An Integrated Laboratory for Computer Architecture and Networking

Takamichi Tateoka, Mitsugu Suzuki, Kenji Kono, Youichi Maeda, and Kôki Abe

Department of Computer Science
The University of Electro-Communications
Tokyo, 182-8585 Japan
Email: cnp@cacao.cs.uec.ac.jp

Abstract

Processors, compilers, and networks – important materials covered by computer science curricula – are often treated independently in laboratories associated with corresponding lecture courses. An integrated laboratory called CNP for juniors majoring in computer science at the University of Electro-Communications has been developed and is now under way, where a networking protocol stack implemented by students is translated into object codes by a compiler implemented by students, which in turn are executed on a processor implemented also by students. The goals of the integrated laboratory are to deal with modern and attractive materials, to provide students with opportunities of collaborating in constructing a large system, as well as to have students share a feeling of accomplishments among them. Responses from students approved our intention and verified the effectiveness. In this paper, we describe the design and development of baseline components to be integrated, laboratory organizations and schedules, and results and evaluations of the laboratory.

1 Introduction

Processors, compilers, and computer networks are important materials covered by computer science curricula. They are often treated independently in laboratories associated with corresponding lecture courses. Many reports on laboratories dealing with microprocessor design and implementation have been published (eg.[1]). Exercises on compiler design are too common to mention. Some reports on computer networking laboratory exist[2], although it has been recognized in the Computing community that academic institutions should treat computer networking to more fully extents[5].

However, in order for improving cost performance of a computer system, tradeoffs between hardware and software must be well understood and the characteristics of applications executed on the system need to be carefully examined. Adjusting interfaces between system components is also required. Thus taking a broad view of entire system is mandatory. For students to acquire the view, separate components need to be integrated into a complete system in a laboratory.

A design problem across areas can effectively be solved in a short term by teamworking, where the problem is divided into parts and works by team members are shared and combined. Providing students with opportunities to have experiences of such teamworking in university laboratories dealing with design and implementation of both hardware and software for modern and attractive applications is of key importance.

An integrated laboratory called CNP for juniors majoring in computer science at the University of Electro-Communications (UEC) has been developed and is now under way, where a networking protocol stack (called TinyIP) implemented by students is translated into object codes by a compiler (called Tiny C) implemented by students, which in turn are executed on a processor (called MinIPS) implemented also by students. The whole system integrated by students in the laboratory is called Tiny J.

Students are organized into several teams. Members of a team cooperatively perform the laboratory experiments. The goals of the CNP laboratory are for each student to understand the interfaces between system modules, to design and implement an assigned one, to integrate cooperatively the components into a system, as well as to discuss and adjust their specifications.

In the following, Section 2 describes laboratory design and developments of baseline components to be implemented and integrated by students. Related courses offered to students are also stated in this section. Section 3 describes the details of the laboratory including stu-

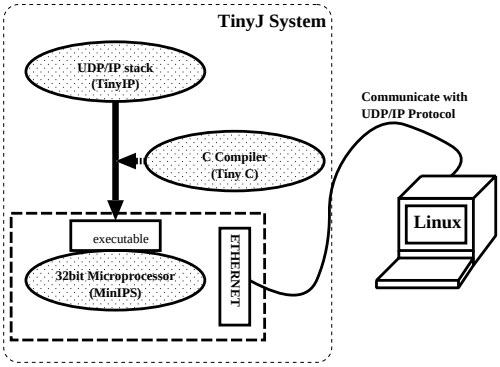

Figure 1: Illustration of the integrated laboratory.

dent organization and schedule of the laboratory course. Section 4 gives results and evaluations of the laboratory. Section 5 closes with a summary and future works.

