MICROPROGRAMMED MACHINE SIMULATION USING REACT

Lappeenranta–Lahti University of Technology LUT

Bachelor's thesis, Software Engineering

2023

Author: Jani Heinikoski

Supervisor: University Lecturer Olli-Pekka Hämäläinen

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

School of Engineering Science

Software Engineering

Jani Heinikoski

Microprogrammed machine simulation using React

Bachelor’s thesis

2023

42 pages, 6 figures, 2 tables and 3 appendices

Examiners: University Lecturer Olli-Pekka Hämäläinen and Associate Professor Antti Knutas

Keywords: simulator application, design science, React, microprogramming, e-learning

This bachelor’s thesis focuses on the study of a microprogrammed machine. It is a simplistic microprogrammable computer used to teach basics of microprogramming in the Foundations of Computer Science (FoCS) -course at LUT University. The main research problem this thesis solves is designing and implementing an accessible way to simulate the microprogrammed machine.

By following a design science research methodology, a web simulator application is designed and implemented in this thesis to solve the main research problem. The goal is to enable users to write arbitrary microprograms, execute them and study how the different components of the computer react in a virtual environment. A Windows operating system dependent simulator application with known compatibility issues already exists. To be easily accessible by most students, a platform independent web simulator application is produced.

The application is intended to be used as a supporting e-learning tool during future FoCS course implementations. This thesis limits to designing and implementing the application. Evaluation through end user testing is done in future course implementations. The application is open-source software allowing possible future contributions from third parties. The application has been developed using React user interface library and hosted on GitHub Pages as a publicly accessible static website. Using the application requires a web browser and Internet access.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Jani Heinikoski

Mikro-ohjelmoitavan tietokoneen simulointi Reactilla

Tietotekniikan kandidaatintyö

2023

42 sivua, 6 kuvaa, 2 taulukkoa ja 3 liitettä

Tarkastajat: Yliopisto-opettaja Olli-Pekka Hämäläinen ja apulaisprofessori Antti Knutas

Avainsanat: simulaattorisovellus, suunnittelutiede, React, mikro-ohjelmointi, e-oppiminen

Tämän kandidaatintyö tutkii yksinkertaista mikro-ohjelmoitavaa tietokonetta, jota käytetään LUT-yliopiston tietojenkäsittelytieteiden perusteet -kurssilla mikro-ohjelmoinnin perusteiden opettamiseen. Työn suurin tutkimusongelma on tuottaa helposti saavutettava mikro-ohjelmoitavaa tietokonetta simuloiva ratkaisu.

Alustariippumaton verkkopohjainen simulaattorisovellus tuotetaan työssä suunnittelutiede-tutkimusmetodologiaa noudattaen. Simulaattorisovelluksen tarkoitus on ratkaista tutkimusongelma mahdollistamalla mikro-ohjelmien tuottamisen ja suorittamisen virtuaaliympäristössä, jossa käyttäjä voi nähdä miten tietokoneen komponentit reagoivat mikro-ohjelman suorituksen aikana. Yhteensopivuusongelmia sisältävä Windows-käyttöjärjestelmästä riippuvainen simulaattorisovellus on jo olemassa. Tässä työssä kehitetään alustariippumaton verkkosovellus, jotta se on helposti saavutettavissa suurimmalle osalle opiskelijoista.

Simulaattorisovellusta on tarkoitus hyödyntää opetusta tukevana e-oppimistyökaluna tulevilla tietojenkäsittelytieteen perusteet -kurssin toteutuksilla. Tämä työ rajoittuu sovelluksen suunnitteluun ja toteutukseen jättäen käyttäjätestauksen tuleville kurssitoteutuksille. Simulaattorisovellus on avointa lähdekoodia, jotta mahdollinen jatkokehitys on mahdollista myös kolmansien osapuolien toimesta. Sovellus on kehitetty React -käyttöliittymäkirjaston avulla, ja julkaistu GitHub Pages -alustalla julkisesti saavutettavana staattisena verkkosivuna. Sovelluksen käyttäminen vaatii verkkoselaimen sekä toimivan Internet-yhteyden.

SYMBOLS AND ABBREVIATIONS

Abbreviations

ALU	Arithmetic Logic Unit
CAGR	Compound Annual Growth Rate
CPU	Central Processing Unit
DS	Design Science
DSRM	Design Science Research Methodology
FoCS	Foundations of Computer Science
ISA	Instruction Set Architecture
LUT	Lappeenranta–Lahti University of Technology
MAR	Memory Address Register
MDR	Memory Data Register
MIR	Microinstruction Register
MIT	Massachusetts Institute of Technology
MM	Main Memory
MPC	Microprogram Counter
MPM	Microprogrammed Machine
RAM	Random-Access Memory
SPA	Single-Page Web Application

Table of contents

1 Introduction

As the world has transformed into a digitalized environment, the need for software engineers has grown. The U.S. Bureau of Labor Statistics stated in 2021:

> “Employment of software developers, quality assurance analysts, and testers is projected to grow 22 percent from 2020 to 2030, much faster than the average for all occupations.” [2]

As a rapidly growing industry, many software developers have chosen to self-learn software engineering. According to a survey conducted by Stack Overflow in 2016, 69% of the survey participants (40183 non-student developers) claim to be at least partially self-taught and 13% only self-taught [3]. It is easy to get overwhelmed by the large amount of information about software engineering on the Internet especially for new software developers. Many articles focus on trending technologies overlooking the importance of understanding the fundamentals of computer science. An extensive mixed-method study by Paul Luo Li, Amy J. Ko and Andrew Begel showed that the top two most important attributes of a software engineer are: “pays attention to coding details” [4] and “mentally capable of handling complexity” [4].

To understand complex systems or understand programming details, it is important for developers to have at least a basic understanding of the underlying system. Understanding how computer systems work at a low abstraction level is required of software developers who intend to work with low level programming but is also useful for other developers. Getting familiar with a simplified microprogrammed machine (MPM) helps to understand arguably the lowest level of programming: microprogramming. Specific details of how modern computers utilize microprogramming are outside the scope of this thesis however typical use cases are considered in related research -chapter. Instead, this thesis focuses on a simplified MPM proposed by J. Boberg et al. (2012) [5]. The structure of the MPM is depicted in (Appendix 1). A web simulator application is designed and produced during this thesis. The application is intended to help understand basic concepts about microprogramming by allowing users to create and execute microprograms on a primitive simulated computer. This thesis refers to “a piece of equipment that is designed to represent real conditions” [6] when referring to a simulator. In this thesis, the piece of equipment is a web application which represents the MPM.

Microprogramming was invented by Maurice Wilkes in 1951 to simplify circuitry in computer systems. Microprogramming is the act of creating microprograms which consist of very primitive microinstructions. Microprogramming is typically used to implement machine instructions instead of implementing them with complex circuitry. It also enables engineers to implement a machine's instruction set differently with varying efficiencies and costs. [7], [8]

The LUT School of Engineering Science at Lappeenranta–Lahti University of Technology (LUT) teaches a course called: Foundations of Computer Science (FoCS). The course focuses on the same MPM as this thesis. Having attended the course and worked on two different course implementations as a teaching assistant, it can be stated that a large portion of students who attend FoCS consider the MPM as a challenging topic. The inability to test microprograms in practice causes a large part of the challenge. Students cannot practically verify if they understood the concepts correctly as the MPM is only defined on paper. Creating a practical way to test microprograms written for the MPM is the main problem this thesis aims to solve. This thesis also provides English literature on the MPM as currently it is only available in Finnish. The design, structure and other materials about the MPM can be found in Finnish from [1].

1.1 Goals and methods

At the time of writing, one simulator program which simulates the MPM has already been created. It is a platform dependent Windows application; it only works on Windows-based operating systems. The simulator also has known compatibility issues with modern versions of Windows. [1] The main goal of this thesis is to produce a platform independent open-source web application which simulates the MPM. The main use case of the simulator application is to be used as a supporting e-learning tool in FoCS. The goal is to allow users to create microprograms, execute them on a simulator application and be able to follow how the different components of the MPM change during execution. A platform independent web application only requires access to a modern Internet browser (such as Google Chrome, Mozilla Firefox, Microsoft Edge etc.) and a working Internet connection which most students have access to. Making the software open-source enables contributions in the future from third parties as well.

The application is developed using React which is an open source frontend JavaScript library primarily intended for creating web user interfaces [9]. React is also an open-source technology as it is licensed under the Massachusetts Institute of Technology (MIT) -license which is a permissive software license [10]. The permissive MIT-license allows other software developers to continue improving the application after it is published. React's component-based workflow should suit well for modelling the MPM which consists of multiple autonomous components. React is also considered as one of the most used web technologies at the time of writing [11].

