University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Emulating Glasgow's First Computer

Gerard Ward
September 14, 2018

Abstract

The English Electric DEUCE was one of the first commercially available computers in the world, but in modern times, there are limited resources available to learn more about it. The aim of this project was to create an emulator of the DEUCE, to provide people with a way of learning how one of the world's earliest computers functions. Written as a web application, this emulator allows users to operate the DEUCE as if they were using an original, but through a modern software solution. In the end, the project captured most of the functionality of the original DEUCE.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Gerard Dominic Ward Date: 20 March 2018

Contents

1 Introduction

This chapter states the motivations and aims behind this project, as well as summarising the contents of the rest of this dissertation. The DEUCE emulator built for this project allows users to operate a web application version of the English Electric DEUCE. The significance of creating this emulator will be explained in this chapter.

1.1 Motivation

In 1951, the English Electric company decided to begin building models of the Digital Electronic Universal Computing Engine, also known as the DEUCE (Vowels 2005a). This computer was based on Alan Turing's Pilot ACE computer but improved on the ACE in several ways, namely by improving upon the speed and reliability of the ACE, adding further storage and adding a large program and subroutine library. Due to these improvements, the DEUCE was considered a commercial success: in total, approximately 33 DEUCEs were created, with the first being installed in 1955. (Vowels 2005a). These were mainly installed at governmental departments, aircraft design facilities and universities. Therefore, as an early, commercially successful stored program computer, the DEUCE has a hugely important part in the history of Computing Science.

Among the universities that installed a DEUCE was the University of Glasgow (University of Glasgow 2019). In 1957, the university established Scotland's first computing lab and chose Dr. Dennis Gilles as its initial Director of Computing, as pictured in Figure 1.1. As director, Gilles was responsible for advising the university to order its first computer and so in 1958, the DEUCE became the first electric computer at a Scottish university. In addition to holding an important place in the history of Computing Science, the DEUCE is also significant in recent Scottish academic history.

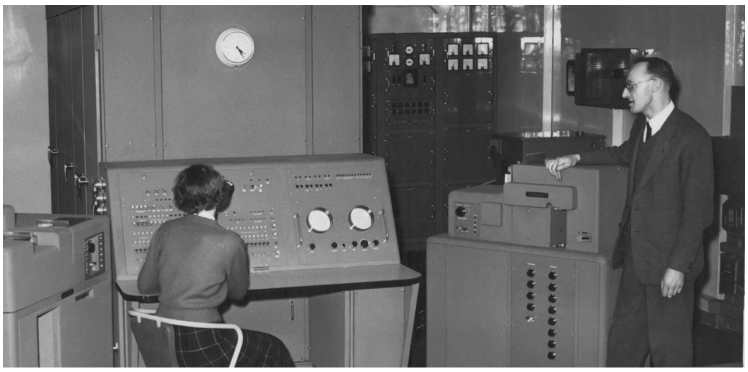


Figure 1.1: *Dr. Dennis Gilles, initial Director of Computing at the University of Glasgow, standing next to the University of Glasgow's DEUCE, c. 1957 (University of Glasgow 2017).*

However, in spite of the DEUCE's importance as an early electric computer, there remain limited resources available on the DEUCE. With all of the machines having stopped being in use since approximately the late 1960s, there are no DEUCEs left to use in the world today, shown in. At the University of Glasgow, the only remaining piece of the DEUCE installed there is one of the mercury delay lines, as shown in Figure 1.2. For this reason, it would be beneficial if there were a modern way of allowing people to operate a DEUCE in order to learn more about an important part of Computing Science history, and also to learn how to operate an example of an early computer.

Figure 1.2: *Mercury delay line amplifier from original Glasgow DEUCE. This piece of equipment sent electrical pulses to encode data inside the mercury delay lines of the DEUCE (University of Glasgow 2017).*

In modern times, it would be impractical to recreate the DEUCE physically. With the DEUCE drawing around 9kw of power and having a clock speed of 1MHz (University of Glasgow 2019), it is a highly inefficient system. Furthermore, its use of mercury delay lines would be dangerous given the potential health problems associated with exposure to mercury (Gao et al. 2017). Therefore, an alternative way of recreating the DEUCE in modern times would be through emulation. According to Smith and Nair (2005), emulation is "the process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality." The creation of a DEUCE emulator would be a modern, efficient and convenient way for the DEUCE to be remade and would solve the problem of the DEUCE being widely unavailable to people.

1.2 Aims

For this project, the key aim is to create a functioning emulator of the DEUCE. The emulator should replicate the behaviour of the original DEUCE, using the same input, processing and output methods. When completed, the emulator should allow the user to run programs on it, as if it were the original DEUCE computer. It should also replicate the user interface of the DEUCE. After the emulator has been created, it should be evaluated against the original DEUCE to discover if it can run programs written for the original computer. It is hoped that the emulator can be used as an educational tool to help people learn more about how early computers, such as the DEUCE, functioned.

1.3 Summary

The purpose of this chapter was to introduce they key motivations behind the project and what the aims of the project should be. The rest of this dissertation will be structured as follows:

- Chapter 2 will examine the background research carried out on how the DEUCE functioned. It will also examine similar early computer emulators and look into what made these emulators examples of good or bad emulators.
- Chapter 3 will examine the requirements gathering process for the emulator and the choices behind the target platform chosen for the project.
- Chapter 4 will examine the initial design choices made for the emulator and the decisions which informed the design of both the graphical and system design of the computer.
- Chapter 5 will discuss the implementation process of the project and how the frontend and backend of the project was implemented.
- Chapter 6 will examine how the emulator was evaluated against the original DEUCE, to discover how successful it was in emulating its behaviour.
- Chapter 7 will reflect on possible future steps for the project and what could have been improved about the emulator.

2 Background

This chapter discusses background research carried out on the DEUCE and on the creation of emulators.

