LIÈGE université Library

https://lib.uliege.be

https://matheo.uliege.be

Master thesis : Designing a web-Based Beta-Assembly Simulator Using Flutter

Auteur : Laperche, Quentin
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2025-2026
URI/URL : http://hdl.handle.net/2268.2/26102

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

UNIVERSITE DE LIEGE
FACULTE DES SCIENCES APPLIQUEES

Designing a web-Based Beta-Assembly Simulator Using Flutter

LAPERCHE Quentin

Thesis presented to obtain the degree of :
Master in Computer Science

Thesis supervisor:
MATHY Laurent

Academic year: **2025 - 2026**

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Prof. Laurent Mathy, Gaulthier Gain, and Benoît Knott, for allowing me to work on this project and for their guidance and valuable advice throughout its development.

I would also like to thank all the professors of the Faculty of Applied Sciences for sharing their knowledge and contributing to my academic development. This master's thesis, as well as my entire university journey, has been an enriching experience. Through a wide variety of courses and projects, I have had the opportunity to explore numerous fields of computer science, allowing me to acquire valuable technical skills, broaden my perspective, and develop both academically and personally.

Finally, I would like to thank everyone who contributed, directly or indirectly, to making these years of study a rewarding and memorable experience.

Contents

List of Figures

Introduction

Computer architecture is a fundamental topic in computer science education. Understanding how processors execute instructions, manipulate memory, and control program flow is essential for students studying low-level systems. However, many of these concepts may remain abstract when presented solely through theoretical lectures.

Educational simulators provide an effective way to bridge the gap between theory and practice by allowing students to observe the internal behavior of a processor during program execution. One such tool is BSim, a simulator of the Beta processor developed as part of the Computation Structures course at MIT. BSim allows users to write assembly programs, execute them step by step, and visualize the internal state of a processor, making the study of computer architecture more accessible and interactive.

Although BSim remains a valuable educational tool, it was originally developed using web technologies and software components that are difficult to maintain and extend. Furthermore, integrating new features or adapting the application to modern development practices can be challenging.

This project consists of a complete reimplementation of BSim using Flutter and Dart.

The work involved several major challenges. The Beta processor had to be simulated accurately despite the difference between the parallel behavior of hardware circuits and the sequential nature of software execution. In addition, a complete assembler capable of translating Beta assembly code into machine instructions had to be developed. Finally, a graphical interface reproducing the workflow of the original application was implemented for web deployment.

The remainder of this thesis presents the design choices, implementation details, and validation of the resulting system. The preparation and architecture of the project are first described, followed by the implementation of the Beta machine, the assembly parser, and the graphical user interface. The thesis concludes with an evaluation of the results and a discussion of possible future improvements.

Summary

This thesis presents the design and implementation of a web-based reimplementation of BSim, an educational simulator of the Beta processor architecture used for teaching computer architecture concepts and the Von Neumann model. The project was developed using Flutter and Dart with the objective of reproducing the main functionalities of the original application while providing a modern, modular, and maintainable architecture compatible with web platforms.

The work primarily focused on two major aspects: the simulation of the Beta processor and the implementation of a custom assembly parsing system. One of the main technical challenges consisted of reproducing the parallel behavior of hardware circuits within a sequential software execution model. To address this problem, a dependency analysis of the Beta machine data flow was performed in order to derive a valid sequential execution order preserving the original processor behavior.

The project also includes the development of a custom two-pass assembler for the uasm language. Instead of relying on external parsing libraries, a dedicated parser and Abstract Syntax Tree (AST) structure were implemented to maintain strict control over the supported instruction set and improve compatibility with the simulator. The assembler supports expressions, macros, labels, and symbol resolution through a two-pass assembly process.

In addition, a graphical interface reproducing the workflow of the original BSim application was developed using Flutter Web. The interface integrates a code editor, execution controls, and real-time visualization of the processor state. Particular attention was given to modularity, maintainability, and synchronization between the simulator and the graphical components.

The final application successfully reproduces the core functionalities of the original BSim software and demonstrates the feasibility of implementing an educational processor simulator entirely as a modern web application.

Objective

The primary objective of this project was to develop a complete web-based reimplementation of the BSim educational simulator while preserving the behavior of the original application.

To achieve this objective, several functional requirements were identified:

- implement a faithful simulation of the Beta processor architecture,
- support the execution of Beta machine code instruction by instruction,
- provide a visual representation of the processor state,
- develop an assembler capable of translating uasm source code into executable machine code,
- reproduce the main workflow of the original BSim application,

Chapter 1

State of the art

1.1 Presentation of BSim

BSim is a software that simulates a Beta machine. The Beta machine follows the Von Neumann architecture model, meaning it is a minimalist computer architecture.
The beta machine consists of five main components:

- The **program counter** points to the current instruction stored in memory.
- The **control logic** retrieves the flags and parameters from the current instruction.
- The **registers** store thirty-two 32-bit integers. They provide faster data access than memory.
- The **ALU** (Arithmetic and Logic Unit) performs arithmetic and logical operations on integers or directly on bits.
- The **memory** stores data. It can retrieve and save data, but it is slower than the registers.

(see Figure 1.1).

The software has two main visual components:

- A code editor to write assembly instructions
- A visualisation of the beta machine to show the effect of the assembly code in the beta machine

And three buttons to be able to switch views:

- **Editor** only shows the code editor
- **Split** shows both view together
- **Simulation** only shows the visualization of the beta machine.

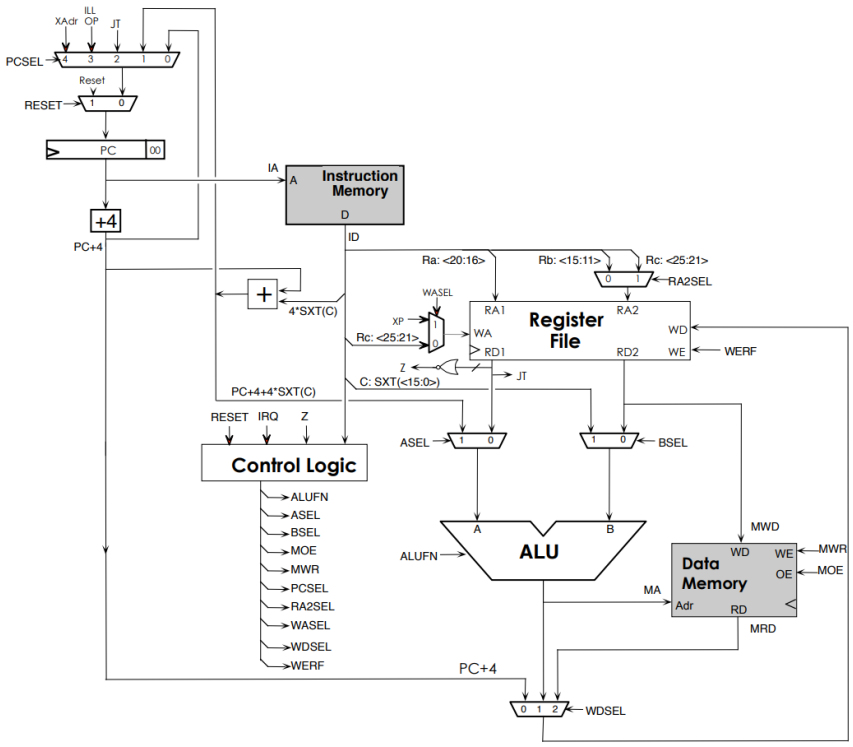

Figure 1.1: Beta machine

The code editor consists of a main editable file, a second read-only file containing the instruction set for the Beta machine. The read-only file contains both variables for registers, and macros corresponding to the beta instruction set. And a button the translate the assembly into binary instructions. Basically all assembly is converted into numbers or nothing; those numbers are all 32-bit integers that can be read by my beta machine.

The visualization component contains a toolbar with buttons to execute the binary, a button to switch the view, another button to display cache statistics, and the currently displayed view. The visualization component provides two different views:

- **Schematic View**: displays useful information in separate panels, such as the registers, the disassembled instructions, the current stack, and the memory.
- **Programmer View**: displays a diagram of the Beta machine with the binary values shown at each important stage of execution.

The toolbar allows the user to execute instructions one step at a time, step backward through the execution, reset the simulation, or run the program continuously until completion (see Figure 1.2).

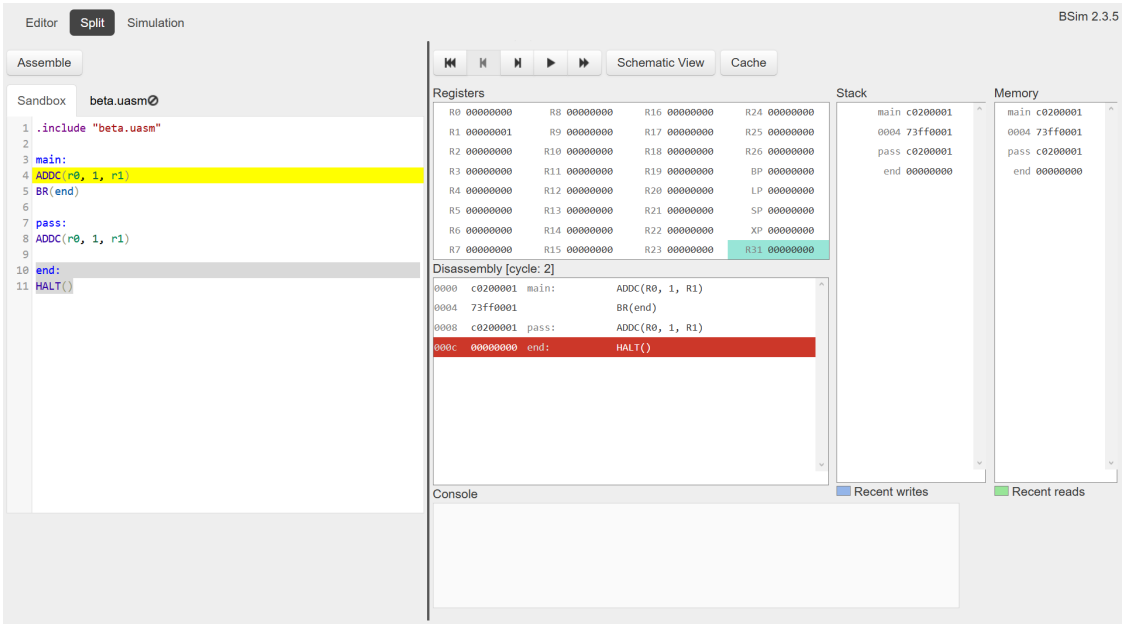

Figure 1.2: BSim

1.2 Formalization of the Assembly Language

The assembly language supported by the simulator is intentionally limited and is derived from the uasm language used by BSim. The language is built around three principal directives:

- The `.include` directive allows the content of another source file to be imported into the current compilation unit. It expects a string representing the path of the file to include. This mechanism is primarily used to provide predefined constants and macro definitions.
- The `.macro` directive is used to define parameterized code expansions. A macro definition consists of a name, an optional list of parameters, and one or more expressions forming the macro body. During assembly, macro invocations are expanded into the corresponding expressions before machine code generation.
- The `.align` directive specifies an alignment constraint. It indicates that the following generated value should be aligned on a specified number of bytes in memory. This is particularly useful for instructions that must begin at addresses respecting the Beta architecture's alignment requirements.