With these considerations in mind, the main research questions for this thesis are:

- How to create a web simulator application which simulates the MPM introduced by J. Boberg et al. (2012) [5] using React?
- Can the simulator application be used with nothing but a web browser and Internet access?
- Can users create and execute arbitrary microprograms in the simulator application?

As this thesis aims to answer a research question which calls for designing and developing software, design science (DS) research has been chosen as the research paradigm. DS research focuses on the creation of artifacts that answer questions related to human/organizational problems. The artifact that solves the research problems in this thesis is the simulator application. DS research has gained traction among information systems researchers since the early 1990s and many frameworks/methodologies have been created for conducting it. [12], [13] This thesis follows the design science research methodology (DSRM) proposed by Peffers et. al. (2008) [12]. The research methodology is presented in detail in chapter three.

1.2 Work structure

This thesis is divided into six chapters. The first chapter is the introduction which gives the reader required background knowledge, explains the motives that lead to this thesis, sets goals for the thesis’ outcome, and justifies why this thesis is important. The second chapter covers related research on the thesis’ topic. The third chapter covers the DSRM (the research methodology followed in this thesis) in-depth. The fourth chapter is dedicated for the design and development of the artifact i.e., the simulator application which is the primary product of this thesis. The fifth chapter discusses the produced artifact and its key findings. The sixth and final chapter concludes this thesis.

2 Related research

This chapter explains how the MPM defined by J. Boberg et al. (2012) [5] and the simulator application as a modern e-learning tool relates to existing research. The MPM is a simplified computer which does not execute machine language instructions directly, instead it utilizes microprogramming. Computers that are built in such a way are called microprogrammed as they use an interpreter-like microprogram to execute machine language instructions. The MPM is the primary subject of study in this thesis overall however there is little research about it outside of the defining document. This chapter focuses on microprogramming in general as the MPM is intimately related to the concept of microprogramming. [5]

After the invention of microprogramming by Maurice Wilkes in 1951 [7], it has been researched and applied extensively mainly in the context of control units of computers. Articles such as *Microprogrammed Control for Computing Systems* by G. B. Gerace (1963) have been written to improve upon the Wilkes' original scheme [14]. Before discussing microprogramming more, it is important to understand its' main use case in computers.

For computers to be able to execute programs, they must first be compiled into instructions understandable by the computer called machine language instructions. The instructions are typically in a binary format as bits can easily be mapped into electric signals which are ultimately required for controlling computer hardware. A special computer program called assembler is designed to compile symbolic assembly language into machine language instructions which often have near one-to-one mapping. Similarly, higher level programming languages can be compiled into assembly language by compiler programs and then into machine language by the assembler. [15]

After programs have been compiled into these machine language instructions, the computer still needs to be able to turn them into electric signals that operate the actual hardware. An instruction set architecture (ISA) can be considered as an abstract model to which a computer's central processing unit (CPU) must conform to. ISAs define salient parts about the capabilities of the CPU such as the instruction set. Instruction set contains all the architecture's machine language instructions which an implementing CPU is capable of executing. [15], [16] Modern ISAs such as x86 and its 64-bit extension x86-64 contain complex instructions such as multiplication and division which are non-trivial to implement directly in the hardware.

Implementing large and/or complex instruction sets with sequential logic circuitry is a complex, inflexible and an expensive approach typically called hardwired control. Microprogrammed control on the other hand can be used to alleviate the need for such complex circuitry by implementing machine instructions by breaking them down into a sequence of primitive microinstructions. A microprogram can act as an interpreter which fetches a machine instruction from the main memory of the computer and executes a sequence of microinstructions which results in the same goal as implementing the machine instruction with complex circuitry. [17]–[19]

As mentioned earlier, the simulator application can be considered as an e-learning tool. To better understand the growing global e-learning market, compound annual growth rate (CAGR) is introduced. CAGR is the annualized average rate of revenue growth over the course of $n > 0$ years. CAGR between years y_0 and y_t where $y_t > y_0$ is calculated as follows:

$$CAGR = (\frac{r_t}{r_0})^{\frac{1}{y_t - y_0}} - 1$$

where r_0 is the annual revenue of year y_0 and r_t is the annual revenue of year y_t. [20], [21] According to a report by Polaris Market Research published in February 2022 “The global E-learning market was valued at USD 214.26 billion in 2021 and is expected to grow at a CAGR of 20.5% during the forecast period.” [22] where the forecast period is from year 2022 up to year 2030. Such a rapid growth amounts to approximately fivefold increase from the current market value until the end of year 2030.

E-learning which is considered as computer assisted learning has existed for decades however it is still a prominently researched topic as the rapidly growing global market suggests. The simulator application falls under the category of asynchronous e-learning tools which can be used for self-paced learning. [23] Although this thesis largely focuses on the design and development of the simulator application, future research can be done from the perspective of effectiveness as an e-learning tool.

3 Research methodology

This chapter introduces the design science research methodology followed throughout this thesis. DSRM is a relatively new research methodology proposed by Peffers et al. (2008) [12]. DSRM process model consists of six core activities [12], all of which are covered in this chapter. The six activities are represented in Figure 1 which visualizes the DSRM process model. This thesis follows the process model relatively strictly but as with all DS projects, “a certain level of creativity” [13] is needed.

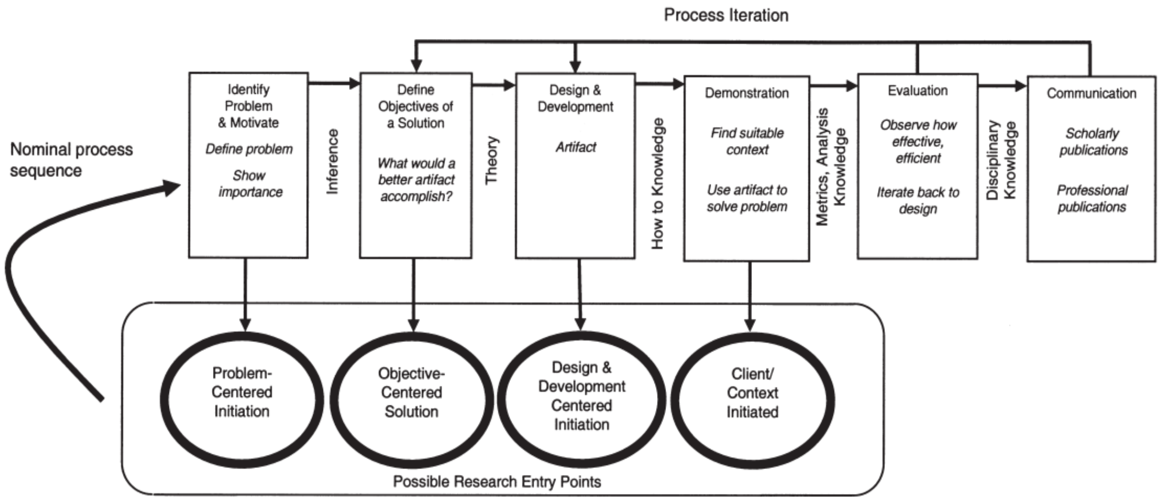

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 1. DSRM process model from [12].

3.1 Problem identification and motivation

The first activity of the DSRM process model Figure 1 is problem identification and motivation. In the first activity, the researcher should: “Define the specific research problem and justify the value of a solution.” [12]. This activity consists of two distinct steps: identifying and defining the research problem, and justifying the value of a solution. Both steps are covered in the introduction chapter. Justifying the value of a solution not only motivates the researcher but also possible future researchers. Considering the solution’s value can also help in accepting the results of the research. Well defined problem on the other hand is required to be able to produce an artifact which effectively solves the research problem. It can also be helpful to break down a complex problem on a conceptual level to better understand what it is that the artifact needs to solve. [12]

3.2 Define the objectives for a solution

After specifying the research problem, the objectives for the solution are defined by taking the research problem specification, knowledge about what is possible/feasible, and determining the objectives and goals for the solution [12]. The most important goals and objectives for the solution have already been stated in the introduction chapter however more specific objectives will be formed later. As the solution consists of implementing software, knowledge about what is feasible or not is gained during initial prototyping during the design and implementation. As with any software project, the requirements of the artifact are expected to change as the implementation progresses.

