Further Experiences Teaching an FPGA-based Embedded Systems Class

Stephen A. Edwards[0000−0003−2609−4861]

Columbia University, New York City NY, USA
sedwards@cs.columbia.edu
http://www.cs.columbia.edu/~sedwards

Abstract. I describe thirteen years of teaching an embedded systems class at Columbia University that spans three "board eras." Students now develop Linux systems with custom FPGA-based peripherals. The soaring complexity of these systems has enabled more ambitious projects at the expense of making it impossible for students to learn "everything" about what they are developing. As such, should students be learning similar skills or specializing in hardware or software?

Keywords: Embedded Systems · Undergraduate education · Graduate education · FPGAs · Linux · Device Drivers · VHDL · System Verilog

1 Introduction

In 2004, I started teaching an embedded systems project course with hardware implemented on FPGAs. I reported my initial experiences at WESE in 2005 [2]; in this paper I summarize how my course has evolved since then.

The class has always centered around students implementing a self-designed project on a supplied FPGA board; as such, the choice of board has been both a key driver and limiter of the class. This paper discusses my experiences with the class chronologically, which I think of as consisting as three "eras," one for each FPGA board.

The basic outline of the class has always consisted of a sequence of canned lab assignments designed to familiarize the students with the board and its development tools followed by work in teams on the project. This division has worked well, although it means the projects themselves are often quite rushed given that the students only effectively work on them for half the semester. Given the complexity of the boards and the development tools, however, there is probably no alternative short of extending the class across two semesters.

We started with a Xilinx Spartan IIE board that quickly showed it limitations brought on by too few pins for many peripherals, switched to an Altera Cyclone II-based board that eliminated the pin limitations to expose the next problem: insufficient software support for complex peripherals, then switched to an Altera Cyclone V with on-chip ARM processors to enable the use of Linux to ameliorate the software challenges and replace them with sheer complexity. Below, I discuss details of these three eras.

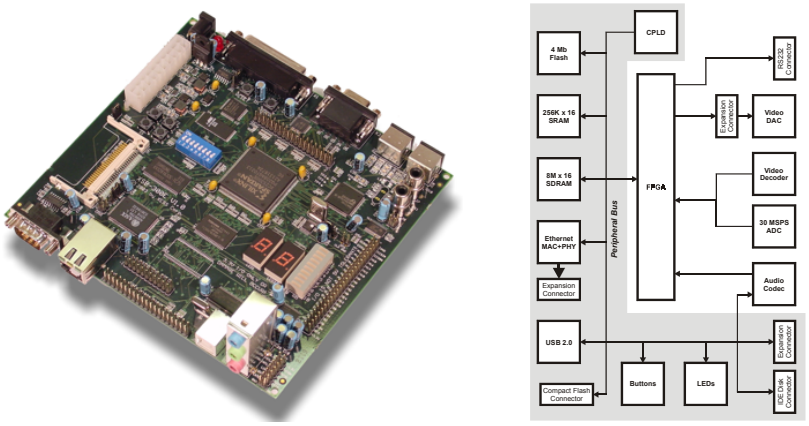

Fig. 1. The XESS XSB-300E Board and its block diagram [5]. The limited number of pins on the Spartan IIE FPGA demanded a bus-based architecture that made using multiple peripherals challenging.

2 2003–2006: The XESS XSB-300E Era

We used a Xilinx Spartan IIE (XC2S300E) board from XESS Corporation for the first four years of the class [2]. While well-equipped with peripherals, including video input and output, Ethernet, USB 2.0, a serial port, SRAM, DRAM, Flash, and an audio CODEC, these peripherals were connected to a shared bus (i.e., each peripheral's data lines were connected to the same FPGA pins; see Fig. 1), which made using multiple peripherals difficult since students would have to develop their own bus arbiter. The board designers likely did this because of the paucity of pins (only 208, including power) on the QFP package, but it constrained the designs students could implement.

During this era, students did six lab assignments, hardware-only, software-only, and hardware/software interfacing, before moving on to projects that had to incorporate both custom hardware and software. This structure has remained to this day, but I have changed the details. During this era, students built video games with custom video-generation hardware (generally without a frame-buffer, given the limited on-chip memory and difficulty of using the large off-chip SDRAM), audio projects, and networking projects.