2 Design and Developments

2.1 Laboratory Design

The laboratory is designed to amalgamate UDP/IP protocol stack, a C compiler, and a 32bit RISC processor. Students integrate these materials into a complete system (Tiny J) to construct a complete small computer system which is capable of communicating with Linux OS through Internet standard UDP/IP protocol. A simple protocol stack (TinyIP) coded in a simplified C language is translated into object codes by a compiler (Tiny C) implemented by students, which in turn are executed on a processor (MinIPS) implemented also by students. The overview of the integrated laboratory is illustrated in Figure 1.

We design the laboratory so that students can design, implement, and modify all parts of the system components. Students, however, are not assigned to design every submodule because of restricted laboratory hours. We provide students with information enough for them to inspect any part. For example HDL descriptions of peripherals used for console function and source codes of original compiler which are to be extended by students are given to them.

We summarize related lectures and laboratories referring to the Computing Curricula 2001 by IEEE CS[5]. Lectures covering AR1 to AR5 of Architecture and Organization and PL1 to PL6 of Programming Language are offered as core, accompanied with corresponding labs. AR6 (Functional organization) is covered by an elective sophomore course, where [6] is used as a textbook. PL8 (Language translation system) is covered by an elective junior course, where a simplified C compiler Tiny C[8] is introduced and designed. Fundamentals of OS are introduced in a requisite course but the topics are intensively treated in an elective junior course. Topics in Net-Centric Computing are treated in an elective senior course, where principles of communication networks with OSI layered architecture are introduced.

The CNP is offered as a requisite junior laboratory course. Since we can not expect special knowledges given by elective lectures or lectures offered in future courses, when necessary we provide students with practical lectures required to complete the assignments in laboratory hours.

2.2 MinIPS Computer System

Requirements for the processor to be developed are: 1)To be simple and modern as an educational processor given to computer science juniors; 2)To have enough performance that allows building a computer system using the processor as a CPU core in Tiny J System; 3)To conform to the processor dealt with by [6] used as a textbook in the corresponding lecture course.

Requirements for the computer system based on the processor are: 1)To accommodate a communication port through which the protocol stack transmits and receives packets; 2)To equip with enough amounts of memory for programming TinyIP; 3)To have functions for loading programs and acting as a console.

We utilize a SRAM based FPGA as the implementing device used by students. It makes students to redesign the processor any times without care of making errors. An evaluation board, system-on-a-programmable-chip (SOPC) Development Board by Altera, equips with an FPGA which is capable of realizing 400,000 gate logic circuits using Logic Elements (LEs) and 20 KB memory using Embedded System Blocks (ESBs). The board also equips with Synchronous SRAM (SSRAM), RS232C, and Ethernet transceiver, which enable to organize a system meeting the requirements with no other hardware supplements. For those reasons we have chosen to utilize the SOPC Board for implementing the system.

We do not use any commercially available intellectual properties for FPGA configurations such as Ethernet controllers because intellectual properties would introduce black boxes. Thus all the prototype modules including the peripherals have been designed by ourselves. Although students are not assigned to design the peripherals, our descriptions are given to students so that whoever interested is ready for reading.

The organization we have designed is shown in Figure 2.

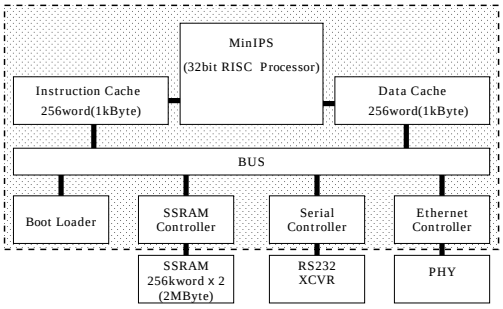

Figure 2: MinIPS system organization. (Enclosed with dashed box is the part implemented on an FPGA.)

The MinIPS processor[3] is a 32 bit RISC which conforms with MIPS[6]. The MinIPS instruction set is reduced to a minimum. For example, multiplication and division are not provided as machine instructions but are compiled to subroutine calls. The block diagram of the processor is shown in Figure 3. Conforming to the textbook, the structure is composed of five pipeline stages; instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB). A forwarding unit (FW Unit) is equipped. The load delay is one, and the branch delay is also one. The MinIPS processor conforms to the textbook but it is simpler.

