Using Embedded Xinu and the Raspberry Pi 3 to Teach Parallel Computing in Assembly Programming

Benjamin Levandowski
CIS Department
Valparaiso University
Valparaiso, IN, USA
benjamin.levandowski@valpo.edu

Debbie Perouli
MSCS Department
Marquette University
Milwaukee, WI, USA
despoina.perouli@mu.edu

Dennis Brylow
MSCS Department
Marquette University
Milwaukee, WI, USA
dennis.brylow@mu.edu

***Abstract*—The 2013 ACM/IEEE curriculum guidelines highlight the importance of teaching parallel computing to undergraduates in higher education. Universities have responded to this need, but there remains a hole in most curricula. While techniques for parallelizing code and using tools such as MPI and OpenMP seem to be well covered, concepts of shared versus distributed memory and race conditions are often taught only at an abstracted level.**

Using the educational operating system Embedded Xinu, students can write ARM assembly code that makes use of instructions meant specifically for multicore interaction. By having students implement exercises such as the *coupon collector's problem*, they can gain a much deeper understanding of potential issues with sharing memory and maintaining cache coherence. This added curriculum fits with existing computer organization courses and helps to prepare students for future work in parallel computing.

This paper presents specific enhancements used with the Embedded Xinu operating system and its associated ARM Playground, as well as a pilot study using the new curriculum with undergraduates in a typical hardware systems course.

***Keywords*-Raspberry Pi; Embedded Xinu; Embedded systems education; Multicore computing; ARM Assembly**

I. Introduction

The port of the Embedded Xinu operating system to the Raspberry Pi 3 platform [1], provided the first example of an established educational operating system executing on genuine multicore commodity hardware. Embedded Xinu had been used on the ARM-based Raspberry Pi platform since 2013 [2], and on several other embedded RISC platforms previously. While a previous explicitly multicore non-RISC Xinu port exists [3], the Intel Single-Chip Cloud (SCC) platform never became widely available commercially, and its production successor, the Xeon Phi board, proved to depart substantially from the earlier design.

Like Embedded Xinu's predecessor, Xinu, the operating system aims to be understandable and approachable by students while maintaining the core fundamentals of a modern operating system [4]. As such, Xinu has been used to teach operating systems [5], internetworking [6], compilers [7], computer architecture [5], embedded computing [8], and unit testing [9]. Expanding its use to parallel computing is a natural next step. Parallel computing with an education-focused operating system affords students the opportunity to develop a deeper understanding of what is required to support parallel execution.

The Raspberry Pi 3 contains a Broadcom BCM2837 chip that houses an ARM Cortex A53 microprocessor [10]. This ubiquitous device is used by Qualcomm for some of its smartphone chips [11], the Nintendo Switch [12], and in many other commercial products. Because of its popularity in industry, the ARM Cortex is an excellent teaching choice with regards to availability, while still offering a gentler learning curve than non-RISC platforms such as the x86_64.

Unfortunately, the ever-increasing complexity of modern computer hardware tends to edge some foundational topics out of the undergraduate curriculum. Computer organization and low level programming are often taught, especially to students with a hardware focus, but the topics of assembly programming and parallel computing rarely mix.

In our study, we aim to find out what gaps in education could be filled by teaching parallel computing in assembly, and to determine more effective methods for teaching. We found that this is not difficult to integrate into existing curricula, and it provides another route to introduce parallel computational thinking early in an undergraduate's careers.

II. Motivation

Parallel computing can be and is taught without too much focus on the specific hardware that allows parallel computing to happen [13]. To provide students a stronger foundation, we want to make genuinely concurrent programming in assembly more approachable and common in the classroom.

A. Prior and Related Work

The 2013 ACM/IEEE Joint Task Force on Computing Curricula updated their recommendations for computer science curricula for undergraduates. Their recommendations are divided into a set number of hours for each concept within a knowledge area, each hour is labeled Tier-1 or Tier-2, and elective topics are listed. As per the guidelines,

all Tier-1 material should be covered and "all or almost all topics in the Tier-2 core" need to be taught as well.