3.3 Design and development

Activity three is called design and development which is the most important activity for this thesis. The fourth chapter is primarily dedicated to how this activity was executed in this thesis. As the name suggests, the third activity focuses on designing and developing the artifact. In Hevner’s and Chartterjee’s book *Design Research in Information Systems* (2010), the artifacts are broadly defined as:

- constructs: vocabulary and symbols,
- models: abstractions and representations,
- methods: algorithms and practices,
- instantiations: implemented and prototype systems and
- better design theories. [13]

However, the artifacts in DS research can be any designed objects if they possess research contribution in the design [12]. The artifacts need to generate new knowledge by solving a problem or by improving existing solutions to be considered DS research instead of just routine design practice [13]. It is through “rigorous evaluation methods and comparison with similar artifacts” [13] that the researcher may conclude that new knowledge has indeed been created with the development of a new artifact which is conducted in the fifth activity. This thesis does not focus on the evaluation of the artifact as proper end user testing can only be done in future FoCS course implementations. Naturally, the artifact is evaluated during its implementation however it does not suffice as rigorous evaluation.

3.4 Demonstration

The fourth activity consists of demonstrating the artifact’s value by using it to solve one or more of the research problems [12]. The main research problem of this thesis is to design and implement the simulator application which solves the problem of not being able to practically test microprograms created for the MPM. It is used to find a solution to the research problem and research questions which are later discussed in the fifth chapter thus covering the demonstration activity.

3.5 Evaluation

As mentioned before, to verify that new knowledge has been created by the development of an artifact it must be rigorously evaluated [13]. The fifth activity of the process model in Figure 1 consists of comparing the objectives set in activity two to the results gained from the demonstration in activity four. As the artifact is a software project, this activity consists mainly of software testing and requirements verification. Each artifact calls for distinct evaluation methods as each artifact has different metrics, context, and objectives. Due to the broadly varying case-specific evaluation methodologies, “any appropriate empirical evidence or logical proof” [12] can be included in the evaluation. [12] End user testing and creating automated tests to evaluate the software is a part of future work and outside the scope of this thesis however continuous testing is performed during the development of the artifact.

3.6 Communication

The final activity of the DSRM process model Figure 1 is communication. This activity, as the name yet again suggests, consists of communicating:

> “the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and other relevant audiences such as practicing professionals, when appropriate.” [12]

Because this bachelor’s thesis is a scholarly publication, it will be published for appropriate audiences which satisfies the sixth and final activity. Furthermore, the artifact is intended to be an open-source project which allows thorough inspection of its design and implementation by all interested audiences. Relevant information on where the to find the artifact will be attached as appendices to this thesis after it has been finished.

3.7 DSRM process model order

The DSRM process model has a sequential order regarding the different activities however it is possible to start at different activities depending on the situation. The different entry points are depicted in Figure 1. Because the research problem originated from observations, it is natural to select the problem-centered initiation approach for this thesis. Definition of the problem is given in the first chapter along with the most important objectives which cover the first two activities. Further chapters cover the rest of the activities however evaluation is considered as a part of future work.

An objective-centered approach is meant for already well defined research problems that can be solved by developing an artifact. A design and development -centered approach starts with activity number three. There must be previous research on the problem to jump directly into activity three. An artifact which has clear objectives but has not yet been designed as a solution for the problem is a common reason for adopting the design and development - centered approach. Finally, there is the client/context-initiated approach starting with activity four: demonstration. Such an approach is usually triggered by observing a working solution for the same or similar research problem. Researchers can then use the results of the working solution to improve upon the existing one or use them to create a new one by working backward in the process model. [12]

4 Design and implementation

This chapter covers the fourth activity of the DSRM process model: design and implementation. As the name suggests, the design and the implementation of the artifact are presented. Technologies used for the development of the simulator application are presented first. The microprogrammed machine and its basic working principles are discussed in more detail. Components of the MPM are introduced and their implementation in the simulator application is presented.

4.1 Simulator application architecture

Before proceeding to the details of the MPM and how they are implemented in the simulator application, a high level explanation of the simulator application's architecture is in order. It is a single-page web application (SPA) developed using React which is a JavaScript library intended for creating user interfaces. SPAs are web applications that update themselves dynamically using JavaScript APIs without the need of refreshing the whole page [24]. In React applications, pieces of the user interface are broken down to autonomous reusable components. The components can be nested and thus form a hierarchical structure which suits well for modelling the MPM. [25]

Figure 2 represents the React components' hierarchy in the simulator application. Some of the less relevant components have been omitted in Figure 2 for the sake of brevity. The application is first divided into two components which represent complete web pages: HomePage and SimulatorPage. As their names suggest, HomePage is the landing page of the website and SimulatorPage contains the simulator application. SimulatorPage is further divided into two components: Editor and MPM. The Editor component represents the view where the users can create and edit their microprograms. It also allows users to manipulate initial conditions of the MPM. The main application logic resides in a component called MPM which represents the microprogrammed machine. It consists of multiple smaller child components which represent different parts of the MPM.

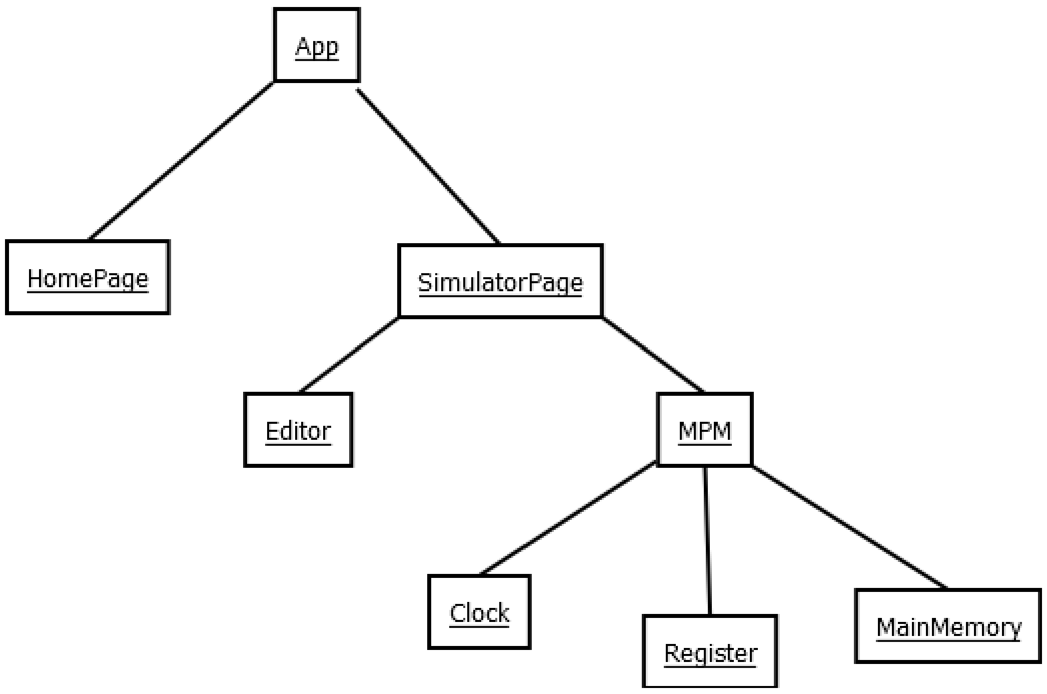


Figure 2. Components of the simulator application

Most of the application’s state also lives within the MPM component. In React, data that changes over time and needs to be remembered is stored using state. For example, the values of registers and data buses are stored using state variables within the MPM component. Because data flows one way from top down in React (parent components pass data down to child components), the child components of the MPM are mostly representations of the state of the MPM component without much added functionality. [26]

4.2 Basic working principle of the MPM

The MPM discussed in this thesis is a simplistic microprogrammed computer. Microprogrammed computers execute complex instructions (i.e., machine language instructions) as a series of primitive microinstructions. An earlier approach is called hardwired implementation where complex instructions are implemented using sequential logic circuits. Microprogrammed approach reduces cost and complexity of the control unit which is why it is the basis of many modern computers. [5], [19] An analogous way to understand the difference is that microprogrammed computers interpret machine instructions whereas hardwired computers already understand machine instructions.

The microinstructions of the MPM defined by J. Boberg et al. (2012) [5] consist of 22 control bits described in Table 1. Together the microinstructions form a microprogram which the MPM executes. When the MPM is built, microprograms are stored in a read-only memory called microprogram memory which can contain a maximum of 256 microinstructions. A five phase clock controls the computer by activating the current microinstruction's control bits in five distinct phases. Activating a control bit means that the control signal represented by the control bit is passed through to the control bus which in turn activates the logic described by the control bit. For example, if the control bit c17 is activated and its value is one then the integer 1 is written to data bus DC1 (see the description of c17 in Table 1). [5] If the control bit's value is zero, it does not affect the state of the computer when activated as zero represents low voltage which the control bus's wires have already by default. With that said, programming consists of creating a series of microinstructions where the programmer chooses what primitive operations i.e., control bits should be carried out in each instruction.