Although the MinIPS processor is based on Harvard Architecture, it does not allow fetching instruction and data simultaneously because the memory is not physically divided into instruction and data submemories. As a solution to the problem we provide instruction and data caches which we implement using ESB memories on FPGA. Two SSRAM chips each of 256k words capacity mounted on the SOPC board are utilized for the main memory. It has enough capacity for programming TinyIP.

The specification of the RS232C controller is based on the simulator SPIM[6], enabling a smooth shift from simulations to executions on real machine. We use the RS232C for program loading and console function.

For the communication port, we adopt the commonly used Ethernet where a link layer address is provided. As the physical layer a PHY chip on the SOPC is utilized. Descriptions of the link layer for controlling the PHY has been developed by ourselves[4] and given to students. The interface is driven by hardware interrupts since polling is not a practical method for receiving Ethernet packets.

For loading programs and acting as a console, two programs, boot loader and monitor, are provided. The boot loader is stored in a ROM area implemented on ESB. It is initially executed for loading programs upon turning on the power. The monitor is loaded through the boot loader. It provides such console functions as displaying the memory contents, loading programs and data into memory, displaying the contents of registers, modifying the contents of program counter, and handling interrupts as well as dispatching the corresponding processes.

Using Quartus II ver.1.1, a development software tool by Altera, we describe and compile the design in Verilog-HDL, configuring the FPGA on the SOPC board. About 57% of the LE resources have been used for the configuration. The compilation requires about 25 minutes with a platform of Pentium4 1.7GHz CPU with RDRAM 512MB memory. The MinIPS system is operating at a clock rate of 16.5MHz.

2.3 Tiny C Compiler

Requirements for the language compiler to be developed are: 1)To be simple enough to understand; 2)To have enough ability that allows compiling network protocol stack and applications for Tiny J System; 3)To conform to compiler design lectured in the corresponding lecture course.

Tiny C[8] is a small subset of C language developed by Prof. Watanabe as an illustrative compiler for his compiler course. It almost meets our requirements except for lacking of support for some operators such as bitwise operators. We supplemented support for unary address operator (“&”), bitwise operators (“&”,“|”), modulo operator(“%”) and shift operators (“<<”, “>>”) into Tiny C. The supplemented version is denoted by Tiny C hereafter unless otherwise noted.

We also introduced minor modifications into SPIM emulator[6] to use it as a MinIPS emulator. The modifications include appending a memory image snapshot function which is used as a substitution for assembler and linker to obtain MinIPS object codes. Signed multiplication and division routines were added into “`trap.handler`” containing startup codes for MinIPS, since MinIPS does not support these instructions.

2.4 TinyIP protocol stack

Requirements for the protocol stack to be developed are: 1)To be simple enough to understand and easy to describe in Tiny C with simple syntax; 2)To be realistic and practical so that students feel a sense of accomplishments; 3)To be educational so that students understand features and benefits of the protocol with layered architecture; 4)To be extensible so that students can append their own ideas to the protocol; 5)To be independent of

