Employing Literate Programming Instruction in a Microprocessors Course

Dr. Bryan A. Jones, Mississippi State University

Bryan A. Jones received the B.S.E.E. and M.S. degrees in electrical engineering from Rice University, Houston, TX, in 1995 and 2002, respectively, and the Ph.D. degree in electrical engineering from Clemson University, Clemson, SC, in 2005. He is currently an Associate Professor at Mississippi State University, Mississippi State, MS.

From 1996 to 2000, he was a Hardware Design Engineer with Compaq, where he specialized in board layout for high-availability redundant array of independent disks (RAID) controllers. His research interests include engineering education, robotics, and literate programming.

Dr. Mahnas Jean Mohammadi-Aragh, Mississippi State University

Dr. Jean Mohammadi-Aragh is an assistant research professor with a joint appointment in the Bagley College of Engineering dean's office and the Department of Electrical and Computer Engineering at Mississippi State University. Through her role in the Hearin Engineering First-year Experiences (EFX) Program, she is assessing the college's current first-year engineering efforts, conducting rigorous engineering education research to improve first-year experiences, and promoting the adoption of evidence-based instructional practices. In addition to research in first year engineering, Dr. Mohammadi-Aragh investigates technology-supported classroom learning and using scientific visualization to improve understanding of complex phenomena. She earned her Ph.D. (2013) in Engineering Education from Virginia Tech, and both her M.S. (2004) and B.S. (2002) in Computer Engineering from Mississippi State. In 2013, Dr. Mohammadi-Aragh was honored as a promising new engineering education researcher when she was selected as an ASEE Educational Research and Methods Division Apprentice Faculty.

