Department of Computer Science

Submitted in part fulfilment for the degree of BEng

SimpleCPU Instruction Set Simulator

Craig Slomski

02 May 2024

Supervisor: Mike Freeman

Department of Computer Science

ACKNOWLEDGEMENTS

I would like to acknowledge my supervisor Mike Freeman for his work on the SimpleCPU project, which without this project would not have been possible, additionally I would like to thank Mike for his role in supporting this as project supervisor, as well as for developing my understanding and interest in Computer Architecture.

I would also like to thank my family for their continued support.

Department of Computer Science

STATEMENT OF ETHICS

For this project, a study was conducted to assess the usability of the final program produced, all participants were above the age of 18 and once no longer needed, all data collected will be destroyed. Outside of this there were no other significant ethical concerns relevant for the undertaking of this project.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF FIGURES

Executive summary

The aim of this project was to produce an instruction set simulator for the SimpleCPU architecture designed by Mike Freeman that was capable of accurately simulating programs, and was suitable for undergraduate teaching.

This work was motivated by the importance of computer architecture, instruction set architecture and assembly programming in the field of computer science and within the curricula, as well as by the shortcomings of current ways to simulate programs for the SimpleCPU architecture. Namely, that the current ways programs can be simulated for the architecture, using Xilinx ISE, are slow to execute and can be difficult for students to understand.

Thus, an instruction set simulator was produced using the Python 3 language. This language was chosen as all students studying the module where the relevant topics are taught in the University of York, SYS1, will have already taken a software module, SOF1, where the fundamentals of Python are taught, thus the students should be able to develop a better understanding of how a processor function by reading the Python code that simulates one.

The development of this project followed an agile methodology, where ‘sprints’ were performed and evaluated based on how well the current solution met user requirements, so that what was prioritised remained flexible throughout development.

For this project, simulators were produced for v1a and v1d of SimpleCPU, as well as an additional simulator that used the same architecture as v1d, however, allowed the user to remap instructions freely. To justify the inclusion of this, the simulator included 22 additional instructions for the user to freely map, wherever the addressing mode of the architecture and instruction matched.

To interface with the simulators, a command line interface and graphical user interface were produced. The graphical user interface offered some additional features, such as the ability to connect a GPIO peripheral, an integrated development environment where programs could be written and assembled with built-in support for linking macros.

The final program was able to execute programs thousands of times faster than the Xilinx ISE simulation of SimpleCPU, and was able to

execute programs several times faster than SimpleCPU when synthesised onto an FPGA, through use of PyPy, a Just-in-Time compiled implementation of Python.

Additionally, a user review was conducted to assess the usability of the program, for these 6 participants used the program to carry out a short exercise, taken from a SYS1 sample examination paper. All user data gathered is planned to be destroyed once no longer necessary for this project. The results of this short survey indicated that the program was accessible to those with experience in assembly programming, with all participants who had experience using the Xilinx tools stating that they found the instruction set simulator program easier to use. Furthermore, most of the participants stated that the program was suitable for use when teaching computer architecture, or specifically assembly programming. However, as the sample size of the study was small, these results are someone limited and a more thorough study with a larger sample size could be useful to conclusively assess the final programs usability.

It is reasonable to say that this project was predominantly successful in meeting its aims, as a program was produced that successfully and accurately simulated the instruction set of various SimpleCPU iterations, offered additional features and indicated a high ease of use when used by participants of a user study.

1 Introduction

Computer Science is a relatively young academic discipline, compared to other disciplines such as Mathematics or Physics, which has increasing importance as computing becomes more ubiquitous in people's lives over time. The growing importance of Computer Science can be seen clearly with the rapid growth Computer Science courses across all levels of education. At GCSE level, the number of Computer Science entrants increased by 12% in 2023, compared to the overall entrants increase of 3%, making it the fastest growing STEM subject [1]. These trends are similarly reflected at University level, where 2023 saw a 9.6% increase in people applying to study Computer Science, which was the highest year-on-year increase among any degree [2]. Thus, as a rapidly growing discipline, the way Computer Science is taught is an issue of high importance.

In particular for this project, the issue of how Computer Architecture and low-level underlying hardware concepts, are taught is of importance. Computer Architecture is a fundamental part of Computer Science, and ought to be understood by those involved in the discipline. This view is supported by the Computer Science Curricula 2013, a document which lays out what students studying the discipline should cover in their time studying the discipline. In the Architecture and Organisation section, it states "Computing professionals should not regard the computer as just a black box that executes programs by magic [3]."

The way in which Computer Architecture has been taught has varied drastically across the history of the discipline, as Computer Architecture itself has also varied in terms of what is available, affordable and practical. Early courses varied in focus from teaching large IBM mainframes and similar systems, without the students having direct experience of computers, then to teaching assembly language programming on mini-computers and then to teaching specific microprocessors [4]. As processors became more complex the focus on specific systems was dropped as the post-RISC processors used in industry were deemed too complex to practically teach students, and courses moved to teaching Computer Architecture using hypothetical or simpler architectures [4]. A major benefit of teaching using hypothetical machines, is that the architecture can be designed prioritising suitability for teaching, thus resulting in an architecture that best conveys of hardware concepts to students, even if the architecture may not be the most performant. This can even be valuable for teaching; an architecture with shortcomings can be used to demonstrate why design practices are used or avoided.

SimpleCPU is an example of a hypothetical architecture used for teaching. Currently at the University of York, SimpleCPU is used to teach undergraduate students about computer architecture and assembly programming, in the Systems and Devices 1 (SYS1), module. Currently, SimpleCPU is implemented using Xilinx ISE. The benefit of this approach is that students are easily able to see how many components of a processor exist on a logic gate level. However, this implementation has some shortcomings for use in education, including that Xilinx ISE requires 85GB of disk space to run on modern operating systems [5], and hasn't received a major update since October 2013 [6]. Additionally, once assembled and loaded into a schematic, programs can be simulated using Xilinx ISim, a component of Xilinx ISE. As simulating programs is done predominantly on a logic gate level, it is a computationally significant process, resulting in simulating programs being a slower process than necessary.

Thus, this project aims to produce a solution that addresses the identified problems with the current implementation of SimpleCPU. In order to establish clear motivation and aims for the project a literature review will be conducted, focusing on computer architecture in the computer science curriculum, and existing implementations of instruction set simulators.

2 Literature review

2.1 The importance of Computer Architecture in Computer Science

2.1.1 Computer Architecture in the Computer Science Curricula

As mentioned in the introduction, Computer Science is a rapidly expanding academic discipline. This is true not only in the intake of students, but of the domain it covers as well. This is seen in the most recent revision, Version Gamma at the time of writing, of the upcoming Computer Science Curricula 2023 report produced by the Association for Computing Machinery (ACM), IEEE Computer Society (IEEE-CS) and the Association for the Advancement of Artificial Intelligence (AAAI), which identifies recent growth in the fields of Quantum Computation and Artificial Intelligence specifically [7].

Additionally, the expansion of the domain of Computer Science can also be seen through the increase in the number of hours to cover the content of the curriculum increasing with each edition of the curriculum, from 280 hours in CC2001 to 308 hours in CS2013. As a result of this the decision was taken to move the curriculum from the previous method of two tiers, with tier I being fully covered and tier II being 80% covered, to a new system consisting of a single main ‘CS core’ that includes all content necessary for students to cover in a Computer Science course, totalling 268 hours of teaching. There is also an additional ‘KA core’ that contains optional content, if a course wants to go into more depth of an area in Computer Science without imposing a requirement.

The content that this project aims to aid the teaching of is included in the Computer Architecture knowledge area, in the Computer Science Curricula 2023 report, specifically content from the Assembly Level Machine Organization knowledge unit including von Neumann machine architecture and Control unit; instruction fetch, decode, and execution from the CS core but predominantly Instruction set architecture (ISA) and Subroutine call and return mechanisms from the KA core. Some of the following Illustrative Learning Outcomes outlined in CS2023 that this project seeks to support the teaching of includes:

- Contextualize the classical von Neumann functional units in embedded systems, particularly on-chip and off-chip memory. (CS Core)
- Comment on how instruction is executed in a classical von Neumann machine (CS Core)
- Comment on how instructions are represented at the machine level and in the context of a symbolic assembler. (KA Core)

- Comment on different instruction formats, such as addresses per instruction and variable-length vs fixed-length formats (KA Core)
- Code a simple assembly language program for string array processing and manipulation (KA Core)

2.1.2 The need for Computer Architecture on the Computer Science Curricula

Computer Architecture is deserving of a substantial role on the Computer Science Curricula. This view is perhaps best supported by the writings of Alan Clements who has wrote extensively advocating for teaching Computer Architecture and for how to teach it [4] [8] [9] [10] and played a major part in shaping the Computer Science Curricula, not only through his published work but also through being part of the CS2008 review task force [11] and through writing his book on Computer Architecture, Computer Organization and Architecture: Theme and Variations [12].

A point that Clements reiterates several times in his published work is that the 'computer lies at the heart of computer science, and that without the computer, computer science would simply be a branch of theoretical mathematics' [8] [10]. Consequently, Clements argues, regardless of industry application, a university teaching Computer Science ought to educate students how a computer is organised.

There is much discussion on what the purpose of universities should be in today's modern society [13] [14] [15]. An area of discussion specifically relevant for this project is whether universities should prioritise educating students or training students to be employable. This section is written with the view that the role of universities ought to be more than a conveyer belt into industry for students, that universities primary aim should be to educate students and that employability is merely one of many positive side effects of education. Consequently, even if a student never writes an assembly program in industry, teaching students to write basic assembly programs is worthwhile for the purpose of education itself. To not take this view is to suggest that in a world ubiquitous with computers, a person who has dedicated three years of their life to studying computer science should not understand even on a basic level, instructions as they are processed by computers and how they are executed, which does not seem reasonable. Again, this view is supported by the writing of Clements:

"The very idea of a computer science program that did not provide students with an insight into the computer would be strange in a

university that purports to educate students rather than merely train them." [10]

However, even if the view is taken that universities sole purpose should be to prepare students for industry, having a good understanding of Computer Architecture and assembly language programming is absolutely necessary for those who pursue careers in developing for embedded systems. Where programming in assembly is common practice across many different architectures including ARM and RISC-V. Another domain of Computing where assembly knowledge is required, is for writing compilers and for compiler optimisation, which is necessary to power programming languages such as C, C++, Haskell and Rust [16].

Additionally, as Computer Science is often seen as the default discipline across the computing curricula, it tends to be broad in focus and many students will choose to study Computer Science due to not wanting to narrow their focus early in their academic career or due to unavailability of other computing disciplines. Doing so should not preclude these students from gaining relevant experience for fields concerned with the low-level details of computer architecture, such as embedded systems. Furthermore, understanding Computer Architecture is useful for programmers as an understanding of architectural features can be used to employ techniques, such as parallelism, in code to improve system performance [7].

Supporting this view, when the CS2023 Task Force on Computer Science Curricula surveyed industry professionals on how important they viewed the knowledge areas, on a Likert scale of (0 – not important, up to 3 – very important) Computer Architecture was rated 1.93, in the short-term and saw a 30% increase, being rated 2.50 in the long-term [17]. Therefore, there is ample evidence to show Computer Architecture and assembly programming are important for their educational benefits and industrial applications and that they are deserving of having a role in the Computer Science Curriculum.

2.2 Computer Architecture at the University of York

2.2.1 SimpleCPU

SimpleCPU is a processor architecture used for undergraduate teaching, in the SYS1 module at the University of York, designed by Mike Freeman. For the purposes of this project there exist 2 relevant iterations of SimpleCPU: SimpleCPU_v1a and SimpleCPU_v1d. Both iterations have relatively small instruction sets and exist primarily so that students can understand how a processor functions at a low-level. The simplicity of their designs makes it easy for students to understand how they function at this level.

