Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2023

Imperial College London

Project Title:	**A High Performance Digital Circuit Simulator for ISSIE**
Student:	**Yujie Wang**
CID:	**01714652**
Course:	**Electronic and Information Engineering**
Project Supervisor:	**Dr Thomas Clarke**
Second Marker:	**Dr Christos Papavassiliou**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

I have used GitHub Copilot and ChatGPT as an aid in the preparation of my report. I have used GitHub Copilot to generate Latex template code for my report. I have used ChatGPT v4 to improve the quality of my English throughout. However all technical content and references comes from my original text.

Acknowledgements

I would like to express my sincere gratitude to Dr. Thomas Clarke for his guidance, support, and supervision throughout this project.

Abstract

ISSIE is a successful CAD tool used for teaching undergraduates digital circuit design. However, its current simulation engine suffers from slow performance on large circuits. This project aims to develop an enhanced simulator to address this issue. The new simulator aims to achieve significant improvements in time and memory complexity, providing a speedup of at least 10 times and 10 times more memory efficiency compared to the existing simulator. Additionally, technical debts present in the current implementation will be addressed. The new simulator should be compatible with existing functionality in ISSIE and maintain robustness and reliability. The emphasis will be on enhancing logic simulation, which is more crucial for user experience compared to symbolic simulation. Overall, the new simulator meets the specified requirements and offers improved performance and maintainability.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Motivation

ISSIE[7] is a highly successful CAD tool for teaching undergraduates - with aspirations to be of use professionally using Verilog input etc. Its simulation engine is critical for acceptable performance on large circuits. The current simulator uses a very interesting algorithm section 2.3 which is for a number of reasons slower than it should be (detailed in subsection 3.1.1).

On typical computer, the existing simulator can process about 3000 component *times* tick per millisecond. This is not fast enough for large circuits and long runs. For example, simulating a design with 10^4 components for 10^6 clock cycles would take about one hour to finish.

This project therefore aims to create an enhanced simulator, building upon the foundation of the existing one, with a specific emphasis on achieving significant time complexity improvements (at least a $\times 10$ speedup in simulation speed). In addition, the new simulator should also improves memory efficiency, enabling the simulation of larger designs and the retention of long simulation history for backtracking.

1.2 Report Structure

This report will first go through the background of the project, including its tech stack and the core data structures and algorithms used in the simulator. Then it will discuss the performance bottlenecks in the existing simulator and how the new simulator addresses them. After that, the report will discuss the implementation details of the new simulator in stages, including the extensions that have been added to this project to explore future directions. Then, the later chapters will cover how testing and benchmark are done to ensure the functional correctness of the new simulator, and measure its performance. Finally, the

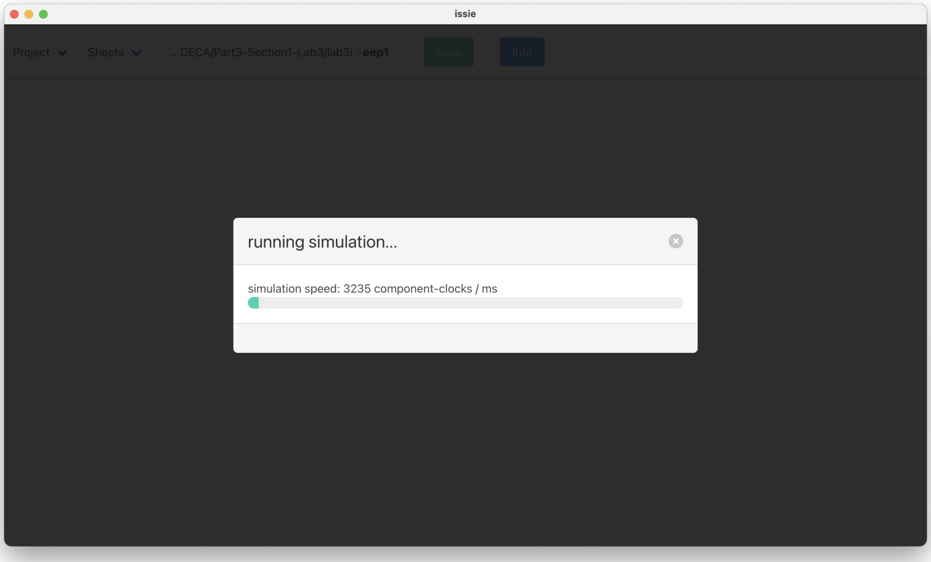


Figure 1.1: ISSIE simulating EEP1 on a laptop with Apple M2 chip.

report will discuss the evaluation of the new simulator and conclude the project with a summary of the project and future work.

Chapter 2

Background

This section will first go through the tech stack used by ISSIE, then gives a brief introduction to the data structure and algorithms used in the current implementation of ISSIE simulator.

2.1 Tech Stack

In brief, ISSIE is built upon the Elmish [12] architecture, the core logic of ISSIE is implemented in FSharp [13] and expose to React [22] for user interface, these FSharp code will be compiled to JavaScript [8] during build time by Fable [14]. Also during build time, the generated JavaScript code will be bundled by Webpack [32] and be integrated into an Electron [4] application. At runtime, the Electron application will use Chromium V8 [27] engine to executes JavaScript code.

2.1.1 Elmish

Elmish is a functional programming library for building user interfaces in FSharp. It provides a model-update-view architecture, similar to the Elm Architecture in Elm [11] programming language. The idea behind Elmish is to allow developers to build web applications in a functional, type-safe, and predictable manner. Elmish follows the principles of functional reactive programming (FRP) and provides a set of primitives and abstractions to manage application state and update the UI in response to user actions and events. Elmish can be used with a variety of front-end frameworks, such as React and Blazor and has become a popular choice for developing modern, scalable, and maintainable web applications in FSharp.

2.1.2 FSharp(F#)

FSharp is a functional-first, multi-paradigm programming language that was developed by Microsoft. It is a member of the ML family of languages.

Its strong-typed nature and powerful type inference system makes it best suitable to be used in projects like ISSIE that put robustness and maintainability at first place.

ISSIE coding guideline discourages the use of mutable fields because they make data and control flow difficult to understand and mutable variables are against the functional programming paradigm. However, there does exist a couple of special mutable fields in the simulation states for performance consideration. For large scale circuit simulation which involves both large amount of components and large amount of simulation steps, copying and creating immutable states in the simulation in each iteration would lead to an unacceptable performance penalty. For example, increment one element in an immutable list of 1000 element would need to copy the entire list with the target element updated.

Therefore, mutable fields such as `ClockTick` are introduced to allow low-cost updating of simulation data.

2.1.3 JavaScript

JavaScript is a high-level, single-threaded, dynamic language that is widely used for creating interactive and responsive web applications [18]. It was originally developed by Netscape in the mid-1990s and is now supported by all modern web browsers.

ISSIE is transpiled to JavaScript to be able to run on most platforms using Electron 2.1.5. This maximises reachability of ISSIE to its target customers in education sector.

JavaScript possesses several interesting features, the following sections will give a brief overview to some of them that are encountered in this project.

Type Coercion

The dynamic nature of JavaScript implies that variables in JavaScript are not assigned specific types, and certain operators, like the `+` operator, contribute to this dynamic behaviour, making it somewhat unpredictable. For instance, the `+` operator can be used to add two numeric values, such as `1 + 2`, but it can also be employed to concatenate strings or combine a string with a numeric value, as illustrated in Listing 1.

Numeric Types

JavaScript has two numeric types, Number and BigInt. Number represent numeric value using IEEE-754 double-precision format [20], this means it only support integer of 52 bits (mantissa in double-precision format has 52 bits), from $-(2^{53}-1)$ to $2^{53}-1$. BigInt can

```
var a = 1 + 2;     // a = 3
var b = 1 + "a";   // b = "1a"
var c = 1.1 + "a"; // c = "1.1a"
var d = "a" + "b"; // d = "ab"
```

Listing 1: Different use cases of JavaScript + operator.

represent integers with arbitrary magnitude [2]. Unlike Number whose arithmetic computation can be performed using hardware as most modern CPUs support double-precision arithmetics, arithmetics of BigInt are implemented in software [1] as no CPU has register to hold data of any length.

Array Types

JavaScript also has two types of arrays: `Array` and typed arrays. The former can store any JavaScript objects, while elements of the latter are raw binary values of supported numeric types, e.g. 8-bit unsigned integer, 32-bit unsigned integer, 32-bit floating point number.

Error Handling

Zero division in JavaScript e.g. `42/0` results `Infinity` because division of JavaScript Number is performed according to IEEE-754 binary double-precision arithmetic [9].

2.1.4 Fable

Fable is a FSharp to JavaScript transpiler that allows developers to write web applications and services using the functional-first FSharp language. It allows developers to take advantage of the many benefits that FSharp provides, such as strong type-checking, functional programming, and its powerful type inference system, while also being able to run their code in web browsers or JavaScript environments.

2.1.5 Electron

Electron is an open-source framework for building cross-platform desktop applications using web technologies, such as HTML, CSS, and JavaScript. Electron allows developers to create native-looking applications for Windows, macOS and Linux with a single codebase. It uses Chromium as the web rendering engine and Node.js for server-side scripting, providing access to the full suite of Node.js APIs.

With its simplicity and compatibility, Electron have made it the best choice for developing ISSIE which aims to provide first-class experience for all modern operating systems and devices.

Two changes in recent releases of Electron limit the total size of simulation data to 4 GB, both are compile time options for V8.

Since Electron V14, pointer compression [21] is enabled which improves performance by reducing the size of pointers on 64-bit platforms but restricts heap memory of each v8 process to 4 GB.

Since Electron V21, sandboxed pointers [29][28] is enabled which protects V8 from memory corruption but blocks use of off-heap memory as backing store to hold ArrayBuffer.

2.1.6 V8 Engine

V8 is Google's open source high-performance Just-in-Time JavaScript and WebAssembly [31] engine, written in C++. It is used in Chromium and in Node.js, therefore V8 is also the core dependency of Electron. It implements ECMAScript and WebAssembly, and runs on Windows, macOS and Linux systems that use x64, ARM or RISC-V processors.

The version of V8 (v10.8) [23] used in the latest ISSIE (v3.0.11) has two different engines to run JavaScript code, Ignition interpreter and TurboFan optimising compiler. A new mid-tier compiler maglev is planed to be added in later version of V8.

After been parsed into abstract syntax tree, a given piece of JavaScript code will first be run by the Ignition interpreter to make sure fast first time response. At the same time, feedback will be collected to track the type of variables used since variables are not typed in JavaScript.

If a region of code is repeatedly called and keeps passing type check, it will be spotted as hot code and will be compiled into optimised machine code by the optimising compiler TurboFan.

More interestingly, as JavaScript is a dynamically typed language, the type of a variable appeared in hot code could change and V8 will de-optimise that piece of code for the program to execute correctly.

2.1.7 Git

ISSIE project is managed by git [16] and hosted on GitHub [5]. This project creates branches from the master branch by `git checkout -b` and use `git rebase` to sync new commits from master branch. This makes sure no conflicts would be created when the new simulator get merged into master.

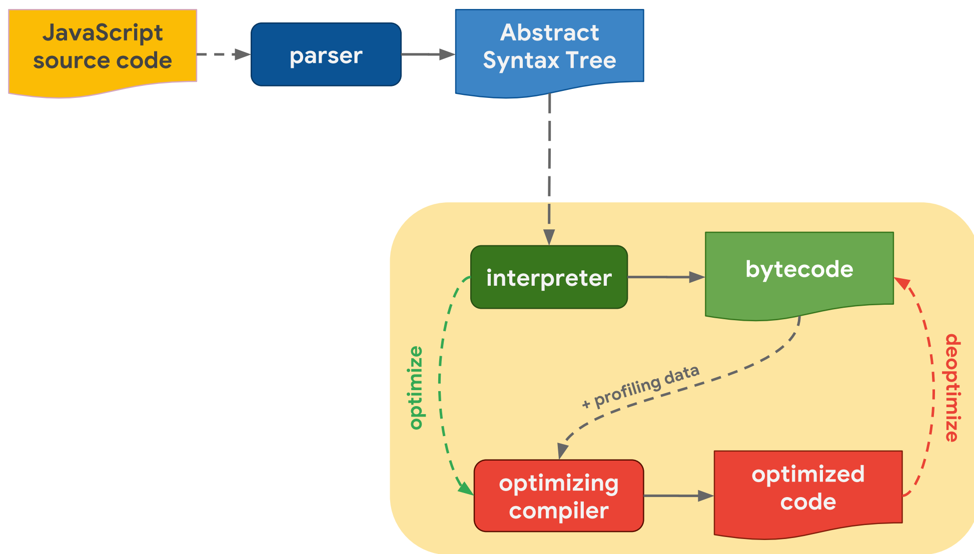


Figure 2.1: High level overview of the V8 engine pipeline [19].

2.2 Data Structures

2.2.1 Types Used in Schematic Editor

ISSIE defines three record types as shown in Table 2.1 and Listing 2 to represent components, wires and ports in schematic editor. They serves three important jobs,

- form a connected graph so that components can be easily found by their ids
- allow design sheets can be easily serialised and deserialised to and from JSON
- contain all information needed to render schematic design sheets and perform simulation

Table 2.1: Record types used to represent circuit schematic.

Type	**Scope of unique Id**	**Usage**
`Port`	Sheet-wise	Represent IO ports on components, form two-way link between ports and components via `HostId`.
`Component`	Sheet-wise	Represent components, allow user created design sheets to be used in other design sheets as `CustomComponent`.
`Connection`	Project-wise	Represent directional wires, holds Ids of the two connected ports.

Each design sheet in ISSIE is represented by a `CanvasState`, which is tuple of a list of `Components` and a list of `Connections`.

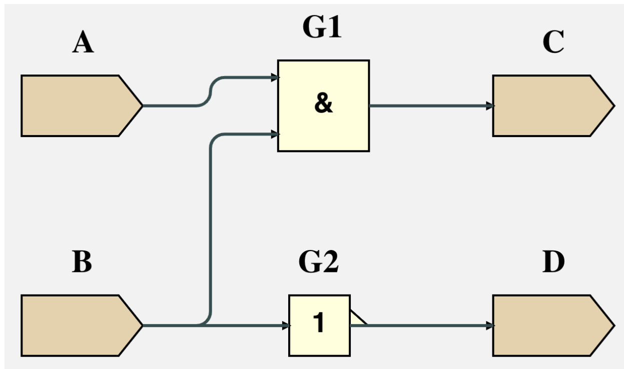


Figure 2.2: Schematic of a simple circuit that outputs $C = A \wedge B$, $D = \neg B$.