2.1 Learning how the DEUCE functions

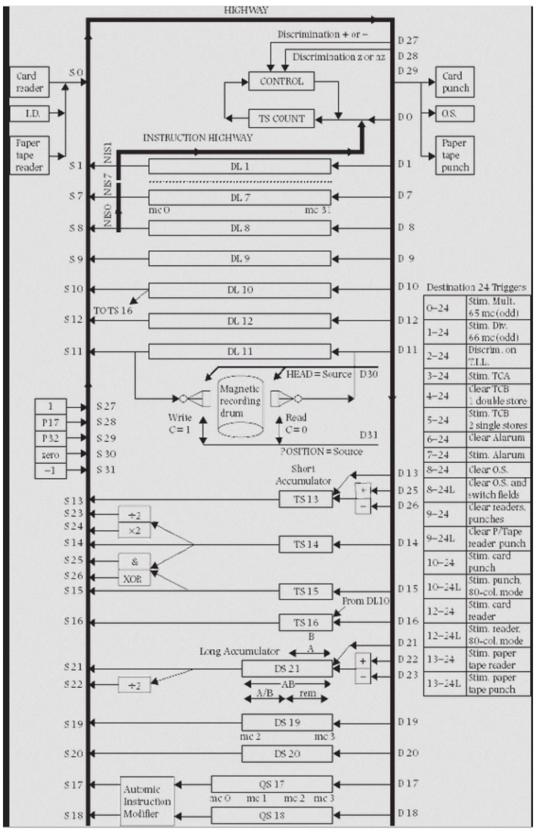


Figure 2.1: *Architecture diagram of the DEUCE (Vowels 2005a). This image shows the memory structure of the delay lines, registers and special memory locations used to carry out special functions in the DEUCE.*

Firstly, it was important to learn how the DEUCE functioned as a computer. The architecture of the DEUCE is shown in Figure 2.1. As an early computer, it is extremely different to modern computers in several ways. For example, for input, the instructions read in by the DEUCE were essentially a series of move instructions from a memory source to a memory destination. Each instruction was a 32-bit binary word (Wetherfield 2010). Instructions could be read in either via a card reader, which read in lines of instructions on special cards, or through the Input Staticiser, a special row of switches which allowed for single word input. Several special source and memory locations specified special functions, such as arithmetic functions, discrimination functions etc., and through using these special memory locations, the computer could be coded to run programs.

The DEUCE stored instructions in delay lines and registers. It consisted of:

- 32 mercury delay lines, each storing 32 words.
- 4 Temporary Store registers, each storing 1 word.
- 3 Double Store registers, each storing 2 words.
- 2 Quadruples Store registers, each storing 4 words.

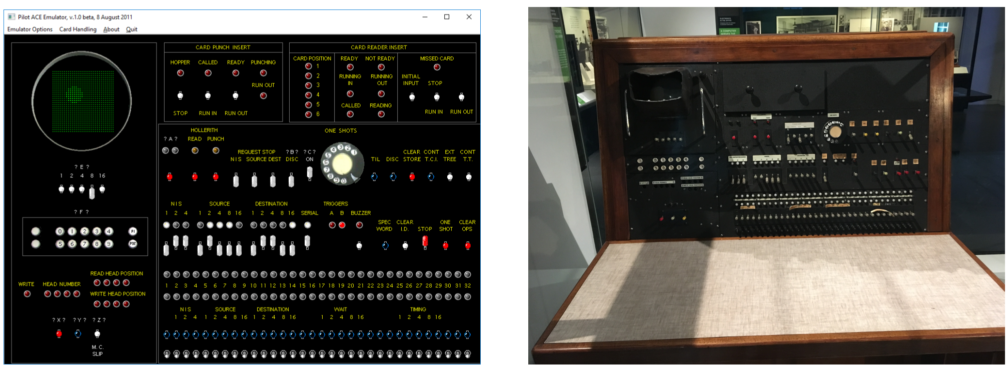
For output, the DEUCE could display information through the Output Staticiser, which was a row of lights that could display a single 32-bit word, or through a screen which could display a matrix of bits.

The complex mechanics of the DEUCE meant that the architecture of the computer had to be studied carefully in order to fully understand how to emulate its behaviour.

2.2 Comparison of early computer emulators

To gain a better understanding of how early computers worked, several emulators of other early computers were used. This gave good insight into how similar early computers functioned and explore the design choices made by the creators of these emulators.

2.2.1 Pilot ACE Emulator



(a) Pilot ACE Emulator from pilotaceonline.com (Green 2011), accessed using Wayback Machine (The Internet Archive 2019). This image shows the emulator console featuring switches and lights used for input and output respectively. It also shows the toolbar used for entering CRD "punch card" files and a monitor for displaying output.

(b) Image of console from real Pilot ACE, taken at Science Museum, London. Emulator shown in Figure 2.2a strongly resembles real front panel of the machine.

Figure 2.2: *Images of Pilot ACE emulator by David Green and the original Pilot ACE machine at the Science Museum, London.*

As described by Vowels (2005b), the Pilot ACE was one of the earliest stored-program computers and was based on a design by Alan Turing, running its first program in 1950. The DEUCE was a commercial version of this machine, so there are several similarities between the two computers. For example, both machines can read input from a Hollerith card reader or from the input dynamiciser keys on the front panel of the machine. The main memory of the Pilot ACE consisted of 11 32-bit delay lines, 5 32-bit temporary stores and 2 64-bit double stores.

This emulator, as seen in Figure 2.2a is a very faithful recreation of the original Pilot ACE, as seen in Figure 2.2b. The panel is a replica of the original panel of the Pilot ACE. To recreate some of the physical actions of operating the Pilot ACE, such as inserting a card into the card reader, the author has instead provided a toolbar which allows some of these features to be carried out.