2.2.2 SimpleCPU_v1a

SimpleCPU_v1a utilises just four stores of data (external memory, the instruction register, the program counter and the accumulator) in order to carry out a fetch-decode-execute instruction cycle. SimpleCPU_v1a has 11 instructions (Move, Add, Sub, And, Load, Store, Addm, Subm, JumpU, JumpZ, JumpNZ) which use three modes of addressing (Intermediate, Absolute, Direct) [18]. This iteration of the architecture has many limitations, such as only having one register for performing operations, moreover, this register only consists of 8 bits meaning that only 256 unique permutations, or values that can be represented exist. Additionally, the memory only contains 256 addresses, which each consist of 16 bits. Normally, these limitations would make an architecture unsuitable to use. However, in a teaching context these limitations are valuable because, firstly, it means SimpleCPU_v1a can be fully explained over the course of a lecture, as opposed to an entire course, allowing students to spend more time in teaching learning by doing. Secondly, it allows v1a to be contrasted with v1d, to show the accommodations that need to be made to allow for more complex architectures.

2.2.3 SimpleCPU_v1d

SimpleCPU_v1d expands on v1a by introducing register addressing modes and featuring four general purpose registers, which are now 16 bits. Another way that v1d expands on v1a is by increasing the memory size to 4096 unique addresses. In order to accommodate this, the program counter has needed to be expanded to 12 bits, so that every memory address can be accessed. Additionally, a ‘LIFO stack’ has been added to store program counter addresses in order to allow subroutines to be executed, using new call and ret instructions. In total 7 additional instructions have been implemented (Call, Ret, Move [register], Load [register], Store [register], Rol, Xor). 7 more instructions are defined in the reference guide, but not implemented (Or, Ror, Add [register], Sub [register], And [register], Or [Register], Asl). There is also room for 5 more instructions that are undefined however are restricted to certain addressing modes (1 × immediate, 2 × register, 2 × register indirect) [19]. This allows students to define new instructions, facilitating the learning of instruction set architecture, in a way which would not be possible using pre-defined architectures.

2.2.4 Xilinx ISE

SimpleCPU is predominantly implemented and used through the program, Xilinx ISE. The Xilinx toolchain allows for SimpleCPU to be synthesised onto a FPGA, or simulated through Xilinx ISim. Xilinx ISE is a large complex program, primarily meant for designing schematics for FPGAs, which results in the program being more confusing than

necessary when used to simply simulate assembly programs. Simulating the behaviour of SimpleCPU using Xilinx ISE is also a computationally significant process, which results in simulation being slower than using an FPGA.

2.3 Instruction Set Simulators

Developing software based Instruction simulators of architectures (e.g. MIPS or ARM) is a common practice [20] [21] [22] [23] [24] in Computer Science and these software solutions are often used for teaching purposes across many different architectures. Utilising such simulations come with advantages over running assembly code over actual hardware, commonly speed, ease of use and ease of debugging however some Instruction Set Simulators go further and offer additional features such as visualising pipelining [21].

The standard implementation of an Instruction Set Simulator (ISS) involves analysing a given processor and its instructions and then constructing a program that for every instruction, mimics the output of the processor with respect to registers and memory [25].

2.3.1 CPU Sim

Rather than an ISS designed around one specific instruction set architecture, CPU Sim is a program that allows users to design and alter architectures at a register-transfer level including assembly language instructions and machine instructions which exist as a sequence of microinstructions. The user can then write and run programs on the CPU that they have specified [26].

A major strength of CPU Sim over traditional ISS's is that allowing users to define architectures makes it extensible to the user. The benefit of this in an education sense is that it allows the user to interact and engage with architecture beyond assembly programming, it is easy to imagine a series of exercises where the user makes a series of improvements to SimpleCPU_v1a, (e.g. expanding memory size), until the final architecture is functionally equivalent to SimpleCPU_v1d. This allows students to easily learn about computer architecture in a practical way, without needing to use physical hardware, which comes with significantly more difficulties than using a software simulation.

Some other features of CPU Sim include the ability to go through assembly programs step by step, the ability to view contents of memory and the ability to view contents of registers.

While CPU Sim, allows for users to define an architecture, there are some severe limitations on what the user is able to do using the program. One major limitation relevant for SimpleCPU, is that CPU

Sim only allows users to define instructions using a single opcode. This is relevant for implementing SimpleCPU_v1d, which explicitly relies on having both a left and right opcode for instructions with register addressing modes.

2.3.2 emulsiV

emulsiV is a web based [27] ISS of the RISC-V instruction set, a common choice of architecture for simulators designed to teach computer architecture, due to its application in industry [28].

One of the core features of emulsiV is its clear visualisation of the computer architecture. This allows the user to see how components of the architecture interact and change state step by step as an assembly program is ran.

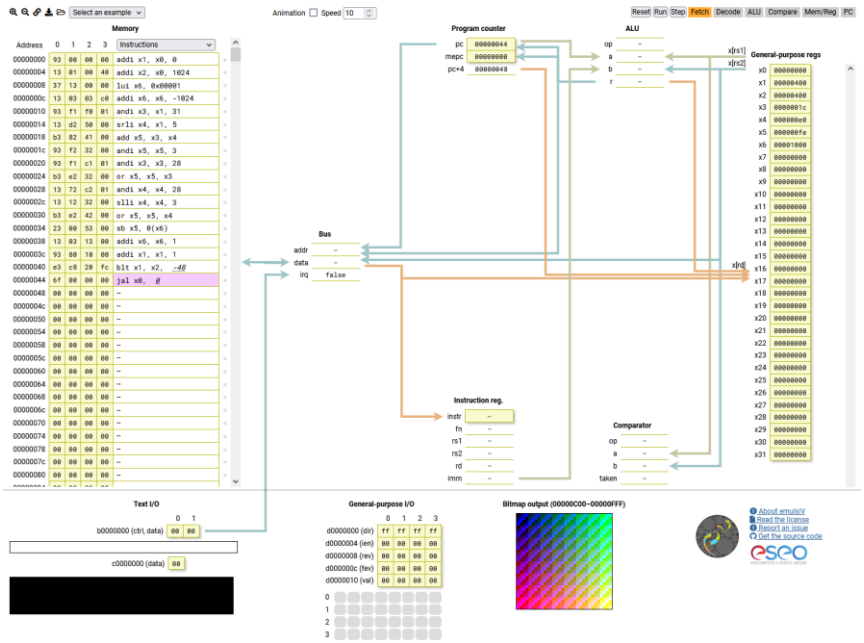


Figure 1 The user interface for emulsiV

Additionally, emulsiV also offers basic I/O peripherals, including text I/O, general purpose I/O (GPIO), and Bitmap output. The Bitmap output uses the data stored between addresses 00000c00 – 00000FFF to display an image. Offering peripherals extends the functionality of the simulator by increasing the potential of programs that can be wrote, for example there are several programs that use the functionality of the peripherals that are available by default with the software including one that uses the four buttons with the GPIO to move a cursor over the bitmap image depending on direction of the button pressed.

emulsiV is limited in its potential for teaching, particularly for first year undergraduate students, due to its use of the RISC-V architecture. While the architecture is sometimes used for education of computer architecture, the architecture is significantly more complicated than older architectures or SimpleCPU, and requires significantly more information to understand. This is perhaps illustrated best through the RISC-V Instruction set manual [29], where volume 1 alone is 145 pages. Comparatively, SimpleCPU_v1d's reference guide is 11 pages. SimpleCPU allows students to quickly fully understand an architecture, and write assembly for it, making it easy to learn and apply, which has major advantages for teaching purposes.

2.3.3 Python 6502 Emulator

Hans Spaans 6502 Emulator is a Python implementation of MOS Technologies 1975 6502 Microprocessor [30]. The Python program emulates the 6502 by reading the memory address specified by a program counter variable, and using two lookup tables with the value in the memory address to construct a string that is the same as a function name representing a given instruction and then executing said function through use of an eval function. This is repeated in a while loop, and is a sensible, readable and understandable way of implementing the functionality of the processor. Additionally, this implementation does not require any external Python libraries to run, and thus the code should be easy for an undergrad student to run and understand.

A feature that this emulator has that the previously mentioned solutions do not is unit testing. This is important in development to ensure that instructions have been implemented correctly.

A drawback of this solution is the lack of a graphical user interface that the other implementations had, making it somewhat harder to use and view than the aforementioned implementations.

2.4 Advantages of using Instruction Set Simulators for undergraduate teaching of Computer Architecture

After reviewing previous implementations of Instruction Set Simulators, it is possible to clearly state why they are valuable for supporting undergraduate teaching of computer architecture.

Speed – ISS's are not restricted by hardware; they are portable by design, meaning they can run on state of the art modern CPUs, even if the hardware they are simulating is from many decades ago or runs at lower clock rates for power consumption reasons (common in embedded systems [31]), this means that Instruction Set Simulators

can simulate assembly programs many times faster through utilisation of superior hardware.

Convenience – ISS's allow users to run assembly programs on many different architectures, which can be significantly more convenient than requiring on specific hardware, which adds an additional cost on running programs. An important factor in this is that use of an ISS, in most cases, removes the step of transferring a program to a piece of hardware, instead allowing the user to simulate a program directly from the hardware they have developed it on.

Flexibility – ISS's can provide the user with the capabilities to alter the instruction set or other parts of the systems architecture such as registers or memory size. This can be valuable for teaching purposes, as it can be used to demonstrate the advantages and drawbacks of designing hardware in certain ways. For example, increasing the number of registers, can require a larger number of bits to address register in the instruction set architecture.

Usability – ISS's can offer usability features not available from hardware alone, such as a clear view into contents of registers and memory, the ability to step through programs instruction by instruction. These features are particularly useful for debugging programs. Another usability feature is easy access to simulated peripherals, such as those offered by emulsiV [27].

2.5 Summary of Project Aims

Following a review of relevant literature, it is possible to surmise aims for the outcome of this project.

1. The project should produce an instruction set simulator program, capable of accurately simulating the behaviour of SimpleCPU_v1a and v1d, for the purpose of supporting undergraduate teaching.
2. The program should offer some support for additional features not currently offered by the Xilinx implementation of SimpleCPU.
3. The program should be more accessible to use than the Xilinx implementation of SimpleCPU.
4. The program should be able to outperform existing implementations of the SimpleCPU architecture.

3 Methodology

3.1 User Requirements

In order to define the user requirements, the users of the system, and what they want from the system, needs to be well understood. During requirements elicitation two separate user groups were identified, students and educators. The student user group is characterised by wanting to use a program that facilitates their understanding of computer architecture in a frictionless way. The educator user group considers the reception of the student user group, however, they have additional wants for the system; broadly speaking the educators aim to use a system with extensibility, so that more can be done using the system to facilitate learning and understanding. To best understand the needs of each group, use case diagrams were produced, with user stories and what features that user story implies a need for.