Within the parallel and distributed computing knowledge area, 15 hours are considered Tier-1 or Tier-2. Within the Parallel Architecture grouping, "shared vs. distributed memory" is a Tier-1 topic, and "Instruction level support" is an elective topic. We argue that neither is receiving the attention that they deserve.

Work in 2013 at Purdue University [14] allowed the introduction of parallel computing concepts into computer engineering courses for undergraduates, and led to the development of a concept map for parallel computing. *Microprocessor and Assembly Programming* is identified as a potential course to include parallel computing concepts, but no further information on how or how much is offered.

More recent work at the University of Guelph [15] highlights the need to cover the entire breadth of the parallel computing, including topics related to hardware. Specifically, the authors underline the importance of understanding parallel memory models, multilevel cache design, parallel machine instructions, and hardware support for vector operations. Despite this, little explicit information on how to teach these concepts is offered; rather, the work goes on to discuss approaches for teaching parallel using high level message passing libraries.

Ultimately, researchers have identified a need to teaching parallel computing from a hardware standpoint, but we argue that this need does not receive the attention that it deserves. Consider the website CSinParallel.org [1]: Their assignments are commonly referenced by educators who teach parallel computing. While they house ten modules under the topic of "shared memory", all of these are high level, and their only module in "Computer Architecture" features GPU programming. Often, parallel assignments for students are integrated into introductory computer science, data structures, and/or operating systems courses [16], [17]. Texas State is one exception in this regard, as their efforts to teaching parallel computing modularly has resulted in parallel modules in their assembly course. These modules focus on architecture and performance, with a planned module for fundamentals [18]. The researchers volunteer no more information on the specifics of these particular modules. Despite this, there is still a lack of readily available assembly curricula, and adding such for hardware level parallelism fills an identified gap.

B. Teaching Parallel Modularly

The two most common approaches to teaching parallel computing to undergraduates are to integrate parallel concepts into existing courses or to offer an upper level capstone course on the subject. The former is widely preferred, or at least some synthesis thereof. To return to the model employed at Texas State, their approach emphasizes the advantages of early introduction followed by reinforcement of ideas that provide a solid foundation for a capstone class. Data collected on students show that this approach promotes both learning and interest. In part, motivation for this modular format is preferred because it is more feasible. As the researchers point out, adding an additional course to the curriculum is difficult for educators, especially the faculty member(s) responsible for the relevant curriculum. For students, the inconvenience (albeit smaller) lies in creating room in their schedules [18].

Introducing students to parallel computational thinking earlier in their careers can both promote more flexible approaches to problem solving, and lay a stronger foundation for later advanced coursework in parallelism [19]. The primary challenge lies in carefully selecting the right parallel exercises such that the added complexity does not threaten to overwhelm the students, or crowd out other essential concepts.

C. Embedded Xinu on the Raspberry Pi 3

Using the Embedded Xinu educational operating system is an attractive option. As stated earlier, Embedded Xinu has extensive uses in the classroom [20], [21]. As our focus is parallel programming in assembly, having full knowledge of the operating system is a great benefit. To fully demonstrate the nuances of multicore computing, we need to have the ability to set memory attributes and cache policies. Further, it helps to know exact details of process scheduling and resource management, which are all hidden by industrial grade operating systems.

The Raspberry Pi has already seen success in the classroom due to its simplicity of design and low cost to produce. In a 2018 conference publication [21], researchers highlight the potential of the Pi boards and contrast them to other forms of instruction. Though laptops, virtual machines, and remote servers are all listed as potential forms of instruction, the researchers show each to be inferior to the single-board computer design and recommend that each student has their own. At Marquette University, each student is not required to have their own Raspberry Pi. Instead, there are Pi's in a laboratory connected to the network—this gives nearly the same experience without requiring each student to purchase a Raspberry Pi and prepare it to load Embedded Xinu.

