Using Custom Hardware and Simulation to Support Computer Systems Teaching

Murray Pearson, Dean Armstrong and Tony McGregor
Department of Computer Science
University of Waikato
Hamilton
New Zealand
{mpearson,daa1,tonym}@cs.waikato.nz

Abstract

Teaching computer systems, including computer architecture, assembly language programming and operating systems implementation, is a challenging occupation. At the University of Waikato we require all computer science and information systems students study this material at second year. The challenges of teaching difficult material to a wide range of students have driven us to find ways of making the material more accessible. The corner-stone of our strategy for delivering this material is the design and implementation of a custom CPU that meets the needs of teaching. In addition to the custom CPU we have developed several simulators that allow specific topics to be studied in detail.

This paper describes our motivation for devloping a custom CPU and supporting tools. We present our CPU and the teaching board and describe the implementation of the CPU in an FPGA. The simulators that that have been developed to support the teaching of the course are then described.

The paper concludes with a description of the current status of the project.

1 Introduction

Teaching computer systems is a challenging but vital part of the computer science curriculum. In 1997 the Department of Computer Science at the University of Waikato decided that computer systems was important to all computer science and information science students and made its computer systems course compulsory for all second year students. Like most computer systems courses Waikato's uses assembly language programming as a vehicle to understanding the inter-relationships and interactions between the different components of a computer system. The brief of the course is quite different to an introductory computer architecture course, even though it contains many of the same components. The difference lies in the audience and motivation. Our course is intended to be useful to all computer professionals, not just those who specialise in computer architecture. Our use of assembly language programming is an example of the impact of this difference. Very few of the students will continue to program in assembly language after the course, however, we believe that it is important that they have an understanding of computer operation at this level of abstraction. While we want to teach a coherent and realistic architecture we have no fundamental interest in details such as delay slots, addressing modes and word alignments. These are important topics for a specialist, but do not significantly add to the understanding of the operation of a computer system as a whole, which is the goal of our course. Assembly language *is* essential to this goal but many students find assembly language programming difficult and this detracts from the main thrust of the course, which is not to teach assembly language per say.

We wish to focus on the role of the machine and the interactions between the hardware and software components including compilers, libraries and the operating system, rather than spending a lot of time describing a partticular manufactures performance oriented features. This has led us to develop our own instruction set architecture called WRAMP. As described later the course has a practical component; practical exercises reinforce the content of the lecture material. To support the practical component of the course using the WRAMP instruction set has required the development of a platform to allow students to assemble and execute WRAMP programs. The two choices considered were the development a WRAMP simulator or a custom hardware platform.

Using a simulator is easier and cheaper however we believe that the lack of real hardware distorts the learning

environment by adding an extra, unnecessary, abstraction when many students are struggling to come to grips with the essential content of the course. A simulator it is itself a program running on a computer. This makes it difficult for students to readily identify the target system and they tend to confuse the role of components of the system. When this happens there is a risk that students will focus on the most obvious difference between practical work in this area and others: the programming language. When real hardware is used, the real focus is more likely to be on the target system.

For this reason, we believe that students should have the benefit of real hardware when they first learning assembly language programming. Until recently this would have excluded a custom CPU design, however it has been made possible by advances in reconfigurable logic. We have used FPGA technology to develop a single board computer (called REX) with with our own custom designed CPU and IO devices.

Once the students have developed a clear mental model of the components of a computer system, simulation can be used to enhance their understanding of the more complex topics in the course. To this end we have developed simulators for use in the course, two of which are presented here. The first of these, called RTLsim, is used to simulate a simple non-pipelined MIPS processor to demonstrate how instructions can be fetched from memory and executed. The second of the simulators is a multi-tasking simulator that introduces students to the ideas behind task swapping in a multitasking kernel.

The next section gives an outline of our computer systems course. Section 3 then describes, in more detail, the motivation for developing a processor and board to support the teaching this course. Sections 4 and 5 describe the design of the CPU and board. Section 6 then describe the simulators that that are used in the course followed by Section 7 which briefly describes the exercises carried out by students on the course.