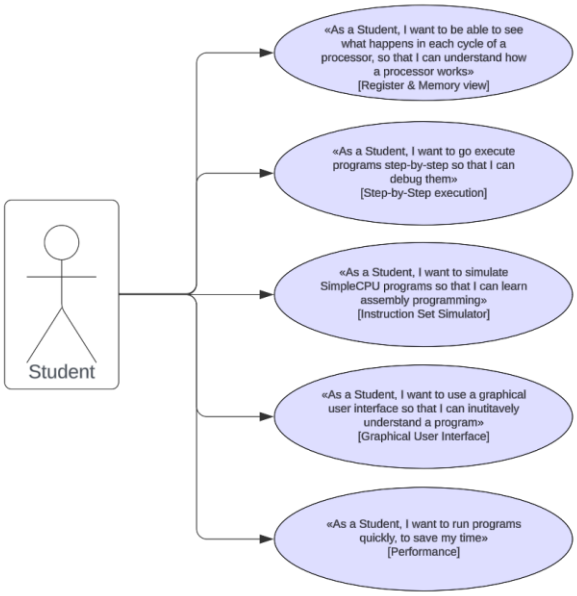


Figure 2 Use Case Diagram for a student

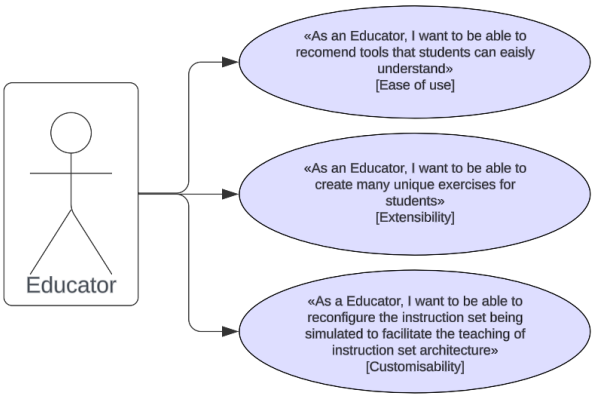


Figure 3 Use Case Diagram for an educator

Following the creation of use case diagrams, and user stories, it was now possible to reasonably define useful user requirements for the system.

Requirement ID	Requirement Type	Description
1.1	Functional	The program must be able to accurately simulate on a register level the behaviour of SimpleCPU_v1a for all instructions.
1.2	Functional	The program must be able to accurately simulate on a register level the behaviour of SimpleCPU_v1d for all implemented instructions.
1.3	Functional	The program should support an implementation of the instructions defined in the reference guide.
2.1	Functional	The program should allow the users to customise the instruction set of SimpleCPU_v1d and implement new instructions.
2.2	Functional	The program should allow the user to assemble programs for SimpleCPU_v1d with an altered instruction set, so that programs can be run.
2.3	Functional	The program should allow users to dump the contents of memory.
2.4	Functional	The program should offer support for emulating a GPIO peripheral.
2.5	Functional	The program should offer support for emulating a display peripheral.
2.6	Functional	The program should offer an Integrated Development Environment for developing assembly programs.
2.7	Non-Functional	Developing, assembling and running programs should be seamless and frictionless.
3.1	Functional	The user must be able to interface with the program through a command line.
3.2	Functional	The user must be able to interface with the program through a graphical user interface.
3.3	Non-Functional	The program should be easy to use and understand.
4.1	Functional	The program should be able to outperform existing ways of running or simulating SimpleCPU programs.

Figure 4 User Requirements

3.2 Design

The program will be written in Python 3, as it is a language that every undergraduate student at the University of York will have experience with, due to undertaking a module, SOF1, the semester before this project would be relevant, which teaches Python and programming fundamentals. Thus, use of Python will allow students to best understand the code and consequently best, facilitate their understanding of computer architecture. Another reason that Python was deemed suitable was due to having the TKinter library built in. Tkinter is a library that interfaces with the Tcl/Tk toolkit [32], which facilitates the building of graphical user interfaces necessary for this project.

A constraint that will be adhered to for the development of this project, is that the project may not use any external libraries that do not come preinstalled with Python. The reason for this constraint is so that undergraduate students who may only have a few months of experience with Python, will be able to run the program without needing to potentially diagnose, and remedy runtime errors through installing relevant dependencies, which could potentially be a barrier for use of the program, going against the aims of the project to make a program that is more accessible for use in teaching Computer Architecture.

An issue to consider when using Python is its performance as a language, due to being an interpreted language Python is typically magnitudes slower than many other languages, such as Java or C++ [33] [34] which can cause some difficulty for real-time systems [35]. However, because the system being simulated has a significantly lower clock rate (10Mhz) than modern systems, which tend to be in the 3Ghz range, Python should come close too, or better the performance of SimpleCPU synthesised onto an FPGA.

In terms of architecture, the system will be composed of five major components. The first three of these components will be classes corresponding to versions of the processor, that are able to simulate, one for v1a, v1d and a class that follows the v1d architecture while allowing the user to alter the instruction set. The other two components will be responsible for allowing the user to interface with these classes, and will function through creating an instance of one of the classes. One of these components will be a Python file that allows the user to interface through a command line, and the other will be a Python file that implements a graphical user interface which offers additional features, such as peripherals.

The processor classes will consist of variables, to represent the contents of registers and memory, with each individual register or address being represented by a corresponding int value, a function to load data (including programs) into memory, a function to reset the processor, a function corresponding to each instruction, a function to increment the program counter and an execute function responsible for evaluating and performing the correct instruction.

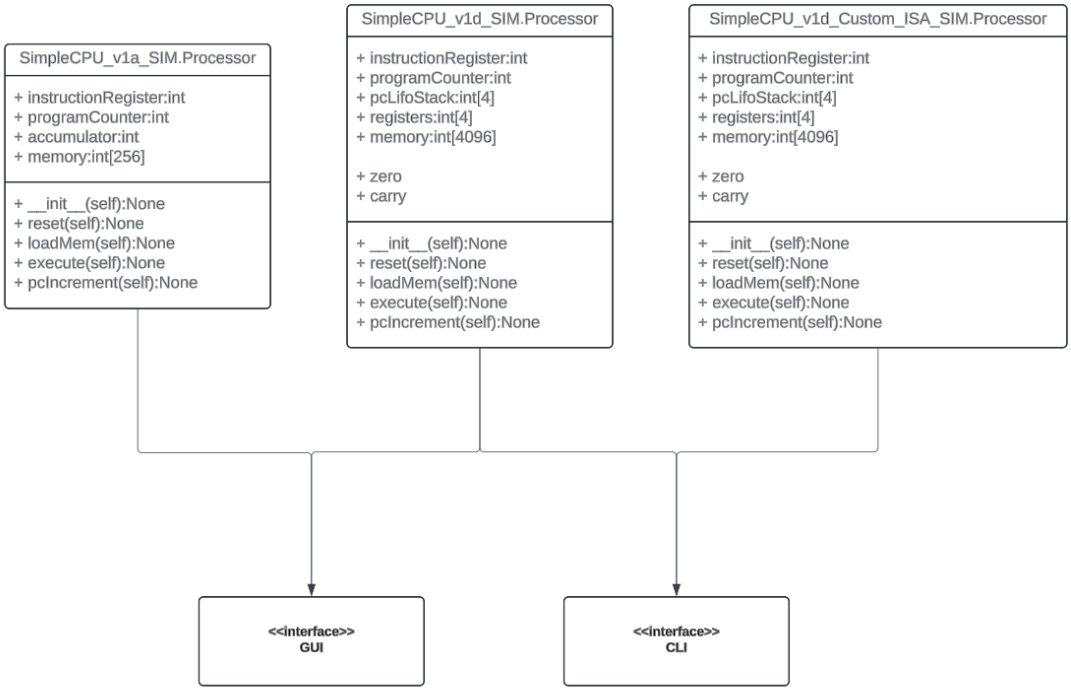


Figure 5 UML Component Diagram

4 Implementation

In the development for this project an agile approach was taken, where features went through a develop, test and bug-fix cycle, leading to an incremental development process which fell into ‘sprints’. This seemed to be the most appropriate way to structure development as agile is most suited for scenarios where the software being developed is new and development is small scale, i.e. does not involve large teams or an organisation [36]. Perusing this approach towards development later proved warranted as it allowed priorities to be shifted based on how well the program met the criteria specified in the user requirements.

4.1 Sprint 1: SimpleCPU_v1a and SimpleCPU_v1d Simulator

The first Sprint focused on developing a Python implementation of SimpleCPU_v1a and SimpleCPUv1d. The implementation of v1a was relatively straight forward and implemented without issue.

```
def execute(self: object) -> None:
    self.instructionRegister = self.memory[self.programCounter]
    opcode = int((self.instructionRegister & 61440) / 4096)
    print(opcode)
    eval("self."+self.opcodes[opcode]+"()")
    self.pcIncrement()

def pcIncrement(self: object) -> None:
    self.programCounter = (self.programCounter + 1) % 256

def move(self: object) -> None:
    value = self.instructionRegister % 256
    self.accumulator = value
```

Figure 6 SimpleCPU_v1a early code snippet

The initial implementation of v1a consisted of an array strings representing each opcode which were ordered according to their appearance in the instruction set and shared names with functions that carried out that instruction. Thus, the index could be obtained through manipulation of the opcode in the instruction register, and the instruction could be executed through an eval function that looked up the index in the opcode array. This look up method was chosen because it was used in other ISS implementations [30]. It came with drawbacks, most significantly, it was slow. However, this was not yet a concern during this stage of development. The final class featured many changes over the code snippet featured, such as moving the pcIncrement() function to before the evaluation and changing the names of instruction functions to include the addressing mode of the

instruction it corresponded to. The final execute function in SimpleCPU_v1d can be viewed in Figure 24.

The implementation of v1d began by using the v1a implementation as a base, and altering variables and instructions to reflect the architecture of v1d, such as by expanding the memory array size to 4096 and implementing a Program Counter Lifo Stack. Due to the additional complexity of v1d, some edge cases were not implemented correctly, which was noticed at the end of this sprint during testing. Some examples of bugs that were caught included some direct instructions not padding the most significant bit and the Rol instruction not producing a carry bit when the highest bit is active, which was inaccurate to the version of v1d used for testing. By the end of Sprint 1, two Instruction Set Simulators were produced, that could accurately simulate all instructions (including the non-implemented instructions on v1d) for each processor version.

To load programs '.dat' files were chosen as the ideal format, because they represented each instruction as 16 binary bits, which is closest in format to how instructions are typically represented as machine code, including in the SimpleCPU reference guides, therefore it is reasonable to assume it is the most important representation for students to be able to understand, and the one they are most likely to understand, when compared to other instruction formats produced by the SimpleCPU assembler. Additionally, a decision was made not to allow the program to accept and automatically assemble '.asm' files, because it was deemed important to force the user to assemble their programs, to facilitate the understanding of how programs exist in hardware and memory.

4.2 Sprint 2: Interfaces

The next Sprint began by implementing the interfaces that the user would interface with the ISS. The first was the Command Line Interface (CLI), which was simple to implement from the pre-existing class structure of the ISS. The CLI was implemented so that it took three arguments, ("-i": The file that would be loaded into memory and ran by the ISS, "-o": The name of the .dat file that would be produced once the program had halted, "-v": the version on SimpleCPU to use (assumed to be v1d if left blank)). To determine when a program was halted two functions were written in the CLI, a program was deemed halted before two consecutive 'move (RA/ACC) 0' instruction cycles, (the instruction when memory is blank, typically at the end of a program), or when it jumps unconditionally to its current address (an otherwise never-ending loop).

Development of the Graphical User Interface (GUI) began by using the Visual TK web application [37], to design a simple interface. Although

this tool was not used for the majority of the user interface design, the overall structure of the code remained similar to the base file when exported.

It was when running a simple program, ‘silly-long-program.asm’ (Shown in Figure 14) a program written to run absurdly long, that performance clearly became an issue. When running this program, it took the ISS approximately 9 seconds to execute ~1,000,000 instructions. This was an issue as using this metric it can be calculated that the ISS was running 29.7 times slower than a program on an FPGA. Thus, focus shifted to optimising performance.

