18544: Network Design and Evaluation

Intel IXA SDK and Programming

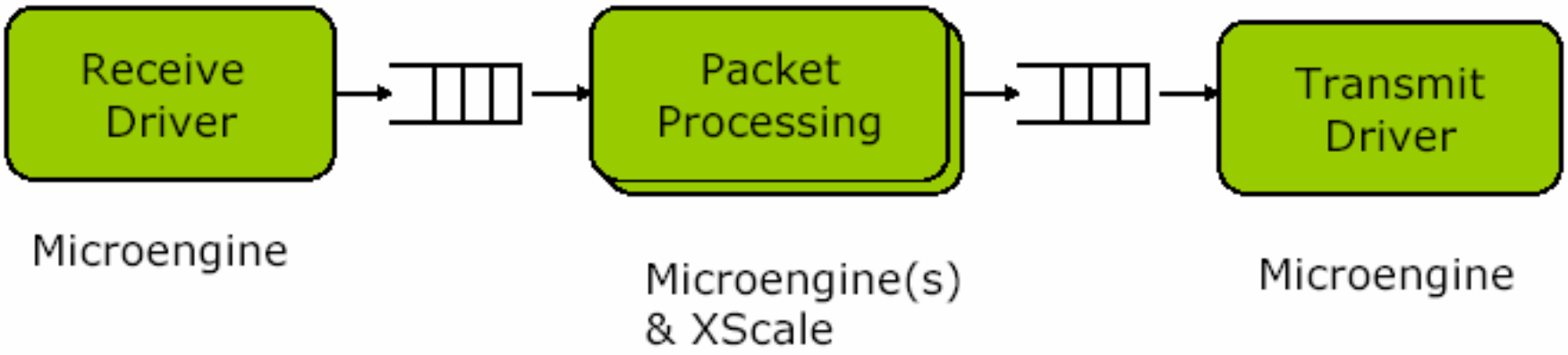
The Core Components

Sept 12, 2006

Agenda

- General concept of the IXA software framework
- Core component programming
  - Memory management
  - Message handling
  - Patching symbols
  - Packet handling
- Examples and References

