Mastery-Based Learning in Undergraduate Computer Architecture

Ellen Spertus
Mills College
Oakland, California, USA
ellen.spertus@gmail.com

Zachary Kurmas
Grand Valley State University
Allendale, Michigan, USA
kurmasz@gvsu.edu

ABSTRACT

Mastery-based learning is an approach in which students are graded based on their demonstrated mastery of explicit learning outcomes rather than by being awarded points less directly connected to course goals. Instead of submitting an assignment only once, students can use feedback to improve their work to increase their learning and grades. We describe our approach to mastery-based grading in introductory computer organization/architecture courses at two different institutions. Specifically, we allowed students to retake tests of basic skills as many times as needed, which was facilitated by programmatically-generated questions. For course projects, students were expected to refine and resubmit projects until they demonstrated mastery by passing all of the provided automated tests. We share our materials and experiences, including the challenge of loosening deadlines to provide students time for continued work without enabling them to fall irretrievably behind.

CCS CONCEPTS

• **Applied computing** → **Interactive learning environments**; *Computer-assisted instruction*; • **Social and professional topics** → **Computer science education**; **Student assessment**; • **Computer systems organization** → *Architectures.*

KEYWORDS

computer architecture education, grading, mastery-based grading, student assessment, DLUnit, MUnit

ACM Reference Format:
Ellen Spertus and Zachary Kurmas. 2021. Mastery-Based Learning in Undergraduate Computer Architecture. In *Proceedings of Workshop on Computer Architecture Education 2021 (WCAE '21).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We would like for our students to approach learning with a "growth mindset," where assignments, tests, and feedback are viewed as positive opportunities to increase one's knowledge rather than as unpleasant, arbitrary, or punitive practices. We especially want students to view course work primarily as a learning opportunity (where mistakes lead to better understanding) rather than as an assessment (where mistakes hurt the student's course grade). To this end, we have adopted *mastery-based grading*, which has been described as "a complete paradigm shift for most teachers. It means thinking about grading as a way to provide feedback, and not a random act that we do because the quarter is ending" [3].

Principles of mastery-based learning include [2, 10]:

- Providing students with learning goals and the objective criteria for measuring them.
- Grading students based on these objective criteria and not on soft skills, such as attendance, effort, or class participation.
- Giving students multiple opportunities to demonstrate mastery, rather than grading them on their initial attempt.

These practices can place a burden on teachers by requiring them to create multiple versions of exercises and grade students' work multiple times — as many times as it takes for a student to achieve the learning outcomes. We present and share tools that we built to manage this load.

We describe how we applied mastery-based learning principles in computer organization/architecture courses at two different institutions. Ellen Spertus used automated problem generation and grading for exercises at Mills College, enabling students to retake quizzes until they demonstrated mastery of the material. At Grand Valley State University, Zachary Kurmas used GitHub Classroom and GitHub Actions to automatically verify the correctness of student submissions. Submissions were not graded until they passed all of the automated tests, thereby providing students an incentive to learn the material by fixing their mistakes. Both professors share their resources and experiences so others can adopt and adapt them.

2 MILLS COLLEGE

Mills College is a minority-serving women's college in Oakland, California. After teaching Computer Architecture for over 20 years using conventional grading techniques, Spertus made a radical shift in her Fall 2020 Computer Architecture class, as described on the syllabus:

> Grading will be solely on demonstrated mastery of the material, based on the principles in *Grading for Equity* by Joe Feldman, as I understand them:
>
> - Students should not be graded on soft skills or subjective behaviors, such as class participation, timeliness, and attendance.
> - Students should not be penalized for making mistakes while learning, so final grades should not include formative assessments (homework), only summative assessments (lab assignments and tests).

WCAE '21, June 17–19, 2021, Valencia, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- What matters is whether students learn material, not when they learn it, so students will be able to redo test portions.

Students will be required to complete homework assignments to advance to tests and later material, but homework grades will not count directly toward the semester grade. Of course, much of the learning will take place when doing the homework or receiving feedback on it, so it is the best way to prepare for tests.

The remainder of this section will describe how these idealistic policies worked out.

