

# **Network Systems Design (Intel IXP2xxx)**

**Douglas Comer**

**Computer Science Department  
Purdue University  
250 N. University Street  
West Lafayette, IN 47907-2066**

**<http://www.cs.purdue.edu/people/comer>**

© Copyright 2004. All rights reserved. This document may not be reproduced by any means without written consent of the author.

Copy permission: these materials are copyright © 2004 by Pearson Education and Douglas Comer, and may not be reproduced by any means without written permission from the author or the publisher. Permission is granted to use the materials in any course for which Comer's text *Network Systems Design Using Network Processors* is a required textbook. In addition to use for in-class presentation, each student who purchases a copy of the textbook is authorized to receive an electronic or paper copy. For permission to use the materials in any way other than the above, contact the author or the publisher.

# I

## **Course Introduction And Overview**

# Topic And Scope

The concepts, principles, and technologies that underlie the design of hardware and software systems used in computer networks and the Internet, focusing on the emerging field of network processors.

# You Will Learn

- Review of
  - Network systems
  - Protocols and protocol processing tasks
- Hardware architectures for protocol processing
- Software-based network systems and software architectures
- Classification
  - Concept
  - Software and hardware implementations
- Switching fabrics

# You Will Learn

## (continued)

- Network processors: definition, architectures, and use
- Design tradeoffs and consequences
- Survey of commercial network processors
- Details of one example network processor
  - Architecture and instruction set(s)
  - Programming model and program optimization
  - Cross-development environment

# What You Will NOT Learn

- EE details
  - VLSI technology and design rules
  - Chip interfaces: ICs and pin-outs
  - Waveforms, timing, or voltage
  - How to wire wrap or solder
- Economic details
  - Comprehensive list of vendors and commercial products
  - Price points

# Background Required

- Basic knowledge of
  - Network and Internet protocols
  - Packet headers
- Basic understanding of hardware architecture
  - Registers
  - Memory organization
  - Typical instruction set
- Willingness to use an assembly language



# Schedule Of Topics

- Quick review of basic networking
- Protocol processing tasks and classification
- Software-based systems using conventional hardware
- Special-purpose hardware for high speed
- Motivation and role of network processors
- Network processor architectures

# Schedule Of Topics

## (continued)

- An example network processor technology in detail
  - Hardware architecture and parallelism
  - Programming model
  - Testbed architecture and features
- Design tradeoffs
- Scaling a network processor
- Survey of network processor architectures

# Course Administration

- Textbook
  - D. Comer, *Network Systems Design Using Network Processors*, Intel IXP2xxx version, Prentice Hall, 2005.
- Grade
  - Quizzes 5%
  - Midterm and final exam 35%
  - Programming projects 60%

# Lab Facilities Available

- Extensive network processor testbed facilities
- Donations from
  - Agere Systems
  - IBM (now sold to Hifn)
  - Intel
- Includes hardware and cross-development software

# What You Will Do In The Lab

- Write and compile software for an NP
- Download software into an NP
- Monitor the NP as it runs
- Interconnect Ethernet ports on an NP board
  - To other ports on other NP boards
  - To other computers in the lab
- Send Ethernet traffic to the NP
- Receive Ethernet traffic from the NP

# Example Programming Projects

- A packet analyzer
  - IP datagrams
  - TCP segments
- An Ethernet bridge
- An IP fragmenter
- A classification program
- A bump-in-the-wire system using low-level packet processors



**Questions?**

# **A QUICK OVERVIEW OF NETWORK PROCESSORS**



# The Network Systems Problem

- Data rates keep increasing
- Protocols and applications keep evolving
- System design is expensive
- System implementation and testing take too long
- Systems often contain errors
- Special-purpose hardware designed for one system cannot be reused

# The Challenge

Find ways to improve the design and manufacture of complex networking systems.

# The Big Questions

- What systems?
  - Everything we have now
  - New devices not yet designed
- What physical communication mechanisms?
  - Everything we have now
  - New communication systems not yet designed / standardized
- What speeds?
  - Everything we have now
  - New speeds much faster than those in use

# More Big Questions

- What protocols?
  - Everything we have now
  - New protocols not yet designed / standardized
- What applications?
  - Everything we have now
  - New applications not yet designed / standardized

# The Challenge (restated)

*Find flexible, general technologies that enable rapid, low-cost design and manufacture of a variety of scalable, robust, efficient network systems that run a variety of existing and new protocols, perform a variety of existing and new functions for a variety of existing and new, higher-speed networks to support a variety of existing and new applications.*

# Special Difficulties

- Ambitious goal
- Vague problem statement
- Problem is evolving with the solution
- Pressure from
  - Changing infrastructure
  - Changing applications

# Desiderata

- High speed
- Flexible and extensible to accommodate
  - Arbitrary protocols
  - Arbitrary applications
  - Arbitrary physical layer
- Low cost

# Desiderata

- High speed
- Flexible and extensible to accommodate
  - Arbitrary protocols
  - Arbitrary applications
  - Arbitrary physical layer
- Low cost



# Statement Of Hope

(1995 version)

*If there is hope, it lies in ASIC designers.*

# Statement Of Hope

(1999 version)

???

*If there is hope, it lies in ASIC designers.*



# Statement Of Hope

(2004 version)

*programmers!*

*If there is hope, it lies in ASIC designers.*

# Programmability

- Key to low-cost hardware for next generation network systems
- More flexibility than ASIC designs
- Easier / faster to update than ASIC designs
- Less expensive to develop than ASIC designs
- What we need: a programmable device with more capability than a conventional CPU

# The Idea In A Nutshell

- Devise new hardware building blocks
- Make them programmable
- Include support for protocol processing and I/O
  - General-purpose processor(s) for control tasks
  - Special-purpose processor(s) for packet processing and table lookup
- Include functional units for tasks such as checksum computation
- Integrate as much as possible onto one chip
- Call the result a *network processor*

# The Rest Of The Course

- We will
  - Examine the general problem being solved
  - Survey some approaches vendors have taken
  - Explore possible architectures
  - Study example technologies
  - Consider how to implement systems using network processors

# Disclaimer #1

In the field of network processors, I am a tyro.

# Definition

Tyro \Ty'ro\, n.; pl. *Tyros*. A beginner in learning; one who is in the rudiments of any branch of study; a person imperfectly acquainted with a subject; a novice.



# By Definition

In the field of network processors, you are all tyros.

# In Our Defense

When it comes to network processors, everyone is a tyro.



**Questions?**

## II

# Basic Terminology And Example Systems (A Quick Review)

# Packets Cells And Frames

- *Packet*
  - Generic term
  - Small unit of data being transferred
  - Travels independently
  - Upper and lower bounds on size

# Packets Cells And Frames (continued)

- *Cell*
  - Fixed-size packet (e.g., ATM)
- *Frame or layer-2 packet*
  - Packet understood by hardware
- *IP datagram*
  - Internet packet

# Types Of Networks

- Paradigm
  - *Connectionless*
  - *Connection-oriented*
- Access type
  - *Shared* (i.e., multiaccess)
  - *Point-To-Point*

# Connection-Oriented Networks

- Telephone paradigm (connection, use, disconnect)
- Examples
  - Frame Relay
  - Asynchronous Transfer Mode (ATM)



# Point-To-Point Network

- Connects exactly two systems
- Often used for long distance
- Example: data circuit connecting two routers

# Data Circuit

- Leased from phone company
- Also called *serial line* because data is transmitted bit-serially
- Originally designed to carry digital voice
- Cost depends on speed and distance
- T-series standards define low speeds (e.g. T1)
- STS and OC standards define high speeds

# Digital Circuit Speeds

<b>Standard Name</b>	<b>Bit Rate</b>	<b>Voice Circuits</b>
—	0.064 Mbps	1
T1	1.544 Mbps	24
T3	44.736 Mbps	672
OC-1	51.840 Mbps	810
OC-3	155.520 Mbps	2430
OC-12	622.080 Mbps	9720
OC-24	1,244.160 Mbps	19440
OC-48	2,488.320 Mbps	38880
OC-192	9,953.280 Mbps	155520
OC-768	39,813.120 Mbps	622080

# Digital Circuit Speeds

<u>Standard Name</u>	<u>Bit Rate</u>	<u>Voice Circuits</u>
–	0.064 Mbps	1
T1	1.544 Mbps	24
T3	44.736 Mbps	672
OC-1	51.840 Mbps	810
OC-3	155.520 Mbps	2430
OC-12	622.080 Mbps	9720
OC-24	1,244.160 Mbps	19440
OC-48	2,488.320 Mbps	38880
<b>OC-192</b>	<b>9,953.280 Mbps</b>	<b>155520</b>
OC-768	39,813.120 Mbps	622080

- Holy grail of networking: devices capable of accepting and forwarding data at 10 Gbps (OC-192).

# Local Area Networks

- Ethernet technology dominates
- Layer 1 standards
  - Media and wiring
  - Signaling
  - Handled by dedicated interface chips
  - Unimportant to us
- Layer 2 standards
  - MAC framing and addressing

# MAC Addressing

- Three address types
  - Unicast (single computer)
  - Broadcast (all computers in broadcast domain)
  - Multicast (some computers in broadcast domain)

# More Terminology

- *Internet*
  - Interconnection of multiple networks
  - Allows heterogeneity of underlying networks
- Network scope
  - *Local Area Network (LAN)* covers limited distance
  - *Wide Area Network (WAN)* covers arbitrary distance

# Network System

- Individual hardware component
- Serves as fundamental building block
- Used in networks and internets
- May contain processor and software
- Operates at one or more layers of the protocol stack



# Example Network Systems

- Layer 2
  - Bridge
  - Ethernet switch
  - VLAN switch

# VLAN Switch

- Similar to conventional layer 2 switch
  - Connects multiple computers
  - Forwards frames among them
  - Each computer has unique *unicast address*
- Differs from conventional layer 2 switch
  - Allows manager to configure *broadcast domains*
- Broadcast domain known as *virtual network*

# Broadcast Domain

- Determines propagation of broadcast / multicast
- Originally corresponded to fixed hardware
  - One per cable segment
  - One per hub or switch
- Now configurable via VLAN switch
  - Manager assigns ports to VLANs

# Example Network Systems (continued)

- Layer 3
  - Internet host computer
  - IP router (layer 3 switch)
- Layer 4
  - Basic Network Address Translator (NAT)
  - Round-robin Web load balancer
  - TCP terminator

# Example Network Systems (continued)

- Layer 5
  - Firewall
  - Intrusion Detection System (IDS)
  - Virtual Private Network (VPN)
  - Softswitch running SIP
  - Application gateway
  - TCP splicer (also known as NAT — Network Address and Protocol Translator)
  - Smart Web load balancer
  - Set-top box

# Example Network Systems (continued)

- Network control systems
  - Packet / flow analyzer
  - Traffic monitor
  - Traffic policer
  - Traffic shaper



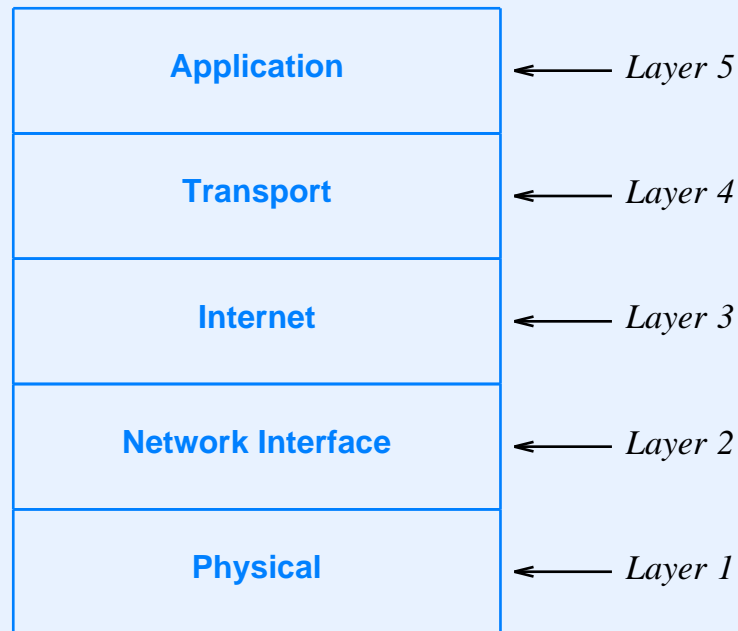
**Questions?**

# III

## Review Of Protocols And Packet Formats



# Protocol Layering



- Five-layer Internet reference model
- Multiple protocols can occur at each layer

# Layer 2 Protocols

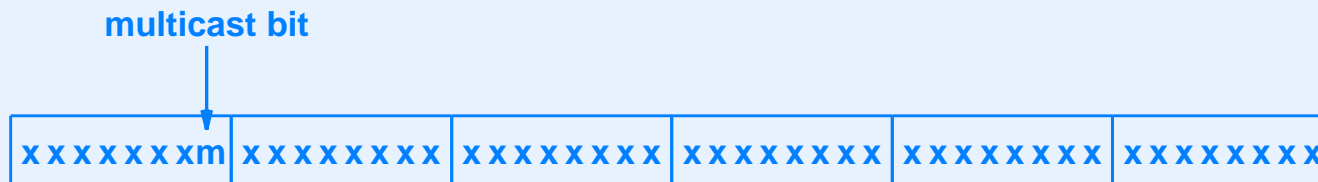
- Two protocols are important
  - Ethernet (widely used)
  - ATM (defines per-flow QoS)
- We will concentrate on Ethernet

# Ethernet Addressing

- 48-bit addressing
- Unique address assigned to each station (NIC)
- Destination address in each packet can specify delivery to
  - A single computer (unicast)
  - All computers in broadcast domain (broadcast)
  - Some computers in broadcast domain (multicast)

# Ethernet Addressing (continued)

- Broadcast address is all 1s
- Single bit determines whether remaining addresses are unicast or multicast



- Multicast bit travels first on the wire



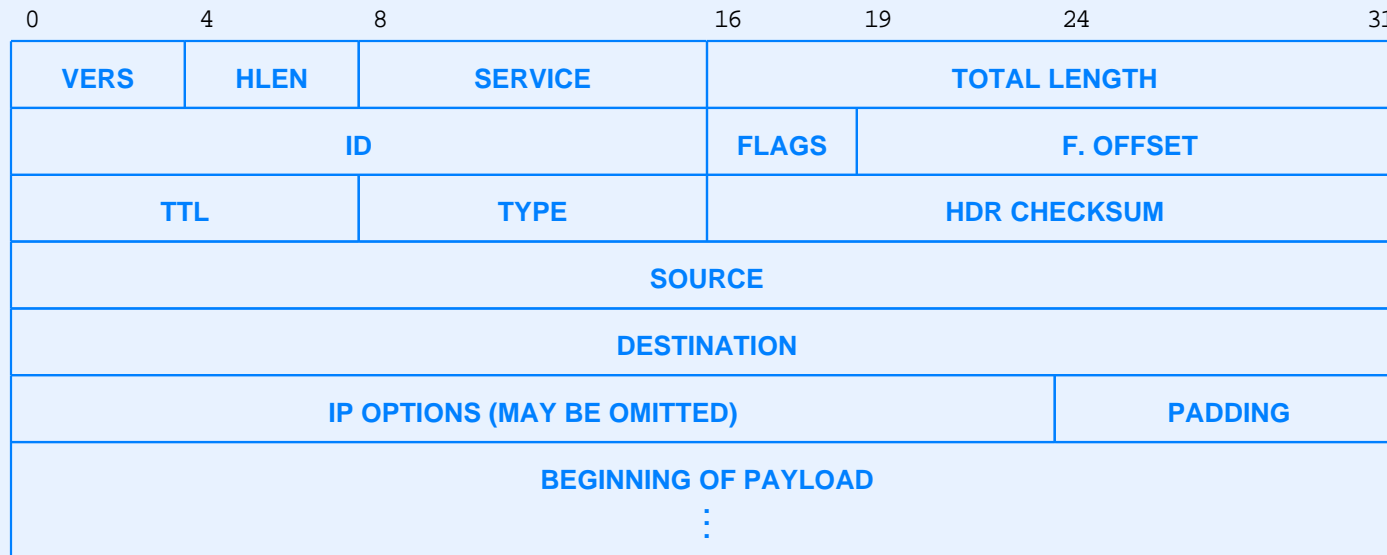
# Internet

- Set of (heterogeneous) computer networks interconnected by *IP routers*
- End-user computers, called *hosts*, each attach to specific network
- Protocol software
  - Runs on both hosts and routers
  - Provides illusion of homogeneity

# Internet Protocols Of Interest

- Layer 2
  - Address Resolution Protocol (ARP)
- Layer 3
  - Internet Protocol (IP)
- Layer 4
  - User Datagram Protocol (UDP)
  - Transmission Control Protocol (TCP)

# IP Datagram Format



- Format of each packet sent across Internet
- Fixed-size fields make parsing efficient



# IP Datagram Fields

Field	Meaning
VERS	Version number of IP being used (4)
HLEN	Header length measured in 32-bit units
SERVICE	Level of service desired
TOTAL LENGTH	Datagram length in octets including header
ID	Unique value for this datagram
FLAGS	Bits to control fragmentation
F. OFFSET	Position of fragment in original datagram
TTL	Time to live (hop countdown)
TYPE	Contents of payload area
HDR CHECKSUM	One's-complement checksum over header
SOURCE	IP address of original sender
DESTINATION	IP address of ultimate destination
IP OPTIONS	Special handling parameters
PADDING	To make options a 32-bit multiple

# IP addressing

- 32-bit Internet address assigned to each computer
- Virtual, hardware independent value
- Prefix identifies network; suffix identifies host
- Network systems use an address mask to specify the boundary between prefix and suffix

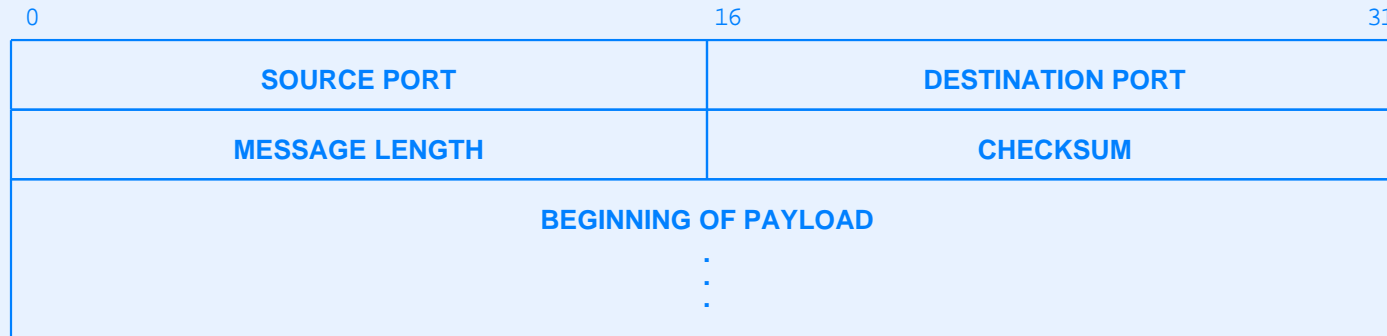
# Next-Hop Forwarding

- Routing table
  - Found in both hosts and routers
  - Stores ( destination, mask, next\_hop ) tuples
- Route lookup
  - Takes destination address as argument
  - Finds next hop
  - Uses longest-prefix match

# Next-Hop Forwarding

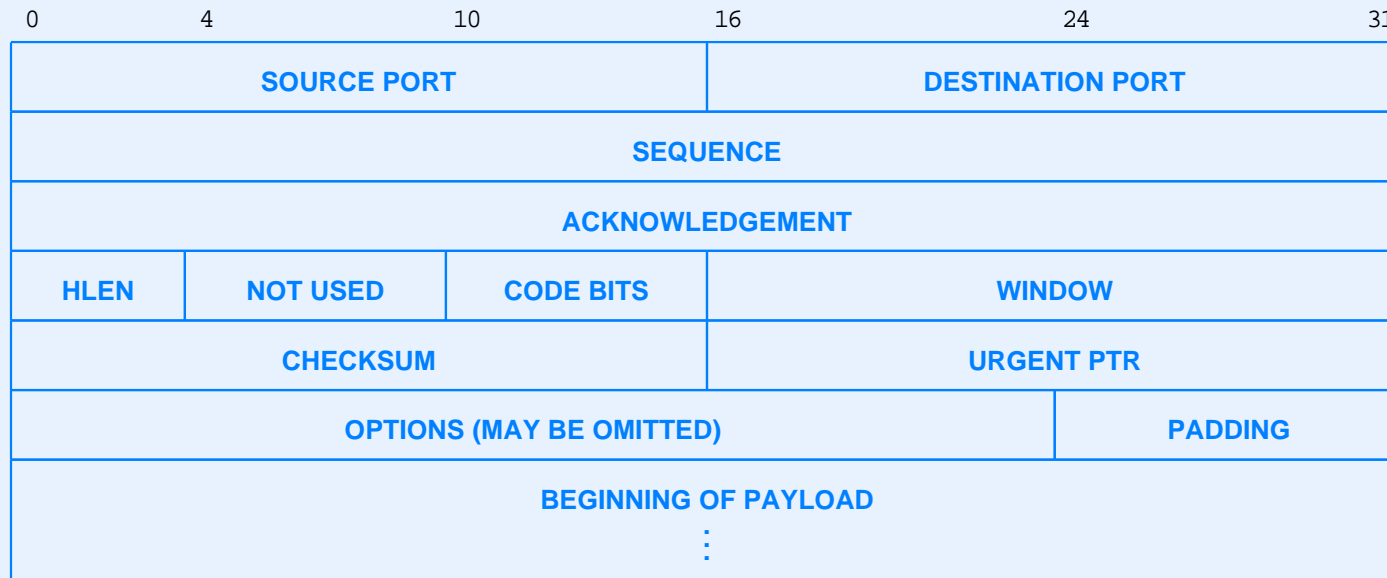
- Routing table
  - Found in both hosts and routers
  - Stores ( destination, mask, next\_hop ) tuples
- Route lookup
  - Takes destination address as argument
  - Finds next hop
  - Uses longest-prefix match

# UDP Datagram Format



Field	Meaning
SOURCE PORT	ID of sending application
DESTINATION PORT	ID of receiving application
MESSAGE LENGTH	Length of datagram including the header
CHECKSUM	One's-complement checksum over entire datagram

# TCP Segment Format

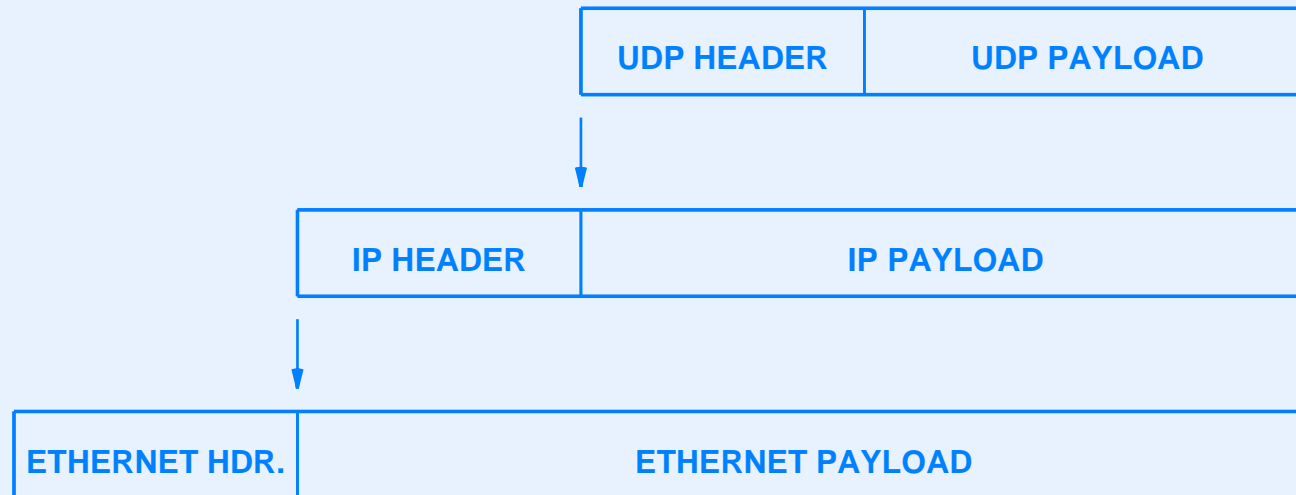


- Sent end-to-end
- Fixed-size fields make parsing efficient

# TCP Segment Fields

Field	Meaning
SOURCE PORT	ID of sending application
DESTINATION PORT	ID of receiving application
SEQUENCE	Sequence number for data in payload
ACKNOWLEDGEMENT	Acknowledgement of data received
HLEN	Header length measured in 32-bit units
NOT USED	Currently unassigned
CODE BITS	URGENT, ACK, PUSH, RESET, SYN, FIN
WINDOW	Receiver's buffer size for additional data
CHECKSUM	One's-complement checksum over entire segment
URGENT PTR	Pointer to urgent data in segment
OPTIONS	Special handling
PADDING	To make options a 32-bit multiple

# Illustration Of Encapsulation



- Field in each header specifies type of encapsulated packet



# Example ARP Packet Format

0	8	16	24	31
ETHERNET ADDRESS TYPE (1)		IP ADDRESS TYPE (0800)		
ETH ADDR LEN (6)	IP ADDR LEN (4)	OPERATION		
SENDER'S ETH ADDR (first 4 octets)				
SENDER'S ETH ADDR (last 2 octets)		SENDER'S IP ADDR (first 2 octets)		
SENDER'S IP ADDR (last 2 octets)		TARGET'S ETH ADDR (first 2 octets)		
TARGET'S ETH ADDR (last 4 octets)				
TARGET'S IP ADDR (all 4 octets)				

- Format when ARP used with Ethernet and IP
- Each Ethernet address is six octets
- Each IP address is four octets

**End Of Review**



**Questions?**



# IV

## Conventional Computer Hardware Architecture

# Software-Based Network System

- Uses conventional hardware (e.g., PC)
- Software
  - Runs the entire system
  - Allocates memory
  - Controls I/O devices
  - Performs all protocol processing

# Why Study Protocol Processing On Conventional Hardware?

- Past
  - Employed in early IP routers
  - Many algorithms developed / optimized for conventional hardware
- Present
  - Used in low-speed network systems
  - Easiest to create / modify
  - Costs less than special-purpose hardware

# Why Study Protocol Processing On Conventional Hardware? (continued)

- Future
  - Processors continue to increase in speed
  - Some conventional hardware present in all systems

# Why Study Protocol Processing On Conventional Hardware? (continued)

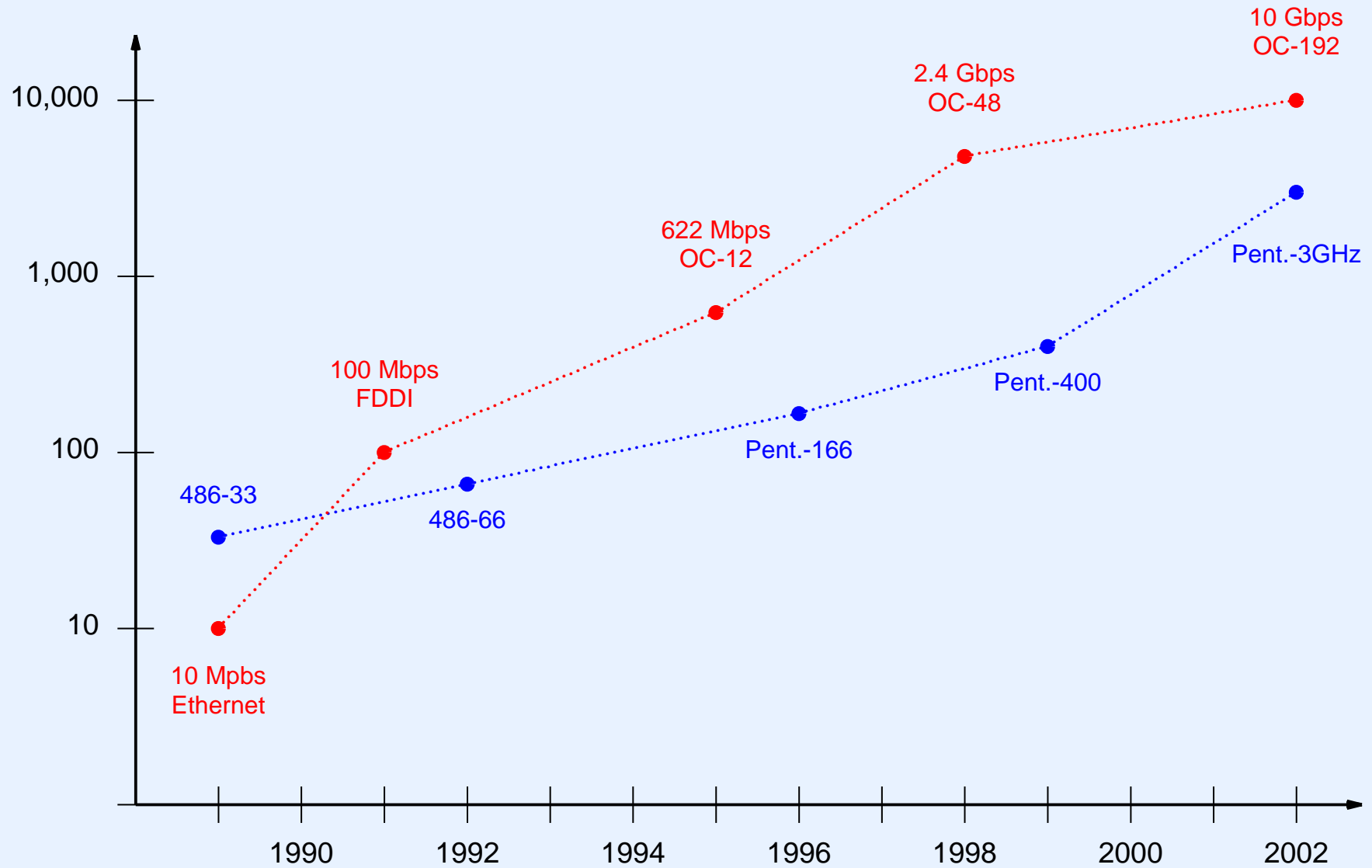
- Future
  - Processors continue to increase in speed
  - Some conventional hardware present in all systems
  - You will build software-based systems in lab!



# Serious Question

- Which is growing faster?
  - Processing power
  - Network bandwidth
- Note: if network bandwidth growing faster
  - Need special-purpose hardware
  - Conventional hardware will become irrelevant

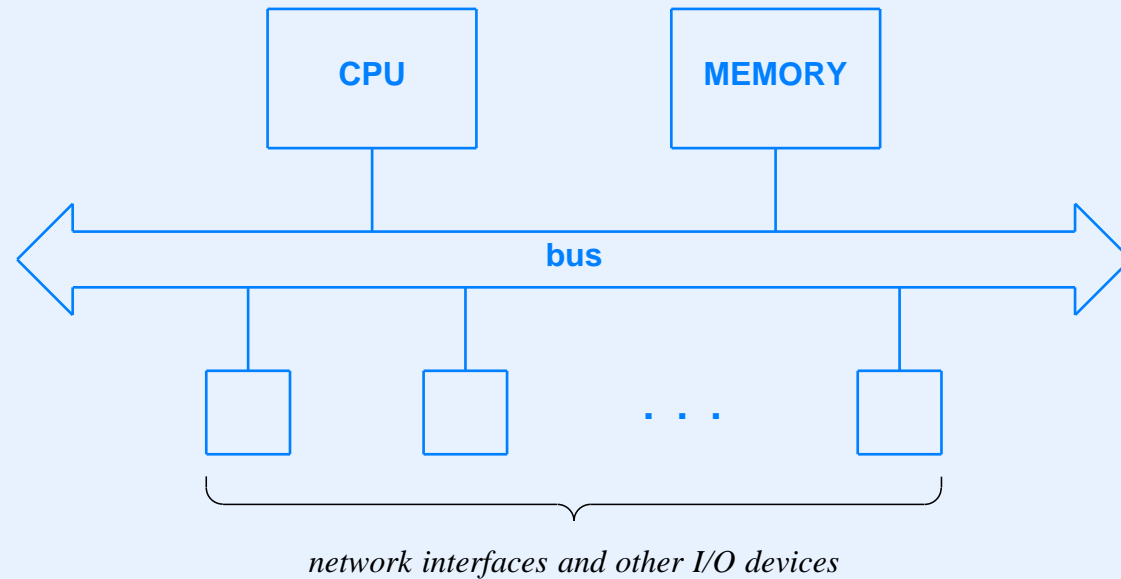
# Growth Of Technologies



# Conventional Computer Hardware

- Four important aspects
  - Processor
  - Memory
  - I/O interfaces
  - One or more buses

# Illustration Of Conventional Computer Architecture



- Bus is central, shared interconnect
- All components *contend* for use

# Bus Organization And Operations



- Parallel wires ( $C+A+D$  total)
- Used to pass
  - Control information ( $C$  bits)
  - An address ( $A$  bits)
  - A data value ( $D$  bits)

# Bus Width

- Number of parallel data bits known as *width* of bus
- Wider bus
  - Transfers more data per unit time
  - Costs more
  - Requires more physical space
- Compromise: to simulate wider bus, use hardware that multiplexes transfers

# Bus Paradigm

- Only two basic operations
  - Fetch
  - Store
- All operations cast as forms of the above

# Fetch/Store

- Fundamental paradigm
- Used throughout hardware, including network processors



# Fetch Operation

- Place address of a device on address lines
- Issue *fetch* on control lines
- Use control lines to wait for device that owns the address to respond
- If operation successful, extract value (response) from data lines
- If not successful, report error

# Store Operation

- Place address of a device on address lines
- Place value on data lines
- Issue *store* on control lines
- Use control lines to wait for device that owns the address to respond
- If operation does not succeed, report error

# Example Of Operations Mapped Into Fetch/Store Paradigm

- Imagine disk device attached to a bus
- Assume disk hardware supports three (nontransfer) operations:
  - Start disk spinning
  - Stop disk
  - Determine current status

# Example Of Operations Mapped Into Fetch/Store Paradigm (continued)

- Assign the disk two contiguous bus addresses  $D$  and  $D+1$
- Arrange for store of nonzero to address  $D$  to start disk spinning
- Arrange for store of zero to address  $D$  to stop disk
- Arrange for fetch from address  $D+1$  to return current status
- Note: effect of store to address  $D+1$  can be defined as
  - Appears to work, but has no effect
  - Returns an error

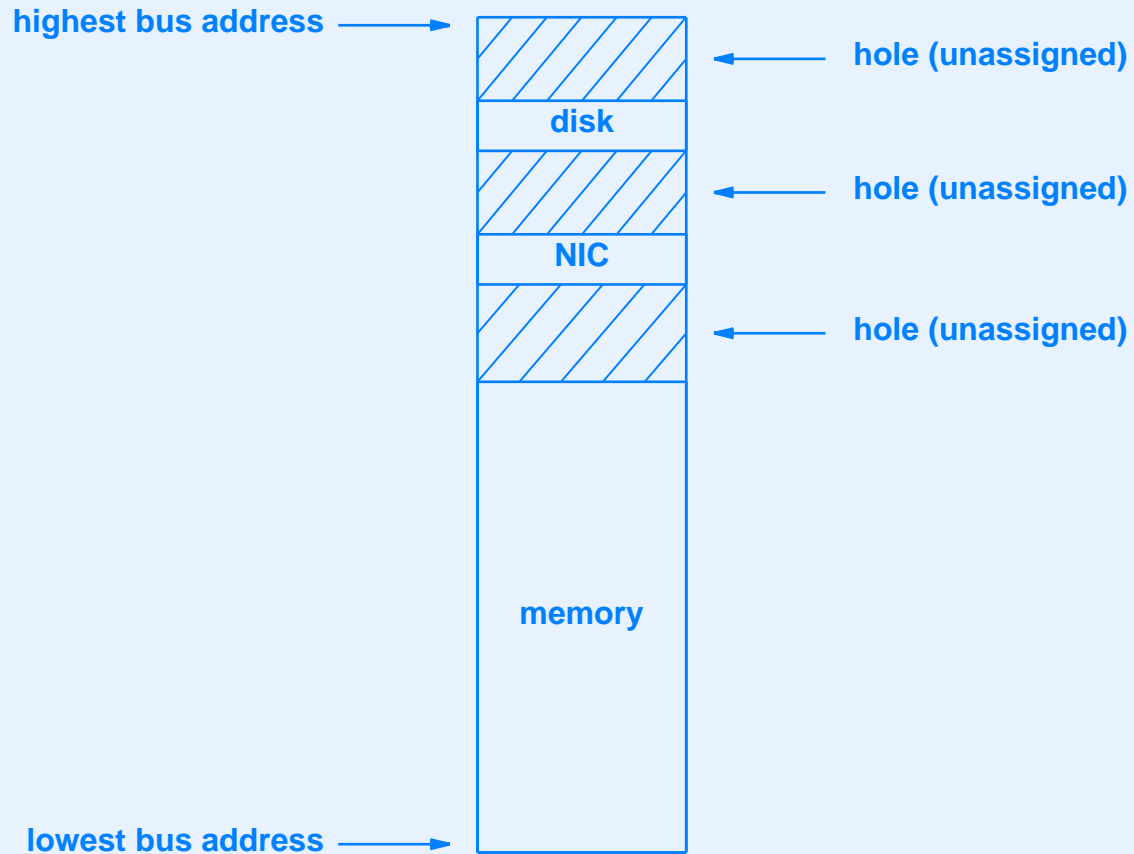
# Bus Address Space

- Arbitrary hardware can be attached to bus
- K address lines result in  $2^k$  possible bus addresses
- Address can refer to
  - Memory (e.g., RAM or ROM)
  - I/O device
- Arbitrary devices can be placed at arbitrary addresses
- Address space can contain “holes”

# Bus Address Terminology

- Device on bus known as *memory mapped I/O*
- Locations that correspond to nontransfer operations known as *Control and Status Registers (CSRs)*

# Example Bus Address Space



# Network I/O On Conventional Hardware

- Network Interface Card (NIC)
  - Attaches between bus and network
  - Operates like other I/O devices
  - Handles electrical/optical details of network
  - Handles electrical details of bus
  - Communicates over bus with CPU or other devices



# Making Network I/O Fast

- Key idea: migrate more functionality onto NIC
- Four techniques used with bus
  - Onboard address recognition & filtering
  - Onboard packet buffering
  - Direct Memory Access (DMA)
  - Operation and buffer chaining

# Onboard Address Recognition And Filtering

- NIC given set of addresses to accept
  - Station's unicast address
  - Network broadcast address
  - Zero or more multicast addresses
- When packet arrives, NIC checks destination address
  - Accept packet if address on list
  - Discard others

# Onboard Packet Buffering

- NIC given high-speed local memory
- Incoming packet placed in NIC's memory
- Allows computer's memory/bus to operate slower than network
- Handles small packet bursts

# Direct Memory Access (DMA)

- CPU
  - Allocates packet buffer in memory
  - Passes buffer address to NIC
  - Goes on with other computation
- NIC
  - Accepts incoming packet from network
  - Copies packet over bus to buffer in memory
  - Informs CPU that packet has arrived

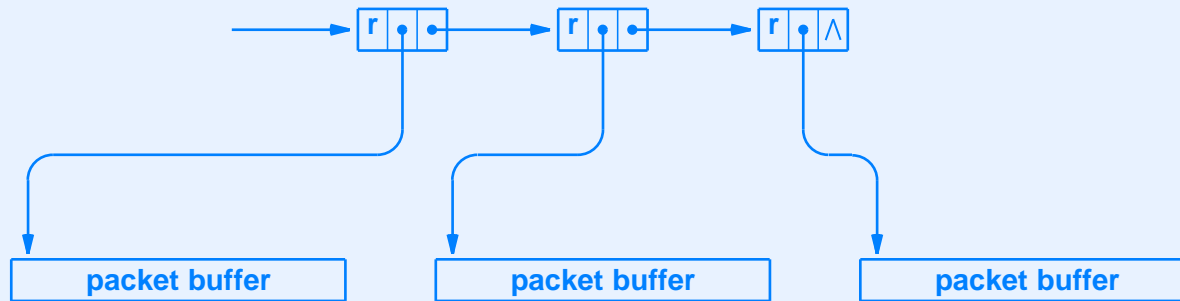
# Buffer Chaining

- CPU
  - Allocates multiple buffers
  - Passes linked list to NIC
- NIC
  - Receives next packet
  - Divides into one or more buffers
- Advantage: a buffer can be smaller than a packet

# Operation Chaining

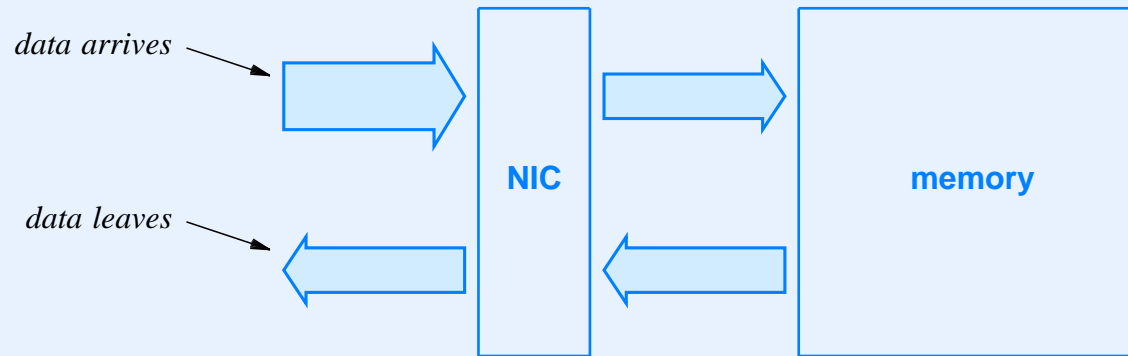
- CPU
  - Allocates multiple buffers
  - Builds linked list of operations
  - Passes list to NIC
- NIC
  - Follows list and performs instructions
  - Interrupts CPU after each operation
- Advantage: multiple operations proceed without CPU intervention

# Illustration Of Operation Chaining



- Optimizes movement of data to memory

# Data Flow Diagram



- Depicts flow of data through hardware units
- Size of arrow represents throughput
- Used throughout the course and text



# Summary

- Software-based network systems run on conventional hardware
  - Processor
  - Memory
  - I/O devices
  - Bus
- Network interface cards can be optimized to reduce CPU load



**Questions?**

# V

## **Basic Packet Processing: Algorithms And Data Structures**

# Copying

- Used when packet moved from one memory location to another
- Expensive
- Must be avoided whenever possible
  - Leave packet in buffer
  - Pass buffer address among threads/layers

# Possibilities For Buffer Allocation

- Fixed-size buffers
  - \* Large enough for largest packet
    - \* Small, with multiple buffers linked together for large packets
- Variable-size buffers

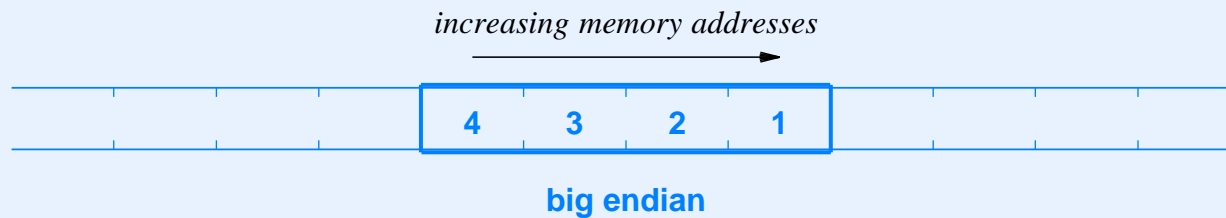
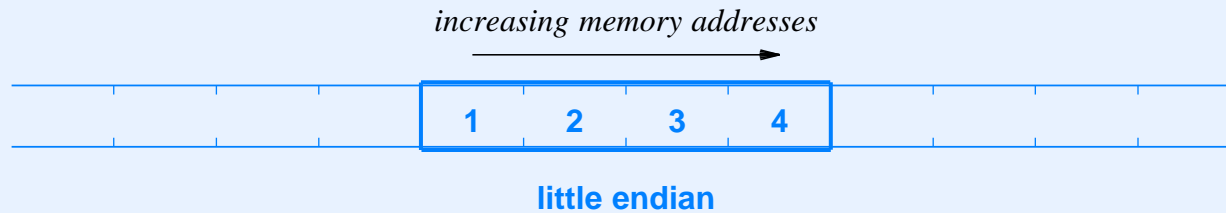
# Buffer Addressing

- Buffer address must be resolvable in all contexts
- Easiest implementation: keep buffers in kernel space

# Integer Representation

- Two standards
  - Little endian (least-significant byte at lowest address)
  - Big endian (most-significant byte at lowest address)

# Illustration Of Big And Little Endian Integers





# Integer Conversion

- Needed when heterogeneous computers communicate
- Protocols define *network byte order*
- Computers convert to network byte order
- Typical library functions

Function	data size	Translation
<b>ntohs</b>	<b>16 bits</b>	<b>Network byte order to host's byte order</b>
<b>htons</b>	<b>16 bits</b>	<b>Host's byte order to network byte order</b>
<b>ntohl</b>	<b>32 bits</b>	<b>Network byte order to host's byte order</b>
<b>htonl</b>	<b>32 bits</b>	<b>Host's byte order to network byte order</b>

# Examples Of Algorithms Implemented With Software-Based Systems

- Layer 2
  - Ethernet bridge
- Layer 3
  - IP forwarding
  - IP fragmentation and reassembly
- Layer 4
  - TCP connection recognition and splicing
- Other
  - Hash table lookup

# Why Study These Algorithms?

# Why Study These Algorithms?

- Provide insight to packet processing tasks

# Why Study These Algorithms?

- Provide insight to packet processing tasks
- Reinforce concepts

# Why Study These Algorithms?

- Provide insight to packet processing tasks
- Reinforce concepts
- Help students recall protocol details

# Why Study These Algorithms?

- Provide insight to packet processing tasks
- Reinforce concepts
- Help students recall protocol details
- **You will need them in lab!**

# Ethernet Bridge



- Used between a pair of Ethernets
- Provides transparent, layer 2 connection
- Listens in promiscuous mode
- Forwards frames in both directions
- Uses addresses to filter



# Bridge Filtering

- Uses source address in frames to identify computers on each network
- Uses destination address to decide whether to forward frame

# Bridge Algorithm

Assume: two network interfaces each operating in promiscuous mode.

Create an empty list, L, that will contain pairs of values;

Do forever {

    Acquire the next frame to arrive;

    Set I to the interface over which the frame arrived;

    Extract the source address, S;

    Extract the destination address, D;

    Add the pair ( S, I ) to list L if not already present.

    If the pair ( D, I ) appears in list L {

        Drop the frame;

    } Else {

        Forward the frame over the other interface;

    }

}

# Implementation Of Table Lookup

- Need high speed (more on this later)
- Software-based systems typically use *hashing* for table lookup

# Hashing

- Optimizes number of *probes*
- Works well if table not full
- Practical technique: *double hashing*

# Hashing Algorithm

Given: a key, a table in memory, and the table size  $N$ .

Produce: a slot in the table that corresponds to the key or an empty table slot if the key is not in the table.

Method: double hashing with open addressing.

Choose  $P_1$  and  $P_2$  to be prime numbers;

Fold the key to produce an integer,  $K$ ;

Compute table pointer  $Q$  equal to  $(P_1 \times K)$  modulo  $N$ ;

Compute increment  $R$  equal to  $(P_2 \times K)$  modulo  $N$ ;

While (table slot  $Q$  not equal to  $K$  and nonempty) {

$Q \leftarrow (Q + R)$  modulo  $N$ ;

}

At this point,  $Q$  either points to an empty table slot or to the slot containing the key.

# Address Lookup

- Computer can compare integer in one operation
- Network address can be longer than integer (e.g., 48 bits)
- Two possibilities
  - Use multiple comparisons per probe
  - Fold address into integer key

# Folding

- Maps N-bit value into M-bit key,  $M < N$
- Typical technique: exclusive or
- Potential problem: two values map to same key
- Solution: compare full value when key matches

# IP Forwarding

- Used in hosts as well as routers
- Conceptual mapping

(next hop, interface)  $\leftarrow f(\text{datagram, routing table})$

- Table driven



# IP Routing Table

- One entry per destination
- Entry contains
  - 32-bit IP address of destination
  - 32-bit address mask
  - 32-bit next-hop address
  - N-bit interface number

# Example IP Routing Table

Destination Address	Address Mask	Next-Hop Address	Interface Number
192.5.48.0	255.255.255.0	128.210.30.5	2
128.10.0.0	255.255.0.0	128.210.141.12	1
0.0.0.0	0.0.0.0	128.210.30.5	2

- Values stored in binary
- Interface number is for internal use only
- Zero mask produces *default* route

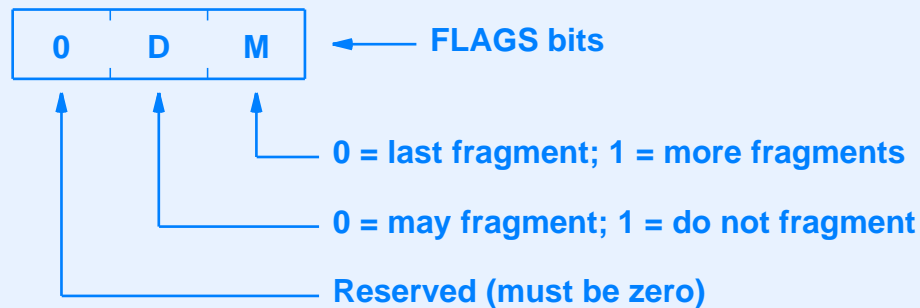
# IP Forwarding Algorithm

```
Given: destination address A and routing table R.  
Find: a next hop and interface used to route datagrams to A.  
For each entry in table R {  
    Set MASK to the Address Mask in the entry;  
    Set DEST to the Destination Address in the entry;  
    If (A & MASK) == DEST {  
        Stop; use the next hop and interface in the entry;  
    }  
}  
If this point is reached, declare error: no route exists;
```

- Note: algorithm assumes table is sorted in longest-prefix order

# IP Fragmentation

- Needed when datagram larger than network MTU
- Divides IP datagram into *fragments*
- Uses FLAGS bits in datagram header



# IP Fragmentation Algorithm

## (Part 1: Initialization)

Given: an IP datagram,  $D$ , and a network MTU.

Produce: a set of fragments for  $D$ .

If the *DO NOT FRAGMENT* bit is set {

    Stop and report an error;

}

Compute the size of the datagram header,  $H$ ;

Choose  $N$  to be the largest multiple of 8 such

    that  $H+N \leq \text{MTU}$ ;

Initialize an offset counter,  $O$ , to zero;

# IP Fragmentation Algorithm

## (Part 2: Processing)

```
Repeat until datagram empty {  
    Create a new fragment that has a copy of D's header;  
    Extract up to the next N octets of data from D and place  
    the data in the fragment;  
    Set the MORE FRAGMENTS bit in fragment header;  
    Set TOTAL LENGTH field in fragment header to be H+N;  
    Set FRAGMENT OFFSET field in fragment header to O;  
    Compute and set the CHECKSUM field in fragment  
    header;  
    Increment O by N/8;  
}
```

# Reassembly

- Complement of fragmentation
- Uses IP SOURCE ADDRESS and IDENTIFICATION fields in datagram header to group related fragments
- Joins fragments to form original datagram

# Reassembly Algorithm

Given: a fragment, F, add to a partial reassembly.

Method: maintain a set of fragments for each datagram.

Extract the IP source address, S, and ID fields from F;

Combine S and ID to produce a lookup key, K;

Find the fragment set with key K or create a new set;

Insert F into the set;

If the set contains all the data for the datagram {

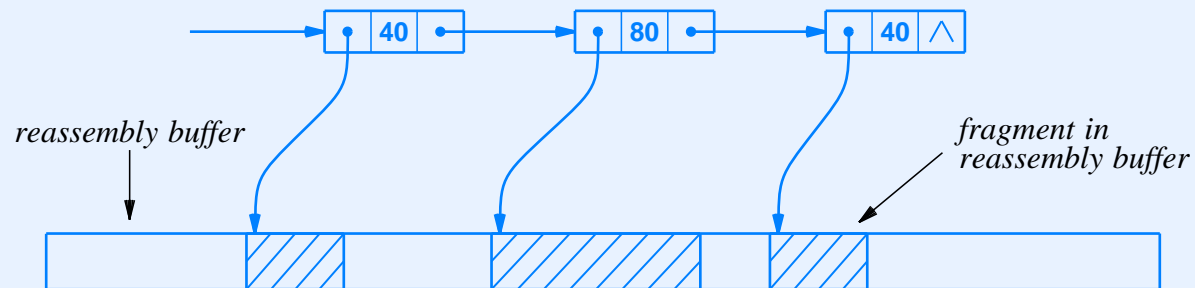
    Form a completely reassembled datagram and process it;

}



# Data Structure For Reassembly

- Two parts
  - Buffer large enough to hold original datagram
  - Linked list of pieces that have arrived



# TCP Connection

- Involves a pair of endpoints
- Started with SYN segment
- Terminated with FIN or RESET segment
- Identified by 4-tuple

( src addr, dest addr, src port, dest port )

# TCP Connection Recognition Algorithm (Part 1)

Given: a copy of traffic passing across a network.

Produce: a record of TCP connections present in the traffic.

Initialize a connection table, C, to empty;

For each IP datagram that carries a TCP segment {

    Extract the IP source, S, and destination, D, addresses;

    Extract the source,  $P_1$ , and destination,  $P_2$ , port numbers;

    Use (S,D, $P_1$ , $P_2$ ) as a lookup key for table C and

        create a new entry, if needed;

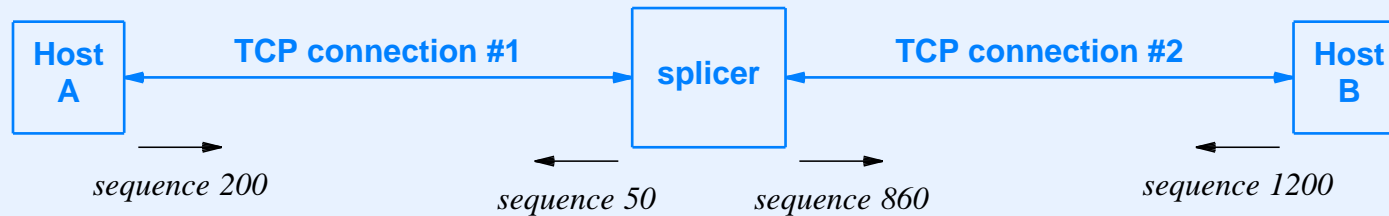
# TCP Connection Recognition Algorithm (Part 2)

```
    If the segment has the RESET bit set, delete the entry;
    Else if the segment has the FIN bit set, mark the
connection
        closed in one direction, removing the entry from C if
        the connection was previously closed in the other;
    Else if the segment has the SYN bit set, mark the
connection as
        being established in one direction, making it completely
        established if it was previously marked as being
        established in the other;
}
```

# TCP Splicing

- Join two TCP connections
- Allow data to pass between them
- To avoid termination overhead translate segment header fields
  - Acknowledgement number
  - Sequence number

# Illustration Of TCP Splicing



Connection & Direction	Sequence Number	Connection & Direction	Sequence Number
Incoming #1	200	Incoming #2	1200
Outgoing #2	860	Outgoing #1	50
Change	660	Change	-1150

# TCP Splicing Algorithm (Part 1)

Given: two TCP connections.

Produce: sequence translations for splicing the connection.