©American Society for Engineering Education, 2016

Employing Literate Programming Instruction in a Microprocessors Course

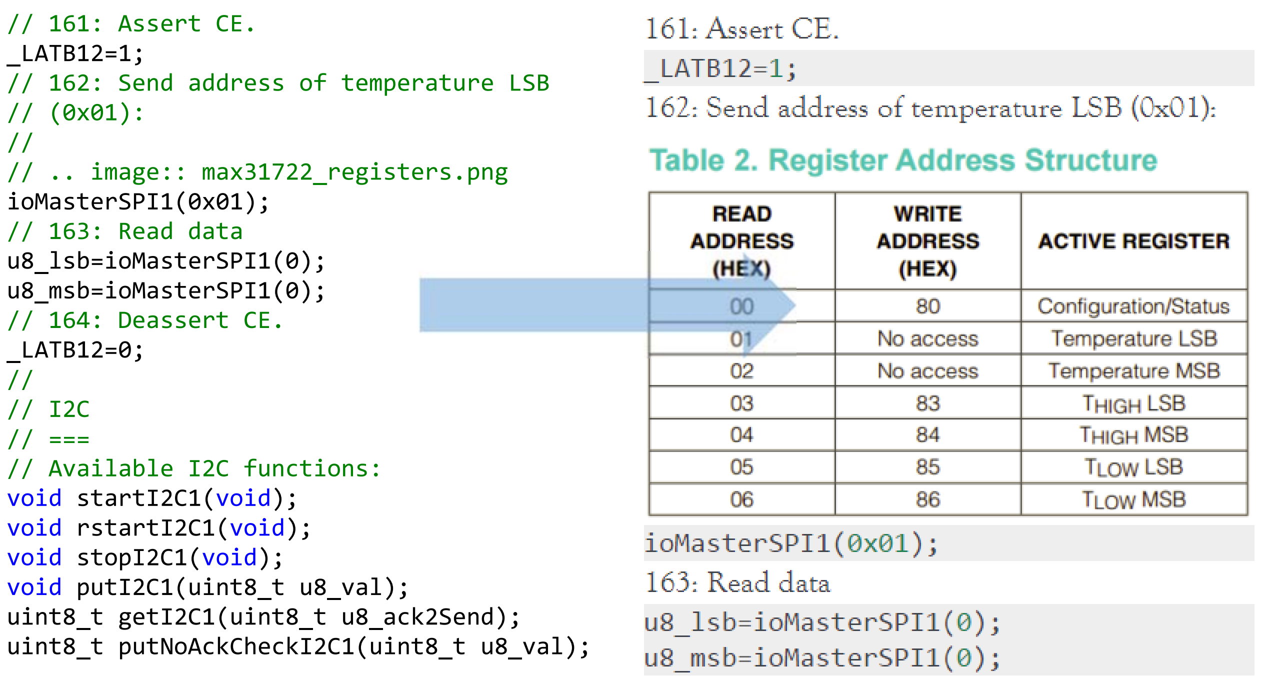
1. Abstract

Learning to program is difficult and has been documented as a persistent problem not just for computer science majors, but also for other engineering majors who use programming as a tool within their disciplines. One solution may lie in Knuth's literate programming paradigm, which treats a program as an essay, intermingling code with explanation in a beautifully typeset document. This stands in contrast to traditional programming pedagogy where difficult-to-understand code is isolated from its explanation, in a separate file from the flowcharts and text which detail the operation of the program. Knuth's literate programming paradigm is consistent with cognitive load theory, which states that keeping related concepts close, temporally or spatially, can improve the ability of students to grasp difficult ideas. Based on this, we hypothesize that programs intermingled with explanation and typeset as a document will improve student learning when compared to traditional instruction in which programs and their explanation remain separated. In this paper, we examine the use of literate programming in a junior-level course on microprocessors in the Department of Electrical and Computer Engineering using both a student survey and analysis of test scores. By alternating between literate programs/documents and traditional programs with accompanying explanations, we evaluate learning gains measured by performance on tests with and without literate programming. Results show a mild preference by students for the literate programming form, and analysis of student performance on tests shows small (but statistically insignificant) gains when using literate programming. Based on these results, we discuss future directions for this new approach to programming pedagogy.

2. 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]. We depend on the integrity of on-line banking, expect Facebook to connect us with our friend's activities, and trust cruise control in cars to drive us safely.

