Conquering the Complexity Mountain: Full-stack Computer Architecture teaching with FPGAs

A. Theodore Markettos, Simon W. Moore, Brian D. Jones, Roy Spliet, Vlad A. Gavrila
Computer Laboratory, University of Cambridge, UK
theo.markettos@cl.cam.ac.uk

***Abstract*—Modern computer systems are exceedingly complex, and increasingly so. This makes it challenging for students with no background in computer systems to climb the mountain of 40 years of design, particularly within a constrained teaching timetable. Through the medium of FPGAs, we have designed an 8-week course to take students from basic digital electronics through to processor design, modern software tools, applications, system-on-chip integration and electronics manufacturing. We recount our experiences with rapidly bringing students up to speed with the modern world of computing systems, and some of the lessons we, as course designers, were taught by the process.**

I. Introduction

"All problems in computer science can be solved by another layer of indirection, except for the problem of too many layers of indirection". David Wheeler's aphorism has never been truer today, where we take for granted many layers of abstraction from cloud computing that 'just happens' (somewhere, somehow) to consumer products that sit atop vast piles of standards, commoditised components, and decades of evolution under market pressures.

For a student, who has only had some limited experience with software development, this can be a daunting mountain to climb. They have not had the experience their professors might, of living through evolution as it happens and accepting developments one by one. Even ask someone in industry and they might reminisce about System/360, VAX, DOS, Windows 95 or XP, based on whatever date they came into the business – and anything before their time is taken for granted.

The Raspberry Pi, a cheap 'simplified' computer for teaching is anything but simple. It has a $25\,\mathrm{mm}^2$ system-on-chip [1] in a $40\,\mathrm{nm}$ process, approximately 50 million gates [2]. It runs an operating system kernel with 17 million lines of code [3]. While you can do simple things with it, there is a lot of complexity hidden behind the scenes.

This means that a course designed to teach modern computer architecture must teach the many abstractions that go up to make modern systems, treading a fine line between oversimplifying and providing so much detail that it becomes overwhelming.

We have designed a course with these principles in mind, taking students with minimal hardware knowledge into modern systems design in a short period.

II. Background

This course was designed for second-year university students who are studying exclusively Computer Science (CS).

Students arrive at university with diverse backgrounds and the course is structured so that they need have no prior experience with CS before admission. While complete beginners are rare, many will have ad-hoc CS experience rather than formal education. Such experience is frequently at the application level - designing mobile apps or websites - rather than systems-level programming or hardware.

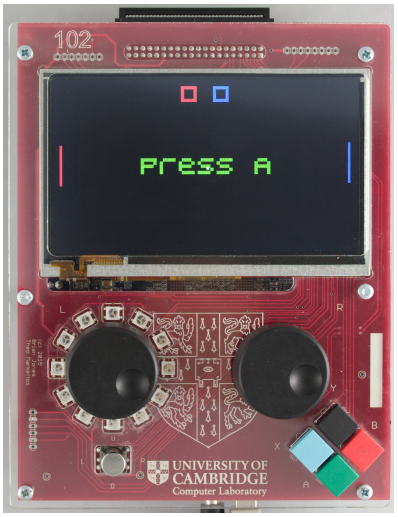

Fig. 1. Our teaching board running a student's code (courtesy Jamie Wood)

The first year course is intended to build foundations of CS: there are courses in functional programming, object-oriented progamming, algorithms, mathematics and operating systems. They are also taught elementary digital electronics: the use of logic gates, clocked logic, state machines, and a small amount of transistor behaviour. The most complex structures they are exposed to are shift registers, RAMs and elementary programmable logic (PALs).

