A perspective on the experiential learning of computer architecture

Ian McLoughlin
School of Computer Engineering,
Block N4, Nanyang Avenue,
Nanyang Technological University,
Singapore

Koji Nakano
Department of Information Engineering,
Hiroshima University,
Kagamiyama 1-4-1, Higashi-Hiroshima,
739-8527, Japan

***Abstract*— In many universities, computer architecture is taught using traditional textbook-based methods. However, it is not easy for students to understand how computers work through lecture style courses alone. This paper describes an experiential approach for teaching masters-level advanced computer architecture with the assistance of hands-on laboratory sessions, leading students to implement performance-enhancing additions to a simple stack-based CPU called TinyCPU [1] originally designed by Nakano of Hiroshima University. From the teaching experience in Nanyang Technological University, analysed in this paper, students manage to quickly grasp the concepts of CPU operation, rapidly investigate the effects of adjusting CPU structure on program execution, and learn the skills-set necessary to enable them to built and improve custom processors later in their careers.**

I. Introduction

Computer architecture is a common subject taught in undergraduate computer engineering, computer science and electronic and electrical engineering courses. Computer architecture normally follows a structure similar to that set out in the CAO (Computer Architecture and Organization) section of the IEEE Computer Society/ACM body of knowledge (BOK) [2].

In many universities, computer architecture is a highler level class that builds upon foundations taught in a computer organisation course. Advanced computer architecture, in this context, is a class that in turn builds upon the foundations of computer architecture. It assumes that students already posses the knowledge and skills to program, design with, and use computer components in both desktop and embedded contexts, and have the tranferrable skils to begin understanding performance-limiting features in computing.

Computer organisation and computer architecture are courses that are backed up by a large number of uncergraduate textbooks (such as [3]), and are generally taught in classroom style, probably augmented through the addition of laboratory sessions, tutoril study and reading assignments.

With this context, the current paper describes an approach that follows on from the foundations of computer organisation and computer architecture, to firstly consolidate the earlier knowledge, and secondly to extend this towards a more system-level apprecation on the one hand, and also that ability to delve into the detail necessary to support that understanding. We call this advanced computer architecture, and deliver it as part of a Masters of science programme in embedded systems [4].

In short, this course can be difficult to teach in a classroom-style format: students enter the course with a diversity of educational levels and prior knowledge, and yet must master both the high level aspectds needed to assess the performance limitations of computer systems in general, but also the basic skills needed to design better processors to alleviate these limitations.

For this reason, in 2009 the first author, in Nanyang Technological University (NTU), Singapore, embarked upon developing a more experiential approach - using hands-on laboratory sessions in which students designed their own CPUs, assessed performance, and implemented a variety of performance improvements. Since it was not the intention to develop a completely new processor, he decided early on to standardise upon an established teaching processor that was freely available, simple, easily understandable and yet extensible. The TinyCPU, developed by Nakano of Hiroshima University, Japan [1], fit this description [5].

This paper will first introduce TinyCPU in Section II, then the educational aspects of using it in Section III. After two years of teaching, this paper offers a retrospective assessment of its effective in Section IV before concluding the paper in Section sec:conclusion.

II. TinyCPU

TinyCPU, as we have seen, was designed by Nakano as a teaching tool for his students in Hiroshima University. It comprises only about 400 lines of relatively simple Verilog, and has been compiled for hardware implementation on Altera and Xilinx FPGA, and explored with tools such as Quartus, ISE and ModelSim.

A block diagram of the CPU is shown in Figure 1, where it can be seen that TinyCPU consists of a stack feeding an ALU, connected to other elements such as input port, output port, memory, instruction register and program counter via a single bus arrangement (abus/dbus). Using a stack to feed operands into the ALU (either one or two), frees up the instruction set and simplifies the data transfer arrangements in much the same way that a load-store architecture does for a simple RISC CPU.

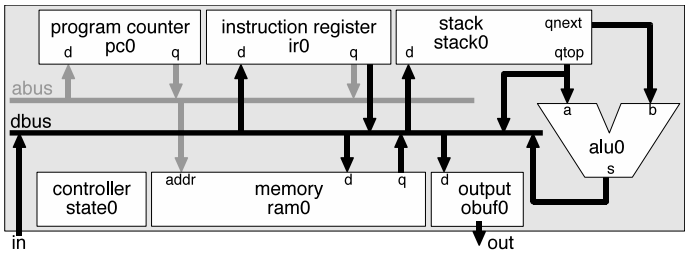

Fig. 1. Block diagram of TinyCPU, showing internal architecture, bus connectivity and basic functional blocks.