Therefore, the Pilot ACE emulator is a very useful model on which to base a DEUCE emulator. Given the similarities in how the Pilot ACE and DEUCE computers functioned, this emulator provided some very good ideas about how to implement a DEUCE emulator. As the DEUCE was a commercial version of this machine, many of the features present in this emulator can be reused in a DEUCE emulator, such as the input dynamiciser and single shot key. While the GUI is an accurate representation of the original Pilot ACE interface, several of the features of the original Pilot ACE appear unknown to the author. For example, the purposes of the keys surrounded by question marks, such as ? E ?, are unknown to the creator. Overall, it would be worthwhile to base several features of a DEUCE emulator on this emulator.

2.2.2 The EDSAC Simulator

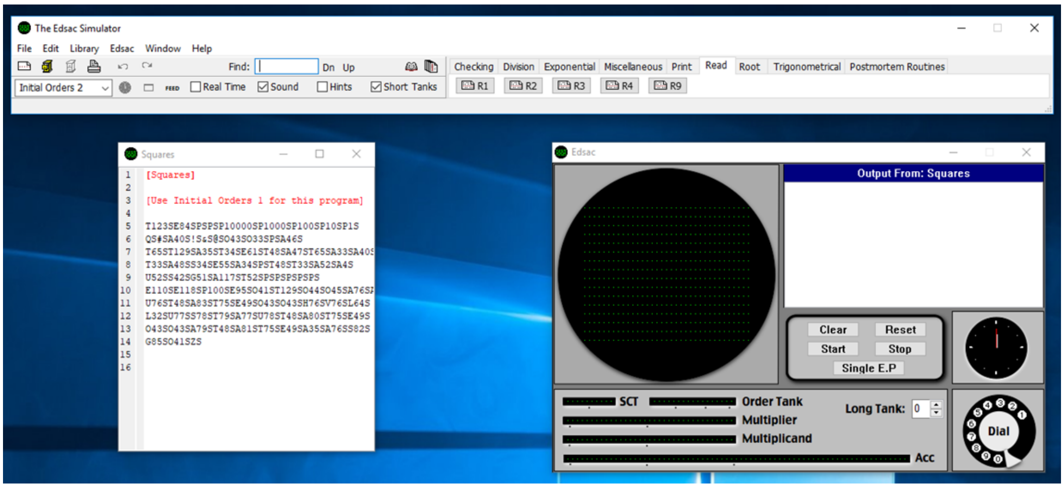


Figure 2.3: EDSAC Simulator from the University of Warwick (Campbell-Kelly 2016). Image shows the toolbar, program text and simulator display.

Like the Pilot ACE and the DEUCE, the EDSAC was another early stored program computer. Its functionality is described by Barron (2011). Running its first program in 1949, it used 32 mercury delay lines for main memory, each storing 32 words of 18 bits. For input, it used a paper-tape reader operating at 50 characters per second and output was achieved through a Creed teleprinter. Control can be achieved through five buttons: clear, reset, start, stop and single E.P. Single E.P functions similarly to the single shot key in the DEUCE, by allowing a program to be executed one instruction at a time.

As shown in Figure 2.3, the EDSAC emulator has three main components: the toolbar, the program text and the simulator display. The toolbar allows control over the features of the EDSAC. For example, it allows files to be read in and allows the user to switch between Initial

Orders 1 and Initial Orders 2. The program text provides the series of instructions to be executed for a program that is being read in. The simulator display shows the output from the programs being executed and allows control over programs through the five main control buttons.

Overall, while there are bigger differences between the DEUCE and the EDSAC than the DEUCE and the Pilot ACE, this is still a useful emulator on which to model some features of the DEUCE. Both computers share common features such as using a reader to take input from files and using delay lines for main memory. One useful feature of this emulator compared to the Pilot ACE emulator is its output display. Rather than outputting to a new file, it is useful to have a console that displays output instead. This would be a good feature to have in a DEUCE emulator, particularly one running as a web app. Therefore, while several features would need to change, there are some features of this emulator that would be useful to base a DEUCE emulator on.

2.2.3 Manchester Baby Simulator

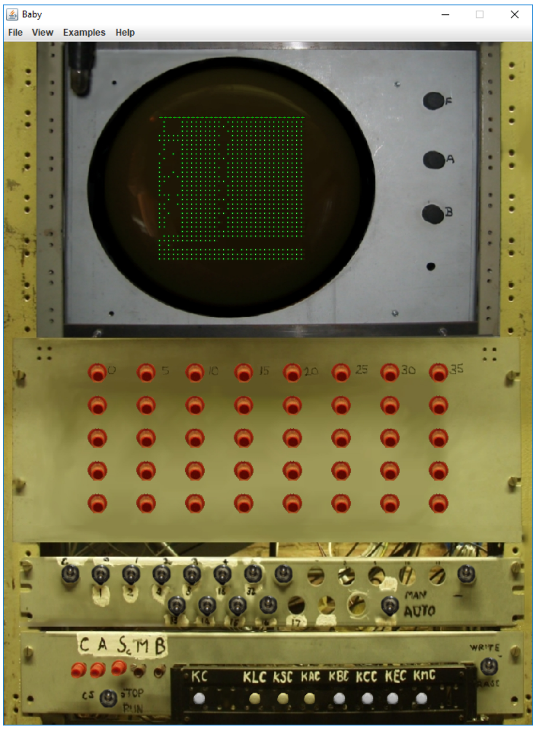


Figure 2.4: *Manchester Baby Simulator by David Sharp (Sharp 2008a). Image shows the photo-realistic interface of the emulator and its forms of input, as well as its monitor for output.*

The Manchester Baby was another early computer, which has its functionality described by Sharp (2008b). It was the first electronic stored program computer in the world, running its first program in 1948 before the Pilot ACE and the EDSAC. Its memory is made up of 32 store lines, each containing 32 bits. In one store line, bits 0–4 represent the operand line, i.e. the number of the line that the instruction will operate on when executed, and bits 13–15 represent the function number, which is essentially the instruction opcode.