Building an Application: The IXP 2400 Programming Models



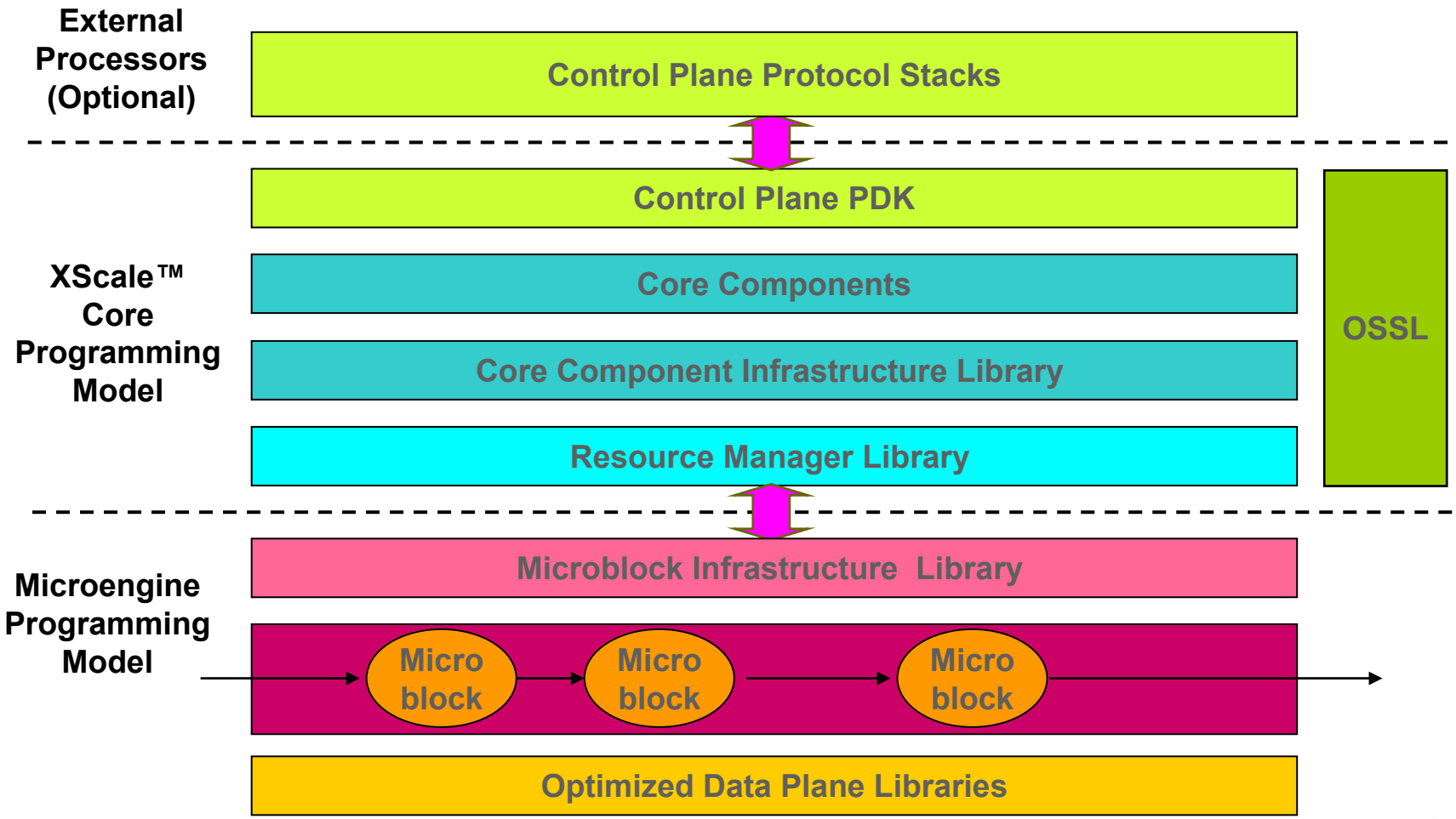
Rule of Thumb

- Microengines:
  - All, or most, of the per-packet processing
  - The data plane processing on the microengines is divided into logical networking functions called *microblocks*.
  - Each microblock should be written independently of the other microblocks to providing modularity and reusability.

Rule of Thumb (Cont.)

- The XScale Core: infrequently-arriving packet types that require more complicated processing
  - Control and configuration packets
  - Packets requiring length processing
    - IP packets with options
  - Correcting erroneous conditions
    - sending ICMP packets
  - Providing configuration and management access for the entire application, including those functions execute on the microengines

IXA Portability Framework



Optimized Data Plane Libraries

- To aid in portability
  - Code can be written for the IXP2XXXX processor and still compile and run on successive generations of the microengines
- To help perform common programming tasks
  - Calculating the checksum of an IP header can be done with a single library routine

Microblock Infrastructure Library

- Provides routines for communication with the core as well as several mechanisms for organizing groups of microblocks into processing groups

For definitive reference → Intel IXA Portability Framework Reference Manual on the CD

Resource Manager (RM)

- Runs on the Intel XScale core and manages:
  - Memory: DRAM, SRAM, scratchpad and local memory can be allocated, freed, and initialized
  - Ring and queues: Both hardware rings and queues, as well as software-based rings and queues can be allocated and accesses
  - Microengines: Microengines can be started, stopped, and loaded with new code
  - Buffers: Packet buffers and buffer freelists can be created and accessed
  - Microblock communication: Move messages between CCs and packets between CCs or between CCs and microblocks.

