SIMULIZER

Final Report

Charlie Street Kelsey McKenna Matthew Broadway
Michael Oultram Theo Styles

Thursday 2nd June, 2016

Abstract

In various Computer Science courses, students learn about processor architecture: how they are designed, how they execute instructions and also how you write assembly code to run on them. A common processor architecture to study is MIPS due to its reduced instruction set. While there are software systems that exist to allow the user to write and execute MIPS programs, they often lack useful features, making them unaccommodating to new users.

In this project we aimed to extend the capabilities of these systems, and additionally provide students with a unique, interactive, and visual way to learn the MIPS assembly language. This has been achieved by creating an IDE-like environment along with a range of visualisations to help with the learning process. In addition, Simulizer provides a visual way to describe difficult concepts, such as instruction pipelining, which opens the potential for the system to be used as a teaching resource.

Contents

1 Introduction

This report aims to discuss the inception and creation stages of our project Simulizer; a piece of software with the goal of visualising and simulating a MIPS-style processor. This report will also analyse the team behind the project, how the project was organised and the decisions the team had to make to bring the software to light.

1.1 Task

Our task was to select an algorithm and visualise it somehow. We tried to think outside the box with our choice of algorithm, challenging the brief set in the process. The Instruction Execute Cycle in both non-pipelined and pipelined form was the algorithm we picked as it is crucial for enabling any other algorithm/program to run on a computer. The algorithm is very suitable for visualising as it can be conceptualised through the use of data moving through a block diagram of a processor. Additionally, the MIPS programs written using the simulator will implement algorithms which will be visualised indirectly through the visualisation of the processor, as well as other means. This potentially allows for unlimited scope as any algorithm has the potential to be implemented in MIPS and visualised using our system.

Software packages exist which allow the user to run their own MIPS programs, for example Spim Simulator, but we felt it was lacking in both functionality, and usability. Therefore, we set our sights on developing a system that would improve on these aspects. In addition, comparing our system to Spim gave us a good target for the basic functionality required to be considered usable by our target audience.

Visualisation is also a critical aspect of this project (for specification reasons as well as making a highly usable system), so Simulizer contains multiple forms of visualisation. The Instruction Execute Cycle is being visualised via the movement of data and use of components on a fairly typical block diagram of a MIPS processor. As an example, for an add instruction, one might see data being read from the registers and being passed to the ALU. The computations would then be seen, and the result written back to a specific register. In addition to this, there are visualisations with more specific intentions. Simulizer provides a visualisation of a pipeline (i.e. instructions moving through the pipeline and specific hazards being shown as 'bubbles' in the pipeline). To complement this, the system will provide information equivalent to Spim Simulator with regards to register and memory contents. At a higher level, Simulizer displays the algorithms being run in the MIPS programs via the use of appropriate data structures such as lists, graphs, disks etc. This allows a reinforcement of rudimentary algorithms familiar to to a student student, as well as providing a cross-reference for the Instruction Execution visualisation.

1.2 Motivation

The original motivation for the project arose from a second year Computer Science module 'Computer Systems & Architecture' taught at the University of Birmingham. The module aims to teach students about 'the fundamental concepts and principles of computer architectures' as well as to 'introduce the basic components of computer systems, their internal design and operation and their interactions' (School of Computer Science, 2015). During the early stages of this module (of which 4 out of 5 members of the development team are students), the inner workings of a processor are taught. There is an emphasis on topics such as the components of a

CPU, the Instruction Execution cycle, pipelining and also how to write basic assembly code to replicate some of the algorithms learnt in the first year of study.

During the CPU sections of the module (which focus mainly on the MIPS R4000 processor) students are taught to write assembly code. Because they are unlikely to have a physical MIPS processor to use, it is recommended that they use a simulator named Spim Simulator. Spim is a 'self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for this processor' (Larus, 2011). Although the students can test the functionality of their assembly code using Spim, it doesn't give them much of a visual aspect of what their program is actually doing. Everything displayed in the software is largely text-driven, with the different segments of primary memory and the register contents shown as text displays. This makes the assignment to implement a familiar algorithm in assembly at lot more difficult when Spim is used (previous algorithms assignments have been bubble sort and binary search). The majority of the motivation for Simulizer has come from this point.

The Computer Systems & Architecture module contains a lot of content that is completely new to students who, based on their first year studies, won't have dealt with anything at such a low level before. This lead to the idea of Simulizer: as well as providing a usable interface to running your own MIPS assembly program, it would be beneficial to students if in-depth visualisations of the processor were running asynchronously with the running of their program. This allows for a level of reinforcement of the content taught in lectures, and would even potentially allow a visual aid during lectures.

1.3 Interest

In order to gauge the interest and potential benefit for this system, communications were held with the module leader for Computer Systems & Architecture (Mr Ian Batten). It was gathered that there were certain areas of the module that students either found difficult to understand, or the lecturer found difficult to teach: the most challenging topic being pipelining. From these discussions, the functionality and goals of Simulizer were determined.

1.4 Overview

As a general overview, this report intends to explain in detail the processes and techniques used during the creation of Simulizer and its assembly language (which we call 'SIMP'), as well an explanation of how it was made. This will involve discussing how the back-end of the system was designed. As well as this, a full evaluation of the project will be provided, along with the methods used to ensure good performance as a team. Good HCI techniques were used to design the user interface and the visualisations, and the Software Engineering practices used helped to maximise the quality of the software. Finally, at the end of this report, one can find a summary and reflection from each of the 5 team members on how the project went for them personally. Attached as an appendix to this report are further useful documents, such as textual use cases, persona and further UML class diagrams.

2 A note on external software/languages

Throughout the development of Simulizer, in order to aid in the production of a highly functional system in a limited time, external software and libraries have been utilised in some specific cases. This section of the report will explicitly declare these, where they have been used in Simulizer, the license they released under, as well as a link to the software to be examined.

Ace Editor - a highly customisable text editor:

- **Where it is used**: The Ace editor is the basis for the editor component used in Simulizer. It provides us with many more options for extensibility than the alternatives we considered. Beyond simply rendering coloured characters of the source code and the line numbers, all of the logic was created by us on top of the framework that Ace offers us. This logic includes rules for syntax highlighting (including the nesting of pre-packaged JavaScript highlighting rules inside SIMP comments), code folding, problem analysis & feedback and interactivity with line highlighting during the simulation. (Note: the colour schemes and JavaScript highlighting rules are not our own and are also BSD licensed)
- **License:** The Ace Editor is released under the BSD license.
- **Link:** `https://ace.c9.io`

ANTLR - a parser generator for structured text:

- **Where it is used**: ANTLR is used in the back-end of the system, providing structured input into the assembler. After defining our own grammar, ANTLR then generated a parser which could analyse a user's code and produce a parse tree. The assembler can traverse the parse tree and perform analysis and error checking of the user's program. Using ANTLR allowed us to achieve more useful analysis and error checking than would have been possible due to the limited time for the project. The parser is also fast enough to continuously parse and feed to our error checking code which provides real-time feedback for problems as the user types.
- **License:** ANTLR is released under the BSD license.
- **Link:** `http://www.antlr.org/index.html`

JavaFX - a set of graphics and media packages for Java:

- **Where it is used**: JavaFX provides a GUI framework along with several utilities. It is a modern alternative to the old standard GUI libraries: Swing, with more of a focus on web technologies (including themability using css). Because of the better default look and feel along with the potential for interesting extensions (see JFXtras) we decided to use JavaFX over other considered libraries such as Swing or Qt. This choice paid off when we decided to use the Ace editor, as it is a web based component rendered inside a JavaFX web view, which is one of the main selling points of JavaFX.
- **License:** JavaFX is released under the GPL v2 license
- **Link:** `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm`

JFXtras - a set of high quality controls and add-ons for JavaFX:

- **Where it is used**: Built on top of JavaFX, This library provides (among other things) the capability for drawing 'internal windows' which our application uses extensively to provide maximum UI flexibility.
- **License**: JFXtras is released under the New BSD License
- **Link**: `http://jfxtras.org/`

GSon - a Java serialization library that can convert Java Objects into JSON and back.

- **Where it is used**: GSON is used as an easy way to store and retrieve the way that internal windows are laid out.
- **License**: GSon is released under the Apache 2.0 License
- **Link**: `https://github.com/google/gson`

In addition to these two pieces of external software, for a small part of the project a different programming language has been used. The language used is JavaScript. JavaScript is being used to provide 'annotations' in the text editor. These are written into the comments of the MIPS program, and then are parsed as well. Once parsed, this JavaScript code allows the user to debug their programs by logging register values for example. It additionally allows easy control over the running of the high-level visualisations. Given that it is expected that a large number of second year Computer Science students will have at least basic experience with JavaScript, and the fact that a scripting language is a suitable language for this task, JavaScript seemed to be an appropriate choice. As a rough approximate, JavaScript contributes about 1.5% to the entire code base of Simulizer.

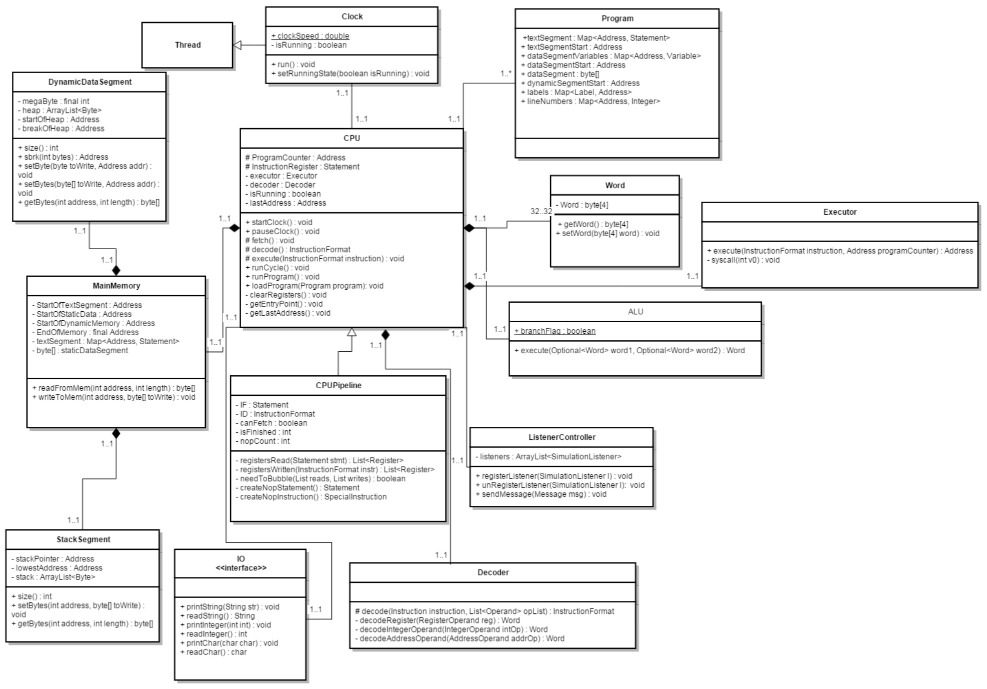
3 Software Design

3.1 Simulation Design

A carefully planned and logical design was crucial to ensure the implementation of Simulizer would be manageable. This careful consideration reduced further complexities later in the project. The design of the software, especially the back-end, went through multiple revisions before a final decision was settled upon. In particular, there were initially multiple ideas for the design of the simulation. These were both focused on different aspects of the implementation. One of the proposed designs was to fully and faithfully recreate the MIPS block diagram within Java, with Bus classes used for communication between components and separate classes for every single component (and additionally a thread for each stage of the pipeline). The alternative design was to reduce the entire simulation state into essentially one class, and then have a separate class for the visualisation state. As a first attempt, a compromise was made that kept the highly faithful design but reduced the threading in the pipeline.

A prototype was implemented based on this design in order to test if it was suitable at a lower level. It soon became apparent that this design would not be fully suitable for multiple reasons: Firstly, the method of communication through buses added a lot of unwanted complexity to the simulation. There was also a level of unnecessary encapsulation made in some of the classes (in particular the program counter, which was just a data class, a bad code 'smell'). This encapsulation would have made the testing of the system incredibly complicated when it needn't be. Also, it was the design stage that discovered that the Observer/Observable model would not be sufficient to connect the model and views; more information would be needed. Based on this, the design was revised further.