A brief description is then given of how we intend to use the board in the third and fourth year computer architecture courses.

2 Course Outline

When the Department decided to make the second year computer systems course compulsory, its curriculum committee established a set of key topics that should be covered by the course. These included: data representation, machine architecture (including assembly language programming), memory and IO, operating systems and data communications.

Figure 1 shows the order of the topics that make up the course and the relative levels of abstraction used to describe them. The main content of the course can be broken into two parts. The first part illustrates what happens to a high level program when it is compiled and executed on a computer system. This serves two purposes. First, it demonstrates some of the major issues which determine the performance of a computer system. Second, it shows the likely consequences of writing a particular construct in a high level programming language in terms of speed and size of the code generated.

The aim of the second part of the course is to produce an understanding of operating system principles and components, their role in supporting the user, and in the execution of programs written in high level languages such as C (the starting point of the course). The focus is on achieving an understanding with the operating system and the implications of hardware and software choices, rather than an ability to write a new one.

There is a strong theme of interactions and relationships between the components of a computer system. To support this we base the whole course around a single processor architecture so that the students could more easily see the way the individual components of the system contribute to the complete computer system.

3 Background

Because the goal of the course is to explain the role and interaction of the components of a computer system, not to teach assembly language programming for its own sake, there are two main requirements for a architecture:

1. a simple, easy to learn instruction set
2. an architecture that can easily demonstrate the relationship between high and low level languages, and user and kernel space.

These goals are at odds with most modern CPU architectures which have been optimised to maximise performance and not simplicity. To help achieve high performance modern CPUs contain many performance oriented techniques including the use of reorder buffers, register renaming and reservation stations[6]. Because of the complexity of these architectures it would not be possible to fully describe the structure and functionality of one of them in an introductory course.

While most architectures are optimised for performance some (such as the 8-bit processors -e.g. the Motorola HC11) are designed to be very cheap and simple. However, this very simplicity often raises the complexity required to program the CPU. For example, performing 16-bit indexed address access on an 8-bit processor that only has an 8-bit ALU requires a series of instructions to support the 16 bit addition rather than the single instruction available on larger word sized machine. Because of the way CPUs developed through the late ’80s and early ’90s, processors with a large enough word size to make

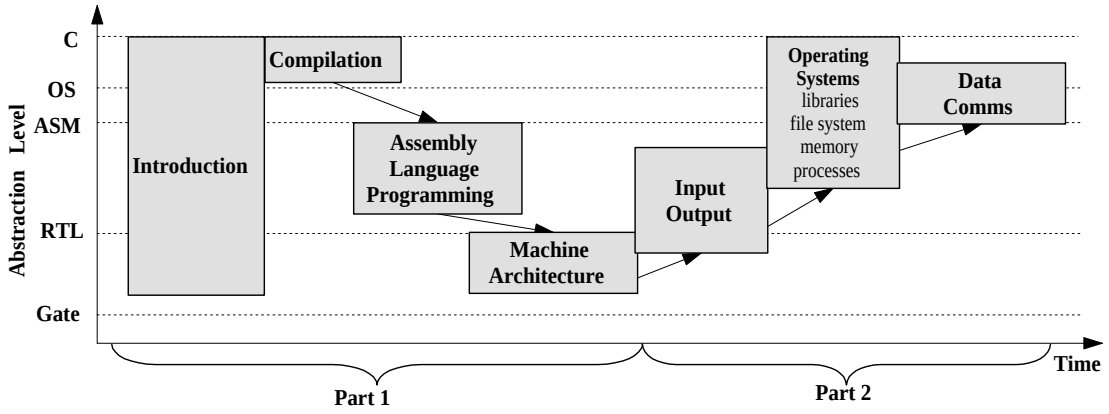


Figure 1. Topics Covered in the Course

those aspects of programming easy have other complexities, such as many addressing modes, that are not available across all instructions or complex interrupt processing. Although many modern CPUs are simpler, because of the influence of the RISC philosophy, they have other disadvantages, including branch and load delays as described below.