2.2.2 Types Used in Simulation

The current implementation of ISSIE simulator uses `uint32` as the underlying data type to store simulation data that has width less than or equal to 32 bit and `bigint` otherwise. `uint32` is used instead of `uint64` because JavaScript does not natively support 64 bit (unsigned) integers as mentioned in subsection 2.1.3.

Three discriminated union [6] types are defined to provide a unified view to simulation data. A FSharp discriminated union type can hold heterogeneous data in one of its named cases. For example, a variable of type `FastBits` in Listing 3 can either be case `Word` which holds a `uint32` or case `BigWord` which holds a `bigint`.

Table 2.2: Discriminated unions used to represent simulation data

Type	Usage
`FastBits`	A unified view to the two underlying numeric types.
`FastAlgExp`	A unified view to the supported algebraic expressions.
`FData`	A unified view to numeric and algebraic simulation data.

These abstraction to simulation data leads to performance overhead. F# discriminated unions are transpiled to JavaScript objects by Fable therefore arrays of `FData` are JS Arrays but not the more efficient Typed arrays. When reading simulation data, each discriminated union object requires one `switch` to determine which case a variable is in. Therefore, extraction of simulation data from a `FData` takes two `switch` statements. The first `switch` determines whether the `FData` holds a `FastData` or a `FastAlgExp`, the second `switch` identifies whether the `FastData` holds a `uint32` or a `bigint`. When updating simulation data, three

```
type Port = {
    Id : string
    // For example, an And would have input ports 0 and 1, and output port 0.
    // If the port is used in a Connection record as Source or Target, the Number is None.
    PortNumber : int option
    PortType : PortType
    HostId : string
}

type Component = {
    Id : string // Id uniquely identifies the component within a sheet.
    Type : ComponentType
    Label : string // All components have a label that may be empty.
    InputPorts : Port list // position on this list determines inputPortNumber
    OutputPorts : Port list // position in this list determines OutputPortNumber
    X : float
    Y : float
    H : float
    W : float
    SymbolInfo : SymbolInfo option
}

type Connection = {
    Id : string // Id uniquely identifies connection globally and is used by library.
    Source : Port
    Target : Port
    Vertices : (float * float * bool) list
}

/// F# data describing the contents of a single schematic sheet.
type CanvasState = Component list * Connection list
```

Listing 2: Core data structures used in schematic editor.

new objects must to be created when updating a `FData`, a new `FastBits`, a new `FastData` and a new `FData`.

Simulation history of each component port is store in `StepArray`, a mutable array that

```
type FastBits =
    | Word of dat: uint32
    | BigWord of dat: bigint

type FastData =
    { Dat: FastBits
      Width: int }

type FastAlgExp =
    | SingleTerm of SimulationIO
    | DataLiteral of FastData
    | UnaryExp of Op: UnaryOp * Exp: FastAlgExp
    | BinaryExp of Exp1: FastAlgExp * Op: BinaryOp * Exp2: FastAlgExp
    | ComparisonExp of Exp: FastAlgExp * Op: ComparisonOp * uint32
    | AppendExp of FastAlgExp list

type FData =
    | Data of FastData
    | Alg of FastAlgExp

type StepArray<'T> =
    { mutable Step: 'T array
      Index: int }
```

Listing 3: Core data structures used in simulation.

acts as a circular buffer storing the value of a signal in simulation up to `MaxArraySize` steps which is configured to 550 in `SimulationView.Constants` module. Its circular nature is implemented by using modulo indexing when read and write its elements, i.e. `let res = outputs[ClockTick`

2.3 Algorithms

In preparation phase of simulation, ISSIE first parses `CanvasState` to obtain all involved components and their connections. In the next step, ISSIE expands `CustomComponents` to their underlying `Components` and create `SimulationComponent` (an intermediate representation) for every `Component` to obtain a flattened graph of simulation components. This graph is called flatten because it does not contain any nested graph for user defined components.

The flatten graph is then used to create `FastSimulation` where each component is represented in its final form, `FastComponent`. Each `FastComponent` contains `StepArrays` to store simulation data of its ports. `StepArrays` of input ports except those of global inputs are links to their corresponding outputs as shown in Figure 2.4 to minimise data copying during simulation. In this stage, ISSIE stores `FastComponents` into different arrays in `FastSimulation` based on their types, e.g. `FClockedComps` for clocked components. One special array is the `FOrderedComps` array which stores components in the order of their dependencies.

`FOrderedComps` improves overall simulation performance by avoiding dynamically finding connected components during simulation. It allows ISSIE to update outputs and state of components correctly by just iterating through ordered `FastComponent` array and call `FastReduce.fastReduce` for each of them.

In simulation, ISSIE simulates the entire circuit in three passes for every clock tick. The first pass is to update the output ports of global inputs. Then, the second pass is to update the states of `AsyncRAM1` components. Finally, the third pass is to update the outputs of all components.

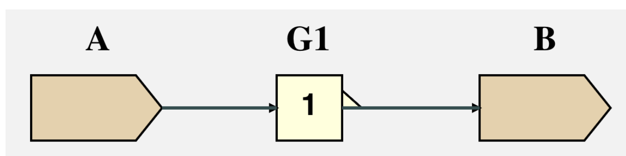


Figure 2.3: A simple circuit that outputs $B = \neg A$.

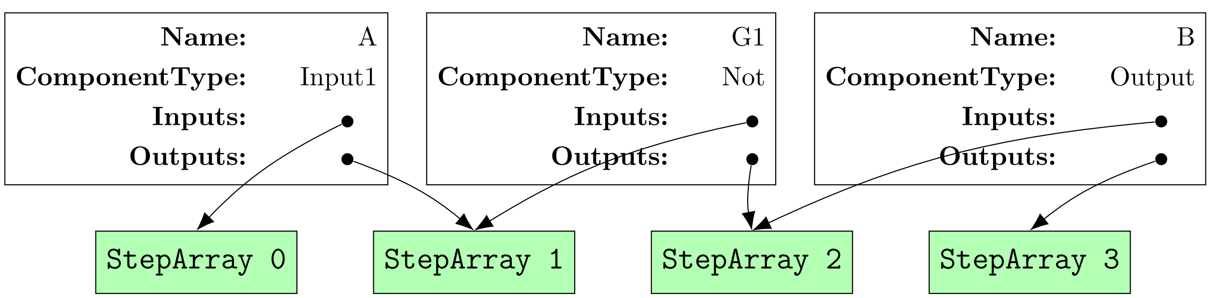


Figure 2.4: Illustration of how simulation data (`StepArray`) is shared between inputs and their corresponding outputs in Figure 2.3 in `FastSimulation`.

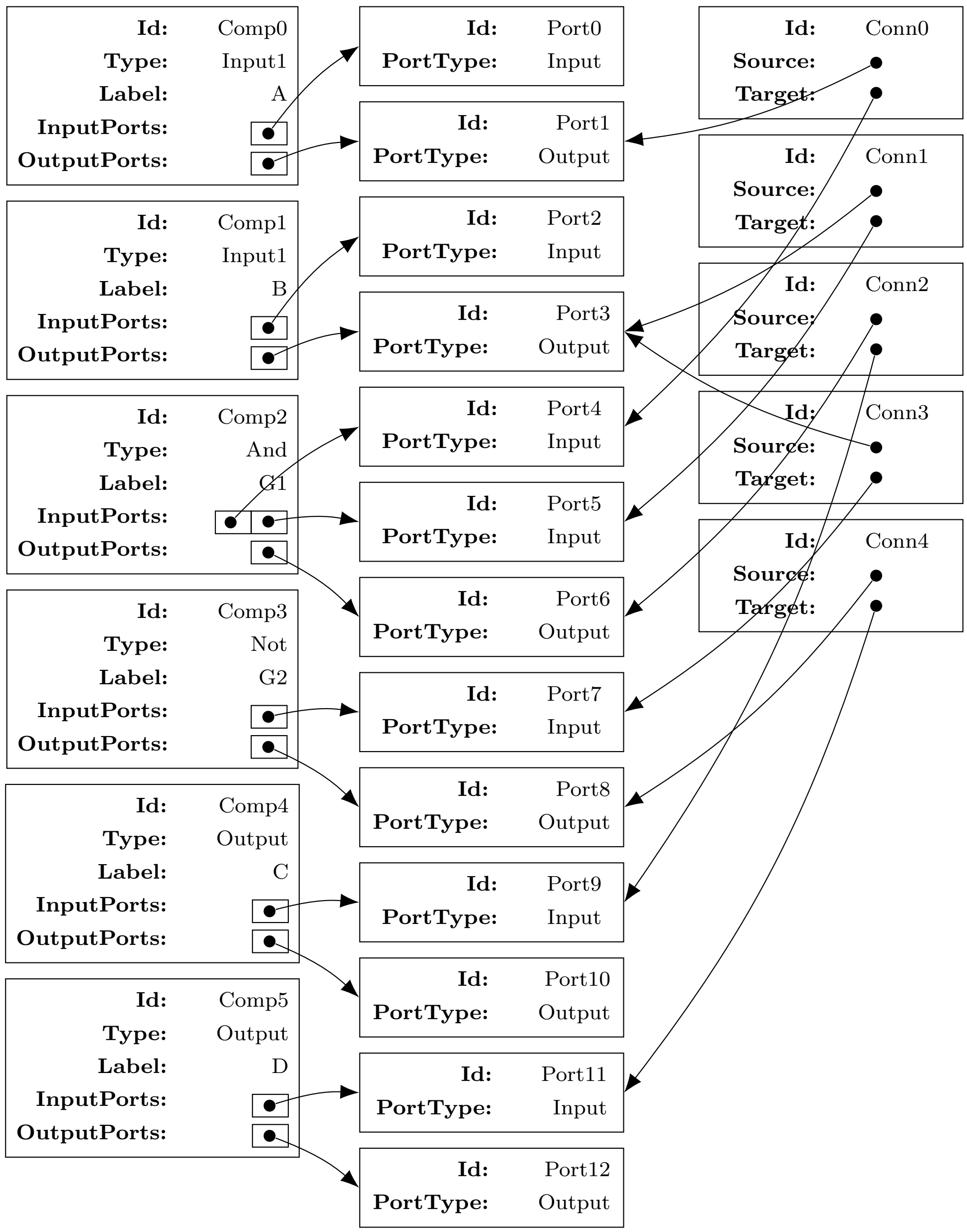


Figure 2.5: Illustration of how different components in Figure 2.2 are linked in `CanvasState`.

Chapter 3

Requirements Capture

This chapter focuses on addressing the issues present in the existing ISSIE simulator and outlining the requirements for the new simulator.

3.1 Problem of the Current Implementation

3.1.1 Insufficient Performance

The existing simulator in ISSIE (Baseline simulator) is already an performant simulator equipped with many optimising techniques as discussed in section 2.2 and section 2.3.But the existing simulator only evaluates about 2000 component per millisecond.

The insufficient performance primarily stems from the existing approach compromising efficiency to provide a universal solution, which is intended to support both symbolic and logic simulation. For example, ISSIE uses `FData` to represent simulation data which can either be a numeric value (`FastData`) or a symbolic value (`FastAlgExp`). This design choice makes the simulator use one more match statement in `FastReduce.fastReduce` for every component type to determine whether the component is evaluated under logic simulation or symbolic simulation. This is a trade-off between performance and generality. The abstraction of simulation data by `FData` also leads to the storage of simulation history using `array<FData>`, which is not memory efficient as this would be transpiled to JS Arrays instead of the more efficient typed array.

3.1.2 Technical Debts

Another problem of the existing simulator is the presence of several technical debts. Two notable examples include:

- `StepArray` was supposed to be a resizeable circular buffer. Thus `StepArray` would be initialised with a small length and can increase to a larger length when needed, once all the elements in the array are used, the array would be overwrite from the beginning. However, functions that were supposed to resize `StepArray` but are no longer used in the simulator. These deprecated functions are confusing to new contributors and should be removed.

- Output Widths of components are dynamically inferred from components with assigned widths by calling `FastReduce.fastReduce`. However, a more optimal approach is available through the use of `BusWidthInferer.inferConnectionsWidth` function which can statically determine all output widths of a given design. To address this technical debt, the simulator should be refactored to utilise the `BusWidthInferer.inferConnectionsWidth` function instead of relying on dynamic inference.

3.2 Requirements for the New Simulator

In order to be a high performance simulator, the new simulator must achieve improved time complexity (simulation speed) and space complexity (memory efficiency). The requirements for the new simulator is summarised as follows shown in Table 3.1.

The new simulator will primarily focus on enhancing the performance of logic simulation rather than symbolic simulation. This emphasis on logic simulation arises from the fact that the user experience in logic simulation is considerably more reliant on simulation speed compared to symbolic simulation. This can be attributed to the following reasons:

- Symbolic simulation is only applicable to combinational circuits, which tend to be smaller in scale, comprising fewer components. In contrast, large circuit designs often incorporate sequential components to prevent timing failures or implement state machines.

- Symbolic simulation only needs to be run one clock tick for each design, whereas logic simulation needs to be run for as many clock ticks as the user's interest demands. For instance, a user might use ISSIE to build and simulate a CPU for calculating the Fibonacci sequence. Consequently, the time spent on symbolic simulation is negligible when compared to the extensive duration required for logic simulation.

Table 3.1: Requirements for the new simulator.

Criteria	**Qualitative and Quantitative Requirement**
Time complexity improvements	By removing the overhead of symbolic simulation, the new logic-simulation only simulator is expected to be at least 10 times faster than the existing simulator.
Memory efficiency improvements	By employing typed arrays to store simulation data, the new simulator is expected to be at least 10 times more memory efficient than the existing simulator.
Reduced technical debts	All technical debts mentioned in subsection 3.1.2 should be addressed. No new technical debts should be introduced.
Compatibility	The new simulator should be compatible with existing functionality in ISSIE. Wave simulator, step simulator and truth table should be able to correctly use the new simulator.
Robustness and Reliability	The new simulator should be able to handle all the designs that the existing simulator can handle and produce the same results. Tests environment should be setup to allow the new simulator to be tested against the existing simulator.

