Network Systems Design (CS490N)

Douglas Comer

Computer Science Department
Purdue University
West Lafayette, IN 47907

http://www.cs.purdue.edu/people/comer

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 implementation
  - Special languages
- 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
- Classification languages and programs

Course Administration

- Textbook
  - D. Comer, *Network Systems Design Using Network Processors*, Prentice Hall, 2003.
- Grade
  - Quizzes 5%
  - Midterm and final exam 35%
  - Programming projects 60%

Lab Facilities Available

- Extensive network processor testbed facilities (more than any other university)
- Donations from
  - Agere Systems
  - IBM
  - 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

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
(2003 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 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*

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

Standard Name	Bit Rate	Voice Circuits
–	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

Standard Name	Bit Rate	Voice Circuits
–	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

- 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 NAPT — Network Address and Protocol Translator)
  - Smart Web load balancer
  - Set-top box

Example Network Systems (continued)

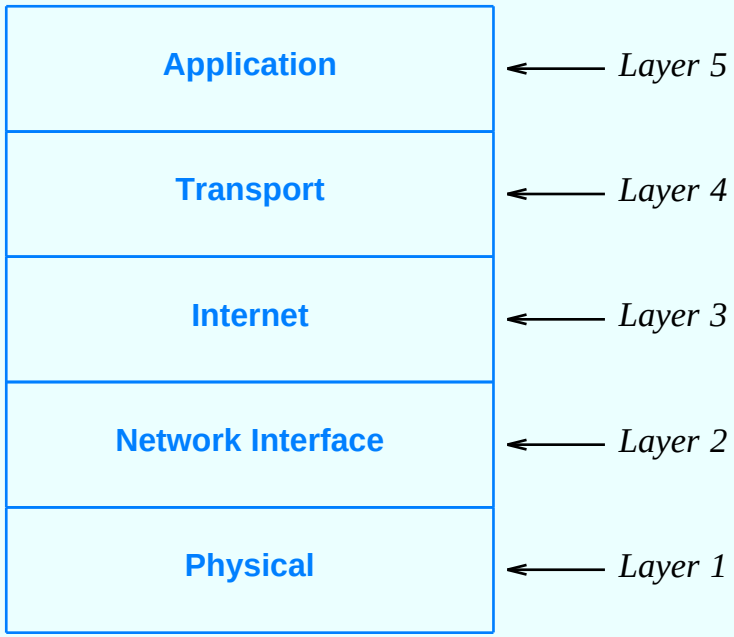
- Network control systems
  - Packet / flow analyzer
  - Traffic monitor
  - Traffic policer
  - Traffic shaper


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

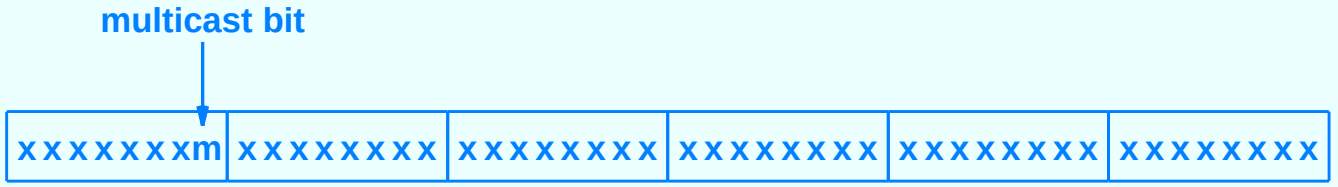
- Two protocols are important
  - Ethernet
  - ATM
- We will concentrate on Ethernet

Ethernet Addressing

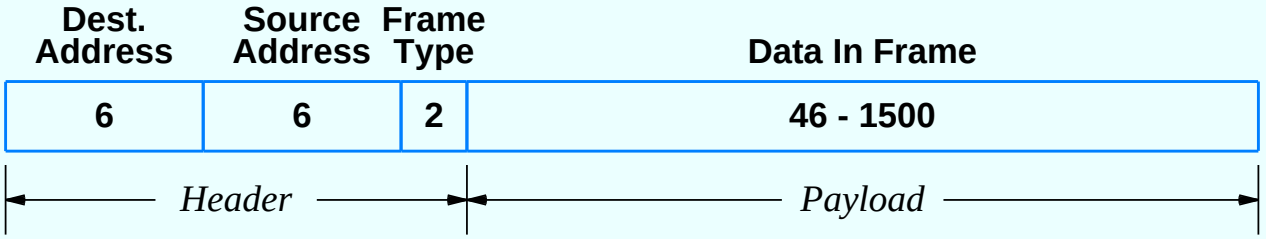
- 48-bit addressing
- Unique address assigned to each station (NIC)
- Destination address in each packet can specify delivery to
  - A single computer (unicast)
  - All computers in broadcast domain (broadcast)
  - Some computers in broadcast domain (multicast)

Ethernet Addressing
(continued)

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



Ethernet Frame Processing



- Dedicated physical layer hardware
  - Checks and removes preamble and CRC on input
  - Computes and appends CRC and preamble on output
- Layer 2 systems use source, destination and (possibly) type fields

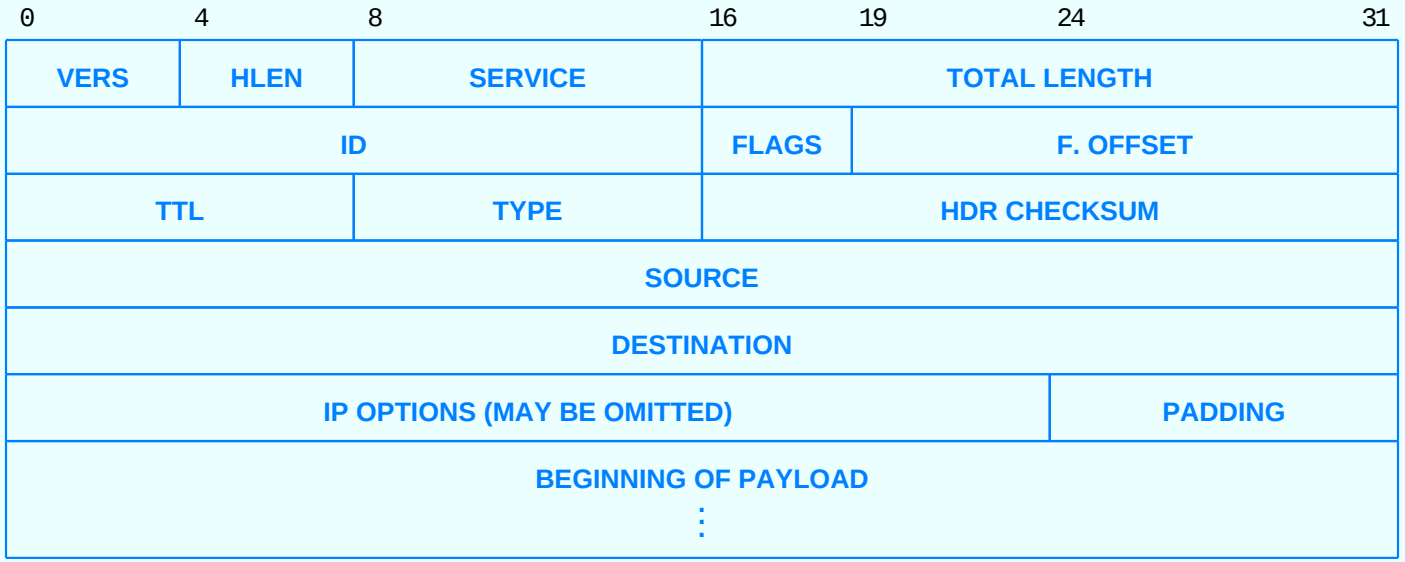
Internet

- 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 address mask to specify boundary between prefix and suffix

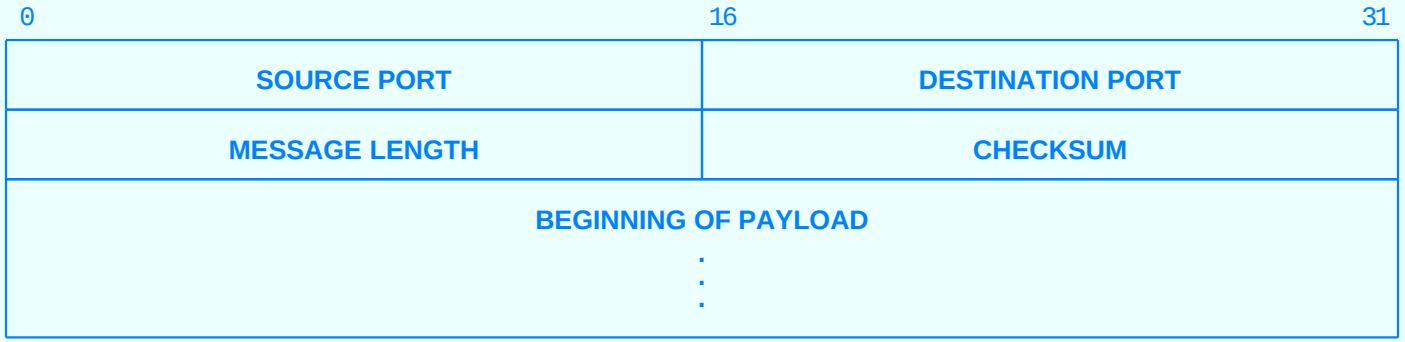
Next-Hop Forwarding

- 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

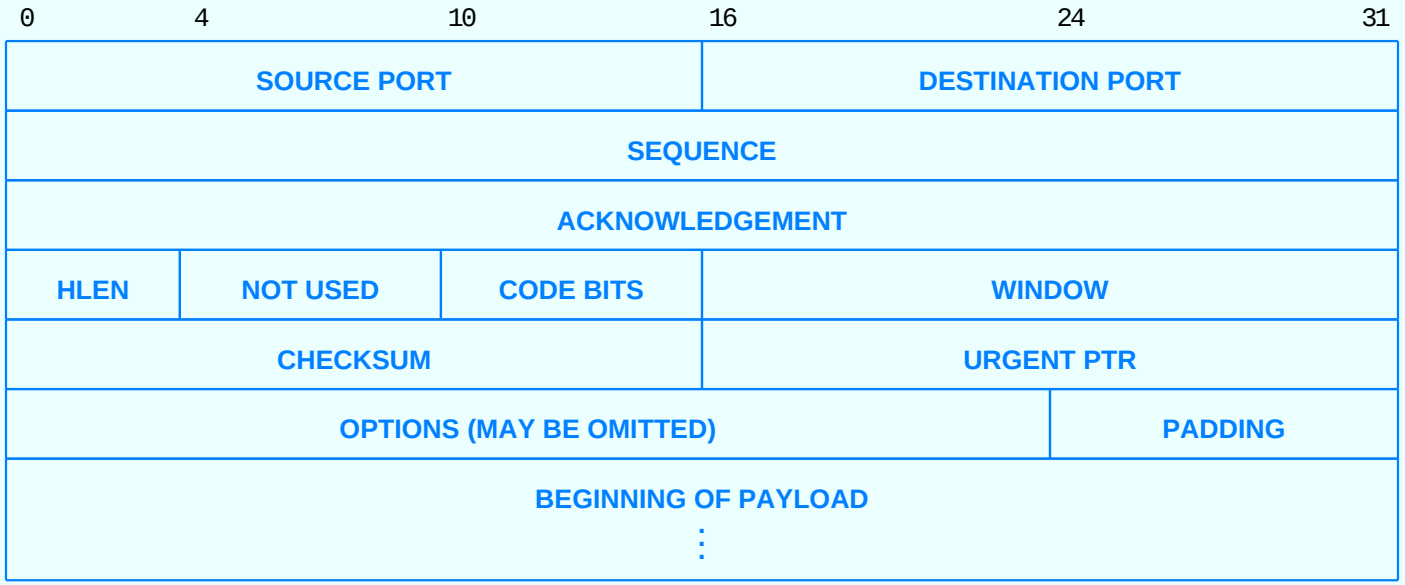
- Routing table
  - Found in both hosts and routers
  - Stores (destination, mask, next_hop) tuples
- Route lookup
  - Takes destination address as argument
  - Finds next hop
  - Uses longest-prefix match

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

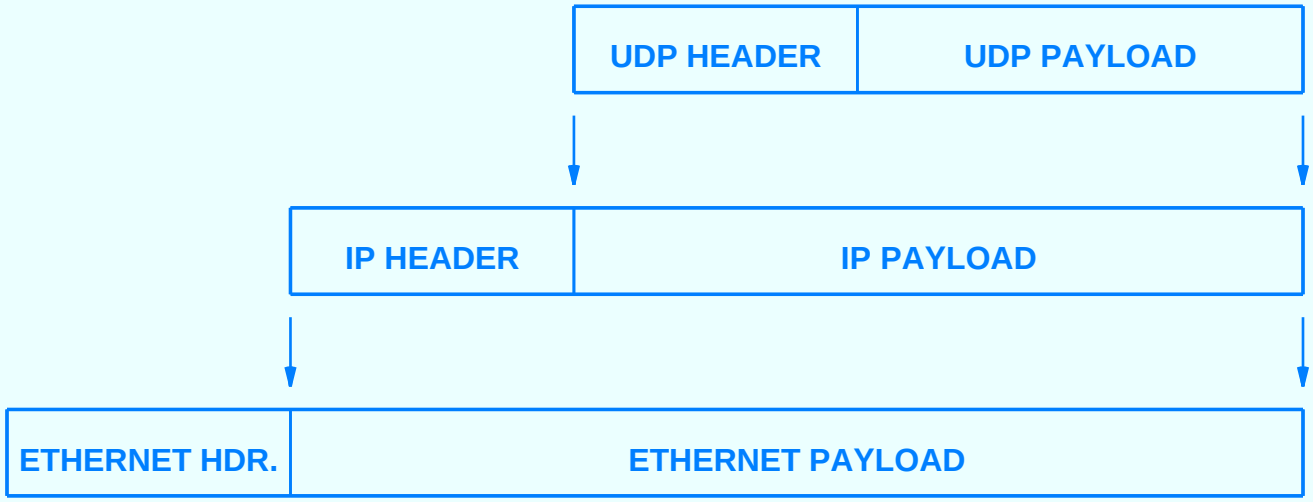


- Sent end-to-end
- Fixed-size fields make parsing efficient

TCP Segment Fields

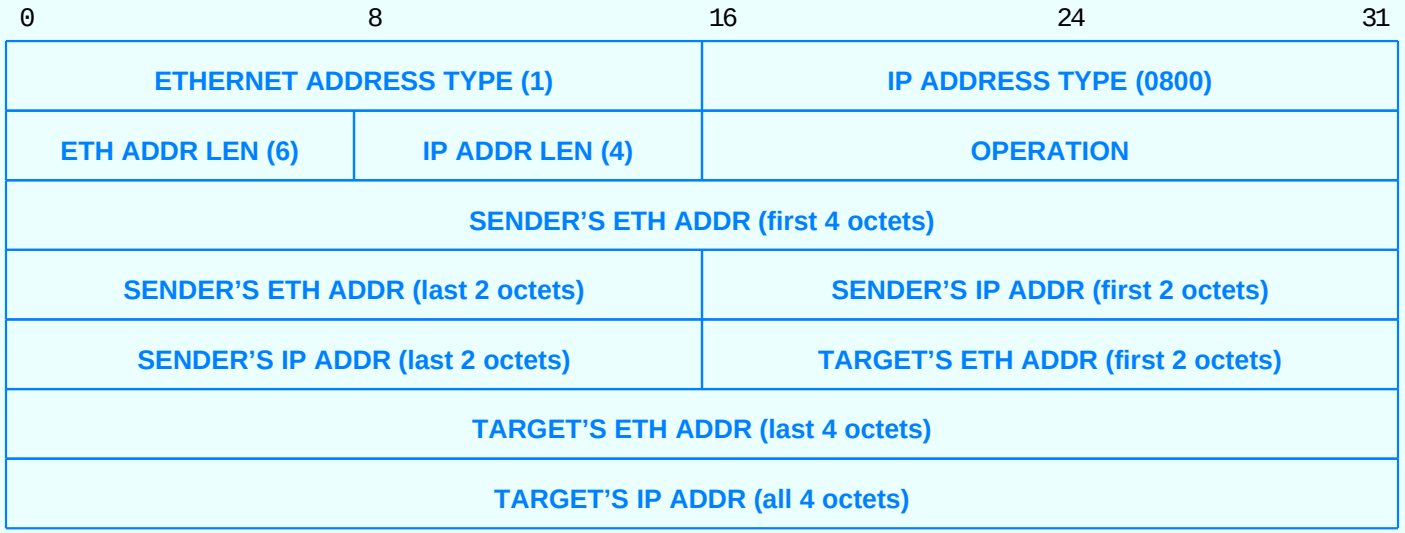
Field	Meaning
SOURCE PORT	ID of sending application
DESTINATION PORT	ID of receiving application
SEQUENCE	Sequence number for data in payload
ACKNOWLEDGEMENT	Acknowledgement of data received
HLEN	Header length measured in 32-bit units
NOT USED	Currently unassigned
CODE BITS	URGENT, ACK, PUSH, RESET, 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



- 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

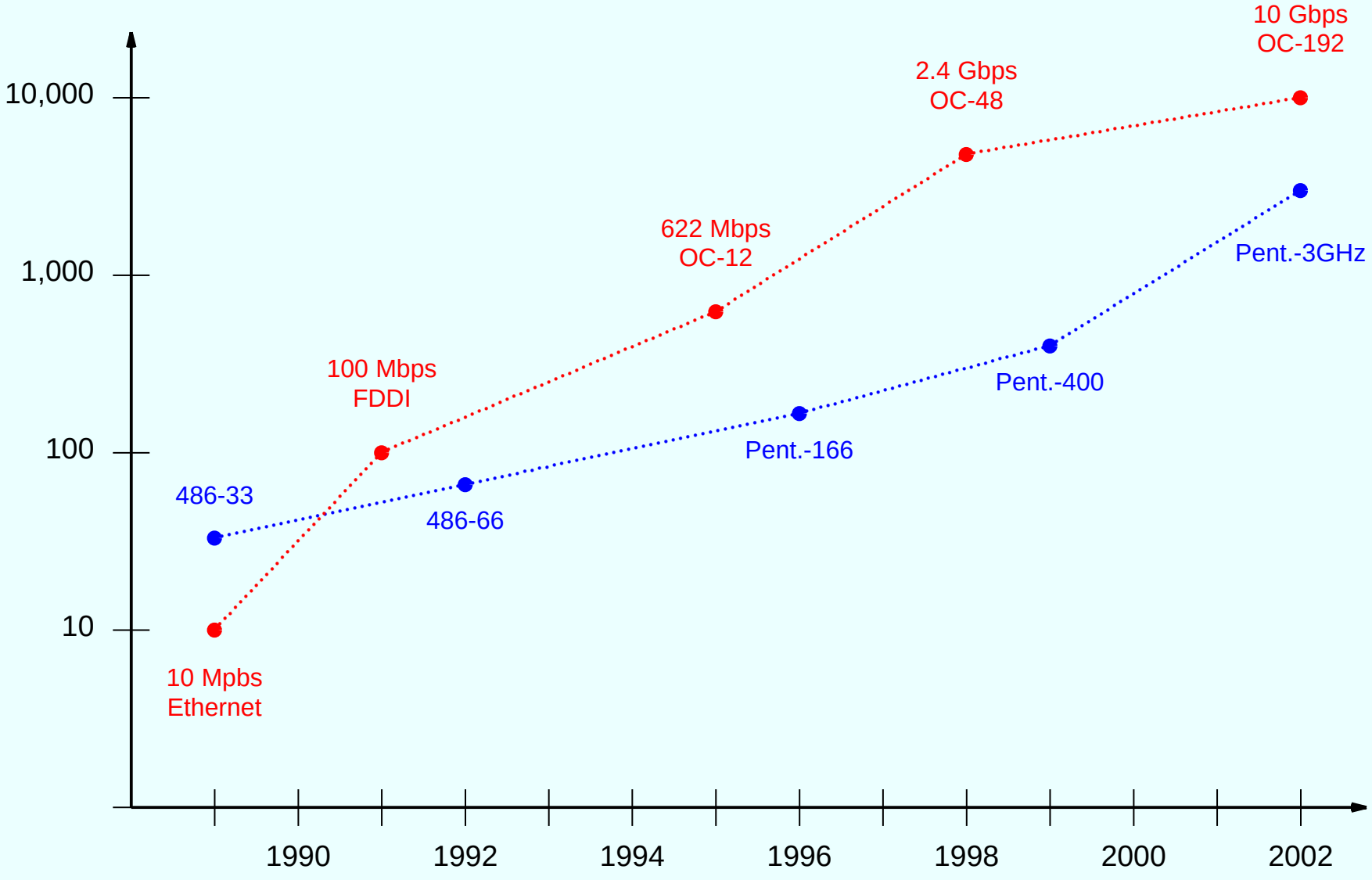
Why Study Protocol Processing On Conventional Hardware? (continued)

- 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

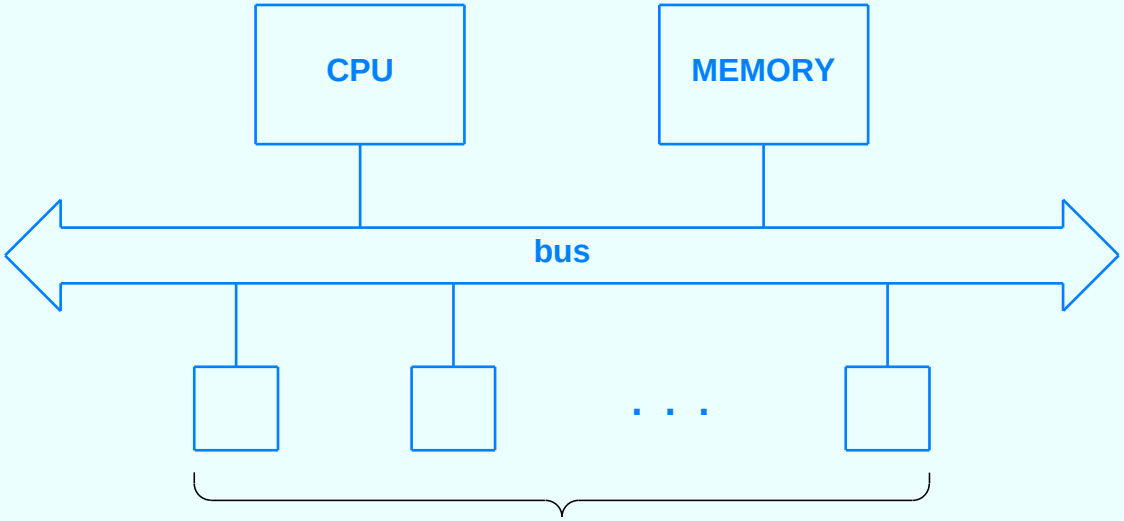
Growth Of Technologies



Conventional Computer Hardware

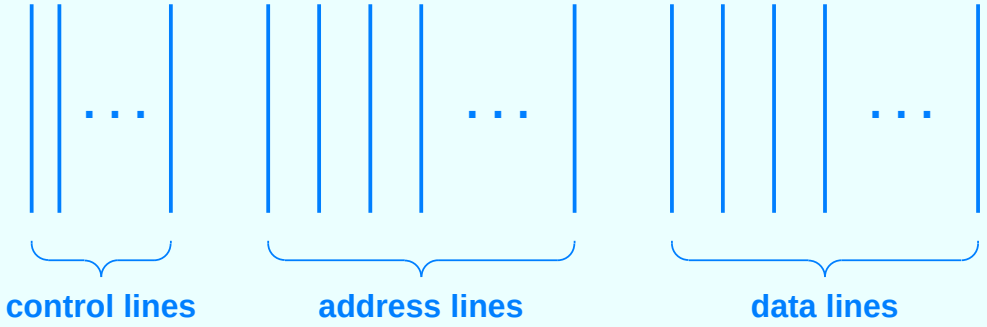
- Four important aspects
  - Processor
  - Memory
  - I/O interfaces
  - One or more buses

Illustration Of Conventional Computer Architecture



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

Bus Organization And Operations



- Parallel wires (K+N+C total)
- Used to pass
  - An address of K bits
  - A data value of N bits (width of the bus)
  - Control information of C bits

Bus Width

- 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
- Wait for device that owns the address to respond
- If successful, extract value (response) from data lines

Store Operation

- Place address of a device on address lines
- Place value on data lines
- Issue *store* on control lines
- Wait for device that owns the address to respond
- If unsuccessful, report error

Example Of Operations Mapped Into Fetch/Store Paradigm

- Imagine disk device attached to a bus
- Assume the hardware can perform 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

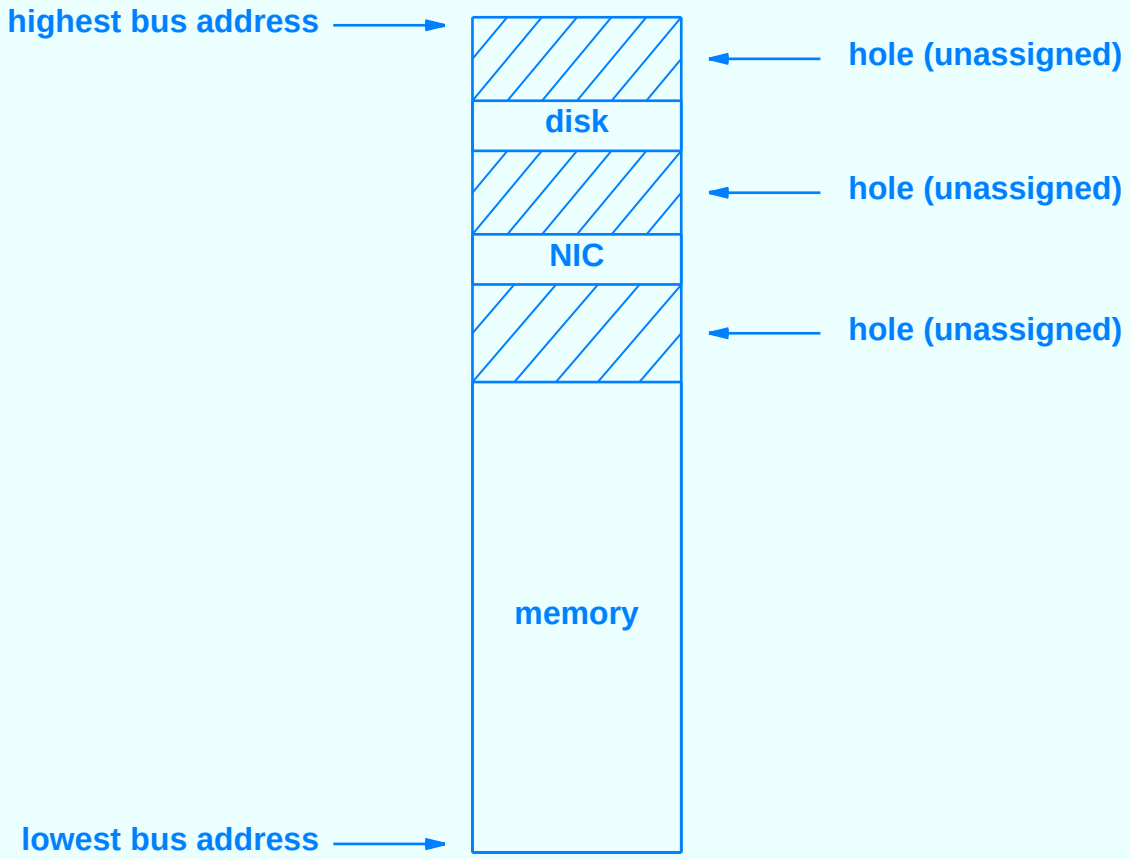
Bus Address Space

- 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

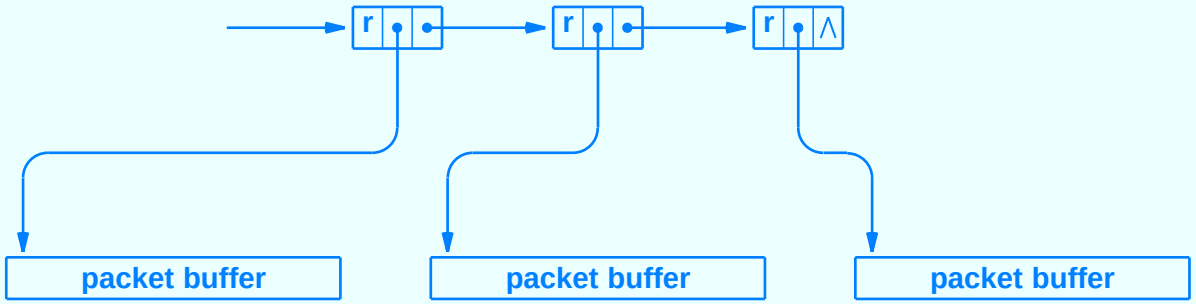
Buffer Chaining

- 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 packet

Operation Chaining

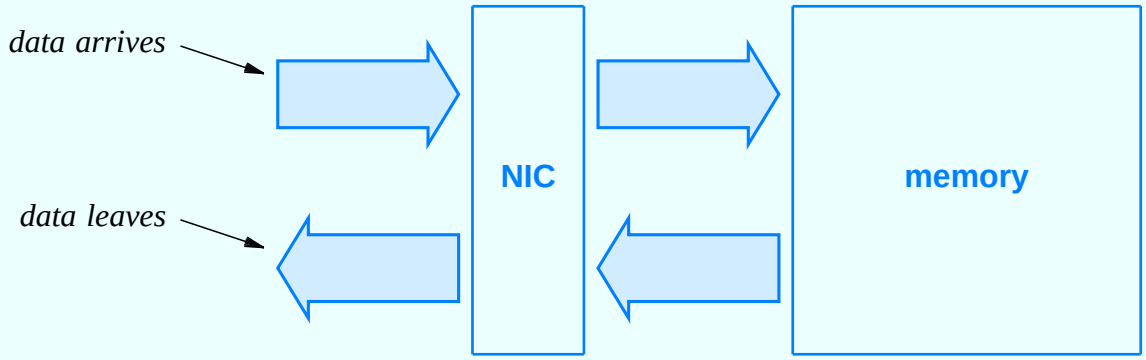
- CPU
  - Allocates multiple buffers
  - Builds linked list of operations
  - Passes list to NIC
- NIC
  - Follows list and performs instructions
  - Interrupts CPU after each operation
- Advantage: multiple operations proceed without CPU intervention

Illustration Of Operation Chaining



- Optimizes movement of data to memory

Data Flow Diagram



- Depicts flow of data through hardware units
- 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

Buffer Allocation

- Possibilities
  - Large, fixed buffers
  - Variable-size buffers
  - Linked list of fixed-size blocks

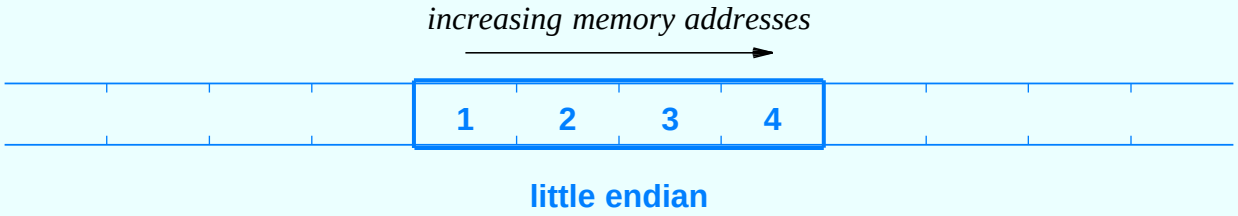
Buffer Addressing

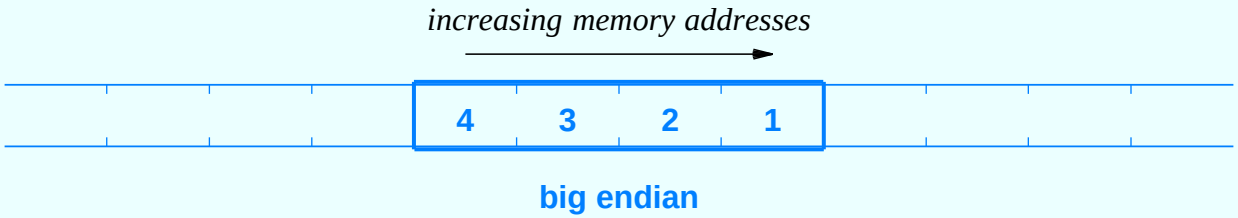
- Buffer address must be resolvable in all contexts
- Easiest implementation: keep buffers in kernel space

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
ntohs	16 bits	Network byte order to host's byte order
htons	16 bits	Host's byte order to network byte order
ntohl	32 bits	Network byte order to host's byte order
htonl	32 bits	Host's byte order to network byte order

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 on packet processing tasks

Why Study These Algorithms?

- Provide insight on packet processing tasks
- Reinforce concepts

Why Study These Algorithms?

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

Why Study These Algorithms?

- Provide insight on 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 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

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

- 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

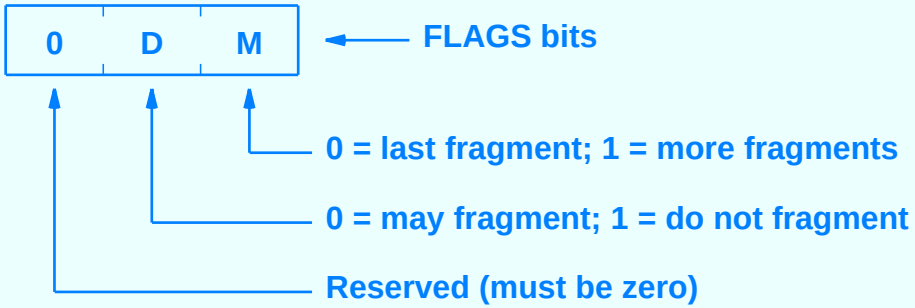
- Values stored in binary
- Interface number is for internal use only
- Zero mask produces *default* route

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;
```

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 $H+N \leq MTU$;
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

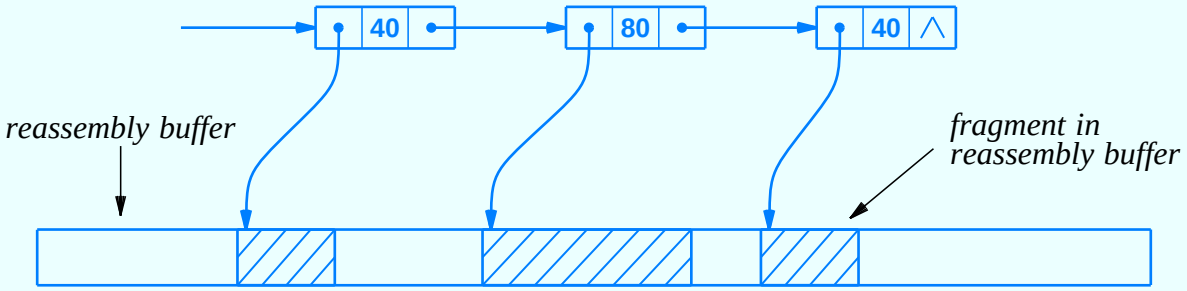
- Complement of fragmentation
- Uses IP SOURCE ADDRESS and IDENTIFICATION fields in datagram header to group related fragments
- Joins fragments to form original datagram

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;

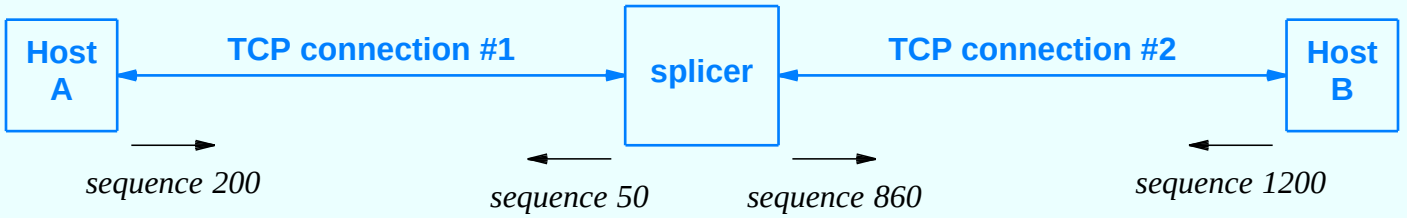
TCP Connection Recognition Algorithm (Part 2)

```
        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

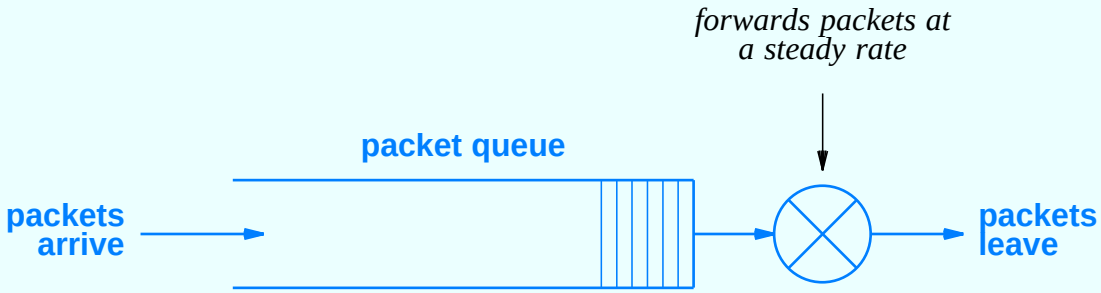
Example Traffic Shaping Mechanisms

- 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()

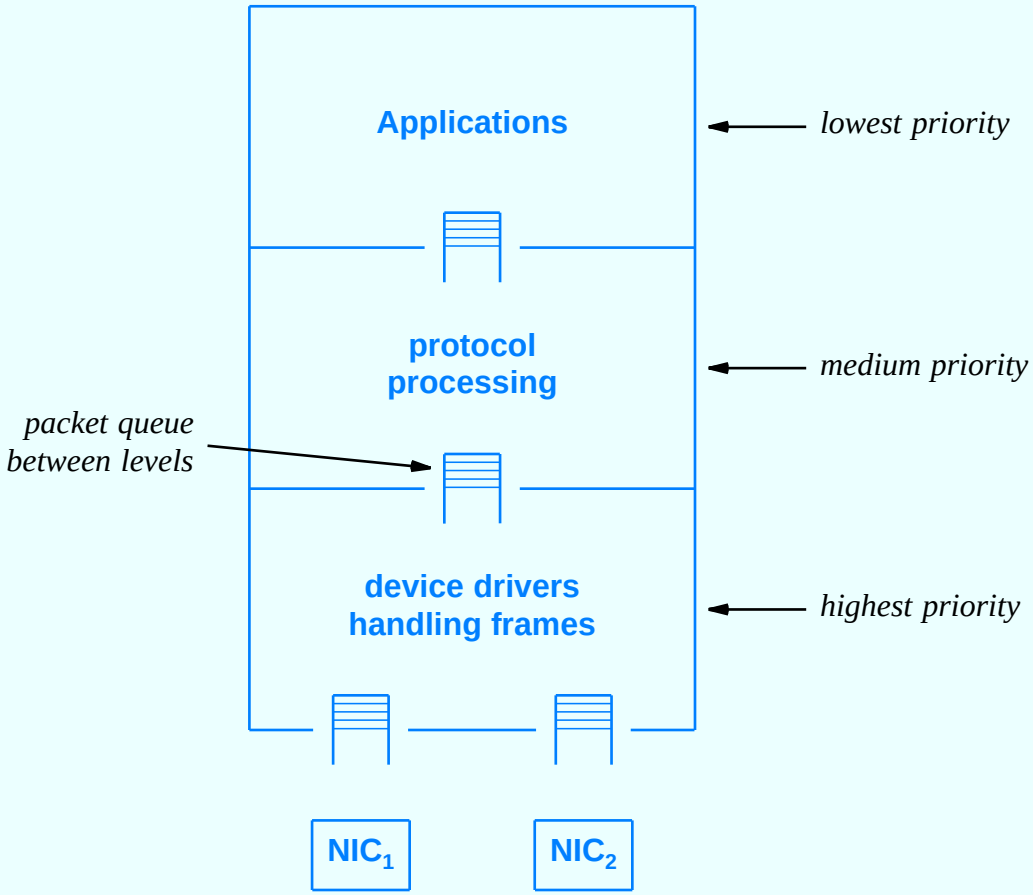
Example Asynchronous API (continued)

