ASEE CoED Journal
Published 1 Dec 2020

Exploring Literate Programming in Electrical Engineering Courses

Bryan A Jones[1], J W Bruce[2] and Mahnas Jean Mohammadi-Aragh[1*]

[1]Department of Electrical and Computer Engineering, Mississippi State University, Mississippi State, MS, USA

[2]Department of Electrical and Computer Engineering, Tennessee Technological University Cookeville, TN, USA

ORIGINAL RESEARCH

Abstract
Knuth's literate programming paradigm positions source code as a work of literature for which communication to a human is prioritized over communication to a computer. A primary pedagogical value of literate programming lies with the act of writing, especially good writing, leading to good thinking. Issues with early literate programming tool implementations plagued the classroom adoption of literate programming. Advances in technology have warranted a reinvestigation of the benefits of the paradigm. To complement existing inquiry of literate programming in computer programming courses, we investigate, "How can literate programming support student learning in microprocessors and digital system design courses?" In our examination of microprocessors, the instructor used principles of literate programming during in-class demonstrations of assembly programming. In our examination of digital system design, students used the tool to engage in literate programming while writing in a hardware description language. Our results indicated students had a slight preference for instructors to utilize literate programming when presenting in-class examples, and we observed small improvements for graded assignments in sections in which literate programming examples were employed. We also observed a difference in preferences for literate programming by major (computer versus electrical engineering) and noted multiple instructor-observed challenges with introducing a drastically different pedagogical technique in upper-level courses. While our examination did not produce statistically significant results, student and instructor perceptions can be used to guide future literate programming implementations and investigations.

Keywords: Literate Programming, Computing Education, Writing

OPEN ACCESS

Volume
11

Issue
2

****Corresponding author***
Submitted *1 Aug 2019*

Accepted *18 Apr 2020*

Citation
Jones B.A., Bruce J.W. and Mohammadi-Aragh M.J. "Exploring Literate Programming in Electrical Engineering Courses," *Computers in Education Journal*, vol. 11, no. 2, 2020.

Related ASEE Publications

[7] J. W. Bruce, B. A. Jones, and M. J. Mohammadi-Aragh. "A Literate Programming Approach for Hardware Description Language Instruction,". In *2019 ASEE Annual Conference & Exposition*. ASEE Conferences, 2019. URL 10.18260/1-2--31966.

[8] B. A. Jones and M. J. Mohammadi-Aragh. "Employing Literate Programming Instruction in a Microprocessors Course,". In *ASEE Annual Conference & Exposition*. ASEE Conferences, 2016. URL 10.18260/p.26941.

[10] M. J. Mohammadi-Aragh, P. J. Beck, A. K. Barton, D. Reese, B. A. Jones, and M. Jankun-Kelly. "Coding the Coders: A Qualitative Investigation of Students' Commenting Patterns,". In *Proceedings of the 125th ASEE Annual Conference and Exposition*, 2018.

1 Introduction

A glance at current events shows our society's dependence on software. In late January 2016, Nest, Inc., which developed and produces a thermostat controllable via smartphones and accessible via WiFi, pushed a software update to all their devices. Unfortunately, bugs in this update caused users' thermostats to drain the battery, disabling the thermostat and leaving heaters turned off in the middle of winter [1]. Angry complaints of babies waking up at 4 AM to a room at 62°F and panicked calls to restore their homes to operation flooded Twitter and on-line forums. As another example, Volkswagen now faces billion-dollar fines for "defeat device" software in its diesel cars which caused them to pass EPA emissions tests in the lab while emitting much higher pollutants during actual driving conditions [2]. Digital devices and the programs that control them are ubiquitous in modern life.

With advances in technology resulting in complex digital circuitry embedded everywhere in modern life, the importance of programming and digital design knowledge continues to grow. Computer science, computer engineering, and electrical engineering students are expected to master the art of programming and learn multiple domain-specific languages including Python, C++, MATLAB, and R to perform analysis across a number of engineering disciplines. Further, digital system designers often need to use hardware description languages (HDLs) such as ABEL [2], VHDL (now described by IEEE Standard 1076), and Verilog (now described by IEEE Standard 1364). These HDLs are used to describe both behavior and structure of a digital circuit, serving as documentation tools that possess characteristics similar to traditional computer programming languages. The introduction and use of modern HDLs, predominantly Verilog and VHDL, are a hallmark of modern computer engineering curricula [3]. However, most engineering students have no prior exposure to such languages and, as least initially, find HDLs challenging. Students can be further confused when learning Verilog, as Verilog is syntactically similar to the sequential procedural C programming commonly taught to engineering students.

In spite of the pressing need for capable, creative, and above all competent programmers and digital device designers, educators struggle to effectively train students in these essential skills. McCraken's comprehensive examination of first-year CS students reports that only approximately 20% of the surveyed students could solve programming problems expected by their instructors [4]. "Issues impacting students learning how to program" was the topic of an entire Computers in Education Division technical session at the 2019 American Society for Engineering Education (ASEE) Annual Meeting [5]. Clearly, there is a need to explore new pedagogical approaches for teaching students how to program and design digital systems.

Authors Jones and Mohammadi-Aragh are actively exploring literate programming (LP) as one approach to improving programming pedagogy [6–8]. In the LP approach, the programmer (author) composes the program (document) in a form that is readable by humans. In short, the program should be like literature, an essay that contains both explanation for human readers and executable statements for compilers [9]. Jones and Mohammadi-Aragh's exploration of LP in introductory programming courses produced positive results [10]. Due to similarities between programming languages and HDLs, we were motivated to consider whether literate programming could improve student experiences learning programming and HDLs in hardware-focused courses. This work was driven by two initial questions: 1) How could LP principles be implemented in hardware-focused course? and 2) How would students respond to LP in hardware-focused courses?

In this paper, we discuss results of our efforts to explore the use of literate programming (LP) methods in two electrical engineering courses: a microprocessors and a digital system design course. Our implementation of microprocessors required students to program in C++ and PIC24 Assembly. Our implementation of digital system design required VHDL.