Compute  $D1$ , the difference between the starting sequences on incoming connection 1 and outgoing connection 2;

Compute  $D2$ , the difference between the starting sequences on incoming connection 2 and outgoing connection 1;

# TCP Splicing Algorithm (Part 2)

```
For each segment {  
  If segment arrived on connection 1 {  
    Add D1 to sequence number;  
    Subtract D2 from acknowledgement number;  
  } else if segment arrived on connection 2 {  
    Add D2 to sequence number;  
    Subtract D1 from acknowledgement number;  
  }  
}
```



# Summary

- Packet processing algorithms include
  - Ethernet bridging
  - IP fragmentation and reassembly
  - IP forwarding
  - TCP splicing
- Table lookup important
  - Full match for layer 2
  - Longest prefix match for layer 3



**Questions?**

**For Hands-On Experience With  
A Software-Based System:  
Enroll in CS 636 !**

# VI

## Packet Processing Functions

# Goal

- Identify functions that occur in packet processing
- Devise set of operations sufficient for all packet processing
- Find an efficient implementation for the operations

# Packet Processing Functions We Will Consider

- Address lookup and packet forwarding
- Error detection and correction
- Fragmentation, segmentation, and reassembly
- Frame and protocol demultiplexing
- Packet classification
- Queueing and packet discard
- Scheduling and timing
- Security: authentication and privacy
- Traffic measurement, policing, and shaping

# Address Lookup And Packet Forwarding

- Forwarding requires address lookup
- Lookup is table driven
- Two types
  - Exact match (typically layer 2)
  - Longest-prefix match (typically layer 3)
- Cost depends on size of table and type of lookup

# Error Detection And Correction

- Data sent with packet used as verification
  - Checksum
  - CRC
- Cost proportional to size of packet
- Often implemented with special-purpose hardware



# An Important Note About Cost

*The cost of an operation is proportional to the amount of data processed. An operation such as checksum computation that requires examination of all the data in a packet is among the most expensive.*

# Fragmentation, Segmentation, And Reassembly

- IP fragments and reassembles datagrams
- ATM segments and reassembles AAL5 packets
- Same idea; details differ
- Cost is high because
  - State must be kept and managed
  - Unreassembled fragments occupy memory

# Frame And Protocol Demultiplexing

- Traditional technique used in layered protocols
- Type appears in each header
  - Assigned on output
  - Used on input to select “next” protocol
- Cost of demultiplexing proportional to number of layers

# Packet Classification

- Alternative to demultiplexing
- Crosses multiple layers
- Achieves lower cost
- More on classification later in the course

# Queueing And Packet Discard

- General paradigm is *store-and-forward*
  - Incoming packet placed in queue
  - Outgoing packet placed in queue
- When queue is full, choose packet to discard
- Affects throughput of higher-layer protocols

# Queueing Priorities

- Multiple queues used to enforce priority among packets
- Incoming packet
  - Assigned priority as function of contents
  - Placed in appropriate priority queue
- *Queueing discipline*
  - Examines priority queues
  - Chooses which packet to send

# Examples Of Queueing Disciplines

- Priority Queueing
  - Assign unique priority number to each queue
  - Choose packet from highest priority queue that is nonempty
  - Known as *strict priority* queueing
  - Can lead to starvation

# Examples Of Queueing Disciplines (continued)

- Weighted Round Robin (WRR)
  - Assign unique priority number to each queue
  - Process all queues round-robin
  - Compute  $N$ , max number of packets to select from a queue proportional to priority
  - Take up to  $N$  packets before moving to next queue
  - Works well if all packets equal size



# Examples Of Queueing Disciplines (continued)

- Weighted Fair Queueing (WFQ)
  - Make selection from queue proportional to priority
  - Use packet size rather than number of packets
  - Allocates priority to amount of data from a queue rather than number of packets

# Scheduling And Timing

- Important mechanisms
- Used to coordinate parallel and concurrent tasks
  - Processing on multiple packets
  - Processing on multiple protocols
  - Multiple processors
- Scheduler attempts to achieve fairness

# Security: Authentication And Privacy

- Authentication mechanisms
  - Ensure sender's identity
- Confidentiality mechanisms
  - Ensure that intermediaries cannot interpret packet contents
- Note: in common networking terminology, *privacy* refers to confidentiality
  - Example: Virtual Private Networks

# Traffic Measurement And Policing

- Used by network managers
- Can measure aggregate traffic or per-flow traffic
- Often related to Service Level Agreement (SLA)
- Cost is high if performed in real-time

# Traffic Shaping

- Make traffic conform to statistical bounds
- Typical use
  - Smooth bursts
  - Avoid packet trains
- Only possibilities
  - Discard packets (seldom used)
  - Delay packets

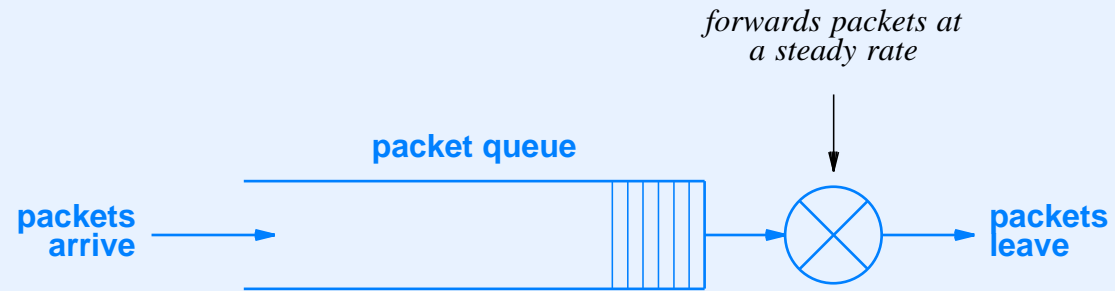
# Example Traffic Shaping Mechanisms

- Leaky bucket
  - Easy to implement
  - Popular
  - Sends steady number of packets per second
  - Rate depends on number of packets waiting
  - Does not guarantee steady data rate

# Example Traffic Shaping Mechanisms (continued)

- Token bucket
  - Sends steady number of bits per second
  - Rate depends on number of bits waiting
  - Achieves steady data rate
  - More difficult to implement

# Illustration Of Traffic Shaper



- Packets
  - Arrive in bursts
  - Leave at steady rate



# Timer Management

- Fundamental piece of network system
- Needed for
  - Scheduling
  - Traffic shaping
  - Other protocol processing (e.g., retransmission)
- Cost
  - Depends on number of timer operations (e.g., set, cancel)
  - Can be high

# Summary

- Primary packet processing functions are
  - Address lookup and forwarding
  - Error detection and correction
  - Fragmentation and reassembly
  - Demultiplexing and classification
  - Queueing and discard
  - Scheduling and timing
  - Security functions
  - Traffic measurement, policing, and shaping



**Questions?**

# VII

## **Protocol Software On A Conventional Processor**

# Possible Implementations Of Protocol Software

- In an application program
  - Easy to program
  - Runs as user-level process
  - No direct access to network devices
  - High cost to copy data from kernel address space
  - Cannot run at *wire speed*

# Possible Implementations Of Protocol Software (continued)

- In an embedded system
  - Special-purpose hardware device
  - Dedicated to specific task
  - Ideal for stand-alone system
  - Software has full control

# Possible Implementations Of Protocol Software (continued)

- In an embedded system
  - Special-purpose hardware device
  - Dedicated to specific task
  - Ideal for stand-alone system
  - Software has full control
  - **You will experience this in lab!**

# Possible Implementations Of Protocol Software (continued)

- In an operating system kernel
  - More difficult to program than application
  - Runs with kernel privilege
  - Direct access to network devices



# Interface To The Network

- Known as *Application Program Interface (API)*
- Can be
  - *Asynchronous*
  - *Synchronous*
- Synchronous interface can use
  - *Blocking*
  - *Polling*

# Asynchronous API

- Also known as *event-driven*
- Programmer
  - Writes set of functions
  - Specifies which function to invoke for each event type
- Programmer has no control over function invocation
- Functions keep state in shared memory
- Difficult to program
- Example: function  $f()$  called when packet arrives

# Synchronous API Using Blocking

- Programmer
  - Writes main flow-of-control
  - Explicitly invokes functions as needed
  - Built-in functions block until request satisfied
- Example: function *wait\_for\_packet()* blocks until packet arrives
- Easier to program

# Synchronous API Using Polling

- Nonblocking form of synchronous API
- Each function call returns immediately
  - Performs operation if available
  - Returns error code otherwise
- Example: function *try\_for\_packet()* either returns next packet or error code if no packet has arrived
- Closer to underlying hardware

# Typical Implementations And APIs

- Application program
  - Synchronous API using blocking (e.g., socket API)
  - Another application thread runs while an application blocks
- Embedded systems
  - Synchronous API using polling
  - CPU dedicated to one task
- Operating systems
  - Asynchronous API
  - Built on interrupt mechanism

# Example Asynchronous API

- Design goals
  - For use with network processor
  - Simplest possible interface
  - Sufficient for basic packet processing tasks
- Includes
  - I/O functions
  - Timer manipulation functions

# Example Asynchronous API (continued)

- Initialization and termination functions
  - `on_startup()`
  - `on_shutdown()`
- Input function (called asynchronously)
  - `recv_frame()`
- Output functions
  - `new_fbuf()`
  - `send_frame()`

## Example Asynchronous API (continued)

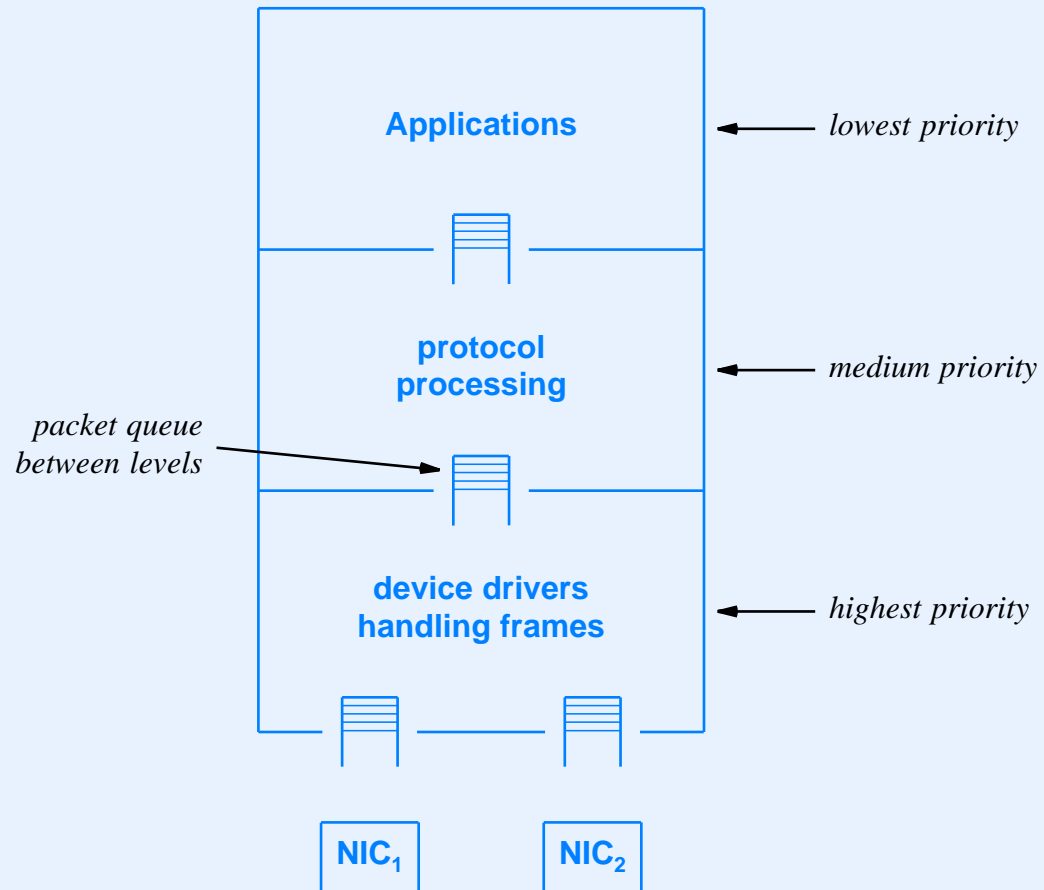
- Timer functions (called asynchronously)
  - `delayed_call()`
  - `periodic_call()`
  - `cancel_call()`
- Invoked by outside application
  - `console_command()`



# Processing Priorities

- Determine which code CPU runs at any time
- General idea
  - Hardware devices need highest priority
  - Protocol software has medium priority
  - Application programs have lowest priority
- Queues provide buffering across priorities

# Illustration Of Priorities



# Implementation Of Priorities In An Operating System

- Two possible approaches
  - Interrupt mechanism
  - Kernel threads

# Interrupt Mechanism

- Built into hardware
- Operates asynchronously
- Saves current processing state
- Changes processor status
- Branches to specified location

# Two Types Of Interrupts

- *Hardware interrupt*
  - Caused by device (bus)
  - Must be serviced quickly
- *Software interrupt*
  - Caused by executing program
  - Lower priority than hardware interrupt
  - Higher priority than other OS code

# Software Interrupts And Protocol Code

- Protocol stack operates as software interrupt
- When packet arrives
  - Hardware interrupts
  - Device driver raises software interrupt
- When device driver finishes
  - Hardware interrupt clears
  - Protocol code is invoked

# Kernel Threads

- Alternative to interrupts
- Familiar to programmer
- Finer-grain control than software interrupts
- Can be assigned arbitrary range of priorities

# Conceptual Organization

- Packet passes among multiple threads of control
- Queue of packets between each pair of threads
- Threads synchronize to access queues



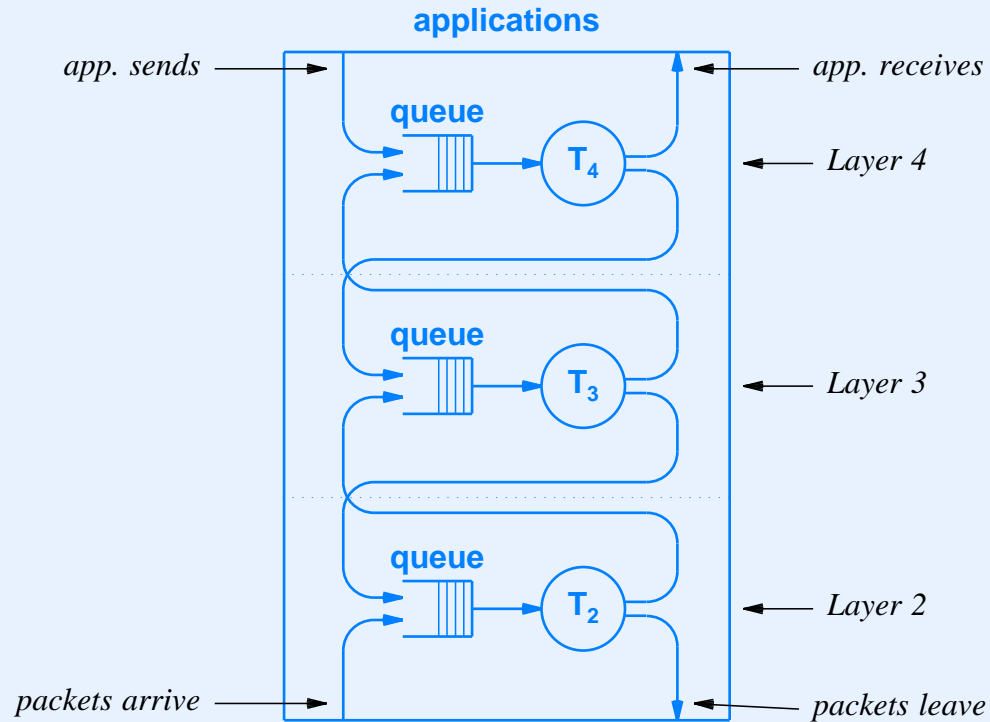
# Possible Organization Of Kernel Threads For Layered Protocols

- One thread per layer
- One thread per protocol
- Multiple threads per protocol
- Multiple threads per protocol plus timer management thread(s)
- One thread per packet

# One Thread Per Layer

- Easy for programmer to understand
- Implementation matches concept
- Allows priority to be assigned to each layer
- Means packet is enqueued once per layer

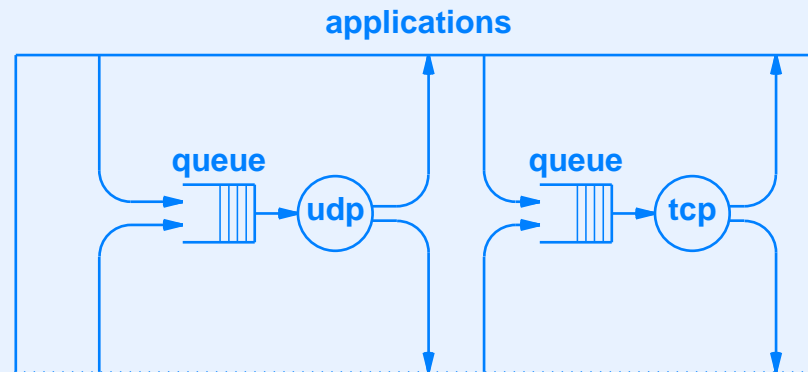
# Illustration Of One Thread Per Layer



# One Thread Per Protocol

- Like one thread per layer
  - Implementation matches concept
  - Means packet is enqueued once per layer
- Advantages over one thread per layer
  - Easier for programmer to understand
  - Finer-grain control
  - Allows priority to be assigned to each protocol

# Illustration Of One Thread Per Protocol

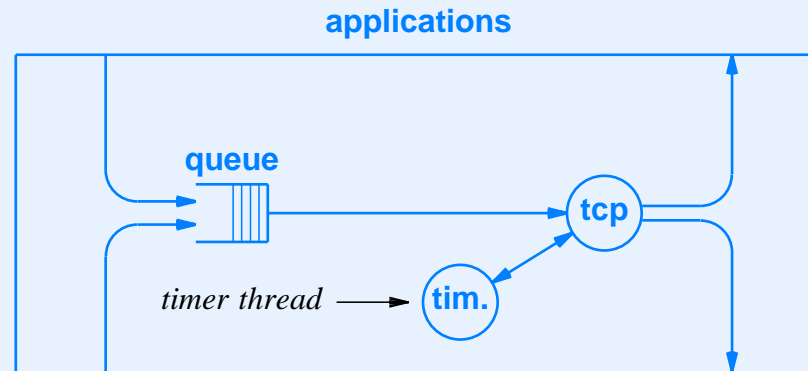


- TCP and UDP reside at same layer
- Separation allows priority

# Multiple Threads Per Protocol

- Further division of duties
- Simplifies programming
- More control than single thread
- Typical division
  - Thread for incoming packets
  - Thread for outgoing packets
  - Thread for management/timing

# Illustration Of Multiple Threads Used With TCP



- Separate timer makes programming easier

# Timers And Protocols

- Many protocols implement timeouts
  - TCP
    - \* Retransmission timeout
    - \* 2MSL timeout
  - ARP
    - \* Cache entry timeout
  - IP
    - \* Reassembly timeout



# Multiple Threads Per Protocol Plus Timer Management Thread(s)

- Observations
  - Many protocols each need timer functionality
  - Each timer thread incurs overhead
- Solution: consolidate timers for multiple protocols

# Is One Timer Thread Sufficient?

- In theory
  - Yes
- In practice
  - Large range of timeouts (microseconds to tens of seconds)
  - May want to give priority to some timeouts
- Solution: two or more timer threads

# Multiple Timer Threads

- Two threads usually suffice
- Large-granularity timer
  - Values specified in seconds
  - Operates at lower priority
- Small-granularity timer
  - Values specified in microseconds
  - Operates at higher priority

# Thread Synchronization

- Thread for layer  $i$ 
  - Needs to pass a packet to layer  $i + 1$
  - Enqueues the packet
- Thread for layer  $i + 1$ 
  - Retrieves packet from the queue

# Thread Synchronization

- Thread for layer  $i$ 
  - Needs to pass a packet to layer  $i + 1$
  - Enqueues the packet
- Thread for layer  $i + 1$ 
  - Retrieves packet from the queue
- **Context switch required!**

# Context Switch

- OS function
- CPU passes from current thread to a waiting thread
- High cost
- Must be minimized

# One Thread Per Packet

- Preallocate set of threads
- Thread operation
  - Waits for packet to arrive
  - Moves through protocol stack
  - Returns to wait for next packet
- Minimizes context switches

# Summary

- Packet processing software usually runs in OS
- API can be synchronous or asynchronous
- Priorities achieved with
  - Software interrupts
  - Threads
- Variety of thread architectures possible





**Questions?**

# VIII

## **Hardware Architectures For Protocol Processing And Aggregate Rates**

# A Brief History Of Computer Hardware

- 1940s
  - Beginnings
- 1950s
  - Consolidation of von Neumann architecture
  - I/O controlled by CPU
- 1960s
  - I/O becomes important
  - Evolution of third generation architecture with interrupts

# I/O Processing

- Evolved from after-thought to central influence
- Low-end systems (e.g., microcontrollers)
  - Dumb I/O interfaces
  - CPU does all the work (polls devices)
  - Single, shared memory
  - Low cost, but low speed

# I/O Processing

## (continued)

- Mid-range systems (e.g., minicomputers)
  - Single, shared memory
  - I/O interfaces contain logic for transfer and status operations
  - CPU
    - \* Starts device then resumes processing
  - Device
    - \* Transfers data to / from memory
    - \* Interrupts when operation complete

# I/O Processing (continued)

- High-end systems (e.g., mainframes)
  - Separate, programmable I/O processor
  - OS downloads code to be run
  - Device has private on-board buffer memory
  - Examples: IBM channel, CDC peripheral processor

# Networking Systems Evolution

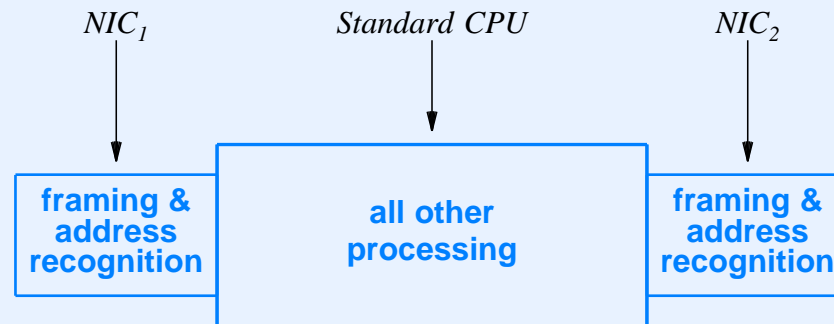
- Twenty year history
- Same trend as computer architecture
  - Began with central CPU
  - Shift to emphasis on I/O
- Three main generations

# First Generation Network Systems

- Traditional software-based router
- Used conventional (minicomputer) hardware
  - Single general-purpose processor
  - Single shared memory
  - I/O over a bus
  - Network interface cards use same design as other I/O devices



# Protocol Processing In First Generation Network Systems



- General-purpose processor handles most tasks
- Sufficient for low-speed systems
- Note: we will examine other generations later in the course

# How Fast Does A CPU Need To Be?

- Depends on
  - Rate at which data arrives
  - Amount of processing to be performed

# Two Measures Of Speed

- Data rate (bits per second)
  - Per interface rate
  - Aggregate rate
- Packet rate (packets per second)
  - Per interface rate
  - Aggregate rate

# How Fast Is A Fast Connection?

- Definition of fast data rate keeps changing
  - 1960: 10 Kbps
  - 1970: 1 Mbps
  - 1980: 10 Mbps
  - 1990: 100 Mbps
  - 2000: 1000 Mbps (1 Gbps)
  - 2004: 2400 Mbps

# How Fast Is A Fast Connection?

- Definition of fast data rate keeps changing
  - 1960: 10 Kbps
  - 1970: 1 Mbps
  - 1980: 10 Mbps
  - 1990: 100 Mbps
  - 2000: 1000 Mbps (1 Gbps)
  - 2004: 2400 Mbps
  - **Soon: 10 Gbps???**

# Aggregate Rate Vs. Per-Interface Rate

- Interface rate
  - Rate at which data enters / leaves
- Aggregate
  - Sum of interface rates
  - Measure of total data rate system can handle
- Note: aggregate rate crucial if CPU handles traffic from all interfaces

## A Note About System Scale

*The aggregate data rate is defined to be the sum of the rates at which traffic enters or leaves a system. The maximum aggregate data rate of a system is important because it limits the type and number of network connections the system can handle.*

# Packet Rate Vs. Data Rate

- Sources of CPU overhead
  - Per-bit processing
  - Per-packet processing
- Interface hardware handles much of per-bit processing



## A Note About System Scale

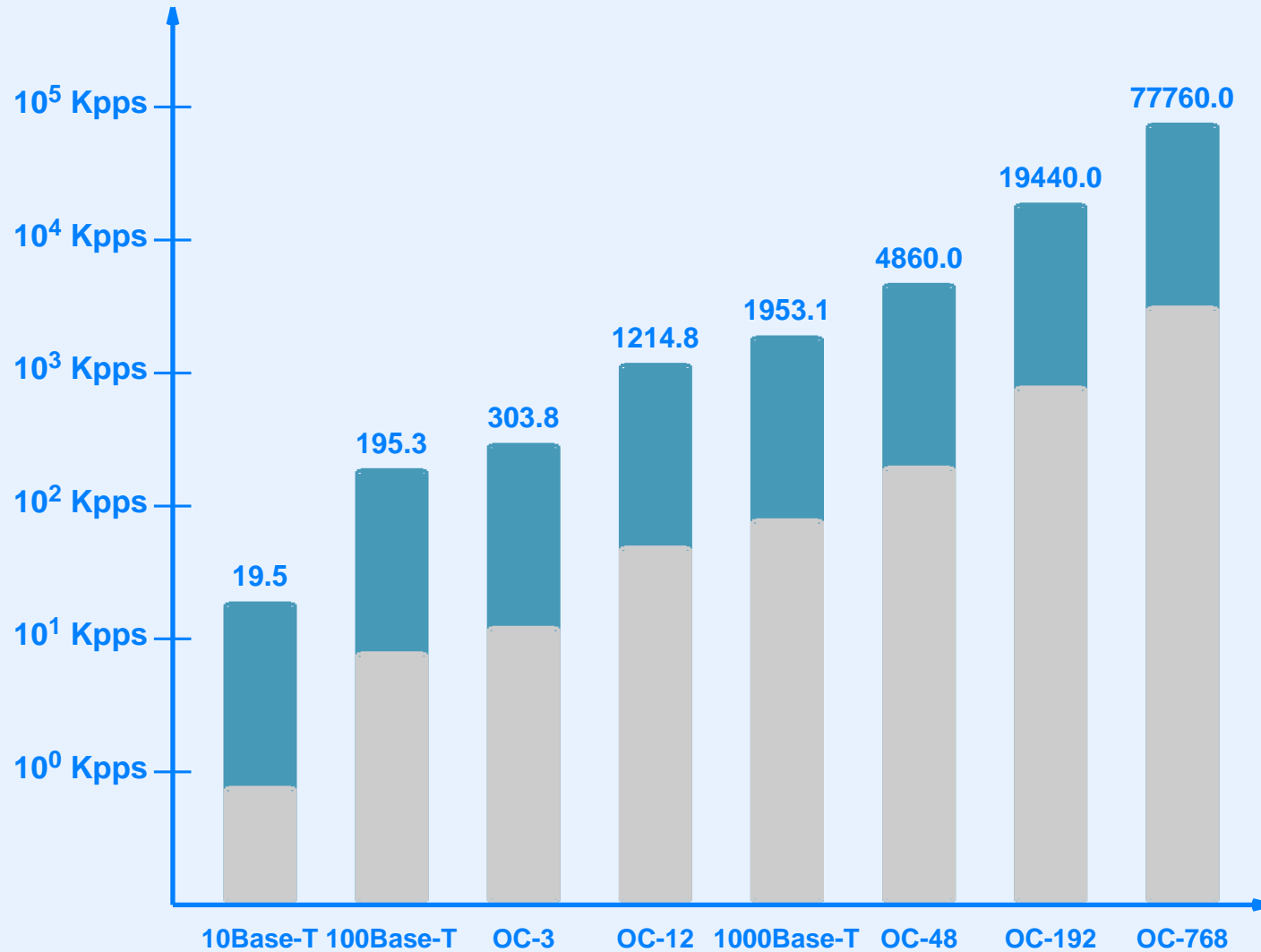
*For protocol processing tasks that have a fixed cost per packet, the number of packets processed is more important than the aggregate data rate.*

# Example Packet Rates

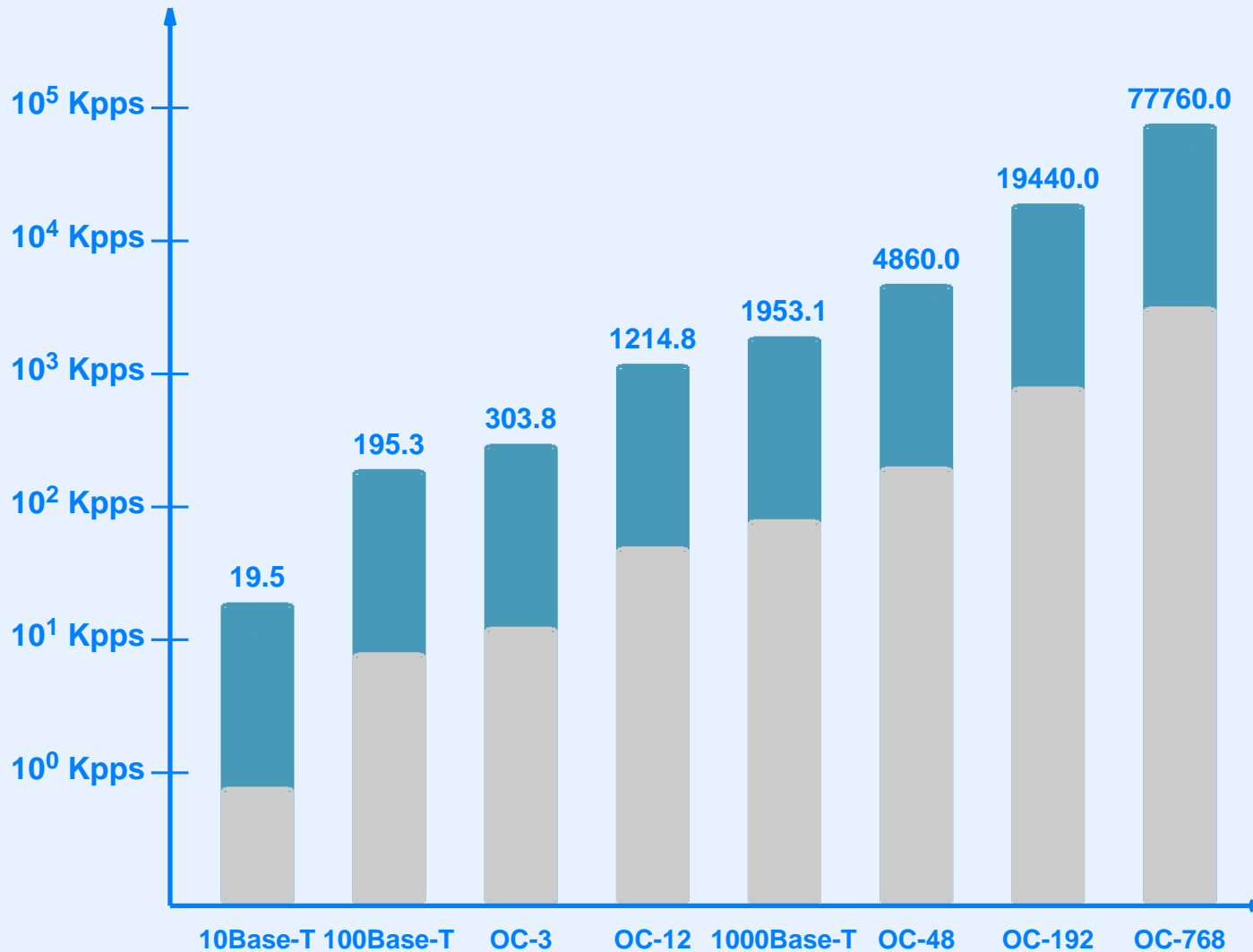
Technology	Network Data Rate In Gbps	Packet Rate For Small Packets In Kpps	Packet Rate For Large Packets In Kpps
10Base-T	0.010	19.5	0.8
100Base-T	0.100	195.3	8.2
OC-3	0.156	303.8	12.8
OC-12	0.622	1,214.8	51.2
1000Base-T	1.000	1,953.1	82.3
OC-48	2.488	4,860.0	204.9
OC-192	9.953	19,440.0	819.6
OC-768	39.813	77,760.0	3,278.4

- Key concept: maximum packet rate occurs with minimum-size packets

# Bar Chart Of Example Packet Rates



# Bar Chart Of Example Packet Rates



- Gray areas show rates for large packets

# Time Per Packet

Technology	Time per packet for small packets (in $\mu\text{s}$ )	Time per packet for large packets (in $\mu\text{s}$ )
10Base-T	51.20	1,214.40
100Base-T	5.12	121.44
OC-3	3.29	78.09
OC-12	0.82	19.52
1000Base-T	0.51	12.14
OC-48	0.21	4.88
OC-192	0.05	1.22
OC-768	0.01	0.31

- Note: these numbers are for a single connection!

# Conclusion

*Software running on a general-purpose processor is an insufficient architecture to handle high-speed networks because the aggregate packet rate exceeds the capabilities of a CPU.*

# Possible Ways To Solve The CPU Bottleneck

- Fine-grain parallelism
- Symmetric coarse-grain parallelism
- Asymmetric coarse-grain parallelism
- Special-purpose coprocessors
- NICs with onboard processing
- Smart NICs with onboard stacks
- Cell switching
- Data pipelines

# Fine-Grain Parallelism

- Multiple processors
- Instruction-level parallelism
- Example:
  - Parallel checksum: add values of eight consecutive memory locations at the same time
- Assessment: insignificant advantages for packet processing



# Symmetric Coarse-Grain Parallelism

- Symmetric multiprocessor hardware
  - Multiple, identical processors
- Typical design: each CPU operates on one packet
- Requires coordination
- Assessment: coordination and data access means  $N$  processors cannot handle  $N$  times more packets than one processor

# Asymmetric Coarse-Grain Parallelism

- Multiple processors
- Each processor
  - Optimized for specific task
  - Includes generic instructions for control
- Assessment
  - Same problems of coordination and data access as symmetric case
  - Designer must choose how many copies of each processor type

# Special-Purpose Coprocessors

- Special-purpose hardware
- Added to conventional processor to speed computation
- Invoked like software subroutine
- Typical implementation: ASIC chip
- Choose operations that yield greatest improvement in speed

# General Principle

*To optimize computation, move operations that account for the most CPU time from software into hardware.*

# General Principle

*To optimize computation, move operations that account for the most CPU time from software into hardware.*

- Idea known as *Amdahl's law* (performance improvement from faster hardware technology is limited to the fraction of time the faster technology can be used)

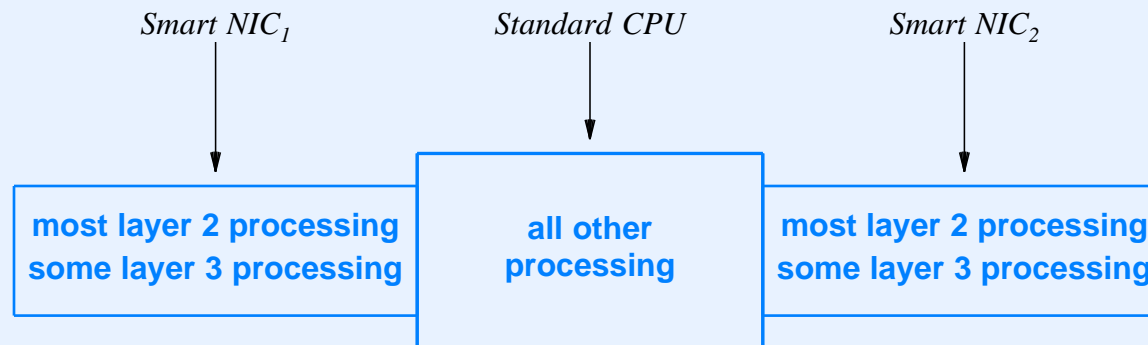
# NICs And Onboard Processing

- Basic optimizations
  - Onboard address recognition and filtering
  - Onboard buffering
  - DMA
  - Buffer and operation chaining
- Further optimization possible

# Smart NICs With Onboard Stacks

- Add hardware to NIC
  - Off-the-shelf chips for layer 2
  - ASICs for layer 3
- Allows each NIC to operate independently
  - Effectively a multiprocessor
  - Total processing power increased dramatically

# Illustration Of Smart NICs With Onboard Processing



- NIC handles layers 2 and 3
- CPU only handles exceptions



# Cell Switching

- Alternative to new hardware
- Changes
  - Basic paradigm
  - All details (e.g., protocols)
- Connection-oriented

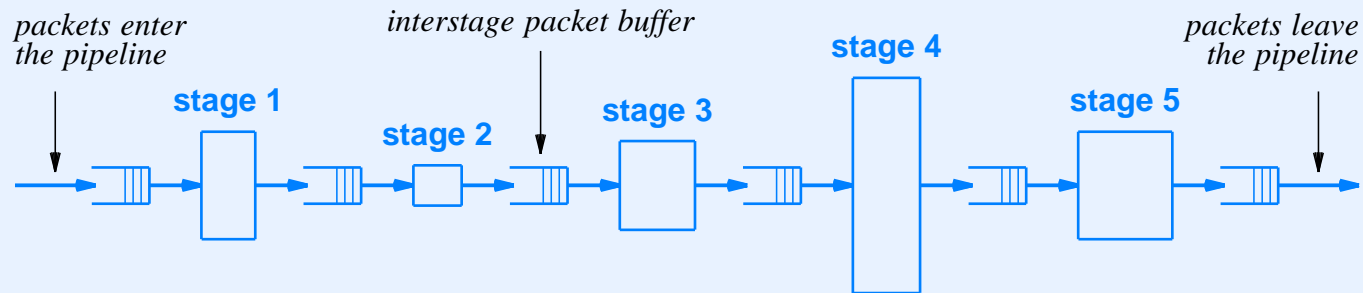
# Cell Switching Details

- Fixed-size packets
  - Allows fixed-size buffers
  - Guaranteed time to transmit/receive
- Relative (connection-oriented) addressing
  - Smaller address size
  - Label on packet changes at each switch
  - Requires connection setup
- Example: ATM

# Data Pipeline

- Move each packet through series of processors
- Each processor handles some tasks
- Assessment
  - Well-suited to many protocol processing tasks
  - Individual processor can be fast

# Illustration Of Data Pipeline



- Pipeline can contain heterogeneous processors
- Packets pass through each stage

# Summary

- Packet rate can be more important than data rate
- Highest packet rate achieved with smallest packets
- Rates measured per interface or aggregate
- Special hardware needed for highest-speed network systems
  - Smart NIC can include part of protocol stack
  - Parallel and pipelined hardware also possible



**Questions?**

# **IX**

## **Classification And Forwarding**

# Recall

- Packet demultiplexing
  - Used with layered protocols
  - Packet proceeds through one layer at a time
  - On input, software in each layer chooses module at next higher layer
  - On output, type field in each header specifies encapsulation



# The Disadvantage Of Demultiplexing

*Although it provides freedom to define and use arbitrary protocols without introducing transmission overhead, demultiplexing is inefficient because it imposes sequential processing among layers.*

# Packet Classification

- Alternative to demultiplexing
- Designed for higher speed
- Considers all layers at the same time
- Linear in number of fields
- Two possible implementations
  - Software
  - Hardware

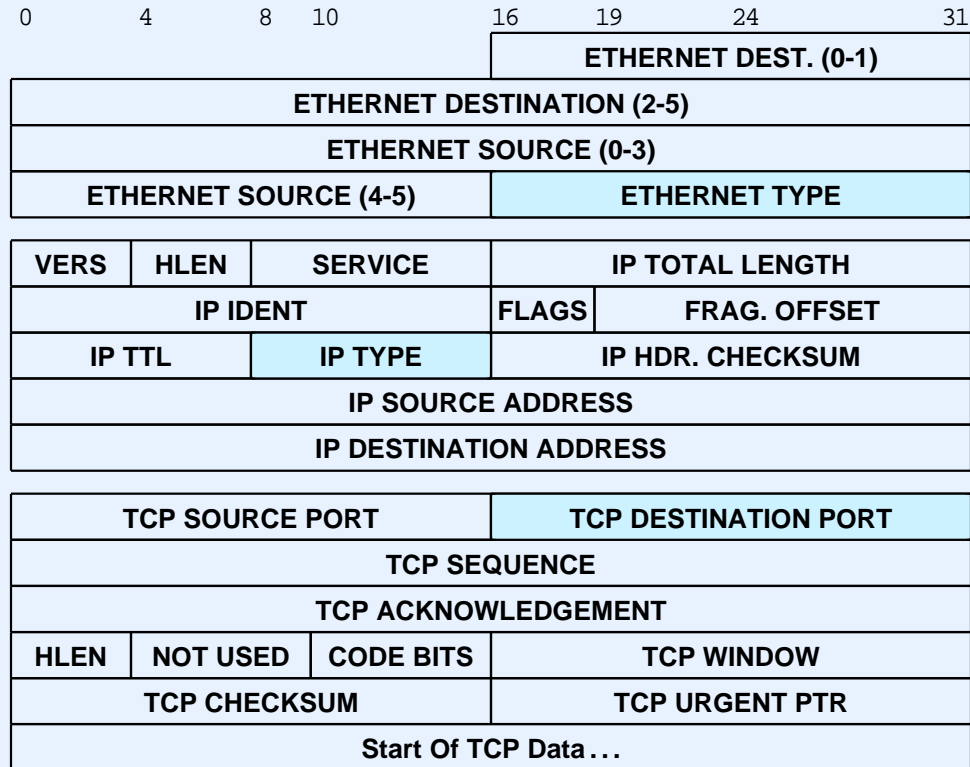
# Example Classification

- Classify Ethernet frames carrying traffic to Web server
- Specify exact header contents in *rule set*
- Example
  - Ethernet type field specifies IP
  - IP type field specifies TCP
  - TCP destination port specifies Web server

# Example Classification (continued)

- Field sizes and values
  - 2-octet Ethernet type is  $0800_{16}$
  - 1-octet IP type is 6
  - 2-octet TCP destination port is 80

# Illustration Of Encapsulated Headers



- Highlighted fields are used for classification of Web server traffic

# Software Implementation Of Classification

- Compare values in header fields
- Conceptually a *logical and* of all field comparisons
- Example

```
if ( (frame type == 0x0800) && (IP type == 6) && (TCP port == 80) )  
    declare the packet matches the classification;  
else  
    declare the packet does not match the classification;
```

# Optimizing Software Classification

- Comparisons performed sequentially
- Can reorder comparisons to minimize effort

# Example Of Optimizing Software Classification

- Assume
  - 95.0% of all frames have frame type  $0800_{16}$
  - 87.4% of all frames have IP type 6
  - 74.3% of all frames have TCP port 80
- Also assume values 6 and 80 do not occur in corresponding positions in non-IP packet headers
- Reordering tests can optimize processing time



# Example Of Optimizing Software Classification (continued)

```
if ((TCP port == 80) && (IP type == 6) && (frame type == 0x0800))  
    declare the packet matches the classification;  
else  
    declare the packet does not match the classification;
```

- At each step, test the field that will eliminate the most packets

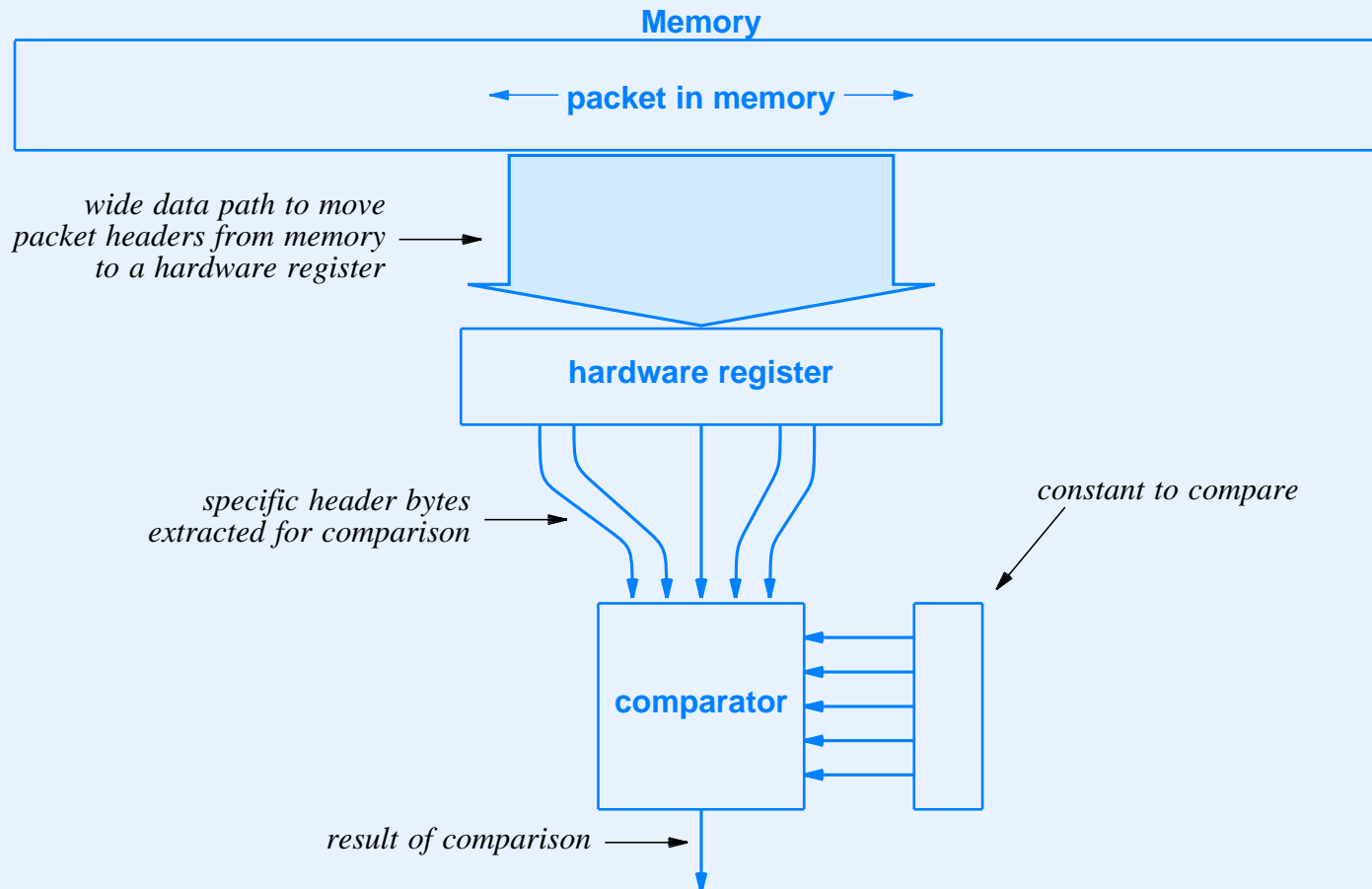
## Note About Optimization

*Although the maximum number of comparisons in a software classifier is fixed, the average number of comparisons is determined by the order of the tests; minimum comparisons result if, at each step, the classifier tests the field that eliminates the most packets.*

# Hardware Implementation Of Classification

- Can build special-purpose hardware
- Steps
  - Extract needed fields
  - Concatenate bits
  - Place result in register
  - Perform comparison
- Hardware can operate in parallel

# Illustration Of Hardware Classifier



- Constant for Web classifier is  $08.00.06.00.50_{16}$

# Special Cases Of Classification

- Multiple categories
- Variable-size headers
- Dynamic classification

# In Practice

- Classification usually involves multiple categories
- Packets grouped together into *flows*
- May have a default category
- Each category specified with rule set

# Example Multi-Category Classification

- Flow 1: traffic destined for Web server
- Flow 2: traffic consisting of ICMP echo request packets
- Flow 3: all other traffic (default)

# Rule Sets

- Web server traffic
  - 2-octet Ethernet type is  $0800_{16}$
  - 1-octet IP type is 6
  - 2-octet TCP destination port is 80
- ICMP echo traffic
  - 2-octet Ethernet type is  $0800_{16}$
  - 1-octet IP type is 1
  - 1-octet ICMP type is 8



# Software Implementation Of Multiple Rules

```
if (frame type != 0x0800) {
    send frame to flow 3;
} else if (IP type == 6 && TCP destination port == 80) {
    send packet to flow 1;
} else if (IP type == 1 && ICMP type == 8) {
    send packet to flow 2;
} else {
    send frame to flow 3;
}
```

- Further optimization possible

# Variable-Size Packet Headers

- Fields not at fixed offsets
- Easily handled with software
- Finite cases can be specified in rules

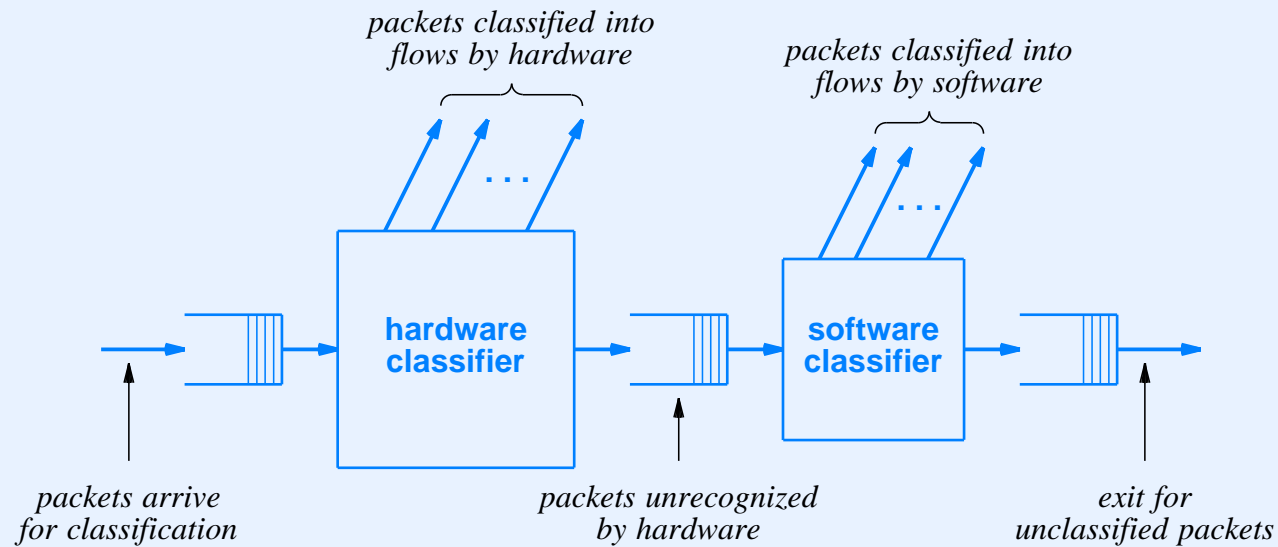
# Example Variable-Size Header: IP Options

- Rule Set 1
  - 2-octet frame type field contains  $0800_{16}$
  - 1-octet field at the start of the datagram contains  $45_{16}$
  - 1-octet type field in the IP datagram contains 6
  - 2-octet field 22 octets from start of the datagram contains 80
- Rule Set 2
  - 2-octet frame type field contains  $0800_{16}$
  - 1-octet field at the start of the datagram contains  $46_{16}$
  - 1-octet type field in the IP datagram contains 6
  - 2-octet field 26 octets from the start of datagram contains 80

# Effect Of Protocol Design On Classification

- Fixed headers fastest to classify
- Each variable-size header adds one computation step
- In worst case, classification no faster than demultiplexing
- Extreme example: IPv6

# Hybrid Classification



- Combines hardware and software mechanisms
  - Hardware used for standard cases
  - Software used for exceptions
- Note: software classifier can operate at slower rate

# Two Basic Types Of Classification

- Static
  - Flows specified in rule sets
  - Header fields and values known a priori
- Dynamic
  - Flows created by observing packet stream
  - Values taken from headers
  - Allows fine-grain flows
  - Requires state information

# Example Static Classification

- Allocate one flow per service type
- One header field used to identify flow
  - IP TYPE OF SERVICE (TOS)
- Use DIFFSERV interpretation
- Note: Ethernet type field also checked

# Example Dynamic Classification

- Allocate flow per TCP connection
- Header fields used to identify flow
  - IP source address
  - IP destination address
  - TCP source port number
  - TCP destination port number
- Note: Ethernet type and IP type fields also checked



# Implementation Of Dynamic Classification

- Usually performed in software
- State kept in memory
- State information created/updated at wire speed

# Two Conceptual Bindings

*classification: packet  $\rightarrow$  flow*

*forwarding: flow  $\rightarrow$  packet disposition*

- Classification binding is usually 1-to-1
- Forwarding binding can be 1-to-1 or many-to-1

# Flow Identification

- Connection-oriented network
  - Per-flow SVC can be created on demand
  - Flow ID equals connection ID
- Connectionless network
  - Flow ID used internally
  - Each flow ID mapped to ( *next hop, interface* )

# Relationship Of Classification And Forwarding In A Connection-Oriented Network

*In a connection-oriented network, flow identifiers assigned by classification can be chosen to match connection identifiers used by the underlying network. Doing so makes forwarding more efficient by eliminating one binding.*

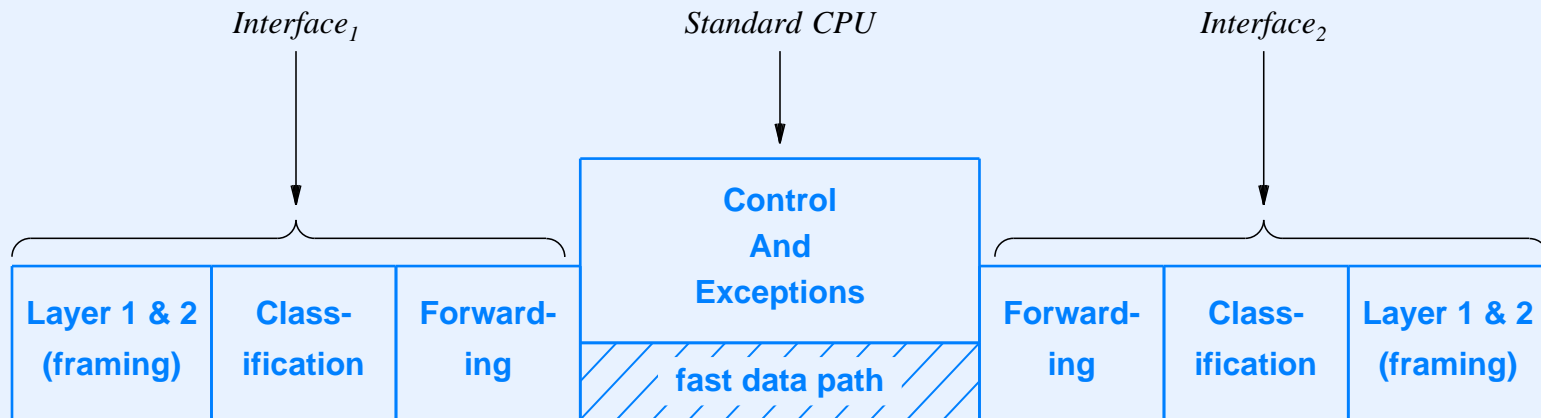
# Forwarding In A Connectionless Network

- Route for flow determined when flow created
- Indexing used in place of route lookup
- Flow identifier corresponds to index of entry in forwarding cache
- Forwarding cache must be changed when route changes

# Second Generation Network Systems

- Designed for greater scale
- Use classification instead of demultiplexing
- Decentralized architecture
  - Additional computational power on each NIC
  - NIC implements classification and forwarding
- High-speed internal interconnection mechanism
  - Interconnects NICs
  - Provides *fast data path*

# Illustration Of Second Generation Network Systems Architecture



# Classification And Forwarding Chips

- Sold by vendors
- Implement hardware classification and forwarding
- Typical configuration: rule sets given in ROM



# Summary

- Classification faster than demultiplexing
- Can be implemented in hardware or software
- Dynamic classification
  - Uses packet contents to assign flows
  - Requires state information



**Questions?**

# XI

## **Network Processors: Motivation And Purpose**

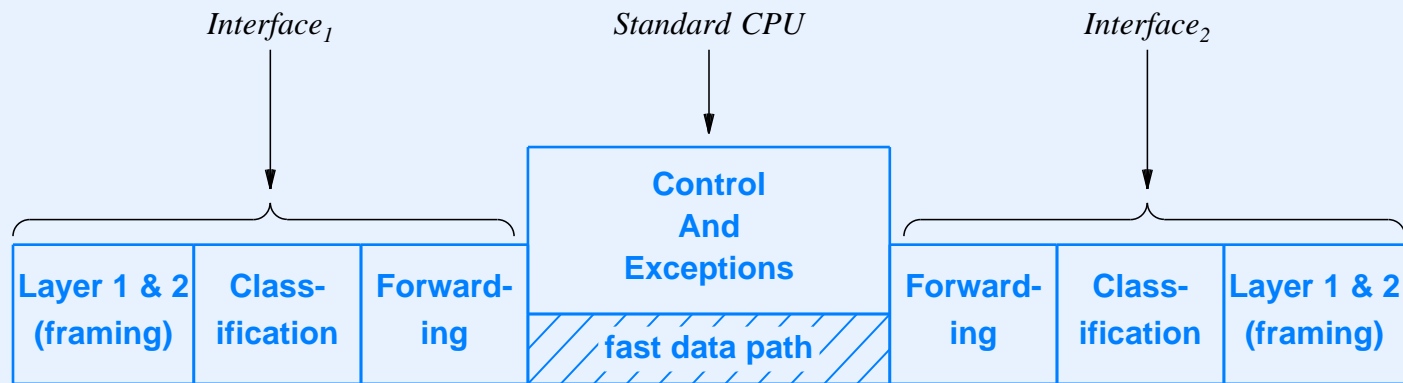
# Second Generation Network Systems

- Concurrent with ATM development (early 1990s)
- Purpose: scale to speeds faster than single CPU capacity
- Features
  - Use classification instead of demultiplexing
  - Decentralized architecture to offload CPU
  - Design optimized for fast data path

# Second Generation Network Systems (details)

- Multiple network interfaces
  - Powerful NIC
  - Private buffer memory
- High-speed hardware interconnects NICs
- General-purpose processor only handles exceptions
- Sufficient for medium speed interfaces (100 Mbps)

# Reminder: Protocol Processing In Second Generation Network Systems



- NIC handles most of layers 1 - 3
- Fast-path forwarding avoids CPU completely

# Third Generation Network Systems

- Late 1990s
- Functionality partitioned further
- Additional hardware on each NIC
- Almost all packet processing off-loaded from CPU

# Third Generation Design

- NIC contains
  - ASIC hardware
  - Embedded processor plus code in ROM
- NIC handles
  - Classification
  - Forwarding
  - Traffic policing
  - Monitoring and statistics



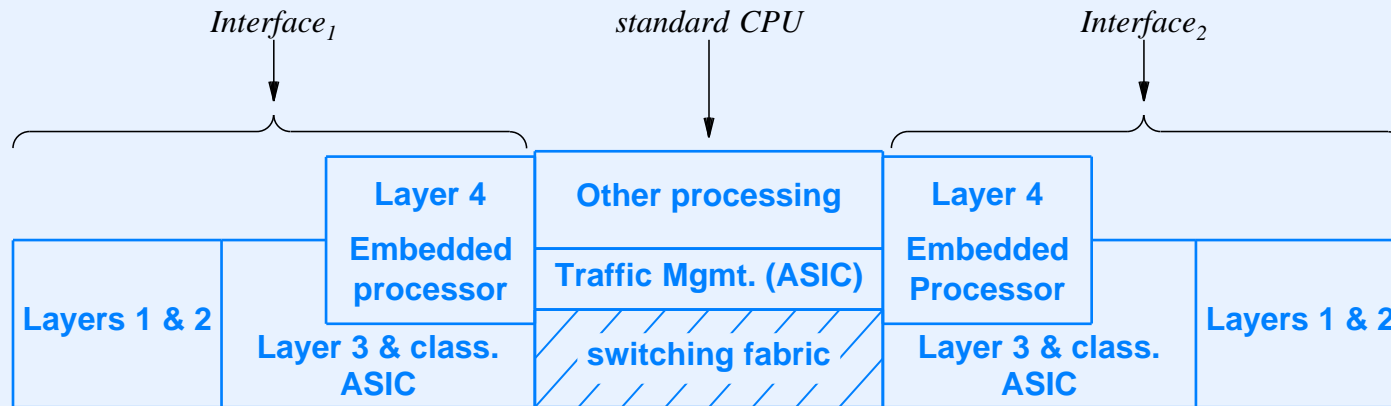
# Embedded Processor

- Two possibilities
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)
- RISC used often because
  - Higher clock rates
  - Smaller
  - Lower power consumption

# Purpose Of Embedded Processor In Third Generation Systems

*Third generation systems use an embedded processor to handle layer 4 functionality and exception packets that cannot be forwarded across the fast path. An embedded processor architecture is chosen because ease of implementation and amenability to change are more important than speed.*

# Protocol Processing In Third Generation Systems



- Special-purpose ASICs handle lower layer functions
- Embedded (RISC) processor handles layer 4
- CPU only handles low-demand processing

# Are Third Generation Systems Sufficient?

# Are Third Generation Systems Sufficient?

- Almost

# Are Third Generation Systems Sufficient?

- Almost ... but not quite.