In the past, we have used the MIPS R3000 family as a compromise between the needs of our course and available CPU designs [4]. The MIPS CPUs have a relatively simple programmer's abstraction. The teaching process is also supported by a number of very popular text books including those written by Hennessey and Patterson [3] [2] and Goodman and Millar [1]. For this reason our computer systems course has been based around this processor for the last six years. While we have found this processor reasonably well suited to our needs, we have identified a number of aspects of the architecture that many students find difficult to understand and which are not central to our teaching goals. These include:

- the presence of *load delay slots* which mean that the instruction directly after a load instruction cannot use the result of the load as it isn't available yet.
- the presence of *branch delay slots* which mean that the instruction directly after a branch instruction is always executed regardless of whether the branch is taken or not.
- the use of an *intelligent assembler* which is capable of reordering instructions and breaking some assembler instructions in two so that they can all be encoded using a single 32-bit word.
- the requirement that all *memory accesses to word values are word aligned*.
- the *parameter passing conventions* that are designed to minimise the number of stack manipulations in a MIPS program.

While we do not believe that the complexities described above are insurmountable, they do detract from the goal of the course, that is to give a complete coverage of the computer systems area at an introductory level without being distracted by the complexities associated with describing a particular manufacturers quirks. This is in keeping with the introductory level and broad audience that this course is intended for. Other courses at the University are intended for students who will specialise in computer architecture, and these do cover commercial architectures, including exposure to many of these issues.

We have been unable to find a suitable commercial CPU architecture to support the teaching of our computer systems course so we developed our own.

Before discussing the architecture of the CPU we have designed we consider the question of whether to use a real CPU or a simulator. Most courses that teach computer architecture or assembly language teaching make use of CPU simulators. Using a simulated system offers two main advantages. Firstly, it is possible to develop a simulator for any CPU. This allows a CPU that is tailored to the goals of the course to be used rather than being limited to those that are available commercially. The second advantage of using a simulator is that simulators normally offer better debugging facilities and visualisations of a program. These can be used to help reinforce important concepts.

As noted in Section 1, using a simulator also introduces difficulties for students. It is more likely that students will confuse the boundries between the host system and the simulated system. Our experience suggests there is a tendency for students to focus on the programming language when a course introduces a new language, rather than conceptual material in the course. The use of real hardware makes the distinctions between the target

system and the development tools concrete. The work presented in this paper largely removes the disadvantages of using a real CPU and enables both a simpler working model and a CPU designed to meet the needs of teaching. This includes good debugging facilites such as the ability to single step and observe register and memory values as the system executes.

4 Processor Design

In designing the processor a great deal of care has been taken to keep the design as simple and regular as possible while still being able support the complete range of practical experiences we wish the students to be exposed to. These experiences start with the writing of simple assembly language programs and build up to the development of a very simple multi-tasking kernel.

The resulting CPU design uses a 32 bit word, and is based around a register-register load-store architecture, very similar to the MIPS and DLX [5] processors. Most computational instructions have a three operand format, where the target and first source are general purpose registers, and the second source is either a register or an immediate value. Regularity of the instruction set was a key factor in maintaining the simplicity. Immediate flavours of all computational instructions are provided, as well as unsigned versions of all arithmetic instructions.

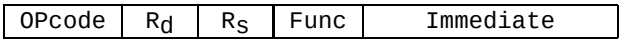
Care was taken to keep the correspondence between assembly language instructions and actual machine instructions as a one-to-one relationship. To this end a major feature of this CPU is the reduction of the address width to 20 bits, and the number of registers to 16. This allows an address, along with two register identifiers and an opcode to fit into a single instruction word, removing the need for assembler translation when a program label is referenced.