In designing the 'ECAD and Architecture' course, the objective was to introduce the second year students to modern programmable hardware (FPGAs), to the 'Electronic CAD' (ECAD) tools used to design for it (Altera's Quartus II suite), and to modern computer architecture. The class has about 100 students, and the course consists of 24 hours of laboratory sessions over an 8 week teaching period. It is concurrent with a 'Computer Design' course consisting of 18 hours of lectures. Each week we ran two 3-hour afternoon laboratory sessions, with a student expected to attend one session or perform equivalent work at home. Experienced staff and PhD students are present at the laboratory sessions to help out with problems.

In our department we choose not to grade practical work in favour of encouraging collaboration and learning from the experience. However all students are expected to complete the work and they are penalised if they do not. Thus the course must be designed so that all students can achieve the goals, which are assessed by an oral interview and demonstration. The goals are not thus objectives in themselves but a structure for their learning.

978-1-4673-8584-8/16/$31.00

III. Course design

Allied with the Computer Design lectures, we wished to introduce students to a number of concepts:

Hardware description languages	
Large scale logic design	*Modern EDA tools*
Logic simulation	*Test-driven development*
Behavioural modelling	*Processor architecture*
Assembly programming	*C programming*
Modern compiler/assembler toolchains	*Unix development environment*
System-on-chip construction	*Whole-system evaluation*
Relationship to mainstream architectures like x86	

In addition, as a byproduct of our approach, we also introduced students to modern hardware manufacturing techniques though this did not form a formal part of the course. In doing so we also challenged the view that computer scientists are downstream 'consumers' of electronics from large tech firms, that it is possible to design your own.

Constraints on our course design included the restricted timetable, concurrency with the lecture course meaing material could not be used before being lectured, the assessment requirement requiring exercises that students of all abilities could complete, and the need for a primarily bring-your-own-device (BYOD) environment to support students working at home on their own laptops with varied hardware and software (and encourage students to explore beyond the course).

IV. FPGA board selection

The original motivation for redesigning the course was that the class size had grown such that we had insufficient hardware to run our previous course. This used Terasic tPad boards, consisting of a DE2-115 board with an Altera Cyclone IV FPGA and a touchscreen display that are no longer sold.

One feature of this course is we believe each student should have their own board that they can take home and keep for the academic year. This means they can do development in their own time, not having to rely entirely on class hours. They do not have to share equipment that would have to remain in the lab. They can also usefully overlap FPGA build times with other activities. By keeping the board beyond their course submission they are encouraged to explore further than the taught materials.

We looked for other low-cost FPGA boards of a comparable rôle. We were particularly attracted by Terasic's DE1-SoC board, which combined a Cyclone V FPGA that has an ARM processor onboard. The ARM processor can run Linux and has a good collection of 'hard' peripherals (Ethernet, USB, SD card) that are necessary for modern computing but awkward to build in FPGA. The ARM and FPGA fabrics each have dedicated on-chip memory, while sharing a cache-coherent memory interconnect between ARM and FPGA. The FPGA can be used standlone, or the ARM used standalone, or the ARM can be used with FPGA-built hardware plugged into its memory map to build innovative peripherals. The board is competetively priced and in a small form factor that students can carry around. Thanks to a generous contribution from Altera we were able to purchase 130 DE1-SoC boards.

V. Add-on hardware development

During FPGA board selection we identified a gap in the marketplace. Many 'beginner' boards contain small FPGAs that limit scope of what may be achieved. Boards with larger FPGAs have peripherals that are often very complex making them a daunting prospect for students. USB, for example, appears to the naïve to be simple, but is highly complex with data rates ranging from 1.5Mb/s to 10Gb/s, complex protocols and a frequent need to deal with bugs in devices.

Additionally, the simpler devices provided on many boards (VGA output, PS/2 keyboard) typically require external hardware that students lack (keyboards are now USB, most students use laptops and don't have their own monitors) and cannot be easily carried around. Other boards were not portable enough.

The DE1-SoC lacks such simple onboard input/output (I/O): there are switches and LEDs, but not much more that can be used without either a complex hardware/software stack or external hardware plugged in. After surveying the market for possible other options we decided that we could build our own add-on board containing simpler peripherals that we could package with the DE1-SoC into a handheld unit.