In spite of the pressing need for capable, creative, and above all competent programmers, educators struggle to effectively train students in these essential skills. McCraken's comprehensive examination of first-year CS students[3] reports that only approximately 20% of the surveyed students could solve programming problems expected by their instructors. In addition, the importance of programming continues to grow; not only are CS and ECE students expected to master the art of programming, but student mastery of domain-specific languages such as MATLAB, R, Maple, and Mathematica are now required to perform analysis across a number of engineering disciplines.



```
// 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 1: CodeChat, the literate programming implementation used to conduct research for this paper, transforms traditional source code in (a) to the web page shown in (b) as shown by the arrow.

The sad state of programming pedagogy may well be the result of the elimination of writing when writing a program. That is, textbooks present a program in well-crafted essays, instructors coach students in developing flow charts and design documents, but all this beautiful writing is sadly abandoned when the actual program is written, resulting in a monospaced monstrosity only a compiler could love, as shown in Figure 1(a).

We assert that good writing leads to good thinking, and good thinking to good programs. In order to improve computer science education, writing must be reintroduced into writing a program. This approach follows Knuth's literate programming paradigm, which views a program as an essay, intermingling code with explanation in a beautifully typeset document, as shown in Figure 1(b). This transformational idea produces a radical shift in beliefs about what defines a program: a program is, first and foremost, an essay written to fellow programmers explaining the purpose behind and structure of a specific implementation, which also captures that implementation in an executable form. That is, a program is not primarily written for a computer, but for a person.

Unfortunately, Knuth's literate programming system closed more doors than it opened. While it produced beautiful documents, composing these documents and understanding the code they produced proved difficult (see Figure 2), prompting all prior investigations of pedagogical use of literate programming to report that students found the literate programming system hard to use. If students can't easily interact with their code in its document form, they will not think of their code as a document; tools must therefore support this code-as-document belief to support effective pedagogical innovations. This paper therefore re-examines pedagogical use of literate programming techniques with CodeChat, a simple, easy-to-use literate programming system.

As a first step in the process of reintroducing writing into writing a program, this paper discusses the results of the use of literate programming techniques by instructors as models for students, in the form of a series of in-class exercises. We hypothesize that students will prefer the literate programming form of a program (illustrated in Figure 1(b)) to the traditional form of a program shown in Figure 1(a). In addition, we hypothesize that students will demonstrate greater understanding of material presented in literate programming form, instead of presented as a combination of traditional code supplemented by data sheets.

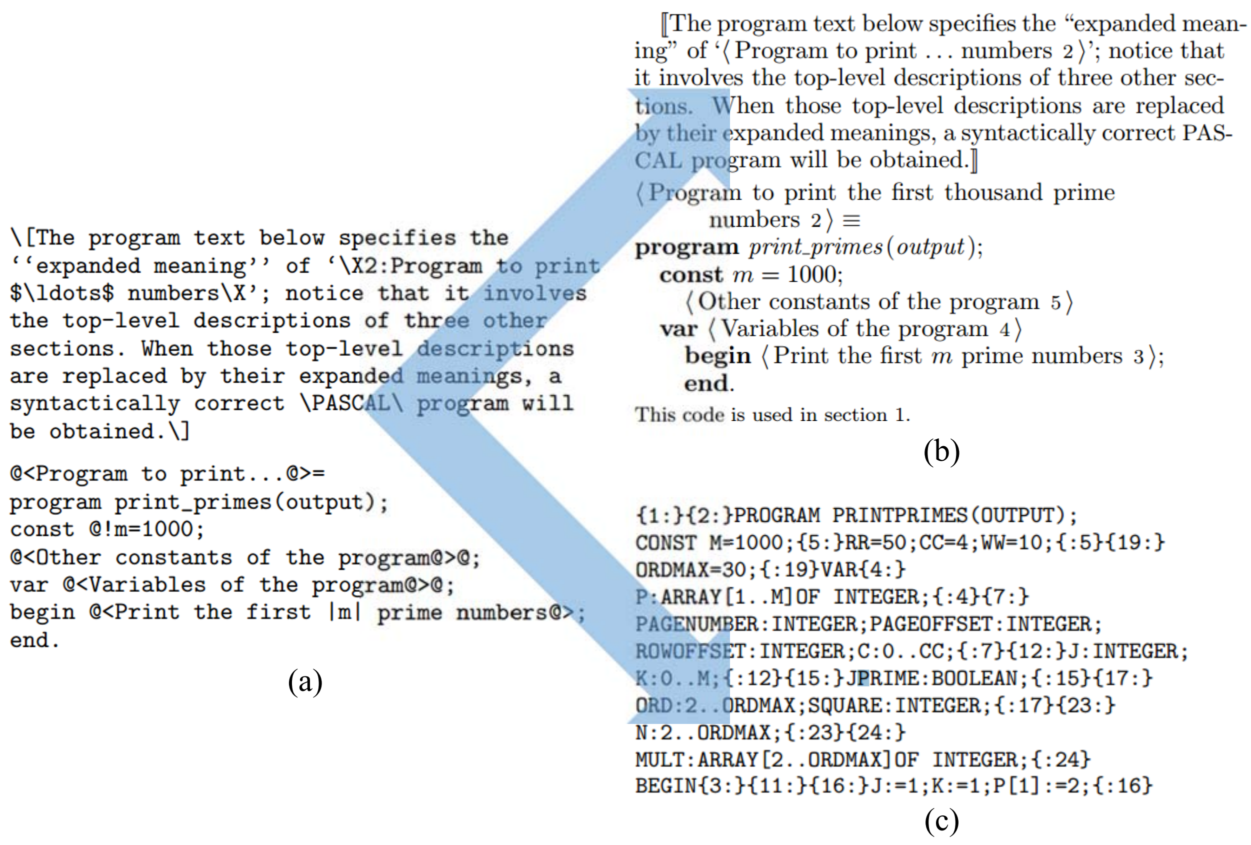
3. Background

3.1 Literate programming

Knuth coined the term literate programming (LP)[4], in which an author composes an essay containing both explanation for readers and executable statements for compilers. This creation of a document consisting of intermingled code and prose differs radically from the traditional software development model, which views comments as an optional accompaniment to the code and makes no attempt to connect these scattered comments into a coherent whole. That is, "you do not document programs (after the fact), but write documents that contain the programs."[5] For the first time, software developers could view a program as a document; code with comments alone simply does not look like nor read like an essay, no matter how well written, as illustrated in Figure 1(a). Educators can provide their students with an executable textbook – high-quality prose with detailed explanations, which at the same time is a program students can compile, execute, and explore.

This transformation of perspective – the understanding that a program is and, at its core, should be an essay – succeeds at several levels. First, at an esthetic level, the typeset results shown in Figure 1(b) are surprisingly beautiful, particularly when compared with a monospaced font in which source code is otherwise shown. This typeset output employs a proportionally-spaced font which is easier to read, includes headings, contains typographical emphasis such as bold or italics, and allows the inclusion of diagrams, all of which increase comprehension. At a deeper, more experiential level, this typeset output reinforces the belief that a program *is* a document, encouraging programmers to write documents, instead of disjointed comments. Finally, this underlying belief that a program is a document then opens to the authors the multitude of advantages which accrue to writers: creation of and reflection on the overall structure of the essay; the ability to easily include others in the development process; the inclusion of the creative ideas which produced a particular implementation.