The other main differences from MIPS and DLX are the removal of the branch and load delay slots, and the fact that the CPU is 32 bit word addressable rather than byte addressable. Making the machine word addressable only, greatly simplifies the operation of the CPU, and allows us to present students with an easily understandable model of it. Another advantage of a word addressable machine is that it removes the possibility of word access alignment problems which new students frequently encounter on a byte addressable machine.

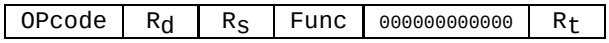
The CPU only supports three instruction formats as shown in Figure 2. It can also be seen from this figure that the instructions have been encoded to allow for easy manual disassembly from a hexadecimal number, with all fields aligned on 4 bit boundaries.

While the CPU has been made as simple as possible for the tasks we require it does include external and software interrupts and has supervisor and user modes with protection. These mechanisms are accessed through a

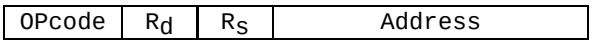
I-Type instruction



R-Type instruction



J-Type instruction



OPCode	4 bit operation code
R_d	4 bit destination register specifier
R_s	4 bit source register specifier
R_t	4 bit source register specifier
Func	4 bit function specifier
Immediate	16 bit immediate field
Address	20 bit address field

Figure 2. Instruction encoding formats

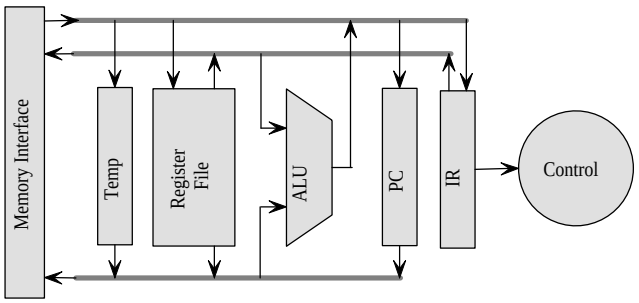


Figure 3. Processor Block Diagram

special register file, similar to the MIPS' coprocessor 0. This means that these concepts need not be discussed for students to begin programming in assembler, and when desired, they can be introduced by describing the special register file, and the two instructions needed to access its contents.

The data-path of the processor is based around a three-bus structure (as shown in Figure 3) and instructions take multiple clock cycles to execute. As can be seen from Figure 3 the CPU's data-path is very simple making it possible to completely explain the operation of the data-path to second year students. In particular it is possible to explain in detail how machine code instructions stored in memory can be fetched, decoded, and executed on the data-path.

The CPU has been represented in VHDL so that it can be targeted to a reconfigurable logic device. The CPU design when synthesised consumes a large portion of a 200 thousand gate Xilinx Spartan II FPGA device.