The default operating system for the Raspberry Pi family architecture is Raspbian[2], a Debian-derived Linux distribution specific to the Pi boards. Even though Linux is open source, the code is massive and not readily approachable; conversely, Embedded Xinu is designed to be understood by undergraduate students, and contains three orders of magnitude fewer lines of source code. Of course, another

[1] https://csinparallel.org/

[2] https://www.raspbian.org/

seemingly feasible alternative may be bare metal programming. Unfortunately, without a hardware debugger, students would have to write a serial port driver just to receive output in order to debug their programs. Further, some basic (but not obvious) setup needs to occur. To complicate matters, each core needs to be set up independent of the other cores.

III. The Concurrent ARM Playground

Embedded Xinu provides a useful compromise between bare metal computing on the Pi 3 boards and the Raspbian distribution of the Linux operating system. Embedded Xinu takes care of the necessary setup and provides access to the device's peripherals without putting restrictions in place for arbitrary code. Previous incarnations of the *MIPS Playground* and the *ARM Playground* have enabled students to run their RISC assembly programs directly on remote target platforms with a minimum of hassle. For this work, we have produced a concurrent variant of the ARM Playground environment, adding basic features to the underlying Embedded Xinu kernel to directly support genuine concurrent programming.

A. Setting Up The Cores

In order to make parallel computing in assembly accessible to students in a computer organization course, the Embedded Xinu operating system must allow students to run code on an emulated bare metal environment. For this reason, we removed some of the more complex conveniences from the operating system, including the serial port shell interface and the process scheduler. These modules are useful in a more advanced course, such as operating systems, but have little added value when the goal is to enable basic assembly-level programming. The core startup files of Embedded Xinu remain in place in order for students to understand the necessary steps for preparing the multicore assembly playground.

Like the original port of XINU to an embedded RISC device (namely the Linksys WRT54G), the Pi 3 board can boot remotely, so the kernel image containing student code can quickly be uploaded and then executed [5]. This eliminates the need to boot from the Pi's onboard SD card, which would be significantly more time-consuming for student users. (Nearly a factor of three longer kernel boot times.) Further, the network setup allows dozens of students to seamlessly share a bounded number of Pi boards.

Before student code is executed, the setup procedure initializes the interrupt vector table on core 0, the serial driver, and initial memory configuration. Once this initial setup is finished, the playground environment is executed, which saves the current state and prepares the other cores for execution. While ARM provides bare-metal boot code for its processors, details on how to start execution on other cores are more elusive.

Prior work [1] provided a framework to execute arbitrary code on the other cores. This is accomplished by writing an address to the specific core's *mailbox* – a hardware communication mechanism provided by the Broadcom implementation of this platform. Execution begins at that written address. In Xinu, execution always begins with the `setupCores()` function in order to properly initialize the other three cores. This setup is similar to setup done on core 0 when Xinu first launches, but the other cores delay their setup in order to remain idle until called upon. Once `setupCores()` is called, the core will switch the processor to system mode, initialize its stack pointer, then setup its memory (which is mostly invalidating relevant data and instruction caches). From here, the arguments passed are loaded, the link register is loaded with the address of an instruction that loops to itself, and the program counter jumps to the location indicated when called.

Inside the assembly playground file, `main.S`, other cores begin processing when core 0 calls `unparkcore()` (which itself calls `setupCores()` after checking for a valid core number). A pointer to arguments may optionally be passed. Finally, the instruction `bx lr` will return control to `main.c` to end the program when executed on core 0, and will send cores 1, 2, or 3 into an infinite loop if executed there. For this reason, the ARM *link register*, ("`lr`"), is written to in `setupCores()` and provides a simple and intuitive way to "park" a core. Because of this, `bx lr` can be safely called without checking the core number.

B. Student Programming Environment

Within the `main.S` assembly file, a user can quickly get code to execute on all four cores at once with each starting from the same instruction, or each core can begin at a different location in code. Each core has its own Multiprocessor Affinity Register which can report the ID of the specific core. The programmer also has access to built-in Playground `getnum()` and `printnum()` functions to allow for basic I/O and debugging via the serial port.