- 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

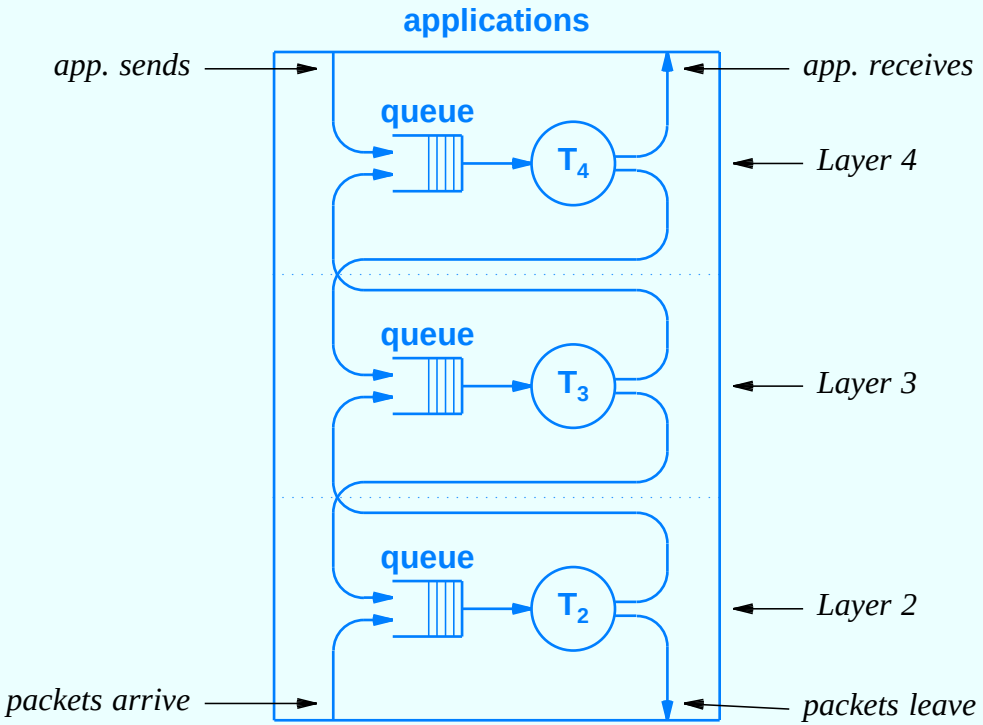
Possible Organization Of Kernel Threads For Layered Protocols

- 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

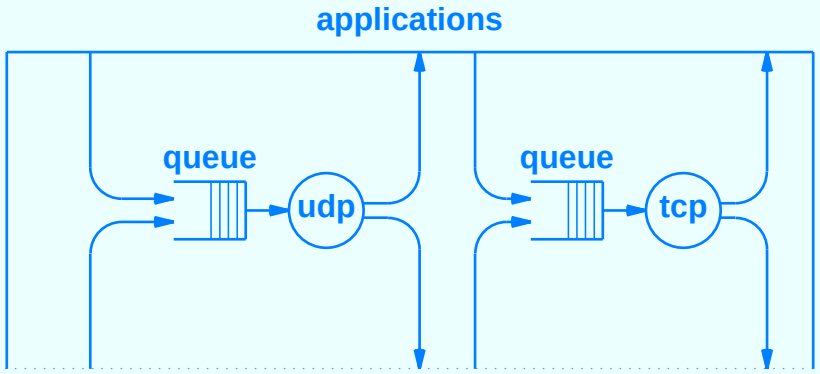
Illustration Of One Thread Per Layer



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

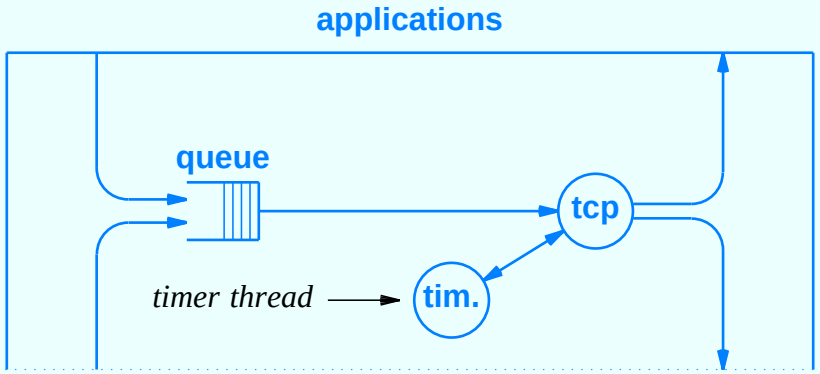


- TCP and UDP reside at same layer
- Separation allows priority

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 on 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

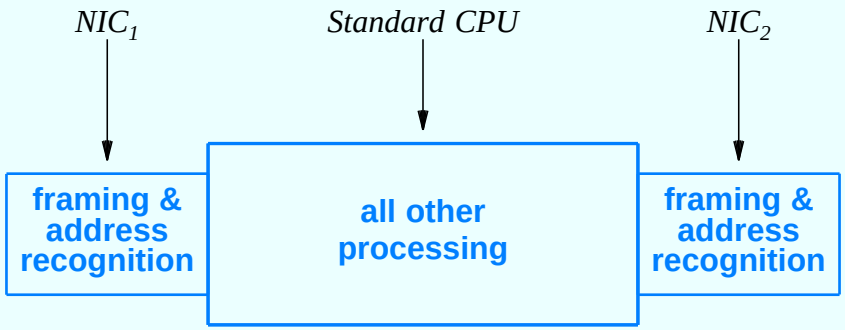
Networking Systems Evolution

- 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)
  - 2003: 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)
  - 2003: 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

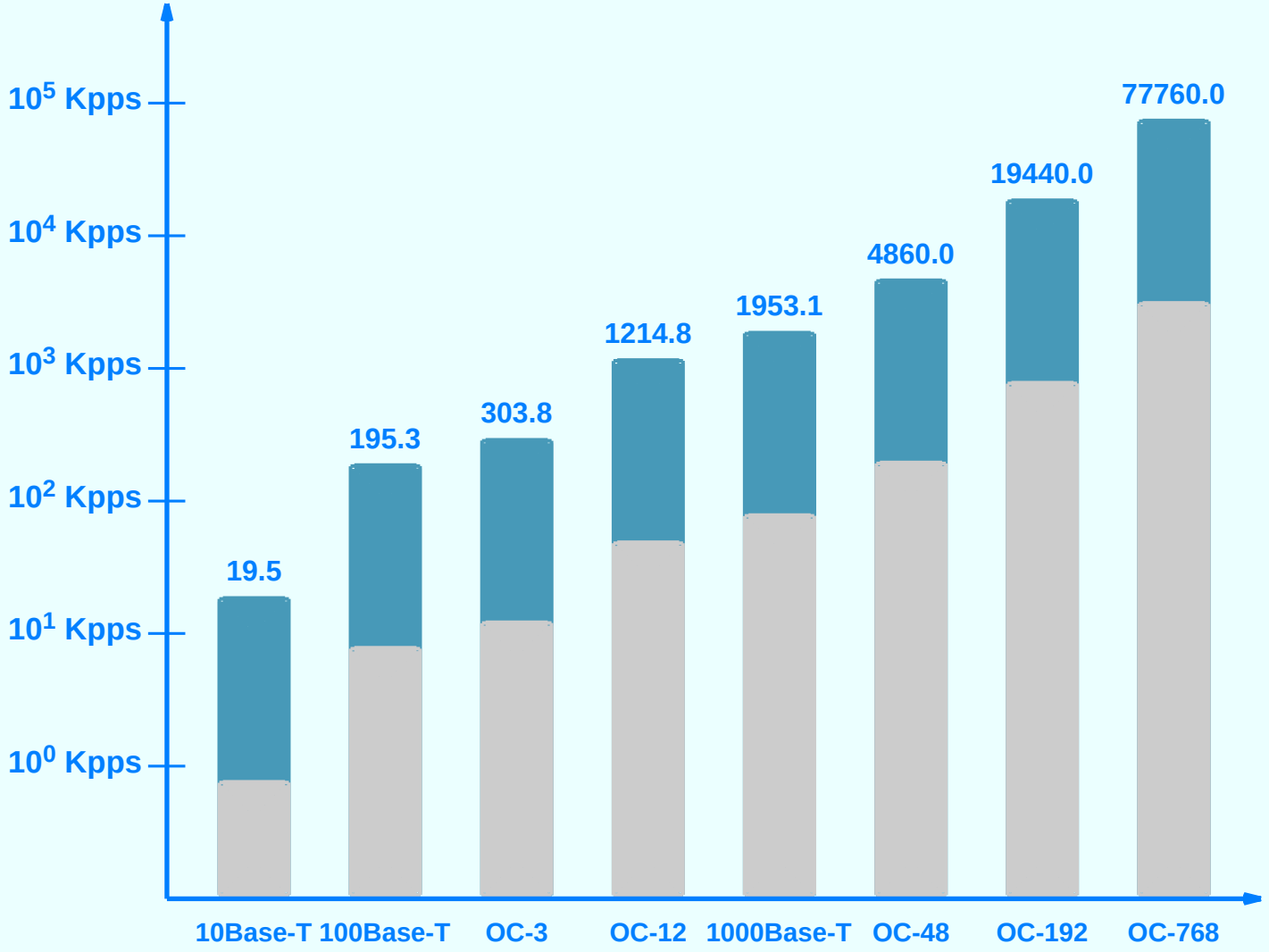
For protocol processing tasks that have a fixed cost per packet, the number of packets processed is more important than the aggregate data rate.

Example Packet Rates

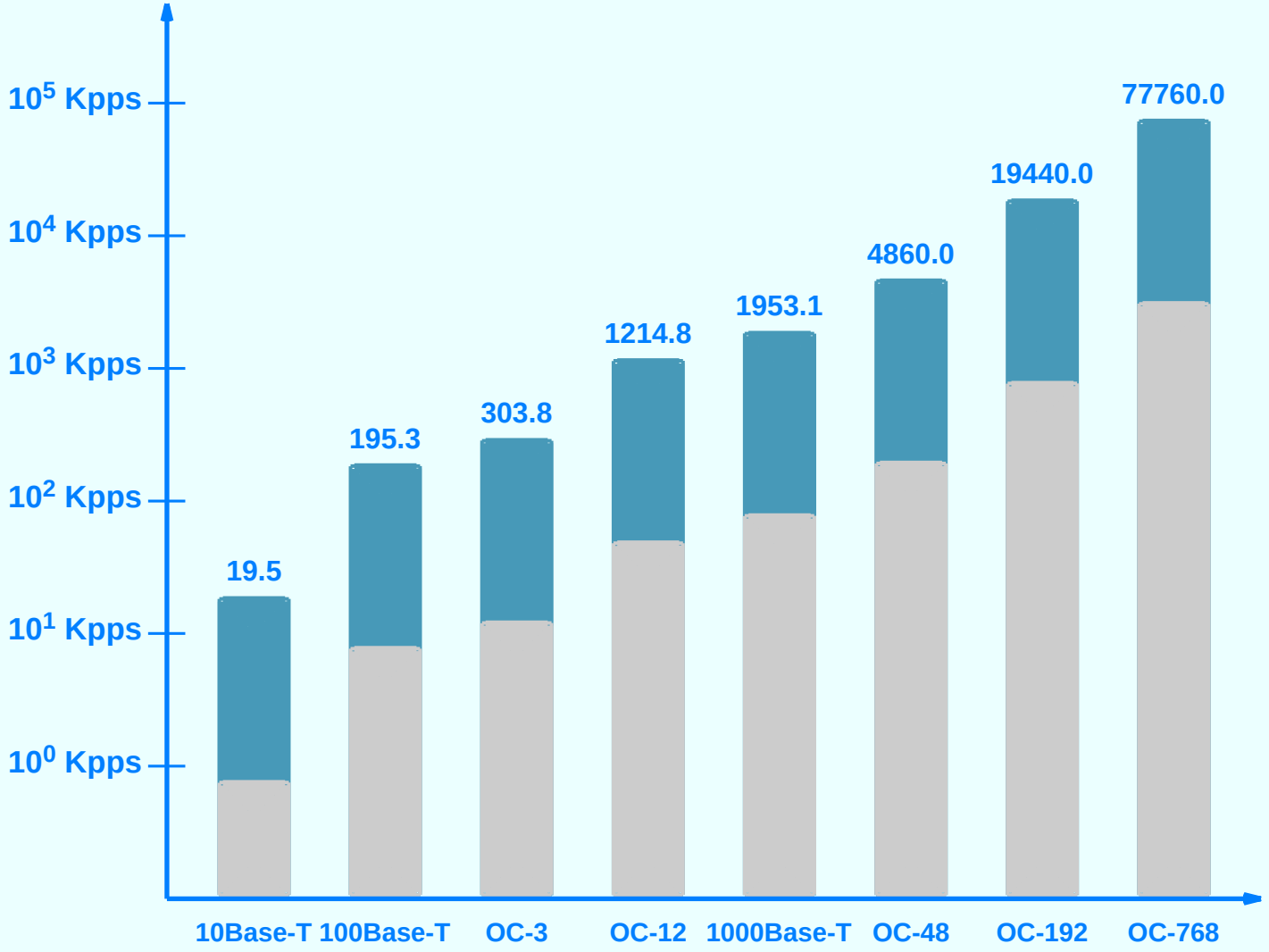
Technology	Network Data Rate In Gbps	Packet Rate For Small Packets In Kpps	Packet Rate For Large Packets In Kpps
10Base-T	0.010	19.5	0.8
100Base-T	0.100	195.3	8.2
OC-3	0.156	303.8	12.8
OC-12	0.622	1,214.8	51.2
1000Base-T	1.000	1,953.1	82.3
OC-48	2.488	4,860.0	204.9
OC-192	9.953	19,440.0	819.6
OC-768	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 μs)	Time per packet for large packets (in μs)
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 much 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)

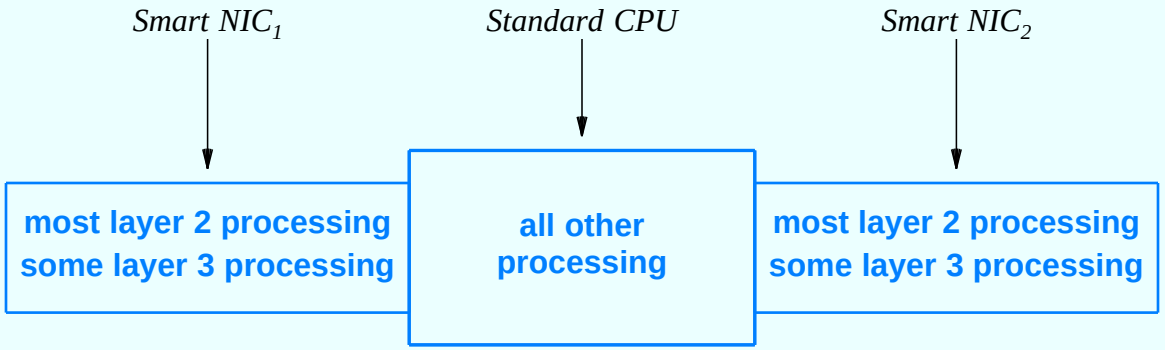
NICs And Onboard Processing

- 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

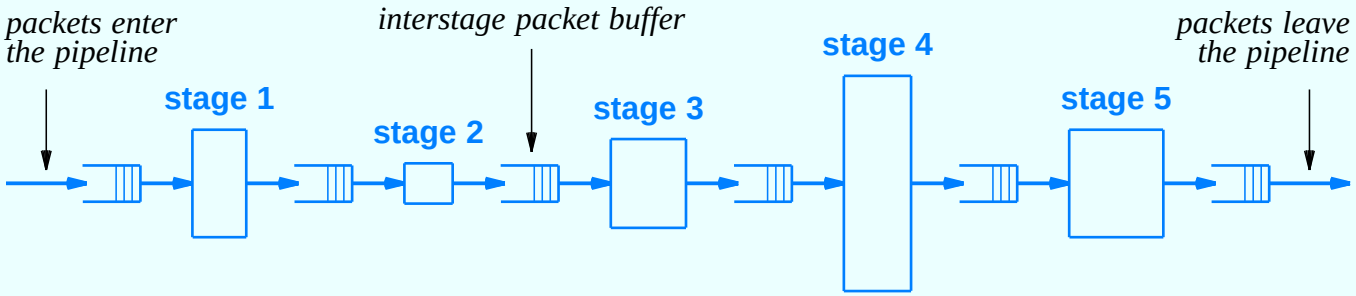
Cell Switching Details

- 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

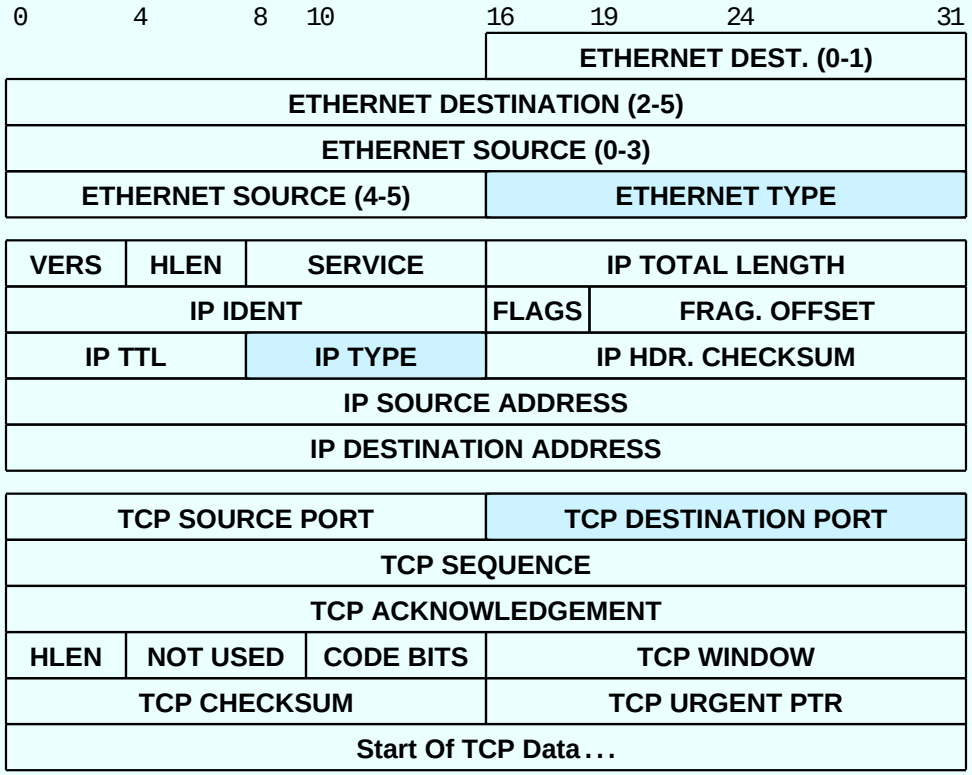
Example Classification

- 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}
  - 2-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

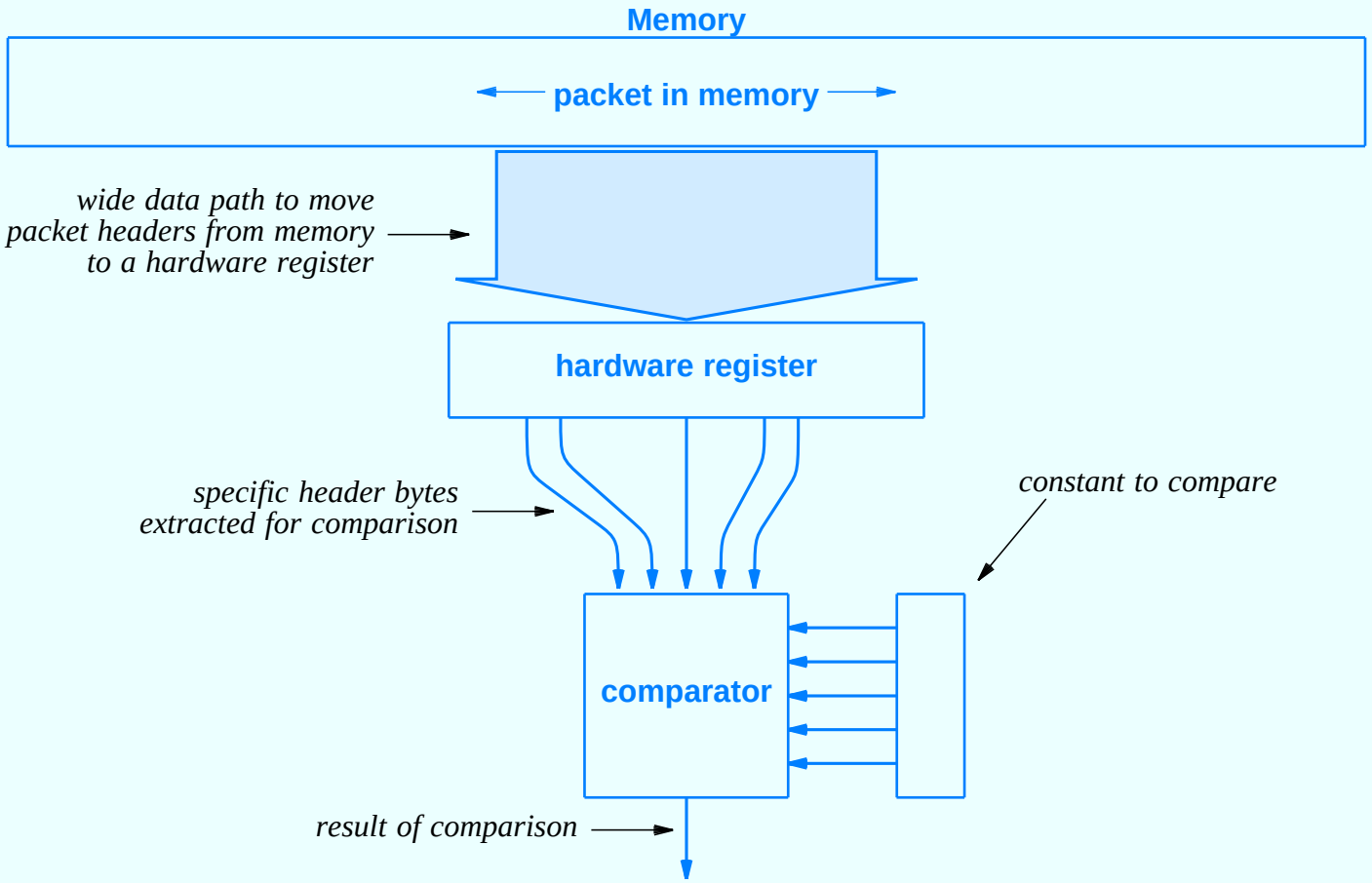
Note About Optimization

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 $08.00.06.01.50_{16}$

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

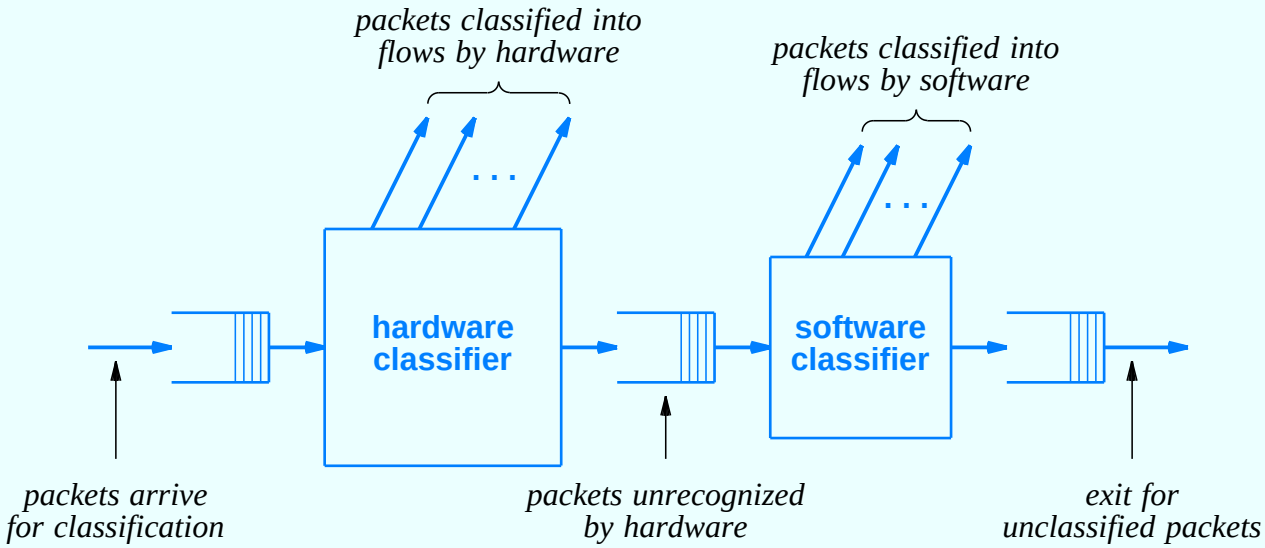
Example Variable-Size Header: IP Options

- 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.

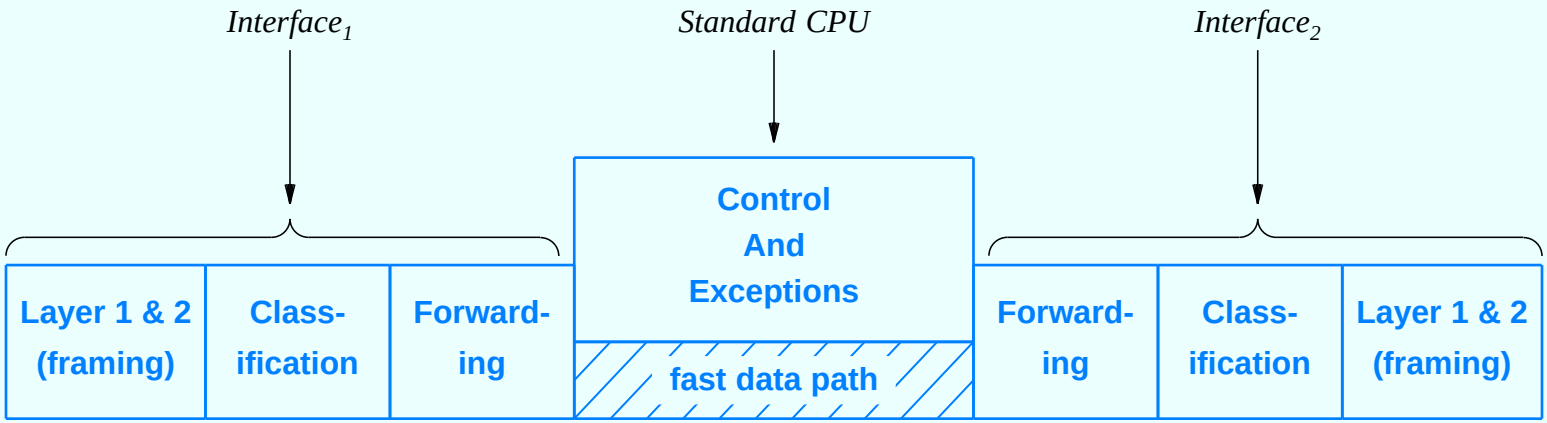
Forwarding In A Connectionless Network

- 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

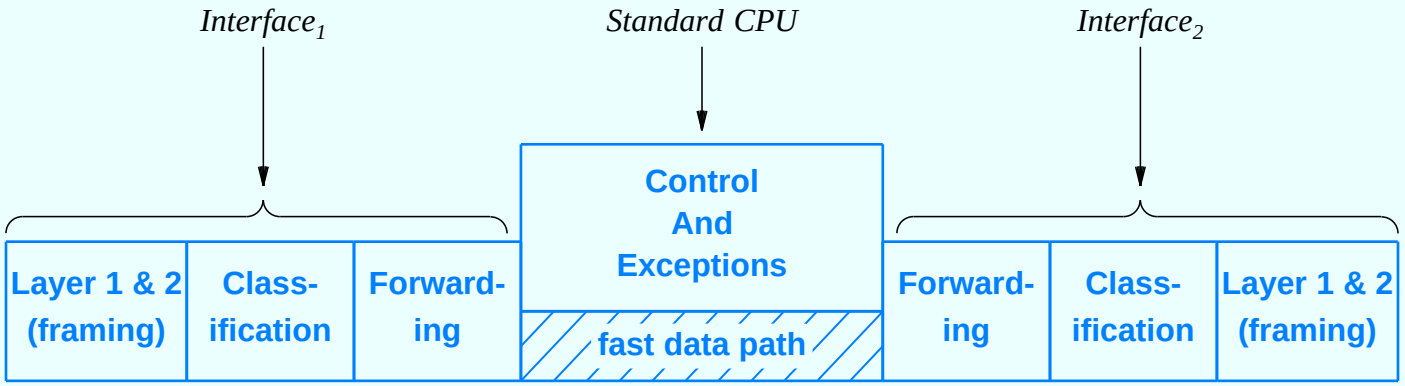
Second Generation Network Systems

- 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

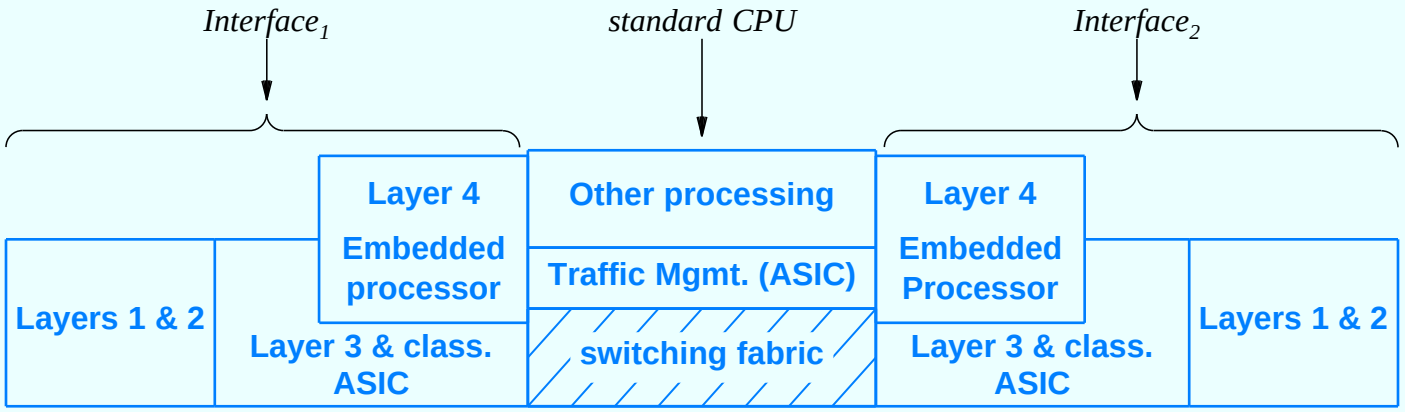
Embedded Processor

- 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

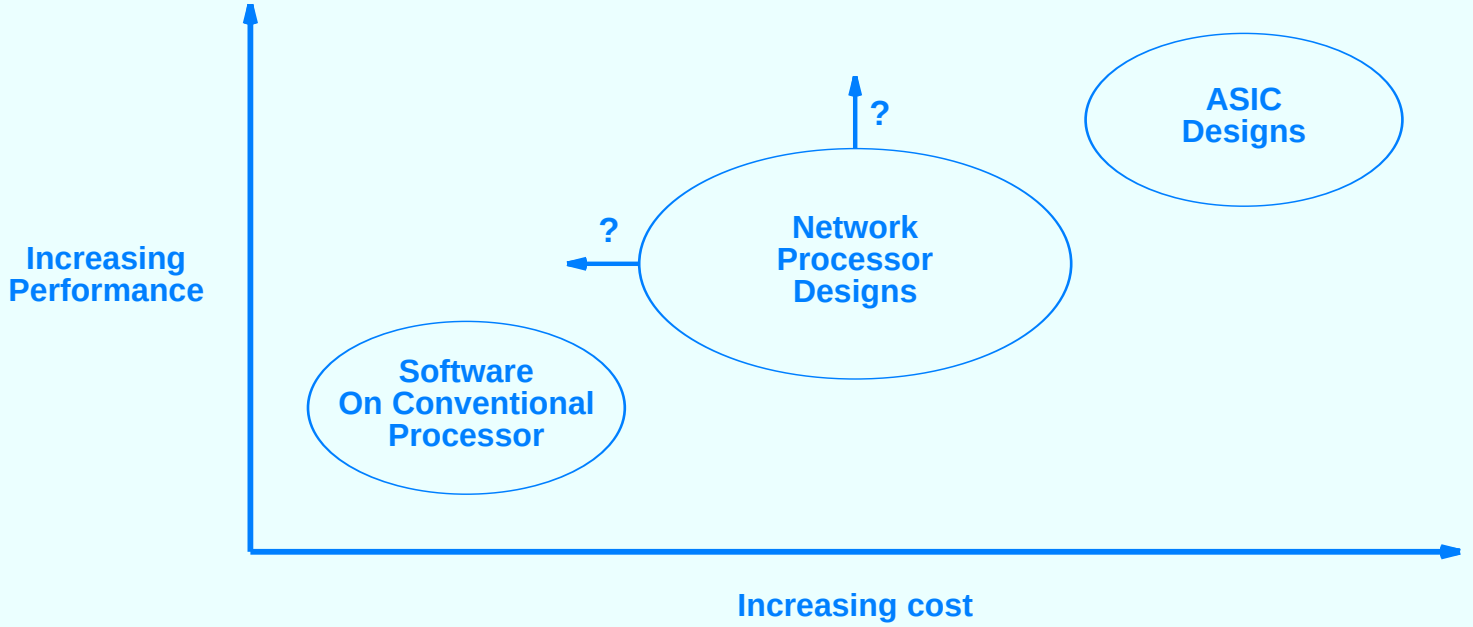
Scalability

- 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

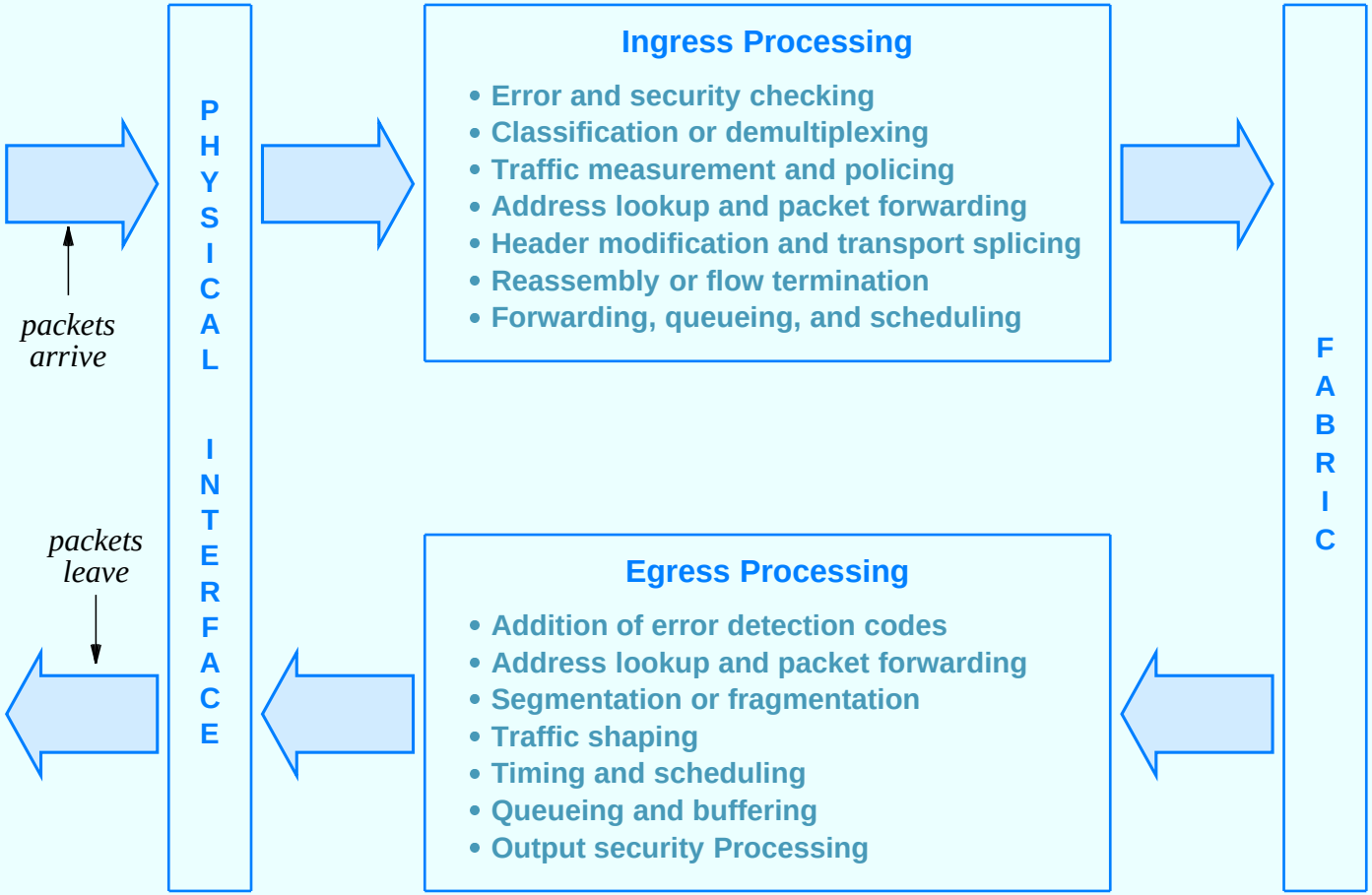
Ingress Processing

- 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


Questions?

XVI

Languages Used For Classification

Languages For Building A Network Stack

- Many questions
  - What language(s) are best?
  - How should processing be expressed?
  - How important is efficiency?
  - Must code be optimized by hand?