The control of the Manchester Baby contains two values: the control instruction (CI) and the present instruction (PI). The CI contains the line number of the instruction executed previously. The PI contains the line representing the instruction currently being executed. Input can also be controlled using the switches on the front panel. Output can be displayed through the monitor above the control panel.

Regarding this emulator of the Manchester Baby, the creator chose to make it photo realistic, so it resembles the interface of the original Manchester Baby. While this is a faithful recreation, the interface feels slightly more intimidating to use than the other emulators used so far. For this reason, it would probably be better to recreate the DEUCE using a cleaner interface rather than faithfully restoring the interface as above. The toolbar of this emulator provides control so the user can load different example programs and view the different parts of the computer, such as the store, the control, the accumulator and the disassembler. The examples tab is a useful way of loading in new programs easily so new users can quickly learn how to use the machine. It would be beneficial to replicate a feature such as this in a DEUCE emulator.

2.3 Summary

In carrying out background research, this allowed for the discovery of how the DEUCE functioned compared to modern computers and what makes for a good emulator of an early computer. Overall, the design of the DEUCE emulator for this project would be similar to the Pilot ACE emulator, as the two original computers shared much of the same design. Using this research, this allowed for the creation of formal requirements as specified in Chapter 3.

3 Analysis/Requirements

This chapter examines the process of analysing how to create the DEUCE emulator. The scope of the project is discussed, as well as the requirements gathered to create the emulator.

3.1 Platform selection

The original plan for this project was to create a DEUCE emulator for mobile platforms, but after considering several factors concerning the feasibility of creating a mobile emulator, it was agreed instead to make a web application. Firstly, it would have been extremely difficult to recreate a faithful DEUCE emulator on mobile devices given the limitations of mobile device screen sizes. For example, the DEUCE's Input Dynamisciser was a row of 32 switches, so trying to create a good interface for this on a mobile device would have been beyond the scope of this project.

Furthermore, creating this emulator as a web application is a modern solution to making the DEUCE accessible to a wide number of people. By hosting the emulator on a website, users can access it through their web browser on any device, including desktop or mobile. Therefore, the decision to change the scope of the project early on was important in order to recreate the DEUCE as faithfully as possible.

3.2 Requirements Gathering

Requirements elicitation was an important early step in the Analysis stage of the project. This helped to define what could feasibly be completed within the scope and time frame of the project. Initial general requirements were firstly created. They are as follows:

1. Replicate physical features of the DEUCE, such as delay lines, card reader etc., within a modern software solution.
2. Have a graphical user interface resembling that of the original DEUCE
3. Allow users to interact with input switches, most likely through a pointing device.
4. Read in programs written as "punch cards".
5. Execute programs in the same way as the original DEUCE.
6. Display output through representations of lights on console.
7. Be intuitive enough for those familiar with the DEUCE to be able to operate it.

While these requirements sufficed as early, broad requirements, it was necessary to then refine them and categorise them into functional and non-functional requirements.

3.2.1 Functional Requirements

The functional requirements of the project define what features DEUCE emulator should have. These were categorised into Must Have, Should Have, Could Have and Want to Have requirements, using the MoSCoW method (Ashmore and Runyan 2014). They are as follows:

Must Have:

1. A row of 32 Input Dynamiciser switches, to allow for single instruction and data input.
2. A row of 32 Input Dynamiciser lights, to display the status of the Input Dynamiciser.
3. A row of 32 Output Staticiser lights, to display output from instructions.
4. A memory storage system comprised of 12 Delay Lines, 4 Temporary Stores, 3 Double Stores and 2 Quadruple Stores. These must store the following:
   (a) Delay Lines must store 32 32-bit words each.
   (b) Temporary Stores must store 1 32-bit word each.
   (c) Double Stores must store 2 32-bit words each.
   (d) Quadruple Stores must store 4 32-bit words each.
5. A clock to simulate delay line acoustic pulses, which are needed to access delay line memory.
6. An instruction decoder that calculates the NIS, Source, Destination, Wait, Timing and Go values of an instruction.
7. A single shot switch to execute a single instruction.
8. A graphical user interface resembling that of the original DEUCE.
9. A series of special Source and Destination addresses, as follows:
   (a) Source 0 - Take user input from Input Dynamiciser.
   (b) Source 23 - Divide contents of TS14 by 2 and place in selected destination address.
   (c) Source 24 - Multiply contents of TS14 by 2 and place in selected destination address.
   (d) Source 25 - LOGICAL AND contents of TS14 and TS15 and place in selected destination address.
   (e) Source 26 - EXCLUSIVE OR contents of TS14 and TS15 and place in selected destination address.
   (f) Source 27 - Place 1 (UNITY) in selected destination address.
   (g) Source 28 - Place 2^{16} in selected destination address.
   (h) Source 29 - Place 2^{31} in selected destination address.
   (i) Source 30 - Place 0 (ZERO) in selected destination address.
   (j) Source 31 - Place -1 in selected destination address.
   (k) Destination 25 - Add the contents of the Source address to TS13 and store result in TS13.
   (l) Destination 26 - Subtract the contents of the Source address from TS13 and store result in TS13.
   (m) Destination 27 - Discriminate between positive and negative numbers.
   (n) Destination 28 - Discriminate between zero and non-zero numbers.
   (o) Destination 29 - Display output on Output Staticiser.

Should Have:

1. A single shot dial to execute a given number of single shots in succession.
2. A card reader to take external input from a pre-written DEUCE program.
3. A card punch to write programs to external files.
4. A Clear ID switch to clear the Input Dynamiciser lights to 0.
5. A Clear Ops switch to clear the Output Staticiser lights to 0.

6. A Clear Store switch to clear all data held in delay line storage.
7. A Read and Single Read switch to read in punch cards.
8. A Punch switch to write to output files.
9. An Initial Input switch to read a new program.

Could Have:

1. A user manual to instruct users on how to operate the DEUCE.
2. A more convenient way of viewing the current memory contents of the DEUCE. While not an original feature of the DEUCE, this could help new users to view the state of their programs.