4.3 Sprint 3: Optimisation

To improve performance significant changes were made. First, the eval() function was replaced with a case statement, that executed a function if a opcode matched a case, which led to a significant improvement in performance, secondly the SimpleCPU_v1d ISS was rewrote to use operations that were theoretically quicker than those previously used, for example, most modulus operators were replaced with, and operators with the value being reduced by one, however this did not measurably improve performance to any significant degree. Finally, the number of instructions to be executed was passed into the execute function, instead of calling the function, the number of times required, which resulted in a small but considerable improvement to performance.

Upon implementing these optimisations, the program was able to execute ~1,115,500 instructions in a second, which was around a ~935% increase in performance. While this was a significant increase in performance, this still meant that the ISS executed instructions at around ~35% of the rate of SimpleCPU synthesised onto an FPGA, which execution, ideally should have been faster than. Without any clear ways to further optimise the program, the next sprint began with a focus on implementing additional features to the program.

4.4 Sprint 4: Additional Features

The next stage in development shifted priorities to implementing additional features. The first additional feature implemented was an Integrated Development Environment, which allowed the user to write programs for SimpleCPU in the program, and assemble them with a single button through a visual interface, the use of .m4 macro files was also supported.

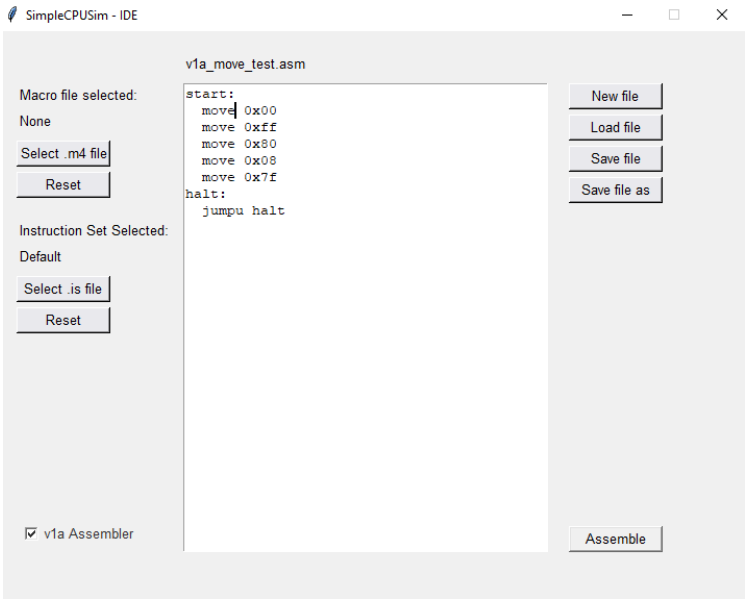


Figure 7 Interface for IDE

To implement this the program used an altered version of the pre-existing assembler for SimpleCPU produced by Mike Freeman [38], additionally the program M4 for Windows [39] was used to translate macros in programs.

The next additional feature implemented was to allow the user to customise the instruction set of SimpleCPU_v1d, this was a significantly more complex task. The first stage of implementation for this feature was to implement additional instructions in a new processor class, this followed the same architecture as v1d but instructions were no longer called in the evaluation function and the class featured 22 additional instructions (Additional instructions shown in Figure 15). Although 5 of these were simple nop instructions, most provided some operation not present such as multiplication, division and modulus, offered a novel feature, such as a ret instruction that didn't increment the LIFO stack counter, or offered instructions in addressing modes that they otherwise were not available in.

To allow the user to define an instruction set, a menu was implemented in the GUI that allowed the user to assign instructions to opcodes, as long as the addressing modes matched. This could be used to save '.is' files that defined the details of an instruction set with the name, arguments (operand and/or register(s)), opcode (with the bits for 'arguments' labelled with 'R' or 'K') and addressing type of each instruction being listed. an example of which is shown in Figure 13. From here a new python file was produced, 'selfModifying.py' that

would load a '.is' file and would use this to construct a new processor.py file, that adhered to the details provided.

This approach was heavily inspired by model-to-text automated code generation in the eclipse epsilon stack [40] (with the .is file being equivalent to the model in this instance) and had the advantage of allowing for instruction customisation, whilst still remaining performant.

To assemble programs written for the custom instruction set, a new assembler also needed to be produced, as the current assembler was hardcoded to support the given instruction set of SimpleCPU. With the given details from the '.is' file a dynamic assembler was able to be produced by reading assembly code, checking if a line of assembly code matched the format of a given instruction in the '.is' file and then inserting the relevant arguments in the relevant format in the opcode.

With these functions implemented it became possible for a user to define an instruction set using the total 47 available instructions, assemble a program using this instruction set and simulate the program using an auto-generated processor ISS using the instruction set.

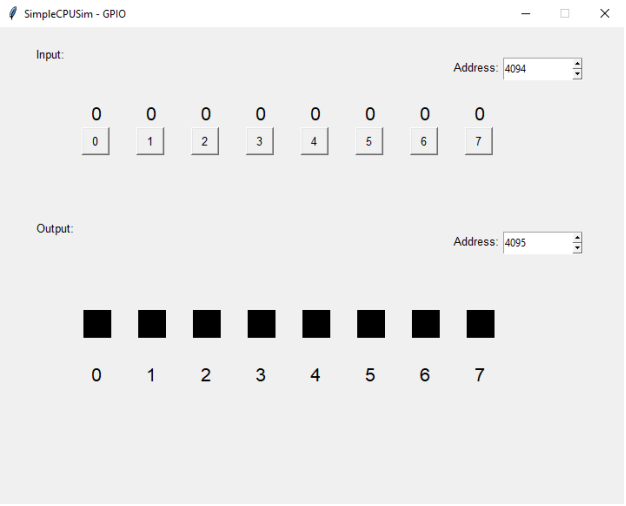


Figure 8 Interface for GPIO

Additionally, peripheral simulation support was implemented through the GUI. Due to time constraints the only peripheral implemented was a GPIO device that allowed the user to choose an address in memory for an input to be written to and another for an output, where outputs were simulated through virtual 'lights' when a bit was turned on.

The last additional feature implemented was an option to dump contents of memory to a .dat file, accessible from a button in the GUI.

4.5 Sprint 5: Migration to PyPy

It was during the later stages of development, that research into the efficiencies of programming languages that led to the discovery of PyPy, a replacement for CPython [41], (The Python implementation distributed by the Python Software Organisation [42]). Rather than implementing Python through an interpreter it, like CPython does, PyPy implements Python through a Just-in-Time compiler, which results in significantly better performance for this use case. When utilising PyPy an increase in performance of several magnitudes was observed, all while the program remained fully compatible using default CPython, which is important so that students can run the program without altering their Installation of Python. However, when using the PyPy implementation, the custom Instruction Set implementation would behave inconsistently, due to the program being modified mid-execution. Because of this the implementation was rewrote.

The processor object and the instructions were split into two separate files. To execute instructions the processor was given an additional variable (instruction: an array of strings which consisted of an instruction's addressing mode and name split by an underscore) and function (loadInstructions: a function that constructed and assigned a value to the new variable, so that each instructions index corresponded to the value of its opcode). With these implemented the execute function was altered so that it would calculate the integer value of an opcode from the instruction register value, and use this value to return the string located at that index in the instruction variable. Finally, the function performs a match-case statement with the string obtained from the instruction, and performs the instruction that it corresponds to, then increments the program counter.

Additionally, because PyPy is released under a MIT license, it is possible to redistribute its pre-compiled binaries with the program, allowing for simple scripts to be produced to launch the program using the PyPy implementation of Python. This was done to remove possibilities for error from the end user, enhancing the usability of the program.

5 Evaluation

5.1 Accuracy

In-order to ensure that user requirements 1.1 and 1.2 were met, it was important that the ISS would be able to fully accurately simulate instructions identically to how they ran on SimpleCPU. In order to ensure this, comprehensive testing was carried out, through the writing of 16 test programs (6 for SimpleCPU_v1a, 10 for SimpleCPU_v1d), which were designed to monitor behaviour in novel scenarios, for example, v1d_subroutine_test.asm, executed 5 call instructions, 4 of which were nested, causing the LIFO stack pointer of v1d to overflow. The behaviour of SimpleCPU executing these programs, as simulated through the Xilinx ISim software, was logged and Python unit tests were written that executed the same programs using the Python ISS simulator class of the processor version that program was written for. Each unit test checked for equivalency in the relevant CPU data stores for each stage of execution. An example of an assembly program wrote for unit testing is shown in Figure 29, and a Python unit test is shown in Figure 31. All unit tests successfully passed.

Test name	Instruction count	Result
v1a_add_test	6	Pass
v1a_and_test	6	Pass
v1a_jump_test	10	Pass
v1a_mem_test	19	Pass
v1a_move_test	6	Pass
v1a_sub_test	7	Pass
v1d_add_test	19	Pass
v1d_and_test	28	Pass
v1d_jump_test	23	Pass
v1d_mem_test	30	Pass
v1d_move_test	13	Pass
v1d_register_test	13	Pass
v1d_rol_test	26	Pass
v1d_sub_test	19	Pass
v1d_subroutine_test	22	Pass
v1d_xor_test	8	Pass

Figure 9 ISS accuracy tests

Following these tests, it is highly likely that the instruction set simulators for v1a and v1d are cycle accurate, in that the representation of the processor matches the Xilinx implementation of SimpleCPU after every instruction, for every instruction. Therefore, it is reasonable to say that user requirements 1.1 and 1.2 have been successfully met.

5.1.1 Accuracy of Unimplemented v1d instructions

There are 7 instructions on v1d, that are implemented in the ISS, however are not implemented in the Xilinx ISE implementation of SimpleCPU, thus these instructions cannot be tested in the same way, For these instructions randomly generated values were used, and tested by asserting these values were equivalent to values that could be reasoned, for example it could be reasoned that a given value will be identical after 1 rotate right instruction, or 15 rotate left instructions. Additionally, the instructions tested were executed through inserting the instruction register value into memory, in order to ensure that the execute function evaluated each instruction correctly. 6 test methods were produced, one for each previously unimplemented instruction, in the python file ‘v1dUnimplementedTests.py’, with each method attempting the instruction with random values 10,000 times to ensure rigor in the testing. An example test is shown in Figure 30. All tests executed successfully every time the test program was ran.

5.2 Performance

In-order to test performance, the ‘silly-long-program.asm’ was replaced with benchmarking programs for each processor, as the aforementioned program had a major shortcoming for testing overall performance, in that it did not use the full instruction set in its main loop, meaning that if instructions took different amounts of time to execute, which they inevitably would in a Python implementation, it would not be an accurate benchmark for the entire system. The full benchmark programs are available in the appendix.