Possible Programming Paradigms

- Declarative
- Imperative
  - With explicit parallelism
  - With implicit parallelism

Choices Depend On

- Underlying hardware architecture
- Data and packet rates
- Software support available (e.g., compilers)

Desiderata

- High-level
- Declarative
- Data-oriented
- Efficient
- General or extensible
- Explicit links to actions

Possible Hardware Architectures

- RISC processor dedicated to classification
- RISC processor shared with other tasks
- Multiple, parallel RISC processors
- State machine or FPGA

Example Classification Languages

- We will consider two examples
  - NCL
  - FPL

NCL

- Expands to *Network Classification Language*
- Developed by Intel
- Runs on StrongARM
- Not part of fast path

NCL

- Network Classification Language
- Developed by Intel
- Runs on StrongARM
- Not part of fast path

NCL Characteristics

- High level
- Mixes declarative and imperative paradigms
- Associates *action* with each classification
- Optimized for protocols with fixed-size header fields
- Basic unit of data is eight-bit byte

NCL Notation

- Item specified by tuple
  - *Offset* beyond a base
  - *Size*
- Syntax:

  base [offset : size]

- Example
  - Two-octet field fourteen octets beyond symbol *FRAME*

FRAME [14 : 2]

NCL Measures Of Data Size

- Octets
  - Default
  - Syntax consists of integers
  - Usually more efficient

NCL Measures Of Data Size
(continued)

- Bits
  - Can specify arbitrary bit position / length
  - Syntax uses less-than and greater-than
  - Usually less efficient
  - Example of a four-bit string six bits beyond *FRAME*

```
FRAME [ <6> : <4> ]
```

NCL Comments

- C-style

  /* ... */

- C++-style

  // ...

NCL Primitives

- Can name header fields
- Names defined by offset / length
- No predefined protocol specifications
  - NCL does not understand IP, TCP, Ethernet, etc.
- No predefined network data types
  - NCL does not understand IP addresses, Ethernet addresses, etc.

Example NCL Code

```
protocol  ip {                  /* declaration of datagram header */
    vershl      { ip[0:1] }
    vers        { (vershl & 0xf0) >> 4 }
    tmphl       { (vershl & 0x0f) }
    hlen        { tmphl << 2 }
    totlen      { ip[2:2] }
    ident       { ip[4:2] }
    frags       { ip[6:2] }
    ttl         { ip[8:1] }
    proto       { ip[9:1] }
    cksum       { ip[10:2] }
    source      { ip[12:4] }
    dest        { ip[16:4] }

    demux       {
    ( proto == 6 )          { tcp at hlen }
    ( proto == 17 )         { udp at hlen }
    ( default )   { ipunknown at hlen }

    // note: other protocol types can be added here

    }
}                               // end of datagram header declaration
```

IP Header Length

- Found in second four bits of IP header
- Gives header size in 32-bit multiples
- NCL can
  - Define fields on byte boundaries
  - Extract bit fields
  - Perform basic arithmetic

Example NCL Code To Extract Header Length

```
vershl   { ip[0:1] }
vers     { (vershl & 0xf0) >> 4 }
tmphl    { (vershl & 0x0f) }
hlen     { tmphl << 2 }
```

Encapsulation

- Needed for layered protocols
- Type field from layer *N* specifies type of message encapsulated at layer *N + 1*
- Specified using *at* keyword

Example NCL Code For Encapsulation

```
( proto == 6 )  { tcp at hlen }
( proto == 17 ){ udp at hlen }
( default )      { ipunknown at hlen }
```

- If IP protocol is 6, *tcp*
- If IP protocol is 17, *icmp*
- Other cases classified as *ipunknown*

Actions

- Refers to imperative execution
- Names function to execute
- Associated with each classification
- Specified by *rule* statement

Example NCL Code To Invoke An Action

```
rule  web_filter  {  ip && tcp.dport == 80  }  {  web_proc(ip.src)  }
```

- Specifies
  - Name is *web_filter*
  - Predicate is *ip && tcp.dport == 80*
  - Action to be invoked is *web_proc(ip.src)*

Intrinsic Functions

- Built into language
- Handle checksum
  - Verification
  - Calculation
- Understand specific protocols

Available Intrinsic Functions

Function Name	Purpose
ip.chksumvalid	Verifies validity of IP checksum
ip.genchksum	Generates an IP checksum
tcp.chksumvalid	Verifies validity of TCP checksum
tcp.genchksum	Generates a TCP checksum
udp.chksumvalid	Verifies validity of UDP checksum
udp.genchksum	Generates a UDP checksum

Named Predicates

- Bind name to predicate
- Can be used later in program
- Associated with specific *protocol*

Example Predicate

predicate ip.fragmentable { ! ((ip.frags >> 14) & 0x01) }

Conditional Rule Execution

- Specifies order for predicate testing
- Linearizes classification
- Helps optimize processing
- Example use: verify IP checksum before proceeding

Illustration Of NCP
Conditional Rule Execution

```
predicate TCPvalid { tcp && tcp.cksumvalid }
with { (TCPvalid) {
    predicate SynFound { (tcp.flags & 0x02) }
    rule NewConnection { SynFound } { start_conn(tcp_dport) }
}
```

Incremental Protocol Definition

- Allows programmer to add definition of *field* to pre-existing *protocol* specification
- Works well with included files
- Useful when generic definition insufficient
- General form:

field protocol_name.field_name { definition }

- Example

field ip.flags { ip[6:<3>] }

NCL Set Facility

- Table search mechanism
- Tables grow dynamically
- Up to seven *keys* (each key is 32-bits)
- Programmer gives table size
  - Size must be power of two
  - Set can grow larger than estimate

Example NCL Set Declaration

- To declare set *my_addrs*:

```
set  my_addrs /* define a set of IP addresses */
     < 1 > {     /* number of keys in each entry */
     size_hint {  256  }
}
```

Example NCL Set Declaration (continued)

- To search *my_addrs*:

```
search  my_addrs.ip_src_lookup
        (ip.source) /* search key is IP source addr. */
```

NCL Preprocessor

- Adopts syntax from C-preprocessor
- Provides
  - #include
  - #ifndef


Questions?

FPL

- Functional Programming Language
- Developed by Agere Systems
- Runs on Agere FPP
- In the fast path

FPL

- Functional Programming Language
- Developed by Agere Systems
- Runs on Agere FPP
- In the fast path

FPL Characteristics

- High level
- Unusual, declarative language
- Follows pattern paradigm
- Designed for networking
- Differs from conventional syntactic pattern languages like SNOBOL or Awk
- Permits parallel evaluation

FPL Syntax

- C++ comments // *I can't understand this...*
- C-like preprocessor (#define)
- Variety of constants
  - Decimal
  - Hexadecimal (begins with 0x)
  - Bit string (begins with 0b)
  - Dotted decimal IP addresses
  - Dashed hexadecimal addresses

Two Pass Processing

- Fundamental idea in FPL
- Program contains two independent parts
- Incoming data processed by both parts
- Motivation: underlying interface hardware

Interface Hardware

- Divides each incoming packet into blocks of 64 octets
- Transfers one block at a time to FPP chip
- Sends additional information with each block
  - Port number on which received
  - Flags
    * Hardware detected error (e.g., invalid CRC)
    * Block is the first block of packet
    * Block is the final block of packet

Pass 1 Processing (blocks)

- Program invoked once for each block
- Accommodates blocks from multiple interfaces
- Collects the blocks for each packet into a separate queue
- Forwards complete queue to Pass 2 when
  - Final block of packet arrives
  - Error detected and processing aborted

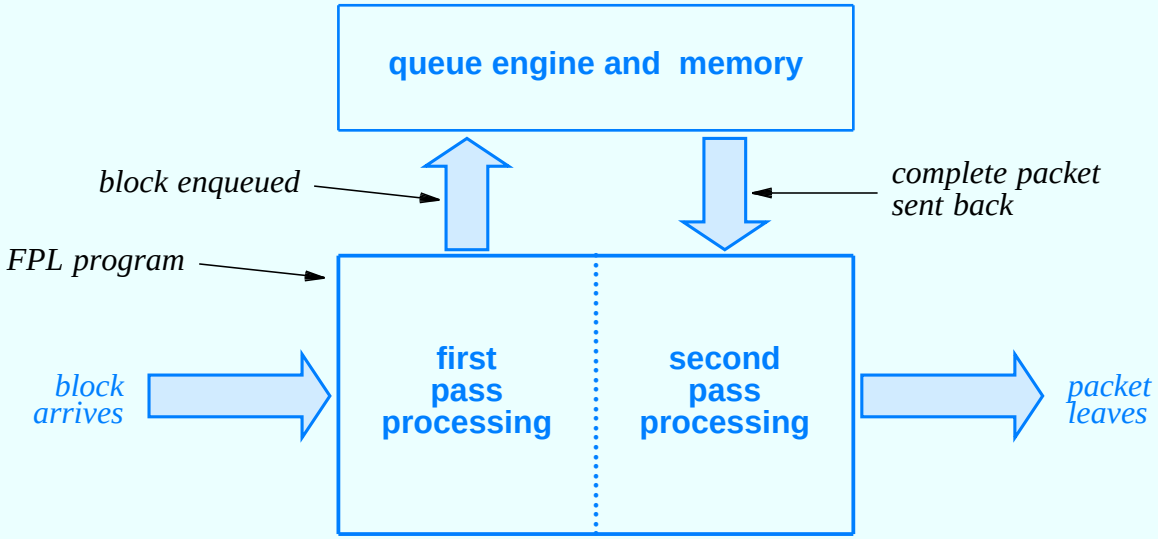
Collecting Blocks

- Blocks arrive asynchronously
  - From arbitrary port
  - In arbitrary order
- Typical trick: use incoming port number as queue ID

Pass 2 Processing (packets)

- Program invoked once for each packet
- Handles complete packet
- Receives
  - Packet from Pass 1
  - Status information for the packet
- Chooses disposition
  - Forward packet for transmission
  - Discard packet

Illustration Of FPL Processing



- Every FPL program contains code for two passes
- Status information kept with packet

FPL Invocation

- Handled by hardware
- Pass 1 invoked when block arrives
- Pass 2 invoked when packet ready
- No explicit polling

Designating Passes

- Programmer specifies starting label for each pass
- Handled by language directives
- To designate first pass:

SETUP ROOT(*starting_label*)

- To designate second pass:

SETUP REPLAYROOT(*starting_label*)

Using A Pattern Match For Conditional Processing

- Fact 1: FPL does not provide a conditional statement
- Fact 2: FPL does provide matching among a set of patterns
- Trick: use pattern selection to emulate conditional execution

Example Conditional In First Pass Processing

- Hardware sets variable *$FramerEOF*
  - Value is 1 if block is final block of packet
  - Value is 0 for other blocks
- Built-in functions used to enqueue block
  - *fQueue* adds block to queue
  - *fQueueEOF* adds block to queue and marks queue ready for second pass

Example Conditional In First Pass Processing (continued)

- Conceptual algorithm for enqueuing a block:

```
if ($FramerEOF) {
        fQueueEOF();
} else {
        fQueue();
}
```

Pattern Used For Conditional

```
#include "fpp.fpl"

// Code for the first pass of a program that handles Ethernet packets

SETUP   ROOT(HandleBlock);

// Pass 1

HandleBlock:  EtherBlock($framerEOF:1);

EtherBlock:   0b0  fQueue(0:2, $portNumber:16, $offset:6, 0:1, 0:2);
EtherBlock:   0b1  fQueueEof(0:2, $portNumber:16, $offset:6, 0:1, 0:2, 0:24);
```

Code For Second Pass

```
// Code for the second pass of a program that classifies Ethernet frames

// Symbolic constants used for classifications

#define   FRAMETYPEA  1      // ARP frames
#define   FRAMETYPEI   2      // IP frames
#define   FRAMETYPEO  3      // Other frames (not the above)

SETUP   REPLAYROOT(HandleFrame);

// Pass 2

HandleFrame:
          fSkip(96)                 // skip past dest & src addresses
          cl = CLASS                // compute class from frame type
          fSkipToEnd()
          fTransmit(cl:21, 0:16, 0:5, 0:6);

CLASS:  0x0806:16           fReturn(FRAMETYPEA);
CLASS:  0x0800:16           fReturn(FRAMETYPEI);
CLASS:  BITS:16             fReturn(FRAMETYPEO);
```

Conceptual Pattern Match

- Program specifies sequence of patterns
- Pointer moves through packet
- Processing proceeds if bits in packet match pattern

Two Basic Types Of Pattern Functions

- Control function
  - Sequence of items to match
  - Processed in order
  - Exactly one path
- Tree function
  - Set of patterns
  - Data in packet compared to all patterns
  - Exactly one must match

Example Control Function

```
HandleFrame:
        fSkip(96)               // skip past dest & src addresses
        cl = CLASS              // compute class from frame type
        fSkipToEnd()
        fTransmit(cl:21, 0:16, 0:5, 0:6);
```

Example Tree Function

```
CLASS: 0x0806:16        fReturn(FRAMETYPEA);
CLASS: 0x0800:16        fReturn(FRAMETYPEI);
CLASS: BITS:16          fReturn(FRAMETYPEO);
```

Special Patterns

- *fSkip*
  - Skips bits in input
- *fSkipToEnd*
  - Skips to end of current packet
- *fTransmit*
  - Sends packet to RSP chip
- *BITS*
  - Matches arbitrary bits (default case)
- Note: second pass must match entire packet

Pattern Optimization

- FPL compiler optimizes patterns
- Typical optimization: eliminate common prefix

Example Pattern Optimization

```
IP:      0x4  0x5  fSkip(152)   fReturn(IPOPTIONS0);
IP:      0x4  0x6  fSkip(160)   fReturn(IPOPTIONS1);
IP:      0x4  0x7  fSkip(168)   fReturn(IPOPTIONS2);
IP:      0x4  BITS:4            fReturn(IPUNKNOWN);
```

(a)

```
IP:      0x4  IPHDR;
IPHDR:   0x5  fSkip(152)        fReturn(IPOPTIONS0);
IPHDR:   0x6  fSkip(160)        fReturn(IPOPTIONS1);
IPHDR:   0x7  fSkip(168)        fReturn(IPOPTIONS2);
IPHDR:   BITS:4                 fReturn(IPUNKNOWN);
```

(b)

IP Address Example

```
ipAddrMatch:  *.*.*.*       fReturn(0);
ipAddrMatch:  10.*.*.*      fReturn(1);
ipAddrMatch:  128.10.*.*    fReturn(2);
ipAddrMatch:  128.211.*.*   fReturn(2);
ipAddrMatch:  128.210.*.*   fReturn(3);
```

FPL Variables

Variable	Pass	Meaning
$framerSOF	1	Is the current block the first of a frame?
$ferr	1	Did the hardware framer detect an error?
$portNumber	1	Port number from which block arrived
$framerEOF	1	Is the current block the last of a frame?
$offset	1 or 2	Offset of data within a buffer
$currOffset	1 or 2	Current byte position being matched
$currLength	1 or 2	Length of item being processed
$pass	1 or 2	Pass being executed
$tag	1 or 2	Status value sent from pass 1 to pass 2

Dynamic Classification

- Can extend tree function at run-time
- Requires use of ASI
- Pattern converted to internal form

Summary

- Programming languages for classification are
  - Special-purpose
  - High-level
  - Declarative
  - Data-oriented
  - Provide links to actions
- We examined two languages
  - NCL, the Network Classification Language from Intel
  - FPL, the Functional Programming Language from Agere


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?

XVIII

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 IXP1200

- Name of first generation IXP chip
- Four models available

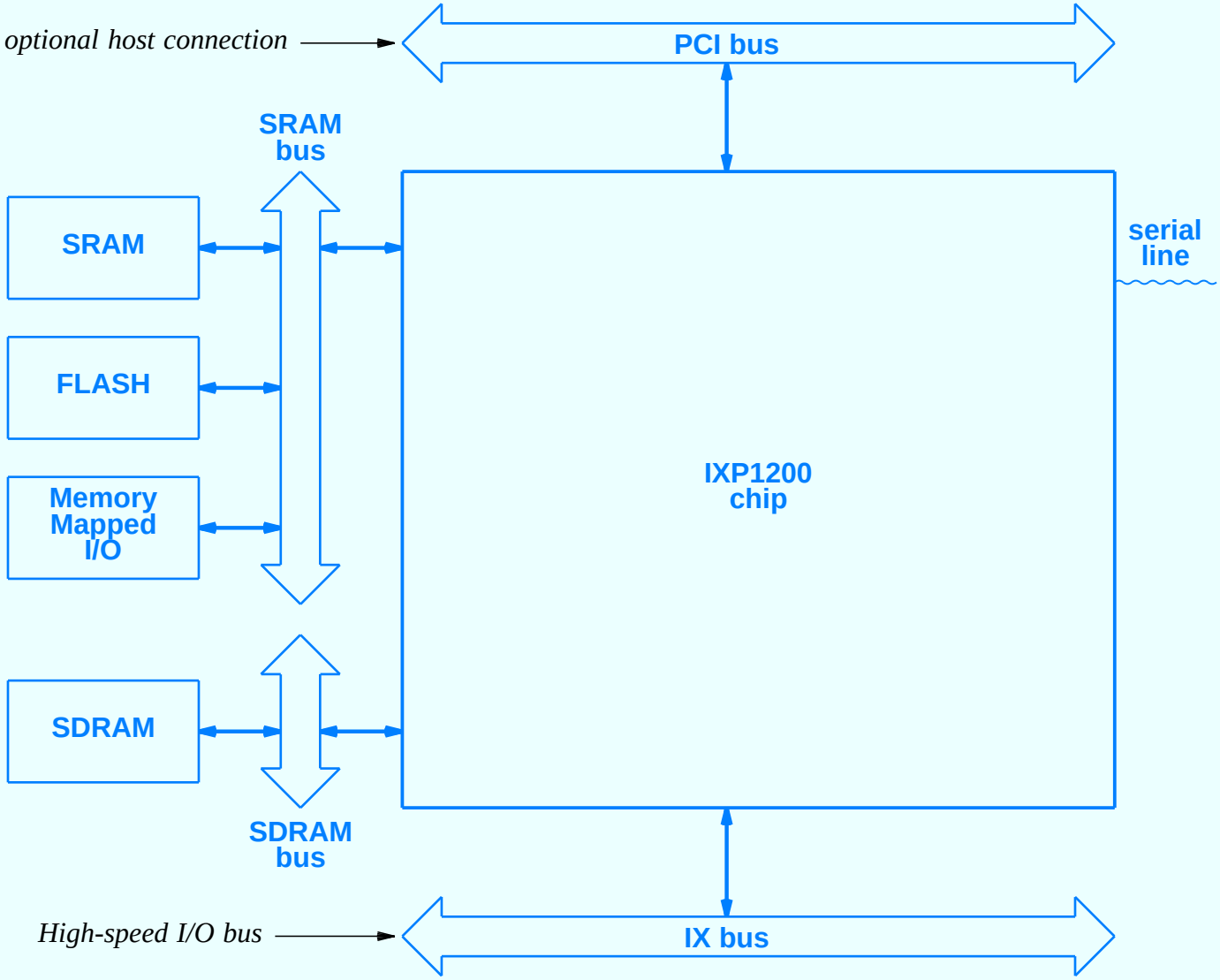
Model Number	Pin Count	Support For CRC	Support For ECC	Possible Clock Rates (in MHz)
IXP1200	432	no	no	166, 200, or 232
IXP1240	432	yes	no	166, 200, or 232
IXP1250	520	yes	yes	166, 200, or 232
IXP1250	520	yes	yes	166 only

- Differences in speed, power consumption, packaging
- Term *IXP1200* refers to any model

IXP1200 Features

- One embedded RISC processor
- Six programmable packet processors
- Multiple, independent onboard buses
- Processor synchronization mechanisms
- Small amount of onboard memory
- One low-speed serial line interface
- Multiple interfaces for external memories
- Multiple interfaces for external I/O buses
- A coprocessor for hash computation
- Other functional units

IXP1200 External Connections



IXP1200 External Connection Speeds

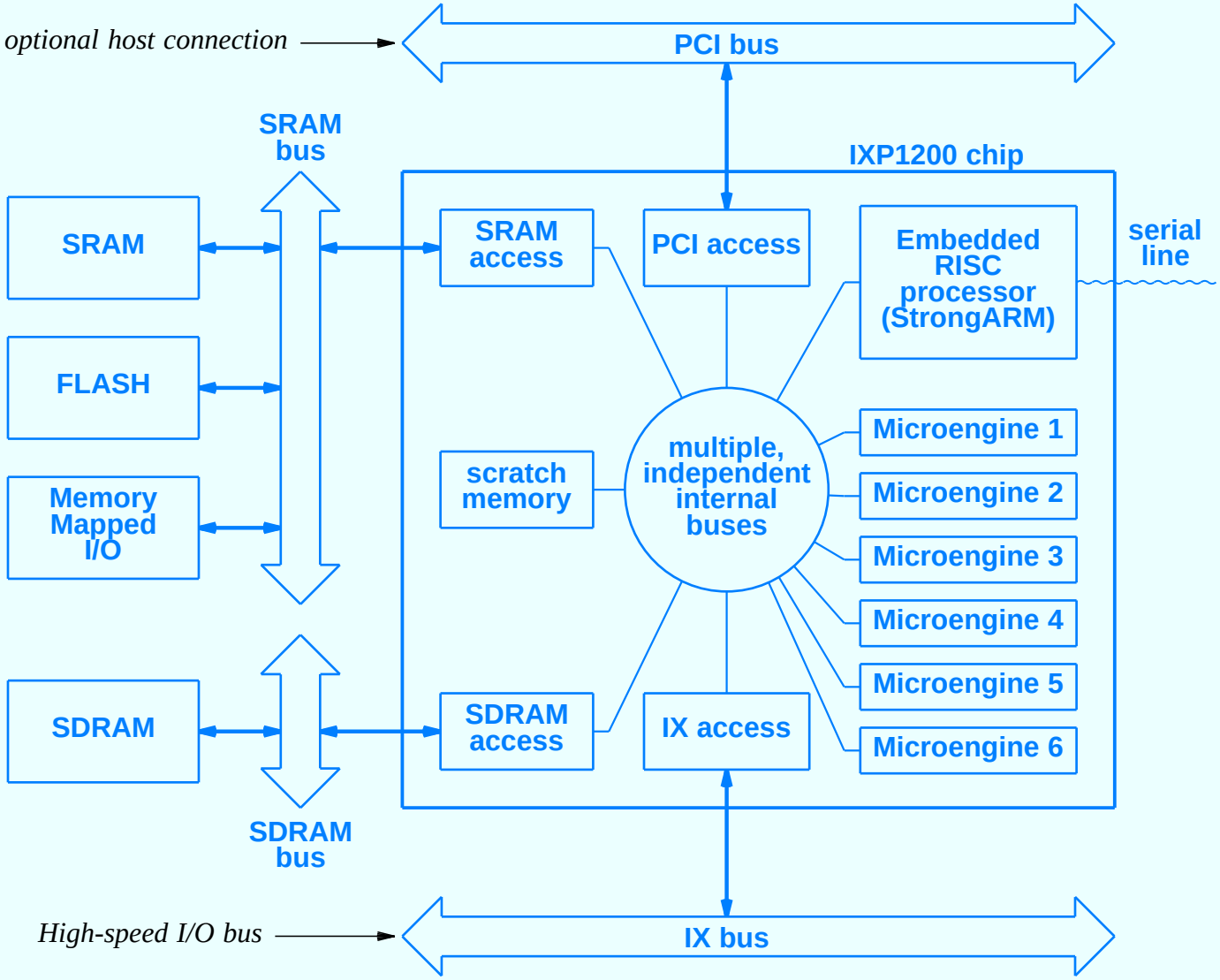
Type	Bus Width	Clock Rate	Data Rate
Serial line	(NA)	(NA)	38.4 Kbps
PCI bus	32 bits	33-66 MHz	2.2 Gbps
IX bus	64 bits	66-104 MHz	4.4 Gbps
SDRAM bus	64 bits	≤ 232 MHz	928.0 MBps
SRAM bus	16 or 32 bits	≤ 232 MHz	464.0 MBps

- Note: MBps abbreviates *Mega Bytes per second*

IXP1200 Internal Units

Quantity	Component	Purpose
1	Embedded RISC processor	Control, higher layer protocols, and exceptions
6	Packet processing engines	I/O and basic packet processing
1	SRAM access unit	Coordinate access to the external SRAM bus
1	SDRAM access unit	Coordinate access to the external SDRAM bus
1	IX bus access unit	Coordinate access to the external IX bus
1	PCI bus access unit	Coordinate access to the external PCI bus
several	Onboard buses	Internal control and data transfer

IXP1200 Internal Architecture



Processors On The IXP1200

Processor Type	Onboard?	Programmable?
General Purpose Processor	no	yes
Embedded RISC Processor	yes	yes
Microengines	yes	yes
Coprocessors	yes	no
Physical Interfaces	no	no

IXP1200 Memory Hierarchy

Memory Type	Maximum Size	On Chip?	Typical Use
GP Registers	128 regs.	yes	Intermediate computation
Inst. Cache	16 Kbytes	yes	Recently used instructions
Data Cache	8 Kbytes	yes	Recently used data
Mini Cache	512 bytes	yes	Data that is reused once
Write buffer	unspecified	yes	Write operation buffer
Scratchpad	4 Kbytes	yes	IPC and synchronization
Inst. Store	64 Kbytes	yes	Microengine instructions
FlashROM	8 Mbytes	no	Bootstrap
SRAM	8 Mbytes	no	Tables or packet headers
SDRAM	256 Mbytes	no	Packet storage

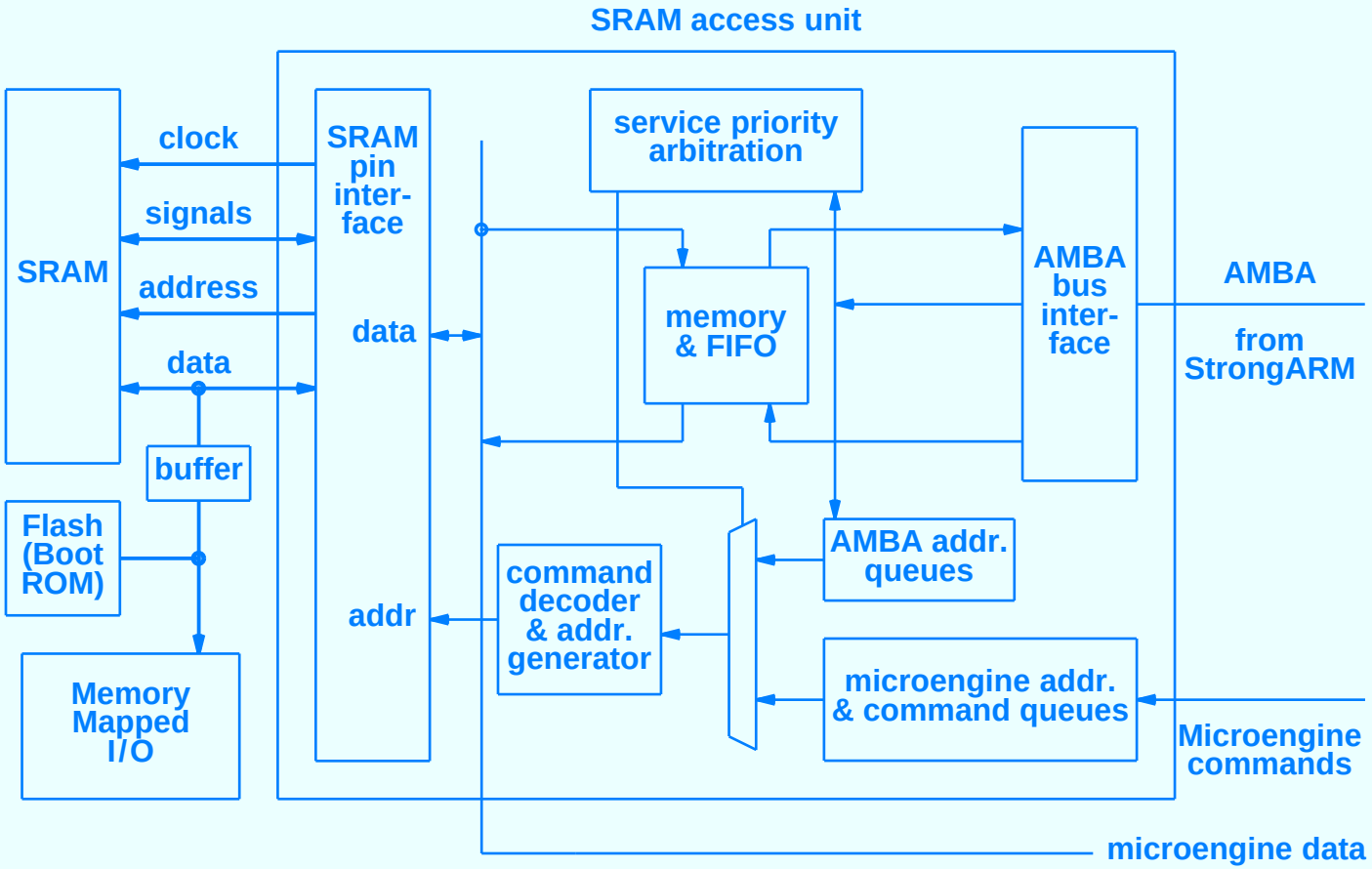
IXP1200 Memory Characteristics

Memory Type	Addressable Data Unit (bytes)	Relative Access Time	Special Features
Scratch	4	12 - 14	synchronization via test-and-set and other bit manipulation, atomic increment
SRAM	4	16 - 20	queue manipulation, bit manipulation, read/write locks
SDRAM	8	32 - 40	direct transfer path to I/O devices

Memory Addressability

- Each memory specifies *minimum addressable unit*
  - SRAM organized into 4-byte *words*
  - SDRAM organized into 8-byte *longwords*
- Physical address refers to word or longword
- Examples
  - SDRAM address 1 refers to bytes 8 through 15
  - SRAM address 1 refers to bytes 4 through 7

Example Of Complexity: SRAM Access Unit



Summary

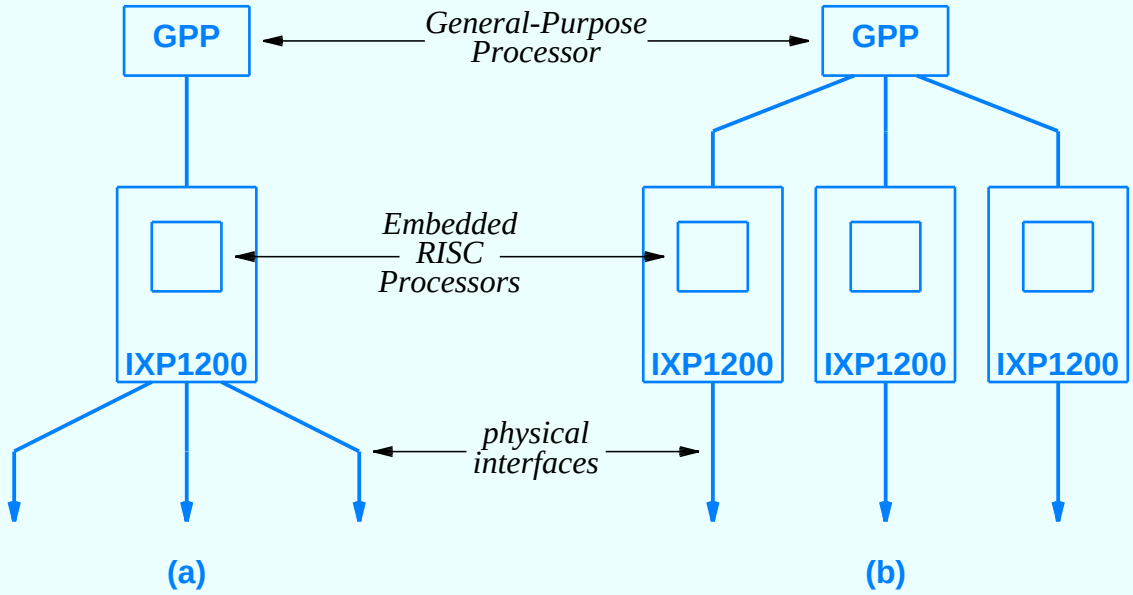
- We will use Intel IXP1200 as example
- IXP1200 offers
  - Embedded processor plus parallel packet processors
  - Connections to external memories and buses


Questions?

XIX

Embedded RISC Processor (StrongARM Core)

StrongARM Role



- (a) Single IXP1200
- (b) Multiple IXP1200s
- Role of StrongARM differs

Tasks That Can Be Performed By StrongARM

- 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

StrongARM Characteristics

- Reduced Instruction Set Computer (RISC)
- Thirty-two bit arithmetic
- Vector floating point provided via a coprocessor
- Byte addressable memory
- Virtual memory support
- Built-in serial port
- Facilities for a kernelized operating system

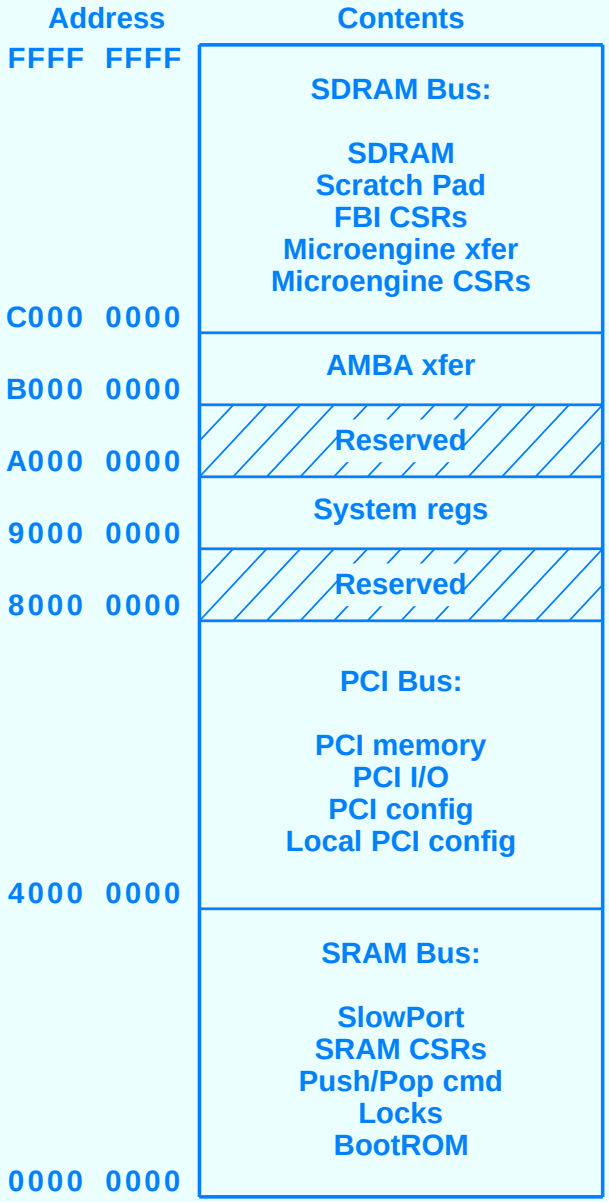
Arithmetic

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

StrongARM Memory Organization

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

StrongARM Address Space



Summary

- Embedded processor on IXP1200 is StrongARM
- StrongARM addressing
  - Single, uniform address space
  - Includes all memories
  - Byte addressable


Questions?

XXI

Reference System And Software Development Kit (Bridal Veil, SDK)

Reference System

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

Intel Reference Hardware

- Single-board network processor testbed
- Plugs into PCI bus on a PC

Intel Reference Hardware (continued)

Quantity or Size	Item
1	IXP1200 network processor (232MHz)
8	Mbytes of SRAM memory
256	Mbytes of SDRAM memory
8	Mbytes of Flash ROM memory
4	10/100 Ethernet ports
1	Serial interface (console)
1	PCI bus interface
1	PMC expansion site

Intel Reference Software

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

Software	Purpose
C compiler	Compile programs for the StrongARM
MicroC compiler	Compile programs for the microengines
Assembler	Assemble programs for the microengines
Downloader	Load software into the network processor
Monitor	Communicate with the network processor and interact with running software
Bootstrap	Start the network processor running
Reference Code	Example programs for the IXP1200 that show how to implement basic functions

Basic Paradigm

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

Bootstrapping Procedure

1. Powering on host causes network processor board to run boot monitor from Flash memory
2. Device driver on host communicates across the PCI bus with boot monitor to load operating system (Embedded Linux) and initial RAM disk configuration into network processor's memory
3. Host signals boot monitor to start the StrongARM
4. Operating system runs login process on serial line and starts telnet server
5. Operating systems on host and StrongARM configure PCI bus to act as Ethernet emulator; the StrongARM uses NFS to mount two file systems, R and W, from a server on the host

Starting Software

1. Compile code for StrongARM and microengines
2. Create system configuration file named *ixsys.config*
3. Copy code and configuration file to read-only public download directory, R
4. Run terminal emulation program on host and log onto StrongARM
5. Change to NSF-mounted directory R, and run shell script *ixstart* with argument *ixsys.config*
6. Later, to stop the IXP1200, run *ixstop* script

Summary

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


Questions?