Chapter 4

Implementation

Implementation of the new simulator is based on the original simulator in ISSIE and consists 4 stages. The first stage aims to clear technical debts of the existing simulator. Then in the second stage, the logic simulation part of the original simulator is extracted and form a logic simulation only simulator. The third stage focuses on using numeric arrays to store simulation data to achieve better performance. Finally, 3 extensions (memory compression, WASM and direct link across custom component boundary) are done in the fourth stage to explore different possibilities to further improve the performance of the ISSIE simulator.

The following sections will describe the implementation details of the new simulator and the optimisation process.

4.1 Stage 1: Clearing Technical Debts

Two types of technical debts are cleared in this stage, deprecated code as mentioned in subsection 3.1.2 and Hidden bugs. To enhance maintainability, the simulator code is formatted using Fantomas [15]. This practice takes inspiration from the V8 project, which enforces code formatting on all code submissions to ensure consistency throughout the codebase. The refactored simulator will be referred as Baseline (v0) simulator in this report.

4.1.1 Removal of Deprecated Code

As mentioned in subsection 3.1.2, functions that were used to resize `StepArray` are removed to make it clear that `StepArray` is just a circular buffer and not resizable.

In addition, `src/Simulator`, `src/Common` and all the files inside these two directories are removed as they are no longer used.

4.1.2 Bug Fixes

JavaScript is designed to not throw exceptions when error occurs as F# does. This silent error handling behaviour of JavaScript makes it hard to find bugs in the original simulator as we do not run or test the simulator on dotnet runtime. In this section, 2 bugs that are found when running the original simulator on dotnet runtime are described.

Infinite Indexing

JavaScript outputs `infinity` when a number is divided by zero as mentioned in subsection 2.1.3. This behaviour is different from the behaviour of F# which throws an exception when zero division happens. In the preparation phase of simulation, `FastReduce.fastReduce` is used to infer output widths from input widths. Originally, `FastReduce.fastReduce` is called with `maxArraySize = 0` and `clockTicks % maxArraySize` is used to index `StepArray`, so `infinity` is used to index `StepArray` resulting in `undefined`. This bug is fixed by calling `FastReduce.fastReduce` with `maxArraySize = 1`.

Incorrect Number of Ports for `RAM1`

`FastCreate.getPortNumbers` defines a search table to find the number of input and output ports for a given `SimulationComponent` by pattern matching its `ComponentType`. The original version would return 2 for `RAM1`, but should be 3 as `RAM1` has inputs for memory address, data in and write enable.

Circular Buffer Indexing

In the original simulator, there is an issue with how `stepSimulation` updates the outputs of global inputs. It currently calls `propagateInputsFromLastStep` to update outputs with `fs.ClockTick + 1` as the index, but in fact it should be `(fs.ClockTick + 1) % maxArraySize` since the outputs (represented by `StepArrays`) are circular buffer.

4.2 Stage 2: Logic Simulation Only Simulator

In this stage, a duplicate of the existing `FastReduce.fastReduce` is created and it is streamlined to only support logic simulation. This is a necessary step to improve the performance of logic simulation because it helps remove the overhead of `FData` as described in subsection 3.1.1 and section 2.2. By just using `FastData` for simulation data, the new simulator needs one less match statement to read simulation data, creates one less object when update simulation data. The refactored simulator will be referred as Version 1 (v1) simulator in this report.

4.2.1 New Representation for Simulation Data

The need for a type to hold simulation data raises from the fact that after the `InputLinks` and `Outputs` of each component now need to be able to hold both array of `FData` and array of `FastData`. Although `StepArray<'T>` is a generic type, it can not be used as `StepArray<FData>` and `StepArray<FastData>` at the same time.

An easy solution would be to use a DU type as shown in Listing 4. However, this approach introduces new overheads as it uses one more discriminated union to wrap the simulation data.

```
type NewStepArray =
    | FDataArray of StepArray<FData>
    | FastDataArray of StepArray<FastData>
```

Listing 4

Alternatively, `StepArray` can be changed from a generic record to a normal record, and have multiple fields each holds a different type of array. By this way, no extra overhead is introduced. Because `StepArray` is used in many other places in the simulator, the new extended version of `StepArray` is named `IOArray` to avoid breaking existing code. `IOArray` is a record type with two fields `FDataArray` and `FastDataArray` as shown in Listing 5.

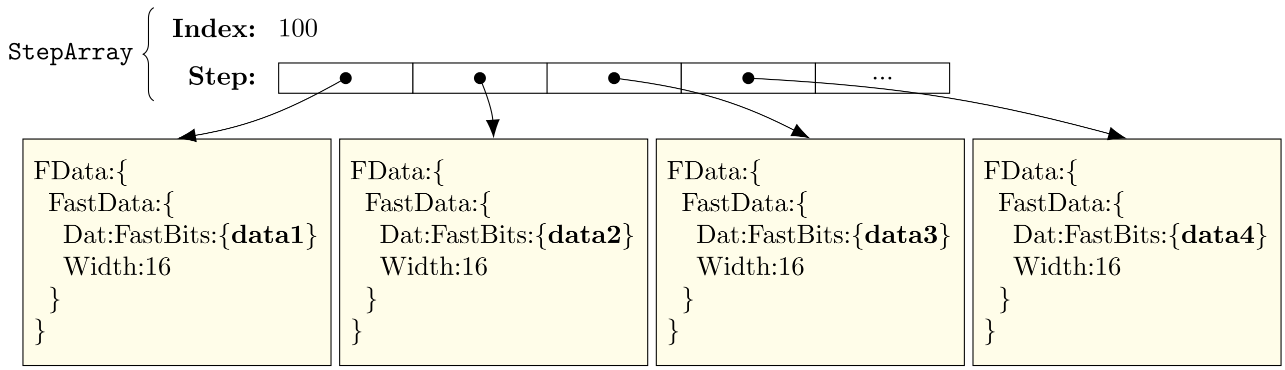
Figure 4.1a and Figure 4.1c illustrates how `IOArray` helps flatten the simulation data structure for logic simulation. When used in logic simulation, the `FDataStep` field is initialised to an empty array to reduce memory footprint.

Based on the memory snapshot of the v1 simulator as shown in Figure 1, `IOArray` that stores 550 data of width less than 33 bit takes 41984 bytes. This is a significant improvement over StepArray in v0, which occupies 68296 bytes as shown in Figure 2. In terms of memory efficiency for storing simulation data, v1 is able to store approximately 63% more data than v0 within the same size of heap memory.

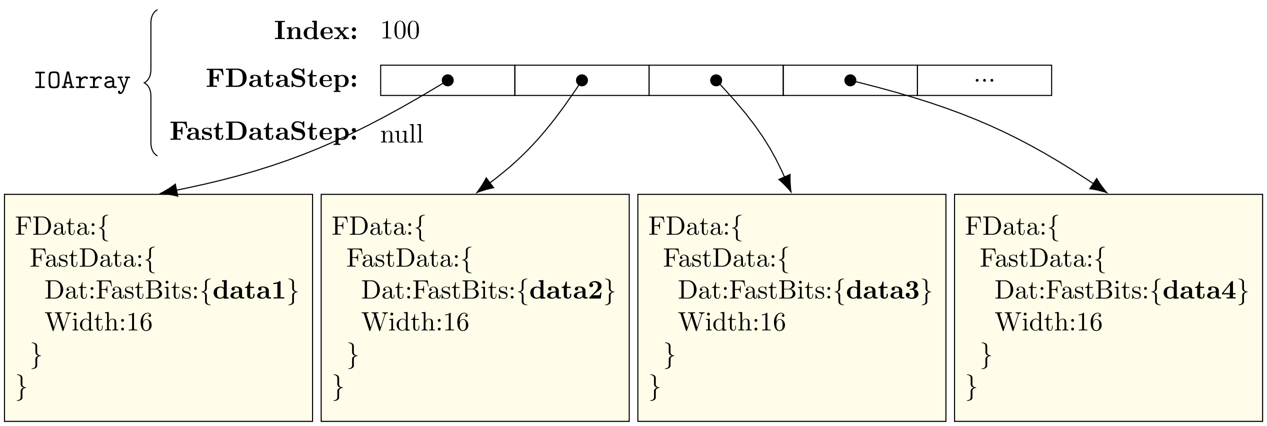
Another benefit of using `IOArray` is that it allows easy extension to support more types of array, just by introducing more fields. For example, it will be extend to support numeric arrays in section 4.3.

4.3 Stage 3: Numeric Array

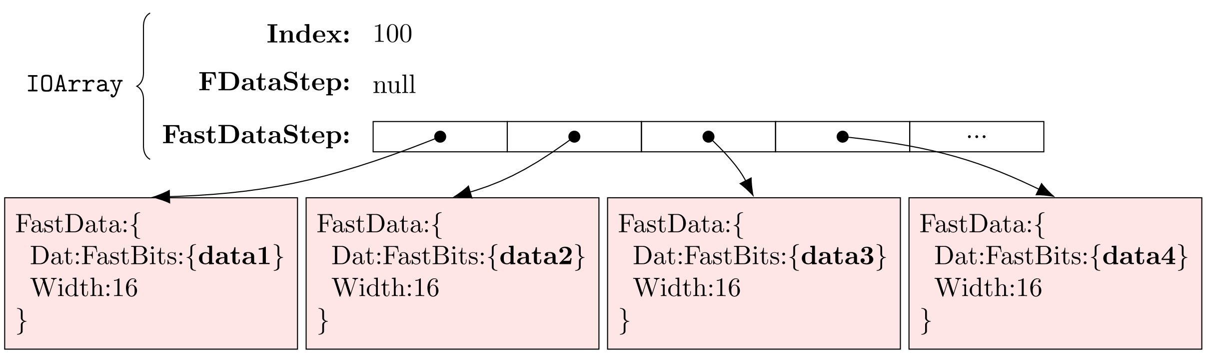
Section 4.2 removes the unnecessary `FData` wrapper from logic simulation data. However, the simulation data is still stored as arrays of objects in JavaScript `Array` objects. This



(a) Data structure of `StepArray<FData>`.



(b) Data structure of `IOArray` when used in truth table generation



(c) Data structure of `IOArray` when used in logic simulation.

Figure 4.1: Difference between `StepArray<FData>` and `IOArray`

```
type IOArray = {
    FDataStep: FData array;
    FastDataStep: FastData array;
    Index: int
}
```

Listing 5: Type declaration of IOArray.

section describes how the simulation data structure can be further optimised, removing all unnecessary wrapper and use numeric arrays to improve performance.

As briefly mentioned in section 2.1.3, JavaScript has two types of arrays, Array and Typed arrays. Typed arrays are more efficient for storing numeric values due to their storage as a contiguous block of memory called `ArrayBuffer`, allowing direct index-based access. On the other hand, Array is stored as a list of pointers to objects, requiring additional steps to access an element by following the pointer to the corresponding object.

The refactored simulator will be referred as Version 2 (v2) simulator in this report.

4.3.1 Static width inference

In order to store simulation data in numeric arrays, the width of each port must be known before `IOArrays` are created so that only the corresponding arrays are initialised to the required length. This requires static width inference as mentioned in subsection 3.1.2.

ISSIE is already equipped with a static width inferrer, `inferConnectionsWidth`, it is currently used by `checkConnectionsWidths` to verify the consistency of connection widths in the simulated design. As part of the modification, `checkConnectionsWidths` is updated to include the output of `inferConnectionsWidth`, specifically `ConnectionsWidth`, in its output. `ConnectionsWidth` is a map that associates connection IDs with their corresponding widths

4.3.2 Updated IOArray

To further optimise the storage of simulation data, the `IOArray` type is modified to store data in raw binary format using typed arrays. The updated `IOArray` type, as shown in Listing 6, replaces the `FDataStep` field with two typed arrays: `UInt32Step`, `BigIntStep`. These arrays are used to store simulation data with widths less than or equal to 32 and greater than 32, respectively. Figure 4.3 illustrates how data is stored in the new `IOArray` type.

Based on the memory snapshot of the v1 simulator as shown in Figure 3, `IOArray` that stores 550 data of width less than 33 bit takes 2520 bytes. This is a significant improvement over StepArray in v0, which occupies 41984 bytes as shown in Figure 2. In terms of memory efficiency for storing simulation data, 21 is able to store approximately 15.7 times more data than v1 within the same size of heap memory. Compare to v0, v2 is able to store approximately 26.1 times more data within the same size of heap memory.

```
type IOArray = {
    FDataStep: FData array;
    UInt32Step: uint32 array;
    BigIntStep: bigint array;
    Index: int
}
```

Listing 6: Type declaration of updated IOArray with typed arrays.