We wished to allow students to design hardware for useful devices without delving into too much complexity. We selected a number of devices for them to drive:

Serial output frequently requires high performance hard serializer/deserializer (SERDES) blocks. To reduce complexity and add a visual component, we included some LEDs capable of 24b colour individually controlled via a serial protocol (the Shenzhen Worldsemi WS2812B branded by Adafruit as 'NeoPixels'). Whilst it is possible to use software-based bit bashing to produce a valid serial stream, a simple Verilog serial output module makes timing reliable.

Rotary shaft encoder requires debouncing and state machines.

Serial input was explored as both a teaching goal and a board-design necessity. Push buttons and a joystick were connected to a shift register chip which was used to serialise the 13-bit input over a restricted number of wires. Verilog on the FPGA could parallel-load the shift register and read out its inputs.

Touch screen to add a display. This is more complex but we can provide code to provide a VGA-like raster scan for the LCD and an I^2C interface for the touch controller.

I^2C EEPROM with temperature sensor was included to hold any necessary per-board calibration data for the touch screen, but also the temperature sensor allows real-world sensing applications.

After some user interface mockups, we decided to design a board to sit on the back of the DE1-SoC and be like a Gameboy-style handheld games console (figure 1). This is a form factor students are likely familiar with and comfortable to hold. We added a small navigation joystick as well as push buttons in the X/Y/A/B style favoured by games. The two rotary shaft encoders are sited above the buttons and at the top is the screen. Around one rotary encoder is a ring of 12 tri-colour addressable LEDs, two notches of the encoder relating to each LED. Each rotary dial also has a centre-click position - this provides a 'find and select' behaviour.

By asking students to write hardware to drive these inputs and outputs, we could both introduce them to hardware design principles and give them components they could use in later application programming.

VI. Hardware manufacturing

The decision to design and manufacture our own I/O board was a reluctant one. Reluctant because manufacturing added quite a lot of uncertainty and risk to the project – we started

design in July and we had to deliver the course in October; we could change the course content but the dates were fixed.

After deciding to build a custom PCB in volume of 150 (to allow for production wastage, class growth and boards for development and test), a major headache was sourcing the components. While it's possible to source components like liquid crystal displays (LCDs) in single units in the UK, finding a vendor who would sell 150 at acceptable prices and in our constrained timeframe was difficult. In the end we found a Chinese supplier (buydisplay.com) with a good range and prices who said they could ship within our schedule (two weeks). Similarly the tri-colour LEDs were not available in volume in the UK (we needed about 2000) so we found suppliers on AliExpress. Ordering from these sources was challenging to our university accounting practices which were not designed for agile parts sourcing.

The dependencies within the project being complex, it made for pressure on the tight timescales. For instance, we did not have time for samples of mechanical parts like knobs to be shipped from the US to test in our UI mockups so we 3D printed approximate copies to try out. We could then order production volumes without having seen samples.

Having identified (and while sourcing) parts, including from traditional suppliers such as Farnell, Mouser and Digikey (direct shipped from the US), we designed a PCB that would sit on the back of the DE1-SoC to take the I/O and connect by ribbon cable to a general purpose I/O (GPIO) port. We made two prototypes before the production version to adjust our design and fix various bugs.

One challenge was that 150 is an awkward volume. It is large enough that the time spent retroactively fixing bugs on the board becomes uncomfortable (5 mins per board $\times$ 150 is painful for one person, and we don't have a factory of staff to parallelise this), but the volume is not large enough to start building small batches, optimise the production process, and then ramp up volumes as bugs are ironed out.

On the other hand we did consider how to optimise the assembly process to take the least time. We designed a sandwich of two pieces of plastic we laser-cut to hold the display in front of the PCB and act as front panel. As we have in-house laser cutting facilities we could rapidly adjust the plastic design to accelerate assembly times. Having done so, we laser cut assembly jigs so that the layer stack could be rapidly built upside down and then all screwed together.