Aside from these directives, the language is composed of expressions involving numbers, arithmetic operators, and labels.

Numeric values and arithmetic expressions behave similarly to those found in most programming languages. The supported operators allow the construction of more complex expressions that are evaluated during assembly.

Labels provide symbolic names for memory locations. Conceptually, a label can be viewed as a variable whose value corresponds to the address of the instruction or data item that

follows its declaration. Labels are primarily used to reference code locations without requiring explicit memory addresses.

The language also defines a special symbol ".", commonly referred to as the location counter. Its value corresponds to the current memory address during assembly. It therefore behaves similarly to a label automatically representing the current position in the generated program.

BSim provides two assembly source files by default. The first file, `main.uasm`, is intended to be edited by the user. The second file, `beta.uasm`, is read-only and contains a collection of predefined macros implementing the Beta instruction set.

A representative excerpt of `beta.uasm` is shown below:

```
.macro SHORT(x) x%0x100 (x>>8)%0x100
.macro WORD(x) SHORT(x) SHORT(x >> 16)

.macro betaop(OP,RA,RB,RC) {
    .align 4
    WORD((OP<<26)+((RC%0x20)<<21)+((RA%0x20)<<16)+((RB%0x20)<<11))
}

.macro betaopc(OP,RA,CC,RC) {
    .align 4
    WORD((OP<<26)+((RC%0x20)<<21)+((RA%0x20)<<16)+(CC%0x10000))
}

.macro ADD(RA, RB, RC) betaop(0x20,RA,RB,RC)
.macro ADDC(RA, C, RC) betaopc(0x30,RA,C,RC)
```

This excerpt illustrates how assembly instructions are ultimately constructed from a hierarchy of macro expansions. The macros `ADD` and `ADDC`, which appear to the programmer as native instructions, are in fact defined in terms of the more generic macros `betaop` and `betaopc`. These macros encode the instruction fields according to the Beta instruction format and then rely on the `WORD` and `SHORT` macros to generate the corresponding binary representation.

Figure 1.3 presents a concrete example of this translation process for the `ADD` instruction. The figure illustrates the successive macro expansions leading to the final binary representation of the instruction. For completeness, it also shows the hexadecimal representation obtained after applying the endianness conversion performed by the `swapEndian` function, which will be discussed in Section 3.1.3.

This design demonstrates that, despite the apparent complexity of the instruction set, every assembly construct ultimately reduces to the generation of numeric values representing machine instructions. Consequently, from the assembler's perspective, the language can be viewed as a combination of directives, arithmetic expressions, labels, and macro expansions.

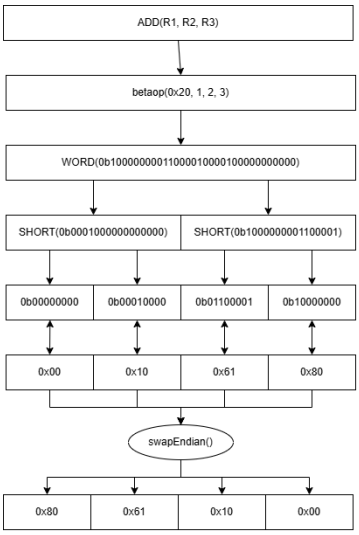

Figure 1.3: ADD instruction to binary

1.3 Other Existing Simulators

Several other assembly and processor simulators already exist. Only browser-based simulators were considered, since this project also targets a web environment. Simulators requiring a local installation were not evaluated.

- `https://web.edumips.org` EduMIPS64 is an educational MIPS64 simulator featuring an integrated assembly editor, execution controls, and pipeline visualization.
- `https://sonic-rv.ics.jku.at/example/new-ics-edu-rv32i-sc/hello_world` SonicRV is a RISC-V simulator that provides interactive visualizations of the processor state and instruction execution.
- `https://github.com/mortbopet/ripes` Ripes is a RISC-V processor simulator with an integrated code editor, pipeline visualization, cache simulation, and debugging tools.
- `https://vonsim.github.io` VonSim is an educational simulator combining assembly programming, processor visualization, and debugging features.
- `https://p3js.goncalomb.com` P3JS is a browser-based assembly simulator featuring an integrated editor, an assembler, execution controls, and memory/register visualization.

Most existing simulators focus on realistic architectures such as MIPS or RISC-V and provide debugging and visualization tools. However, the level of visualization and pedagogical focus differs between simulators. BSim distinguishes itself by emphasizing the visualization of the Beta architecture and its internal execution process.

Chapter 2

Preparation of the Work

Before starting the implementation phase, several preparation and planning tasks were carried out in order to organize the development process and define the overall structure of the application.

The preparation phase included:

- the creation of a task management system with multiple progression states (*to-do*, *doing*, *done*, *to test*, *to correct*, and *validated*),
- the design of a global widget architecture diagram,
- and the establishment of a development schedule with intermediate deadlines.

The task management system was primarily used to monitor development progress and identify unfinished or unstable features. Separating implementation, testing, and validation stages made it easier to track the state of each component throughout the project.

A structural diagram of the application's widgets was also created before implementation in order to better organize interactions between graphical components and anticipate dependencies between interface elements.

The project was developed using the Flutter framework. Flutter was selected because it allows the creation of reactive cross-platform applications while maintaining a unified codebase for web deployment. The program uses different packages `flutter_monaco` to have a VS Code-like editor, and `google_fonts` to have a custom monospace font.

The application architecture was divided into two principal folders: `models` and `views`.

The `views` folder contains all graphical components of the application and therefore represents the frontend layer of the project. This folder was further divided into specialized subfolders:

- the `layout` folder, containing widgets responsible for interface organization and workspace management,

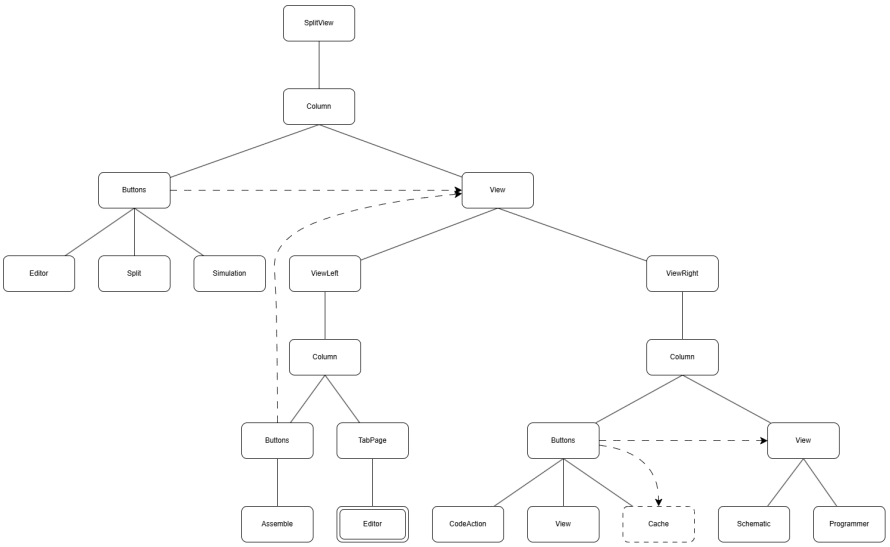

Figure 2.1: Widget structure

- and the `custom_paint` folder, containing the `CustomPainter` implementation used to draw the Beta machine schematic, as well as helper functions for rendering recurring graphical components such as wires, multiplexers, and logic elements.

The `models` folder contains the application's data structures and simulation logic, corresponding to the backend layer of the project. This folder notably contains the Beta machine simulation, the assembly parser, and a dedicated `AST` subfolder containing all classes related to the Abstract Syntax Tree representation of the assembly language.

This separation between frontend and backend components was introduced to improve modularity and maintainability. Grouping files according to their responsibilities also simplified navigation inside the project and reduced coupling between unrelated components.

Chapter 3

Implementation

3.1 Implementing the beta machine

The first component implemented during the project was the Beta machine simulator itself. This part was written entirely in pure Dart and represents the core of the application.

3.1.1 Making the execution sequential

The main challenge of this implementation was reproducing the behavior of electronic circuits using sequential software execution. In hardware, signals propagate through wires almost simultaneously, meaning that many components compute their outputs in parallel during the same clock cycle. In contrast, software instructions are executed sequentially and therefore require an explicit execution order.

However, not all signals are independent. Some values depend on the results of previous computations, meaning that the execution order must respect the dependencies between signals. Consequently, some hardware components had to be decomposed into multiple sequential software steps.

One important example is the register file. In the Beta architecture, several operations involving the registers conceptually occur simultaneously:

- the computation of `RD1`, which depends on the current instruction,
- the computation of `RD2`, which depends on the control logic,
- and the write operation to register `RC`, which depends on the ALU result.

Although these operations occur during the same hardware cycle, they cannot be directly reproduced sequentially without first analyzing their dependencies.

To determine which components needed to be separated and in which order computations should occur, a dependency graph of the Beta machine was created.