Program Name	**Instruction Cycles taken to execute**
v1a_benchmark	182786810
v1d_benchmark	448154091

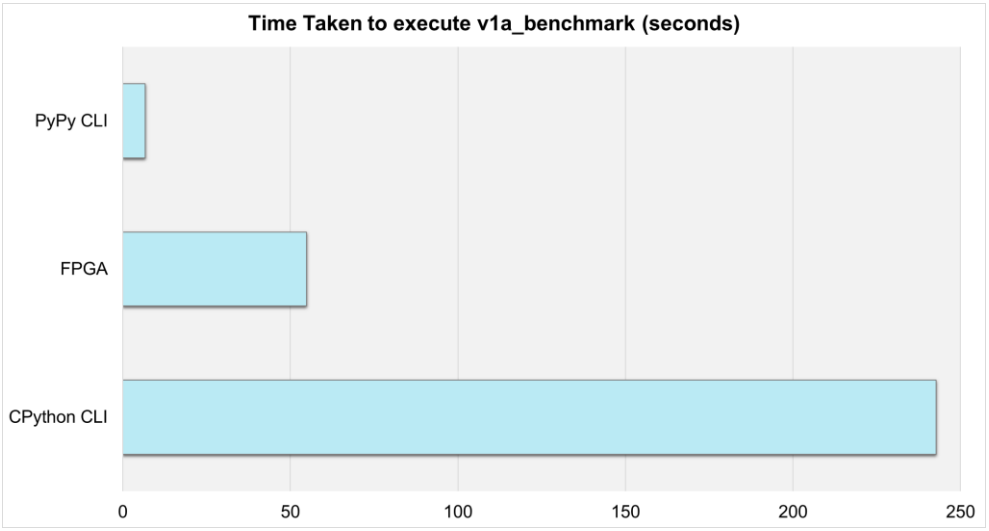


Figure 10 v1a_Benchmark Results

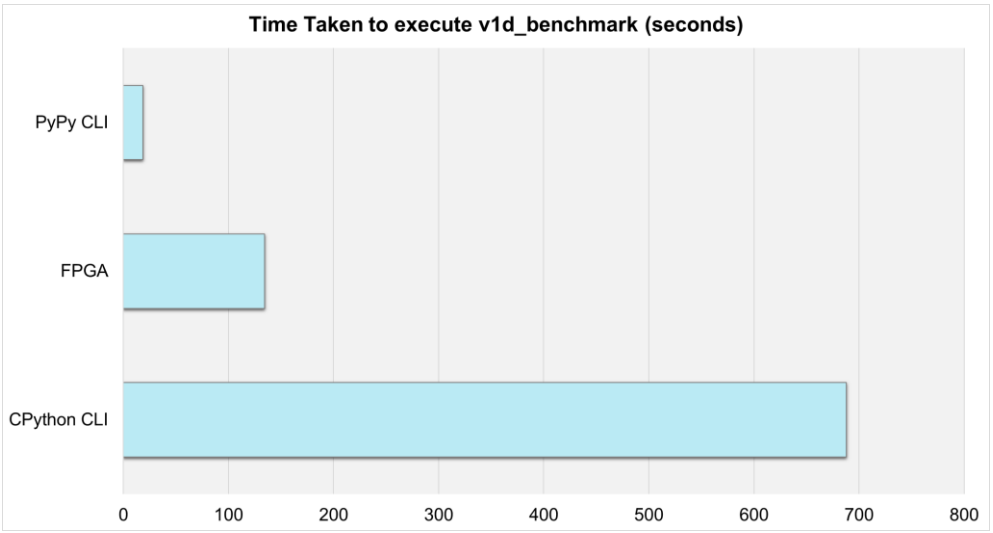


Figure 11 v1d_Benchmark Results

For these results a simulation time was calculated based on simulating a period of execution using Xilinx ISim, however the result calculated was not included in the charts for scaling purposes. It was calculated it would take Xilinx ISim 11329 seconds to fully simulate the v1a_benchmark simulation and 79000 seconds to fully simulate the v1d_benchmark simulation. Thus, it can be said the program was able to massively improve simulating SimpleCPU in terms of performance when compared to existing methods, with the PyPy implementation running the benchmarks thousands of times faster than Xilinx ISim. The Python implementation benchmarks were executed four times with a Windows Desktop PC utilising an Intel Core i5-10500 CPU with a clock rate of 3.1GHz. From this the average result was then taken. Full results are available and calculations are shown in Figure 18-23. Following these benchmarks it is reasonable to say that user

requirement 4.1 has been met as the implementation of the ISS outperforms both Xilinx ISim and an FPGA, however this must be said with the caveat that this is only the case when the PyPy implementation of Python is used, rather than the more common CPython implementation. If PyPy is not used, the program will fail to meet this requirement. The performance requirement is the only requirement that is dependent on Python implementation used, and PyPy has been included with the program, with launch files to mitigate some of the inconveniences caused by requiring a different Python implementation than most users are familiar with for this requirement to be met.

5.3 Features

In terms of instructions customisability, a user is able to remap instructions, so long as they use the addressing mode of a given opcode and 22 additional instructions were added to support and justify this. The additional instructions implemented were designed to encourage users to think about the challenges when designing an instruction set architecture, and how these designs are shaped by the underlying hardware. A good example of this is the 3 different implementations of the rol instruction.

[Direct] rol - the default instruction implemented in SimpleCPU_v1d, shifts all bits in a given register by 1 bit. This can be very slow, in the use case where a program has to rotate a register left several places, many rol instructions need to be executed, taking many instruction cycles. This also takes up a significant amount of room in SimpleCPU's memory.

[Direct] rol8 - shifts the registers bits left by 8 spaces in a single instruction cycle. This improves the aforementioned problems, however, the instructions usability for other purposes is significantly worse.

[Immediate] rol - allows the programmer to specify how many units should be shifted, however uses a valuable immediate instruction slot, of which there is only six available.

These three implementations of the rol instruction all have advantages and drawbacks, which facilitates thought from the user as to which instruction should be used for a specific use case, thus in this way the program facilitates education of the various factors considered when designing an instruction set architecture.

A drawback with how instruction customisability was implemented was that the program does not allow users to define instructions themselves. Although the way that the custom_v1d ISS has been

designed allows for extensibility, by adding an additional method, case and relevant instruction details to the instruction formats file, this cannot be done from within the program itself. Therefore, it can be said that the final program has only partially succeeded in meeting user requirement 2.1.

To support the implementation of user requirement 2.1, a custom assembler was produced, that used the details defined in a user's instruction set to assemble .dat files, that followed the given instruction formats defined by users. The custom assembler used the existing SimpleCPU assembler, produced by Mike Freeman, to perform a first pass, which made the assembly code easier to parse, and then compared the code to the instruction set file, and wrote the assembled .dat machine code interpretation when the assembly code matched the instruction set format provided in the '.is' file. Therefore, the program was able to meet user requirement 2.2.

In terms of how users were able to interact with the program, both a Graphical User Interface and Command Line Interface were produced. The command line simply loads a .dat file into memory, and executes using a specified SimpleCPU version until the program is deemed halted. Additionally, the user has the option to specify an instruction set file, to use a custom instruction set, and the option to dump the contents of memory after execution. Therefore, user requirement 3.1 has been met.

The Graphical user interface offers significantly more features. The user is able to select a .dat file to execute, however the user has more options for execution. The user is able to go through a given program step by step, or can select run to automatically run it, or pause to pause execution. The speed that a program is run can also be decided by the user, with the program executing (2^x-1) instructions per main loop of the program with x being a number the user decides between 1 and 25. Interestingly, the GUI was able to execute programs faster than the CLI, as the GUI executes with the assumption the program being run could be continuous rather than one which halts, it foregoes a halt check until the end of a main loop, if the program is halted, it then reverts to the previous state of the processor and checks for a halt at every cycle. This can have a draw back for longer programs, in that theoretically, if the program didn't contain a self-referential jumpu instruction, the ISS may not halt at the end of a program, but would continue to execute unpredictably. This was not deemed a problem however as firstly, this could easily be circumvented if the user follows the convention of a self-referential jumpu instruction and secondly, because the program would not halt on the Xilinx implementation of SimpleCPU; halting in this instance is merely a convention applied onto the ISS based on assumed programmer intent.

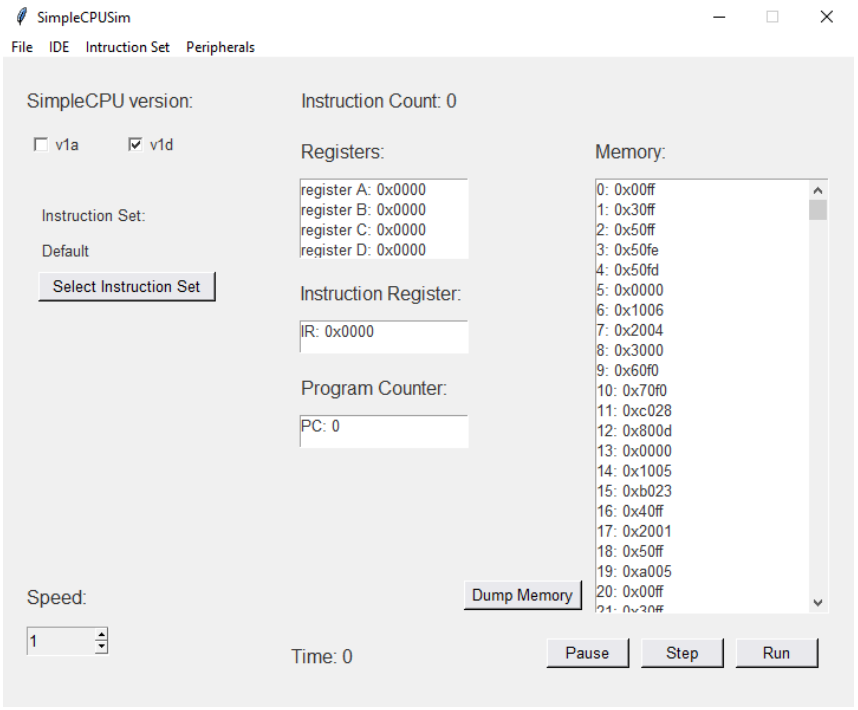


Figure 12 Final Main GUI

Other features implemented in the GUI include a display of all contents of registers (including the instruction register), the program counter, and all memory locations. Additionally, the GUI also supported dumping memory. Therefore, the program clearly met user requirements 3.2.

In terms of peripherals, a basic GPIO peripheral was implemented, that provided 8 buttons, that wrote a bit to a specified memory address, and provided 8 ‘lights’ that turned red if a bit was on. This facilitates the writing of programs that branch based on user input or perform operations based on the user input in some way, extending the usability of the program, and ensuring that the program met user requirement 2.4.

Due to time constraints, a display peripheral was not implemented consequently, the program was unable to meet user requirement 2.5.

An integrated development environment (IDE) was fully implemented for the program. This greatly streamlined the process of developing assembly programs, as they could be written assembled and ran all within one application, and without the need to use a command-line which has some obvious drawbacks from a usability point of view [43] [44]. The IDE allowed users to load files, save files and ‘save as’ files. It also had support for .m4 files, in terms of both writing and linking for

assembly, further streamlining development. Thus, the program clearly met user requirement 2.6.

5.4 Usability

To assess the usability of the program a study was carried out, where students with experience of the computing discipline would use the program to complete an exercise from a sample SYS1 paper that involved drawing a red cross onto an image. To facilitate this, two python programs were produced that stored an image into a .dat file and loaded an image from a .dat file. The full exercise given to participants is showcased in Figure 26-29.

Due to time constraints only, a small sample size of 6 participants were able to be gathered. To attempt to mitigate this, the study aimed to collect rich qualitative data, with the understanding that it offered more potential for insights [45]. This was collected through the form of an unstructured interview which was conducted after the participants had used the program. The unstructured interview consisted predominantly of four questions, which the study sought to answer:

- How accessible did you find use of the instruction simulator program?
- If applicable, how would you compare the usability of the instruction set simulator program to other pre-existing solutions you have used?
- Are there any other aspects of the program you would like to share your thoughts about?
- Do you think the program has potential for use in facilitating the teaching of assembly programming, instruction set architecture or computer architecture?

5.4.1 Analysis of Participant Response

To evaluate the participant responses, a reflexive thematic analysis was carried out, as this allowed patterns to be identified across different participant responses [46].

Theme 1: Ease of use - All participants bar one stated that they found the program to be accessible or easy to use. Some of the reasons mentioned for this included the graphical user interface being clearly labelled with buttons being easy to read and understand their purpose and the programs built in user protection, such as not allowing a user to load .dat files into memory until an iteration of SimpleCPU is selected and giving the user a warning if they attempt to open a new file in the integrated development environment without saving the current file.