Want to have:

1. A monitor to display output. This requirement would be extremely challenging to implement within the scope of this project and so will not be included this time.
2. A subroutine library. The original DEUCE came with a large library of subroutines to be used by DEUCE programmers but this would be too much to implement within the time frame of the project.

3.2.2 Non-functional Requirements

The non-functional requirements of the project describe how the emulator should work, rather than describing what the emulator will do.

1. The emulator should be easy enough to understand how to operate for those familiar with the DEUCE.
2. It should be intuitive so that new users can learn how to operate a DEUCE.
3. It should be designed mainly for desktop computer users but also be available on other devices such as tablets and mobile phones.
4. It should be maintainable so that development can be continued in future.
5. It should be able to be used as a tool to aid learning about early computers.

3.3 Summary

This chapter gives an overview of the steps taken during the Analysis stage of the development of the project. The decision to change the project scope from being a mobile development project to a web development project was highly important, as it showed the steps taken to calculate the feasibility of the project on web compared to mobile. This stage was also key in the development process as it involved the creation of formal requirements for the project. Following these requirements would prove highly important for the rest of the project.

4 Design

This chapter examines the design of the DEUCE emulator. The design of the emulator can broadly be split into two parts: the system architecture and the user interface. The design of the system architecture is concerned with the logical design of the components which make up the computer, while the user interface design examines the graphical aspect of the emulator and how to present it to users.

4.1 System Architecture

Recreating the system architecture of the DEUCE entirely within software was an important step in the design process. With much of the system architecture of the original DEUCE being physical, it was necessary to consider abstract alternatives to how the system could be imitated entirely within software. The main design concepts concerning the DEUCE emulator are detailed within this section.

4.1.1 Memory Storage

The creation of the memory storage of the DEUCE emulator presented several design questions at an early stage. Firstly, the DEUCE had two levels of main memory: delay lines and registers. Access to each of these levels was different as the delay line used clock pulses to access data, whereas for Temporary Stores, access was instant. Delay lines could also hold a vast amount of words compared to registers. Furthermore, within the registers themselves, there were varying sizes between Temporary Stores, Double Stores and Quadruple Stores. Therefore, a solution was needed to represent these different levels of memory entirely within software.

An initial design solution to this problem was to use a dictionary data structure to hold each type of storage. With keys for each type of storage, this would allow a way of accessing each level of memory entirely within software. For every key, the value pair would be an array holding the number of words each type of storage could hold. For example, keys "DL1" to "DL12" would represent the 12 Delay Lines of the DEUCE and hold an array of 32 words, while "TS13" being Temporary Store 13 could hold 1 word.

Another proposed solution to this was to simply store all levels of memory within one array of "Memory" objects. Each memory object could hold a given number of words and so could be used to represent each type of storage. The index of each type of storage would have to be tracked rather than named but given the labels of each type of storage in the architecture diagram of the original DEUCE (DL1, TS13, QS17 etc.) seen in Figure 2.1, it would not be difficult to do so. The number of each type of storage from the labels could be simply used as indexes within the array.

For this reason, it was decided that an array would be used over a dictionary. Ultimately, using a dictionary would overcomplicate the design of memory as software and an array would be a more efficient and effective way of keeping track of memory.

4.1.2 Delay Line Architecture

Following on from the overall design of the memory storage of the DEUCE, another large design challenge regarding the system architecture of the DEUCE emulator was the design of access to the delay lines. In early computers, delay lines provided access to data using mercury and acoustic pulses. With the data constantly flowing in the mercury tanks, it was necessary to keep track of where the data was in the delay lines using minor and major cycles. One minor cycle represented one pulse, while one major cycle represented how long it took for data to flow around an entire delay line. This is why DEUCE instructions feature Wait and Timing numbers, as these numbers calculate where in the delay lines certain data will be.

Within a web application version of the DEUCE, the design of the delay line storage of the DEUCE posed a problem. Without physical mercury delay line storage available to this emulator, a suitable substitute was needed to represent this behaviour. To emulate the behaviour of minor and major cycles of delay lines, it was decided that there should be a clock to measure minor cycles. This would be designed as a counter that would increment by 1 every minor cycle. Using this counter, it would be possible to track where data was being held in each delay line. Essentially, this counter would act like a pointer to an array. This solved the design problem of using an alternative to physical delay line storage.

4.1.3 Instruction Processing

The design of how instructions would be processed in the DEUCE emulator was another challenging task. Each memory location in the DEUCE could hold one 32-bit word that could represent an instruction or data. A 32-bit instruction was broken down into several parts: the NIS, the Source, the Destination, the Characteristic, the Wait, the Timing and the Go bit. In an emulator, this would mean parsing a 32-bit number to retrieve each part of the instruction. Using these parts of the instruction, a word could be moved from one location in memory to another.

To help visualise the instruction processing of the emulator more clearly, a UML diagram was created in the early stages of design following rules established by Torre et al. (2018), shown in Figure 4.1. This diagram was based on the description of how the DEUCE functioned in Vowels (2005a)The diagram shows the two forms of input a user can use to provide the emulator with data: either using a pre-written punch card, or the input switches on the console. If a punch card is entered, the card will be read in and the console will show the status of the card, either as being "Called", "Ready" or as "Being read". After the card has been read in, the Initial Input switch can be pressed to load the card into storage, so that program execution can begin. Storage can be cleared using the "clear store" switch.

For input via the Input Dynamiciser switches, the current state of the Input Dynamiciser will be displayed on the ID lights. These lights can be reset to 0 using the "Clear ID" switch. Each ID switch represents one bit of a 32-bit word. When the user wants to enter an instruction, they would press the "One Shot" switch to execute said instruction.

Regarding output, the DEUCE should display output on the Output Staticiser lights. These lights display the output of the last instruction executed as a 32-bit reverse binary word. The original DEUCE could also write output to punched cards or to the system's monitor. These features may not be implemented in this version of the DEUCE but it was important to show that these features were part of the original DEUCE architecture.