2.1 Learning Outcomes

There were 24 learning outcomes divided among 6 groups:

- Digital logic (5)
- Computer arithmetic (6)
- Assembly language (1)
- Processors (6)
- Verilog (2)
- Memory (4)

Learning outcomes were judged on an integer scale from 0–4, as advocated by Joe Feldman in *Grading for Equity*. Feldman argues that the traditional percentage scale is flawed because [4, 5]:

- 60% of the scale is devoted to different levels of failure.
- A single missed assignment (0%) can have a devastating effect on the semester grade and students' subsequent motivation.
- Research has shown that "when teachers used the 0–100 scale to score student work, there was enormously wide variance from teacher to teacher"[4, p. 79].

In *Specifications Grading*, Linda B. Nilson observes [9]:

> Point totals do not easily map on outcomes. If a student earns 72 or 80 or 88 out of 100 points, that number does not indicate what she can and cannot do at the end of the course.

Feldman's 0–4 scale is shown in Table 1 applied to a computer architecture learning outcome, writing working assembly code. While different teachers might choose different criteria, the scale has the virtue of being objective. Clear specification of grading criteria shuts down arguments from students that they deserve more points.

Automated grading was used for 8 of the learning outcomes, shown with their criteria in Table 2. The first column indicates a learning outcome, the second column the criteria for full credit, the third column the criteria for $\frac{3}{4}$ credit, and the final column the maximum number of tries attempted by any student. Numbers in parentheses indicate how many students (out of 6) eventually earned that score. The criteria are discussed further below.

The Canvas learning management system (LMS) lets instructors specify learning outcomes with criteria for different scores. These can be inserted into rubrics on quizzes that draw questions at random from a question bank. Further details, including importable learning outcomes, question-generating scripts, and sample questions, can be found in the author's GitHub repository [12].

2.2 Ordering Problems

While many of the Computer Arithmetic outcomes (Table 2) are straightforward, the ordering problems are worth discussion. Here is a sample problem:

> The following numbers are in two's complement notation. Please order them from lowest (most negative) to highest (most positive). For example, if they were already correctly ordered, your answer would be ABCD.
>
> A = 00111101
> B = 11100000
> C = 10110101
> D = 00101100
>
> For full credit, answer the problem without converting all of the numbers to decimal.

In addition to providing an ordering (e.g., "CBDA") that could be graded automatically, students were required to provide an explanation, such as:

> B and C must be negative because their sign bit is 1. Because the next bit (representing +64) is 1 in B but not C, B must have a higher value than C, therefore C < B. A and D are both positive because their sign bits are 0. Because the next three bits are 011 and 010 respectively, A > D. Therefore, the final order is C < B < D < A.

While theoretically the teacher need not take the time to manually grade the explanation unless the student got the ordering correct, providing feedback on an incorrect explanation was very helpful to students. Another benefit of requiring an explanation was that it made it harder to cheat. While there was no way to stop a student from checking the ordering with a calculator, generating an explanation required understanding the material.

Table 1: Criteria for grading the ability to write assembly code

Grade)	Criterion
4 (A)	Can implement non-leaf procedures with loops or other complex control flow
3 (B)	Can implement non-leaf procedure without complex control flow
2 (C)	Can implement simple leaf procedures
1 (D)	Not used
0 (F)	Cannot write working code

Table 2: Grading Criteria for Learning Outcomes

Group/Outcome	4 (A)	3 (B)	max tries
Computer Arithmetic			
Converting among bases 2, 8, 10, and 16	no errors (6)	minor error	1
Converting between decimal and 2C	no errors (5)	minor error (1)	4
Ordering 2C numbers	without conversion (6)	with conversion	3
Converting between decimal and FP	no errors (6)	minor error	4
Ordering FP numbers	without conversion (6)	with conversion	3
Choosing the best representation	most efficient (4)	acceptable (1)	3
Processors			
Encoding MIPS instructions	correct (5)	minor error (1)	3
Memory hierarchy			
Computing average memory access time	correct (5)	—	7

2.3 Choosing a Representation

The final computer arithmetic learning objective was for students to be able to choose the best (most efficient and sufficiently accurate) numeric representation for a given scenario. While the problems could be graded automatically, they were created manually and there were only 6 versions. Here is an example:

> You are working with a team of scientists trying to keep track of the mass, position, and velocity of every artificial and natural satellite in earth's orbit, including space junk (abandoned equipment) and asteroids. It is vital to model this information as accurately as possible. The mass of satellites is recorded in integer multiples of grams. The largest satellite is the moon, with an estimated mass of about 7.342×10^{22} kilograms. You are asked to evaluate the suitability of different representations of the mass of each satellite.