XXII

Programming Model (Intel ACE)

Active Computing Element (ACE)

- Defined by Intel's SDK
- Not part of hardware

ACE Features

- Fundamental software building block
- Used to construct packet processing systems
- Runs on StrongARM, microengine, or host
- Handles control plane and fast or slow data path processing
- Coordinates and synchronizes with other ACEs
- Can have multiple inputs or outputs
- Can serve as part of a pipeline

ACE Terminology

- Library ACE
  - Built by Intel
  - Available with SDK
- Conventional ACE
  - Built by Intel customers
  - Can incorporate items from *Action Service Libraries*
- MicroACE
  - *Core component* runs on StrongARM
  - *Microblock component* runs on microengine

More Terminology

- Source microblock
  - Handles ingress from I/O device
- Sink microblock
  - Handles egress to I/O device
- Transform microblock
  - Intermediate position in a pipeline

Four Conceptual Parts Of An ACE

- Initialization
- Classification
- Actions associated with each classification
- Message and event management

Output Targets And Late Binding

- ACE has set of outputs
- Each output given *target name*
- Outputs bound dynamically at run time
- Unbound target corresponds to packet discard

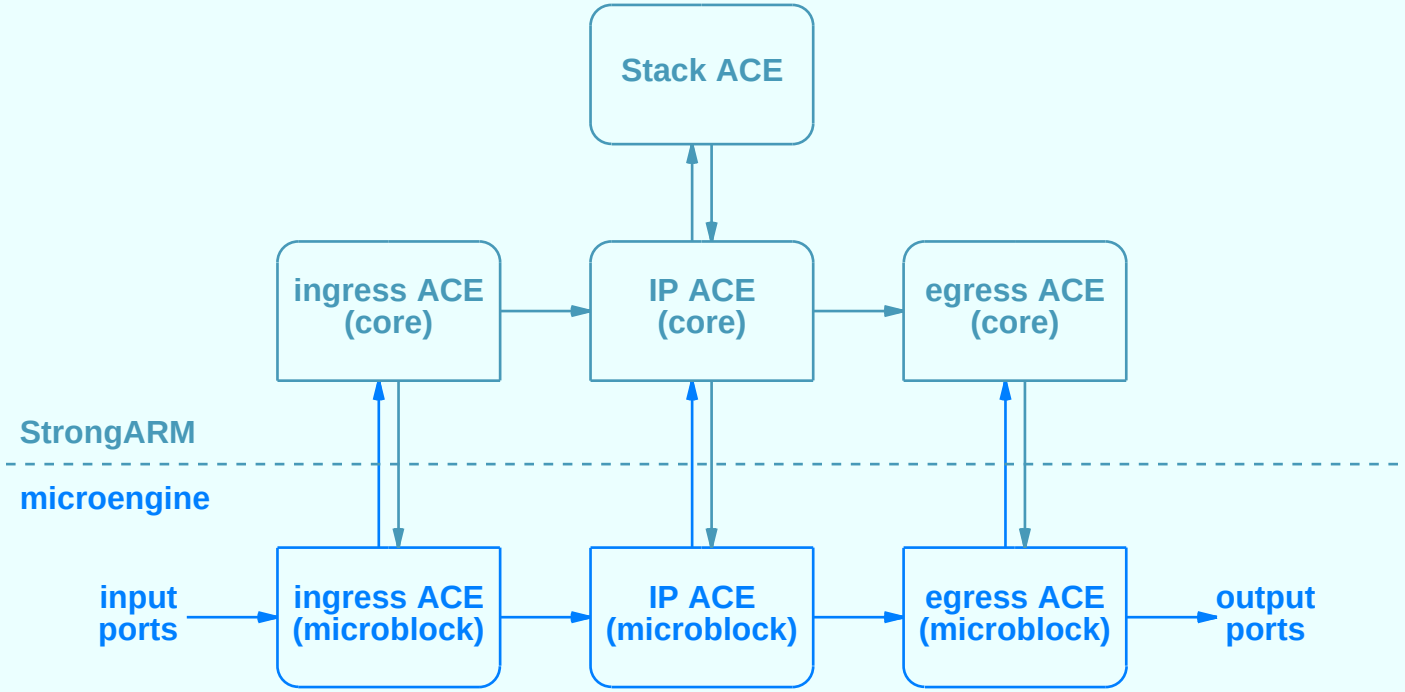
Conceptual ACE Interconnection

- Ingress ACE acts as *source*
- Process ACE acts as *transform*
- Egress ACE acts as *sink*
- Connections created at run time

Components Of MicroACE (review)

- Single ACE with two components
  - StrongARM (core component)
  - Microengines (microblock component)
- Communication possible between components

Division Of ACE Into Components

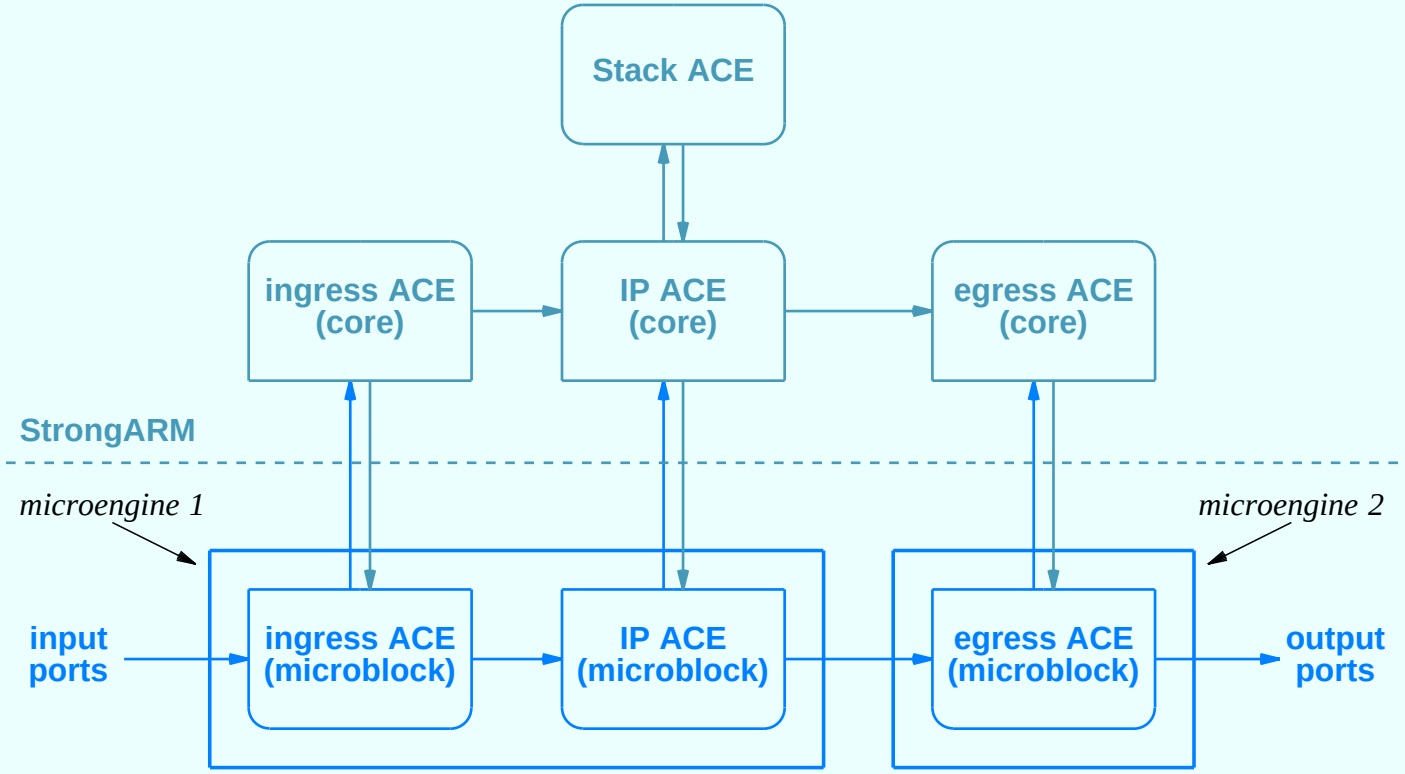


- Microblocks form *fast path*
- Stack ACE runs entirely on StrongARM

Microblock Group

- Set of one or more microblocks
- Treated as single unit
- Loaded onto microengine for execution
- Can be replicated on multiple microengines for higher speed

Illustration Of Microblock Groups

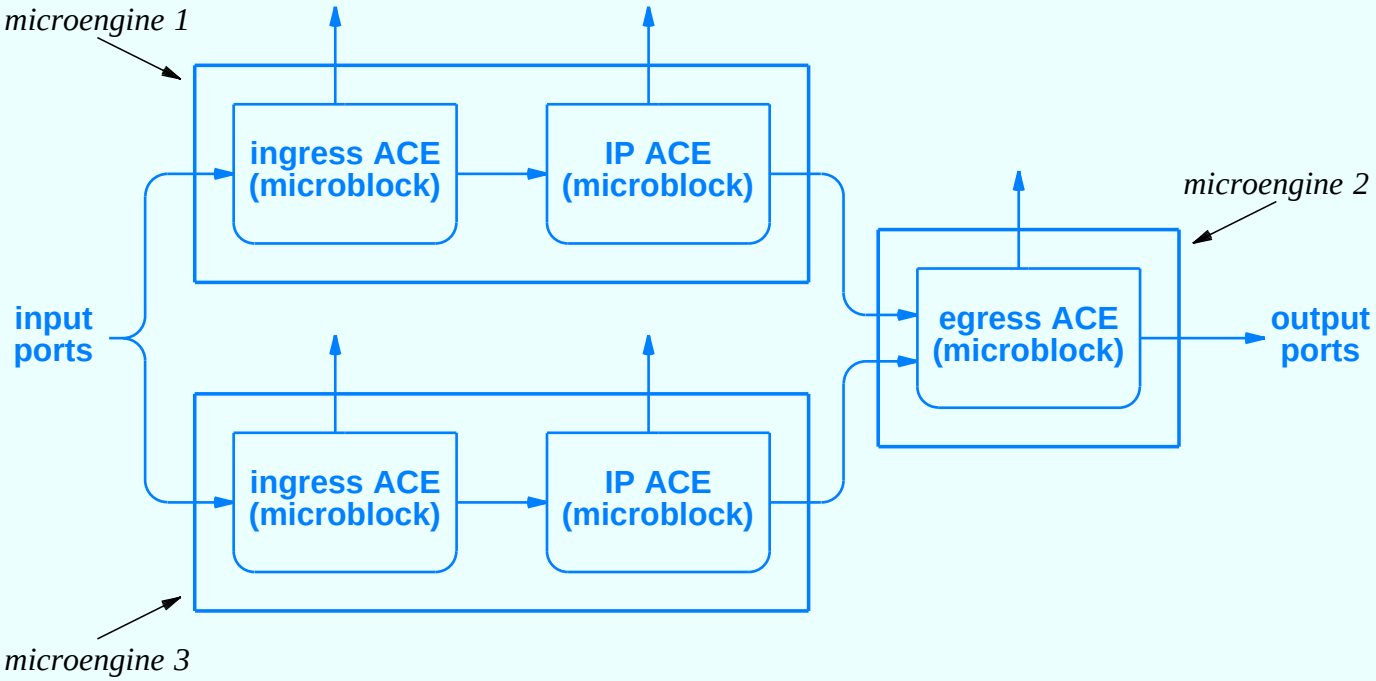


- Entire microblock group assigned to same microengine

Replication Of Microblock Group

- Typically used for
  - Ingress microblock group
  - Egress microblock group
- Replications depend on number and speed of interfaces
- SDK software computes replications automatically

Illustration Of Replication



- Ingress microblock group replicated on microengines 1 & 3
- Incoming packet taken by either copy

Microblock Structure

- Asynchronous model
- Programmer creates
  - Initialization macro
  - Dispatch loop

Dispatch Loop

- Determines disposition of each packet
- Uses *return code* to either
  - Send packet to StrongARM
  - Forward packet to ‘‘next’’ microblock
  - Discard packet

Dispatch Loop Algorithm

```
Allocate global registers;
Initialize dispatch loop;  Initialize Ethernet devices;
Initialize ingress microblock;  Initialize IP microblock;
while (1)  {
        Get next packet from input device(s);
        Invoke ingress microblock;
        if  ( return code == 0 )  {
                Drop the packet;
        }  else if  ( return code == 1 )  {
                Send packet to ingress core component;
        }  else  {  /* IP packet */
                Invoke IP microblock;
                if  ( return code == 0 )  {
                        Drop packet;
                }  else if ( return code == 1 )  {
                        Send packet to IP core component;
                }  else  {
                        Send packet to egress microblock;
                }
        }
}
```

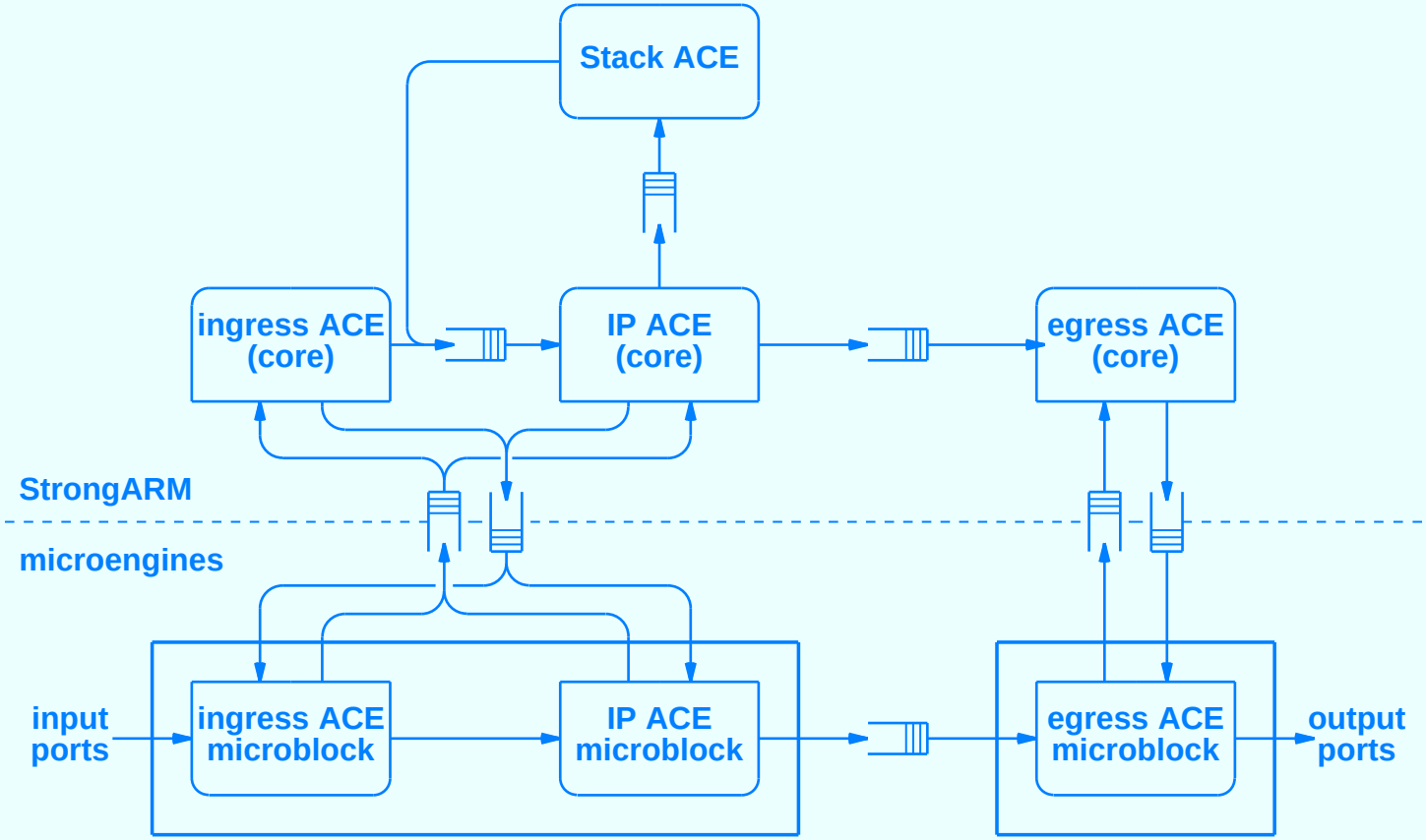
Arguments Passed To Processing Macro

- A *buffer handle* for a frame that contains a packet
- A set of state registers
  - Contain information about the frame
  - Can be modified
- A variable named *dl_next_block*
  - Used to store return code

Packet Queues

- Placed between ACE components
- Buffer packets
- Permit asynchronous operation

Illustration Of Packet Queues



Exceptions

- Packets passed from microblock to core component
- Mechanism
  - Microcode sets dl_next_block to *IX_EXCEPTION*
  - Dispatch loop forwards packet to core
  - ACE tag used to identify corresponding component
  - *Exception handler* is invoked in core component

Possible Actions Core Component Applies To Exception

- Consume the packet and free the buffer
- Modify the packet before sending it on
- Send the packet back to the microblock for further processing
- Forward the packet to another ACE on the StrongARM

Crosscall Mechanism

- Used between
  - Core component of one ACE and another
  - ACE core component and non-ACE application
- Not intended for packet transfer
- Operates like Remote Procedure Call (RPC)
- Mechanism known as *crosscall*

Crosscall Implementation

- Both caller and callee programmed to use crosscall
- Declaration given in *Interface Definition Language* (*IDL*)
- IDL compiler
  - Reads specification
  - Generates *stubs* that handle calling details

Crosscall Implementation
(continued)

- Three types of crosscalls
  - Deferred: caller does not block; return notification asynchronous
  - Oneway: caller does not block; no value returned
  - Twoway: caller blocks; callee returns a value
- Twoway call corresponds to traditional RPC
- Type of call determined at compile time

Twoway Calling And Blocking

Because the core component of an ACE is prohibited from blocking, an ACE cannot make a twoway call.

Summary

- Intel SDK uses ACE programming model
- ACE
  - Basic unit of computation
  - Can include code for StrongARM (core) and microengines (microblock)
- Packet queues used to pass packets between ACEs
- Crosscall mechanism used for nonpacket communication


Questions?

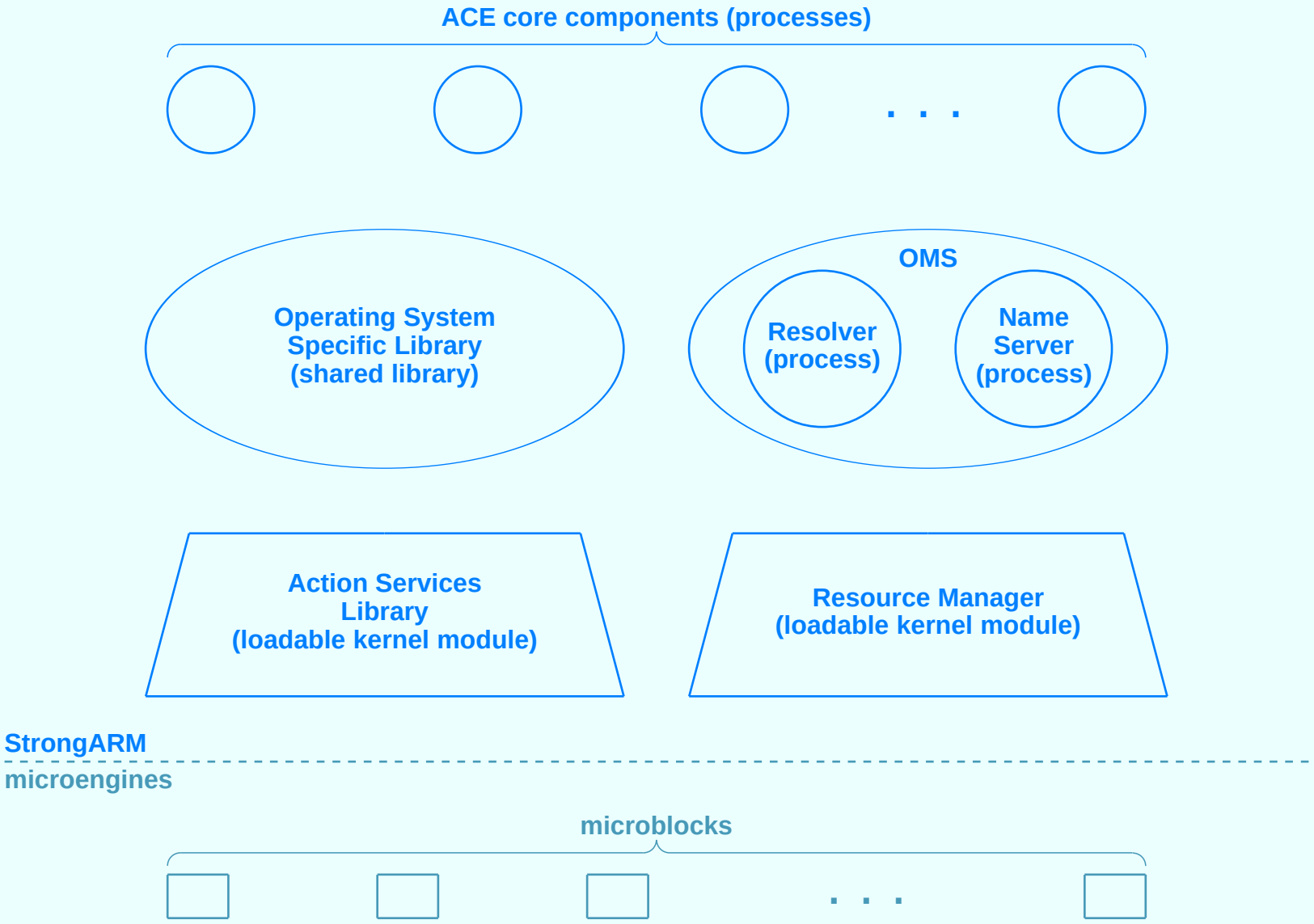
XXIII

ACE Run-Time Structure
And
StrongARM Facilities

StrongARM Responsibilities

- Loading ACE software
- Creating and initializing ACE
- Resolving names
- Managing the operation of ACEs
- Allocating and reclaiming resources (e.g., memory)
- Controlling microengine operation
- Communication among core components
- Interface to non-ACE applications
- Interface to operating system facilities
- Forwarding packets between core component and microblock(s)

Illustration Of ACE Components



Components

- Object Management System
  - Binds names to objects
  - Contains
    - Resolver
    - Name server
- Resource Manager
  - Access to OS
  - Communication with microengines
  - Memory management

Components
(continued)

- Operating System Specific Library
  - Shared library
  - Provides virtual API
  - Should be named *OS independent library*
- Action Services Library
  - Loadable kernel module
  - Run-time support for core component
  - TCP/IP support

Microengine Assignment

- Automated by SDK
- Programmer specifies
  - Type (speed) of each port
  - Type of each ACE
- SDK chooses
  - Number of replications for each microblock
  - Assignment of microblocks to microengines

Type Values Used In Configuration File

- Only two port types
  - *Slow* corresponds to 10 / 100 Ethernet
  - *Fast* corresponds to Gigabit Ethernet

Numeric Value	Meaning
0	Slow ingress file
1	Slow egress file
2	Fast ingress file for Port 1
3	Fast ingress file for Port 2
4	Fast egress file

Pieces Of ACE Code Programmer Writes

- *Initialization function* called when ACE begins
- *Exception handler* receives packets sent by microblock
- *Action functions* that correspond to packet classifications
- *Crosscalls* ACE accepts
- *Timer functions* for timed events
- *Callback functions* for returned values
- *Termination function* called when the ACE terminates

Reserved Names

- SDK software uses fixed names for some functions
- Examples
  - ACE initialization function must be named *ix_init*
  - ACE termination function must be named *ix_fini*

Overall Structure Of Core Component

- Core component of ACE runs as separate Linux process
- Always the same structure
- Programmer does *not* write main program
- SDK supplies structure and main program
- Uses an *event loop*

Conceptual Structure Of ACE Core Component

```
main( )                     /* Core component of an ACE        */
{
      Intel_init( );        /* Perform internal initialization */
      ix_init( );           /* Call user's initialization function */
      Intel_event_loop( ):  /* Perform internal event loop     */
      ix_fini( );           /* Call user's termination function */
      Intel_fini( );        /* Perform internal cleanup        */
      exit( );              /* Terminate the Linux process     */
}
```

Event Loop

- Central to asynchronous programming model
- Uses polling
- Repeatedly checks for presence of event(s) and calls appropriate handler
- Can be hidden from programmer
- In ACE model
  - Explicit
  - Programmer can modify / extend

Illustration Of ACE Event Loop

```
Intel_event_loop( )
{
        do forever  {
                E = getnextevent( );
                if (E is termination event)  {
                        return to caller;
                }  else if (E is exception event)  {
                        call exception handler function;
                }  else if (E is timer event)  {
                        call timer handler function;
                /* Note: additional event tests can be added here*/
                }
        }
}
```

A Note About Event Loops

Beware: although it may seem trivial, the event loop mechanism has several surprising consequences for programmers.

Event Loop Processing

- If event loop stops
  - All processing stops
  - The arrival of a new event will not trigger invocation of an event handler
- Conclusion: the event loop must go on

Asynchronous Programming, Event Loops, Threads, And Blocking

Because each core component executes as a single thread of control, no handler function is permitted to block because doing so will stop the event loop and block the entire core component.

Prohibition On Blocking Calls

- Programmer must use asynchronous call model
- Calling program specifies
  - Function to invoke
  - Arguments to pass
  - *Callback* function
- Called program
  - Invoked asynchronously
  - Receives copy of arguments, and computes result
  - Specifies callback to be invoked

Illustration Of Asynchronous Callback (part 1)

```
h {                             /* Synchronous version          */
        y = f(x);               /* Call f (potentially blocking)  */
        z = g(y);               /* Use the result from f to call g  */
        q += z;                 /* Use the value of z to update q*/
        return;
}

h1 {                            /* Asynchronous version         */
        allocate global variables y, z, and q;
        establish cbf1 as the callback function for f1;
        establish cbg1 as the callback function for g1;
        Start f1(x) with a nonblocking call;
        return;
}
```

Illustration Of Asynchronous Callback (part 2)

```
function cbf1(retval) {          /* Callback function for f1          */
        y = retval;
        start g1(y) with a nonblocking call;
        return;
}

function cbg1(retval) {          /* Callback function for g1          */
        z = retval;
        q += z;
        return;
}
```

Code Complexity

Because it uses the asynchronous paradigm, the core component of an ACE is usually significantly more complex and difficult to understand than a synchronous program that solves the same problem.

Some Good News

- ACE core component structured around event loop
- Underlying system is sequential
- No mutual exclusion needed!

Some Good News

- ACE core component structured around event loop
- Underlying system is sequential
- No mutual exclusion needed!

Memory Allocation

- Performed by StrongARM
- Function *RmMalloc* (in Resource Manager) allocates memory
- Request must specify type of memory
  - SRAM
  - SDRAM
  - Scratch
- Function *RmGetPhysOffset* maps virtual address to physical address
- Resulting physical address passed to microblock

Steps Taken At System Startup

- Load and start ASL kernel module, name server, and resolver
- Load and start device drivers for the network interfaces
- Load and initialize the Resource Manager which determines how many copies of each ingress and egress microblock group to run
- Parse and check configuration file *ixsys.config*
- Start core component of each ACE, which causes ACE to invoke *ix_init* function

Steps Taken At System Startup (continued)

- Turn on interfaces, assign microblock groups to microengines, and resolve external references (known as *patching*)
- Bind the ACE targets and physical interfaces
- Start microengines running

ACE Data Structure

- Stores information needed by Intel SDK software
- Must be allocated by *ix_init* and deallocated by *ix_fini*
- Uses structure *ix_ace*
- Can be extended with data defined by programmer

Example ACE Data Declaration

```
#include <ix/asl.h>             /* Include Intel's library declarations*/
structmyace   {                 /* Programmer's control block      */
        struct     ix_ace ace;  /* Intel's ace embedded as first item*/
        int        mycount;     /* Counter used by programmer's code*/

        /* Programmer can insert additional data items here...   */
}
```

Example Data Structure Allocation

```
ix_init  (  ... , ix_ace **app , ... )
{
        struct myace  *ap;    /* Ptr to programmer's control block*/

        ap  =  malloc ( sizeof ( struct myace ) ) ;
        *app = &ap->ace ;
        ix_ace_init ( &ap->ace ) ;

        /* Other initialization code goes here... */
}
```

Crosscall

- Uses OMS
- Three types
  - Oneway: no return
  - Twoway: conventional procedure call
  - Deferred: asynchronous return through callback

Possible Crosscall Communication

Caller	Called Procedure	Block?
ACE core component	ACE core component	no
ACE core component	non-ACE application	no
non-ACE application	ACE core component	no
non-ACE application	non-ACE application	yes

Crosscall Declaration

- Uses *Interface Definition Language (IDL)*
- Declares type of
  - Exported functions
  - Arguments

Example IDL Declaration

```
interface PacketCounter
{
        struct packetinfo  {
                int  numpackets;      /* Count of total packets      */
                int  numbcasts;       /* Count of broadcast packets*/
        };

        oneway void incTcount ( in  int  addedpackets ) ;
        deferred int resetTcount ( in  int  newnumpkts ) ;
        twoway int resetPinfo ( in  struct  packetinfo ) ;
};
```

Timer Management

- Performed on StrongARM
- Uses OMS
- Follows asynchronous model

Using A Timer

1. Initialize H, a handle for an ix_event.
2. Associate handle H with a callback function, CB.
3. Calculate T, a time for the event to occur.
4. Schedule event H at time T.
5. At any time prior to T, event H can be cancelled.
6. At time T, function CB will be called if the event has not been cancelled.

Summary

- Core component of ACE
  - Runs as Linux process
  - Provides asynchronous API
  - Uses event loop mechanism
  - Includes functions named *ix_init* and *ix_fini*
  - Responsible for memory allocation and timer management


Questions?

XX

Packet Processor Hardware (Microengines And FBI)

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
- One hundred twenty-eight general-purpose registers
- One hundred twenty-eight transfer registers
- Hardware support for four threads and context switching
- Five-stage execution pipeline
- Control of an Arithmetic Logic Unit (ALU)
- Direct access to various functional units

Microengine Level

- Not a typical CPU
- Does not contain native instruction for each operation
- 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

Instruction	Description
Arithmetic, Rotate, And Shift Instructions	
ALU	Perform an arithmetic operation
ALU_SHF	Perform an arithmetic operation and shift
DBL_SHIFT	Concatenate and shift two longwords
Branch and Jump Instructions	
BR, BR=0, BR!=0, BR>0, BR>=0, BR<0, BR<=0, BR=count, BR!=count	Branch or branch conditional
BR_BSET, BR_BCLR	Branch if bit set or clear
BR=BYTE, BR!=BYTE	Branch if byte equal or not equal
BR=CTX, BR!=CTX	Branch on current context
BR_INP_STATE	Branch on event state
BR_!SIGNAL	Branch if signal deasserted
JUMP	Jump to label
RTN	Return from branch or jump
Reference Instructions	
CSR	CSR reference
FAST_WR	Write immediate data to thd_done CSRs
LOCAL_CSR_RD, LOCAL_CSR_WR	Read and write CSRs
R_FIFO_RD	Read the receive FIFO
PCI_DMA	Issue a request on the PCI bus
SCRATCH	Scratchpad memory request
SDRAM	SDRAM reference
SRAM	SRAM reference
T_FIFO_WR	Write to transmit FIFO
Local Register Instructions	
FIND_BST, FIND_BSET_WITH_MASK	Find first 1 bit in a value
IMMED	Load immediate value and sign extend
IMMED_B0, IMMED_B1, IMMED_B2, IMMED_B3	Load immediate byte to a field
IMMED_W0, IMMED_W1	Load immediate word to a field
LD_FIELD, LD_FIELD_W_CLR	Load byte(s) into specified field(s)
LOAD_ADDR	Load instruction address
LOAD_BSET_RESULT1, LOAD_BSET_RESULT2	Load the result of find_bset
Miscellaneous Instructions	
CTX_ARB	Perform context swap and wake on event
NOP	Skip to next instruction
HASH1_48, HASH2_48, HASH3_48	Perform 48-bit hash function 1, 2, or 3
HASH1_64, HASH2_64, HASH3_64	Perform 64-bit hash function 1, 2, or 3

Microengine View Of Memory

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

Five-Stage Instruction Pipeline

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

Example Of Pipeline Execution

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

- 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

Time

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

Sources Of Delay

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

Memory Access Delays

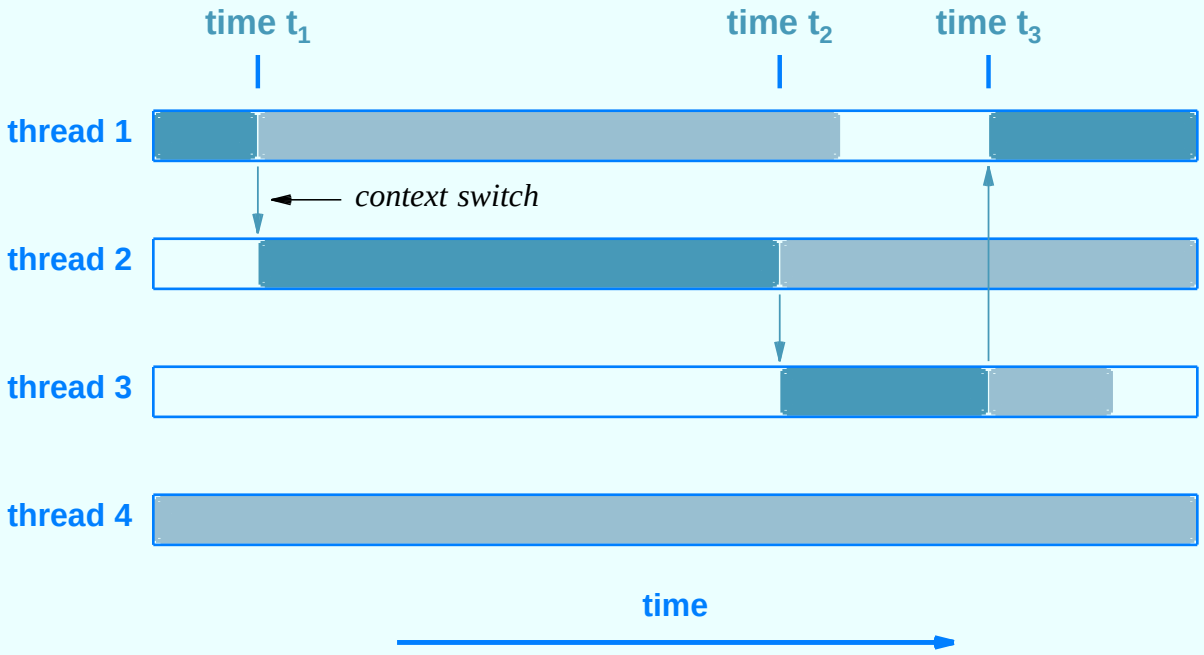
Type Of Memory	Approximate Access Time (in clock cycles)
Scratchpad	12 - 14
SRAM	16 - 20
SDRAM	32 - 40

- 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*

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
- IXP1200 offers zero-overhead context switch

Microengine Instruction Store

- Private instruction store per microengine
- Advantage: no contention
- Disadvantage: small (1024 instructions)

General-Purpose Registers

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

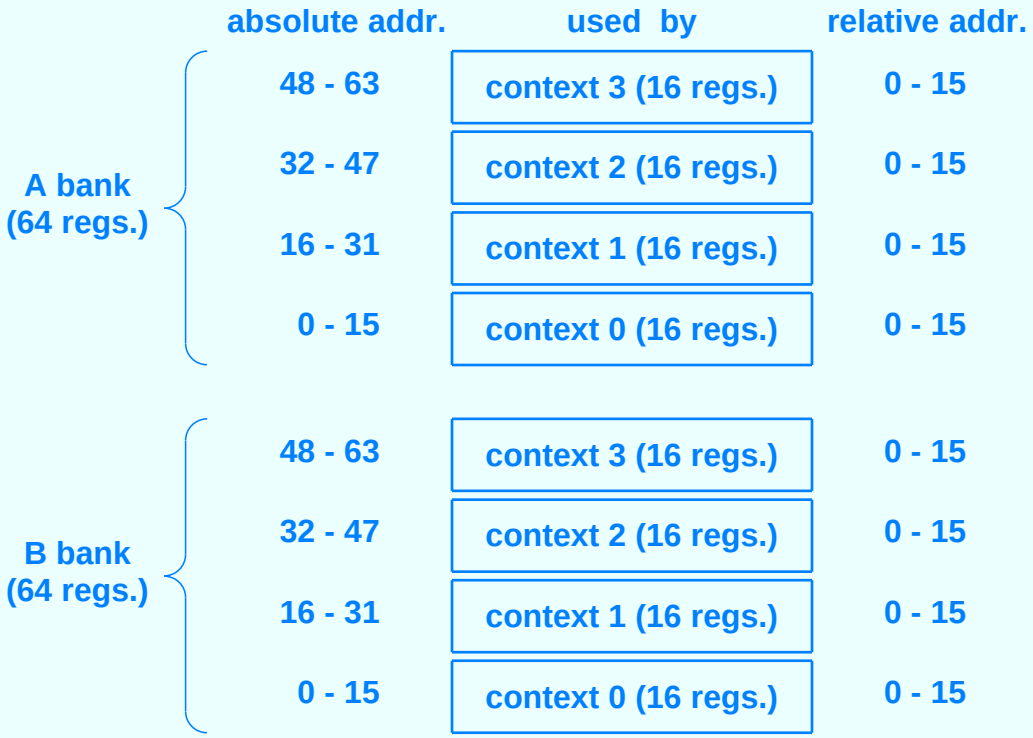
Forms Of Addressing

- Absolute
  - Entire set available
  - Uses integer from 0 to 127
- Context-relative
  - One quarter 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



- 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
- One hundred twenty-eight per microengine
- Divided into four types
  - SRAM or SDRAM
  - Read or write

Transfer Register Addresses

	absolute addr.	used by	relative addr.
SDRAM read (32 regs.)	24 - 31	context 3 (8 regs.)	0 - 7
	16 - 23	context 2 (8 regs.)	0 - 7
	8 - 15	context 1 (8 regs.)	0 - 7
	0 - 7	context 0 (8 regs.)	0 - 7
SDRAM write (32 regs.)	24 - 31	context 3 (8 regs.)	0 - 7
	16 - 23	context 2 (8 regs.)	0 - 7
	8 - 15	context 1 (8 regs.)	0 - 7
	0 - 7	context 0 (8 regs.)	0 - 7
SRAM read (32 regs.)	24 - 31	context 3 (8 regs.)	0 - 7
	16 - 23	context 2 (8 regs.)	0 - 7
	8 - 15	context 1 (8 regs.)	0 - 7
	0 - 7	context 0 (8 regs.)	0 - 7
SRAM write (32 regs.)	24 - 31	context 3 (8 regs.)	0 - 7
	16 - 23	context 2 (8 regs.)	0 - 7
	8 - 15	context 1 (8 regs.)	0 - 7
	0 - 7	context 0 (8 regs.)	0 - 7

Local Control And Status Registers

- Used to interrogate or control the IXP1200
- All mapped into StrongARM address space
- Microengine can only access its own local CSRs

Local CSRs

Local CSR	Purpose
USTORE_ADDRESS	Load the microengine control store
USTORE_DATA	Load a value into the control store
ALU_OUTPUT	Debugging: allows StrongARM to read GPRs and transfer registers
ACTIVE_CTX_STS	Determine context status
ENABLE_SRAM_JOURNALING	Debugging: place journal in SRAM
CTX_ARB_CTL	Context arbiter control
CTX_ENABLES	Debugging: enable a context
CC_ENABLE	Enable condition codes
CTX_*n*_STS	Determine context status
CTX_*n*_SIG_EVENTS	Determine signal status
CTX_*n*_WAKEUP_EVENTS	Determine which wakeup events are currently enabled

Note: *n* is digit from 0 through 3 (hardware contains a separate CSR for each of the four contexts).