The first step consisted of modeling the flow of binary signals through the Beta machine. In Figure 3.1, rectangular nodes represent hardware components while oval nodes represent

inputs, outputs, and intermediate signals exchanged between components.

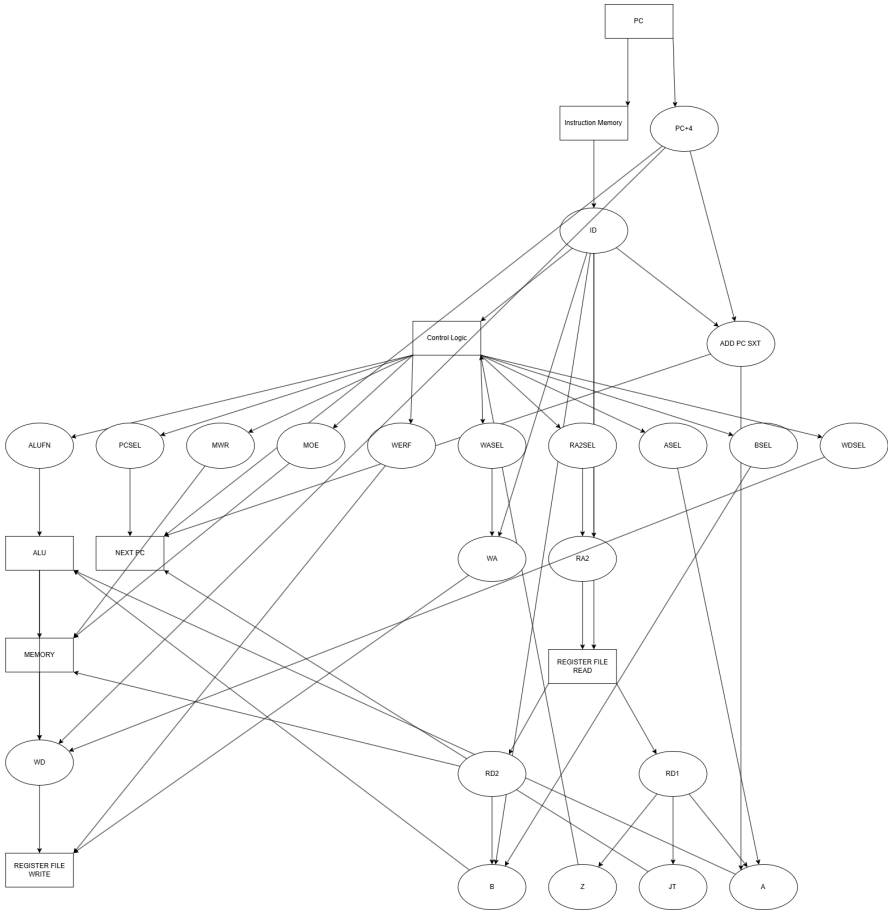

Figure 3.1: Beta signal flow

Once the dependency graph was constructed, a simplified version was generated using a process equivalent to a transitive reduction on a directed acyclic graph.

This simplification removes redundant dependency edges while preserving the reachability relationships between nodes.

More precisely, for each dependency edge (A, C):

- if there exists an intermediate node B such that:
  - A depends on B,
  - and B depends on C,
- then the direct dependency between A and C is redundant and can be removed.

For example, if a computation A depends on B, and B itself depends on C, then computing A already implicitly requires the computation of C. The direct edge between A and C

therefore does not provide additional ordering information.

This process significantly reduced the complexity of the graph and produced a clearer unidirectional dependency flow suitable for sequential execution.

The resulting simplified graph is shown in Figure 3.2.

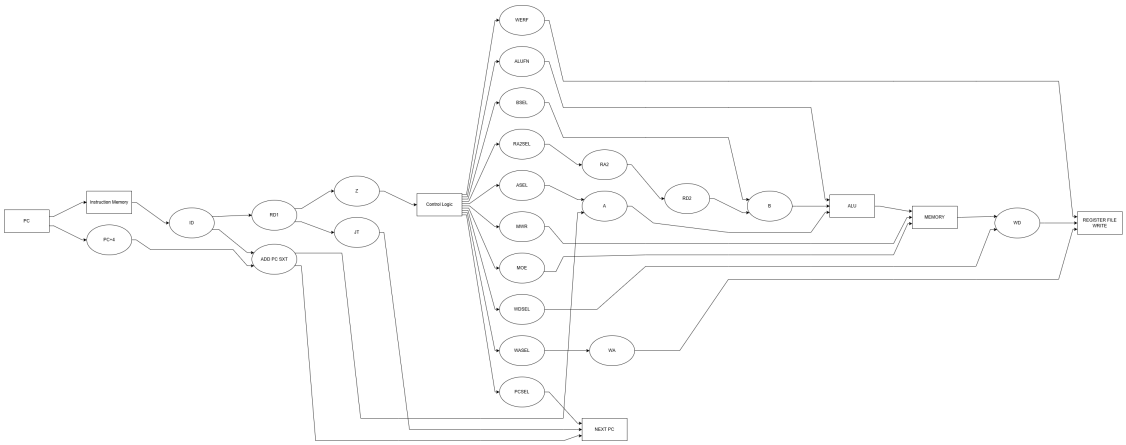

Figure 3.2: Simplified Beta dependency graph

This simplification was not performed solely for readability purposes. Its primary objective was to eliminate backward dependencies and produce a graph with a unidirectional flow. This property is essential because it allows the simultaneous behavior of hardware circuits to be transformed into a valid sequential execution order.

One notable modification visible in the simplified graph is the decomposition of the `REGISTER READ` component into two independent operations: `RD1` and `RD2`. This decomposition was necessary because `RD2` depends on the control logic while `RD1` does not. Without this separation, the dependency graph would contain cycles, making sequential execution impossible.

3.1.2 Making sure the sequential code covered every part

Before starting the implementation, all components were analyzed to ensure that no additional decomposition was required.

Using the simplified dependency graph from Figure 3.2 and the reference Beta machine schematic shown in Figure 1.1, the following sequential execution model for one processor cycle was obtained:

```
id = Instructions.at(pc)
rd1 = registers(ra(id))
z = ra(id) == 31
jt = rd1
pcSxt = pc + 4 + 4*sxt_C(id)
control = ControlLogic(id, z)
ra2 = multiplexer(control.ra2sel, [rb(id), rc(id)])
rd2 = registers[ra2]
a = multiplexer(control.asel, [rd1, pcSxt])
b = multiplexer(control.bsel, [rd2, sxt_C(id)])
aluOutput = alu(a, b, control.alufn)
mrd = memoryLogic(rd2, aluOutput, control.mwr, control.moe)
wa = multiplexer(control.wasel, [rc, 30])
wd = multiplexer(control.wdsel, [pc + 4, aluOutput, mrd])
registerWrite(wa, wd, control.werf)
pc = multiplexer(control.pcsel, [pc + 4, pcSxt, jt, 4, 8])
```

To verify that all signals, inputs, and outputs of the Beta machine were correctly represented in the sequential implementation, each wire involved in the pseudocode execution flow was highlighted directly on the processor schematic.

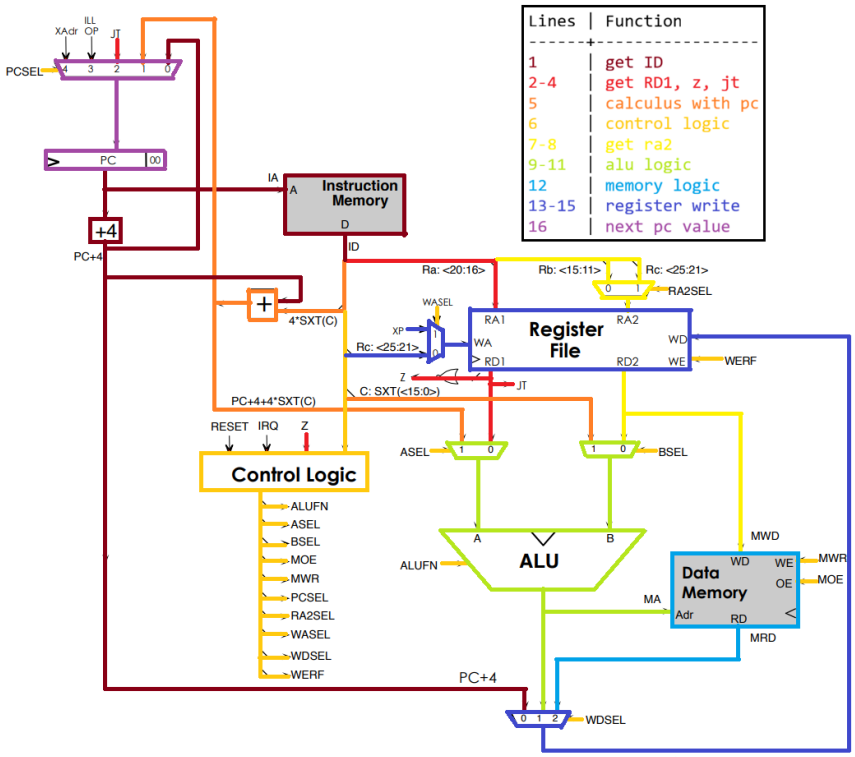

Figure 3.3: Sequential decomposition of the Beta execution cycle

Figure 3.3 also includes a color-coded table identifying the different sequential stages of the execution cycle. Each color corresponds to a specific step in the pseudocode and highlights the associated signal paths directly on the Beta machine schematic.

This visualization serves two purposes. First, it provides a direct correspondence between the sequential software implementation and the original hardware data flow. Second, it helps verify that every signal involved in the execution cycle is accounted for and executed in the correct order.

The table additionally indicates the role of each step within the processor cycle, making the sequential decomposition easier to analyze and validate.

3.1.3 Writing the code

After defining the execution model and validating the sequential decomposition of the Beta machine, the implementation phase could begin.

One utility function that proved necessary was the conversion between big-endian and little-endian representations:

```
String _swapEndian(String bin) {
    if (bin.length != 32) throw "Wrong binary $bin";
    String hex = "";
    for (int i = 24; i >= 0; i -= 8) {
        hex += bin.substring(i, i + 8).padLeft(8, '0');
    }
    return hex;
}
```

The code assembler generates binary instructions in big-endian form, while hexadecimal values appearing in assembly source files are represented in little-endian order. This function reverses the byte order of a 32-bit value, ensuring consistent interpretation between assembly source code and the internal representation used by the simulator.

