Co-Lecturing With the DED

Explaining Circuit Design via the Draw Encode Display Loop

Alasdair Lambert
alasdair.lambert@strath.ac.uk
University of Strathclyde
Glasgow, UK

Guillaume Allais
guillaume.allais@strath.ac.uk
University of Strathclyde
Glasgow, UK

Conor Mc Bride
conor.mcbride@strath.ac.uk
University of Strathclyde
Glasgow, UK

Abstract

When representing digital circuits, 2 dimensional hand drawings free us from the linear structure of hardware description languages, enabling intuitive reasoning and making structure explicit. However these drawings are imprecise and inert: they do not enforce that the circuits are well defined and cannot be tested.

We want both intuitive visual representations and well defined testable ones but students can struggle to link one ot the other.

To bridge this gap we present the Draw Encode Display Loop (DED), a Co-Lecturing dynamic which equips students with a systematic method to tackle natural language specifications:

(1) **Draw** visually informative intermediate representations (truth tables, characteristic tables) to generate a structured circuit diagram.
(2) **Encode** the diagram by labelling inputs, outputs and intermediate values which can be directly converted to code.
(3) **Display** the code using an in-house diagrammatic renderer.

This is supported by Syrup, an education-focused hardware description language which allows students to define, experiment with and display their own circuits. We support these approaches with survey data gathered from two cohorts of students.

CCS Concepts

• **Applied computing** → **Computer-assisted instruction**.

Keywords

Co-Lecturing, Digital Logic, Programming Education

ACM Reference Format:
Alasdair Lambert, Guillaume Allais, and Conor Mc Bride. 2018. Co-Lecturing With the DED: Explaining Circuit Design via the Draw Encode Display Loop. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

By focusing on high-level understanding, a visual representation of a complex and technical topic can reduce the cognitive load that would be associated with a textual implementation. Clearly this has pedagogical value as high level understanding provides a foundation on which to build detailed knowledge [5]. But visualisations are a double edged sword for Computer Science educators: heavy use of visual strategies can mean that students are less confident when presented with text-based programming languages [9].

When teaching digital logic to first year undergraduate students we like to first draw a rich and colourful diagram showing the problem's structure, and then translate it into a formal description language which allows us to check our solution has the correct behaviour. This is because small digital circuits have a clear flow of information from inputs, through wires, into logic gates, and towards outputs. Correspondingly, diagrams are a natural visualisation tool for such circuits with varied colours making the information flow explicit and a well chosen layout highlighting their structural properties. To wit, fig. 1 shows how distinct the diagrams for the sequential and parallel versions of a circuit taking the conjunction of its 4 inputs look.

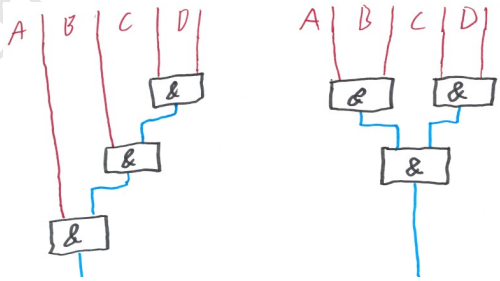

Figure 1: Sequential and Parallel And for 4 inputs as diagrams

We also use textual hardware desription languages because they can easily be processed by computers. They enable users to make sure the wiring is correct, and interactively simulate these circuits' behaviour. These languages may however obfuscate the physical structure of the circuits and make it harder to distinguish between functionally equivalent but structurally different components as demonstrated by fig. 2. This figure shows how the hardware description implementations[1] for the sequential and parallel circuits pictured above only differ in how parentheses are arranged. A difference that could easily be missed in a much bigger description.

But by using this visual-first approach we are cut by the aforementioned double edged sword: we present a high-level visualisation of circuits to solve a given problem and we need our students to convert these visualisations to a hardware description language.