# Problems With Third Generation Systems

- High cost
- Long time to market
- Difficult to simulate/test
- Expensive and time-consuming to change
  - Even trivial changes require silicon respin
  - 18-20 month development cycle
- Little reuse across products
- Limited reuse across versions

# Problems With Third Generation Systems (continued)

- No consensus on overall framework
- No standards for special-purpose support chips
- Requires in-house expertise (ASIC designers)



# A Fourth Generation

- Goal: combine best features of first generation and third generation systems
  - Flexibility of programmable processor
  - High speed of ASICs
- Technology called *network processors*

# Definition Of A Network Processor

*A network processor is a special-purpose, programmable hardware device that combines the low cost and flexibility of a RISC processor with the speed and scalability of custom silicon (i.e., ASIC chips). Network processors are building blocks used to construct network systems.*

# Network Processors: Potential Advantages

- Relatively low cost
- Straightforward hardware interface
- Facilities to access
  - Memory
  - Network interface devices
- Programmable
- Ability to scale to higher
  - Data rates
  - Packet rates

# Network Processors: Potential Advantages

- Relatively low cost
- Straightforward hardware interface
- Facilities to access
  - Memory
  - Network interface devices
- **Programmable**
- Ability to scale to higher
  - Data rates
  - Packet rates

# The Promise Of Programmability

- For producers
  - Lower initial development costs
  - Reuse software in later releases and related systems
  - Faster time-to-market
  - Same high speed as ASICs
- For consumers
  - Much lower product cost
  - Inexpensive (firmware) upgrades

# Choice Of Instruction Set

- Programmability alone insufficient
- Also need higher speed
- Should network processors have
  - Instructions for specific protocols?
  - Instructions for specific protocol processing tasks?
- Choices difficult

# Instruction Set

- Need to choose one instruction set
- No single instruction set best for all uses
- Other factors
  - Power consumption
  - Heat dissipation
  - Cost
- More discussion later in the course

# Scalability

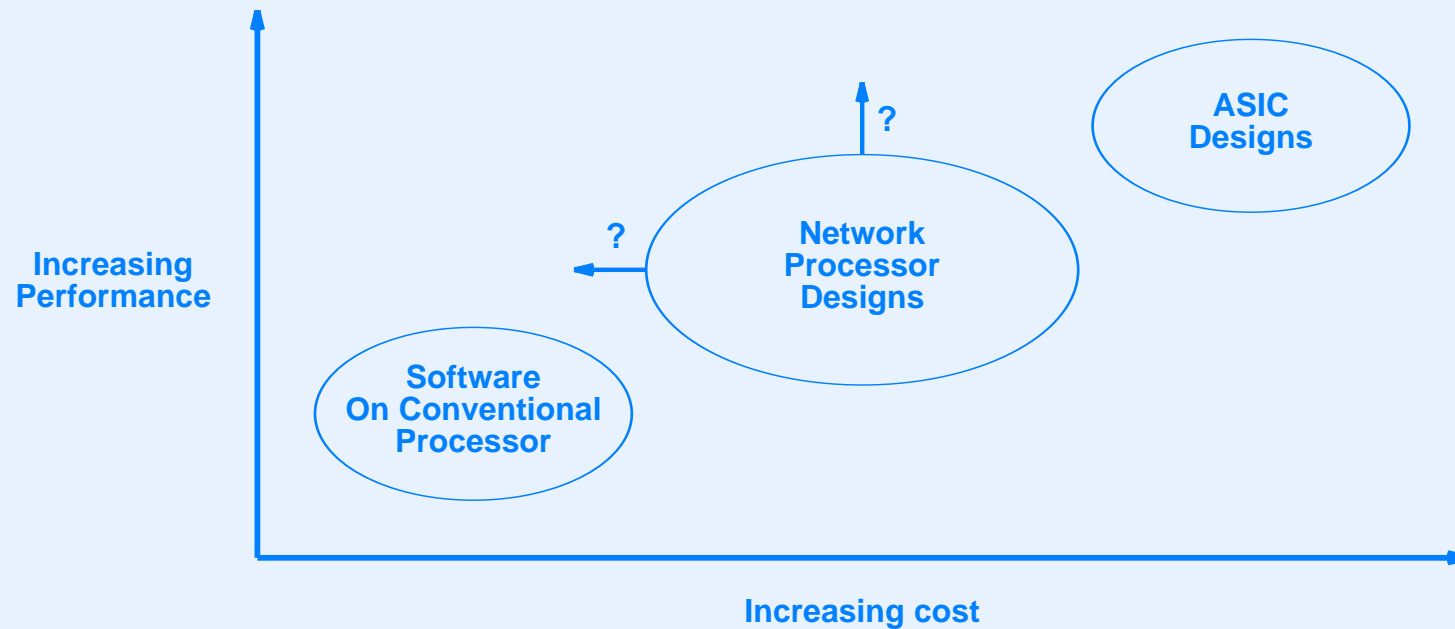
- Two primary techniques
  - Parallelism
  - Data pipelining
- Questions
  - How many processors?
  - How should they be interconnected?
- More discussion later



# Costs And Benefits Of Network Processors

- Currently
  - More expensive than conventional processor
  - Slower than ASIC design
- Where do network processors fit?
  - Somewhere in the middle

# Where Network Processors Fit



- Network processors: the middle ground

# Achieving Higher Speed

- What is known
  - Must partition packet processing into separate functions
  - To achieve highest speed, must handle each function with separate hardware
- What is unknown
  - Exactly what functions to choose
  - Exactly what hardware building blocks to use
  - Exactly how building blocks should be interconnected

# Variety Of Network Processors

- Economics driving a gold rush
  - NPs will dramatically lower production costs for network systems
  - A good NP design potentially worth lots of \$\$
- Result
  - Wide variety of architectural experiments
  - Wild rush to try yet another variation

# An Interesting Observation

- System developed using ASICs
  - High development cost (\$1M)
  - Lower cost to replicate
- System developed using network processors
  - Lower development cost
  - Higher cost to replicate
- Conclusion: amortized cost favors ASICs for most high-volume systems

# Summary

- Third generation network systems have embedded processor on each NIC
- Network processor is programmable chip with facilities to process packets faster than conventional processor
- Primary motivation is economic
  - Lower development cost than ASICs
  - Higher processing rates than conventional processor



**Questions?**

# **XII**

## **The Complexity Of Network Processor Design**



# How Should A Network Processor Be Designed?

- Depends on
  - Operations network processor will perform
  - Role of network processor in overall system

# Goals

- Generality
  - Sufficient for all protocols
  - Sufficient for all protocol processing tasks
  - Sufficient for all possible networks
- High speed
  - Scale to high bit rates
  - Scale to high packet rates
- Elegance
  - Minimality, not merely comprehensiveness

# The Key Point

*A network processor is not designed to process a specific protocol or part of a protocol. Instead, designers seek a minimal set of instructions that are sufficient to handle an arbitrary protocol processing task at high speed.*

# Network Processor Design

- To understand network processors, consider problem to be solved
  - Protocols being implemented
  - Packet processing tasks

# Packet Processing Functions

- Error detection and correction
- Traffic measurement and policing
- Frame and protocol demultiplexing
- Address lookup and packet forwarding
- Segmentation, fragmentation, and reassembly
- Packet classification
- Traffic shaping
- Timing and scheduling
- Queueing
- Security: authentication and privacy

# Questions

- Does our list of functions encompass all protocol processing?
- Which function(s) are most important to optimize?
- How do the functions map onto hardware units in a typical network system?
- Which hardware units in a network system can be replaced with network processors?
- What minimal set of instructions is sufficiently general to implement all functions?

# Division Of Functionality

- Partition problem to reduce complexity
- Basic division into two parts
- Functions applied when packet arrives known as  
*ingress processing*
- Functions applied when packet leaves known as  
*egress processing*

# Ingress Processing

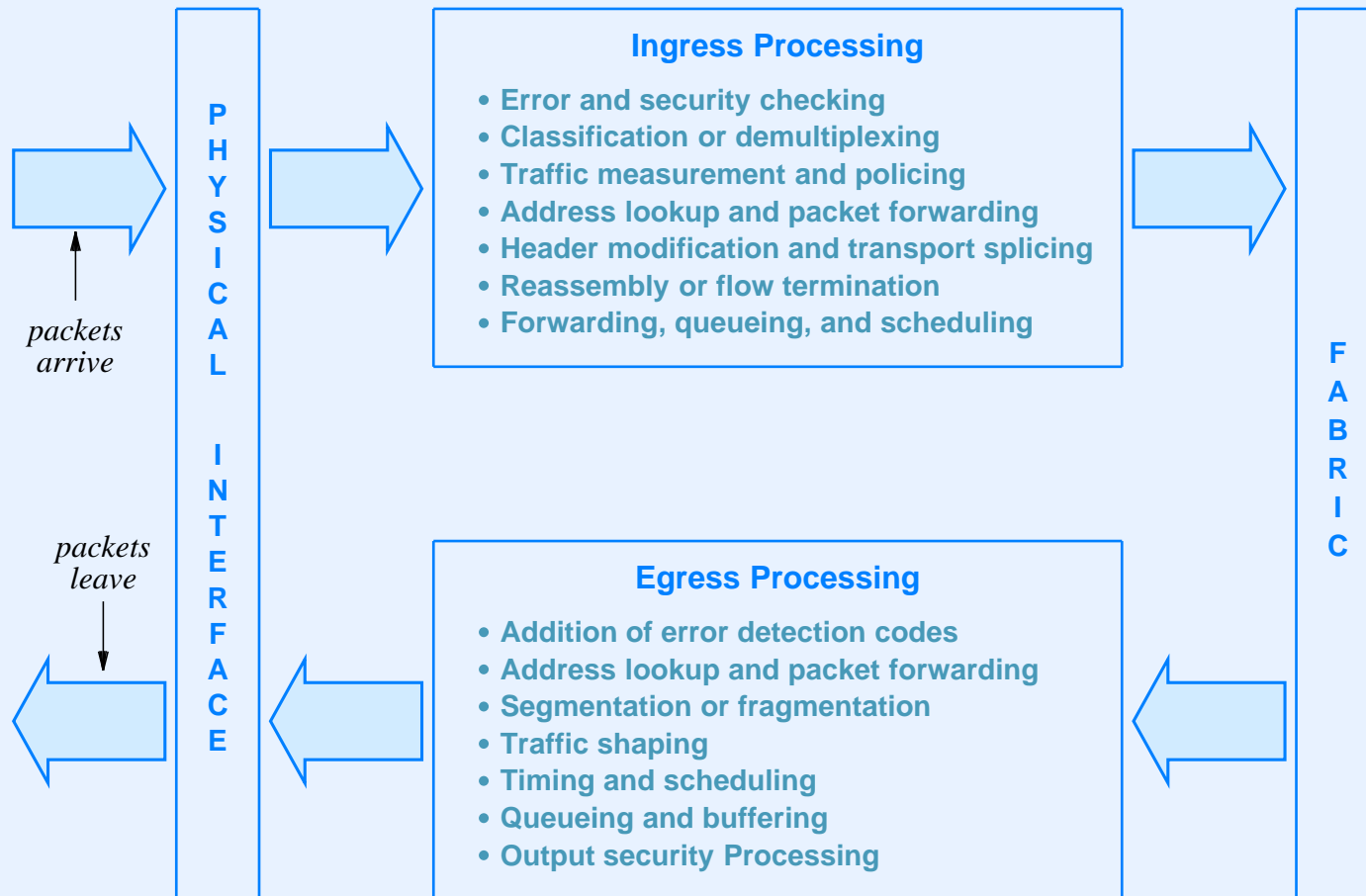
- Security and error detection
- Classification or demultiplexing
- Traffic measurement and policing
- Address lookup and packet forwarding
- Header modification and transport splicing
- Reassembly or flow termination
- Forwarding, queueing, and scheduling



# Egress Processing

- Addition of error detection codes
- Address lookup and packet forwarding
- Segmentation or fragmentation
- Traffic shaping
- Timing and scheduling
- Queueing and buffering
- Output security processing

# Illustration Of Packet Flow



# A Note About Scalability

*Unlike a conventional processor, scalability is essential for network processors. To achieve maximum scalability, a network processor offers a variety of special-purpose functional units, allows parallel or pipelined execution, and operates in a distributed environment.*

# How Will Network Processors Be Used?

- For ingress processing only?
- For egress processing only?
- For combination?

# How Will Network Processors Be Used?

- For ingress processing only?
- For egress processing only?
- For combination?
- Answer: No single role

# Potential Architectural Roles For Network Processor

- Replacement for a conventional CPU
- Augmentation of a conventional CPU
- On the input path of a network interface card
- Between a network interface card and central interconnect
- Between central interconnect and an output interface
- On the output path of a network interface card
- Attached to central interconnect like other ports

# An Interesting Potential Role For Network Processors

*In addition to replacing elements of a traditional third generation architecture, network processors can be attached directly to a central interconnect and used to implement stages of a macroscopic data pipeline. The interconnect allows forwarding among stages to be optimized.*

# Conventional Processor Design

- Design an instruction set,  $S$
- Build an emulator/simulator for  $S$  in software
- Build a compiler that translates into  $S$
- Compile and emulate example programs
- Compare results to
  - Extant processors
  - Alternative designs



# Network Processor Emulation

- Can emulate low-level logic (e.g., Verilog)
- Software implementation
  - Slow
  - Cannot handle real packet traffic
- FPGA implementation
  - Expensive and time-consuming
  - Difficult to make major changes

# Network Processor Design

- Unlike conventional processor design
- No existing code base
- No prior hardware experience
- Each design differs

# Hardware And Software Design

*Because a network processor includes many low-level hardware details that require specialized software, the hardware and software designs are codependent; software for a network processor must be created along with the hardware.*

# Summary

- Protocol processing divided into ingress and egress operations
- Network processor design is challenging because
  - Desire generality and efficiency
  - No existing code base
  - Software designs evolving with hardware



# **XIII**

## **Network Processor Architectures**

# Architectural Explosion

*An excess of exuberance and a lack of experience have produced a wide variety of approaches and architectures.*

# Principle Components

- Processor hierarchy
- Memory hierarchy
- Internal transfer mechanisms
- External interface and communication mechanisms
- Special-purpose hardware
- Polling and notification mechanisms
- Concurrent and parallel execution support
- Programming model and paradigm
- Hardware and software dispatch mechanisms



# Processing Hierarchy

- Consists of hardware units
- Performs various aspects of packet processing
- Includes onboard and external processors
- Individual processor can be
  - Programmable
  - Configurable
  - Fixed

# Typical Processor Hierarchy

Level	Processor Type	Programmable?	On Chip?
8	General purpose CPU	yes	possibly
7	Embedded processor	yes	typically
5	I/O processor	yes	typically
6	Coprocessor	no	typically
4	Fabric interface	no	typically
3	Data transfer unit	no	typically
2	Framer	no	possibly
1	Physical transmitter	no	possibly

# Memory Hierarchy

- Memory measurements
  - Random access latency
  - Sequential access latency
  - Throughput
  - Cost
- Can be
  - Internal
  - External

# Typical Memory Hierarchy

Memory Type	Rel. Speed	Approx. Size	On Chip?
Control store	100	$10^3$	yes
G.P. Registers†	90	$10^2$	yes
Onboard Cache	40	$10^3$	yes
Onboard RAM	7	$10^3$	yes
Static RAM	2	$10^7$	no
Dynamic RAM	1	$10^8$	no

# Internal Transfer Mechanisms

- Internal bus
- Hardware FIFOs
- Transfer registers
- Onboard shared memory

# External Interface And Communication Mechanisms

- Standard and specialized bus interfaces
- Memory interfaces
- Direct I/O interfaces
- Switching fabric interface

# Example Interfaces

- System Packet Interface Level 3 or 4 (SPI-3 or SPI-4)
- SerDes Framing Interface (SFI)
- CSIX fabric interface

Note: The Optical Internetworking Forum (OIF) controls the SPI and SFI standards.

# Polling And Notification Mechanisms

- Handle asynchronous events
  - Arrival of packet
  - Timer expiration
  - Completion of transfer across the fabric
- Two paradigms
  - Polling
  - Notification



# Concurrent Execution Support

- Improves overall throughput
- Multiple threads of execution
- Processor switches context when a thread blocks

# Support For Concurrent Execution

- Embedded processor
  - Standard operating system
  - Context switching in software
- I/O processors
  - No operating system
  - Hardware support for context switching
  - Low-overhead or zero-overhead

# Concurrent Support Questions

- Local or global threads (does thread execution span multiple processors)?
- Forced or voluntary context switching (are threads preemptable)?

# Hardware And Software Dispatch Mechanisms

- Refers to overall control of parallel operations
- Dispatcher
  - Chooses operation to perform
  - Assigns to a processor

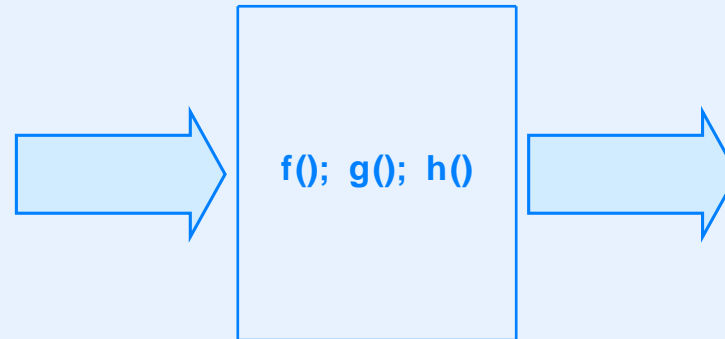
# Implicit And Explicit Parallelism

- Explicit parallelism
  - Exposes parallelism to programmer
  - Requires software to understand parallel hardware
- Implicit parallelism
  - Hides parallel copies of functional units
  - Software written as if single copy executing

# Architecture Styles, Packet Flow, And Clock Rates

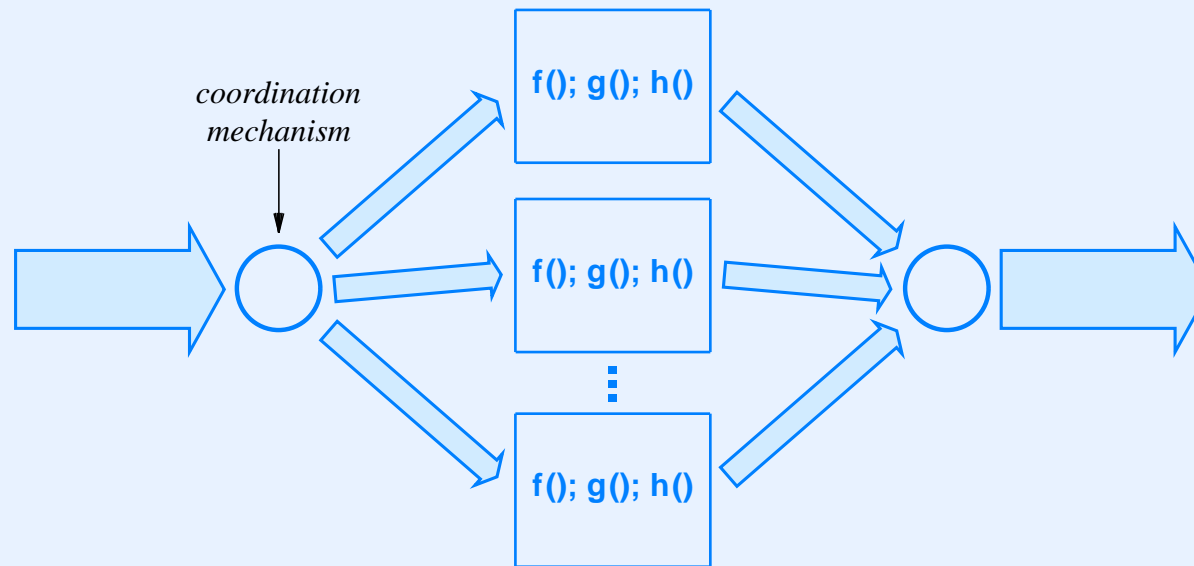
- Embedded processor plus fixed coprocessors
- Embedded processor plus programmable I/O processors
- Parallel (number of processors scales to handle load)
- Pipeline processors
- Dataflow

# Embedded Processor Architecture



- Single processor
  - Handles all functions
  - Passes packet on
- Known as run-to-completion

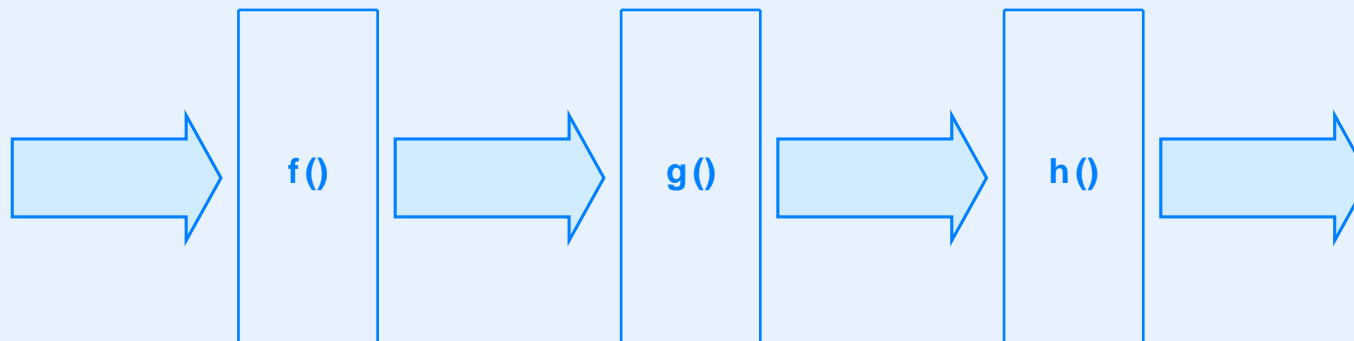
# Parallel Architecture



- Each processor handles  $1/N$  of total load



# Pipeline Architecture



- Each processor handles one function
- Packet moves through “pipeline”

# Clock Rates

- Embedded processor runs at  $>$  wire speed
- Parallel processor runs at  $<$  wire speed
- Pipeline processor runs at wire speed

# Software Architecture

- Central program that invokes coprocessors like subroutines
- Central program that interacts with code on intelligent, programmable I/O processors
- Communicating threads
- Event-driven program
- RPC-style (program partitioned among processors)
- Pipeline (even if hardware does not use pipeline)
- Combinations of the above

# Example Uses Of Programmable Processors

## General purpose CPU

- Highest level functionality
- Administrative interface
- System control
- Overall management functions
- Routing protocols

## Embedded processor

- Intermediate functionality
- Higher-layer protocols
- Control of I/O processors
- Exception and error handling
- High-level ingress (e.g., reassembly)
- High-level egress (e.g., traffic shaping)

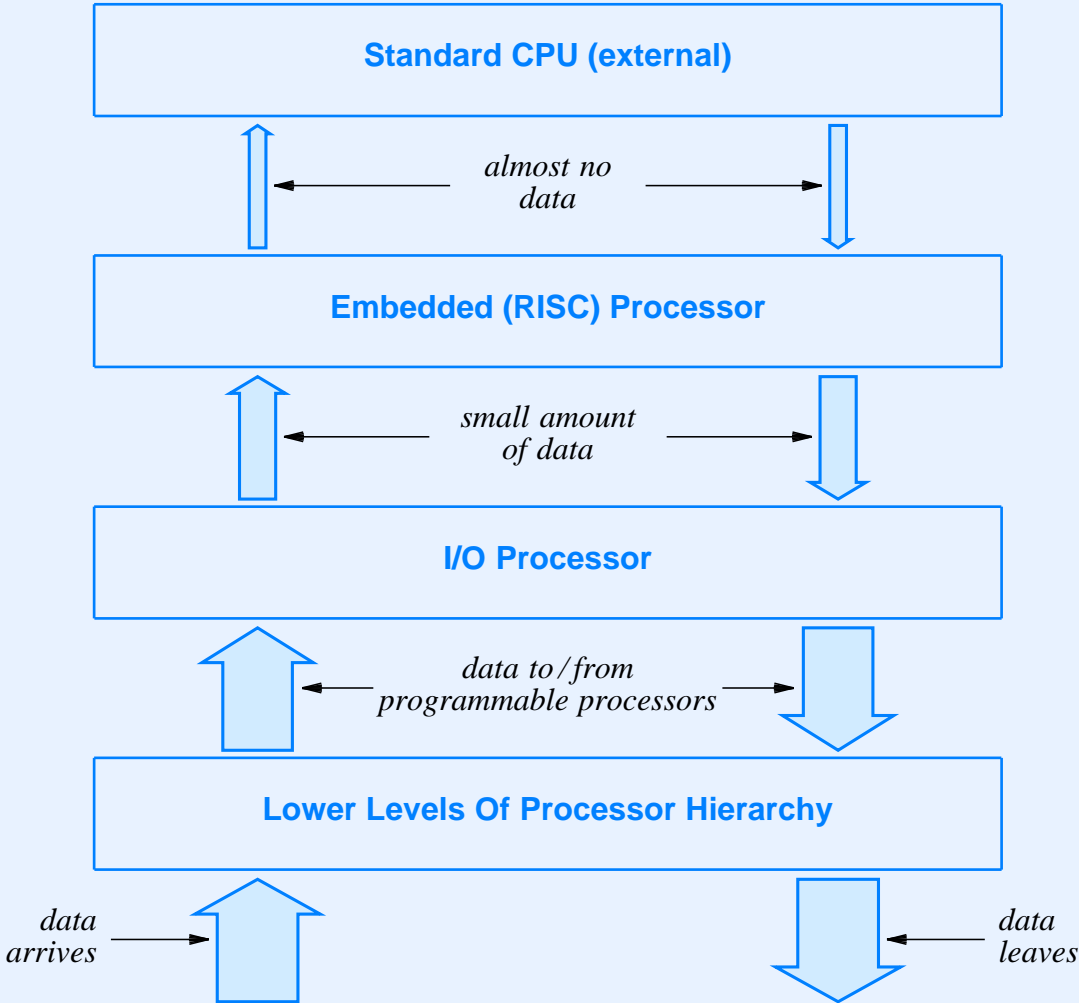
## I/O processor

- Basic packet processing
- Classification
- Forwarding
- Low-level ingress operations
- Low-level egress operations

# Using The Processor Hierarchy

*To maximize performance, packet processing tasks should be assigned to the lowest level processor capable of performing the task.*

# Packet Flow Through The Hierarchy



# Summary

- Network processor architectures characterized by
  - Processor hierarchy
  - Memory hierarchy
  - Internal buses
  - External interfaces
  - Special-purpose functional units
  - Support for concurrent or parallel execution
  - Programming model
  - Dispatch mechanisms



**Questions?**



# **XVII**

## **Overview Of The Intel Network Processor**

# An Example Network Processor

- We will
  - Choose one example
  - Examine the hardware
  - Gain first-hand experience with software
- The choice: Intel

# Intel Network Processor Terminology

- *Intel Exchange Architecture (IXA)*
  - Broad reference to architecture
  - Both hardware and software
  - Control plane and data plane
- *Intel Exchange Processor (IXP)*
  - Network processor that implements IXA

# Intel IXP2xxx

- Refers to second generation IXP chip
- Several models available

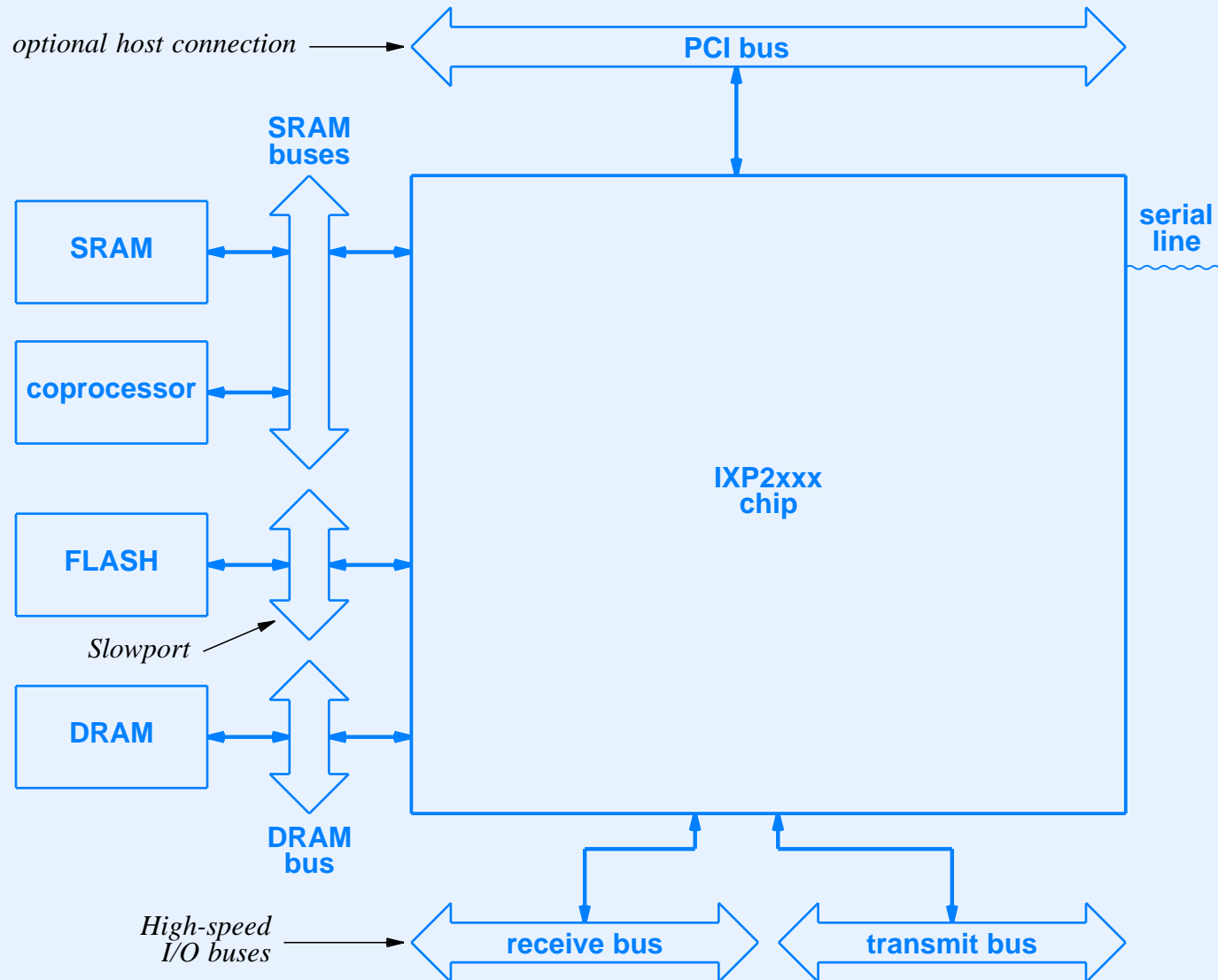
Model Number	Intended Use	Typical Input	Data Rate	Support For cryptography
IXP2400	Access & edge	OC-12 to OC-48	2.5 Gbps	no
IXP2800	Edge & core	OC-48 to OC-192	10.0 Gbps	no
IXP2850	Edge & core	OC-48 to OC-192	10.0 Gbps	yes

- Differences in speed, power consumption, parallelism, interfaces, packaging
- Term *IXP2xxx* refers to any model

# IXP2xxx Features

- One embedded RISC processor
- Eight to sixteen programmable packet processors
- Multiple, independent onboard buses
- Processor synchronization mechanisms
- Shared and non-shared onboard memory
- One low-speed serial line interface
- Multiple interfaces for external memories
- Multiple interfaces for external I/O buses
- Coprocessor for hash computation and cryptography
- Other functional units

# IXP2xxx External Connections



# IXP2400 External Connection Speeds

Type	Bus Width	Clock Rate	Data Rate
Serial line	(NA)	(NA)	38.4 Kbps
PCI bus	64 bits	66 MHz	2.2 Gbps
MSF interface	32 bits in and out	unspecified	unspecified
DDR DRAM	64 bits	150 MHz	2.4 GBps
QDR SRAM	32 bits	200 MHz	1.6 GBps

†GBps abbreviates *Giga Bytes per second*.

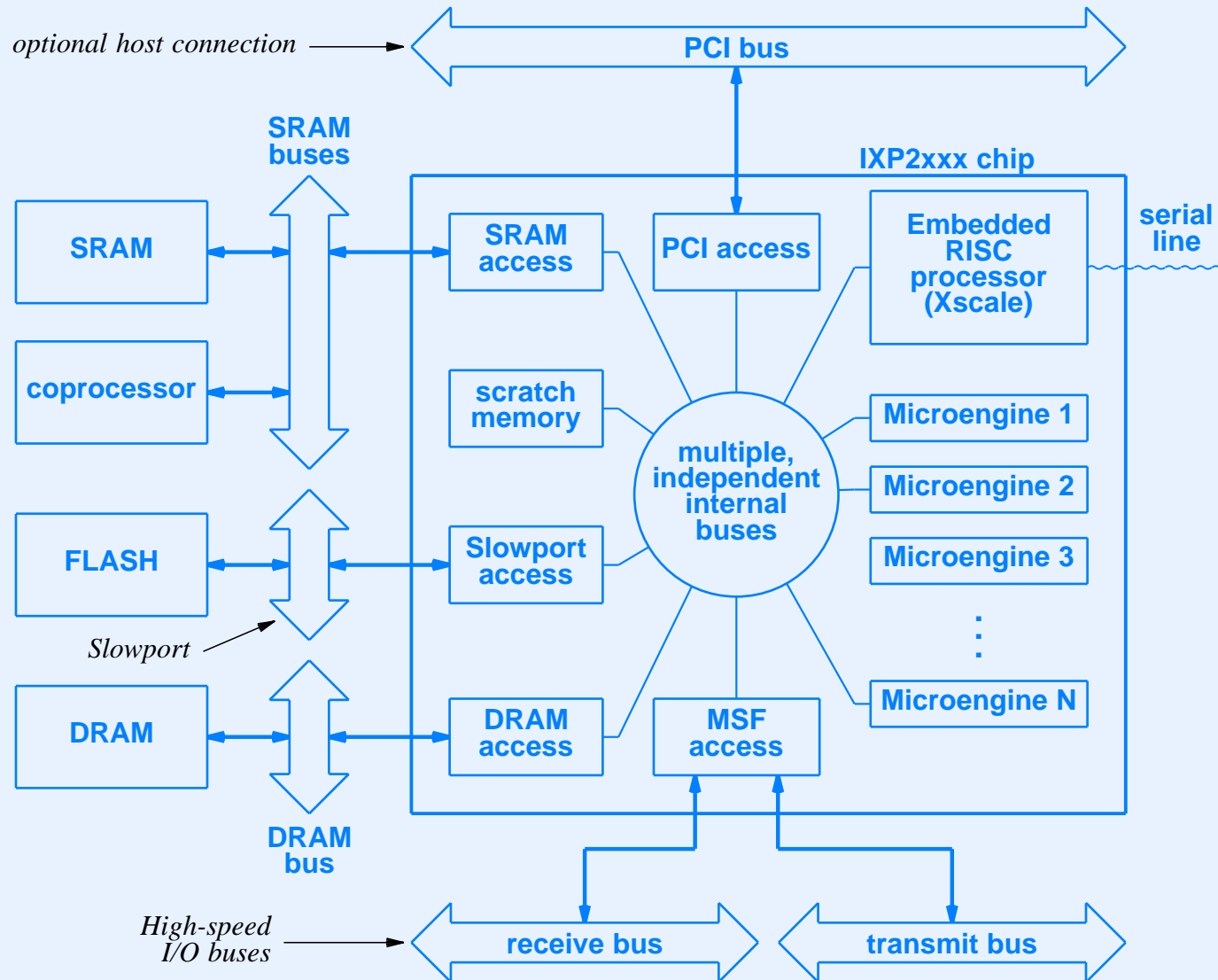
- Notes
  - MBps abbreviates *Mega Bytes per second*
  - IXP2800 operates at higher speed

# IXP2xxx Internal Units

Quantity	Component	Purpose
1	Embedded RISC processor	Control, higher layer protocols, and exceptions
8/16	Packet processing engines	I/O, basic packet processing, and packet forwarding
1+	SRAM access unit	Coordinate access to the external SRAM bus
1+	DRAM access unit	Coordinate access to the external DRAM bus
1	Media / Switch Fabric access unit	Coordinate access to the external I/O devices
1	PCI bus access unit	Coordinate access to the external PCI bus
1	Hash unit	Compute a hash function for high-speed lookup
0 or 1	Crypto unit	Compute cryptographic encoding for secure transfer
several	Onboard buses	Internal control and data transfer



# IXP2xxx Internal Architecture



# Processors On The IXP2xxx

<b>Processor Type</b>	<b>Onboard?</b>	<b>Programmable?</b>
General-Purpose Processor	no	yes
Embedded RISC Processor	yes	yes
I/O Processors	yes	yes
Coprocessors	yes	no
Physical Interfaces	no	no

# IXP2xxx Memory Hierarchy

Memory Type	Maximum Size	On Chip?	Typical Use
GP Registers	256 (2 banks)	yes	Intermediate computation
Inst. Cache	32 Kbytes	yes	Recently used instructions
Data Cache	32 Kbytes	yes	Recently used data
Mini Cache	2 Kbytes	yes	Data that is reused once
Write buffer	unspecified	yes	Write operation buffer
Local memory	2560 bytes / $\mu$ eng.	yes	Register spills and caching
Scratchpad	16 Kbytes	yes	IPC and synchronization
Inst. Store	4 Kbytes / $\mu$ eng.	yes	Microengine instructions
FlashROM	unspecified	no	Bootstrap
SRAM	64 Mbytes	no	Tables or packet headers
DRAM	2 Gbytes	no	Packet storage

# IXP2xxx Memory Characteristics

Memory Type	Access Unit (bytes)	Relative Access Time	Special Features
local	4	1	accessed using the LM_ADDR registers
Scratch	4	10	synchronization via atomic read-modify-write support for IPC (rings) push/pull reflector mode
SRAM	4	14	follows QDR specification atomic operations support for queues and rings bit manipulation
DRAM	8	20	connects to: Xscale, microengines, and PCI bus master

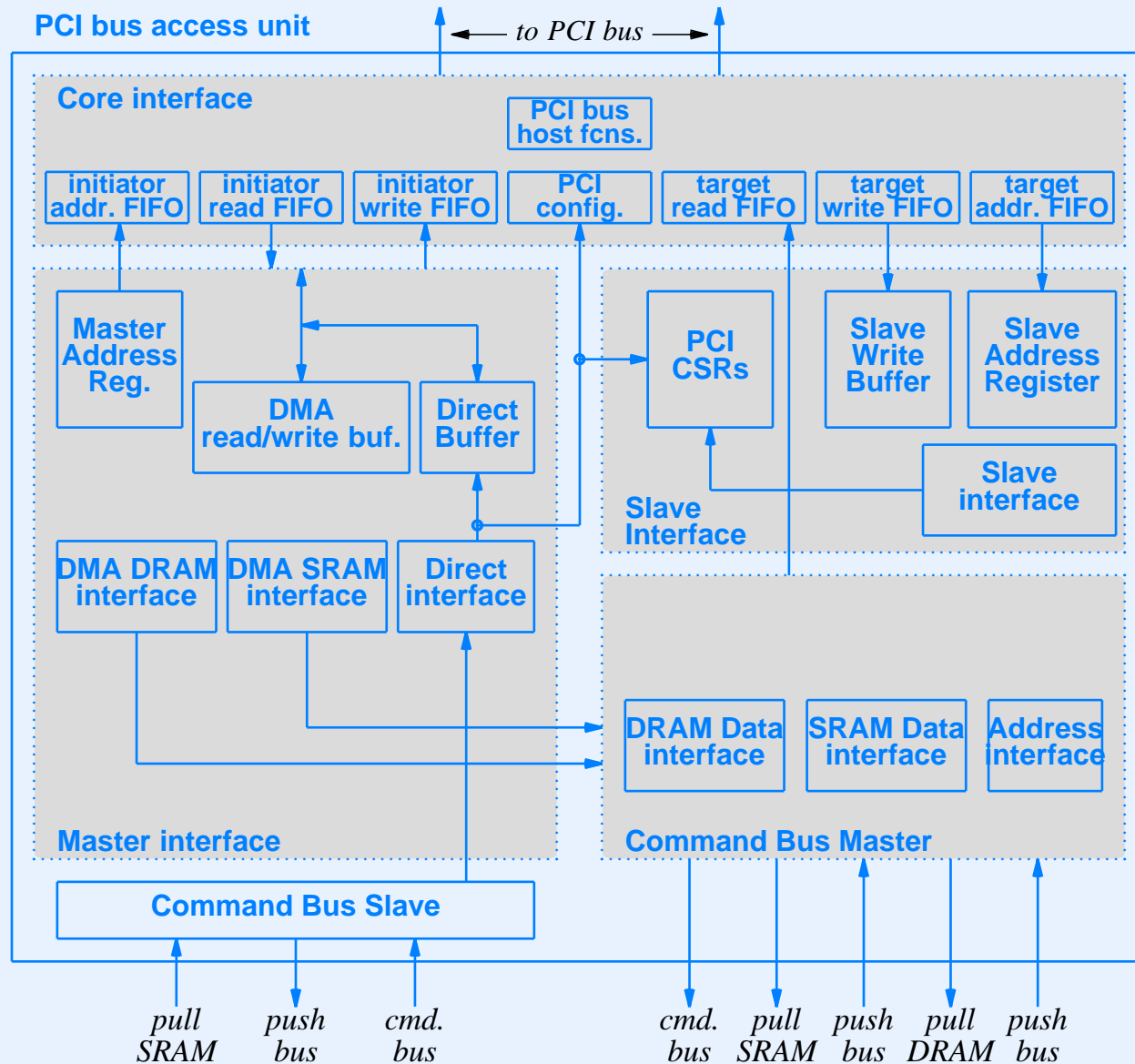
# Memory Access

- Each memory specifies *minimum access unit*
  - Two-byte unit is *word*
  - Four-byte unit is *longword*
  - Eight-byte unit is *quadword*
- When program accesses item in memory, physical memory system fetches entire access unit

# The Point About Memory Access

*The underlying memory is organized into data units of words or longwords. To achieve optimal performance, programmers must understand the memory organization and allocate items to minimize access times.*

# Example Of Complexity: PCI Access Unit



# Summary

- We will use Intel IXP2xxx as example
- IXP2xxx offers
  - Embedded processor plus parallel packet processors
  - Connections to external memories and buses





**Questions?**

**XX**

**Reference System  
And  
Software Development Kit  
(ENP-2611, SDK)**

# Reference System

- Provided by vendor
- Targeted at potential customers
- Usually includes
  - Hardware testbed
  - Development software
  - Simulator or emulator
  - Download and bootstrap software
  - Reference implementations

# Intel Reference Hardware

- Single-board network processor testbed
- Plugs into PCI bus on a PC
- Part number *ENP-2611*
- Internal code name *Mt. Hood*
- Manufactured by Radisys Corporation

# Items On The Intel ENP-2611 Reference System

<b>Quantity or Size</b>	<b>Item</b>
<b>1</b>	<b>IXP2400 network processor (600MHz)</b>
<b>8</b>	<b>Mbytes of QDR-SRAM memory</b>
<b>256</b>	<b>Mbytes of DDR-SDRAM memory</b>
<b>16</b>	<b>Mbytes of Flash ROM memory</b>
<b>1</b>	<b>SPI-3 bridge FPGA to connect to:</b> <ul style="list-style-type: none"><li><b>– PM3386/7 Gigabit Ethernet MACs</b></li><li><b>– MSF running in SPI-3 mode</b></li></ul>
<b>3</b>	<b>10/100/1000 optical Ethernet ports</b>
<b>1</b>	<b>10/100 Ethernet management port</b>
<b>1</b>	<b>Serial interface (on the XScale)</b>
<b>1</b>	<b>PCI bus interface</b>

# Intel Reference Software

- Known as *Software Development Kit (SDK)*
- Runs on PC
- Includes:

Software	Purpose
C compiler	Compile C programs for the XScale
MicroC compiler	Compile C programs for the microengines
Assembler	Assemble programs for the microengines
Simulator	Simulate an IXP2xxx for debugging (Windows)
Resource Manager	XScale kernel module used to control and communicate with hardware
Workbench Server	Load software into the network processor
Workbench Backend Svr	Remote (Windows) application that controls and communicates with the network processor
Bootstrap	Start the network processor running
Reference Code	Example programs for the IXP2xxx that show how to implement basic functions

# Operating System On XScale

- XScale processor powerful enough to run an OS
- Version of *Embedded Linux* used that supports
  - Telnet server that allows remote login
  - Shell that allows a user to run commands
  - Access to a remote file system via NFS
  - Other servers that are used for control and status

# Operating System On External Host

- Cross-development tools run on external host (PC)
  - Some SDK compilers require Linux
  - Workbench Backend Server (*WB Backend Server*) used for download runs under Windows
- Site can avoid using Workbench software



# External File Access And Storage

- DRAM accessed via DRAM bus
- SRAM and Flash accessed via SRAM bus
- Ethernet ports accessed via MSF
- Code and data downloaded via control Ethernet
- NFS accessed via control Ethernet
- XScale accessed via
  - Serial line (console)
  - Telnet

# Basic Paradigm

- Build software on conventional computer
- Load into reference system
- Test / measure results

# Bootstrapping Procedure

1. Restarting the ENP-2611 causes the boot manager on the XScale to load and run a copy of *RedBoot* program out of the Flash ROM.
2. The RedBoot program running on the XScale sends a BOOTP request to obtain an IP address for the management Ethernet port.
3. The RedBoot program running on the XScale uses the IP address obtained via BOOTP to contact a TFTP server and download a copy of the Linux kernel image.
4. The Linux kernel boots and uses NFS to mount a remote file system.
5. After the kernel is operational, a script runs that starts a telnet server on the management interface as well as other servers that accessible to external hosts.

# Starting Software

1. Compile code for the XScale and microengines, and place the resulting files in a directory, D, on the computer that runs the NFS server. The Intel SDK uses the terms *core component* and *microblock* for the compiled files.
2. Copy the entire contents of directory D to the read-write public download directory, W, that has been mounted by the testbed. (This step is not necessary if only one programmer has access to the testbed.)
3. Run a telnet client program on the host that forms a connection to the testbed system, and log onto the XScale.
4. Load a set of Linux kernel modules, including microengine drivers, the Resource Manager library, and a module to configure the SPI-3 interface.

# Starting Software

## (continued)

5. Load a module that reads microblock code and places the code into microengines. Note that no such module comes with the SDK; a programmer must write the module. However, the module does not need to access hardware directly because the *Resource Manager* can be used to load code into microengines.
6. Start the programs on the XScale that were compiled in Step 1.