The participant who did not find the program accessible did not have experience with assembly programming or SYS1, suggesting that pre-existing knowledge on SimpleCPU or assembly programming is necessary to understand the program. The view that background knowledge was required was also expressed by another participant, solidifying this view.

Theme 2: Advantages over Xilinx - Every participant with experience of the Xilinx toolchain, said that they found the instruction set simulator more immediately accessible. Many participants expressed reservations about using Xilinx, although one participant praised Xilinx ISE's waveforms as useful for debugging, they also stated that the Xilinx toolchain came with a high learning curve, one described using Xilinx as a nightmare and another criticized the complexity of Xilinx, and praised the program for having more clarity.

Theme 3: Room for improvement – Participants identified some shortcomings of the program, while most found the UI clear, some described it as ugly or basic and suggested ways to restructure it. Some more suggestions for what could be improved with the graphical user interface included making it clearer when a program had stopped running. Additionally, some criticized the exercise given during the for being somewhat unclear.

Theme 4: Suitability for teaching assembly programming – While one participant criticized the program for being confusing, most participants praised the programs suitability for teaching assembly programming, and one participant praised the interface for being easy to use, due to the included assembler. However most acknowledged that background information is necessary with one participant suggesting a help menu or tutorial.

5.4.2 Conclusion of User Review

While the reliability of the analysis is somewhat limited by the small sample size, it is clear that all participants with relevant background knowledge found the program easy to use. Thus, it is possible that the program met user requirement 3.3 but further analysis is needed to conclusively state this. Additionally, it is reasonable to surmise that the program is insufficient as a resource for teaching independently and should be supplemented with other resources on computer architecture and assembly programming to best support undergraduate education.

To summarise the evaluation section, Figure 32 has been included in the appendix, displaying which user requirements were met.

6 Conclusion

The primary aim of this project was to produce an instruction set simulator that could accurately replicate the behaviour of v1a and 1vd of SimpleCPU. This was motivated by an appreciation for the importance of understanding computer architecture and a desire to streamline the process of simulating assembly programming on these architectures. The final program produced was clearly able to meet this aim successfully.

The second aim of this project was to extend the functionality of the instruction set simulator beyond what was possible before the project was undertaken. The program was able to meet this aim mostly successfully through the addition of additional instructions, instruction remapping, peripheral support with a GPIO device implementation and in integrated development environment. However, unfortunately the program was unable to meet every criterion that was identified as part of the user requirements.

The third aim of this project was to make an instruction set simulator that was more accessible than the existing implementation of SimpleCPU. Although, a significant enough sample size was not gathered in time to fully reach a conclusion on whether or not this aim was met, the results of the thematic analysis conducted seem to indicate that this was the case.

A significant effort in this project was the attempts to optimise the performance of the instruction set simulator, so that it was able to outperform the 3,333,333.33 instructions performed per second by the FPGA implementation of SimpleCPU. In the end, this was achieved by switching Python implementation, from the CPython interpreted implementation, to the PyPy just-in-time compiler implementation. This had some drawbacks in that it required PyPy to be bundled with the program to ensure that users were able to simply run the program with good performance without needing to go through a potentially complicated install process, however it easily outperformed utilising an FPGA when running benchmarks.

6.1 Further Work

Due to time restrictions, no form of display peripheral was implemented into the program. This is unfortunate as implementing a display peripheral would extend the usability of the instruction set simulator by offering more possibilities for what types of programs can be written to run on the simulator, for example, the GPIO and a display could be used to create simple video game programs.

Additionally, during the development of this program, Mike Freeman began working on a new iteration of SimpleCPU, SimpleCPU_v1e, which changed how the Call-Ret stack was implemented and featured additional registers [47]. It would be a worthwhile endeavour to produce a new processor class that followed this updated architecture so that it could be integrated into the instruction set simulator.

Another potential focus for further work could be a new segment of the graphical user interface that allows for auto generation of new instructions in the custom instruction set, by using techniques similar to those used in the epsilon eclipse stack [40] or those employed through the ‘selfModifying.py’ program in an early iteration of the program. This would allow self-defined instructions to work easily without needing to significantly overhaul how the current program functions.

Appendix

```
move,(register,operand),0000RR00KKKKKKKK,immediate
add,(register,operand),0001RR00KKKKKKKK,immediate
sub,(register,operand),0010RR00KKKKKKKK,immediate
and,(register,operand),0011RR00KKKKKKKK,immediate
load,(register,operand),0100KKKKKKKKKKKK,absolute
store,(register,operand),0101KKKKKKKKKKKK,absolute
addm,(register,operand),0110KKKKKKKKKKKK,absolute
subm,(register,operand),0111KKKKKKKKKKKK,absolute
jumpu,(operand),1000KKKKKKKKKKKK,direct
jumpz,(operand),1001KKKKKKKKKKKK,direct
jumpnz,(operand),1010KKKKKKKKKKKK,direct
jumpc,(operand),1011KKKKKKKKKKKK,direct
call,(operand),1100KKKKKKKKKKKK,direct
or,(register,operand),1101RR00KKKKKKKK,immediate
xop1,(),1110000000000000,immediate
ret,(),1111000000000000,direct
move,(register,register),1111RRRR00000001,register
load,(register,register),1111RRRR00000010,register_indirect
store,(register,register),1111RRRR00000011,register_indirect
rol,(register),1111RR0000000100,register
ror,(register),1111RR0000000101,register
add,(register,register),1111RRRR00000110,register
sub,(register,register),1111RRRR00000111,register
and,(register,register),1111RRRR00001000,register
or,(register,register),1111RRRR00001001,register
xor,(register,register),1111RRRR00001010,register
asl,(register),1111RR0000001011,register
xop2,(),1111000000001100,register_indirect
xop3,(),1111000000001101,register
xop4,(),1111000000001110,register_indirect
xop5,(),1111000000001111,register
```

Figure 13 SimpleCPU_v1d in '.is' format

```
start:
    move ra 0xff
    store ra 0xfff
    store ra 0xffe
    store ra 0xffd
    store ra 0xffc
    load ra 0xfff
    sub ra 0x01
    store ra 0xfff
    jumpnz 0x05
    load ra 0xffe
    sub ra 0x01
    store ra 0xffe
    jumpnz 0x05
    load ra 0xffd
    sub ra 0x01
    store ra 0xffd
    jumpnz 0x05
    load ra 0xffc
    sub ra 0x01
    store ra 0xffc
    jumpnz 0x05
```

Figure 14 silly-long-program.asm

Instruction	Addressing mode	Description
Mul	Immediate	Performs a multiplication on a specified register by a specified operand
Div	Immediate	Performs a divide on a specified register by a specified operand
Mod	Immediate	Performs a modulus operation on a specified register by a specified operand
Xor	Immediate	Immediate version of Xor Instruction
Rol	Immediate	Version of Rol instruction that shifts bits specified by an operand
Ror	Immediate	Version of Ror instruction that shifts bits specified by an operand
Asl	Immediate	Version of Asl instruction that shifts bits specified by an operand
Nop	Immediate	Performs no operation
Nop	Absolute	Performs no operation
Nop	Direct	Performs no operation
Nop	Register	Performs no operation
Nop	Register Indirect	Performs no operation
Sec	Direct	Sets the carry flag to a specified operand
Sez	Direct	Sets the zero flag to a specified operand
Pop	Direct	Pops the Program Counter LIFO Stack without returning
RetNp	Direct	Returns without performing a pop on the Program Counter LIFO stack
Mul	Register	Performs a multiplication on a specified register by a specified register
Div	Register	Performs a division on a specified register by a specified register
Mod	Register	Performs a modulus on a specified register by a specified register
Rol8	Register	Performs a Rol on a specified register, but shifts the bits 8 spaces
Asl8	Register	Performs a Ror on a specified register, but shifts the bits 8 spaces
ClrLifo	Direct	Clears values in the Program Counters LIFO stack

Figure 15 CustomProcessor additional instructions

```
start:
 move 0xff
 store 0xff
 store 0xfe
 store 0xfd
part1:
 move 0x00
 add 0x06
 sub 0x04
 and 0x00
 addm 0xf0
 subm 0xf0
 jumpu part2
part2:
 load 0xff
 sub 0x01
 store 0xff
 jumpnz part1
 move 0xff
 store 0xff
 load 0xfe
 sub 0x01
 store 0xfe
 jumpnz part1
 move 0xff
 store 0xfe
 load 0xfd
 sub 0x01
 store 0xfd
 jumpnz part1
 jumpz 0x20
 jumpu 0x1c
 .data 0xfe
 .data 0xff
 .data 0xfe
 jumpu 0x20
```

Figure 16 v1a_benchmark.asm

```
start:
 move ra 0xff
 and ra 0xff
 store ra 0xff
 store ra 0xfe
 store ra 0xfd
part1:
 move ra 0x00
 add ra 0x06
 sub ra 0x04
 and ra 0x00
 addm ra 0xf0
 subm ra 0xf0
 call subRoutine
 jumpu part2
part2:
 move ra 0x00
 add ra 0x05
 jumpc wrongHalt
 load ra 0xff
 sub ra 0x01
 store ra 0xff
 jumpnz part1
 move ra 0xff
 and ra 0xff
 store ra 0xff
 load ra 0xfe
 sub ra 0x01
 store ra 0xfe
 jumpnz part1
 move ra 0xff
 and ra 0xff
 store ra 0xfe
 load ra 0xfd
 sub ra 0x01
 store ra 0xfd
 jumpnz part1
 jumpz halt
wrongHalt:
 .data 0xfe
 .data 0xff
 .data 0xfe
 jumpu wrongHalt
halt:
 jumpu halt
subRoutine:
 move rb 0x7f
 or rb 0x7e
 rol rb
 add rb 0x01
 move rb 0x59
 move rc 0x5a
 store rb rc
 load rc rb
 add rc rb
 sub rc rb
 ror rc
 ret
```

Figure 17 v1d_benchmark.asm

Simulator	*Time Taken to execute v1a_benchmark in seconds (s)*				
	1st run	*2nd run*	*3rd run*	*4th run*	*Average*
CPython CLI	241.594	247.568	239.844	241.885	242.723
PyPy CLI	6.623	6.688	6.609	6.687	6.652

Figure 18 Time taken to execute v1a_benchmark using the instruction set simulator

$$\text{Instructions to execute} = 182786810$$

$$\text{FPGA Board Cycles per second} = 10000000$$

$$\text{SimpleCPU Cycles per Instruction} = 3$$

$$\text{Instructions per Second} = \frac{\text{FPGA Board Cycles per second}}{\text{SimpleCPU Cycles per Instruction}} = \frac{10000000}{3}$$

$$\text{Time taken to execute (s)} = \frac{\text{Instructions to execute}}{\text{Instructions per Second}} = \frac{182786810}{\frac{10000000}{3}} = 54.836s$$

Figure 19 Calculation for time taken to execute v1a_benchmark using an FPGA

$$\text{Time taken (s) to simulate 50ms} = 10.33s$$

$$\text{Time taken (s) to execute v1a benchmark on FPGA} = 54.836s$$

$$\text{Time taken (s) to simulate full program} = \frac{\text{Total time taken to execute}}{\text{Time Simulated}} \times \text{Total time taken to simulate}$$

$$\text{Time taken (s) to simulate full program} = \frac{54.836}{0.05} \times 10.33 = 11329.11s$$

Figure 20 Calculation of time taken to simulate v1a_benchmark using Xilinx ISim

Simulator	*Time Taken to execute v1d_benchmark in seconds (s)*				
	1st run	*2nd run*	*3rd run*	*4th run*	*Average*
CPython CLI	685.899	688.686	686.052	691.405	688.01
PyPy CLI	18.357	19.031	18.672	18.682	18.686