Without diving too deep into electric engineering, it is helpful to understand how bits and electric signals relate to one another – specifically how binary values are used to represent electric signals. Each digit of a binary value is called a bit which can have a value of zero or one. Bits can be trivially translated into electric signals and vice versa where zero represents low voltage and one represents high voltage. This kind of a voltage which corresponds to a discrete number of values is called a digital signal [27]. When designing complex logic circuits, it is easier to use ideal zeros and ones to represent electric signals instead of using true voltages or other electric properties. Going forward, this thesis refers to bits and binary values, but it is important to keep in mind that in the context of the MPM bits are merely an encoded representation of electric signals. [28] For example, if it is said that the integer value one is written to a bus, it means that the rightmost wire of that bus has high voltage whereas other wires of the bus are low voltage thus representing the binary value one: $\ldots 00001$.

Table 1. Control bits of the MPM. Adapted from [5, p. 133]

Control bit	**Description**
C1	Value of register A is written to data bus DC2.
C2	Value of register B is written to data bus DC2.
C3	Value of register C is written to data bus DC2.
C4	Value of register D is written to data bus DC2.
C5	Integer one is written to data bus DC1.
C6	Value of register MDR is written to data bus DC1.
C7	Value of data bus DC1 is negated.
C8	A logical left shift is performed on the value of data bus DC3.
C9	Value of data bus DC3 is written to register A.
C10	Value of data bus DC3 is written to register B.
C11	Value of data bus DC3 is written to register C.
C12	Value of data bus DC3 is written to register D.
C13	Value of data bus DC3 is written to register MDR.
C14	Value of data bus DC3 is written to register MAR.
C15	Value at main memory address pointed to by register MAR is written to MDR.
C16	Value of register MDR is written to main memory address pointed to by MAR.
C17	Integer value 1 is written to data bus DC1.
C18	The eight most significant bits of register MIR are written to DC1.
C19	Integer value 1 is written to data bus DC1 if the value of register A is zero otherwise integer value 2 is written to data bus DC1.
C20	Integer value 1 is written to data bus DC1 if the most significant bit of register A is 1 otherwise integer value 2 is written to data bus DC1.
C21	The four most significant bits of register MDR are written to data bus DC1.
C22	Value of register MPC is written to data bus DC2.

All 22 control bits in a microinstruction are activated in the first four clock phases. The clock phases are executed sequentially from clock phase one to five. The fifth clock phase has a special purpose. In the fifth clock phase, the microprogram counter (MPC) which points to the currently executed microinstruction in the microprogram memory gets a new value from data bus DC3. After the new value has been written to the MPC, the microinstruction pointed to by MPC is fetched from the microprogram memory and written to the microinstruction register (MIR). MIR is a 22-bit register which stores the microinstruction that is currently being executed. The MPM works in an infinite loop where the current microinstruction in MIR is executed in the first four clock phases after which a new instruction is fetched for execution in the fifth clock phase. [5]

During each clock phase, each of the control bits that belong to it are executed simultaneously. Simultaneous activation of the control bits might lead to conflicting situations. For example, the first four control bits all write a different register’s value to data bus DC2 which results in an undefined value in DC2. However, as long as the computer does not crash, it does not matter if the value is not stored anywhere. [5] To provide users the ability to execute each control bit separately, the simulator application does not execute the control bits simultaneously in each clock phase.

In conclusion, the MPM is a simplistic computer which executes a read-only microprogram from the microprogram memory. The simulator application allows users to create arbitrary microprograms and test them by showing how the computer’s state (values of registers, memory, data buses etc.) changes during the execution of the microprogram. Without a real or virtual implementation of the MPM, there would be very limited options for verifying whether microprograms work as the programmer intended which is what the simulator aims to solve.

4.3 Components of the MPM

In short, the MPM comprises of the following components: registers, memories, an arithmetic logic unit (ALU), a clock and buses [5]. Each of them has their own purpose and together they form the microprogrammed machine depicted in (Appendix 1). Table 2 gives a high level description of each of the components. After Table 2, all the components’ implementations in the simulator application are explained. Finally, features that are not included in the definition of the MPM are presented.

Table 2. Components of the MPM

Component	Description
Microinstruction register	Microinstruction register is a 22-bit register which contains the microinstruction currently being executed. Each bit maps to a specific control bit from left to right e.g., 20^{th} bit of MIR corresponds to control bit c20. If the value of a control bit is one then the operation of the corresponding control bit will be carried out in its respective clock phase otherwise it does not affect the computer's state. [5]
Clock	The MPM has a five-phase clock that controls the computer's actions by activating the control bits in the MIR. Control bits c1–c8 are activated in the first phase, c9–c14 in the second phase, c15–c16 in the third phase and c17–c22 in the fourth phase. The bits are activated simultaneously in each phase. The fifth clock phase has a constant function in which the value of DC3 is written to the MPC after which a new microinstruction is written to MIR from the microprogram memory address pointed to by the MPC. [5]
Buses	The MPM has three data/address buses DC1, DC2 and DC3 and a control bus CC. The data/address buses are 16-bit as is the word size of the MPM. The control bus is a 22-bit bus as it must be able to contain microinstructions which consist of 22 control bits. [5]
Registers A, B, C and D	The MPM has four 16-bit general purpose registers called A, B, C and D. They are meant for storing operands and results of arithmetic operations. The A register can also be used in clock phase 4 to modify control flow of the microprogram. [5]
Microprogram memory	Microprogram memory is a read-only memory which contains the microprogram. Each memory location is 22-bit wide as each location contains a microinstruction consisting of 22 control bits. There is a total of $2^8 = 256$ memory locations because MPC is an 8-bit wide register which means that the microprogram can consist of 256 microinstructions maximum. [5]
Microprogram counter	Microprogram counter is an 8-bit wide register which contains the address of the currently executed microinstruction. The address points to a memory location in the microprogram memory. In the fifth clock phase, the value of data bus DC3 is written to the MPC after which the microinstruction pointed to by the MPC is written to the MIR. [5]

Main memory	Main memory (MM) is a read-write memory which has $2^{12} = 4096$ 16-bit wide memory locations. It is typically used to store machine language instructions and data. [5]
Memory data and address registers	Memory data register (MDR) and memory address register (MAR) are registers that are used to read and write data from and to the main memory. MDR is a 16-bit wide register which can be used in the first clock phase for arithmetic operations as it is the only register that can be written to data bus DC1. MAR is a 12-bit address register which is used to point to the MM. In the third clock phase, it is possible to write data to the MDR from the MM location pointed to by the MAR or write the value of MDR to the MM location pointed by the MAR. [5]
Arithmetic logic unit	The arithmetic logic unit is a full adder with few other abilities: it can negate the value of DC1, sum two 16-bit integers in data buses DC1 and DC2, and perform a logical left shift of the resulting sum in DC3. Subtraction is handled by using two's complement representation in the MPM. In short, it can perform all the arithmetic operations of the MPM. [5]

4.3.1 Registers and buses

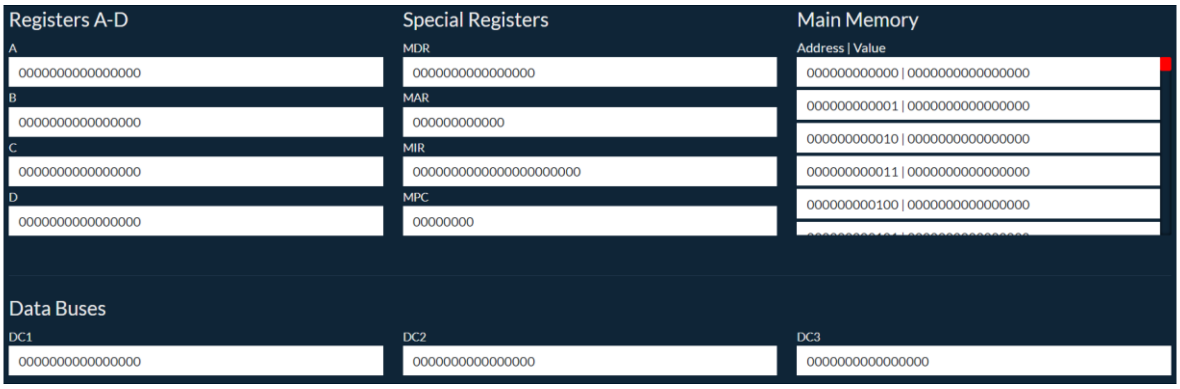
Registers are components that can store binary information. An n-bit register consisting of n flip-flops can store an n-bit binary value. Flip-flops are logic circuits that can store a single bit of information. [27, p. 324] Storing integers in variables is a trivial task in modern programming languages however some extra considerations must be taken because registers have a minimum and a maximum value they can store. The MPM handles most values in two's complement binary representation which means that registers' values must be within the range:

$$\text{value} \in [-2^{n-1},\ 2^{n-1} - 1]$$

where n is the number of bits. If the user inputs a value outside of the range, the application displays a warning message modal to the user.