The Intel XScale Core Components

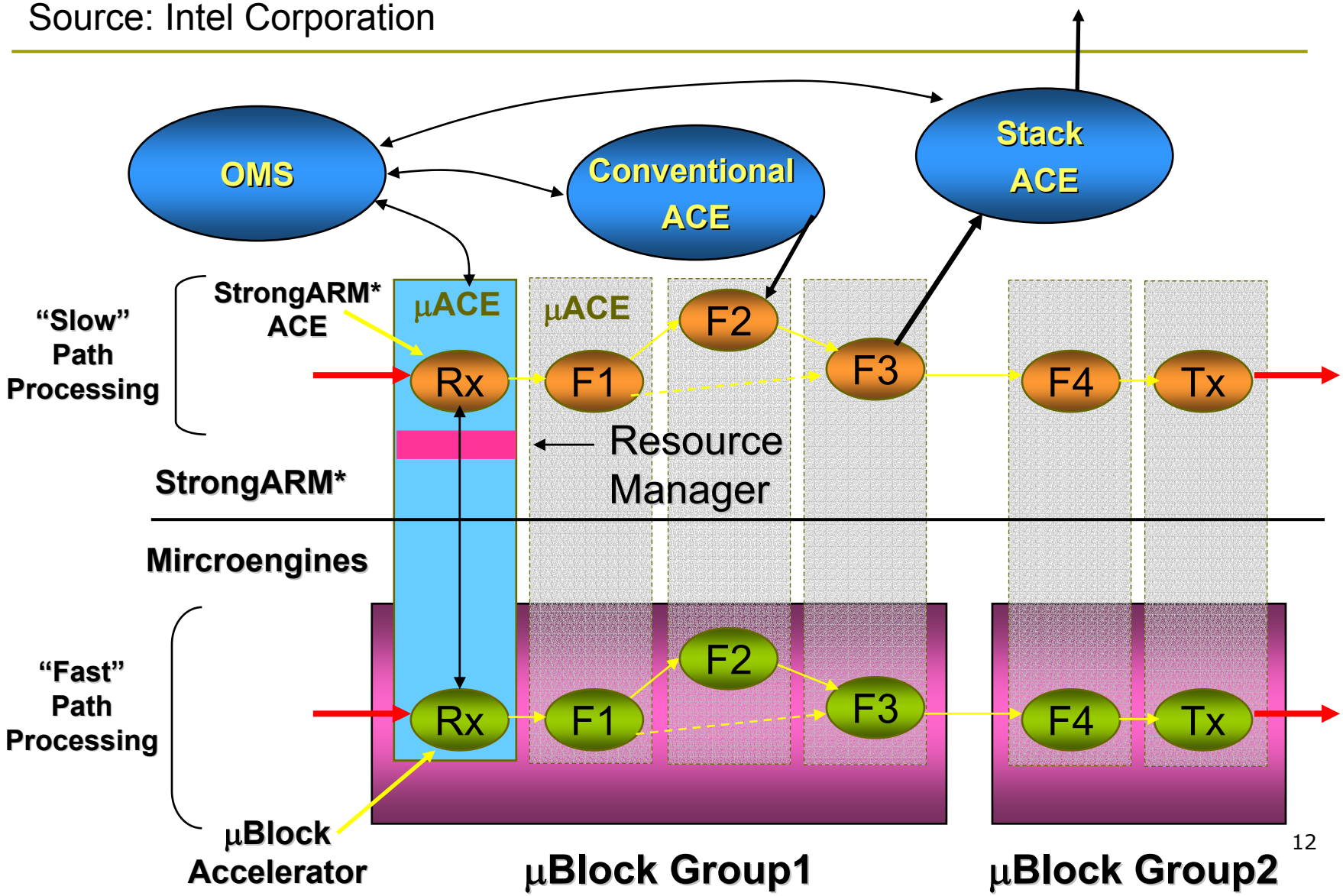
- Processes exception packets
  - May be control plane related, such as routing update message
  - May be data plane related requiring extra processing, such as IP packet with options
  - Processing packets not handled by microblocks
- Performs table management and configuration
- Configure and control microblocks

Core Component Infrastructure (CCI) Library

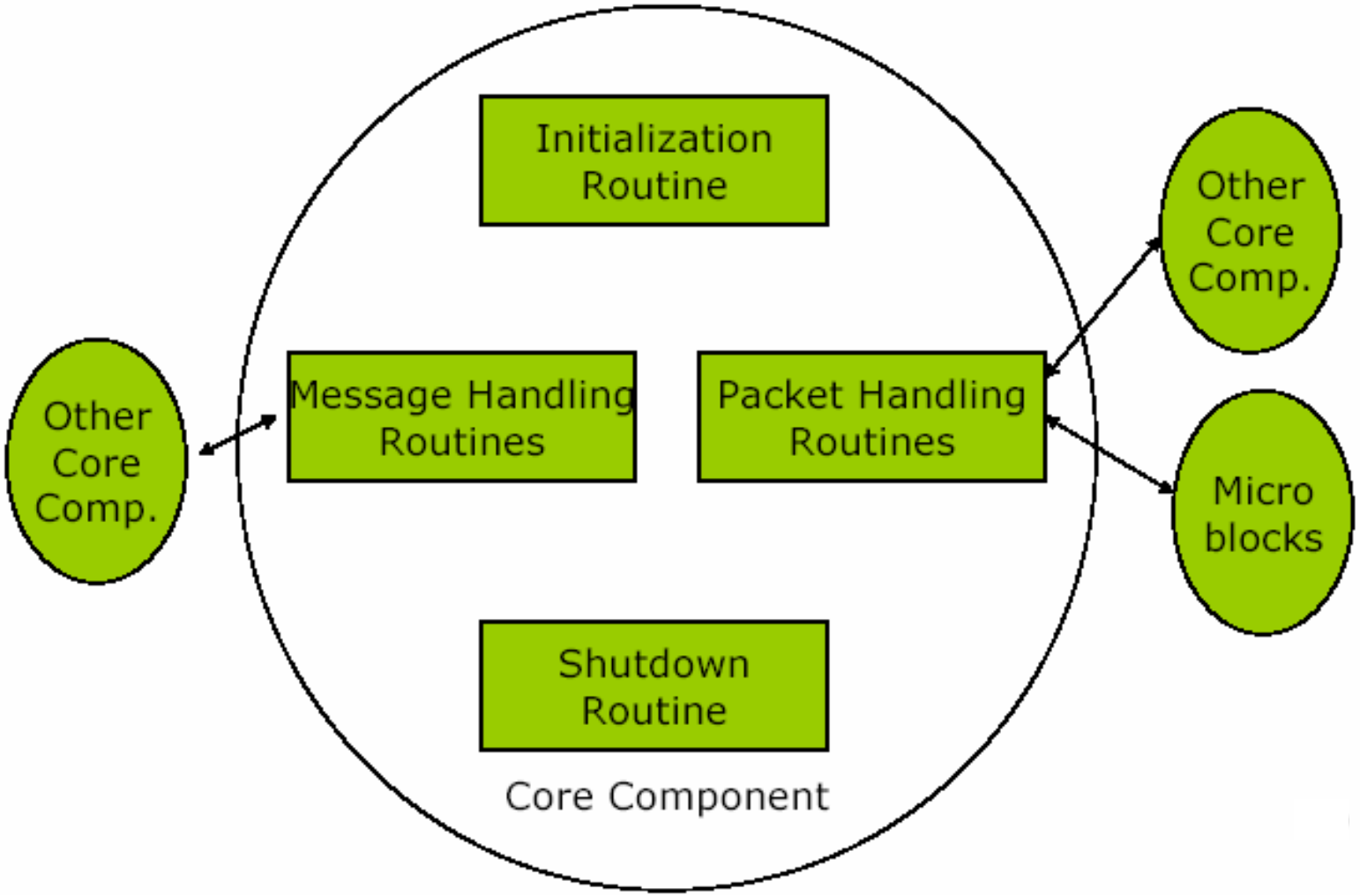
- Aids in development of core components
  - Packet communication channels to microblocks from core components and vice versa
  - Message passing between core components
  - Generic communication channels between core components
  - Flexible model for controlling the scheduling and execution of core components in the Intel XScale operating system threads
- Insulates programmers from details of resource manager