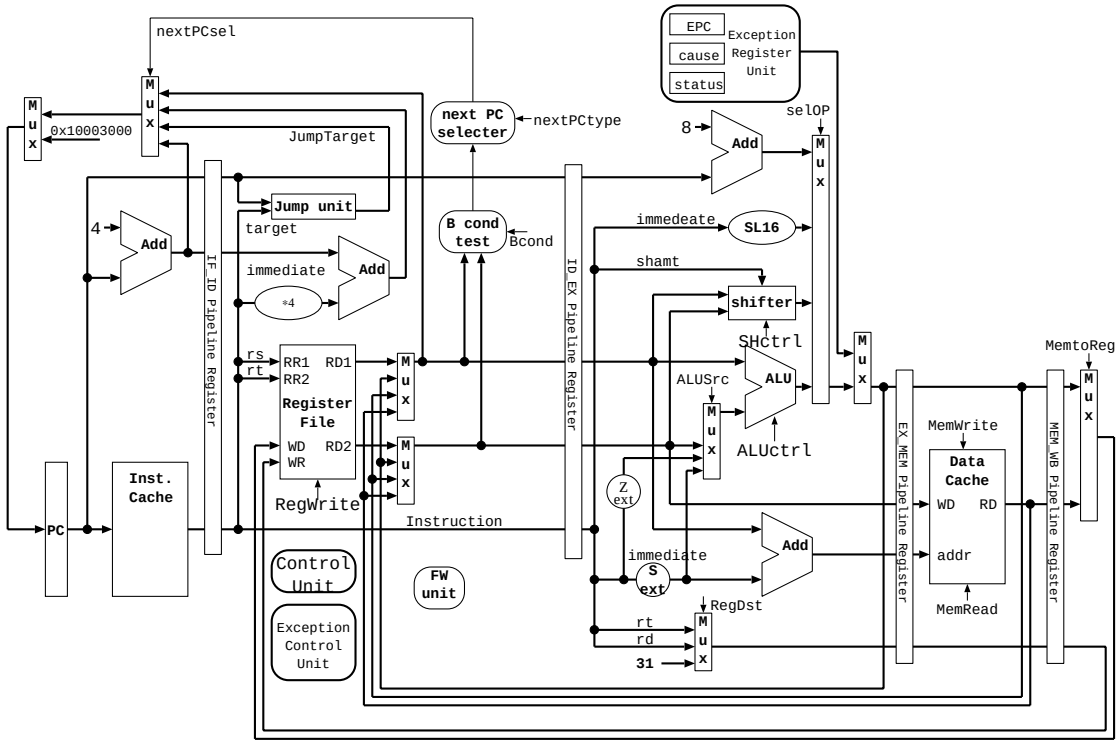

Figure 3: Block diagram of MinIPS processor.

hardware so as to allow testing before the processor becomes available.

To simplify the protocol stack, we use Internet Protocol (IP) as a network layer, and User Datagram Protocol (UDP) as a transport layer. We do not support packet fragmentation, Address Resolution Protocol (ARP), nor Internet Control Message Protocol (ICMP) processing. In spite of the limited functions, it is still capable of communicating with standard IP such as the one implemented in the Linux kernel.

We provide students with two versions of the implementations: one written in standard C language to illustrate the design of the protocol stack, the other written in Tiny C to be integrated into Tiny J System. Both of them have almost the same structure except that the latter calls for works on differences between standard C and Tiny C compilers. We basically describe the former in this section.

The stack consists of ten modules whose functions and calling flows are shown in Figure 4. They are described in separate C source files with well-defined interfaces. Receiving functions are driven by interrupts. The method of using interrupt mechanism is practical and keeps the control flow simple and conforming to the OSI seven-layer model. The structure facilitates appending new features to the stack, making the stack extensible. Hardware dependent routines are collected into one module (`hardware.c`), resulting in portability to new hardware.

The core routine of the stack consists of about 800 lines in C language with additional 400 lines of instructive comments. Tiny C version consists of about 500 lines since some features such as generic FIFO routines were omitted.

We also supplied a TinyIP compatible library for standard UDP/IP stacks on Linux. Students can build and execute application programs before completion of their own stacks.

We developed a monitoring tool `etherpeep` which allows observing ethernet frames in hexadecimal numbers (and ascii characters). It is similar to `tcpdump` command on Linux with `-x` option except that `etherpeep` displays ethernet headers. It displays whole ethernet frames in simple format.

