

GEORGIA INSTITUTE OF TECHNOLOGY

College of Computing

CS8803J — High-Performance Communication

Spring 2002

CS8803J
Project 1

Issued: January 4, 2002
Due: January 23, 2002

Purpose: This project introduces networking components with concrete examples. The approach is at the lowest available level of abstraction – assembly programming with device registers – in order to reveal (and to force you to understand) all details. The work is mostly done on a simulator with the final product demonstrated on hardware.

Reading: IXP1200 Dev Tool UG
IXP1200 Datasheet
IXP1200 HW Ref Manual
IXP1200 Prog Ref Manual
IXF440 Datasheet

Problems:

1. IXP1200/simulator warmup: memory copy.
2. IXF440/packet generator warmup: 100T receiver.
3. Performance measurement: transmitter + receiver.
4. Hardware performance measurement.

To turn in: Turn in a write-up that describes your solutions to the problems including all codes and performance data.

Collaboration: (*As in the syllabus*) collaboration on projects and homework in **pairs** is encouraged. If you work in a pair, turn in one write-up with the names of both collaborators. You're welcome to discuss high-level concepts with other groups, but all homework solutions must be worked out and written up separately.

NOTE: support files for this assignment are available via NFS on CoC machines in the following directory:

`/net/hp82/softarch/cs8803j`

In this project you will build several pieces of networking infrastructure that will be useful in subsequent projects: a programmable packet generator and a statistic-gathering packet receiver. The main point of the project, however, is to cause you to absorb a *large* amount of information about the IXP system: the hardware, the programming model and the workflow used to prototype applications.

The approach of this first project is deliberately low-level so that no details are obscured. Subsequent projects will use programming abstractions that will give you greater productivity and power but for now we start with the bare metal.

Do not underestimate the amount of time needed for this project: the individual problems below are simple to describe and simple in concept but will require a lot of reading and experimenting to implement.

Problem 1: Treasure Hunt

This problem could be a beforehand-warmup or an afterward-cooldown problem, your choice. While the other problems have you implement various features (thus causing you to read lots 'o stuff), this problem is more like a treasure hunt through the documentation for architectural or software goodies. Note that these items are hardly exhaustive or even representative; just a few things I think are cool.

A: Where are the two memory banks in DRAM and how do you contrive (in microengine code) to generate requests to alternating banks? How should you place data structures like packet queues in memory for best performance? Note that both the “HW Ref” and “Prog Ref” manuals have sections on the DRAM controller.

B: How many outgoing packets can be buffered internally in the 100T MAC (not the TFIFO in the IXP, the MAC itself? How do you avoid overrunning this internal buffer? Note that in addition to the HW Ref and Prog Ref manuals for the IXP chip, you can also learn much from the data sheet for the MAC chip (IXF440).

C: How do you implement a critical section (e.g. with a mutex) between microengine threads on different microengines? How do you implement a mutex between the microengines and the StrongARM?

D: How do you capture outgoing network packets from the IXP under simulation?

E: The IXP has an elaborate status collection mechanism for the IX bus – somewhat like a fancy interrupt controller – called the “ready bus”. What has to happen to let a thread read MAC status from a local register?

Problem 2: Memory Copy

This problem is intended to introduce the IXP1200 microengine assembly language and Intel's Integrated Design Environment (IDE) for the IXP1200 microengines. The problem given (a memory copy) is contrived but introduces many of the odd features of the microengines. Also, it's *hard* to get it to work at full memory speed. We haven't figured out how to do so yet and we're curious to see what you come up with!

A: Find the IDE (on the NT/Win2K machines in the first floor of the CCB) and figure out how to write microengine assembly language. Aside from the manuals (HW Ref Manual, Prog Ref Manual), probably the best way to get started is to look at the example codes for the next to problems, load that into the IDE and then start modifying it.

B: Your task is to copy a 1MB block of memory from one place in SDRAM to another place in SDRAM. Use the IDE to write microengine code and simulate it. Set the IDE to a 232MHz chip to match the ENP2505 boards we have. You can use the cycle counts reported by the simulator to compute MB/S (megabytes per second).

C: Tune the memory copy for maximum memory system performance by exploiting any techniques you choose – two-phase requests, memory banks, multiple threads, multiple microengines, etc. You may split up the 1MB block of memory into separate blocks if that helps. Note that based on the SDRAM bus bandwidth, the maximum rate should be 466MB/S – the eight-byte bus running at half the chip core frequency gives 932MB/S and the memcopy will get half of that because data must cross the bus twice.