# Summary

- Reference systems
  - Provided by vendor
  - Targeted at potential customers
  - Usually include
    - \* Hardware testbed
    - \* Cross-development software
    - \* Download and bootstrap software
    - \* Reference implementations



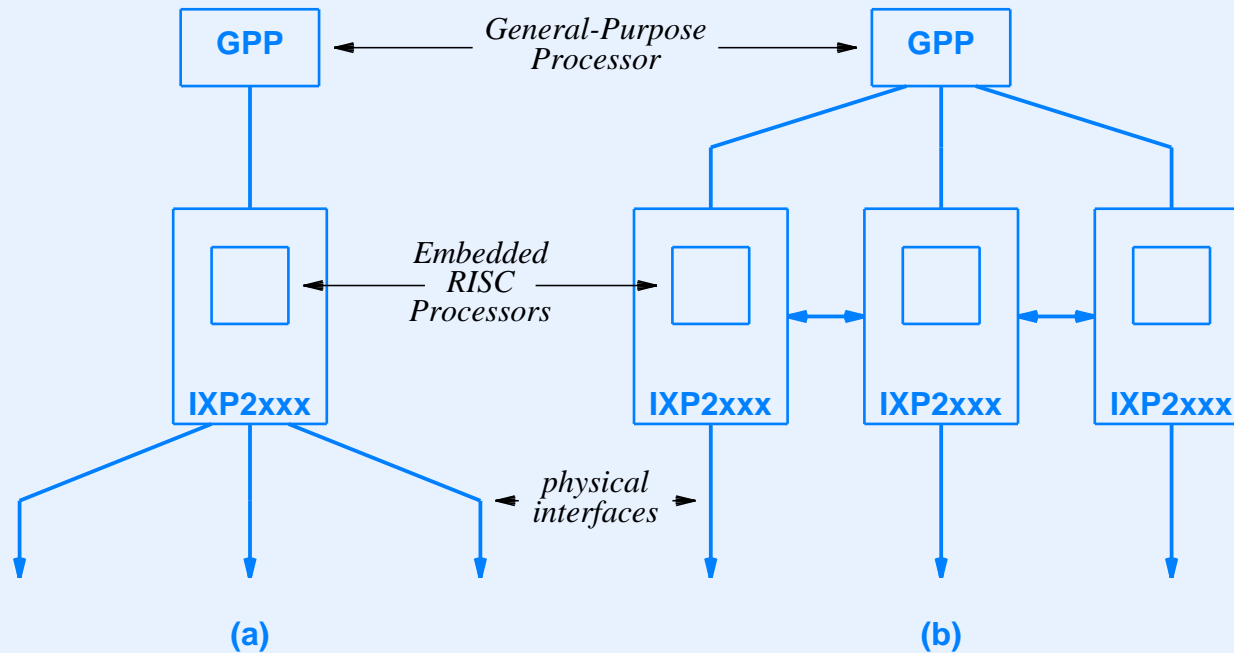
**Questions?**

# **XVIII**

## **Embedded RISC Processor (XScale Core)**



# XScale Role



- (a) Single IXP2xxx
- (b) Multiple IXP2xxxs
- Role of XScale differs

# Tasks That Can Be Performed By XScale

- Bootstrapping
- Exception handling
- Higher-layer protocol processing
- Interactive debugging
- Diagnostics and logging
- Memory allocation
- Application programs (if needed)
- User interface and/or interface to the GPP
- Control of packet processors
- Other administrative functions

# XScale Characteristics

- Reduced Instruction Set Computer (RISC)
- Thirty-two bit arithmetic
- Extra functionality can be provided via a coprocessor
- Byte addressable memory
- Virtual memory support
- Built-in serial port
- Facilities for a kernelized operating system
- Performance monitoring unit

# Arithmetic

- XScale is configurable in two modes
  - Big endian
  - Little endian
- Choice made at run-time

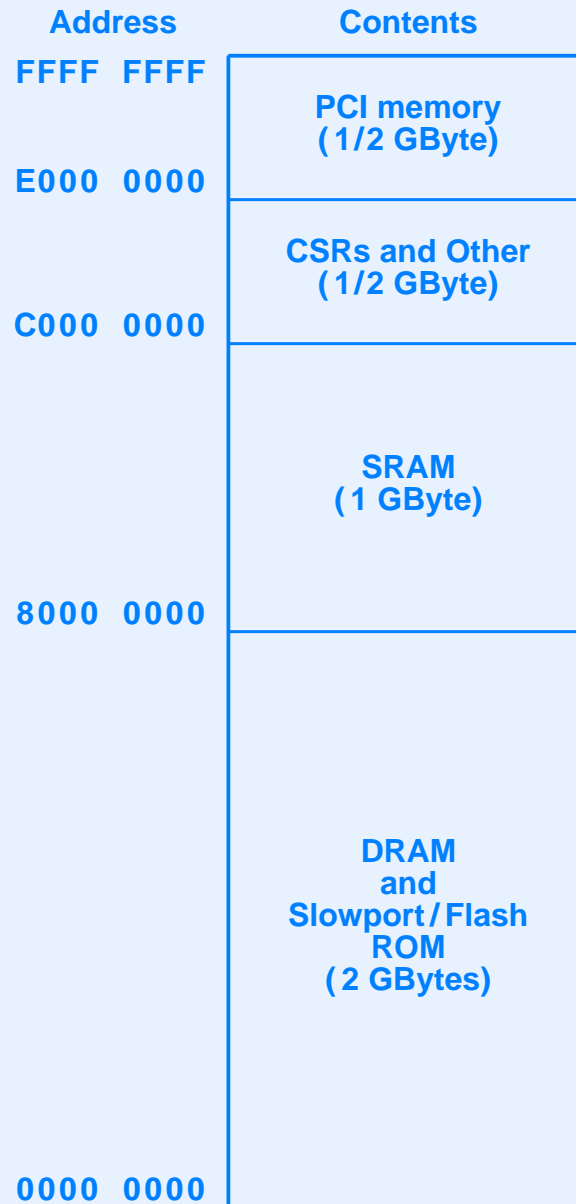
# Performance Monitoring Unit

- Can measure
  - Instruction cache miss rate
  - Translation Lookaside Buffer (TLB) miss rate
  - Stalls in the instruction pipeline
  - Number of branches initiated by software
- Useful for program tuning

# XScale Memory Organization

- Single, uniform address space
- Includes memories and devices
- Byte addressable

# XScale Address Space



# Shared Memory And Address Translation

- Memory shared with microengines
- Microengines use separate physical address spaces



# Consequence For Programmers

*Because the Xscale and packet processors do not use the same memory architecture, linked lists and other data structures in which pointers cross from one memory to another do not make sense in in the microengine address space.*

# Internal Peripheral Units

- One UART
- Four 32-bit countdown timers (one watchdog)
- Eight General-Purpose I/O (GPIO) pins
- One Slowport interface

# Summary

- Embedded processor on IXP2xxx is XScale
- XScale addressing
  - Single, uniform address space
  - Includes all memories
  - Byte addressable



**Questions?**

# **XIX**

## **Packet Processor Hardware (Microengines)**

# Microengines

- Parallel hardware units
  - Eight on IXP2400
  - Sixteen on IXP28x0
- Handle fast data path processing
- Known as *microengine version 2 (MEv2)*

# Role Of Microengines

- Packet ingress from physical layer hardware
- Checksum verification
- Header processing and classification
- Packet buffering in memory
- Table lookup and forwarding
- Header modification
- Checksum computation
- Packet egress to physical layer hardware

# Microengine Characteristics

- Programmable microcontroller
- RISC design
- Two hundred fifty-six general-purpose registers
- Five hundred twelve transfer registers
- One hundred twenty-eight Next Neighbor registers
- Hardware support for four threads and context switching
- Six hundred forty words of local memory
- Sixteen entry CAM with thirty-two bits per entry



# Microengine Characteristics

## (continued)

- Control of an Arithmetic Logic Unit (ALU)
- Direct access to various functional units
- A unit to compute a Cyclic Redundancy Check (CRC)

# Microengine Level

- Not a typical CPU
- Does not contain native instruction for each operation
- Controls other units on the chip
- Really a *microsequencer*

# Consequence Of Microsequencing

*Because it functions as a microsequencer, a microengine does not provide native hardware instructions for arithmetic operations, nor does it provide addressing modes for direct memory access. Instead, a program running on a microengine controls and uses functional units on the chip to access memory and perform operations.*

# Microengine Instruction Set (Part 1)

Description	Instruction
<b>General instructions (Arithmetic, Rotate, And Shift)</b>	
ALU	Perform an ALU operation
ALU_SHF	Perform an ALU operation and shift
ASR	Perform an arithmetic right shift
BYTE_ALIGN_BE, BYTE_ALIGN_LE	Concatenate registers and select bytes
CRC_LE, CRC_BE	Compute CRC (big or little endian)
DBL_SHF	Concatenate and shift two longwords
MUL_STEP	Multiply two unsigned integers
FFS	Find position of LSB in register
POP_COUNT	Count 1 bits in a register
IMMED	Load immediate 16-bit value to register
IMMED_B0 through IMMED_B3	Load immediate byte to a field
IMMED_W0, IMMED_W1	Load immediate 16-bit word to a field
LD_FIELD, LD_FIELD_W_CLR	Load bytes to specified fields
LOAD_ADDR	Load instruction address
LOCAL_CSR_RD, LOCAL_CSR_WR	Read or write local microengine CSRs
NOP	No operation

# Microengine Instruction Set (Part 2)

Description	Instruction
<b>Branch and Jump Instructions</b>	
<b>BCC</b>	Branch on condition code
<b>BR</b>	Branch unconditionally
<b>BR_BCLR, BR_BSET</b>	Branch if bit clear or set
<b>BR=BYTE, BR!=BYTE</b>	Branch if byte equal or not equal to literal
<b>BR=CTX, BR!=CTX</b>	Branch on current context
<b>BR_INP_STATE, BR_!INP_STATE</b>	Branch on event state
<b>BR_SIGNAL, BR_!SIGNAL</b>	Branch if signal deasserted
<b>JUMP</b>	Jump to label
<b>RTN</b>	Return from branch or jump

# Microengine Instruction Set (Part 3)

Description	Instruction
<b>Content Addressable Memory (CAM) Instructions</b>	
<b>CAM_CLEAR</b>	<b>Clear all entries in local CAM</b>
<b>CAM_WRITE_STATE</b>	<b>Write state bits into specified CAM entry</b>
<b>CAM_READ_TAG</b>	<b>Read tag for specified CAM entry</b>
<b>CAM_READ_STATE</b>	<b>Read state bits for specified CAM entry</b>
<b>CAM_LOOKUP</b>	<b>Search local CAM for tag value</b>
<b>CAM_WRITE</b>	<b>Write tag value for specified CAM entry</b>

# Microengine Instruction Set (Part 4)

Instruction	Description
<b>I/O And Context Swap Instructions</b>	
DRAM (read and write) DRAM (RBUF and TBUF) CAP (CSR addressing) CAP (calculated addressing) CAP (reflect) CTX_ARB HALT HASH MSF PCI SCRATCH (read and write) SCRATCH (atomic operation) SCRATCH (ring operation) SRAM (read and write) SRAM (atomic operation) SRAM (CSR) SRAM (read queue descriptor) SRAM (write queue descriptor) SRAM (enqueue) SRAM (dequeue) SRAM (ring operation) SRAM (journal operation)	Perform an ALU operation Perform an ALU operation and shift Perform an arithmetic right shift Concatenate registers and select bytes Compute CRC (big or little endian) Concatenate and shift two longwords Multiply two unsigned integers Find position of LSB in register Count 1 bits in a register Load immediate 16-bit value to register Load immediate byte to a field Load immediate 16-bit word to a field Load bytes to specified fields Load instruction address Read or write local microengine CSRs Load or store values in CSR registers Access queue in SRAM Change queue in SRAM Enqueue item in SRAM queue Dequeue item from SRAM queue Manipulate a communication ring in SRAM Perform atomic operation in SRAM

# Microengine View Of Memory

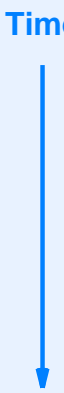
- Separate address spaces
- Specific instruction to reference each memory type
  - Instruction *dram* to access DRAM memory
  - Instruction *sram* to access SRAM memory
  - Instruction *scratch* to access Scratchpad memory
- Consequence: early binding of data to memory



# Six-Stage Instruction Pipeline

<b>Stage</b>	<b>Description</b>
1	Fetch the next instruction (part 1)
2	Fetch the next instruction (part 2)
3	Decode the instruction and get register address(es)
4	Extract the operands from registers
5	Perform ALU, shift, or compare operations and set the condition codes
6	Write the results to the destination register

# Example Of Pipeline Execution



clock	stage 1	stage 2	stage 3	stage 4	stage 5	stage 6
1	inst. 1	-	-	-	-	-
2	inst. 2	inst. 1	-	-	-	-
3	inst. 3	inst. 2	inst. 1	-	-	-
4	inst. 4	inst. 3	inst. 2	inst. 1	-	-
5	inst. 5	inst. 4	inst. 3	inst. 2	inst. 1	-
6	inst. 6	inst. 5	inst. 4	inst. 3	inst. 2	inst. 1
7	inst. 7	inst. 6	inst. 5	inst. 4	inst. 3	inst. 2
8	inst. 8	inst. 7	inst. 6	inst. 5	inst. 4	inst. 3

- Once pipeline is started, one instruction completes per cycle

# Instruction Stall

- Occurs when operand not available
- Processor temporarily stops execution
- Reduces overall speed
- Should be avoided when possible

# Example Instruction Stall

- Consider two instructions:
  - K: ALU operation to add the contents of R1 to R2
  - K+1: ALU operation to add the contents of R2 to R3
- Second instruction cannot access R2 until value has been written
- Stall occurs

# Effect Of Instruction Stall

clock	stage 1	stage 2	stage 3	stage 4	stage 5	stage 6
1	inst. K	inst. K-1	inst. K-2	inst. K-3	inst. K-4	inst. K-5
2	inst. K+1	inst. K	inst. K-1	inst. K-2	inst. K-3	inst. K-4
3	inst. K+2	inst. K+1	inst. K	inst. K-1	inst. K-2	inst. K-3
4	inst. K+3	inst. K+2	inst. K+1	inst. K	inst. K-1	inst. K-2
5	inst. K+3	inst. K+2	inst. K+1	-	inst. K	inst. K-1
6	inst. K+3	inst. K+2	inst. K+1	-	-	inst. K
7	inst. K+4	inst. K+3	inst. K+2	inst. K+1	-	-
8	inst. K+5	inst. K+4	inst. K+3	inst. K+2	inst. K+1	-

- Bubble develops in pipeline
- Bubble eventually reaches final stage

# A Note For Programmers

*Understanding the execution pipeline is important for programmers because dependencies among instructions can cause the processor to stall, which lowers performance.*

# Sources Of Delay

- Access to result of previous / earlier operation
- Conditional branch
- Memory access

# Memory Access Delays

Type Of Memory	Approx. Access Time In Clock Cycles	
	IXP2400	IXP28x0
Local Memory	1	1
Scratchpad	60	60
SRAM	150	90
DRAM	300	120

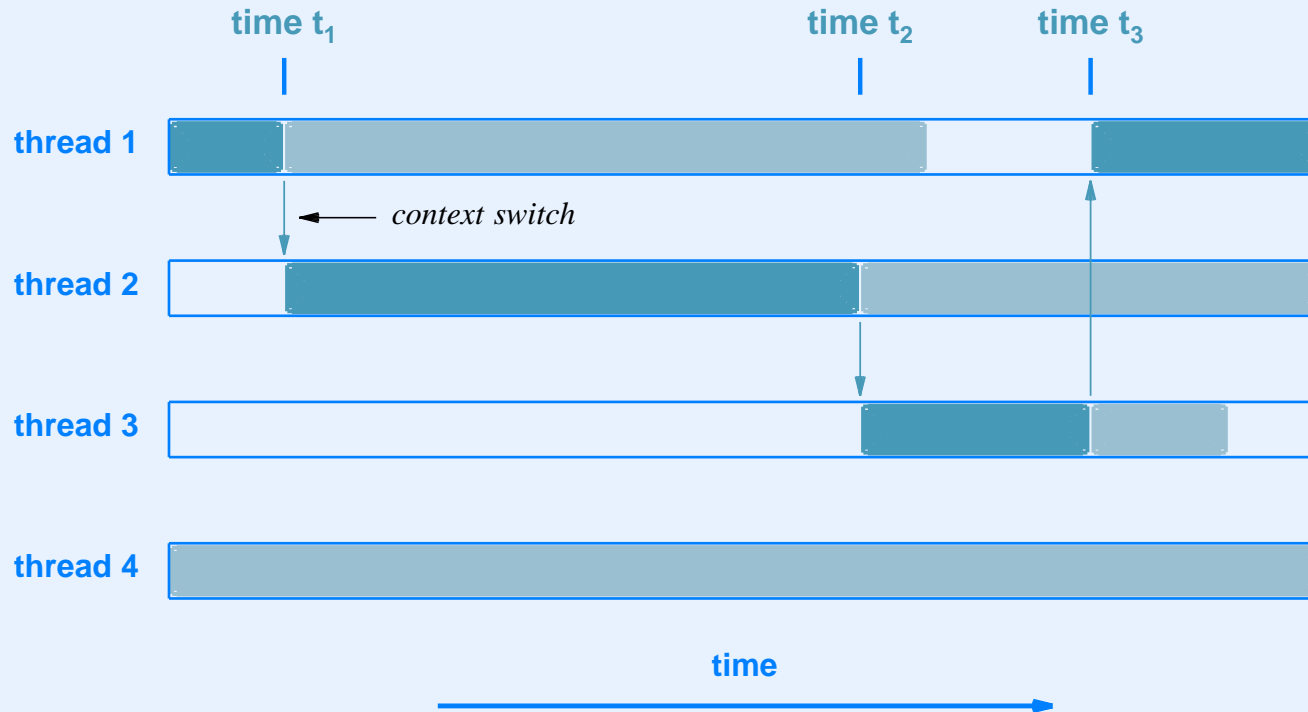
- Delay is surprisingly large



# Threads Of Execution

- Technique used to speed processing
- Multiple *threads of execution* remain ready to run
- Program defines threads and informs processor
- Processor runs one thread at a time
- Processor automatically switches context to another thread when current thread blocks
- Known as *hardware threads*
- Microengine has eight threads

# Illustration Of Hardware Threads



- White - ready but idle
- Blue - being executed by microengine
- Gray - blocked (e.g., during memory access)

# The Point Of Hardware Threads

*Hardware threads increase overall throughput by allowing a microengine to handle up to four packets concurrently; with threads, computation can proceed without waiting for memory access.*

# Context Switching Time

- *Low-overhead context switch* means one instruction delay as hardware switches from one thread to another
- *Zero-overhead context switch* means no delay during context switch
- IXP2xxx offers zero-overhead context switch

# Microengine Instruction Store

- Private instruction store per microengine
- Advantage: no contention
- Disadvantage: smaller size (4000 instructions)

# General-Purpose Registers

- two hundred fifty-six per microengine
- Thirty-two bits each
- Used for computation or intermediate values
- Divided into banks
- Context-relative or absolute addresses

# Forms Of Addressing

- Absolute
  - Entire set available
  - Uses integer from 0 to 255
- Context-relative
  - One eighth of set available to each thread
  - Uses integer from 0 to 31
  - Allows same code to run on multiple microengines

# Register Banks

- Mechanism commonly used with RISC processor
- Registers divided into *A bank* and *B bank*
- Maximum performance achieved when each instruction references a register from the A bank and a register from the B bank



# Summary Of General-Purpose Registers

Number Of Active Contexts	Active Context Number	General Purpose Register Absolute Addresses		S Transfer or Neighbor Index	D Transfer Index
		A Port	B Port		
8	0	0 - 15	0 - 15	0 - 15	0 - 15
	1	16 - 31	16 - 31	16 - 31	16 - 31
	2	32 - 47	32 - 47	32 - 47	32 - 47
	3	48 - 63	48 - 63	48 - 63	48 - 63
	4	64 - 79	64 - 79	64 - 79	64 - 79
	5	80 - 95	80 - 95	80 - 95	80 - 95
	6	96 - 111	96 - 111	96 - 111	96 - 111
	7	112 - 127	112 - 127	112 - 127	112 - 127
4	0	0 - 31	0 - 31	0 - 31	0 - 31
	2	32 - 63	32 - 63	32 - 63	32 - 63
	4	64 - 95	64 - 95	64 - 95	64 - 95
	6	96 - 127	96 - 127	96 - 127	96 - 127

- Note: half of the registers for each context are from A bank and half from B bank

# Transfer Registers

- Used to buffer external memory transfers
- Example: read a value from memory
  - Copy value from memory into transfer register
  - Move value from transfer register into general-purpose register
- Five hundred twelve per microengine
- Divided into four types
  - SRAM or DRAM
  - Read or write

# Next Neighbor Registers

- Provide high-speed, synchronized communication
- Allows data to pass between microengines
- Handle small values
- Typically used to pass buffer address, not entire packet
- Used to build software pipeline

# Next Neighbor Register Hardware

Primitive	Type	Purpose
NN_Get	Register	Extract next item from next neighbor ring
NN_Put	Register	Insert item in a next neighbor ring
NN_FULL	Signal	Test whether a next neighbor ring is full
NN_EMPTY	Signal	Test whether a next neighbor ring is empty

- Hardware polls state bit

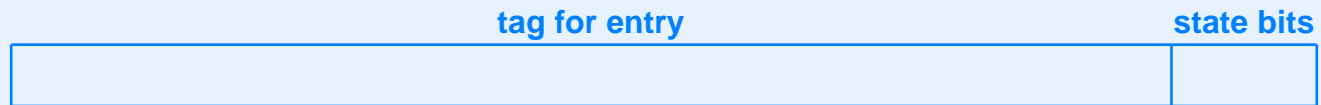
# Local Memory

- Private (one per microengine)
- Small size (2560 bytes)
- Low latency (one instruction cycle after setup)
- Read or written under program control
- Accessed via special hardware registers
  - Address placed in *LM\_ADDR0*
  - Value accessed via *LM\_ADDR1*

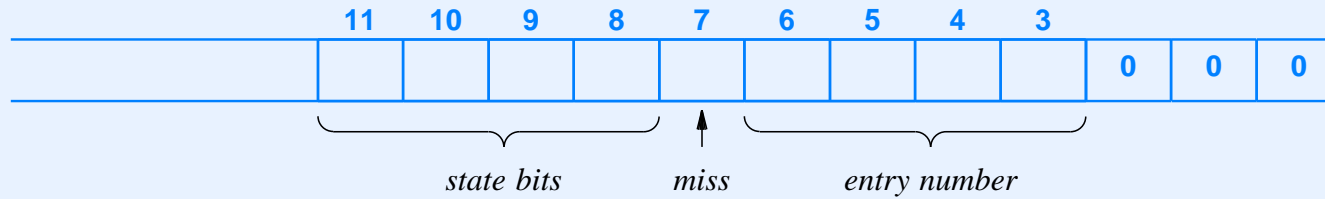
# Content Addressable Memory (CAM)

- Used to speed searches
- Characteristics
  - Sixteen entries
  - Thirty-two bit search key per entry
  - Four-bit status value per entry
  - Single instruction lookup
  - Hardware reports first entry that matches

# Organization Of CAM



# Hardware Bits Returned For CAM Operation





# Local Control And Status Registers

- Used to interrogate or control the IXP2xxx
- All mapped into XScale address space
- Microengine can only access its own local CSRs

# Example Local CSRs

Local CSR	Purpose
USTORE_ADDRESS	Load the microengine control store
USTORE_DATA_LOWER	Lower 20 bits of the instruction
USTORE_DATA_UPPER	Upper 12 bits of the instruction
USTORE_ERROR_STATUS	Error status bits
ALU_OUT	Debugging: allows XScale to read GPRs and transfer registers
CTX_ARB_CTL	Context arbiter control
CTX_ENABLES	Context arbiter control
CC_ENABLE	Debugging: read condition codes
CSR_CTX_POINTER	Used to modify context-specific CSRs
INDIRECT_CTX_STS	To access context-specific PC
ACTIVE_CTX_STS	Find context currently running
TIMESTAMP_HIGH	Clock (high-order bits)
TIMESTAMP_LOW	Clock (low-order bits)
PSEUDO_RANDOM_NUMBER	Random value

- Note:  $n$  is digit from 0 through 7 (hardware contains a separate CSR for each of the eight contexts).

# Interprocessor Communication Mechanisms

- Context-to-XScale communication
- Context-to-Context communication (one or more IXP2xxx's)

# Context-To-XScale Communication

- Interrupts (SHaC unit)
- Shared memory
- Memory ring mechanism
  - SRAM
  - Scratchpad

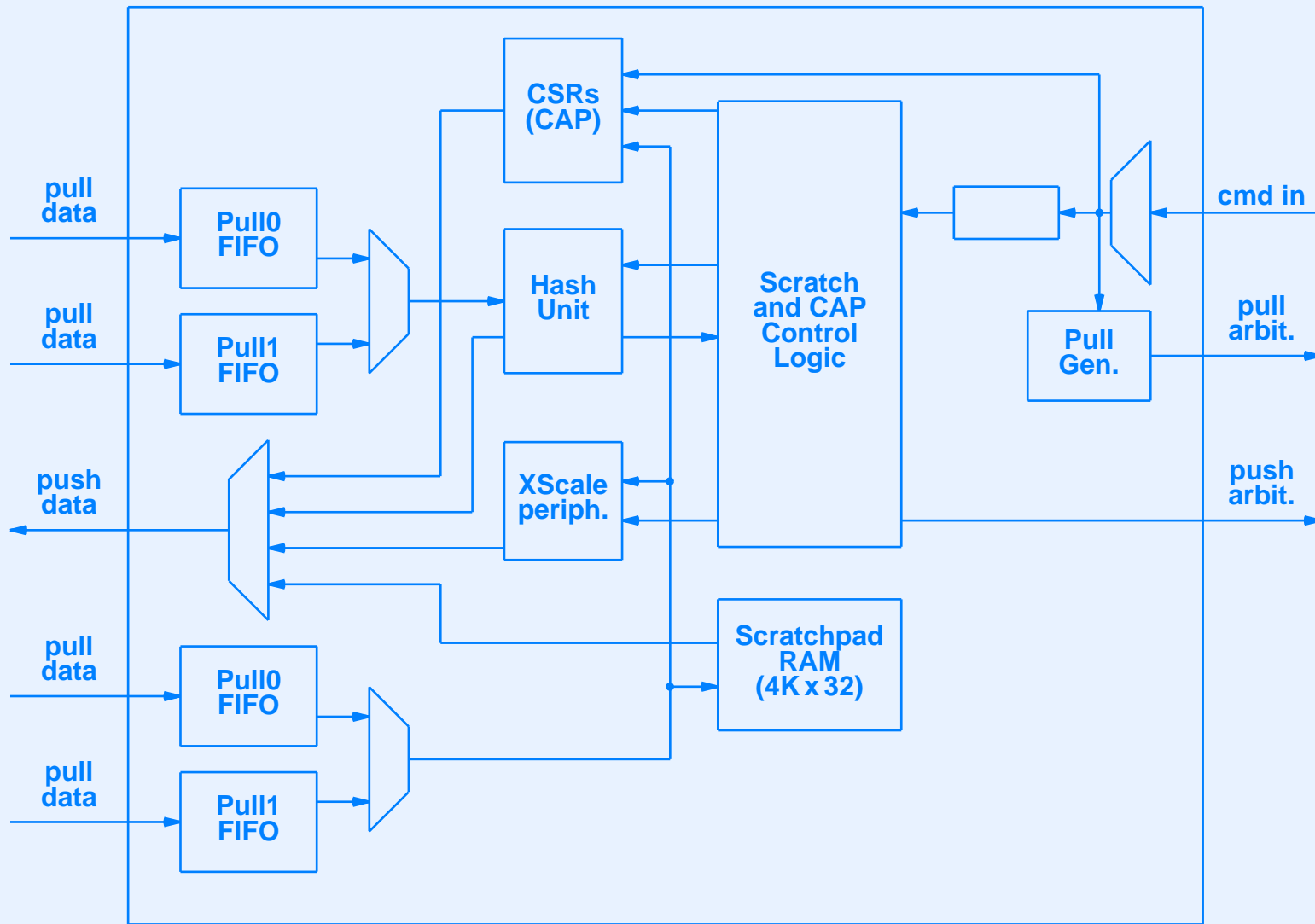
# Context-to-Context communication

- Signal event mechanism
- Memory ring mechanism
- Next neighbor registers
- Reflector bus mechanism

# SHaC Unit

- Operates as coprocessor
- Controls
  - Scratchpad memory
  - Hash unit
  - Communication mechanism used by microengines
  - CSR bus interface
  - Push / pull reflector

# SHaC Architecture (simplified)



# ScratchPad Memory

- Organized into 4K words of 4 bytes each
- Offers special facilities
  - Atomic operations
    - \* Set or clear bits
    - \* Increment, decrement, add, or subtract
    - \* Swap values
  - Communication rings



# Hash Unit

- Configurable coprocessor
- Operates asynchronously
- Intended for fast table lookup

# Hash Unit Computation

- Computes quotient  $Q(x)$  and remainder  $R(x)$ :

$$A(x) * M(x) / G(x) \rightarrow Q(x) + R(x)$$

- $A(x)$  is input value
- $M(x)$  is hash multiplier (configurable)
- $G(x)$  is built-in value
- Three values for  $G$  (48-bit, 64-bit, or 128-bit hash)

# Hash Mathematics

- Integer value interpreted as polynomial over field  $[0,1]$
- Example:

$$20401_{16}$$

- Is interpreted as

$$x^{17} + x^{10} + 1$$

- Similarly, value  $G(x)$  used in 48-bit hash

$$1001002000401_{16}$$

- Is interpreted as

$$x^{48} + x^{36} + x^{25} + x^{10} + 1$$

# Hash Example

$$A = 8000000000001_{16} \quad (x^{47} + 1)$$

$$G = 1001002000401_{16} \quad (x^{48} + x^{36} + x^{25} + x^{10} + 1)$$

$$M = 20D_{16} \quad (x^9 + x^3 + x^2 + 1)$$

- Hash computes R, remainder of M times A divided by G

$$H(X) = R = A * M \% G$$

# Hash Example (continued)

- We see that

$$A(x) * M(x) = x^{56} + x^{50} + x^{49} + x^{47} + x^9 + x^3 + x^2 + 1$$

- Furthermore:

$$A * M = Q * G + R$$

# Hash Example (continued)

- Where

$$Q(x) = x^8 + x^2 + x^1$$

- Thus, Q is  $106_{16}$  and R is

$$R(x) = x^{47} + x^{44} + x^{38} + x^{37} + x^{33} + x^{27} + x^{26} \\ + x^{18} + x^{12} + x^{11} + x^9 + x^8 + x^3 + x^1 + 1$$

- The hash unit returns R as the value of the computation:

$$H(A) = R = 90620C041B0B_{16}$$

## Other IXP2xxx Hardware

- The IXP2xxx contains registers used for
  - Configuration and bootstrapping
  - Control of functional units and buses
  - Checking status of processors, threads, and onboard functional units

# The Point About Registers

*In addition to basic functional units, the IXP contains hundreds of registers that allow software to configure, control, or interrogate the status of functional units, buses, and attached devices.*



# Media Switch Fabric Interface

- Complex unit
- Primary interface to high-speed external devices
- Configurable to handle standard MACs such as
  - UTOPIA0 (IXP2400 only)
  - SPI-3 (IXP2400 only)
  - SPI-4.2 (IXP28x0 only)

# Transmit And Receive BUFs

- Used for I/O
- Contained in MSF unit
- Function as randomly accessible memory
- Transfer in chunks of 64, 128, or 256 bytes
- Two types
  - *Receive BUFs (RBUFs)* handle input
  - *Transmit BUFs (TBUFs)* handle output

# Crypto Unit

- Available on the IXP2850
- Two units
- Can be used for
  - Two 3DES/DES (*Data Encryption Standard*) cores for data encryption/decryption
  - One AES (*Advanced Encryption Standard*) core for data encryption/decryption that can use 128, 192 or 256 bit keys
  - Two SHA-1 (*Secure Hash Algorithm*) cores for authentication
- Programmer chooses

# Crypto Unit

## (continued)

- Support for
  - The *Electronic Code Book* standard (*ECB*)
  - The *Cipher Block Chaining* standard (*CBC*)
- Sufficient for
  - IPsec
  - SSL

# Crypto Unit API

- Input RAM (read/write to on-board RAM)
- State (set crypto parameters, e.g. keys)
- Cipher (initiate cypher algorithm execution)
- Hash (initiate hash algorithm execution)
- Utils (functions such as checksum calculation)

# Summary

- Microengines
  - Low-level, programmable packet processors
  - Use RISC design with instruction pipeline
  - Have hardware threads for higher throughput
  - Use transfer registers to access memory
  - Use BUFs for I/O
  - Have access to hash and crypto units



**Questions?**

**XXI**

# **Programming Model**



# Assumptions About Support Software And Overall Structure

- XScale runs MontaVista Embedded Linux®
- Code for XScale compiled to run under Linux
- Microengines do not run any OS
- Code for microengines compiled to run on bare machine
- Consequences for programmers:
  - Microcode handles all hardware details
  - XScale code relies on libraries and OS

# Major Pieces Of Software

- One or more *microblocks* that run on the microengines
- A *core component* that runs on the XScale
- User interface code that runs on the XScale
- Note: we will concentrate on the first two

# Interconnections Among Microblocks

- Each microblock is asynchronous
- Fast data path code built as series of microblock stages
- Basic pipeline architecture

# Typical Network Systems

- At least three microblocks
  - Ingress
  - Processing
  - Egress

# Example Microblock Pipeline



- Ingress and egress microblocks
  - Interface with MSF
  - Handle packet I/O
- Process microblock
  - Performs protocol processing

# Assignment Of Microblocks To Microengines

- Approach #1
  - Multiple types of microblocks run on each microengine
  - Each thread runs one microblock
- Approach #2
  - Each microblock runs on separate microengine
  - Multiple copies (threads) used to increase performance
- In practice, most systems use approach #2

# Mpackets And Transfers

- MSF divides incoming packet into fixed-size units called *mpackets*
- Mpacket size can be configured to be 64, 128, or 256 octets
- Each mpacket received independently
- Hardware sets bit to indicate first and last mpacket of a packet
- Software divides outgoing packet into mpackets
- Each mpacket transmitted independently

# Ingress And Egress Microblocks

- Available from *building blocks library*
- Ingress microblock
  - Named *Receive (RX)*
  - Invoked whenever an mpacket arrives
  - Places mpacket in buffer in memory
- Egress microblock
  - Named *Transmit (TX)*
  - Invoked whenever an mpacket is ready for egress
  - Releases buffer after mpacket is sent



# Microblocks And Parallel Execution

- Microengine can be dedicated to run exactly one microblock
- Microengine can run multiple microblocks in a pipeline
- Multiple microengines can run copies of a pipeline
- Each approach has advantages and disadvantages

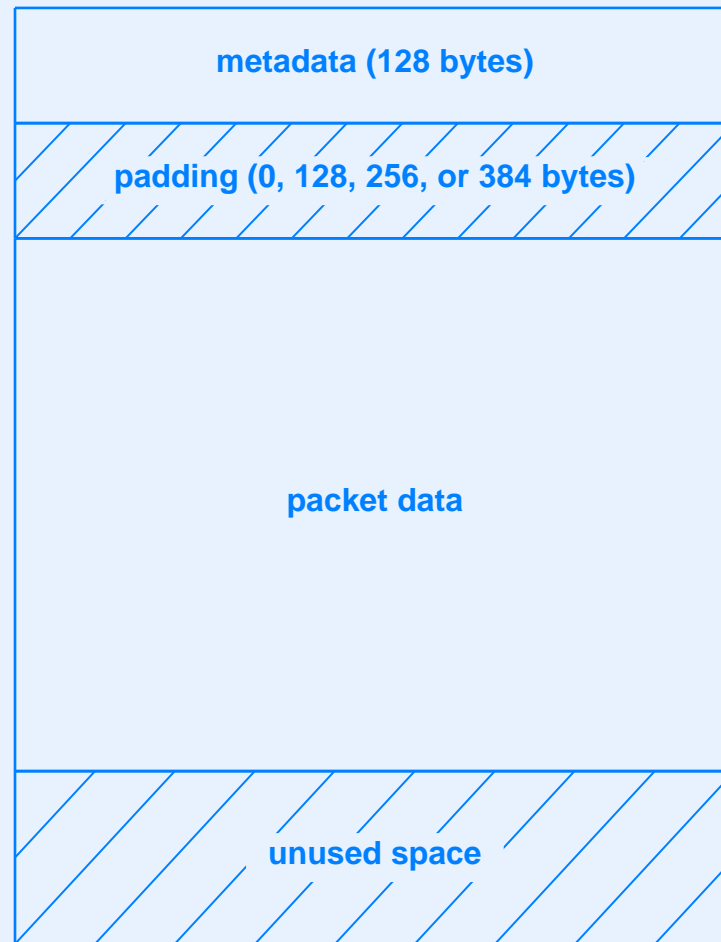
# Packet Buffers

- Mechanism provided by SDK
- Set of fixed-size buffers allocates in DRAM
- Typical buffer size is 2048
- Buffer holds
  - Packet (e.g., Ethernet frame)
  - Control information called *metadata*

# Placement Of A Packet In A Buffer

- Metadata occupies first 128 bytes of buffer
- Padding separates metadata from packet
- Optimizes memory access
  - DRAM organized into four banks
  - All banks can be accessed in parallel
  - Bank starts at 128 bytes beyond previous
- Goal: distribute packet headers evenly over banks

# Illustration Of Buffer Layout



- Padding is selected at random

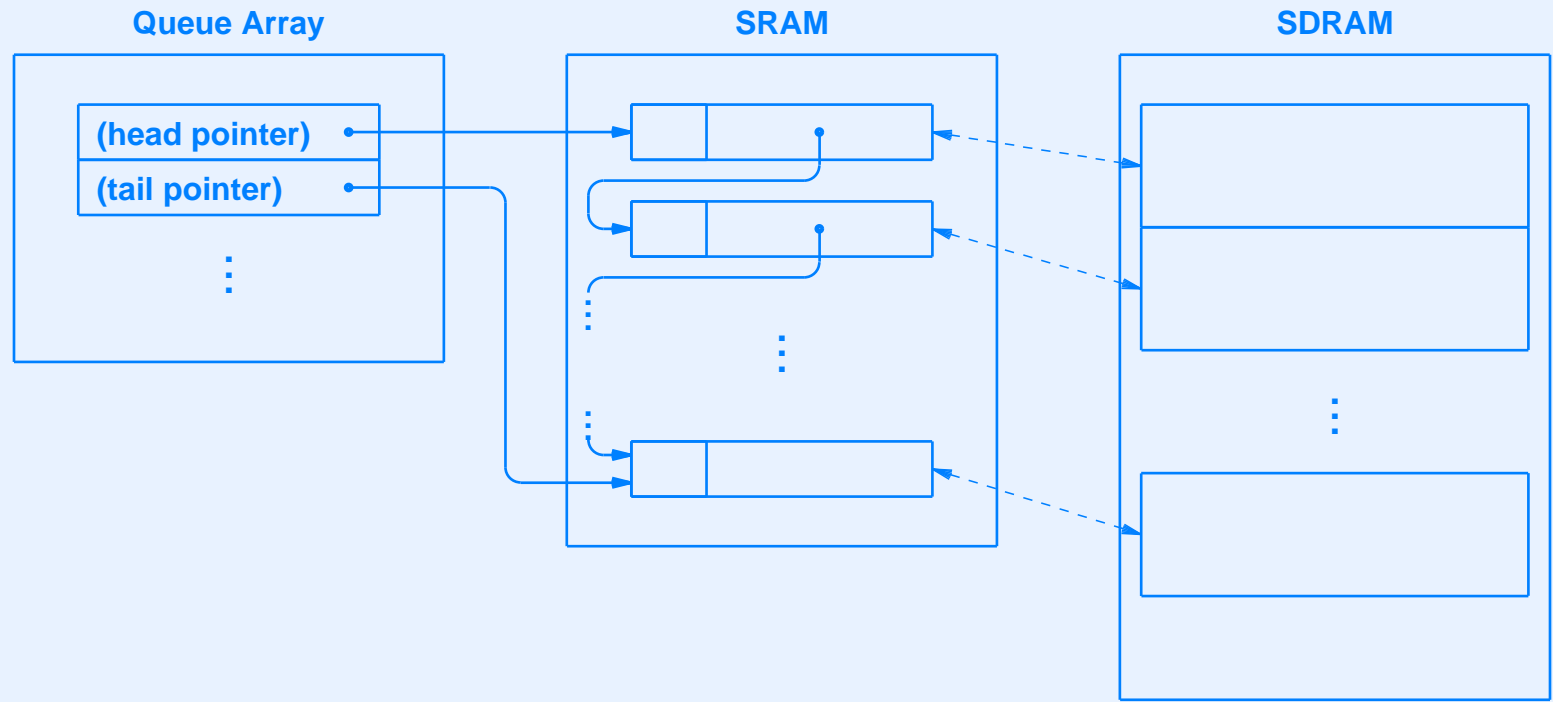
# Buffer Queues And Buffer Allocation

- Hardware support for high-speed buffer allocation
- Mechanism implements *First-In-First-Out (FIFO)*
  - Singly-linked list
  - Head and tail pointers
  - Two basic operations
    - \* Enqueue adds item at tail
    - \* Dequeue removes item from head

# Buffer Queues And Buffer Allocation (continued)

- FIFO mechanism
  - Provided by SRAM memory controller
  - Known as *Queue Array*
- Packet queues kept in DRAM
- Linear mapping between items in SRAM Queue Array and DRAM buffers

# Mapping Between Queue Array And DRAM Buffers



- Linear mapping between address of queue element in SRAM and address of buffer in DRAM

# Example Address Mapping

- Thirty-two buffers allocated in DRAM
  - Let  $B$  denote starting location
  - Assume buffers are contiguous
- Queue of thirty-two list elements allocated in SRAM
  - Let  $F$  denote starting location
  - Assume elements are contiguous
  - Known as *free buffer list*
- Let  $A$  be address of list element in SRAM returned by dequeue operation



## Example Address Mapping (continued)

- Address of corresponding buffer in DRAM is

$$\text{buffer address} = B + \frac{A - F}{\text{free list element size}} \times \text{buffer size}$$

- Where *buffer size* denotes the size of a packet buffer
- Optimization: to avoid division, precompute

$$DL\_DS\_RATIO = \frac{\text{buffer size}}{\text{free list element size}}$$

and

$$DL\_REL\_BASE = B - F \times DL\_DS\_RATIO$$

## Example Address Mapping (continued)

- Once constants are precomputed, a buffer address is found by:

$$\textit{bufferaddress} = A \times \textit{DL\_DS\_RATIO} + \textit{DL\_REL\_BASE}$$

- Note: if buffer and free list elements are powers of two, multiplication can be replaced by bitwise shift

# Buffer Handle

- List element address in SRAM
- Used as packet ID
- Is translated to buffer address as needed
- Consists of four bytes
  - Three bytes correspond to SRAM address
  - One byte used to pass additional information
    - \* Beginning of packet
    - \* End of packet
    - \* Count of segments in the packet

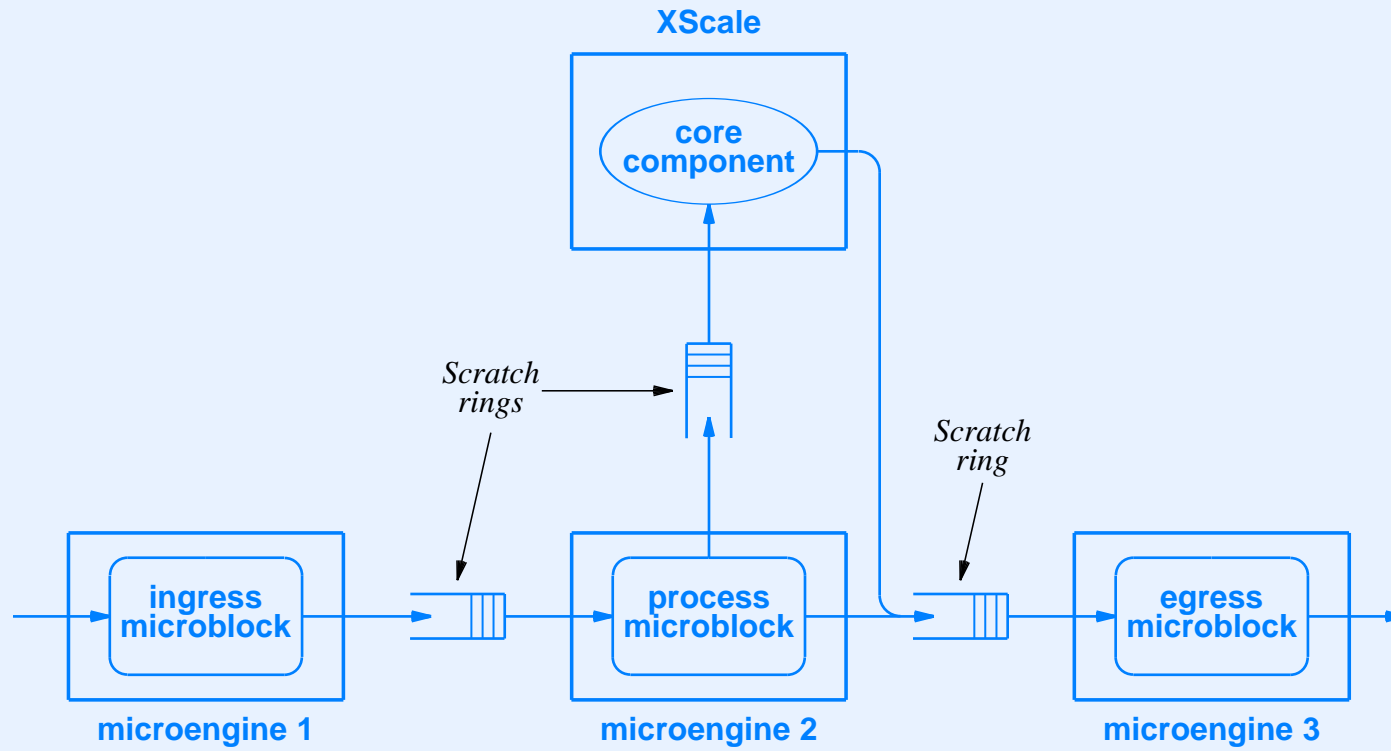
# Packet Discard

- Uses buffer handle
- Extremely efficient
- To discard, dequeue buffer handle on free list
- No other action required

# Packet Forwarding Mechanism

- Hardware mechanism used for interprocess communication
- Supported by both SRAM and Scratch memories
- Known as *Memory Ring*
- Controller implements insertion and extraction
- Requests serialized and atomic
- Used for high-speed forwarding of packets among microengines and XScale

# Illustration Of Scratch Rings



# Queue Array Hardware Limitation

- Hardware only has sixty-four Queue Array entries per SRAM channel.
  - IXP2400 has 2 SRAM channels
  - IXP28xx has 4 SRAM channels
- Consequence: cannot have arbitrarily many Rings

# Overcoming The Queue Array Limitation

- Resource manager allocates backup store in SRAM for each Queue Array entry
- When Queue Array exhausted, software
  - Chooses one Queue Array entry (typically LRU)
  - Copies entry to backup store
  - Uses hardware slot for new Queue Array
- Process is known as *spilling*
- CAM can be used used to choose entry to spill



# Core Processing

- Terminology
  - XScale is referred to as *core processor*
  - Software running on the XScale is called *core component*
- Core processing
  - Insufficient speed for fast path
  - Reserved for *exceptions*
- Note: core processor has access to memory rings and queues

# Summary

- Two major types of software
  - One or more *microblocks*
  - Core component
- Microblocks interconnected in pipeline
- SDK includes ingress and egress microblocks
- Packet buffers allocated in DRAM; free list kept in SRAM
- Address of list element in SRAM called *buffer handle*

# Summary

## (continued)

- Linear mapping translates buffer handle to DRAM address when needed
- Packet discard is trivial (return buffer handle to free list)
- Memory ring mechanism used for IPC
- Queue array hardware provides finite set of queues; values spilled to SRAM when necessary



**Questions?**

# **XXII**

## **XScale Facilities**

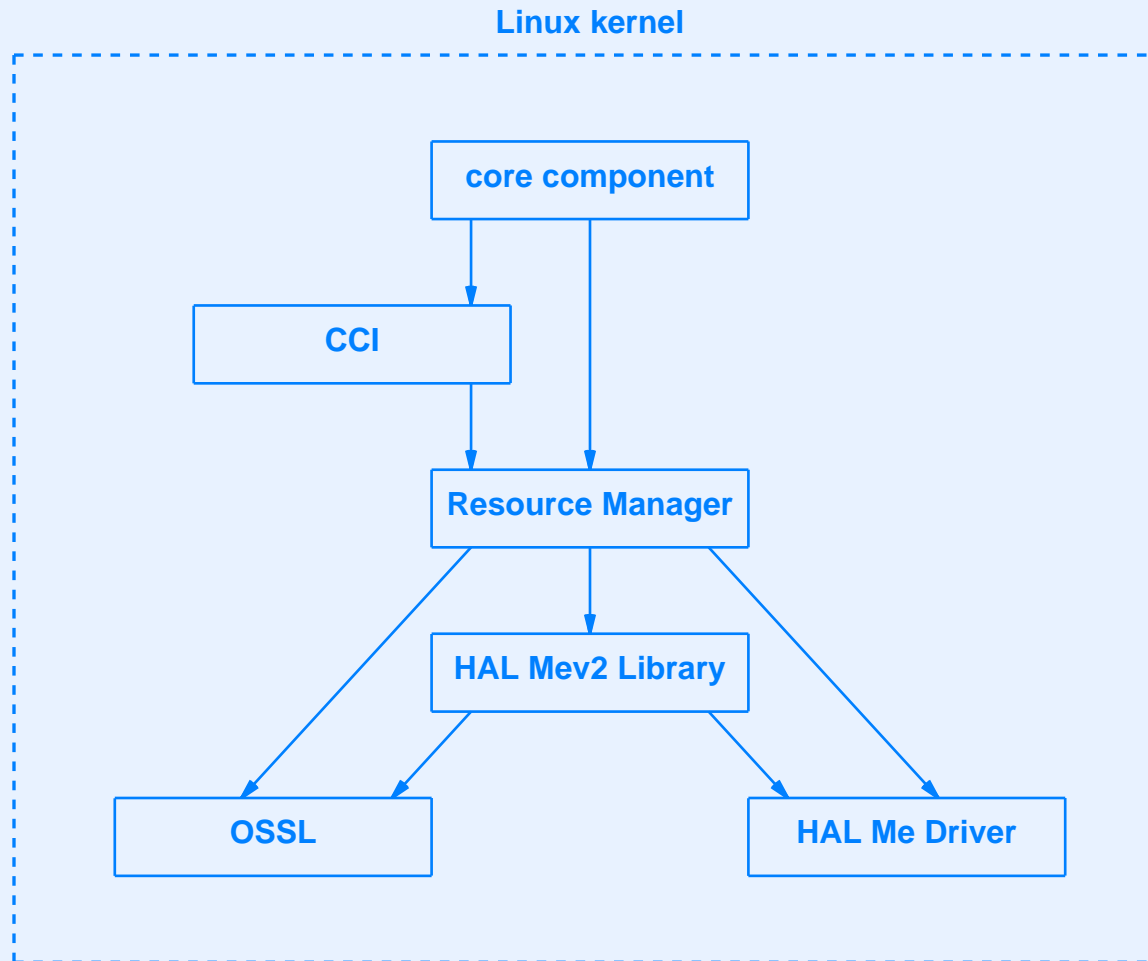
# XScale Responsibilities

- Loading microengine code
- Creating rings and / or queues for communication
- Allocating and reclaiming resources such as memory
- Patching symbols in microcode
- Starting and controlling microengine operation
- Providing an external interface for management
- Providing an interface to operating system facilities
- Doing slow path processing of exception packets

# Conceptual Organization Of XScale Software

- Several pieces of support software available
- Core component uses each piece directly or indirectly
- Each piece of support software implemented as a Linux *loadable kernel module*
- Core component also implemented as a kernel module

# Organization Of Software On XScale





# Core Component Infrastructure (CCI)

- Support module
- Provides facilities to
  - Create and run core components
  - Setup timers
  - Give each core component a private *execution engine*

# Resource Manager (RM)

- Support module
- Among most important
- Provides facilities used to
  - Access operating system services
  - Manage memory and translate addresses
  - Control microengines
  - Load code into microengines
  - Manage queue of exception packets sent to core component

# Operating System Specific Library (OSSL)

- Unfortunate name
- Role of the OSSL is operating system independence
- Core component calls OSSL function
- OSSL function calls underlying OS function
- Allows core component to remain independent of the underlying OS

# Hardware Abstraction Layer (HAL)

- Two HAL modules
  - Library *halMev2\_lib* provides control functions
  - Module *halMeDrv* provides access to microengine CSRs
- Isolate programmer from hardware details

# Memory Management

- SRAM, DRAM, and Scratch memories shared among microengines and XScale
- However
  - Addressing schemes used by microengines and XScale differ
  - Parallel processors cannot attempt to allocate memory without mutual exclusion
- SDK solution: all memory management (allocation and deallocation occurs through Resource Manager on XScale

# Memory Management Functions In The Resource Manager

- `ix_rm_mem_alloc`
  - Argument specifies DRAM, SRAM, or Scratch memory
  - Although XScale uses single address space, Resource Manager ensures allocation is made from specified memory
- `ix_rm_mem_free`

# Memory Allocation By Microengine

- Microengine
  - Can make an allocation directly
  - Must inform Resource Manager after allocation complete
  - Known as a *reservation*
- To make a reservation, core component calls
  - `ix_rm_mem_reserve`

# Allocation Of Local Memory

- Only visible to individual microengine
- However, Resource Manager provides functions that allow microengine to allocate and free Local Memory
  - `ix_rm_mem_local_alloc`
  - `ix_rm_mem_local_free`
  - `ix_rm_mem_local_reserve`



# Address Translation

- Functions in Resource Manager always return a virtual address in the XScale's address space
- Address must be translated to physical address before microengine can use it
- Address from microengine must be translated to virtual address before core component can use it
- All translation performed by core component using Resource Manager functions
  - `ix_rm_get_physical_offset`
  - `ix_rm_get_virtual_address`

# Ring And Queue Creation

- Provided by Resource Manager
- Three functions
  - ix\_rm\_hw\_queue\_create
  - ix\_rm\_hw\_sram\_ring\_create
  - ix\_rm\_hw\_scratch\_ring\_create
- Return a *handle*
- Value of handle can be predeclared and used as argument

# Ring And Queue Deletion

- Also performed by Resource Manager
- Two functions
  - ix\_rm\_hw\_queue\_delete
  - ix\_rm\_hw\_ring\_delete

# Ring And Queue Manipulation

- Performed by Resource Manager
- Functions are
  - ix\_rm\_hw\_enqueue
  - ix\_rm\_hw\_dequeue
  - ix\_rm\_hw\_ring\_put
  - ix\_rm\_hw\_ring\_get

# Buffer Management Facilities

- Special case of queue
- Function to create free list is
  - `ix_rm_buffer_free_list_create`

# Basic Form Of Core Component

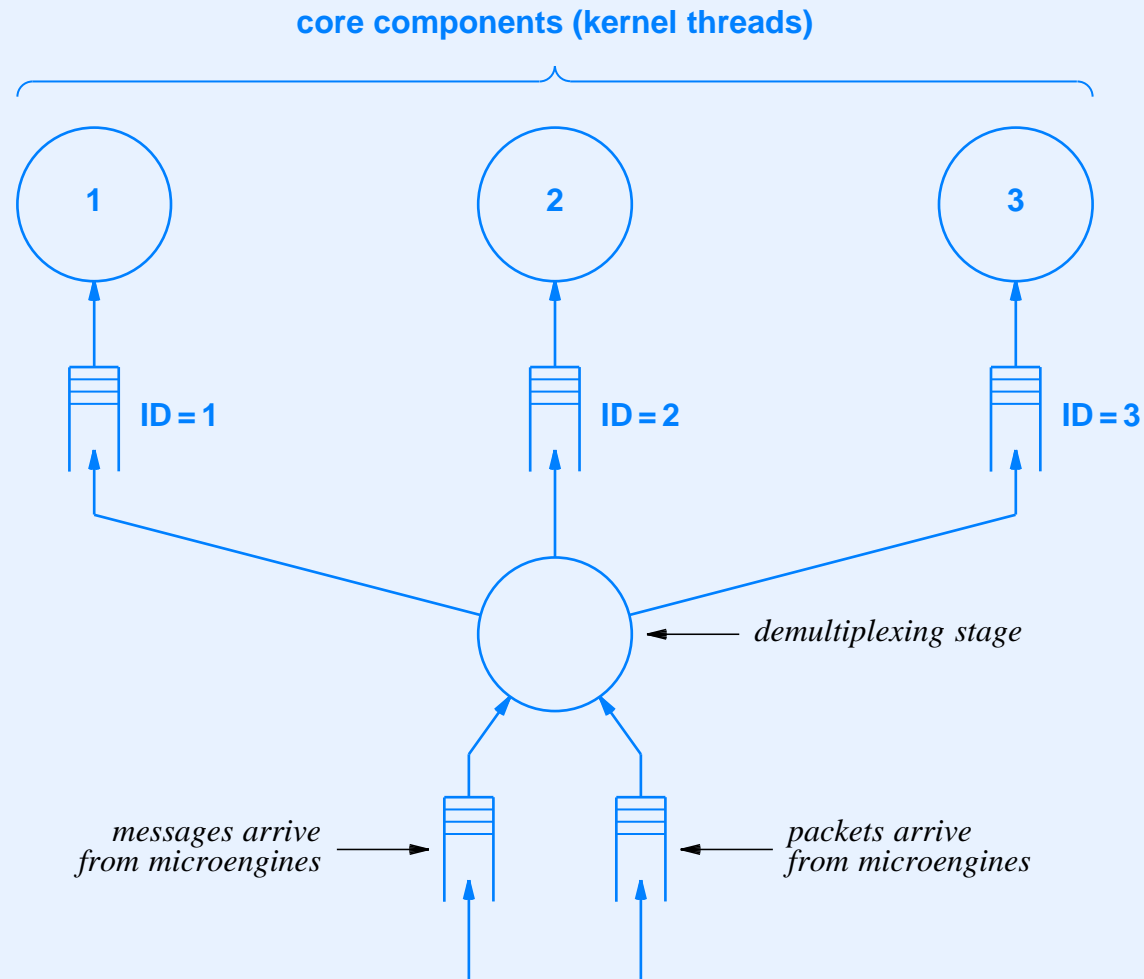
```
do forever {  
    wait for next packet from the microengines;  
    process the packet;  
}
```

- Note: although macroengines typically send a packet, the mechanism allows a microengine to send a “message”

# Core Processing

- Questions
  - How does a core component avoid using the CPU while waiting for a packet from the microengines?
  - If multiple core components exist to process packets, how is a given packet sent to the correct core component?
  - Can messages that arrive from microengines be processed out-of-order?
- Answer
  - Introduce an additional demultiplexing stage
  - To send a message, microengine interrupts the demultiplexing stage.

# Illustration Of Core Architecture





# Patching Symbols And Loading Microcode

- XScale loads microengine control store
- Symbolic references in code replaced by constant value
- Known as *patching*
- Allows values to change without recompilation
- Assembly *import* directive used to specify name and value to the assembler
  - `.import_var MY_CONSTANT`

# Difference Between Import And Defined Constants

- The code below produces an error (constant is too large)

```
#define MY_CONSTANT 0x12345678  
alu [ addr, MY_CONSTANT, +, 4 ]
```

- The following code compiles without an error (constant is truncated during patching)

```
.import_var MY_CONSTANT  
alu [ addr, MY_CONSTANT, +, 4 ]
```

# Use Of Immed32

- To avoid compiler error

```
.import_var MY_CONSTANT  
.reg tmp_value  
immed32(tmp_value, MY_CONSTANT)  
alu [ addr, tmp_value, +, 4 ]
```

# Resource Manager API

- Allows a core component to
  1. Read microcode from an external binary file and place it in a structure.
  2. Define the names and values of a set of imported symbols.
  3. Patch the microcode by replacing each imported variable reference with the appropriate value.
  4. Load the patched microcode into the microengine instruction store.