This last point bears further investigation. Traditional programming focuses on the what – the code, which defines a specific implementation. This information provides a compiler all the information needed to blindly execute these instructions. To borrow an analogy[6], if Aunt Matilda bakes a cake, the tools of science can determine the what – chemical composition, material properties, and nutritional content. These tools, by nature, cannot determine the why; to determine this, we must ask Aunt Matilda the purpose for which she baked the cake. Likewise, code alone tells us what, but not why. This why represents a vital, missing piece in the traditional software development process. Why was this specific implementation chosen? Was it a casual, unthinking decision, or the result of hours of debugging to find and correct a rare mode of failure? This explanation, stating



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

@<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 2: Knuth's original literate programming system, WEB[4], transforms the input source document in (a) to the formatted output in (b) and the source code in (c) as illustrated by the large arrows. To edit (b) or (c), the corresponding source text in (a) must be located, edited, then (b) and (c) regenerated to verify the edit, making any changes laborious.

the purpose behind a snippet of code, provides vital insight and must be communicated through writing, a practice central to the exercise of literate programming.

While brilliant, WEB, Knuth’s initial literate programming implementation, hindered its adoption. First, as shown in Figure 2(a), the source document consists of a series of cryptic formatting instructions, some of which allow the inclusion of Pascal code, the only programming language supported by WEB. These cryptic TeX-based formatting commands raised a high barrier of entry, discouraging adoption. Second, the choice of an additional source document means that the formatted document shown in Figure 2(b) and the source code shown in Figure 2(c) are derived using LP tools (WEB’s tangle and weave) from the source document: neither can be directly edited. Making a minor edit to the formatted document therefore requires finding the text in the source document which produced it, editing that, then regenerating the formatted document to verify that the correct edit was performed. Likewise, modifying the source code requires a similarly laborious process. Minor textual edits become major chores, and the use of traditional development tools such as debuggers and profilers becomes extremely difficult.

Later literate programming implementations addressed the first problem: 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). Others provide simpler formatting syntax: nuweb uses LaTeX; noweb a simpler set of literate programming directives and also allows use of LaTeX; pyWeb uses restructured text. Pieterse's survey reviews other literate programming approaches[7]. However, the second problem exists by Knuth's design: choosing a source document from which LP tools produce both source code and a formatted document prevents direct modification of either the source code or the formatted document, isolating authors from the writing they must do. For these reasons, no literate programming tool has gained widespread acceptance in the programming community or for sustained pedagogical use.

This last point is substantiated by noting that most education-focused research using literate programming tools took place in the 1990s. Efforts in this area include using LP tools to grade homework submissions[8], teach programming[9] (with success, but accompanied by student complaints about the difficulty of LP tools), or write better comments[10] (with success, but again including students complaints on the difficulty of debugging LP-generated code).

In contrast, documentation generators, such as Doxygen and JavaDoc, produce a document directly from formatted comments within source code, thus overcoming the second problem. They are widely used by programmers, with thousands of programs which employ these tools or their variants[a, b]. While these tools were inspired by LP principles[11], they can only be used to document the external interface (the application programming interface, or API), not the internal working of a program, thus losing the core benefits of literate programming. Therefore, this paper employs the CodeChat tool[12] which combines the strengths of both documentation generators and LP tools: it builds a formatted document directly from source code, using human-readable restructured text as markup in comments. In addition, it contains a synchronization mechanism which matches source code with the corresponding web output, making editing of either straightforward. This simplified approach provides a viable platform for conducting research into the use of literate programming in computer science pedagogy.