Constructing Network Applications

Source: Intel Corporation



Structure of a Core Component



Working of a Core Component

- Allocate memory for the context
- Allocate memory required by the application
- Handle packets or messages as and when required
- Send packets or messages as and when required

Using a Core Component

- Anything with a lookup table in memory needs core to maintain these in-memory data structure
- Intel IXA SDK provides framework for implementing core components
- Single core component can service multiple microblocks (though this may limit reusability)

Initializing a Core Component

- An initialization routine
  - Allocate memory to be shared with a microblock
  - Establish communication channels with other core components, the control plane or the microblock
- Signature for init function must follow this specification

```
typedef ix_error (*ix_cc_init)
    (ix_cc_handle hCC, void** pContext);
```

Example of an Init. Routine

```
//Allocate memory for Context
eth_context = (ethernet_context*)
  ix_ossl_malloc(sizeof(ethernet_context));
```

Where is memory allocated?

```
if (eth_context == NULL) {
  return IX_ERROR_WARNING(IX_CC_ERROR_OOM,
     ("Failed to allocate memory for context"));
}
*context = eth_context;
```

Terminating a Core Component

- A termination/shutdown routine
  - Release any resource owned by the core component
- The shutdown function is called when the core component is being shutdown
- Signature for shutdown function must follow this specification

```
typedef ix_error (*ix_cc_fini)
    ix_cc_handle hCC, void*arg_pContext);
```

CC Init/Shutdown Routine

- Initialization:

  ./tx/core/ethernet_tx/source/ix_cc_eth_tx_init.c
  Ln 671:

```
ix_error ix_cc_eth_tx_init(
    ix_cc_handle arg_hCcHandle,void**
    arg_ppContext)
```

- Shutdown/Termination:

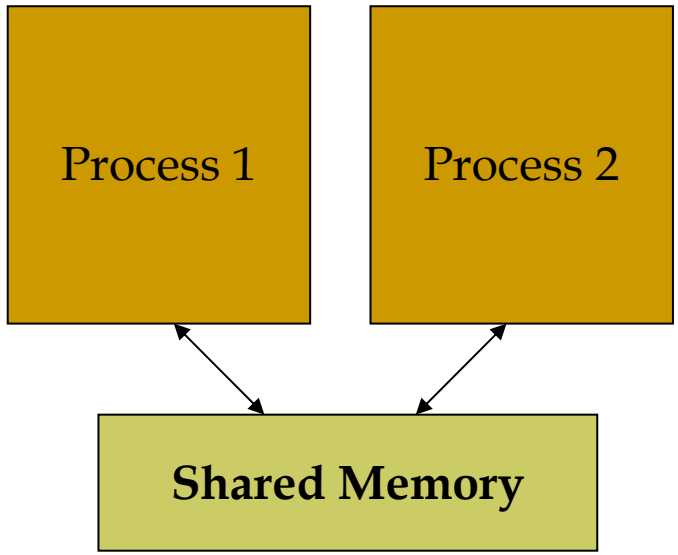
  ./tx/core/ethernet_tx/source/ix_cc_eth_tx_fini.c
  Ln 293:

```
ix_error ix_cc_eth_tx_fini(
    ix_cc_handle arg_hCcHandle, void*
    arg_pContext)
```


Trace code

Recap: Shared Memory

- In software the term shared memory refers to memory that is accessible by more than one process, where a process is a running instance of a program.