In its basic form, TinyCPU is a 16-bit machine, with a 16-bit instruction set, although memory addresses are limited to 12-bits by the space available within a single 16-bit instruction word to encode an absolute address value.

All memory locations, stack entries and internal data buses are 16-bits in size, and can perform one transfer per clock cycle. For executing instructions, there is thus one cycle required to fetch the instruction, one cycle to fetch any operands and either one or two cycles (depending upon the exact instruction being handled) to perform the execution stage. The CPU state diagram, illustrating this, is shown in Figure 2, where the state transition control signals - each readily identifiable in the Verilog source code - are also marked. The run signal is a one-time trigger to begin execution, and halt is likewise a one-time stop signal initiated by the `HALT` instruction.

TinyCPU supports 9 basic instructions plus a further 19 data processing instructions (which are ALU functions), can handle immediate constants and addresses, variables stored in memory, and is equipped with a 16-bit output register, and input port.

In terms of development flow, the various Verilog source files that make up TinyCPU (defs.v, alu.v, counter.v, ram.v, stack.v, state.v, tinycpu.v) are compiled using FPGA design tools, and then either simulated, or programmed into an FPGA. In both cases a top level file is required which instantiates TinyCPU (for example tinycpu-testbench.v or tinycpu-fpga-top.v respectively). As long as TinyCPU is reset, is fed with a clock, and then triggered with the external `run` signal, it will then operate.

Once TinyCPU starts, it will read instructions sequentially from memory, as defined in the Verilog source file ram.v,

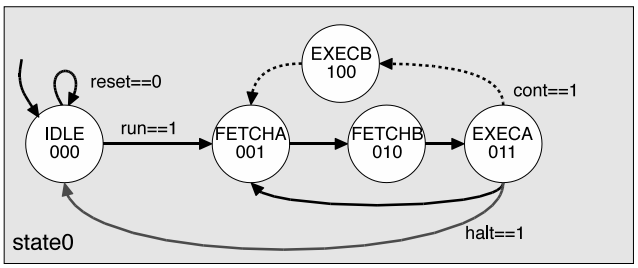

Fig. 2. TinyCPU state transition diagram, enumerating the five possible states, and identifying the trigger signals between them.

in other words, TinyCPUs program is encoded in ram.v, and compiled up as part of the Verilog code. For simulation and testing purposes, as used in the laboratory sessions described here, this approach is fine, but for use in something like a commercial FPGA-based product, it would probably benefit from an external memory interface. This would allow a new program to be loaded without having to totally recompile the entire CPU!

Nakano has also released an assembler for TinyCPU (which itself is just a few tens of lines of Perl code), and a compiler which supports a useful subset of the C programming language. The assembler implements two-pass assembly (thus allowing use of symbols) and outputs a type of object code. A separate conversion utility can then transform the object code into Verilog, suitable for pasting into memory file ram.v (and it even has the feature of line-by-line commenting the machine code to indicate the original assembler instructions corresponding to each machine code instruction).

For the purposes of the present paper, the assembler is taught, briefly, and then used as a tool. Students need to understand the operation of the assembler to fully appreciate the TinyCPU instruction set – and indeed will need to appreciate this fully when they are asked to extend the instruction set, and modify the assembler to support their extensions.

TinyCPU is a complete, working CPU. Every student in a laboratory session can, working individually, instantiate and run one or more TinyCPU processors either on real hardware, or simulated using a tool such as ModelSim. The typical 'view' of TinyCPU seen by students is shown in Figure 3, where a waveform display window shows the cycle-by-cycle contents of various wires and registers.

From this base, students first learn how TinyCPU works, appreciate the internal architecture, and then begin to discover its limitations and potential improvement areas. These are discussed further in the following section.

III. An MSc level advanced computer architecture education

Within Nanyang Technological University School of Computer Engineering, advanced computer architecture is taught as the ES6191 subject, a part of the MSc programme in Embedded Systems [4]. This comprises 39 hours of teaching/laboratory evening sessions, plus significant home study.