3.2 Cognitive load theory

Sweller's Cognitive Load Theory[13, 14, 15] (CLT) posits that new concepts (termed schema) must be 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.

[a] See http://www.stack.nl/~dimitri/doxygen/projects.html.
[b] JavaDoc is included in Oracle's Java implementation and used to document its API, resulting in wide adoption in the Java community. See, for example, https://docs.oracle.com/javase/8/docs/api/overview-summary.html.

Much of the instruction in the microprocessors course requires students to integrate material from multiple sources. For example, the function shown in Figure 1(a) also depends on understanding of a timer provided in the textbook used in the course, and specific reference to the definition of bits in timer 2's control register which is provided in the microcontroller's datasheet. Cognative load theory predicts that when these elements are spatially or temporally separated, such as referring to a textbook, a datasheet, and traditional source code, additional extraneous load is imposed to successfully integrate these elements. Because literate programming encourages including all these elements as a part of the document, as shown in Figure 1(b), we hypothesize that the use of literate programs will reduce extraneous load, therefore improving students' ability to master these concepts, which will lead to higher test scores.

4. Approach

The authors instruct ECE 3724[c], a course offered within the Department of Electrical and Computer Engineering at Mississippi State University, which focuses on introducing students to microprocessors through both lecture and laboratory exercises. The first half of the course focuses on instruction in assembly language, and is accompanied by labs in which students translate a C program to assembly, then simulate their assembly code to demonstrate its correctness. The second half of the course requires students build a simple microcontroller and its supporting components on a wireless protoboard, then to develop C programs to interface with external peripherals connected to the microcontroller. The first two tests cover assembly skills, while the third test and portions of the cumulative final focus on peripheral interfacing in C.

Students typically perform well in introductory assembly instruction covered in test 1, then struggle with more advanced assembly, such as pointers and extended-precision operations, in test 2. Likewise, introductory peripheral interfacing in C covered in test 3 typically produces good scores, while students struggle with more advanced concepts (I^2C and SPI busses, A/D and D/A converters) and their implementation in C on the final.

This paper made use of a cohort of 58 students enrolled in two sections of ECE 3724 during the Fall 2015 semester. One instructor, leading the control group of 27 students, employed no literate programming methods in their section. The other instructor alternated use of traditional with literate programming techniques in their section of 31 students. 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[d], a social coding website, providing convenient web access to the code for the students. Next, material for test 2 was developed using literate programming techniques to produce a document during in-class exercises; both the resulting assembly code and its rendering to a literate programming document were posted on Github and the course website[e], respectively. The same approach was taken for the second half of the course, which employs the C programming language. Test 3 material was covered using traditional techniques, while the material for the final was developed as a set of literate programming documents; for both cases, the C source and the resulting

[c] All course materials are available at http://courses.ece.msstate.edu/ece3724.
[d] See https://github.com/bjones1/ece3724_inclass.
[e] See http://courses.ece.msstate.edu/ece3724/in_class.

literate programming documents were available via the web. Figure 3 shows a comparison between traditional source code (in this case, assembly) used for Test 1 in-class exercises and its literate programming form used for Test 2 instruction. Likewise, Figure 1 compares traditional C code for Test 3 preparation with the literate programming variant for the final.

4.1 Discussion

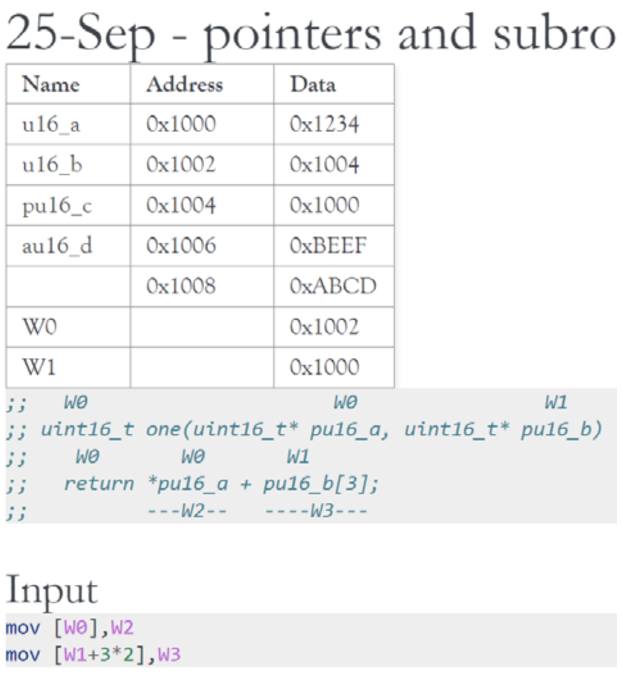
Test 1 material[f], consisting of introductory assembly, can typically be taught by focusing exclusively on the code itself. While literate programming provides better formatting and organization, little gain should be expected from a cognitive load perspective because the source code itself contains 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 literate programming to integrate memory maps into the source code should reduce extraneous cognitive load and improve comprehension. Figure 3 compares these two approaches.

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 1(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 I^{2}C and SPI, both of which must be used to interact with peripherals. The ability to integrate