D: In your writeup note how close you get to full bandwidth (400MB/s) and the resources (# of threads) required. Detail your approach and critique your approach in retrospect.

E: OPTIONAL: test the memcopy on the real hardware. It's not clear that the simulator properly models the details of the SDRAM controller.

Problem 3: Packet Receiver

This problem uses a simple packet receiver to introduce the 100T MAC chip (the IXF440), the IXP1200's interface to MAC chips and the IDE's support for generating test packets to the simulator. These items are important to understand for themselves and also as building blocks for later work. Also, the code you generate here is useful for collecting statistics about network streams so you can use it as a piece of "test equipment" in subsequent projects.

A: Use the packet generator in the IDE to test the simple receiver code in `octalrecv.uc`. The code is available on-line. Test the code by sending it a stream of minimum-size packets.

B: Modify the receiver to receive packets of any size, to receive from all ports simultaneously, to count the packets received from each port, to record the last source MAC address seen and to histogram the sizes of packets from any port. The data collected should be placed in SRAM where the StrongARM core can read them. A C definition of the structure is on the next page (excerpted from `receiver.h`, available on-line).

C: Verify that your code works at full speed using a packet generator script on the simulator. In the writeup, describe your approach and critique it.

D: OPTIONAL: you can't really try out the receiver on real hardware until you have a transmitter. However, if you do the first part of the next problem, you could use that code to transmit packets then come back here and test them on the hardware.

```

typedef struct
{
    unsigned int npackets;        /* count of packets received */
    unsigned int nbytes;         /* count of total bytes received */
    char lastmac[6];             /* last ethernet MAC seen */
} portstats_t;

typedef struct
{
    portstats_t portstats[4];    /* statistics for each port */
    unsigned int histogram[24]; /* histogram of packet sizes seen */
} receiver_t;

/*
 * Address in SRAM. This address is arbitrary but must be agreed-upon
 * by both the StrongARM and the microengines. To the microengines,
 * An SRAM address is exactly an index in longwords. On the
 * StrongARM, the SRAM is mapped at byte addresses into a region of
 * virtual memory.
 */

#define RECEIVER_SRAM_INDEX 0x2000 /* index in longwords into SRAM */

#define thereceiver ((receiver_t *) (SRAM_BASE
                                     + (RECEIVER_SRAM_INDEX
                                          * sizeof(long))))

```

Problem 4: Packet Transmitter

This problem uses a simple packet transmitter to complete your set of test equipment.

A: Test the simple transmitter code in `octaltrans.uc` using the IDE with the option to log packets sent. As written, the code should send a stream of minimum-sized packets with fixed data in them. Verify that you see these packets in the IDE.

B: Modify the transmitter code to transmit packets of various sizes from buffers in DRAM to various ports as directed by instructions in SRAM. The structure to use is given below (`transmitter.h` online).

```
typedef struct
{
    int port;                /* port for this packet */
    int next;               /* index of next packetinfo_t (or -1) */
    unsigned int sdram_index; /* ptr to packet in SDRAM */
    unsigned int sdram_nbytes; /* total length of packet in bytes */
} packetinfo_t;

#define TRANSMITTER_SRAM_INDEX 0x3000 /* offset from base of SRAM */

#define thetransmitter ((transmitter_t *) (SRAM_BASE
                                           + (TRANSMITTER_SRAM_INDEX
                                              * sizeof(long))))
```

C: Verify that your code works at full speed on the simulator. You will have to spend some effort building the structures in memory to get this to work.

Problem 5: Hardware

The final problem is to get the transmitter and receiver you've built above to work on the real hardware. When you get it working, demonstrate your implementation to the instructor. We suggest the following procedure:

A: Verify that the two pieces of your code run using the IDE connected to one of the ENP2505 boards in the lab. You can telnet to the StrongARM boards (ilab[1-6]-ixp1) and you can point the IDE at these boards also. Note that despite all efforts the boards frequently require rebooting. We'll post instructions on how to do this.

There are several options for testing the two programs together

- Use two IDE sessions to connect to two separate ENP2505 boards.
- Smoosh both programs onto one IXP by using multiple microengines. Then you can use one IDE session and one ENP2505 board.
- Use locally developed tools (to be described later) to load and start the microengines from the StrongARM.