In this environment, the potential for race conditions exists for both shared memory and shared resources. Memory can easily be declared and available for all cores, but if precautions are not taken, simultaneous reads and writes may produce incorrect or undesired results. Having simultaneous writes to the serial output may cause multiple messages to attempt to be printed at once, and none may be readable.

The solution to race conditions is to make use of the *load exclusive* and *store exclusive* instructions that ARM provides on multicore processors. Each of these instructions ensures that it is the only core reading or writing to a specific memory address. This allows programmers to implement basic mutual exclusion objects (mutexes) to regulate memory access. While high level languages may still require programmers to implement mutexes, the details and specific machine instructions required are handled by the compiler.

Here, students can see exactly what the processor has to do to share resources, and therefore the cost of doing so is more obvious to the programmer.

C. Memory Model

Separate from the `main.S` file is `mmu_util.S`, the memory management unit (MMU) utilities file, which controls the properties of each chunk of memory. The default state is to mark the memory mapped peripherals as device memory, and to mark everything else as normal, shareable memory. That said, these properties are easy to adjust, and ARM provides extensive documentation on the translation table's short-descriptor section formats beginning in section G5.4.1 of the ARM Architecture Reference Manual ARMv8 [22]. Namely, the type extension, bufferable, and cacheable bit fields provide numerous options for determining the amount of sharing and cache policies. Even if taking correct measures in ensuring load and store exclusive instructions are used, they will fail if the user selects the wrong memory policies. Another potential issue that can arise is if the peripherals are marked as cacheable. This would inevitably cause the serial port (amongst other devices) to fail, corrupting all I/O with the device.

The ARM Architecture Reference Manual (version *da*) for ARMv8-A describes how to set properties of specific sections of memory. In Figure G4-4, the section short-descriptor translation table format shows the bit fields that correspond to various memory properties. The S bit and the type extension bits are the most relevant for our purposes. We need to ensure that the memory used by the programs we write is properly marked as shareable. In our case, we want to set the memory to be normal, inner cacheable with write-back with write-allocate. The difficult part in this is that as noted in section G4.7.2, some of the names of the bit fields are inherited, so their labels do not necessarily accurately reflect their function. After the translation table is initialize, we can make use of exclusive access instructions, namely `ldrex` and `strex`.

In regards to the distributed memory model, the results are computed by using ARM *vector instructions*. To start, we have to use `VLD` to load the vectors into the special vector and floating point registers. Even though we are using the 32-bit memory model, each of these registers is 64 bits and has designated subdivisions that can access half of a register at a time, for example. Once loaded, we could combine the results from each of the four cores using `VADD`. Using these vector instructions demonstrates parallel computing by use of specially designed assembly instructions.

IV. Pilot Study

The Concurrent ARM Playground version of Embedded Xinu on the Raspberry Pi 3 has many potential uses. At Marquette University, we fielded an initial pilot study in the Fall 2018 semester with students in the COSC 2200 "Hardware Systems" course. To dovetail with existing assignments, we tasked students midway through the course with implementing the *coupon collector's problem* for $n = 8$. This allows students to experience the pseudorandom number generator (PRNG) implemented by Embedded Xinu, and this ties in well with an existing assignment that deals with encryption.

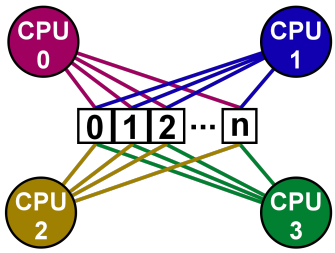

Figure 1. Visualization of a shared memory model with four cores

The *coupon collector's problem* was selected because of its potential for frequent race conditions. This is a classic urn problem that tasks someone with randomly drawing coupons one at a time, with replacement, from an urn containing n coupons. This is repeated until all n coupons have been drawn at least once, and the number of attempts is recorded. Many trials are then conducted. Using a shared memory model may not be optimal, but it lets us demonstrate to students many potential problems without assigning an overly difficult problem. (If students get caught up on the logic and have trouble producing working code, then lessons on parallelism at the hardware level are lost on them).