The implementation of the execution cycle itself was relatively straightforward because most of the complexity had already been addressed during the dependency analysis phase described previously.

The simulator is represented by a `Beta` class responsible for storing the processor state and executing instructions.

A dedicated `ControlLogic` class was also introduced to separate instruction decoding from execution. This separation improves readability and closely follows the architecture of the original Beta machine. The `ControlLogic` object is constructed from the instruction opcode and the value of the `Z` signal. Its role is to compute all control signals according to the control table shown in Figure 3.4.

The core of the simulator is implemented in the private method `_executeOneCycle()`, which corresponds directly to the sequential execution model established in the previous chapter. During a single cycle, the method:

- fetches the instruction located at the current program counter,

	RESET	IRQ	OP	OPC	LD	LDR	ST	JMP	BEQ	BNE	ILLOP
ALUFN[5:0]	--	--	F(op)	F(op)	"+"	"A"	"+"	--	--	--	--
ASEL	--	--	0	0	0	1	0	--	--	--	--
BSEL	--	--	0	1	1	--	1	--	--	--	--
MOE	--	--	--	--	1	1	0	--	--	--	--
MWR	0	0	0	0	0	0	1	0	0	0	0
PCSEL[2:0]	--	4	0	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	3
RA2SEL	--	--	0	--	--	--	1	--	--	--	--
WASEL	--	1	0	0	0	0	--	0	0	0	1
WDSEL[1:0]	--	0	1	1	2	2	--	0	0	0	0
WERF	--	1	1	1	1	1	0	1	1	1	1

Figure 3.4: Beta control logic

- computes all intermediate signals,
- executes the ALU operation,
- performs memory accesses,
- updates registers,
- computes the next program counter,
- and updates the visualized state of the processor.

The method is intentionally private because it represents a low-level primitive used internally by higher-level execution controls such as step-by-step execution, continuous execution, and state restoration.

The implementation is shown below:

```
void _executeOneCycle([bool notify = true]) {
    if (!isStopped) {
      id = _memory[pc]?.toRadixString(2).padLeft(32, "0") ?? "";
      pc4 = pc + 4;
      sxtC = _sxtC(id);
      pcSxt = pc4 + 4 * sxtC;
      ra = _ra(id);
      rb = _rb(id);
      rc = _rc(id);
      rd1 = _registers[ra];
      z = ra == 31;
      int jt = rd1;
      control = ControlLogic(id.substring(0, 6), z);
      ra2 = _mux(control.ra2sel, [rb, rc]);
      rd2 = _registers[ra2];
      a = _mux(control.asel, [rd1, pcSxt]);
      b = _mux(control.bsel, [rd2, sxtC]);
      aluOutput = _alu(a, b, control.alufn);
      mrd = _memoryLogic(rd2, aluOutput, control.mwr, control.moe);
      wa = _mux(control.wasel, [rc, 30]);
      wd = _mux(control.wdsel, [pc4, aluOutput, mrd]);
      _registerWrite(wa, wd, control.werf);
      pc = _mux(control.pcsel, [pc4, pcSxt, jt, 4, 8]);
      isStopped = _memory[pc] == 0;
      _cycles++;
      if (notify) {
        notifyListeners();
      }
    }
  }
```

Several data structures were chosen to represent the internal state of the processor:

- Registers are represented as a fixed-size list of 32 integers. This choice reflects the hardware architecture directly and provides constant-time access to register values.
- Labels are stored in a hash map associating label names as strings with memory addresses as integers. This structure provides efficient lookup during assembly and execution while naturally representing the relationship between symbolic identifiers and concrete memory locations.
- Memory is represented using a hashmap associating memory addresses with stored values. The key is an integer which corresponds to the address, and the value is also an integer which corresponds to the data stored at that address. Since only allocated addresses need to be stored, this representation avoids reserving a large contiguous memory space and therefore reduces memory usage.
- The assembled program is stored as a list of integers, where each integer represents a complete 32-bit machine instruction. Storing instructions as integers rather than binary strings reduces memory usage.

3.1.4 Access to the beta machine

Finally, most internal variables of the Beta machine class were intentionally made publicly accessible. This design choice was motivated by the future integration of the schematic visualization interface, where internal processor states such as registers, ALU outputs, and control signals needed to be dynamically displayed to the user.
They could have been private with getters for each, thus making the possibility of setting them to another value from outside the class impossible, but the setting of those variables would not break anything because they are always written before read during the execution of a cycle of the beta machine.

There are many methods to communicate with the beta. The following are simply a byproduct of not saving the actual disassembly in the beta. They have been created to be able to retrieve some information previously fed to the beta:

- getDisassemblyAt(index): reforms the assembly from the binary
- getHexAt(index): returns the hexadecimal value corresponding to the binary with the swapped endian
- getLabelOf(int index): returns the label or null at the given index

The following methods are getter methods that provide an abstraction layer over the internal register representation:

- `getRegisterValue(int regNum)` returns the current value stored in the specified register.
- `getRegisterName(int regNum)` returns the symbolic name associated with a register. Although most registers are identified by their numerical index, registers 27, 28, 29, and 30 correspond respectively to `BP` (Base Pointer), `LP` (Link Pointer), `SP` (Stack Pointer), and `XP` (Exception Pointer). This method centralizes the naming logic and avoids duplicating register-name mappings throughout the user interface.

These methods allow control of the execution.

- restart(notify=true) cleans memory, sets the pc, and other fields to 0, and resets the program
- assemble(programParse) restarts the beta machine and stores the program in memory
- run(cycles, notifyAtEachStep=false, microsecondsPerCycles=1) runs a timer asynchronously that ticks periodically with the given microseconds; at each tick the beta calls the method executeCycle.
- executeCycle(cycles=1, notifyAtEachStep=false) performs a given number of executions; each execution can notify the frontend components depending on the value of the argument notifyAtEachStep.

The Beta machine is responsible for managing its own execution timer. Storing the timer directly within the Beta model simplifies the handling of asynchronous execution, as multiple

interface components need to interact with it.

In particular, both the code editor and the Beta visualization provide controls for starting, stopping, and stepping through program execution. Since the Beta machine is the only component shared between these views, placing the timer logic inside the Beta class ensures that all execution-related operations remain centralized and synchronized.

3.2 Implementing a compiler

The next logical step was to feed binary code to the beta machine. Bsim is made to perform that. In order to avoid making the user write binary by itself, it provides a code editor where the user can write in assembly; the assembly code is then compiled and translated into binary.

The decision was made to implement a custom compiler rather than rely on existing alternatives, because available compilers may not be compatible with Flutter and adapting seemed too complex and out of the scope of the project. Additionally, an analysis of the existing compiler indicated that certain assembly instructions appeared irrelevant for the beta machine architecture. As a result, these instructions were deliberately excluded to prevent potential issues, such as enabling unintended machine states, causing user confusion regarding ignored instructions, or producing unexpected behavior.

An additional motivation for this implementation was to gain hands-on experience in compiler development.

Compiling systems are commonly divided into two stages:

- lexical analysis, responsible for token generation,
- and syntactic analysis, responsible for constructing an Abstract Syntax Tree (AST).

Those two steps can detect errors either grammatically (writing something that does not belong to the grammar of the language) or syntactically (something that does not respect the structure of a specific pattern in the language).

3.2.1 Formalisation of the grammar

The first step is to write a grammar formally in order to verify that the language can be parsed, and in order to know what to implement.

Before implementing the compiler, the supported subset of the uasm language was formally described using an Extended Backus-Naur Form (EBNF) grammar.

```
PROGRAM ::= empty
          | DIRECTIVE PROGRAM

DIRECTIVE ::= MACRO
            | INCLUDE
            | ASSIGNMENT
            | EXPRESSION
            | CALL

MACRO ::= ".macro" ID "(" OPTIONAL_PARAMS ")" MACROEXP

INCLUDE ::= ".include" "\"" STRING "\""

MACROEXP ::= "{" MACRO_INSTRS "}"
           | MACRO_INSTRS
```

```
16
17 MACRO_INSTRS ::= MACRO_INSTRUCTION OPTIONAL_MACRO_INSTRS
18
19 OPTIONAL_MACRO_INSTRS ::= empty
20                          | MACRO_INSTRS
21
22 MACRO_INSTRUCTION ::= EXPRESSION
23                     | ASSIGNMENT
24                     | CALL
25                     | ".align" number
26
27 OPTIONAL_PARAMS ::= empty
28                   | PARAMS
29
30 PARAMS ::= ID
31          | ID "," PARAMS
32
33 OPTIONAL_ARGUMENTS ::= empty
34                       | ARGUMENTS
35
36 ARGUMENTS ::= EXPRESSION
37             | EXPRESSION "," ARGUMENTS
38
39 ASSIGNMENT ::= ID "=" EXPRESSION
40
41 CALL ::= ID "(" OPTIONAL_ARGUMENTS ")"
42
43 EXPRESSION ::= EXPRESSION OP EXPRESSION
44              | "(" EXPRESSION ")"
45              | NUMBER
46              | ID
47
48 OP ::= "+"
49      | "-"
50      | "*"
51      | "/"
52      | "%"
53      | ">>"
54      | "<<"
55
56 ID ::= LETTER
57      | LETTER ID_TAIL
58
59 ID_TAIL ::= empty
60           | ID_CHAR ID_TAIL
61
62 ID_CHAR ::= LETTER
63           | DIGIT
64           | "_"
```

3.2.2 Abstract Syntax Tree

General Structure

Rather than implementing a traditional compiler architecture composed of a lexer followed by a parser, the language is parsed directly through the Abstract Syntax Tree (AST).

This decision was motivated by the simplicity of the supported uasm language. The language contains relatively few keywords and its grammar is sufficiently unambiguous to allow direct syntactic recognition through regular expressions. Consequently, introducing a dedicated tokenization stage would have increased the complexity of the implementation without providing significant benefits. Instead, each AST node is responsible for recognising and parsing the syntax corresponding to its own grammar rule. In practice, regular expressions are used directly within the AST node classes to identify valid constructs.