Buses are a set of lines that transfer data between different components in a computer. Buses are controlled by selection logic where control signals select the source and destinations in between which data is transferred. For example, in the MPM control bits $c9 - c14$ can be used to transfer data from the same source (data bus DC3) to multiple different registers using a shared transfer path i.e., a bus. [5], [27, p. 359]

In the simulator application, both the registers and buses use the same Register component. This is because the relevant information of both buses and registers is the binary information that they hold at a certain time. What React components the user interface composes of is completely transparent to the end user. The component is simply a read-only HTML input -element which displays the value it currently has. It can display the value in either binary representation as seen in Figure 3 or as a base 10 integer. The registers' and buses' actual values are stored in state variables within the parent MPM component where the logic for transferring data between registers is implemented as well. The visual appearance of registers and buses can be seen in Figure 3.


Registers A-D

A: 0000000000000000
B: 0000000000000000
C: 0000000000000000
D: 0000000000000000

Special Registers

MDR: 0000000000000000
MAR: 000000000000
MIR: 0000000000000000000000
MPC: 00000000

Main Memory

Address | Value
000000000000 | 0000000000000000
000000000001 | 0000000000000000
000000000010 | 0000000000000000
000000000011 | 0000000000000000
000000000100 | 0000000000000000

Data Buses

DC1: 0000000000000000
DC2: 0000000000000000
DC3: 0000000000000000

Figure 3. Registers, buses, and main memory components in the simulator application.

4.3.2 Memory

Figure 3 also shows the visual appearance of the main memory component. Memory in the context of digital systems is simply "a collection of cells capable of storing binary information" [27, p. 403]. The MPM has two separate memory components: microprogram memory and main memory. The microprogram memory is a read-only memory where microprograms are stored. Read-only memory cannot be written to as the name suggests; values can only be read from it. Main memory on the other hand is a random-access memory (RAM) which can be both written to and read from. The time it takes to access RAM is independent of the location which is why it is called random-access memory [27, p. 404]. Each location of the main memory can store a 16-bit binary value, and it has 4096 memory locations. Addressing memory locations is done using the MAR register which is a 12-bit register hence the MM has $2^{12} = 4096$ memory locations. [5]

As each location of the main memory stores a 16-bit value, the MainMemory component can be thought of as a list of 16-bit registers that have a unique address. In Figure 3, the MainMemory component can be seen as a list of similar components as the Register component. It has not been implemented using the Register component in the simulator application however it is a list of read-only HTML input elements. Each of the elements displays the address and value of the memory location in the following format: address | value. The MainMemory component is also capable of displaying its addresses and values in either binary or base 10 representations.

4.3.3 Arithmetic logic unit and clock

The arithmetic logic unit of the MPM is a component capable of doing a logical left shift, summation, and subtraction by negating the other operand and then performing summation. The tasks are simple to implement using most programming languages however just summing or subtracting two variables in JavaScript is not enough. Once again, the minimum and maximum values come into play because the summation is performed by a 16-bit full adder [5, p. 132]. As discussed before, the minimum and maximum value that can be represented in two's complement binary representation is dependent on the number of bits. The simulator application displays a warning modal if the arithmetic operation performed by the ALU produces a result outside the range:

$$[-2^{16-1},\ 2^{16-1} - 1] = [-32768, 32767]$$

which is the range of values a 16-bit binary value can be represented in two's complement representation.

As stated in Table 2, the MPM has a five-phase clock which controls the MPM. A clock is a type of signal generator which produces a rectangular pulse wave. The MPM's multiphase clock generates clock pulses in five distinct phases similarly as illustrated in [29, Fig. 1b] which are then passed to five separate wires. The five wires are connected such that the clock pulses activate the control bits in MIR in five phases. Because the clock controls when the control bits are activated and thus progresses the microprogram's execution, it makes the computer's speed dependent on the clock's frequency. Naturally, physical properties of the other components limit the maximum clock frequency the computer can handle. [5, pp. 129, 133]

The Clock component somewhat differs from the previous definition of the MPM's clock. In the application, the Clock component only displays which clock phase is currently in execution. The intention is to give the user the ability to advance the clock by activating control bits. To allow users to activate the control bits one by one, they are not executed simultaneously in clock phases. When the user activates control bits, the Clock component is updated by the MPM component to match the clock phase of whichever the last control bit activated belongs to. For example, after executing the control bit c15 which is the first control bit of the third phase, the clock phase is updated to three.

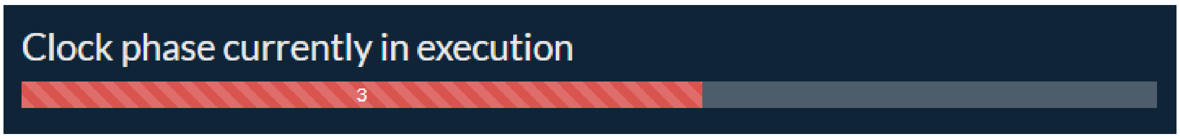


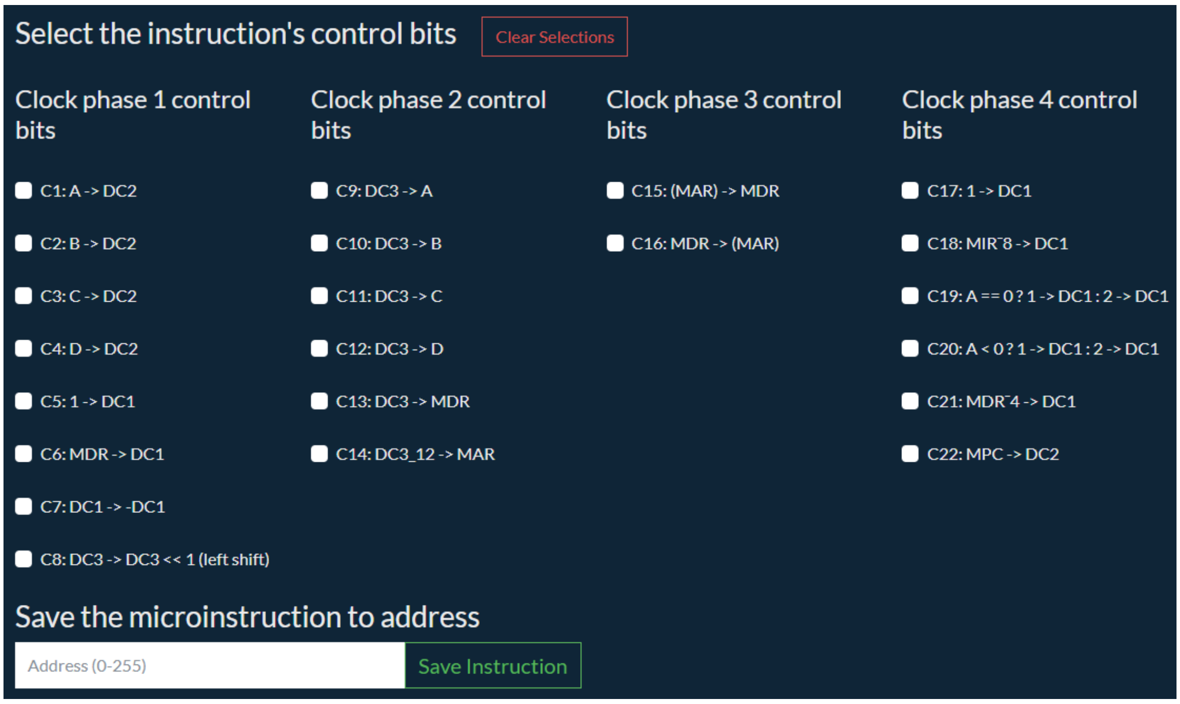
Figure 4. Visual appearance of the simulator application's Clock component.