[1]We will explain the syntax of Syrup definitions in section 4; for now it is enough to play 'spot the differences' between the two definitions.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

```
and4seq(<Bit>, <Bit>, <Bit>, <Bit>) -> <Bit>
and4seq(A, B, C, D) = A & (B & (C & D))

and4par(<Bit>, <Bit>, <Bit>, <Bit>) -> <Bit>
and4par(A, B, C, D) = (A & B) & (C & D)
```

Figure 2: Sequential and Parallel And for 4 inputs in Syrup

To resolve this we present the Draw-Encode-Display (DED) loop, a co-lecturing dynamic for delivering circuit design lectures which highlights the connection between visual circuit representations and their formal implementation. Here one lecturer solves a problem using pen and paper (Draw), discussing how to approach and resolve problems with the available tools, before handing the diagram over. The other lecturer can then focus on formalising the diagram (Encode). Finally, they close the loop by checking that the code corresponds to the hand drawn solution by using the display feature (Display).

The DED loop has been used for two years during semester one of CS106 Computer Systems and Organisation. This is a core first year module with approximately 180 students per year, the course covers digital logic and basic instruction set architecture

By surveying two cohorts of students we present preliminary data addressing the following research questions.

RQ1 Do students find the DED loop helpful?
RQ2 Do students prefer the DED loop to a code-only approach?
RQ3 Did students find Co-Lecturing helpful?

2 Related Work

The DED loop is a co-lecturing strategy involving visual representations of computational objects and live programming.

Visual Representations of Programs. Block-based programming, where learners build programs by composing compatible blocks which perform small actions, is an established tool for beginner programmers [9, 16–20]. These tools, due to their visual nature, allow beginners to write code more easily than with text-based programming languages [19, 20] and have a small but positive impact on academic outcomes [17, 20]. These language are engaging and make composition of effects easier [19] but they suffer from a perceived inauthenticity as they are not industrial strength [19] and can cause frustration when moving to text-based languages [9]. Circuit diagrams function in much the same way: small components are wired together to create larger circuits. We can therefore expect the same benefits and drawbacks when using hand drawn circuits, however with a lack of machine verified feedback for students.

Live Programming. Our work relies heavily on the Cognitive Apprenticeship (CA) model [2] combined with live programming [6, 11–13, 21]. The CA model focuses on making thinking explicit by having educators talk through their thought process as they solve problems [2]. This can help learners understand topics that can otherwise seem theoretical, or inaccessible [6]. It has a natural pairing with live programming, the engaging practice of interactively building a program from a specification for the benefit of the audience. However live programming is far from a panacea for programming education [13] and has been shown to be similar in terms of student outcomes to static delivery [14]. Similarly, the CA approach has proven benefits in CS education however it struggles to scale well in the scaffolding and fading stages [15].

Co-Lecturing. Co-lecturing [4] is a form of team teaching where multiple lecturers present each lecture. Team teaching has an established track record at all education levels [1, 8, 10, 22]. In primary education it has been shown to have a positive impact on educational attainment [1]. However at tertiary education it is less well explored and often focuses on groups of teaching assistants [10]. Co-Lecturing has been shown to be useful tool for training [8], where new lecturers can join existing teams, shoring up resilience to staffing changes while acting as a critical friend [3] for module enhancement. Co-Lecturing can be deployed for particular examples [7] but has been shown to be particularly successful when combined with the CA model for live programming [4].

The DED loop uses aspects of visual programming and live programming, delivered via co-lecturing with an emphasis on CA delivery. Taken as a group we believe these techniques harmonise to provide a holistic delivery method. Pen and paper drawing are used to exploit the benefits of visual programming such as lower cognitive load by reduced syntactic noise and ease of compostition. A hardware description language is then taught via co-lecturing and live programming. The final stages, encode and display are intended to show students how to convert from diagrams to code helping to avoid common problems with visual languages.

3 Approach

The DED loop has been used for two years during semester one of CS106 Computer Systems and Organisation. As the class is only taken by Computer Science or Software Engineering Students, we do not use standard Electrical Engineering notations for logic gates. This reduces the time students spend learning notation, but also means lectures focus on the logic and meaning of circuits, as well as building large complex systems from small well understood ones.

Students use Syrup[2], a bespoke hardware description language written in Haskell, to define circuits that we describe in section 4.