The revised design can be viewed below. Although this design still aims for a faithful implementation of the processor, in some cases smaller components, such as the program counter, are not in their own class. Instead, they are a field in the main CPU class. This design kept the most complex components separate, but gave a central class to work around. This also allowed the easy sending of messages through this central class and also made the components much easier to unit test, since each major component could be tested in isolation. One could see these revisions as a refactoring for testability. Other features of this design include a much simpler method for changing the operation of the CPU through inheritance and a significantly reduced thread usage, which reduces the complexity of the simulation significantly. The following is the UML class diagram for the simulation design:


Thread
Clock
+ clockSpeed : double
- isRunning : boolean
+ run() : void
+ setRunningState(boolean isRunning) : void
Program
+textSegment : Map<Address, Statement>
+ textSegmentStart : Address
+ dataSegmentVariables : Map<Address, Variable>
+ dataSegmentStart : Address
+ dataSegment : byte[]
+ dynamicSegmentStart : Address
+ labels : Map<Label, Address>
+ lineNumbers : Map<Address, Integer>
DynamicDataSegment
- megaByte : final int
- heap : ArrayList<Byte>
- startOfHeap : Address
- breakOfHeap : Address
+ size() : int
+ sbrk(int bytes) : Address
+ setByte(byte toWrite, Address addr) : void
+ setBytes(byte[] toWrite, Address addr) : void
+ getBytes(int address, int length) : byte[]
CPU
# ProgramCounter : Address
# InstructionRegister : Statement
- executor : Executor
- decoder : Decoder
- isRunning : boolean
- lastAddress : Address
+ startClock() : void
+ pauseClock() : void
# fetch() : void
# decode() : InstructionFormat
# execute(InstructionFormat instruction) : void
+ runCycle() : void
+ runProgram() : void
+ loadProgram(Program program): void
- clearRegisters() : void
- getEntryPoint() : void
- getLastAddress() : void
Word
- Word : byte[4]
+ getWord() : byte[4]
+ setWord(byte[4] word) : void
Executor
+ execute(InstructionFormat instruction, Address programCounter) : Address
- syscall(int v0) : void
ALU
+ branchFlag : boolean
+ execute(Optional<Word> word1, Optional<Word> word2) : Word
MainMemory
- StartOfTextSegment : Address
- StartOfStaticData : Address
- StartOfDynamicMemory : Address
- EndOfMemory : final Address
- textSegment : Map<Address, Statement>
- byte[] : staticDataSegment
+ readFromMem(int address, int length) : byte[]
+ writeToMem(int address, byte[] toWrite) : void
CPUPipeline
- IF : Statement
- ID : InstructionFormat
- canFetch : boolean
- isFinished : int
- nopCount : int
- registersRead(Statement stmt) : List<Register>
- registersWritten(InstructionFormat instr) : List<Register>
- needToBubble(List reads, List writes) : boolean
- createNopStatement() : Statement
- createNopInstruction() : SpecialInstruction
ListenerController
- listeners : ArrayList<SimulationListener>
+ registerListener(SimulationListener l) : void
+ unRegisterListener(SimulationListener l): void
+ sendMessage(Message msg) : void
StackSegment
- stackPointer : Address
- lowestAddress : Address
- stack : ArrayList<Byte>
+ size() : int
+ setBytes(int address, byte[] toWrite) : void
+ getBytes(int address, int length) : byte[]
IO
<<interface>>
+ printString(String str) : void
+ readString() : String
+ printInteger(int int) : void
+ readInteger() : int
+ printChar(char char) : void
+ readChar() : char
Decoder
# decode(Instruction instruction, List<Operand> opList) : InstructionFormat
- decodeRegister(RegisterOperand reg) : Word
- decodeIntegerOperand(IntegerOperand intOp) : Word
- decodeAddressOperand(AddressOperand addrOp) : Word
1..1
1..*
32..32

In this design, the CPU class is the most important class, providing the central 'hub' for all other classes in the simulation. With respect to the actual CPU, this class abstractly represents the Control Unit, the Program Counter and the Instruction Register as well as the 32 general purpose registers. At a high level, this class also deals with the Instruction Execution Cycle. The fetch, decode and execute methods carry out each of the three main stages of the cycle respectively and the `runSingleCycle` and `runProgram` methods will go about running the given program, in accordance to the clock.

The clock class is crucial to keep all of the system components, visual or non-visual, in time. The clock determines how long one round of the Instruction Execution cycle takes, and so ultimately decides how long a program will take to run. Consequently, the speed of the animations is also determined by the clock. The clock, understandably, is itself a Thread, so it can keep 'ticking' asynchronously.

The `ALU` class maps fairly closely to a real ALU in a MIPS CPU; it carries out the operations on words of data, such as arithmetic operations or branch comparisons. This class needs only a single method, which takes the two (or one) words of data, and carries out an operation on those words. The other significant CPU component not included in the main `CPU` class is the main memory. For its design, the memory had to be considered in 'segments', and then an appropriate data structure was chosen for each. The `MainMemory` class has four different segments, each a field in the class. The first is the text segment. This represents the code of the program and is easiest represented as a Map. This allows swift retrieval of instructions at spread out addresses (if a jump instruction is executed for example), and so a map is appropriate as it provides functionality close to that of a hash table. The second 'segment' is the static data segment. This is being represented as an array of bytes. The reason for this is that the size of this segment is fixed, and there is also a requirement for flexible access (i.e. one byte or groups of 4 indexes not divisible by 4 for example). Therefore, a standard Java array seems most suitable for this application.

The final two 'segments' for the `MainMemory` class are the Stack and the Heap, both of which grow dynamically, and are byte addressable, therefore Java `ArrayList` seemed a reasonable data structure to use, providing the growth element, and byte addressing if Byte is the type of the `ArrayList`. The methods in each of these classes are essentially just intended as read/write methods, with the exception of `sbrk` in the `DynamicDataSegment` class which at a high level is used for memory allocation.

The `Program` class is another crucial class. It essentially provides all of the information needed by the simulation to run a program, including the new memory layout, and boundary addresses. This information is sent to the simulation from the assembler. It enables the easy switching of programs, giving CPU objects a level of re-usability.

The Decoder and Executor class are reasonably self-explanatory. They contain methods to deal with decoding and executing respectively. They were originally intended to be directly in the CPU class, but extensive refactoring brought them out. The `MessageManager` is the class that deals with the sending of messages between the simulation and views; it can register a visual component to listen to the CPU and then messages will begin to be sent to that component to process. The IO interface is used for dealing with I/O between the user and the simulation. By using an interface, it allows the creation of different forms of IO for different purposes (an additional use for this was within unit tests).

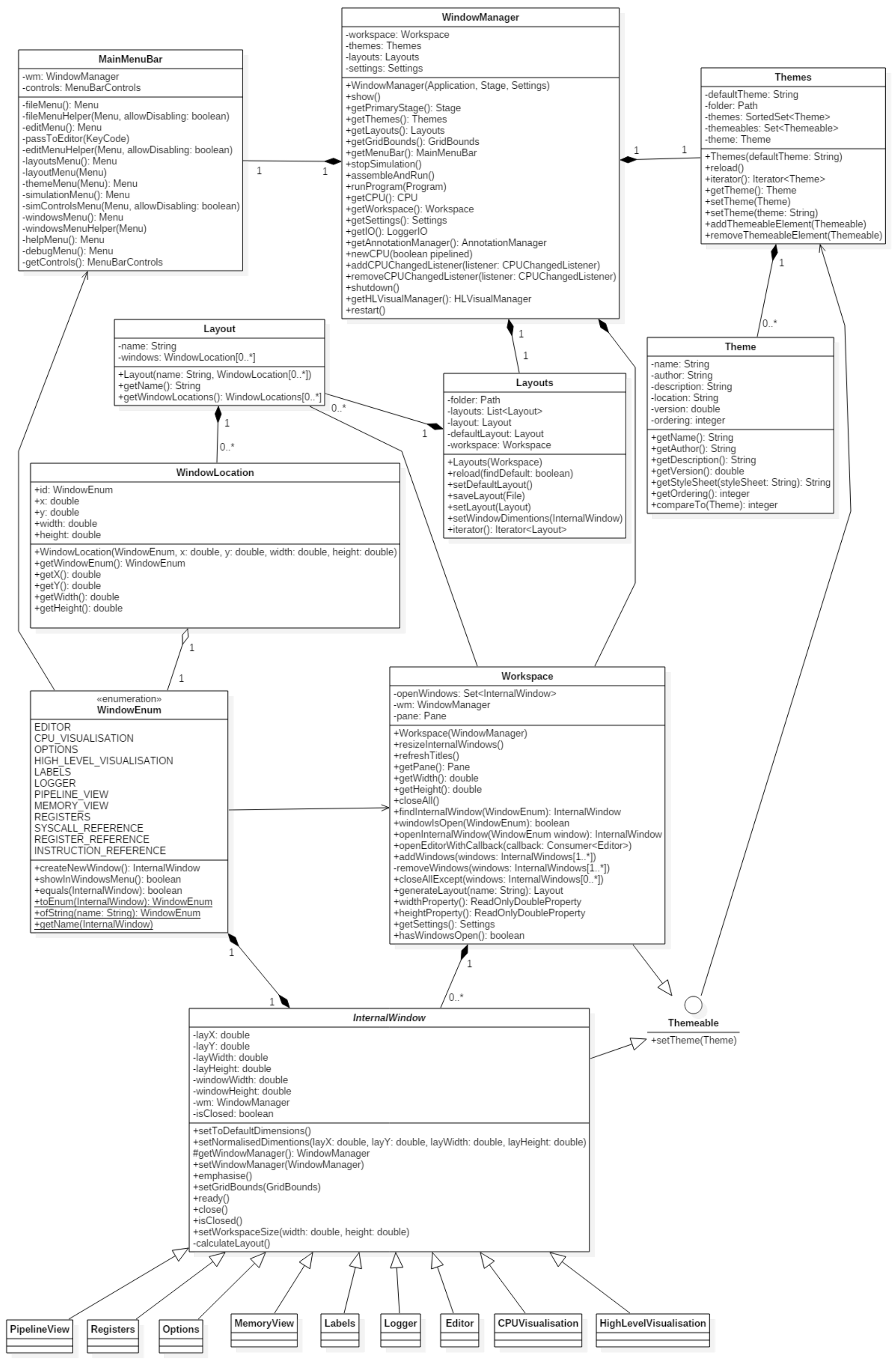
The last main component in the Simulation is the CPU pipeline class. This class is intended to provide the same functionality as the `CPU` class so they can be used interchangeably (hence the use of inheritance). However, the `CPUPipeline` contains the major difference that the `runSingleCycle` is changed such that the instruction execution is now carried out by way of a 3 stage pipeline, a feature that is particularly desired by Ian Batten.

In addition to the classes used by the simulation as shown above, there are also two groups of classes which have proven very useful, but are subtler in use. One of these groups is a set of simulation-oriented exception classes. These include `HeapException`, `DecodeException`, `StackException`, `MemoryException`, `InstructionException` and `ExecuteException`. These classes are used to report errors related to their namesake, and when caught, can be used to display error messages to the user. This is a particularly helpful feature if the written code contains logical errors, rather than syntax errors.

The second group of classes are used for representing instruction formats, such as load instructions, instructions to read/write from registers, jumping instructions, instructions using a base/offset for addressing, and special instructions such as `syscall`. These all have a common base class and are used primarily by the decoder and executor to determine what should be done with a given instruction.

3.2 User Interface Design

The second important section of the design is that of the User Interface structure, the part of the system that was eventually linked to the simulation. Fairly early on, it was unanimously decided that the interface for the system would take a highly modular approach, with windows that can be switched/modified/resized at the user's own wish. As well as providing a high level of usability (the user can personalise the system to whatever suits their needs best), it also allows a more straightforward design at the class level. The following is the class diagram for the UI structure:


WindowManager
-workspace: Workspace
-themes: Themes
-layouts: Layouts
-settings: Settings
+WindowManager(Application, Stage, Settings)
+show()
+getPrimaryStage(): Stage
+getThemes(): Themes
+getLayouts(): Layouts
+getGridBounds(): GridBounds
+getMenuBar(): MainMenuBar
+stopSimulation()
+assembleAndRun()
+runProgram(Program)
+getCPU(): CPU
+getWorkspace(): Workspace
+getSettings(): Settings
+getIO(): LoggerIO
+getAnnotationManager(): AnnotationManager
+newCPU(boolean pipelined)
+addCPUChangedListener(listener: CPUChangedListener)
+removeCPUChangedListener(listener: CPUChangedListener)
+shutdown()
+getHLVisualManager(): HLVisualManager
+restart()
MainMenuBar
-wm: WindowManager
-controls: MenuBarControls
-fileMenu(): Menu
-fileMenuHelper(Menu, allowDisabling: boolean)
-editMenu(): Menu
-passToEditor(KeyCode)
-editMenuHelper(Menu, allowDisabling: boolean)
-layoutsMenu(): Menu
-layoutMenu(Menu)
-themeMenu(Menu): Menu
-simulationMenu(): Menu
-simControlsMenu(Menu, allowDisabling: boolean)
-windowsMenu(): Menu
-windowsMenuHelper(Menu)
-helpMenu(): Menu
-debugMenu(): Menu
-getControls(): MenuBarControls
Themes
-defaultTheme: String
-folder: Path
-themes: SortedSet<Theme>
-themeables: Set<Themeable>
-theme: Theme
+Themes(defaultTheme: String)
+reload()
+iterator(): Iterator<Theme>
+getTheme(): Theme
+setTheme(Theme)
+setTheme(theme: String)
+addThemeableElement(Themeable)
+removeThemeableElement(Themeable)
Theme
-name: String
-author: String
-description: String
-location: String
-version: double
-ordering: integer
+getName(): String
+getAuthor(): String
+getDescription(): String
+getVersion(): double
+getStyleSheet(styleSheet: String): String
+getOrdering(): integer
+compareTo(Theme): integer
Layout
-name: String
-windows: WindowLocation[0..*]
+Layout(name: String, WindowLocation[0..*])
+getName(): String
+getWindowLocations(): WindowLocations[0..*]
Layouts
-folder: Path
-layouts: List<Layout>
-layout: Layout
-defaultLayout: Layout
-workspace: Workspace
+Layouts(Workspace)
+reload(findDefault: boolean)
+setDefaultLayout()
+saveLayout(File)
+setLayout(Layout)
+setWindowDimentions(InternalWindow)
+iterator(): Iterator<Layout>
WindowLocation
+id: WindowEnum
+x: double
+y: double
+width: double
+height: double
+WindowLocation(WindowEnum, x: double, y: double, width: double, height: double)
+getWindowEnum(): WindowEnum
+getX(): double
+getY(): double
+getWidth(): double
+getHeight(): double
«enumeration»
WindowEnum
EDITOR
CPU_VISUALISATION
OPTIONS
HIGH_LEVEL_VISUALISATION
LABELS
LOGGER
PIPELINE_VIEW
MEMORY_VIEW
REGISTERS
SYSCALL_REFERENCE
REGISTER_REFERENCE
INSTRUCTION_REFERENCE
+createNewWindow(): InternalWindow
+showInWindowsMenu(): boolean
+equals(InternalWindow): boolean
+toEnum(InternalWindow): WindowEnum
+ofString(name: String): WindowEnum
+getName(InternalWindow)
Workspace
-openWindows: Set<InternalWindow>
-wm: WindowManager
-pane: Pane
+Workspace(WindowManager)
+resizeInternalWindows()
+refreshTitles()
+getPane(): Pane
+getWidth(): double
+getHeight(): double
+closeAll()
+findInternalWindow(WindowEnum): InternalWindow
+windowIsOpen(WindowEnum): boolean
+openInternalWindow(WindowEnum window): InternalWindow
+openEditorWithCallback(callback: Consumer<Editor>)
+addWindows(windows: InternalWindows[1..*])
-removeWindows(windows: InternalWindows[1..*])
+closeAllExcept(windows: InternalWindows[0..*])
+generateLayout(name: String): Layout
+widthProperty(): ReadOnlyDoubleProperty
+heightProperty(): ReadOnlyDoubleProperty
+getSettings(): Settings
+hasWindowsOpen(): boolean
Themeable
+setTheme(Theme)
InternalWindow
-layX: double
-layY: double
-layWidth: double
-layHeight: double
-windowWidth: double
-windowHeight: double
-wm: WindowManager
-isClosed: boolean
+setToDefaultDimensions()
+setNormalisedDimentions(layX: double, layY: double, layWidth: double, layHeight: double)
#getWindowManager(): WindowManager
+setWindowManager(WindowManager)
+emphasise()
+setGridBounds(GridBounds)
+ready()
+close()
+isClosed()
+setWorkspaceSize(width: double, height: double)
-calculateLayout()
PipelineView
Registers
Options
MemoryView
Labels
Logger
Editor
CPUVisualisation
HighLevelVisualisation
1
0..*