3 Integrated Laboratory

The laboratory course offered in second semester of junior year started from year 2001. Hours assigned to the

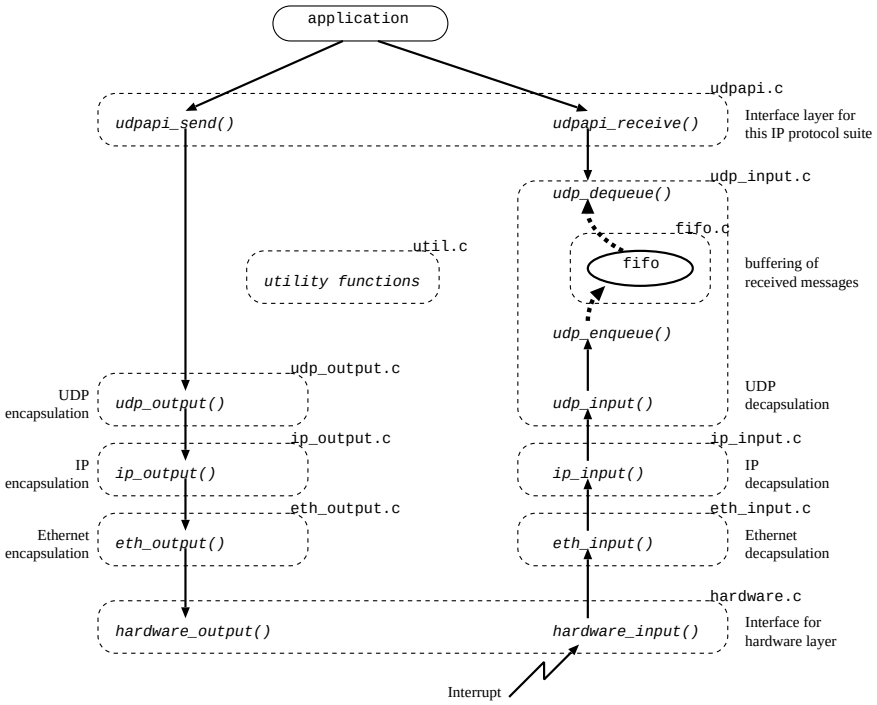

Figure 4: Functions and their calling flows of TinyIP modules.

course are divided into two periods each consisting of 12 three-hour classes. The laboratory course completes within individual periods. In each period 30 students take the course. Thus 60 students in total participate in the laboratory during the semester.

In each period students are grouped into five teams each of six students. A team consists of N (networking), C (compiler), and P (processor) subgroups who are in charge of working on corresponding sub-laboratories and cooperatively develop a complete Tiny J system.

In the P sub-laboratory students proceed along the following steps: 1)Learning how to use design tools; 2)Designing small submodules; 3)Designing arithmetic logic unit; 4)Designing pipelined processor; 5)Compiling the MinIPS system and verifying the function.

For coordinating them with other sub-laboratory assignees, we fix a minimum specification given to students. After getting familiarized with the development tools, students first design simple modules such as multiplexors and adders, and then gradually shift to designing more complex modules. When completing all the necessary modules, they start designing the entire processor.

In designing the processor they are not assigned to describe the whole of the processor. Instead, they are given a processor description with several parts taken out in such a way that the behavior of the pipelined processor can still be understood. They are assigned to supplement the incomplete design with proper descriptions. For such modules as the RS232C and Ethernet controllers, descriptions are given to students so as to enable them to verify that the processor is operating.

For testing and verifying the design several tools are provided: 1)A graphical simulator embedded in Quartus has been used throughout the experiments; 2)LEDs equipped on the board are used in the preliminary experiments as well as for monitoring states of programming execution; 3)The boot loader given to students has been effective in checking whether the MinIPS system works as a whole; 4)Test programs such as calculating prime numbers in Tiny C are given to students for more extensive debugging.

Discussions among team members on the specification of MinIPS are expected. For example, extra instructions may be added to MinIPS instruction set if an agreement is reached between C and P assignees.