The first half of the curriculum covers the major elements presented in this paper, namely understanding, designing, constructing and testing a simple stack based CPU, implemented in Verilog: TinyCPU, plus an extensive theoretical treatment of pipelining, pipelining issues, ILP and IPC-targeted enhancement techniques (since these are not easily learnt through TinyCPU – see later). Hands-on laboratories extend for four evening sessions of three hours each (plus an additional session on cache design – also described briefly later). The first of these sessions introduces TinyCPU and takes students on a tour of its features and design. The second session introduces the assembler, written by Nakano et. al. in Perl [1]. Students are expected to master the assembly language during this session,

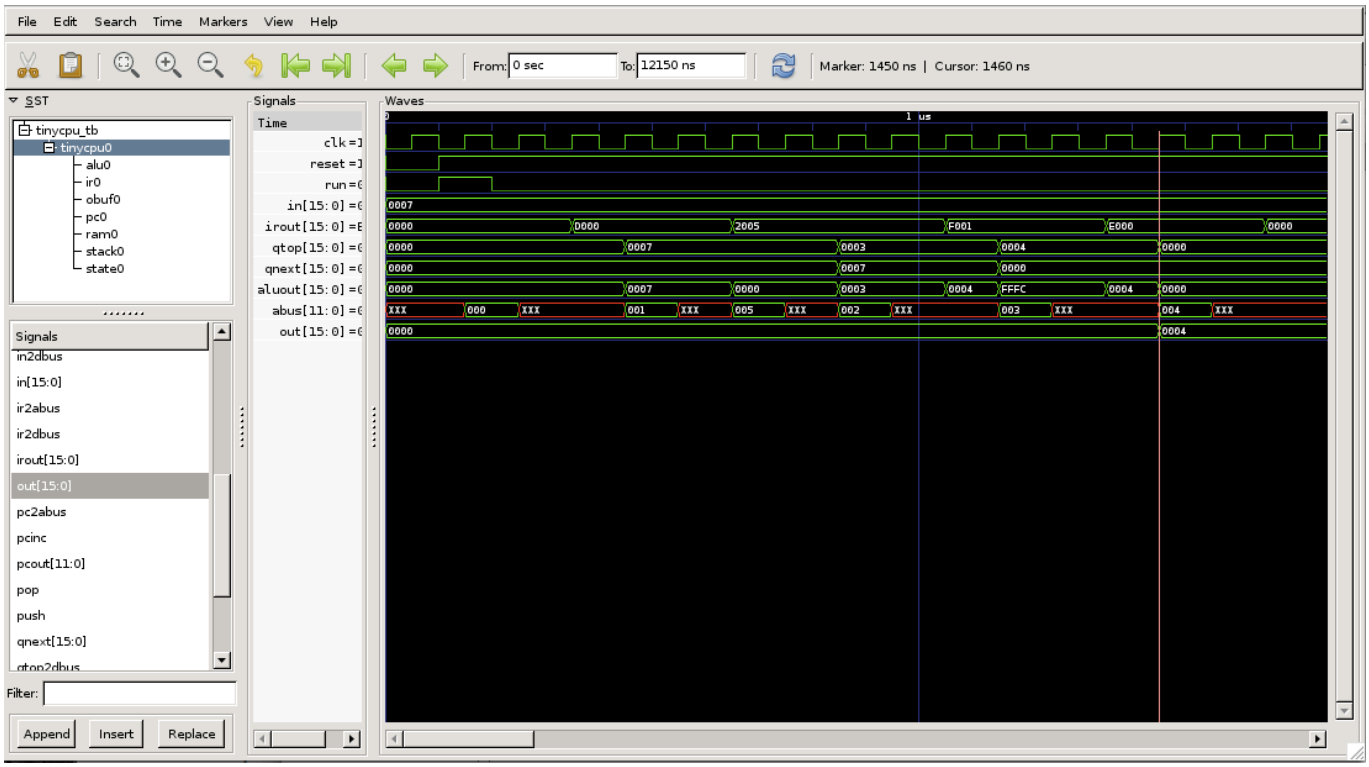

Fig. 3. Screen capture showing the waveform view of TinyCPU as it is executing a program from ram.v. The waveform viewer is GTKWAVE, and the wave data has been generated from Icarus Verilog. Both are freely available open source programs that run on Linux.

extend the assembler to catch various errors and implement an additional instruction. They then modify TinyCPU to include additional hardware for that instruction. During past years, this instruction has either been a selectable signed/unsigned ALU operation, or a saturating ADD (SADD).