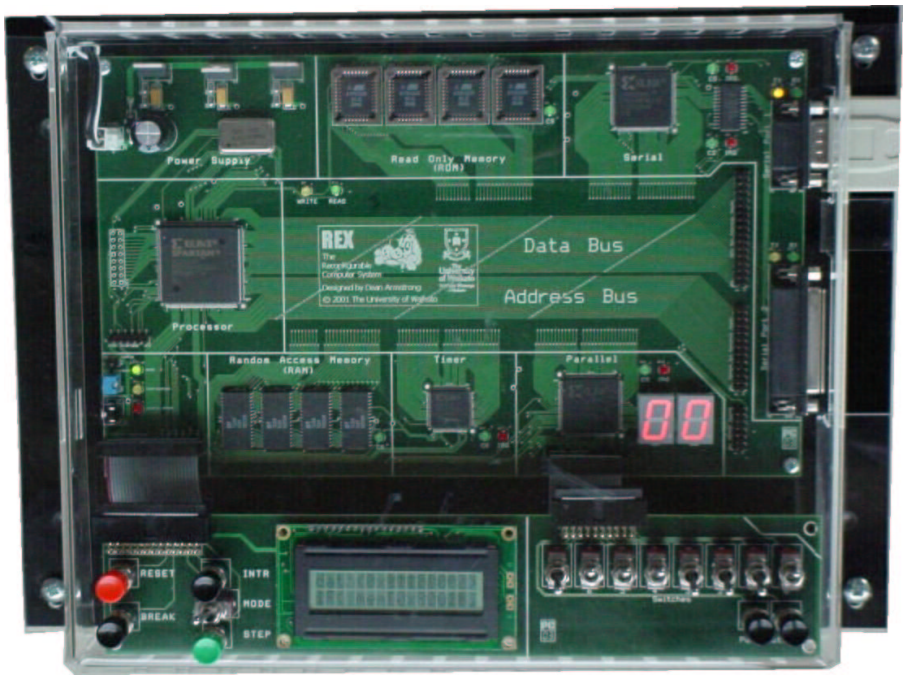


Figure 4. The Teaching Kit

5 Board Design

Figure 4 shows the REX board designed to support the CPU described in the previous section. As can be seen from the picture we have been careful to layout the board so that the main components that make up a computer system can be clearly identified. The main data-paths that connect these components are also visible on the board.

Reconfigurable logic is used wherever possible on the board to allow it to be as flexible as possible. In addition to making the design of our own CPU and IO devices possible, this allows the architecture of these components that students are presented with to be fine tuned as the course develops. As explained later, it also allows the board to be used for multiple teaching functions, including FPGA and CPU design.

While it would have been possible to place most or all of the reconfigurable designs into a single chip the decision was made to use a separate chip for each IO device and the CPU, making it possible for the students to physically identify each of these devices on the board. The choice to use multiple RAM and ROM chips to provide the 32 bits of data rather than employing multiple accesses to a single chip was also made with the intention of clarifying the operation for the students. Effort was made, however to keep the number of non-essential support components to a minimum.

The boards are intended to be connected to a workstation where students can write and assemble programs, which can then be loaded and run on the board. Because we wanted to build a laboratory for a large class it was important to make reconfiguration easy. In particular we designed the board to support remote reconfiguration of all programmable devices and the stored bootstrap program code. Scripts have been developed that enable all of the REX boards in a laboratory environment to be completely reconfigured from a single command. Cost has also been kept to a reasonable level.

Although there are a number of features that support teaching, one that had a large impact on both the board and CPU design is support for cycle-by-cycle stepping of the processor with an LCD display to indicate bus contents, and LEDs to show device selection and exceptions. We believe this feature will be a major asset for students struggling with the many new abstractions and concepts presented by the course.

6 Simulators

In the course we use a number of simulators to reinforce some of the more complex conceptual material. The first simulator (RTLsim) has been developed to reinforce the ideas associated with the execution of machine code instructions on a data-path. The second simulator is a multi-tasking simulator that introduces students to the ideas behind task swapping in a multitasking kernel.