The best answer to this question is `BigInteger` (4 points), and an acceptable answer is the less efficient `BigDecimal` (3 points). The other variants of this question are available to instructors on request.

The author considered this the most important of the outcomes, and weighted it more heavily, because it was the only one that required higher-level understanding of the material and the most likely to be applicable in the students' subsequent careers.

Because the set of possible answers was limited (`short`, `int`, `long`, `float`, `double`, `BigInteger`, and `BigDecimal`) and students were allowed to retake the quiz without penalty, it was important to make sure students could not get the answer by guessing. Consequently, students were required to explain their choice. It was also hoped that requiring an explanation would make students more likely to make the right choice, although there were not enough students in the class to test this hypothesis.

An unexpected implementation problem occurred when, after failing the quiz with two different questions, the next two versions of the quiz generated at random by Canvas repeated the same questions rather than selecting one of the other four. The instructor needed to manually create a quiz with an unselected question for the student. This was not a problem with any of the other learning outcomes because there were at least 20 automatically-generated versions of each question in the bank (although the instructor never checked whether there were repeats for individual students). It would be better if the LMS avoided repeating questions when letting a student retake a quiz.

2.4 Processors: Instruction Encoding

It was harder to generate questions related to processors, so the only learning outcome with automated questions was the ability to encode MIPS instructions (e.g., "`add $r10, $r8, $r6`") into binary. Because the quizzes were all open-book and the instructor wanted to be helpful to students, an encoding reference sheet was included. All students eventually received full credit for this question, with the exception of one who filled the `shamt` portion of the instruction with 4 0s instead of 5 0s and chose not to redo the quiz to raise their score.

2.5 Memory hierarchy: AMAT Calculation

Similarly, there was only one automated learning outcome involving the memory hierarchy: calculating the average memory access time of a system. Here is a sample problem (with randomly set values underlined):

> A processor has 2 levels of caches. The L1 cache has an access time of 1 cycle and a hit rate of 93%. The L2 cache has an access time of 4 cycles and a cumulative hit rate of 99%. Main memory accesses take 73 cycles. All of the access times are concurrent (done in parallel). What is the average memory access time in cycles?

While all of the students eventually got a version of this problem correct, there were more retries than for any other outcome. One student took and failed the quiz 5 times in succession one night before meeting with the instructor to discuss what they were doing wrong, as no feedback had been given besides whether the final answer was correct or incorrect.

We have since realized that giving students different instantiations of this same template may not let them effectively demonstrate that they have mastered the learning outcome because once a student is shown how to solve one instance of the problem, they could apply the same formula to the values in the next version of the

Table 3: Survey respondents' level of agreement with statements about course policies.

	Strongly agree	Agree	Neutral	Disagree	Strongly disagree
Being able to redo quizzes...					
...reduced my stress level.	5	3	1	0	0
...caused me to get a better grade.	6	4	0	0	0
...caused me to learn more.	6	3	1	0	0
...penalizes students with many demands on their time.	0	0	1	5	4
I was able to do well on retakes without understanding the material.	0	0	1	6	3
You should keep this policy.	4	6	0	0	0
The lack of due dates...					
...lowered my stress level.	6	1	1	1	1
...gave me flexibility when I needed it.	5	2	1	2	0
...caused me to prioritize other classes over this one.	3	4	2	1	0
...led me to procrastinate.	2	3	2	2	1
...led me to spend more time on assignments.	4	2	3	1	0
...led to my having to withdraw from the course.	1	0	1	4	4
I liked having explicit rubrics (grading guides) for assignments.	5	2	2	0	1
I liked that the rubrics were based on achievement of learning outcomes.	4	3	1	1	1
I understood the grading system.	5	1	1	2	1
It was hard to know how I was doing in class.	1	3	0	4	2

question without necessarily understanding the underlying principles. In the future, we would like to have more templates for AMAT calculations, varying the number of levels and whether access times are concurrent or sequential.

2.6 Discussion

The instructor chose to give a follow-up survey to better understand how the policies affected students, especially because the course had high attrition. Ten of the thirteen students who attempted the course completed a detailed survey about the course, for which they each received a $15 gift card.