The PCBs were etched and assembled in Ireland using an outside manufacturer from parts we had supplied. We then organised an afternoon for students to assemble the boards they would be using for the labs. After some initial training in the assembly line we had built, approximately 10 student volunteers assembled 130 boards in about 3 hours. Students showed a keen interest in how manufacturing is organised and the steps required to minimise production time. After testing, the successful boards were made available to students as they progressed through the exercises, and we reworked test failures to fix production issues.

VII. Exercise design

Having set the hardware design and production in motion, in parallel we considered how to design the exercises.

As students had no background in hardware systems or embedded development, we adopted a bottom-up approach. Starting from elementary digital electronics, we gradually built up through the complexity stack.

A. On-line SystemVerilog tutor

Some years ago we realised that lecture courses are not a good medium for introducing students to programming languages, particularly those with unfamiliar concepts like hardware description languages (HDLs) e.g. SystemVerilog. Lectures are good at introducing concepts, but not good at explaining syntax or how to link those concepts together. In a previous project [4] we designed an on-line tutor that instructs students in how to write SystemVerilog and how to use it to describe hardware. Built into the tutor are exercises where the students' code is run in a Verilog simulator and the outputs checked against a good solution. The outputs are represented visually (for example, waveforms or state transition diagrams) and, if not correct, advice is given as to what problems need to be addressed. Students work through exercises involving combinational and sequential logic, the students checking their SystemVerilog as they go.

Instead of introducing HDL design through lectures, we prescribed the tutor as the first exercise in our course. After this it is remarkable that, across the aptitude range, they hit the ground running and are able to write valid code: bugs tend to be in describing the wrong hardware rather than invalid semantics. We no longer tend to see fundamental problems such as confusing hardware module instantiation with software function calls, though syntactical misconceptions such as confusing vectors with multi-bit data types still occur.

VIII. Virtual machines

With foundations from the web-based tutor, we transition students to a development environment on their laptop. There are a number of logistical challenges in doing so.

First is that EDA tools are not intended for classroom use, or for use on mobile devices. Typically they demand workstation-class desktop machines with large monitors. Additionally newer FPGAs typically require more RAM to synthesise (eg 6GiB for the Cyclone V against 1.5GiB for the Cyclone IV). A key driver in the development of our exercises was to minimise build times. A previous lab had 20-30 minute build times, which are quick for FPGA builds in industry (multi-hour builds are the norm). We found this destroyed productivity and interest: in past surveys some students said they had been put off the hardware industry as a result.

Students' laptops are often designed for portability over performance. Students are also cost-conscious. While many students have good specification laptops (Apple MacBook Pro and similar), our baseline for development was an inexpensive Core i3 machine as might be sold in a supermarket. The gulf between commercial tools hungry for machine resources, restricted-resources laptops owned by students, and the wish to deliver a good teaching experience, remains an ongoing tension.

Additionally, EDA tools are complex to install and platform-sensitve, but we cannot mandate the operating system students run on their laptop. To avoid this problem we configured the tools in an Ubuntu image in a VirtualBox virtual machine and distributed this to students. Inside the virtual machine we installed the Quartus II FPGA synthesis tool, the ModelSim Verilog simulator, the GCC toolchain for RISC-V, and the SDL libraries for our device simulator.

For students without a suitable laptop we had a small number of Ubuntu-based workstations in the classroom for

students to work on. While the machines were of a reasonable specification, the network-based fileserver architecture we had to use severely limited performance: operations could be 8× slower with files and tools coming from the fileserver. EDA tools are designed for fast disc and plenty of RAM rather than disc kilometres away over a campus LAN.