As seen in Figure 4, the Clock component of the simulator application is a progress bar which advances incrementally in steps of one from phase one to phase five. The clock phase is also displayed as a numeric label within the progress bar for clarity. The Clock component itself does not contain any state. The current clock phase is managed by the MPM component and passed as props to the Clock component.

4.3.4 Other features

Most of the simulator application comprises of the components that have been modelled by following the definition of the MPM. However, few additional components have been created to provide users functionality for creating microprograms, switching between decimal and binary number systems, initializing registers and main memory, and saving or loading microprograms. The implementation of these features is presented here.

One of the most important features is to allow users to create microprograms as without it the simulator provides little to no value. Microprogramming consists of creating the 22-bit microinstructions where in each microinstruction each of the 22 control bits' values are set to zero or one. If a control bit's value is one, the operation described in Table 1 is carried out when the control bit is activated. After creating the microinstruction, it is saved into the microprogram memory to a certain address. To provide this functionality, a form consisting of checkboxes representing the control bits is shown in Figure 5. At the bottom of Figure 5, an input element and a button are provided for the user to input the address they want their microinstruction to be saved to.



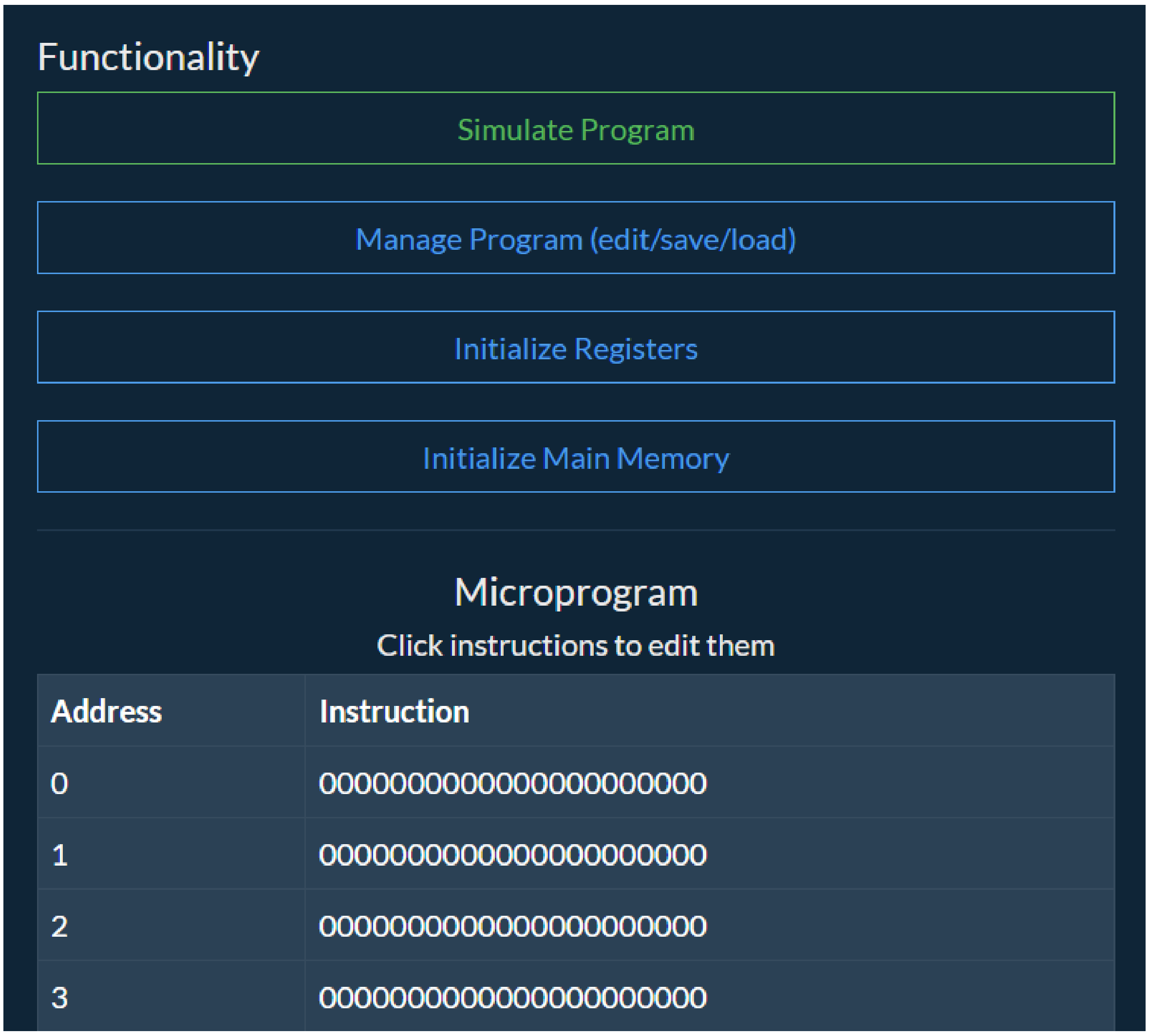
Select the instruction's control bits

Clock phase 1 control bits	Clock phase 2 control bits	Clock phase 3 control bits	Clock phase 4 control bits
C1: A -> DC2	C9: DC3 -> A	C15: (MAR) -> MDR	C17: 1 -> DC1
C2: B -> DC2	C10: DC3 -> B	C16: MDR -> (MAR)	C18: MIR¯8 -> DC1
C3: C -> DC2	C11: DC3 -> C		C19: A == 0 ? 1 -> DC1 : 2 -> DC1
C4: D -> DC2	C12: DC3 -> D		C20: A < 0 ? 1 -> DC1 : 2 -> DC1
C5: 1 -> DC1	C13: DC3 -> MDR		C21: MDR¯4 -> DC1
C6: MDR -> DC1	C14: DC3_12 -> MAR		C22: MPC -> DC2
C7: DC1 -> -DC1			
C8: DC3 -> DC3 << 1 (left shift)			

Save the microinstruction to address

Figure 5. Editor component's microinstruction section.

To be able to view and modify the microprogram, it has been added as a table consisting of the addresses and microinstructions at the bottom of the Editor and MPM components as seen in Figure 6. The table represents the microprogram memory of the MPM and the four first memory locations are shown in Figure 6. It can be freely modified in the application however it is important to remember that it is a read-only memory (see Table 2 and chapter 4.3.2).



Microprogram

Click instructions to edit them

Address	Instruction
0	0000000000000000000000
1	0000000000000000000000
2	0000000000000000000000
3	0000000000000000000000

Figure 6. Bottom of the Editor component.

As also seen in Figure 6, buttons for editing, saving, and loading microprograms, and initializing registers or the main memory are provided in the Editor component. They open modals which allow users to give initial values for the general purpose registers A, B, C and

D or initialize the main memory with arbitrary values. The manage program -button opens a modal where users can edit the microprogram in a free-form text representation. Such functionalities are not defined by J. Boberg et al. (2012) [5] however have been added to better work as a learning tool. Exercises which prerequisite some initial conditions can be created when users are allowed to modify the registers’ and main memory’s values. Finally, a button for switching between decimal and binary number systems is given in the MPM component to eliminate the need of manually converting values between them.

5 Discussion

This chapter discusses key points of the design, development, and implementation of the finished artifact i.e., the simulator application. Key points include the software development process, software development technologies, limitations of the artifact, deviations from the specification of the MPM, and future work. After discussing the key findings, this chapter is concluded by deliberating the research questions.

5.1 Software development process

The software development process followed during the development of the artifact can be considered a loose lightweight agile process. Simplicity and development speed were preferred over a robust process and detailed documentation which is common for agile software processes. [30, p. 19] The most significant reason for choosing a lightweight development process was time. As the only developer contributing to the development of the artifact, not much time could have been allocated for following a strenuous process. The process consisted of a cycle of quick small releases which were then tested and validated immediately. In the context of agile practices, iteratively developing, testing and validating the software artifact are listed as best practices in [31, Tbl. 2].

A minimalistic prototyping approach was first taken with the simulator application to try and evaluate how realistic developing it was. After finishing few iterations of the prototype, an approximation of required work to produce the simulator application was estimated and deemed realistic. The finished prototype worked as the basis on top of which the finished artifact was developed. The first prototype can be found by going back to the first commit in the code repository of the simulator application (Appendix 2).

A bottom-up approach was followed during the whole development process. The MPM consists of easily identifiable autonomous components such as registers and buses that can be considered and modelled as separately. The different components of the MPM were implemented first and then used to construct the MPM component itself. In the context of

React, it means that the leaf components at the component hierarchy in (Figure 2) were developed first i.e., bottom up.