Sessions one and two, which each last three hours, involve students working individually, using Modelsim and Perl tools. A Linux PC is preferred for this work, since it is faster, more reliable and far more convenient, however the laboratory has been designed so that it can also work with the windows OS. Since we encourage students to begin the laboratory work at home, this is especially useful for less capable students who are still using windows on their own computers.

Both initial sessions are assessed identically: through hand-written (and hand drawn) answers to a sequence of questions inserted into the laboratory manual, which is provided electronically to each student. Students who hand in the work before the end of the sessions receive a bonus. No bonus is given for work delivered up to one week late, but thereafter a penalty is imposed which increases linearly with degree of lateness.

The third lab session originally required students to implement a multiply-accumulate instruction, and hardware, either through incorporating additional circuitry, through re-use of existing circuitry or even through a microcode-type approach. However from 2010 onwards, the session has been discontinued, and instead combined with session four.

In session four, students are given instructions as shown in Table I. They are asked to extend and expand TinyCPU. This is where in-depth analysis starts, clever ideas and insights are rewarded, and a relatively divergent thought process is encouraged.

Students have two weeks to complete the major changes to TinyCPU, including laboratory sessions three and four, and again can choose to work at home or access the laboratory at other times. For this, they are to work as pairs, self selected, but hand in one report between them. Reports are limited to a maximum length of six pages, plus up to six pages of waveforms and listings: with such tight length guidelines, students need to think carefully about exactly what they intend to write, and how they will present their arguments.

Some guidelines for the report content are as follows:

- list the enhancements chosen.
- note, in one sentence each, how the enhancements were implemented.
- what files were changed (students may print out the files and highlight the changes). Identify why the changes were made, and, if not obvious, and explain the rationale behind the changes.
- convince the reader that each enhancement works. To do this, students might need to give a test bench, and show the waveforms for executing it. They need to ensure that the test bench really does demonstrate the enhancement working, and clearly describe what is happening in the testbench.

TABLE I

LABORATORY INSTRUCTIONS: ENHANCE AND IMPROVE TINYCPU

Working in groups of two, over sessions 3 and 4 (you can also access the lab freely at other times, or use Modelsim at home), you need to enhance TinyCPU by **four difficulty levels (D)**, spanning at least **3 categories**. Several ideas are given below, along with their **/D/** score:
Cat A: Enable TinyCPU to respond to real-time events. This category includes changes to its instruction set **/1/**, implementing of interrupt vector(s) **/1/**, adding an interrupt signal **/1/**, maybe shadow register bank(s) **/1/** or even a shadow stack **/1/**.
Cat B: Improve the number-crunching capabilities of TinyCPU. Make TinyCPU able to handle at least four multiply-accumulate instructions per cycle **/1/**. This can be done in many ways, including through a superscalar-type arrangement, though implementing an SIMD unit, or by adding a co-processor.
Cat C: Instruction set change: implement a compressed instruction set (maybe use Huffman coding?) for TinyCPU **/1/**, or perhaps include a microcode translation unit so that TinyCPU can execute ARM, x86 code, or code designed for some other microprocessor **/1/**.
Cat D: Addressing: increase the number of addressing modes supported **/1/**, implement dual memory areas (separate instruction and data) **/1/**, allow vector operations **/1/**. Allow zero-overhead looping through data memory areas **/1/**.
Cat E: Programming support: give the assembler full error reporting **/1/**, integrate everything with the eclipse IDE **/1/**, implement a built-in debugger for the system **/2/** (this could be a boundary-scan unit, or a breakpoint/trace unit with dedicated interface [3]).
Cat F: Hardware: demonstrate TinyCPU working on a real FPGA **/3/**, showing it running a program that handles user I/O.

- critically analyse the changes – how much improvement do they really make? Are there better ways of achieving this? Are there any disadvantages incurred through the 'enhancement'?

A marking scheme is given to all students to enhance the clarity of assessment, as follows (out of 100):

- 8 marks for choosing the right set of enhancements and describing a correct method to implement them.
- 8 marks for clearly identifying all the changes made, 8 marks for making the rationale clear.
- 15 marks for each difficult level chosen, comprising 5 marks for identifying a valid test methodology, 5 marks for showing the evidence of testing, 5 marks for interpreting the evidence and making clear what is happening.
- 4 marks for critical analysis of each difficulty level, and revealing potential disadvantages.