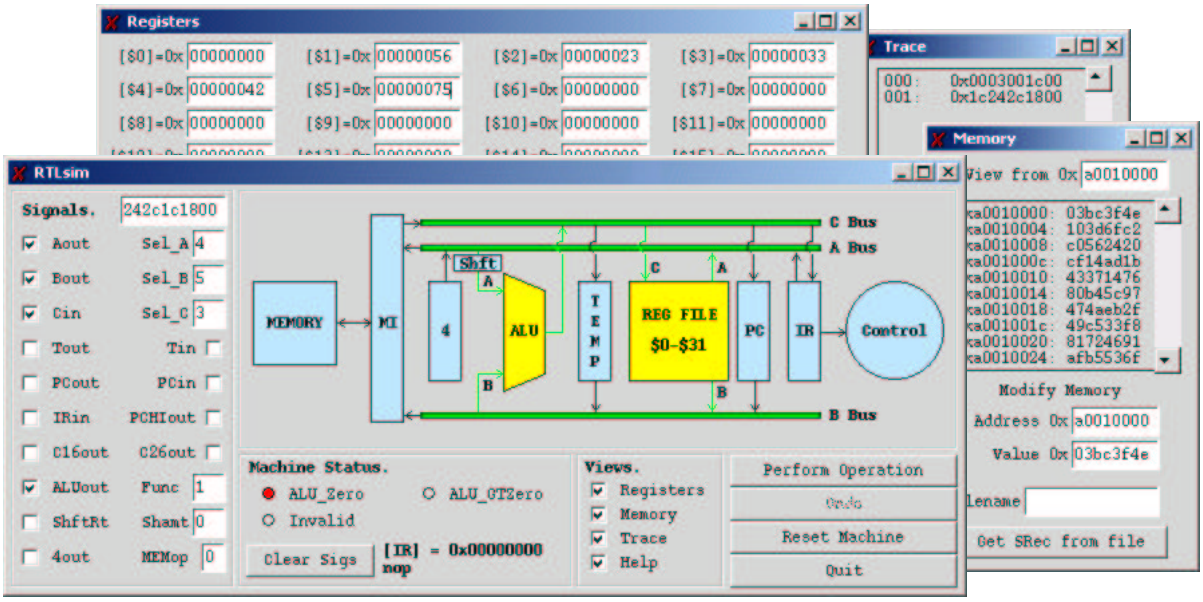


Figure 5. Screendump showing RTLsim in operation

6.1 RTLsim

In the first part of the course the students learn the relationships between a program written in a high level language such as "C" and its representation in assembler and machine code. Following on from this we show the students how a machine code instructions can be executed on a simple processor data-path. In previous years a simulator called RTLsim which simulates the data-path of a simple non-pipelined MIPS like processor has been used to support the teaching of this component of the course. Currently we are in the process of developing a WRAMP version of the simulator. The rest of this section describes the MIPS version of RTLsim.

RTLsim is written in C for a UNIX system running X-windows. When the simulator is run the student (user) acts as the control unit for the data path by selecting the control signals that will be active in each control step.

Figure 5 shows the main window for the simulator that comprises of two main components, a visual representation of the data-path and a control signals window. The data-path is made up of a 32-register register file, ALU, Memory interface and a number of other registers to store values such as the program counter and the current instruction being executed. Three internal buses are used to connect to connect these components together. This combination of components and buses is sufficient to fetch and execute most of the instructions in the MIPS R3000 instruction set. The control signals section of at the left hand end the main window is used by the student to set the values of control signals that are going to be active in the current control step. For example consider the execution of the MIPS instruction `add $3, $4, $5` that adds the contents of register 4 to register 5 and store the result into register 3. Assuming the instruction has been fetched into the instruction register during earlier control steps then the settings shown in the controls signals window of 5 would cause the necessary actions to occur to execute this instruction. As the student sets the control signals for a control step they are given visual feedback on the data-path of what will occur when the control step is executed. For example if the PCout signal is selected the colours of the PC register and the B Bus would change to show that the PC register is going to output a value onto the Bbus. If two components try to output to the same bus at the same time the bus would turn red to indicate an illegal operation.

From the main window, other windows may be opened that show the contents of memory and the register file. In the case of the memory window it is possibly to preload memory image from an file in s-record format before starting a simulation. This is the same file format used to upload programs to the MIPS board. This enables the students to upload and execute the same program on both a MIPS board and RTLsim, allowing the executions to be compared.

The simulator can also record a trace of the operations that are performed in each control step. This trace can be used by the student to playback the operations in the simulator or used as input to an automated marking system.