5.2 Limitations and future work

The simulator application has been divided into React components and a collection of utility functions written in vanilla JavaScript i.e., not using the React's JSX extension which extends the syntax of JavaScript by allowing HTML-like markup to be mixed with plain JavaScript [32]. The utility functions provide functionality that is required in multiple components such as converting a base 10 integer into binary representation; they are not considered as a part of any single component. They are then imported and used within the React components wherever needed.

Separately testing, improving or completely rehauling the components or utility functions is possible as long as the components' interfaces do not change. Because most components and functions are small and serve a single clear purpose, future contributors can easily work on them. However, by design most of the application's state and logic resides in the MPM component which is a relatively large component compared to the other ones. The MPM component could be better refactored into smaller components in the future.

As was the original goal, the simulator application is open-source software to allow contributions from anyone. Typically, open-source software is a two-edged sword where on the other hand vulnerabilities are found, reported and thus patched more frequently as more people contribute to the project however it also allows malicious users access to the source code. Resistance to vulnerabilities is inherent to the simulator application because it is a static website which does not use any third party services nor a backend server. The website is hosted using GitHub Pages which only serves the static files from the gh-pages branch of the code repository in (Appendix 2). This means that any security vulnerabilities should not arise from the application's source code itself however there is always a possibility of vulnerabilities being found in the application's dependencies in the future.

Even though vulnerabilities should not arise from the source code, bugs are likely to exist whether discovered or not as with any software project. For numerous reasons such as to prevent bugs, the application should be migrated to use TypeScript in the future. TypeScript can be thought of as a superset of ECMAScript which is the language standard for JavaScript [33]. TypeScript being a superset of ECMAScript means that valid JavaScript is also valid TypeScript which allows migration in small increments. JavaScript itself is not well suited for maintaining large software projects which TypeScript tries to solve by extending it. TypeScript introduces static typing and many other language constructs such as interfaces to JavaScript. [34] Mainly with TypeScript, static analysis could prevent bugs already at compile time and integrated development environment support would improve significantly.

The simulator application does not persist any session data which means that microprograms created by users are discarded each time the web page is refreshed. The application warns users about this behaviour and encourages to save the microprogram locally using the application's Manage Program -feature. The feature allows users to save and load the microprogram they have created in the application's own microprogram editor. Retaining the microprogram and/or the simulation's state automatically within the browser's web storage could be beneficial for better user experience. Web storage allows web applications to persist data in the user's web browser. It is possible to save data for the current session or without any expiration date using web storage. [35], [36]

5.3 Deviations from the MPM

The simulator application strives to conform to the definition of the MPM given by J. Boberg et al. (2012) [5] however few minor degrees of freedom had to be taken. The most significant difference with the specification and the application is how the clock component works. Because in the simulator application the user controls the MPM by activating control bits individually, the clock is not responsible for activating them. Instead, the Clock component in the application only displays the current clock phase to which the latest executed control bit belongs to. This gives users more control by allowing them to execute microprograms on their own phase and makes observing changes in the machine's state easier as control bits can be activated individually. This deviation can be justified as it makes the simulator application a better e-learning tool.

The application initializes registers and memory locations to zero as any initial values have not been stated by J. Boberg et al. (2012) [5] and thus presumed to be an implementation detail. However, whether the registers or memory components have been initialized with random values or not, it can be handled in the microprogram. For example, it is possible to begin the microprogram with microinstructions which initialize registers to certain values. Furthermore, the application allows users to initialize registers and the main memory with arbitrary values. This feature has been added to allow exercises with arbitrary initial conditions to be created.

5.4 Research questions

To recap, the research questions of this thesis are:

1. How to create a web simulator application which simulates the MPM introduced by J. Boberg et al. (2012) [5] using React?
2. Can the simulator application be used with nothing but a web browser and Internet access?
3. Can users create and execute arbitrary microprograms in the simulator application?

This subchapter answers the research questions based on the findings in this thesis. The first research question is largely answered in chapter 4 as the design and implementation of the web simulator application is presented in it. As mentioned in the previous subchapter 5.3, few minor deviations from the definition of the MPM had to be taken but overall, the simulator application was created by modelling the components in React using the definitions given by J. Boberg et al. (2012) [5].

The implementation given in chapter 4 does not utilize any third party services or a backend server and as such can be constructed completely as a static website. At the time of writing, it is hosted on GitHub Pages which is a service allowing users to host websites directly from git code repositories in GitHub. The simulator application can be accessed using nothing but a web browser and Internet access from the URL provided in (Appendix 3) thus answering research question number 2.

Finally, as shown in subchapter 4.3.4, a component for creating arbitrary microprograms has been added to the simulator application. A more graphical programming oriented solution is given by allowing users to select the control bits of each microinstruction using a collection of checkboxes as well as a free form approach where users can edit the microprogram in a textual representation.

6 Conclusions

This thesis focuses on a microprogrammed machine defined by J. Boberg et al. (2012) [5]. It is a simplified computer which can be microprogrammed using primitive microinstructions. The research problem solved in this thesis is to create an easily accessible and practical way of creating and executing microprograms for the MPM. A web simulator application for the MPM was created to solve the research problem. As the research problem ensued a need for developing a software artifact, the design science research methodology proposed by Peffers et. al. (2008) [12] was followed throughout the thesis from a problem-centered entry point.

The design and implementation of the simulator application is presented in chapter 4. The simulator application was implemented using React and is currently freely accessible as a static website from the URL in (Appendix 3). The simulator application is an open-source software project hosted at the URL in (Appendix 2). Most of the implementation follows the definition of the MPM however few minor deviations from the original design were taken and few additional features were created.

This thesis limits to the design and implementation of the simulator application. More rigorous evaluation of the artifact is considered a part of future work. The simulator application is going to be used as a supporting e-learning tool at LUT on the FoCS course where end user testing is going to be performed on future course implementations.

The application itself can be contributed to by either creating new issues or pull requests on the code repository in (Appendix 2). Creating unit tests, refactoring larger components, and migrating the project to use TypeScript are currently the biggest ways to contribute to the project.

References

[1] J. Boberg, ‘Jorma Boberg: Materiaaleja’, Sep. 29, 2016. http://staff.cs.utu.fi/staff/jorma.boberg/Mat/ (accessed Jun. 02, 2022).

[2] U.S. Bureau Of Labor Statistics, ‘Software Developers, Quality Assurance Analysts, and Testers’, Sep. 08, 2021. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm (accessed Jun. 21, 2022).

[3] Stack Overflow, ‘Stack Overflow Developer Survey 2016 Results’, *Stack Overflow*, 2016. https://insights.stackoverflow.com/survey/2016 (accessed Jun. 22, 2022).

[4] L. P. Luo, A. Begel, and A. J. Ko, ‘What distinguishes great software engineers?’, *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 322–352, Jan. 2020, doi: https://doi-org.ezproxy.cc.lut.fi/10.1007/s10664-019-09773-y.

[5] J. Boberg, M. Penttonen, T. Salakoski, and J. Teuhola, ‘Johdatus tietojenkäsittelytieteeseen’. Jun. 25, 2012. [Online]. Available: http://staff.cs.utu.fi/staff/jorma.boberg/Mat/JTKTMoniste_25_06_2012.pdf

[6] Cambridge University Press & Assessment, ‘simulator’, *Cambridge Dictionary*, Jul. 19, 2023. https://dictionary.cambridge.org/dictionary/english/simulator (accessed Jul. 21, 2023).

[7] Association for Computing Machinery (acm), ‘Maurice V. Wilkes Additional Materials’, 2019. https://amturing.acm.org/info/wilkes_1001395.cfm (accessed Jun. 22, 2022).

[8] Britannica, The Editors of Encyclopaedia, ‘microprogramming | Definition & Facts | Britannica’, Sep. 14, 2021. https://www.britannica.com/technology/microprogramming (accessed Jun. 22, 2022).

[9] Facebook, ‘React – A JavaScript library for building user interfaces’. https://reactjs.org/ (accessed Jun. 29, 2022).

[10] ‘React – License’. Meta, Jun. 02, 2022. Accessed: Jun. 02, 2022. [Online]. Available: https://github.com/facebook/react/blob/d300cebde2a63e742ccb8b6aa7b0f61db1ae29b4/LICENSE

[11] L. Vailshery, ‘Most used web frameworks among developers 2021’, *Statista*, Feb. 23, 2022. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/ (accessed Jun. 02, 2022).

[12] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, ‘A Design Science Research Methodology for Information Systems Research’, *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, 2008, doi: 10.2753/MIS0742-1222240302.