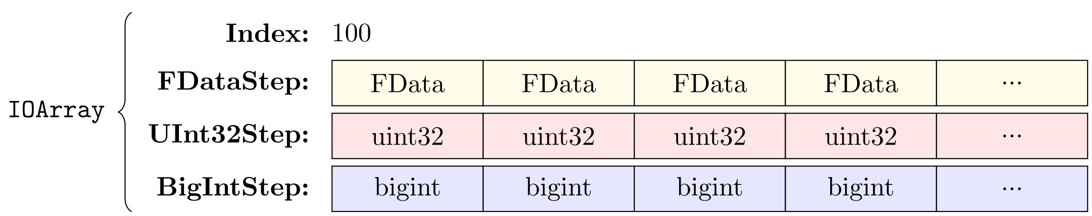
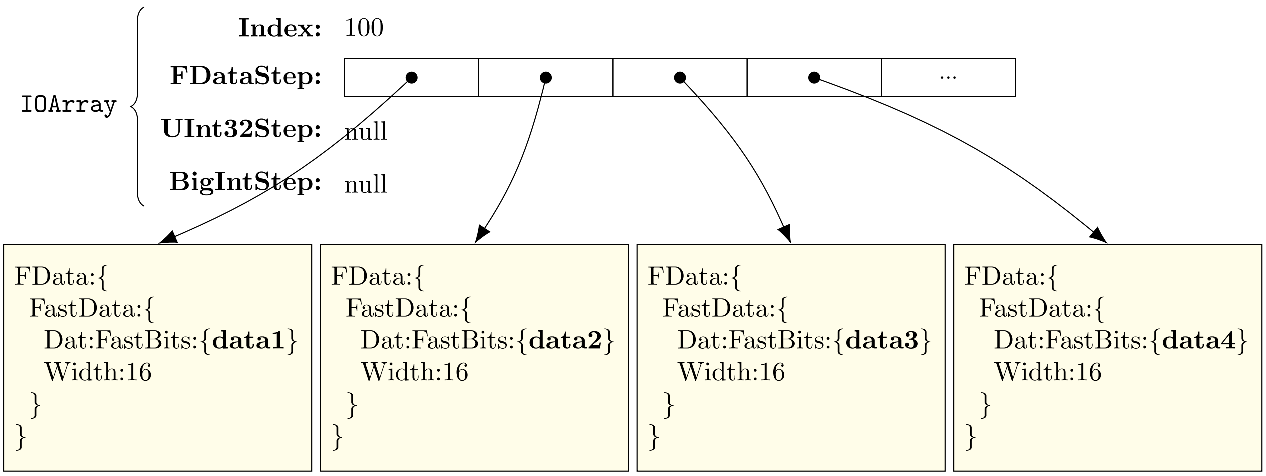
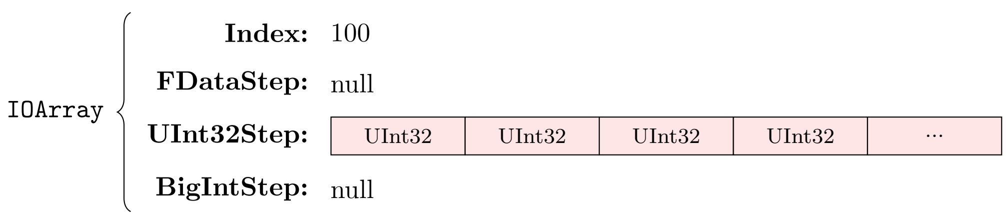


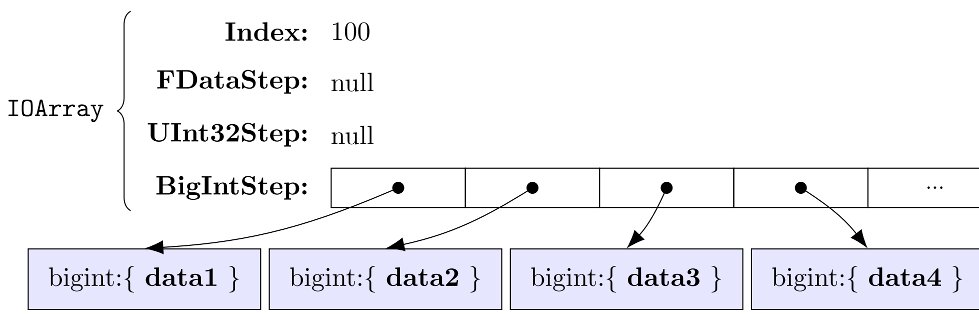
Figure 4.2: `IOArray` with typed arrays to store data in logic simulation.



(a) Data structure of `IOArray` when used in truth table generation.



(b) Data structure of `IOArray` when used for data of width ≤ 32 in logic simulation.



(c) Data structure of `IOArray` when used for data of width > 32 in logic simulation.

Figure 4.3: Difference between `StepArray<FData>` and `IOArray`

4.4 Stage 4: Extensions

4.4.1 Memory Compression

This extension focuses on further optimising heap memory usage in simulation, with compromise on simulation speed. The main idea is to store as many data as possible into each 32-bit word instead of storing one only data in every 32-bit word.

In section 4.3, IOArray is updated to store simulation data of any width into one of its three typed-array fields: UInt32Step, BigIntStep, FDataStep. UInt32Step is an array of 32-bit unsigned integers, like the other two buffers, each element of UInt32Step stores data of one port from one clock tick. This however means memory space is wasted for data with width less than 32 bits. For example, if a component has an output port with width 1 bit, then only 1 bit of each 32-bit unsigned integer in its UInt32Step is used, the other 31 bits are wasted. This is a huge waste of memory space, especially when the design has many components with IO ports of small width.

To address this issue, auxiliary methods of UInt32Step are modified to be capable of efficiently packing multiple small-width data into each 32-bit unsigned integer. This is achieved by using bit manipulation operations to store and retrieve data from UInt32Step.

For example, if a component has an output port with width 1 bit, then its data is stored from the least significant bit of each 32-bit unsigned integer in its UInt32Step. Each 32-bit unsigned integer can store data of that port for 32 clock ticks. In the most extreme case, i.e., when all ports are 1-bit, memory usage for simulation data can be reduced by 32 times.

This save in heap size however comes with a cost of performance as extra bit manipulations must be done to store and retrieve data from UInt32Step. This cost is measured in chapter 6.

The refactored simulator will be referred as Version 3 (v3) simulator in this report.

4.4.2 Compression methods

Two different ways to read and write data from UInt32Step have been considered.

1. Dense Compression.

   Data is stored in UInt32Step with all bits utilised. Data can span multiple 32-bit word, but complex bit manipulation is required. In the worse case, two read operations are required to read one data from UInt32Step, two write operations are required to write one data to UInt32Step.

   For example, if a port has width 12 bits, then its data for the first 2 clock ticks are stored in the first 32-bit word in UInt32Step. However, the data for the fifth clock tick will be stored in split between the first and second word, its first 8 bits are stored in the first word and the remaining 4 bits are stored in the second word.

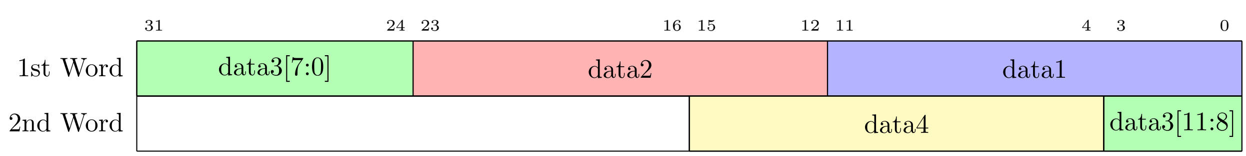


Figure 4.4: Layout of 12-bit data using dense compression.

2. Sparse Compression.

his compression method sacrifices bits at the end of each word in trade for less IO operations and simpler bit manipulation. Data is stored in UInt32Step without spanning multiple 32-bit word. In the worse case, one read operation is required to read one data from UInt32Step, one write operation is required to write one data to UInt32Step.

For example, if a port has width 12 bits, then its data for the first 2 clock ticks are stored in the first 32-bit word in UInt32Step. The data for the fifth clock tick will be stored in the second 32-bit word in UInt32Step, 8 bits in the end of the first word are unused.

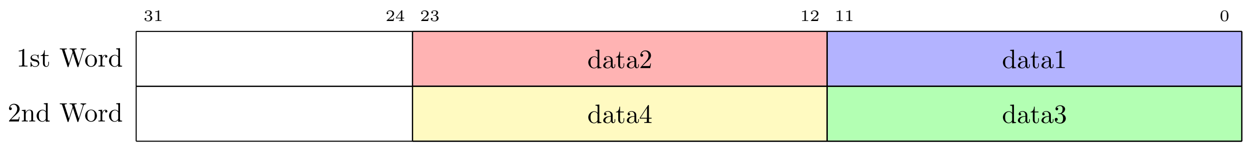


Figure 4.5: Layout of 12-bit data using sparse compression.

3. Numeric Array of Other Types.

   Instead of using UInt32Step (Uint32Array) to store all data with width less than 33, UInt16Buffer and UInt8Buffer can be added to store data with width less than 16 and 8 respectively using Uint16Array and Uint8Array. This method requires neither bit manipulation nor more IO operations, but it requires extra operations to decide which one of the buffers to use when read or write data.

   For example, if a port has width 7 bits, then the memory usage of its data stored in Uint16Array would be half of that stored in Uint32Array, its data stored in Uint8Array would be one fourth of that stored in Uint32Array.

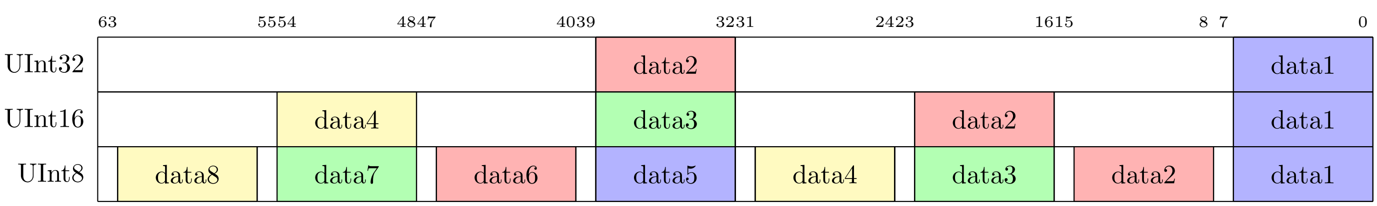


Figure 4.6: Layout of 7-bit data in Uint32Array, Uint16Array and Uint8Array.

4.4.3 New Read and Write Operations for UInt32Step

This section will describe the implementation of UInt32Step using sparse compression and discuss design choices been made to achieve the best performance.

The length of each UInt32Step (`numWords`) is calculated as follows using known data width `w` and maximum clock ticks of data kept in buffer `maxArraySize`.

$$\texttt{samplesPerWord} = \lfloor \frac{32}{\texttt{w}} \rfloor \tag{4.1}$$

$$\texttt{numWords} = \lceil \frac{\texttt{maxArraySize}}{\texttt{samplesPerWord}} \rceil \tag{4.2}$$

Read Data from UInt32Step

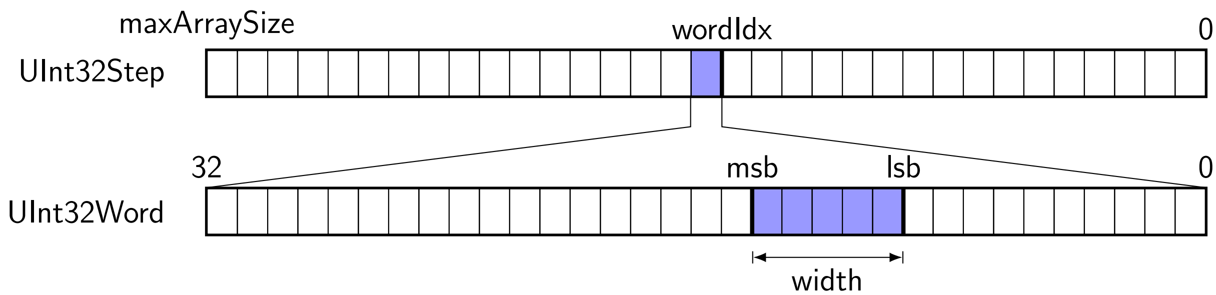


Figure 4.7: Illustration of how to extract an arbitrary data in a UInt32Step

To extract arbitrary data from UInt32Step, we first need to find out two indexes,

1. `arrayIdx`, index of the word that contains the desired data in the circular buffer.

$$\texttt{arrayIdx} = \texttt{clockTick} \bmod \texttt{maxArraySize} \tag{4.3}$$

2. `bitOffset`, bit offset of the data within the word, i.e. the least significant bit.

$$\texttt{wordIdx} = \lfloor \frac{\texttt{arrayIdx}}{\texttt{samplesPerWord}} \rfloor \tag{4.4}$$

$$\texttt{bitOffset} = (\texttt{arrayIdx} \bmod \texttt{samplesPerWord}) \times \texttt{w} \tag{4.5}$$

With `arrayIdx` and `bitOffset`, we can extract the desired data from UInt32Step as follows.

$$\texttt{word} = \texttt{UInt32Step[wordIdx]} \tag{4.6}$$

$$\texttt{data} = \texttt{word[lsb:lsb + w]} \tag{4.7}$$

JavaScript does not have native support to do bit slicing as in Equation 4.7, so we have to use shifts and integer divisions to extract `data` from `word`. Listing 7 and Listing 8 shows three different implementations of bit slicing in F# and their Fable transpiled JS code. The performance of these three implementations are measured by running them 1000000 times on a random UInt32 word. As the transpiled code suggested, `bitSlice1` is the fastest as it takes less operations which can also be processed taking advantage of parallelism, followed by `bitSlice3` and `bitSlice2`. This is probably because `bitSlice1` and `bitSlice3` both contain two parts that can be computed independently whose results are then merged to get the final result, while the computation in `bitSlice2` is sequential. `bitSlice1` is faster than `bitSlice3` because it has fewer operations overall.

```
// shift right and mask
let bitSlice1 bits lsb width =
    (bits >>> lsb) &&& (0xFFFFFFFFu >>> (32 - width))

// shift left and right
let bitSlice2 bits lsb width =
    (word <<< (32 - lsb - width)) >>> (32 - lsb)

// integer division
let bitSlice3 bits lsb width =
    (uint32 (bits / (1u <<< lsb))) % (1u <<< width)
```

Listing 7: Three different implementations of bit slicing in F#.

```
// shift right and mask
function bitSlice1(bits, lsb, width) {
    return ((bits >>> lsb) & (4294967295 >>> (32 - width))) >>> 0;
}

// shift left and right
function bitSlice2(bits, lsb, width) {
    return ((word << ((32 - lsb) - width)) >>> 0) >>> (32 - lsb);
}

// integer division
function bitSlice3(bits, lsb, width) {
    return (~~(bits / ((1 << lsb) >>> 0)) >>> 0) % ((1 << width) >>> 0);
}
```

Listing 8: JavaScript code transpiled by Fable from Listing 7.

Write Data to UInt32Step

The same algorithm as in section 4.4.3 can be used to find out where in the circular buffer to put `data`. Once `word` is found, we need to carefully merge `data` into `word` without change other bits in the word. This can be done in two ways,

1. Extract the bits in `word` before and after where `data` will be written to and then construct the new `word` from these old bits and `data`.

$$\texttt{newWord} = \{\texttt{word}[31 : \texttt{bitOffset} + \texttt{w}], \texttt{data}, \texttt{word}[\texttt{bitOffset} : 0]\} \tag{4.8}$$

2. Clear the bits in `word` that will be written to, shift `data` to the correct position and then merge `data` into `word` by bitwise and.

$$\texttt{mask} = \neg\left((\texttt{0xFFFFFFFFu} \ll (32 - \texttt{w})) \gg (32 - \texttt{bitOffset} - \texttt{w})\right) \tag{4.9}$$

$$\texttt{newWord} = (\texttt{word} \wedge \texttt{mask}) \vee (\texttt{data} \ll \texttt{bitOffset}) \tag{4.10}$$

4.4.4 WebAssembly

This extension focuses on leveraging WebAssembly (WASM) to enhance the ISSIE simulator's simulation speed and enable access to more heap memory.