Lectures are delivered via co-lecturing with a heavy focus on the CA delivery model. Each lecture is presented as a series of problems to be solved which are tailored to illustrate the topics of the week. Lectures are split between the two lecturers, one introduces the ideas or problems for the lecture and then works through them using pen and paper. Colours, selected from a colourblind friendly palette to assure accessibility, are used to indicate meaning e.g. place value. Here the focus is to talk through the high level approach to a problem, often starting with a english language problem description, generating a truth table and then designing a circuit which implements that behaviour. This forms the Draw section of the lecture and aims to cement the strategy for solving the problem in the minds of the students before any code is written.

Once a pen and paper solution is generated it is handed to the second lecturer and the second part of the lecture begins, formalising the pen and paper circuit via Syrup. This is the Encode section of the lecture. First they begin by listing the inputs and outputs, the bits that are expected to be fed into the circuit and the ones that are expected as output. This forms the first line of the Syrup definition.

[2]Syrup is freely available at https://github.com/pigworker/Syrup, and an interactive tutorial page can be found at https://personal.cis.strath.ac.uk/guillaume.allais/syrup/.

Intermediate wires are then given names that bear some meaning within the circuit, as are the inputs and outputs. This means that within the circuit definition the lecturer can show that generating the code is simply a matter of chasing outputs back through the diagram. This shows students that if they have the diagram the code can be generated systematically. Once the circuit definition is complete the display feature is used to show that the defined function is indeed the same as the pen and paper solution. This is the Display section of the lecture. The ability to simulate the formalised circuit is also used in this stage to make sure that the behaviour of the circuit is correct.

The DED loop is used during every lecture which involves building circuitry. Roles for each lecturer are clearly defined, however both lectures can interject or clarify during the others section. Generally more time is spent on the Draw section as new topics are introduced and the proposed solution needs to be thoroughly explained. This allocation can change if new Syrup features need to be introduced, or if class feedback indicates student need more support with Syrup.

4 Syrup syntax

Syrup is a text-based statically typed programming language of circuit descriptions built by connecting components using named wires.

Types. Each wire has a type describing its structure: it can either be carrying a single bit (`<Bit>`), or be a cable packing together n wires, each with a given type T_i (`[`$T_1, \ldots, T_n$`]`).

Declarations. A component is declared by specifying its name and its interface (i.e. the types of its inputs and outputs). For instance, in the code below: line 1 declares a gate named `not` with one input and one output, both of type `<Bit>`, and line 2 and 3 respectively declare gates named `and` and `or` both of which take two inputs and return one output, all of type `<Bit>`.

```
1 not(<Bit>) -> <Bit>
2 and(<Bit>, <Bit>) -> <Bit>
3 or(<Bit>, <Bit>) -> <Bit>
```

Without knowing the components' respective implementation, their interfaces alone already tell us how they can be wired to other components to build bigger circuits.

Definitions. A component is defined by giving a name to each of its input and using various pre-existing components to produce its outputs. For instance, in the code below: line 1 defines the `not` gate declared above by stating that its one input is named `X`, and that `not` is defined by duplicating this input, feeding it into both inputs of a `nand` gate[3] and immediately returning that result. Once defined, a component can be freely reused in subsequent definitions. For instance, line 2–3 give the standard definitions of `and` and `or` in terms of `nand` and the `not` gate we just introduced.

```
1 not(X) = nand(X, X)
2 and(X, Y) = not(nand(X, Y))
3 or(X, Y) = nand(not(X), not(Y))
```

Declarations are type checked to ensure our wiring respects every component's interface.

[3]The universal gate `nand` comes pre-defined in Syrup; everything else is ultimately defined in terms of it

Where blocks. Larger definitions become difficult to define without breaking up the logic. To ameleorate this Syrup supports **`where`** blocks which allow students to name intermediate outputs. This means that outputs can be implemented in a step-by-step manner. See the definition of `hadd`, a half-adder which adds two bits together. Note that in the where block meaningful names are given to the bit, representing their place value.

```
1 hadd(<Bit>, <Bit>) -> <Bit>, <Bit>
2 hadd(X, Y) = twos, ones where
3   twos = and(X, Y)
4   ones = xor(X, Y)
```

Experiments. Once we have defined components, we can experiment with them. In this paper we are particularly interested in the **`display`** function which allows us to visualise (cf. fig. 3) the circuit diagrams corresponding to the textual representations given above in order to check that our code does indeed describe the circuits we expect. Note that the box in `not`'s diagram is an explicit duplication node taking the signal named `X` and outputting two copies of it.