Most project teams encountered similar challenges when designing custom hardware blocks. Limited on-chip memory forced them to either become parsimonious with their storage of data or utilize off-chip memory. Using the 512K of off-chip SRAM was straightforward but often still insufficient; the 16 MB of SDRAM was much larger but difficult to use because if its complex protocols, timing constraints, and refresh requirements. Moreover, using either of these memories from hardware meant either creating an OPB bus master and vying for control of the bus, or simply dominating the off-chip bus and foregoing the use of any other bussed peripherals. Most groups took the latter route.

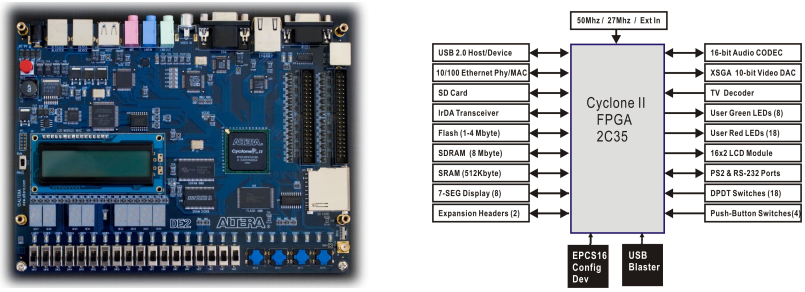

Fig. 2. The Terasic DE2 Board and its block diagram [1]. Ample pins on its Cyclone II FPGA enabled a star-based topology that made using external peripherals vastly easier than with the XSB, but the need for complex software for peripherals such as the USB controller was limiting.

3 2007–2013: The Terasic DE2 Era

After four years of grappling with the shared bus topology of the XSB board, in 2007 we made a big switch and adopted the DE2, an Altera Cyclone II-based development board developed by Terasic Technologies in 2005 (Fig. 2). Ample pins (672 total; 475 for I/O) allowed the DE2 designers to connect the peripherals on the DE2 in a star configuration, making it far easier to use multiple peripherals from VHDL compared to the bus-based XSB board. Moreover, the array of peripherals on the DE2 was similar to that on the XSB, so the knowledge of which projects would work transferred over easily.

The switch from Xilinx to Altera ecosystems was annoying but largely lateral: Xilinx had XST, the Microblaze processor, the OPB, and Platform Generator; Altera had Quartus, the Nios II processor, the Avalon bus, and SOPC Builder.

3.1 Lab Assignemnts

For the DE2 era, we streamlined the initial lab assignments down to three:

1. *A pure hardware problem:* an old-style computer front panel that allows the user to examine and modify the the contents of on-chip memory locations. The students found it challenging to turn the level-sensitive signals from the pushbuttons into events (i.e., single-cycle pulses).
2. *A pure software problem:* a network chat client that receives, displays, and sends broadcast UDP packets interpreted as text. We supplied custom hardware consisting of a Nios II processor with a VGA character display, a PS/2 keyboard receiver, and an interface to the Ethernet controller along with skeletal software that exercised each of these.
3. *A mixed hardware/software problem*: a custom video hardware peripheral that displays a ball on the screen and moves the ball under the control of a C program running on a Nios II. The students could also implement a sound synthesizer or an image convolver, but virtually all chose the ball.

3.2 Projects

Many projects of the DE2 era featured video. The on-chip memory of the FPGA was generally too small for a framebuffer, but some groups used the off-chip SRAM, which allowed a byte per pixel at VGA resolutions (640×480).

One group did a digital picture frame that displayed JPEG files read from an SD card with a FAT filesystem. They used the SRAM for a 15-bit-per-pixel color framebuffer, the SDRAM as Nios II memory, and communicated with the SD card through a "bit banged" SPI interface.

Another group did a Pac-man clone with custom video hardware for tile backgrounds, sprite foregrounds and interfaced classical NES controllers to the FPGA. At the last minute, they switched the sprites to a "Pac-Edwards" theme.