# Example Of Using The Resource Manager

- Load code from *my\_file* and patch imported constant *MY\_CONSTANT*

```
ix_rm_ueng_set_ucose(my_name);  
importSymbols[0].m_Name = "MY_CONSTANT";  
importSymbols[0].m_Value = 0x12345678;  
ix_rm_ueng_patch_symbols(me_number, 1, importSymbols);  
ix_rm_ueng_load();
```

# Resource Manager Functions To Control A Microengine

- To start a microengine

`ix_rm_ueng_start()`

- To stop a microengine

`ix_rm_ueng_stop()`

# Summary

- XScale software includes
  - Base operating system (Linux)
  - Core components
  - Support software
- Core component runs as a loadable Linux loadable kernel module
- Each support software system also runs as Linux loadable kernel module

# Summary

## (continued)

- Support software includes
  - Resource Manager
  - Core Component Infrastructure
  - Operating System Specific Library
  - Hardware Abstraction Layer
- Resource Manager offers API for items such as
  - Memory management and address translation
  - Queue and Ring allocation
  - Control of microengines





**Questions?**

# **XXIII**

## **Microengine Programming I**

# Microengine Code

- Many low-level details
- Close to hardware
- Written in (micro)assembly language

# Features Of Intel's Microengine Assembler

- Directives to control assembly
- Symbolic register names
- Macro preprocessor (extension of C preprocessor)
- Set of structured programming macros

# Statement Syntax

- General form:

*label: operator operands tokens*

- *label* is optional
- Interpretation of *tokens* depends on instruction

# Comment Statements

- Three styles available
  - C style (between `/*` and `*/`)
  - C++ style (`//` until end of line)
  - Traditional assembly style (`;` until end of line)
- Only traditional comments remain in code for intermediate steps of assembly

# Assembler Directives

- Begin with period in column one
- Can
  - Generate code
  - Control assembly process
- Example: associate *myname* with register five in the A register bank

```
.areg    myname    5    a
```

# Example Operand Syntax

- Instruction *alu* invokes the ALU

*alu* [ *dst*, *src*<sub>1</sub>, *op*, *src*<sub>2</sub> ]

- Four operands
  - Destination register
  - First source register
  - Operation
  - Second source register
- Two minus signs (--) can be specified for destination, if none needed



# Major ALU Operations

Operator	Meaning
+	Result is $src_1 + src_2$
-	Result is $src_1 - src_2$
B-A	Result is $src_2 - src_1$
B	Result is $src_2$
~B	Result is the bitwise inversion of $src_2$
AND	Result is bitwise <i>and</i> of $src_1$ and $src_2$
OR	Result is bitwise <i>or</i> of $src_1$ and $src_2$
XOR	Result is bitwise <i>exclusive or</i> of $src_1$ and $src_2$
+carry	Result is $src_1 + src_2 +$ carry from previous operation
-carry	Result is $src_1 - src_2 -$ carry from previous operation
~AND	Result is bitwise ( <i>not</i> $src_1$ ) <i>and</i> $src_2$
AND~	Result is bitwise ( $src_1$ <i>and</i> ( <i>not</i> $src_2$ ))
+8	Result is $src_1 + src_2$ with the first 24 bits set to zero
+16	Result is $src_1 + src_2$ with the first 16 bits set to zero

# ALU Shift Operations

- Shifts or rotates  $src_2$  before operation
- Syntax is

*alu\_shf [ dst, src<sub>1</sub>, op, src<sub>2</sub>, src<sub>2</sub>\_shift\_op ]*

# Memory Operations

- Programmer specifies
  - Type of memory
  - Direction of transfer
  - Address in memory (two registers used)
  - Starting transfer register
  - Count of words to transfer
  - Optional tokens

# Memory Operations

## (continued)

- General forms

sram [ *direction*, *xfer\_reg*, *addr<sub>1</sub>*, *addr<sub>2</sub>*, *count* ], *optional\_tokens*

dram [ *direction*, *xfer\_reg*, *addr<sub>1</sub>*, *addr<sub>2</sub>*, *count* ], *optional\_tokens*

scratch [ *direction*, *xfer\_reg*, *addr<sub>1</sub>*, *addr<sub>2</sub>*, *count* ], *optional\_tokens*

# Special Memory Operations

- Some memories offer special operations such as
  - Test-and-set
  - Atomic increment
- Operand *direction* used to specify special operations

# Memory Addressing

- Specified with operands  $addr_1$  and  $addr_2$
- Each operand corresponds to register
- Use of two operands can be used to
  - Scale to large memory
  - Use base + offset form

# Immediate Instruction

- Place constant in thirty-two bit register

*immed [ dst, ival, shift ]*

- Upper sixteen bits of *ival* must be all zeros or all ones
- Operand *shift* specifies bit shift

0	No shift
<<0	No shift (same as 0)
<<8	Shift to the left by eight bits
<<16	Shift to the left by sixteen bits

# Other Forms Of Immed Instruction

- Used to load part of a register

immed\_b0      Load byte zero (low-order byte) only

immed\_b1      Load byte one only

immed\_b2      Load byte two only

immed\_b3      Load byte three only

immed\_w0      Load word zero (low-order 16 bits) only

immed\_w1      Load word one only



# Register Names

- Usually automated by assembler
- Directives available for manual assignment

Directive	Type Of Assignment
<code>.addr name a</code>	Manual assignment to bank A
<code>.addr name b</code>	Manual assignment to bank B
<code>.reg name</code>	Automatic assignment

# Automated Register Assignment

*Intel's microengine assembler uses symbolic names for registers, and then maps each name to a specific register. A programmer can use directives to specify the mapping manually or can allow the assembler to choose a mapping; for general-purpose and next-neighbor registers, a programmer cannot mix automatic and manual assignments.*

# Register Names And Meanings

- Name denotes type of register

<b>Register Type</b>	<b>Relative</b>	<b>Absolute</b>
<b>General-purpose</b>	<b>register_name</b>	<b>@register_name</b>
<b>SRAM transfer</b>	<b>\$register_name</b>	-
<b>DRAM transfer</b>	<b>\$\$register_name</b>	-
<b>Next neighbor</b>	<b>n\$register_name</b>	-

# Register Allocation

- Hardware provides both *read* and *write* transfer registers
- Same numbers used
- Separate allocation functions

*.xfer\_read name*

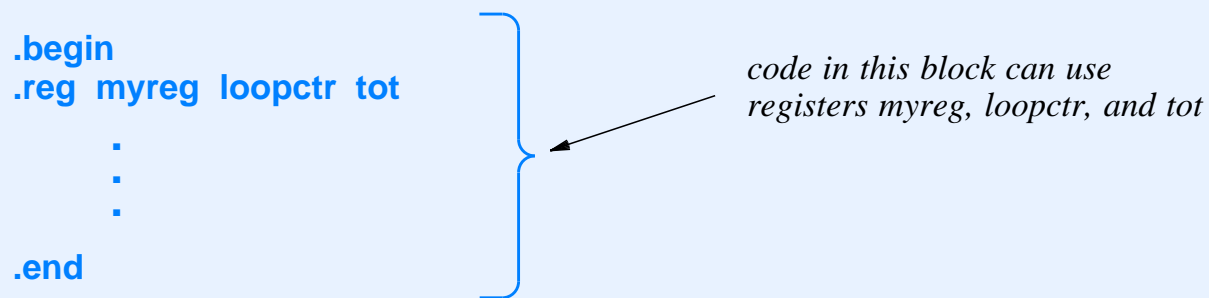
*.xfer\_write name*

# Local Register Scope, Nesting, And Shadowing

- Programmer
  - Uses *.begin* directive to declare register names
  - Defines register names
  - References names in instructions
  - Uses *.end* to terminate scope
- Assembler
  - Assigns registers
  - Chooses bank for each register
  - Replaces names in code with correct reference

# Illustration Of Automated Register Naming

- One or more register names specified after `.begin`
- Example

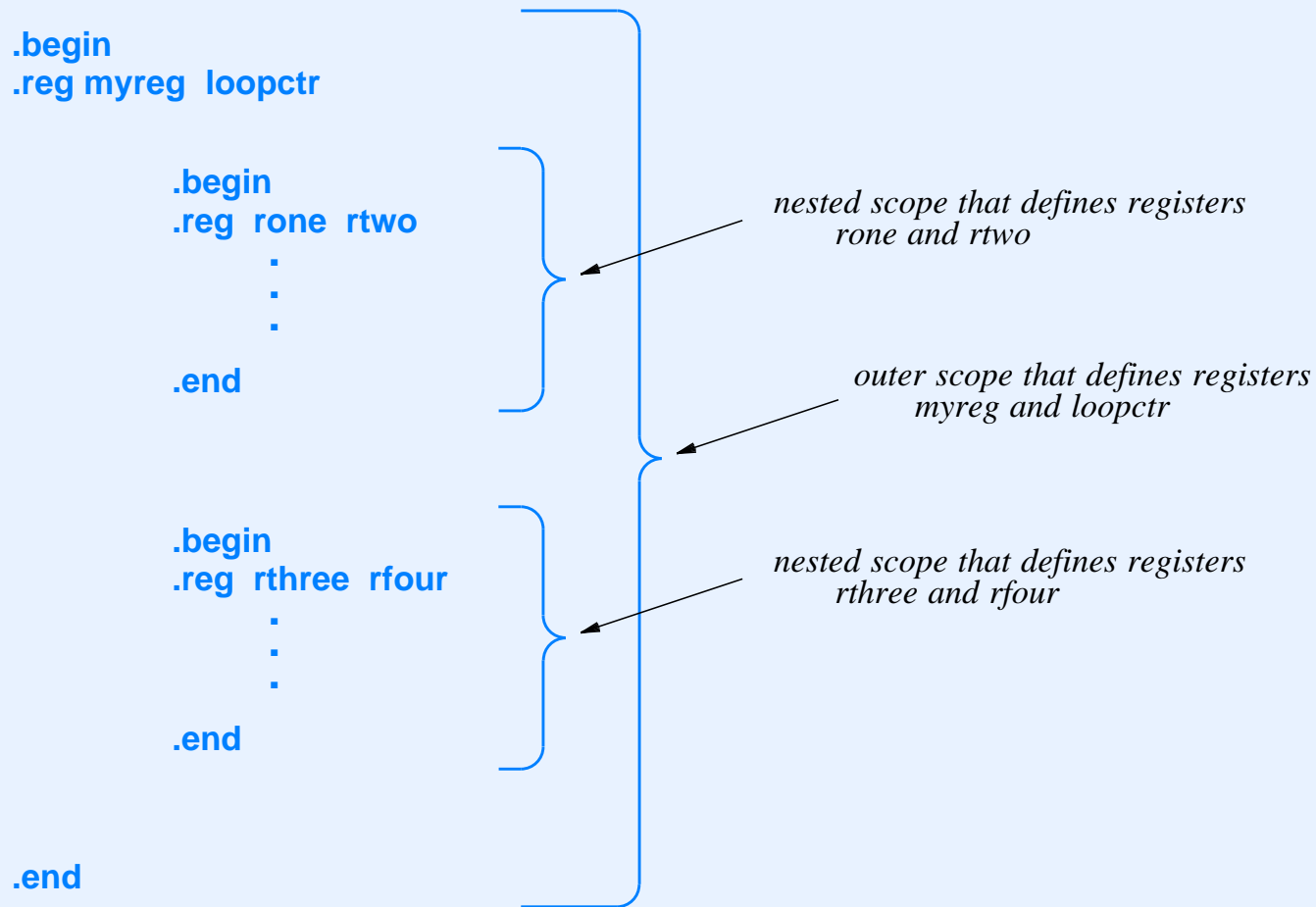


- Names valid only within scope

# Nested Scopes

- Programmer specifies `.begin` and `.end` pair inside a `.begin` `.end` pair
- Innermost scope has precedence
- Intel says inner declarations *shadow* outer declarations

# Illustration Of Nested Register Scope





# Register Assignments And Conflicts

- Operands must come from separate banks
- Some code sequences cause *conflict*
- Example:

$$Z \leftarrow Q + R;$$
$$Y \leftarrow R + S;$$
$$X \leftarrow Q + S;$$

- No assignment is valid
- Programmer must change code

# Macro Preprocessor Features

- File inclusion
- Symbolic constant substitution
- Conditional assembly
- Parameterized macro expansion
- Arithmetic expression evaluation
- Iterative generation of code

# Macro Preprocessor Statements

Keyword	Use
<b>#include</b>	Include a file
<b>#define</b>	Definition of a symbolic constant (unparameterized)
<b>#define_eval</b>	Definition of a symbolic constant equal to an arithmetic expression
<b>#undef</b>	Remove a previous symbolic constant definition
<b>#macro</b>	Start the definition of a parameterized assembly language macro
<b>#endm</b>	End a macro definition started with <b>#macro</b>
<b>#ifdef</b>	Start conditional compilation if specified symbolic constant has been defined
<b>#ifndef</b>	Start conditional compilation if specified symbolic constant has not been defined
<b>#if</b>	Start conditional compilation if expression is true
<b>#else</b>	Terminate current conditional compilation and start alternative part of conditional compilation
<b>#elif</b>	Terminate current conditional compilation and start another if expression is true
<b>#endif</b>	Terminate current conditional compilation
<b>#for</b>	Start definite iteration to generate a code segment a fixed number of times
<b>#while</b>	Start indefinite iteration to generate a code segment while a condition holds
<b>#repeat</b>	Start indefinite iteration to repeat a code segment as long as a condition holds
<b>#endloop</b>	Terminate an iteration

# Macro Definition

- Can occur at any point in program
- General form:

```
#macro name [ parameter1, parameter2, ... ]  
    lines of text  
#endm
```

# Macro Example

- Compute  $a = b + c + 5$

```
/* example macro add5 computes a=b+c+5 */  
#macro add5[a, b, c]  
    .begin  
    .reg tmp  
        alu[tmp, c, +, 5]  
        alu[a, b, +, tmp]  
    .end  
#endm
```

- Assumes values a, b, and c in registers

# Macro Expansion Example

- Call of *add5[*var1*, *var2*, *var3*]* expands to:

```
.begin
.reg tmp
    alu[tmp, var3, +, 5]
    alu[var1, var2, +, tmp]
.end
```

- Warning: because macros use textual substitution, illegal arguments can generate illegal code

# Repeated Generation Of A Code Segment

- Macro preprocessor
  - Supports *#while* statement for iteration
  - Uses *#define\_eval* for arithmetic evaluation
- Can be used to generate sequence of code blocks

# Example Of Repeated Code

- Preprocessor code:

```
#define LOOP 1
#while (LOOP < 4)
    alu_shf[reg, -, B, reg, >>LOOP]
#define_eval LOOP LOOP + 1
#endloop
```

- Expands to:

```
alu_shf[reg, -, B, reg, >>1]
alu_shf[reg, -, B, reg, >>2]
alu_shf[reg, -, B, reg, >>3]
```



# Structured Programming Directives

- Make code appear to follow structured programming conventions
- Include *break* statement ala C

Directive	Meaning
<b>.if</b>	<b>Conditional execution</b>
<b>.if_unsigned</b>	<b>Unsigned version of .if</b>
<b>.elif</b>	<b>Terminate previous conditional execution and start a new conditional execution</b>
<b>.elif_unsigned</b>	<b>Unsigned version of .elif</b>
<b>.else</b>	<b>Terminate previous conditional execution and define an alternative</b>
<b>.endif</b>	<b>End .if conditional</b>
<b>.while</b>	<b>Indefinite iteration with test before</b>
<b>.while_unsigned</b>	<b>Indefinite iteration (unsigned)</b>
<b>.endw</b>	<b>End .while loop</b>
<b>.repeat</b>	<b>Indefinite iteration with test after</b>
<b>.until</b>	<b>End .repeat loop</b>
<b>.until_unsigned</b>	<b>Unsigned version of .until</b>
<b>.break</b>	<b>Leave a loop</b>
<b>.continue</b>	<b>Skip to next iteration of loop</b>

# Example Of Conditional Compilation

```
.if ( conditional_expression )
    /* block of microcode statements */
.elif ( conditional_expression )
    /* block of microcode statements */
.elif ( conditional_expression )
    /* block of microcode statements */
    .
    .
    .
.else
    /* block of microcode statements */
.endif
```

# Tests That Can Be Used In A Conditional Expression

<b>Operator</b>	<b>Meaning</b>
<b>BIT</b>	Test whether a bit in a register is set
<b>BYTE</b>	Test whether a byte in a register equals a constant
<b>COUT</b>	Test whether a carry occurred on the previous operation
<b>CTX</b>	Test the currently executing thread number
<b>SIGNAL</b>	Test whether a specified signal has arrived for a thread
<b>INP_STATE</b>	Test whether the thread is in a specified state

# Mechanisms For Context Switching

- Context switching is voluntary
- Thread can execute:
  - *ctx\_arb* instruction
  - Reference instruction (e.g., memory reference)

# Argument To `ctx_arb` Instruction

- Determines disposition of thread
  - *voluntary*: thread suspended until later
  - *signal\_event*: thread suspended until specified event occurs
  - *kill*: thread terminated

# Context Switch On Reference Instruction

- Token added to instruction to control context switch
- Two possible values
  - *ctx\_swap*: thread suspended until operation completes
  - *sig\_done*: thread continues to run, and signal posted when operation completes
- Signals available for SRAM, DRAM, PCI bus, etc.

# Example Of Context Switch

- To perform context switch while waiting for DRAM access:

```
dram [read, $$rbuf0, base, 2, 4], sig_done [sig_name]
```

# Indirect Reference

- Poor choice of name
- Hardware optimization
- Found on other RISC processors
- Result of one instruction modifies next instruction
- Avoids stalls
- Typical use
  - Compute  $N$ , a count of words to read from memory
  - Modify memory access instruction to read  $N$  words



# Fields That Can Be Modified

- Microengine associated with a memory reference
- Starting transfer register
- Count of words of memory to transfer
- Context number of the hardware context executing the instruction (i.e., context to signal upon completion)
- The mask that specifies a set of signals

# How Indirect Reference Operates

- Programmer codes two instructions
  - ALU operation
  - Instruction with *indirect reference* set
- Note: destination of ALU operation is -- (i.e., no destination)
- Hardware
  - Executes ALU instruction
  - Uses result of ALU instruction to modify field in next instruction

# Example Of Indirect Reference

- Example code

```
alu_shf [ --, --, b, 0x13, << 16 ]  
scratch [ read, $reg0, addr1, addr2, 0 ], indirect_ref
```

- Memory instruction coded with count of zero
- ALU instruction computes count

# External Transfers

- Microengine cannot directly access
  - Memory
  - Buses ( I/O devices )
- Intermediate hardware units used
  - Known as *transfer registers*
  - Multiple registers can be used as large, contiguous buffer

# External Transfer Procedure

- Allocate contiguous set of transfer registers to hold data
- Start reference instruction that moves data to or from allocated registers
- Arrange for thread to wait until the operation completes

# Allocating Contiguous Registers

- Registers assigned by assembler
- Programmer needs to ensure transfer registers contiguous
- Assembler provides *.xfer\_order* directive
- Example: allocate four continuous SRAM input transfer registers

```
.reg $reg1 $reg2 $reg3 $reg4  
.xfer_order_rd $reg1 $reg2 $reg3 $reg4
```

- Notes
  - Ordering affects both read and write registers
  - Directive *.xfer\_order\_wr* available for output

# Summary

- Microengines programmed in assembly language
- Intel's assembler provides
  - Directives for structuring code
  - Macro preprocessor
  - Automated register assignment
- External data access performed through transfer registers



**Questions?**



# **XXIV**

## **Microengine Programming II**

# Specialized Memory Operations

- Ring and Queue manipulation
- Processor coordination (e.g., via atomic bit operations)
- Atomic memory operations (e.g *incr*, and *decr*)

# Ring And Queue Manipulation

- SRAM controller provides Queue Array mechanism
- XScale used to create buffers in Queue Array
- Microengine can allocate buffer from free list

*sram [ dequeue, xfer, src\_op<sub>1</sub>, src\_op<sub>2</sub> ], tokens*

- Handle placed in *xfer* register
- *src* operands encode memory channel and Queue Array number

# Ring And Queue Manipulation

- Microengine can return buffer to free list

`sram [ enqueue, --, src_op1, src_op2 ]`

- Note: macros assume variables *DL\_DS\_RATIO* and *DL\_REL\_BASE* have been defined

# Processor Coordination Via Bit Testing

- Provided by SRAM and Scratchpad memories
- Atomic operations on individual bits
- Mask used to specify bit in a word
- General form

*scratch [ cmd, \$xfer, addr<sub>1</sub>, addr<sub>2</sub> ]*

- Operands *addr<sub>1</sub>* and *addr<sub>2</sub>* added to form address
- Register *\$xfer* contains 32-bit mask

# Bit Manipulation Commands

<b>Operation</b>	<b>Meaning</b>
<b>set</b>	<b>Set the specified bits to one</b>
<b>clr</b>	<b>Set the specified bits to zero</b>
<b>test_and_set</b>	<b>Place the original word in the read transfer register, and set the specified bits to one</b>
<b>test_and_clr</b>	<b>Place the original word in the read transfer register, and set the specified bits to zero</b>

# Atomic Memory Operations

- Memory shared among
  - XScale
  - Microengines
- Need atomic increment to avoid incorrect results
- General form

*scratch [ operation, optional\_value, addr<sub>1</sub>, addr<sub>2</sub> ]*

- *optional\_value* depends on operation being performed

# Atomic Memory Operations (continued)

- Possible atomic operations include:

<b>Operation</b>	<b>Meaning</b>
<b>incr</b>	<b>Increment the specified word in memory</b>
<b>decr</b>	<b>Decrement the specified work in memory</b>
<b>add</b>	<b>Add the value in a transfer register to the specified word in memory</b>
<b>sub</b>	<b>Subtract the value in a transfer register to the specified word in memory</b>



# Critical Sections And Folding

- Piece of code that referenes shared variables known as *critical section*
- To ensure correctness, only one thread can execute a critical section at any time (*mutual exclusion*)
- IXP2xxx solution: *sequencing*
  - Set of threads placed in circular order
  - Thread passes control the “next” thread

# Steps Used For Sequencing

```
Let C be the context number (thread ID);
if ( C == 0 ) {
    wait for signal from “previous” microengine;
} else {
    wait for signal from context C - 1;
}

access the critical section;

if ( C == 7 ) {
    signal “next” microengine;
} else {
    signal context C + 1;
}
```

# Optimized Sequencing

- Steps for sequencing assume threads on a given microengine in consecutive positions of sequence
- To optimize data access
  - First thread on microengine copies shared variable into Local Memory
  - Threads on microengine sequence and use local copy
  - Last thread on microengine copies value back to external memory
- Can be dynamic: use CAM to test whether variable is in Local Memory

# Control And Status Registers (CSRs)

- IXP2xxx has dozens of CSRs
- Provide access to hardware units on the chip
- Allow processors to
  - Configure
  - Control
  - Interrogate
  - Monitor
- Access
  - XScale: mapped into address space
  - Microengines: special instructions

# Cap Instruction

- Used on microengines to access CSRs
- General form

*cap [ cmd, \$xfer\_data, csr\_address ]*

- *cmd* is *read*, *write*, or *fast\_wr*

# High-Speed CSR Access

- Some CSRs reachable through fast data path
- Command *fast\_wr* provides fast-path access
- General form

*cap [ fast\_wr, immediate\_data, CSR ]*

# Reflection

- Move data from a transfer register on one microengine to a transfer register on another microengine
- Uses *cap* instruction
- General form

*cap [ cmd, xfer, rem\_ME, rem\_reg, rem\_ctx, ref\_count ]*

# Local CSRs

- Refer to individual microengine
- Can be accessed in single cycle
- Microengine issues *local\_csr\_rd* or *local\_csr\_wr* instruction
- Example

`local_csr_wr [ CSR, src ]`



# Local CSRs

- Reading from local CSR requires two steps

`local_csr_rd [ CSR ]`

`immed [ destreg, 0 ]`

# Intel Dispatch Loop Macros

- Each microengine executes infinite loop
  - Each iteration checks for event and processes event
  - Events are low level (e.g., hardware device becomes ready)
  - Known as *dispatch loop*
- SDK includes over forty predefined macros related to dispatch loop

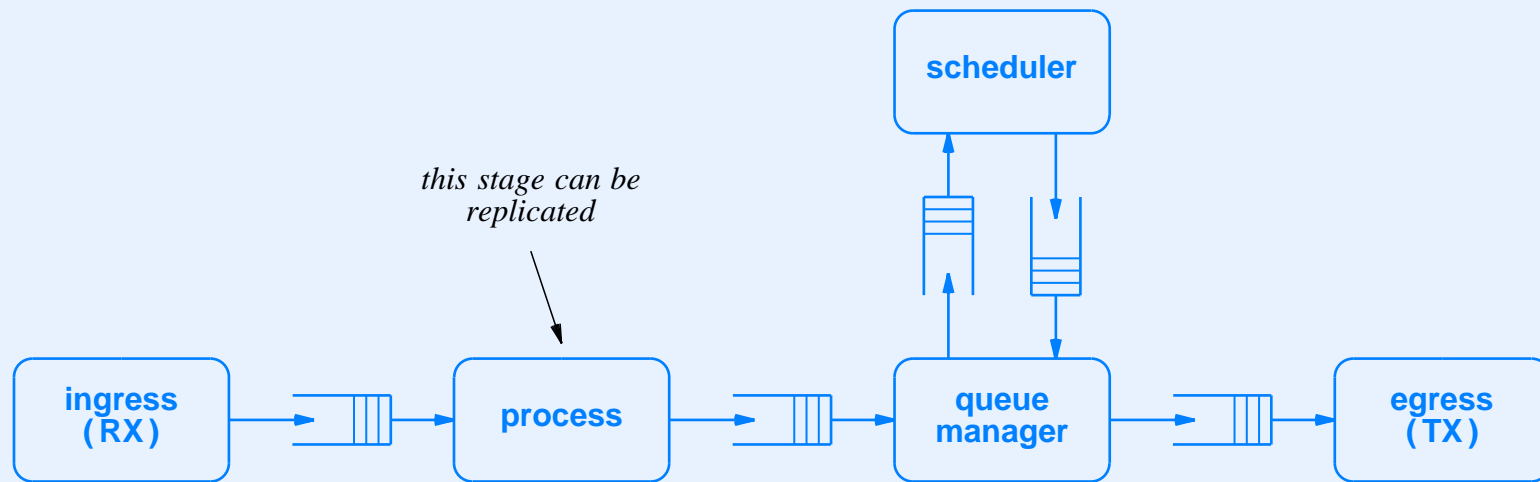
# Examples Of Predefined Dispatch Loop Macros

Macro	Purpose
<code>dl_buf_init</code>	Initialize the buffer API
<code>dl_buf_alloc</code>	Allocate a packet buffer
<code>dl_buf_free</code>	Deallocate a packet buffer
<code>dl_buf_get_desc</code>	Return SRAM pointer to metadata from a buffer
<code>dl_buf_get_data</code>	Return DRAM pointer to buffer data area
<code>dl_meta_init_cache</code>	Populate a metadata cache
<code>dl_meta_flush_cache</code>	Flush metadata cache to SRAM
<code>dl_meta_get_buffer_next</code>	Move to next buffer in a chain
<code>dl_meta_get_offset</code>	Find offset of data within a buffer
<code>dl_meta_get_free_list</code>	Find free list from which buffer was allocated
<code>dl_meta_get_rx_stat</code>	Extract receive status from a buffer
<code>dl_meta_get_buffer_size</code>	Find the size of data in a given buffer
<code>dl_meta_get_packet_size</code>	Find the total size of a packet
<code>dl_meta_get_input_port</code>	Find the input port over which packet arrived
<code>dl_meta_set_output_port</code>	Set the output port to which packet will be sent

# Traffic Management And Packet Scheduling

- Scheduling requires keeping one queue per scheduled flow
- Cannot be achieved with straightforward data pipeline
- Solution: add microblock outside main pipeline

# Arrangement Of Microblocks When Packet Scheduling Used



- Queue manager and scheduler operate independently

# Accessing Packet Header Fields

- Microengine instructions do not address memory directly
- Packet header loaded into transfer registers
- Many details

# Example Header Access Code (Part 1)

```
/* Allocate eight DRAM transfer registers to hold the packet header */
xbuf_alloc [ $$hdr, 8 ]

/* Reserve two general-purpose registers for the computation */
.begin
.reg base offset

/* Compute the DRAM address of the data buffer */
dl_buf_get_data [ base, dl_buffer_handle ]

/* Compute the byte offset of the start of the packet in the buffer */
dl_meta_get_offset [ offset ]
```

## Example Header Access Code (Part 2)

```
/* Load thirty-two bytes of data from DRAM into eight DRAM */  
/* transfer registers. Start at DRAM address base + offset */  
dram [ read, $$hdr0, base, offset, 4 ]
```

```
/* Inform the assembler that we have finished using the two */  
/* registers: base and offset */  
.end
```

```
/* Process the packet header in the DRAM transfer registers  
/* starting at register $$hdr */
```

...

```
/* Free the DRAM transfer registers when finished */  
xbuf_free [ $$hdr ]
```



# Dispatch Loop And Associated Variables

- Typical operation
  - Check for arrival of packet on Hardware Ring from previous microengine
  - Invoke procedure to process packet
  - Place packet on Hardware Ring that leads to next microengine
- Set of variables (registers) control operation of dispatch loop

# Examples Of Intel Dispatch Loop Variables

Variable	Size	Value And Meaning
exception_id	8 bits	ID of an exception handler on the XScale
exception_code	8 bits	A value passed with an exception packet
dl_next_block	8 bits	ID of next logical block for a packet
dl_buf_handle	32 bits	Buffer handle for start of the packet
dl_eop_buf_handle	32 bits	Buffer handle for end of the packet
buffer_size	16 bits	Length of the buffer containing the packet
packet_siz	16 bits	Total length of packet (across all buffers)
buffer_offset	16 bits	Offset of data from the start of the buffer
input_port	16 bits	Logical port over which the packet arrived
rx_stat	4 bits	Status flag bits (unicast, broadcast, etc.)
output_port_egress	24 bits	Port over which packet is to be sent
output_port_fabric	8 bits	Blade ID when multiple blades used
output_port_type	4 bits	Hardware type of output interface
cache_flags	4 bits	Control header caching (64 bytes of packet)
next_hop_id	32 bits	ID of the next hop for the packet
flow_id	32 bits	Flow ID for metering / policing
queue_id	16 bits	Output queue for traffic management

# Header Caching

- Packets reside in DRAM
- Accessing header fields is expensive
- To optimize access, copy header into Local Memory
- Think of copy as a *cache*
- SDK includes mechanisms to perform header caching

# Packet I/O

- Physical frame divided into sixty-four octet units for transfer
- Each unit known as *mpacket*
- Division performed by interface hardware
- Microengine uses MSF interface to transfer each mpacket separately
- Hardware set two bits in RBUF to specify whether
  - Mpacket is first packet of a frame
  - Mpacket is last packet of a frame
- Note: cell or small packet has both bits set

# Packet I/O (continued)

- No interrupts
- No DMA
- Dispatch loop in ingress or egress microblock uses polling
- Microengine performs transfer

# Ingress Packet Transfer

- Incoming mpacket moved from Receive BUF into
  - SRAM transfer registers
  - Directly into DRAM
- DRAM transfer has form

*msf [ cmd, --, addr<sub>1</sub>, addr<sub>2</sub>, count ], tokens*

# Ingress Packet Transfer (continued)

- Transfer to SRAM transfer register has the form

*msf [ cmd, \$xfer, addr<sub>1</sub>, addr<sub>2</sub>, count ]*

## Other I/O Details

- Microengine must
  - Check status of mpacket to determine if
    - \* MAC hardware detected problem (e.g., bad CRC)
    - \* Mpacket arrived with no problems
  - Check whether mpacket is first mpacket of a frame



# Example Of Packet Processing

- If mpacket is first of a frame, branch to *start\_of\_packet#*

```
alu [--, $rc, AND, 1 ]
```

```
br!=0 [ start_of_packet# ]
```

# Summary

- Special hardware facilities support
  - Hardware Queues and Rings
  - Bit testing
  - Atomic memory operations
  - Sequencing and folding
  - CSR access
- Microengine executes event loop known as *dispatch loop*
  - Checks for packets arriving
  - Calls macro(s) to process each packet
  - Sends packets to next specified destination

# Summary

## (continued)

- Intel supplies large set of dispatch loop macros
- Intel's SDK provides microblocks for ingress and egress
- Frame is divided into mpackets for transfer
- Hardware sets bits to specify whether incoming mpacket is first or last of a frame
- Microengine can transfer mpacket to SRAM transfer registers or directly to DRAM



**Questions?**

**XXV**

# **An Example Program**

# We Will

- Consider an example
- Examine all the user-written code
- See how the pieces fit together

# Choice Of Network System

- Used to demonstrate
  - Basic concepts
  - Code structure and organization
- Need to
  - Minimize code size and complexity
  - Avoid excessive detail
  - Ignore performance optimizations
- Example: Network Address Translator (NAT)

# NAT System Assumptions

- Only two connections: one to the ISP and one to a local network
- Both connections are Ethernet
- Traffic restricted to
  - TCP
  - UDP
  - ICMP echo and reply (ping)
- Applications do not pass IP address or protocol port information in the data stream

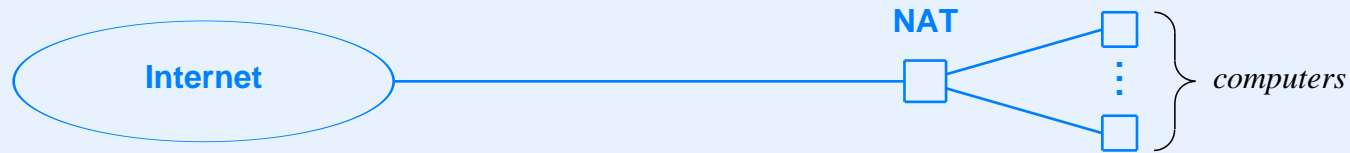


# NAT System Assumptions

## (continued)

- System will not handle fragmented datagrams or datagrams with IP options
- System will only handle communication initiated from local computers (i.e., computers within the site)
- Use XScale to handle all exceptions
- Will translate port numbers as well as addresses (NAPT)

# Conceptual NAT Topology

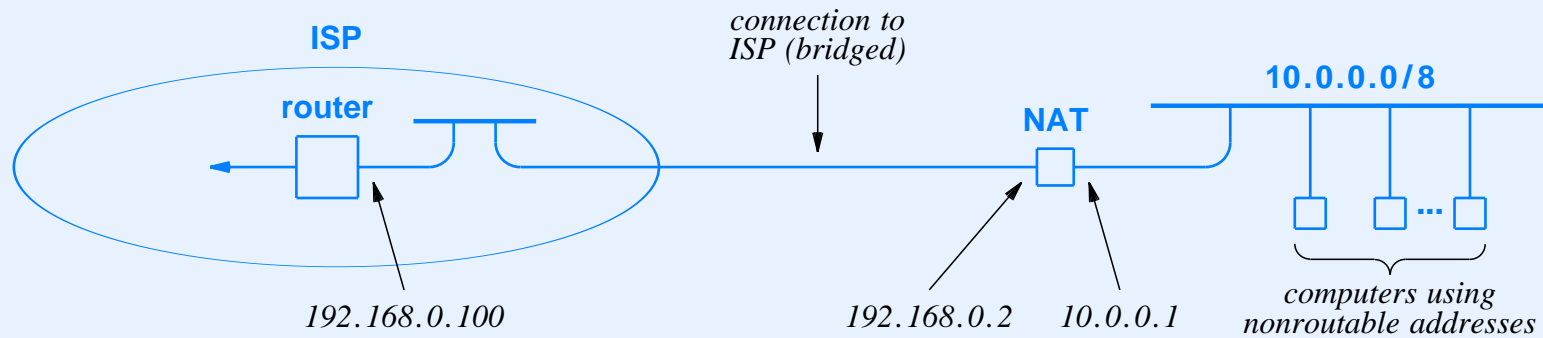


- NAT located between site and rest of Internet
- All packets between the site and the Internet pass through the NAT box

# Assumptions About Addresses

- Site has single valid IP address 192.168.0.2
- Default router at ISP has IP address 192.168.0.100
- Computers behind NAT box use net 10 addresses such as
  - 10.0.0.1
  - 10.0.0.5
  - 10.0.0.13

# Illustration Of NAT Addressing



# NAT

- Changes fields in packet headers
  - Source fields in outgoing packet
  - Destination fields in incoming packet
- Uses a table to store translation information

# Illustration Of NAT Translation Table

Local IP Address	Local Port or ID	Remote IP Address	Remote Port or ID	Protocol	New Port or ID
10.0.0.2	29000	128.10.2.1	80	TCP	1180
10.0.0.3	29000	128.10.2.1	80	TCP	1239
10.0.0.4	12	192.5.3.1	–	ICMP	1630

- Table shows three simultaneous connections
  - Computer 10.0.0.2 contacts 128.10.2.1:80
  - Computer 10.0.0.3 contacts 128.10.2.1:80
  - Computer 10.0.0.4 pings 192.5.3.1

# Ports, Identifiers, And Ping

- Each entry in NAT table corresponds to flow
- For TCP or UDP, flow is identified by
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number
  - Replacement source port used by NAT
  - Protocol

# Ports, Identifiers, And Ping (continued)

- For ping, flow is identified by
  - Source IP address
  - ID value in packet
  - Destination IP address
  - Replacement ID used by NAT
  - Protocol



# Dynamic NAT Table

- Outgoing packet used to create entry in NAT table
- Table is fixed size
- Consequence: when table is full, must delete old entry when adding a new entry

# NAT Table Management

- Each entry contains countdown timer field
- Timer value
  - Reset whenever entry used
  - Decrement every second
- When timer reaches zero, entry available for reuse
- When entry must be removed from full table, entry with oldest timer value is selected (LRU)

# Optimization

- To avoid arithmetic operations: use bit shift
- Timer value initialized with high-order bit set
- On each tick of the clock, shift right one bit
- When bit is shifted all the way to right, value becomes zero

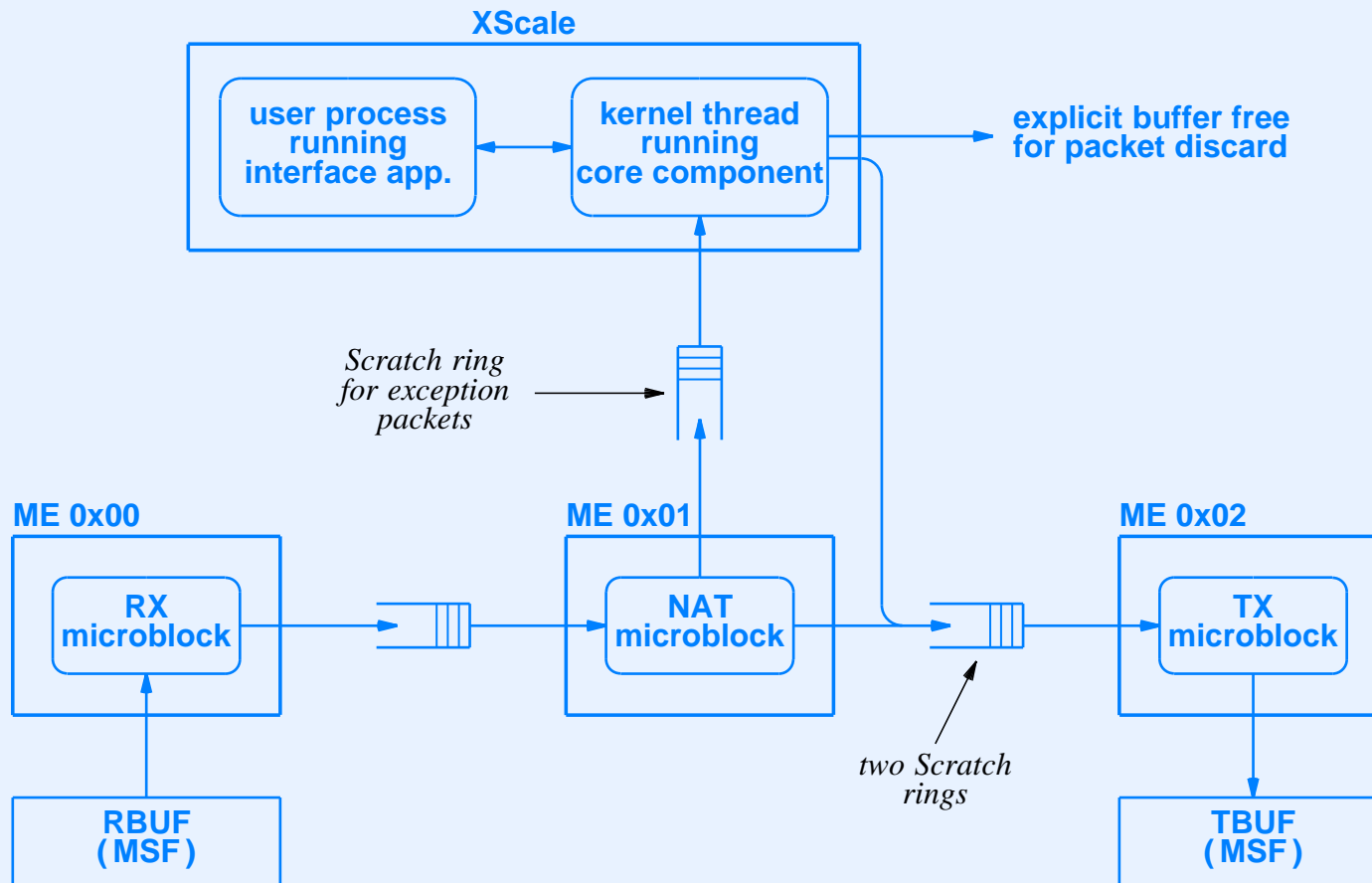
# Organization Of The Code

- Uses Intel's RX and TX microblocks to receive and send packets
- Single NAT microblock handles fast-path translation and forwarding
- Core component handles exceptions.

# Five Main Pieces Of Code

- Ingress (RX) microblock from Intel's SDK
- NAT microblock to handle the fast data path
- Egress (TX) microblock from Intel's SDK
- Core Component to handle exceptions ,User interface

# Illustration Of Interconnections



- Hardware rings used for interconnection

# Purpose Of Core Component

- System initialization. The core component performs the usual startup tasks by patching symbols in the microcode, loading microcode into microengines, and allocating memory.
- Exception packet processing. The core component handles packets for which address translation fails, and inserts new entries in the address translation table as necessary.
- Timer aging. Once each second, the core component decrements the timer associated with each entry in the address translation table.
- User interface interaction. The code component interacts with the user interface application to provide information or respond to commands.

# ARP Processing

- ARP processing needed to find hardware addresses of
  - Router at ISP
  - Local computers
- Local computers
  - Treat NAT box as default router
  - Send ARP request
- Router at ISP
  - Is default router for NAT box
  - Expects to receive ARP request



# Handling ARP

- NAT box assumes local computers will send ARP requests
- Single ARP request sent to router at ISP
  - Performed at startup
  - Packets for the ISP are discarded until a response arrives
- Values left in ARP cache indefinitely

# Implementation Of The NAT Microblock

- Poll *POS\_RX\_RING\_OUT* in infinite loop
- When packet available, extract buffer pointer from ring
- Read and classify packet
  - Place first 40 octets of packet in DRAM transfer registers
  - Note: caching described later
- Check destination Ethernet address
- Verify packet is IP carrying TCP, UDP, or ping

# Steps Taken In NAT Microblock (1)

```
do forever {
  if (input ring nonempty) {
    obtain buffer handle for next packet;
    if (Ethernet destination address invalid)
      discard the packet;
      continue;
  }
  if (not an IP packet || not one of TCP,
      UDP, or ICMP echo ) {
    send packet to core component;
    continue;
  }
  if (packet originates from local computer) {
    if (destination is local) {
      send packet to core component;
      continue;
    }
  }
  Check NAT table for outgoing match;
```

## Steps Taken In NAT Microblock (2)

```
} else /* packet originates from Internet */ {  
  if (destination is not the NAT system) {  
    send packet to core component;  
    continue;  
  }  
  Check NAT table for incoming match;  
}  
if (NAT table lookup failed) {  
  send packet to core component;  
  continue;  
}
```

## Steps Taken In NAT Microblock (3)

```
Replace fields in packet headers;  
Perform ARP lookup and set the Ethernet source  
    address and Ethernet destination address;  
Pass packet to TX microblock;  
}
```

# Header Caching And Alignment

- DRAM access extremely slow
- To optimize: cache packet header in Local memory
- Alignment
  - Local memory optimized for access by multiples of 4 bytes
  - Ethernet header contains 14 bytes
- Further performance enhancement: shift header right by two bytes when moving to Local memory (and shift back when storing in DRAM).
- Hardware instruction available

# Summary Of Comparisons Performed

<b>Field In Packet Header</b>	<b>Field In NAT Table</b>
<b>For outgoing packet (to the Internet)</b>	
Source IP address	Local IP address
Source Port (or ID)	Local Port or ID
Destination IP address	Remote IP address
Destination Port (or ID)	Remote Port or ID
IP Proto field	Protocol
<b>For incoming packet (from the Internet)</b>	
Source IP address	Remote IP address
Source Port (or ID)	Remote port or ID
Destination port (or ID)	New Port or ID
IP Proto field	Protocol

# Implementation Of NAT Lookup

- Hashing used to identify bucket
  - Extract fields from packet header
  - Hash to get bucket number  $0$  through  $N-1$
- Sequential search within bucket



# Hashing Details

- Fields selected for hashing depend on direction of packet
- Two hash tables
  - Forward table for packets traveling to the Internet (f\_nat\_table)
  - Reverse table for packets arriving from the Internet (r\_nat\_table)
  - Must be linked together

# Fields In NAT Table Entry

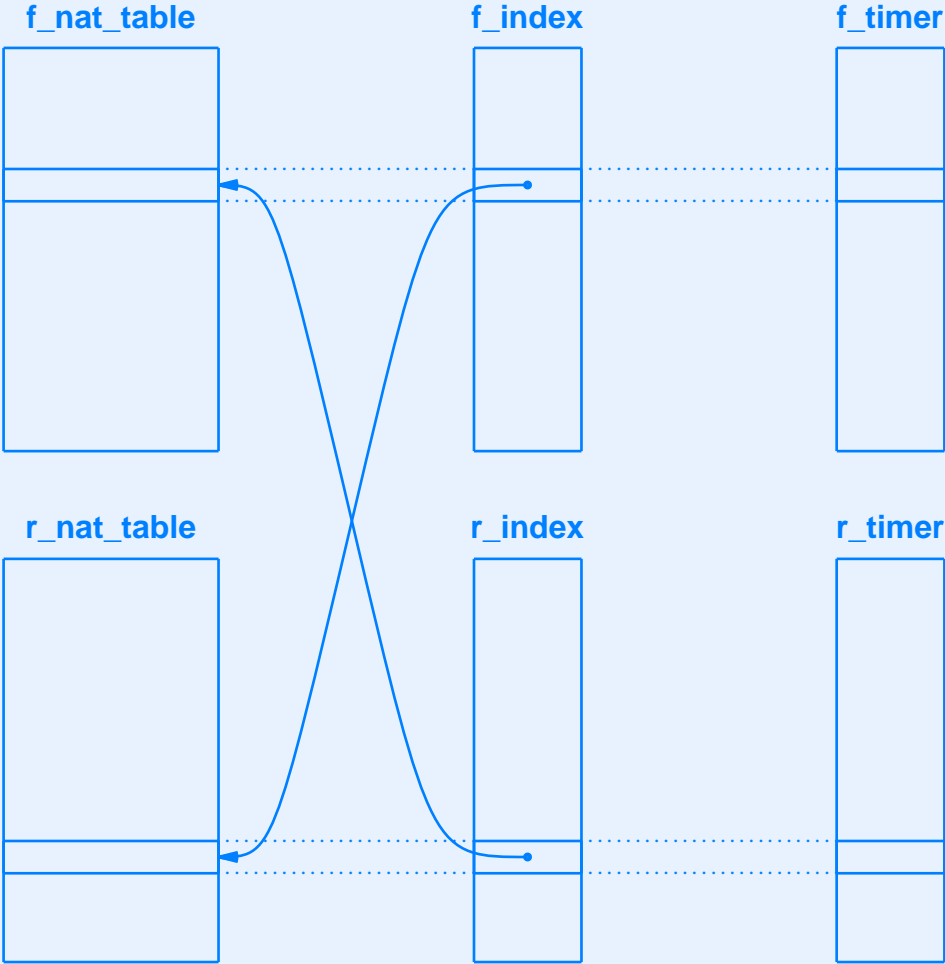
Size	Purpose
1 byte	Valid flag (only left-most bit is used)
1 byte	protocol
2 bytes	New port or ID
2 bytes	Local port (or ID)
2 bytes	Remote port (or ID)
4 bytes	Local IP address
4 bytes	Remote IP address

- Entry is exact multiple of DRAM access size

# Auxiliary Parallel Arrays

- Store timers and pointers
- Are parallel to the hash tables: the  $i$  item in an auxiliary table corresponds to the  $i$  item in a hash table
- Four auxiliary arrays used
  - Timer for forward entries,  $f\_timer$
  - Timer for reverse entries,  $r\_timer$
  - Index for forward entries,  $f-index$
  - Index for reverse entries,  $r-index$

# Illustration Of Auxiliary Arrays



# Header Fields That NAT Changes

## Outgoing packet (to the Internet)

<b>SOURCE IP</b>	←	<b>NAT system IP address</b>
<b>SOURCE PORT (or ID)</b>	←	<b>NAT New Port (or ID)</b>
<b>IP CHECKSUM</b>	←	<b>Adjusted IP header checksum</b>
<b>TCP or UDP CHECKSUM</b>	←	<b>Adjusted transport checksum</b>

## Incoming packet (from the Internet)

<b>DEST. IP Address</b>	←	<b>Local IP Address</b>
<b>DEST. PORT (or ID)</b>	←	<b>Local Port (or ID)</b>
<b>IP CHECKSUM</b>	←	<b>Adjusted IP header checksum</b>
<b>TCP or UDP CHECKSUM</b>	←	<b>Adjusted transport checksum</b>

# Definition Of Constants For Entire System (1)

```
/* NAT_shared_defs.h - constants shared by microcode and core code */
#define NAT_DEF_MAJOR_NUMBER 50 /* major number of NAT psuedo-device */
#define NAT_DRIVER_NAME "NAT" /* Name of driver for NAT pseudo-device */
#define NAT_DEV_FILE "/dev/NAT" /* File name for NAT pseudo-device */
#define PORTS_NUM 2 /* number of network interfaces */
#define NAT_IFC 0 /* external interfaces to outside world */
#define GW_IP 0xC0A80064 /* Router's IP address (192.168.0.100) */
#define NAT_CC_ID 65 /* ID of core component for exceptions */

/* Packet buffer parameters: 64MB of buffers, 2048 bytes per buffer */
#define NUM_BUFFERS 32*1024
#define BUF_SIZE 2048

/* Memory channels for free buffer list */
#define BUF_SRAM_CHAN 0
#define BUF_DRAM_CHAN 0

/* Counter sizes. These are implicitly defined in TX and RX building */
/* blocks. Namely, there are four 4-byte counters per port, which, for */
/* three ports gives 4*4*3=48 bytes for each counter region */
#define RX_CNTR_SIZE 48
#define TX_CNTR_SIZE 48
```

# Definition Of Constants For Entire System (2)

```
/* NAT table size, which must be a power of two */
#define NAT_TABLE_SIZE 128*1024 /* 128K entries */

/* Hash bucket size for NAT table = 2^HASH_BUCKET_SHIFT */
#define HASH_BUCKET_SHIFT 3
#define HASH_BUCKET_SIZE (1<<HASH_BUCKET_SHIFT)

/* NAT table bit mask */
#define NAT_TABLE_BIT_MASK ((NAT_TABLE_SIZE>>HASH_BUCKET_SHIFT)-1)

/* ARP table size, which must be a power of two */
#define ARP_TABLE_SIZE 256

/* ARP table bit mask */
#define ARP_TABLE_BIT_MASK (ARP_TABLE_SIZE-1)

/* Ethernet packet types that are recognized */
#define ETH_ARP 0x0806 /* ARP */
#define ETH_IP 0x0800 /* IP */

/* IP protocol types that are recognized */
#define IPT_UDP 17
#define IPT_TCP 6
#define IPT_ICMP 1

/* ICMP message types that are recognized */
#define ICMP_ECHO_REQ 8
#define ICMP_ECHO_REP 0
```

# Definition Of Constants For Entire System (3)

```
/* ARP operation types */
#define ARP_REQ 1
#define ARP_REP 2

/* timer aging interval in ms */
#define AGING_INTERVAL 1000 /* 1 sec */

/* maximum number of attempts to select a new (unused) NAT port value */
#define NEW_NPORT_ATTEMPS 30

/* maximum number of attempts to resolve gateway MAC address */
#define GW_MAC_RES_ATTEMPTS 3

/* number of microengines */
#define ME_NUM 8

/* size of one microengine cluster */
#define ME_CL_SZ 4

/* macro to represent Ethernet address as a byte array */
#define ETH2B(X) \
((char*)&(X))[0],((char*)&(X))[1],((char*)&(X))[2],((char*)&(X))[3],\
((char*)&(X))[4],((char*)&(X))[5]

/* macro to represent IP address as a byte array */
#define IP2B(X) \
((char*)&(X))[0],((char*)&(X))[1],((char*)&(X))[2],((char*)&(X))[3]
```



# Constants And Types For The User Interface (1)

```
/* NAT_types.h - types used by the core component and user interface */
```

```
typedef struct nat_entry_s { /* an entry in a NAT table */
    unsigned int valid : 1;
    unsigned int unused : 7;
    unsigned int prot : 8;
    unsigned int nport : 16;
    unsigned int lport : 16;
    unsigned int rport : 16;
    unsigned int ip_addr_loc;
    unsigned int ip_addr_rem;
} nat_entry;
```

```
typedef struct arp_entry_s { /* an entry in the ARP cache */
    unsigned int ip_addr;
    unsigned int eth_w0;
    unsigned short eth_w1;
    unsigned short ifnum : 15;
    unsigned short valid : 1;
    unsigned int unused;
} arp_entry;
```

# Constants And Types For The User Interface (2)

```
typedef struct net_if_s {          /* network interface structure */
    unsigned int ip_addr;
    unsigned int eth_w0;
    unsigned int eth_w1;
} net_if;

typedef enum nat_cmd_t {          /* possible ioctl commands for
                                /* the NAT pseudo-device          */
    SILENT=0, VERBOSE,
    GET_RX_COUNTER, CLR_RX_COUNTER,
    GET_TX_COUNTER, CLR_TX_COUNTER,
    GET_ARP_TABLE,  GET_NAT_TABLE,  GET_TIMER_TABLE
} nat_cmd;
#define INVALID_CMD -1
```

# Definitions Of Scratch Ring Constants

```
/* NAT_scratch_rings.h - constants used for Scratch Memory rings */

/* Ring used between the PACKET_RX and NAT microblocks */

#define PKT_RX_TO_NAT_SCR_RING          4
#define PKT_RX_TO_NAT_SCR_RING_SIZE    IX_SCRATCH_RING_SIZE_1K

/* First ring for communicating between NAT and Packet TX microblocks */

#define PACKET_TX_SCR_RING_0           6
#define PACKET_TX_SCR_RING_0_SIZE     IX_SCRATCH_RING_SIZE_256

/* Second ring for communicating between NAT and Packet TX microblocks */

#define PACKET_TX_SCR_RING_1           7
#define PACKET_TX_SCR_RING_1_SIZE     IX_SCRATCH_RING_SIZE_256

/* Third ring for communicating between NAT and Packet TX microblocks */

#define          PACKET_TX_SCR_RING_2           8
#define          PACKET_TX_SCR_RING_2_SIZE     IX_SCRATCH_RING_SIZE_256

/* Fourth ring for communicating between NAT and Packet TX microblocks */

#define PACKET_TX_SCR_RING_3           9
#define PACKET_TX_SCR_RING_3_SIZE     IX_SCRATCH_RING_SIZE_256
```

# Basic NAT Microblock (1)

```
/* NAT_microblock.uc - microcode for NAT processing */

/*****
/* Note: this file contains code for overall NAT processing, including */
/* code to obtain incoming packets from the packet_rx[] microblock, */
/* check the packet, perform NAT processing, if needed, and forward */
/* each packet to the correct transmit queue for the sphy_mphy4_tx[] */
/* microblock. */
*****/

#include <dl_system.h>
#include <stdmac.uc>
#include <dispatch_loop.uc>
#include <hardware.h>
#include <NAT_shared_defs.h>
#include <NAT_macros.uc>

/* Define NAT table location and parameters */
.import_var NAT_TABLE_BASE
#define_eval NAT_TABLE_BM NAT_TABLE_BIT_MASK
#define_eval HASH_BUCKET_SZ HASH_BUCKET_SIZE
#define_eval SHIFT_VAL 4+HASH_BUCKET_SHIFT
#define_eval NAT_TABLE_SZ NAT_TABLE_SIZE

/* Define ARP table location and parameters */
.import_var ARP_TABLE_BASE
#define_eval ARP_TABLE_BM ARP_TABLE_BIT_MASK
```

## Basic NAT Microblock (2)

```
/* Define timer table location */
.import_var TIMER_TABLE_BASE

/* Obtain the default gateway IP address */
.import_var GATEWAY_IP_ADDR

/* Obtain configurations for each network interface */
.import_var IF0_IP
.import_var IF1_IP
.import_var IF0_ETH_W0
.import_var IF0_ETH_W1
.import_var IF1_ETH_W0
.import_var IF1_ETH_W1

#define_eval NAT_IP_ADDR IF/**/NAT_IFC/**/_IP

/* Define Local memory addresses */
#define STEP 64
#define LM_ADDR0_0 0
#define_eval LM_ADDR0_1 LM_ADDR0_0+STEP
#define_eval LM_ADDR0_2 LM_ADDR0_1+STEP
#define_eval LM_ADDR0_3 LM_ADDR0_2+STEP
#define_eval LM_ADDR0_4 LM_ADDR0_3+STEP
#define_eval LM_ADDR0_5 LM_ADDR0_4+STEP
#define_eval LM_ADDR0_6 LM_ADDR0_5+STEP
#define_eval LM_ADDR0_7 LM_ADDR0_6+STEP
```

# Basic NAT Microblock (3)

```
/* **** */
/* Specify signals and registers */
/* **** */

.sig sig_scr_get           ; signal for scratch get
.sig sig_scr_put          ; signal for scratch put
.sig sig_pkt_hdr          ; signal for packet header read
.sig sig_dram_wr          ; signal for dram write done
.reg temp                 ; GPR for intermediate data
.reg zero                 ; GPR containing constant value 0
.reg one                  ; GPR containing constant value 1
.reg ring                 ; scratch ring
.reg port                 ; input port number
.reg $txreq               ; tx request to put on scratch rings
.reg eth_ip               ; GPR containing ETH_IP constant (0x0800)
.reg NAT_ip               ; GPR containing NAT box IP address
.reg ctx_num              ; context number of the current thread
.reg if_out               ; output interface to forward packet to
.reg EthDstW0 EthDstW1   ; Ethernet address registers
.reg f_nat_table          ; GPR with NAT table base
.reg nat_tab_bit_mask     ; GPR with NAT table bit mask (size - 1)
.reg r_nat_table          ; GPR with reverse NAT table base
.reg arp_tab              ; GPR with ARP table base
.reg arp_tab_bit_mask     ; GPR with ARP table bit mask (size - 1)
.reg f_timer              ; GPR with timer table base
.reg r_timer              ; GPR with reverse timer table base
.reg gateway_ip           ; GPR with default gateway IP address
.reg nat_port             ; GPR with port to substitute
.reg if_ip if_eth_w0 if_eth_w1 ; network interface settings
.reg IpHlen IpSrc IpDst IpProt SrcPort DstPort ; flow 5-tuple
```

# Basic NAT Microblock (4)

```
/* Allocation of transfer registers */
xbuf_alloc[ $\$$ pkt_hdr,2,read_write]
xbuf_alloc[ $\$$ entry_w,4,read_write]
xbuf_alloc[ $\$$ iphdr,10,read_write]
xbuf_alloc[ $\$$ rdata, RX_TO_FUNC_MSG_SIZE, read]

/*****/
/* Data initialization          */
/*****/

/* Frequently used constants */

immed[zero, 0] /* 0 */
immed[one, 1]  /* 1 */
immed32(eth_ip,ETH_IP) /* Ethernet type IP */

/* Constants that are specific to NAT */
immed32(NAT_ip,NAT_IP_ADDR)
immed32(f_nat_table,NAT_TABLE_BASE)
immed32(nat_tab_bit_mask,NAT_TABLE_BM)
immed32(arp_tab,ARP_TABLE_BASE)
immed32(arp_tab_bit_mask,ARP_TABLE_BM)
immed32(f_timer,TIMER_TABLE_BASE)
immed32(gateway_ip,GATEWAY_IP_ADDR)
#define_eval NAT_TABLE_SZ_B (NAT_TABLE_SZ<<4)
immed32(temp,NAT_TABLE_SZ_B)
alu[r_nat_table,f_nat_table,+,temp]
immed32(temp,NAT_TABLE_SZ)
alu[r_timer,f_timer,+,temp]
```