2 Background Literature

2.1 Literate Programming (LP)

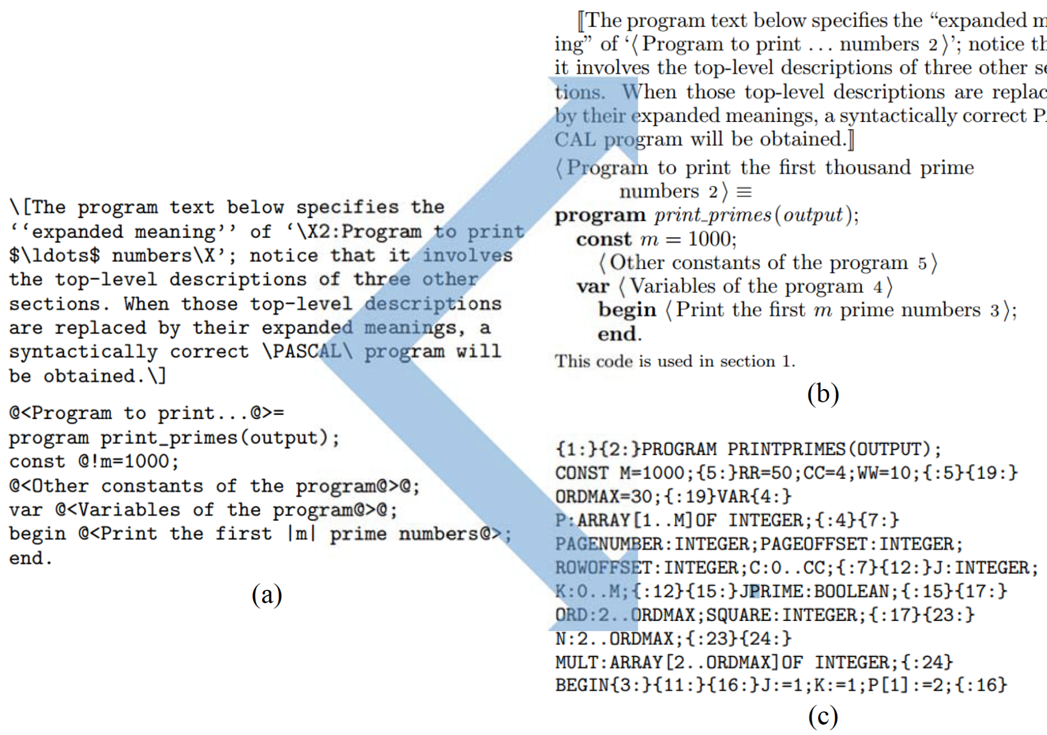
Donald Knuth proposed LP [11] in 1984. In the literate programming paradigm, code and comments are equally important. The literate program mandate that a programmer's creative output consist of a document which contains intermingled code and prose differs radically from typical software development models that view the creative output as code in which scattered comments are not required to connect into a coherent whole. A quick glance at the current state of computer programming paradigms will reveal that Knuth's efforts in LP were not widely adopted. It has been suggested that the failure of adoption of LP for pedagogical use is due, in part, to the lack of sophistication of early LP tools [9] . This last point is substantiated by noting that most education-focused research using LP tools took place in the 1990s. Efforts in this area include using LP tools to grade homework submissions [12] , teach programming [13] , or write better comments [14] . The work in [14] and [15] reports some success, but both cite student complaints about the difficulty of using LP tools.

Knuth's initial LP implementation, called WEB, incorporated ideas of LP but had major weaknesses that limited adoption. The first weakness was related to language support. Pascal was the only supported language. The second weakness was related to writing and formatting. WEB used macros and preprocessor directives to create a meta-language "source document" that would render a WEB file into two separate outputs: a human-readable document and source code for compilation. The WEB source document consisted of a series of cryptic formatting instructions (Figure 1(a)), some of which allowed the inclusion of Pascal code. Knuth's choice of an additional source document meant that the formatted human- readable document (Figure 1(b)) and the source code (Figure 1(c)) had to be created by WEB. Neither the rendered text nor the source code could be edited directly and minor textual edits become major chores. Making minor edits to the formatted document in Figure 1(b) required finding the text in the source document in Figure 1(a) which produced it, editing the offending lines, then regenerate the formatted document to verify that the correct edit was performed. Likewise, modifying the source code in Figure 1(c) requires a similarly laborious process. The "source document" design choice completely isolated authors from their writing. Finally, a third weakness with WEB was that traditional development tools such as debuggers and profilers were extremely difficult to deploy for WEB documents and their associated programs.

Today's popular document generators do not easily allow elaboration about the internal workings of a program — the primary goal of LP. LP implementations developed after WEB somewhat addressed WEB's weaknesses in language support and formatting. Some variants support additional programming languages: CWEB (for C), FWEB (Fortran, C, and C++), xmLP (XML), pyWeb, FunnelWeb, nuweb, and noweb (any language). Other tools provided simpler formatting syntax: nuweb uses LaTeX; noweb uses a simpler set of LP directives and also allows use of LaTeX; pyWeb uses restructured text. Pieterse's survey reviews these and other LP approaches [12]. More recently, documentation generated directly from code has become widespread. Documentation generators, such as Doxygen and JavaDoc, produce a document directly from formatted comments within source code, thus overcoming WEB's second problem. These document generation tools are widely used by programmers, with thousands of programs employing these tools or variants. However, while tools like Doxygen, JavaDoc, and others, were inspired by LP principles [16], they are currently only used to document a program's external interface (e.g., application programming interface (API)). Investigations into the use of documentation generators to support LP principles are needed.

2.2 Cognitive Load Theory

Knuth's LP paradigm is consistent with cognitive load theory [17], which states that keeping related concepts close, temporally or spatially, can improve the ability of students to grasp difficult ideas. Sweller's Cognitive Load Theory [17–19] posits that new concepts (termed schema) must be



```
\[The program text below specifies the
``expanded meaning'' of `\X2:Program to print
$\ldots$ numbers\X'; notice that it involves
the top-level descriptions of three other
sections. When those top-level descriptions
are replaced by their expanded meanings, a
syntactically correct \PASCAL\ program will
be obtained.\]

@<Program to print...@>=
program print_primes(output);
const @!m=1000;
@<Other constants of the program@>@;
var @<Variables of the program@>@;
begin @<Print the first |m| prime numbers@>;
end.
```

(a)

⟦The program text below specifies the "expanded meaning" of '⟨ Program to print . . . numbers 2 ⟩'; notice that it involves the top-level descriptions of three other sections. When those top-level descriptions are replaced by their expanded meanings, a syntactically correct PASCAL program will be obtained.⟧

⟨ Program to print the first thousand prime numbers 2 ⟩ ≡
program *print_primes*(*output*);
  const $m = 1000$;
    ⟨ Other constants of the program 5 ⟩
  var ⟨ Variables of the program 4 ⟩
    begin ⟨ Print the first m prime numbers 3 ⟩;
    end.

This code is used in section 1.

(b)

```
{1:}{2:}PROGRAM PRINTPRIMES(OUTPUT);
CONST M=1000;{5:}RR=50;CC=4;WW=10;{:5}{19:}
ORDMAX=30;{:19}VAR{4:}
P:ARRAY[1..M]OF INTEGER;{:4}{7:}
PAGENUMBER:INTEGER;PAGEOFFSET:INTEGER;
ROWOFFSET:INTEGER;C:0..CC;{:7}{12:}J:INTEGER;
K:0..M;{:12}{15:}JPRIME:BOOLEAN;{:15}{17:}
ORD:2..ORDMAX;SQUARE:INTEGER;{:17}{23:}
N:2..ORDMAX;{:23}{24:}
MULT:ARRAY[2..ORDMAX]OF INTEGER;{:24}
BEGIN{3:}{11:}{16:}J:=1;K:=1;P[1]:=2;{:16}
```

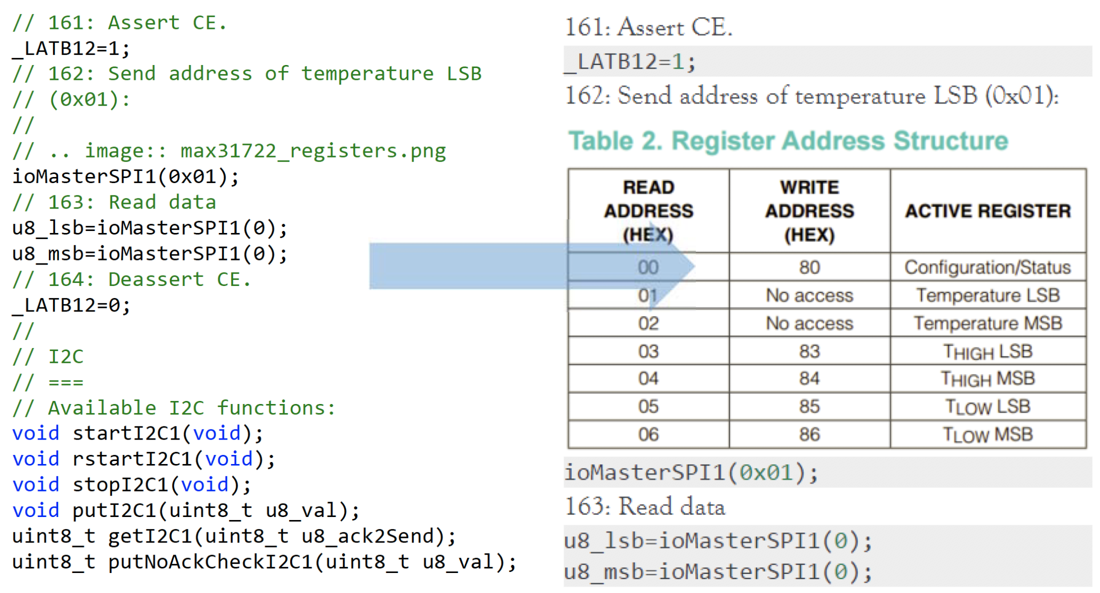
(c)