As mentioned in subsection 2.1.5, latest version of Electron limits heap memory to 4GB and blocks use of off-heap memory for typed array. By utilising WASM with 64-bit memory indexes [26], ISSIE can address these memory limitations. It facilitates the simulation of more extensive and complex designs.

However, it is important to note that support for compiling F# to WebAssembly is still in the experimental phase. Both the official Blazor project and the third-party dotnet-wasi-sdk project do not currently offer mature support for F#-to-WASM compilation. The following sections delve into the details of the exploration process.

Compile to WASM using Dotnet-Wasi-Sdk [25]

This is a third party package that aims to compile dotnet code to WASM that is compliant with WebAssembly System Interface(WASI), this means the generated wasm binary can be executed not only in browser but in any WASI compliant runtime, e.g. Docker. Unfortunately, this package is still in early development stage and has a known issue [24] that it cannot call `Grow` or `GC` function to increase and manage unused memory. This issue causes runtime error when ISSIE simulator parses designs from JSON as shown in Listing 9.

```
[wasm_trace_logger] * Assertion at
↪  /home/yujie/workspace/dotnet-wasi-sdk/modules/runtime/src/mono/mono/metadata/sgen-stw.c:77,
↪  condition `info->client_info.stack_start >= info->client_info.info.stack_start_limit
↪  && info->client_info.stack_start < info->client_info.info.stack_end' not met
```

Listing 9: Error message printed when parsing design from JSON using ISSIE simulator compiled with dotnet-wasi-sdk.

Compile to WASM using Blazor [3]

Blazor offers two ways to run dotnet code in WASM: Ahead-of-time (AOT) compilation and interpretation. AOT compilation trims unused code from source code and optimises the rest of code using LLVM compiler before generate WASM binary. In interpretation mode, the dotnet runtime is compiled to WASM, the source code is still compiled to Dlls and executed in the WASMed dotnet runtime. Running AOT compiled WASM should be significantly faster than interpretation.

Both of these two options only support C# to WASM compilation. As a result, source code of ISSIE simulator must be wrapped in a C# project to be compiled to WASM by Blazor. This makes it difficult to read file system from WASM code, for example, ISSIE simulator need to read and validate Memory content before simulation starts. To work around this, all files needed for simulation are read in JavaScript driver code and passed to WASM as byte arrays as arguments of entry function.

AOT compilation is not feasible for ISSIE because Fable had problem compiling to WASM. Fable is not used directly by ISSIE but is a dependency of Thoth.Json which ISSIE uses to parse JSON files.

Chapter 5

Testing

Two tests are used to check the functional correctness of the new simulator automatically and manually. All versions of simulator developed in this project (including WASM simulator and version 3 simulator) pass manual test, only version 1 and version 2 simulators have been tested under automated test as it requires extra effort to setup the automated test for the specific data structure in each version.

Table 5.1: Test related files in `simulation_test/js`.

File	**Usage**
`index.js`	Defines `main` function that loads all design sheets a given path and calls `startCircuitSimulation` from transpiled JavaScript file to start simulation with specified parameters.
`utils.js`	Defines several helper functions that uses Node APIs to interact with local file system. These functions are designed to replace Electron APIs ISSIE simulator uses.
`runTest.sh`	Bash script that invokes `index.js` with Node to run simulation for a specific design sheet in a given ISSIE project. It can be configured to validate simulation result against a reference, or set Node and V8 options to collect various debugging information during the simulation process.
`runTests.sh`	A wrapper to `runTest.sh` that run simulation for all design in a given design and check whether all of them pass test.
`runTest.ps1`	PowerShell equivalent to `runTest.sh`
`generateRef.sh`	Bash script that invokes `index.js` with Node to run simulation for a specific design sheet in a given ISSIE project, and save all simulation data to a reference file in JSON format.

5.1 Automated Correctness Test

This tests is done in Node.js by comparing the simulation results of the new simulator with the results produced by the golden reference, the Baseline simulator. All test related files are located in the `simulation_test/js` directory. Table 5.1 lists all files in this directory and their usages.

To run the correctness test, reference outputs must be generated first by running `generateRef.sh` on `yw2919-simulator/baseline` branch for the desired designs. Following that, run `runTests.sh` on testing branch to check whether the new simulator produces identical results to the reference outputs.

This automatic test replies on helper functions defined in `utils.js` to extract simulation data according the the data structure used by the tested simulator. For example, the simulation data in version 1 simulator needs to be unwrapped from `FastData` and `FastBits` whereas the simulation data in version 2 simulator is already unwrapped and can be extracted by indexing directly.

5.2 Manual Correctness Test

Manual test is a complementary to the automated test to obtain quick feedback in development. It is done by simulating the EEP1 CPU design with a program to compute fibonacci sequence and checking whether the result for the 10th term equals 55.

Chapter 6

Results

This chapter presents the results of the performance benchmark of the new simulators. Table 6.1 lists the actions that have been taken to make sure the benchmark results are representative.

Table 6.1: Criteria considered to make benchmark results representative.

Criteria	**Actions Taken to Fulfil the Criteria**
Sample Selection	The benchmark result is collected by running the benchmark on two machines with x86 and ARM CPU, respectively.
Sample Size	The benchmark is repeated 10 times for each test case to reduce the effect of random noise, e.g. randomness in JIT compilation.
Realistic Workloads	This benchmark uses EEE1labs as the test suite [10], it includes a complete implementation of EEP1 CPU and 23 design sheets that covers various common digital logic circuits such as arithmetic logic unit, register file, decoder, etc.
Fair Comparison	Warm up runs are performed before the actual benchmark for JavaScript based benchmark to make sure potential optimisation has been applied by the JIT compiler. For speed benchmark, geometric mean is used to compute the benchmark score from simulation speed of each design sheet. This is because the simulation speed of each design sheet varies a lot, using arithmetic mean would skew the result towards the design sheets that are faster to simulate. Design sheets that contained fewer clocked components are faster to simulate because clocked components need extra reads to history data. For memory benchmark, garbage collection is forced before each measurement to eliminate the effect of garbage collection on memory usage.

6.1 Simulation Speed Benchmark

This benchmark is design to compare simulation speed of the new simulators with the Baseline simulator, measured in `tick` $\times$ `component/second`. All designs in EEE1labs are used as the test suite. This benchmark run each design for 2000 clock ticks and repeat 10 times, the final score is the geometric mean of the simulation speed of each design sheet.

The Electron benchmark is designed to be triggered by clicking the "Benchmark" button in the "View" menu of ISSIE app as shown in Figure 6.1, or by pressing "Ctrl + Shift + B". Benchmarks for other platforms need to be run in the terminal under their respective directories in `simulator_tests`, for example, `simulator_tests/dotnet` for dotnet benchmark.

Figure 6.2 shows the speed up from version 1 to version 3 relative to Baseline in different platforms. From Figure 6.2a and Figure 6.2b, the same trend is observed on both MacOS ARM and Windows x86 Electron, version 1 alone achieves more than $\times 9$ speed up to Baseline. Version 2 further increases the speed up to 15 on both testing machine. Version 3 has a drop in speed, as expected in subsection 4.4.1, from $\times 15$ to $\times 8$ as it takes more instructions to extract packed data from `UInt32Step`.

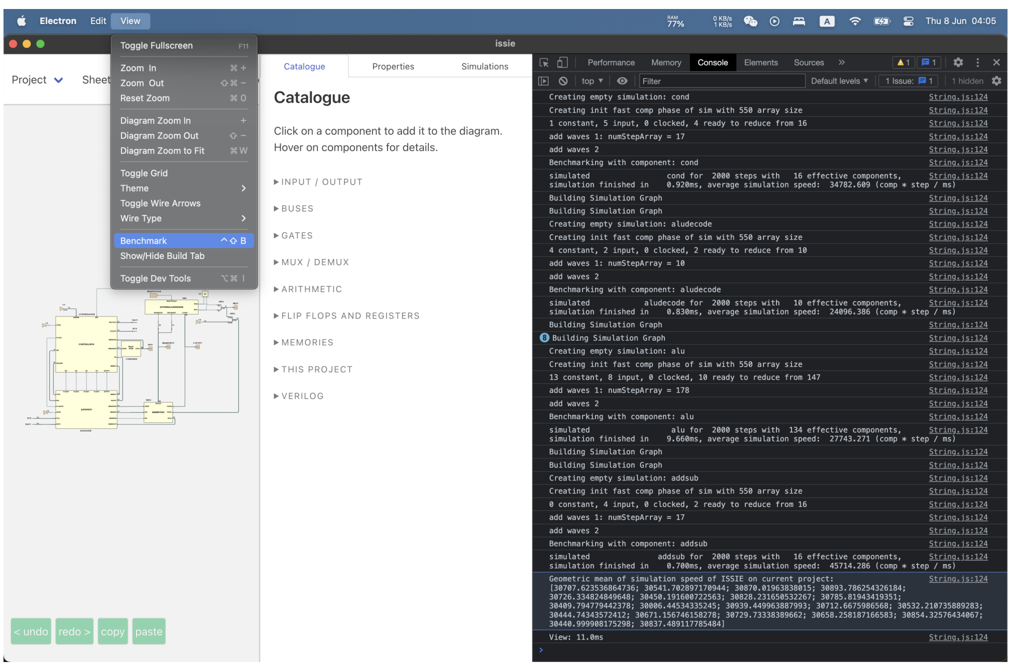
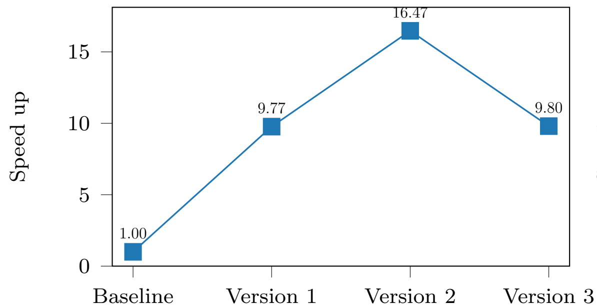
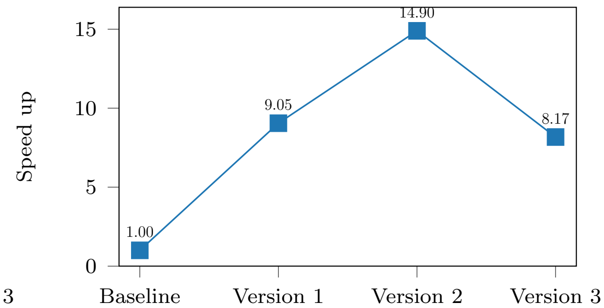


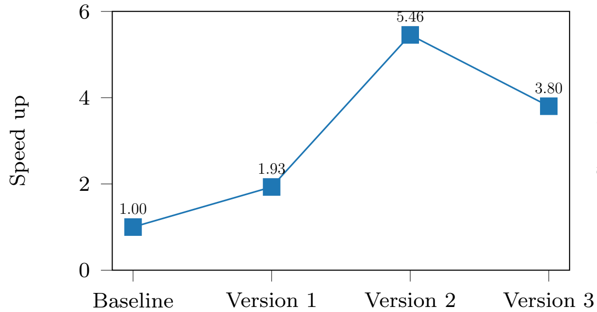
Figure 6.1: Screenshots of Electron simulation speed benchmark



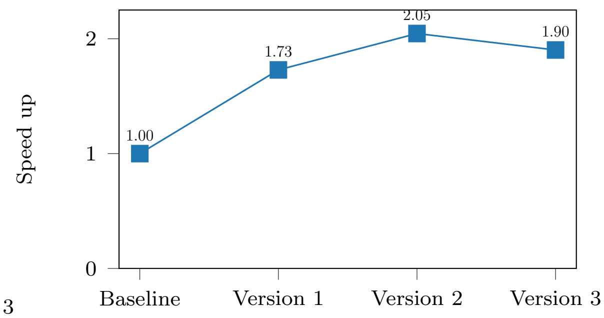
(a) Simulation speed improvement on MacOS ARM Electron, simulation speed of Baseline is 3127 tick × component / second.



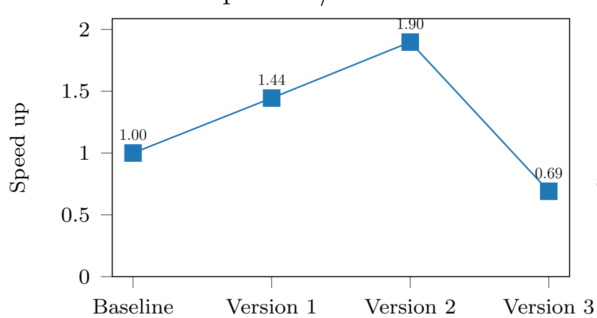
(b) Simulation speed improvement on Windows x86 Electron, simulation speed of Baseline is 2020 tick × component / second.



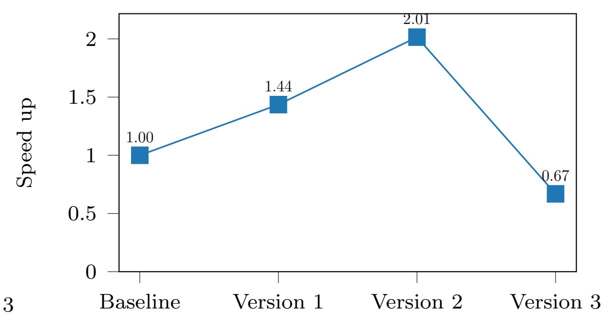
(c) Simulation speed improvement on MacOS ARM Dotnet, simulation speed of Baseline is 25754 tick × component / second.



(d) Simulation speed improvement on Windows x86 Dotnet, simulation speed of Baseline is 19118 tick × component / second.



(e) Simulation speed improvement on MacOS ARM WASM, simulation speed of Baseline is 3818 tick × component / second.



(f) Simulation speed improvement on Windows x86 WASM, simulation speed of Baseline is 2371 tick × component / second.

Figure 6.2: Speed up of simulation speed on different versions of the simulator compared to the Baseline.