## Basic NAT Microblock (5)

```
/* Byte alignment setting */
local_csr_wr[BYTE_INDEX,2]

/* Obtain the current context number */
local_csr_rd[active_ctx_sts]
immed[ctx_num,0]
alu[ctx_num, ctx_num, AND, 0x07]

/* Set a Local memory address */
.if (ctx()==0)
    immed[temp,LM_ADDR0_0]
.elif (ctx()==1)
    immed[temp,LM_ADDR0_1]
.elif (ctx()==2)
    immed[temp,LM_ADDR0_2]
.elif (ctx()==3)
    immed[temp,LM_ADDR0_3]
.elif (ctx()==4)
    immed[temp,LM_ADDR0_4]
.elif (ctx()==5)
    immed[temp,LM_ADDR0_5]
.elif (ctx()==6)
    immed[temp,LM_ADDR0_6]
.else
    immed[temp,LM_ADDR0_7]
.endif
local_csr_wr[ACTIVE_LM_ADDR_0,temp]
```



# Basic NAT Microblock (6)

```
/*
*****
*/
/*      Main loop      */
/*
*****
*/
start#:
/* Read a packet from RX scratch ring */
alu_shf[ring, --, B, PKT_RX_TO_NAT_SCR_RING, <<2]
scratch[get,$rdata0,0,ring,RX_TO_FUNC_MSG_SIZE],
        sig_done[sig_scr_get]

/* Reset the exception register */
alu[dl_exception_reg, --, b, 0]

/* Wait for the RX ring read to finish */
ctx_arb[sig_scr_get]

/* Check if ring is empty */
alu[--, $rdata0, -, 0]
beq[ring_empty#]

/* Ring is not empty */
alu[dl_buf_handle,--,b,$rdata0] /* set buffer handle */
alu[dl_eop_buf_handle, --,b,$rdata1] /* get eop parameter */
alu[dl_meta1,--,b,$rdata2] /* get data offset */
alu[port, 0xF, AND, $rdata4, >>16] /* get input port */

/* Ignore packets from ports other than 0 or 1 */
alu[--,port,-,PORTS_NUM]
bge[drop#]
```

# Basic NAT Microblock (7)

```
/* Read the packet header (40 bytes) and assume Ethernet */
eth_iphdr_load(dl_buf_handle, sig_pkt_hdr)

/* If frame type is not IP, send to the core */
alu[temp,--,b,$$iphdr3,>>16]
alu[--,temp,xor,eth_ipt]
bne[exception#],defer[2] /* defer - save some cycles here */
alu[EthDstW0,--,b,$$iphdr0]
ld_field_w_clr[EthDstW1,1100,$$iphdr1]

/* At this point the code has an IP packet; check the type */
alu[IpProt,0xFF,and,$$iphdr5]
br=byte[IpProt,0,IPT_TCP,tcp_udp_icmp#] /* check for TCP */
br=byte[IpProt,0,IPT_UDP,tcp_udp_icmp#] /* check for UDP */
br!=byte[IpProt,0,IPT_ICMP,exception#] /* check for ICMP */
```

tcp\_udp\_icmp#:

```
/* The packet carries TCP, UDP or ICMP */

/* Find the network interface data for the input port */
net_if_data_get(port,if_ip,if_eth_w0,if_eth_w1)

/* Verify that the Ethernet destination matches our address */
alu[--,if_eth_w0,xor,EthDstW0]
bne[exception#],defer[1]
alu[--,if_eth_w1,xor,EthDstW1]
bne[exception#],defer[2]
```

## Basic NAT Microblock (8)

```
/* Compute the IP header size */
alu[IpHlen,0xF,and,$$iphdr3,>>8]

/* To simplify the code, we do not deal with IP options. */
/* If options are present, drop the packet */
alu[--,IpHlen,-,5]
bgt[exception#],defer[3]

/* Store a copy of the IP header in local memory */
byte_align_be[--,$$iphdr3]
byte_align_be[*l$index0[0],$$iphdr4]
byte_align_be[*l$index0[1],$$iphdr5]
byte_align_be[*l$index0[2],$$iphdr6]
byte_align_be[*l$index0[3],$$iphdr7]
byte_align_be[*l$index0[4],$$iphdr8]
byte_align_be[*l$index0[5],$$iphdr9]
byte_align_be[*l$index0[6],0]

/* Obtain the IP source and destination addresses */
alu[IpDst,--,b,*l$index0[4]]
alu[--,if_ip,xor,IpDst]
/* Branch if destination IP is local (i.e., the NAT box) */
beq[local_dst#],defer[1]
alu[IpSrc,--,b,*l$index0[3]]
```

# Basic NAT Microblock (9)

```
/* At this point the packet contains TCP,UDP or ICMP, and has */
/* a non-local destination address. If the packet is incoming, */
/* drop it. If the packet is outgoing, perform NAT translation */
/* and send the packet to the Internet. */
alu[--,port,xor,NAT_IFC]
beq[exception#]

/* Read the source and destination ports (or ICMP type and ID) */
read_src_and_dst_ports(NON_LOCAL_DST,IpHlen,IpProt,
                      SrcPort,DstPort)
.if (IpProt == IPT_ICMP)
    /* If the packet is ICMP, but not an echo request, */
    /* send the packet to the core as an exception */
    alu[--,DstPort,xor,ICMP_ECHO_REQ]
    bne[exception#],defer[1]
    alu[DstPort,--,b,0]
.endif
/* Perform NAT lookup for an outgoing packet */
nat_lookup_outgoing(IpSrc,SrcPort,IpDst,DstPort,IpProt,
                   nat_port,if_out)
alu[SrcPort,--,b,nat_port]
```

# Basic NAT Microblock (10)

```
tx_pkt#:
/* If NAT lookup failed, send the packet to core */
/* as an exception */
alu[--,--,~b,nat_port]
beq[exception#]

.set if_out /* inserted to prevent an assembler warning */

/* At this point, NAT lookup has been successful, and the ARP */
/* table must be consulted to determine the correct Ethernet */
/* address for the frame. */
alu[dl_exception_reg, --, b, 1,<<10]
arp_lookup(if_out,IpDst,EthDstW0,EthDstW1)
alu[dl_exception_reg, --, b, 0]

/* Modify the packet header */
modify_and_save_packet_header(if_out,EthDstW0,EthDstW1,IpHlen,
                               IpProt,IpSrc,IpDst,SrcPort,DstPort)

/* Create a TX request for transmit queue */
alu[temp, --, b, if_out, <<24] /* 27:24 output port */
ld_field[temp, 0111, dl_buf_handle] /* 23:00 buffer handle */
alu[$txreq, temp, OR, one, <<31] /* 31 valid bit */
/* bits 31:28 reserved */

/* Jump to Scratch ring write for the corresponding port */
alu[temp, --, b, if_out, <<2]
jump[temp,write_ring0#],targets[write_ring0#,write_ring1#,\
                               write_ring2#,write_ring3#]
```

# Basic NAT Microblock (11)

```
write_ring0#:  
    write_tx_ring(0,start#)  
write_ring1#:  
    write_tx_ring(1,start#)  
write_ring2#:  
    write_tx_ring(2,start#)  
write_ring3#:  
    write_tx_ring(3,start#)  
  
    /* If Scratch ring is full -- wait voluntarily */  
full_ring0#:  
    ctx_arb[voluntary],br[write_ring0#]  
full_ring1#:  
    ctx_arb[voluntary],br[write_ring1#]  
full_ring2#:  
    ctx_arb[voluntary],br[write_ring2#]  
full_ring3#:  
    ctx_arb[voluntary],br[write_ring3#]  
  
local_dst#:  
    /* If the Destination IP address in an incoming datagram is */  
    /* not the address of the NAT box address, send the packet */  
    /* to the core as an exception. */  
    alu[--,IpDst,xor,NAT_ip]  
    bne[exception#]
```

# Basic NAT Microblock (12)

```
/* At this point the incoming packet contains TCP, UDP, */
/* or ICMP and has a local IP destination. Read the source */
/* and destination ports. */
read_src_and_dst_ports(NON_LOCAL_SRC, IpHlen, IpProt,
                      SrcPort, DstPort)
.if (IpProt == IPT_ICMP)
    /* If the packet is ICMP, but not an echo reply, */
    /* send the packet to the core as an exception. */
    alu[--, SrcPort, xor, ICMP_ECHO_REP]
    bne[exception#], defer[1]
    alu[SrcPort, --, b, 0]
.endif
/* Perform NAT lookup for an incoming packet */
nat_lookup_incoming(IpDst, DstPort, IpSrc, SrcPort, IpProt,
                   nat_port, if_out)
alu[DstPort, --, b, nat_port]
br[tx_pkt#] /* jump to the transmission code */
```

# Basic NAT Microblock (13)

```
exception#:  
    /* send to the NAT core component */  
    dl_exception_set(NAT_CC_ID, 0)  
    /* this is a packet (not message) */  
    dl_exception_set_priority(0)  
    dl_exception_send(dl_buf_handle)  
  
ring_empty#:  
    br[start#] /* jump back to the main loop to continue probing */  
  
drop#: /* Drop the packet by freeing its buffer */  
    dl_buf_free(dl_buf_handle,BUF_FREE_LIST0)  
    br[start#] /* go back to the main loop start */
```



# Macros Used To Implement NAT (1)

```
/* NAT_macros.uc - Microassembly macros used with NAT */

/*****/
/* Macro to read source and destination ports from UDP or TCP packet */
/*****/

#macro read_src_and_dst_ports(callsite,hdr_len,IpProt,SrcPort,DstPort)
    .begin
    .reg buf_offset sdram_offset pkt_offset
    /* assume no IP options */
    .if (IpProt == IPT_ICMP)
#if (streq(callsite,'NON_LOCAL_SRC'))
        alu[DstPort,--,b,*l$index0[6],>>16]
        alu[SrcPort,0xFF,and,*l$index0[5],>>24]
#else
        alu[SrcPort,--,b,*l$index0[6],>>16]
        alu[DstPort,0xFF,and,*l$index0[5],>>24]
#endif
    .else
        alu[SrcPort,--,b,*l$index0[5],>>16]
        ld_field_w_clr[DstPort,0011,*l$index0[5]]
    .endif
    .end
#endm
```

## Macros Used To Implement NAT (2)

```
/* **** */
/* Macro to do NAT lookup for packet with local src */
/* **** */

#macro nat_lookup_outgoing(ip_addr_loc,lport,ip_addr_rem,\
                           rport,prot,nport,if_out)
    .begin
    .reg cnt tmp offset entry_w1 tm_offset $timer_bm
    .sig hash_done read_done write_done
    xbuf_alloc[$hash128_w,4,read_write]
    /* hash IP address, port and protocol */
    alu[$hash128_w0,--,b,ip_addr_rem]
    alu[$hash128_w1,--,b,ip_addr_loc]
    alu[entry_w1,rport,or,lport,<<16]
    alu[$hash128_w2,--,b,entry_w1]
    alu[$hash128_w3,--,b,prot]
    hash_128[$hash128_w0,1], sig_done[hash_done]
    alu[cnt,--,b,zero]
    alu[nport,--,~b,zero]
    ctx_arb[hash_done]
    /* compute the hash value mod the number of buckets */
    /* in the hash table */
    alu[offset,$hash128_w0,and,nat_tab_bit_mask]
    /* computer byte offset into NAT table */
    alu[offset,--,b,offset,<<SHIFT_VAL]
    alu[offset,offset,-,16]
```

# Macros Used To Implement NAT (3)

```
/* search the bucket linearly */
search_start#:
    alu[--,HASH_BUCKET_SZ,-,cnt]
    ble[search_done#]
    alu[offset,offset,+,16]
    dram[read,$$entry_w0,f_nat_table,offset,2],
        sig_done[read_done]

    ctx_arb[read_done]
    /* Verify that values in the entry match the */
    /* search keys                                     */
    /* check valid bit */
    br_bclr[$$entry_w0,31,search_start#],defer[3]
    alu[cnt,cnt,+,one]
    alu[tmp,0xFF,and,$$entry_w0,>>16]
    alu[--,tmp,xor,prot] /* check protocol */
    bne[search_start#],defer[3]
    alu[tm_offset--,b,offset,>>4]
    alu[tmp,tm_offset,and,3]
    alu[--,$$entry_w1,xor,entry_w1] /* check ports */
    bne[search_start#],defer[3]
    alu[tmp--,b,tmp,<<3]
    alu[tmp,31,-,tmp]
    alu[--,$$entry_w2,xor,ip_addr_loc] /* check local IP */
    bne[search_start#],defer[3]
    alu[--,tmp,or,zero] /* dummy instruction for */
        /* indirect shift          */
    alu[$timer_bm--,b,one,<<indirect]
    alu[--,$$entry_w3,xor,ip_addr_rem] /* check remote IP */
    bne[search_start#]
```

# Macros Used To Implement NAT (4)

```
        /* at this point, the code has found a match in the NAT */
        /* table, and must update timer for the entry          */
        sram[set,$timer_bm,f_timer,tm_offset],sig_done[write_done]
        ld_field_w_clr[nport,0011,$$entry_w0]
        alu[if_out,--,b,NAT_IFC]
        alu[ip_addr_loc,--,b,NAT_ip]
        ctx_arb[write_done]
search_done#:
        .end
#endm
```

# Macros Used To Implement NAT (5)

```
/* **** */
/* Macro to perform NAT lookup for a packet with a local destination */
/* **** */
#macro nat_lookup_incoming(ip_addr_loc, nport, ip_addr_rem, rport, \
                          prot, lport, if_out)
    .begin
    .reg cnt tmp offset port_tmp tm_offset $timer_bm
    .sig hash_done read_done write_done
    xbuf_alloc[$hash128_w,4,read_write]
    /* hash IP address, port and protocol */
    alu[$hash128_w0,--,b,ip_addr_rem]
    alu[$hash128_w1,rport,or,nport,<<16]
    alu[$hash128_w2,--,b,prot]
    alu[$hash128_w3,--,b,zero]
    hash_128[$hash128_w0,1], sig_done[hash_done]
    alu[cnt,--,b,zero]
    alu[lport,--,~b,zero]
    ctx_arb[hash_done]
    /* compute the hash value mod the number of buckets */
    /* in the hash table */
    alu[offset,$hash128_w0,and,nat_tab_bit_mask]
    /* compute byte offset into NAT table */
    alu[offset,--,b,offset,<<SHIFT_VAL]
    alu[offset,offset,-,16]
endmacro
```

## Macros Used To Implement NAT (6)

```
/* search the bucket linearly */
search_start#:
    alu[--,HASH_BUCKET_SZ,-,cnt]
    ble[search_done#]
    alu[offset,offset,+,16]
    dram[read,$$entry_w0,r_nat_table,offset,2],
        sig_done[read_done]

    ctx_arb[read_done]
    /* Verify that values in the entry match */
    /* the search keys */
    /* check valid bit */
    br_bclr[$$entry_w0,31,search_start#],defer[3]
    alu[cnt,cnt,+,one]
    alu[tmp,0xFF,and,$$entry_w0,>>16]
    alu[--,tmp,xor,prot] /* check protocol */
    bne[search_start#],defer[3]
    alu[tm_offset--,b,offset,>>4]
    alu[port_tmp,0,+16,$$entry_w1]
    alu[--,port_tmp,xor,rport] /* check remote port */
    bne[search_start#],defer[3]
    alu[tmp,tm_offset,and,3]
    alu[port_tmp,0,+16,$$entry_w0]
    /* check NAT (destination) port */
    alu[--,port_tmp,xor,nport]
    bne[search_start#],defer[3]
    alu[tmp,--,b,tmp,<<3]
    alu[tmp,31,-,tmp]
    alu[--,$$entry_w3,xor,ip_addr_rem] /* check remote IP */
    bne[search_start#],defer[2]
```

# Macros Used To Implement NAT (7)

```
        alu[--,tmp,or,zero] /* dummy instruction for */
                               /* indirect shift      */
        alu[$timer_bm,--,b,one,<<indirect]
        /* at this point, the code has found a match in the NAT */
        /* table, and must update timer for the entry          */
        sram[set,$timer_bm,r_timer,tm_offset],sig_done[write_done]
        alu[if_out,one,+,NAT_IFC] /* so that if_out!=NAT_IFC */
        alu[ip_addr_loc,--,b,$$entry_w2]
        alu[lport,--,b,$$entry_w1,>>16]
        ctx_arb[write_done]

search_done#:
        .end
#endm

/*****
/* Macro to read Ethernet and IP headers from DRAM          */
/*****/
#macro eth_iphdr_load(buf_handle, req_sig)
        .begin
        .reg    sdram_offset buf_offset
        /* Read 40 bytes of ETH/IP Header from DRAM */
        /* (DRAM reads are in quadwords -- 8 bytes */
        dl_buf_get_data[sdram_offset, buf_handle]
        dl_meta_get_offset[buf_offset]
        dram[read,$$iphdr0,sdram_offset,buf_offset,5],sig_done[req_sig]
        ctx_arb[req_sig]
        .end
#endm
```

# Macros Used To Implement NAT (8)

```
/* **** */
/* Macro to write modified Ethernet and IP headers from Local */
/* memory back into DRAM */
/* **** */
#macro eth_iphdr_store(sdram_offset, buf_offset, req_sig)
    byte_align_be[--,eth_ipt]
    byte_align_be[$$iphdr3,*1$index0[0]]
    byte_align_be[$$iphdr4,*1$index0[1]]
    byte_align_be[$$iphdr5,*1$index0[2]]
    byte_align_be[$$iphdr6,*1$index0[3]]
    byte_align_be[$$iphdr7,*1$index0[4]]
    byte_align_be[$$iphdr8,*1$index0[5]]
    byte_align_be[$$iphdr9,*1$index0[6]]
    dram[write,$$iphdr0,sdram_offset,buf_offset,5],sig_done[req_sig]
#endm
```



# Macros Used To Implement NAT (9)

```

/*****/
/* Macro to update IP checksum */
/* cksum_new = cksum_old + old_val + ~new_val */
/*****/
#macro cksum_upd(cksum,old_val,new_val)
    .begin
    .reg x not_new_val
    alu[x,--,b,old_val,>>16]
    alu[cksum,cksum,+,x]
    alu[cksum,cksum,+16,old_val]
    alu[not_new_val,--,~b,new_val]
    alu[x,--,b,not_new_val,>>16]
    alu[cksum,cksum,+,x]
    alu[cksum,cksum,+16,not_new_val]
    .end
#endm
/*****/
/* Macro to add carry into checksum */
/* cksum = cksum>>16 + cksum&0xffff */
/* cksum = cksum>>16 + cksum&0xffff */
/*****/
#macro cksum_carry(cksum)
    .begin
    .reg x
    alu[x,--,b,cksum,>>16]
    alu[cksum,x,+16,cksum]
    alu[x,--,b,cksum,>>16]
    alu[cksum,x,+16,cksum]
    .end
#endm

```

# Macros Used To Implement NAT (10)

```

/*****
/* Macro to modify and store Eth frame header, IP packet header      */
/* and UDP, TCP, or ICMP packet header                               */
/*****
#macro modify_and_save_packet_header(if_out,EthDstW0,EthDstW1,IpHlen,\
                                     IpProt,IpSrc,IpDst,SrcPort,DstPort)

    .begin
    .reg tmp cksum buf_offset pkt_offset sdram_offset
    .reg if_ip if_eth_w0 if_eth_w1 oldId oldPorts oldIpSrc oldIpDst
    .sig dram_rd dram_wr1 dram_wr2

    /* Compute sdram offset for the buffer */
    dl_buf_get_data[sdram_offset, dl_buf_handle]

    /* Set the Ethernet header */
    net_if_data_get(if_out,if_ip,if_eth_w0,if_eth_w1)
    alu[$$iphdr0,--,b,EthDstW0]
    alu[$$iphdr1,EthDstW1,or,if_eth_w0,>>16]
    dbl_shf[$$iphdr2,if_eth_w0,if_eth_w1,>>16]

    /* Update the IP header */
    ld_field_w_clr[cksum,0011,*1$index0[2]]
    alu[oldIpSrc,--,b,*1$index0[3]]
    alu[oldIpDst,--,b,*1$index0[4]]
    /* calculate new checksum */
    cksum_upd(cksum,oldIpSrc,IpSrc)
    cksum_upd(cksum,oldIpDst,IpDst)
    cksum_carry(cksum)

```

# Macros Used To Implement NAT (11)

```
/* Save the new checksum and IP addresses */
ld_field[*1$index0[2],0011,cksum]
alu[*1$index0[3],--,b,IpSrc]
alu[*1$index0[4],--,b,IpDst]

/* Update the TCP, UDP, or ICMP header. Note: we assume that */
/* the datagram does not contain IP options */
.if (IpProt == IPT_ICMP)
    /* Update the ICMP header */
    ld_field_w_clr[cksum,0011,*1$index0[5]]
    alu[oldId,--,b,*1$index0[6],>>16]
    .if (IpSrc == NAT_ip)
        cksum_upd(cksum,oldId,SrcPort)
        ld_field[*1$index0[6],1100,SrcPort,<<16]
    .else
        cksum_upd(cksum,oldId,DstPort)
        ld_field[*1$index0[6],1100,DstPort,<<16]
    .endif
    cksum_carry(cksum)
    ld_field[*1$index0[5],0011,cksum]
    dl_meta_get_offset[buf_offset]
    /* Save the modified Ethernet and IP headers */
    eth_iphdr_store(sdram_offset,buf_offset,dram_wrl)
```

# Macros Used To Implement NAT (12)

```
.else
    /* Update the TCP or UDP header */
    dl_meta_get_offset[buf_offset]
    .if (IpProt == IPT_TCP)
        alu[pkt_offset,buf_offset,+,48]
    .else
        alu[pkt_offset,buf_offset,+,40]
    .endif
    /* Read the old UDP or TCP checksum */
    dram[read,$$pkt_hdr0,sdram_offset,pkt_offset,1],
        sig_done[dram_rd]

    alu[oldPorts,--,b,*l$index0[5]]
    alu[*l$index0[5],DstPort,or,SrcPort,<<16]
    /* Save the modified Ethernet and IP headers */
    eth_iphdr_store(sdram_offset,buf_offset,dram_wr1)
    /* Wait for the checksum to be read */
    ctx_arb[dram_rd]
    .if (IpProt == IPT_TCP)
        ld_field_w_clr[cksum,0011,$$pkt_hdr0]
    .else
        alu[cksum,--,b,$$pkt_hdr0,>>16]
        /* If UDP checksum is 0, no update needed */
        alu[--,cksum,xor,zero]
        bne[wait#]
    .endif
    cksum_upd(cksum,oldIpSrc,IpSrc)
    cksum_upd(cksum,oldIpDst,IpDst)
```

# Macros Used To Implement NAT (13)

```
alu[tmp,DstPort,or,SrcPort,<<16]
cksum_upd(cksum,oldPorts,tmp)
cksum_carry(cksum)
.if (IpProt == IPT_TCP)
    alu[tmp,--,b,cksum]
    ld_field[tmp,1100,$$pkt_hdr0]
.else
    alu[tmp,--,b,cksum,<<16]
    ld_field[tmp,0011,$$pkt_hdr0]
.endif
alu[$$pkt_hdr0,--,b,tmp]
alu[$$pkt_hdr1,--,b,$$pkt_hdr1]
dram[write,$$pkt_hdr0,sdram_offset,pkt_offset,1],
                                     sig_done[dram_wr2]
ctx_arb[dram_wr1,dram_wr2],br[done#]
.endif
wait#:
ctx_arb[dram_wr1]
done#:
.end
#endm
```

# Macros Used To Implement NAT (14)

```

/*****
/* Macro to obtain a copy of network interface settings */
*****/
#macro net_if_data_get(ifnum,if_ip,if_eth_w0,if_eth_w1)
    alu[temp,--,b,ifnum,<<3]
    jump[temp, if_0#], targets[if_0#,if_1#]
    /* set network interface parameters */
if_0#:
    immed[if_ip, IF0_IP]
    immed_w1[if_ip, IF0_IP>>16]
    immed[if_eth_w0,IF0_ETH_W0]
    immed_w1[if_eth_w0,IF0_ETH_W0>>16]
    br[end_of_if_table#],defer[2]
    immed[if_eth_w1,IF0_ETH_W1]
    immed_w1[if_eth_w1,IF0_ETH_W1>>16]
    nop /* added for alignment */
if_1#:
    immed[if_ip, IF1_IP]
    immed_w1[if_ip, IF1_IP>>16]
    immed[if_eth_w0,IF1_ETH_W0]
    immed_w1[if_eth_w0,IF1_ETH_W0>>16]
    immed[if_eth_w1,IF1_ETH_W1]
    immed_w1[if_eth_w1,IF1_ETH_W1>>16]
end_of_if_table#:
#endm

```

# Macros Used To Implement NAT (15)

```

/*****
/* Macro to perform ARP table lookup */
*****/
#macro arp_lookup(port, IpDst, EthAddrW0, EthAddrW1)
    .begin
    .reg ip_addr cnt tmp offset $hash48_w0 $hash48_w1
    .reg $entry_w0 $entry_w1 $entry_w2
    .sig hash_done read_done
    .xfer_order $hash48_w0 $hash48_w1
    .xfer_order $entry_w0 $entry_w1 $entry_w2
    .if (port == NAT_IFC)
        alu[ip_addr,--,b,gateway_ip]
    .else
        alu[ip_addr,--,b,IpDst]
    .endif
    /* Hash the IP address */
    alu[$hash48_w0,--,b,ip_addr]
    alu[$hash48_w1,--,b,zero]
    hash_48[$hash48_w0,1], sig_done[hash_done]
    alu[cnt,--,b,zero]
    ctx_arb[hash_done]
    /* Compute the hash value mod the size of the ARP table */
    alu[offset,$hash48_w0,and,arp_tab_bit_mask]
    /* Compute the byte offset into the ARP table */
    alu[offset,--,b,offset,<<4]
    /* Adjust the start of the table */
    alu[offset,offset,-,16]

```

# Macros Used To Implement NAT (16)

```
/* Search the table sequentially */
arp_search_start#:
    alu[--,arp_tab_bit_mask,-,cnt]
    blt[exception#] /* lookup failed */
    alu[offset,offset,+,16]
    alu[offset,offset,and,arp_tab_bit_mask,<<4]
    sram[read,$entry_w0,arp_tab,offset,3],ctx_swap[read_done]
    alu[--,$entry_w0,xor,ip_addr]
    bne[arp_search_start#],defer[1]
    alu[cnt,cnt,+,one]
    br_bclr[$entry_w2,0,arp_search_start#]
arp_search_end#:
    /* Set the word 0 of the Ethernet address */
    alu[EthAddrW0,--,b,$entry_w1]
    /* Set word 1 of the Ethernet address */
    ld_field_w_clr[EthAddrW1,1100,$entry_w2]
    /* Set the output port */
    ld_field_w_clr[if_out,0011,$entry_w2,>>1]
    .end
#endm

/*****
/* Macro to write the current packet on the TX ring */
*****/
#macro write_tx_ring(ring_num,label)
#define_eval RN PACKET_TX_SCR_RING /**/ring_num
    br_inp_state[SCR_RING/**/RN/**/_FULL, full_ring/**/ring_num/**/#]
    scratch[put,$txreq,zero,(RN<<2),1],sig_done[sig_scr_put]
    ctx_arb[sig_scr_put],br[label]
#endm
```



# Core Component Responsibilities

- Initialization (performed once at startup)
- Processing exception packets
  - ARP request with the NAT system as the target
  - ICMP echo request with the NAT system as the destination
  - TCP, UDP, or ICMP Echo, packet for which no NAT table entry exists
  - Note: other packets are dropped
- Processing requests from the user interface
- Cleanup (performed once at shutdown)

# Core Component Implementation

- Divided into three files
- Conceptual purpose
  - Initialization and driver for pseudo device
  - Packet handler
  - Definitions of protocol headers (include file)

# Core Initialization And Pseudo-Device Driver (1)

```
/* NAT_pseudo_dev.c - NAT core comp. & driver (Linux kernel module) */

#include <linux/kernel.h> /* Code runs in the Linux kernel */
#include <linux/module.h> /* The code runs as a kernel module */
#include <linux/fs.h> /* NAT pseudo device is a character device */
#include <asm/uaccess.h> /* Needed for communication with user space */

#include <enpv2_types.h>

#include <ix_rm.h> /* Code uses the IXA Resource Manager */
#undef LINUX /* Prevents the compiler from complaining */
#include <ix_cc.h> /* Code uses the IXA CCI */
#include <ix_cci.h>

#include "NAT_shared_defs.h"
#include "NAT_types.h"
#include "NAT_scratch_rings.h"

#define ME_MASK 0x07 /* system uses microengines 0, 1 and 2 */
#define CONTEXT_MASK 255 /* context mask -- enable all contexts */

#define HASH_MULT_W0 0x12345678 /* hash multiplier -- word 0 */
#define HASH_MULT_W1 0x87654321 /* hash multiplier -- word 1 */
#define HASH_MULT_W2 0x56781234 /* hash multiplier -- word 2 */
#define HASH_MULT_W3 0x43218765 /* hash multiplier -- word 3 */
```

# Core Initialization And Pseudo-Device Driver (2)

```
/* macro to log error message and terminate resource manager */
#define panic(...) { printk("%s: ",NAT_DRIVER_NAME);\
                    printk(__VA_ARGS__);\
                    ix_rm_term();\
                    unregister_chrdev(NAT_major, NAT_DRIVER_NAME);\
                    return(-1); }

/* macro to clear block of kernel memory */
#define bzero(buf,size) ix_ossl_memset(buf,0,size)

/* macro to convert microengine sequence number */
/* into microengine ID */
#define ME_ID(i) ((i%ME_CL_SZ)|((i/ME_CL_SZ)<<4))

/* macro to convert SRAM offset and memory channel into */
/* microengine addressing */
#define ME_SRAM_ADDR(offset,memChan)\
( (memChan)?(offset|(0x20000000<<memChan)):offset )

/* Module parameter -- a UOF file name */
static char * Uof_file;
/* Module parameter -- Linux major device number for NAT psuedo device */
static unsigned int NAT_major = NAT_DEF_MAJOR_NUMBER;

MODULE_AUTHOR("Networking Lab, CS, Purdue University");
MODULE_DESCRIPTION("NAT pseudo-device driver for IXP2XXX");
MODULE_PARM(Uof_file, "s");
MODULE_PARM(NAT_major, "i");
```

# Core Initialization And Pseudo-Device Driver (3)

```
/* Static gateway IP address */
unsigned int gateway_ip= GW_IP;

/* Static network interface configuration */
net_if iface_table[PORTS_NUM]={
    {0xC0A80002 /* 192.168.0.2 */,
     0x01010101,0x01010000 /* 01:01:01:01:01:01 */},
    {0x0A000001 /* 10.0.0.1 */,
     0x02020202,0x02020000 /* 02:02:02:02:02:02 */} };

/* External procedures */

extern ix_error nat_pkt_handler(ix_buffer_handle,ix_uint32,void*);
extern ix_error resolve_arp(unsigned int);

/* Local procedures */

static int patch_microblocks(ix_buffer_free_list_info);
static int create_scr_rings();
static int init_hash();
static ix_error nat_table_timer(void*);
static ix_error exe_init_f(ix_exe_handle,void**);
static ix_error exe_fini_f(ix_exe_handle,void*);
static ix_error cc_init_f(ix_cc_handle,void**);
static ix_error cc_fini_f(ix_cc_handle,void*);
static ix_exe_handle exeHandle;
static ix_cc_handle ccHandle;
static ix_event_handle eveHandle;
static int nat_open(struct inode *, struct file *);
```

# Core Initialization And Pseudo-Device Driver (4)

```
static int nat_release(struct inode *, struct file *);
static int nat_ioctl(struct inode *, struct file *,
                    unsigned int, unsigned long);

/* Verbosity level */
int verb=SILENT;

/* Pointers to various run-time data structures */
nat_entry *f_nat_table, *r_nat_table;
unsigned int *f_index, *r_index;
arp_entry *arp_table;
unsigned char *f_timer, *r_timer;
void *rx_cntr, *tx_cntr;

/* Scratch rings */
ix_hw_ring_handle rxToNatRing, txScrRing[4];

/* List of free buffers */
ix_buffer_free_list_handle hwFreeList = 0;

/* Operations for the NAT pseudo-device */
static struct file_operations nat_fops = {
    ioctl:      nat_ioctl,
    open:       nat_open,
    release:    nat_release
};
```

# Core Initialization And Pseudo-Device Driver (5)

```
static int nat_open(struct inode *inode, struct file *filp)
{
    MOD_INC_USE_COUNT;
    return 0;
}
static int nat_release(struct inode *inode, struct file *filp)
{
    MOD_DEC_USE_COUNT;
    return 0;
}

static int nat_ioctl(struct inode *inode, struct file *fp,
                    unsigned int cmd, unsigned long buf)
{
    switch (cmd) {
        case SILENT:
            verb = SILENT;
            break;
        case VERBOSE:
            verb = VERBOSE;
            break;
        case GET_ARP_TABLE:
            if ((char *)buf != NULL)
                return copy_to_user((char *)buf, arp_table,
                                    ARP_TABLE_SIZE*sizeof(arp_entry));
            break;
    }
}
```

# Core Initialization And Pseudo-Device Driver (6)

```
case GET_NAT_TABLE:
    if ((char *)buf != NULL)
        return copy_to_user((char *)buf,
                            (void*)f_nat_table,
                            NAT_TABLE_SIZE*sizeof(nat_entry));
    break;
case GET_TIMER_TABLE:
    if ((char *)buf != NULL)
        return copy_to_user((char *)buf,
                            (void*)f_timer, 2*NAT_TABLE_SIZE);
    break;
case GET_RX_COUNTER:
    if ((char *)buf != NULL)
        return copy_to_user((char *)buf,rx_cntr,
                            RX_CNTR_SIZE);
    break;
case GET_TX_COUNTER:
    if ((char *)buf != NULL)
        return copy_to_user((char *)buf,tx_cntr,
                            TX_CNTR_SIZE);
    break;
case CLR_RX_COUNTER:
    bzero(rx_cntr,RX_CNTR_SIZE);
    break;
case CLR_TX_COUNTER:
    bzero(tx_cntr,TX_CNTR_SIZE);
    break;
default:
    return INVALID_CMD;
}
```



# Core Initialization And Pseudo-Device Driver (7)

```
        return 0;
    }

int init_module()
{
    ix_error err;
    ix_buffer_free_list_info hwFreeListInfo;
    int i;

    if (Uof_file == NULL) {
        printk("%s: no microcode file specified!\n",
              NAT_DRIVER_NAME);
        return -1;
    }

    /* Register the pseudo-device with Linux */
    if (register_chrdev(NAT_major, NAT_DRIVER_NAME, &nat_fops) < 0) {
        printk("%s: can't get major number %d\n",
              NAT_DRIVER_NAME, NAT_major);
        return -1;
    }

    /* Initialize Intel's Resource Manager */
    printk("%s: Initializing Resource Manager\n", NAT_DRIVER_NAME);
    err=ix_rm_init(0);
    if (err != IX_SUCCESS) {
        printk("Error: ix_rm_init failed\n");
        return -1;
    }
}
```

# Core Initialization And Pseudo-Device Driver (8)

```
/* Register the exception packet handler */
printk("%s: Setting packet receive mode (to callback)\n",
       NAT_DRIVER_NAME);
err = ix_rm_packet_set_receive_mode(NAT_CC_ID,
                                   IX_COMM_ID_MODE_CALLBACK);
if (err != IX_SUCCESS)
    panic("ix_rm_packet_set_receive_mode failed\n");
printk("%s: Registering packet handler\n", NAT_DRIVER_NAME);
err = ix_rm_packet_handler_register(NAT_CC_ID, nat_pkt_handler,
                                   NULL);

if (err != IX_SUCCESS)
    panic("ix_rm_packet_handler_register failed\n");

/* Allocate a free buffer list */
err = ix_rm_hw_buffer_free_list_create(NUM_BUFFERS,
                                       sizeof(ix_hw_buffer_meta), BUF_SIZE,
                                       BUF_SRAM_CHAN, BUF_DRAM_CHAN, &hwFreeList);
if (err != IX_SUCCESS)
    panic("ix_rm_hw_buffer_free_list_create failed\n");
/* Read freelist info (it will be needed later) */
err = ix_rm_buffer_free_list_get_info(hwFreeList,
                                     &hwFreeListInfo);

if (err != IX_SUCCESS)
    panic("ix_rm_hw_buffer_free_list_get_info failed\n");
```

# Core Initialization And Pseudo-Device Driver (9)

```
/* Allocate RX counters (SRAM, channel 0) */
err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 0,
                     RX_CNTR_SIZE, &rx_cntr);
if (err != IX_SUCCESS)
    panic("ix_rm_mem_alloc failed for RX counters\n");
/* Clear RX counters */
bzero(rx_cntr,RX_CNTR_SIZE);

/* Allocate TX counters (SRAM, channel 1) */
err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 1,
                     TX_CNTR_SIZE, &tx_cntr);
if (err != IX_SUCCESS)
    panic("ix_rm_mem_alloc failed for TX counters\n");
/* Clear TX counters */
bzero(tx_cntr,TX_CNTR_SIZE);

/* Allocate the NAT table in DRAM */
err = ix_rm_mem_alloc(IX_MEMORY_TYPE_DRAM, 0,
                     2*NAT_TABLE_SIZE*sizeof(nat_entry),
                     (void**)&f_nat_table);
if (err != IX_SUCCESS)
    panic("ix_rm_mem_alloc failed for NAT table\n");
/* Clear the NAT table */
bzero((void*)f_nat_table,2*NAT_TABLE_SIZE*sizeof(nat_entry));
/* Set the base address for reverse NAT table */
r_nat_table=f_nat_table+NAT_TABLE_SIZE;
```

# Core Initialization And Pseudo-Device Driver (10)

```
/* Allocate the NAT index table in DRAM */
err = ix_rm_mem_alloc(IX_MEMORY_TYPE_DRAM, 0,
                    2*NAT_TABLE_SIZE*sizeof(unsigned int),
                    (void**)&f_index);
if (err != IX_SUCCESS)
    panic("ix_rm_mem_alloc failed for NAT index table\n");
/* Clear the NAT index table */
bzero((void*)f_index,2*NAT_TABLE_SIZE*sizeof(unsigned int));
/* Set the base for the reverse NAT index table */
r_index=f_index+NAT_TABLE_SIZE;

/* Allocate the timer table in SRAM */
err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 0, NAT_TABLE_SIZE,
                    (void**)&f_timer);
if (err != IX_SUCCESS)
    panic("ix_rm_mem_alloc failed for timer table\n");
/* Clear the timer table */
bzero((void*)f_timer,2*NAT_TABLE_SIZE);
/* Set the base address for the reverse timer table */
r_timer=f_timer+NAT_TABLE_SIZE;

/* Allocate the ARP table in DRAM */
err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 1,
                    ARP_TABLE_SIZE*sizeof(arp_entry), (void**)&arp_table);
if (err != IX_SUCCESS)
    panic("ix_rm_mem_alloc failed for ARP table\n");
/* Clear the ARP table */
bzero((void*)arp_table,ARP_TABLE_SIZE*sizeof(arp_entry));
```

# Core Initialization And Pseudo-Device Driver (11)

```
/* Reset the microengines */
printk("%s: Resetting all microengines\n", NAT_DRIVER_NAME);
ix_rm_ueng_reset_all();

/* Get the microcode from the UOF file */
printk("%s: Setting ucode\n", NAT_DRIVER_NAME);
err = ix_rm_ueng_set_ucode(Uof_file);
if (err != IX_SUCCESS)
    panic("ix_rm_ueng_set_ucode failed\n");

/* Patch the microcode symbols before actually */
/* loading microcode. */
if (patch_microblocks(hwFreeListInfo) < 0)
    return(-1);

/* Create scratch rings */
if (create_scr_rings() < 0)
    return(-1);

/* Load microcode into microengines */
printk("%s: Loading ucode\n", NAT_DRIVER_NAME);
err = ix_rm_ueng_load();
if (err != IX_SUCCESS)
    panic("ix_rm_ueng_load failed\n");

/* Initialize hash unit */
if (init_hash() < 0)
    return(-1);
```

# Core Initialization And Pseudo-Device Driver (12)

```
/* Start the assigned microengines */
for (i=0;i<ME_NUM;i++) {
    if ((ME_MASK>>i)&0x1) {
        printk("%s: Starting ME%i\n",NAT_DRIVER_NAME, i);
        err = ix_rm_ueng_start(ME_ID(i),CONTEXT_MASK);
        if (err != IX_SUCCESS)
            panic("ix_rm_ueng_start failed for ME %i\n",i);
    }
}

/* Resolve an ARP entry for the gateway */
if (resolve_arp(GW_IP) != IX_SUCCESS)
    panic("can't resolve ARP entry for the gateway\n");

/* Create an execution engine (i.e., a kernel thread) for the */
/* NAT timer aging procedure */
err=ix_cci_init(); /* Initialize Intel's CCI */
if (err != IX_SUCCESS)
    panic("ix_cci_init failed\n");
printk("%s: Creating timer thread\n",NAT_DRIVER_NAME);
err=ix_cci_exe_run(NULL,exe_init_f,exe_fini_f,"NAT timer",
    &exeHandle);
if (err != IX_SUCCESS) {
    ix_cci_fini();
    panic("ix_cci_exe_run failed\n");
}
return 0;
}
```

# Core Initialization And Pseudo-Device Driver (13)

```
/* Cleanup */
void cleanup_module()
{
    ix_error err;
    int i;

    /* Stop each of the assigned microengines */
    for (i=0;i<ME_NUM;i++) {
        if ((ME_MASK>>i)&0x1) {
            printk("%s: Stopping ME%i\n",NAT_DRIVER_NAME,i);
            err = ix_rm_ueng_stop(ME_ID(i));
            if (err != IX_SUCCESS)
                printk(
                    "%s: ix_rm_ueng_stop failed for ME %i\n",
                    NAT_DRIVER_NAME,i);
        }
    }

    /* Unregister the packet handler */
    ix_rm_packet_handler_unregister(NAT_CC_ID);

    /* Terminate the timer thread */
    printk("%s: Stopping timer thread\n",NAT_DRIVER_NAME);
    ix_cci_exe_shutdown(exeHandle);
    ix_cci_fini();
}
```

# Core Initialization And Pseudo-Device Driver (14)

```
    /* Terminate the Resource Manager */
    ix_rm_term();

    /* unregister pseudo-device */
    unregister_chrdev(NAT_major, NAT_DRIVER_NAME);
}

/* NAT table management: periodically go through the timer table */
/* and update (age) each of the timers */
ix_error nat_table_timer(void* dummy)
{
    int i;
    for (i=0;i<2*NAT_TABLE_SIZE;i++)
        if (f_nat_table[i].valid)
            f_timer[i]=f_timer[i]>>1;
    return(IX_SUCCESS);
}

ix_error exe_init_f(ix_exe_handle exeHandle,void** ppContext)
{
    ix_cc_init_context dummy;
    return ix_cci_cc_create(exeHandle,cc_init_f,cc_fini_f,
        (void*)&dummy,&ccHandle);
}

ix_error exe_fini_f(ix_exe_handle exeHandle,void* pContext)
{
    return ix_cci_cc_destroy(ccHandle);
}
```



# Core Initialization And Pseudo-Device Driver (15)

```
ix_error cc_init_f(ix_cc_handle ccHandle,void** ppContext)
{
    /* Age each timer every AGING_INTERVAL */
    return ix_cci_cc_add_event_handler(ccHandle,AGING_INTERVAL,
        nat_table_timer,IX_EVENT_TYPE_PERIODIC,1,&eveHandle);
}

ix_error cc_fini_f(ix_cc_handle ccHandle,void* pContext)
{
    return ix_cci_cc_remove_event_handler(ccHandle,eveHandle);
}
```

# Core Initialization And Pseudo-Device Driver (16)

```

/*****
/* Patch the microcode symbols before actually loading microcode. */
/*
/* The imported variables that must be patched are:
/* BUF_FREE_LIST0    -- get from freelist allocation,
/*                   used by all microblocks
/* BUF_SRAM_BASE    -- get from freelist allocation,
/*                   used by all microblocks
/* DL_REL_BASE      -- compute from freelist allocation
/*                   parameters, used by all microblocks
/* FREE_LIST_ID     -- get from freelist allocation (RX ublock)
/* PACKET_COUNTERS_SRAM_BASE -- get from memory allocation for
/*                   RX counters, used by RX ublock
/* PACKET_TX_COUNTER_BASE -- get from memory allocation for
/*                   TX counters, used by TX ublock
/* ARP_TABLE_BASE   -- get from memory allocation,
/*                   used by NAT microblock only
/* NAT_TABLE_BASE   -- get from memory allocation,
/*                   used by NAT microblock only
/* TIMER_TABLE_BASE -- get from memory allocation,
/*                   used by NAT microblock only
/* GATEWAY_IP_ADDR  -- gateway IP address, hardcoded,
/*                   used by NAT microblock only
/* IFO_IP, IF1_IP,
/* IFO_ETH_W0, IFO_ETH_W1,
/* IF1_ETH_W0, IF1_ETH_W1 -- interface settings from interface
/*                   table, used by NAT microblock only
*****/

```

# Core Initialization And Pseudo-Device Driver (17)

```
int patch_microblocks(ix_buffer_free_list_info  hwFreeListInfo)
{
    ix_error err;
    ix_imported_symbol importSymbols[15];
    ix_uint32      memChan;
    ix_uint32      offset;

    /* Set common symbols */
    importSymbols[0].m_Name="BUF_FREE_LIST0";
    importSymbols[0].m_Value = hwFreeListInfo.m_FreeListInfo;

    importSymbols[1].m_Name="BUF_SRAM_BASE";
    err = ix_rm_get_phys_offset( hwFreeListInfo.m_pMetaBaseAddress,
                                NULL,&memChan,&offset,NULL);
    if (err != IX_SUCCESS)
        panic("ix_rm get_phys_offset failed for %s\n",
              importSymbols[1].m_Name);
    importSymbols[1].m_Value = ME_SRAM_ADDR(offset,memChan);

    importSymbols[2].m_Name = "DL_REL_BASE";
    err = ix_rm_get_phys_offset(hwFreeListInfo.m_pDataBaseAddress,
                                NULL,&memChan,&offset,NULL);
    if (err != IX_SUCCESS)
        panic("ix_rm get_phys_offset failed for %s\n",
              importSymbols[2].m_Name);
    importSymbols[2].m_Value = offset -
        ((importSymbols[1].m_Value*hwFreeListInfo.m_DataElementSize)/
         hwFreeListInfo.m_MetaElementSize);
}
```

# Core Initialization And Pseudo-Device Driver (18)

```
/* Set RX specific symbols */
importSymbols[3].m_Name="PACKET_COUNTERS_SRAM_BASE";
err = ix_rm_get_phys_offset(rx_cntr,NULL,&memChan,&offset,NULL);
if (err != IX_SUCCESS)
    panic("ix_rm_get_phys_offset failed for %s\n",
        importSymbols[3].m_Name);
importSymbols[3].m_Value = ME_SRAM_ADDR(offset,memChan);

importSymbols[4].m_Name="FREE_LIST_ID";
importSymbols[4].m_Value = hwFreeListInfo.m_FreeListInfo1;

/* Patch ME 0x00 -- RX microblock */
err = ix_rm_ueng_patch_symbols(0x00,5,importSymbols);
if (err != IX_SUCCESS)
    panic("ix_rm_ueng_patch_symbols failed for RX microblock\n");

/* Set NAT specific symbols */
importSymbols[3].m_Name="NAT_TABLE_BASE";
err = ix_rm_get_phys_offset((void*)f_nat_table,
    NULL,&memChan,&offset,NULL);
if (err != IX_SUCCESS)
    panic("ix_rm_get_phys_offset failed for %s\n",
        importSymbols[3].m_Name);
importSymbols[3].m_Value = offset;
importSymbols[4].m_Name="ARP_TABLE_BASE";
err = ix_rm_get_phys_offset((void*)arp_table,
    NULL,&memChan,&offset,NULL);
```

# Core Initialization And Pseudo-Device Driver (19)

```
if (err != IX_SUCCESS)
    panic("ix_rm_get_phys_offset failed for %s\n",
        importSymbols[3].m_Name);
importSymbols[4].m_Value = ME_SRAM_ADDR(offset,memChan);
importSymbols[5].m_Name="GATEWAY_IP_ADDR";
importSymbols[5].m_Value=gateway_ip;
importSymbols[6].m_Name="IF0_IP";
importSymbols[6].m_Value=iface_table[0].ip_addr;
importSymbols[7].m_Name="IF0_ETH_W0";
importSymbols[7].m_Value=iface_table[0].eth_w0;
importSymbols[8].m_Name="IF0_ETH_W1";
importSymbols[8].m_Value=iface_table[0].eth_w1;
importSymbols[9].m_Name="IF1_IP";
importSymbols[9].m_Value=iface_table[1].ip_addr;
importSymbols[10].m_Name="IF1_ETH_W0";
importSymbols[10].m_Value=iface_table[1].eth_w0;
importSymbols[11].m_Name="IF1_ETH_W1";
importSymbols[11].m_Value=iface_table[1].eth_w1;
importSymbols[12].m_Name="TIMER_TABLE_BASE";
err = ix_rm_get_phys_offset((void*)f_timer,
    NULL,&memChan,&offset,NULL);
if (err != IX_SUCCESS)
    panic("ix_rm_get_phys_offset failed for %s\n",
        importSymbols[3].m_Name);
importSymbols[12].m_Value = ME_SRAM_ADDR(offset,memChan);
```

# Core Initialization And Pseudo-Device Driver (20)

```
/* Patch ME 0x01 -- NAT microblock */
err = ix_rm_ueng_patch_symbols(0x01,13,importSymbols);
if (err != IX_SUCCESS)
    panic("ix_rm_ueng_patch_symbols failed for NAT microblock\n");

/* Set counter base for TX */
importSymbols[3].m_Name="PACKET_TX_COUNTER_BASE";
err = ix_rm_get_phys_offset((void*)tx_cntr,
                           NULL,&memChan,&offset,NULL);
if (err != IX_SUCCESS)
    panic("ix_rm_get_phys_offset failed for %s\n",
          importSymbols[3].m_Name);
importSymbols[3].m_Value = ME_SRAM_ADDR(offset,memChan);

/* Patch ME 0x02 -- TX microblock */
err = ix_rm_ueng_patch_symbols(0x02,4,importSymbols);
if (err != IX_SUCCESS)
    panic("ix_rm_ueng_patch_symbols failed for TX microblock\n");
return(1);
}
```

# Core Initialization And Pseudo-Device Driver (21)

```
/* Function to create RX and TX scratch memory rings */
int create_scr_rings()
{
    ix_error err;
    err = ix_rm_hw_scratch_ring_create(0,
                                       (PKT_RX_TO_NAT_SCR_RING_SIZE>>9),
                                       PKT_RX_TO_NAT_SCR_RING, &rxToNatRing);
    if (err != IX_SUCCESS)
        panic("ix_rm_hw_scratch_ring_create failed for Rx->Nat ring\n");

    err = ix_rm_hw_scratch_ring_create(0,
                                       (PACKET_TX_SCR_RING_0_SIZE>>9),
                                       PACKET_TX_SCR_RING_0, &txScrRing[0]);
    if (err != IX_SUCCESS)
        panic("ix_rm_hw_scratch_ring_create failed for TX 0 ring\n");
    err = ix_rm_hw_scratch_ring_create(0,
                                       (PACKET_TX_SCR_RING_1_SIZE>>9),
                                       PACKET_TX_SCR_RING_1, &txScrRing[1]);
    if (err != IX_SUCCESS)
        panic("ix_rm_hw_scratch_ring_create failed for TX 1 ring\n");
    err = ix_rm_hw_scratch_ring_create(0,
                                       (PACKET_TX_SCR_RING_2_SIZE>>9),
                                       PACKET_TX_SCR_RING_2, &txScrRing[2]);
    if (err != IX_SUCCESS)
        panic("ix_rm_hw_scratch_ring_create failed for TX 2 ring\n");
    err = ix_rm_hw_scratch_ring_create(0,
                                       (PACKET_TX_SCR_RING_3_SIZE>>9),
                                       PACKET_TX_SCR_RING_3, &txScrRing[3]);
}
```

# Core Initialization And Pseudo-Device Driver (22)

```
    if (err != IX_SUCCESS)
        panic("ix_rm_hw_scratch_ring_create failed for TX 3 ring\n");
    return(1);
}
```



# Core Initialization And Pseudo-Device Driver (23)

```
/* Function to initialize the 128-bit and 48-bit hash multipliers */
int init_hash()
{
    ix_error err;
    ix_hash_multiplier_128 hash128m;
    ix_hash_multiplier_48 hash48m;
    hash128m.m_LW0=HASH_MULT_W0;
    hash128m.m_LW1=HASH_MULT_W1;
    hash128m.m_LW2=HASH_MULT_W2;
    hash128m.m_LW3=HASH_MULT_W3;
    printk("%s: Setting hash 128 multiplier to 0x%08X%08X%08X%08X\n",
        NAT_DRIVER_NAME,
        (unsigned int)hash128m.m_LW3, (unsigned int)hash128m.m_LW2,
        (unsigned int)hash128m.m_LW1, (unsigned int)hash128m.m_LW0);
    err=ix_rm_hash_128_multiplier_set(&hash128m);
    if (err != IX_SUCCESS)
        panic("ix_rm_hash_128_multiplier_set failed\n");
    if (err != IX_SUCCESS)
        panic("ix_rm_hash_64_multiplier_set failed\n");
    hash48m.m_LW0=HASH_MULT_W0;
    hash48m.m_LW1=HASH_MULT_W1;
    printk("%s: Setting hash 48 multiplier to 0x%08X%08X\n",
        NAT_DRIVER_NAME,
        (unsigned int)hash48m.m_LW1, (unsigned int)hash48m.m_LW0);
    err=ix_rm_hash_48_multiplier_set(&hash48m);
    if (err != IX_SUCCESS)
        panic("ix_rm_hash_48_multiplier_set failed\n");
    return(1);
}
```

# Packet Formats Used By The Core (1)

```
/* NAT_net.h - protocol declarations used by the core component */

/* Ethernet packet header */
typedef struct eth_s {
    unsigned char e_dst[6];
    unsigned char e_src[6];
    unsigned short e_type;
    unsigned short data[1];
} eth;

/* ARP packet header      */
typedef struct arp_s {
    unsigned short ar_hrd;
    unsigned short ar_pro;
    unsigned char ar_hln;
    unsigned char ar_pln;
    unsigned short ar_op;
    unsigned char ar_sha[6];
    /* declared as two shorts for alignment */
    unsigned short ar_spa1;
    unsigned short ar_spa2;
    unsigned char ar_tha[6];
    unsigned int ar_tpa;
} arp;
```

## Packet Formats Used By The Core (2)

```
/* IP packet header      */
typedef struct ip_s {
    unsigned char ip_v : 4;
    unsigned char ip_hl : 4;
    unsigned char ip_tos;
    unsigned short ip_len;
    unsigned short ip_id;
    unsigned short ip_frag;
    unsigned char ip_ttl;
    unsigned char ip_p;
    unsigned short ip_sum;
    unsigned int ip_src;
    unsigned int ip_dst;
    unsigned int data[1];
} ip;

/* ICMP packet header    */
typedef struct icmp_s {
    unsigned char icmp_type;
    unsigned char icmp_code;
    unsigned short icmp_cksum;
    unsigned short icmp_id;
    unsigned short icmp_seq;
    unsigned int data[1];
} icmp;
```

## Packet Formats Used By The Core (3)

```
/* TCP packet header      */
typedef struct tcp_s {
    unsigned short tcp_sport;
    unsigned short tcp_dport;
    unsigned int tcp_seq;
    unsigned int tcp_ack;
    unsigned char tcp_offset;
    unsigned char tcp_code;
    unsigned short tcp_window;
    unsigned short tcp_cksum;
    unsigned short tcp_urgptr;
    unsigned int data[1];
} tcp;

/* UDP packet header      */
typedef struct udp_s {
    unsigned short udp_sport;
    unsigned short udp_dport;
    unsigned short udp_len;
    unsigned short udp_cksum;
} udp;

/* Transmit request structure */
typedef struct tx_req_s {
    unsigned int valid : 1;
    unsigned int reserved : 3;
    unsigned int port : 4;
    unsigned int buff_handle : 24;
} ix_tx_req;
```

# Core Component Packet Handler (1)

```
/* NAT_pkt_handler.c - packet handler and table management functions */

#include <enpv2_types.h>
#include <ix_rm.h>          /* Code uses the IXA Resource Manager      */
#define LINUX              /* Prevents the compiler from complaining */
#include <ix_cc.h>         /* Code uses the IXA CCI                  */
#include <ix_cci.h>
#include "NAT_shared_defs.h"
#include "NAT_types.h"
#include "NAT_net.h"

/* macro to drop a packet and quit packet handler */
#define drop(arg_hBuffer) { ix_rm_buffer_free(arg_hBuffer);\
                           return IX_SUCCESS; }

/* Verbosity level */
extern int verb;

/* Static gateway IP address */
extern unsigned int gateway_ip;

/* Static network interface configuration */
extern net_if iface_table[];

/* Global NAT port */
static unsigned short global_nport=0;

/* Scratch rings */
extern ix_hw_ring_handle rxToNatRing, txScrRing[];
```