We start by initializing an array of about 150 items to hold the frequencies generated from each trial. Each time a trial completes, the value at the index corresponding to the number of coupons required is incremented by one. Since we choose 8 coupons to collect (as modular arithmetic in binary is easier with powers of two), indices 0 through 7 should remain at zero if the code executes correctly. Now, unless mutexes (or other safeguards against race conditions) are implemented, not all trials will be recorded if a sufficient number are ran across all four cores simultaneously. Implementing these correctly requires some precision, but the Arm Compiler (v6.10) User Guide provides example code [23].

Once the student has code has been written and tested, the memory model can be tweaked so changes in output can be observed. The relevant value to adjust is the section memory descriptor (described in section 3.3). The default is the same as the example value provided in the ARM Bare-metal Boot Code application note, which is `0xXXX15C06` [24] (the first 12 bits determine the mapping from the virtual to the physical address space and are not related to the shareability and other attributes). Changing this to `0xXXX00C0E` will make about 20% of the writes to memory lost, and changing the value to `0xXXX00C0A` will make about 0.02% of the

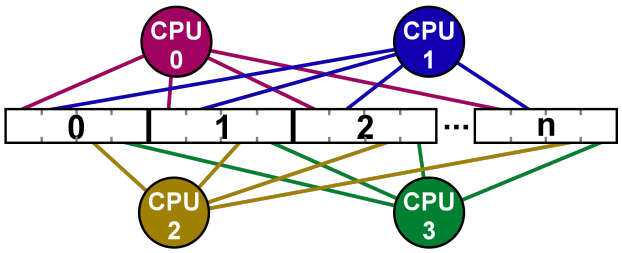

Figure 2. Visualization of a distributed memory model with four cores

values lost when running a sufficient number of trials. The exact details of what is being altered is described in the ARM Architecture Reference Manual ARMv8 starting in section G5.7.1 [22].

Another second exercise to do after writing working code is to compare the current program with a program that implements a distributed memory model. Again, the effects of doing so are witnessed first hand as four times the amount of memory is used, but so are fewer mutexes. Students should see that the speedup vastly outweighs the extra space cost (and may wonder why they were asked to implement a shared memory model in the first place). Basic visualizations of each model are provided in Figure 1 and Figure 2.

V. Evaluation

We introduced the 43 students of the Fall 2018 COSC 2200 Hardware Systems class at Marquette University to a multicore assignment using the ARM architecture and assembly language. COSC 2200 is a sophomore level class and is the first course in the major that students typically register for after one or two freshman programming courses. We have recently updated the second freshman programming course to include the topic of concurrency and synchronization in Java. As a result, 24 students that participated in our study had written a few concurrent programs in Java the prior semester.

A. Assignment Description

The objectives of our pilot study assignment were for students to:

1) realize the benefit of concurrency by solving a simple, but interesting problem;
2) explore a given design and identify at a high level the purpose of each core in solving the problem;
3) recognize how the cores' tasks intermingle with each other in a shared memory model;
4) implement the low level details for synchronization in assembly given functions for acquiring and releasing a mutex.

Section V-B describes an additional objective that can be achieved through this assignment, although not part of the original goals.

In this assignment, we asked students to use all four cores of the Raspberry Pi 3 and to run a total of ten million experiments for the coupon collector's problem. Each core should run the same number of experiments (2.5 million). The students' code should output the values of the array that stores the number of trials necessary to draw all eight coupons for each experiment. We called this array `counts`. If students followed these instructions, then the screen output should look like a vertical histogram. To help students test their code, we also asked them to print the sum of the values stored in `counts`, which would need to be equal to the total number of experiments (10 million).