6.2 Memory Usage Benchmark

Memory usage benchmark only focuses on the Electron version of the simulator as it is the one that is delivered by users. Dotnet version is not able to run under the current framework of ISSIE app, and WASM version is too immature to be used in production.

To compare the memory usage of the new simulators with the Baseline simulator, all versions of simulators simulates the same design, "eep1" from EEE1labs repository which consists of 538 components, for 20000 clock ticks with `maxArraySize` set to 10000. Memory snapshots are taken at different stages of simulation to better understand how heap memory is used. Statistics of these memory snapshots are collected from the Chrome DevTools Memory panel as shown in Figure 6.3 as recorded in Table 6.3. `maxArraySize = 10000` and `clockTicks = 20000` is chosen because this corresponds to the largest size of memory snapshot that my computer can handle without crashing DevTools.

The size of `FastSimulation` is recorded separately in Table 6.2 to better study the memory efficiency of `Array` and typed arrays.

It is worth noting that the memory usage of the Baseline and Version 1 simulators keeps increasing as the simulation continues, even after clock ticks reaching the maximum length of IOArray, 10000. This is because the simulator keeps allocating new memory to create new objects, e.g. `FastBits`, when updating IOArrays, which holds JavaScript `Array` objects. This is not the case for Version 2 and Version 3 simulators as they use typed arrays which have fixed size.

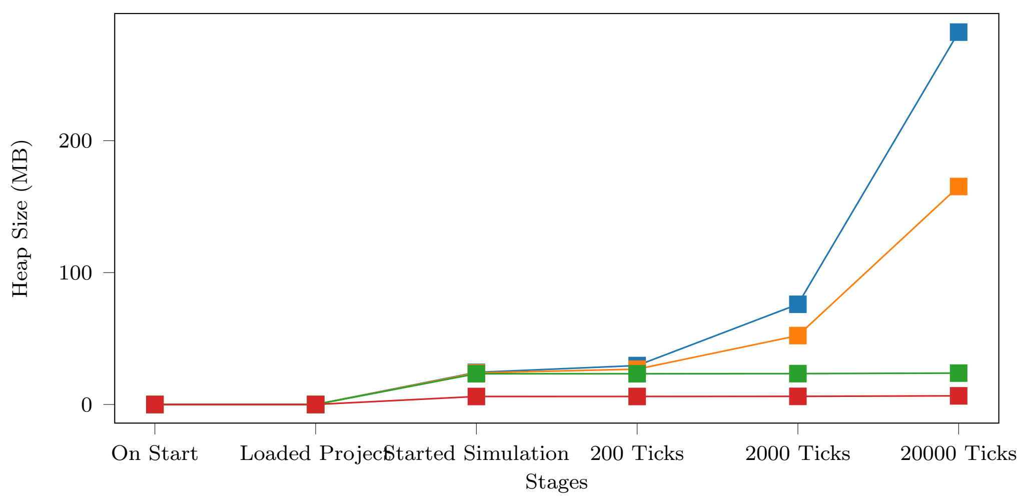
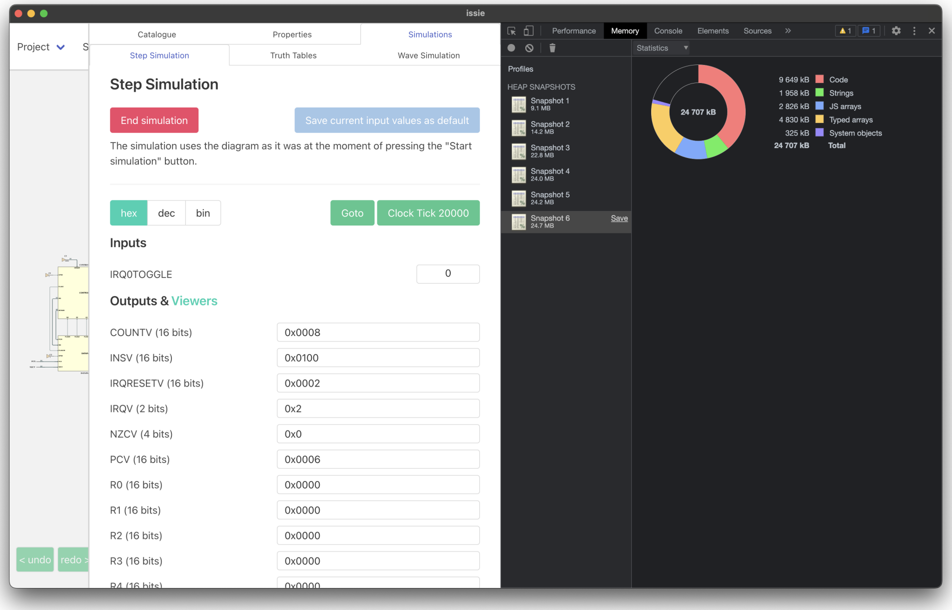
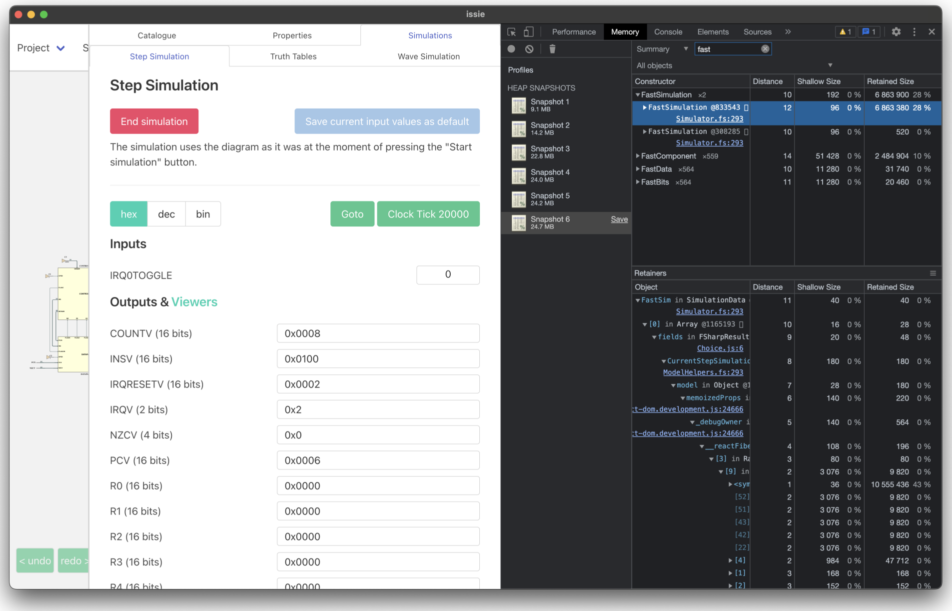


Figure 6.4: Heap size of FastSimulation (in MB) in different versions of ISSIE at different stages of step simulation. Blue line is Baseline simulator, yellow line is Version 1, green line is Version 2 and red line is Version 3.

From Figure 6.5, it is clear that the total heap usage is significantly affected by the heap



(a) Memory Statistics.



(b) Memory Summary with `FastSimulation` expanded.

Figure 6.3: Screenshots of Memory Panel in Chrome DevTools.

Stage \ Version	Baseline	Version 1	Version 2	Version 3
On Start	1812	1764	1764	1764
Loaded Project	1812	1764	1764	1764
Started Simulation	25541968	25133484	24430592	6355052
2000 Ticks	30943180	28081472	24439920	6364888
20000 Ticks	79653984	54773456	24526428	6451288
200000 Ticks	296027380	173299752	24938520	6863380

Table 6.2: Heap size of FastSimulation (in Bytes) in different versions of ISSIE simulator at different stages of running step simulation for eep1 [10].

usage of `FastSImulation`, it accounts for 92% of the total heap usage for Baseline simulator in benchmark conditions. Follows the blue line in Figure 6.5, the total heap usage is reduced by 41% from Baseline to Version 1, 86% from Version 1 to Version 2, 72% from Version 2 to Version 3 and 98% from Baseline to Version 3. The greatest drop in heap usage is from Version 1 to Version 2.

According to this result, Version 2 and Version 3 simulator can hold 11 and 42 times more simulation data than the Baseline simulator within the same memory limit under the benchmark conditions. It is also worth noting that the improve in memory efficiency observed here for Version 2 is lower than what is estimated in section 4.3, this is because `FastSimulation` does not only consist of `IOArrays`, for example, state of clocked components is still stored as `StepArray` in Version 2.

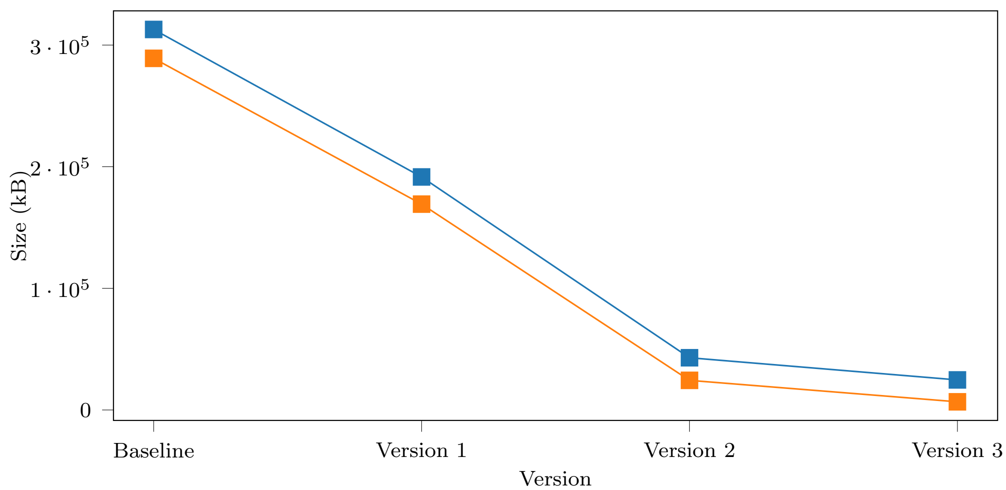


Figure 6.5: Heap usage of FastSimulation and total heap usage (in kB) in different versions of ISSIE after simulating eep1 for 20000 clockTicks, plotted using data from Table 6.3 and Table 6.2. Blue line is total heap usage and yellow line is heap usage of FastSimulation.

Stage \ Category	Code	Strings	JS arrays	Typed arrays	System objects	Total
On Start	4131	1397	99	697	321	9220
Loaded Project	6402	1921	479	1042	323	14352
Started Simulation	8607	1973	25073	1070	324	42601
200 Ticks	8301	1932	27696	967	325	47446
2000 Ticks	8554	1922	51161	962	325	96182
20000 Ticks	8821	1923	155551	969	326	312839

(a) Heap statistics of Baseline (in kB) at different stages of step simulation.

Stage \ Category	Code	Strings	JS arrays	Typed arrays	System objects	Total
On Start	4025	1395	98	697	321	9111
Loaded Project	5614	1895	480	1044	324	13496
Started Simulation	7720	1938	24811	1022	553	41358
200 Ticks	9422	1927	26007	967	532	46004
2000 Ticks	9468	1918	36649	975	529	72536
20000 Ticks	9696	1923	84120	1018	525	191572

(b) Heap statistics of Version 1 (in kB) at different stages of step simulation.

Stage \ Category	Code	Strings	JS arrays	Typed arrays	System objects	Total
On Start	4123	1400	99	695	321	9215
Loaded Project	6379	1912	473	984	323	14089
Started Simulation	8377	1955	2543	22881	324	40914
2000 Ticks	9241	1962	2571	22887	324	42219
20000 Ticks	9556	1961	2624	22892	324	42480
200000 Ticks	9815	1954	2829	22891	325	42935

(c) Heap statistics of Version 2 (in kB) at different stages of step simulation.

Stage \ Category	Code	Strings	JS arrays	Typed arrays	System objects	Total
On Start	3990	1400	98	697	321	9084
Loaded Project	6223	1926	476	1042	323	14184
Started Simulation	8328	1954	2536	4812	324	22788
2000 Ticks	9265	1961	2567	4818	324	23990
20000 Ticks	9388	1962	2620	4822	324	24230
200000 Ticks	9649	1958	2826	4830	325	24707

(d) Heap statistics of Version 3 (in kB) at different stages of step simulation.

Table 6.3: Heap statistics of different versions of ISSIE simulator at different stages of running step simulation for eep1 [10].

```
[27878:0x138008000]      232 ms: Scavenge 2.0 (2.3) -> 1.3 (3.3) MB, 0.3 / 0.0 ms  (average mu = 1.000, current mu = 1.000) allocation failure;
[27878:0x138008000]      259 ms: Scavenge 2.7 (3.8) -> 2.3 (4.3) MB, 0.5 / 0.0 ms  (average mu = 1.000, current mu = 1.000) allocation failure;
[27878:0x138008000]      346 ms: Scavenge 4.4 (5.6) -> 4.0 (6.1) MB, 0.9 / 0.0 ms  (average mu = 1.000, current mu = 1.000) allocation failure;
```

Listing 10

6.3 More Profilings

Benchmarks in section 6.1 only shows the overall speed up of the new simulator. In order to exploit what exactly causes the speed up, two extra profiling runs were performed, one measures the influence of the garbage collector on simulation speed, the other measures the average number of Ignition bytecode [17] `fastReduce` takes to process a component for one tick.

6.3.1 Time Spent on Garbage Collection (GC)

The total time spent on garbage collection is obtained by first running benchmark (`runTest.sh`) with `--gc-stat` flag, which prints out brief summary of each garbage collection operation, as shown in Listing 10. Then, these statistics are feed to a python script which uses regex to extract time consumption of each operation according to the format specified in `gc-tracer.cc` [30] and sums them up. Each test runs simulation of EEP1 CPU for 10^6 clock ticks with `maxArraySize = 1e5`.

Figure 6.6 shows the time spent on garbage collection compared to the total time consumption to run the simulation. As expected, Baseline simulator spends the most time on garbage collection, followed by Version 1. Time spent on GC is negligible for Version 2 and Version 3. This can be further explained by the patterns shown in Figure 4, Figure 5, Figure 6 and Figure 7. This series of figures show that Baseline simulator and Version 1 simulator not only spent the most time on GC, but also call GC the most times. GC is most frequently called in Baseline simulator as it quickly runs out of memory and have to call GC to free up memory.

Reducing the need for GC contributes to 31% of the speed up from Baseline to Version 1, and 30% of the speed up from Version 1 to Version 2.