Before RTLsim was introduced to the course the students where given a paper-based exercise where they had

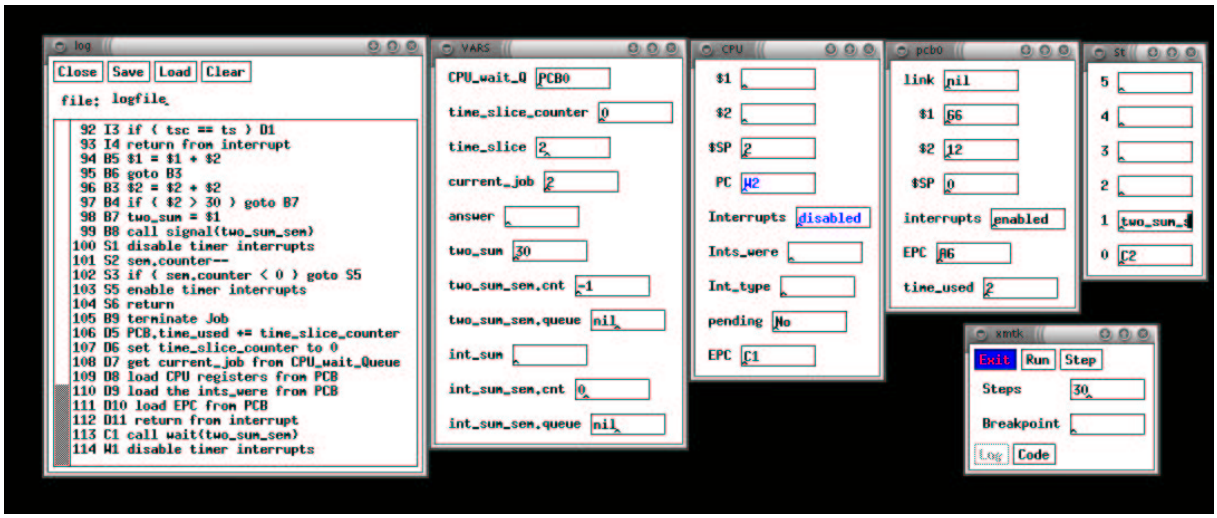


Figure 6. Multi-tasking simulator

to define the sequence of control steps necessary to execute a set of MIPS instructions they were given. If the students had not grasped the main concepts they completed the entire exercise incorrectly and were not given any feedback until the assignments were marked and returned to them several weeks later. However with the introduction of RTLsim the students are given immediate feedback at several levels. Firstly as the students set the control signals they are given visual feedback on the data-path. Once they believe they have the necessary signals to execute the control step they can try it and observe the outcome in the registers and memory. If the outcome is incorrect the simulator provides undo operations so they can try again. Lastly, an automated marking system is used. If the exercise is not completed correctly the marking system generates a set of comments that tells the students where they went wrong so they can try again.

6.2 The Multi-tasking Simulator

One of the assignments undertaken by students in the second year course using the boards is the development of a very simple multi-tasking kernel. The kernel does not include memory management, task creation or termination but it does share the CPU between three tasks, including the saving and restoring of state and changing of stacks between tasks. The tasks are designed to use different parts of the hardware. One reads the switches and writes the value read to the seven segment display, another reads characters from the secondary terminal and writes the uppercase values to the terminal. The third task displays the time on the primary serial port. Students have already written these tasks in a single task environment, in earlier assignments.

Although the multi-tasking kernel does not require very many lines of code, there are conceptual and coding barriers to its implementation. We address these issues in classes but have found it useful to re-enforce the ideas using a multi-tasking simulator, before students attempt their own implementation. The simulator is written in C for X-windows and creates a number of windows. An example of the windows is shown in figure 6. Each task has two windows associated with it, the first is the stack and the second is the saved state of the task (its process descriptor). An example for one task is shown in the right most two windows in figure 6. When the students use the simulator there are three tasks; two have been omitted here to save space. The `link` field is used to form a linked list of tasks waiting for the CPU or waiting on a semaphore for an event.

Moving to the left in figure 6 the middle window shows the CPU registers. The simulated machine has only two general purpose registers, a stack pointer, a program counter, a status register and a saved program counter which shows the value of the program counter as it was at the last interrupt. The status register is divided into the interrupt status (masked or enabled), the interrupt status before the last interrupt (software interrupts are taken even if interrupts are masked), the type of interrupt (e.g. timer interrupt) and whether there is an interrupt pending (when interrupts are disabled).