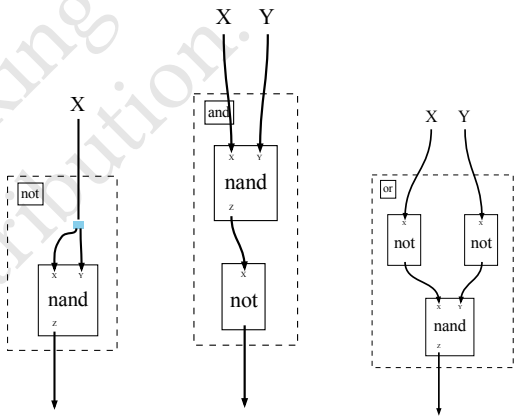

Figure 3: Syrup circuit diagrams of `not`, `and`, and `or` defined in terms of `nand`

By using **`display`** to visualise the sequential and parallel variations on and4 shown in fig. 2 we can see (in fig. 4) that we indeed obtain circuits that have radically different structures.

4.1 A DED-Easy Worked Example

This example takes place at a point in the class where the students have learned about a range of basic logic circuits, basic binary arithmetic, half adders, and full adders. For this example let us consider a ripple carry adder a circuit which adds two cables of three bit numbers and a carry in together, returning a cabled three bit number and a carry out. We will refer to this circuit as rca3. While lecturers often have names, we will refer to the lecturer responsible for the draw section of the loop as the Architect and the lecturer responsible for encoding the diagram as the Technician. Before the lecture starts two projector screens are setup, the first uses a document camera to display a blank sheet of paper and the other a Syrup editor. This means that students can see both the code and the written solution at the same time.

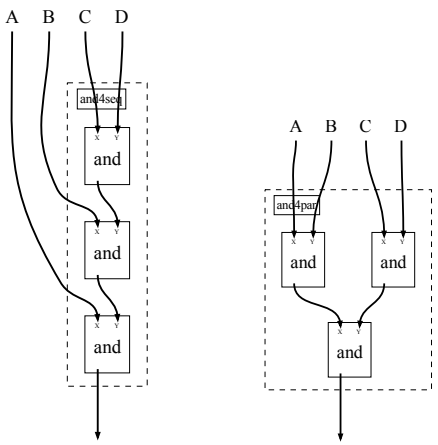

Figure 4: Variations on and4

4.2 D for Draw

The Architect begins with a problem statement and its context. They recap that the class has covered how to add two bits together using a half adder, and that they have improved on that with a full adder to add three bits together. They point out that the carry-in of a full adder makes it a composable component so that carry-forwards can be accomodated. The Architect then describes the problem, how do we add cables of bits together?

The Architect sketches out the inputs to the circuit. Multiple colours are used, the Architect asks the students why this is the case? In cases of stunned silence the Technician can chime in to ask "is it to do with place value?". Bits which have the same place value are given the same colour, the Architect points out that this makes the problem simple, all we have to do is deal with one place value at a time. With this in mind the class is asked another question, "Do we know of a way to add three bits together?". The lecturers hope to hear the words "full adder", any bold and correct student is rewarded with a caramel wafer. Armed with the solution the Architect shows on paper how to tie up all of the values of place value 1 using a full adder. This produces a value of place value 1 together with a carry-forward of place value 2 thus allowing the Architect to use the same trick again for the (now three) bits of place value 2. They pause to reinforce how useful the carry-in is for composition. They can then finish the problem showing the solution given in figure 5.

With the finished solution the Architect pauses, asking the class if they understand this solution. Hopefully the class will air any questions they have, however, should they fail to, the Technician can jump in to ask questions. In the case where the Technician feels the Architect was unclear with any part of their explanation they can offer an alternative explanation, or asked questions which tease out more information. When both of the lecturers feel comfortable that the class has a good grasp on the problem, or at least that the students have no further questions, and proposed solution the Architect asks the Technician to write the code that represents this diagram.