2.6.1 Redo Policy. As shown in Table 3, students indicated that the ability to redo quizzes reduced their stress and caused them to learn more and earn better grades. One student wrote:

> [B]eing able to correct work and retake quizzes helped me to learn the material more deeply. Rather than just seeing I did something wrong and not being able to do anything about it, I had the opportunity to go back and understand the material that I hadn't fully grasped.

Students did not feel they were able to do well on retakes without understanding the material. They thought that the same redo policy should be used in future years.

2.6.2 Due dates. By the second week of class, the instructor was concerned about her decision to eliminate late penalties. Most of the students immediately fell behind, some irretrievably so, although they continued to insist they could catch up when the instructor raised concerns. If the withdraw deadline had not been moved to the last day of class because of the pandemic, a record number of students would have failed the course. Instead, there was record attrition. Of the 13 students who began the course, 7 withdrew, 5 completed the course by the end of the semester, and 1 took an Incomplete that was satisfied before the start of the next semester.

Students' opinions are shown in the second part of Table 3. Most, but not all, students said the absence of late penalties lowered their stress, gave them needed flexibility, caused them to prioritize other classes, and led them to procrastinate. The 6 students who strongly agreed (4) or agreed (2) that it caused them to spend more time on assignments were asked whether the extra time spent was productive. Their responses were:

- Yes, I learned the material better. (3)
- Yes because I needed the additional time to work through problems.^(1)
- Most of the time.^(1)
- No, it was unproductive perfectionism. (1)

The responses followed by a caret were written in; the other responses were selected with check boxes.

While the instructor thought the policy played a major role in students' not completing the course, only one student (strongly) agreed. An optional survey question revealed that many students were struggling with stress, mental illness (including anxiety and depression), physical illness (of themselves or a loved one), a need to earn money, and a lack of a good place to work, Internet connectivity, or a reliable computer, caused by their moving back home for the pandemic. More information about the course and the follow-up survey appear in a separate paper [13].

2.6.3 Learning outcomes. In hindsight, it was not necessary to create as many different grading criteria (levels) for each learning outcome. Because the most common scores were either full credit or no credit, grading time and complexity could have been reduced by taking an all-or-nothing specifications grading approach [9].

Additionally, it is unclear whether students deserve any college-level credit for incorrectly converting between bases, for example, particularly when they get multiple tries.

The policy of not counting homework toward the final grade worked surprisingly well. Students were willing to complete homework assignments even though there was no explicit penalty for not doing so. The instructor did not detect any decline in homework completion or quality, and she was able to spend less time grading homework, focusing on giving students feedback rather than assigning points.

As shown in Table 3, most students liked having explicit rubrics based on grading outcomes, but some did not understand the grading system, and most found it hard to know how they were doing in the class. This is a clear area for improvement.

3 GRAND VALLEY STATE UNIVERSITY

Zachary Kurmas at Grand Valley State University takes a mastery-based approach to grading projects in the undergraduate 300-level Computer Organization course: Student projects and assembly language programs don't receive a grade until they pass all of the automated tests.[1] With a traditional grading model, many students simply "submit it and forget it." By the time the graded projects are returned, these students have moved on to other topics and mentally "written off" the lost points. We have not conducted a formal study, but our experience suggests that mandating that students fix their mistakes provides the motivation needed to seek help and address the misconceptions that led to those mistakes in the first place [6, 7].

The course currently assigns four projects and four assembly language assignments. The four projects are:

(1) Build a signed adder with overflow detection
(2) Build a subtractor using the adder from Project 1; and build a circuit that compares both signed and unsigned integers
(3) Build a simple ALU containing the circuits from Projects 1 and 2, as well as other basic operations (`and`, `or`, `not`, etc.)
(4) Implement the single-cycle CPU presented in the Patterson and Hennessy text

When we introduced these projects and assignments, the scores were quite low. Over the course of several years, we attempted to address these low scores by clarifying and expanding the instructions, as well as by allocating additional time during lecture to discus the requirements and possible strategies. These changes had almost no effect on scores. Furthermore, we noticed that, overall, students scored reasonably well on related exam questions. It turned out that the main cause of low scores was not poor high-level understanding, but poor testing: The students were not thoroughly testing their circuits and code before submitting them and, therefore, did not realize that they had made low-level mistakes.