Operator precedence

Operator precedence is handled during AST construction. When parsing an expression, operators are searched in reverse precedence order. Lower-precedence operators therefore become higher-level nodes in the AST, while higher-precedence operators remain deeper in the tree.

For example:

$$1 + 2 \times 3$$

is represented as:

$$(1 + (2 \times 3))$$

rather than:

$$((1 + 2) \times 3)$$

The implementation is shown below.

```
1  OperatorExp.parse(String text, final HashSet<String> scope) {
2    // operators in reverse order of precedence for AST
3    const List<String> precedence = [">>", "<<", "-", "+", "/", "*", "
       %"];
4    bool parsed = false;
5    for (String op in precedence) {
6      if (parsed = _splitForOp(text, op, scope)) {
7        return;
8      }
9    }
10   if (!parsed) {
11     throw "Unknown operator";
12   }
13 }
```

AST classes

All AST nodes inherit from a common abstract base class:

```
abstract class ASTNode{
  int size();
}
```

The `size()` method returns the number of generated bits produced by the node. This information is required during the first assembly pass to compute label addresses before code generation begins.

Each AST node defines two static methods:

- `isValid()`, responsible for determining whether a piece of text matches the node grammar,
- `parse()`, responsible for constructing the corresponding AST object.

This approach replaces the role traditionally played by a lexer. Rather than producing tokens and dispatching them to a parser, each node class can recognise and instantiate itself directly.

The method was intentionally named `parse()` rather than using a constructor in order to emphasise that syntactic analysis is being performed before object creation.

According to my grammar, here are the classes of the abstract class ASTNode:

- *ASTNode*
  - IncludeNode
  - MacroNode
  - *CompilableNode*
    - CallNode
    - *ExpressionNode*
      - NumberExp
      - IDExp
      - OperatorExp
  - LabelNode
  - AssignmentNode
  - *MacroInstruction*
    - AlignInstruction
    - ExpressionInstruction
    - AssignmentInstruction
    - CallInstruction

Parser

The central entry point of the assembler is the `ProgramParser` class.

Its responsibilities are:

- processing include directives,

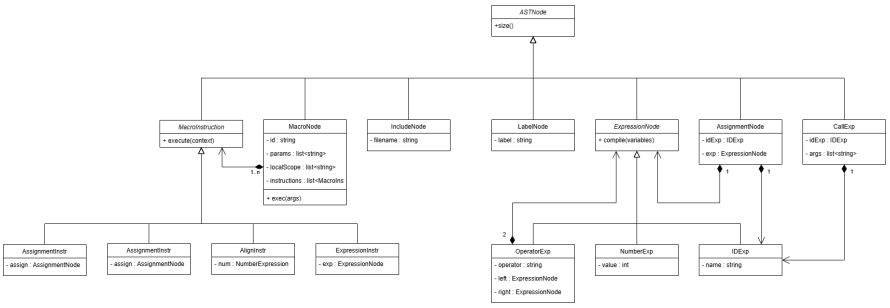

Figure 3.5: AST

- registering macro definitions.
- constructing the AST,
- collecting labels,
- collecting expressions,
- registering variables,

To construct the abstract syntax tree, there is a class called ProgramParser. It contains some variables for which specific data structures were picked in order to optimise lookup speed as well as memory.

```
class ProgramParser {
  // filenames already included
  static final Set<String> includeTrace = {};
  // { name: value }
  static final HashMap<String, String> variables = HashMap();
  // { name: { args_nb : Macro } }
  static final HashMap<String, HashMap<int, MacroAstNode>> macros =
      HashMap();
  // { name: value }
  final HashMap<String, String> labels = HashMap();
  // { name: value }
  final Map<int, CompilableNode> children = {};
  // { name: value }
  MacroAstNode? insideBraces;
}
```

The `labels` structure associates label names with their corresponding addresses. Labels are stored as binary strings rather than integers because they are evaluated in the same way as variables during expression compilation. Storing them directly in binary form avoids repeated conversions and ensures that their exact bit representation is preserved when used in expressions.

The `children` structure contains all top-level compilable nodes of the program. A map indexed by source-code line number was chosen instead of a simple list. This allows compilation errors to be reported with precise line information while still preserving the order in

which instructions must be evaluated.

Macros are stored in a two-level map. The first level associates a macro name with its definitions, while the second level indexes these definitions by their number of arguments. This design enables macro overloading, allowing multiple macros to share the same name provided that they accept different numbers of parameters.

The `insideBraces` field is used during parsing to support multi-line macro definitions. When a macro body begins with an opening brace, subsequent lines are accumulated until the corresponding closing brace is encountered.

The structures `macros`, `variables`, and `includeTrace` are declared as static because they must be shared across all parser instances. This is particularly important for the implementation of the `.include` directive, where included files must have access to the same symbol table and macro definitions as the file that includes them. The shared `includeTrace` structure is additionally used to detect cyclic inclusions and prevent infinite recursion during parsing.

The main parsing process is implemented in the constructor `ProgramParser.parse()`.

The parser processes the source file line by line. For each line, it determines which AST node type can recognize the corresponding grammar construct through its `isValid()` method. Once the correct node type has been identified, the associated `parse()` method is invoked to create the corresponding object.

The handling of `.include` directives is delegated to `IncludeAstNode`. During parsing, the included file is recursively parsed, and its variables and macro definitions are added to the shared parser state. This makes the contents of included files immediately available to the remainder of the compilation process.

Similarly, `MacroAstNode.parse()` not only constructs the macro representation but also registers the resulting definition in the global macro table. As a consequence, macro declarations become available as soon as they are encountered.

When a label is parsed, its address must be determined immediately. This address is computed by summing the sizes of all previously parsed compilable nodes. The resulting value corresponds to the position, in bytes, of the instruction that follows the label. This mechanism allows labels to be referenced later during code generation.

Assignments are handled differently from other AST nodes. Although they participate in parsing, they do not generate machine code and therefore are not considered compilable nodes. For this reason, the assignment logic is separated into a dedicated `compile()` method, whose purpose is to evaluate the assigned expression and update the global variable table.

Finally, expression nodes and macro call nodes are inserted into the `children` collection together with their source-code line number.

```
class ProgramParser {
    ProgramParser.parse([String filename = "main.uasm"]) {
        for (int i = 0 ; i < lines.length ; i++) {
            String line = lines[i];
            if (IncludeAstNode.isValid(line)) {
              IncludeAstNode.parse(line);
            } else if (MacroAstNode.isValid(line)) {
              MacroAstNode node = MacroAstNode.parse(line);
            } else if (LabelAstNode.isValid(line)) {
              LabelAstNode labelNode = LabelAstNode.parse(line);
              labels[labelNode.label.name] = (children.values.fold(0,
                  (value, element) => value+ element.size())~/8)
            } else if (AssignmentAstNode.isValid(line)) {
              AssignmentAstNode node = AssignmentAstNode.parse(line,
                  HashSet.from(ProgramParser.variables.keys));
              node.compile(ProgramParser.variables);
            }else if (CallNode.isValid(line)) {
              children[i+1] = CallNode.parse(line, HashSet.from(
                  ProgramParser.variables.keys));
            }else if (ExpressionAstNode.isValid(line)) {
              children[i+1] = ExpressionAstNode.parse(line, HashSet.
                  from(ProgramParser.variables.keys));
            } else {
              throw "Unknown command";
            }
        }
    }
}
```

Once the parsing phase has completed, the abstract syntax tree can be evaluated to generate the final machine code. This task is performed by the `eval()` method of `ProgramParser`.

For each node stored in `children`, the method invokes `compile()`, which returns the corresponding binary string. The generated binary fragments are concatenated to produce the complete machine code of the program.

Compilation errors are intercepted and enriched with the source-code line number associated with the node. This considerably improves the quality of error reporting, as users can directly locate the instruction responsible for the failure.

Before compiling each node, a dictionary of literals is constructed. This dictionary contains both labels and variables and is passed to the `compile()` method. Consequently, every expression has access to the complete symbol table and can resolve references to variables, labels, and the special symbol `.`, which represents the current memory address.

```
class ProgramParser {
  List<String> eval() {
    for (int i = 0; i < children.length; i++) {
      CompilableNode exp = children.values.elementAt(i);
      try{binary.write(exp.compile(literals));}
      catch(e){
        throw "Compilation error: $e with expression at line ${
            children.keys.elementAt(i)}";
      }
    }
    return RegExp(
      r'.{1,32}',
    ).allMatches(binary).map((m) => m.group(0)!).toList();
  }
}
```

Finally, once the complete binary output has been generated, it is divided into chunks of 32 bits corresponding to individual Beta instructions. The resulting list of binary strings is then returned and can be loaded directly into the Beta machine memory.

Include handling

The `.include` directive allows source files to import declarations from other files.

To prevent infinite recursion caused by cyclic includes, the parser maintains a global inclusion trace. Whenever a file is included, its name is inserted into this trace. If an attempt is made to include a file already present in the trace, an exception is raised.

On the following frame is the code in IncludeNode.parse that prevents infinite recursion:

```
IncludeAstNode.parse(String text)
{
    if(ProgramParser.includeTrace.contains(filename)){
        throw "Include cycle detected ${ProgramParser.includeTrace}";
    }
    ProgramParser.includeTrace.add(filename);
}
```

Macros

Macros constitute the core mechanism used by BSim to define assembly instructions.

Rather than being hardcoded into the assembler, instructions such as `ADD` and `ADDC` are themselves defined as macros. Consequently, implementing macro expansion was essential to support the Beta assembly language.

A macro definition is represented by the `MacroAstNode` class. A macro can be viewed as a parameterized code-generation function: when invoked, it receives a list of arguments and produces a binary output according to the instructions contained in its body.

```
class MacroAstNode extends ASTNode{
    late final IDExp ID;
    late final String Function(List<String> args, HashMap<String,
        String> scope) exec;
    final List<String> params = [];
    final List<MacroInstruction> instructions = [];
    final HashSet<String> localScope = HashSet();

    @override
    int size() => instructions.fold(0, (value, element) => value +
        element.size());
}
```

Rather than interpreting the macro definition each time it is called, the parser compiles the macro into a callable function stored in the `exec` field. This function receives the evaluated arguments and the current scope, executes the macro instructions, and returns the generated binary representation.
The `size()` method computes the number of generated bits produced by the macro. Since the supported macros only affect the values of generated instructions and not their quantity, the size of a macro depends solely on its body and is independent of the arguments supplied during invocation. This property is essential for the first assembly pass, during which label addresses are computed before code generation.