Students start the C sub-laboratory by tracing the parser in the original Tiny C source code to draw a chart illustrating the syntax of the language processed by the compiler. Then they are assigned to refine the original Tiny C compiler so as to accept additional operators and literals required to implement TinyIP. The assignments are the necessary supplements described in the

previous section. Discussions among team members on the specification of their Tiny C is expected also in this sub-laboratory. Agreements in the team members may lead to changing the specification of their final version of Tiny C.

In the N sub-laboratory students proceed along the following steps: 1)Writing simple applications; 2)Analyzing ethernet frames; 3)Implementing and enhancing TinyIP for Linux; 4)Implementing TinyIP in Tiny C for MinIPS; 5)Combining TinyIP with other sub-laboratories. The fourth and the last are steps for the CNP integration requiring collabrations of team members. The integration steps are to be led by N assignees.

Students start the N sub-laboratory from learning basic network architecture and writing some simple applications. They connect two Linux boxes with an ethernet cross cable, configuring a LAN isolated from the campus network. We provide a TinyIP compatible library for Linux and a sample application with detailed documents. They utilize the library to write client and server programs satisfying echo protocol[7]. The programs are used as applications later in Tiny J.

Next they learn how the frame is encapsulated and decapsulated. They capture and analyze ethernet frames produced by standard UDP/IP stack with `etherpeep` command. They also get a good reference of working UDP/IP frames.

Students then proceed to implementing TinyIP in standard C. We provide a template of TinyIP implementation missing core functions such as encapsulation and decapsulation of ethernet, IP, and UDP frames. For verifying the implementation they connect two Linux boxes: one is configured to use standard UDP/IP stacks while the other is to use TinyIP. On both boxes they execute their client and server applications developed at the first step. They enhance their TinyIP implementation by adding some features such as ICMP and ARP, and/or by making improvements on memory consumption.

After discussions among team members to fix the final language specification of Tiny C, they implement TinyIP in Tiny C for MinIPS. They are given a template of TinyIP implementation in Tiny C and write the missing code in accordance with Tiny C specifications. They compile and test their implementations in the following three environments: 1)gcc for compilation and Linux for execution; 2)Tiny C for compilation and MinIPS emulator for execution; 3)Tiny C for compilation and MinIPS real hardware for execution. In the first environment, they can test TinyIP stack independently of Tiny C and MinIPS. In the second environment, they can test TinyIP stack and Tiny C independently of MinIPS real hardware.

Finally in the third environment they integrate results developed by team members into a complete Tiny J System. The integration follows the steps: 1)N, C, and P subgroups demonstrate respectively that TinyIP is running on Linux, that Tiny C generates code executable on MinIPS emulator, and that the MinIPS processor works by executing LED blinking program. 2)They synthesize MinIPS and load the TinyIP compiled by Tiny C. 3)They execute an echo server on Tiny J System and confirm that it can communicate with an echo client on the Linux box.

Screens displaying UDP/IP communications between MinIPS and Linux are shown in Figure 5. A window of the MinIPS console displays loading and execution of Tiny J object codes for the echo server. The Linux screen displays execution of the echo client on a window while monitoring the communications by `etherpeep` on the other window. The echo server in this figure turns upper/lower cases of received alphabets in transmission for ease of verification.

4 Results and Discussions

In the P sub-laboratory, according to steps reached by individual students, we gave hints at early stages to adjust their paces. In the first and second periods, eight of ten and ten of ten assignees completed the P sub-laboratory, respectively. Even students who could not complete the laboratory expressed in their reports a strong sense of accomplishments.

We accepted seventeen reports from the C assignees. All of them completed all the requisite assignments, and twelve students tried the optional enhancements. Examples of the enhancements made by students are: adding pre-increment/pre-decrement operators; extending the lexical analyzer so as to accept various sorts of integer literals.

All of the twenty N assignees succeeded in implementing TinyIP stacks for Linux. They enhanced their stacks for Linux in various ways. Table 1 shows the enhancements and the numbers of students who tried and finished the enhancements. Some of them made multiple enhancements.