Figure 21 Time taken to Simulate v1d_benchmark using the Instruction Set Simulator

$$\text{Instructions to execute} = 448154091$$

$$\text{FPGA Board Cycles per second} = 10000000$$

$$\text{SimpleCPU Cycles per Instruction} = 3$$

$$\text{Instructions per Second} = \frac{\text{FPGA Board Cycles per second}}{\text{SimpleCPU Cycles per Instruction}} = \frac{10000000}{3}$$

$$\text{Time taken to execute (s)} = \frac{\text{Instructions to execute}}{\text{Instructions per Second}} = \frac{448154091}{\frac{10000000}{3}} = 134.446s$$

Figure 22 Calculation for time taken to execute v1d_benchmark using an FPGA

$$\text{Time taken (s) to simulate 50ms} = 29.38s$$

$$\text{Time taken (s) to execute v1d benchmark on FPGA} = 134.446s$$

$$\text{Time taken (s) to simulate full program} = \frac{\text{Total time taken to execute}}{\text{Time Simulated}} \times \text{Total time taken to simulate}$$

$$\text{Time taken (s) to simulate full program} = \frac{134.446}{0.05} \times 29.38 = 79000.4696s$$

Figure 23 Calculation of time taken to simulate v1d_benchmark using Xilinx ISim

```
58
59      def execute(self: object, noOfExecutions=1) -> None:
60          for x in range(noOfExecutions):
61              self.instructionRegister = self.memory[self.programCounter]
62              opcode = self.instructionRegister >> 12
63              self.programCounter = (self.programCounter + 1) & 4095
64              match opcode:
65                  case 0: #0000 [immediate]
66                      self.immediate_move()
67
68                  case 1: #0001 [immediate]
69                      self.immediate_add()
70
71                  case 2: #0010 [immediate]
72                      self.immediate_sub()
73
74                  case 3: #0011 [immediate]
75                      self.immediate_and()
76
77                  case 4: #0100 [absolute]
78                      self.absolute_load()
79
80                  case 5: #0101 [absolute]
81                      self.absolute_store()
82
83                  case 6: #0110 [absolute]
84                      self.absolute_addm()
85
86                  case 7: #0111 [absolute]
87                      self.absolute_subm()
88
89                  case 8: #1000 [direct]
90                      self.direct_jumpU()
91
92                  case 9: #1001 [direct]
93                      self.direct_jumpZ()
94
95                  case 10: #1010 [direct]
96                      self.direct_jumpNZ()
97
98                  case 11: #1011 [direct]
99                      self.direct_jumpC()
100
101                 case 12: #1100 [direct]
102                     self.direct_call()
103
104                 case 13: #1101 [immediate]
105                     self.immediate_or()
106
107                 case 14: #1110 [immediate]
108                     self.immediate_xop1()
109
```

Figure 24 Final execute() function in v1d [1st half]

```
109
110             case 15: #1111
111                 #All instruction with left and right opcode bits
112                 opcode = self.instructionRegister & 15
113                 match opcode:
114                     case 0: #1111 + 0000 [direct]
115                         self.direct_ret()
116 
117                     case 1: #1111 + 0001 [register]
118                         self.register_move()
119 
120                     case 2: #1111 + 0010 [register indirect]
121                         self.register_indirect_load()
122 
123                     case 3: #1111 + 0011 [register indirect]
124                         self.register_indirect_store()
125 
126                     case 4: #1111 + 0100 [register]
127                         self.register_rol()
128 
129                     case 5: #1111 + 0101 [register]
130                         self.register_ror()
131 
132                     case 6: #1111 + 0110 [register]
133                         self.register_add()
134 
135                     case 7: #1111 + 0111 [register]
136                         self.register_sub()
137 
138                     case 8: #1111 + 1000 [register]
139                         self.register_and()
140 
141                     case 9: #1111 + 1001 [register]
142                         self.register_or()
143 
144                     case 10: #1111 + 1010 [register]
145                         self.register_xor()
146 
147                     case 11: #1111 + 1011 [register]
148                         self.register_asl()
149 
150                     case 12: #1111 + 1100 [register indirect]
151                         self.register_indirect_xop2()
152 
153                     case 13: #1111 + 1101 [register]
154                         self.register_xop3()
155 
156                     case 14: #1111 + 1110 [register indirect]
157                         self.register_indirect_xop4()
158 
159                     case 15: #1111 + 1111 [register]
160                         self.register_xop5()
```

Figure 25 Final execute() function in v1d [2nd half]

SimpleCPU Instruction Set Simulator Exercise

April 25, 2024

1 Introduction

For the purposes of aiding the teaching of computer architecture and assembly programming, an instruction set simulator for SimpleCPU has been produced. In order to assess the usability of this program, you have been asked to use this program to perform a simple exercise. Below is the exercise, along with some details on how to use the program.

2 Exercise

The exercise below is taken from question 2 of the Systems 1 example assessment, from the 2021-22 period of assessment.

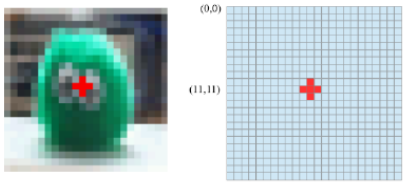


Figure 3 : mark centre subroutine output.

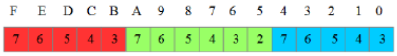


Figure 4 : 16-bit packed RGB data format

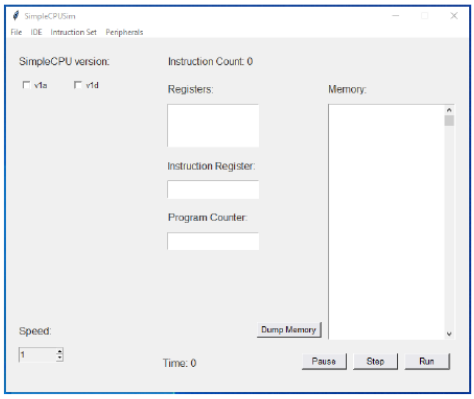
Write a program to place a red cross in the middle of an RGB image stored in memory, as shown in figure 3. The centre of the image is define as co-ordinate position (11,11). This image is stored in memory using the packed RGB data type shown in figure 4. To aid with the completion of this task a .bat file has been produced which stores the image 'bug24x24.ppm' into a .dat file starting from base address 1024, (this .dat file contains the contents of memory including your assembled code that is ran by the program) where the order that pixels are stored in goes horizontally, then vertically, and a .bat file has been produced that loads the image from a .dat file (this .dat file will be your the dumped memory contents after execution of your program). Resources on SimpleCPU, helpful information for the exercise and a solution to the exercise, will be made available in order to assist you in completing this task. In order for loadImage.bat to function, the .dat file produced must be named 'program.dat'

3 How to use the instruction set simulator

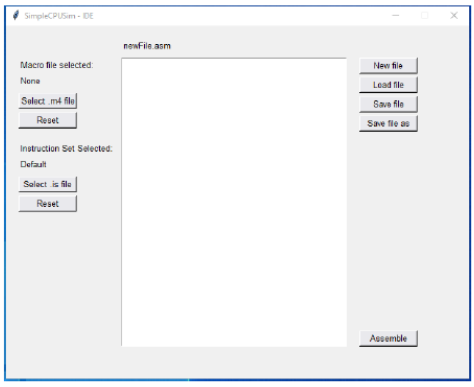
To open the instruction set simulator a .vbs file (WindowsGUI.vbs) has been produced that opens the program using PyPy, a Just-in-Time implementation of Python that results in significant performance boosts when executing programs.

1

Figure 26 SimpleCPU Instruction Set Simulator Exercise page 1



Before a program can be loaded a version of SimpleCPU must first be selected, for the purposes of this exercise you should select version v1d. The program offers an integrated development environment. You should use this to write your program, to open a window for the integrated development environment, select 'IDE' on the menu bar at the top of the program, then select new window.



You should select New file in this window and name your program as 'program.asm' in the same directory as the storeImage and loadImage .bat files. Once you have wrote your program, it can be assembled by pressing the Assemble button. This will produce a file in the same name and location with the .dat file type, which can be read by the program. IMPORTANT: Remember to load the image into the .dat file before executing the program.

To select a program to run, select 'File' in the menu bar, and select open, then navigate to the directory containing the 'program.dat' file, and select it. Following this the program can be run step by step, or executed continuously ran using the run button, to increase the speed of execution of the program the speed box can be incremented.

2

Figure 27 SimpleCPU Instruction Set Simulator Exercise page 2

Once a program has reached a self-referential unconditional jumpu instruction, or 2 blank memory addresses, it will halt. Once it has reached this point, or once you are satisfied with the programs execution, select the dump memory icon to dump the contents of memory. You should save this file as 'output.dat' in the same directory as the 'loadImage.bat' file. Upon saving the file run the 'loadImage.bat' file, a file title 'output.ppm' should have been loaded, look at it and verify your solution. (.ppm files can be loaded by ImageJ a program installed on the lab computers.)

3.1 Exercise summary

- Use the Instruction Set Simulator to write and assemble a program (named program.dat) that meets the requirements of the exercise
- Use the 'StoreImage.bat' file to include the image with the program
- Execute the program using the Instruction Set Simulator
- Use the dump memory feature to write the contents of memory to 'output.dat'
- Use the 'LoadImage.bat' file to write the contents of the new image to 'output.ppm'
- Use ImageJ to verify results

4 Useful Information

- To write the red colour to the image the store instruction can be used, where the contents of a given register are equivalent to the colour red in the given format in figure 4.
- To move the value of the colour red into a register at the correct position the rol instruction can be used after a move instruction (remembering that move instructions are bit padded, so that the upper 8 bits not represented in the operand are always the same as the highest bit of the operand).
- The address of the first memory location where the colour red must be written to is 0x4FB.

Figure 28 SimpleCPU Instruction Set Simulator Exercise page 3

```
start:
 add ra 0x00
 add ra 0x01
 add ra 0x80
 add ra 0xff
 add ra 0x80
 add ra 0xff
 add rb 0x00
 add rb 0x01
 add rb 0xff
 add rc 0xff
 add rc 0x00
 add rc 0x01
 add rc 0x80
 add rc 0xff
 add rc 0xff
 add rd 0x00
 add rd 0xf1
 add rd 0x00
halt:
 jumpu halt
```

Figure 29 v1d_add_test.asm

```
def testRegRor(self):
    for x in range(10000):
        cpu = processor()
        regValue = random16bit()
        cpu.registers[0] = regValue
        cpu.registers[1] = regValue
        cpu.memory[0] = int("1111000000000101",2)
        cpu.execute()
        for y in range(15):
            cpu.programCounter = 0
            cpu.memory[0] = int("1111010000000100",2)
            cpu.execute()
        self.assertEqual(cpu.registers[0], cpu.registers[1])
```