The structure of the UI (due to the decision made on modularity) makes this design very hierarchical. The `Workspace` class manages each separate window, which may or may not be open at any given time. The `WindowManager` class contains the `Workspace` as well as the `MainMenuBar`. `WindowManager` is the class that deals with the CPU object currently being used (in the respect that if the user changes the CPU execution type, a new object will have to be created, additionally it may register components to listen to the CPU).

The window interface is used to describe a general template of what any component displaying on screen needs to implement. Any of the ‘modules’ displayed on screen is therefore a Window.

Some of the Windows displayed include a window for the logger. The logger is the 'gateway' for any input and output between the simulator (as an example, the logger would be used if the user writes a program that calls `syscall` with code 5 to read an integer value into the program). This logger will additionally implement the IO interface discussed previously such that it is compatible with the simulator. The code editor is the window that is the location for the user to write their programs into (and save/load etc.). This window uses the Ace editor to provide an attractive editor (please see 'A note on external software'). Any other window on screen is additionally a visualisation of some sort and so also has to implement the visualisation interface.

The visualisations part of this diagram are all components on the screen that listen to the CPU such that it can, in some way, provide some educational insight into the running of the processor. The `MemoryView` class will provide an insight into the current states of the data segment, the stack and the heap. The `Registers` class will simply look at the contents of all 32 of the general purpose registers (a useful debugging tool alongside the memory view). The `PipelineView` will display the instructions running through the pipeline, with 'bubbles' running through the pipeline on the occurrence of pipeline hazards. The `CPUVisualisation` is the class which visualises our central visualisation in the application: the MIPS block diagram with the data moving through it and highlighting on the use of specific components such as the ALU.

In addition to the aforementioned visualisations, there are also the high-level visualisations, which are the classes providing a more familiar view of the running of the algorithm the users will be used to. There will be multiple different types of high-level visualisation within the system such as list visualisation, towers of Hanoi visualisation etc. These will be discussed further in the 'Visualisation Design' section of this report.

Another crucial component of the simulation is how it communicates to many different components at once. As already discussed, the Observer/Observable pattern was not sufficient for this scenario, and so we sought an alternative approach. The simulation is linked to a `MessageManager` that handles the dispatching and processing of messages sent from the simulation. Components that wish to listen to specific messages register with the `MessageManager`.

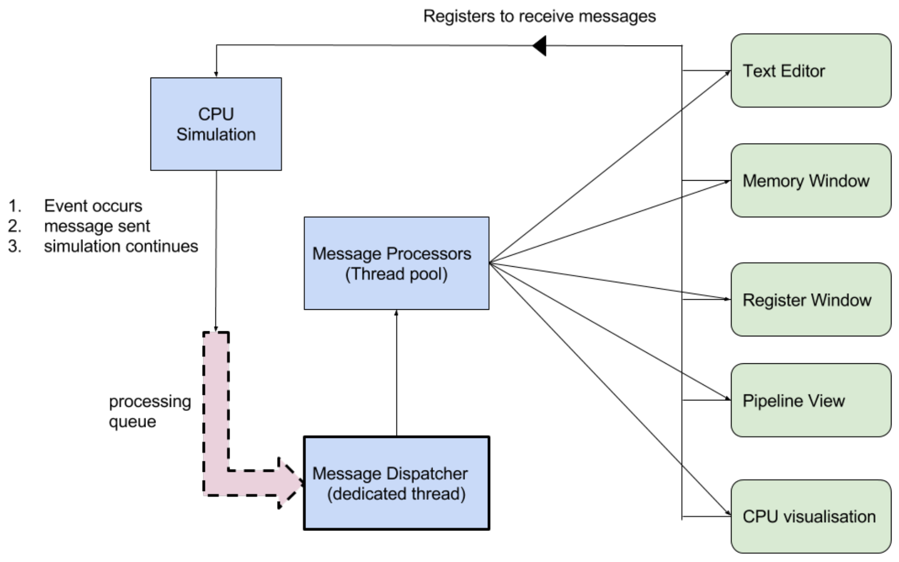
Simulizer uses the producer-consumer pattern to send a large number of messages very quickly and process them concurrently. Whenever an event occurs within the simulation (the producer), it creates a new message object and pushes it onto a processing queue managed by the `MessageManager`. The manager manages a thread pool, which contains a dedicated 'dispatching' thread. This thread is the consumer of the processing queue and repeatedly pops messages off of the queue and dispatches the work of processing the message in each of the listeners to other threads in the pool. The manager guarantees that the simulation will not move on to the next cycle before processing of the messages is complete but does not guarantee the order in which messages are processed.

Variations on the approach were trialled, such as each listener processing in a separate thread in the pool, or having multiple dispatching threads. Ultimately these methods required more synchronisation which drastically decreased performance, so these prototypes were abandoned.

Each listener of the CPU extends the `SimulationListener` abstract base class and only implements the methods corresponding to the type of message they are interested in.

The messages contain useful information about the type of event. This is used to great effect in the annotation system, which passes the javascript code in the message to the annotation processor which executes it. We tried to keep the information provided as general as possible to make the system extremely extensible. Furthermore, information that could easily be looked up using the simulation object was not included for performance reasons.

The data movement message is a good example of our generality. It signals that a piece of data has moved and gives the data. It doesn't specify the CPU component which is involved in the transfer (main memory / register file etc) as the simulation has a different concept of these components to the actual hardware. The hardware-faithful behaviour can be inferred in the code for a visualisation that wants to represent that information. The idea behind the system is to perform no additional work to cater to specific listeners. The listeners decide what of the available information they need, and compute the information based on that. This again helps with the extensibility as the simulation has high cohesion; it performs the simulation and little else.



4 Visualisation Design

From a usability standpoint, there are certain unwritten HCI rules that makes software feel familiar to a new user. We wanted the UI to be highly flexible, and so we had a look at how this could be achieved. One piece of software that has a very flexible UI is the desktop environment, and users of Simulizer should be very familiar with how that works. We decided to mimic a "desktop like" feel by putting each component of the system in its own "Internal Window". Like a desktop environment, these Internal Windows can be moved, resized and closed based on the user's preference. When the main Simulizer window resizes, all of the Internal windows resize to the new dimensions. This means if the resolution of the screen changes (maybe a projector is connected), then all the Internal Windows resize accordingly, keeping the window layout consistent.

Like a desktop environment, we wanted Simulizer to be customisable. There are many use cases on why customisation is a requirement. Take the example of a colour blind user; a more high contrast theme would be required, and so by adding this functionality we have increased the accessibility of the project. The customisation of this software has been designed to come in many forms. Firstly, at a fine grain level of detail, the user is able to configure screen sizes and initial simulation settings, for example, but also fonts and editor themes etc. At the level of the interface, the user is able to create their own layouts for the software, allowing the user to be able to return to the layout that works best for them. Finally, Simulizer supports the changing of the aforementioned themes, providing an even higher level of customisation: the user can change the system as much or as little as they wish.

The development of a user interface in such a manner has given strong HCI attributes to the project. The use of internal windows gives a sense of familiarity to the interface. Because of the knowledge that every component in the system can be opened in the same way in its own internal window, the system is consistent, suggesting that once the user has used the system for a brief time, they will have fully acquired knowledge to utilise the entire system. The consistency of the interface is additionally reinforced via the ability to switch themes/layouts. With a lot of information available, Simulizer has aimed itself on providing clarity throughout its use. This includes helpful error messages should something erroneous occur during the user's time with the system; this is particularly apparent with error messages provided from the simulation to the interface as this in particular was a requirement on improving upon the existing software. Simplicity has also lead to clarity. For example, where possible colour is used instead of text to explain an idea. Further detail is available through the interface but only if the user opts to see it. One final desirable HCI attribute is attractiveness. We believe that Simulizer provides an attractive interface: clean separation of components, and good use of colour has partly achieved this. Further to this, JavaFX has been used for the main interface. We believe that JavaFX is a much superior set of libraries for designing interfaces over its competitor, Swing, which to us, doesn't allow the creation of professional looking interfaces.

The system satisfies Nielsen's Heuristics for HCI. As an example, we keep the user informed about the system status throughout the program: an animated dialog box is displayed while the user's program is being assembled. By having various parts of the system satisfy these heuristics, Simulizer creates an enjoyable experience for the user.

4.1 CPU Visualisation

The CPU Visualisation is one of the most visual parts of the system, and allows the user to learn a great deal about how the CPU is working on instructions at a low level. For this reason it was crucial that a great level of detail and thought was put into the design in order to provide ease of use.

One of the initial considerations to be made with regards to this visualisation was how it was going to be designed and how it would look. By looking at what content would already be familiar with the user base, it was decided that a reasonably faithful block diagram of the R3000 processor would be most appropriate. However, recreating an accurate block diagram without abstraction would not be appropriate level of detail for our target audience. The main reason for this is the excess of communication links within the diagram. As a result of this, these links would be cut down, with only one link from one component to another. This lead to a well justified abstraction of the bus communication of the processor, which has only aided the cause of providing software that is easy to learn from.

When viewing the visualisation, the separate components can easily be seen against the light background and allow the user to quickly get a grasp on how the CPU works. Varying component sizes specify their relative importance, for example the mux unit is small, as this performs simple operations, whereas the registers are large. This allows the user to quickly get an idea of the importance of various components. Different shapes were also used, for example the ALU and adder components are the typical shape shown in existing resources, which allows the user to quickly understand the function of various components. Wires can also be seen on the diagram and link various components together, these wires are easy to see and are all organised and positioned to provide a clean design. Distinct arrowheads can also be seen at the end of each wire, which signifies the direction of flow for data, these are fairly large and allow them to easily be seen by the user.

Tool-tips were used extensively in the CPU block diagram to give more information to the user. These tool-tips are revealed when the user hovers over a CPU component. They provide detail about the function for each component, and allow the user to learn more about why different components are needed. For example, the program counter is used to store the address of the next instruction. The tool-tips have a dark background which allows the user to clearly differentiate them from components and allow the information to be read easily.

Once the user starts running the simulation, they will start to see red circles moving along the wires in the block diagram. These red circles are easy to see against the blue wires and show the movement of data/signals through the CPU. The movement of data is also synchronised with the text editor, for example when an instruction is being fetched, the visualisation will animate to show this, and once the editor moves to decode an instruction, the visualisation will also show a decode. The user will also notice components highlighting when the visualisation is running, this shows when a specific component is being used, for example the ALU is performing a comparison. This works well when combined with the data movement animations and allows the user to see how data is being directed through the CPU.

To provide more information, a rectangular box appears at the top of the visualisation. This contains in information about the current instruction or stage of the simulation. This box is a different colour to the CPU components allowing it to be easily distinguished. The box also fades in and out at different points in the simulation, which draws attention to it when required. The information is used to describe instructions and stages, for example that the current value of the program counter is being passed to memory.

Overall the CPU visualisation has many different visual aspects to aid in the user's learning and allow them to easily observe how a CPU functions inside a computer. As each instruction is executed, detailed information regarding the movement of data and the interaction of the CPU components is shown to the user. This information is also reinforced by the informative tool-tips.

4.2 Pipeline View

The pipeline view is a unique and innovative approach to visualising instruction pipelining, which is traditionally represented in a static form. Simulizer's dynamic pipeline view gives the user a more natural way to experience a multi-stage process as opposed to static resources which attempt to show multiple stages in a single diagram.

The basis of the design for this pipeline view arose from existing pipeline diagrams available to the students through common resource sources. By designing the component in such a way, it provided a form of compatibility in the scenario in which the user would like to carry out 'further reading' on pipelining (a very large topic which can go into great detail). Additionally, special attention was made in the design of this component; the team had been informed pipelining was typically a highly difficult topic to teach and so special provisions were made to keep this visualisation simple, only boosting its ability as a resource for learning and teaching.