Tools for checking individual components, for testing effects caused by interactions between components, and for verifying Tiny J integration as a system are required. Although some of them have been provided for students as mentioned in the previous section, the testing environment is still poor as a whole. Particularly in verifying the integration, it is not easy for students to create programs for checking expected behaviors, because situations covering exhaustive failures are difficult to produce. If we provided better test tools, they could verify their implementations more easily.

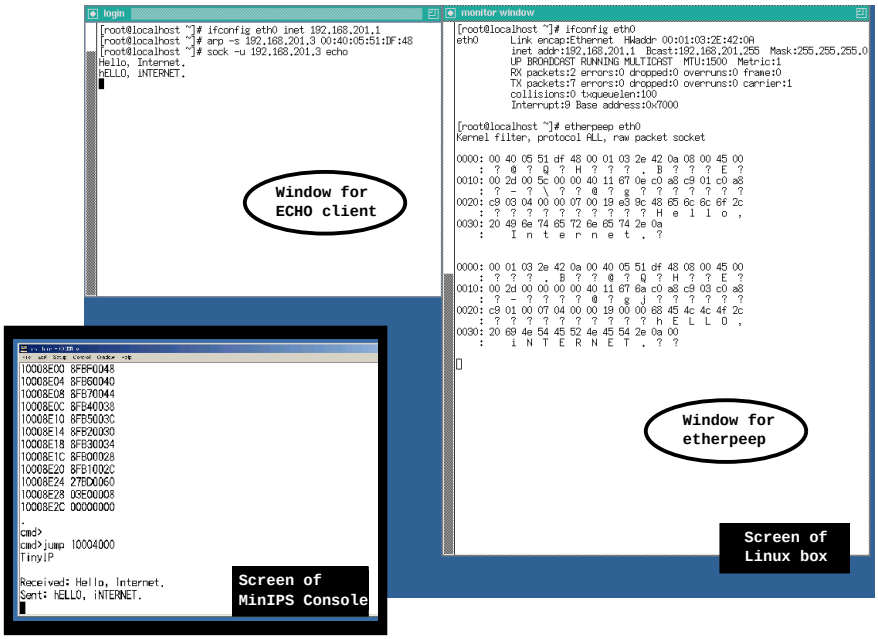

Figure 5: Screens displaying UDP/IP communications between MinIPS and Linux.

Table 1: Enhancements made by students.

Features	# of students who tried (finished)
Optimizing memory usage	8(8)
IP fragment transmission	4(2)
IP fragment reception	3(2)
ARP request	7(6)
ARP reply	5(5)
ICMP echo reply	4(3)
ICMP port unreachable	2(2)

We asked students to fill out a questionary provided by us to evaluate the laboratory from a student's point of view. A summary of the answers collected from P assignees after the first period of the laboratory is shown in Figure 6. The results show that it took long time for students to complete the laboratory compared to regular 36 hours: for example, 10 to 15 extra hours needed for 60% of the P assignees. However, we can see that 90% of the students understood the laboratory and 100% of them enjoyed it. Almost the same responses have been obtained about levels of understanding and attractiveness from other sub-laboratory assignees. In spite of the large and tough laboratory, three teams out of five were successful in the integration of C, N, and P components. We observed many scenes where shouts for joy arose from around upon succeeding in the CNP integration. This is considered to be another proof that CNP laboratory is successful in giving the students a sense of accomplishment.

Some students in the first period, however, complained that they did not understand well what other subgroups were doing. This suggests a need of some devices for students to be more aware of other subgroups.

From the suggestion as well as our experiences on the first period of the CNP laboratory, we introduced *progress check sheet*, a sheet to record the progress of each member in a team. Three columns of the sheet list steps of P, C, and N sub-laboratories in time sequence. A row of the sheet shows the current progress of the members in a team. They put the date when they have finished a step. All the students in the same team share a sheet and can see what other subgroups are currently working on. We expected that by sharing the sheet they will feel a sense of cooperations, stimulating more active communications. We also intended by the sheet that students obtain a cross-cutting view of the Tiny J system.