[13] A. Hevner and S. Chatterjee, *Design Research in Information Systems*, vol. 22. in Integrated Series in Information Systems, vol. 22. 233 Spring Street, New York, NY 10013, USA: Springer Science+Business Media, 2010. Accessed: Jun. 30, 2022. [Online]. Available: https://link-springer-com.ezproxy.cc.lut.fi/book/10.1007/978-1-4419-5653-8

[14] G. B. Gerace, ‘Microprogrammed Control for Computing Systems’, *IEEE Trans. Electron. Comput.*, vol. EC-12, no. 6, pp. 733–747, Dec. 1963, doi: 10.1109/PGEC.1963.263557.

[15] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016.

[16] A. Ltd, ‘What is Instruction Set Architecture (ISA)?’, *Arm | The Architecture for the Digital World*. https://www.arm.com/glossary/isa (accessed Jul. 01, 2022).

[17] D. A. Patterson, ‘Reduced instruction set computers’, *Commun. ACM*, vol. 28, no. 1, pp. 8–21, tammikuu 1985, doi: 10.1145/2465.214917.

[18] D. A. Patterson, ‘Microprogramming’, *Sci. Am.*, vol. 248, no. 3, pp. 50–57, 1983.

[19] A. K. Agrawala and T. G. Rauscher, *Foundations of Microprogramming: Architecture, Software, and Applications*. Academic Press, 2014.

[20] Gartner, ‘Definition of Compound Annual Growth Rate (CAGR) - Gartner Information Technology Glossary’, *Gartner Glossary*. https://www.gartner.com/en/information-technology/glossary/cagr-compound-annual-growth-rate (accessed Aug. 24, 2023).

[21] Wall Street Prep, ‘Compound Annual Growth Rate (CAGR)’, *Wall Street Prep*. https://www.wallstreetprep.com/knowledge/cagr-compound-annual-growth-rate/ (accessed Aug. 24, 2023).

[22] Polaris Market Research, ‘E-learning Market Size Global Report, 2022 - 2030’, *Polaris*, Feb. 2022. https://www.polarismarketresearch.com/index.php/industry-analysis/e-learning-market (accessed Jul. 21, 2023).

[23] Z. Bezovski and S. Poorani, ‘The Evolution of E-Learning and New Trends’, *Inf. Knowl. Manag.*, vol. 6, no. 3, Art. no. 3, Mar. 2016.

[24] MDN, ‘SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN’, Jun. 08, 2023. https://developer.mozilla.org/en-US/docs/Glossary/SPA (accessed Jul. 13, 2023).

[25] Meta Open Source, ‘Describing the UI – React’, *React*. https://react.dev/learn/describing-the-ui (accessed Jul. 13, 2023).

[26] Meta Open Source, ‘Thinking in React – React’, *React*. https://react.dev/learn/thinking-in-react (accessed Jul. 13, 2023).

[27] M. M. Mano, C. R. Kime, and T. Martin, *Logic and computer design fundamentals*, Fifth Edition. Boston: Pearson, 2016.

[28] B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*. Springer, 2019.

[29] K. D. Wagner, ‘Clock system design’, *IEEE Des. Test Comput.*, vol. 5, no. 5, pp. 9–27, Oct. 1988, doi: 10.1109/54.7979.

[30] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, ‘Agile Software Development Methods: Review and Analysis’. arXiv, Sep. 25, 2017. doi: 10.48550/arXiv.1709.08439.

[31] P. Meso and R. Jain, ‘Agile Software Development: Adaptive Systems Principles and Best Practices’, *Inf. Syst. Manag.*, vol. 23, no. 3, pp. 19–30, Jun. 2006, doi: 10.1201/1078.10580530/46108.23.3.20060601/93704.3.

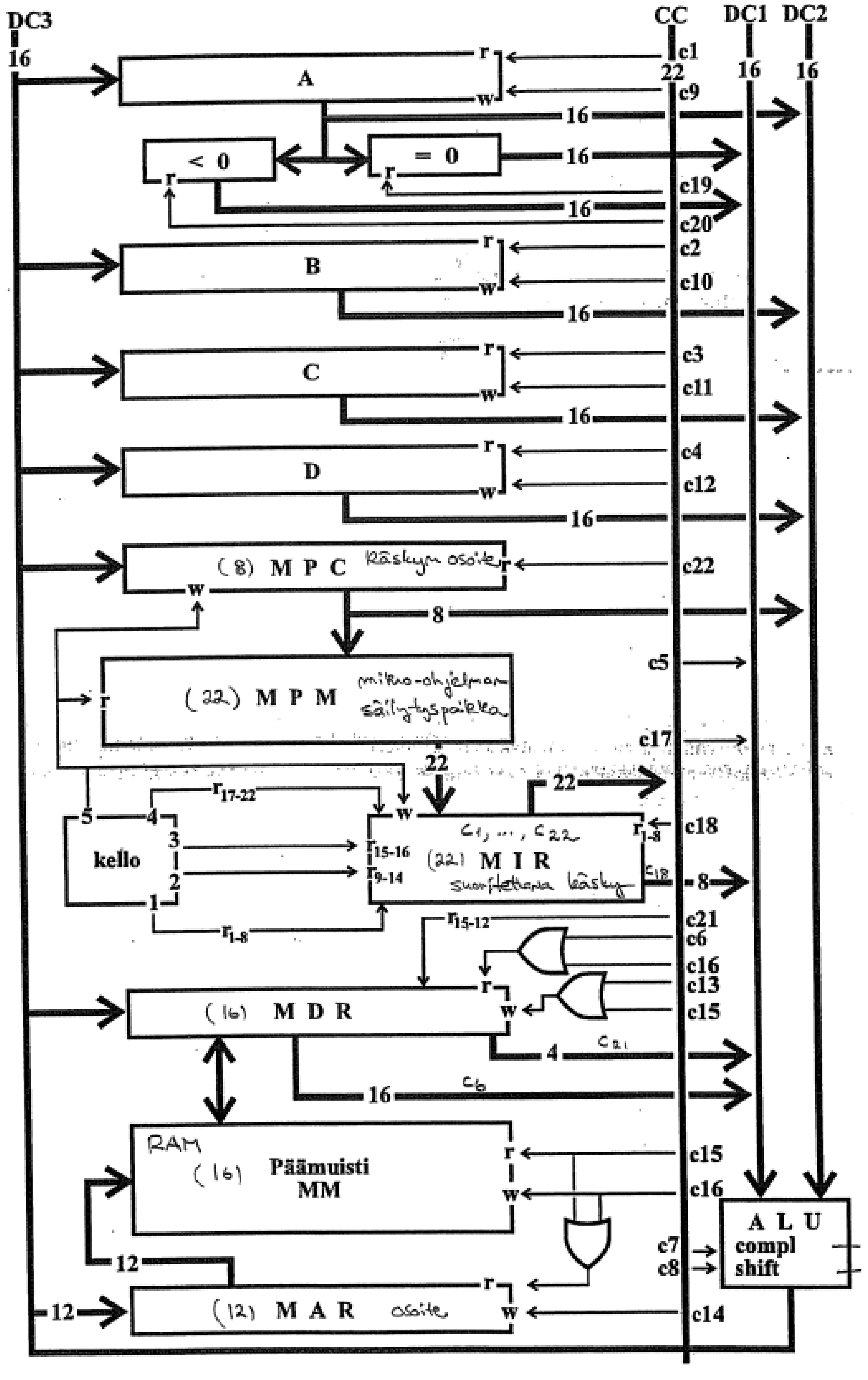
[32] Meta Open Source, ‘Writing Markup with JSX – React’, *React*. https://react.dev/learn/writing-markup-with-jsx (accessed Jul. 19, 2023).

[33] MDN, ‘JavaScript technologies overview - JavaScript | MDN’, Feb. 21, 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview (accessed Jul. 19, 2023).

[34] G. Bierman, M. Abadi, and M. Torgersen, ‘Understanding TypeScript’, in *ECOOP 2014 – Object-Oriented Programming*, R. Jones, Ed., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 257–281. doi: 10.1007/978-3-662-44202-9_11.

[35] W3Schools, ‘HTML Web Storage API’. https://www.w3schools.com/html/html5_webstorage.asp (accessed Jul. 21, 2023).
[36] MDN, ‘Window: localStorage property - Web APIs | MDN’, Apr. 08, 2023. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (accessed Jul. 21, 2023).

Appendix 1. Structure of the microprogrammed machine from [1].



Appendix 2. URL to the git code repository of the simulator application.

https://github.com/jani-heinikoski/mpm-simulator

Appendix 3. URL to the deployed simulator application.

https://jani-heinikoski.github.io/mpm-simulator/