Interprocessor Communication Mechanisms

- Thread-to-StrongARM communication
  - Interrupt
  - Signal event
- Thread-to-thread communication within one IXP1200
  - Signal event
- Thread-to-thread communication across multiple IXP1200s
  - Ready bus

FBI Unit

- Interface between processors and other functional units
  - Scratchpad memory
  - Hash unit
  - FBI control and status registers
  - Control and operation of the Ready bus
  - Control and operation of the IX bus
  - Data buffers that hold data arriving from the IX bus
  - Data buffers that hold data sent to the IX bus
- Operates like DMA controller
- Uses FIFOs

FIFOs

- Hardware buffers
- Transfer performed in 64 byte blocks
- Misleading name: access is random
- Only path between external device and microengine
- Receive FIFO (RFIFO)
  - Handles input
  - Physical interface moves data to RFIFO
  - FBI moves data from RFIFO to memory

FIFOs
(continued)

- Transmit FIFO (TFIFO)
  - Handles output
  - FBI moves data from memory to TFIFO
  - Physical interface extracts data from TFIFO

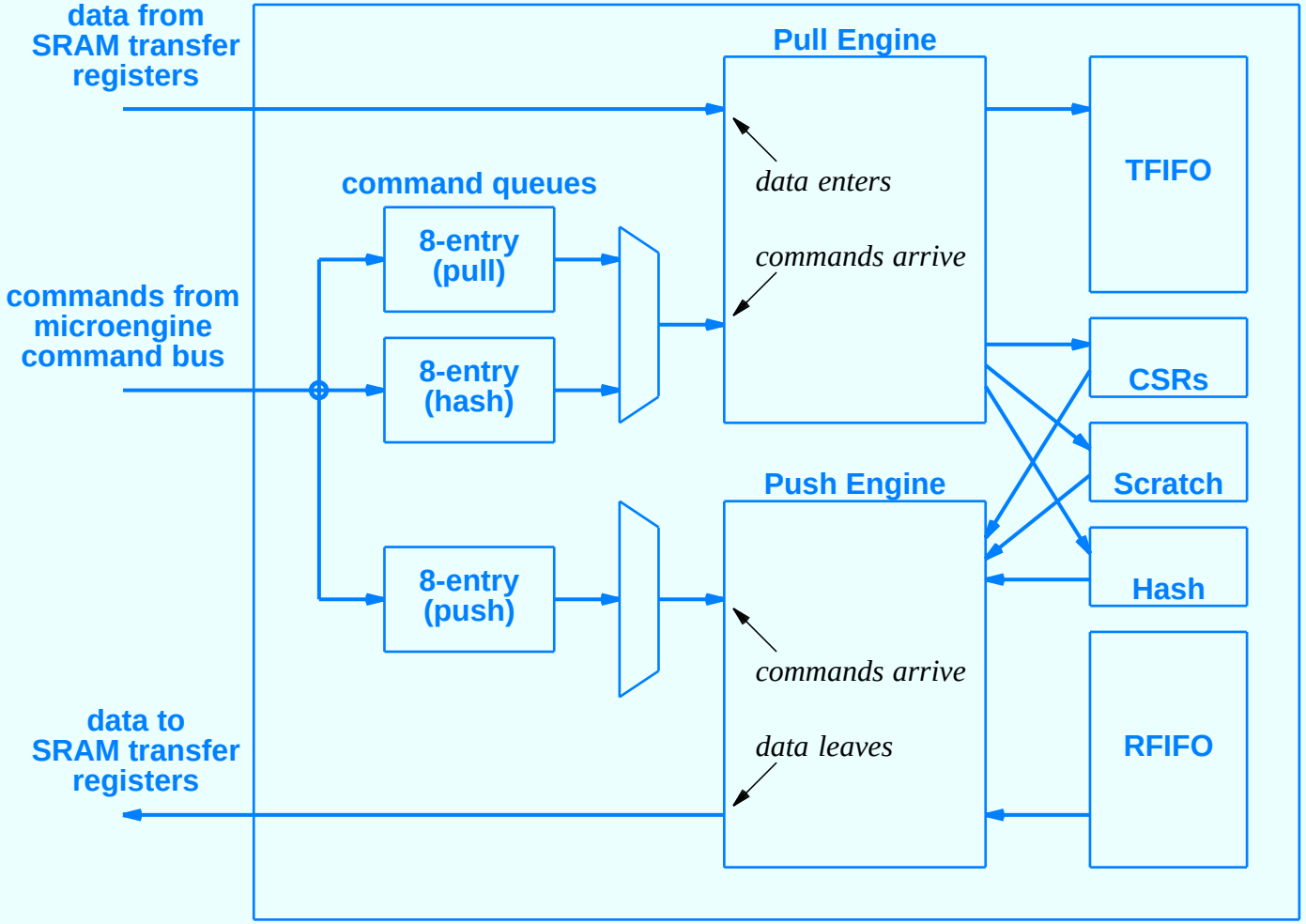
A Note About FIFO Transfer

It is possible for a microengine to transfer data to or from a FIFO without going to memory.

FBI

- Initiates and controls data transfer
- Contains active components
  - Push engine
  - Pull engine

FBI Architecture (simplified)



ScratchPad Memory

- Organized into 1K words of 4 bytes each
- Offers special facilities
  - Test-and-set of individual bits
  - Autoincrement (low latency)

Hash Unit

- Configurable coprocessor
- Operates asynchronously
- Intended for table lookup when multiplication or division required (ALU does not have multiply instruction)

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
- Two values for G — one for 48-bit hash, one for 64-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

$A = 800000000001_{16} \quad (x^{47} + 1)$

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

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

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

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

Hash Example
(continued)

- Multiplying yields

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

- Furthermore

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

- Where

$$Q = x^{8} + x^{2} + x^{1}$$

Hash Example
(continued)

- So, Q is:

$$106_{16}$$

- And R is:

$$x^{47} + x^{44} + x^{38} + x^{37} + x^{33} + x^{27} + x^{26} + x^{18} + x^{12} +$$

$$x^{11} + x^{9} + x^{8} + x^{3} + x^{1} + 1$$

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

$$90620C041B0B_{16}$$

Other IXP1200 Hardware

- The IXP1200 contains over 1500 registers used for
  - Configuration and bootstrapping
  - Control of functional units and buses
  - Checking status of processors, threads, and onboard functional units
- Example: IX bus registers used to
  - Control bus operation
  - Configure the bus for 64-bit or 32-bit mode operation
  - Control devices attached to the bus
  - Specify whether a MAC device or the IXP handles control signals

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 FIFOs for I/O
  - Do not have multiply, but have access to hash unit


Questions?

XXIV

Microengine Programming I

Microengine Code

- Many low-level details
- Close to hardware
- Written in 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 token

- Interpretation of *token* 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
```

Example Operand Syntax

- Instruction *alu* invokes the ALU

alu [*dst*, src_1, *op*, src_2]

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

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 token

Memory Operations
(continued)

- General forms

sram [*direction*, *xfer_reg*, $addr_1$, $addr_2$, *count*], *optional_token*

sdram [*direction*, *xfer_reg*, $addr_1$, $addr_2$, *count*], *optional_token*

scratch [*direction*, *xfer_reg*, $addr_1$, $addr_2$, *count*], *optional_token*

Memory Addressing

- Specified with operands $addr_1$ and $addr_2$
- Each operand corresponds to register
- Use of two operands allows
  - Base + offset
  - Scaling to large memory

Immediate Instruction

- Place constant in thirty-two bit register

immed [*dst*, *ival*, *rot*]

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

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

Register Names

- Usually automated by assembler
- Directives available for manual assignment

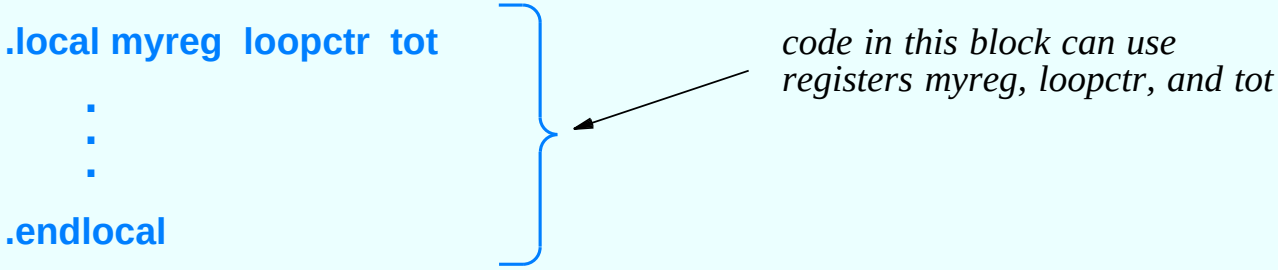
Directive	Type Of Register Assigned
.areg	General-purpose register from the A bank
.breg	General-purpose register from the B bank
.$reg	SRAM transfer register
.$$reg	SDRAM transfer register

Automated Register Assignment

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

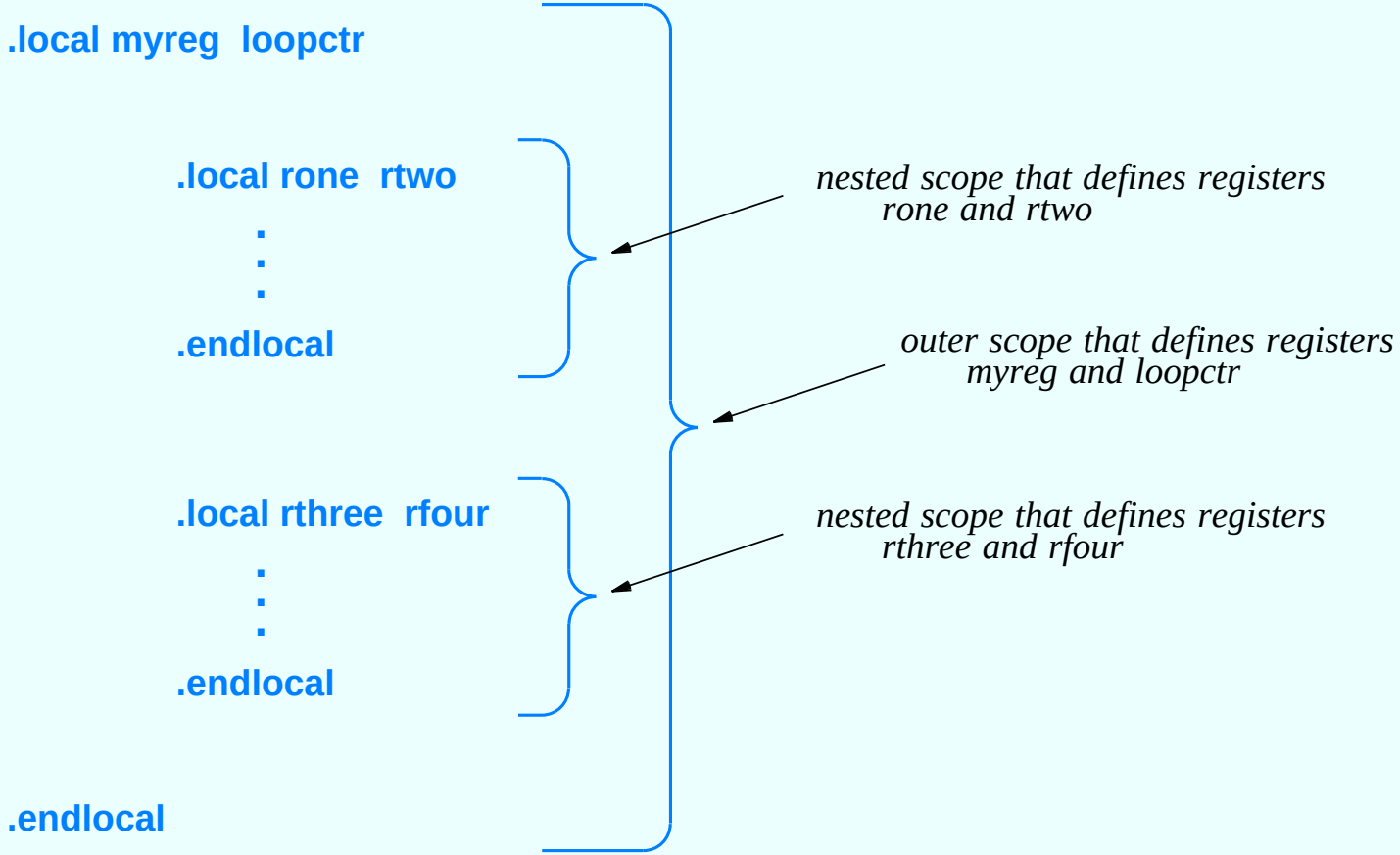
Illustration Of Automated Register Naming

- One or more register names specified on .local
- Example



- Names valid only within scope
- Scopes can be nested

Illustration Of Nested Register Scope



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
#include	Include a file
#define	Define symbolic constant
#define_eval	Eval arithmetic expression & define constant
#undef	Remove previous definition
#macro	Start macro definition
#endm	End a macro definition
#ifdef	Start conditional compilation if defined
#ifndef	Start conditional compilation if not defined
#if	Start conditional compilation if expr. true
#else	Else part of conditional compilation
#elif	Else if part of conditional compilation
#endif	End of conditional compilation
#for	Start definite iteration
#while	Start indefinite iteration
#repeat	Start indefinite iteration (test after)
#endloop	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]
        .local tmp
              alu[tmp, c, +, 5]
              alu[a, b, +, tmp]
        .endlocal
#endm
```

- Assumes values a, b, and c in registers

Macro Expansion Example

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

```
.local tmp
      alu[tmp, var3, +, 5]
      alu[var1, var2, +, tmp]
.endlocal
```

- Warning: can generate illegal code

Generation Of Repeated Code

- Macro preprocessor
  - Supports *#while* statement for iteration
  - Uses *#define_eval* for arithmetic evaluation
- Can be used to generate repeated code

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 al la C

Directive	Meaning
.if	Conditional execution
.elif	Terminate previous conditional execution and start a new conditional execution
.else	Terminate previous conditional execution and define an alternative
.endif	End .if conditional
.while	Indefinite iteration with test before
.endw	End .while loop
.repeat	Indefinite iteration with test after
.until	End .repeat loop
.break	Leave a loop
.continue	Skip to next iteration of loop

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, SDRAM, FBI, PCI, etc.

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
- Thread ID of the hardware thread (i.e., thread to signal upon completion)

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 transfer registers

```
.local $reg1 $reg2 $reg3 $reg4
.xfer_order $reg1 $reg2 $reg3 $reg4
```

Summary

- Microengines programmed in assembly language
- Intel's assembler provides
  - Directives for structuring code
  - Macro preprocessor
  - Automated register assignment


Questions?

XXV

Microengine Programming II

Specialized Memory Operations

- Buffer pool manipulation
- Processor coordination via bit testing
- Atomic memory increment
- Processor coordination via memory locking

Buffer Pool Manipulation

- SRAM facility
- Eight linked lists
- Operations *push* and *pop*
- General form of pop

sram [pop, *$xfer*, ––, ––, *listnum*]

Processor Coordination Via Bit Testing

- Provided by SRAM and Scratchpad memories
- Atomic test-and-set
- Mask used to specify bit in a word
- General form

scratch [bit_wr, *$xfer*, $addr_1$, $addr_2$, *op*]

Bit Manipulation Operations

Operation	Meaning
set_bits	Set the specified bits to one
clear_bits	Set the specified bits to zero
test_and_set_bits	Place the original word in the read transfer register, and set the specified bits to one
test_and_clear_bits	Place the original word in the read transfer register, and set the specified bits to zero

Atomic Memory Increment

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

scratch [incr, – –, $addr_1$, $addr_2$, 1]

Processor Coordination Via Memory Locking

- Word of memory acts as mutual exclusion lock
- Achieved with *memory lock* instruction
- Single *read_lock* instruction
  - Obtains a lock on location *X* in memory
  - Reads values starting at location *X*
- Single *write_unlock* instruction
  - Writes values to memory
  - Unlocks the specified location

Processor Coordination Via Memory Locking (continued)

- General form of read_lock

  sram [read_lock, *$xfer*, $addr_1$, $addr_2$, *count*], ctx_swap

- Token *ctx_swap* required (thread blocks until lock obtained)
- General form of write_unlock

  sram [unlock, −−, $addr_1$, $addr_2$, 1]

Implementation Of Memory Locking

- Achieved with a CAM that holds eight items
- Only releases contiguous items in CAM
- Consequences
  - At most eight addresses can be locked at any time
  - Thread can remained blocked even if its request can be satisfied

Control And Status Registers

- Over one hundred fifty
- Provide access to hardware units on the chip
- Allow processors to
  - Configure
  - Control
  - Interrogate
  - Monitor
- Access
  - StrongARM: mapped into address space
  - Microengines: special instructions

Csr Instruction

- Used on microengines
- General form

csr [*cmd, $xfer, CSR, count*]

- Alternatives
  - Instruction *fast_wr* provides access to subset of CSRs through the FBI unit
  - Instruction *local_csr_rd* provides access to local CSRs

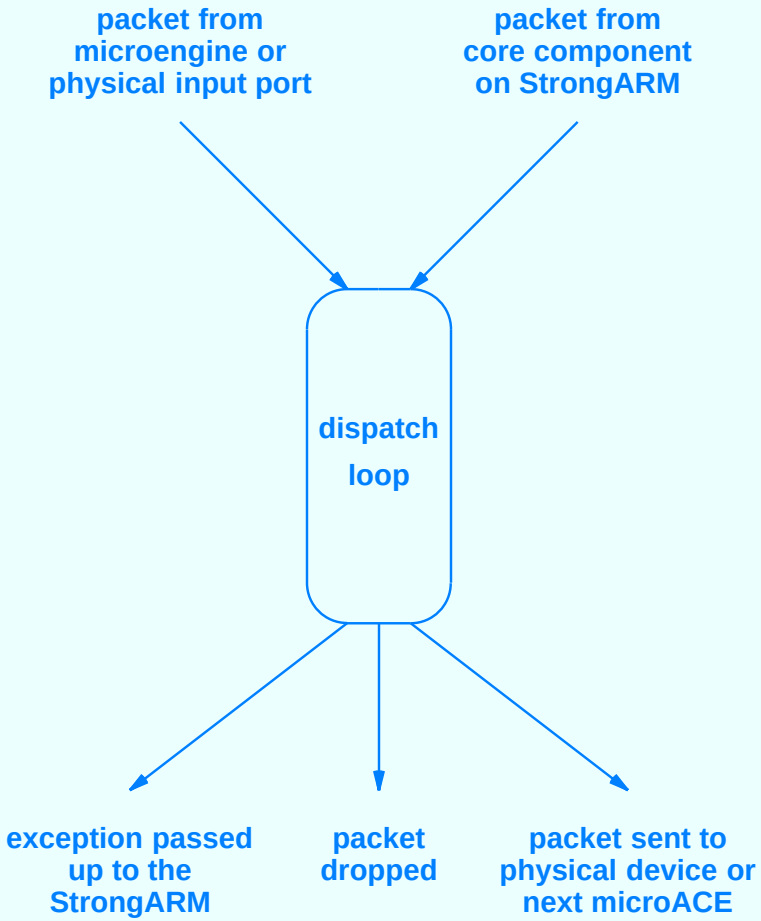
Intel Dispatch Loop Macros

- Each microengine executes *event loop*
  - Analogous to core event loop
  - Events are low level (e.g., hardware device becomes ready)
  - Known as *dispatch loop*
- SDK includes predefined macros related to dispatch loop

Predefined Dispatch Loop Macros

Macro	Purpose
Buf_Alloc	Allocate a packet buffer
Buf_GetData	Get the SDRAM address of a packet buffer (note: the name is misleading)
DL_Drop	Drop a packet and recycle the buffer
DL_GetBufferLength	Compute the length (in bytes) of the packet portion of a buffer
DL_GetBufferOffset	Compute the offset of packet data within a buffer
DL_GetInputPort	Obtain the input port over which the packet arrived
DL_GetOutputPort	Find the port over which the packet will be sent
DL_GetRxStat	Obtain the receive status of the packet
DL_Init	Initialize the dispatch loop macros
DL_MESink	Send a packet to next microblock group
DL_SASink	Send a packet to the StrongARM
DL_SASource	Receive a packet from the StrongARM
DL_SetAceTag	Specify the microblock that is handling a packet so the StrongARM will know
DL_SetBufferLength	Specify the length of packet data in the buffer
DL_SetBufferOffset	Specify the offset of packet data in the buffer
DL_SetExceptionCode	Specify the exception code for the StrongARM
DL_SetInputPort	Specify the port over which the packet arrived
DL_SetOutputPort	Specify the port on which the packet will be sent
DL_SetRxStat	Specify the receive status of the packet

Illustration Of Packet Flow



Packet Selection Macros

- Also supplied by SDK
- Allow selection of packet from queues
- Two approaches
  - Round-robin from a set of queues
    * Used for 10 / 100 Ethernet
    * Macro is *RoundRobinSource*
  - Strict FIFO ordering
    * Used for Gigabit Ethernet
    * Macro is *FifoSource*

Obtaining Packets From StrongARM

- Microengine can receive packets from
  - Input port (ingress) or another microengine
  - StrongARM
- Dispatch loop tests for each possibility
- Packets from StrongARM should have lower priority

Implementation Of Priority

- Macro *EthernetIngress* obtains Ethernet frame
- Macro *DL_SASource* obtains packet from StrongARM
- Each returns *IX_BUFFER_NULL*, if no packet waiting
- To achieve priority, *DL_SASource* counts *SA_CONSUME_NUM* times before returning a packet

Packet Disposition

- Auxiliary variable used
- Dispatch loop finds a packet and calls macro *X* to process
- Macro *X* sets variable *dl_buffer_next*
- Dispatch loop examines *dl_buffer_next* and invokes
  - *DL_Drop* to drop packet
  - *DL_SASink* to send to StrongARM
  - *DL_MESink* to send to ‘‘next’’ ACE

Other Predefined Macros

- Macros needed for
  - Buffer manipulation
  - Packet processing
  - Other tasks
- Predefined macros available

Example Code To Process Packet Header

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

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

        /* Compute the SDRAM address of the data buffer */
        Buf_GetData[base, dl_buffer_handle]

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

        /* Convert the byte offset to SDRAM words by dividing by eight */
        /* (shift right by three bits) */
        alu_shf[offset, --, B, offset, >>3]

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

Example Code To Process Packet Header (continued)

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

/* Process the packet header in the SDRAM transfer registers
/* starting at register $$hdr */
. . .

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

Using Intel Dispatch Macros

- Programmer must perform five steps
  - Define three symbolic constants
  - Declare registers with a *.local* directive
  - Use a *.import_var* directive to name tag values (optional)
  - Include the macros pertinent to the microblock
  - Initialize the macros as the first part of a dispatch loop
- Note: three constants must be defined before macros are included
  - SA_CONSUME_NUM
  - IX_EXCEPTION
  - SEQNUM_IGNORE

Required Register Declarations

- The following declarations are required

```
.local  dl_reg1 dl_reg2 dl_reg3 dl_buffer_handle dl_next_block
```

Naming Tag Values

- Required in microblock that sends packets (exceptions) to core component
- Uses .*import_var* directive
- Example

```
.import_var IPV4_TAG
```

Including Intel Macros

- Must include two files
  - DispatchLoop_h.uc
  - DispatchLoopImportVars.h
- Ingress microblock includes
  - EthernetIngress.uc
- Egress microblock includes
  - EthernetEgress.uc

Initialization Of Intel Macros

- Program calls *DL_Init* to perform initialization
- Typical initialization sequence

```
DL_Init[ ]
EthernetIngress_Init[ ]
. . .     /* Other microblock initialization calls */
```

Packet I/O

- Physical frame divided into sixty-four octet units for transfer
- Each unit known as *mpacket*
- Division performed by interface hardware
- Microengine transfers each mpacket separately
- Header uses two bits to specify position of mpacket
  - Start Of Packet (SOP) set for first mpacket of frame
  - End Of Packet (EOP) set for last mpacket of frame
- Note: cell can have both SOP and EOP set

Packet I/O
(continued)

- No interrupts
- Dispatch loop uses polling
  - Ready Bus Sequencer checks devices and sets bit
  - Dispatch loop tests bit

Ingress Packet Transfer

- Incoming mpacket placed in Receive FIFO (RFIFO)
- Microengine can transfer from RFIFO to
  - SRAM transfer registers
  - Directly into SDRAM
- SDRAM transfer has form

sdram [r_fifo_rd, *$$xfer*, $addr_1$, $addr_2$, *count*], indirect_ref

Packet Egress

- Steps required
  - Reserve space in Transmit FIFO (TFIFO)
  - Copy mpacket from memory into TFIFO
  - Set SOP and EOP bits
  - Set *valid* flag in *XMIT_VALIDATE* register
- General form used to copy from SDRAM to TFIFO

sdram [t_fifo_wr, *$$xfer*, $addr_1$, $addr_2$, *count*], indirect_ref

Setting The Valid Flag In A XMIT_VALIDATE Register

- Valid bit is in CSR
- Use *fast_wr* instruction to access

```
fast_wr [ 0, xmit_validate ], indirect_ref
```

Other Details

- Microengine must check status of mpacket to determine if
  - MAC hardware detected problem (e.g., bad CRC)
  - Mpacket arrived with no problems
- Information found in *RCV_CNTL* CSR

Summary

- Special operations used for
  - Synchronization
  - Memory access
  - 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
- Intel supplies large set of dispatch loop macros

Summary
(continued)

- Many details required to use Intel dispatch macros
- Packet I/O performed on sixty-four byte units called *mpackets*
- Mpacket can be transferred from RFIFO to
  - SRAM transfer registers
  - SDRAM
- Many details required to perform trivial operations on packet


Questions?

XXVI

An Example ACE

We Will

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

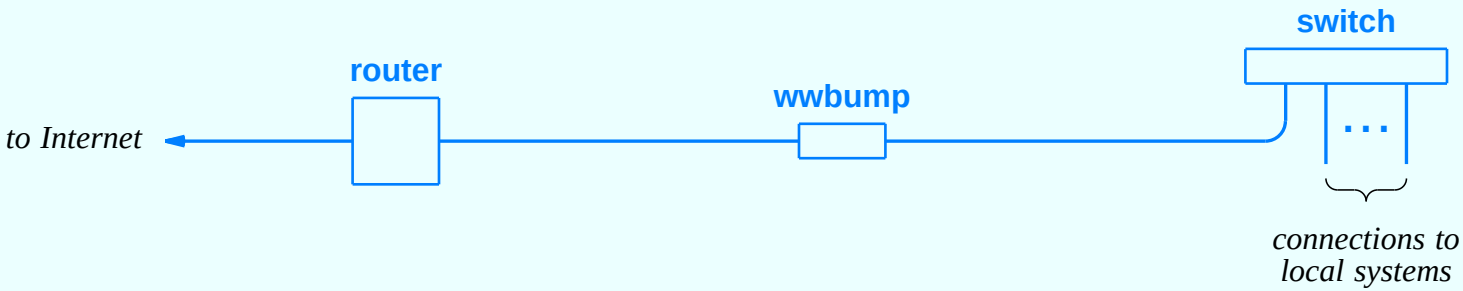
Choice Of Network System

- Used to demonstrate
  - Basic concepts
  - Code structure and organization
- Need to
  - Minimize code size and complexity
  - Avoid excessive detail
  - Ignore performance optimizations

The Example

- Trivial network system
- In-line paradigm using two ports
  - Ethernet-to-Ethernet connectivity
  - Known as *bump-in-the-wire*
- Count Web packets
  - Frame carries IP
  - Datagram carries TCP
  - Destination port is 80
- Named *WWBump* (*Web Wire Bump*)

Illustration Of WWBump

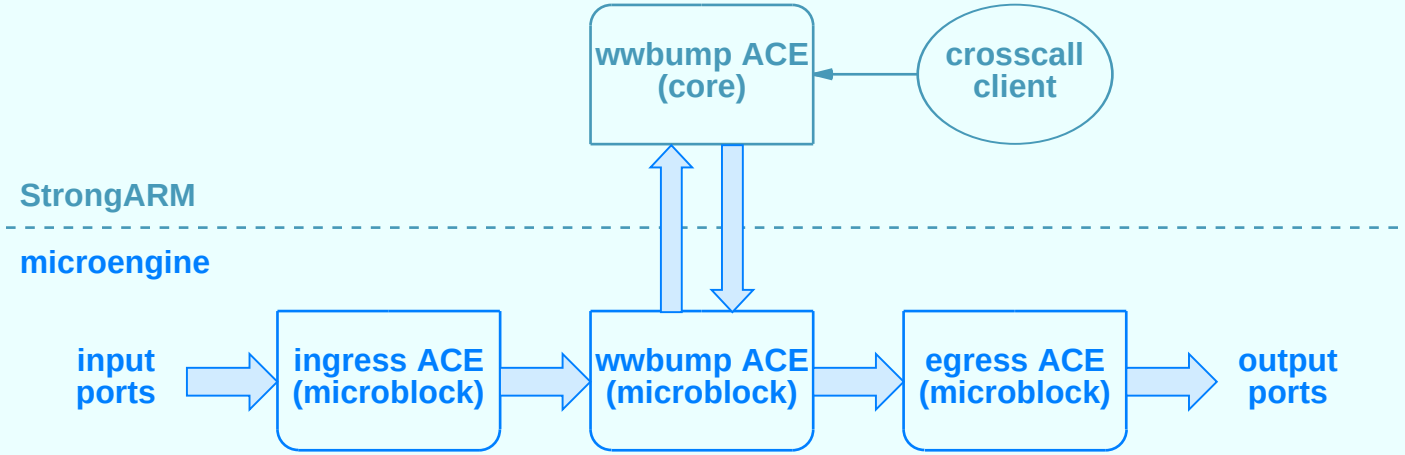


- Passes *all* traffic in either direction
- Counts Web packets

Design

- Code written for *bridal veil* testbed
- Accepts input from either port zero or port one
- Forwards incoming traffic to opposite port
- Uses the ingress and egress library ACEs supplied by Intel
- Defines a wwbump MicroACE
- Passes each Web packet to the core component (exception)
- Provides access to current packet count via the crosscall
- Uses an *ixsys.config* file to specify binding of targets

Organization Of ACEs



Static ACE Data Structure

- Intel software requires
  - Static control block for ACE
  - Structure is named (*ix_ace*)
- Programmer can add additional fields to control block

Fields In The Wwbump Control Block

Field	Type	Purpose
name	string	Name of the ACE in ixsys.config
tag	int	ID of ACE assigned by Resource Manager
ue	int	Bitmask of microengines running the ACE
bname	string	TAG name that the microblock uses
ccbase	ix_base_t	Handle for crosscall service

Example Declarations (wwbump.h)

```
/* wwbump.h -- global constants and declarations for wwbump */

#ifndef __WWBUMP_H
#define __WWBUMP_H

#include <ix/asl.h>
#include <ix/microace/rm.h>
#include <ix/asl/ixbasecc.h>
#include <wwbump_import.h>

typedef struct wwbump wwbump;
struct wwbump
{
        ix_ace  ace;        /* Mandatory handle for all ACEs            */

        char    name[128];  /* Name of this ACE given in ixsys.config   */
        int     tag;        /* ID assigned by RM for SA-to-ME communic. */
        int     ue;         /* Bitmask of MEs this MicroACE runs on     */
        char    bname[128]; /* Block name (need .import_var $bname_TAG) */
        ix_base_t ccbase;   /* Handle for cross call service            */
};

/* Exception handler prototype */
int     exception(void *ctx, ix_ring r, ix_buffer b);

#endif /* __WWBUMP_H */
```

Shared Symbolic Constants

- Core component programmed in C
- Microblock component programmed in assembly language
- Typical trick: bracket lines of code with #ifndef to prevent multiple instantiations
- Unusual trick
  - Both languages share C-preprocessor syntax
  - Can place all constant definitions in shared file

Distinguishing Code At Compile Time

- Preprocessor can distinguish
  - Code for core component
  - Code for microblock component
- Technique: symbolic constant *MICROCODE* only defined when assembling microcode
- Can use *#ifdef*

Example File Of Symbolic Constants (wwbump_import.h)

```
/* wwbump_import.h -- define or import tag for wwbump MicroACE */

#ifndef _WWBUMP_IMPORT_H
#define _WWBUMP_IMPORT_H

#ifdef MICROCODE

.import_var WWBUMP_TAG

#else

#define WWBUMP_TAG_STR          "WWBUMP_TAG"

#endif

#endif /* _WWBUMP_IMPORT_H */
```

Division Of Microcode Into Files

- Two main files
- Follow Intel naming convention: macro and file names related to microcode start with uppercase letter
- WWBump.uc
  - Code for packet processing
  - Defines macro WWBump
- WWB_dl.uc
  - Contains code for dispatch loop

WWBump Macro

- Performs two functions
  - Specifies eventual output port for packet
  - Classifies the packet
- Classification
  - Represented as disposition
    - StrongARM (exception)
    - Next microblock (Egress)
  - Stored in register *dl_next_block*

WWBump Details

- Uses Intel *ingress* macro
- Calls *DL_GetInputPort* to determine input port (0 or 1)
- Calls *DL_SetOutputPort* to specify output port
- When passing packet to core component
  - Set exception to *WWBUMP_TAG*
  - Set return code to *IX_EXCEPTION*

Steps WWBump Takes During Classification

- Read packet header into six SDRAM transfer registers
- Check
  - Ethernet type field (0800_{16})
  - IP type field (6)
  - TCP destination port (80)
- Difficult (ugly) in microcode
  - Must use IP header length to calculate port offset
  - Port number is 16 bits; transfer register is 32

File WWBump.uc (part 1)

```
/* WwBump.uc - microcode to process a packet */

#define ETH_IP          0x800           ; Ethernet type for IP
#define IPT_TCP         6               ; IP type for TCP
#define TCP_WWW         80              ; Dest. port for WWW

#macro WwBumpInit[]
        /* empty because no initialization is needed */
#endm

#macro WwBump[]
    xbuf_alloc[$$hdr,6]         ; Allocate 6 SDRAM registers

    /* Reserve a register (ifn) and compute the output port for the     */
    /* frame; a frame that arrives on port 0 will go out port 1, and    */
    /* vice versa                                                       */

    .local ifn
        DL_GetInputPort[ifn]            ; Copy input port number to ifn
        alu [ ifn, ifn, XOR, 1 ]        ; XOR with 1 to reverse number
        DL_SetOutputPort[ifn]           ; Set output port for egress
    .endlocal
```

File WWBump.uc (part 2)

```
/* Read first 24 bytes of frame header from SDRAM */

.local base off
    Buf_GetData[base, dl_buffer_handle] ; Get the base SDRAM address
    DL_GetBufferOffset[off]         ; Get packet offset in bytes
    alu_shf[off, --, B, off, >>3]   ; Convert to Quad-words
    sdram[read, $$hdr0, base, off, 3], ctx_swap ; Read 3 Quadwords
                                                ;  (six registers)
.endlocal

/* Classify the packet.  If any test fails, branch to NotWeb#       */

/* Verify frame type is IP (1st two bytes of the 4th longword)      */

.local etype
    immed[etype, ETH_IP]
    alu_shf[ --, etype, -, $$hdr3, >>16]    ; 2nd operand is shifted
    br!=0[NotWeb#]
.endlocal

/* Verify IP type is TCP (last byte of the 6th longword)            */

br!=byte[$$hdr5, 0, IPT_TCP, NotWeb#]
```

File WWBump.uc (part 3)

```
/* Verify destination port is web (offset depends on IP header size */

.local base boff dpoff dport

    /* Get length of IP header (3rd byte of 4th longword), and      */
    /* convert to bytes by shifting by six bits instead of eight    */

    ld_field_w_clr[dpoff, 0001, $$hdr3, >>6] ; Extract header length

    /* Mask off bits above and below the IP length                  */

    .local mask
        immed[mask, 0x3c]           ; Mask out upper and lower 2 bits
        alu [ dpoff, dpoff, AND, mask ]
    .endlocal
```

File WWBump.uc (part 4)

```
/* Register dpoff contains the IP header length in bytes.  Add  */
/* Ethernet header length (14) and offset of the destination    */
/* port (2) to obtain offset from the beginning of the packet   */
/* of the destination port.   Add to SDRAM address of buffer,   */
/* and convert to quad-word offset by dividing by 8 (shift 3).  */

alu[dpoff, dpoff, +, 16]                ; Add Ether+TCP offsets
Buf_GetData[base, dl_buffer_handle]     ; Get buffer base address
DL_GetBufferOffset[boff]                ; Get data offset in buf.
alu[boff, boff, +, dpoff]               ; Compute byte address
alu_shf[boff, --, B, boff, >>3]         ; Convert to Q-Word addr.
sdram[read, $$hdr0, base, boff, 1], ctx_swap    ; Read 8 bytes

/* Use lower three bits of the byte offset to determine which   */
/* byte the destination port will be in.  If value >= 4, dest.  */
/* port is in the 2nd longword; otherwise it's in the first.    */

alu[ dpoff, dpoff, AND, 0x7 ]           ; Get lowest three bits
alu[ --, dpoff, -, 4]                   ; Test and conditional
br>=0[SecondWord#]                      ;   branch if value >=4
```

File WWBump.uc (part 5)

```
FirstWord#:     /* Load upper two bytes of register $$hdr0 */
        ld_field_w_clr[dport, 0011, $$hdr0, >>16] ; Shift before mask
        br[GotDstPort#]                           ; Check port number

SecondWord#:    /* Load lower two bytes of register $$hdr1 */

        ld_field_w_clr[dport, 0011, $$hdr1, >>16] ; Shift before mask

GotDstPort#:    /* Verify destination port is 80 */

        .local wprt
            immed[wprt, TCP_WWW]                  ; Load 80 in reg. wprt
            alu[--, dport, -, wprt]               ; Compare dport to wprt
            br!=0[NotWeb#]                        ;  and branch if not equal
        .endlocal
    .endlocal
```

File WWBump.uc (part 6)

```
IsWeb#:         /* Found a web packet, so send to the StrongARM */

    /* Set exception code to zero (we must set this)           */
    .local exc                                  ; Declare register exc
        immed[exc, 0]                           ; Place zero in exc and
        DL_SetExceptionCode[exc]                ;  set exception code
    .endlocal

    /* Set tag core component's tag (required by Intel macros) */
    .local ace_tag                              ; Declare register ace_tag
        immed32[ace_tag, WWBUMP_TAG]            ; Place wwbump tag in reg.
        DL_SetAceTag[ace_tag]                   ;  and set tag
    .endlocal

    /* Set register dl_next_block to IX_EXCEPTION to cause dispatch    */
    /*   to pass packet to StrongARM as an exception                   */
    immed[dl_next_block, IX_EXCEPTION]          ; Store return value
    br[Finish#]                                 ; Done, so branch to end

NotWeb#:        /* Found a non-web packet, so forward to next microblock*/
    immed32[dl_next_block, 1]                   ; Set return code to 1
```

File WWBump.uc (part 7)

```
Finish#:        /* Packet processing is complete, so clean up          */
xbuf_free[$$hdr]                                ; Release xfer registers
#endm
```

Dispatch Loop Algorithm

```
initialize dispatch loop macros;
do forever  {
        if (packet has arrived from the StrongARM)
                Send the packet to egress microblock;
        if (packet has arrived from Ethernet port) {
                Invoke WWBump macro to process the packet;
                if (return code specifies exception) {
                        Send packet to StrongARM;
                } else {
                        Send packet to egress microblock;
                }
        }
}
```

Dispatch Loop Code

- Contained in file *WWB_dl.uc*
- Not defined as separate macro
- Includes several files
  - Standard (Intel) header files to define basic macros
  - Our packet processing macro (file WWBump.uc)

File WWB_dl.uc (part 1)

```
/* WwB_dl.uc - dispatch loop for wwbump program */

/* Constants */

#define IX_EXCEPTION    0       ; Return value to raise an exception
#define SA_CONSUME_NUM  31      ; Ignore StrongARM packets 30 of 31 times
#define SEQNUM_IGNORE   31      ; StrongARM fastport sequence num

/* Register declarations (as required for Intel dispatch loop macros)   */
.local  dl_reg1 dl_reg2 dl_reg3 dl_reg4 dl_buffer_handle dl_next_block

/* Include files for Intel dispatch loop macros */
#include "DispatchLoop_h.uc"
#include "DispatchLoopImportVars.h"
#include "EthernetIngress.uc"
#include "wwbump_import.h"

/* Include the packet processing macro defined previously */
#include "WWBump.uc"
```