Figure 1. Knuth'sWEB system for LP transforms the input source document in (a) to the formattedoutput in (b) and the source code in (c) as illustrated by the large arrows

learned in the limited capacity of short-term memory before they can be stored in and employed by long-term memory. The cognitive load of acquiring a new schema can be categorized into several distinct loads. When a schema consists of multiple, interconnected ideas, they must all fit in the limited capacity of short-term memory for learning to occur. The intrinsic cognitive load refers to the number of interconnected ideas which are required to learn a new schema. Poor instructional design can impose extraneous cognitive load, in which the instructional technique unnecessarily increases the number of interconnected ideas, making a schema harder to acquire. When intrinsic load is low, the additional extraneous loading has little effect on learning. However, high intrinsic loading combined with extraneous loading hinders learning. Therefore, instructors should strive to reduce extraneous load produced by their presentation of the material to improve student learning. LP supports instructor efforts by positioning student writing consisting of design decisions, formulas, and figures in close proximity to the source code they are writing.

3 CodeChat: A Modern LP Approach

One of the authors of this paper has developed a modern tool — CodeChat [20] — that attempts to address WEB's failings while still incorporating the main premise of LP. The CodeChat tool combines the strengths of both documentation generators and LP tools. CodeChat builds a formatted document directly from source code, using human-readable ReStructuredText [21] as markup contained in the language's native comments. CodeChat contains a synchronization mechanism which matches source code with the corresponding web output, making editing of either straightforward. CodeChat users simply edit a programming language source file in the left pane, shown in Figure 2(a) and Figure 3(a). The user places very intuitive ReStructuredText markup in the language's comment lines. The CodeChat tool parses the source code file, locates the markup, and renders the program and annotations into human-readable form in real-time into the right-hand pane (see Figure 2(b) and Figure 3(b)). The rendered output can be generated in many different formats, including HTML, TeX, LaTeX, and DocBook. This single-view paradigm is understood by today's computer users within a few moments of use. CodeChat provides an easy-to-use and viable platform for LP techniques in programming. CodeChat supports LP in more than 200 programming and scripting languages "right out of the box." Additionally, it is OpenSource and freely available, making CodeChat a viable tool for conducting research into the use of LP in programming education pedagogy. Figure 2 and Figure 3 illustrate the application of CodeChat in the two courses examined for this study.



```
// 161: Assert CE.
_LATB12=1;
// 162: Send address of temperature LSB
// (0x01):
//
// .. image:: max31722_registers.png
ioMasterSPI1(0x01);
// 163: Read data
u8_lsb=ioMasterSPI1(0);
u8_msb=ioMasterSPI1(0);
// 164: Deassert CE.
_LATB12=0;
//
// I2C
// ===
// Available I2C functions:
void startI2C1(void);
void rstartI2C1(void);
void stopI2C1(void);
void putI2C1(uint8_t u8_val);
uint8_t getI2C1(uint8_t u8_ack2Send);
uint8_t putNoAckCheckI2C1(uint8_t u8_val);
```

(a)

161: Assert CE.

```
_LATB12=1;
```

162: Send address of temperature LSB (0x01):

Table 2. Register Address Structure

READ ADDRESS (HEX)	WRITE ADDRESS (HEX)	ACTIVE REGISTER
00	80	Configuration/Status
01	No access	Temperature LSB
02	No access	Temperature MSB
03	83	THIGH LSB
04	84	THIGH MSB
05	85	TLOW LSB
06	86	TLOW MSB

```
ioMasterSPI1(0x01);
```

163: Read data

```
u8_lsb=ioMasterSPI1(0);
u8_msb=ioMasterSPI1(0);
```

(b)

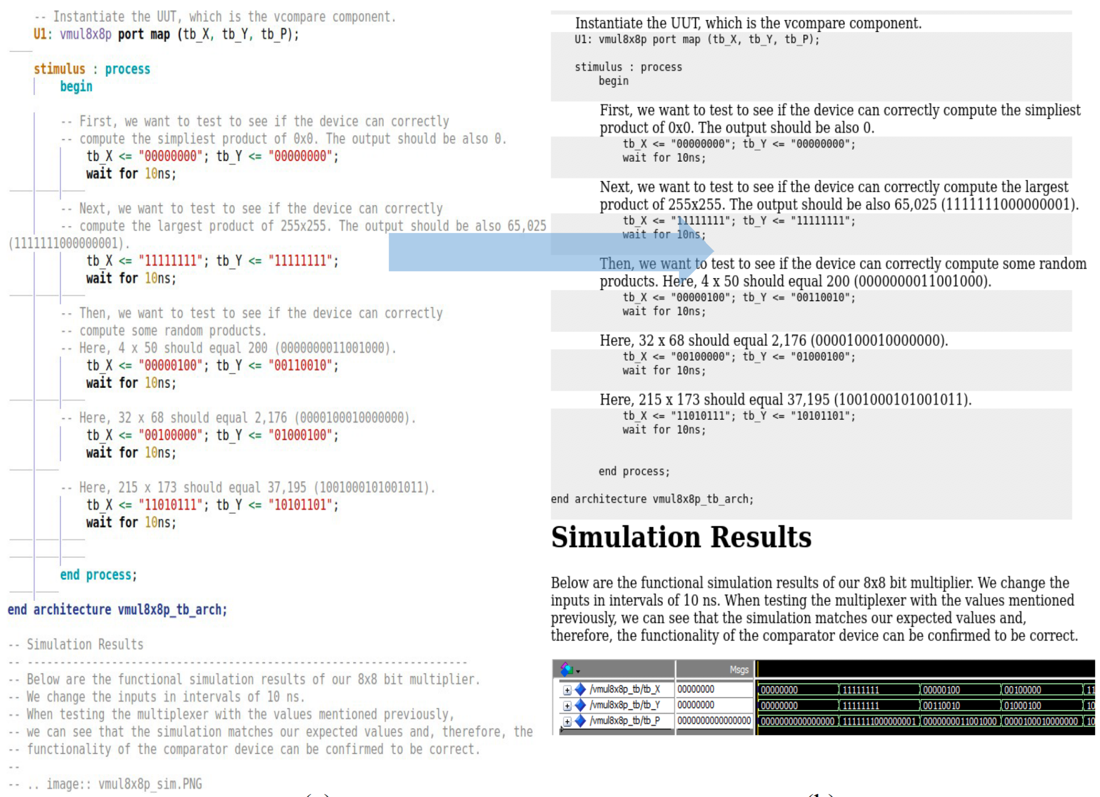
Figure 2. CodeChat, the literate programmingimplementation used to conduct research for this paper, transforms traditionalmicroprocessors course source code in (a) into the web page shown in (b) as shown by the arrow.