```
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 3: Traditional assembly code in (a), compared to its literate programming form in (b).

[f] All tests and solutions for the course are publicly available at http://courses.ece.msstate.edu/ece3724/old/history.html.

	Control section	**Experimental section**		**Difference**	
	Traditional	Traditional	LP		
Test 1, #12-20	91%±8%	94%±12%		3%	5%
Test 2, #17-28	69%±13%		78%±17%	8%	
Test 3, #1-11, 16-29	70%±15%	84%±14%		14%	−1%
Final, #24-29	58%±19%		72%±22%	13%	

Table 1: Student test scores. The control section employs only traditional source code, while the experimental section alternates between traditional and literate programming. Values are given as mean±standard deviation. Horizontal differences are reported between the control and experimental section means.

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 1(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 I^{2}C bus.

4.2 Data collection

To evaluate student performance, specific questions were selected on the tests and final which require writing snippets of code in C or assembly. Table 1 shows the test questions used to examine student performance.

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 literate programming versus traditional code in the course. The questions are:

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 literate programming 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. Results

Because students could provide multiple free-form answers to the questions above, the results were coded based on the categories provided in the questions. Table 2 shows a summary of the results.

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

1. Students would like 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 weren't aware that the literate programming pages were available on the web, thinking they were only provided for in-class exercises. Students were sent several e-mails giving the address of the LP pages; for the Spring 2016 semester, students must access the LP website for every in-class assignment.
3. Students would like live updates available via the web, 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 has been implemented in the Spring 2016 semester.
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.

Based on these survey results, students refer to both traditional source code and its literate programming 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 student 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 literate programming format, and noting that all their previous instruction and programming assignments employ the traditional form, students seem willing to embrace the literate programming approach. This preference is also notable in that several students reported they were unaware of the existence of the literate programming webpages.

How often did you refer to:	**1. Traditional**	**4. LP**
Frequently	3%	4%
Several times	16%	17%
A few times	61%	42%
Once	11%	13%
Never	11%	25%
# responses	38	24

7. Which format?	
Traditional	27%
LP	41%
Neither	5%
Don't care	27%
# responses	22

How helpful?	**2. Traditional**	**5. LP**
Mean	7.576923	7
Std. Dev.	1.498026	2.075498
# responses	26	26

Table 2. Survey results. Numbers, such as "1. Traditional", refer to the numbered questions given in §4.2.