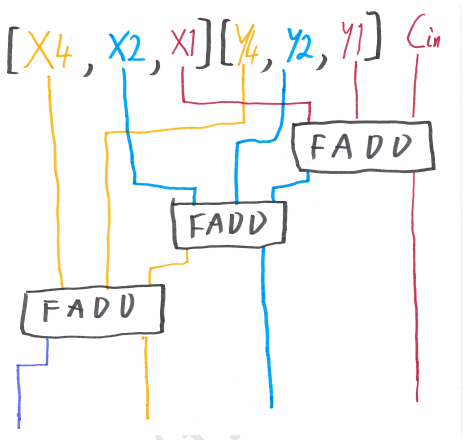

Figure 5: Solved RCA3

4.3 E for Encode

The Technician begins by explaining what we need to implement a Syrup circuit. First, a type signature specifying the circuit's interface. The Technician takes a pen and draws a dotted box around the circuit, being careful to make sure input and output wires overrun the box. They then explain that this shows how the circuit interacts with the outside world, or in other words what the circuit expects as inputs and what it gives back as outputs. The Technician then annotates each input and output with a type written using Syrup syntax. In our running example, these annotations are respectively cables of size 3 (`[<Bit>, <Bit>, <Bit>]`) and single bits (`<Bit>`) as seen in figure 6.

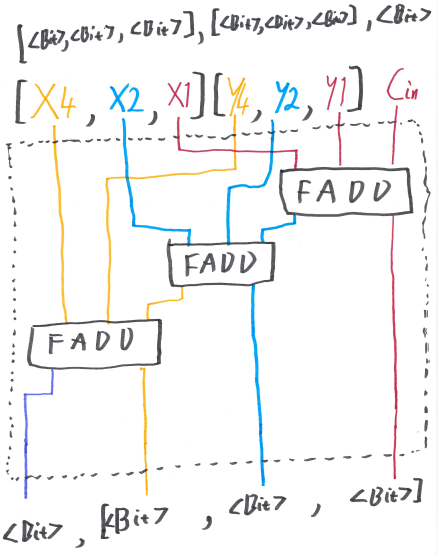

Figure 6: Input and Output Wires

With the diagram updated, they move to the editor and start to define the circuit. They explain again that Syrup needs to know the inputs and outputs to define a circuit, but happily they can simply

read this from the diagram! Noting that repeatedly writing the type of cables of three bits is a little bit tedious, they start by introducing a type synonym `<Bit3>` for it (line 1 below). This has the added benefit of making the meaning of the circuit clearer: an `rca3` takes two 3 bit numbers and a carry in and return a carry out and a 3 bit number (line 2).

```
1 type <Bit3> = [<Bit>, <Bit>, <Bit>]
2 rca3(<Bit3>, <Bit3>, <Bit>) -> <Bit>, <Bit3>
```

We now have the beginning of a circuit definition. The Technician explains that we have told Syrup the type of our circuit but not its implementation. So the Technician returns to the diagram. They explain to the class that we know how the circuit works at a high level but it would be better to have small, easy to understand sections that we can implement, rather than trying to write the whole thing in one line. Using the diagram they explain that it would be helpful to name all of the outputs so that we can explain how to build them. This helps, they explain, but we still have unnamed wires. These wires correspond to intermediate results that are produced by some gates and then fed into others. They explain really these ought to have names so that we can tell Syrup how to build them as well. The Technician then labels the intermediate wires C2 and C4 to show that they are carry-forwards into the 2s and 4s respectively, this is shown in figure 7.