## Core Component Packet Handler (2)

```
/* List of free buffers */
extern ix_buffer_free_list_handle hwFreeList;

/* Pointers to various run-time data structures */
extern nat_entry *f_nat_table, *r_nat_table;
extern unsigned int *f_index, *r_index;
extern arp_entry *arp_table;
extern unsigned char *f_timer, *r_timer;

/* global procedures */
ix_error nat_pkt_handler(ix_buffer_handle, ix_uint32, void*);
ix_error resolve_arp(unsigned int);

/* local procedures */
static void process_arp_req(arp*, ix_hw_buffer_meta*,
                           ix_buffer_handle, eth*);
static void process_arp_rep(arp*, ix_hw_buffer_meta*);
static void send_icmp_echo_rep(ip*, icmp*, ix_hw_buffer_meta*,
                               ix_buffer_handle, eth*);

static int process_icmp(icmp*, nat_entry*);
static int process_udp(udp*, nat_entry*);
static int process_tcp(tcp*, nat_entry*);
static char* find_arp_entry(unsigned int);
static int add_arp_entry(arp_entry*);
static int add_nat_entry(nat_entry*);
static void add_r_nat_entry(unsigned int);
static void del_nat_entry(unsigned int);
```

# Core Component Packet Handler (3)

```
static int  set_new_nport(nat_entry*);
static void send_pkt(void*, unsigned int, eth*,
                    unsigned char *, unsigned short);

/* Packet handler called when an exception packet arrives */
ix_error nat_pkt_handler(
    ix_buffer_handle arg_hBuffer,
    ix_uint32 arg_UserData,      /* exception code */
    void* arg_pContext
)
{
    ix_hw_buffer_meta *meta_data;
    void *buf;
    int cksum;
    eth *eth_pkt;
    arp *arp_pkt;
    arp_entry ae;
    ip* ip_pkt;
    icmp* icmp_pkt;
    tcp* tcp_pkt;
    udp* udp_pkt;
    nat_entry ne;
    char * gw_eth;
    ix_rm_buffer_get_data(arg_hBuffer, &buf);
    ix_rm_buffer_get_meta(arg_hBuffer, (void **)&meta_data);
    eth_pkt=(eth*)((int)buf+(int)(meta_data->m_Offset));
```

# Core Component Packet Handler (4)

```
if (eth_pkt->e_type == ETH_ARP) { /* got ARP */
    if (verb == VERBOSE)
        printk("Got ARP packet\n");
    arp_pkt=(arp*)(eth_pkt->data);
    if ( arp_pkt->ar_op == ARP_REQ &&
        arp_pkt->ar_tpa ==
            iface_table[meta_data->m_InputPort].ip_addr) {
        /* Process ARP request */
        process_arp_req(arp_pkt,meta_data,
            arg_hBuffer,eth_pkt);
        return IX_SUCCESS;
    } else
    if ( arp_pkt->ar_op == ARP_REP &&
        arp_pkt->ar_tpa ==
            iface_table[meta_data->m_InputPort].ip_addr) {
        /* Process an ARP reply */
        process_arp_rep(arp_pkt,meta_data);
        return IX_SUCCESS;
    } else drop(arg_hBuffer);
}
if (eth_pkt->e_type != ETH_IP)
    drop(arg_hBuffer);
/* got IP packet */
ip_pkt=(ip*)(eth_pkt->data);
```



# Core Component Packet Handler (5)

```
if (verb == VERBOSE) {
    printk("Got IP packet\n");
    printk("\tIP src = %i.%i.%i.%i\n",IP2B(ip_pkt->ip_src));
    printk("\tIP dst = %i.%i.%i.%i\n",IP2B(ip_pkt->ip_dst));
    printk("\tprotocol: %i\n",ip_pkt->ip_p);
    printk("\tingress port = %i\n", meta_data->m_InputPort);
}
/* For simplicity, the code drops a datagram that */
/* has IP options */
if ( ip_pkt->ip_hl > 5)
    drop(arg_hBuffer);
if ( ip_pkt->ip_dst ==
    iface_table[meta_data->m_InputPort].ip_addr) {
    /* The packet is destined to the NAT box itself */
    if (ip_pkt->ip_p == IPT_ICMP) { /* got ICMP */
        icmp_pkt=(icmp*)(ip_pkt->data);
        if (icmp_pkt->icmp_type == ICMP_ECHO_REQ) {
            /* received ping */
            if (verb == VERBOSE)
                printk("Got ping for us\n");
            /* Send an echo reply */
            send_icmp_echo_rep(ip_pkt,icmp_pkt,
                               meta_data,
                               arg_hBuffer,eth_pkt);
            return IX_SUCCESS;
        }
    }
}
}
```

# Core Component Packet Handler (6)

```
if ( /* Packet not from the gateway */
    meta_data->m_InputPort != NAT_IFC &&
    /* An entry is present in ARP table for the gateway */
    (gw_eth = find_arp_entry(GW_IP)) != NULL ) {
    /* The packet is an exception because the NAT lookup */
    /* failed, so add a new entry to the NAT table      */
    ne.valid=0; /* will be set to 1 later */
    ne.prot=ip_pkt->ip_p;
    ne.ip_addr_loc=ip_pkt->ip_src;
    ne.ip_addr_rem=ip_pkt->ip_dst;
    switch (ip_pkt->ip_p) {
        case IPT_ICMP:
            /* packet is ICMP */
            icmp_pkt=(icmp*)(ip_pkt->data);
            if (process_icmp(icmp_pkt,&ne) < 0)
                drop(arg_hBuffer);
            break;
        case IPT_TCP:
            /* received TCP */
            tcp_pkt=(tcp*)(ip_pkt->data);
            if (process_tcp(tcp_pkt,&ne) < 0)
                drop(arg_hBuffer);
            break;
        case IPT_UDP:
            /* received UDP */
            udp_pkt=(udp*)(ip_pkt->data);
            if (process_udp(udp_pkt,&ne) < 0)
                drop(arg_hBuffer);
            break;
    }
```

# Core Component Packet Handler (7)

```
        default: drop(arg_hBuffer);
    }
    /* Create an ARP entry for this packet in case a reply */
    /* comes later */
    ae.ip_addr=ip_pkt->ip_src;
    ae.eth_w0=*(int*)eth_pkt->e_src;
    ae.eth_w1=*(short*)&eth_pkt->e_src[4];
    ae.ifnum = meta_data->m_InputPort;
    ae.valid = 1;
    /* Update the IP checksum */
    cksum = ip_pkt->ip_sum+(ip_pkt->ip_src>>16)
        + (ip_pkt->ip_src&0xFFFF)
        + ((~iface_table[NAT_IFC].ip_addr)>>16)
        + ((~iface_table[NAT_IFC].ip_addr)&0xFFFF);
    cksum=(cksum&0xFFFF)+(cksum>>16);
    cksum=(cksum&0xFFFF)+(cksum>>16);
    /* Update the IP packet */
    ip_pkt->ip_src = iface_table[NAT_IFC].ip_addr;
    ip_pkt->ip_sum=cksum&0xFFFF;
    /* Transmit the packet */
    send_pkt((void*)arg_hBuffer,NAT_IFC,eth_pkt,
            gw_eth,ETH_IP);
    /* Add the ARP entry that was created above */
    add_arp_entry(&ae);
    return IX_SUCCESS;
}
/* Drop the packet */
drop(arg_hBuffer);
}
```

## Core Component Packet Handler (8)

```
/* Function to pass packet to TX microblock */
static void send_pkt(void* buf, unsigned int ifnum, eth* eth_pkt,
    unsigned char * eth_addr, unsigned short eth_type)
{
    ix_tx_req txreq;
    ix_uint32 txreq_size;
    /* set ethernet addresses */
    *(int*)eth_pkt->e_dst=*(int*)eth_addr;
    *(short*)((int*)eth_pkt->e_dst+1)=*(short*)((int*)eth_addr+1);
    *(int*)eth_pkt->e_src=iface_table[ifnum].eth_w0;
    *(short*)((int*)eth_pkt->e_src+1)=(iface_table[ifnum].eth_w1>>16);
    eth_pkt->e_type = eth_type;
    /* prepare Tx request */
    txreq.valid = 1;      /* valid request */
    txreq.reserved=0;    /* reserved -- set to zero */
    txreq.port = ifnum;  /* outgoing interface */
    txreq.buff_handle = (unsigned int)buf; /* buffer handle */
    txreq_size=1;
    /* put Tx request on appropriate Tx scratch ring */
    ix_rm_hw_ring_put(txScrRing[txreq.port], &txreq_size,
        (ix_uint32 *)&txreq);
}
```

# Core Component Packet Handler (9)

```
/* ARP lookup function */
char * find_arp_entry(unsigned int ipaddr)
{
    ix_hash_48 hash48v;
    int i,j;
    hash48v.m_LW0 = ipaddr;
    hash48v.m_LW1 = 0;
    ix_rm_hash_48_hash(&hash48v);
    j = hash48v.m_LW0&ARP_TABLE_BIT_MASK;
    for (i=0;i<ARP_TABLE_SIZE;i++) {
        if (ipaddr == arp_table[j].ip_addr && arp_table[j].valid)
            return((char*)&(arp_table[j].eth_w0));
        j=(j+1)&ARP_TABLE_BIT_MASK;
    }
    return NULL;
}

/* Function to resolve an ARP entry for given IP address (used */
/* to obtain an ARP entry for the gateway) */
ix_error resolve_arp(unsigned int ipaddr)
{
    char eth_bcast[]={0xff,0xff,0xff,0xff,0xff,0xff};
    void * buf;
    eth *eth_pkt;
    arp *arp_pkt;
    ix_buffer_handle hBuffer;
    ix_hw_buffer_meta *meta_data;
    int i;
```

# Core Component Packet Handler (10)

```
for (i=0;i<GW_MAC_RES_ATTEMPTS;i++) {
    ix_rm_buffer_alloc(hwFreeList,&hBuffer);
    ix_rm_buffer_get_data(hBuffer,(void*)&buf);
    ix_rm_buffer_get_meta(hBuffer,(void **)&meta_data);
    meta_data->m_Offset=0;
    meta_data->m_BufferSize=60;
    meta_data->m_PacketSize=60;
    eth_pkt=(eth*)((int)buf+(int)meta_data->m_Offset);
    arp_pkt=(arp*)(eth_pkt->data);
    arp_pkt->ar_hrd = 1; /* Ethernet */
    arp_pkt->ar_hln = 6;
    arp_pkt->ar_pro = ETH_IP; /* IPv4 */
    arp_pkt->ar_pln = 4;
    arp_pkt->ar_op = ARP_REQ;
    *(int*)arp_pkt->ar_tha=0;
    *(short*)((int*)arp_pkt->ar_tha+1)=0;
    arp_pkt->ar_tpa=GW_IP;
    *(int*)arp_pkt->ar_sha=iface_table[NAT_IFC].eth_w0;
    *(short*)((int*)arp_pkt->ar_sha+1)=
        (iface_table[NAT_IFC].eth_w1>>16);
    arp_pkt->ar_spa1=
        (short)(iface_table[NAT_IFC].ip_addr>>16);
    arp_pkt->ar_spa2=
        (short)(iface_table[NAT_IFC].ip_addr&0xFFFF);
    send_pkt((void*)hBuffer, NAT_IFC, eth_pkt,
            eth_bcast, ETH_ARP);
    printk("%s: Resolving gateway MAC address...\n",
            NAT_DRIVER_NAME);
}
```

# Core Component Packet Handler (11)

```
        ix_oss1_sleep(500);
        if (find_arp_entry(GW_IP) != NULL)
            return IX_SUCCESS;
    }
    return(-1);
}

/* Function to process an ARP request */
void process_arp_req(arp* arp_pkt, ix_hw_buffer_meta* meta_data,
    ix_buffer_handle arg_hBuffer, eth *eth_pkt)
{
    arp_entry ae;
    ae.ip_addr = (arp_pkt->ar_spa1<<16) | arp_pkt->ar_spa2;
    ae.eth_w0 = *(int*)arp_pkt->ar_sha;
    ae.eth_w1 = *(short*)((int*)arp_pkt->ar_sha+1);
    ae.ifnum = meta_data->m_InputPort;
    ae.valid = 1;
    arp_pkt->ar_op = ARP_REP;
    *(int*)arp_pkt->ar_tha=*(int*)arp_pkt->ar_sha;
    *(short*)(arp_pkt->ar_tha+4)=*(short*)(arp_pkt->ar_sha+4);
    arp_pkt->ar_tpa=(arp_pkt->ar_spa1<<16) | arp_pkt->ar_spa2;
    *(int*)arp_pkt->ar_sha=iface_table[meta_data->m_InputPort].eth_w0;
    *(short*)(arp_pkt->ar_sha+4)=
        iface_table[meta_data->m_InputPort].eth_w1>>16;
    arp_pkt->ar_spa1=iface_table[meta_data->m_InputPort].ip_addr>>16;
    arp_pkt->ar_spa2=
        iface_table[meta_data->m_InputPort].ip_addr&0xFFFF;
```

# Core Component Packet Handler (12)

```
send_pkt((void*)arg_hBuffer, meta_data->m_InputPort,
        eth_pkt, eth_pkt->e_src, ETH_ARP);
if (verb == VERBOSE)
    printk("Sent ARP reply\n");
/* Also add an entry into arp table */
if ( !add_arp_entry(&ae) )
    printk("%s: ARP table full!", NAT_DRIVER_NAME);
}

/* Function to process an ARP reply */
void process_arp_rep(arp* arp_pkt, ix_hw_buffer_meta* meta_data)
{
    arp_entry ae;
    ae.ip_addr = (arp_pkt->ar_spa1<<16) | arp_pkt->ar_spa2;
    ae.eth_w0 = *(int*)arp_pkt->ar_sha;
    ae.eth_w1 = *(short*)(arp_pkt->ar_sha+4);
    ae.ifnum = meta_data->m_InputPort;
    ae.valid = 1;
    if ( !add_arp_entry(&ae) )
        printk("%s: ARP table full!", NAT_DRIVER_NAME);
}
```



# Core Component Packet Handler (13)

```
/* Function to insert an entry into the ARP table. */
/* Note: because our code uses a simplified ARP table in which entries */
/* do not expire, there is no need to check for duplicate entries. */
int add_arp_entry(arp_entry *ae)
{
    ix_hash_48 hash48v;
    int i,j;
    hash48v.m_LW0 = ae->ip_addr;
    hash48v.m_LW1 = 0;
    ix_rm_hash_48_hash(&hash48v);
    j = hash48v.m_LW0&ARP_TABLE_BIT_MASK;
    for (i=0;i<ARP_TABLE_SIZE;i++) {
        if (ae->ip_addr == arp_table[j].ip_addr &&
            arp_table[j].valid )
            return(1);
        if ( !arp_table[j].valid ) {
            arp_table[j]=*ae;
            return(1);
        }
        j=(j+1)&ARP_TABLE_BIT_MASK;
    }
    return(0);
}
```

# Core Component Packet Handler (14)

```
/* Function to insert an entry into the NAT table */
int add_nat_entry(nat_entry* ne)
{
    ix_hash_128 hash128v;
    unsigned char timer, del_timer;
    int i,j,del_cand;
    hash128v.m_LW0 = ne->ip_addr_rem;
    hash128v.m_LW1 = ne->ip_addr_loc;
    hash128v.m_LW2 = (ne->lport<<16)|ne->rport;
    hash128v.m_LW3 = ne->prot;
    ix_rm_hash_128_hash(&hash128v);
    j = (hash128v.m_LW0&NAT_TABLE_BIT_MASK)<<HASH_BUCKET_SHIFT;
    del_cand=j;
    for (i=0;i<HASH_BUCKET_SIZE;i++,j++) {
        if ( f_nat_table[j].valid &&
            ne->ip_addr_loc == f_nat_table[j].ip_addr_loc &&
            ne->ip_addr_rem == f_nat_table[j].ip_addr_rem &&
            ne->lport == f_nat_table[j].lport &&
            ne->rport == f_nat_table[j].rport &&
            ne->prot == f_nat_table[j].prot ) {
            ne->nport=f_nat_table[j].nport;
            return(1);
        }
    }
}
```

# Core Component Packet Handler (15)

```
    if ( !f_nat_table[j].valid ) {
        if (set_new_nport(ne) < 0)
            return(-1);
        f_nat_table[j]=*ne;
        add_r_nat_entry(j);
        f_nat_table[j].valid=1;
        return(1);
    }
    /* No free slot was found; choose a candidate */
    /* for deletion */
    timer=f_timer[j]|r_timer[f_index[j]];
    del_timer = f_timer[del_cand]|r_timer[f_index[del_cand]];
    if ( timer < del_timer ||
        ( timer == del_timer &&
          f_nat_table[j].prot!=f_nat_table[del_cand].prot &&
          ( f_nat_table[j].prot == IPT_ICMP ||
            ( f_nat_table[j].prot == IPT_UDP &&
              f_nat_table[del_cand].prot == IPT_TCP )))
          del_cand=j;
    }
    del_nat_entry(del_cand);
    if (set_new_nport(ne) < 0)
        return(-1);
    f_nat_table[del_cand]=*ne;
    add_r_nat_entry(del_cand);
    f_nat_table[del_cand].valid=1;
    return(1);
}
```

# Core Component Packet Handler (16)

```
/* Function to delete an entry from the NAT table */
void del_nat_entry(unsigned int entry_index)
{
    f_nat_table[entry_index].valid=0;
    f_timer[entry_index]=0;
    r_nat_table[f_index[entry_index]].valid=0;
    r_timer[f_index[entry_index]]=0;
}

/* Function to add an entry to the reverse NAT table */
void add_r_nat_entry(unsigned int entry_index)
{
    ix_hash_128 hash128v;
    int i, j, k, del_cand, r_del_cand;
    unsigned char timer, del_timer;
    nat_entry *ne=&f_nat_table[entry_index];
    hash128v.m_LW0 = ne->ip_addr_rem;
    hash128v.m_LW1 = (ne->nport<<16)|ne->rport;
    hash128v.m_LW2 = ne->prot;
    hash128v.m_LW3 = 0;
    ix_rm_hash_128_hash(&hash128v);
    j=(hash128v.m_LW0&NAT_TABLE_BIT_MASK)<<HASH_BUCKET_SHIFT;
    del_cand=r_index[j];
    r_del_cand=j;
}
```

# Core Component Packet Handler (17)

```
for (i=0;i<HASH_BUCKET_SIZE;i++,j++) {
    /* Check whether the slot is empty */
    if (!r_nat_table[j].valid) {
        /* we found an empty slot in reverse NAT table */
        r_nat_table[j]=f_nat_table[entry_index];
        f_index[entry_index]=j;
        r_index[j]=entry_index;
        r_nat_table[j].valid=1;
        return;
    }
    /* Find a candidate for deletion */
    k = r_index[j];
    timer = f_timer[k]|r_timer[j];
    del_timer=f_timer[del_cand]|r_timer[r_del_cand];
    if ( timer < del_timer ||
        ( timer == del_timer &&
          f_nat_table[k].prot!=f_nat_table[del_cand].prot &&
          ( f_nat_table[k].prot == IPT_ICMP ||
            ( f_nat_table[k].prot == IPT_UDP &&
              f_nat_table[del_cand].prot == IPT_TCP )))) {
        del_cand=k;
        r_del_cand=j;
    }
}
```

# Core Component Packet Handler (18)

```
/* This point is reached if no slot is empty */
del_nat_entry(del_cand);
r_nat_table[r_del_cand]=f_nat_table[entry_index];
r_index[r_del_cand] = entry_index;
f_index[entry_index] = r_del_cand;
r_nat_table[r_del_cand].valid=1;
}

/* Function to calculate a value for a new NAT port */
int set_new_nport(nat_entry* ne)
{
    ix_hash_128 hash128v;
    int i,j,k;

    ne->nport=++global_nport;
    /* Try at most NEW_NPORT_ATTEMPS values, and then give up */
    for (i=0;i<NEW_NPORT_ATTEMPS;i++) {
        hash128v.m_LW0 = ne->ip_addr_rem;
        hash128v.m_LW1 = (ne->nport<<16)|ne->rport;
        hash128v.m_LW2 = ne->prot;
        hash128v.m_LW3 = 0;
        ix_rm_hash_128_hash(&hash128v);
        j=(hash128v.m_LW0&NAT_TABLE_BIT_MASK)<<HASH_BUCKET_SHIFT;
```

# Core Component Packet Handler (19)

```
for (k=0;k<HASH_BUCKET_SIZE;k++,j++) {
    if ( r_nat_table[j].valid &&
        r_nat_table[j].ip_addr_rem == ne->ip_addr_rem &&
        r_nat_table[j].rport == ne->rport &&
        r_nat_table[j].nport == ne->nport &&
        r_nat_table[j].prot == ne->prot )
        break;
    }
if (k==HASH_BUCKET_SIZE)
    /* An unused NAT port value has been found */
    return(ne->nport);
/* try the next NAT port value */
ne->nport=++global_nport;
}
return(-1);
}
```

# Core Component Packet Handler (20)

```
/* Function to send echo response */
void send_icmp_echo_rep(ip* ip_pkt,icmp* icmp_pkt,
    ix_hw_buffer_meta* meta_data,ix_buffer_handle arg_hBuffer,
    eth *eth_pkt)
{
    unsigned int cksum;
    icmp_pkt->icmp_type = ICMP_ECHO_REP;
    cksum = icmp_pkt->icmp_cksum+(ICMP_ECHO_REQ<<8)
        + ((~(ICMP_ECHO_REP<<8))&0xFFFF);
    cksum = (cksum&0xFFFF)+(cksum>>16);
    cksum = (cksum&0xFFFF)+(cksum>>16);
    icmp_pkt->icmp_cksum = cksum&0xFFFF;
    ip_pkt->ip_dst = ip_pkt->ip_src;
    ip_pkt->ip_src = iface_table[meta_data->m_InputPort].ip_addr;
    send_pkt((void*)arg_hBuffer,meta_data->m_InputPort, eth_pkt,
        eth_pkt->e_src, ETH_IP);
}
```



# Core Component Packet Handler (21)

```
/* Function to translate ICMP packet */
int process_icmp(icmp* icmp_pkt,nat_entry* ne)
{
    unsigned int cksum;
    /* If this is not an echo request -- drop the packet */
    if (icmp_pkt->icmp_type != ICMP_ECHO_REQ)
        return(-1);
    /* For ICMP echo request we do ID field translation */
    ne->lport=icmp_pkt->icmp_id;
    ne->rport=0;
    if (add_nat_entry(ne) < 0)
        return(-1);
    icmp_pkt->icmp_id=ne->nport;
    /* update ICMP checksum */
    cksum= icmp_pkt->icmp_cksum+ne->lport
        + ((~icmp_pkt->icmp_id)&0xFFFF);
    cksum=(cksum&0xFFFF)+(cksum>>16);
    cksum=(cksum&0xFFFF)+(cksum>>16);
    icmp_pkt->icmp_cksum = cksum&0xFFFF;
    return(1);
}
```

# Core Component Packet Handler (22)

```
/* Function to translate TCP packet */
int process_tcp(tcp* tcp_pkt,nat_entry* ne)
{
    unsigned int cksum;
    if (verb == VERBOSE) {
        printk("\tTCP source port = %i\n", tcp_pkt->tcp_sport);
        printk("\tTCP dest. port = %i\n", tcp_pkt->tcp_dport);
    }
    /* Perform TCP source port translation */
    ne->lport=tcp_pkt->tcp_sport;
    ne->rport=tcp_pkt->tcp_dport;
    if (add_nat_entry(ne) < 0)
        return(-1);
    tcp_pkt->tcp_sport=ne->nport;
    /* Update the TCP checksum */
    cksum = tcp_pkt->tcp_cksum+ne->lport+(ne->ip_addr_loc>>16)
        + (ne->ip_addr_loc&0xFFFF)+((~tcp_pkt->tcp_sport)&0xFFFF)
        + ((~iface_table[NAT_IFC].ip_addr)>>16)
        + ((~iface_table[NAT_IFC].ip_addr)&0xFFFF);
    cksum=(cksum&0xFFFF)+(cksum>>16);
    cksum=(cksum&0xFFFF)+(cksum>>16);
    tcp_pkt->tcp_cksum = cksum&0xFFFF;
    return(1);
}
```

# Core Component Packet Handler (23)

```
/* Function to translate UDP packet */
int process_udp(udp* udp_pkt,nat_entry* ne)
{
    unsigned int cksum;
    if (verb == VERBOSE) {
        printk("\tUDP source port = %i\n", udp_pkt->udp_sport);
        printk("\tUDP dest. port = %i\n", udp_pkt->udp_dport);
    }
    /* Perform UDP source port translation */
    ne->lport=udp_pkt->udp_sport;
    ne->rport=udp_pkt->udp_dport;
    if (add_nat_entry(ne) < 0)
        return(-1);
    udp_pkt->udp_sport=ne->nport;
    /* Update the UDP checksum */
    if (udp_pkt->udp_cksum) {
        cksum = udp_pkt->udp_cksum+ne->lport+(ne->ip_addr_loc>>16)
            + (ne->ip_addr_loc&0xFFFF)
            + ((~udp_pkt->udp_sport)&0xFFFF)
            + ((~iface_table[NAT_IFC].ip_addr)>>16)
            + ((~iface_table[NAT_IFC].ip_addr)&0xFFFF);
        cksum=(cksum&0xFFFF)+(cksum>>16);
        cksum=(cksum&0xFFFF)+(cksum>>16);
        udp_pkt->udp_cksum = cksum&0xFFFF;
    }
    return(1);
}
```

# User Interface Application

- Allows user to interact with core component
- Core component
  - Defines pseudo-device in Linux kernel
  - Installs driver for pseudo-device
- To execute a command, user interface performs an operation on the pseudo-device

# Code For User Interface (1)

```
/* NAT_control.c -- user interface and control functions for NAT */

#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <enpv2_types.h>
#include <sys/mman.h>

#include "NAT_shared_defs.h"
#include "NAT_types.h"

#define USAGE \
"Usage: %s [ show | clear | silent | arp | nat | verbose ]\n", argv[0]

void AppShow( void )
{
    int i;
    UINT32 *p1, *p2;
    char buf[1024];
    int natfd;
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n", NAT_DEV_FILE);
        return;
    }
}
```

## Code For User Interface (2)

```
void AppSilent()
{
    int natfd;
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,SILENT,NULL);
    close(natfd);
}

void AppVerbose()
{
    int natfd;
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,VERBOSE,NULL);
    close(natfd);
}
```

## Code For User Interface (3)

```
void AppSilent()
{
    int natfd;
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,SILENT,NULL);
    close(natfd);
}

void AppVerbose()
{
    int natfd;
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,VERBOSE,NULL);
    close(natfd);
}
```

# Code For User Interface (4)

```
void AppClear( void )
{
    int natfd;
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,CLR_RX_COUNTER,NULL);
    ioctl(natfd,CLR_TX_COUNTER,NULL);
    close(natfd);
    printf("Counters cleared\n");
    return;
}
```



## Code For User Interface (5)

```
void AppGetArpTbl( void )
{
    int natfd,i;
    arp_entry buf[ARP_TABLE_SIZE];
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,GET_ARP_TABLE,buf);
    close(natfd);
    for (i=0;i<ARP_TABLE_SIZE;i++) {
        if (buf[i].valid)
            printf(
                "%i.%i.%i.%i -> %02X:%02X:%02X:%02X:%02X:%02X  iface: %i\n",
                IP2B(buf[i].ip_addr), ETH2B(buf[i].eth_w0),
                buf[i].ifnum);
    }
    return;
}
```

## Code For User Interface (6)

```
void AppGetNatTbl( void )
{
    int natfd,i;
    nat_entry buf[NAT_TABLE_SIZE];
    unsigned char timer[2*NAT_TABLE_SIZE];
    natfd = open(NAT_DEV_FILE, O_RDWR, 0);
    if ( natfd == -1 ) {
        printf("Failed to open %s\n",NAT_DEV_FILE);
        return;
    }
    ioctl(natfd,GET_NAT_TABLE,buf);
    ioctl(natfd,GET_TIMER_TABLE,timer);
    for (i=0;i<NAT_TABLE_SIZE;i++) {
        if (buf[i].valid) {
            printf("IP local: %i.%i.%i.%i port local: %i\n",
                IP2B(buf[i].ip_addr_loc), buf[i].lport);
            printf("IP remote: %i.%i.%i.%i port remote: %i\n",
                IP2B(buf[i].ip_addr_rem), buf[i].rport);
            printf("protocol: %i port (NAT): %i timer: %i index: %i\n\n",
                buf[i].prot, buf[i].nport,
                timer[i]|timer[NAT_TABLE_SIZE+i],i);
        }
    }
    close(natfd);
    return;
}
```

# Code For User Interface (7)

```
int main(int argc, char **argv)
{
    if (argc != 2) {
        printf(USAGE);
        return 0;
    }
    if (strncmp(argv[1], "show", 4) == 0) {
        AppShow();
    } else if (strncmp(argv[1], "clear", 5) == 0) {
        AppClear();
    } else if (strncmp(argv[1], "silent", 6) == 0) {
        AppSilent();
    } else if (strncmp(argv[1], "verbose", 7) == 0) {
        AppVerbose();
    } else if (strncmp(argv[1], "arp", 3) == 0) {
        AppGetArpTbl();
    } else if (strncmp(argv[1], "nat", 3) == 0) {
        AppGetNatTbl();
    } else {
        printf("Invalid parameter\n");
        printf(USAGE);
    }
    return 0;
}
```

# Summary

- Example system implements NAT
- Code uses RX and TX microblocks from Intel's SDK
- NAT microblock implements NAT in fast path
- Core component handles exceptions
- User interface provides interaction with core component



**Questions?**

# X

## Switching Fabrics

# Physical Interconnection

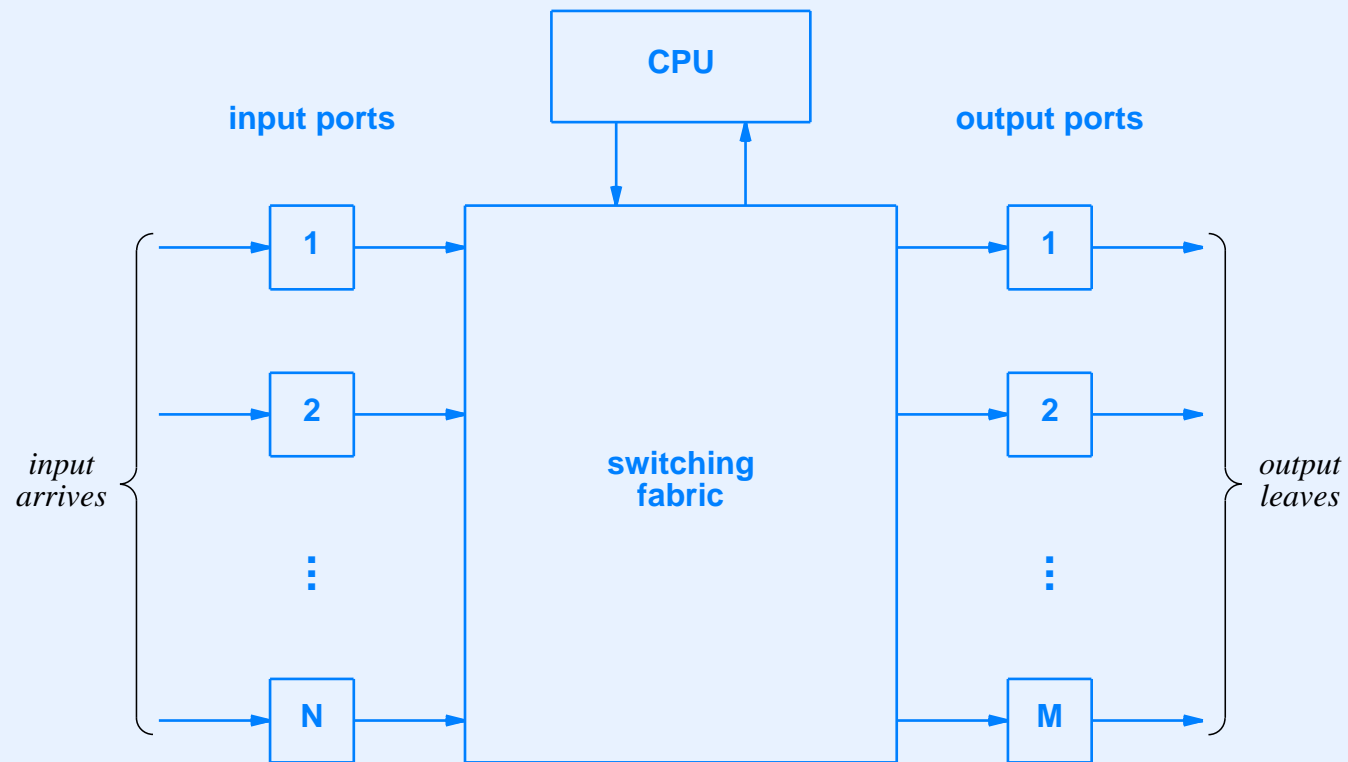
- Physical box with backplane
- Individual *blades* plug into backplane slots
- Each blade contains one or more network connections

# Logical Interconnection

- Known as *switching fabric*
- Handles transport from one blade to another
- Becomes bottleneck as number of interfaces scales



# Illustration Of Switching Fabric



- Any input port can send to any output port

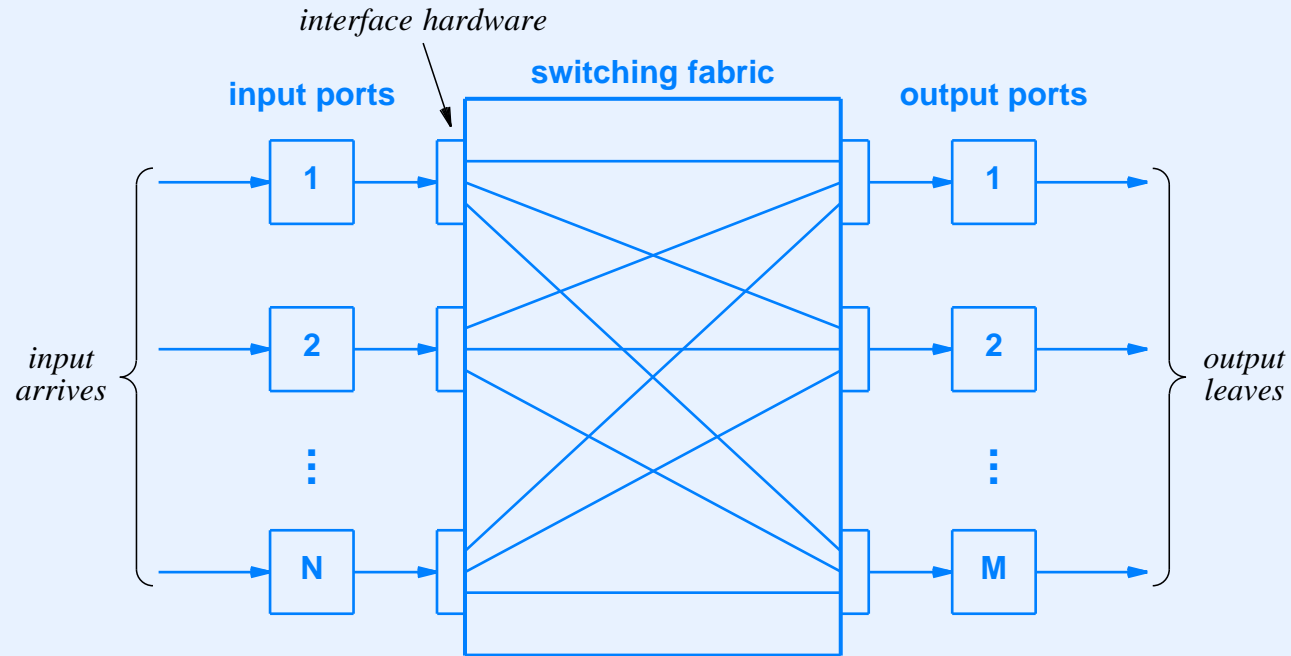
# Switching Fabric Properties

- Used inside a single network system
- Interconnection among I/O ports (and possibly CPU)
- Can transfer unicast, multicast, and broadcast packets
- Scales to arbitrary data rate on any port
- Scales to arbitrary packet rate on any port
- Scales to arbitrary number of ports
- Has low overhead
- Has low cost

# Types Of Switching Fabrics

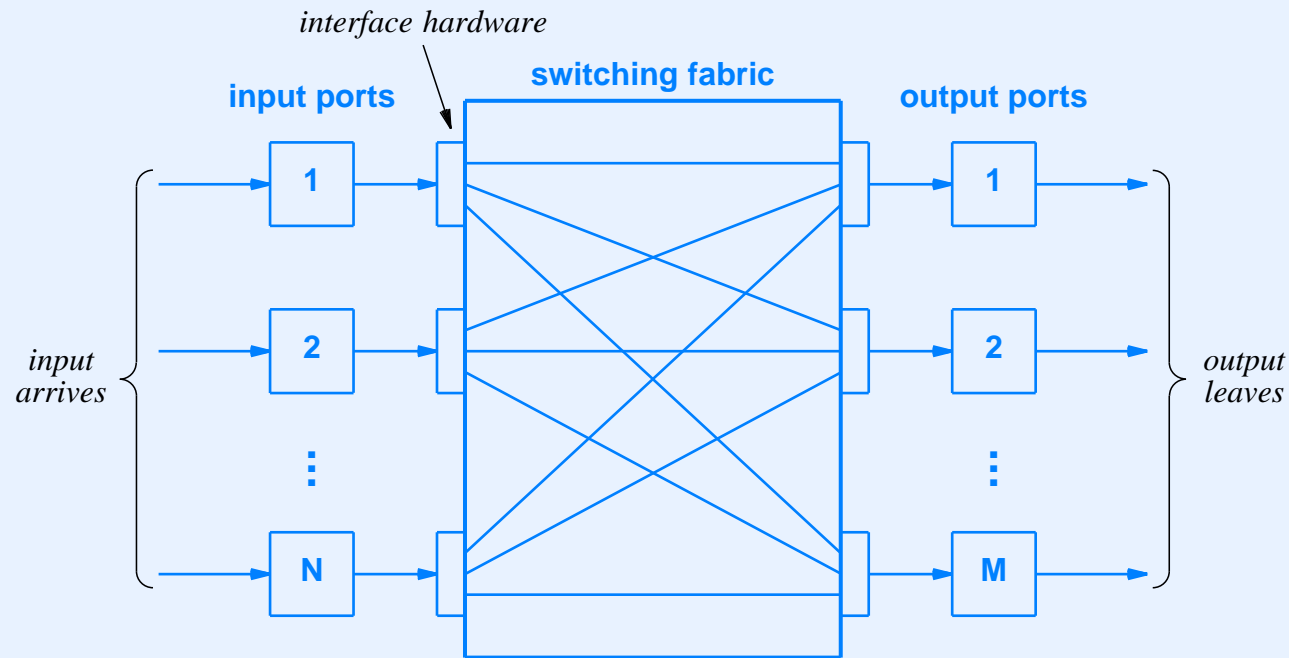
- Space-division (separate paths)
- Time-division (shared medium)

# Space-Division Fabric (separate paths)



- Can use multiple paths simultaneously

# Space-Division Fabric (separate paths)



- Can use multiple paths simultaneously
- *Still have port contention*

# Desires

# Desires

- High speed

# Desires

- High speed
- Low cost



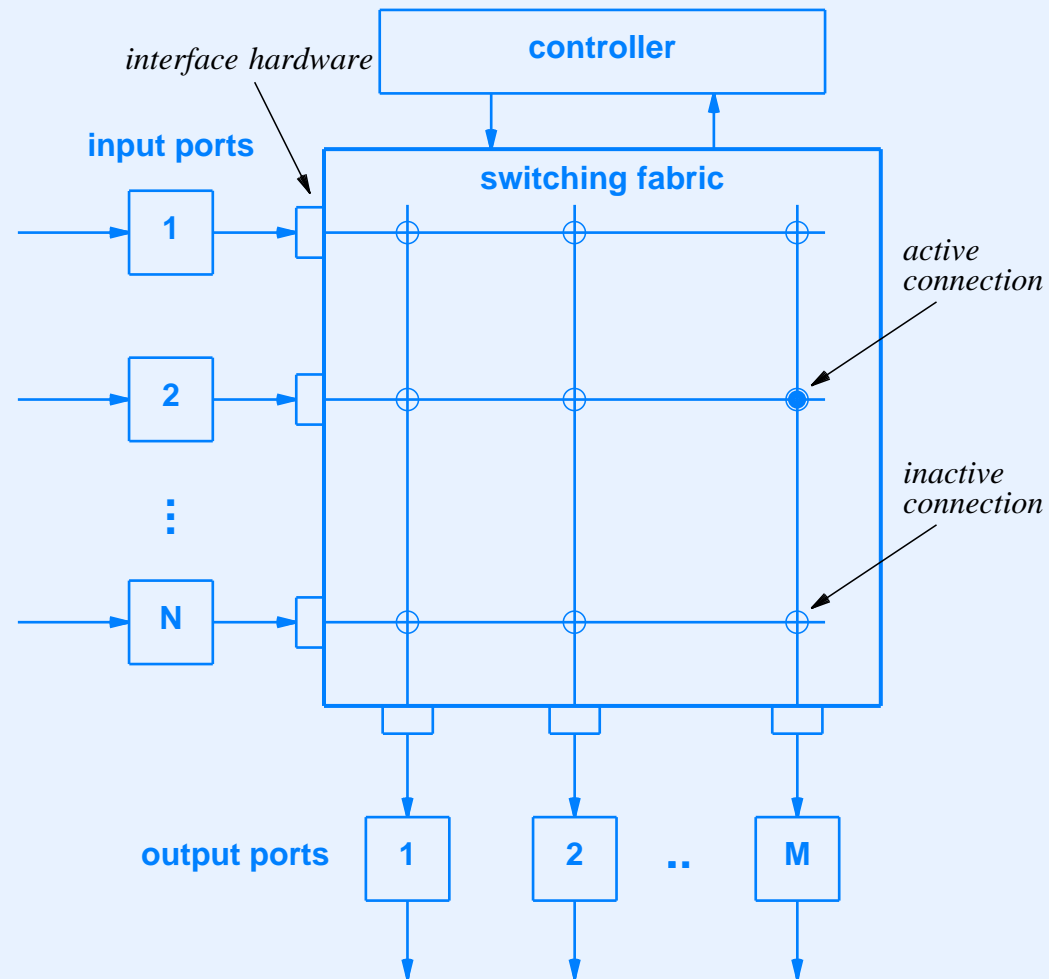
# Desires

- High speed *and* low cost!

# Possible Compromise

- Separation of physical paths
- Less parallel hardware
- Crossbar design

# Space-Division (Crossbar Fabric)



# Crossbar

- Allows simultaneous transfer on disjoint pairs of ports
- Can still have *port contention*

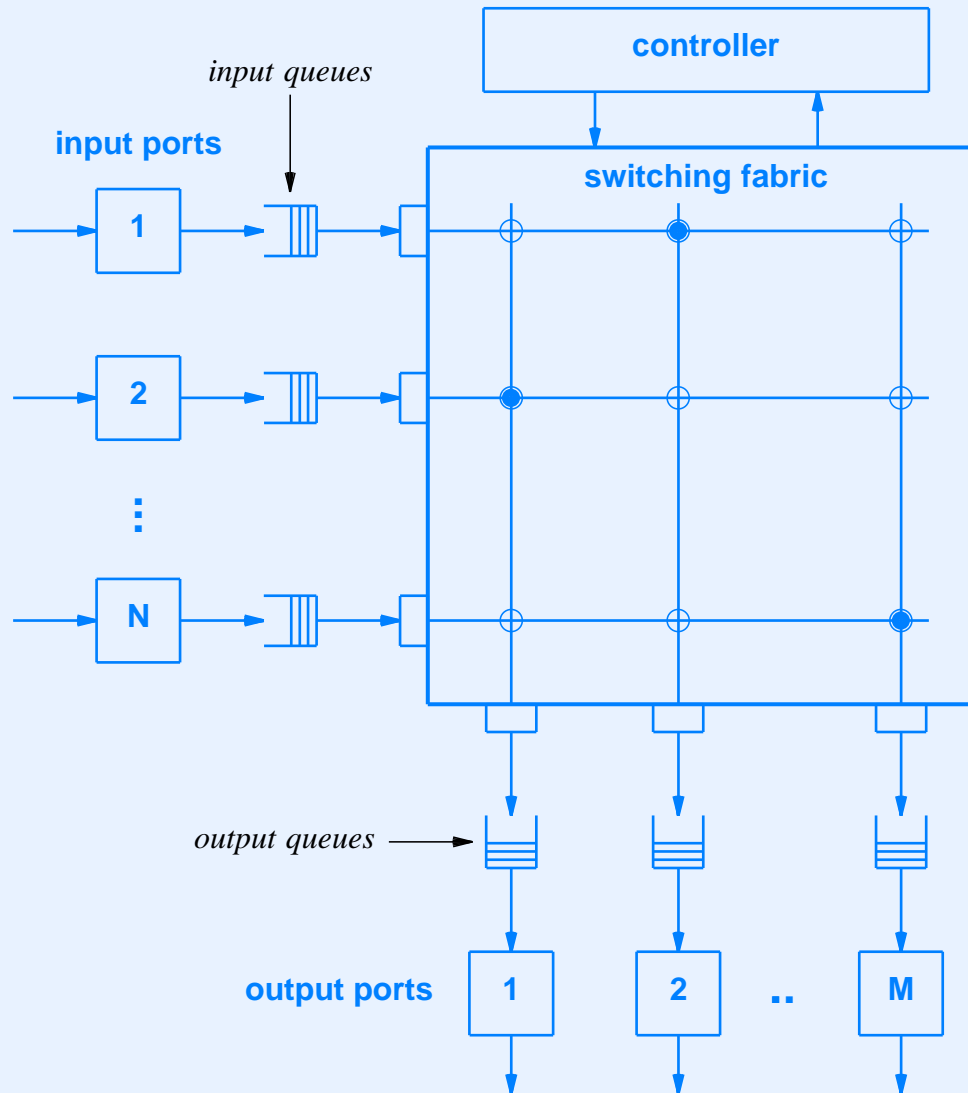
# Crossbar

- Allows simultaneous transfer on disjoint pairs of ports
- Can still have *port contention*

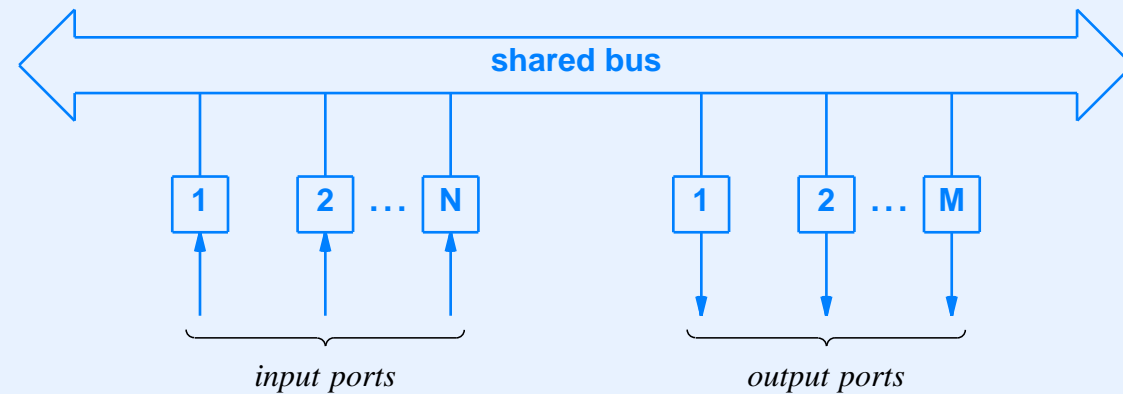
# Solving Contention

- Queues (FIFOs)
  - Attached to input
  - Attached to output
  - At intermediate points

# Crossbar Fabric With Queuing



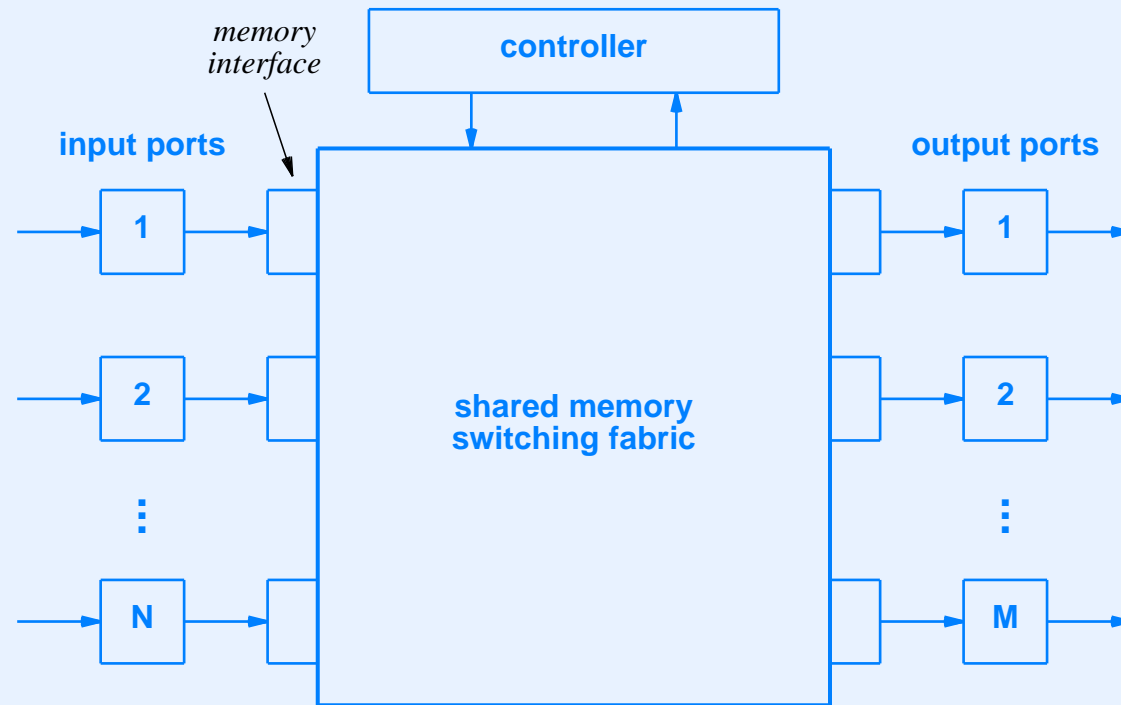
# Time-Division Fabric (shared bus)



- Chief advantage: low cost
- Chief disadvantage: low speed



# Time-Division Fabric (shared memory)



- *May* be better than shared bus
- Usually more expensive

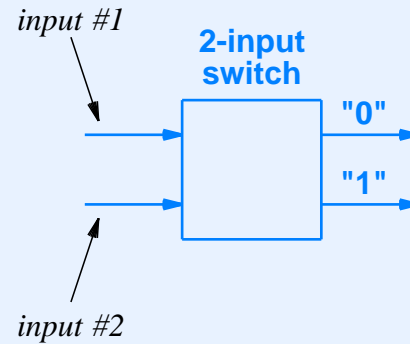
# Multi-Stage Fabrics

- Compromise between pure time-division and pure space-division
- Attempt to combine advantages of each
  - Lower cost from time-division
  - Higher performance from space-division
- Technique: limited sharing

# Banyan Fabric

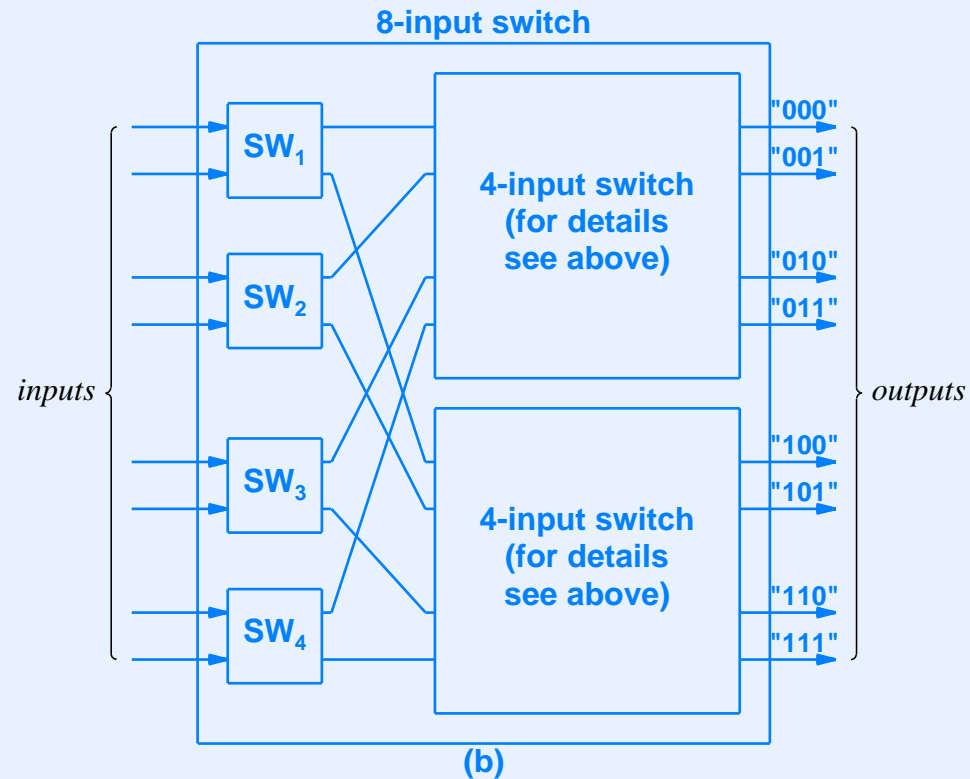
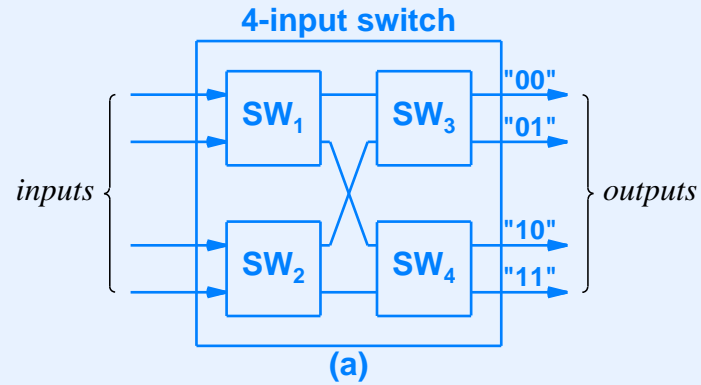
- Example of multi-stage fabric
- Features
  - Scalable
  - Self-routing
  - Packet queues allowed, but not required

# Basic Banyan Building Block



- Address added to front of each packet
- One bit of address used to select output

# 4-Input And 8-Input Banyan Switches



# Summary

- Switching fabric provides connections inside single network system
- Two basic approaches
  - Time-division has lowest cost
  - Space-division has highest performance
- Multistage designs compromise between two
- Banyan fabric is example of multistage



**Questions?**

# **XIV**

## **Issues In Scaling A Network Processor**



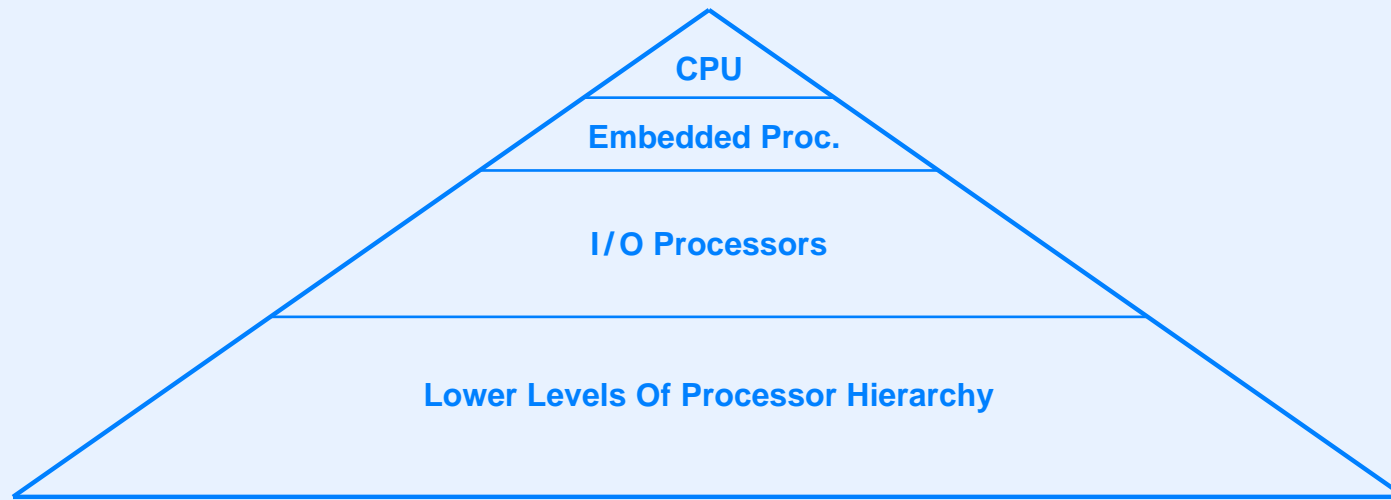
# Design Questions

- Can we make network processors
  - Faster?
  - Easier to use?
  - More powerful?
  - More general?
  - Cheaper?
  - All of the above?
- Scale is fundamental