Macro execution is performed within a dedicated execution context.
The execution context isolates the state of a macro invocation from the rest of the compilation process. Each call therefore executes with its own set of local variables while sharing access to the symbols available in the current scope.

```
class MacroExecutionContext {
  final HashMap<String, String> variables;
  final StringBuffer output = StringBuffer();
  int padding = 0;

  MacroExecutionContext(this.variables);
}
```

All instructions appearing inside a macro body inherit from the abstract class `MacroInstruction`. Besides defining a common type hierarchy, the class provides a factory constructor responsible for identifying and constructing the appropriate instruction subclass.

```
abstract class MacroInstruction extends ASTNode{
    factory MacroInstruction.parse(String text, HashSet<String>
        localScope){
        if(AlignInstruction.isValid(text)){
            return AlignInstruction.parse(text);
        }
        if(AssignmentInstruction.isValid(text)){
            return AssignmentInstruction.parse(text, localScope);
        }
        if(CallInstruction.isValid(text)){
            return CallInstruction.parse(text, localScope);
        }
        if(ExpressionInstruction.isValid(text)){
            return ExpressionInstruction.parse(text, localScope);
        }
        throw "Unknown macro instruction";
    }
}
```

Macros are stored according to both their identifier and their number of parameters. Consequently, multiple macros may share the same name provided that they accept a different number of arguments. The call to a macro is resolved with the compile method of CallNode; it is worth mentioning the method size, which returns the size of the corresponding macro. Before execution, the compiler verifies that the requested macro exists and that a definition matching the supplied number of arguments is available. Any mismatch results in a compilation error.

```
class CallNode extends CompilableNode {
  String compile(HashMap<String, String> variables) {
    if(!ProgramParser.macros.containsKey(ID.name)){
      throw "Unknown macro ${ID.name}";
    }
    else if(!ProgramParser.macros[ID.name]!.containsKey(args.length)){
      throw "Macro ${ID.name} can have ${ProgramParser.macros[ID.name
          ]!.keys.toList()} args but ${args.length} were given";
    }
    return ProgramParser.macros[ID.name]![args.length]!.exec(
      args.map((e) => e.compile(variables)).toList(), variables);
  }

  @override
  int size() => ProgramParser.macros[ID.name]?[args.length]?.size() ??
      0;
}
```

This macro system proved sufficiently expressive to implement the complete Beta instruction set without introducing any instruction-specific logic into the assembler itself. As a result, the assembler remains generic and only requires support for expressions, labels, directives, and macro expansion in order to generate executable machine code.

Prepocessing

Instead of implementing a complete lexer, a lightweight preprocessing stage was introduced through a function called `cleanProgram(text)`.

This preprocessing step performs:

- comment removal,
- whitespace normalization,
- line segmentation,
- and elimination of unnecessary carriage returns.

The resulting cleaned instruction list is then directly processed by the AST parser.

Because the supported uasm grammar remains relatively simple and unambiguous, this lightweight preprocessing stage was sufficient and avoided the complexity of a full tokenization system.

Using an AST representation provided several advantages:

- separation between syntax analysis and code generation,
- simplified operator precedence handling,
- easier implementation of macros and expressions,
- and improved extensibility for future language features.

3.2.3 Label linking

One major challenge during the implementation of the assembler was handling label resolution. Assembly instructions may reference labels whose final addresses are not yet known during the initial parsing stage. This creates a forward reference problem: the address of a label is required to generate machine code, but the final address itself depends on the total size of previously assembled instructions.

This issue was further complicated by the fact that some assembly expressions could generate multiple machine instructions, meaning that the final program counter value could not be determined incrementally in a single pass.

To solve this problem, the assembler was implemented as a two-pass assembly process.

During the first pass, the source code is traversed to:

- determine the binary size of each instruction or expression,
- compute the corresponding program counter progression,
- and construct the symbol table associating labels with their final addresses.

This is done thanks to the method size() in the abstract class ASTNode (see 3.2.2). This method returns the number of bits that will result from the evaluation depending on the type:

Type		Size
assignmentNode	⇒	0
includeNode	⇒	0
labelNode	⇒	0
macroNode	⇒	$\sum expression.size()$
IDExpression	⇒	32
numberExpression	⇒	binaryNumber.length
callExpression	⇒	macro.size()
operatorExpression	⇒	operator == "%" ? right.size()-1 : max(left.size(), right.size())

Table 3.1: Size of each type

Assignment, labels, and include do not return any binary; they just create variables. Id always has a size of 32 bits; this means that a variable alone by default would be parsed as a single instruction. The size of a macro is determined by the sum of the size of each expression composing it. Because a macro can only contain 3 types of instructions:

- AlignInstruction that does not return a binary
- AssignInstruction that does not return a binary
- ExpressionInstruction that does return a binary

The size of a number is given by its representation in binary; this is necessary because a number might not always be 32 bits. They can voluntarily be written in binary. A call is obviously given by the size of the corresponding macro. Finally, the operator size is either given by the maximum size between the expressions around the operator or by the expression on the right of the modulo.

During the second pass, the previously computed symbol table is used to resolve label references and generate the final binary instructions.

A bidirectional map structure was initially considered for associating labels and addresses. However, this solution was rejected because multiple labels may legally reference the same program counter value.

The computation to binary is done thanks to the file beta.uasm that needs to be included in the main file, in order to be parsed so that instructions can be recognised. Then the expression node has an eval() method that returns the number they evaluate to; it returns one or multiple 32-bit numbers. This can then be fed to my beta machine.

3.3 Implementing the code Editor

One important component of the project was the integration of a code editor allowing users to write and modify assembly programs directly inside the web application. Several solutions were evaluated before selecting the final implementation.

3.3.1 Using TextField

The first approach consisted of implementing a custom editor using Flutter's standard `TextField` widget. Although this solution initially appeared simple, several limitations quickly became apparent.

The first issue concerned performance. The `TextField` widget is designed for conventional text input rather than large source-code editing workloads. When assembly programs reached approximately 400 lines, the editor became noticeably slow and less responsive.

Another major difficulty involved synchronizing the line-number gutter with the displayed code. Since Flutter's standard text components do not provide direct access to the internal rendering metrics of the `TextField`, the gutter had to be implemented manually using a separate column widget placed next to the editor.

This approach required estimating line spacing and character dimensions to align the line numbers with the corresponding code lines. While small inaccuracies were barely visible for short programs, even minimal offsets could accumulate and produce significant visual misalignment for larger source files.

3.3.2 Using package `flutter_code_editor`

Because of these limitations, a second solution based on the `flutter_code_editor` package was evaluated. This editor provided additional code-editing features, but similar alignment problems remained between the line-number gutter and the text content. In addition, performance issues persisted when handling large files. After inspecting the package implementation, modifying the editor internally was considered but ultimately abandoned.

3.3.3 Using the package `flutter_monaco`

The final solution was based on the `flutter_monaco` package, which integrates the Monaco editor used by Visual Studio Code. This editor provided significantly better usability, syntax highlighting, and overall editing capabilities.

However, integrating Monaco inside Flutter introduced several platform-specific challenges.

On desktop platforms, scrolling behavior was unstable. Scrolling operations occasionally stopped after only a few pixels, preventing smooth navigation through the source code.

On web platforms, the primary issue concerned interaction conflicts between the Monaco editor and the application's resizable split-view interface. When the user attempted to resize

panels while the cursor entered the editor region, the resizing operation stopped immediately.

After consulting the Monaco documentation, the issue was identified as a consequence of the editor being rendered inside an HTML `iframe` on web platforms. By default, the iframe captures mouse-related events and prevents them from propagating to Flutter widgets outside the editor.

The documentation proposed several possible solutions. The selected approach consisted of temporarily disabling mouse-event capture inside the editor by modifying the editor's `enable` property during split-view resizing operations.

This solution integrated naturally with the previously implemented `SplitViewController`, which already managed the state of the resizable layout. During dragging operations, the controller temporarily disables editor interaction, allowing resize events to propagate correctly to the Flutter interface.

The resulting implementation proved stable on web platforms, which represented the primary deployment target of the project. Desktop-specific issues were considered lower priority because the application was primarily intended to be used as a browser-based educational tool.

Despite its advantages, the Monaco integration still presents several limitations. In particular, the Flutter wrapper does not provide full access to all Monaco customization features. This restricts fine-grained theme customization and prevents direct integration of parsing or compilation errors inside the editor interface itself.

3.3.4 Handling Errors

To show compilation or runtime errors, the code editor interface package does not support precise line highlighting for error localization. Instead, a snackbar is used to inform the user of the error type, the corresponding line number, and, in the case of compilation errors, the expected input. The snackbar can be closed either by user interaction or by correcting the code and reassembling it.

A snackbar was chosen because it is highly customizable and, more importantly, it does not interfere with existing UI elements, as it is rendered as an overlay. Alternative solutions, such as displaying errors in a container under the editor, would either shift the layout when an error occurs or require reserving permanent space for a conditionally visible element, resulting in an inconsistent or visually inefficient interface. The snackbar provides a temporary, non-intrusive, and easily dismissible solution, while still allowing for further customization if needed.

3.4 Implementing the visualisation

There are two possible visualizations of the beta execution.

- The programmer view that displays the registers, the disassembly, the stack, the memory, and a console.
- The schematic view that draws the representation of the beta machine with the binary code written at each important step.

Both views rely on the ListenableBuilder widget to receive update messages from the beta machine, which is a ChangeNotifier that notifies its Listeners whenever an execution cycle occurs.