When the user is observing the pipeline view, they will immediately notice the different colours of various blocks. These colours are used to show different instructions across different cycles in the pipeline. The different colours allow the user to very quickly see how each instruction is passing through the pipeline. The user can also click on a single instruction, this will highlight the instruction across all cycles and also jump to the correct line inside the editor. More information about each instruction can also be seen by hovering over it. The data is displayed such as the instruction name, type, address and line number.

Bubbles are shown as red circles in the pipeline view; their colour and shape distinguish them from regular instructions. By hovering over each bubble, the user can see more information about the hazard such as whether it's a control or read after write hazard. Through the including of such a feature, the visualisation provides information for gaining a deeper understanding of pipelining and hazards.

As the program is running, the pipeline view will start to populate and move along in real-time as the program continues to run. This can easily be disabled with the use of a check box to suit the user's preference. The user can also use the left and right arrows to move back to a previous time in the pipeline, or alternatively the user can enter a cycle number to jump to, allowing fast movement along the pipeline.

Overall the pipeline view provides great detail and visual aspects to allow the user to quickly and easily learn more about pipelining, which is extremely difficult with existing resources. The user can easily see different stages of the pipeline such as fetch, decode and execute as well as see how instructions are moving through the different stages and the pipeline hazards that occur.

4.3 Editor

Because our user base of computer science students and lectures will be familiar with an IDE (integrated development environment), we thought that we should follow some of those conventions. One of the first features that people notice when they open an IDE is syntax highlighting. Syntax highlighting provides a link between what the code does and colour. This creates makes code more readable as it provides the user with instant recognition. The syntax highlighting in Simulizer is based heavily upon the grammar of the assembly language, therefore providing highly appropriate and understandable highlighting for different cases.

The second feature that people notice about IDE's is error checking. Simulizer notifies the user of a mistake in their code by putting a red box around the mistake. Simulizer also puts a red X in the gutter (alongside the line numbers). When the user hovers over the X, they are presented with more information about the error. We felt it was important to not display the error all the time as the user might want to work on another part of the program before fixing the error. The designing of the interface to include this was yet another bid to improve upon Spim Simulator, and providing the error messages in this way not only achieved this, but additionally provided familiarity with the Eclipse IDE, which the majority of 2nd year Computer Science students will have had experience with.

When a program is running, the current line that is being fetched/decoded/executed is highlighted in the gutter. We used colours on the red to green scale to indicate the process that each line needs to go through to be finally executed. Green indicates fetch, orange decode and red execute. These colours can be hovered over to describe to the user which stage each colour represents (more of as a reminder than as something that they will need to see all of the time). These descriptions were a HCI design decision made upon the request of a user who had tried out the system.

4.4 Program I/O

The Program I/O had a few design choices to improve its usability. The first iteration of the I/O logger was very primitive. It had a log of the simulation output as well as a method for the user to input data into the simulation. While this was functional, it did not meet our high standards for the interface. For example, we found that we didn't notice when the simulation was waiting for an input. This issue was dealt with by emphasising the Program I/O when input was required. This increased its usability drastically, but we felt more could be done.

We had a look at the information that was being input/output to the logs, and realised that both MIPS and annotation outputs were being sent to the same place. We decided that we should split these into different "streams" of information in the Program I/O to make it clear to the user where each message came from. Since it is only possible to see one of these streams at once, we notify the user with a red dot next to the tab when something is posted to that stream. When running a large program, this makes information output from your program much easier to digest, also removing any potential confusion. If, for example, a user has sent many logs to the logger using the JavaScript annotations, then they can focus their attention on that stream.

4.5 High Level

We felt that the high level visualisations needed to be very colourful, in order to describe what is going on through animation. During the project we wrote two different high level visualisations, one for visualising Towers of Hanoi and one for list visualisation. Both of these visualisations allow for swapping of elements/discs. These swaps are animated so that the user can see where the item started off, and trace it through to where it ends up.

For the Hanoi visualisation, each disc has a different colour (as well as a different size) to easily distinguish between them. This is a lot more visual and usable than having all of the discs look the same and putting a number in each disc.

The list visualisation is a more general visualisation and therefore has a few more HCI components. There are emphasis and markers. Emphasis is a way to attract the user's attention to a particular element of the list. It does this by flashing that element red. Red is a very prominent colour that can be easily seen in frontal vision. The reason we flash the colour is to produce a sense of movement which can attract the user's attention when the element is in the user's peripheral vision.

One of the major design points of these visualisations (and arguably of the whole system) is the generality provided by their design. The list visualisation, for example, isn't hard coded to work with a single algorithm, it can be used for any list-based algorithm that wishes to use it. This is because these visualisations are controlled by the user; specifically the JavaScript annotations they add into their programs. If the user were to write insertion sort in MIPS for example, then they could use the same visualisation for this purpose, annotating the necessary points in their code. These visualisations can be designed and developed completely independently from the rest of the system and then easily integrated with the existing visualisations. This design choice provides incredible extensibility to the system, with the possibility of more complex visualisations being added, such as graphs and Turing machines. Furthermore, once this system is released under a public license, any user could potentially contribute to adding new visualisations without modifying the underlying system. The generality is provided because this allows any algorithm to be visualised and an arbitrary number of visualisations to be created. Therefore, this is considered one of the major selling points of this system with regards to the visualisation design.

5 Software Engineering/Risk Assessment

In order to ensure a high-quality piece of software was produced by the end-point of this project, various software engineering practices have been employed. These practices vary from the way we organised our time developing the system, to how we ensured our code was easy to read/understand and maintainable.

One of the early software engineering related choices to be made within the project was the question of what development model was going to be followed. This project has had a relatively short life-span and so a model was required that would deal with this well, as well as putting safeguards in with respect to some of the risk assessment carried out on the project. For this reasons, an Agile method of development seemed best suited to this project and its team.

Following an Agile methodology allowed the incremental development of the system in short sprints. This was an effective model for Simulizer due to its highly modular design: one week could produce one new visualisation module by one team member for example, and the next week could produce another. The previous week's work could be completed, and, in many cases, the next weeks work by that team member wouldn't too heavily depend on the previous weeks (an exception to this would be the model/CPU simulation, which was built in parts, over multiple sprints, and these parts unavoidably have to depend on other parts in some cases). The sprint length throughout the project did vary very slightly but only between sprint lengths of either 1 week or 2. The length of specific sprints would be determined at team meetings (which occurred on the same day as our sprint start date; Tuesday), dependent on what increment(s) needed to be completed next. As an example, the first major development sprint lasted two weeks for the reason that a lot of initial infrastructure needed to be put into place such as the basic model layout. In contrast, the sprint to finish the prototype was strictly a week long: the overarching goal of the sprint was to achieve a working prototype to be demonstrated (something which was a success due to a well-planned out development schedule; looking at the whole project plan signalled that we were exactly on time with regards to project progress at that point).

Another advantage found through the use of an Agile development cycle was the ease of integration with risk mitigation strategies. Due to the large amount of functionality required by Simulizer, very comprehensive risk analysis had to be carried out on the potential project to ensure that, if any significant issues arose during development, they would be dealt with in an appropriate fashion (the risk analysis for the project can be found in the systems specification document). One of the most straight-forward techniques for minimising risk was by way of compromise – cutting out some of the functionality of the system. With an Agile methodology, if the team is behind time, then the tasks of one members work for a given week can be removed and that team member can be re-focused on to a higher priority task that needs to be done/finished. Fortunately, (quite possibly due to the benefits of an Agile model), no such compromises have had to be made within the development cycle and in fact, Simulizer contains more functionality (particularly in its educational and visual aspect) than the original specification submitted documented (all additional functionality is documented in the final specification).

Yet another risk identified in the project was the risk accompanying if the project wasn't favoured by users. The solution/avoidance for this was through the good software engineering practice of simply interacting with the users of the system (outside of simple user testing). In the case of user interaction, the main source of information was Ian Batten. Throughout the project, semi-regular discussions would be held with Ian regarding certain aspects of the system, describing what features he would find most useful. In addition to this, after the point of having prototype, he would receive live demonstrations of the system, focused on the aspects geared to teaching. Through the use of this practice of user interaction, any user based risk was reduced significantly (as an aside, another risk identified was due to the project being built only shortly after learning the concepts initially; discussions with Ian were also an avoidance strategy for this risk, and having such conversations aided significantly with some of the more conceptually challenging areas of the system (the pipeline being one of them)).

As well as using well-known software development models and keeping in good contact with Simulizer's user base, another software engineering practice employed (and one which proved to be incredibly useful) was that of design patterns, particularly with respect to the linking of the model and the views of the system. In particular, we have utilised message passing, producer-consumer, model-view-control and observable patterns.

The use of design patterns gave a much desired attribute to Simulizer: low levels of coupling. This loose coupling is most apparent in the splitting up of the model and view(s) of the system. As testament to this, the CPU visualisation and the CPU simulation were developed completely independently of each other. The CPU simulation had been tested fully, and the CPU visualisation had been visually tested to ensure all animations were working as intended. Once it was guaranteed that both of these components were stable, then they were linked together (essentially by the visualisation registering to listen to the simulator). However, this wasn't something carried out until week 8 of the project (relatively late in) and it shows how little these components actually depend on each other's internal workings.

As another valuable software attribute, Simulizer also enforces high cohesion. The main cause for this was the way in which Simulizer was designed. In the design stage of the project, multiple designs were considered (for the simulation in particular), and the end design was in fact reasonably faithful to the standard block diagram for a MIPS processor (the same one displayed as part of the visualisation). In being faithful, while avoiding unnecessary levels of encapsulation for the sake of being faithful, all of the components/classes have a very easy to explain reason for having the functionality it does, and the links to the objects it does. As an example, the CPU class has the methods fetch, decode and execute, and a memory object as a field in the class. The CPU needs to be able to access memory and also needs to be able to carry out the Instruction Execution cycle, and so the structure here is very clear, making the entire class structure very easy to understand.

Having gained these valuable attributes has significantly aided in the challenge to make Simulizer a maintainable and extensible piece of software. The low coupling between the model and view alone creates a very maintainable system: if, for example, JavaFX (the library used for the UI) was to become deprecated, then only the view would have to be updated; the model is still completely usable. With regards to extensibility, the system is highly modular in nature. If for example, a developer wanted to have a longer pipeline, then it would be straightforward to do. All that would have to be done is create a new class extending the base CPU class, in particular extending the `runSingleCycle` method; the rest of the code is fully reusable. Another sign of extensibility in the system is how simple it is to add a new high-level visualisation. As an example, if someone wanted to add a visualisation for the travelling salesman problem for example, they would only have to do 4 things: Firstly, they would have to write whatever visual aspects they would like to see in JavaFX. Secondly, they would have to implement the algorithm in MIPS assembly code. The third step is to add any annotations using Simulizer's JavaScript annotation system to interact with the visualisation (messages sent from the simulation). Finally, the visualisation should listen to the simulation and that's it. A new visualisation has been added with no real changes to any of the existing code base.

At a more fine grain level, practices have been used to ensure code is of a high quality and is readable. The general tactics of suitable commenting and code documentation through Javadoc have been employed, as well as using meaningful names for classes and methods. Every non-trivial class and method has a description of its purpose before it's declared and names such as `programCounter` and `instructionRegister` are used as field names, rather than un-helpful names such as `x1`, `x2` etc. Enforcing a good code style has only helped make the system more understandable to an external developer/user reading it: it gives an easy link to the design documentation as well as giving a detailed insight into the way the software behaves in use. As well as this, a common code format has been used within the team. Individual members of the team can write in whatever format/style they desire, but in order to give commonality to the finished code base, format files have been created to be used on the code, to make every Java file conform to every other in terms of code format, improving the code readability further.

6 Evaluation

Through the process of completing this project, a high level overview can be taken and examined, allowing a full evaluation of this project.
Firstly, it is worth looking at what has been achieved in the project, with respect to both the original specification and the software as a whole. When looking back at the specification, it has been shown (please refer to the test report) that all of the functional requirements have been satisfied. This means the final software produced achieves at least what was initially set. In addition to this, working ahead of schedule has allowed the addition of originally unspecified features, such as the replaying of instructions in the CPU visualisation.

Other added features include the addition of windows for labels for example, but also the extensive help windows and references available from within the system. These allow the user to retrieve information on what each register should be used for, the instructions available for use within the simulator as well as information on the system as a whole.

From a design perspective there is one aspect of the system that has worked extremely well and allows an incredible amount of extensibility from the system. This is the generality with respect to the algorithms that can be run and visualised.

From personal experience of adding new algorithms to the system, it has proven very straightforward with no consideration of the internals of the system required. New algorithms can be written via the writing of the MIPS code for it; the annotation language then allows easy integration with visual components through straightforward methods for operations such as swapping, emphasis etc. If a visual component already exists that can be used for visualising an algorithm, then that is the extent of what is required. If a new visualisation needs to be made, that is all that has to be done: it can be written in JavaFX and then used. This has given the system the aforementioned generality by giving the potential to add visualisations for any algorithm desired to be included within the system. Consequently, this had led to the production of a highly flexible system.

As well as the achievement of the functional requirements, there were also the non-functional requirements to be satisfied. The easiest method of testing said requirements is through the delivery of the software to the users. Although this is covered in detail in the test report, the users' opinions on the software are crucial to the evaluation of the system. Two of the biggest (non-functional) aims of this project was to improve upon the existing software (Spim Simulator) and to also provide a system which was helpful to the students in their understanding of the content of the Computer Systems & Architecture module. When asked both of these questions explicitly, the large majority of students said it was preferable over Spim Simulator and also that it would prove helpful. Additionally, when finding users' favourite visualisation, the results came back with a reasonably even distribution, suggesting that, in general, all of the visualisations provided are proving as useful as each other. In general positive user response signals that performance/usability requirements have been met. Any other non-functional requirements were met simply, by keeping to them, for example, not allowing local file access through the JavaScript annotation language.