3.1 Encouraging Mastery while Reinforcing Testing

The main reason many students were failing to master the project's low-level details was because they weren't made aware of their deficiencies in a timely manner (i.e., before the course had moved on to other topics). In order to provide sufficiently rapid feedback, we needed to automate the testing of projects. However, rather than simply providing the students with a complete set of tests, we decided to also help students improve their own testing abilities.

Like programming, testing is a skill that must be developed over time through experience. Few (if any) students become good testers after completing one or two units in a college course. To help students gain valuable testing experience, we decided to introduce into this course a workflow motivated by Test Driven Development (TDD). For these assignments, we ask students to prepare a suite of tests *before* implementing any circuits or writing any code. When the students' code/circuit passes their tests, they submit their work to an automated test system (currently a GitHub Action), which returns a terse pass/fail message. If the submission fails, students need to review their test cases, find an omission, then write their own failing test case. We expect that this process of finding missing test cases will help them write more complete test suites in the future. We also expect that the process of seeing the consequences of their missing tests will help them retain the lesson better than if we were to simply critique their original test cases and provide a list of missing tests.

We believe that the common approach of simply providing students a complete set of tests (either an "open" set of tests that the students can view, or a "blind" set of tests that returns only a pass or fail message) encourages them to focus on getting the tests to pass rather than on the underlying assignment objectives. We hypothesize that requiring students to write their own tests and think critically about the shortcomings of those tests will help redirect their focus back to the project objectives. Writing thorough tests requires a thorough understanding of the problem being addressed. (One of the motivations for Test Driven Development is that the process of writing tests encourages developers to slow down and identify aspects of the problem that are either unclear or under-specified.) We are optimistic that our TDD-motivated workflow will have a similar effect by helping students better recognize which aspects of the project they need to understand better before simply "diving in" and hoping for the best.

It is not clear the extent to which our process has improved students' testing abilities. We have not conducted any formal studies, but, anecdotally, students still continue to make the same number and types of testing mistakes throughout the semester. (In other words, they don't seem to test Project 4 any better than Project 1.) We suspect that, upon receiving a failing report from the automated tester, most students immediately scan their code for bugs rather than scan their tests for oversights. (This suspicion is raised by the number of students who ask for help with their assembly language code even though they have no failing tests of their own.) Because this course is but one small step in a the long process of learning to test well, we don't expect to see a clear, measurable effect until more courses in our curriculum adopt this process.

Although the testing aspect of our process is still a work in progress, there was an immediate improvement in student mastery: Students were forced to acknowledge when they didn't understand an aspect of an assignment. They could no longer simply submit

[1] Of course, exceptions can be made when appropriate.

```
import static edu.gvsu.dlunit.*;

// Simple tests to help students get started
public class UAdderTest {

 @Test
 public void zero_zero_false() {
   setPinUnsigned("InputA", 0);
   setPinUnsigned("InputB", 0);
   setPin("CarryIn", false);
   run();
   assertEquals("Output", 0, readPinUnsigned("Output"));
   assertFalse("Carry Out", readPin("CarryOut"));
 }

 @Test
 public void zero_one_false() {
   setPinUnsigned("InputA", 0);
   setPinUnsigned("InputB", 1);
   setPin("CarryIn", false);
   run();
   assertEquals("Output", 1, readPinUnsigned("Output"));
   assertFalse("Carry Out", readPin("CarryOut"));
 }

 // Place similar tests here....
}
```

Figure 1: Example `DLUnit` test suite

an assignment without assuring that it worked.[2] As a result, assignment grades and engagement have increased significantly. For most assignments, timeliness accounts for 20% of the overall grade. Consequently, the only students who earn less than 80% on an assignment are those who give up completely and never finish it. Most assignment grades are above 90%.[3] Similarly, we have noticed a considerable increase in visits to office hours. Sometimes that engagement comes later than we'd like (i.e., after the due date), but the vast majority of students who need help eventually ask for it.

3.2 Automated Testing Infrastructure

The infrastructure for our projects and assignments requires two key components: (1) Automated testing tools for digital logic circuits and assembly language (where students can easily run their own tests), and (2) a framework with which students can run our tests without being able to see them.