The window second to the left shows the values of some shared memory variables. These include the head of the CPU wait queue, the number of interrupts left in this time slice, the job currently using the CPU the output of two of the tasks (`answer` and `two_sum`), and semaphores that hold task 3 until these two tasks are completed.

The left hand window, which gives a trace of the in-

structions that have been executed. The simulator executes pseudo-code which has been designed to be close enough to WRAMP assembly code that it is easy to imagine the assembly code that matches a pseudo-code instruction, but without some of the confusing detail of assembly code. The number at the left of the log window indicates the sequence number of the instructions that have been executed. The letter/number code next to the sequence number is the address of the instruction. The letter in the address indicates what part of the code (A = task A, F = first level interrupt handler, W = wait, S = signal, etc.) the instruction belongs to.

As each step of the simulation is executed the values that change are hilighted in red in the appropriate window. Students are able to change the values at any time to alter the course of the simulation. The assignment encourages them to do this, including altering the time-slice length.

Readers interested in obtaining the simulator should contact the author at `tonym@cs.waikato.ac.nz`.

7 Assignments

The assignments that make up the practical component of the course are shown in Table 7. Of particular note is the implementation of a multitasking kernel by the students. Given that most students are not computer technology students and that most successfully complete this exercise we belief this is a major indication of the success of the course.

No.	Name
1	Introduction to Unix
2	Data Representation
3	Introduction to REX
4	C and WRAMP assembly
5	RTL Design Exercise
6	Parallel and Serial IO
7	Interrupts
8	Multitasking Kernel Simulator
9	Multitasking Kernel Coding
10	Error Detection

Table 1. Assignments

8 Use of the Board by 3rd and 4th year Students

We are currently teaching students in a third year computer architecture course about design using VHDL. By the end of the course the students will be able to design the main components (ALU, registers, finite state machines, etc) that make up a CPU. In future years we plan to use the REX boards to support the teaching of this course.

In our fourth year computer architecture course, students design and implement their own CPU. Last year the students used a prototype version of the REX board to implement their CPUs. With the introduction of the new board and the experience gained using the board in the second and third year courses, we hope to extend the complexity of the project undertaken in this course.

9 Conclusions

This paper described the range of hardware and software tools that have been developed to support the teaching of the introductory Computer Systems course at the University of Waikato.

There is much merit in the design of custom CPU and IO devices for teaching purposes. Current reconfigurable hardware devices have made it possible to build a single board computer, with a custom CPU and IO devices, to support the teaching of computer systems courses. Using this approach we have removed some of the 'sharp edges' of assembly language programming, like branch delay slots and complex CPU status control, that add complexity to introductory teaching but do not add significant value.

An additional advantage is that the board will provide a consistent teaching platform across a range of courses. We expect that this will considerably enhance the students learning experience.

We have just installed 25 REX boards in one of the Departments Computer Labs. Supporting tools such as a monitor program for the board, a C compiler, an assembler and linker are now largely complete. Over the past couple of weeks students have been using the REX boards to complete their assignments. All of the feedback we have had from the students todate has been very positive and encouraging.

References

[1] J. Goodman and K. Millar. *A Programmer's View of Computer Architecture with Assembly Language examples from the MIPS RISC Architecture*. Oxford Press, 1992.

[2] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan-Kaufman, 1995.

[3] D. A. Patterson and J. Hennessy. *Computer Organisation and Design: The Hardware/Software interface*. Morgan-Kaufman, 1994.

[4] M. Pearson, A. McGregor, and G. Holmes. Teaching computer systems to majors: A MIPS based approach. *IEEE Computer Society Computer Architecture Technical Committee News Letter*, pages 22–24, Feb. 1999.

[5] P. M. Sailer and D. R. Kaeli. *The DLX Instruction Set Architecture Handbook*. Morgan-Kaufmann, 1996.

[6] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11, pages 25–33. 1967.