One pitfall with distributing the tools in a VM is that we had to support the VM software on several versions of Windows, Mac and Linux, and using some advanced features of the VM (USB port passthrough, screen resize). While there was initially some frantic debugging as early-adopter students encountered problems, we rapidly grew our knowledge and adjusted the instructions before the next group reached them.

IX. Simulation environment and Test-Driven Development

In the lectures and exercises we introduced a strong focus on test-driven development (TDD). This was for a number of reasons. From our perspective, we wanted to build in slack into the timetable so that unexpected hardware manufacture delays could be accommodated. From the user experience point of view, test-driven development reduces the reliance on long build times and greatly increases productivity. Finally TDD is commonplace in industry for the latter reason, so introducing it is worthwhile training for students. Therefore we endeavoured to provide test frameworks and simulated components so that students could do their design mostly in simulation.

Once a student was set up with their VM or class workstation, we tested this out by introducing them to the ModelSim simulator. This was acheived by taking an exercise they had already solved in the on-line tutor and transferring it to ModelSim, guiding them through the using the GUI. This bridged the gap from the on-line tutor, where the simulator is hidden, to using a commercial tool.

Throughout our exercises we used marginal 'sticky notes' to explain concepts not relevant to the main thread, but which might be of interest to those who want to go deeper. This enabled us to give directions to other materials and further exploration, without confusing the main text.

X. HDL for hardware peripherals

We then introduced students to some of the peripherals on the add-on board, namely the rotary shaft encoder and the shift register used to read buttons and other I/O.

This was intended to introduce several concepts, notably debouncing, synchronisation and modular development. A testbench was provided and students given scripts to execute their code against the testbench. Students had to first build a debouncer and check it against the testbench, then use the debouncer in a controller to interpret the quadrature code emitted by the rotary encoder. The testbenches were carefully designed to check anomalous behaviour such as bouncing inputs and partial movements. Discussion with students indicated this was a good way of making them think about how to take a specification and implement a real-world state machine.

Another exercise asked students to write a controller for the 16-bit shift register on the add-on board, to read the inputs which it multiplexed onto a single wire. In this case we provided model Verilog of the physical shift register component (a pair of 74HC165s) and asked them to write a module that provided its inputs and outputs. The model could then be replaced later with pins to the real chips. This exercise was made optional in the interests of time.

	Open	Simple	Soft	Compiler
ARM Cortex A9				✓
NIOS II			Unmodifiable	✓
Thacker's Tiny Computer	✓	Too simple	✓	
RISC-V based	✓	✓	✓	✓

TABLE I. Processor candidates

XI. RISC-V processor in simulation

Since we wished to teach computer architecture, we required a processor to teach. We considered several alternatives (table I), including the hard ARM core inside the DE1-SoC; Altera's NIOS-II soft core; Thacker's Tiny Computer (TTC), a simple processor used in previous labs; and a soft core based on the RISC-V architecture. Our criteria were a core where students might view the source code, understand and modify it; and one with a modern toolchain including C compiler.

The ARM and NIOS II source code is not accessible to students, while the TTC has a restrictive instruction set and no C compiler. RISC-V, an architecture growing in popularity in the academic community, has a classical RISC instruction set, GCC compiler support and available soft cores.

After looking for soft cores we chosen the Yarvi ('Yet Another RISC-V Implementation' [5]) as a relatively compact core that implements the RV32I intruction set, the minimal required for a RISC-V. It is GPL licensed and has about 500 lines of SystemVerilog, running at 75MHz on a Cyclone V.

Yarvi has internal data and instruction memories. We substantially expanded it by increasing these to 4KiB each and adding memory-mapped Avalon slave ports to allow another CPU to view and change these memories (loading core into the Yarvi was previously only possible at compile time). We also added an Avalon memory-mapped master port to allow the Yarvi to access external memory and I/O, and a pipeline stall to allow multi-cycle memory accesses. Finally instruction tracing support was added to see the progress of instructions through the pipeline.