3.4.1 Programmer view

5 widgets compose this view. Each widget is computed or updated based on the beta machine, not the code editor. The beta machine does not contain the assembly lines; it only stores the binary translation.
The disassembly widget shows the current assembly line to execute alongside its corresponding hexadecimal translation with the program counter associated with it; it may also display the label name if there is one. The challenge with this widget was to recover the assembly from the hexadecimal value. It is done by a simple lookup table for the instructions. This approach, rather than storing the assembly inside the beta, has two advantages:

Less memory used

Ability to reformulate some expression (display the result instead of a calculus)

3.4.2 Schematic view

This widget builds a derived class of CustomPainter called BetaPainter, which paints the representation of the beta machine on a canvas. This approach allows the same behavior as in Bsim, that is to say, the fact that the beta machine always fits the space it has. In the original software, it relies on the creation of an SVG image that can be scaled without loss. In Flutter, the most equivalent technique is to set a virtual size for the canvas and to draw every element on it using this virtual size. Then the CustomPainter is expanded in its parent and can be resized without needing a rebuild.

3.5 Layout and controllers

The graphical interface relies on two principal layout components: SplitView and TabView.

The SplitView component is responsible for managing the main workspace organization of the application. It allows the interface to dynamically display either the code editor, the Beta machine visualization, or both simultaneously in a split-screen configuration. This behavior reproduces the workflow of the original BSim application while improving flexibility for web usage.

The TabView component manages the code editor tab system and allows users to work with multiple assembly files simultaneously. Each tab contains an independent code editor, enabling easier organization of programs and improving usability during development and testing.

To improve the flexibility and responsiveness of the graphical interface, the application uses dedicated controllers for managing complex layout interactions. Two main controllers were implemented: SplitViewController and TabViewController.

These controllers centralize layout-related state management and allow different widgets to communicate without creating strong dependencies between interface components. This approach improves modularity and simplifies synchronization between the editor and visualization components.

The SplitViewController was implemented as a dedicated state management component responsible for controlling the behavior of the split-screen layout. It extends Flutter's ChangeNotifier class, allowing the interface to react dynamically whenever the layout state changes.

The controller manages two main pieces of information:

- The width ratio between the two panels
- The current dragging state of the separator

```
class SplitViewController extends ChangeNotifier {
  double _leftWidthRatio;
  bool _isDragging;

  SplitViewController({double initialLeftWidthRatio=1})
      : _leftWidthRatio=initialLeftWidthRatio,
        _isDragging=false;

  set leftWidthRatio(double value) {
    _leftWidthRatio = value.clamp(0, 1);
    notifyListeners();
  }

  double get leftWidthRatio => _leftWidthRatio;

  bool get isDragging => _isDragging;

  void drag() {
    _isDragging = true;
    notifyListeners();
  }

  void undrag() {
    _isDragging = false;
    notifyListeners();
  }
}
```

The leftWidthRatio variable stores the proportion of space allocated to the left panel of the split view. The value is clamped between 0 and 1 to ensure that the interface remains within valid layout boundaries and to prevent invalid resizing behavior.

Whenever the ratio changes, the controller calls notifyListeners(), automatically triggering updates in all dependent widgets. This mechanism takes advantage of Flutter's reactive rendering model and ensures immediate synchronization between user interactions and the visual interface.

The controller also maintains an _isDragging state indicating whether the split-view separator is currently being manipulated by the user. This information is used to temporarily restrict certain interactions during resizing operations, reducing the risk of inconsistent interface states or unintended user actions.

The controller allows external widgets to dynamically modify the layout state. For example, after compiling assembly code, the simulator automatically forces the interface into split-view mode in order to display both the source code and the processor visualization simultaneously. This improves usability by immediately exposing the execution environment to the user.

The controller is also used to temporarily disable editor interactions while the split view is being resized or modified. Preventing concurrent interactions during layout updates reduces

the risk of inconsistent interface states and improves overall stability.

Using controllers for layout management proved particularly useful in Flutter because the interface is highly reactive and many widgets may need to respond simultaneously to state changes. This design reduces unnecessary coupling between components and makes future interface extensions easier to implement.

Unlike the split-view system, the tab management system does not use a dedicated controller. Instead, tab updates are handled directly through Flutter's setState() mechanism. Since the tab behavior is relatively localized, even when considering scalability, and simpler than the split-view synchronization logic, introducing a separate controller was considered unnecessary and would have added avoidable architectural complexity.

3.6 File management

The simulator includes a file management system intended to support modular assembly programs. This is particularly relevant in the context of educational use, where programs may need to be split across multiple source files using `.include` directives.

Due to the application being deployed as a web-based system, direct access to the host file system is not available. In addition, persistent storage was not required for the objectives of this project. For these reasons, file management was implemented as an in-memory virtual file system.

All files are stored during runtime using a map-based data structure associating file names with their corresponding content. This approach ensures constant-time access to file data while avoiding external dependencies such as databases or filesystem APIs.

Here is the class responsible for file management:

```
class Files {
  static final HashMap<String, String> _files = HashMap();

  static String? getText(String filename) => _files[filename];
  static void create(String filename) => _files[filename] = "";
  static void write(String filename, String text) {_files[filename] =
      text;}
  static bool doesExist(String filename) => _files.containsKey(
     filename);
}
```

The implementation intentionally remains minimal. The class provides only four operations:

- file creation,
- content retrieval,
- content modification,
- and existence verification.

These operations were sufficient for the needs of the assembler and the include system while keeping the implementation lightweight.

The two files main.uasm and beta.uasm are stored in an assets folder. Although the source files are initially stored as application assets, they are loaded into the virtual file system during startup. This ensures that all file accesses performed by the assembler use the same abstraction layer, regardless of whether the file originates from the bundled assets or from a future user-created source file.

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  Files.write(
    "main.uasm",
    await rootBundle.loadString("assets/uasm/main.uasm"),
  );
  Files.write(
    "beta.uasm",
    await rootBundle.loadString("assets/uasm/beta.uasm"),
  );