Another aspect of the system which has proven successful is the plan for its development. At no point in this project has time been a major issue, and for the most part, it could be seen that components/features of the system were being delivered on time. Although this required a fairly dense work rate (arguably any project being conducted in this timescale would require this), the plan was well organised such that the completion of work was possible, with minimal bottlenecks within the development plan. It is for this reason that Simulizer was able to contain its large array of functionality.

The previous points have discussed what has been achieved with the system, and how the system has been a success with particular regards to the requirements set out in the specification document. However, to provide a balanced evaluation of the system, areas for improvement will be discussed. Possibly one of the biggest occurrences of this in the system was a possible underestimation of the precision required to get a large multi-threaded system correct. In particular, earlier on during the development of the system, less attention may have been given to preventing concurrency errors, such as race conditions and deadlock. As a result, significant amounts of time nearer the end of the project was spent fixing said concurrency errors, and in some cases, certain methods of threading had to be redesigned such that they were more stable and less error prone.

One more possible improvement that may have proven useful is all team members having more knowledge of the whole code base. With no doubt, the separation of work between team members was very effective in maintaining a well-designed system while still providing an efficient workflow. However, later on in development, when most of the threading issues started to appear, certain team members were an unable to provide much aid in fixing these errors due to their work being separate from the threading design. If every team member had an understanding of the whole system, then this may have aided in the fixing of bugs later on. However, this still had an upside that it enabled certain members to focus on these areas, while other team members worked on other important areas of the project.

Reaching the end of the project, there are two defining aspects to be discussed, which measure the team's success over the last 10 weeks. The first of these is the goal of Simulizer being used as a resource in the Computer Systems & Architecture module in the following years. After discussions/demonstrations of the system with the module leader (Ian Batten), there is a real chance of it being used. He is planning to take a copy of the software once the project timeline is over to take and use to see whether it is suitable. As a result, this may entail further additions/changes being made to Simulizer after the project is over. This represents a significant achievement for the team and for the software, and so this is something the team is more than willing to continue with.

In addition to this, there is further potential for Simulizer. After discussions with the team's tutor and Ian Batten, it has been mentioned that there is genuine innovation within Simulizer, giving it the potential to be used as a teaching resource. Included in this are aspects of the system such as an easy to understand visualisation of a pipeline (something which had previously been described by Ian Batten as difficult to teach). In addition, the JavaScript annotation language is an effective method of debugging assembly programs at a high level and allows for fine grain control over the visualisations displayed on screen as well as allowing explicit changes to the simulator (such as register changes for example). It is for this reason that, potentially, a paper could be written discussing the innovation Simulizer is providing as a teaching and learning resource. Given the teams current stage into their respective academic careers, and the relatively short time scale for the project, this would be a great accomplishment.

As a balanced whole, Simulizer can be considered a successful project. It has achieved everything that was originally set to be accomplished (and more so in some cases) and it has been well received by the user base (both sides of it as well). There are some gaps for potential improvement, but given the environment in which Simulizer has been developed, this is almost inevitable.

7 Teamwork

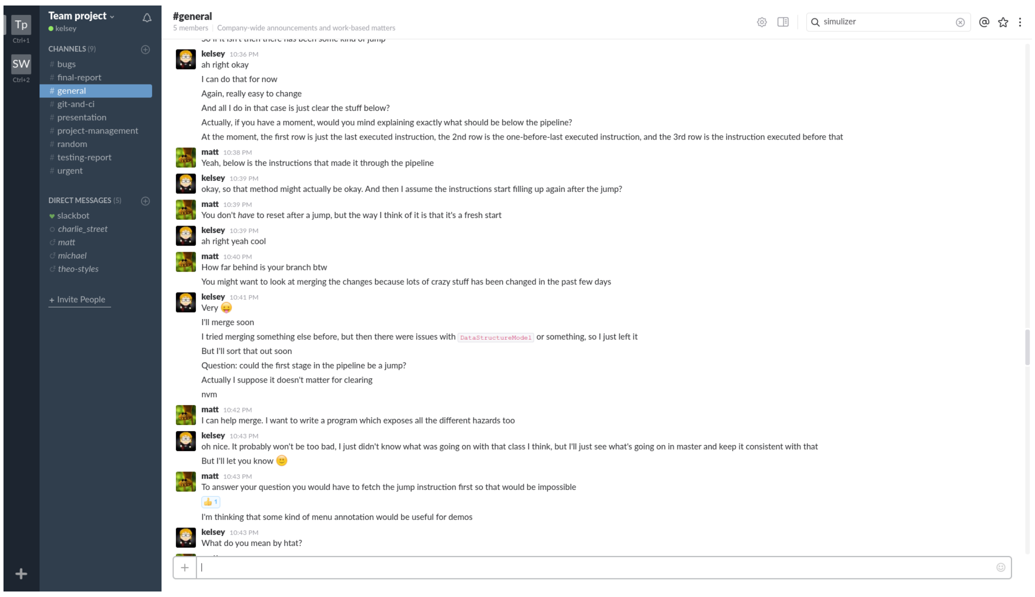
We implemented a number of strategies and tools in order to help us work effectively as a team. Even when we all met for the first time, we were discussing how we would communicate, e.g. through email, social media, etc. Since we had similar goals for achieving the best piece of software we possibly could, it meant our working styles were very compatible. Everyone regularly checked to see if there were new messages or new tasks assigned to them, which allowed for open discussions and swift development. Furthermore, our dedication to the project meant that we were eager to implement the system and resolve any issues.

Shortly after meeting each other, we decided on a weekly meeting time that suited everyone. Every week since the start of the project, we had a team meeting lasting an hour where we discussed our plans for the future of the project, any issues to be resolved etc. One of the first things we decided in these meetings was to follow an agile development methodology. This meant that in future meetings we planned the tasks for the next sprint, and we could review any outstanding tasks/issues from the last sprint.

To achieve our full potential, we wanted to play to the strengths of our team members, and conversely avoid assigning tasks to members who would not be comfortable with that task. For example, Charlie had never used JavaFX before, so it was clear that assigning him heavy visualisation tasks using JavaFX would not be the best strategy; Charlie developed the underlying CPU simulation. Similarly, Kelsey does not do the Computer Systems & Architecture module, meaning that he would be more suited to the visualisation side and more general design tasks rather than the low-level of a CPU.

In order to keep track of the tasks for each sprint, and any issues in the project, we used a tool called Taiga, which is specifically suited to agile projects. Taiga allowed us to create a backlog of tasks to be completed and then assign them to certain team members and place them in sprints. Taiga helped us to stay organised as we could always see which tasks needed to be completed, which issues had not been resolved, and generally helped us keep track of our progress.

We realised immediately that we would need some form of instant messaging for our communication outside meetings, so we wanted to find something that would be well suited to team-work activities. We decided on Slack, which has many features particularly useful for software development. By using Slack, different team members could communicate in different channels about different issues. We were also able to integrate Taiga and our GitHub repository with Slack. This notified us when a commit was made and when a task was changed on Taiga. It was easy to communicate with everyone on Slack, both because the software was useful, but also because everyone would keep Slack open at home, and messages wouldn't go unread or unanswered.



kelsey 10:36 PM
ah right okay
I can do that for now
Again, really easy to change
And all I do in that case is just clear the stuff below?
Actually, if you have a moment, would you mind explaining exactly what should be below the pipeline?
At the moment, the first row is just the last executed instruction, the 2nd row is the one-before-last executed instruction, and the 3rd row is the instruction executed before that

matt 10:38 PM
Yeah, below is the instructions that made it through the pipeline

kelsey 10:39 PM
okay, so that method might actually be okay. And then I assume the instructions start filling up again after the jump?

matt 10:39 PM
You don't *have* to reset after a jump, but the way I think of it is that it's a fresh start

kelsey 10:39 PM
ah right yeah cool

matt 10:40 PM
How far behind is your branch btw
You might want to look at merging the changes because lots of crazy stuff has been changed in the past few days

kelsey 10:41 PM
Very 😛
I'll merge soon
I tried merging something else before, but then there were issues with `DataStructureModel` or something, so I just left it
But I'll sort that out soon
Question: could the first stage in the pipeline be a jump?
Actually I suppose it doesn't matter for clearing
nvm

matt 10:42 PM
I can help merge. I want to write a program which exposes all the different hazards too

kelsey 10:43 PM
oh nice. It probably won't be too bad, I just didn't know what was going on with that class I think, but I'll just see what's going on in master and keep it consistent with that
But I'll let you know 🙂

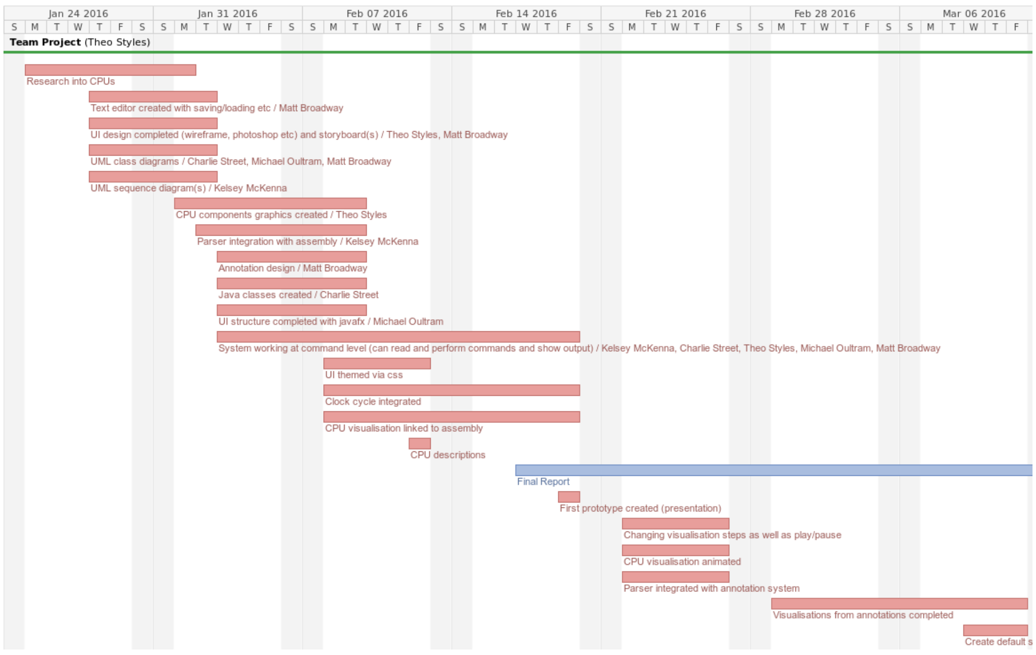
matt 10:43 PM
To answer your question you would have to fetch the jump instruction first so that would be impossible
👍 1
I'm thinking that some kind of menu annotation would be useful for demos

kelsey 10:43 PM
What do you mean by htat?

Occasionally we would use Skype if we needed to talk about something in more depth that we didn't get enough time to discuss during our meetings.

Early in the project, we invested some time into investigating tools that would improve our productivity and efficiency in the future. For example, we looked into using build automation tools like Maven. We believed this would be important because we would be working with various dependencies, IDEs, and operating systems. Once we decided to use Gradle, we configured it to work with our various development environments, and it allowed each member to specify required dependencies in a single file, which would be available to everyone through GitHub, and then everyone was synchronised with the same setup. Investing some time into this research has caused future development to run very smoothly and has allowed us to avoid potential problems with dependencies and configurations.

As a more general plan project management tool, we used Wrike as an idealistic model of how our project would develop. By being able to view the entire span of the project in a single place, it meant we could more selectively choose tasks for our sprints to ensure that we would meet these goals. For example, for the prototype presentation, we knew we had to focus more on visualisation so that we would have something to show for our work, even though a lot had been done in the back end of the system.



The approach of democratically making decisions was consciously chosen shortly after our team's formation. We agreed that if there were any minor disputes about a feature of the system then the team member 'in charge' of that feature would have a larger say in the final decision.

Democratic decision-making gave all ideas equal opportunity within the team. For example, if team members were passionate about an idea but with a differing opinion, each member was able to present their point of view, and then we could decide on a plan of action as a whole. As mentioned, the team is very dedicated to the project, and these discussions helped to manifest even better ideas and designs. Moreover, each discussion was concluded before moving on; there wasn't an instance where a team member decided to do it 'their way' without prior agreement, which meant everyone was aware of project decisions.

Our meetings were always very lively with lots of ideas and discussions, which meant we never came away from a meeting wondering what was expected in the next sprint. Each team member had a task assigned to them before the meeting was over to ensure that everyone was able to contribute in achieving the team's full potential.

A feature of our team-work and communication which proved integral to development is our branch model using git. The master branch would be reserved for stable builds, and if we needed to work on a bug, feature, or component of the system, a separate branch would be created. This allowed us to work independently on our own branches without ever interfering with anyone else's code. We were still able to work on each other's branches if we wanted to collaborate on a piece of functionality, but the ability to evolve, or accidentally corrupt, the system without interrupting anyone else's progress has been crucial to our success. We then regularly transferred our changes over to the svn repository to allow the module coordinators and our tutor to view the source code.

We chose git as our version control system for various reasons. All of us were very familiar with git, meaning that we saved time by not learning everything again in svn. In addition, our branching model was achieved quite painlessly using git. Merges were much easier than they would have been in svn, which allowed us to save valuable time during development. A small amount of research into the comparison of svn and git also helped us make this decision.

Throughout the development of our system we needed a repository for files such as design documents, presentation slides, images, etc. We decided to use a shared Google Drive folder for this rather than our GitHub repository because we wanted to upload binary files and visual-based documents, which otherwise would have been more awkward to view and modify.