After initial Computer Design lectures where students were taught the basics of computer architecture, a lecture was dedicated to showing the Yarvi source code and walked through each section. This took students' knowledge of SystemVerilog from earlier exercises, the previous material on architecture, and linked it together by showing a real processor they would use.

A. Toolchain and exercises

In their VM we had already pre-configured the RISC-V GCC toolchain and standard Unix tools such as 'make'. The Yarvi simulation environment was comprised of several parts:

- The Yarvi Verilog code itself
- Scripts to simulate the Yarvi code in Modelsim and display instruction traces
- Framework assembler and C programs including linker scripts and stack setup to run with no operating system
- Makefiles to build assembler or C programs using the RISC-V GCC tools
- Scripts to convert GCC outputs into memory formats that Modelsim (and later Quartus) could understand

We gave students the simulation environment and explained how to build and run assembly code and execute it in Yarvi in Modelsim. Since some students were unfamiliar with the Unix environment, understanding and using all the different parts proved to be challenging. In addition this was most students first introduction to both C and assembly coding.

We set a number of exercises to build up gradually from a framework we provided. Firstly we gave them code to put in the shell that makes a simple output 'DEBUG PRINT' in the instruction trace, by writing to a magic I/O address. This enabled very simple output of values in a simulated processor with no external I/O.

Secondly to introduce them to assembly code, we asked them to write a 32-bit division algorithm based on some high-level pseudocode and a provided test framework. When their division code did not work, students had to use the instruction trace to debug their program's behaviour.

Finally we introduced the concept of linking, and how to call assembler from C, including use of calling conventions. We asked them to modify their division code slightly to make a 'remainder' function. Then we provided a naïve algorithm for finding prime numbers by trial division, and asked them to implement the algorithm using their remainder function. This turned out to highlight various bugs in their assembly code (a common one was not implementing calling conventions).

B. Applications with emulated peripherals

Following our test-driven development focus, we then wanted to ask them to write code to read the inputs and drive the display of the add-on board, but do so in a simulation environment. For this reason we wrote a Modelsim library that connected to the simulated Yarvi's memory interface, sending Avalon reads and writes over a TCP connection. Another application connected to the TCP port and emulated external peripherals to the Yarvi, popping up a window to show the LCD contents and taking keyboard input instead of rotary dials and buttons. Combined with the slow simulation of Verilog, this is not quick but does allow practical application development without the larger overheads of synthesis.

Drawing this all together, the final simulation exercise was to implement an application in the emulated environment, using the Yarvi core, I/O, C and assembler. The baseline application was Etch-A-Sketch, where simply a pixel can be plotted based on the X and Y coordinates read from the dial inputs, and a button to clear the screen. An optional exercise was to implement something more complex – we suggested Pong, but we also saw Space Invaders and Flappy Birds.

A considerable challenge to complex applications was the 4 KiB of instruction memory (an arbitrary limit we didn't consider) and the slow speed of the emulated system. This caused them to think carefully about code size and about efficiency: for instance only redrawing the pixels that were necessary, not the whole screen.

A working application was one of the two compulsory oral assessments of the course. As part of the assessment we asked them questions about Verilog (troubleshooting some intentionally-miswritten code), the Yarvi split instruction/data memories and their impact on software such as operating systems, and to calculate the complexity of their division routine using the instruction trace. This aimed to consolidate their architectural knowledge so far, from HDL to processor architecture and real-world software complexity.

XII. FPGA implementation

Having completed the first stage, we then moved into FPGA implementation of their systems. This was intentionally shorter than the simulation stages, to reduce the impact of synthesis times on their experience.

First, we provided a walkthrough of the Quartus synthesis tool using a simple LED flasher. This was intended to explain the many detailed concepts of FPGA synthesis, from pin assignments, synthesis, timing analysis, programming, download, and optionally inspecting the generated floorplan.

Once working, we asked students to import their rotary controller and shift register driver and test outputting them on LEDs, purely in logic at this stage. Since the contents of the FPGA are small at this point, synthesis times are quick.