Finally, to finish the course, laboratory session five includes a cache design exercise using Dinero IV [6] and Cacti [7]; but since this is not related to TinyCPU, it will not be described further here.

Overall course assessment is by 2-hour examination (40%[1]), lab project grading for sessions three, four and five (45%), laboratory sessions one and two (10%) and one or more homework quizzes (5%).

[1]Note: the percentages given in this paper are approximate, since they vary a little from year-to-year based upon the presence or absence of quizzes and the relative difficulty level of the exam questions.

IV. PERSEPCTIVE - POSITIVES AND NEGATIVES

After five years of running the course, and two years of using TinyCPU for experiential learning, student feedback indicates that students enjoy the laboratory sessions and feel that they have learnt a significant amount of computer architecture. However it is important to note that the four laboratory sessions that involve TinyCPU were originally 6 hours of lectures plus two sessions that used Trimaran [8] to investigate the effect of various degrees of superscalar enhancement on pipelines in general. In the labs, we have thus replaced a very targeted and advanced computer architecture tool for high-level pipeline analysis with a very low level processor that students need to understand in great detail.

The reduction in lectures by six hours also requires that we teach less, but that students self-study to a greater extent. Our findings indicate that this approach works well as a means of levelling the knowledge of students: i.e. since we have a very heterogeneous international intake, we had found a remarkable diversity in the abilities and knowledge of students from various Asian countries in particular. With the new approach, especially the encouragement to self-study, students, on output, are more consistent in their achievements and knowledge.

Although there have been definite structural improvements gained through using TinyCPU, we note that it is a very simple CPU. It is not pipelined, and thus can not naturally be used to investigate pipelining issues, superscalar, instruction reordering, data forwarding. Even branch speculation and branch prediction become almost meaningless with TinyCPU. To counteract this, in 2009, almost a third of students began their TinyCPU improvements by introducing a pipeline - with either three or four stages. Many also implemented superscalar, multiple-issue and/or dynamic pipeline enhancements.

Teaching feedback scores for this course have risen, student satisfaction ratings have improved, and class size has increased as more students have chosen the course (which is optional).

To summarise, students are exposed to more hands-on computer architecture. They achieve a more consistent level, and appear more satisfied with the learning experience. However there remains a residual concern that the deflection from many of the issues that actually plague modern CPUs (and which are largely irrelevant to TinyCPU due to its simple structure), needs to be counteracted in some way. It may be that, as a generality, any CPU which is sufficiently simple to teach to all students, is insufficiently complex to demonstrate these complex issues.

V. CONCLUSION

TinyCPU, a stack-based processor, written in Verilog, and released by Nakano of Hiroshima University, has been adopted for the experiential teaching of computer architecture at an advanced level in Nanyang Technological University. This paper describes the processor and its method of adoption. The advantages and disadvantages of using TinyCPU have been

presented, and the educational benefits of its use analysed and discussed based upon the experiences of two years of MSc course teaching which incorporate it.

REFERENCES

[1] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an FPGA," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2008, pp. 723–728.

[2] M. Varanasi, "Computing curricula - computer engineering," in *Microelectronic Systems Education, 2003. Proceedings. 2003 IEEE International Conference on*, jun. 2003, pp. 2 – 3.

[3] I. V. McLoughlin, *Computer Architecture: an embedded approach*. McGraw-Hill, Jan. 2011.

[4] I. V. Mcloughlin, D. L. Maskell, S. Thambipillai, and G. W. Boon, "An embedded systems graduate education for Singapore," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2007.

[5] (2010, Aug.) Koji Nakano TinyCPU Wiki. [Online]. Available: http://www.cs.hiroshima-u.ac.jp/nakano/wiki/

[6] (2010, July) Dinero IV trace-driven uniprocessor cache simulator. [Online]. Available: http://pages.cs.wisc.edu/ markhill/DineroIV/

[7] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," HP, Tech. Rep., 2001.

[8] L. N. Chakrapani, J. Gyllenhaal, W.-M. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran: An infrastructure for research in instruction-level parallelism," in *In Instruction-Level Parallelism, Lecture Notes in Computer Science*, 2004, p. 2005.