Students in the second period answered that the progress check sheet helped them be aware of other subgroups and understand what they are doing. They also answered that they were able to collaborate smoothly with other subgroups thanks to the check sheet. Probably due to the boosted collaboration, all of the five teams were successful in the CNP integration in the second period, which is one of the distinguished improvements

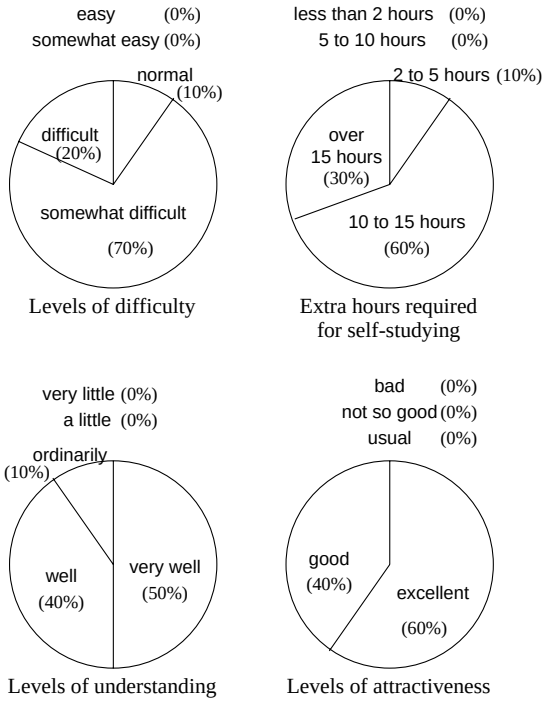

Figure 6: Summary of the questionary.

from the first period.

5 Conclusions

An integrated laboratory dealing with computer networks, compiler design, and computer organization has been developed. In the laboratory, students understood the assigned components and their interfaces with other components. After discussing and adjusting their specifications, they designed and implemented these components, and integrated them cooperatively into a system. The goals of the integrated laboratory have been proven to be fulfilled from the response of students who performed the laboratory, approving our intention of the laboratory and verifying its effectiveness.

Several improvements have been made to encourage students' cooperations. But we are aware of a lack of testing methodology. Two approaches are considered: 1) giving them a set of test suites; 2) teaching the way of testing. Both approaches are to be brought into the laboratory, which belong to future works.

Acknowledgements

The authors are grateful to Prof. Tan Watanabe at UEC, the original TinyC inventor, who has been supporting our work with many respects. Mr. Masato Naraoka at UEC contributed to maintaining laboratory equipments. We also thank to members of Abe lab. for developing many peripherals for Tiny J. Special thanks are due to students who challenged the laboratory with great interests and contributed to many improvements.

References

[1] R. B. Brown, R. J. Lomax, G. Carichner, and A. J. Drake. Microprocessor design project in an introductory VLSI course. *IEEE Trans. of Education*, 43(3):353–361, 2000.

[2] D. Kassabian and A. Albicki. A protocol test system for the study of sliding window protocols on networked UNIX computers. *IEEE Trans. Education*, 38(4):328–334, 1995.

[3] T. Katsu, D. Oosuga, M. Tsuruta, and K. Abe. Design and implementation of a 32 bit RISC processor MinIPS. *Bull. of the Univ. of Electro-Comm.*, 10(2):71–78, 1997.

[4] K. Morita and K. Abe. Implementation of UDP/IP protocol stack on FPGA and its performance evaluation. In *Proc. IPSJ General Conf. Special5*, pages 157–158.

[5] The Joint Task Force on Computing Curricula IEEE-CS and ACM. *Computing Curricula - Final Draft*. http://www.computer.org/education/cc2001/final/index.htm, December 2001.

[6] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann Pub., 1998.

[7] J. Postel. Echo protocol. RFC 862, May 1983.

[8] T. Watanabe. *Composing a compiler*. Asakura Pub., 1998.