A. System-on-chip design

We then introduced Altera's Qsys tool. While many courses stop at Verilog-level implementations, this was intended to show higher-level tools for building a system-on-chip (SoC) and using library parts. Using the simple rotary encoder, shift register driver, and a 7-segment LED converter, students build Qsys components from them. We then showed how Qsys makes it easy to plug these components together, and the type-checking it provides.

B. The Yarvi SoC

Finally, we joined the parts by building Yarvi into an SoC. The FPGA SoC-building process can be quite error prone. We discovered in previous labs that, if you give 100 students a set of instructions to build an SoC, some percentage will make mistakes: omitting steps, inverting resets, using incorrect clocks. The design of the tools sometimes does not help, hiding away critical settings in obscure dialogues. Build times are long, so a mistake can cost many hours. When a mistake is made, often it is very hard to debug and requires much skill and experience on the part of teaching staff. As one past student said 'this trains the staff more than the students'.

For this reason we provided a pre-built project with parts of the Yarvi SoC already installed. This included the Yarvi SoC, a video memory made from block RAMs, an LCD controller to DMA and generate pixels with appropriate timing, and a simple LED PIO. To this students were asked to add the rotary controller and shift register Qsys components they already made, and to install their application software into the Yarvi memory. They then build their FPGA and test it on their board. The end result is running Etch-A-Sketch, Pong or another game on hardware/software stack where they worked at all layers.

The final oral assessment had students demonstrate their system, and then answer questions on FPGA place and route and timing analysis. A final summing up question asked how a many-core Yarvi system might differ from a multicore x86 system, and discuss their behaviour (with particular respect to caching, coherency, and fluidity of memory placement).

XIII. Evaluation

After a last-minute supplier change for accounting reasons, the add-on boards were manufactured in good time. Students were interested to learn about manufacturing techniques and how designs could be optimised for manufacture. We discovered some manufacturing failures: yield of the multi-colour LEDs on the board was low because a failure of one in the ring causes the others to fail, and we did not have equipment to repeatably test touch screens. Students really appreciated the work we had put into designing the hardware and packaging.

The SystemVerilog tutor was remarkably trouble-free: students were able to get up to speed in writing HDL with no major problems. One downside of the test-driven development model is that the tests are never fully comprehensive, and it is

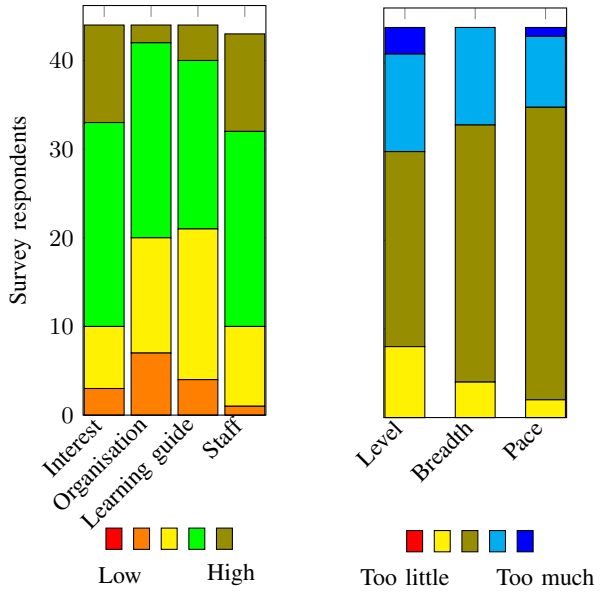

Fig. 2. Student survey results

possible to short-circuit the requirements with a solution that passes the tests.

When students moved on to assembler, many found transitioning from Boolean logic gates to two-dimensional arrays of bits (shifting and masking) challenging. It might have been useful to include a primer on bit manipulation. Many of them made a good attempt at writing division in assembler, however some would have appreciated more time spent on this approach.