File WWB_dl.uc (part 2)

```
/* Microblock initialization */
DL_Init[]
EthernetIngress_Init[]
WWBumpInit[]

/* Dispatch loop that runs forever */
.while(1)

Top_Of_Loop#:   /* Top of dispatch loop (for equivalent of C continue)  */

    /* Test for a frame from the StrongARM */
    DL_SASource[ ]                                  ; Get frame from SA
    alu[--, dl_buffer_handle, -, IX_BUFFER_NULL]; If no frame, go test
    br=0[Test_Ingress#], guess_branch               ;  for ingress frame
    br[Send_MB#]                                    ; If frame, go send it
```

- Macro *DLSAsource*
  - Checks for packets from StrongARM
  - Returns one packet after *SA_CONSUME_NUM* calls

File WWB_dl.uc (part 3)

```
Test_Ingress#: /* Test for an Ethernet frame */

    EthernetIngress[ ]                          ; Get an Ethernet frame
    alu[--, dl_buffer_handle, -, IX_BUFFER_NULL]; If no frame, go back
    br=0[Top_Of_Loop#]                          ;  to start of loop

    /* Check if ingress frame valid and drop if not */
    br!=byte[dl_next_block, 0, 1, Drop_Packet#]

    /* Invoke WWBump macro to set output port and classify the frame    */
    WWBump[]

    /* Use return value from WWBump to dispose of frame:                */
    /*    if exception, jump to code that sends to StrongARM            */
    /*    else jump to code that sends to egress                        */

    alu[ --, dl_next_block, -, IX_EXCEPTION]    ; Return code is exception
    br=0[Send_SA#]                              ;  so send to StrongARM

    br[Send_MB#]                                ; Otherwise, send to next
                                                ;  microblock
```

- Note *dl_next_block* specifies whether the frame should be treated as an exception or forwarded to next microblock

File WWB_dl.uc (part 4)

```
Send_SA#:
    /* Send the frame to the core component on the StrongARM as an      */
    /* exception. Note that tag and exception code are assigned by      */
    /* the microblock WWBump.                                           */
    DL_SASink[ ]
    .continue                                   ; Continue dispatch loop

Send_MB#:
    /* Send the frame to the next microblock (egress).  Note that the   */
    /* output port (field oface hidden in the internal structure) has   */
    /* been assigned by microblock WWBump.                              */
    DL_MESink[ ]
    nop
    .continue

Drop_Packet#:
        /* Drop the frame and start over getting a new frame */
        DL_Drop[ ]
.endw

nop                     ; Although the purpose of these no-ops is
nop                     ; undocumented, Intel examples include them.
nop
.endlocal
```

Code For Core Component

- Written in C
- Defines two basic functions
  - Exception handler called when packet arrives from StrongARM
  - Default action called when another frame arrives (e.g., from another ACE)

Exception Handler

- Function named *exception*
- Receives packets from microblock
- Must call Intel function *RmGetExceptionCode*
- Can call Intel function *RmSendPacket* to forward packet to a microblock
- Three arguments
  - Pointer to ACE control block
  - Ring buffer (not used in our code)
  - Pointer to ix_buffer holding packet

Default Action

- Function named *ix_action_default*
- Required part of every ACE
- Called when frame arrives from source other than microblock component
- Our version merely discards the packet
- Code in file *action.c*

File action.c (part 1)

```
/* action.c -- Core component of wwbump that handles exceptions */

#include <wwbump.h>
#include <stdlib.h>
#include <wwbcc.h>

ix_error exception(void *ctx, ix_ring r, ix_buffer b)
{
    struct wwbump *wwb = (struct wwbump *) ctx;    /* ctx is the ACE */
    ix_error e;
    unsigned char c;
    (void) r;   /* Note:  not used in our example code */


    /* Get the exception code: Note: Intel code requires this step */
    e = RmGetExceptionCode(wwb->tag, &c);
    if ( e ) {
        fprintf(stderr, "%s:  Error getting exception code", wwb->name);
        ix_error_dump(stderr, e);
        ix_buffer_del(b);
        return e;
    }
```

- Call to *RmGetExceptionCode* is required

File action.c (part 2)

```
    Webcnt++;   /* Count the packet as a web packet */

    /* Send the packet back to wwbump microblock */
    e = RmSendPacket(wwb->tag, b);
    if ( e ) {  /* If error occurred, report the error */
        ix_error_dump(stderr, e);
        ix_buffer_del(b);
        return e;
    }

    return 0;
}
```

- First line of this page
  - Performs entire ‘‘work’’ of exception handler
  - Increments global counter (*Webcnt*)
- Once packet has been counted, *RmSendPacket* enqueues packet back to microblock group

File action.c (part 3)

```
/* A core component must define an ix_action_default function that is   */
/* invoked if a frame arrives from the core component of another ACE.   */
/* Because wwbump does not expect such frames, the version of the       */
/* default function used with wwmbump simply deletes any packet that    */
/* it receives via this interface.                                      */

int ix_action_default(ix_ace * a, ix_buffer b)
{
    (void) a;                   /* This line prevents a compiler warning*/
    ix_buffer_del(b);           /* Delete the frame                     */
    return RULE_DONE;           /* This step required                   */
}
```

- Code for default action is trivial: discard any packet that arrives

Initialization And Finalization

- Additional code needed for each ACE
- Function *ix_init* performs initialization
  - Allocates memory for the ACE control block
  - Invokes Intel macros
    * *ix_ace_init* for internal initialization
    * *RmInit* to form connection to Resource Manager
    * *RmRegister* to register with Resource Manager

Initialization And Finalization (continued)

- Function *ix_fini* performs finalization
  - Invokes Intel macros
    * *cc_fini* to terminate crosscalls
    * *RmTerm* to terminate Resource Manager
    * *ix_ace_fini* to deallocate ACE control block

File init.c (part 1)

```
/* init.c -- Initialization and completion routines for the wwbump ACE  */

#include <stdlib.h>
#include <string.h>
#include <wwbump.h>
#include <wwbcc.h>

/* Initialization for the wwbump ACE */
ix_error ix_init(int argc, char **argv, ix_ace ** ap)
{
    struct wwbump *wwb;
    ix_error e;
    (void)argc;

    /* Set so ix_fini won't free a random value if ix_init fails */
    *ap = 0;

    /* Allocate memory for the WWBump structure (includes ix_ace) */
    wwb = malloc(sizeof(struct wwbump));
    if ( wwb == NULL )
        return ix_error_new(IX_ERROR_LEVEL_LOCAL, IX_ERROR_OOM, 0,
                            "couldn't allocate memory for ACE");
```

File init.c (part 2)

```
/* Microengine mask is always passed as the third argument */
wwb->ue = atoi(argv[2]);

/* Set blockname used to associate the ACE with its microblock */
strcpy(wwb->bname, "WWBUMP");

/* The name of the ACE is always the 2nd argument */
/* The first argument is the name of the executable */
wwb->name[sizeof(wwb->name) - 1] = '\0';
strncpy(wwb->name, argv[1], sizeof(wwb->name) - 1);

/* Initializes the ix_ace handle (including dispatch loop, control */
/* access point, etc) */
e = ix_ace_init(&wwb->ace, wwb->name);
if (e) {
    free(wwb);
    return ix_error_new(0,0,e,"Error in ix_ace_init()\n");
}
```

File init.c (part 3)

```
/* Initialize a connection to the resource manager */
e = RmInit();
if (e) {
    ix_ace_fini(&wwb->ace);
    free(wwb);
    return ix_error_new(0,0,e,"Error in RmInit()\n");
}

/* Register with the resource manager (including exception handler) */
e = RmRegister(&wwb->tag, wwb->bname,&wwb->ace, exception, wwb,
               wwb->ue);
if (e) {
    RmTerm();
    ix_ace_fini(&wwb->ace);
    free(wwb);
    return ix_error_new(0,0,e,"Error in RmRegister()\n");
}
```

File init.c (part 4)

```
    /* Initialize crosscalls */
    e = cc_init(wwb);
    if ( e ) {
        RmUnRegister(&wwb->tag);
        RmTerm();
        ix_ace_fini(&wwb->ace);
        free(wwb);
        return e;
    }

    *ap = &wwb->ace;
    return 0;
}

ix_error ix_fini(int argc, char **argv, ix_ace * ap)
{
    struct wwbump *wwb = (struct wwbump *) ap;
    ix_error e;
    (void)argc;
    (void)argv;

    /* ap == 0 if  ix_init() fails */
    if ( ! ap )
      return 0;
```

File init.c (part 5)

```
    /* Finalize crosscalls */
    e = cc_fini(wwb);
    if ( e )
        return e;

    /* Unregister the exception handler and microblocks */
    e = RmUnRegister(wwb->tag);
    if ( e )
        return ix_error_new(0,0,e,"Error in RmUnRegister()\n");

    /* Terminate connection with resource manager */
    e = RmTerm();
    if ( e )
        return ix_error_new(0,0,e,"Error in RmTerm()\n");

    /* Finalize the ix_ace handle */
    e = ix_ace_fini(&wwb->ace);
    if ( e )
        return ix_error_new(0,0,e,"Error in ix_ace_fini()\n");

    /* Free the malloc()ed memory */
    free(wwb);
    return 0;
}
```

The Important Point

- Wwbump performs a trivial task
- The code invokes Intel's ingress and egress ACEs
- The code is written using Intel's dispatch loop macros
- The code is large.

The Important Point

- Wwbump performs a trivial task
- The code invokes Intel's ingress and egress ACEs
- The code is written using Intel's dispatch loop macros
- The code is huge!

An Example Crosscall

- Trivial example
  - Provide access to current Web packet count
  - Wwbump ACE acts as crosscall server
  - Only exports one function
  - Can be called from non-ACE program
- Exported function named *getcnt*

Basic Steps When Building A Crosscall Facility

- Define a set of functions that the crosscall server exports
- Create an *Interface Definition Language* (*IDL*) declaration for each function, and use the IDL compiler to generate the corresponding stub code
- Write code for each exported function with arguments that match the generated stub arguments
- Write an initialization function
- Write a finalization function

Example IDL Specification

```
interface wwbump
{
    twoway long getcnt();
};
```

- IDL specification declares
  - Name and type of exported function
  - Type of each argument

Files Generated By The IDL Compiler

File	Contents	Description
*IFACE*_stub_c.h		Crosscall data types and types for initialization and finalization functions
	*IFACE*_intName	Interface name as a string
	*IFACE_FCN*_opName	Function name as a string
	stub_*IFACE*_init()	Client crosscall initialization
	stub_*IFACE*_fini()	Client crosscall finalization
	*IFACE_FCN*_fptr	Function pointer for FCN
	stub_*IFACE_FCN*()	Client-side crosscall function
	CC_VMT_*IFACE*	Crosscall Virtual Method Table structure
	deferred_cb_*IFACE_IFCN*()	Client callback prototype
	*IFACE_FCN*_cb_fptr	Client callback function pointer
	CB_VMT_*IFACE*	Client callback VMT
	getCBVMT_*IFACE*()	Find VMT for callback
*IFACE*_sk_c.h		Declarations of server-side interfaces
	sk_*IFACE*_init()	Server initialization
	sk_*IFACE*_fini()	Server finalization
	getCCVMT_*IFACE*()	Find the VMT for the interface
	invoke_*IFACE_IFCN*()	Invoke a crosscall
	*IFACE*_stub_c.h	Included file

More Files Generated By The IDL Compiler

File	Contents	Description
*IFACE*_cc_c.h		Prototypes for individual crosscall functions
	IFACE_FCN()	Prototype for function FCN
	*IFACE*_sk_c.h	Included file
*IFACE*_cb_c.h		Prototypes and Virtual Method Tables for client-side callback functions
	*IFACE_FCN*_cb()	Prototype for client callback
	*IFACE*_stub_c.h	Included file
*IFACE*_stub_c.c		Code for client-side initialization and finalization crosscall functions
IFACE_sk_c.c		Code for server-side initialization and finalization and VMT accessor functions
*IFACE*_cc_c.h		Prototypes for individual crosscall functions
	IFACE_FCN()	Prototype for function FCN
	*IFACE*_sk_c.h	Included file
*IFACE*_cb_c.h		Prototypes and Virtual Method Tables for client-side callback functions
	*IFACE_FCN*_cb()	Prototype for client callback
	*IFACE*_stub_c.h	Included file

Even More IDL-Generated Files

File	Description
*IFACE*_cc_c.c	Default code for crosscall functions that merely return an error.
*IFACE*_cb_c.c	Default implementations of client callback functions that merely return an error.

Code For A (Trivial) Crosscall Server

- Programmer must supply
  - Code for exported function(s)
  - Initialization function
  - Finalization function
  - Global variable declarations
  - Code to change pointer(s) in global *virtual method table*
- Example code in file *wwbcc.c*

File wwbcc.c (part 1)

```
/* wwbcc.c -- wwbump crosscall functions */

#include <stdlib.h>
#include <string.h>
#include <wwbump.h>
#include <wwbcc.h>
#include "wwbump_sk_c.h"
#include "wwbump_cc_c.h"


long Webcnt;            /* Stores the count of web packets */

/* Initialization function for the crosscall */

ix_error cc_init(struct wwbump *wwb)
{
    CC_VMT_wwbump *vmt = 0;
    ix_cap *capp;
    ix_error e;

    /* Initialize an ix_base_t structure to 0 */
    memset(&wwb->ccbase, 0, sizeof(wwb->ccbase));

    /* Get the OMS communications access point (CAP) of the ACE */
    ix_ace_to_cap(&wwb->ace, &capp);
```

File wwbcc.c (part 2)

```
    /* Invoke the crosscall initialization function and check for error */
    e = sk_wwbump_init(&wwb->ccbase, capp);
    if (e)
        return ix_error_new(0,0,e,"Error in sk_wwbump_init()\\n");


    /* Retarget incoming crosscalls to our getcnt function */

    /* Get a pointer to the CrossCall Virtual Method Table */
    e = getCCVMT_wwbump(&wwb->ccbase, &vmt);
    if (e)
    {
        sk_wwbump_fini(&wwb->ccbase);
        return ix_error_new(0,0,e,"Error in getCCVMT_wwbump()\\n");
    }

    /* Retarget function pointer in the table to getcnt */
    vmt->_pCC_wwbump_getcnt = getcnt;

    /* Set initial count of web packets to zero */
    Webcnt = 0;

    return 0;
}
```

File wwbcc.c (part 3)

```
/* Cross call termination function */
ix_error cc_fini(struct wwbump *wwb)
{
    ix_error e;
    /* Finalize crosscall and check for error */
    e = sk_wwbump_fini(&wwb->ccbase);
    if ( e )
        return ix_error_new(0,0,e,"Error in sk_wwbump_fini()\\n");

    return 0;   /* If no error, indicate sucessful return */
}

/* Function that is invoked each time a crosscall occurs */
ix_error getcnt(ix_base_t* bp, long* rv)
{
    (void)bp;   /* Reference unused arg to prevent compiler warnings */

    /* Actual work: copy the web count into the return value */
    *rv = Webcnt;

    /* Return 0 for success */
    return 0;
}
```

Tying It All Together

- Need to specify
  - ACEs to be loaded
  - Which ACEs are ingress / egress
  - Number and speed of network ports
- Configuration file used
  - Named *ixsys.config*
  - Programmer usually modifies sample file

Configuration Examples

```
file 0 /mnt/SlowIngressWWBump.uof
```

- Defines file */mnt/lowIngressWWBump.uof* to be
  - Ingress microcode (type 0)
  - Associated with slow ports (10/100 ports)

Configuration Examples
(continued)

```
microace wwbump /mnt/wwbump none 0 0
```

- Specifies
  - *wwbump* is a microace
  - Runs on the ingress side (first zero)
  - Has unknown type (second zero)

Configuration Examples
(continued)

```
bind static ifaceInput/default wwbump
bind static wwbump/default ifaceOutput
```

- Specifies
  - Default binding for ingress ACE is *wwbump*
  - Default binding for output from *wwbump* is ACE named *ifaceOutput*

Configuration Examples
(continued)

`sh` *command*

- Specifies command to be run on the StrongARM
- Typical use: preload ARP cache

```
sh arp -s 10.1.0.2 01:02:03:04:05:06
```

Summary

A huge amount of low-level code is required, even for a trivial ACE that counts Web packets or a trivial crosscall that returns the contents of an integer. In addition to the code, a detailed configuration file must be created to specify bindings among ACEs.


Questions?

X

Switching Fabrics

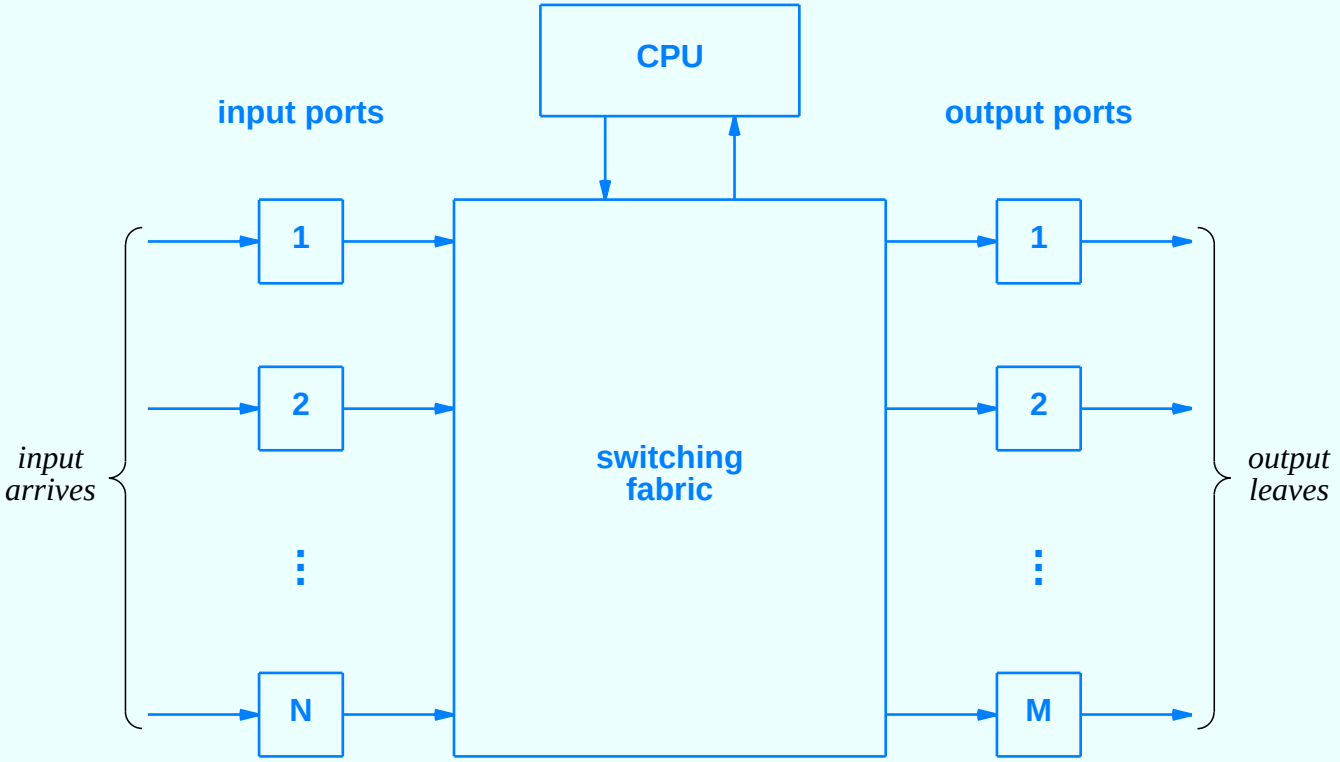
Physical Interconnection

- 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

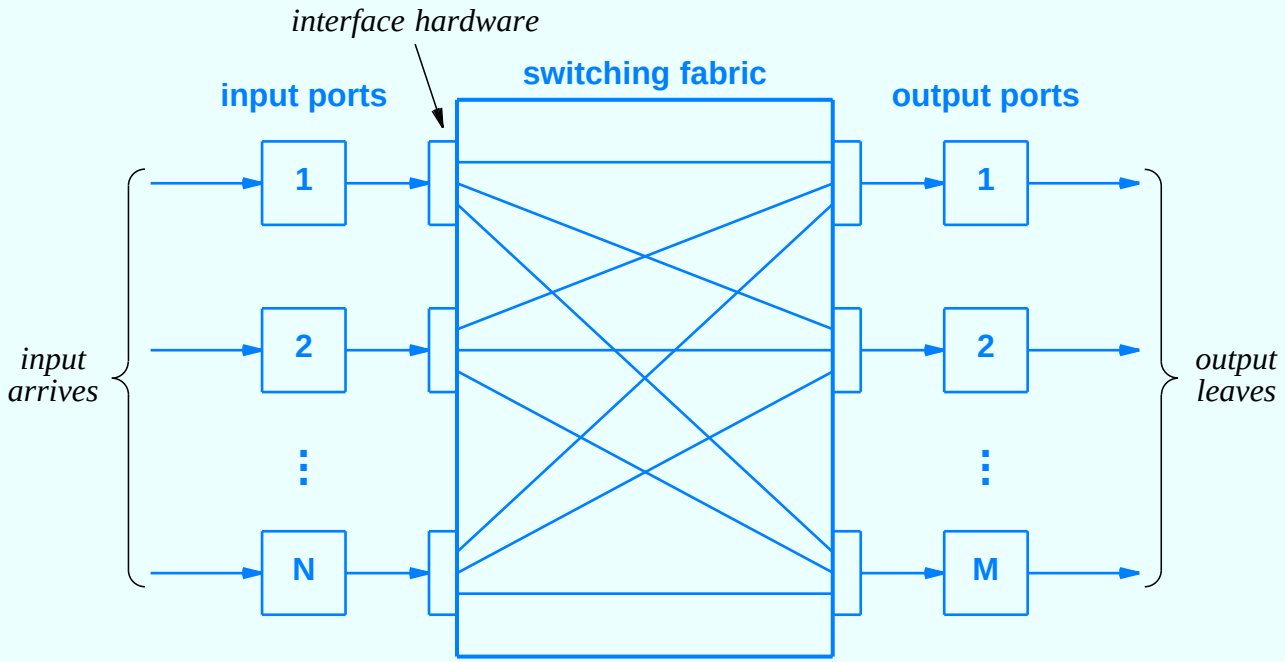
Switching Fabric Properties

- 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

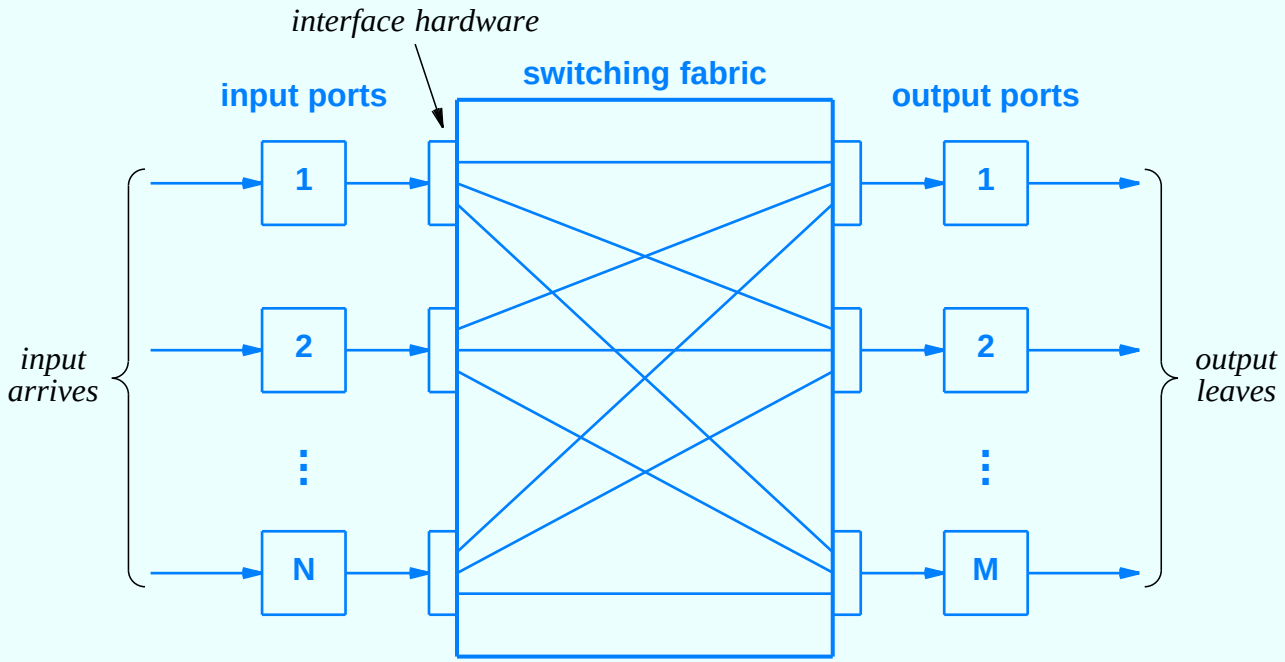
- Space-division (separate paths)
- Time-division (shared medium)

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

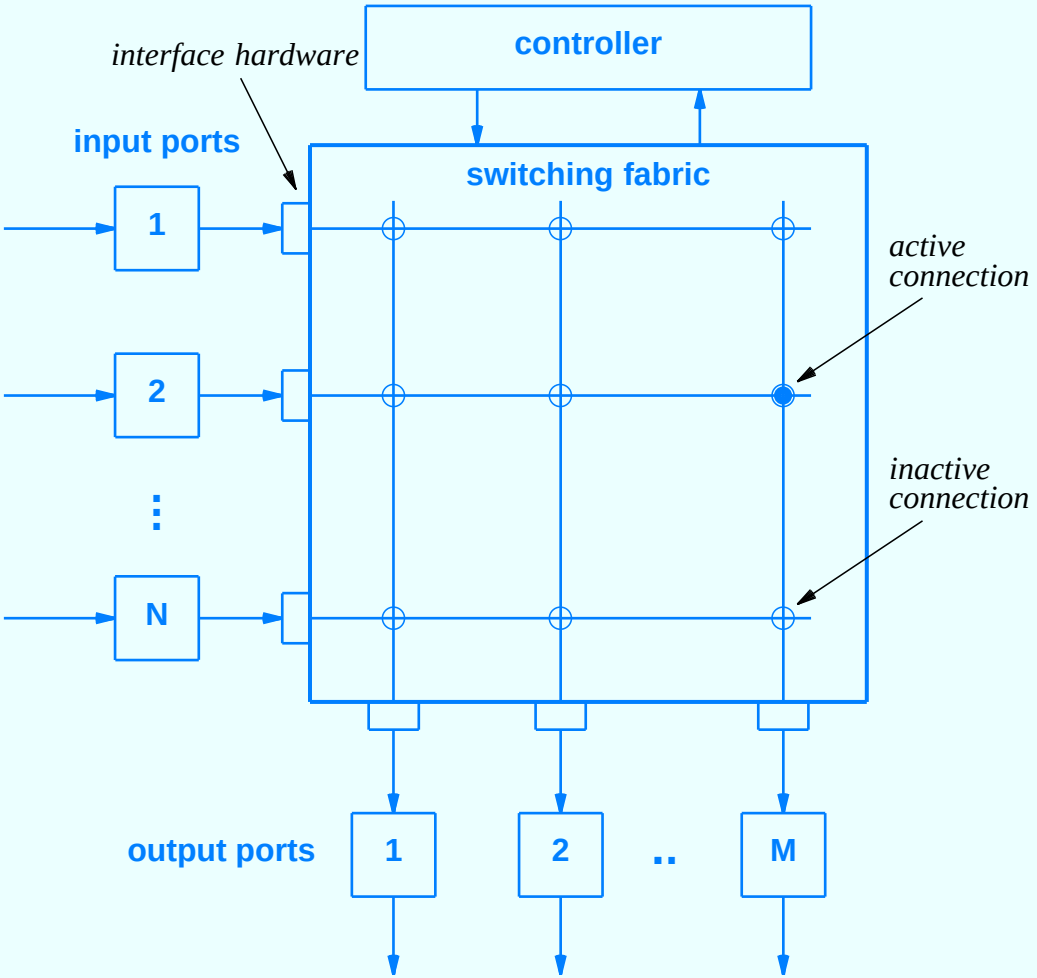
Desires

- 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*

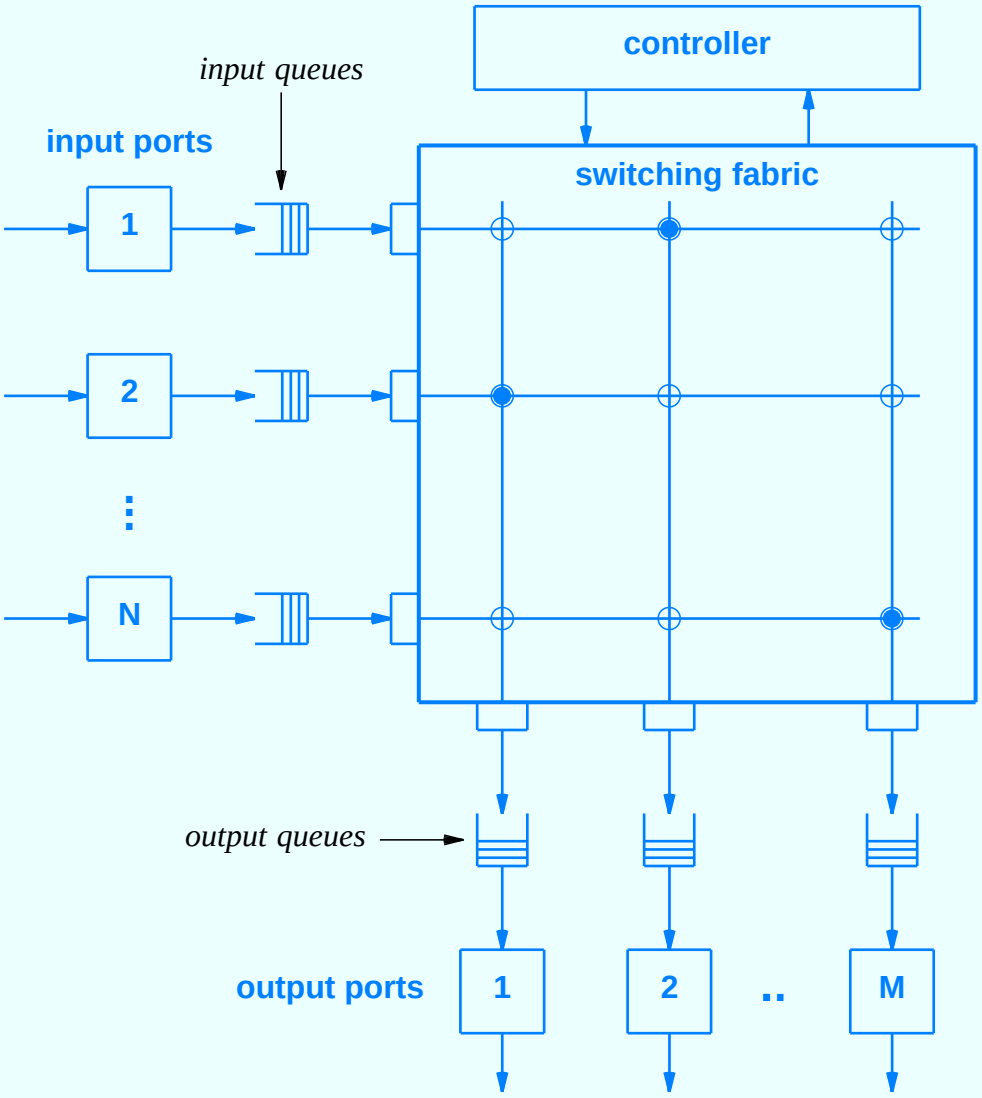
Crossbar

- Allows simultaneous transfer on disjoint pairs of ports
- Can still have *port contention*

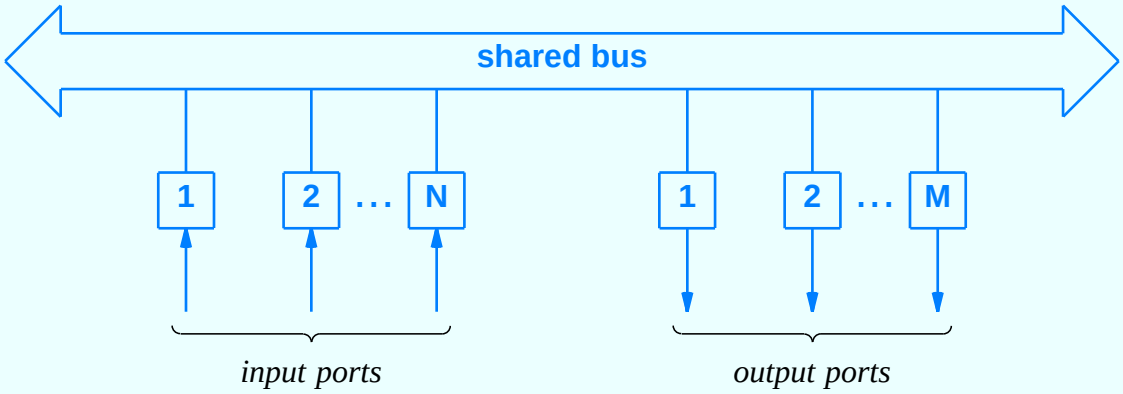
Solving Contention

- Queues (FIFOs)
  - Attached to input
  - Attached to output
  - At intermediate points

Crossbar Fabric With Queuing

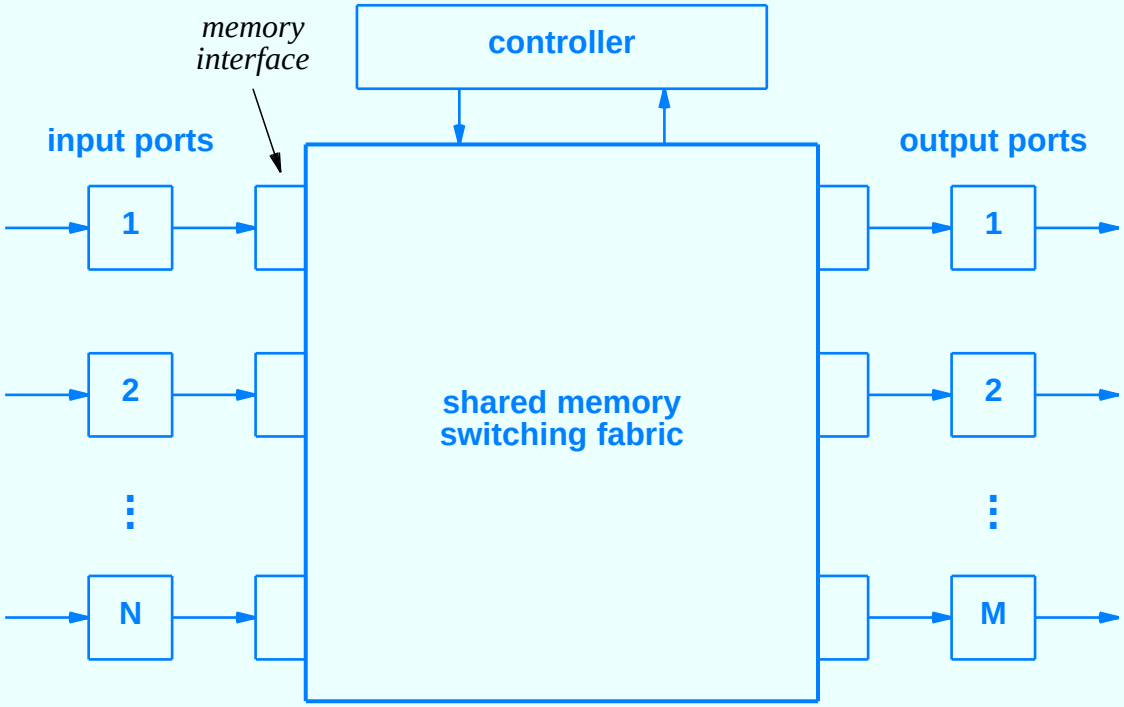


Time-Division Fabric (shared bus)



- Chief advantage: low cost
- Chief disadvantage: low speed

Time-Division Fabric (shared memory)



- *May* be better than shared bus
- Usually more expensive

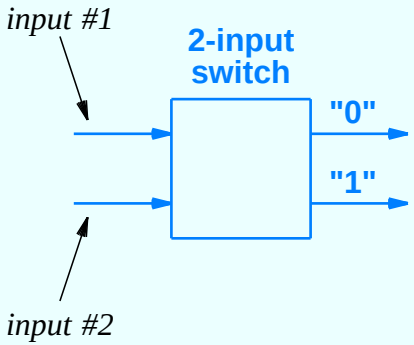
Multi-Stage Fabrics

- 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

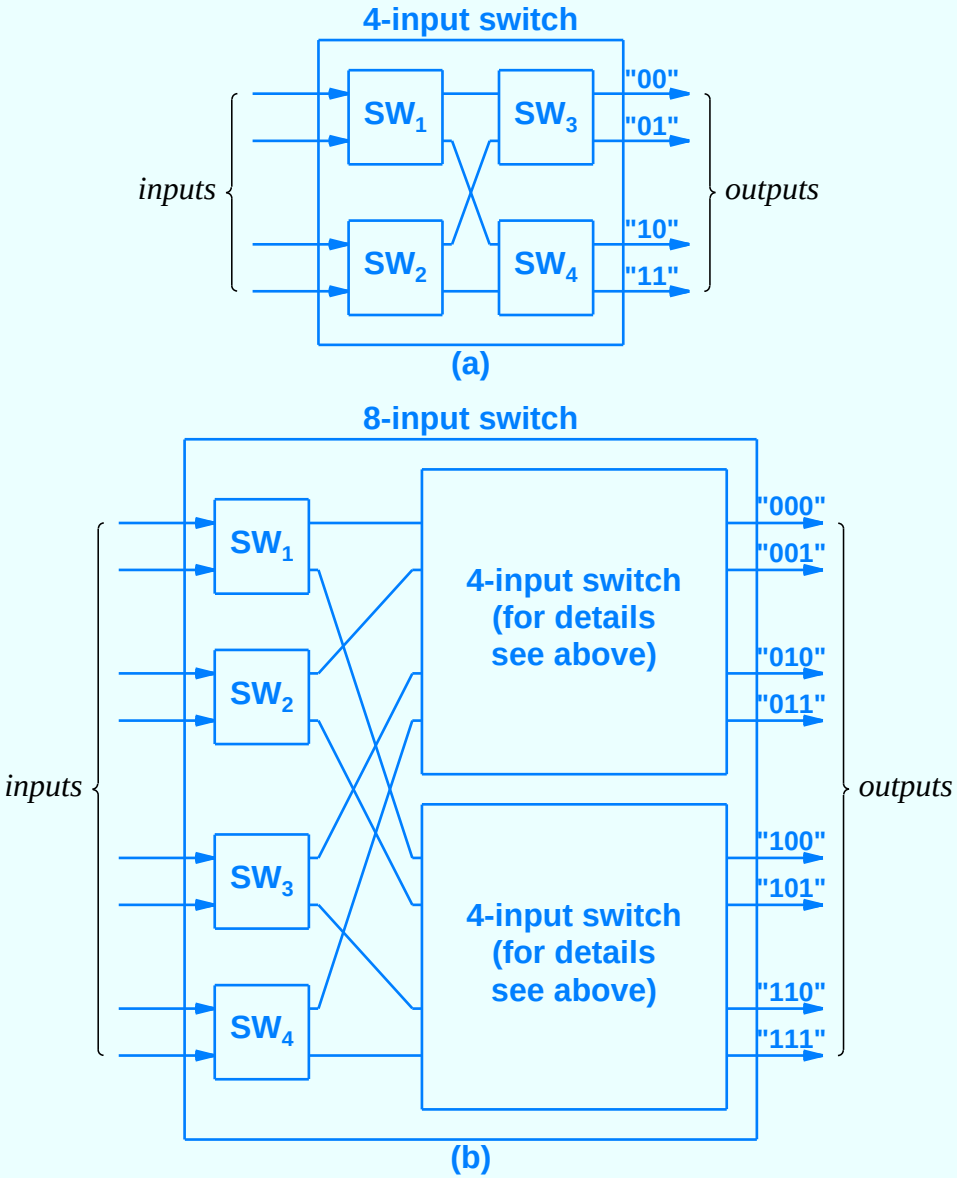
- Example of multi-stage fabric
- Features
  - Scalable
  - Self-routing
  - Packet queues allowed, but not required

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?