# Scaling The Processor Hierarchy

- Make processors faster
- Use more concurrent threads
- Increase processor types
- Increase numbers of processors

# The Pyramid Of Processor Scale



- Lower levels need the most increase

# Scaling The Memory Hierarchy

- Size
- Speed
- Throughput
- Cost

# Memory Speed

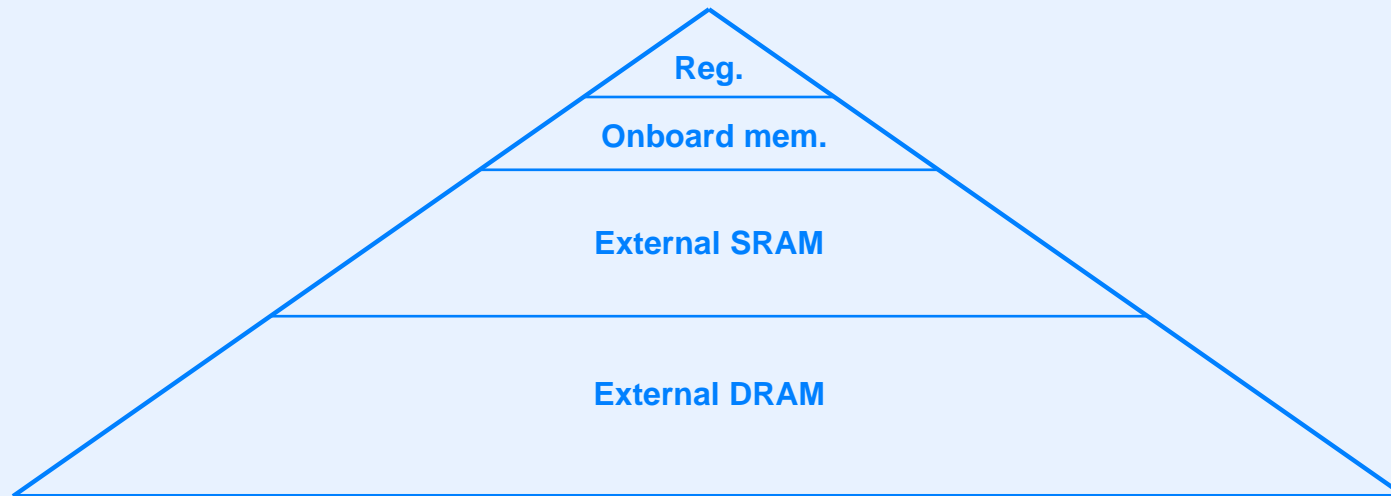
- Access latency
  - Raw read/write access speed
  - SRAM 2 - 10 ns
  - DRAM 50 - 70 ns
  - External memory takes order of magnitude longer than onboard

# Memory Speed

## (continued)

- Memory cycle time
  - Measure of successive read/write operations
  - Important for networking because packets are large
  - Read Cycle time (tRC) is time for successive fetch operations
  - Write Cycle time (tWC) is time for successive store operations

# The Pyramid Of Memory Scale



- Largest memory is least expensive

# Memory Bandwidth

- General measure of throughput
- More parallelism in access path yields more throughput
- Cannot scale arbitrarily
  - Pinout limits
  - Processor must have addresses as wide as bus



# Types Of Memory

<b>Memory Technology</b>	<b>Abbreviation</b>	<b>Purpose</b>
<b>Synchronized DRAM</b>	<b>SDRAM</b>	<b>Synchronized with CPU for lower latency</b>
<b>Quad Data Rate SRAM</b>	<b>QDR-SRAM</b>	<b>Optimized for low latency and multiple access</b>
<b>Zero Bus Turnaround SRAM</b>	<b>ZBT-SRAM</b>	<b>Optimized for random access</b>
<b>Fast Cycle RAM</b>	<b>FCRAM</b>	<b>Low cycle time optimized for block transfer</b>
<b>Double Data Rate DRAM</b>	<b>DDR-DRAM</b>	<b>Optimized for low latency</b>
<b>Reduced Latency DRAM</b>	<b>RLDRAM</b>	<b>Low cycle time and low power requirements</b>

# Memory Cache

- General-purpose technique
- May not work well in network systems

# Memory Cache

- General-purpose technique
- May not work well in network systems
  - Low temporal locality

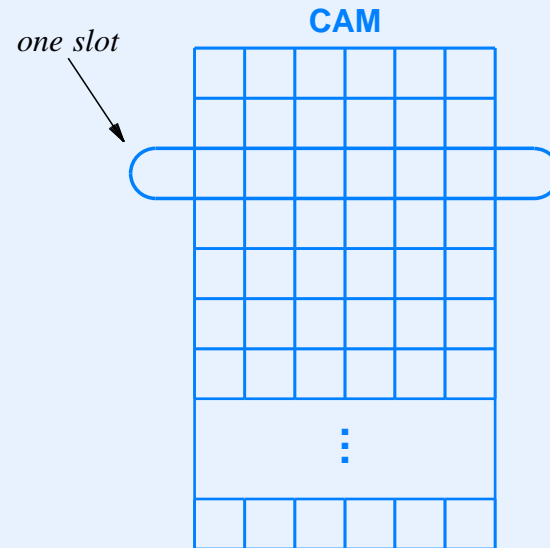
# Memory Cache

- General-purpose technique
- May not work well in network systems
  - Low temporal locality
  - Large cache size (either more entries or larger granularity of access)

# Content Addressable Memory (CAM)

- Combination of mechanisms
  - Random access storage
  - Exact-match pattern search
- Rapid search enabled with parallel hardware

# Arrangement Of CAM



- Organized as array of slots

# Lookup In Conventional CAM

- Given
  - Pattern for which to search
  - Known as *key*
- CAM returns
  - First slot that matches key, or
  - All slots that match key

# Ternary CAM (T-CAM)

- Allows masking of entries
- Good for network processor



# T-CAM Lookup

- Each slot has bit mask
- Hardware uses mask to decide which bits to test
- Algorithm

```
for each slot do {  
    if ( ( key & mask ) == ( slot & mask ) ) {  
        declare key matches slot;  
    } else {  
        declare key does not match slot;  
    }  
}
```

# Partial Matching With A T-CAM

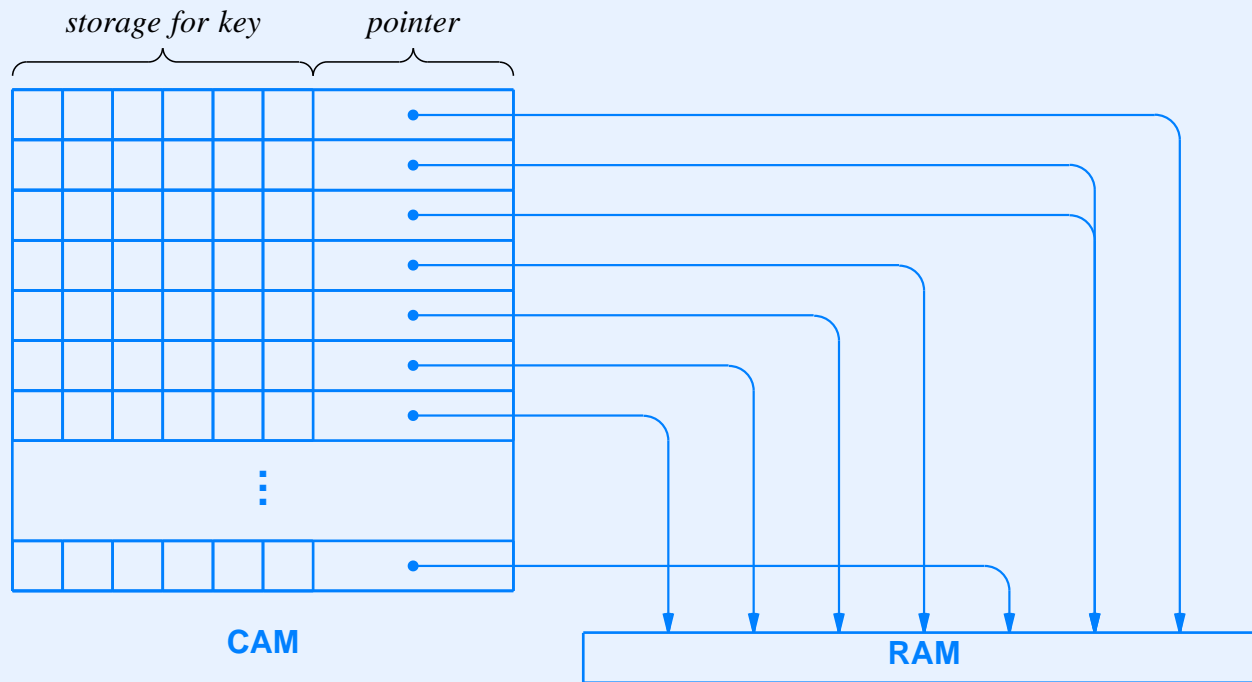


- Key matches slot #1

# Using A T-CAM For Classification

- Extract values from fields in headers
- Form values in contiguous string
- Use a key for T-CAM lookup
- Store classification in slot

# Classification Using A T-CAM



# Software Scalability

- Not always easy
- Many resource constraints
- Difficulty arises from
  - Explicit parallelism
  - Code optimized by hand
  - Pipelines on heterogeneous hardware

# Summary

- Scalability key issue
- Primary subsystems affecting scale
  - Processor hierarchy
  - Memory hierarchy
- Many memory types available
  - SRAM
  - SDRAM
  - CAM
- T-CAM useful for classification



**Questions?**

# **XV**

## **Examples Of Commercial Network Processors**



# Commercial Products

- Emerge in late 1990s
- Become popular in early 2000s
- Exceed thirty vendors by 2003
- Fewer than thirty vendors by 2004

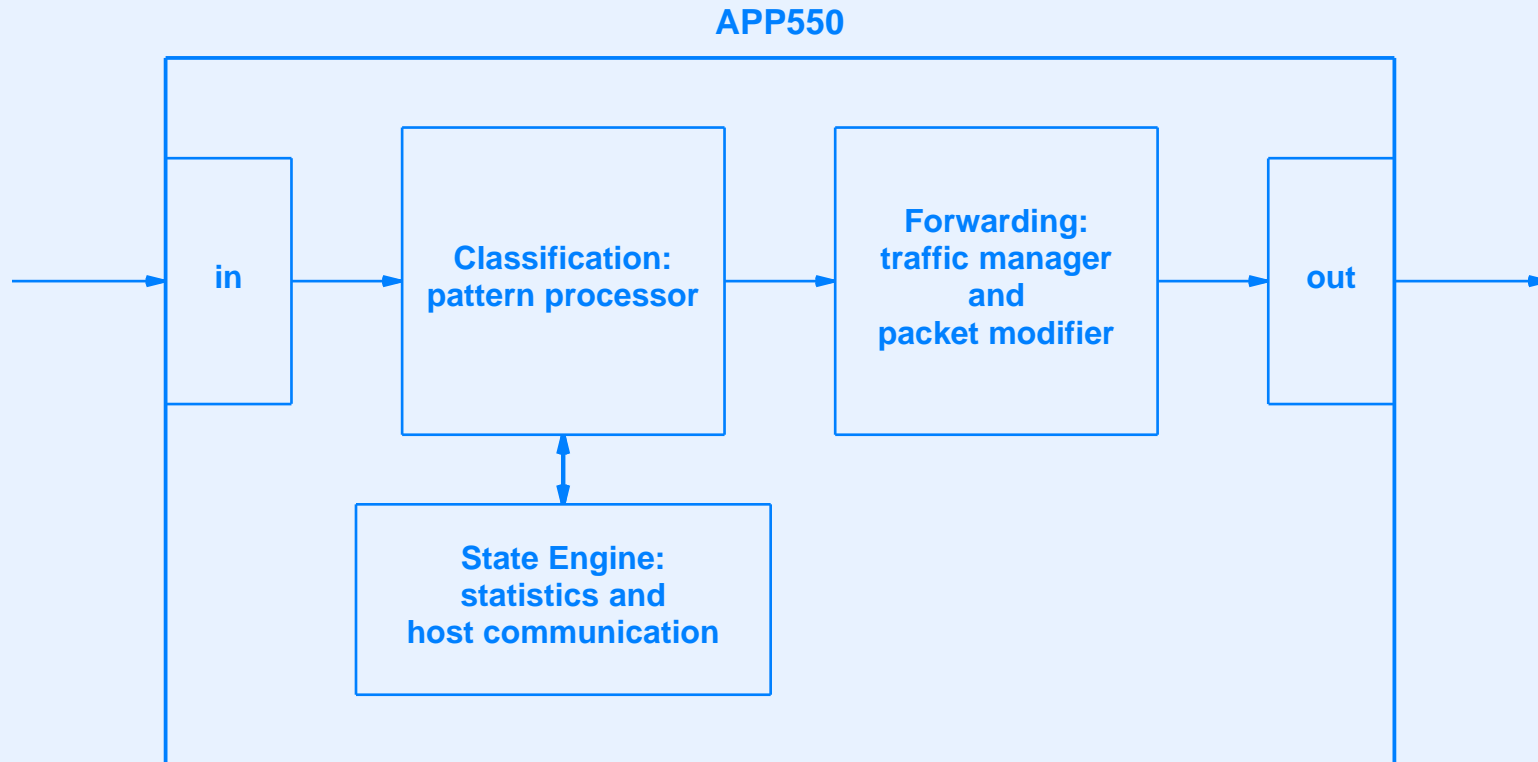
# Examples

- Chosen to
  - Illustrate concepts
  - Show broad categories
  - Expose the variety
- Not necessarily “best”
- Not meant as an endorsement of specific vendors
- Show a snapshot as of 2004

# Short Pipeline Of Unconventional Processors (Agere)

- Two-stage pipeline
  - Classification
  - Forwarding (traffic management)
- Unusual, special-purpose processors
  - Classification uses programmable pattern matching engine
  - Traffic manager uses programmable queue selection mechanism
- Model is *APP550*

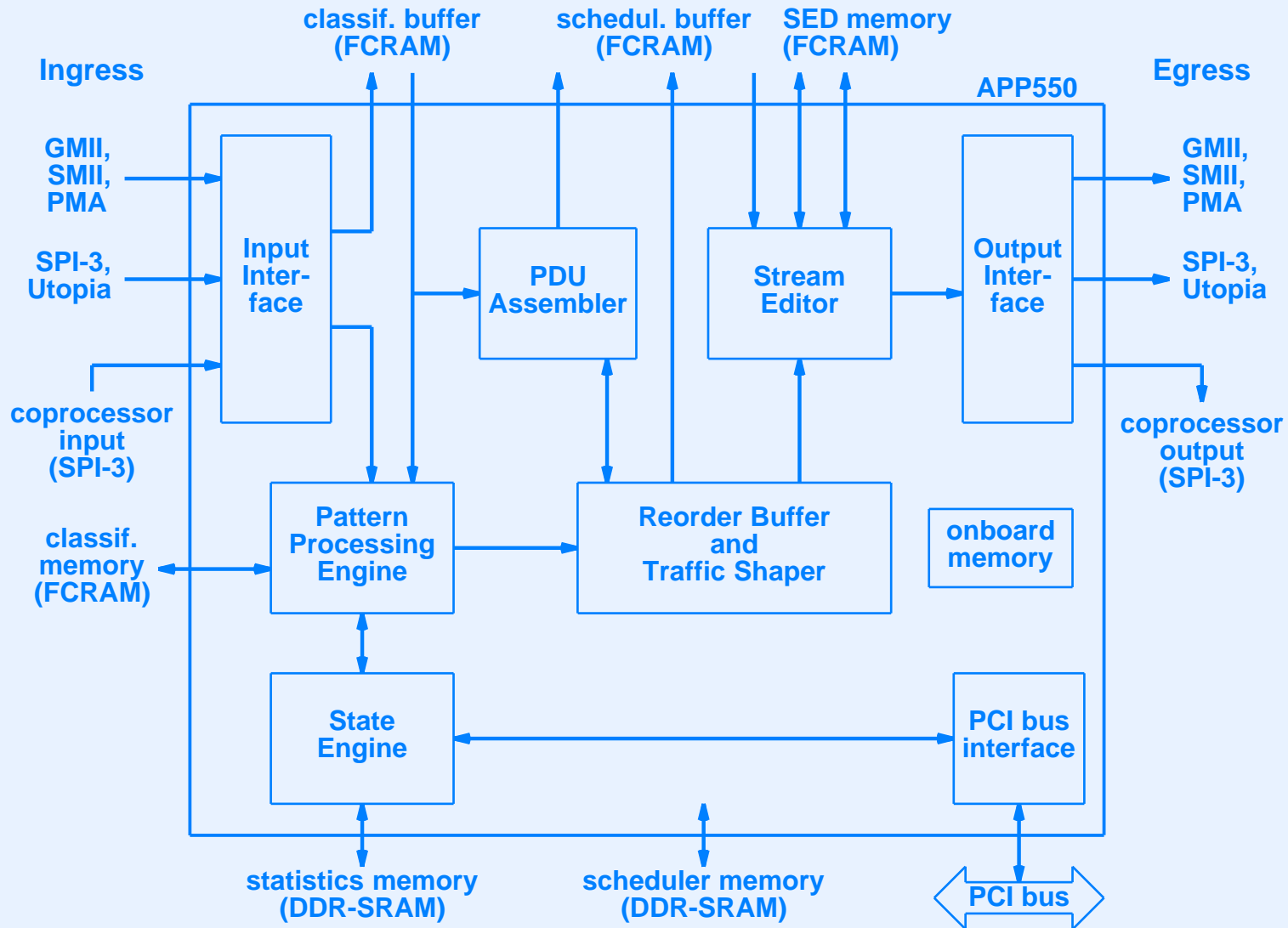
# Agere Architecture



# Languages Used By Agere

- FPL
  - Functional Programming Language
  - Produces code for FPP
  - Non-procedural
- C-NP
  - C for Network Processors
  - Produces code for engines on chip
  - Similar to shell scripts

# Architecture Of Agere's APP550 chip



# Processors On Agere's APP550

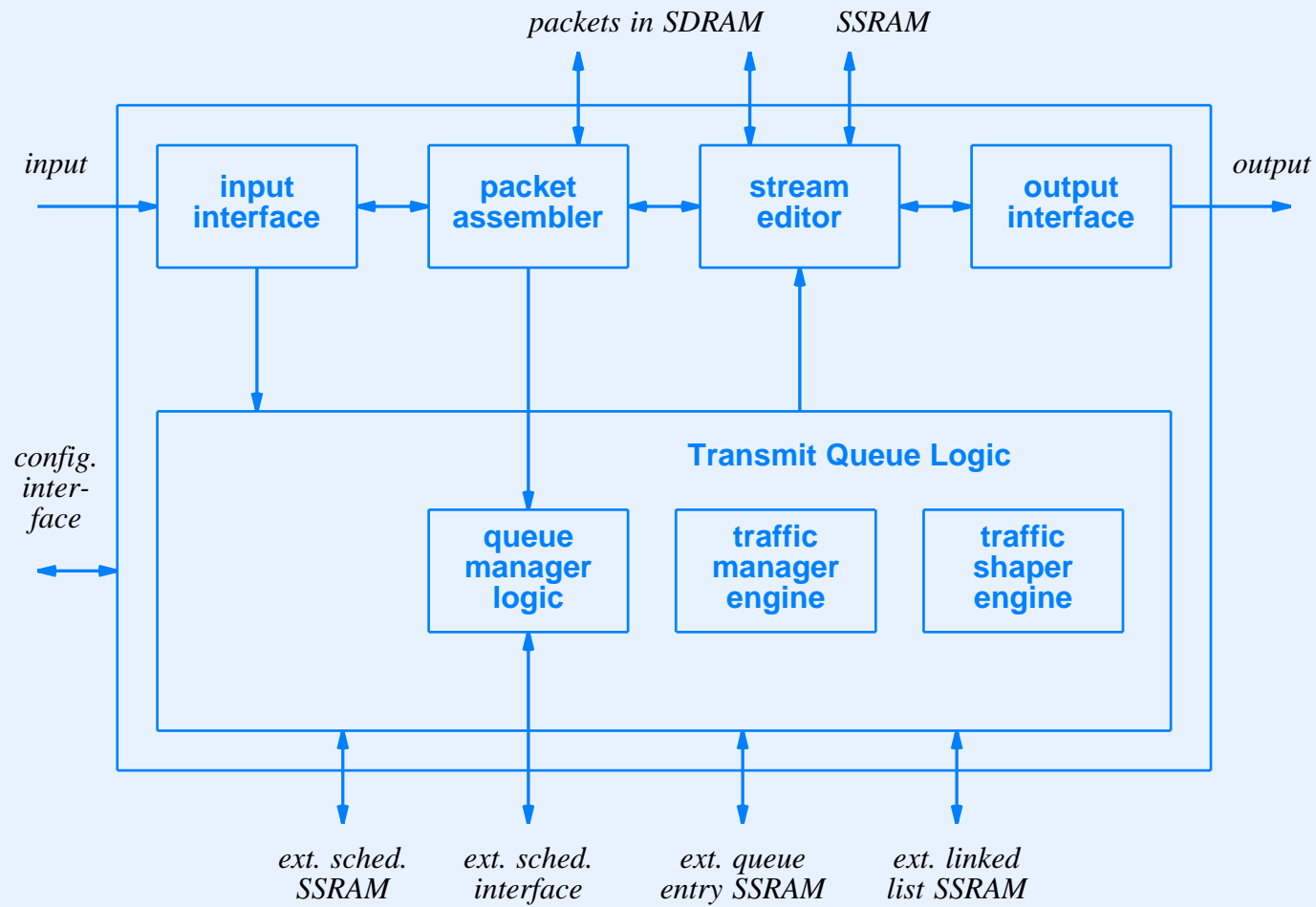
<b>Engine</b>	<b>Purpose</b>
<b>Pattern Processing Engine</b>	<b>Classification</b>
<b>State Engine</b>	<b>Gathering state information for scheduling and verifying flow is within bounds</b>
<b>Reorder Buffer Manager</b>	<b>Ensure packet order</b>
<b>PDU Assembler</b>	<b>Collect all blocks of a frame</b>
<b>Traffic Manager</b>	<b>Schedule packets and shape traffic flow</b>
<b>Stream Editor (SED)</b>	<b>Modify outgoing packet</b>

# Augmented RISC (Alchemy)

- Based on MIPS-32 CPU
  - Five-stage pipeline
- Augmented for packet processing
  - Instructions (e.g. multiply-and-accumulate)
  - Memory cache
  - I/O interfaces



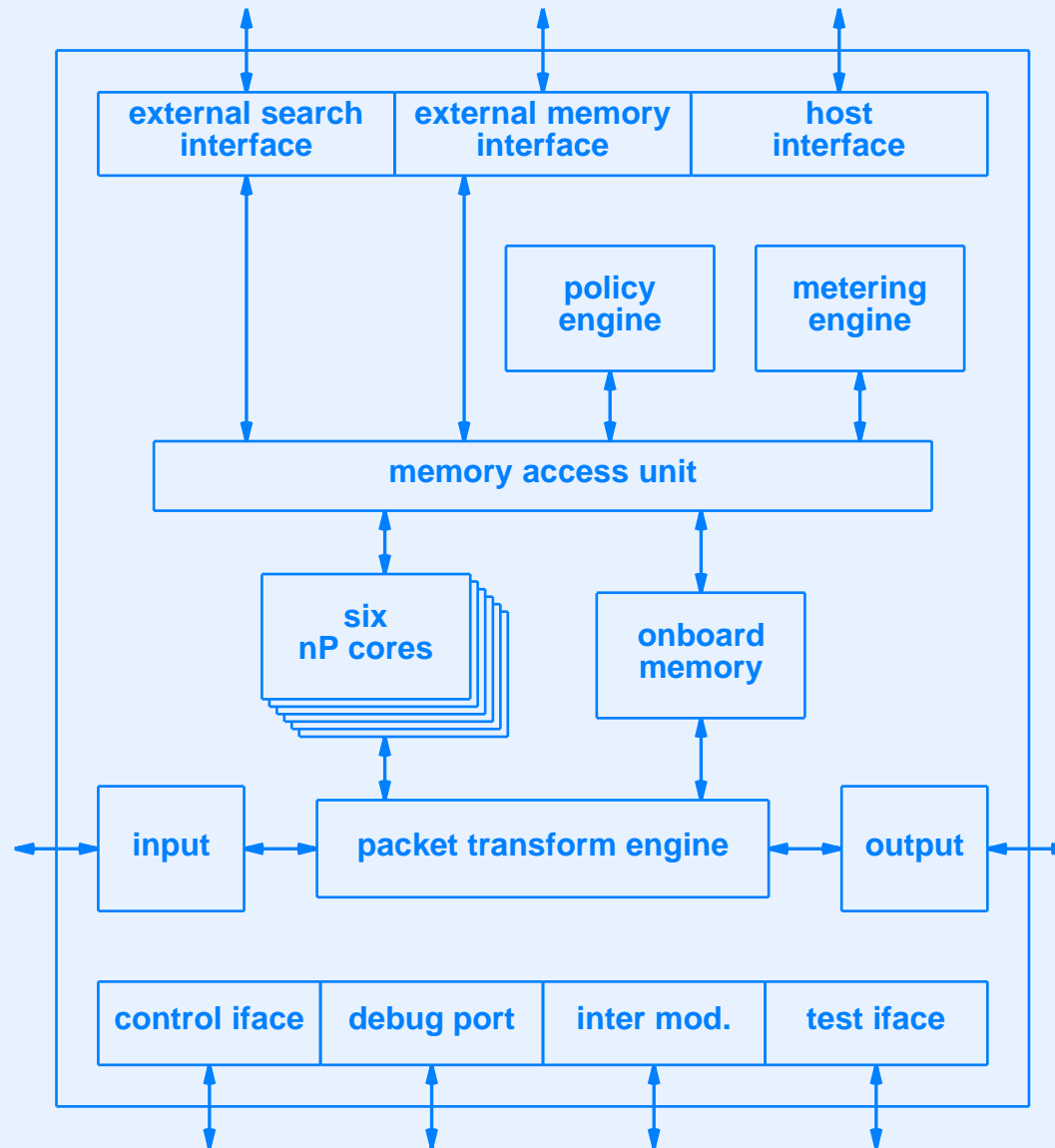
# Alchemy Architecture



# Parallel Embedded Processors Plus Coprocessors (AMCC)

- One to six nP core processors
- Various engines
  - Packet metering
  - Packet transform
  - Packet policy

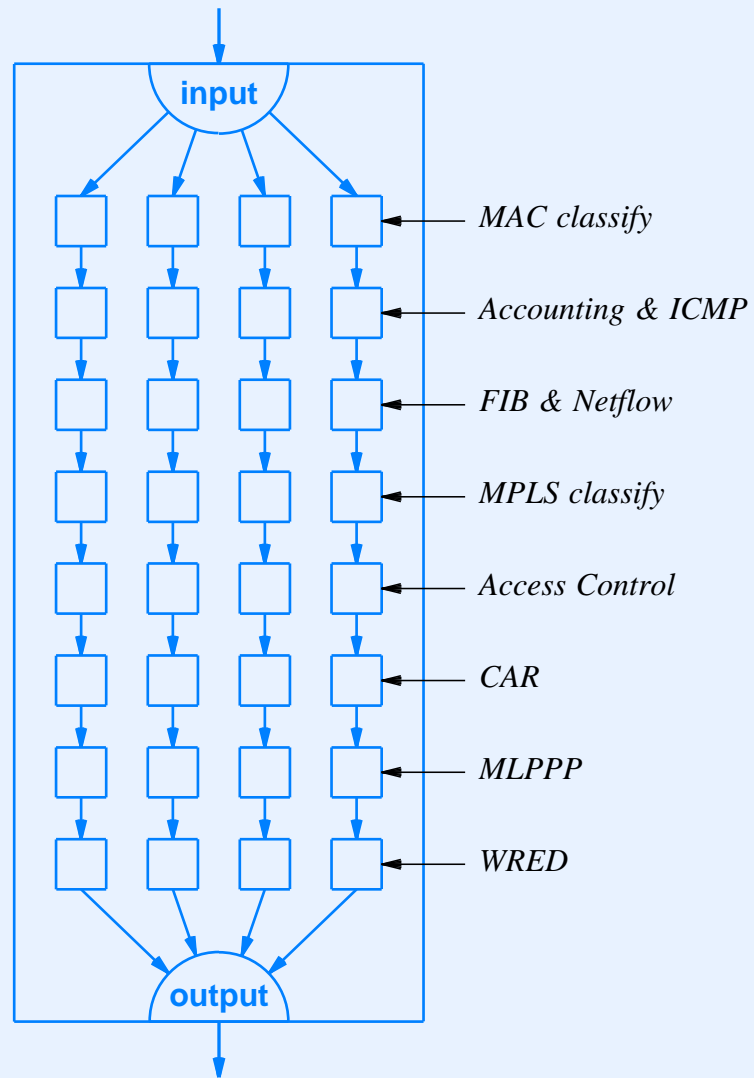
# AMCC Architecture



# Parallel Pipelines Of Homogeneous Processors (Cisco)

- Parallel eXpress Forwarding (PXF)
- Arranged in parallel pipelines
- Packet flows through one pipeline
- Each processor in pipeline dedicated to one task

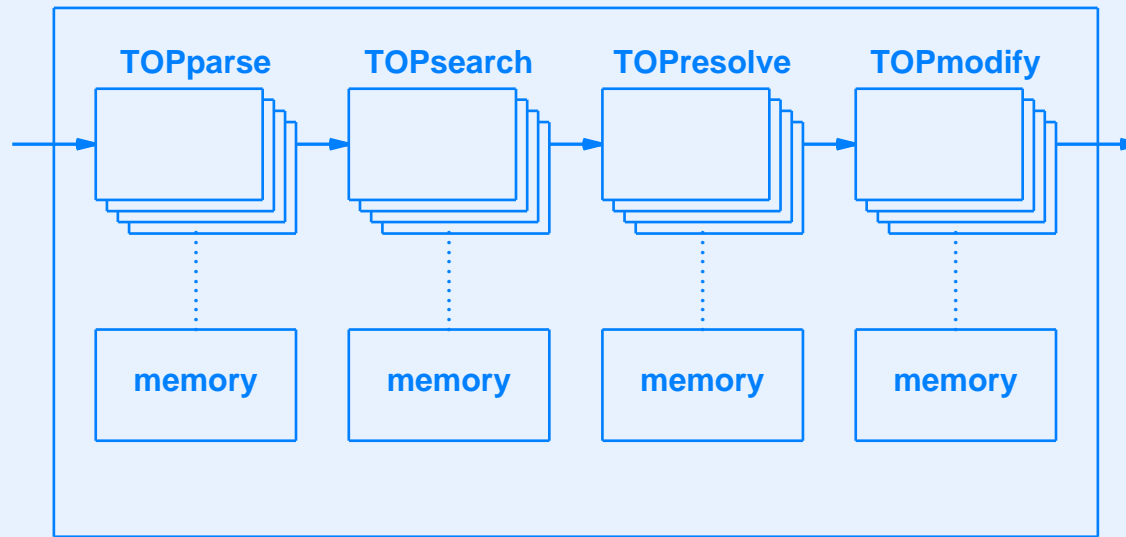
# Cisco Architecture



# Pipeline Of Parallel Heterogeneous Processors (EZchip)

- Four processor types
- Each type optimized for specific task

# EZchip NP-1c Architecture



# EZchip Processor Types

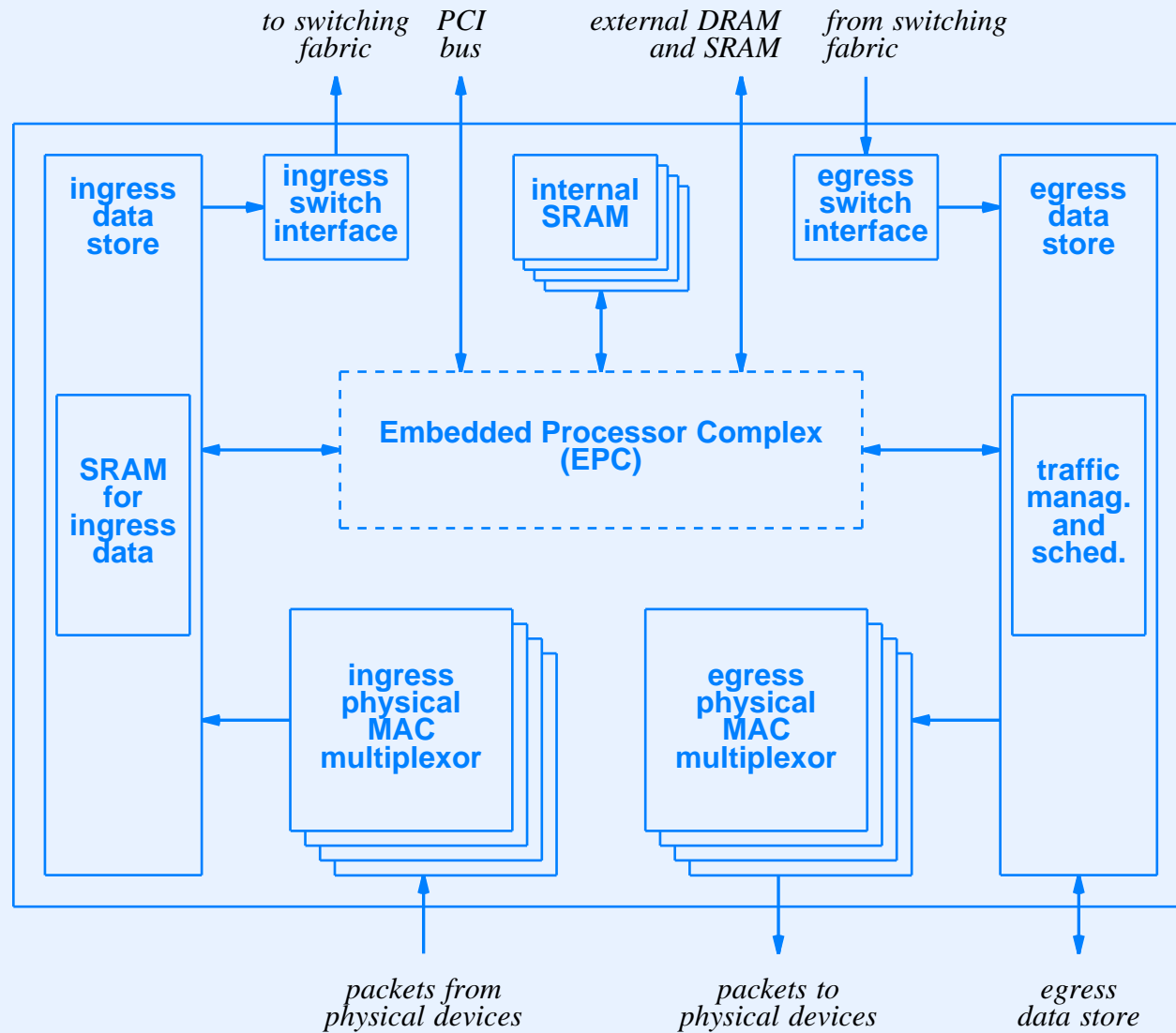
<b>Processor Type</b>	<b>Optimized For</b>
<b>TOPparse</b>	<b>Header field extraction and classification</b>
<b>TOPsearch</b>	<b>Table lookup</b>
<b>TOPresolve</b>	<b>Queue management and forwarding</b>
<b>TOPmodify</b>	<b>Packet header and content modification</b>



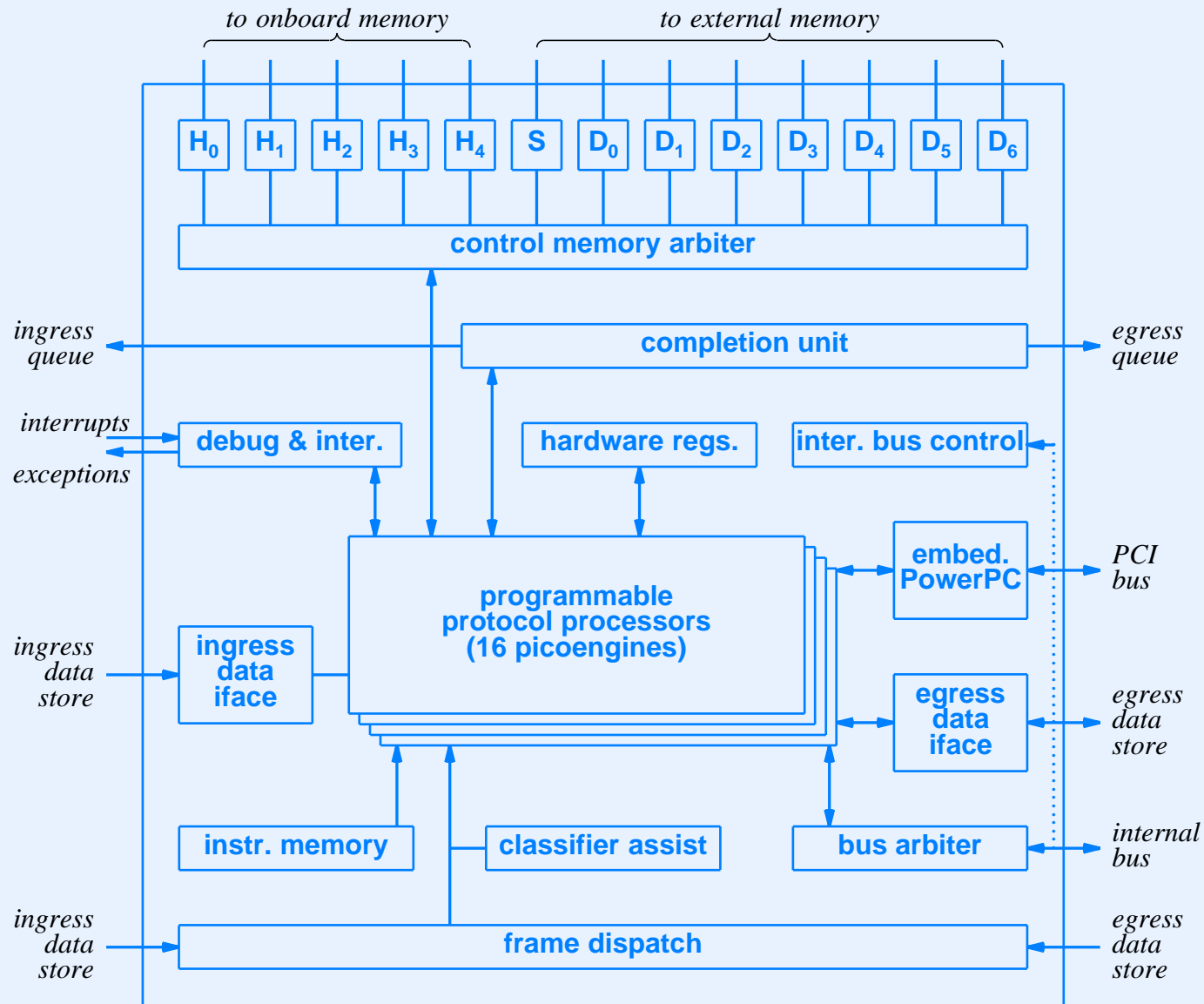
# Extensive And Diverse Processors (Hifn, formerly IBM)

- Multiple processor types
- Extensive use of parallelism
- Separate ingress and egress processing paths
- Multiple onboard data stores
- Model is *NP4GS3*

# Hifn NP4GS3 Architecture



# Hifn's Embedded Processor Complex



# Packet Engines

- Found in Embedded Processor Complex
- Programmable
- Handle many packet processing tasks
- Operate in parallel (sixteen)
- Known as *picoengines*

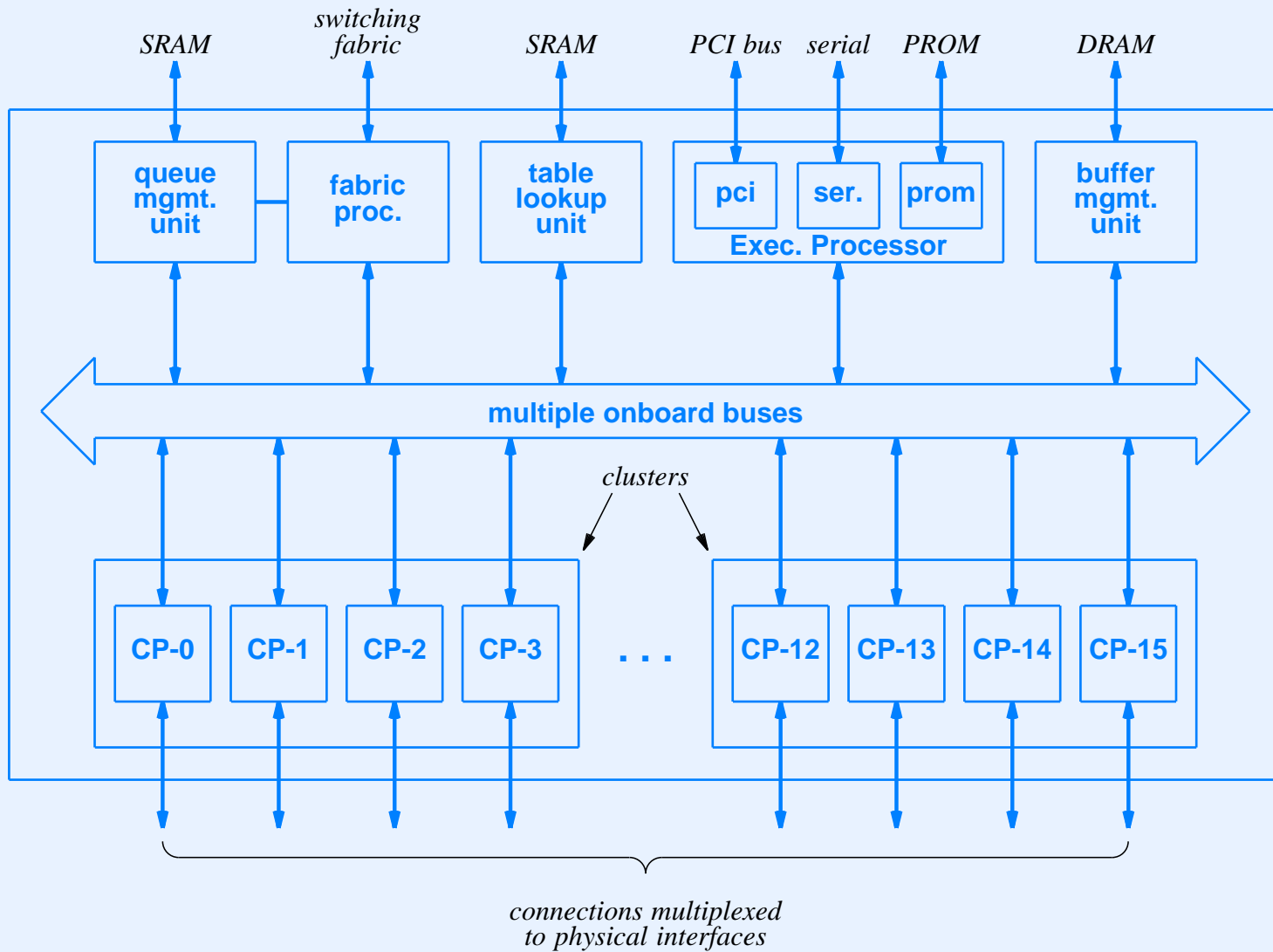
# Other Processors On The IBM Chip

<b>Coprocessor</b>	<b>Purpose</b>
<b>Data Store</b>	<b>Provides frame buffer DMA</b>
<b>Checksum</b>	<b>Calculates or verifies header checksums</b>
<b>Enqueue</b>	<b>Passes outgoing frames to switch or target queues</b>
<b>Interface</b>	<b>Provides access to internal registers and memory</b>
<b>String Copy</b>	<b>Transfers internal bulk data at high speed</b>
<b>Counter</b>	<b>Updates counters used in protocol processing</b>
<b>Policy</b>	<b>Manages traffic</b>
<b>Semaphore</b>	<b>Coordinates and synchronizes threads</b>

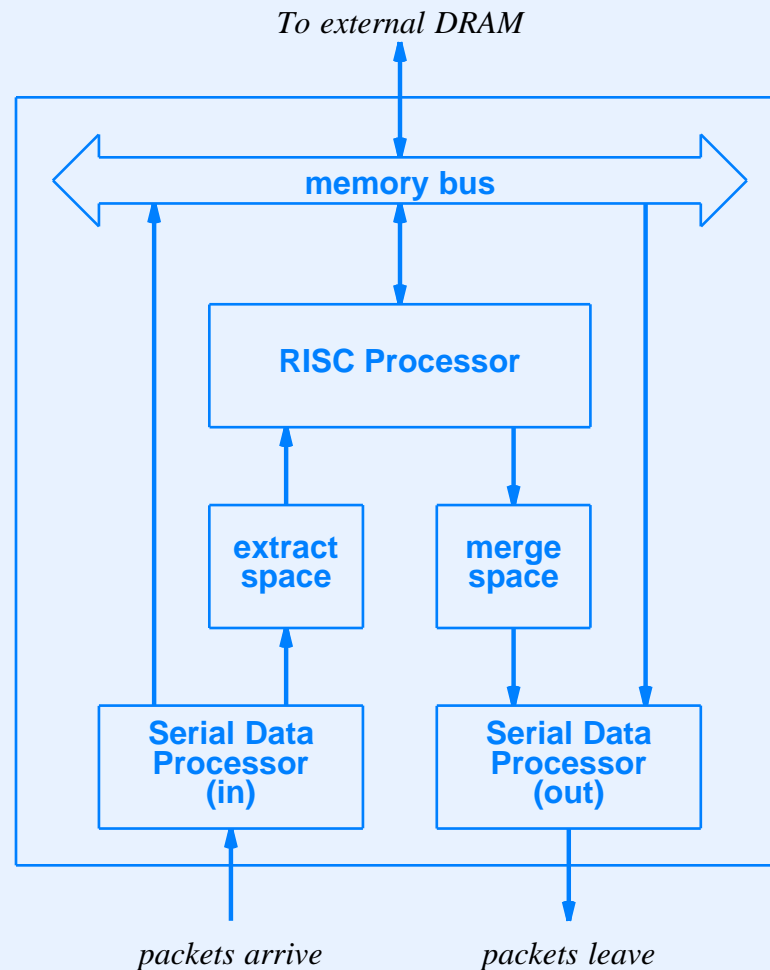
# Flexible RISC Plus Coprocessors (Motorola C-PORT)

- Onboard processors can be
  - Dedicated
  - Parallel clusters
  - Pipeline

# C-Port Architecture



# Internal Structure Of A C-Port Channel Processor



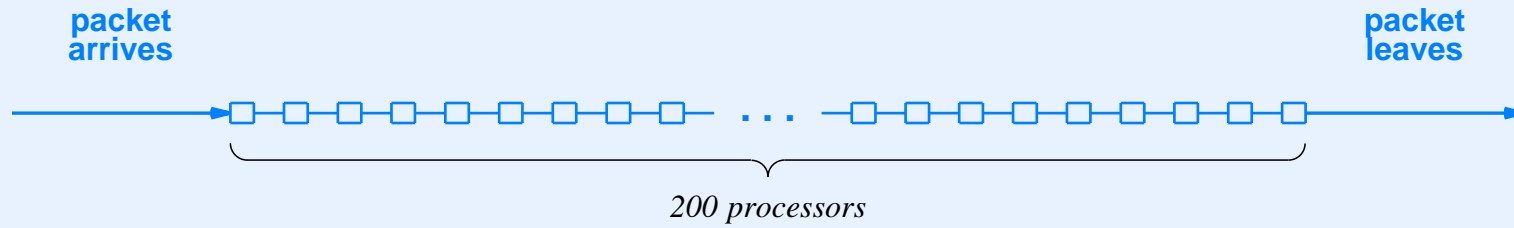
- Actually a processor complex



# Extremely Long Pipeline (Xelerated)

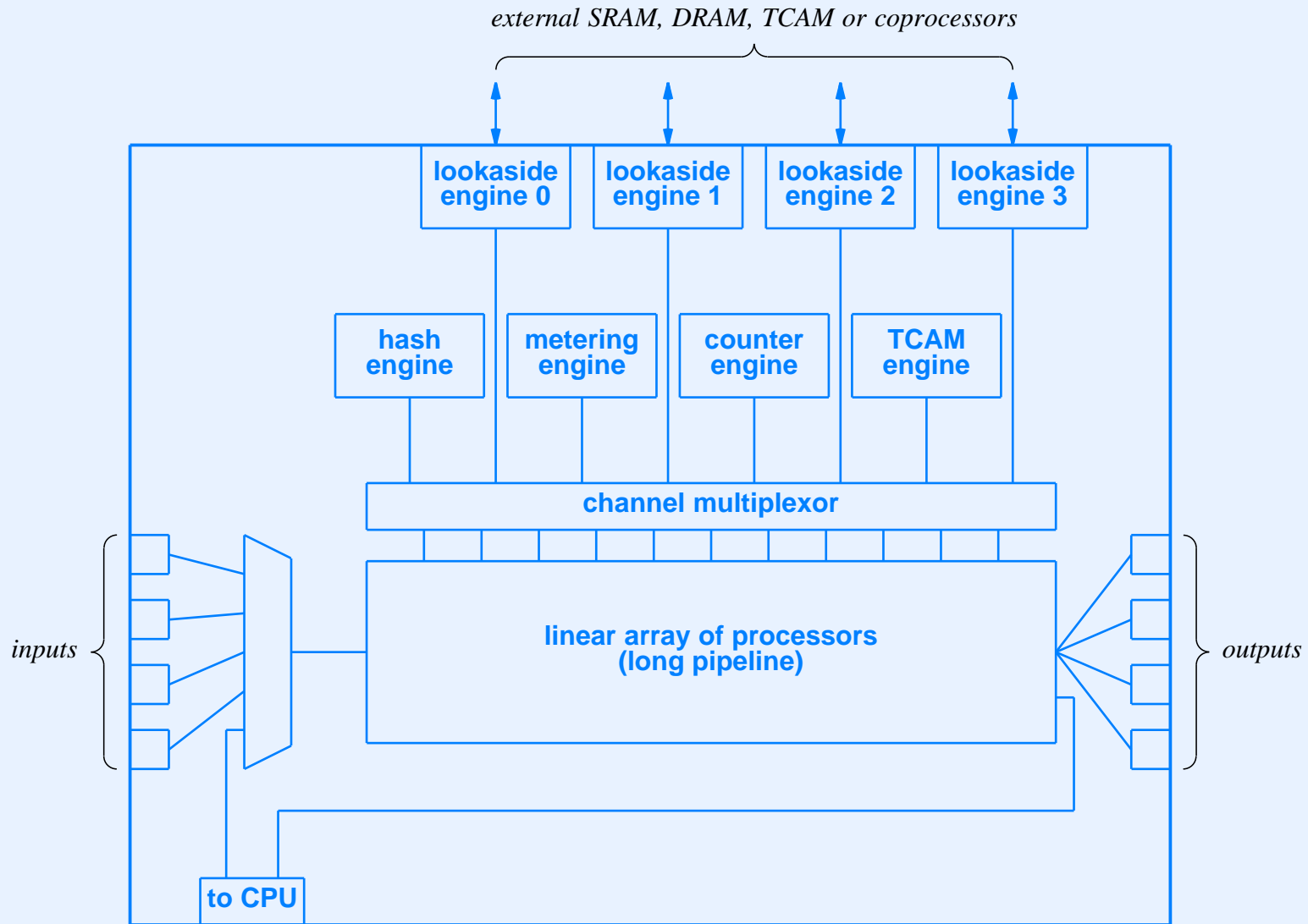
- Pipeline contains 200 processors
- Each processor can execute four instructions per packet
- External coprocessor calls used to pass state

# Xelerated Architecture



- Pipeline has 200 stages

# Xelerated Internal Architecture



# Summary

- Many network processor architecture variations
- Examples include
  - Augmented RISC processor
  - Embedded parallel processors plus coprocessors
  - Parallel pipelines of homogeneous processors
  - Pipeline of parallel heterogeneous processors
  - Extensive and diverse processors
  - Flexible RISC plus coprocessors
  - Extremely long pipeline



**Questions?**

# **XVII**

## **Design Tradeoffs And Consequences**

# Low Development Cost Vs. Performance

- The fundamental economic motivation
- ASIC costs \$1M to develop
- Network processor costs programmer time

# Programmability Vs. Processing Speed

- Programmable hardware is slower
- Flexibility costs...



# Speed Vs. Functionality

- Generic idea:
  - Processor with most functionality is slowest
  - Adding functionality to NP lowers its overall “speed”

# Speed

- Difficult to define
- Can include
  - Packet Rate
  - Data Rate
  - Burst size

# Per-Interface Rates Vs. Aggregate Rates

- Per-interface rate important if
  - Physical connections form bottleneck
  - System scales by having faster interfaces
- Aggregate rate important if
  - Fabric forms bottleneck
  - System scales by having more interfaces

# Increasing Processing Speed Vs. Increasing Bandwidth

*Will network processor capabilities or the bandwidth of network connections increase more rapidly?*

- What is the effect of more transistors?
- Does Moore's Law apply to bandwidth?

# Lookaside Coprocessors Vs. Flow-Through Coprocessors

- Flow-through pipeline
  - Operates at wire speed
  - Difficult to change
- Lookaside
  - Modular and easy to change
  - Invocation can be bottleneck

# Uniform Pipeline Vs. Synchronized Pipeline

- Uniform pipeline
  - Operates in lock-step like assembly line
  - Each stage must finish in exactly the same time
- Synchronized pipeline
  - Buffers allow computation at each stage to differ
  - Synchronization expensive

# Explicit Parallelism Vs. Cost And Programmability

- Explicit parallelism
  - Hardware is less complex
  - More difficult to program
- Implicit parallelism
  - Easier to program
  - Slightly lower performance

# Parallelism Vs. Strict Packet Ordering

- Increased parallelism
  - Improves performance
  - Results in out-of-order packets
- Strict packet ordering
  - Aids protocols such as TCP
  - Can nullify use of parallelism



# Stateful Classification Vs. High-Speed Parallel Classification

- Static classification
  - Keeps no state
  - Is the fastest
- Dynamic classification
  - Keeps state
  - Requires synchronization for updates

# Memory Speed Vs. Programmability

- Separate memory *banks*
  - Allow parallel accesses
  - Yield high performance
  - Difficult to program
- Non-banked memory
  - Easier to program
  - Lower performance

# I/O Performance Vs. Pin Count

- Bus width
  - Increase to produce higher throughput
  - Decrease to take fewer pins

# Programming Languages

- A three-way tradeoff
- Can have two, but not three of
  - Ease of programming
  - Functionality
  - Performance

# Programming Languages That Offer High Functionality

- Ease of programming vs. speed
  - High-level language offers ease of programming, but lower performance
  - Low-level language offers higher performance, but makes programming more difficult

# Programming Languages That Offer Ease Of Programming

- Speed vs. functionality
  - For restricted language, compiler can generate optimized code
  - Broad functionality and ease of programming lead to inefficient code

# Programming Languages That Offer High Performance

- Ease of programming vs. functionality
  - Optimizing compiler and ease of programming imply a restricted application
  - Optimizing code for general applications requires more programmer effort

# **Multithreading: Throughput Vs. Ease Of Programming**

- Multiple threads of control can increase throughput
- Planning the operation of threads that exhibit less contention requires more programmer effort



# Traffic Management Vs. High-Speed Forwarding

- Traffic management
  - Can manage traffic on multiple, independent flows
  - Requires extra processing
- Blind forwarding
  - Performed at highest speed
  - Does not distinguish among flows

# Generality Vs. Specific Architectural Role

- General-purpose network processor
  - Used in any part of any system
  - Used with any protocol
  - More expensive
- Special-purpose network processor
  - Restricted to one role / protocol
  - Less expensive, but may need many types

# Special-Purpose Memory Vs. General-Purpose Memory

- General-purpose memory
  - Single type of memory serves all needs
  - May not be optimal for any use
- Special-purpose memory
  - Optimized for one use
  - May require multiple memory types

# Backward Compatibility Vs. Architectural Advances

- Backward compatibility
  - Keeps same instruction set through multiple versions
  - May not provide maximal performance
- Architectural advances
  - Allows more optimizations
  - Difficult for programmers

# Parallelism Vs. Pipelining

- Both are fundamental performance techniques
- Usually used in combination: pipeline of parallel processors
  - How long is pipeline?
  - How much parallelism at each stage?

# Summary

- Many design tradeoffs
- No easy answers



**Questions?**



**STOP**