4 Research Overview

We assert that good writing leads to good thinking, and good thinking to good programs. We believe that advances in technology and user interface design have warranted new explorations of the benefits of Knuth's LP. We revive Knuth's ideas by developing a new LP tool addressing weaknesses for prior implementations. Then, we use our tool in two electrical engineering courses to answer: "How can LP support student learning in microprocessors and digital system design courses?" In our examination of LP in microprocessors, the instructor used the tool to demonstrate LP during in-class exercises. In our examination of LP in digital system design, the students used the tool to engage in LP for homework and in a design lab environment.

We investigated the impact of LP in a microprocessors course because the course requires students to integrate material from multiple sources creating high cognitive load. For example, the function shown in Figure 2(a) also depends on understanding of a timer and specific reference to the definition of bits in timer 2's control register which is provided in the microcontroller's datasheet. Cognitive load theory predicts that when these elements are spatially or temporally separated, such as refer- ring to a textbook, a datasheet, and traditional source code, additional extraneous load is imposed to successfully integrate these elements. Because LP encourages including all these elements as a part of the document, as shown in Figure 2(b), we hypothesize that the use of literate programs will reduce extraneous load, thereby improving students' ability to master these concepts, which will lead to higher test scores.

We investigated the impact of LP in a digital system design course because modern hardware description languages have similarities with programming languages, and there is limited research



```
    -- Instantiate the UUT, which is the vcompare component.
    U1: vmul8x8p port map (tb_X, tb_Y, tb_P);

    stimulus : process
        begin

        -- First, we want to test to see if the device can correctly
        -- compute the simpliest product of 0x0. The output should be also 0.
            tb_X <= "00000000"; tb_Y <= "00000000";
            wait for 10ns;

        -- Next, we want to test to see if the device can correctly
        -- compute the largest product of 255x255. The output should be also 65,025
(1111111000000001).
            tb_X <= "11111111"; tb_Y <= "11111111";
            wait for 10ns;

        -- Then, we want to test to see if the device can correctly
        -- compute some random products.
        -- Here, 4 x 50 should equal 200 (0000000011001000).
            tb_X <= "00000100"; tb_Y <= "00110010";
            wait for 10ns;

        -- Here, 32 x 68 should equal 2,176 (0000100010000000).
            tb_X <= "00100000"; tb_Y <= "01000100";
            wait for 10ns;

        -- Here, 215 x 173 should equal 37,195 (1001000101001011).
            tb_X <= "11010111"; tb_Y <= "10101101";
            wait for 10ns;

        end process;

end architecture vmul8x8p_tb_arch;

-- Simulation Results
-- ----------------------------------------------------------------------
-- Below are the functional simulation results of our 8x8 bit multiplier.
-- We change the inputs in intervals of 10 ns.
-- When testing the multiplexer with the values mentioned previously,
-- we can see that the simulation matches our expected values and, therefore, the
-- functionality of the comparator device can be confirmed to be correct.
--
-- .. image:: vmul8x8p_sim.PNG
```

(a)

Instantiate the UUT, which is the vcompare component.

```
U1: vmul8x8p port map (tb_X, tb_Y, tb_P);

stimulus : process
    begin
```

First, we want to test to see if the device can correctly compute the simpliest product of 0x0. The output should be also 0.

```
        tb_X <= "00000000"; tb_Y <= "00000000";
        wait for 10ns;
```

Next, we want to test to see if the device can correctly compute the largest product of 255x255. The output should be also 65,025 (1111111000000001).

```
        tb_X <= "11111111"; tb_Y <= "11111111";
        wait for 10ns;
```

Then, we want to test to see if the device can correctly compute some random products. Here, 4 x 50 should equal 200 (0000000011001000).

```
        tb_X <= "00000100"; tb_Y <= "00110010";
        wait for 10ns;
```

Here, 32 x 68 should equal 2,176 (0000100010000000).

```
        tb_X <= "00100000"; tb_Y <= "01000100";
        wait for 10ns;
```

Here, 215 x 173 should equal 37,195 (1001000101001011).

```
        tb_X <= "11010111"; tb_Y <= "10101101";
        wait for 10ns;


    end process;

end architecture vmul8x8p_tb_arch;
```

Simulation Results

Below are the functional simulation results of our 8x8 bit multiplier. We change the inputs in intervals of 10 ns. When testing the multiplexer with the values mentioned previously, we can see that the simulation matches our expected values and, therefore, the functionality of the comparator device can be confirmed to be correct.

```
                      Msgs
/vmul8x8p_tb/tb_X     00000000          00000000         11111111         00000100         00100000         11
/vmul8x8p_tb/tb_Y     00000000          00000000         11111111         00110010         01000100         10
/vmul8x8p_tb/tb_P     0000000000000000  0000000000000000 1111111000000001 0000000011001000 0000100010000000 10
```

(b)

Figure 3. CodeChat transforms digital system design HDL in (a) into the webpage shown in (b) as shown by the arrow.

investigating how LP can support students using hardware description language. The introduction and use of modern HDLs, predominantly Verilog and VHDL, are a hallmark of modern computer engineering curricula [3]. Because digital devices operate concurrently, HDLs allow for description of concurrent operations, similar to programming languages like Haskell and Ada. The complexity of hardware descriptions coupled with HDL similarities to sequential programming languages has led us to propose the idea of introducing LP into HDL education. Using LP in HDL education also reinforces the spirit of why HDLs were created in the first place – to describe hardware behavior. An HDL description should describe – in the most human-readable terms as possible – what the hardware design "looks like" or "how it behaves". The HDL description is, first and foremost, an essay written to fellow designers describing digital hardware. This description also happens to be in an "executable form" thanks to the HDL tool chain. However, the use of LP to improve HDL education has not been widely investigated. One early attempt involved an approach very similar to Knuth's WEB approach [22, 23] using a Prolog logic program to generate a human-readable form and a Verilog HDL file. Our investigation provides further insight into the benefits of LP in support of HDLs.

5 CASE 1: Microprocessors Course

The microprocessors course used for this study focused on introducing students to micro-processors through both lecture and laboratory exercises. The first half of the course focused on instruction in assembly language, and was accompanied by labs in which students translated a C program to assembly, then simulated their assembly code to demonstrate its correctness. The second half of the course required students to build a simple microcontroller and supporting components on a wireless protoboard, then to develop C programs to interface with external peripherals connected to the microcontroller. The first two examinations assessed assembly skills, while the third test and portions of the cumulative final assessed peripheral interfacing in C. Students typically perform well in introductory assembly instruction assessed on Test 1, then struggle with more advanced assembly, such as pointers and extended-precision operations, assessed on Test 2. Likewise, introductory peripheral interfacing in C assessed on Test 3 typically produces good scores, while students struggle with more advanced concepts (I2C and SPI buses, A/D and D/A converters) and their implementation in C on the final.