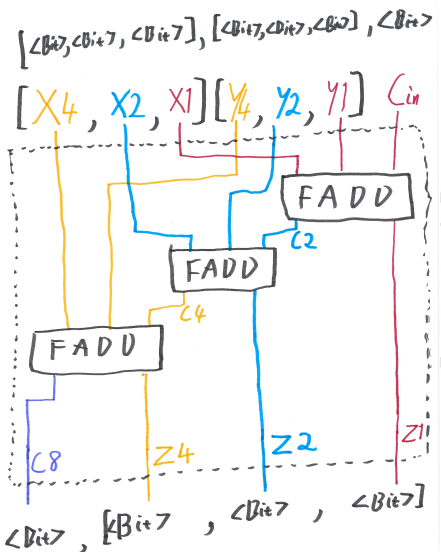

Figure 7: Intermediate Wire Naming and Output Values

With the final labelled diagram in hand the Technician then begins building the circuit. They encode the outputs as a bit representing the a carry-forward to the 8s, `C8`, and the resulting three bit sum packed as a cable of 3 bits `[Z4, Z2 , Z1]`. They then show that we can explain how to compute these signals by building small blocks at a time. As with the draftsman's diagram they tie up the values of place value 1 using a full adder, labelling the output as `Z1`, the bit produced for output of place value 1 and the carry-forward into place value 2 `C2`. In that process, they note that the name `Cin` used in the diagram breaks the mnemonic of labelling each name with its place value and insist on using `C1` instead. The Architect sulks but agrees that the Technician is correct. They carry on, using the labelled diagram to repeatedly ask "How do I define this" and follow on by reading it from the diagram and encoding it as Syrup code. At each step they include a comment (`-- Using this syntax`) to document what the purpose of the intermediate computations is.

```
rca3([X4, X2, X1], [Y4, Y2, Y1], C1)
  = C8, [Z4, Z2, Z1] where
    -- Adding up the 1s
    C2, Z1 = fadd(X1, Y1, C1)
    -- Adding up the 2s
    C4, Z2 = fadd(X2, Y2, C2)
    -- Adding up the 4s
    C8, Z4 = fadd(X4, Y4, C4)
```

Internally, Syrup makes sure that the types are respected, that all of the named wires are properly defined, that the definitions are not circular in nature. Once the code is accepted, we know we have a valid circuit. But is it the one we wanted? The one we really wanted?

4.4 D for Display

The Architect insists on using the display feature to obtain a circuit diagram corresponding to the final piece of Syrup code. The generated diagram, shown in figure 8 can then be compared to the original hand-drawn solution. The lecturers then explain that both of the representations encode the same logic and that if you have one you can generate the other. This routine is performed regularly throughout the course and often more than once during a lecture as solutions are refined or improved upon.