In Table 1, the differences reported between the control and experimental section attempts to reduce the effects of confounding variables between the sections such as instructor style, student background, and student ability. These differences show a negligible delta of 3% between the two sections for test 1, during which both sections employed traditional source code, suggesting that effects of confounding variables are small, particularly given the wide standard deviations. Test 2, which compares traditional to literate programming instruction, shows a slightly larger difference of 8%. Treating test 1 as a baseline difference and test 2 as the expected effect shows a vertical difference of 5%, suggesting a small, statistically insignificant improvement using literate programming. The test 3/final set suggests confounding variables have a greater impact, with a 14% difference for traditional source code. However, in this set the effect of literate programming is small, with a small (but again statistically insignificant) decrease of 1%. Overall, these results (a delta of 5%, then –1%) suggest a small, though statistically insignificant improvement when employing literate programming.

6. Conclusions and future work

In conclusion, these results indicate that literate programing may provide some benefit to students in the context of a microprocessors course. In particular, a student survey shows a mild preference for the literate programming 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 employing literate programming. Larger datasets are needed to verify that this effect holds more generally.

The efforts reported herein represent only the beginning of the pedagogical possibility for literate programming. One promising avenue for exploration involves engaging students in the creation of a literate program, using writing-to-learn strategies in this context of programming[12]. 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 literate programming document, would significantly improve the usability of the system. Further research into the effectiveness of instructor use of literate programming to confirm its benefits, as well as experiments on the usefulness of literate programming techniques for experienced programmers, is also warranted.

References

[1] Bilton, Nick, "'Smart' Home Suffers a Brain Freeze", The New York Times, Jan. 14, 2016, pg. D2. Also available as http://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html.

[2] Warrick, Joby, "EPA: Volkswagen used 'defeat device' to illegally skirt air-pollution controls," The Washington Post, Sep. 18, 2015, available from https://www.washingtonpost.com/news/energy-environment/wp/2015/09/18/epa-volkswagen-used-defeat-device-to-circumvent-air-pollution-controls/.

[3] McKracken, M. et al. (December 2001) *A Multi-national, multi-institutional study of assessment of programming skills of first-year CS Students*, SIGCSE Bulletin, vol. 33, no. 4, pp. 125-180.

[4] Knuth, D. (1992). Literate Programming (1984). Literate Programming. CSLI. p. 99.

[5] Skaller, M. J., in a Charming Python interview. Retrieved from http://gnosis.cx/publish/programming/charming_python_8.html.

[6] Lennox, John, "God's Undertaker: Has Science Buried God," Lion Hudson Plc, 2007, p. 40.

[7] Pieterse, V., Derrick G. K., & Boake, A. (2004). *A case for contemporary literate programming*, in SAICSIT, 75, 2-9, Stellenbosch, Western Cape, South Africa.
[8] Hurst, A. J. (1996). *Literate programming as an aid to marking student assignments*. Proceedings of the 1st Australasian conference on Computer Science education, 280-286.
[9] Childs, B., Dunn, D., & Lively, W. (1995). *Teaching CS/1 Courses in a Literate Manner*. Proceedings of the TeX Users Group Conference, St. Petersburg, Florida, July 24-28, Volume 16, No. 3, p. 300-309.
[10] Shum, S. & Cook, C. (1994). *Using Literate Programming to Teach Good Programming Practices*. Proceedings of the twenty-fifth SIGCSE symposium on Computer Science education, p. 66-70.
[11] See http://rant.gulbrandsen.priv.no/udoc/history.
[12] B. A. Jones, M. Jean Mohammadi-Aragh, A. K. Barton, D. Reese, and H. Pan, "Writing-to-Learn-to-Program: Examining the Need for a New Genre in Programming Pedagogy," *Proceedings of the 122nd ASEE Annual Conference and Exposition*, June 14-17, 2015, Seattle, WA, USA.
[13] Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. Cognitive science, 12(2), 257-285.
[14] Sweller, J., & Chandler, P. (1994). Why some material is difficult to learn. *Cognition and instruction*, *12*(3), 185-233.
[15] Sweller, J., Van Merriënboer, J. J., & Paas, F. G. (1998). Cognitive architecture and instructional design. *Educational psychology review*, *10*(3), 251-296.