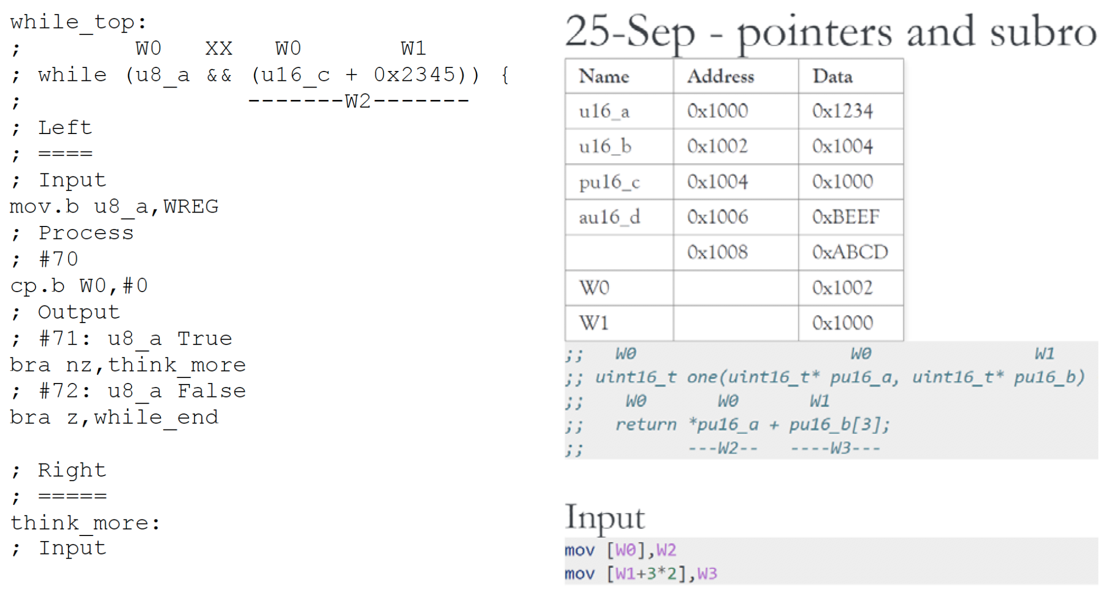
We hypothesized that LP would improve comprehension as the semester progressed to more challenging content distributed across multiple resources. Test 1 material, consisting of introductory assembly, can typically be taught by focusing exclusively on the code itself. While LP provides better formatting and organization, little gain should be expected from a cognitive load perspective because the source code itself contains all the necessary resources. Test 2 material, particularly when discussing pointers, exacts a heavier cognitive load. Students must refer to a memory map (in the form of a table), carefully examine the C code to translate, then write the resulting assembly code. Therefore, the ability of LP to integrate memory maps into the source code should reduce extraneous cognitive load and improve comprehension. While some Test 3 material relies on information available within the source code, some does not. For example, to fully understand the operation of code to configure a timer in Figure 2(a), students must refer to the microcontroller's timer documentation. This higher cognitive load should produce reduced comprehension using traditional techniques, since students' attention must be split between the code and supporting datasheets. Final material includes relatively complex bus protocols such as I2C and SPI, both of which must be used to interact with peripherals. The ability to integrate related information with the code should reduce extraneous cognitive load, producing improved comprehension and scores. For example, comment 162, "Send address of temperature LSB (0x01)" in Figure 2(b) is immediately followed by the table of register addresses, which gives the temperature LSB address as 0x01. Therefore, the following line of code, ioMaster- SPI1(0x01);, clearly sends the temperature LSB address of 0x01 across the I2C bus.

5.1 Approach for Microprocessors

Our investigation made use of a cohort of 58 students enrolled in two sections of the microprocessors course during the Fall 2015 semester. Students self-selected their preferred section based on course scheduling (i.e., timing) preferences. One section, the control group of 27 students, employed no LP methods. The other section of 31 students alternated use of traditional with LP techniques. Specifically, instruction using traditional coding methods was employed for Test 1 material, by developing and discussing code snippets in assembly during in-class exercises. All code written during class was then uploaded to Github [24], a social coding website, providing convenient web access to the code for the students. Next, material for Test 2 was developed using LP techniques to produce a document during in-class exercises. Both the resulting assembly code and its rendering to a LP document were posted on Github and the course website [25], respectively. The same approach was taken for the second half of the course, which employed the C programming language. Test 3 material was covered using traditional techniques, while the material for the final was developed as a set of LP documents. For both cases, the C source and the resulting LP documents were available via the web. Figure 4 shows a comparison between traditional source code (in this case, assembly) used for Test 1 in-class exercises and its LP form used for Test 2 instruction. Likewise, Figure 2 compares traditional C code for Test 3 preparation with the LP variant for the final.

5.2 Data Collection for Microprocessors Case

To evaluate student performance, specific questions were selected on the tests and final which require writing snippets of code in C or assembly. In addition, students in the experimental section participated in an anonymous in-class survey given at the end of the course to provide feedback on the use of LP versus traditional code in the course. The questions were:



```
while_top:
;         W0    XX    W0        W1
; while (u8_a && (u16_c + 0x2345)) {
;                 -------W2-------
; Left
; ====
; Input
mov.b u8_a,WREG
; Process
; #70
cp.b W0,#0
; Output
; #71: u8_a True
bra nz,think_more
; #72: u8_a False
bra z,while_end

; Right
; =====
think_more:
; Input
```

(a)

25-Sep - pointers and subro

Name	Address	Data
u16_a	0x1000	0x1234
u16_b	0x1002	0x1004
pu16_c	0x1004	0x1000
au16_d	0x1006	0xBEEF
	0x1008	0xABCD
W0		0x1002
W1		0x1000

```
;;   W0                     W0               W1
;; uint16_t one(uint16_t* pu16_a, uint16_t* pu16_b)
;;    W0       W0       W1
;;   return *pu16_a + pu16_b[3];
;;           ---W2--   ----W3---
```

Input

```
mov [W0],W2
mov [W1+3*2],W3
```

(b)

Figure 4. Traditional assembly code in (a),compared to its literate programming form in (b).

1. How often did you refer to code discussed in class posted at https://github.com/bjones1/ece3724_inclass? Frequently, several times, a few times, once, or never?
2. On a scale of 1 to 10, where 1 is the least helpful and 10 is the most helpful, how helpful was providing this code on the web?
3. Comments – what would make the code posted more helpful?
4. How often did you refer to the code discussed in class posted in LP form at http://courses.ece.msstate.edu/ece3724/in_class? Frequently, several times, a few times, once, or never?
5. On a scale of 1 to 10, where 1 is the least helpful and 10 is the most helpful, how helpful was providing this code on the web?
6. Comments – what would make the code posted in this form more helpful?
7. Which format did you find more helpful? Traditional, literate programming, neither, or don't care
8. In terms of the code discussed, what would help you learn more during this course?

5.3 Results for Microprocessors Case

Table 1 shows the test questions used to examine student performance. Note that the control section only employs traditional source code, while the experimental section alternates between traditional and LP. Through the differences reported between the control and experimental sections, we attempt to control for the effects of confounding variables between the sections such as instructor style, student background, and student ability. The mean differences (Δ_M) show a statistically insignificant value of 3% between the two sections for Test 1, during which both sections employed traditional source code. This suggests that effects of confounding variables are small, particularly given the wide standard deviations. For Test 2, which compares traditional to LP instruction, a slightly larger difference of 8% was observed. Treating Test 1 as a baseline difference and Test 2 as the expected effect shows a vertical difference of 5%, suggesting a small, but statistically insignificant improvement using LP. The Test 3/Final comparison suggests confounding variables have a greater impact, with a 14% difference for traditional source code. Further, the effect of LP was a statistically insignificant decrease of 1%.