From using this diagram, the flow of program execution in the DEUCE can be seen. It identifies the key components of the DEUCE based on the functional requirements from Chapter 3.

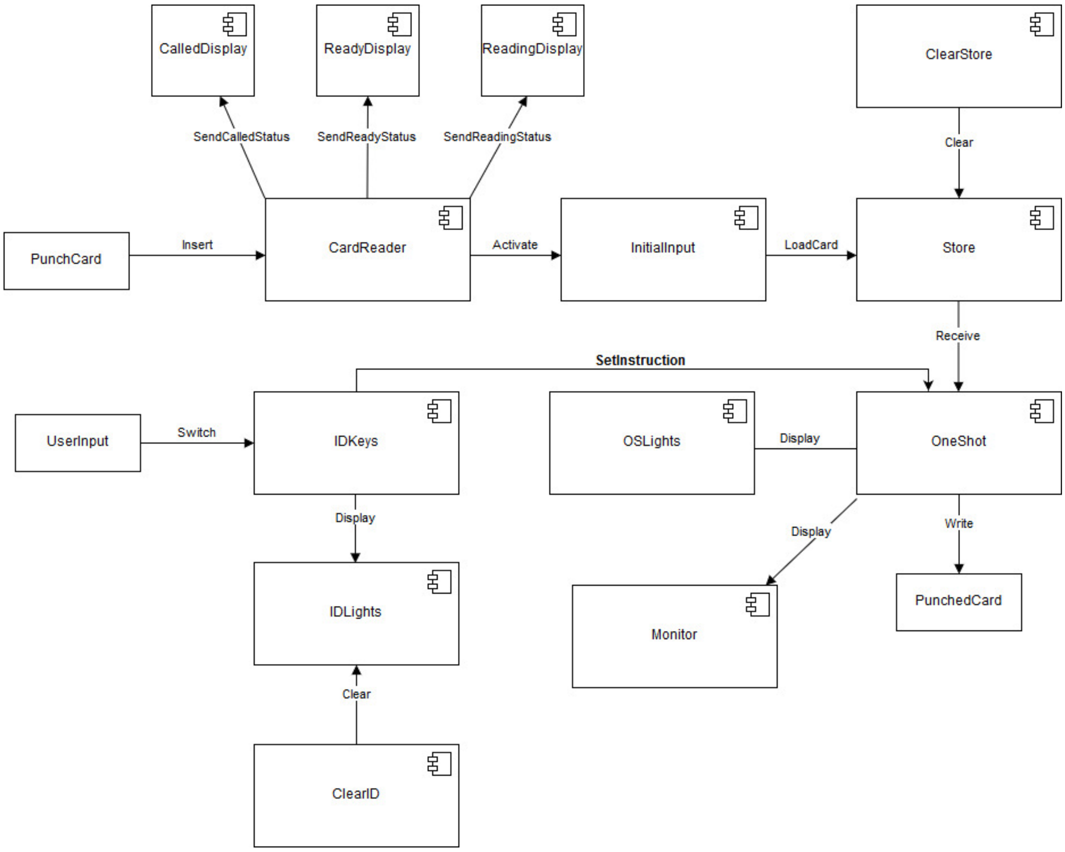


Figure 4.1: *UML diagram describing the system architecture. Rules for creating the diagram were followed from Torre et al. (2018). Input can be given via 2 sources, which is then executed via a series of single shots. On the original DEUCE, output could be given in three forms, so the project would aim to recreate this functionality.*

4.2 User Interface

One of the key differences between the interface of the DEUCE emulator and the original DEUCE interface is the lack of hardware. Given that this project would be written entirely as software, one of they key challenges would be replicating the physical features of the DEUCE within software and representing them correctly to the user. To solve this problem, visual metaphors would need to be used to allow the user to feel as though they were using the original DEUCE.

The DEUCE is comprised of many switches and lights. It also features a dial to set the number of single shots to be executed in succession. While it is not possible to physically recreate these components, it is important to give the user correct feedback when operating them. To solve this challenge, it was decided that switches and lights would need to be animated properly to give the user good feedback on their actions.

Another large difference between a software-only DEUCE and a real DEUCE is the absence of a card reader and card punch. In the original DEUCE, a Hollerith card reader was used to read in pre-written programs and a card punch was used to physically print output on card. To replicate a card reader in a web application, a good solution would be to provide a file upload button and have users browse files to mimic inserting a program into the DEUCE. A better solution would be to use a web framework that supports "drag and drop" features, which would allow a user to "drop" a card into the card reader.

For these reasons, it was necessary to select a web framework that would support these graphical needs. The selection of an appropriate web framework is later discussed in Chapter 5.

4.3 Summary

The design stage of the project was important for planning how each aspect of the project would be created. During this stage, major challenges in the design of each major component of the DEUCE were tackled, and it was decided how these challenges would be solved without reference to specific implementation. Given the complexity of the design of the original DEUCE, it was necessary to think of modern alternatives that could work within a web application. Overall, the steps taken here were very useful for the Implementation stage of the project.

5 Implementation

This chapter discusses the implementation of the DEUCE emulator in terms of both front-end and back-end development. It also discusses the choice of web framework and the decisions behind this choice.

5.1 Web framework choice

The emulator was written as a web application because hosting it on the internet would allow a large number of people to access it from almost anywhere in the world.

5.1.1 Angular

5.1.2 React

5.1.3 Vue.js

5.2 Front-end development

5.3 Back-end development

5.4 Guidance

You can't talk about everything. Cover the high level first, then cover important, relevant or impressive details.

5.5 General points

These points apply to the whole dissertation, not just this chapter.

5.5.1 Figures

Always refer to figures included, like Figure 5.1, in the body of the text. Include full, explanatory captions and make sure the figures look good on the page. You may include multiple figures in one float, as in Figure ??, using `subcaption`, which is enabled in the template.