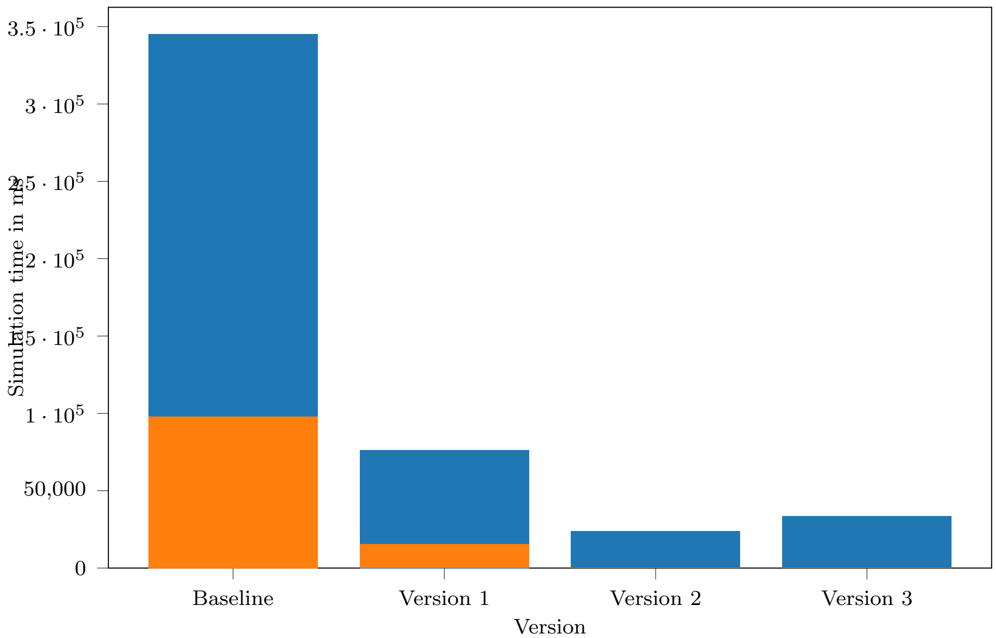


Figure 6.6: Total time spent on garbage collection and total time spent (measured with `--gc-stat` flag) to simulate 1E6 clock ticks with `maxArraySize` = 1E5.

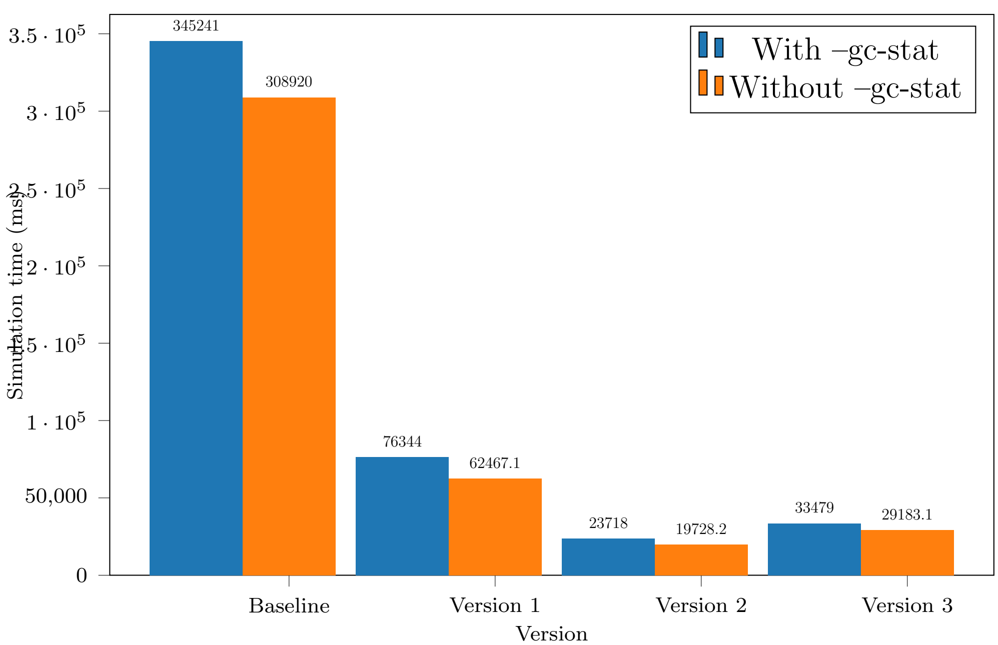


Figure 6.7: Total time spent to simulate 1E6 clock ticks with `maxArraySize` = 1E5, measured with and without `--gc-stat` flag

6.3.2 Average Number of Ignition Bytecode

To measure the average number of Ignition bytecode generated to process any component in `fastReduce`, the `--print-bytecode` is used together with `--print-bytecode-filter=fastReduce` flag to print out bytecode generated by V8 for `fastReduce` only. The output is then feed to a python script which use regex to count bytecode of different categories and use `Jump*` bytecode to recognise branches in the control flow. These information is used to build a tree with each node being a `Jump*` bytecode and each branch being one paragraph of branchless bytecode. The average number of bytecode that `fastReduce` takes to process one component is then calculated by average the number of bytecodes in path in this tree that connects its roots to one of its terminal. The result is shown in Figure 6.8.

From Figure 6.8, it can be seen that average number of bytecodes across all categories decreases from the Baseline version to Version 2, with the exception of arithmetic bytecodes. This observation can be attributed to the fact that arithmetic bytecodes solely process the extracted simulation data and remain unaffected by the change in the data structure that encapsulates the simulation data.

Conversely, the flatter data structure used in new simulators leads to a decrease in the average number of jump, load, and store bytecodes. This is because the new typed data structure necessitates fewer of these operations for processing. For instance, the removal of `FData` from Baseline to Version 1 implies less use of `LdaNamedProperty` to read field of `FData`.

On average, there is a 38% reduction in the number of executed bytecodes for processing one component from the Baseline version to Version 1. Additionally, from Version 1 to Version 2, there is a further 27% reduction in the number of executed bytecodes. These findings highlight the bytecode optimisation achieved through the improvements made in the data structure and processing algorithms in the new simulators.

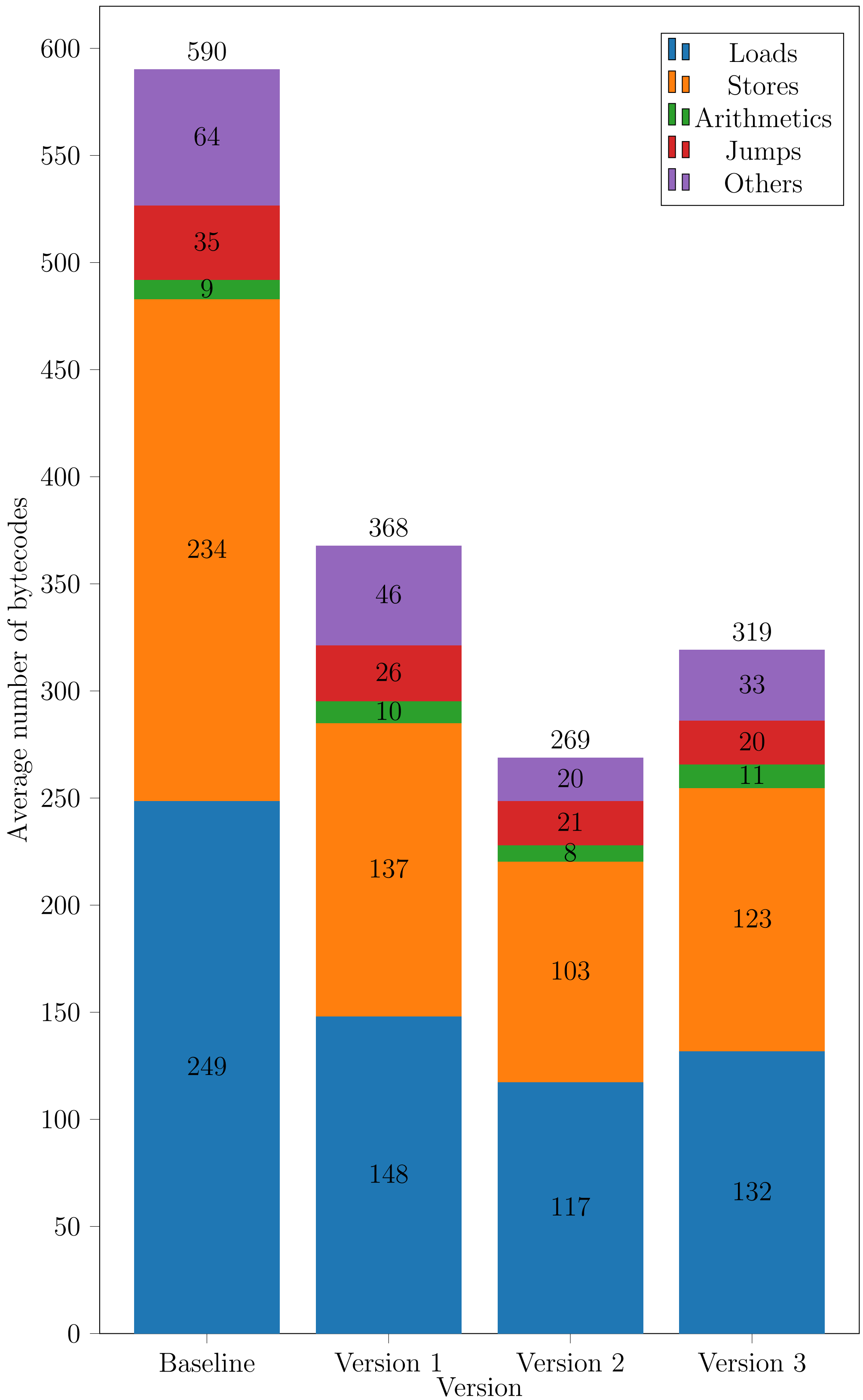


Figure 6.8: Average number of Ignition byte code that `FastReduce.fastReduce` takes to simulate one component for one tick.

Chapter 7

Evaluation

This chapter evaluates the project based on the requirements set in chapter 3.

Despite the outstanding memory efficiency of Version 3 simulator, Version 2 is chosen as the final delivered simulator as it is more balanced in terms of memory and speed performance. The trade off of half speed for 5 time more simulation data is considered not worth for the education purpose of ISSIE simulator. However, the Version 3 simulator is still valuable as a proof of what would be possible if the simulator is to be used to simulate extreme large scale system or extreme long time period. Concepts from version 3 can be brought to version 2 in the future to improve its memory efficiency if needed.

Overall, the new simulator (Version 2) satisfies all the requirements set in chapter 3. It is capable to simulate at 15 times the speed and store 11 times more simulation data in the same heap size compared to the Baseline. It also cleaned up the code base to make it more maintainable and fixed several hidden bugs that were present in the Baseline.

Chapter 8

Conclusion and Future Work

Optimising a digital circuit simulator to achieve over $\times 10$ improvement is a challenging and rewarding task. This project has successfully achieved all the preset requirements and also exploit the source of these performance gain. During the process, tooling and techniques are also developed as explored that can be helpful for future development of ISSIE simulator.

As explored in the extensions, there are still many ways to improve the ISSIE simulator. The following are some of the ideas that could be further explored in the future:

Although dotnet ecosystem has note yet fully support WebAssembly, it is a very promising platform for the future of ISSIE simulator. WASM with 64-bit memory indexes could allow simulation at much larger scale. And it could also be used to develop browser base ISSIE simulator.

Another unoptimised area is the boundaries between CustomComponents and their hosting component. In the current implementation, these boundaries are bridged by IO components on both ends. But this could be improved by allowing direct access between the actually logic of the two components.

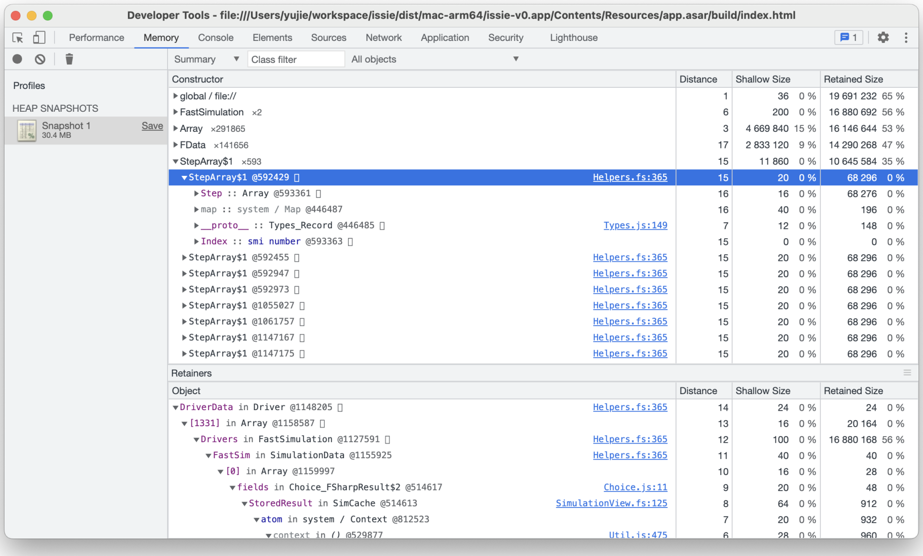


Figure 1: Memory snapshot of v0 simulator with StepArray highlighted.

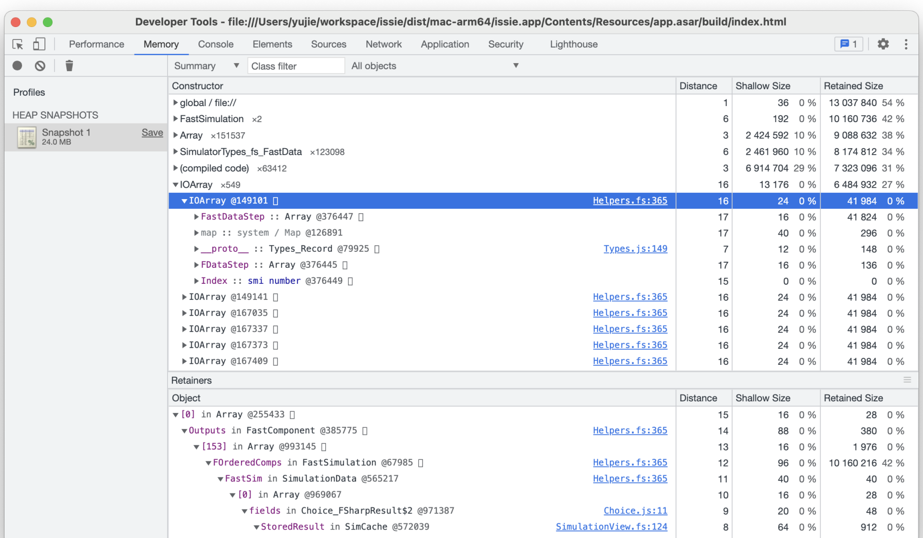


Figure 2: Memory snapshot of v1 simulator with IOArray highlighted.