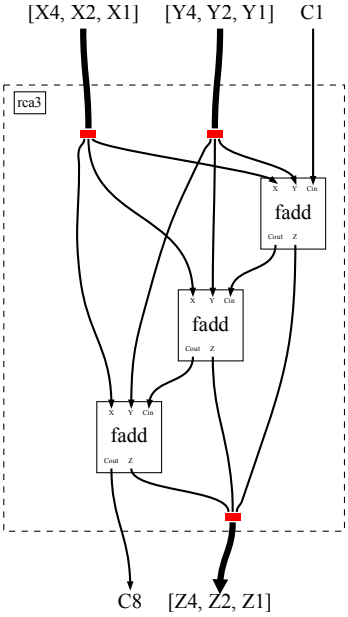

Figure 8: Displaying the `rca3` circuit

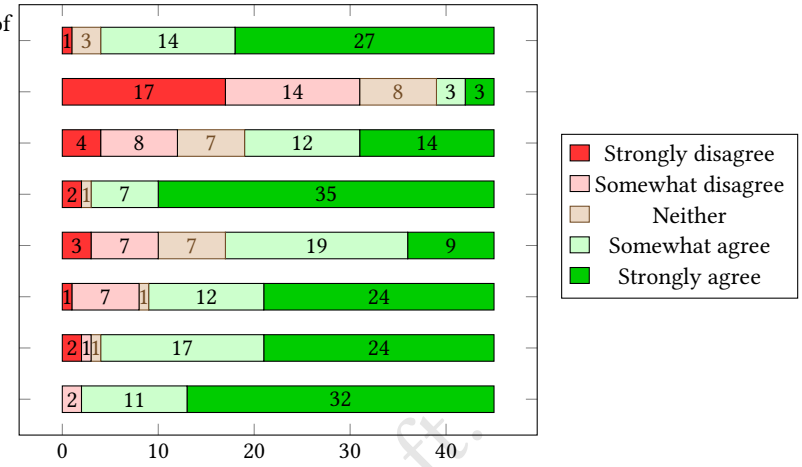

Figure 9: Survey Results

5 Results and Analysis

We aimed to understand whether or not students found the DED loop helpful in their understanding. In particular we wished to understand whether or not students prefered this approach to code first. To this end we designed a survey consistng of Likert scale questions. The survey was distributed to first and second year students that had attended semester 1 of CS106 and therefore experienced the DED loop. The start of the survey contained a description of the DED loop and its deployment. The survey was distributed via email and students were given time during class to complete the survey. Participation was entirely voluntary and ethical consent was granted by the University of Strathclyde Department of Computer and Information Sciences Ethics board.

After dissemination 45 complete responses were obtained, the responses to the Likert scale questions are displayed in figure 9. Responses were highly positive with 43 of the respondents agreeing that the approach was enging, 41 respondents agreeing that the DED loop made it easier to follow lectures and 36 agreeing that the DED loop made it easier to understand Syrup. Less students, though still a majority of 28 responses, agreed to some extent that they used the DED loop when writing their own code. A large majority of students disagreed with the statement "I would have prefered if only Syrup was used". Students were also very positive about the Co-Lecturing approach with 42 of the 45 responses agreeing that two lecturers presenting differing points of view was helpful. 41 of the students agreed that the use of colours was helpful.

Students were overwhelmingly positive about the DED approach to circuit specification. In particular students were heavily in favour of this deployment of Co-Lecturing with 42 of the students agreeing that mutiple perspectives were helpful. This directly answers RQ3, students did find co-lecturing helpful. A majority of 31 students disagreed that they would have prefered if only Syrup was used, 6 students agreed that they would have prefered a Syrup only approach. These results address RQ2 students prefer the DED loop to a code-only approach. Taken together our results answer RQ1, students did find the DED loop helpful.

While our preliminary results were very positive, we do acknowledge some limitations to our work. Our questions focussed on whether or not students found the approach helpful. As this approach was integral to our delivery of the course it would have been very difficult to tease out whether or not student understanding had been improved outside of students perceived understanding. Ideally a controlled experiment would have been used to determine the efficacy of the DED loop, as such our results are preliminary.

Ideally our survey would have had more responses. Given the number of responses it is likely that selection bias has played a role in the positive nature of our responses and that a larger survey may have shown a larger variety of opinions.

6 Conclusion and Future Work

We present evidence that students are highly positive about the DED loop approach to circuit design education. Our results show that the DED loop has merit and see areas for future work.

It would be interesting to understand whether or not students are more capable at generating the circuit diagram from a Syrup specification or vice versa. The DED loops focuses on solving via a diagram and then deriving the code. It would be interesting to understand if this forms a bidirectional understanding or simply a one way derivation technique.

Students were also very positive about the co-lecturing aspect of the DED loop. It would be interesting to see whether this approach functions as well with only one lecturer. It may be that students find it less engaging when the two lecturers cannot discuss ideas but similarly helpful in understanding.

Our work introduces Syrup, a pedagogy focused hardware description language. We also introduce the DED loop, a lecturing dynamic which helps students to bridge the gap from intuitive circuit diagrams to hardware specification code. Our approach is simple, easy to reproduce and, as our data shows, popular with students. We encourage the reader to try this approach themselves, it's DED easy.

References

[1] Marlies Baeten and Mathea Simons. 2014. Student teachers' team teaching: Models, effects, and conditions for implementation. *Teaching and Teacher Education* 41 (7 2014), 92–110. doi:10.1016/J.TATE.2014.03.010

[2] Allan Collins, John Seely Brown, Ann Holum, et al. 1991. Cognitive apprenticeship: Making thinking visible. *American educator* 15, 3 (1991), 6–11.

[3] Arthur L Costa, Bena Kallick, et al. 1993. Through the lens of a critical friend. *Educational leadership* 51 (1993), 49–49.