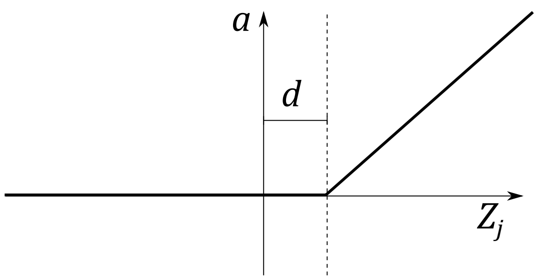


Figure 5.1: *In figure captions, explain what the reader is looking at: "A schematic of the rectifying linear unit, where* a *is the output amplitude,* d *is a configurable dead-zone, and Z_j is the input signal", as well as why the reader is looking at this: "It is notable that there is no activation* at all *below 0, which explains our initial results."* ***Use vector image formats (.pdf) where possible****. Size figures appropriately, and do not make them over-large or too small to read.*

5.5.2 Equations

Equations should be typeset correctly and precisely. Make sure you get parenthesis sizing correct, and punctuate equations correctly (the comma is important and goes *inside* the equation block). Explain any symbols used clearly if not defined earlier.

For example, we might define:

$$\hat{f}(\xi) = \frac{1}{2}\left[\int_{-\infty}^{\infty} f(x)e^{2\pi i x \xi}\right], \tag{5.1}$$

where $\hat{f}(\xi)$ is the Fourier transform of the time domain signal $f(x)$.

5.5.3 Algorithms

Algorithms can be set using `algorithm2e`, as in Algorithm 1.

Data: $f_X(x)$, a probability density function returing the density at x.
σ a standard deviation specifying the spread of the proposal distribution.
x_0, an initial starting condition.
Result: $s = [x_1, x_2, \ldots, x_n]$, n samples approximately drawn from a distribution with PDF $f_X(x)$.

```
begin
    s ⟵ []
    p ⟵ f_X(x)
    i ⟵ 0
    while i < n do
        x' ⟵ N(x, σ^2)
        p' ⟵ f_X(x')
        a ⟵ p'/p
        r ⟵ U(0, 1)
        if r < a then
            x ⟵ x'
            p ⟵ f_X(x)
            i ⟵ i + 1
            append x to s
        end
    end
end
```

Algorithm 1: The Metropolis-Hastings MCMC algorithm for drawing samples from arbitrary probability distributions, specialised for normal proposal distributions $q(x'|x) = \mathcal{N}(x, \sigma^2)$. The symmetry of the normal distribution means the acceptance rule takes the simplified form.

5.5.4 Tables

If you need to include tables, like Table 5.1, use a tool like https://www.tablesgenerator.com/ to generate the table as it is extremely tedious otherwise.

5.5.5 Code

Avoid putting large blocks of code in the report (more than a page in one block, for example). Use syntax highlighting if possible, as in Listing 5.1.

Table 5.1: *The standard table of operators in Python, along with their functional equivalents from the* `operator` *package. Note that table captions go above the table, not below. Do not add additional rules/lines to tables.*

Operation	**Syntax**	**Function**
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`div(a, b)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusive Or	`a ^b`	`xor(a, b)`
Bitwise Inversion	`~a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identity	`a is b`	`is_(a, b)`
Identity	`a is not b`	`is_not(a, b)`
Indexed Assignment	`obj[k] = v`	`setitem(obj, k, v)`
Indexed Deletion	`del obj[k]`	`delitem(obj, k)`
Indexing	`obj[k]`	`getitem(obj, k)`
Left Shift	`a <<b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Negation (Arithmetic)	`- a`	`neg(a)`
Negation (Logical)	`not a`	`not_(a)`
Positive	`+ a`	`pos(a)`
Right Shift	`a >>b`	`rshift(a, b)`
Sequence Repetition	`seq * i`	`repeat(seq, i)`
Slice Assignment	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Slice Deletion	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Slicing	`seq[i:j]`	`getitem(seq, slice(i, j))`
String Formatting	`s % obj`	`mod(s, obj)`
Subtraction	`a - b`	`sub(a, b)`
Truth Test	`obj`	`truth(obj)`
Ordering	`a <b`	`lt(a, b)`
Ordering	`a <= b`	`le(a, b)`

```
def create_callahan_table(rule="b3s23"):
   """Generate the lookup table for the cells."""
   s_table = np.zeros((16, 16, 16, 16), dtype=np.uint8)
   birth, survive = parse_rule(rule)

   # generate all 16 bit strings
   for iv in range(65536):
      bv = [(iv >> z) & 1 for z in range(16)]
      a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p = bv

      # compute next state of the inner 2x2
      nw = apply_rule(f, a, b, c, e, g, i, j, k)
      ne = apply_rule(g, b, c, d, f, h, j, k, l)
      sw = apply_rule(j, e, f, g, i, k, m, n, o)
      se = apply_rule(k, f, g, h, j, l, n, o, p)

      # compute the index of this 4x4
      nw_code = a | (b << 1) | (e << 2) | (f << 3)
      ne_code = c | (d << 1) | (g << 2) | (h << 3)
      sw_code = i | (j << 1) | (m << 2) | (n << 3)
      se_code = k | (l << 1) | (o << 2) | (p << 3)

      # compute the state for the 2x2
      next_code = nw | (ne << 1) | (sw << 2) | (se << 3)

      # get the 4x4 index, and write into the table
      s_table[nw_code, ne_code, sw_code, se_code] = next_code

   return s_table
```

Listing 5.1: *The algorithm for packing the* 3×3 *outer-totalistic binary CA successor rule into a* $16 \times 16 \times 16 \times 16$ *4 bit lookup table, running an equivalent, notionally 16-state* 2×2 *CA.*

6 Evaluation

How good is your solution? How well did you solve the general problem, and what evidence do you have to support that?