3.2.1 Test Utilities. We created `DLUnit` to test simulated digital logic circuits. It is a Java program that allows `JUnit` tests to "drive" either the `JLS` or `Logisim` digital logic simulators [1, 11]. `DLUnit` is a re-write of `JLSCircuitTester` [6] that replaces the older tool's custom syntax with `JUnit` syntax and adds support for `Logisim`. It is available from [8].

Similarly, we created `MUnit` to test MIPS assembly programs [7]. `MUnit` is a Java program that allows `JUnit` tests to "drive" the MARS MIPS simulator. It can also be downloaded from [8].

3.2.2 Test Framework. In order for our process to help students learn to write their own tests, it is important that students be able to run our tests without being able to see them. We do this using GitHub Classroom and GitHub Actions. GitHub Classroom is an instructor-facing tool that automatically creates a separate GitHub repository for each student (or group of students) in a class. Students work on assignments inside the repository created for them. They submit their code by executing a `git push`. When their code is pushed, GitHub automatically executes a GitHub Action that runs the instructor-provided `DLUnit` or `MUnit` test suite. In order to hide the tests from the students, the GitHub Action downloads the tests from the instructor's web site. This web site has a directory that is configured to only accept requests from IP addresses that belong to GitHub.

3.3 Discussion

At the time we made these changes, we had not heard of mastery-based grading; however, our approach is congruent: Students demonstrate mastery over project objectives by submitting code/circuits that pass a suite of tests. Those submissions that pass the tests earn full credit (or close to it). Although some proponents of mastery-based grading discourage the use of factors such as timeliness, our students lose points if their submissions don't pass all the automated tests by a given date. We believe that some students would fall hopelessly behind if project grades did not include a timeliness component. We have not conducted a formal study, but we expect that if projects weren't due until the end of the semester, a significant number of students would fall too far behind to be able to catch up, as was the case at Mills College. We are also concerned that we would receive more submissions than we could grade during the last couple weeks of a semester.

To be clear, we use automated tests, but we do not "autograde" assignments. Each submission is reviewed after it passes the automated tests. For the digital logic assignments, the grader might suggest techniques for making a circuit more efficient and/or readable. For the assembly language assignments, the grader might suggest how to write code that more closely follows established conventions. There is a limit to how many projects we could thoughtfully provide feedback on during the last week or two of a semester, which is another reason deadlines are important. In addition, our feedback would not be useful if students didn't receive it early enough in the semester to apply it to subsequent assignments.

Moving to a mastery-based grading system for the entire course (as opposed to just the aforementioned assignments) would address one of the main shortcomings of our approach: It is difficult to properly balance assignment grades and exam grades. Currently, course grades are simply a weighted average of tests (50%), projects (20%), labs (20%), and homework (10%). Project and lab grades are almost always above 90%. As a result, weak students who work (either formally or informally) with a strong student can earn a final grade that does not reflect their actual level of understanding. Every semester, there are a few students who barely pass the tests but end up with a course average above 75% because their lab and

[2] Of course, as Dijkstra observed, testing can only prove the presence of bugs, not their absence.

[3] For example, this semester, only 2 of the 40 students who completed the course failed to complete Project 1. Among the other 38 students, the lowest grade was 87%.

project grades are well above 90%.[4] The conventional solution of increasing the weight of tests and decreasing the weight of projects is not ideal because doing so would make the projects feel "under valued". (The projects require too much work to be worth "only" 10% of the final grade.)

A mastery-based course grade would eliminate the need to assign a specific weight to projects (or any other category of work). Instead, projects would simply become a task students are expected to complete in order to pass the course. In that sense, their effect would feel proportionate to the amount of work required, but, at the same time, high project grades could not "make up for" low test grades — students would need to do well on both components to earn a high grade in the course.

4 CONCLUSION

The authors applied mastery-based grading to motivate students to create better work and earn higher grades while (we hope) developing a better understanding of both undergraduate computer architecture and practices conducive to success, such as accepting and applying feedback (Mills College) and practicing test-driven development (Grand Valley State University).

Mills students were given rubrics for each learning outcome and the option, in many cases, to retake tests as many times as they chose, which was feasible because test questions were generated programmatically. (This approach would be less applicable in more advanced classes that require deeper understanding.) The ability to retake tests without penalty avoids the frustrating situation where students understand material only after the instructor goes over the graded test, at which point it is too late for grades to improve. Instead, students can decide for themselves when their grade matches their level of mastery and when to move on (instead of giving each student exactly one attempt). While this aspect of mastery-based learning worked well for students and we recommend it, eliminating late penalties proved counterproductive.