By following a collaborative and efficient work-flow throughout the project, we were able to stay motivated and focused in order to accomplish our goals.

8 Summary

After discussing Simulizer in depth throughout the contents of this report, it is worth now summarising the project. This section will discuss what Simulizer is, what its aims are, how the creation of the software has been achieved, and the success of the project as a whole.

At its finished state, Simulizer is a large piece of software that, in the most basic sense, simulates and visualises the operations of a MIPS R3000 processor. Simulizer provides multiple levels of visualisation: lower level visualisations such as the CPU and pipeline visualisations give an insight into the running of the processor (information about registers, memory etc. only complement this); high level visualisations take a look from a different perspective, examining how the algorithms written in assembly code are running.

The most significant aims of this project were to satisfy the needs of our user base, by providing an accessible, usable environment to work in, accommodating for both teaching and learning scenarios. In order to do this, communications with the user base were held, in particular with the module leader, in order to ensure the project was moving in the right direction to satisfy these aims.

In order to create such a piece of software, a attention to detail was required towards the design of the software. The design (and indeed the project as a whole) incorporated many different Software Engineering practices, such as actively trying to enforce loose coupling and high cohesion throughout the software, as well as the use of multiple design patterns, including the producer-consumer pattern, as well as a general listener pattern for the connections between the model and many views of the system. The design of this system was carried out with careful consideration, hence the multiple iterations of design before pursuing one through to the development stage of this project.

The brief for this project was to visualise an algorithm (or suite of algorithms) and so a lot of thought went into ensuring the visualisations were as easy to understand and effective as possible. Arguably the central visualisation of this system is the CPU visualisation and this was designed with the students in mind, via the design incorporating the processor block diagram which the students will be familiar with from their lectures. Furthermore, one of the main selling points with regards to the visualisations of this system is its generality. With respect to the high-level visualisations in particular, a simple system for adding new visualisations has been created; the user writes the algorithm in MIPS, adds the visual component (may not be necessary if a suitable visualisation is already available), then adds the appropriate JavaScript annotations to control the visualisations. Finally the visualisation needs to start listening to the model but this is a very simple process.

As well as managing the development of Simulizer, effective management of the team also had to take place. Different tools were used for this, such as Slack for easy communication, and Taiga for effective project management. Obviously tools alone wouldn't make a highly productive, strongly bonded team and this was achieved through an agreed system of decision making and also through mutual respect for each other within the team. This allowed all team members' ideas to be brought forward and as a result, Simulizer is a culmination of these ideas.

Simulizer has truly tried to innovate as a teaching and learning resource, through the use of visualisations of low level components/concepts related to processors/program execution that, to our knowledge, no other simulator (or other piece of software) has achieved previously. The innovation and general project success has been gauged by the satisfying of the requirements set out at the start of the project, but more so by the response received by the users of the system. The general response was that Simulizer is a useful and highly usable piece of software that succeeds at providing better educational interfaces than its competitors. Proof of this success is demonstrated by the possibility that Simulizer may actually be used as a resource in the teaching of Computer Systems & Architecture next year. This is a major achievement for the team and one that every member is incredibly proud of.

8.1 References

School of Computer Science (2015), 'Module 19430 (2015)', `http://www.cs.bham.ac.uk/internal/modules/2015/19340/` [accessed 09/03/2016]

Larus, J. (2011), 'SPIM: A MIPS32 Simulator', `http://spimsimulator.sourceforge.net/` [accessed 09/03/2016]

9 Glossary

- ALU – Arithmetic and Logic Unit: A component in the CPU that carries out the large majority of computations in the CPU
- Assembly Language – A type of programming language situated in-between high level languages such as Java and machine code. Assembly language programs can directly access registers, memory etc.
- Binary Search – An efficient $O(\log n)$ searching algorithm for searching in an ordered list.
- Branch – Similar to a jump but technically on a smaller address range, and usually based upon the result of some condition being tested.
- Bubble Sort – A highly inefficient sorting algorithm of $O(n^2)$ complexity.
- Bubbling – A name given to a pipeline stall. When running a pipelined CPU, there are cases where certain stages of the pipeline have to be stalled in order to prevent certain types of error (such as reading something before it is written to). In order to prevent this, 'bubbles' are placed in the pipeline: NOP instructions (these do nothing) are placed into the pipeline to stall a certain stage for a cycle (or more). Visually, it is similar to the idea of a rising bubble.
- Bus – A method of transferring data between CPU components. It is essentially just a wire.
- Clock – In the case of the CPU, a clock is used to keep all components in the CPU synchronised. As a rule of thumb with RISC (Reduced Instruction Set Computer) machines, one round of the Instruction Execution cycle lasts one 'tick' of the clock.
- CPU – Central Processing Unit.
- High-Level Visualisation – A visualisation of high level algorithms running, such as showing the movement of data in a linked list as a sort is being run.
- Instruction Execution Cycle – A series of steps that is run in cycles in order to run a program. It consists of multiple stages, usually 4/5 but in the case of Simulizer, simplified down to 3: Fetch, decode and execute.
- Instruction Register – A register which stores the instruction currently being executed.
- Jump – Moving to a different point in the program during execution other than sequentially.
- Memory – When memory is discussed, it refers to the main memory (i.e. RAM). The main memory in a MIPS processor consists of 5 main segments: one for OS reserved data, one for the code to be executed, one for all statically defined data, one for dynamically allocated data (the heap), and one for the stack.
- MIPS – A type of processor not so common in modern times, but the type of processor used for example in the 'Computer Systems & Architecture' module to demonstrate the running of the processor.
- Non-Pipelined Execution: In this case the IE cycle just focuses on one instruction: one instruction is fetched, decoded and then executed, and then the next one and so on.
- Pipelining – A form of instruction execution that significantly speeds up program execution without changing the clock speed. It does this by (in a simplified sense) executing instruction n, while decoding instruction $n + 1$, and fetching instruction $n + 2$.

- Program Counter – A register that stores the address of the next instruction to be executed.
- Register – A very fast piece of memory located in the processor (very close to the components). In a MIPS processor, registers are 4 bytes in size.
- Syscall – An instruction in the MIPS language, which, when given a code which is stored in the v0 register, will perform a certain operation. A large majority of these operations are related to IO operations but also include, and are not limited to, allocating new heap memory and exiting the program.
- Word – 4 bytes.

10 Individual Summary

10.1 Charlie Street

Throughout this project, I have had the opportunity to take on challenges that prior to this project I never would have thought I would have completed. Firstly, 11 weeks ago I would never have considered having the software we ended with; I am incredibly proud of what the 5 of us have achieved over the past 10 weeks and I hope the software reflects the hard work put into it. Personally, throughout the development of this project, my main role has been to develop the simulation that is central to the back end of the system (with respect to the brief, the algorithm, if you will).

In the early stages of the project I had an active role in designing the system, creating UML diagrams for the back end of the system and also providing the user persona for the system, giving the team a detailed insight into the users we were actually making Simulizer for. The design of the simulation was a task I spent a lot of time on. This design was crucial as it would determine how the entire back end would work, and so it needed to be as clean and usable as possible. The difficulty and importance of this meant the design required multiple iterations to get right, but by the end it resulted in a design which was reasonably easy and simple to implement.

Carrying out the task of writing the simulation required me to know in detail about the processor, so as well as being able to write such a simulation in the first place, it required me to really understand how the processor works. The coding of this simulator required me to be fairly faithful to what I was implementing, including the part of the simulator I am most proud of which is the pipeline. This required very careful thought to get right. One of the most important design points of this simulator was that it had to be compliant with other simulators, and thankfully, I think this has been achieved.

As well as developing the simulation, I also played a key role in the writing of the documentation of the system, having written many sections for the original specification as well as the final report and test report. For the test report, I contributed to approximately two thirds of all unit tests for the system (I wanted strong confidence that my simulation was correct, regardless of the fact this meant testing each single instruction, of which there were 80). I also completed a comprehensive set of integration tests for the system as well as organising and carrying out all of the user testing for the system. For the final report I contributed heavily to the Software Design, Software Engineering, Introduction, Evaluation and Summary sections. I also wrote most of the general content and slides for the final presentation.

Reflecting on the work of us as a team, as well as being efficient in the development, I believe we have worked really well, playing off each other's strengths in order to ensure the right person was always assigned to do a certain section/ part of the project. Additionally, I think as a team we have got on really well together and it has been a pleasure to work with everyone over the last 10/11 weeks.

To summarise, this project has lead me to personally stretch myself in my abilities. I didn't think that at the start of the semester I would have written a simulation for a CPU as part of a project like Simulizer. This has been the largest project I have ever completed, and the longest time I have ever spent working in a team, and I can honestly say that I have enjoyed the process of creating and developing Simulizer, with all its difficulties, very much.

10.2 Kelsey McKenna

My main role in our project has been to develop the high-level and low-level visualisations, which complement the key visualisation – the CPU. However, over the course of the project I have also been in charge of the code editor (and its syntax and error highlighting), researching and reporting on key aspects of our software configuration, e.g. Gradle, UI aspects, setting up a framework for developing the final report, among other things.

Since I am the only member of the team not studying the Computer Systems & Architecture module, it made sense for me to focus on parts of the design and implementation that were not heavily based around the low-level architecture, so I have been responsible for more of the visualisation side.

Before starting the project, I was slightly worried about not having the same level of knowledge in CPU architecture as my team members, who all study the module. I even considered trying to catch up on the module content, but after we started assigning tasks to each other, I soon realised that not having detailed knowledge of CPUs was not going to affect the amount I would contribute.

Before we started programming, I was in charge of researching build automation tools, e.g. Maven and Gradle, and deciding which one(s) we should use. After doing some research I decided on Gradle, as it was more appropriate for our purposes, and there was a lot of helpful documentation and various plug-ins. As well as this, we needed a parser to work with our custom assembly language, which includes annotations. I looked into the parsing tool ANTLR and wrote a prototype of the language we would be using (a reduced version of MIPS assembly).

In the design phase, I created various sequence diagrams to help us get a better understanding of how we would operate the system in the high-level, and I wrote a number of use cases to help us extract our requirements. In addition, I wrote a number of UML class diagrams and use cases for general usage of the system.

To develop appropriate high-level visualisations, e.g. tower of hanoi, lists etc., I needed to create a number of prototypes in order to test the capabilities of certain features in JavaFX. Getting a prototype for animations took some time, and I made sure to design the high-level visualisations so that they can easily be adapted and extended, and so that they provided a general interface that was easy to connect with the other parts of the system, especially the annotations. Similarly for low-level visualisations, e.g. the pipeline view, I went through a number of iterations of prototypes in order to achieve the final result. Again, the design helped to separate the model and view meaning that once I had completed the visualisation side, it would be very easy to link with the model in the back end.

Throughout the project I took charge of a number of tasks which I believed improved the efficiency of our group. For example, I wrote a script to transfer all our code from git to the svn repository. I also had the idea of setting up a framework where we could write our final report in markdown format, which is very simple, and then automatically convert to $\LaTeX$. For the last 10 days of the project I created a daily plan, with each day detailing exactly what we should have finished by the end of that day. I felt this was important to help us continue at a good pace, and to make sure that we were on track to finish everything on time. I was also diligent in fixing and reporting bugs, which meant that we could keep track of the stability of the program.

Overall, I feel I have learned a lot about working in a team, about tools and strategies for developing software projects, and I have improved my software engineering abilities. I look forward to working on future projects where I can continue to develop these skills.

10.3 Matthew Broadway

I have been very fortunate to have worked in a team of highly skilled programmers who share my ambition and appreciation for crazy ideas which has allowed us to surpass all initial expectations and create a genuinely useful tool.

Straight after being assigned to the team I brainstormed many ideas into a presentation. When presented to the team they unanimously liked my favourite idea (the CPU visualisation) the best. I then got in touch with our lecturer for the Computer Systems and Architecture and arranged a meeting to discuss what things we could do to make the project interesting or useful.

The very first thing I did towards the code was research into Gradle and created a small JavaFX prototype application which used the internal window system that we ended up using for the rest of the project.

Early on in the project I was responsible for writing the grammar for our language and the assembler which takes the AST and does error checking and validation, bundling the assembly into an easy format for the simulation to consume.

To accompany the assembler I created a very extensive unit test suite for checking the assembler against hand calculated results and also some tests which compares the assembler with the behaviour of SPIM.

Another task that fell on me was to figure out how to encode and decode between binary and Java representations of signed and unsigned integer values. This was very challenging because Java only supports signed integers. I created the `DataConverter` class which provides this functionality.

After that work was done (which was the initial bottleneck before the other members could really get started) I worked on the code editor component. Up until this point we had been using an existing component that we were not happy with, and so I did research into using the web-technology based 'Ace editor'. Initial tests were promising so I spent a week or so creating the syntax highlighting and code folding rules (along with all the other editor features) in javascript. I was the only contributor to this section as the other members were working on the other functionality such as the visualisation components.

Next I worked on the annotation system. I built the framework for extracting the annotations from the source code and binding them to statements (this meant adding to the assembler code) and then moved on to the runtime environment and 'bridges' which would control the Java components from the annotations. This included work on embedding highlighting rules for javascript inside our language. I was the only contributor to the annotation component (except for additions to the bridges when new visualisations were created).

Next I worked on the clock for the simulation. This required a separate thread which would notify the simulation thread at regular intervals and hold back its execution if necessary. This component underwent several revisions and is now reliable and efficient. I was the only contributor to this component.

Finally I worked on the message passing system which up until this point had been rudimentary and did all its processing on the simulation thread. To get around this I prototyped several designs revolving around using a thread pool to distribute the work. The hardest part was the synchronisation of the listeners with the simulation. I created a mechanism for preventing the simulation from advancing to the next cycle before all waiting messages were processed. This move broke many of the visualisations and other listeners. Michael and I spent a few days hunting down all the problems caused by the move.