Figure 30 Register Ror Instruction test

```
def testAdd(self):
    cpu = processor()
    cpu.loadMem('Resources/tests/v1dTests/v1d_add_test.dat')
    cpu.execute()
    self.assertEqual(cpu.registers[0], int("0x0000",16))
    cpu.execute()
    self.assertEqual(cpu.registers[0], int("0x0001",16))
    cpu.execute()
    self.assertEqual(cpu.registers[0], int("0xff81",16))
    cpu.execute()
    self.assertEqual(cpu.registers[0], int("0xff80",16))
    cpu.execute()
    self.assertEqual(cpu.registers[0], int("0xff00",16))
    cpu.execute()
    self.assertEqual(cpu.registers[0], int("0feff",16))
    cpu.execute()
    self.assertEqual(cpu.registers[1], int("0x0000",16))
    cpu.execute()
    self.assertEqual(cpu.registers[1], int("0x0001",16))
    cpu.execute()
    self.assertEqual(cpu.registers[1], int("0x0000",16))
    cpu.execute()
    self.assertEqual(cpu.registers[2], int("0xffff",16))
    cpu.execute()
    self.assertEqual(cpu.registers[2], int("0xffff",16))
    cpu.execute()
    self.assertEqual(cpu.registers[2], int("0x0000",16))
    cpu.execute()
    self.assertEqual(cpu.registers[2], int("0xff80",16))
    cpu.execute()
    self.assertEqual(cpu.registers[2], int("0xff7f",16))
    cpu.execute()
    self.assertEqual(cpu.registers[2], int("0xff7e",16))
    cpu.execute()
    self.assertEqual(cpu.registers[3], int("0x0000",16))
    cpu.execute()
    self.assertEqual(cpu.registers[3], int("0xfff1",16))
    cpu.execute()
    self.assertEqual(cpu.registers[3], int("0xfff1",16))
    cpu.execute()
    self.assertEqual(cpu.registers[3], int("0xfff1",16))
    cpu.execute()
    self.assertEqual(cpu.registers[3], int("0xfff1",16))
    cpu.execute()
```

Figure 31 ISS Python Unit test for v1d_add_test

Requirement ID	Description	Requirement met
1.1	The program must be able to accurately simulate on a register level the behaviour of SimpleCPU_v1a for all instructions.	Achieved
1.2	The program must be able to accurately simulate on a register level the behaviour of SimpleCPU_v1d for all implemented instructions.	Achieved
1.3	The program should support an implementation of the instructions defined in the reference guide.	Achieved
2.1	The program should allow the users to customise the instruction set of SimpleCPU_v1d and implement new instructions.	Partially Achieved
2.2	The program should allow the user to assemble programs for SimpleCPU_v1d with an altered instruction set, so that programs can be run.	Achieved
2.3	The program should allow users to dump the contents of memory.	Achieved
2.4	The program should offer support for emulating a GPIO peripheral.	Achieved
2.5	The program should offer support for emulating a display peripheral.	Not met
2.6	The program should offer an Integrated Development Environment for developing assembly programs.	Achieved
2.7	Developing, assembling and running programs should be seamless and frictionless.	Possibly Achieved
3.1	The user must be able to interface with the program through a command line.	Achieved
3.2	The user must be able to interface with the program through a graphical user interface.	Achieved
3.3	The program should be easy to use and understand.	Possibly Achieved
4.1	The program should be able to outperform existing ways of running or simulating SimpleCPU programs.	Achieved (Dependent on Python Implementation)

Figure 32 Summary of User Requirements met

Bibliography

[1] British Computing Society, “Computer Science fastest growing STEM subject,” 24 August 2023. [Online]. Available: https://www.bcs.org/articles-opinion-and-research/computer-science-fastest-growing-stem-subject/. [Accessed 22 January 2024].

[2] British Computing Society, “University Computing departments met with record applicant numbers as AI hits the mainstream,” 13 February 2023. [Online]. Available: https://www.bcs.org/articles-opinion-and-research/university-computing-departments-met-with-record-applicant-numbers-as-ai-hits-the-mainstream/. [Accessed 22 January 2024].

[3] ACM/IEEE Joint Task Force on Computing Curricula, “Computer science curricula 2013: curriculum guidelines for undergraduate degree programs in computer science,” Association for Computing Machinery, New York, United States, 2013.

[4] A. Clements, “The Undergraduate Curriculum in Computer Architecture,” *IEEE Micro,* vol. 20, no. 3, pp. 13 - 21, 2000.

[5] Xilinx , “ISE 14.7 VM for Windows 10: Installation, Licensing and Release Notes for Virtual Machine Installation,” 19 February 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1227-ise-vm-windows10. [Accessed 27 April 2024].

[6] Xilinx, “https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive-ise.html,” 20 February 2020. [Online]. Available: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive-ise.html. [Accessed 27 April 2024].

[7] The Joint Task Force on Computing Curricula, “Computer Science Curricula 2023 Version Gamma,” Association for Computing Machinery, New York, United States, 2023.

[8] A. Clements, A. A. Shvartsman, W. K. King, C. Lu and D. Q. Dupont, “Towards a modern computer architecture curriculum,”

in *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference.*, San Juan, Puerto Rico, 1999.

[9] A. Clements, “Crafting a curriculum in computer architecture,” in *38th Annual Frontiers in Education Conference*, New York, United States, 2008.

[10] A. Clements, “Has computer architecture exceeded its teach-by date?,” in *31st Annual Frontiers in Education Conference*, Reno, Nevada, United States, 2001.

[11] ACM/IEEE Joint Task Force on Computing Curricula, “Computer Science Curriculum 2008: An Interim Revision of CS 2001,” Association for Computing Machinery, New York, United States, 2008.

[12] A. Clements, Computer Organization and Architecture: Theme and Variations, Cengage Learning, 2013.

[13] R. Barnett, “The Purposes of Higher Education and the Changing Face of Academia,” *London Review of Education,* vol. 2, no. 1, pp. 61-73, 2004.

[14] J. McArthur, “Reconsidering the social and economic purposes of higher education,” *Higher Education Research & Development,* vol. 30, no. 6, pp. 737-749, 2011.

[15] T. McCowan, “Should universities promote employability?,” *Theory and Research in Education,* vol. 13, no. 3, pp. 267-285, 2015.

[16] freecodecamp, “Interpreted vs Compiled Programming Languages: What's the Difference?,” freecodecamp, 10 January 2020. [Online]. Available: https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/. [Accessed 27 April 2024].

[17] R. Simha, A. N. Kumar and R. K. Raj, “Undergraduate Computer Science Curricula,” *Communications of the ACM,* vol. 67, no. 2, p. 29–31, 2024.

[18] M. Freeman, “Simple CPU v1a: Reference Guide,” [Online]. Available: http://www.simplecpudesign.com/simple_cpu_v1a_datasheet/index.html. [Accessed 18 April 2024].

[19] M. Freeman, “Simple CPU v1d: Reference Guide,” [Online]. Available: http://www.simplecpudesign.com/simple_cpu_v1d_datasheet/index.html. [Accessed 18 April 2024].

[20] C. Cooper and P. Werstein, “The Use of Java to Develop a Microprocessor Emulator,” in *International Conference on Software Engineering: Education and Practice*, Dunedin, New Zealand, 1998.

[21] D. Patti, A. Spadaccini, M. Palesi, F. Fazzino and V. Catania, “Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator,” *IEEE Transactions on Education,* vol. 55, no. 3, pp. 406 - 411, 2012.

[22] M. Brox, A. Gersnoviez, M. A. Montijano, E. Herruzo and C. D. Moreno, “SICOME 2.0: A teaching simulator for Computer Architecture,” in *Technologies Applied to Electronics Teaching (TAEE)*, La Laguna, Spain, 2018.

[23] J. T. Chia and S. K. G, “Educational Simulator for Analysing Pipelined LEGv8 (subset of ARMv8) Architecture,” in *IEEE Region 10 International Conference TENCON* , Chiang Mai, Thailand , 2023.

[24] P. W. C. Prasad, A. Alsadoon, A. Beg and A. Chan, “Using simulators for teaching computer organization and architecture,” *Computer Applications in Engineering Education,* vol. 24, no. 2, pp. 215 - 224, 2016.

[25] W. Zaatal and G. E. Nasr, “An implementation scheme for a microprocessor emulator,” in *IEEE International Conference on Electronics, Circuits and Systems*, Jounieh, Lebanon, 2000.

[26] D. Skrien, “CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes,” *Journal on Educational Resources in Computing,* vol. 1, no. 4, pp. 46-59, 2001.

[27] G. Savaton, “emulsiV,” [Online]. Available: https://eseo-tech.github.io/emulsiV/. [Accessed 6 February 2024].

[28] G. Savaton, “emulsiV: A visual simulator for teaching computer architecture using the RISC-V instruction set,” 25 June 2020. [Online]. Available:

https://www.youtube.com/watch?v=QqBmVEBhdco. [Accessed 6 February 2024].

[29] A. Waterman, K. Asanovi´c and S. Inc, *The RISC-V Instruction Set Manual Volume 1: User-Level ISA,* California, Berkeley: University of California, 2017.

[30] H. Spaans, “python-6502-emulator,” 1 January 2022. [Online]. Available: https://github.com/hspaans/python-6502-emulator. [Accessed 2 February 2024].

[31] J. Patterson and J. Dixon, “Optimizing Power Consumption in DSP Designs,” Texas Instruments, Texas, 2006.

[32] Python Software Foundation., “tkinter — Python interface to Tcl/Tk,” [Online]. Available: https://docs.python.org/3/library/tkinter.html. [Accessed 2024 April 18].

[33] K. M, “benchmarks,” 13 October 2014. [Online]. Available: https://github.com/kostya/benchmarks. [Accessed 17 April 2024].

[34] N. Heer, “Speed Comparison,” 15 10 2022. [Online]. Available: https://niklas-heer.github.io/speed-comparison/. [Accessed 17 April 2024].

[35] J. Murphy, “TALK / James Murphy / From 3 to 300 fps: NES Emulation in Python and Cython,” PyCon US, 4 June 2021. [Online]. Available: https://www.youtube.com/watch?v=3of9pY2vovA. [Accessed 2024 April 17].

[36] I. Sommerville, “Agile software development,” in *Software Engineering*, Pearson Education, 2016, pp. 72-100.

[37] “Visual TK,” 2019. [Online]. Available: https://visualtk.com/. [Accessed 18 April 2024].

[38] M. Freeman, “Simple CPU v1a: Assembler,” 21 February 2024. [Online]. Available: http://simplecpudesign.com/simple_cpu_v1a_assembler/index.html. [Accessed 19 April 2024].

[39] GnuWin, “m4 for Windows,” GnuWin, 5 June 2010. [Online]. Available: https://gnuwin32.sourceforge.net/packages/m4.htm. [Accessed 19 April 2024].

[40] Eclipse Foundation, “Epsilon documentation: The Epsilon Generation Language (EGL),” [Online]. Available: https://eclipse.dev/epsilon/doc/egl/. [Accessed 2024 April 19].

[41] The PyPy Team, “PyPy - Features,” 16 April 2024. [Online]. Available: https://www.pypy.org/features.html. [Accessed 19 April 2024].

[42] P. S. Foundation, “Alternative Python Implementations,” [Online]. Available: https://www.python.org/download/alternatives/. [Accessed 19 April 2024].

[43] G. Andrews, “Why the Command Line Is Not Usable,” 5 April 2015. [Online]. Available: https://gusandrews.medium.com/why-the-command-line-is-not-usable-583d54dcb8ea. [Accessed 24 April 2024].

[44] A. Feizi and C. Y. Wong, “Usability of user interface styles for learning a graphical software application,” in *International Conference on Computer & Information Science (ICCIS)*, Kuala Lumpur, Malaysia, 2012.

[45] A. Blandford, D. Furniss and S. Makri, “Gathering Data,” in *Qualitative HCI Research: Going Behind the Scenes*, Morgan & Claypool, 2016, pp. 33-49.

[46] V. Braun, V. Clarke, N. Hayfield and G. Terry, *Thematic Analysis. In: Liamputtong, P. (eds) Handbook of Research Methods in Health Social Sciences.,* Singapore: Springer, 2019.

[47] M. Freeman, “Simple CPU v1e: Reference Guide,” [Online]. Available: http://www.simplecpudesign.com/simple_cpu_v1e_datasheet/index.html. [Accessed 25 April 2024].