Why Shared Memory?

- Used to support system-wide data structures (e.g. route table, arp table, MAC filters etc.) among core components and microengines
- Doing the copy can be time consuming when transferring large quantities of data
- Microengines do not support memory management (e.g. handle concurrent requests)

Your project will definitely need to do memory allocation!

Memory Management [1]

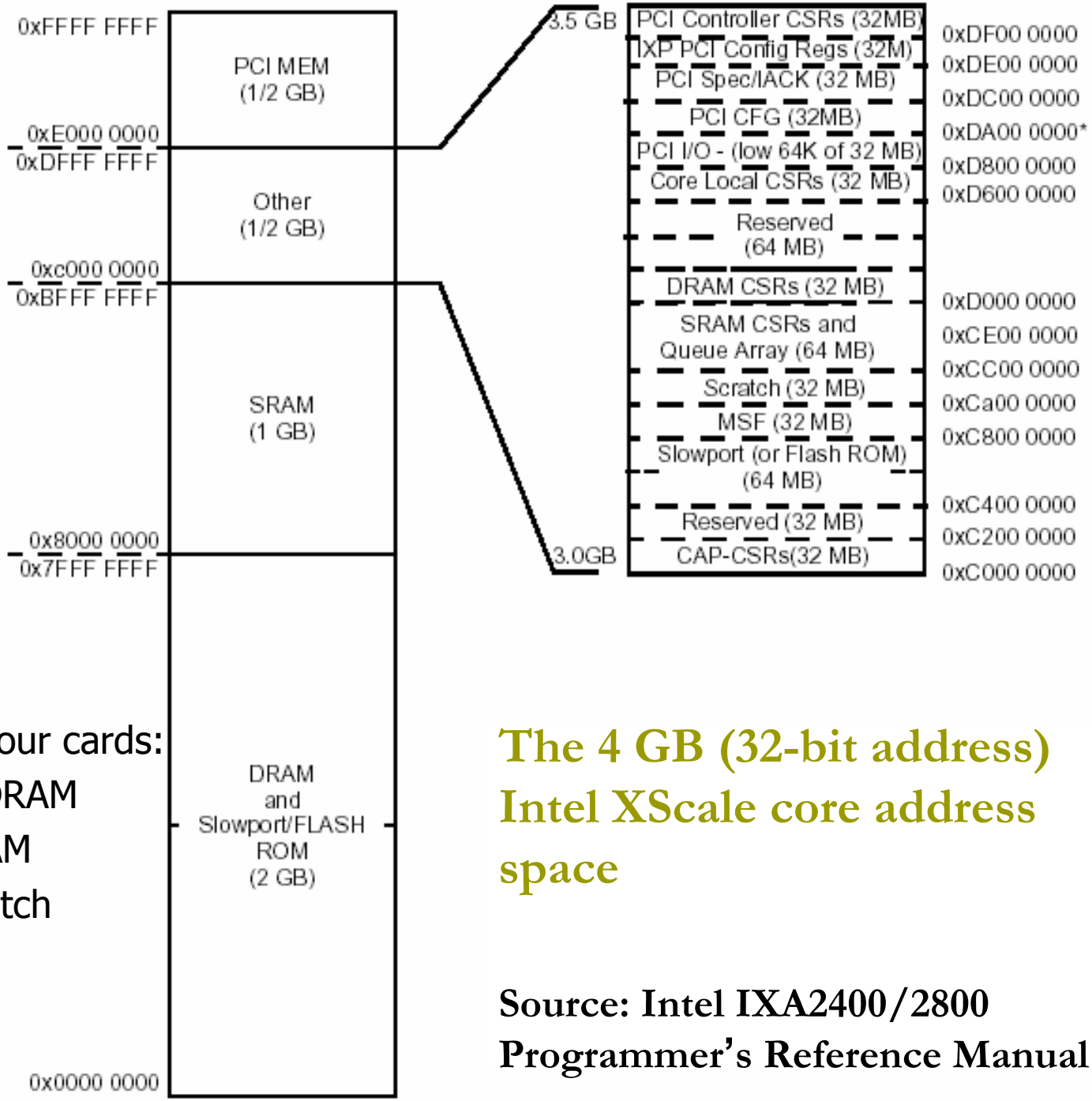
- Memory used by core components managed using malloc and free
- Resource Manager manages memory accessed by the microengines
- To allocate memory from the Intel XScale core using Resource Manager

```
IX_EXPORT_FUNCTION ix_error
   ix_rm_mem_alloc(
        ix_memory_type arg_MemType,
        ix_uint32 arg_MemChannel,
        ix_uint32 arg_Size,
        void** arg_pMemoryAddr);
```