Table 1. Microprocessors Test Scores

Assessment	Control	Experimental		ΔM
	Traditional (M ± SD)	Traditional (M ± SD)	LP (M ± SD)	
Test 1, #12-20	91% ± 8	94% ± 12		3%
Test 2, #17-28	69% ± 13		78% ± 17	8%
Test 3, #1-11, 16-29	70% ± 15	84% ± 14		14%
Final, #24-29	58% ± 19		72% ± 22	13%

Students could provide multiple free-form answers to the eight survey questions listed in the previous section. Thus, the survey results were coded based on the categories provided in the questions. Table 2, Table 3 and Table 4 show a summary of the frequencies reported for questions 1, 2, 4, 5, and 7. Based on these survey results, students refer to both traditional source code and its LP form with essentially equal frequency. The higher response of 61% for traditional versus 42% for LP in the category of "a few times" represents a difference for students who made little use of either system; the usage frequencies for "frequently" and "several times" are almost identical. Likewise, noting the wide standard deviation, students considered both traditional and LP forms equally helpful. Students reported preferring the LP format over the traditional format; however, given a large group (27%) of "don't care" responses, this is a mild preference. Given that this is the first exposure students have to the LP format, and noting that all their previous instruction and programming assignments employ the traditional form, students seem willing to embrace the LP approach. This preference is also notable in that several students reported they were unaware of the existence of the LP webpages.

Table 2. Microprocessors Question 1 and 4 Results

How often did you refer to …?	Q1: Traditional (n=38)	Q4: LP (n=24)
Frequently	3%	4%
Several Times	16%	17%
A Few Times	61%	42%
Once	11%	13%
Never	11%	25%

Table 3. Microprocessors Question 2 and 5 Results

How helpful? (10 = most helpful)	Q2: Traditional (n=26)	Q5: LP (n=26)
m ± S.D.	7.6 ± 1.5	7.0 ± 2.1

Table 4. Microprocessors Question 7 Result

	Tradition	LP	Neither	Don't Care
Q7: Preferred which format? (n=22)	27%	41%	5%	27%

Analyzing the comments provided in response to questions 3, 6, and 8 produced the following four themes:

1. Students liked the code organized by topic, rather than class date. This suggestion has been implemented for the Spring 2016 semester.
2. Several students stated that they were not aware that the literate programming pages were available on the web, thinking they were only provided for in-class exercises. This response

was provided even though students were sent several e-mails giving the address of the LP pages. For the Spring 2016 semester, a change was implemented that required students to access the LP website for every in-class assignment.

3. Students requested live updates as the instructors add examples and notes to the LP document in class. Previously, the LP document wasn't posted to the web until after each class period. This suggestion was implemented for future course offerings.

4. Students requested more explanation and more videos. The class is "flipped," which causes some students to feel cheated that lectures focus on in-class exercises, rather than delivering facts. In addition, some of the older videos need to be updated.

5.4 Discussion of Microprocessors Case

The results of our examination of LP in a microprocessors course indicate that literate programming examples may provide some benefit to students. In particular, a student survey shows a mild preference for the LP form when compared to the traditional form. In addition, an analysis of student performance on examinations shows a small, statistically insignificant improvement in student performance when the instructor employs LP for in-class examples. Larger datasets are needed to verify that this effect holds more generally. We note that in this implementation, students were not writing their own code using LP principles; instead, LP was used during the presentation of in-class exercises. A future examination of learning outcomes when students are writing their own literate programs in microprocessors is warranted and may result in expected learning gains.

6 CASE 2: Digital System Design Course

The digital system design course used for this study was a split-level course targeted at senior undergraduate and introductory graduate students. The course reviewed and revisited digital logic design topics from a prerequisite introductory course and added complexity and practical aspects not covered in the earlier course. All of the coursework was captured in VHDL. The course concluded with a small design project. The first half of the course focused on instruction in VHDL syntax and was accompanied by laboratory sessions in which students wrote descriptions — mostly structural — of simple hardware designs in VHDL. Students composed appropriate test benches to exercise and validate their designs. The second half of the course focused on timing and system design issues along with a more behavioral approach to hardware description. Weekly lab assignments required the students to design ever-larger components that could be reused in their final design project. The course offering described herein was similar to previous semesters with nearly identical topical coverage and pacing using the same text, lecture materials, and lab facilities. The previous iterations of the course were taught by another faculty member who did not employ LP methods. The course is offered once per academic year in a single section.

Our goal with incorporating LP in a digital system design course is to promote HDL "source code" as, first and foremost, an essay written to fellow designers describing digital hardware, which also happens to be in an "executable form" thanks to the HDL tool chain. Furthermore, the more expressive nature of human language and the student's experience with human language should allow the students to more clearly express the behavior and interactions of the digital hardware. Using LP in HDL education reinforces the spirit of why HDLs were created in the first place – to describe hardware behavior. An HDL description should describe in as human-readable terms as possible what the hardware design "looks like" or "how it behaves". The goal was to promote the view that hardware description language "source code" is a product for human consumption instead of tool-chain consumption.

6.1 Approach for Digital System Design

Our investigation made use of a cohort of 36 students enrolled one section of the course during a recent fall semester. The cohort was composed of ten electrical engineering majors who took the class as an elective and twenty-six computer engineering majors for whom the course is required. The course was composed of lecture periods (twice a week; one hour each) and a weekly lab session (once a week; two hours per week). The staffing included a graduate teaching assistant to assist students. Lecture periods were sprinkled with collaborative active exercises. Students were encouraged to work in teams on the lecture active exercises and on laboratory tasks. The instructor often had to expressly direct the collaborative effort as most students would choose to work alone if given the choice. The complexity of VHDL and the overhead of tool chain processes largely prohibited the use of the synthesis tools in-class for the active exercises.

In-class active exercises were mostly focused on the design approach for a particular problem, with students writing "pseudo-code" VHDL. Homework and lab activities consisted of students writing VHDL descriptions annotated with explanations in the CodeChat tool. In previous semesters, students wrote formal lab reports to describe their design approach and results. The addition of LP annotations to their VHDL was the only significant change to VHDL coding assignments compared to previous semesters. The CodeChat tool allowed students to include hyperlinks, figures, tables, equations, FSM diagrams, timing waveforms, etc. directly in their HDL descriptions. Since the code itself was descriptive, and each design's test bench could be annotated with results obtained from the VHDL tools, students were not required to submit a formal lab report. Each design task had a deliverable of VHDL files that were to be liberally annotated with descriptions, explanations, figures, timing diagrams, and observations.

6.2 Data Collection for Digital System Design Case

To ascertain whether LP had an impact on the students' ability to successfully learn VHDL and capture digital systems behavior, at the conclusion of the course students were asked to respond to a survey. Survey questions are provided in Table 5 . Students responded to the survey questions using five-point Likert-type scales. For questions 1 and 2, students could choose a response from the following list: poor, fair, satisfactory, very good, or excellent. The responses were mapped to the values zero (poor) through four (excellent). For questions 3-6, students were given the choice of strongly disagree, disagree, neutral, agree, and strongly agree. These responses were mapped to values -2 (strongly disagree) to +2 (strongly agree), which allows the sign of the response to indicate whether the response is negative or positive.

Table 5. Digital System Design Survey Questions

Question	Question Text
1	I would rate my knowledge of HDL and digital systems design knowledge at the **start** of the course.
2	I would rate my knowledge of HDL and digital systems design knowledge at the **conclusion** of the course.
3	Writing detailed and descriptive comments in my HDL descriptions helped me to learn.
4	I would have rather had traditional Q&A questions instead of writing documented HDL descriptions for course homework.
5	My digital design efforts in this class would have been better summarized with a formal report/paper.
6	Putting my design ideas into words helps me to see errors in my design and improves my overall output.

Degree program information (i.e., major) was collected to examine how differences in background affect a student's view of LP as it was used in the digital systems design course. While computer engineering and electrical engineering are very similar, the computer science and programming skills and experience of the computer engineering majors usually is much more substantial than those of electrical engineering students.