We provided students with the assembly functions for acquiring and releasing a mutex. We also provided them with the code allocating four critical variables:

- `counts`, the array that contains the results;
- `locks`, an array used to implement synchronized access to `counts`;
- `proccnt`, the number of active processes at any given time;
- `prcntlk`, the lock used to implement synchronized access to `proccnt`.

Students, grouped in teams of two, had to address the following tasks:

- Start cores 1, 2, and 3 by calling `unparkcore()` and have each of the four cores execute the same function, which is called `setup()`. Note that students do not need to interact with the `setupCores()` function described in Section III-A, so there is no naming confusion. Students also need to update the value of `proccnt`.
- Implement the `setup()` function. The function, which is supposed to be run by each of the four cores, performs a quarter of the total experiments. After one experiment is completed, the function updates the shared `counts` array using the corresponding `locks` array position as a mutex. Once a core completes all assigned experiments, the core decreases the value of `proccnt` using `prcntlk` as a mutex. We provided students with the code that uses the `mrc` instruction and accesses the Multiprocessor Affinity Register (MPIDR) to identify the core that is about to exit the `setup` function.
- Print the results. After ensuring that other cores have also finished their portion of the experiments, core 0 accesses the *counts* array and prints the results on standard output.

B. Study Outcomes

As already mentioned, several students had written concurrent programs in a high level language in the previous

semester. However, we still found it worth emphasizing the special role of core 0. Students appeared to need the reminder that the program is run by core 0, which does not finish execution after running the `setup()` function unlike the other cores. This emphasis was critical in students achieving Objective 2.

An unanticipated side benefit is that implementing the coupon collector's problem in assembly is not only appropriate for demonstrating multicore principles at the hardware level, but also ideal for students to appreciate the efficiency and simplicity that logical instructions offer when performed at the bit level. In a previous ARM assembly assignment, students learned how to declare arrays dynamically in the heap and fill them with values. In this assignment, most students initially thought of declaring an integer array of eight bytes to determine whether they have drawn all eight coupons at a given trial. Getting introduced to logical ARM assembly instructions and how they can operate individually on the bits of registers was welcomed by students for two reasons: a) they experienced first hand that assembly coding became simpler, since they did not need to allocate and keep updating an array, and b) they realized that bit operators resulted in more efficient space use, since they needed to keep track of eight bits of a register instead of eight integer sized memory locations. Therefore, the assignment objectives can include performing bit level operations that result in less complex and more efficient assembly code.

An equally unanticipated, but less welcome, result was that students found an easy way to circumvent the need of a mutex in some cases. After finishing the execution of `setup()`, core 0 is supposed to verify that the other cores have also finished their experiments before accessing the `counts` array and printing the results. Students were expected to have core 0 repeatedly check the value of `proccnt` until there is only one core running, itself. This is the reason that students needed to initialize the `proccnt` variable in the beginning and then update it at the end of the `setup()` function.

Although most students did perform these two tasks (i.e. initialization and update), many did not use `proccnt` as a tool for core 0 to check how many cores are still running and seemed to have missed the purpose of this variable. Instead, they had core 0 wait for a small amount of time and then access the `counts` array. This small delay after core 0 had completed the experiments was enough for all other cores to also finish in our scenario, where every core executes the same number of trials. As a result, the output of code that only delayed without checking `proccnt` was still correct.

One way to address this concern in the future, other than emphasizing even more the purpose of the `proccnt` variable, is to assign only a small number of experiments to core 0 as opposed to the same number as any other core. The `setup()` function should then accept an argument specifying the number of experiments the core running `setup()` should execute. The `unparkcore()` function does support core functions with arguments as described in Section III-A.