6.1 Comparison of features with original DEUCE

6.2 Guidance

- Ask specific questions that address the general problem.
- Answer them with precise evidence (graphs, numbers, statistical analysis, qualitative analysis).
- Be fair and be scientific.
- The key thing is to show that you know how to evaluate your work, not that your work is the most amazing product ever.

6.3 Evidence

Make sure you present your evidence well. Use appropriate visualisations, reporting techniques and statistical analysis, as appropriate.

If you visualise, follow the basic rules, as illustrated in Figure 6.1:

- Label everything correctly (axis, title, units).
- Caption thoroughly.
- Reference in text.
- **Include appropriate display of uncertainty (e.g. error bars, Box plot)**
- Minimize clutter.

See the file `guide_to_visualising.pdf` for further information and guidance.

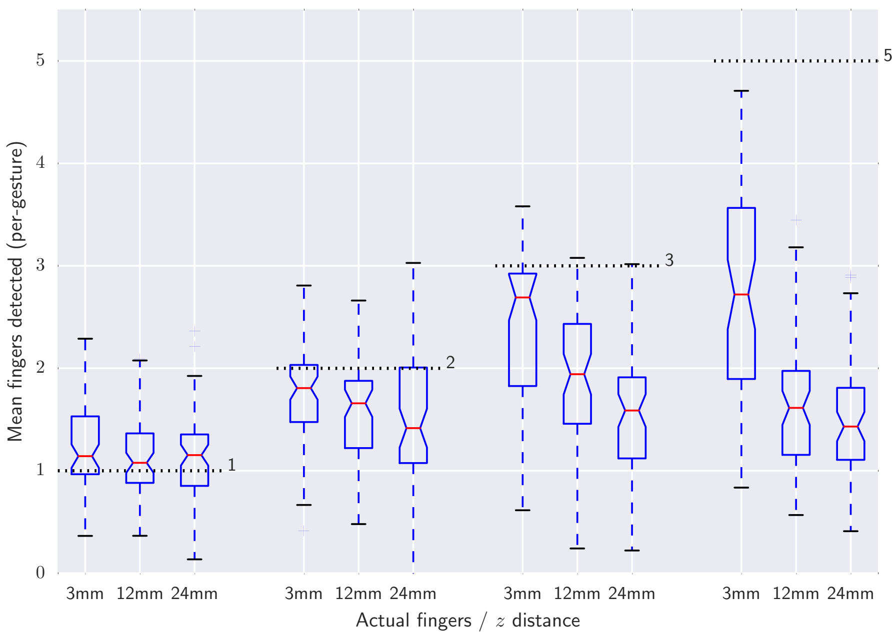


Figure 6.1: *Average number of fingers detected by the touch sensor at different heights above the surface, averaged over all gestures. Dashed lines indicate the true number of fingers present. The Box plots include bootstrapped uncertainty notches for the median. It is clear that the device is biased toward undercounting fingers, particularly at higher z distances.*

7 Conclusion

Summarise the whole project for a lazy reader who didn't read the rest (e.g. a prize-awarding committee).

7.1 Guidance

- Summarise briefly and fairly.
- You should be addressing the general problem you introduced in the Introduction.
- Include summary of concrete results ("the new compiler ran 2x faster")
- Indicate what future work could be done, but remember: **you won't get credit for things you haven't done**.

A Appendices

Typical inclusions in the appendices are:

- Copies of ethics approvals (required if obtained)
- Copies of questionnaires etc. used to gather data from subjects.
- Extensive tables or figures that are too bulky to fit in the main body of the report, particularly ones that are repetitive and summarised in the body.
- Outline of the source code (e.g. directory structure), or other architecture documentation like class diagrams.
- User manuals, and any guides to starting/running the software.

Don't include your source code in the appendices. It will be submitted separately.

7 Bibliography

S. Ashmore and K. Runyan. *Introduction to Agile Methods*. Addison Wesley Professional, 2014.

D. Barron. EDSAC: A Programmer Remembers. In *The Computer Journal*, volume 54, pages 139–142, 2011.

M. Campbell-Kelly. Edsac Simulator, 2016. URL `https://www.dcs.warwick.ac.uk/~edsac/`.

Z. Gao, X. Ying, J. Yan, J. Wang, S. Cai, and C. Yan. Acute mercury vapor poisoning in a 3-month-old infant: A case report. In *Clinica Chimica Acta*, volume 465, pages 119–122, 2017.

D. Green. The Pilot ACE Emulator, 2011. URL `https://web.archive.org/web/20111204234452/http://www.pilotaceonline.com/index.html`.

D. Sharp. Manchester Baby Simulator, 2008a. URL `https://www.davidsharp.com/baby/index.html`.

D. Sharp. Manchester Baby Simulator User Guide, 2008b. URL `https://www.davidsharp.com/baby/babyuserguide.pdf`.

J. E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Publishers, 2005.

The Internet Archive. Internet Archive: Wayback Machine, 2019. URL `https://archive.org/web/`.

D. Torre, Y. Labiche, M. Genero, and E. Maged. A systematic identification of consistency rules for uml diagrams. In *Journal of Systems and Software*, volume 144, pages 121–142, 2018.

University of Glasgow. DEUCE mercury delay line amplifier, 2017. URL `https://www.youtube.com/watch?v=Jf_j3oE5BvE`.

University of Glasgow. What the DEUCE? Scotland's first computer, 2019. URL `https://www.gla.ac.uk/schools/computing/aboutus/60years/scotlandsfirstcomputer`.

R. A. Vowels. The DEUCE - a user's view. In *Alan Turing's Automatic Computing Engine: The Master Codebreaker's Struggle to build the Modern Computer*, pages 299–325, 2005a.

R. A. Vowels. The Pilot ACE: from concept to reality. In *Alan Turing's Automatic Computing Engine: The Master Codebreaker's Struggle to build the Modern Computer*, pages 233–258, 2005b.

M. Wetherfield. Personal Recollections of Programming DEUCE in the Late 1950s. In *The Computer Journal*, volume 53, pages 1535–1549, 2010.