Memory Management [2]

- Address microengines use may be different than Intel XScale core
- For example, if you want to give an address returned by `ix_rm_mem_alloc` to the microengines, convert it to an address which can be used by microengines using `ix_rm_get_phys_offset` function
- Also tells the memory type and channel if required

Trace code



Available on our cards:

- ✓ 256 MB SDRAM
- ✓ 64 MB SRAM
- ✓ 16 KB Scratch

The 4 GB (32-bit address) Intel XScale core address space

Source: Intel IXA2400/2800 Programmer's Reference Manual

Implementation Note

- The Resource Manager memory management is designed to handle one-time memory allocation
- Applications that require handling a large number of allocation and free operations dynamically need to obtain enough memory from the RM and manage it themselves

Patching Load-time Constants [1]

- A way of communication between the XScale core and microengines
- Used in patching memory locations determined when the core component is initialized
  - The values of these constants are not known at compile time, but rather are determined at the code loading time
  - The values cannot be changed once they are set (can be only changed by stopping and reloading the microengines)

Patching Load-time Constants [2]

- Signature of load-time constants:

```
IX_EXPORT_FUNCTION ix_error
    ix_rm_ueng_patch_symbols(
          ix_uint32 arg_MENumber,
          ix_uint32 arg_SymbolsNumber,
          const ix_imported_symbol
          arg_aSymbols[]);
```

- In microengine assembly code:

```
.import_var ETHERNET_DATA
```

or in microengine C:

```
int ETHERNET_DATA =
    LoadTimeConstant ("ETHERNET_DATA");
```


Trace code

Recap: Messages

- Messages, to be short, are various notifications sent to a process in order to notify it of various events.
- Each message has a message handler, which is a function that gets called when the process receives that message. The function is called in "asynchronous mode", meaning that no where in your program you have code that calls this function directly.

Why Using Messages?

- Used by the core component to notify other core components that some event occurred, without these core components needing to poll for the event.
- Examples of message handling jobs
  - Dump/Purge/Modify ARP cache
  - Dump/Purge/Modify route entries

Message Handling in Core Components [1]

- One or more message handler routine
  - Control or configuration message (not necessarily a packet)
- Take input from or give input to other Intel XScale core code
- Core components have functions that are called when receiving messages

```
typedef ix_error (* ix_msg_handler) (
    ix_buffer_handle arg_hDataToken,
    ix_uint32 arg_UserData,
    void* arg_pComponentContext)
```

Message Handling in Core Components [2]

- During initialization, the core component registers its message handlers using the `ix_cci_cc_add_message_handler` function
- The function to add a message handler is:

```
ix_error ix_cci_cc_add_message_handler(
  ix_cc_handle arg_hComponent,
  ix_uint32 arg_InputID,
  ix_msg_handler* arg_Handler,
  ix_input_type arg_SourceType);
```


Trace code

Message Handling in Core Components [3]

- The core component uses `ix_cc_msup_send_async_msg()`, `ix_cc_msup_send_msg()`, `ix_cci_send_message()` to send messages
- The dispatch loop handles packets that come from the Intel XScale core component and steers them to the appropriate microblock.

Trace code

Packet Handling in a Core Component [1]

- One or more packet handler routine
  - Process exception packets, or packets from the control plane or other core components or microblocks
- OS might want to send packets to microengines too
- The signature of the handler function:

```
ix_error (* ix_pkt_handler) (
  ix_buffer_handle arg_hDataToken,
  ix_uint32 arg_ExceptionCode,
  void* arg_pComponentContext)
```

Packet Handling [2]

- Similar to handling messages. Just that `ix_cci_cc_add_packet_handler` is called to register the handler in the core:

```
ix_error ix_cci_cc_add_packet_handler(
  ix_cc_handle arg_hComponent,
  ix_uint32 arg_InputID,
  ix_pkt_handler arg_Handler,
  ix_input_type arg_SourceType );
```


Trace code

Packet Handling [3]

- Core components can send packets to microblocks or other core components using `ix_cci_send_packet()`
- When microblock wants to send packets to its core component:
  - it sets the next block value to the unique packet handler identifier (i.e. Input ID)
  - The microblock can also set a 32-bit exception code

Trace code

Packet Handling [4]