GVSU students were expected to completely debug digital logic projects and assembly language programs before receiving a grade for the assignment. We recommend this mastery-based approach because it encourages students to address gaps in their understanding rather than simply "writing off" points. Anecdotally, this approach appears to be successful because the majority of students eventually submit correct projects and programs. In addition, GVSU students were also asked to follow a TDD-like workflow when working on these assignments in order to improve their testing abilities. We can not yet determine the effects this approach has had on our students' testing abilities. We suspect that this approach would need to be adopted by several other courses before there is a measurable effect.

At both schools, technology was employed to reduce the human testing and grading burden. Mills students took programmatically-generated tests with a combination of automatic and human grading, while GVSU students needed their programs to pass automated tests before they could be submitted and graded by a human being. These tools are available at the authors' websites [8, 12].

[4]The projects cover fewer than half of the topics in the course. These struggling students do fine on the project-based questions but do very poorly on questions covering topics like sequential circuits, cache memory, and pipelining.

REFERENCES

[1] Carl Burch. 2002. Logisim: A Graphical System for Logic Circuit Design and Simulation. *J. Educ. Resour. Comput.* 2, 1 (March 2002), 5–16. https://doi.org/10.1145/545197.545199

[2] Robert Campbell, David Clark, and Jessica OShaughnessy. 2020. Introduction to the Special Issue on Implementing Mastery Grading in the Undergraduate Mathematics Classroom. *PRIMUS* 30, 8-10 (2020), 837–848. https://doi.org/10.1080/10511970.2020.1778824 arXiv:https://doi.org/10.1080/10511970.2020.1778824

[3] Amadou Diallo. 2019. How Mastery-Based Learning Can Help Students of Every Background Succeed. *National Public Radio* (March 11, 2019). Retrieved March 23, 2021 from https://www.kqed.org/mindshift/53241/how-mastery-based-learning-can-help-students-of-every-background-succeed

[4] Joe Feldman. 2019. *Grading for equity: what it is, why it matters, and how it can transform schools and classrooms.* Corwin, a SAGE Publishing Company, Thousand Oaks, CA.

[5] Lory Hough. 2019. Grade Expectations: Why we need to rethink grading in our schools. *Harvard Ed. Magazine* (Summer 2019). Retrieved March 23, 2021 from https://www.gse.harvard.edu/news/ed/19/05/grade-expectations

[6] Zachary Kurmas. 2008. Improving student performance using automated testing of simulated digital logic circuits. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education* (Madrid, Spain). ACM, New York, NY, USA, 265–270. https://doi.org/10.1145/1384271.1384342

[7] Zachary Kurmas. 2017. MIPSUnit: A Unit Testing Framework for MIPS Assembly. In *SIGCSE '17: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Kansas City, Missouri). New York, NY, USA, 351–355. https://doi.org/10.1145/3017680.3017747

[8] Zachary Kurmas. 2021. Automated Testing Tools (web site). https://www.cis.gvsu.edu/~kurmasz/Software.

[9] Linda Burzotta Nilson. 2015. *Specifications grading: restoring rigor, motivating students, and saving faculty time.* Stylus Publishing, Sterling, VA.

[10] Connecticut Department of Education. n.d.. *10 Principles of Mastery-Based Learning.* Retrieved March 23, 2021 from https://portal.ct.gov/SDE/Mastery-Based-Learning/10-Principles-of-Mastery-Based-Learning

[11] David A. Poplawski. 2007. A pedagogically targeted logic design and simulation tool. In *WCAE '07: Proceedings of the 2007 workshop on Computer architecture education.* ACM, San Diego, California, 1–7. https://doi.org/10.1145/1275633.1275635

[12] Ellen Spertus. 2021. Computer Architecture teaching/testing tools (GitHub repository). https://github.com/espertus/comparch-testing.

[13] Ellen Spertus. 2021. Interactive Asynchronous Online Computer Architecture Education. In *WCAE '21: Proceedings of the 2021 Workshop on Computer Architecture Education.* ACM, Valencia, Spain.