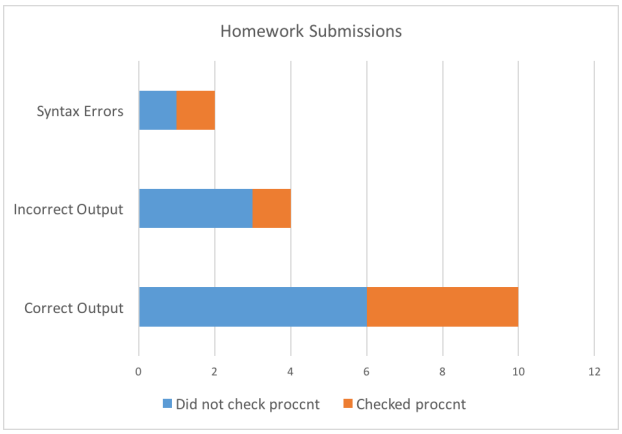

Figure 3. Student submissions that produced the right output, produced wrong output or did not compile due to syntax errors.

Overall, we found the results of the pilot study to be encouraging. We received 19 homework submissions, while 2 teams (4 students) did not make a submission. Students had one week to complete the assignment, which was due the last day of class in the semester. We dismissed 3 submissions that were either made a few hours after the deadline or were related to academic integrity violations. Out of the 16 valid homework submissions, 10 produced the correct output, while 3 more were close. In these 13 submissions, we found that students correctly implemented exclusive access to the `count` array for updates by each core. As mentioned already, several submissions seemed to miss the purpose of the `proccnt` variable, since it was not used to check whether core 0 is the only core running before printing the results, although students updated `proccnt` correctly. Figure 3 shows the number of submissions that checked the value of `proccnt` in all 16 submissions.

VI. Conclusion

We believe that the addition of this and similar curriculum modules can strengthen a student's parallel computational thinking. When students take upper level courses in or involving parallel computing, they will benefit from having experience in programming in parallel in assembly. High level design decisions will be better informed by observations made after programming in a low level. Concepts like shared versus distributed memory will feel less abstract, and when using tools such as OpenMP or MPI, students will have a better grasp on what happens "under the hood".

Using Embedded Xinu and the Concurrent ARM Playground for the Raspberry Pi 3 allows students to dive straight into genuine concurrent programming at a low level, while

having the option to later study what goes on to make the code run on the Pi boards. These environments give students opportunities to read technical manuals and documentation to solve their problems. Finally, there is plenty of room for exploration beyond the assignment as there are many opportunities to go beyond what is required.

VII. Future Work

The next course in the sequence for our group of pilot students is Operating Systems, which at Marquette entails completing major components of Embedded Xinu. In Spring 2019, we plan to pilot a new group of concurrency-focused assignments in this course that build on the students' prior experience with both concurrent Java programming, and concurrent ARM assembly.

This is only one of many ways to do parallel programming on the Pi 3 boards in both C and assembly language. This processor has the capability of what ARM calls NEON-Advanced SIMD, which allows for vector operations acting on 128-bit registers. Making use of these instructions would further increase the parallel capability of the processor.

Parallel computational thinking is likely to only grow more important for computing-centric undergraduates in the foreseeable future. While getting parallel computing assignments and courses into every university may be the desired goal, fleshing out these rigorous yet flexible curriculum modules for a wide variety of courses is a necessary precondition.

Acknowledgment

This material is based upon work supported by the National Science Foundation REU Site Grant #ACI-1461264 "Computation Across the Disciplines" at Marquette University. The authors would also like to thank all the contributors to the Embedded Xinu project across the years.

References

[1] P. Bansal, R. Latinovich, T. Lazar, P. J. McGee, and D. Brylow, "XinuPi3: Teaching multicore concepts using embedded xinu," in *Proceedings of the 6th Computer Science Education Research Conference*, ser. CSERC '17. New York, NY, USA: ACM, 2017, pp. 20–25. [Online]. Available: http://doi.acm.org/10.1145/3162087.3162091

[2] E. Biggers, F. Harunani, Tyler, and D. Brylow, "Xinupi : Porting a lightweight educational operating system to the raspberry pi," in *Proceedings of WESE 2013: Workshop on Embedded Systems Education*, 2013.