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 Framer 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

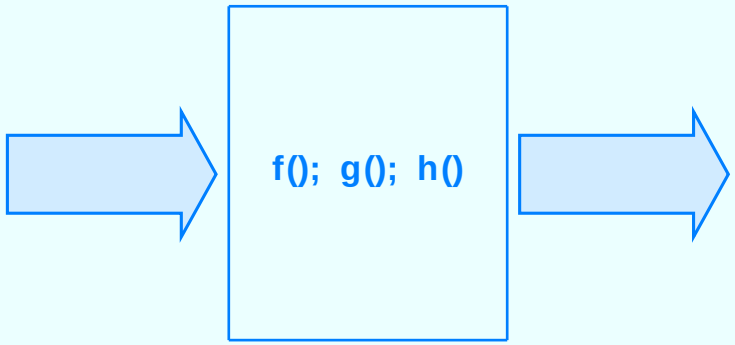
Implicit And Explicit Parallelism

- 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

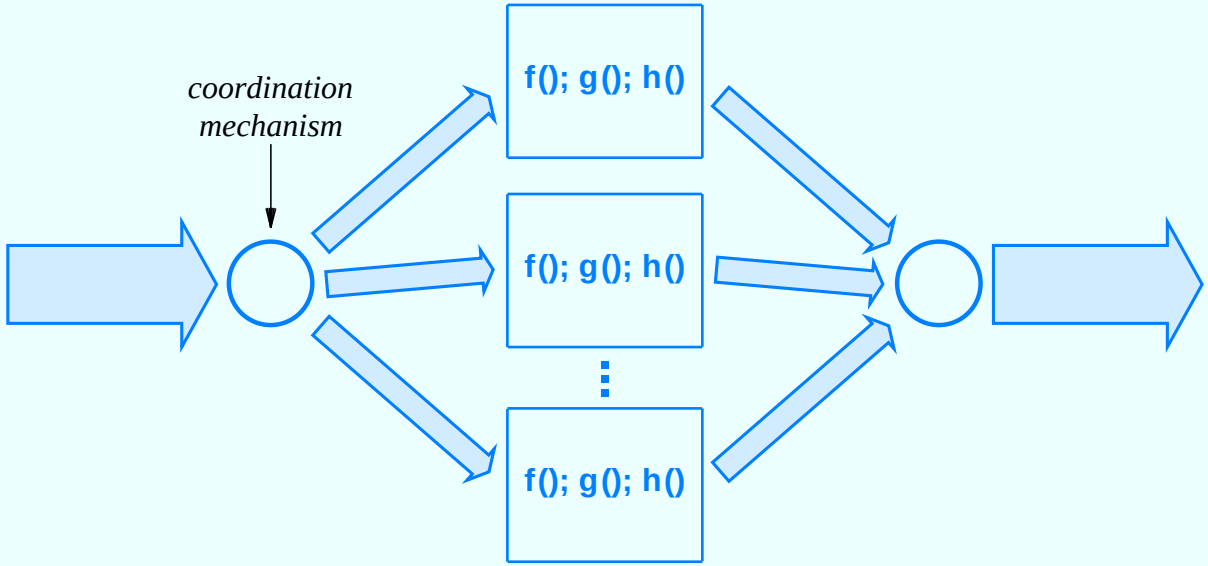
- Embedded processor plus fixed coprocessors
- Embedded processor plus programmable I/O processors
- Parallel (number of processors scales to handle load)
- Pipeline processors
- Dataflow

Embedded Processor Architecture



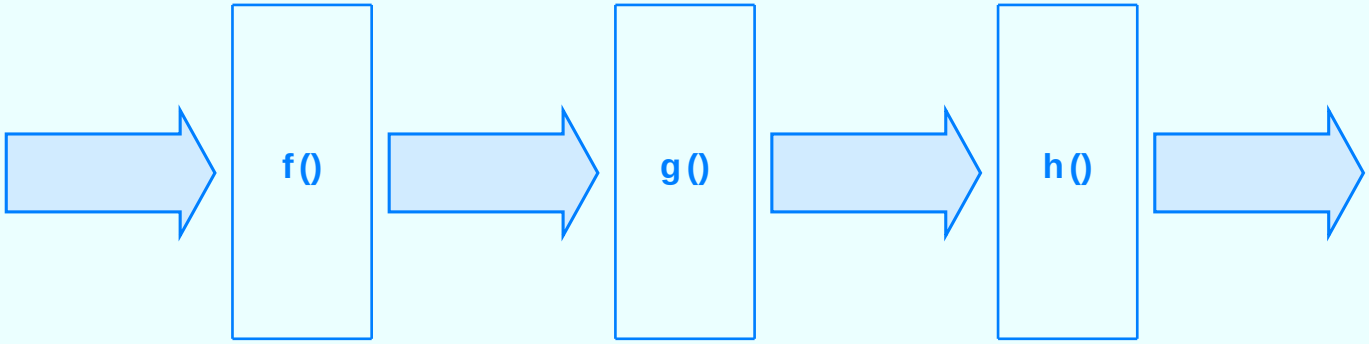
- Single processor
  - Handles all functions
  - Passes packet on
- Known as run-to-completion

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)

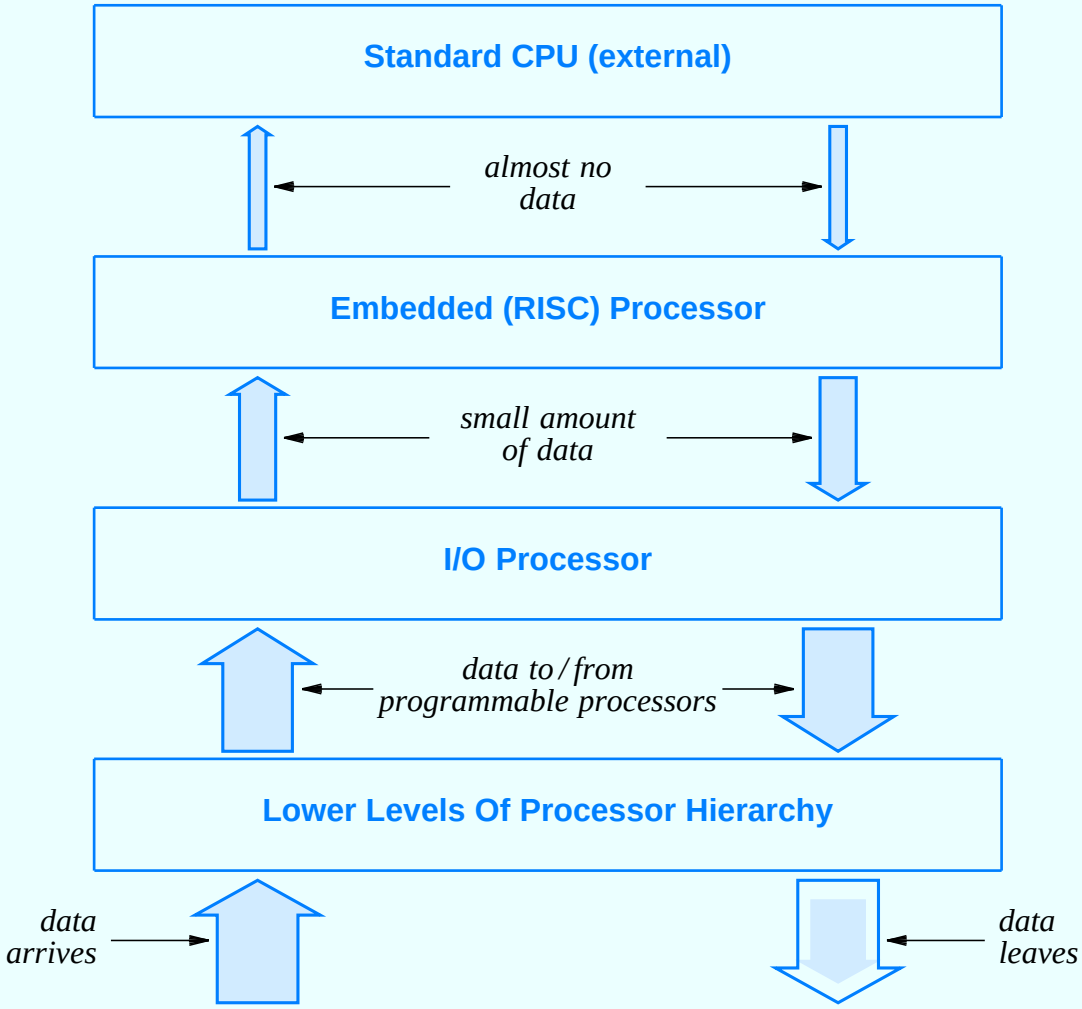
I/O processor

- 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?

XIV

Issues In Scaling A Network Processor

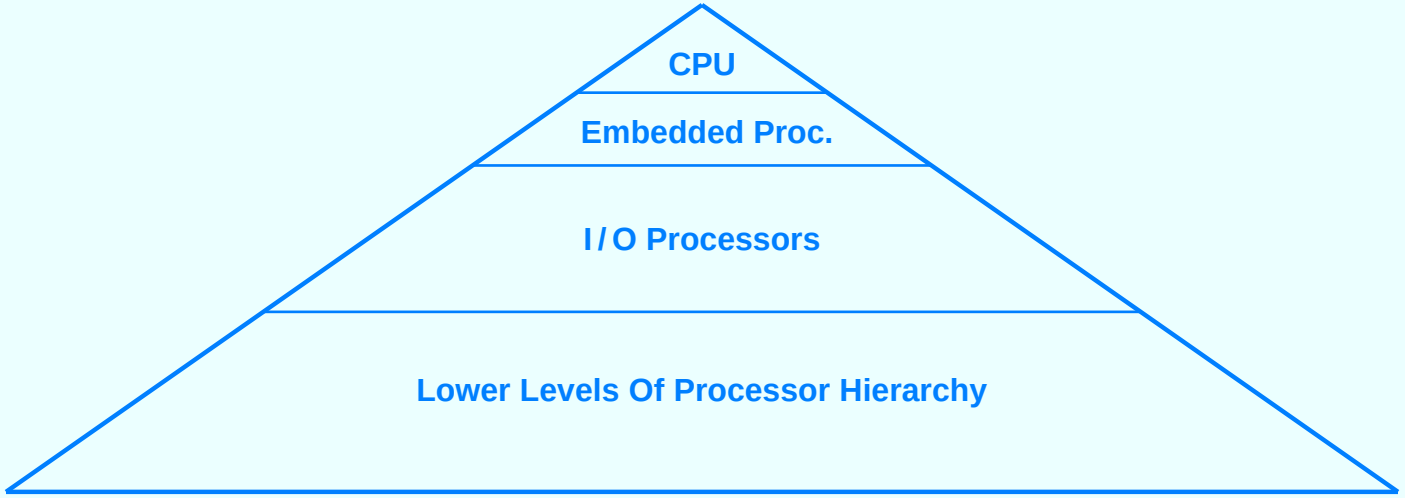
Design Questions

- 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

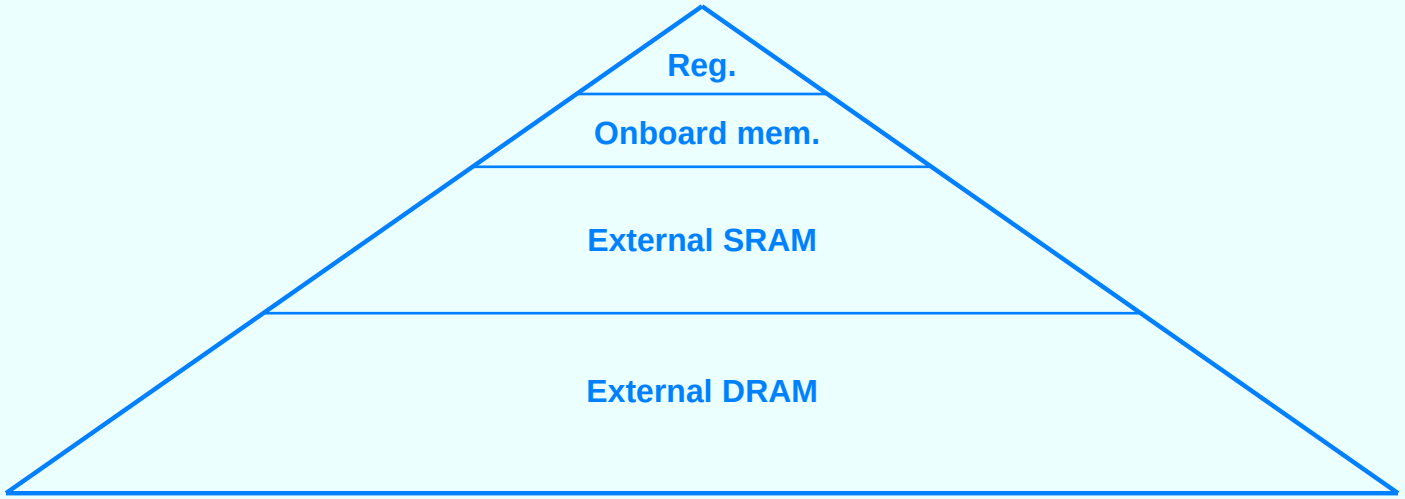
Memory Speed

- 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

Memory Technology	Abbreviation	Purpose
Synchronized DRAM	SDRAM	Synchronized with CPU for lower latency
Quad Data Rate SRAM	QDR-SRAM	Optimized for low latency and multiple access
Zero Bus Turnaround SRAM	ZBT-SRAM	Optimized for random access
Fast Cycle RAM	FCRAM	Low cycle time optimized for block transfer
Double Data Rate DRAM	DDR-DRAM	Optimized for low latency
Reduced Latency DRAM	RLDRAM	Low cycle time and low power requirements

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

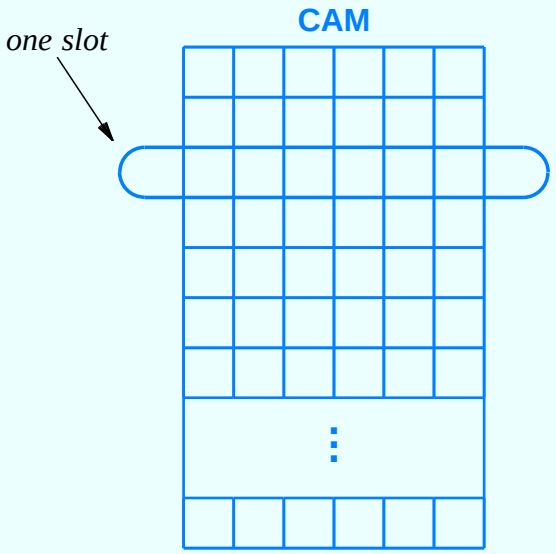
Memory Cache

- 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)

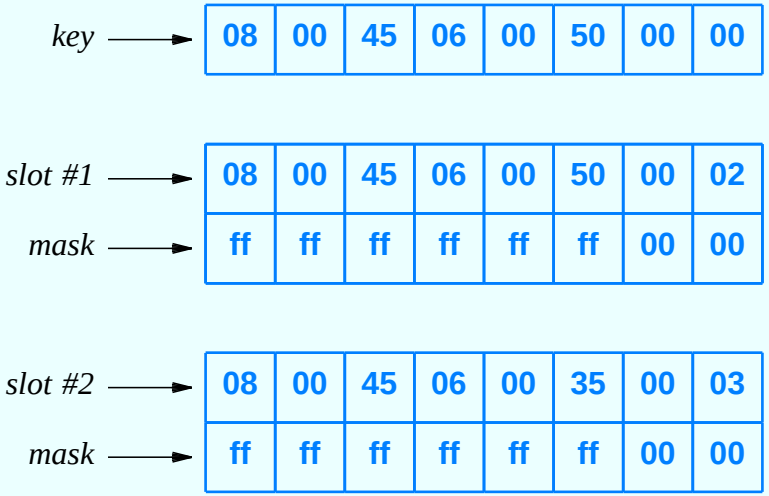
- Allows masking of entries
- Good for network processor

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

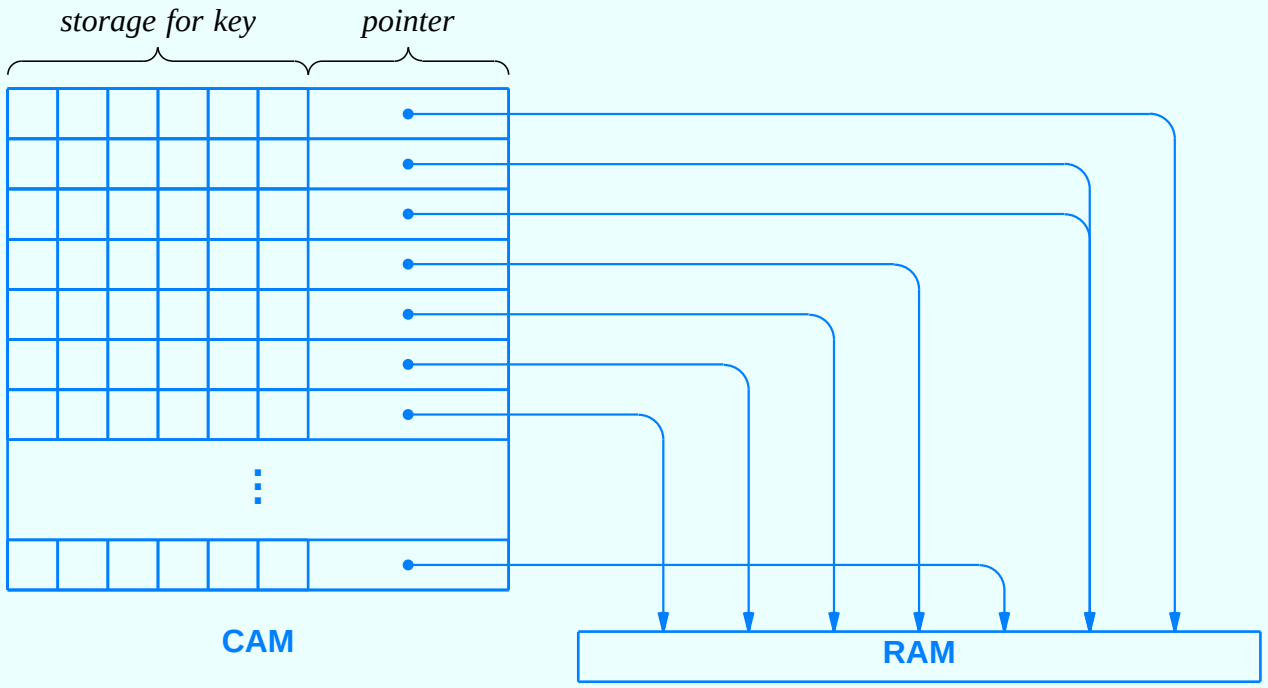


- Key matches slot #1

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

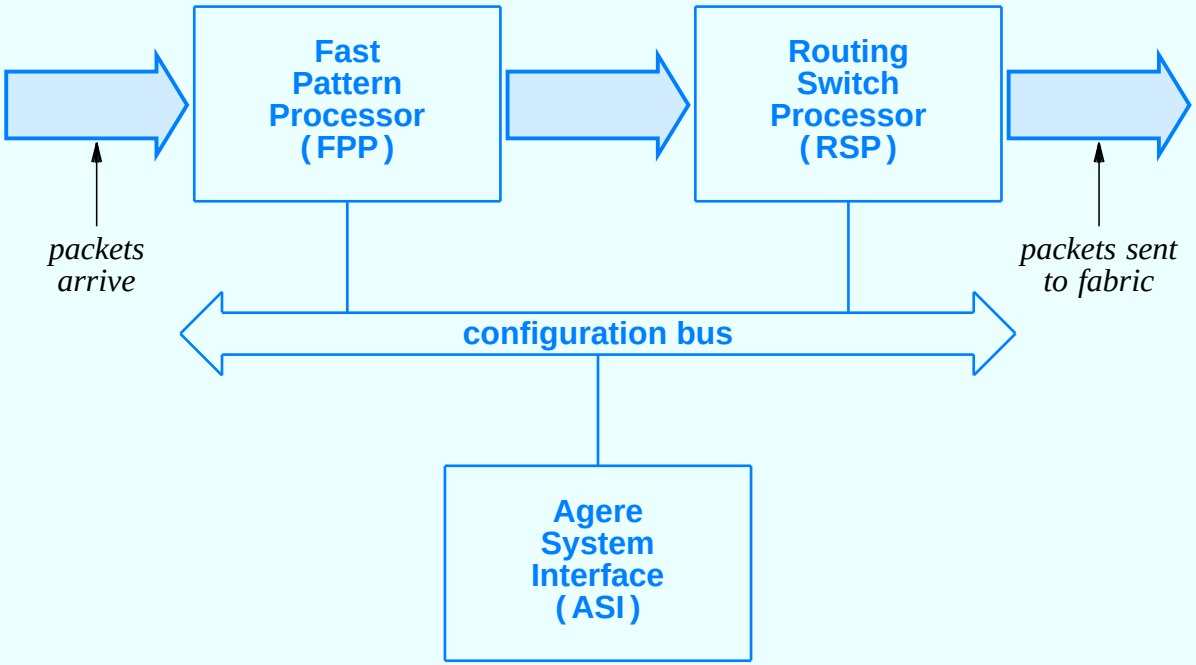
Examples

- 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 2003

Multi-Chip Pipeline (Agere)

- Brand name PayloadPlus
- Three individual chips
  - Fast Pattern Processor (FPP) for classification
  - Routing Switch Processor (RSP) for forwarding
  - Agere System Interface (ASI) for traffic management and exceptions

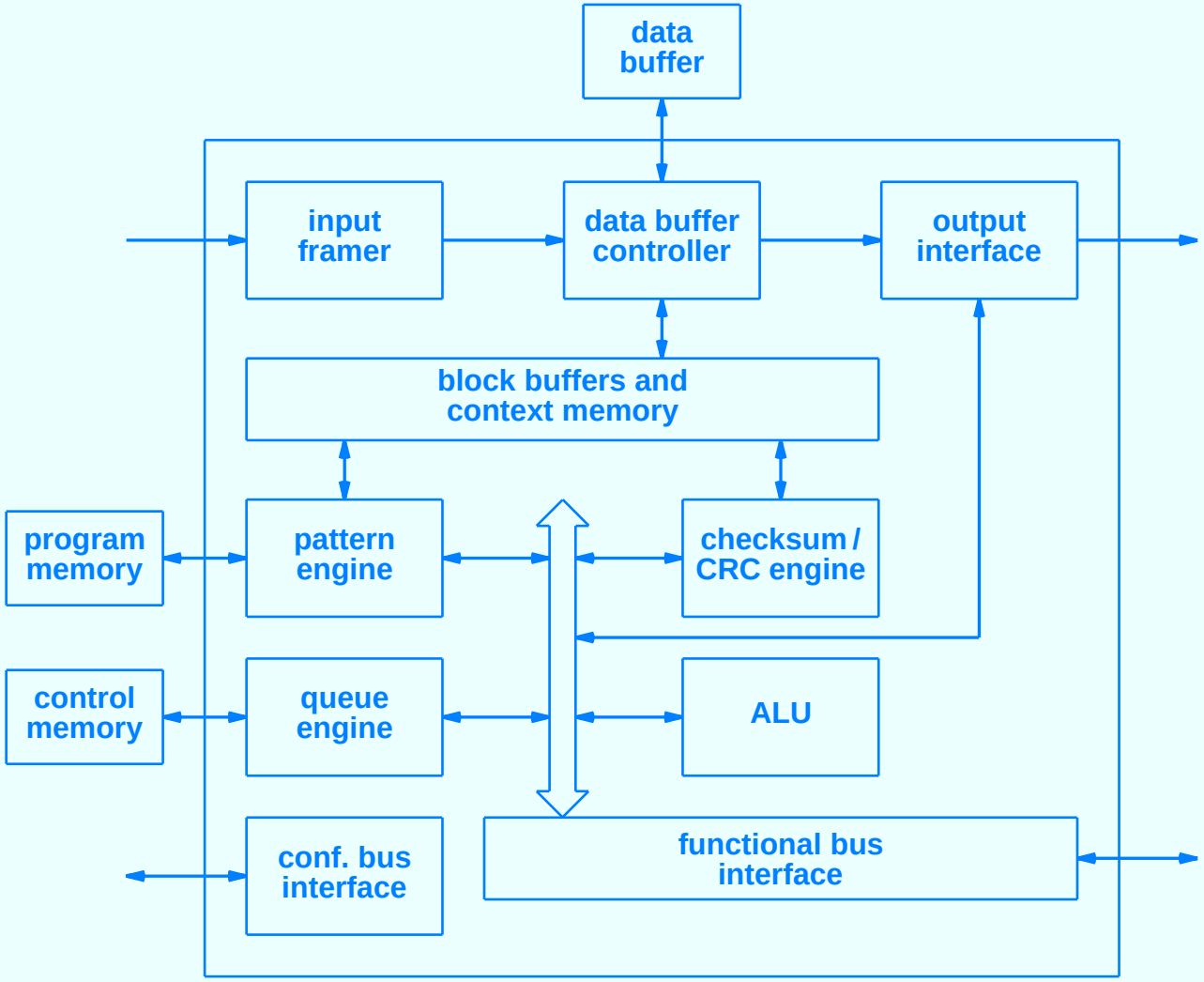
Multi-Chip Pipeline (Agere) (continued)



Languages Used By Agere

- FPL
  - Functional Programming Language
  - Produces code for FPP
  - Non-procedural
- ASL
  - Agere Scripting Language
  - Produces code for ASI
  - Similar to shell scripts

Architecture Of Agere's FPP Chip

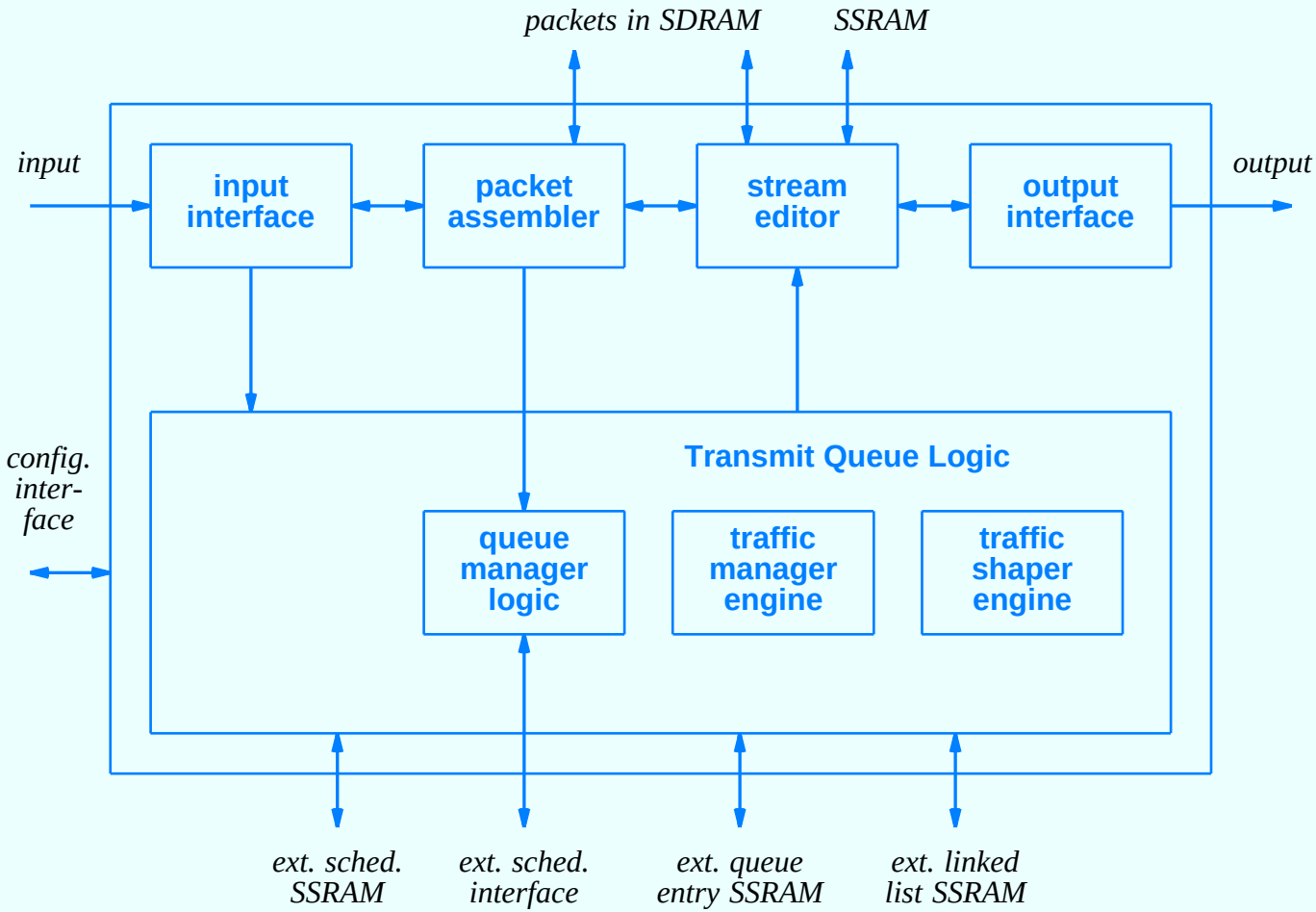


Processors On Agere's FPP

Processor Or Unit	Purpose
Pattern processing engine	Perform pattern matching on each packet
Queue engine	Control packet queueing
Checksum/CRC engine	Compute checksum or CRC for a packet
ALU	Conventional operations
Input interface and framer	Divide incoming packet into 64-octet blocks
Data buffer controller	Control access to external data buffer
Configuration bus interface	Connect to external configuration bus
Functional bus interface	Connect to external functional bus
Output interface	Connect to external RSP chip

- Note: Functional bus interface implements an RPC-like interface

Architecture Of Agere's RSP Chip



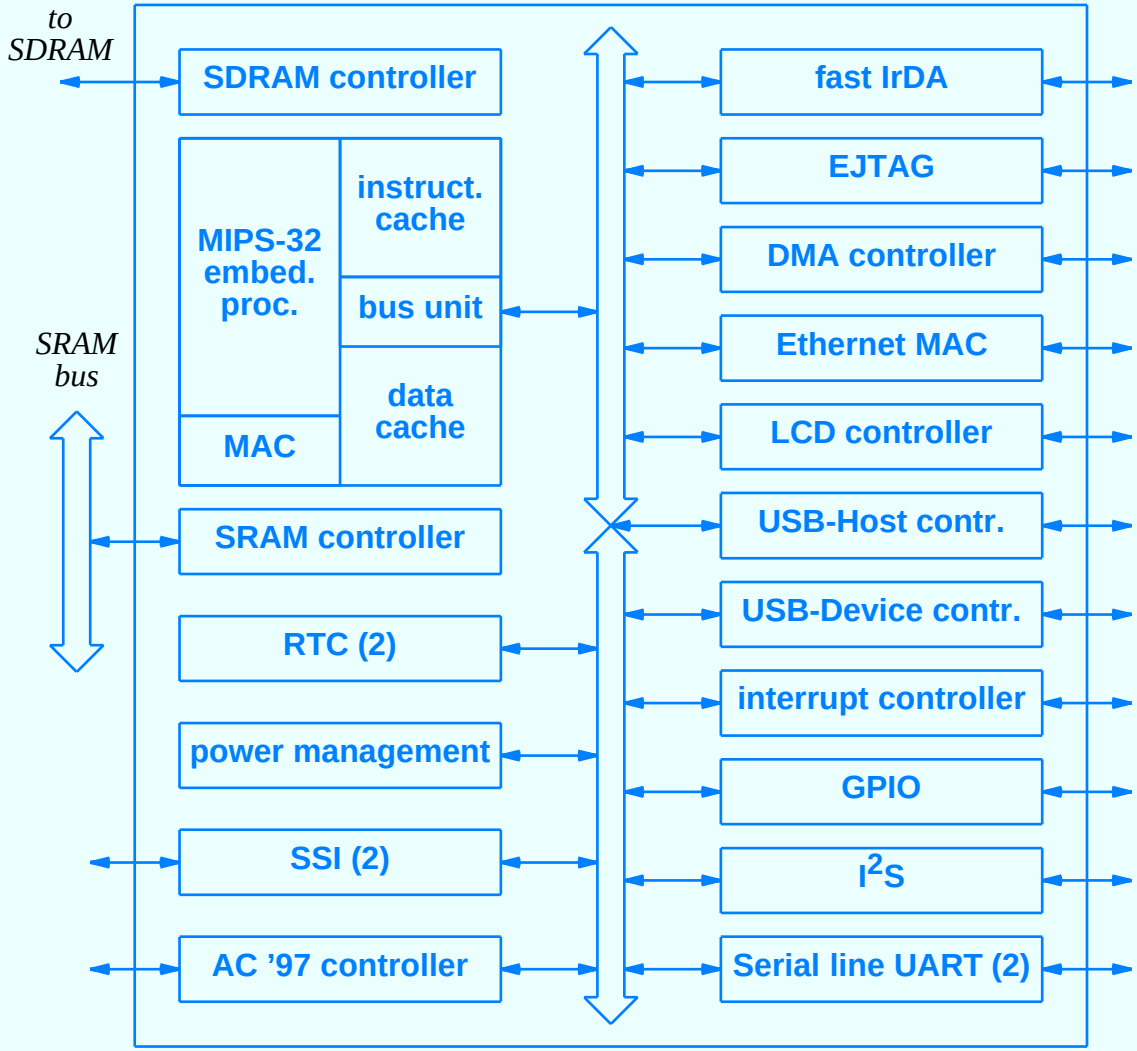
Processors On Agere's RSP

Processor Or Unit	Purpose
Stream editor engine	Perform modifications on packet
Traffic manager engine	Police traffic and keep statistics
Traffic shaper engine	Ensure QoS parameters
Input interface	Accept packet from FPP
Packet assembler	Store incoming packet in memory
Queue manager logic	Interface to external traffic scheduler
Configuration bus interface	Connect to external configuration bus
Output interface	External connection for outgoing packets

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

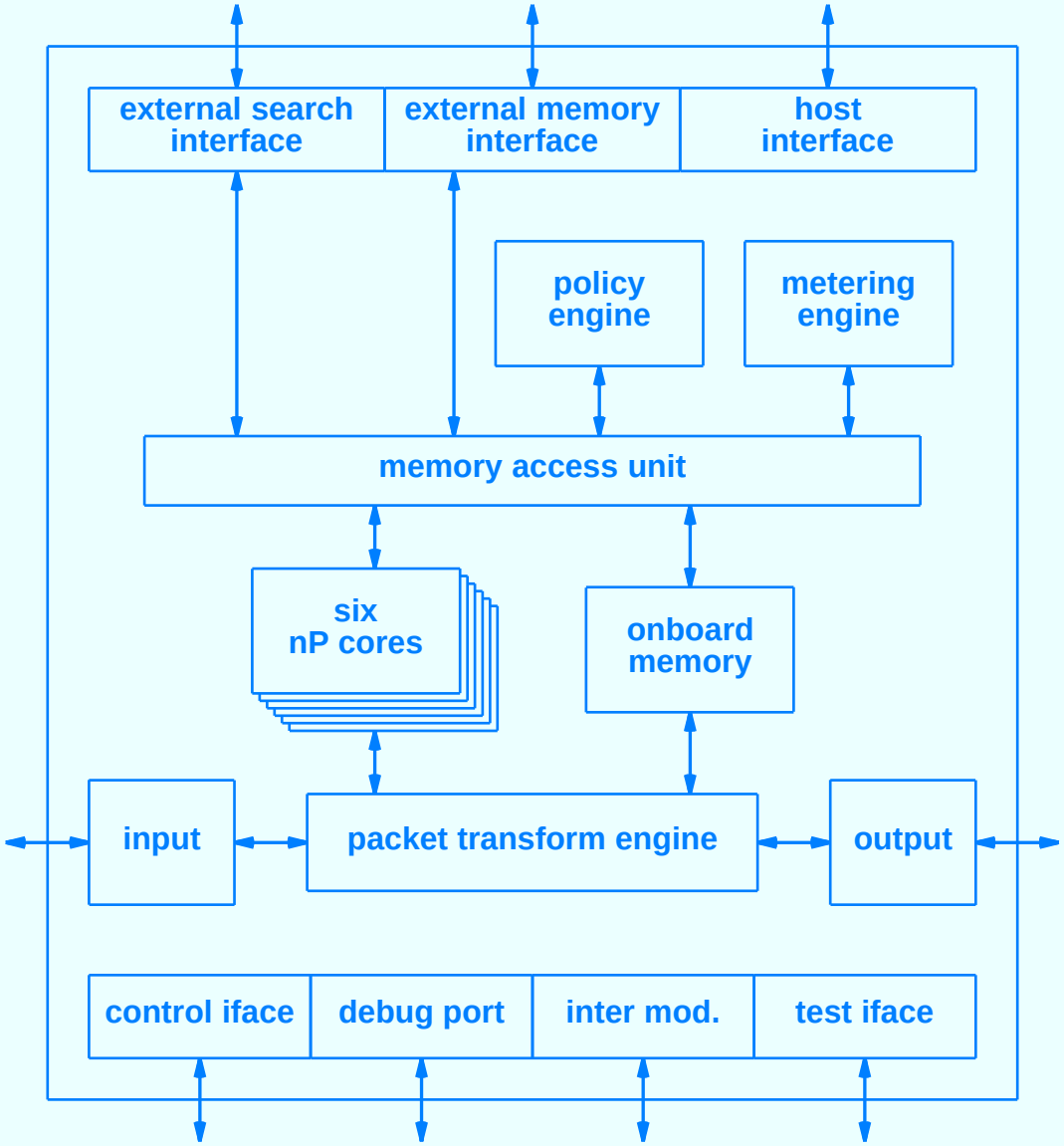
Alchemy Architecture



Parallel Embedded Processors Plus Coprocessors (AMCC)

- One to six nP core processors
- Various engines
  - Packet metering
  - Packet transform
  - Packet policy