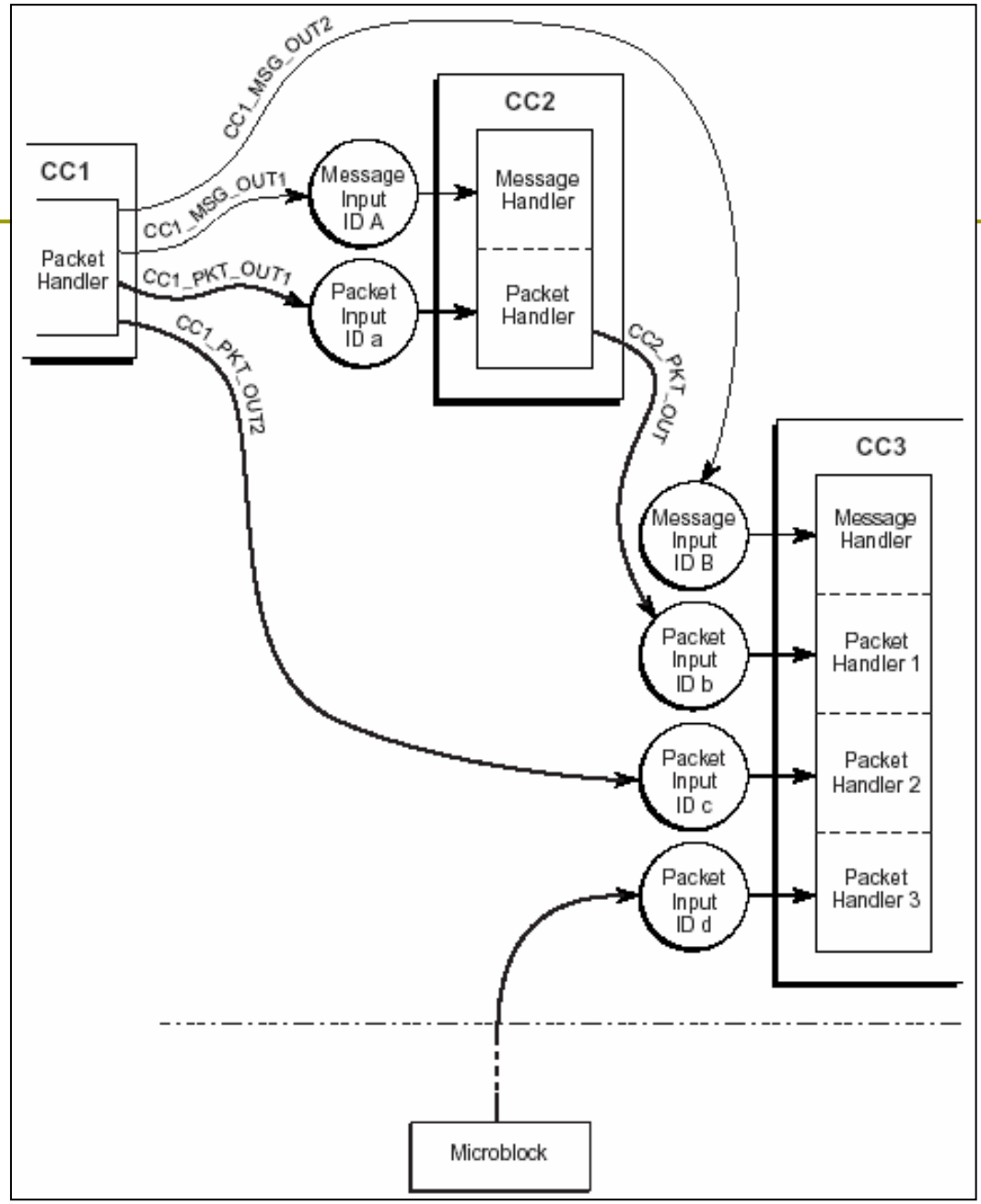
- The following microengine C code sends packet to the ethernet core component:

```
// If the entry is not valid, drop this
packet
if(!table_entry.valid) {
    dl_set_exception(ETHERNET_EXCEPTION_ID,0);
    dlNextBlock = IX_EXCEPTION;
    return;
}
```