The FPGA part was remarkably trouble-free – in fact students criticised it for being too easy. That is in marked contrast to previous years where FPGA build times were the primary complaint and students could waste weeks debugging.

The results of the end-of-course survey are presented in figure 2, which suggests we mostly achieved our goals. 97% of students completed their assessment successfully.

XIV. Conclusion

We delivered a challenging Computer Architecture practical course to 100 students, of entirely new material, in 24 hours of laboratory time.

As part of this effort we designed and fabricated new teaching boards, involving globalised sourcing and modern manufacturing methods, and involved students in this process.

Students began the course with basic knowledge of digital logic gates and sequential circuits. We taught them up through the layers of abstraction, beginning with teaching SystemVerilog using an interactive web tutor. Having gained a good understand of HDL, they designed controllers for physical components such as rotary encoders and shift registers.

We then introduced them to the Yarvi RISC-V processor in simulation. We used this to teach assembly programming, including use of modern tools (GCC) and the Unix environment, and expressing algorithms in assembler. They were then taught how to interwork assembler and C.

To encourage test-driven development and to speed up application writing, we built an emulator of the peripherals on our FPGA board. This enabled students to develop applications while the processor was simulated and memory traffic was sent to simulated peripherals.

Subsequently, we taught students how to use FPGA tools and how to compile their Verilog for FPGA. We also showed them how to turn their Verilog into modular system-on-chip components. Having provided them with processor installed in a framework to take away some of the tricky debugging issues, they then added their components to the FPGA project. After synthesising their design they were able to test their applications on the processor running in FPGA.

As a result they created a project equivalent in complexity to a 1980s arcade game, from the gate level to the application.

We navigated a number of obstacles and learnt numerous lessons along the way. We learnt that medium-scale volume hardware manufacturing is possible in a UK university setting but there are a number of pitfalls, in particular relating to accounting practices. In-house laser cutting and 3D printing solved design problems we could not if we outsourced. While we had no ability to hire assembly staff, it was fortuitous that we could involve students in the process and it became part of the learning experience.

Initial language learning, particularly for languages with unfamiliar paradigms like hardware description, works much better with an online tutor than by lecture or by simple exercises. Test-driven development allows not only rapid feedback of success, but also allows decoupling layers such as hardware and software design. This enables software to be perfected before time-consuming hardware tasks.

Deploying the exercises in a heterogenous bring-your-own-device and workstation-based environment had numerous pitfalls, but none which were insurmountable. EDA tools are not intended for portable use, and there are many technical obstacles they bring up in both access and performance, and this guided many of our decisions. This achieved the goal of most students being able to take their board home and work wherever and whenever was convenient for them.

Some students can find rapid progression through the layers challenging, but were able to make progress nonetheless and with minimal support outside of contact hours. 97% completed their assignments and received credit.

We have published our lab material at
`www.cl.cam.ac.uk/teaching/1516/ECAD+Arch/`

Acknowledgements

We would like to thank Altera for generously supporting our teaching effort, funding summer interns and hardware donations. The Yarvi processor was originally written by Tommy Thorn and Paul J. Fox did some of our initial adaptations. We also thank all students who gave us comments and feedback during the labs.

References

[1] "Raspberry Pi Zero extreme teardown." [Online]. Available: https://www.youtube.com/watch?v=HH5cFqc9OcM

[2] "Standard cell libraries: TSMC." [Online]. Available: http://www.europractice-ic.com/libraries_TSMC.php

[3] "Linux v4.5 statistics." [Online]. Available: https://www.linuxcounter.net/statistics/kernel

[4] S. Moore and K. Taylor, "An intelligent interactive online tutor for computer languages," in *25th Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (SGAI)*, 2005.

[5] T. Thorn, "YARVI - Yet Another RISC-V Implementation." [Online]. Available: https://github.com/tommythorn/yarvi