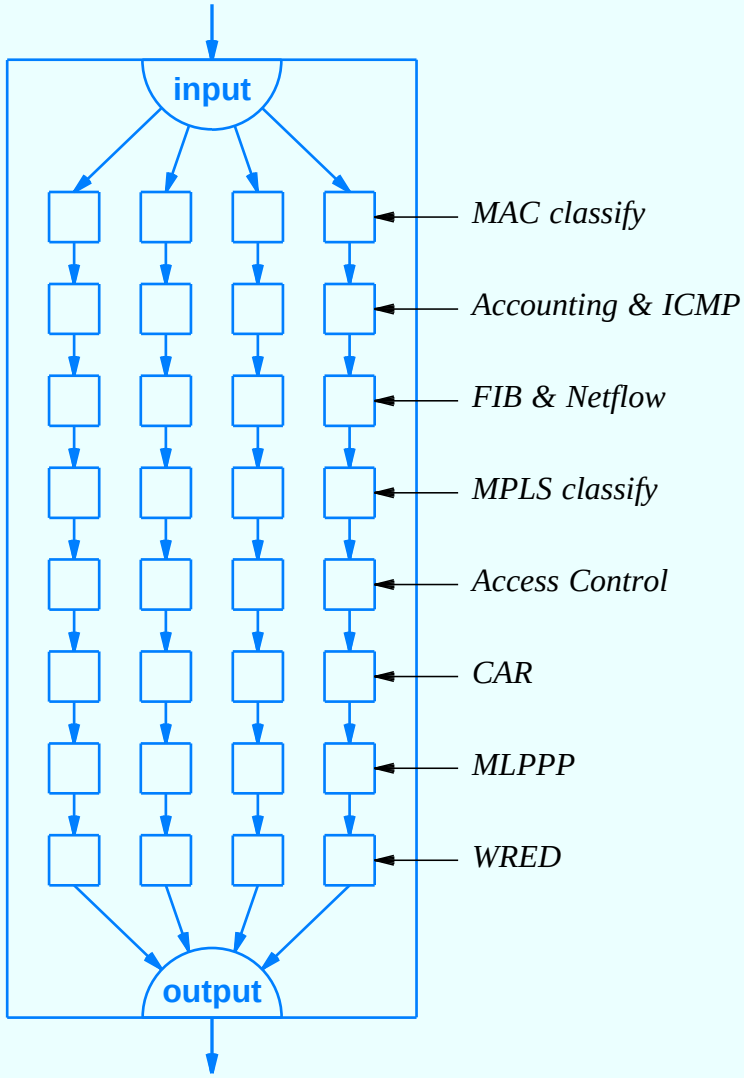
AMCC Architecture



Pipeline 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

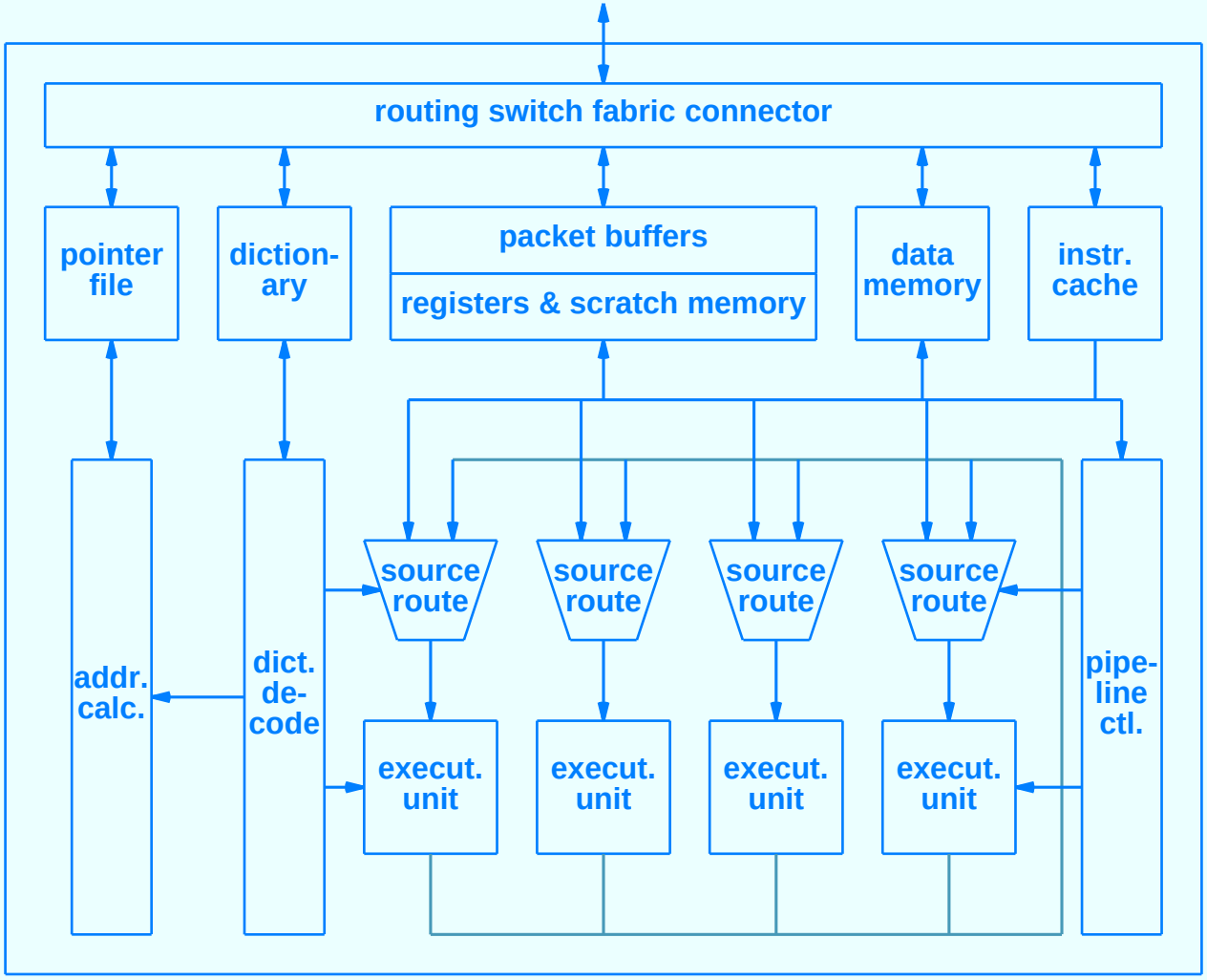
Cisco Architecture



Configurable Instruction Set Processors (Cognigine)

- Up to sixteen parallel processors
- Connected in a pipeline
- Processor called *Reconfigurable Communication Unit (RCU)*
- Interconnected by *Routing Switch Fabric (RSF)*
- Instruction set determined by loadable dictionary

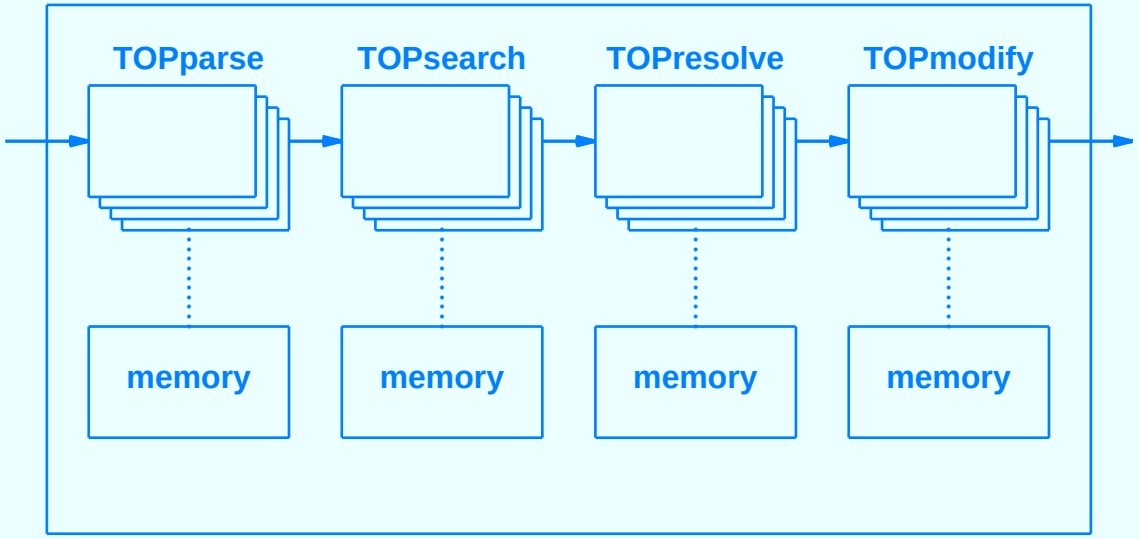
Cognigine Architecture



Pipeline Of Parallel Heterogeneous Processors (EZchip)

- Four processor types
- Each type optimized for specific task

EZchip Architecture



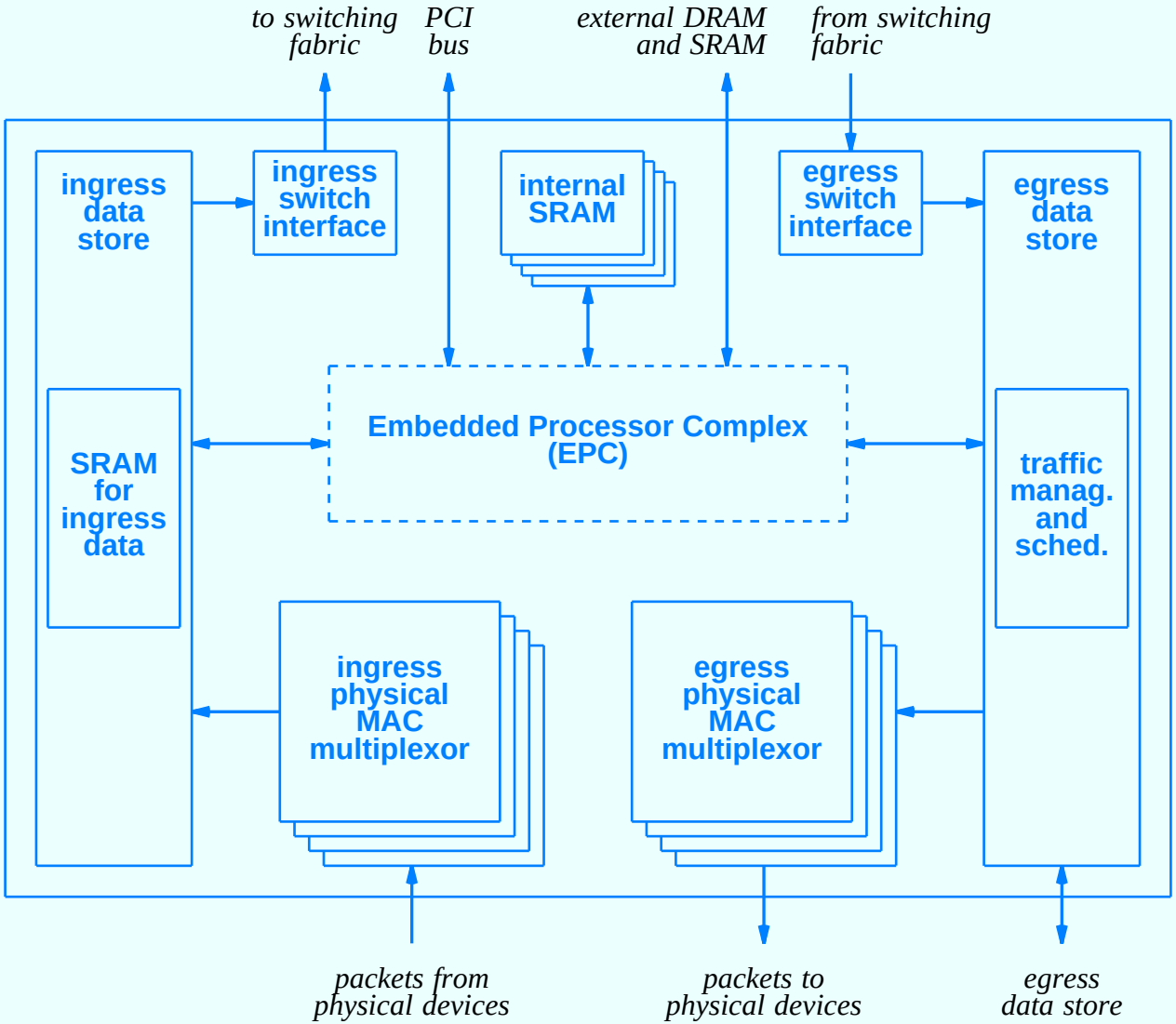
EZchip Processor Types

Processor Type	Optimized For
TOPparse	Header field extraction and classification
TOPsearch	Table lookup
TOPresolve	Queue management and forwarding
TOPmodify	Packet header and content modification

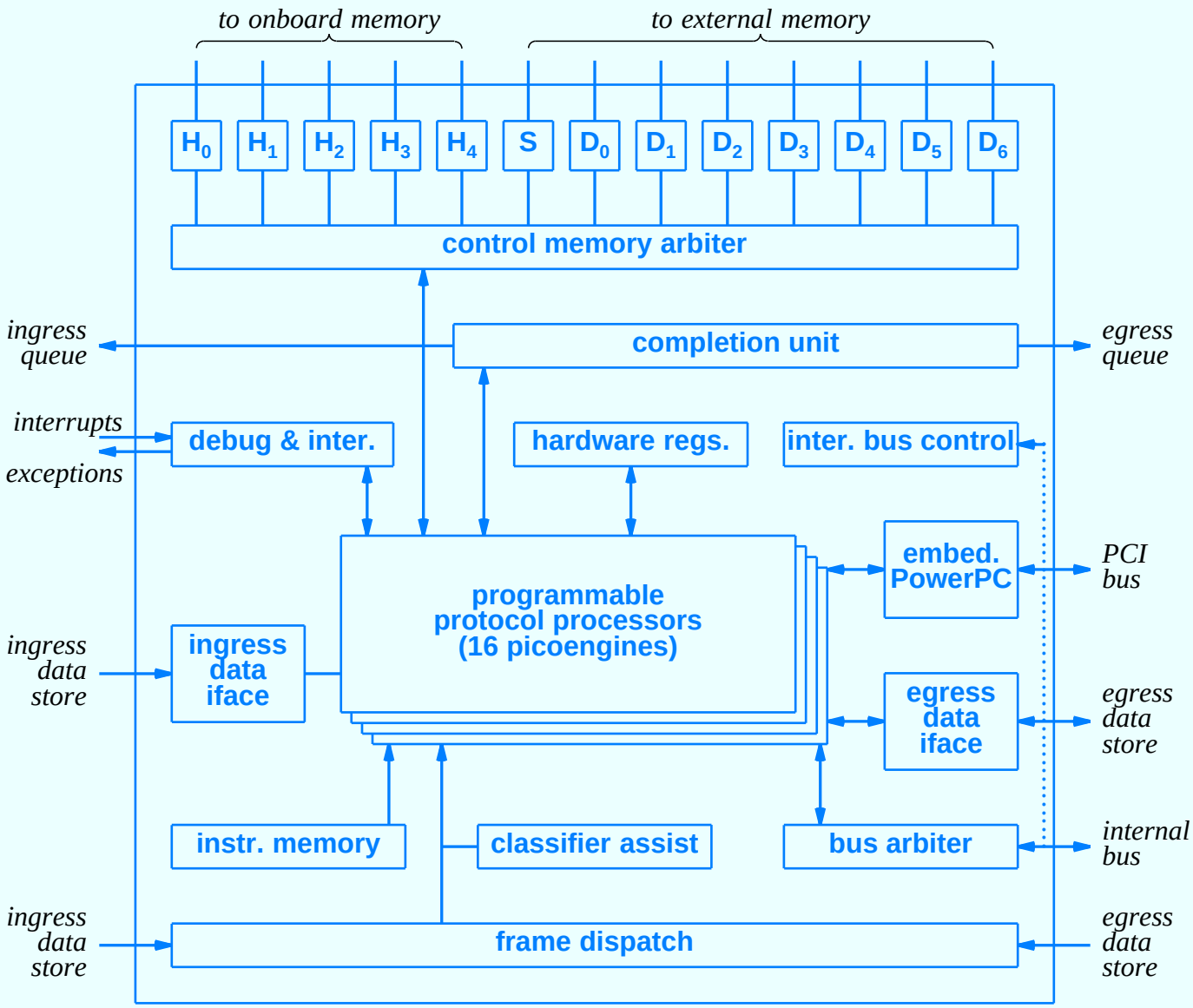
Extensive And Diverse Processors (IBM)

- Multiple processor types
- Extensive use of parallelism
- Separate ingress and egress processing paths
- Multiple onboard data stores
- Model is *NP4GS3*

IBM NP4GS3 Architecture



IBM's Embedded Processor Complex



Packet Engines

- Found in Embedded Processor Complex
- Programmable
- Handle many packet processing tasks
- Operate in parallel (sixteen)
- Known as *picoengines*

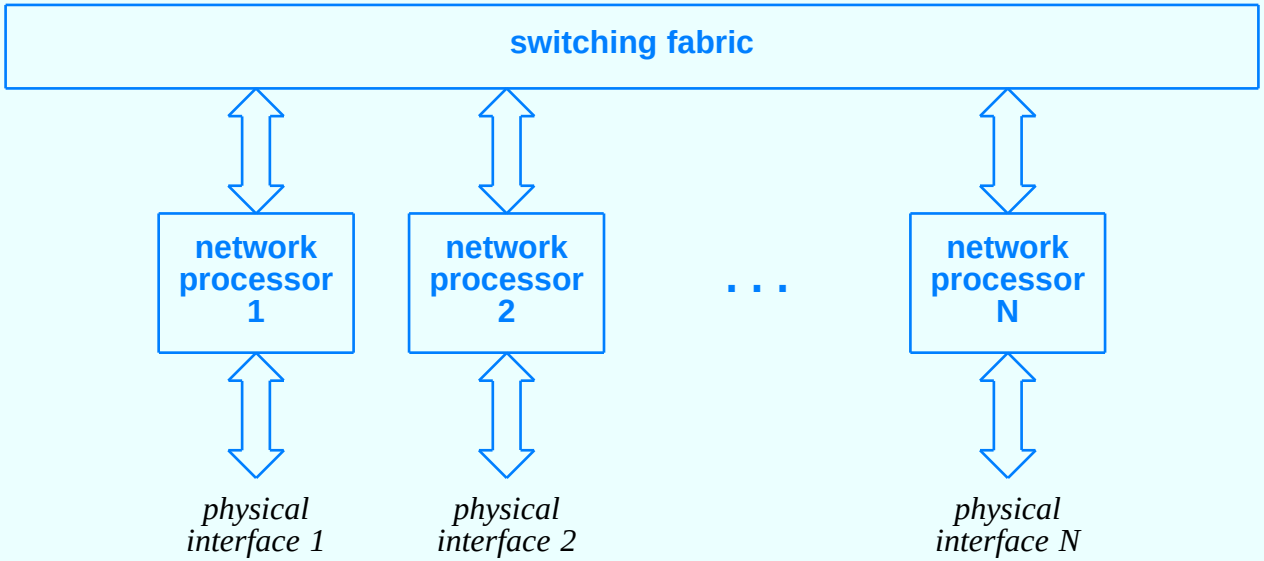
Other Processors On The IBM Chip

Coprocessor	Purpose
Data Store	Provides frame buffer DMA
Checksum	Calculates or verifies header checksums
Enqueue	Passes outgoing frames to switch or target queues
Interface	Provides access to internal registers and memory
String Copy	Transfers internal bulk data at high speed
Counter	Updates counters used in protocol processing
Policy	Manages traffic
Semaphore	Coordinates and synchronizes threads

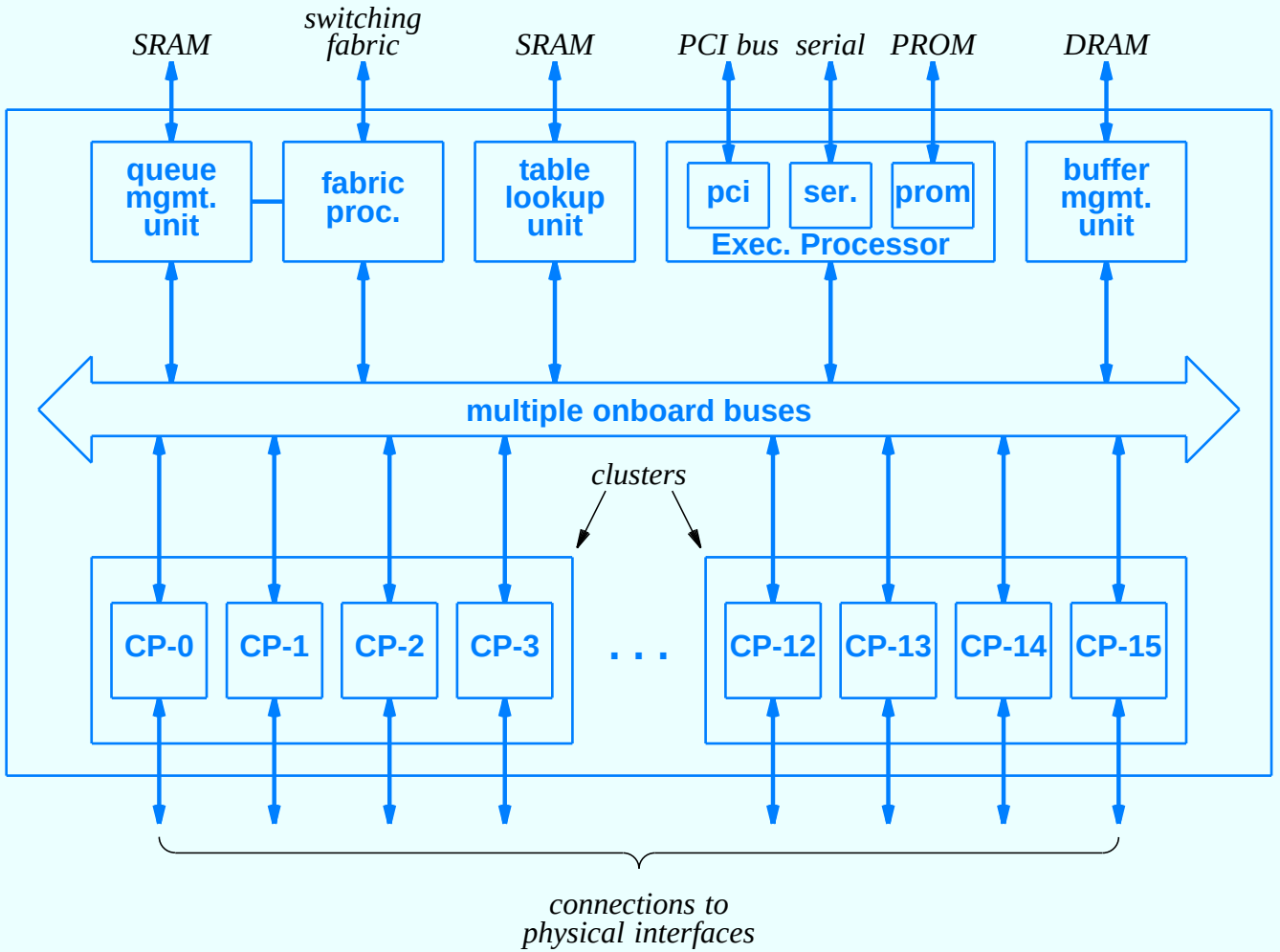
Flexible RISC Plus Coprocessors (Motorola C-PORT)

- Onboard processors can be
  - Dedicated
  - Parallel clusters
  - Pipeline

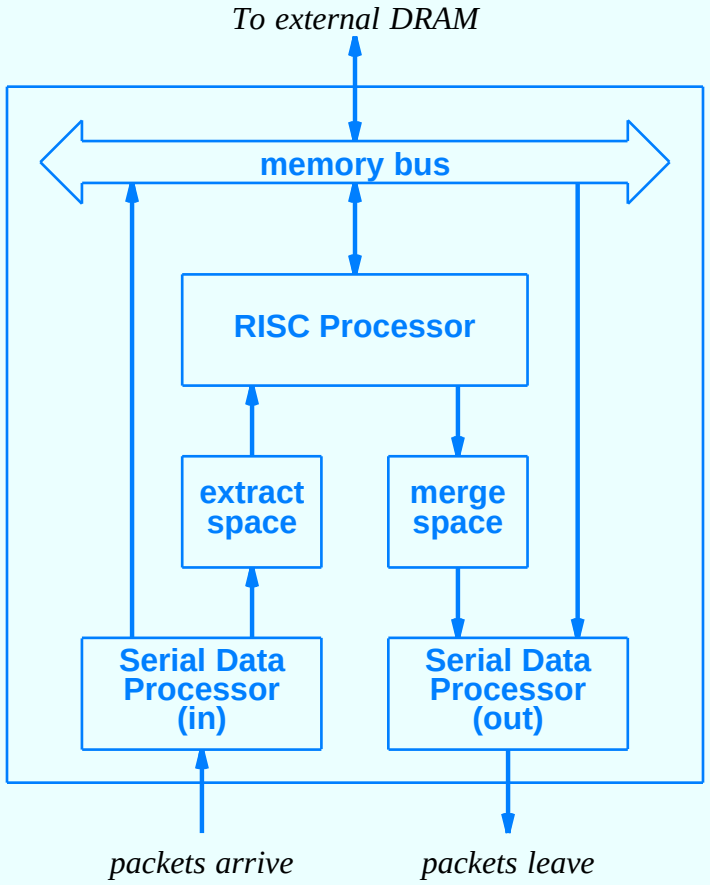
C-Port Architecture



C-Port Configured As Parallel Clusters



Internal Structure Of A C-Port Channel Processor



- Actually a processor complex

Summary

- Many network processor architecture variations
- Examples include
  - Multi-chip and single-chip products
  - Augmented RISC processor
  - Embedded parallel processors plus coprocessors
  - Pipeline of homogeneous processors
  - Configurable instruction set
  - Pipeline of parallel heterogeneous processors
  - Extensive and diverse processors
  - Flexible RISC plus coprocessors


Questions?

XXVII

Intel's Second Generation Processors

Terminology

In the network processor industry, one has to be careful when describing successive versions of network processors because the delay between the announcement of a new product and the actual date at which customers can obtain the product gives a second meaning to the phrase ‘‘later versions of a network processor’’.

General Characteristics

- Same basic architecture
  - Embedded processor
  - Parallel microengines
- Enhancements
  - Speed
  - Amount of parallelism
  - Functionality

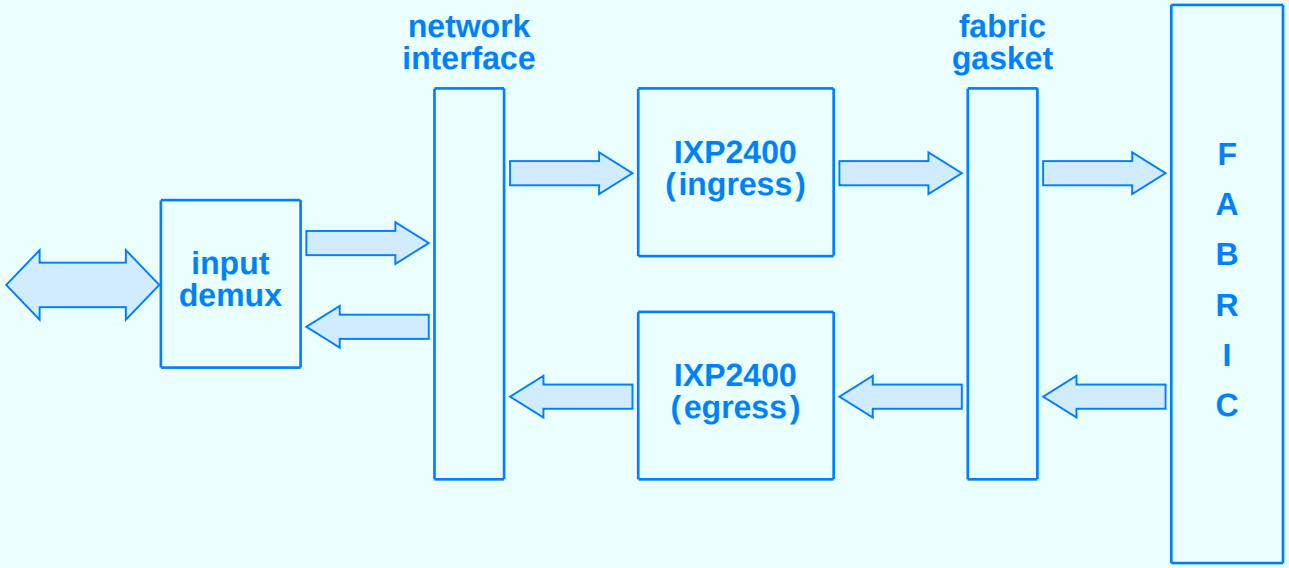
Models Of Next Generation Chips

- Two primary models
  - IXP2400 (2.4 Gbps)
  - IXP2800 (10 Gbps)
- Version of 2800 with onboard encryption processor
  - IXP2850

Use Of Two Chips For Higher Throughput

- Problem
  - One chip insufficient for full duplex
- Solution
  - Increase parallelism
  - Use separate chips for ingress and egress processing
- Need communication path between them

Conceptual Data Flow Through Two Chips



- Many details not shown
  - Inter-chip communication mechanism
  - Memory interfaces

IXP2400 Features

- XScale embedded processor (ARM compliant) with caches
- Eight microengines (400 or 600 MHz)
- Eight hardware threads per microengine
- Multiple microengine instruction stores of one thousand instructions each
- Two hundred fifty-six general-purpose registers
- Five hundred twelve transfer registers
- Addressing for two gigabyte DDR-DRAM
- Addressing for thirty-two megabyte QDR-SRAM
- Sixteen words of Next Neighbor Registers

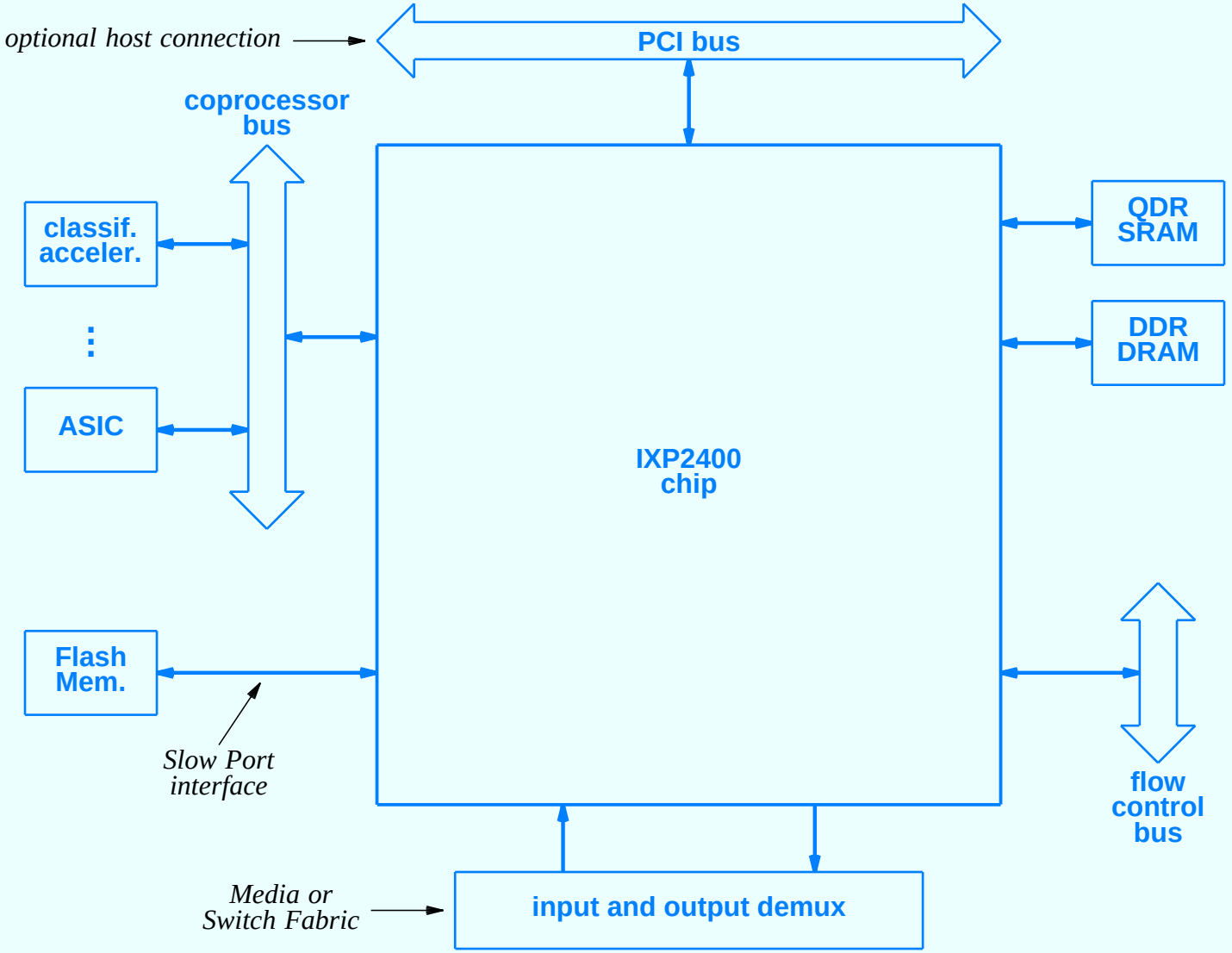
Memory Hierarchy

- External DDR-DRAM
- External QDR-SRAM
  - Q-array hardware in controller
- Onboard Scratchpad memory
- Onboard local memory

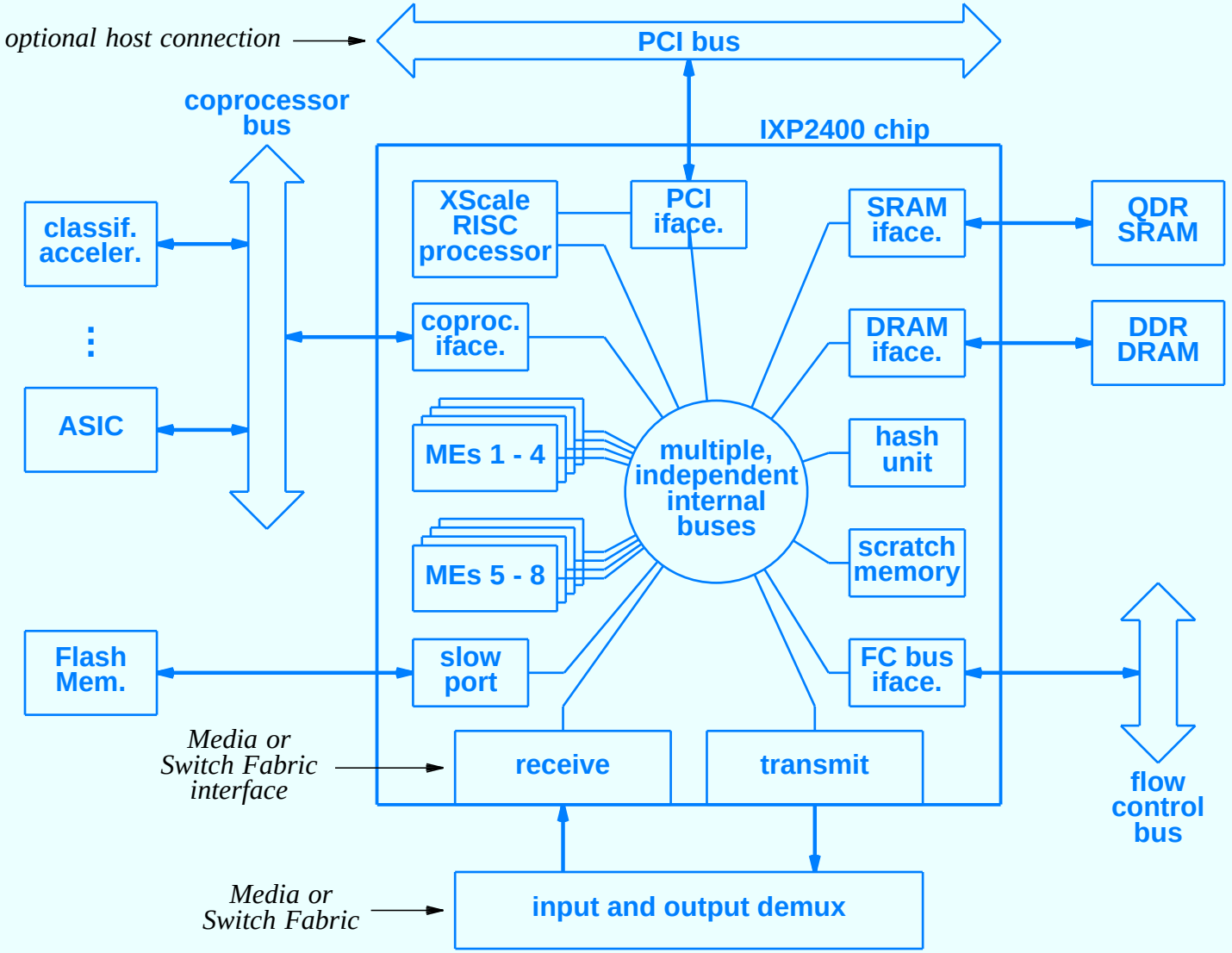
External Connections And Buses

- No IX bus
- High-speed interfaces attach to Media or Switch Fabric (MSF) interface
- MSF configurable to
  - Utopia 1, 2, or 3 interface
  - CSIX-L1 fabric interface
  - SPI-3 (POS-PHY 2/3) interface

Illustration Of External Connections



Internal Architecture



Microengine Enhancements

- Multiplier unit
- Pseudo-random number generator
- CRC calculator
- Four thirty-two bit timers and timer signaling
- Sixteen entry CAM
- Timestamping unit
- Support for generalized thread signaling
- Six hundred forty words of local memory
- Queue manipulation mechanism that eliminates the need for mutual exclusion

Microengine Enhancements (continued)

- ATM segmentation and reassembly hardware
- Byte alignment facilities
- Two ME clusters with independent buses
- Called *Microengine Version 2*

Other Facilities

- Reflector mode pathways
  - Internal, unidirectional buses
  - Allow microengines to communicate
- Next Neighbor Registers
  - Pass data from one ME to another
  - Intended for software pipeline

IXP2800 Features

- Same general architecture as 2400
- Sixteen microengines (1.4 GHz)
- Two unidirectional sixteen-bit Low Voltage Differential Signaling data interfaces that can be configured as
  - SPI-4 Phase 2
  - CSIX switching fabric interface
- Four QDR-SRAM interfaces (1.6 gigabytes per second each)
- Three RDRAM interfaces (1.6 gigabytes per second each)

Summary

- Intel is moving to second generation of network processors
- Three models announced
  - IXP2400
  - IXP2800
  - IXP2850
- Same general architecture, except
  - Faster processors
  - More parallelism
  - More hardware facilities
  - Newer, faster external connections


Questions?

Stop Here!

Stop Here!

Stop Here!

Stop Here!

Stop Here!


Questions?