6.3 Results for Digital System Design Case

Table 6 provides the statistics of the students' self-assessed abilities by major for Q1 (knowledge at start of course) and Q2 (knowledge at end of the course). A small number of computer engineers (four out of 16) reported they had some knowledge of the subject before the course started, while electrical engineering majors all reported no prior knowledge of the course topics. At the end of the course, most students of both majors reported their knowledge as being "satisfactory" or better. The difference between a student's response (Q2-Q1) would indicate the course impact on their knowledge. Computer engineering and electrical engineering majors reported a mean difference between Q1 and Q2 of 1.82 and 1.86, respectively.

Table 6. Digital System Design Question 1 and 2 Results

Knowledge of DSD and HDL	CmpE (M $\pm$ SD, Mdn)	EE (M $\pm$ SD, Mdn)	Combined Cohort (M $\pm$ SD, Mdn)
Q1: Start	0.29 $\pm$ 0.59, 0	0.00 $\pm$ 0.00, 0	0.21 $\pm$ 0.51, 1
Q2: End	2.12 $\pm$ 1.22, 2	1.86 $\pm$ 1.07, 2	2.04 $\pm$ 1.16, 2
ΔM (Q2-Q1)	1.82	1.86	1.83

Table 7 provides a summary of student preferences by major for Q3 (writing helped me learn), Q4 (prefer traditional HW style), Q5 (prefer traditional lab report style), and Q6 (writing helps me improve my design). The results from Q3 indicate that computer engineers were slightly more negative than neutral about the LP approach. Electrical engineers were exactly neutral as a group. Of course, individual students reported "strongly agree" and "agree", but there were roughly equal numbers of students reporting "strongly disagree" and "disagree". In Q4, computer engineers indicated a slight preference for homework based on LP-infused VHDL over a set of traditional homework problems. Electrical engineers indicated an even stronger preference for LP and VHDL, although neither group was overwhelming in their preference. Not surprisingly, Q5 responses indicate that both majors are more emphatic about preferring the LP approach to VHDL compared to writing a formal lab report. As with Q4, the preference among the electrical engineers was stronger than the computer engineers. Q6 was worded to try to elicit the student response about LP without ascribing it to the course and lab experience in support of Q3. Both majors were effectively neutral in their views, with electrical engineering majors being slightly positive on average, and computer engineers as a group were nearly neutral. Note that response in Q6 are a bit more positive than the very similar question Q3. Students appear to recognize that writing and natural language may help them express their designs, but the LP approach or negative feelings due to CodeChat installation drove their opinions a bit more negative when the writing involves commenting their HDL.

Table 7. Digital System Design Questions 3-6 Results

LP and Writing in HDL	CmpE (M $\pm$ SD, Mdn)	EE (M $\pm$ SD, Mdn)	Combined Cohort (M $\pm$ SD, Mdn)
Q3: LP helped	-0.41 $\pm$1.33, 0	0.00 $\pm$0.00, 0	-0.29 $\pm$1.27, 0
Q4: Prefer Traditional	-0.18 $\pm$1.19, 0	0.29 $\pm$1.25, 0	-0.21 $\pm$1.18, 0
Q5: Prefer Formal Report	-0.65 $\pm$1.17, -1	-1.29 $\pm$0.76, -1	-0.83 $\pm$1.09, -1
Q6: Writing helps	-0.06 $\pm$1.09, 0	0.29 $\pm$0.95, 1	0.04 $\pm$1.04, 0

A Mann-Whitney U test was chosen as a non-parametric test to compare the two populations [26]. No significant differences between the two populations were observed on questions Q1-Q6.

6.4 Discussion of Digital System Design Case

Students were somewhat ambivalent about the effectiveness of LP to improve their HDL in the digital design course with computer engineering students somewhat more negative. One theory is the computer engineers are more comfortable with the topics covered in the course and with programming languages and documentation than the electrical engineers. The design tasks in the course were not overly complex, so computer engineers may have felt the LP paradigm was simply too much overhead and work for the reasonably simple technical HDL that followed. Electrical engineers are less comfortable in the course described here, and the LP approach may have allowed them to better express themselves. If this is a valid view, one might expect the student enthusiasm to increase as assignments and designs get more complicated. Such a result would also agree with the observation that computer programmers tend to comment more heavily in complicated code, or code that they do not initially understand [27, 28]. LP would provide the developer with a more suitable mechanism for human-digestible descriptions.

Students, especially the computer engineering majors who have more experience with computing languages, may have also suffered from the mistaken impression that any computer activity time spent not creating "working code" is a waste of time. With more experience, they may realize that documentation of computer code or VHDL descriptions leads to a much higher maintainability and, ultimately, shorter overall development cycles and lower costs [27, 28].

Anecdotal observations by the instructor and other faculty are that student comprehension and performance seemed largely unchanged compared to previous semesters. The next step would be to form a controlled experiment wherein one group will use traditional editors or IDEs to develop largely undocumented HDL descriptions. A treatment group would use the tools and LP approach described here. Incorporation of formal product metrics such as defect production and development time would also give more objective measures to the quality of student output.

Finally, student performance on other course measures or later academic outcomes might indicate a difference. Alternatively, an ideal, but often difficult to deploy, approach to testing the efficacy of LP in the digital systems design course would be a formal experiment with control and treatment groups of students taught under similar educational conditions during the same academic period. This was not possible due to existing scheduling and faculty workloads. Another approach could create control and treatment groups within the same lecture and lab population. This approach is problematic as students may view different assignments with different levels of difficultly or otherwise detrimental to their grade. Control and treatment groups could be formed by student self-selection, but groups of comparable sizes, abilities, and motivation are very suspect with this scheme. Given the circumstances, a decision was made to apply the treatment to the entire cohort.

Similar to prior LP implementations, concerns with the tool may have overshadowed the benefits of LP. The CodeChat tool was created by one of the authors. The tool is not as polished as commercial or mature open-source software applications. The tool itself works as advertised and is quite stable. Students never reported dissatisfaction with tool functionality. However, installation of the CodeChat tool as problematic. To aid in CodeChat development, the CodeChat tool uses a wide variety of open-source libraries and supporting frameworks. Several of these software packages had varying compatibility issues with student laptops. Students were quite vocal in expressing their frustration with the installation process at the beginning of the semester. The students' unhappiness was compounded as installation on one student's laptop would be smooth and flawless, and another student with an identical computer would experience installation problems. Ultimately, several installation problems were a result of students not following the detailed installation instructions provided by the instructor. Eventually, the instructor was able to assist every student and get the tool operational. However, this initial negative experience likely colored the students' impression of the CodeChat tool and the LP paradigm.

7 Conclusions

The hope in undertaking our exploratory investigations of LP in hardware-focused courses was that LP would aid student learning, and improve the quality of the student output. We have presented our results with the hope that other educators and researchers will apply LP in microprocessors and HDL education at their institutions to help build a more complete understanding of how LP can support student learning in electrical engineering courses.

We have presented two cases of how LP could be used in hardware-focused courses. In the first case, microprocessors, LP techniques were used by the instructor to present and explain in-class examples. In the second case, digital system design, students used LP principles while writing HDL. Students were less resistant to the new LP paradigm when the instructor was the one intermingling writing and coding. This could be related to our choice to implement LP in advanced hardware courses, which required a minimum of four semesters of programming prerequisite courses in which students did not use LP. In our prior work with LP implementations in introductory courses we have not observed as much resistance and it will be interesting to examine whether students who used LP in early courses have similar or less resistance to LP in advanced courses. However, based on our current data set, we recommend that instructors who implement LP in upper-level hardware courses transition students to the method by first using LP to present and explain in-class examples.