[3] M. Ziwisky, K. Persohn, and D. Brylow, "A down-to-earth educational operating system for up-in-the-cloud many-core architectures," *Trans. Comput. Educ.*, vol. 13, no. 1, pp. 4:1–4:12, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2414446.2414450

[4] D. Comer, *Operating system design: the Xinu approach*, 2nd ed. CRC Press/Taylor & Francis Group, 2015.

[5] D. Brylow, "An experimental laboratory environment for teaching embedded operating systems," *SIGCSE Bull.*, vol. 40, no. 1, pp. 192–196, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1352322.1352201

[6] D. Brylow and K. Thurow, "Hands-on networking labs with embedded routers," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 399–404. [Online]. Available: http://doi.acm.org/10.1145/1953163.1953283

[7] A. B. Mallen and D. Brylow, "Compiler construction with a dash of concurrency and an embedded twist," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 161–168. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869568

[8] P. Ruth and D. Brylow, "An experimental nexos laboratory using virtual xinu," in *2011 Frontiers in Education Conference (FIE)*, Oct 2011, pp. S2E–1–S2E–6. [Online]. Available: https://doi.org/10.1109/FIE.2011.6143069

[9] M. H. Netkow and D. Brylow, "Xest: An automated framework for regression testing of embedded software," in *Proceedings of the 2010 Workshop on Embedded Systems Education*, ser. WESE '10. New York, NY, USA: ACM, 2010, pp. 7:1–7:8. [Online]. Available: http://doi.acm.org/10.1145/1930277.1930284

[10] Raspberry Pi, "BCM2837 - raspberry pi documentation," 2012. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837

[11] Qualcomm, "Snapdragon 410 processor." [Online]. Available: https://www.qualcomm.com/products/snapdragon/processors/410

[12] PCMag, "Nintendo switch uses a standard tegra x1 processor," 2017. [Online]. Available: https://www.pcmag.com/news/352485/nintendo-switch-uses-a-standard-tegra-x1-processor

[13] D. J. John and S. J. Thomas, "Parallel and distributed computing across the computer science curriculum," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 1085–1090.

[14] C. M. Brown, Y.-H. Lu, and S. Midkiff, "Introducing parallel programming in undergraduate curriculum," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1269–1274. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.270

[15] W. B. Gardner, "Should we be teaching parallel programming?" in *Proceedings of the 22Nd Western Canadian Conference on Computing Education*, ser. WCCCE '17. New York, NY, USA: ACM, 2017, pp. 3:1–3:7. [Online]. Available: http://doi.acm.org/10.1145/3085585.3085588

[16] D. Ernst and D. E. Stevenson, "Concurrent CS: preparing students for a multicore world," pp. 230–234, 06 2008.

[17] J. R. Graham, "Integrating parallel programming techniques into traditional computer science curricula," *SIGCSE Bull.*, vol. 39, no. 4, pp. 75–78, Dec. 2007. [Online]. Available: http://doi.acm.org/10.1145/1345375.1345419

[18] M. Burtscher, W. Peng, A. Qasem, H. Shi, D. Tamir, and H. Thiry, "A module-based approach to adopting the 2013 acm curricular recommendations on parallel computing," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 36–41. [Online]. Available: http://doi.acm.org/10.1145/2676723.2677270

[19] K. Kirkpatrick, "Parallel computational thinking," *Commun. ACM*, vol. 60, no. 12, pp. 17–19, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3148760

[20] P. Turton and T. F. Turton, "PiBrain - a cost-effective supercomputer for educational use," in *5th Brunei International Conference on Engineering and Technology (BICET 2014)*, Nov 2014, pp. 1–4.

[21] S. J. Matthews, J. C. Adams, R. A. Brown, and E. Shoop, "Portable parallel computing with the raspberry pi," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: ACM, 2018, pp. 92–97. [Online]. Available: http://doi.acm.org/10.1145/3159450.3159558

[22] ARM, "ARM architecture reference manual ARMv8, for ARMv8-architecture profile," 2017.

[23] ——, "ARM compiler version 6.10 user guide," 2017.

[24] ——, "Bare-metal boot code for ARMv8-A processors," 2017.