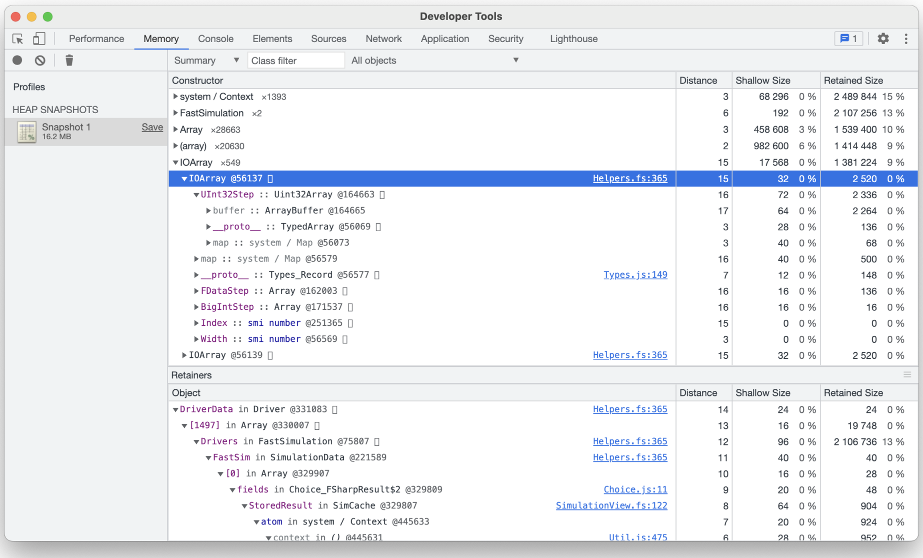


Figure 3: Memory snapshot of v2 simulator with IOArray highlighted.

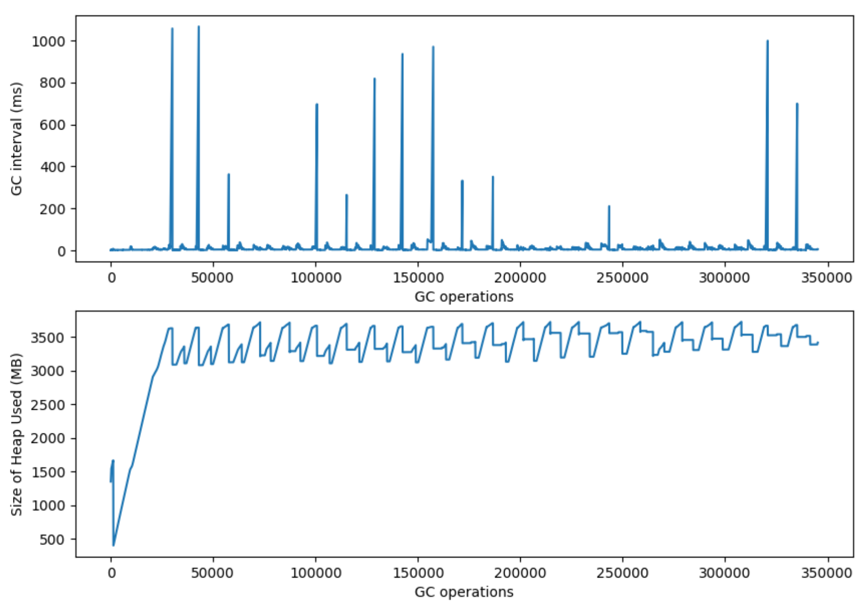


Figure 4: Duration of each GC operation (on top) and change of size of used Heap memory (on bottom) during profiling task and in subsection 6.3.1 for Baseline simulator.

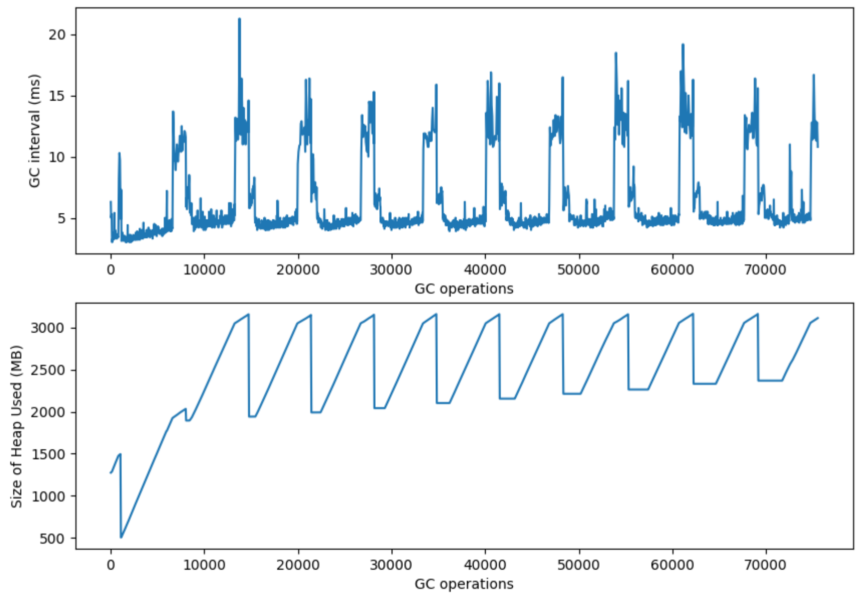


Figure 5: Duration of each GC operation (on top) and change of size of used Heap memory (on bottom) during profiling task and in subsection 6.3.1 for Version 1 simulator.

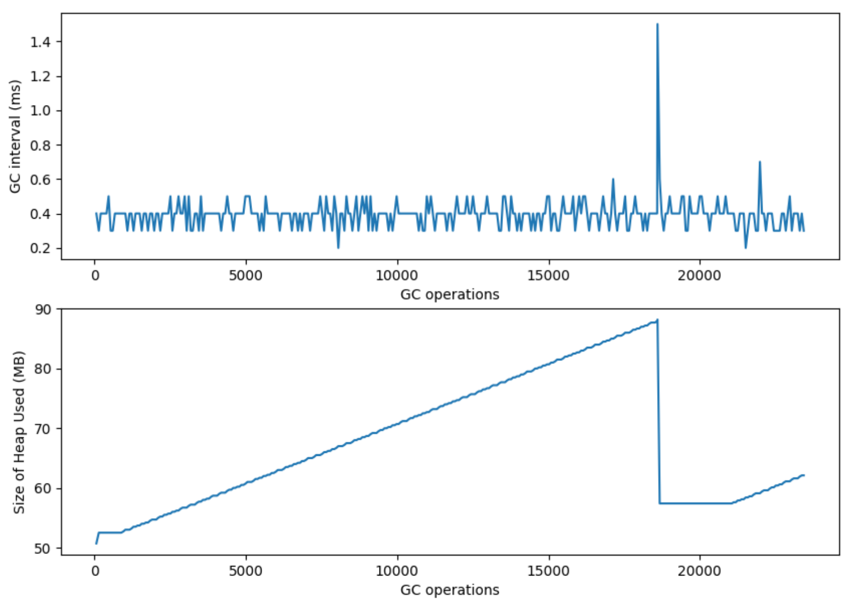


Figure 6: Duration of each GC operation (on top) and change of size of used Heap memory (on bottom) during profiling task and in subsection 6.3.1 for Version 2 simulator.

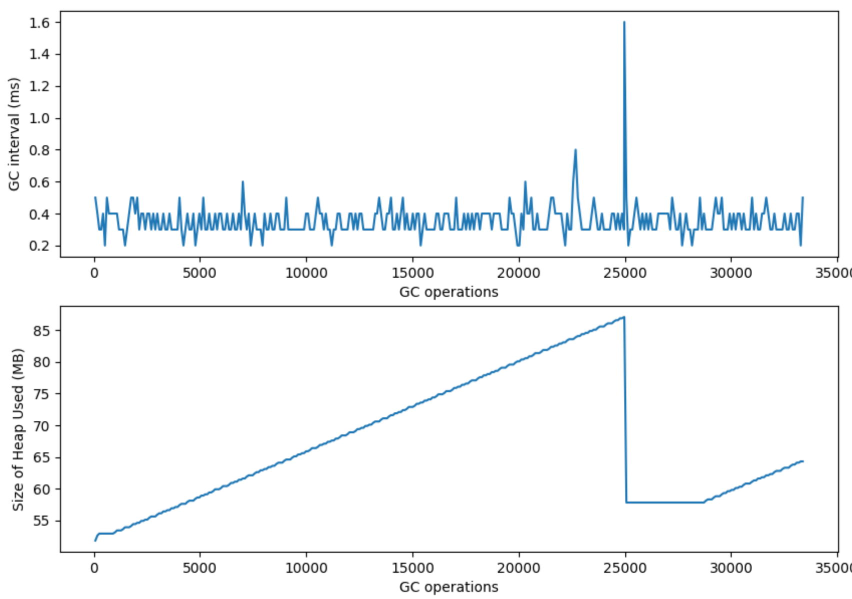


Figure 7: Duration of each GC operation (on top) and change of size of used Heap memory (on bottom) during profiling task and in subsection 6.3.1 for Version 3 simulator.

Bibliography

[1] *Adding BigInts to V8 · V8*. URL: https://v8.dev/blog/bigint (visited on 06/18/2023).

[2] *BigInt - JavaScript — MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt (visited on 01/26/2023).

[3] *Blazor — Build client web apps with C# — .NET*. Microsoft. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor (visited on 06/20/2023).

[4] *Build cross-platform desktop apps with JavaScript, HTML, and CSS — Electron*. URL: https://electronjs.org/ (visited on 06/19/2023).

[5] *Build software better, together*. GitHub. URL: https://github.com (visited on 06/21/2023).

[6] cartermp. *Discriminated Unions - F#*. Sept. 15, 2021. URL: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions (visited on 06/19/2023).

[7] Tom Clarke. *Issie - an Interactive Schematic Simulator with Integrated Editor*. original-date: 2020-06-27T16:00:12Z. Jan. 31, 2023.

[8] *ECMAScript*. original-date: 2014-04-09T19:04:33Z. June 18, 2023.

[9] *ECMAScript® 2024 Language Specification*. URL: https://tc39.es/ecma262/#sec-numeric-types-number-divide (visited on 06/18/2023).

[10] Ed. *EEE1labs*. original-date: 2022-10-05T12:58:54Z. June 17, 2023.

[11] *Elm - delightful language for reliable web applications*. URL: https://elm-lang.org/ (visited on 01/24/2023).

[12] *Elmish · Elmish*. URL: https://elmish.github.io/elmish/ (visited on 01/24/2023).

[13] *F# Software Foundation*. URL: https://fsharp.org/ (visited on 06/19/2023).

[14] *Fable · JavaScript you can be proud of!* URL: https://fable.io/ (visited on 12/26/2022).

[15] *Fantomas*. URL: https://fsprojects.github.io/fantomas/ (visited on 06/20/2023).

[16] *Git*. URL: https://git-scm.com/ (visited on 06/21/2023).

[17] Franziska Hinkelmann. *Understanding V8's Bytecode*. DailyJS. Dec. 19, 2017. URL: https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775 (visited on 01/26/2023).

[18] *JavaScript — MDN*. May 1, 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 06/19/2023).

[19] *JavaScript engine fundamentals: Shapes and Inline Caches · Benedikt Meurer*. URL: https://benediktmeurer.de/2018/06/14/javascript-engine-fundamentals-shapes-and-inline-caches/ (visited on 02/06/2023).

[20] *Numbers and dates - JavaScript — MDN*. June 1, 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates (visited on 06/18/2023).

[21] *Pointer Compression in V8 · V8*. URL: https://v8.dev/blog/pointer-compression (visited on 06/18/2023).

[22] *React – A JavaScript library for building user interfaces*. URL: https://reactjs.org/ (visited on 02/06/2023).

[23] *refs/heads/10.8-lkgr - v8/v8 - Git at Google*. URL: https://chromium.googlesource.com/v8/v8/+/refs/heads/10.8-lkgr (visited on 06/19/2023).

[24] *Running out of memory quite quickly and seemingly not garbage collection · Issue #11 · SteveSandersonMS/dotnet-wasi-sdk*. URL: https://github.com/SteveSandersonMS/dotnet-wasi-sdk/issues/11 (visited on 05/24/2023).

[25] Steve Sanderson. *Experimental WASI SDK for .NET Core*. original-date: 2022-03-25T15:57:08Z. June 20, 2023.

[26] *Timeline update? · Issue #36 · WebAssembly/memory64*. URL: https://github.com/WebAssembly/memory64/issues/36 (visited on 06/18/2023).

[27] *V8 JavaScript engine*. URL: https://v8.dev/ (visited on 01/26/2023).

[28] *V8 Sandbox - Address Space*. Google Docs. URL: https://docs.google.com/document/d/1PM4Zqmlt8ac5O8UNQfY7fOsem-6MhbsB-vjFI-9XK6w/edit?usp=sharing&usp=embed_facebook (visited on 06/14/2023).

[29] *V8 Sandbox - Sandboxed Pointers*. Google Docs. URL: https://docs.google.com/document/d/1HSap8-J3HcrZvT7-5NsbYWcjfc0BVoops5TDHZNsnko/edit?usp=fatal-error-notifier-usp&usp=embed_facebook (visited on 06/14/2023).

[30] *v8/src/heap/gc-tracer.cc at main · v8/v8 · GitHub*. URL: https://github.com/v8/v8/blob/main/src/heap/gc-tracer.cc (visited on 06/21/2023).

[31] *WebAssembly*. URL: https://webassembly.org/ (visited on 06/19/2023).

[32] *webpack*. webpack. URL: https://webpack.js.org/ (visited on 06/19/2023).