10.4 Michael Oultram

I came into team project thinking that I'd prefer to code alone, but leave wondering how I'll ever manage to code without a team. From the beginning this project seemed rather ambitious and definitely something that I wouldn't have tackled alone. My role within the project is to act as the 'glue' between the simulation and the visualisation. While this did mean that I got to work on many different sections of the project, a lot of these sections were a collaboration.

In the first week, Matt and Kelsey created a prototype of the modular UI. I expanded this prototype into the fully featured modular UI we have today. The underlying code structure of the Internal Windows was changed for greater control like automatically resizing the Internal Window and to make it easier for other parts of the application access these Internal Windows. I used this new code structure to create some of the more basic Internal Windows like the Registers, Program I/O and Help Windows.

I also added some features that the original UI prototype did not have, mainly layouts, settings and themes. Layouts took me multiple weeks to get working correctly but fortunately I could work on this feature in a different branch. This stopped everyone having to work with my broken code whilst I'm trying to develop it. Putting each feature in a branch allowed for me to easily switch between different parts of the application, and to priorities what features/bugs fixes were most important at the time.

During one week of the project, I was assigned to bug fixes and performance enhancements. In this week, I spent a lot of time looking at how threads were being used throughout the project. In a few places in the project, threads were not being used optimally. For example, the CPU Visualisation would create 21 threads everytime the window was opened and these threads were never closed. I created a partial fix for this, which Theo fully fixed later on.

Kelsey wrote the visual code for the High Level visualisations. Because he needed to see the visualisations to write the visual code, he just linked it to the annotations in a very quick and easy way. When looking at the threading issues of the project, I realised that this was a bottleneck and so I designed and implemented a better solution. This better solution involved using an MVC pattern, such that the annotations use the model, and when this model changes, the visualisation (or the view) is updated. I designed it in a way that allows us to view multiple visualisations at the same time. More importantly this new design allows for the visualisations to be opened midway through a program execution, and the visualisation would be in the correct place. In addition, the new solution solved some of the threading issues that we were previously experiencing.

I found working in a team quite beneficial in some circumstances. For example, I've had some code that I've written refactored to use the new double colon notation (something I wouldn't have known about otherwise). Team members are also a great way to test my code works, as I'm quickly notified if I've broken anything or if I could write something in a better way and I'd like to think I have returned the favour with some of the thread optimisations that I performed.

10.5 Theo Styles

I would have never imagined at the start of the project we would have the software we do today. The project seemed highly ambitious at the time, but with hard work and determination, we developed a very useful and sophisticated piece of software. I have thoroughly enjoyed working with the team and have learned a great deal about both Java and CPU architecture concepts along the way. My main role in the project was to develop the CPU visualisation, which shows the block diagram of a R3000 processor and the movement of data through it.

In the first week, I created story-boards to demonstrate how the UI would look along with the layout of different components. This allowed us to have a solid base for the UI layout along with how the user would perform various actions and how the UI would change to reflect this. I also created the schedule for the project using a tool called Wrike, this allowed us to easily allocate time and people for different sections of the project.

I then started creation of a basic CPU block diagram using JavaFX, this was created in a separate branch on git and allowed me to work efficiently without affecting other team member's code. Once the basic blocks were created, I started to link each component together with various wires for example connecting the ALU with the registers. Once the basic layout was created along with the simulation, I started to link the two together using the message passing system, implementing animations for separate instructions such as branches and jumps. This required research on the data paths of various instructions and allowed me to learn a lot more about how a CPU actually executes a program at a low level. Once the animations were implemented I quickly realised the block diagram needed to be extended to support operations such as branching. Due to the nature of the design this was done extremely quickly, and adding additional components such as the multiplexer was trivial.

Working as a team has been very beneficial and I feel each person has worked extremely well. Each person has worked on aspects they prefer and specialise in, which has allowed the project to develop quickly and efficiently. Using tools such as Slack has also been very useful and has allowed the team to communicate much more clearly. I have learnt a great deal from looking at various team members coding methods, such as the double colon notation, along with issues between threads and the best way to deal with them.

Overall, I have greatly enjoyed working in a team as well as on a large project such as Simulizer. I had never worked with a team on such a large project before and it has been an amazing experience working with everyone as well as a great learning experience.

11 Appendix

11.1 How to create a new visualisation

Simulizer's visualisations can be easily extended because of the underlying design. The existing visualisations, e.g. tower of hanoi, are completely independent of the simulation and any other visualisations. In this section, we will describe how a new visualisation can easily be added to Simulizer.

In the steps below, we will describe how a potential graph visualisation could be added. The following classes should be written by the developer:

- `GraphModel`
- `GraphVisualiser`

11.1.1 Writing the model

The first step in creating our graph visualisation is creating the model. This should be a subclass of `DataStructureModel`. This class operates as the model in a typical MVC design pattern. For a graph, we might want to model the nodes, the edges, the weights etc.

Any methods that need to be called by the annotations should be public.

For example, we could create a simple model with a single method for the annotations:

```
public void completeGraph(Integer... nodes) {
  // generate the graph
  setChanged();
  notifyObservers();
}
```

which would do the required computations to generate a completed graph whose labels are given by the values of `nodes`, and will also inform the view that a change has taken palce.

In order to access the information in the model, we should have appropriate get methods. In this example, we might want a method like this:

```
public MyGraph getGraph() {
  synchronized (graph) {
    return this.graph;
  }
}
```

Note the `synchronized` keyword to prevent thread interference and memory consistency errors.

11.1.2 Writing the visualisation

The visualiser class should be a subclass of `DataStructureVisualiser`. In our example, `GraphVisualiser` should have a constructor like this:

```
public GraphVisualiser(GraphModel model, HighLevelVisualisation vis) {
  super(model, vis);
  ...
}
```

If, for example, you want to visualise the graph using the `Canvas` component from the JavaFX API, then you could add the line

```
getChildren().add(canvas);
```

to the constructor and then use the canvas in visualisations.

The `DataStructureVisualiser` class handles the animation timer, receiving messages, etc. We need to implement the `processChange` and `repaint` methods. The `processChange` method should contain any logic, if any, for updating parameters for the view. For example, if animations are required, an animation time-line could be instantiated. You should call `setUpdatePaused(true)` at the end of the `processChange` method, and `setUpdatePaused(false)` should be called after any animations finish. The `repaint` method should repaint/resize the components in the window.

By implementing the `getName` method you can give your visualisation a name, e.g.

```
@Override
public String getName() {
  return "Graph";
}
```

11.1.3 Linking it to the system

Now we need to link our model and visualiser to the rest of the system so that they can be accessed via annotations. We need to modify the following classes:

1. `ModelType`: we need to add a constant to the enum, e.g.

   ```
   public enum ModelType {
     HANOI, LIST, FRAME, GRAPH;
   }
   ```

2. `HighLevelVisualisation`: we need to add our `GRAPH` enum as a case in the switch statement in the `addNewVisualisation` method:

   ```
   switch (model.modelType()) {
     ...
     case GRAPH:
   vis = new GraphVisualiser((GraphModel) model, this);
   break;
     ...
   }
   ```

3. `HLVisualManager`: we need to allow the user to load the visualisation from their annotations using a user-friendly name, e.g. “graph” in the `create` method:

   ```
   switch (visualiser) {
   ...
   case "graph":
     model = new GraphModel(io);
   ```

```
        break;
      }
```

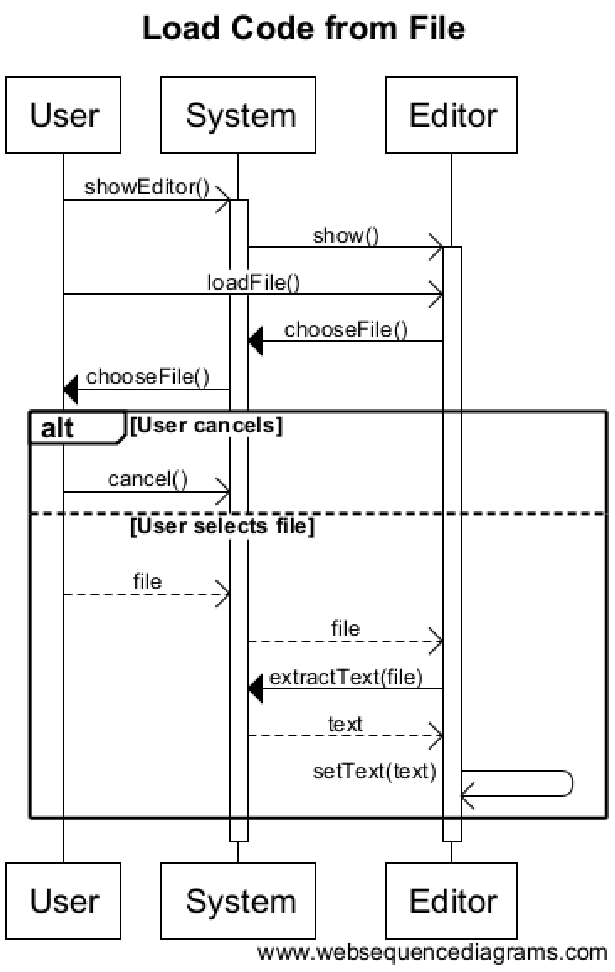
11.1.4 Usage

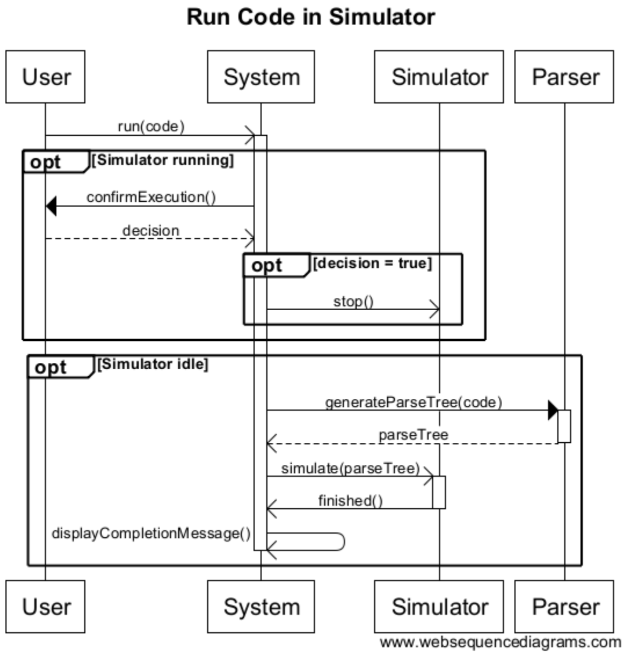
Now the visualisation can be called via annotations in the user's code! To load the graph visualisation, use

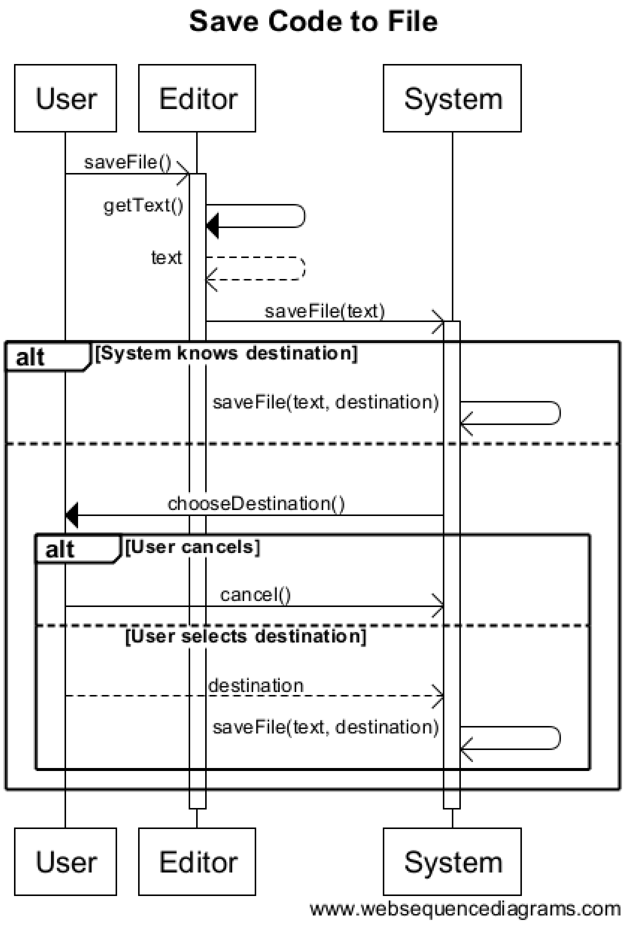
```
# @{ var g = vis.load('graph') }@
```

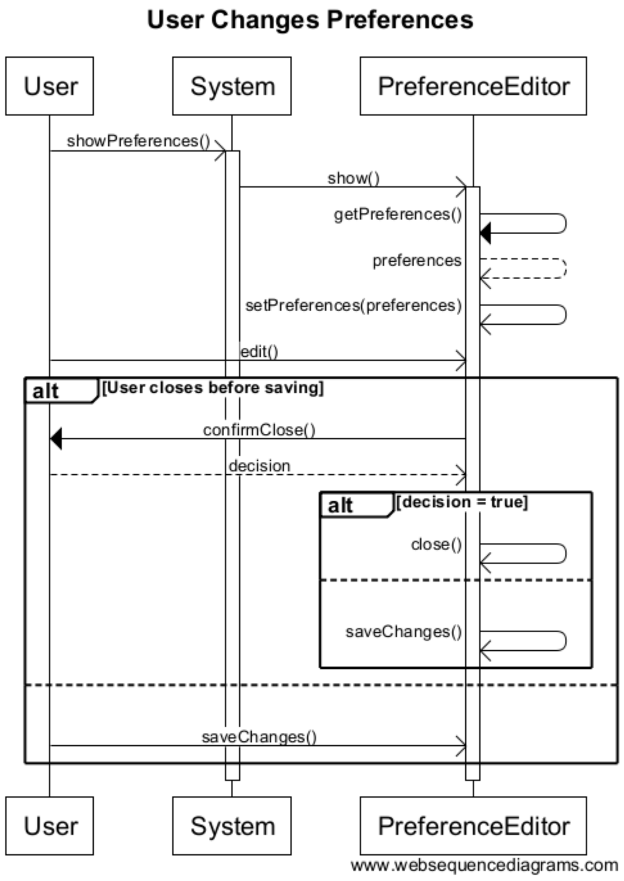
More details on how to use visualisations can be found in the user guide.