Another group's project read AES encrypted (uncompressed) monochrome video from a raw-formatted SD card for display. Their resolution and frame rate were modest, but they implemented the AES decoding algorithm in hardware.

A real-time hardware ray tracer was one of the few projects that actually consumed most of programmable logic of the DE2's Cyclone II (others routinely exhausted on-chip memory). They had to restrict their rendered models to just six rectangles to achieve the performance they wanted, but were able to achieve color and reflections at 60 frames per second by running 20 ray units in parallel.

3.3 Challenges

The use of communication peripherals was the big challenge with the DE2. While the students could develop code to send UDP packets through the Davicom DM9000A Ethernet controller, it was not realistic for them to code a TCP/IP stack. The Philips ISP1362 USB controller was even worse. Furthermore, networking or USB controller software is not something that is easily coded as a C library; most implementations rely on the facilities of a multitasking operating system. So the use of an existing library was not an option.

Running Linux on the DE2 was theoretically possible, but very difficult. Linux normally relies on hardware virtual memory support to deliver multitasking with memory protection, but the μClinux project has developed kernels designed to run on processors without MMUs. David Lariviere and I experimented with the use of μClinux on the DE2 in 2008 [3], but we were never able to get this working well enough for students to use it routinely. The option of an MMU was later added to Altera's Nios II processor, making a full Linux port possible, but we never experimented with getting this to work.

The DE2 board ultimately grew obsolete. In 2012, we managed to order additional DE2s to accomondate our growing enrollments and replace broken boards, but this was the last gasp for them. Operating a lab with multiple, incompatible boards was not realistic for the students, so it was time to move on. Terasic and Altera had started moving on long before: Terasic released the similar, but slightly incompatible DE2-70 board (with a 2C70 Cyclone II FPGA) in 2009, and in 2012, the much more incompatible DE2-115 (with a Cyclone IV EP4CE115 FPGA). Altera discontinued software support for the Cyclone II family of chips after Quartus 13.0sp1 (June 2013).

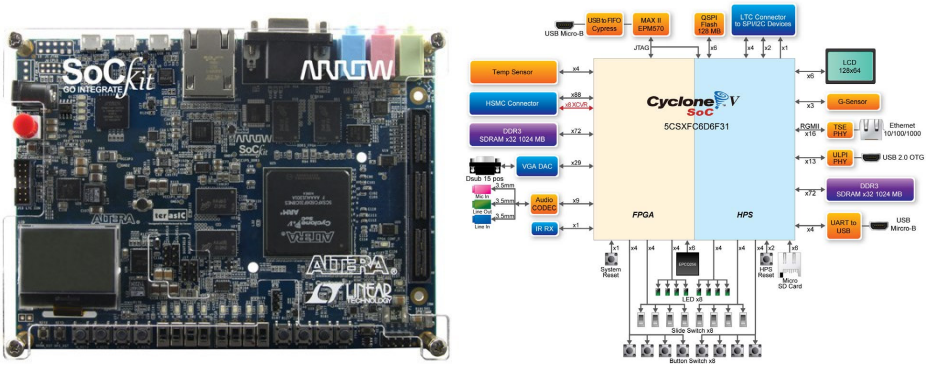

Fig. 3. The Terasic SoCKit Board [4] brought dual hard ARM9 cores via a Cyclone V FPGA, enabling the use of Linux software and its device drivers. The main failing of these boards has been the fragile micro-USB connectors.

4 2014–2016: The Terasic SoCKit Era

In 2014, we switched to Terasic's SoCKit board, one of the first to incorporate Altera's System-on-chip Cyclone V FPGA (a 5CSXFC6D6F31), which integrated dual ARM9 Cortex cores with standard peripherals (collectively, the "HPS" or Hard Processor System) with a traditional programmable FPGA fabric. The HPS portion of the chip was capable of directly running a stock Linux distribution (i.e., without any custom FPGA hardware) and exposed an Avalon bus to the FPGA fabric that could be used to build peripherals.