[4] Andrew Fagan, Alasdair Lambert, and Martin Goodfellow. 2025. Learning Programming Languages by Pantomime. In *Proceedings of the 9th Conference on Computing Education Practice (CEP '25)*. Association for Computing Machinery, New York, NY, USA, 1–4. doi:10.1145/3702212.3702213

[5] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 21–50. doi:10.1145/3437800.3439202

[6] Maria Knobelsdorf, Christoph Kreitz, and Sebastian Böhne. 2014. Teaching theoretical computer science using a cognitive apprenticeship approach. *SIGCSE 2014 - Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (2014), 67–72. doi:10.1145/2538862.2538944

[7] Alasdair Lambert, Stuart Gale, and Ezra Schoen. 2025. Double Acts for Demystifying Subroutine Calling Conventions. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 2* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 733–734. doi:10.1145/3724389.3731255

[8] Grischa Liebel, Håkan Burden, and Rogardt Heldal. 2017. For free: continuity and change by team teaching. *Teaching in Higher Education* 22 (1 2017), 62–77. Issue 1. doi:10.1080/13562517.2016.1221811

[9] Luke Moors, Andrew Luxton-Reilly, and Paul Denny. 2018. Transitioning from Block-Based to Text-Based Programming Languages. In *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 57–64. doi:10.1109/LaTICE.2018.000-5

[10] Elizabeth Patitsas. 2013. A case study of the development of CS teaching assistants and their experiences with team teaching. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '13)*. Association for Computing Machinery, New York, NY, USA, 115–124. doi:10.1145/2526968.2526981

[11] John Paxton. 2002. Live programming as a lecture technique. *J. Comput. Sci. Coll.* 18, 2 (dec 2002), 51–56.

[12] Adalbert Gerald Soosai Raj, Jignesh M Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of live-coding in learning introductory programming. In *Proceedings of the 18th koli calling international conference on computing education research*. 1–8.

[13] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 164–170. doi:10.1145/3430665.3456382

[14] Anshul Shah, Emma Hogan, Vardhan Agarwal, John Driscoll, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. 2023. An Empirical Evaluation of Live Coding in CS1. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 476–494. doi:10.1145/3568813.3600122

[15] Anshul Shah and Adalbert Gerald Soosai Raj. 2024. A Review of Cognitive Apprenticeship Methods in Computing Education Research. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1202–1208. doi:10.1145/3626252.3630769

[16] Glenn Strong, Nina Bresnihan, and Brendan Tangney. 2025. Supporting learners in the transition from block-based to text-based programming, a systematic review. *Journal of Computer Languages* 84 (2025), 101342. doi:10.1016/j.cola.2025.101342

[17] Tarık Talan, Yunus Doğan, Yusuf Kalinkara, and Veli Batdi. 2025. To block or not to block? : a meta-analysis of effectiveness of block-based programming. *Research in Science and Technological Education* 0, 0 (2025), 1–20. arXiv:https://doi.org/10.1080/02635143.2025.2593936 doi:10.1080/02635143.2025.2593936

[18] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (July 2019), 22–25. doi:10.1145/3341221

[19] David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (Boston, Massachusetts) *(IDC '15)*. Association for Computing Machinery, New York, NY, USA, 199–208. doi:10.1145/2771839.2771860

[20] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Trans. Comput. Educ.* 18, 1, Article 3 (Oct. 2017), 25 pages. doi:10.1145/3089799

[21] Chih-Chang Yu and Leon Yufeng Wu. 2024. Instructing with Cognitive Apprenticeship Programming Learning System (CAPLS) for novice computer science college freshmen: An exploration study. *Educational Technology & Society* 27 (2024), pp. 183–196. Issue 2. https://www.jstor.org/stable/48766170

[22] Daniela Zehetmeier, Axel Böttcher, and Anne Brüggemann-Klein. 2018. Designing Lectures as a Team and Teaching in Pairs. *4th International Conference on Higher Education Advances (HEAD'18)* (7 2018), 873–880. doi:10.4995/HEAD18.2018.8103