12 UML Sequence Diagrams

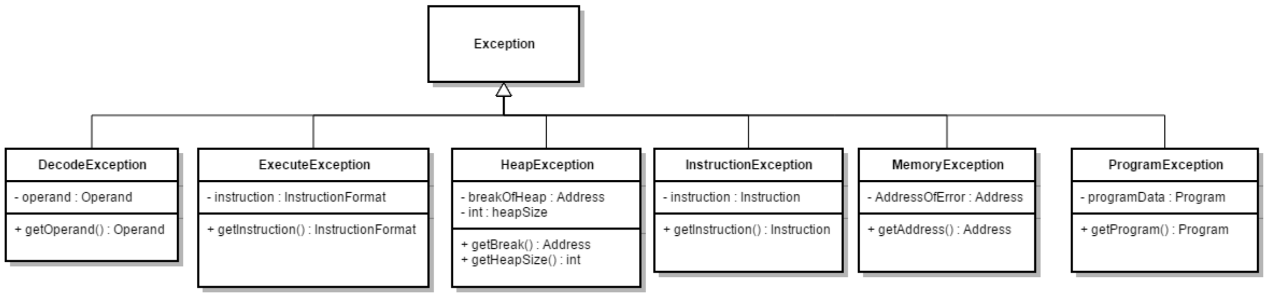


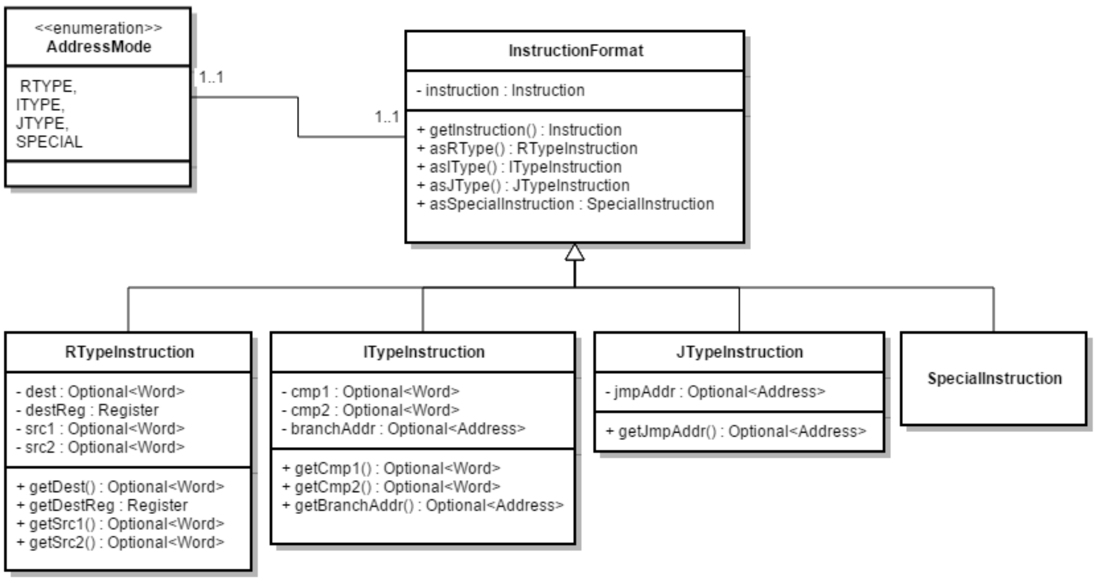

Run Code in Simulator
User
System
Simulator
Parser
run(code)
opt [Simulator running]
confirmExecution()
decision
opt [decision = true]
stop()
opt [Simulator idle]
generateParseTree(code)
parseTree
simulate(parseTree)
finished()
displayCompletionMessage()
www.websequencediagrams.com


Save Code to File
User
Editor
System
saveFile()
getText()
text
saveFile(text)
alt [System knows destination]
saveFile(text, destination)
chooseDestination()
alt [User cancels]
cancel()
[User selects destination]
destination
saveFile(text, destination)
www.websequencediagrams.com



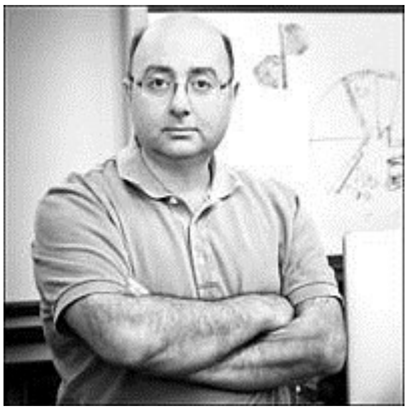
13 Further UML Class Diagrams





<<enumeration>>
AddressMode
RTYPE,
ITYPE,
JTYPE,
SPECIAL
1..1
1..1
InstructionFormat
- instruction : Instruction
+ getInstruction() : Instruction
+ asRType() : RTypeInstruction
+ asIType() : ITypeInstruction
+ asJType() : JTypeInstruction
+ asSpecialInstruction : SpecialInstruction
RTypeInstruction
- dest : Optional<Word>
- destReg : Register
- src1 : Optional<Word>
- src2 : Optional<Word>
+ getDest() : Optional<Word>
+ getDestReg : Register
+ getSrc1() : Optional<Word>
+ getSrc2() : Optional<Word>
ITypeInstruction
- cmp1 : Optional<Word>
- cmp2 : Optional<Word>
- branchAddr : Optional<Address>
+ getCmp1() : Optional<Word>
+ getCmp2() : Optional<Word>
+ getBranchAddr() : Optional<Address>
JTypeInstruction
- jmpAddr : Optional<Address>
+ getJmpAddr() : Optional<Address>
SpecialInstruction

Nicolas Yates - Lecturer (School of Computer Science)



"I'd really like to give my students something more useful for their studies.

Background

Age: 50

Occupation: University Lecturer

Location: School of Computer Science, University of Birmingham

Technology Level: As someone who has worked/taught in Computer Science for many years, Nicolas is fully capable of using any technology, old or new

Description

Nicolas is a lecturer in the School of Computer Science at the University of Birmingham. He has been a lecturer for multiple years now after studying there during his education. Nicolas specialises in low-level programming and hardware and although he still carries out research, he much prefers teaching undergraduates at the university. He teaches 3 different modules during the year, one of which is the Computer Systems & Architecture module.

Being someone who is generally well prepared, Nicolas spends a lot of time prior to his lectures working on slides, handouts and other supplementary material such as code for his students to run and edit at their own will. For this reason, he ends up writing a lot of MIPS assembly code for the module, which he does by typing it up in his favourite text editor and then tediously debugging it within Spim simulator. This takes him a very long time and he would very much like to speed up this process.

Nicolas likes to keep things simple in his lectures, even when difficult content is being covered and so actively utilises simple diagrams, usually colour coded, to help the student's understanding of the material. Students have responded to positively to this, and so he would like to translate into the way he show's his assembly code working.

Goals

- Wants to help his students gain as high marks as possible in the Computer Systems & Architecture module.
- Wants to be able to use visual representations as aids to his teaching/lecture material.
- Wants a nicer environment to write all of his demonstration assembly code, so he can write and run his code in one single package.

Current Frustrations

- At this current moment in time, Nicolas has to use Spim simulator for all his demonstrations and also to give to his students to use for their assignments etc. because it is the most suitable for running MIPS assembly code.
- He doesn't like teaching using Spim Simulator because firstly, it is really not suitable for projecting onto a screen since everything is highly text based. Secondly it is not very self-explanatory. In order to give the students a good understanding, he needs a simple to understand projection on screen to aid what he is saying.

Scenarios

- Write and run (with error checking) some MIPS compatible assembly code in a single easy to use package, saving him time, allowing him more time to prepare for lectures.
- Run some code/algorithms on the simulator in a lecture scenario so that it benefits the student's learning/ understanding.
- Giving some context to his lectures by running algorithms already known by the students.

John Talbot - Undergraduate Student (School of Computer Science)

“I’d love to be able to *see* what’s happening”

Background

Age: 20

Occupation: Undergraduate Student

School: School of Computer Science, University of Birmingham

Technology Level: As a computer scientist, John is highly comfortable using any kind of technology and has good intuition if he needs to pick up anything new.

Description

John is a second year Computer Science Student at the University of Birmingham. As part of his 2nd semester John is taking the Computer Systems & Architecture module. Although he is taking this as a compulsory module, he looks forward to it as, to him, it is a bit of a change of pace compared to some of his other modules.

John is a very prepared student, and so likes to keep up to date with all of the module content, and reinforce this knowledge into his head so that, come the time for revision, he need only refresh his memory, rather than essentially learn the course from scratch. For this reason, he really likes to make sure he has a good understanding of all the content.

Although John is a very highly skilled programmer in high-level programming languages, he never did a Computing course at A-level (he learnt and honed his programming skills in his personal time over the few years before arriving at university). This meant that all of the content on the components of the CPU and assembly language were all completely new to him and so he feels he has to spend more time than others going through all of the lecture material to gain a good understanding.

In order to help him wit h this, he commonly uses the simulator software recommended by the course lecturer known as Spim simulator. This lets him run his code and check information about the registers as it is running. However, he finds the presentation of this information really bland and in some cases he finds himself getting more confused about the material.

John has looked about on the internet to try and find better software (he always checks online for good supplementary material for all of his courses) but couldn’t find anything useful. He would love to see a more visual version of the simulator released such that he can improve his knowledge of the computers architecture and also hopefully boost his marks in the process.

Goals

- Wants to get a really good mark in the Computer Systems & Architecture module.
- Wants to be able to write (and understand) low level assembly language code in an easy to use, user-friendly environment.
- Wants a better understanding of the internal running of a CPU, particularly with topics such as pipelining and the fetch-decode-execute cycle.

Current Frustrations

- At the moment, in order to help him complete the assembly language coding part of the module, John is having to use the Spim-simulator software. Although he understands it simulates well, he finds the software really difficult to use as there is no visual aspect, just text displays of registers and memory. This makes him quite annoyed when using the software for any substantial amount of time. The software doesn’t help him understand the content any better, he just uses it to check his assignments work and that is it.

Scenarios

- Write and run (with error checking) some MIPS compatible assembly code in a comfortable easy to use way to complete assignments.
- Run some algorithms on a simulator to watch how the internals of the CPU are working
- Get descriptions of the all of the CPU components so that he can learn and remember them for his exam.
- Revise his knowledge of high level algorithms.

Name: Run Code in Simulator
Identifier: UC 01
Initiator: User
Goal: Simulate the execution of the code supplied in the code editor.
Preconditions: The user has entered some code in the code editor.
Main Success Scenario:
1. User clicks the “Run” button in the code editor.
2. Include use case *Parse Code*.
3. Include use case *Run Simulation*
4. System informs user that the execution has finished.

Extension:
1. Another simulation is running
    1. System asks user if they want to cancel the current simulation.
    2. User selects yes.
    3. Resume 2.

1.1. User doesn’t cancel the running simulation
    1. Fail.

2. Use case *Parse Code* fails
    1. System informs user that the code wasn’t able to be parsed, and gives information about the line number where the error occurred.
    2. Fail.
3. Use case *Run Simulation* fails
    1. System informs user that an error occurred while executing the code, and displays any relevant information about the error.
    2. Fail
3. User cancels execution
    1. Resume 4.

Name: Load Code from File
Identifier: UC 02
Initiator: User
Goal: Extract the text from the specified file and place it in the code editor.
Preconditions: The user has entered some code in the code editor.
Main Success Scenario:
1. User opens the code editor.
2. User clicks the “Load from File” button in the code editor.
3. System presents user with a file chooser dialog.
4. User selects a “.s” file and confirms their choice.
5. System copies text from file into code editor.

Extension:
4. User clicks “Cancel” button
    1. Fail.
5. System could not read from file
    1. System informs user about the error, e.g. the file has special permissions.
    2. Fail.

Name: Save Code to File
Identifier: UC 03
Initiator: User
Goal: Save the text in the code editor to file
Main Success Scenario:
1. User clicks the “Save to File” button.
2. System presents user with a file saving dialog.
3. User selects the destination directory and types a name for the file.
4. System saves the file to the specified destination.

Extension:
2. System already knows file destination
    1. Resume 4.
3. User clicks cancel button
    1. Fail.
4. System could not write the file to the specified destination
    1. System informs user about the error, e.g. the directory is read-only
    2. Fail.

Name: User Changes Preferences
Identifier: UC 04
Initiator: User
Goal: Update the system preferences based on user input
Main Success Scenario:
1. User requests to view a dialog box of preferences
2. Editor retrieves preferences and displays them in a dialog box.
3. User edits preferences
4. User saves changes.

Extension:
2. Editor could not retrieve system preferences
    1. Fail.
4. User closes window before saving changes
    1. Editor confirms user’s choice to close without saving changes.

4.1. User closes window without saving changes.
    1. Stop

4.1. User cancels closing window.
    1. Resume 3.