While not flawless, the Linux distribution for the SoCKit board was a breath of fresh air compared to software that could run on the DE2. Networking, USB, and the package manager just worked, enabling students to easily make use of USB peripherals, modern network services, and the like. It was no longer necessary for students to pour over the hundreds of registers in the Ethernet chip to write software to received a single packet.

We switched to the System Verilog hardware description language. We had chosen VHDL in 2003 because another instructor was using it, but by 2014 that class was gone and had been replaced with other classes that were using System Verilog. The synthesizable subset of System Verilog is more succinct and less error-prone than the equivalent VHDL, so we are happy with the switch.

4.1 Device Drivers

The move to the SoCKit board with Linux made it harder to access custom peripherals from software. We had been running "bare metal" C code on the DE2, allowing custom hardware peripherals to be accessed directly, e.g., by casting integer literals to `volatile` pointers. While this can be done in the Linux world (e.g., by running programs as root and using `mmap`() to map the peripheral memory into userspace), device drivers are the preferred mechanism.

I now spend a lecture on Linux device drivers. I begin with the structure of modern operating systems (userspace processes are isolated from hardware by the kernel and device drivers), then move on to the Unix device driver model (everything is a file; devices have major and minor numbers; the distinction between block and character devices), the Linux kernel module system, the catch-all `ioctl`() function, the device tree system (*.dtb* files and the in-memory database of peripheral memory layout), and all the various functions for managing resources. Most of the code in students' device drivers is a mix of boilerplate (e.g., modules must have both *init*() and *exit*() functions that are called when the module is loaded and unloaded) and error-handling (when a request for a resource fails, release all the resources that were acquired earlier and terminate).

4.2 Booting Linux

The SoCKit has a dizzying array of options (selected by a difficult-to-explain mix of jumpers and absurdly tiny DIP switches) for booting Linux and configuring its FPGA. One easy way is to boot from an SD card that carries a bootloader (uboot), a Linux kernel image, a bitstream file for the FPGA, and the Linux root filesystem. We considered this for the lab, but because we do not have the room or budget to assign each team its own board and workstation, teams have to share workstations and boards, meaning they would have to manage one SD card per team, which seemed problematic (e.g., "I thought you had the card," or "Fred has the card but couldn't make it to lab this time so we can't work").

Instead, we configured our boards to be "diskless workstations" that booted and ran off a fileserver. Each board was configured to boot from a small on-board flash chip containing a modified uboot image that would use DHCP to configure its network interface, then use PXE to download both a Linux kernel image and an FPGA bitstream. The Linux kernel would then boot and mount its root filesystem via NFS from the same server. The result was that each team could have one or more board configurations (kernel, root filesystem, and FPGA bitstream) and access them from any board. When first booted, the bootloader would download a list of available PXE images, display them on the serial console, and then allow the user to select one (i.e., the team's image) to boot.

This "diskless" configuration ultimately worked, but was challenging to set up. DHCP, PXE, and NFS servers all had to be configured to work in concert; the uboot source and boot script had to be modified because the existing PXE mechanism did not know about FPGA bitstreams; a custom bootloader had to be flashed to each board by booting yet another version of uboot and running the appropriate flashing commands; NFS had to be set up not only to serve the root filesystems to the boards but the root filesystems, kernel images, and FPGA bitstream files to the workstations so students could modify them as needed. Security was incomplete. While the fileserver and network for the boards were local to our lab and fairly well-protected from the public Internet (the fileserver functioned as a firewall between the boards and our campus network), there is little to stop one team of students from seeing or corrupting another team's files.

4.3 Lab Assignments

For the SoCKit era, we continued to start the class with three lab assignments adapted from the DE2 era. We supplied a starting point for each lab: a code skeleton that implemented a rudimentary version of the lab for students to modify.