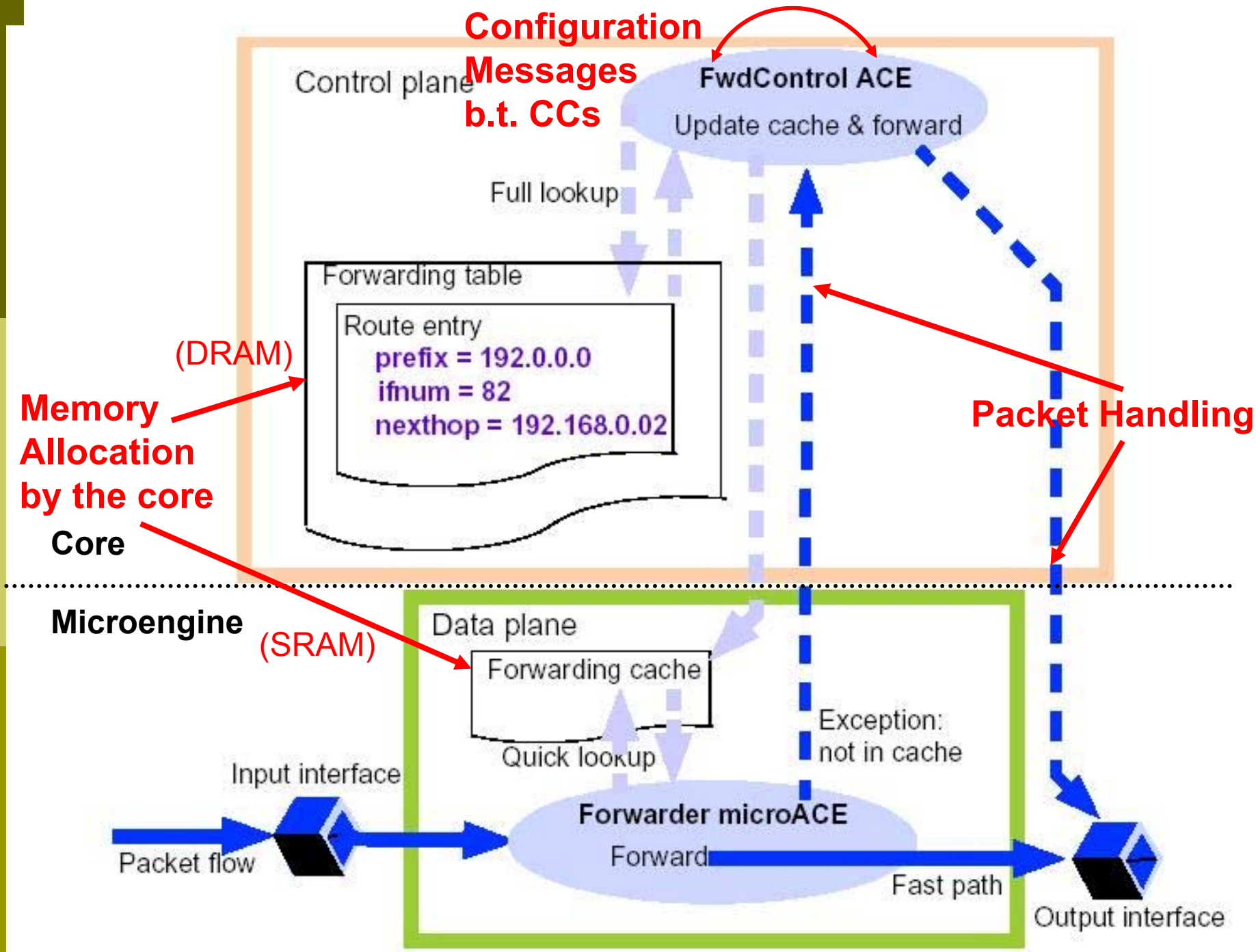
Dispatch loop will catch this packet and notify the core

Good Reference: IAX Framework Development Manual

Example:
Packet and Message Data Paths between core components



Source: IXA Portability Framework Developer's Manual


Configuration Messages b.t. CCs
Control plane
FwdControl ACE
Update cache & forward
Full lookup
Forwarding table
Route entry
prefix = 192.0.0.0
ifnum = 82
nexthop = 192.168.0.02
(DRAM)
Memory Allocation by the core
Packet Handling
Core
Microengine
(SRAM)
Data plane
Forwarding cache
Quick lookup
Exception: not in cache
Input interface
Packet flow
Forwarder microACE
Forward
Fast path
Output interface

Summary

- Understand the basic concept of a core component
- Inter-communication between core components and microengines – shared memory, message and packet handling
- Introduce basic core component programming
- Programming Examples

Project Base Code

- http://www.cs.cmu.edu/afs/andrew.cmu.edu/course/18/544/www/dl/proj_544fall04_core.tar.gz

References

- **IXA Architecture Portability Framework Developer's Manual**
  - Resource Manager overview and API list
  - Core Component overview and API list
  - Microblock architecture and functions
  - Other useful information for development and debugging
- **IXA Architecture Portability Framework Reference Manual**
  - Signatures and Syntax of APIs

References (Cont.)

- **Useful APIs for programming:** http://www.cs.cmu.edu/afs/andrew.cmu.edu/course/18/544/www/handouts/CoreCompMemMgmt.pdf
  - Most commonly used Message/Packet Handling APIs that you will need in doing projects
- **Intel® IXP2400 and IXP2800 Network Processor Programmer's Reference Manual**
  - MEv2 instruction set
  - Assembler
- **Intel IXAP SDK Software Framework CD** (also available on the course site)

Questions?