In spite of limitations with the current available tool, students only expressed a slight preference for more traditional pedagogical methods. Additionally, especially for non-computing majors, students recognized that writing about their designs in their natural language could help them express their designs in HDL. This result is promising and motivates further examinations of LP in hardware-focused courses. We challenge instructors to consider ways of implementing natural language writing tasks in conjunction with hardware design activities. While we wait for LP tools to be refined, instructors could consider implementing journaling or reflection prompts into their hardware design assignments.

The efforts reported herein represent only the beginning of the pedagogical possibility for LP in hardware courses. As we continue our parallel work that is exploring how LP and writing-to-learn strategies support students who are at the early stages of learning to program, we will consider how those findings can inform learning gains in upper level hardware courses. Additionally, the results herein support additional research into the effectiveness of instructor use of LP when presenting in-class examples. The results herein also provide evidence to support future investigations into the differences in student perceptions of LP by student background (e.g., novice versus expert, differences by major). From a tool perspective, extending CodeChat from its current state, in which edits can only be made to the source code, to enable direct modification of the LP document, would significantly improve the usability of the system and may assist students in separating their perceptions about the tool from their view of LP benefits.

Acknowledgments

Portions of this material is based upon work supported by the National Science Foundation under Grant No. DUE-1612132. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Bilton, N. (2016) ‘Smart’ Home Suffers a Brain Freeze. *The New York Times*.

[2] Warrick, J. (2015). URL https://www.washingtonpost.com/news/energy-environment/wp/2015/09/18/epa-volkswagen-used-defeat-device-to-circumvent-air-pollution-controls/.

[3] Ardis, M., Budgen, D., Hislop, G.W., Offutt, J., Sebern, M., and Visser, W. (2015) SE 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. *Computer*, **11**, 106–109.

[4] Mckracken, M. (2001) A Multi-national, multi-institutional study of assessment of programming skills of first-year CS Students. *SIGCSE Bulletin*, **33** (4), 125–180.

[5] ASEE (2019), Online Session Locator: M308 Technical Session 1: Issues Impacting Students Learning How to Program. URL https://www.asee.org/public/conferences/140/registration/view_session?session_id=11337.

[6] M, Mohammadi-Aragh, J., Beck, P.J., Barton, A.K., and Jones, B.A. (2019) A Case Study of Writing to Learn to Program, in *Proceedings of the 126th ASEE Annual Conference and Exposition*.

[7] Bruce, J.W., Jones, B.A., and Mohammadi-Aragh, M.J. (2019) A Literate Programming Approach for Hardware Description Language Instruction, in *2019 ASEE Annual Conference & Exposition*, ASEE Conferences. URL 10.18260/1-2--31966.

[8] Jones, B.A. and Mohammadi-Aragh, M.J. (2016) Employing Literate Programming Instruction in a Microprocessors Course, in *ASEE Annual Conference & Exposition*, ASEE Conferences. URL 10.18260/p.26941.

[9] Mertz, D. (2000), Charming Python #8: Interviews with Creators of Vyper and Stackless Python. URL http://gnosis.cx/publish/programming/charming_python_8.html, Interview with M.J. Skaller.

[10] Mohammadi-Aragh, M.J., Beck, P.J., Barton, A.K., Reese, D., Jones, B.A., and Jankun-Kelly, M. (2018) Coding the Coders: A Qualitative Investigation of Students' Commenting Patterns, in *Proceedings of the 125th ASEE Annual Conference and Exposition*.

[11] Knuth, D.E. (1984) Literate Programming. *The Computer Journal*, **27** (2), 97–111. URL 10.1093/comjnl/27.2.97.

[12] Pieterse, V., Kourie, D.G., and Boake, A. (2004) A Case for Contemporary Literate Programming, in *SAICSIT, Stellenbosch, Western Cape, South Africa*, *SAICSIT '04*, vol. 75 (ed. and others),
South African Institute for Computer Scientists and Information Technologists, ZAF, *SAICSIT '04*, vol. 75, pp. 2–9.

[13] Hurst, A.J. (1996) Literate programming as an aid to marking student assignments, in *Proceedings of the 1st Australasian Conference on Computer Science Education* (ed. and others), Association for Computing Machinery, New York, NY, USA, ACSE '96, pp. 280–286. URL 10.1145/369585.369650.

[14] Childs, B., Dunn, D., and Lively, W. (1995) Teaching CS/1 Courses in a Literate Manner, in *Proceedings of the TeX Users Group Conference*, *3*, vol. 16, *3*, vol. 16, pp. 300–309.

[15] Shum, S. and Cook, C. (1994) Using literate programming to teach good programming practices. *ACM SIGCSE Bulletin*, **26** (1), 66–70. URL 10.1145/191033.191059;https://dx.doi.org/10.1145/191033.191059.

[16] Gulbrandsen, A. and Personal Website (2009), The History of udoc. URL http://rant.gulbrandsen.priv.no/udoc/history.

[17] Sweller, J. (1988) Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, **12** (2), 257–285. URL 10.1207/s15516709cog1202_4;https://dx.doi.org/10.1207/s15516709cog1202_4.

[18] Sweller, J. and Chandler, P. (1994) Why Some Material Is Difficult to Learn. *Cognition and Instruction*, **12** (3), 185–233. URL 10.1207/s1532690xci1203_1;https://dx.doi.org/10.1207/s1532690xci1203_1.

[19] Sweller, J., Merriënboer, J.J.V., and Paas, F.G. (1998) Cognitive architecture and instructional design. *Educational Psychology Review*, **10** (3), 251–296.

[20] Jones, B.A., Mohammadi-Aragh, M.J., Barton, A.K., Reese, D., and Pan, H. (2015) Writing-to-Learn-to-Program: Examining the Need for a New Genre in Programming Pedagogy, in *Proceedings of the 122nd ASEE Annual Conference and Exposition*.

[21] Goodger, D. (2016), Restructuredtext: Markup Syntax and Parser Component of Docutils. URL http://docutils.sourceforge.net/rst.html.

[22] Bowen, J.P. (2000) Combining Operational Semantics, Logic Programming and Literate Programming in the Specification and Animation of the Verilog Hardware Description Language. *Integrated Formal Methods. IFM 2000*, **1945**. URL 10.1007/3-540-40911-4_16.

[23] Bowen, J.P., Jifeng, H., and Qiwen, X. (2000) An animatable operational semantics of the Verilog hardware description language, in *ICFEM 2000. Third IEEE International Conference on Formal Engineering Methods* (ed. and others), pp. 199–207. URL 10.1109/ICFEM.2000.873820.

[24] Jones, B.A. (2019), CodeChat - ECE 3724 on GitHub. URL https://github.com/bjones1/ece3724_inclass, Down.

[25] Jones, B.A. (2019), ECE 3724 Microprocessors. URL https://sites.google.com/site/ece3724/Home.

[26] Winter, J.C.D. and Dodou, D. (2010) Five-point Likert items: t test versus Mann-Whitney-Wilcoxon (Addendum added October 2012). *Practical Assessment, Research and Evaluation*, **15**. URL https://doi.org/10.7275/bj1p-ts64;https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1237&context=pare.

[27] Pressman, R.S. (2001) Software Engineering: A Practitioner's Approach, McGraw-Hill, McGraw-Hill series in computer science.

[28] Mcconnell, S. (2004) *Code Complete 2/e*, Best Practices, Microsoft Press.