1. *Hardware: An old-style computer front panel.* Unlike the DE2, the SoCKit has no seven-segment LEDs, so I provided a VGA LED emulator that displayed eight seven-segment digits on a VGA monitor. This component exposed a signal for each segment of each digit, making it easy to use.
2. *Software: A text chat client.* I supplied the students with a VGA framebuffer that they accessed after calling `mmap`(). We switched to USB keyboards accessed through *libusb* because the SoCKit did not have the DE2's PS/2 port. The students used the standard socket API to connect to a simple chat server I implemented in Python. They used pthreads to listen to the keyboard in one thread and handle incoming network communication on another.
3. *A mixed hardware/software system: The bouncing ball.* Students had to write System Verilog for a VGA peripheral that would display a raster with a ball whose position was controlled by registers written from an ARM processor through an Avalon bus. They also had to write a device driver that would ferry coordinates from userspace to those registers, update the .dtb (Device Tree Blob) file that characterized the memory map visible to the processors to include the VGA peripheral, and finally write a C program that bounced the ball around.

4.4 Projects

In the SoCKit era, projects grew more elaborate and utilized fancier peripherals.

Accelerators grew more common in this era. One groupt implemented an accelerator that could compute "inverse kinematics," i.e., angles for robot arm joints to reach a desired position in space for the end effector. Another group implemented a cryptographic accelerator for RSA able to perform modular multiplication and exponentiation.

One team implement a 3D pottery game in which a player manipulated a virtual pot on a turntable. The novelty here was a custom graphics controller that would display a whole pot as a series of overlapping ellipses.

The SoCKit's standard hardware and software made it possible to integrate fancier periperhals. One group used a Leap Motion controller that, like Microsoft's Kinect, uses multiple cameras to capture a 3D model of object in space, such as your hands. They used the output of this controller to simulate the operation of a digital piano, displaying the image of a piano and your fingers on the screen and synthesizing sound through the audio CODEC on the SoCKit.

Another group used the SoCKit as a controller for an automous vehicle built from a LEGO kit. They interfaced the SoCKit to the various LEGO motors and sensors, ultimately enabling it to park itself in a space while avoiding the curb and nearby vehicles. While this group ultimately succeeded, they managed to destroy a SoCKit board by connecting the 3.3V I/O of the SoCKit to 5V devices.

4.5 Challenges

Complexity has been the challenge of the SoCKit era. In addition to digital hardware design an low-level C programming, students now need to understand Linux, the boot process, device drivers, and a variety of high-level APIs (sockets, libusb, etc.). While the projects have become more modern, the amount of knowledge needed to implement a project has also grown substantially.

The fragile micro USB connectors on the SoCKit boards break easily in a public student lab. Despite adding a Terasic-supplied metal bracket and securing the USB cables to the board with nylon twist-ties, students still found a way to routinely break off these connectors, rendering the US$250 boards irreparable.

5 Conclusions and Next Steps

After 2016, I took a two-year hiatus from teaching Embedded Systems because the department needed me to teach another course, but I will resume it again in the spring of 2019. I plan to switch to another SoCKit-like board, likely Terasic's DE10-Standard or DE1-SoC, but I have not made a final decision as I write this. These boards have addressed the physical failings of the SoCKit board while providing an almost identical development platform.

I am constantly faced with wondering how much to provide to the students versus how much to ask them to develop themselves. The trend is to supply more, isolating the students from implementation details to allow them to assemble more complex projects. But at what point does this stop being embedded systems with its emphasis on domain-specific peripherals and real-world data and simply turns into another class on software development for desktop machines?

Another big question is the extent to which students should be allowed to specialize in what they work on and thus learn about in the class. Practically, students typically split their teams into software and hardware portions and only learn more about what they already prefer. This is somehow realistic from a professional standpoint, but is it good pedagogy?

References

1. Altera, San Jose, California: DE2 Development and Education Board User Manual (2006), version 1.4
2. Edwards, S.A.: Experiences teaching an FPGA-based embedded systems class. In: Proceedings of the Workshop on Embedded Systems Education (WESE). pp. 52–58. Jersey City, New Jersey (Sep 2005)
3. Lariviere, D., Edwards, S.A.: uClinux on the Altera DE2. Tech. Rep. CUCS–055–08, Columbia University, Department of Computer Science, New York, New York, USA (Dec 2008)
4. Terasic Technologies, Hsinchu City, Taiwan: SoCKit User Manual (2013)
5. XESS Corporation, Franklinton, North Carolina: XSB Board V1.0 Manual (Aug 2003)