  runApp(MainView());
}
```

The system does not provide persistence between sessions. This design choice is justified by both technical and conceptual considerations. From a technical perspective, browser-based environments do not provide direct unrestricted access to the host file system, and introducing a persistence layer such as IndexedDB or a backend service was unnecessary for the objectives of this project. From a design perspective, persistent storage was not required for the simulator's intended use case, which focuses on interactive execution rather than long-term project storage.

Although not currently exposed to the user interface, the underlying architecture supports the creation of multiple files. This design decision was intentionally left open for future extension. In a potential educational setting, such a feature would allow instructors to design larger projects distributed across multiple files, improving readability and organization compared to a single monolithic source file.

The `.include` directive is implemented by resolving file names directly within this in-memory structure. Since file lookup is performed through a hash map, include resolution has an average-case complexity of $O(1)$, making the mechanism effectively independent of the number of files currently loaded in the virtual file system.

Conceptually, this structure behaves as a lightweight virtual file system (VFS), abstracting file access behind a simple interface and decoupling the assembler from the underlying storage mechanism. Although this system remains simple, it provides a flexible foundation for future extensions such as persistent projects, user-created source files, or more advanced project management features.

Chapter 4

Testing and Validation

4.1 Testing strategy

The project was validated through systematic manual testing. Each component was tested independently before being integrated into the complete application. Particular attention was given to the correctness of the Beta machine execution cycle and the assembly parser, as these components constitute the core functionality of the simulator.

4.2 Beta machine validation

The Beta machine was validated through a set of systematic manual test cases covering arithmetic operations, memory access, control flow, and program counter management. Every instruction type was tested independently before integration into larger programs in order to isolate potential sources of errors.

4.2.1 Arithmetic instructions

Arithmetic instructions were tested by verifying that operations such as addition and subtraction correctly produced expected results. The correctness of the ALU was validated by checking that computed values were properly written to the destination registers and could be subsequently retrieved without corruption. Edge cases involving boundary values were also tested to ensure consistent behavior.

4.2.2 Memory instructions

Memory-related instructions were validated by testing both read and write operations. The simulator was verified to correctly store values in the internal memory structure and retrieve them when required. Additional tests were performed to ensure that invalid memory accesses were either handled gracefully or produced a controlled failure depending on the configuration of the simulator. This also confirmed that the internal memory model remained consistent after multiple consecutive operations.

4.2.3 Branch and jump instructions

Control flow instructions were tested with particular attention to label resolution and program counter updates. Both forward and backward label references were used to verify correct address resolution in the assembler. Special cases were also tested where a single assembly instruction expands into multiple binary instructions due to macro expansion. These cases ensured that branch offsets remained correct even when instruction size was not uniform at the source level. The program counter behavior was validated for all execution paths, ensuring that it increments by 4 for sequential execution and is correctly updated in the case of jumps and branches.

4.2.4 Program correctness validation

Full execution traces were used to validate the overall consistency of the Beta machine. Small assembly programs were executed step-by-step to verify that register states, memory state, and program counter transitions matched the expected theoretical behavior of the Beta architecture.

4.3 Assembler validation

The assembler was validated through a collection of manual test programs designed to exercise all supported language constructs. Particular attention was given to label resolution, macro expansion, expression evaluation, and error detection, as these components form the core functionality of the compilation process.

4.3.1 Expression evaluation

Arithmetic expressions were tested using combinations of operators with different precedence levels. The generated binary values were compared against manually computed results to verify the correctness of the AST construction and expression evaluation process. Operator precedence and parenthesized expressions were tested to ensure that the AST correctly reflected the intended order of evaluation.

4.3.2 Label resolution

Label handling was validated using both forward and backward references. Programs containing jumps to labels defined before and after the current instruction were assembled and executed to verify that the generated addresses were correct. Additional tests were performed using instructions generated through macro expansion. Since a single assembly instruction may expand into multiple machine instructions, these tests verified that the two-pass assembly process correctly computed label addresses despite variable instruction sizes.

4.3.3 Macro expansion

Macros were tested using different types or numbers of parameters. The generated binary output was compared with the expected instruction encoding to verify that parameter substitution and macro execution were functioning correctly. The recognition of each instruction defined in `beta.uasm` was tested.

4.3.4 Include directives

The `.include` directive was validated by distributing variable definitions and macro definitions across multiple files. These tests verified that symbols defined in included files were correctly accessible from the including file. Include cycle detection was also tested to ensure that recursive inclusion generated an appropriate compilation error rather than causing infinite recursion.

4.3.5 Error handling

Invalid assembly programs were intentionally created to verify the robustness of the parser and assembler. These tests included syntax errors, unknown macros, incorrect numbers of macro arguments, undefined labels, and malformed expressions.

4.3.6 Validation through execution

In addition to testing the assembler independently, the generated machine code was executed on the Beta simulator. Successful execution of representative assembly programs provided end-

to-end validation of the complete compilation process, from source code to machine execution.

4.4 User interface validation

The graphical user interface was validated through manual interaction testing. Since the user interface primarily serves as a visualization and interaction layer, testing focused on usability, responsiveness, and synchronization between the different components of the application.

4.4.1 Code editor

The code editor was tested using assembly programs of various sizes. These tests verified that text editing, scrolling, selection, and code modification behaved as expected. Many tests of the integration of the Monaco editor on the web platform were performed. The interaction between the editor and the surrounding layout components was tested extensively to ensure that resizing operations and editor interactions remained stable.

4.4.2 Compilation workflow

The complete workflow from source code editing to program execution was tested repeatedly. After modifying the assembly code, compilation was triggered, and the resulting machine state was inspected to verify that changes were correctly propagated through the assembler and into the Beta machine. These tests ensured that the editor, assembler, and simulator remained properly synchronized.

4.4.3 Layout components

The custom layout components were tested independently.

The SplitView component was tested by repeatedly resizing panels and switching between single-panel and dual-panel modes, and with different children widgets. These tests verified that the layout remained stable and that editor interactions did not interfere with resizing operations.

The TabView component was tested by opening, selecting, and switching between tabs. These tests ensured that the correct file contents were displayed and that modifications remained associated with the appropriate tab.

4.4.4 Programmer and schematic views

The Beta machine views were tested by executing programs step-by-step and verifying that displayed values matched the internal state of the simulator.

4.4.5 Cross-platform testing

The application was tested on both desktop and web environments. Although some limitations were observed on the desktop version due to differences in Monaco editor behavior, the web version, the primary deployment target of the project, was successfully validated and remained stable during normal usage.

4.5 Limitations

No automated test suite was implemented during the project. While Flutter provides support for unit and widget testing, the project's priority was the implementation of the simulator itself. Consequently, verification relied on manual testing. Future work could include the development of automated unit tests for the assembler and the Beta machine components.

Chapter 5

Conclusions

5.1 Results

Figure 5.1 presents the final application. The interface is divided into two main areas: the assembly editor on the left and the Beta machine visualization on the right.

The editor allows users to write and modify assembly programs using the syntax supported by the custom assembler. Once assembled, the generated machine code is loaded into the simulator and can be executed interactively.

The visualization panel displays the internal state of the processor during execution. In particular, it provides access to the disassembled instructions, the register file, and the contents of memory. These elements are updated dynamically after each execution cycle, allowing users to observe the effects of each instruction on the processor state.

The example shown in Figure 5.1 illustrates a program that has already executed several instructions. The modifications visible in the register file and memory confirm that the simulator correctly propagates the effects of executed instructions throughout the machine state. In this example, the first memory location has been overwritten with the value `0`. Since this address originally contained the first instruction of the program, the modification is immediately reflected in the disassembly view, demonstrating the shared-memory nature of the Von Neumann architecture implemented by the Beta processor.

Overall, the final application successfully achieves the objectives defined at the beginning of the project. It provides a functional assembler, an executable Beta machine simulator, and an interactive visualization environment reproducing the core functionality of the original BSim software.

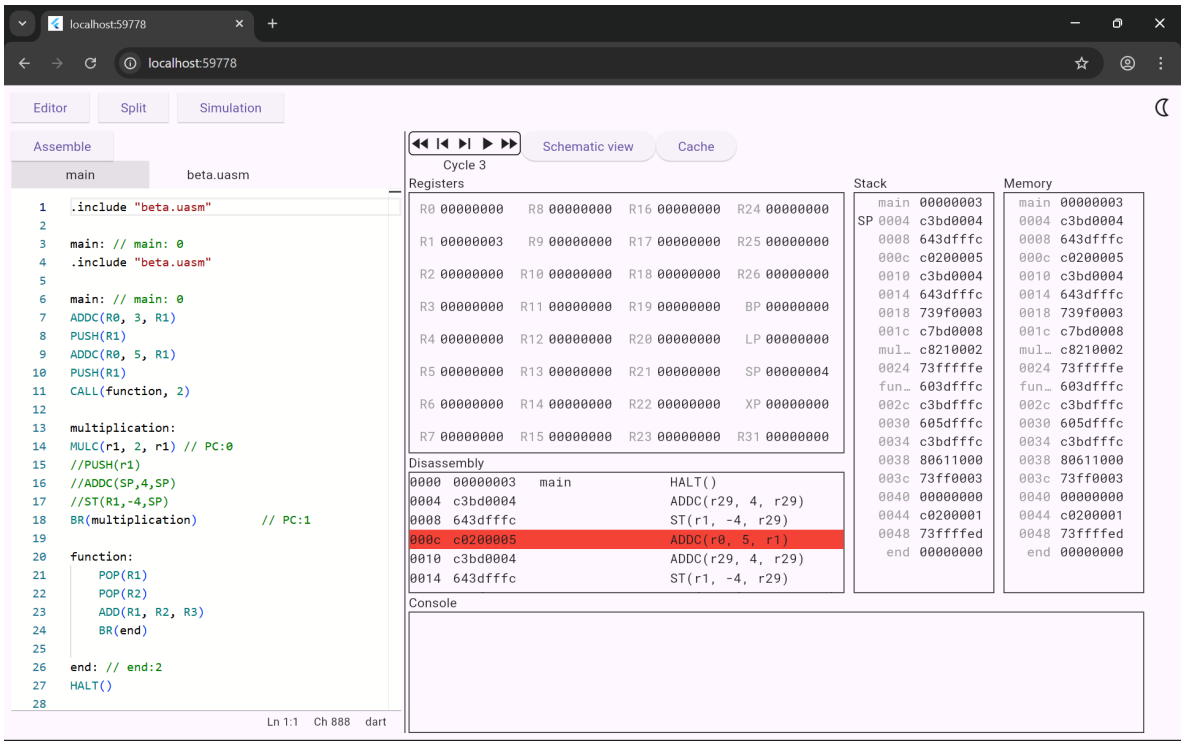

Figure 5.1: Final application(light theme)

5.2 Security

Since the application executes entirely on the client side and does not rely on user accounts, databases, or external services, its attack surface remains limited. The primary risks are related to resource exhaustion through excessively large assembly programs, recursive macro expansions, or infinite execution loops. These issues can be mitigated by introducing limits on program size, recursion depth, and execution cycles. Furthermore, the use of an in-memory virtual file system prevents unauthorized access to the host file system through assembly include directives.

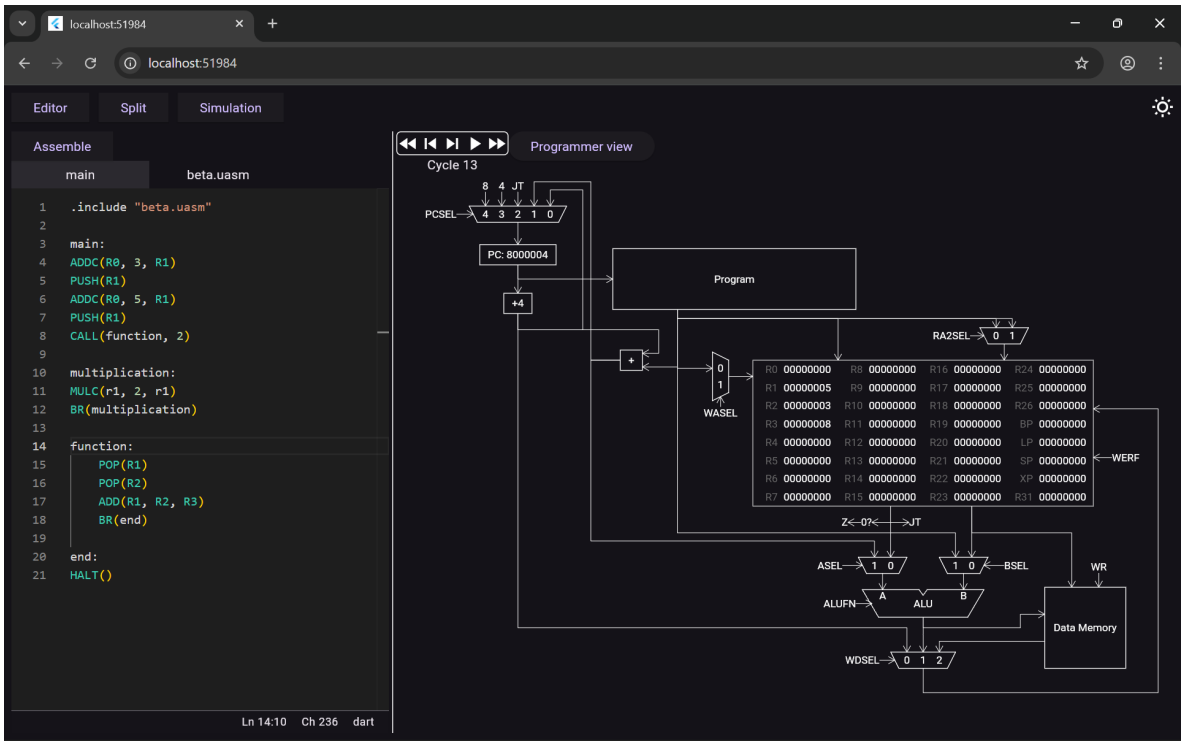

Figure 5.2: Final application (dark theme)

5.3 Perspectives

Several improvements can be considered to extend the current implementation and enhance both the usability and the educational value of the simulator.

A first improvement concerns the schematic visualization of the Beta machine. Currently, the visualization focuses on the structural representation of the processor. A possible extension would be to display additional runtime information directly on the schematic, such as the binary values circulating through internal wires, the current instruction being executed, and selected memory contents. This would provide a closer correspondence between the execution state of the processor and its graphical representation, making the simulator more useful as a teaching tool.

Another important area of future work is the implementation of cache memory. The current simulator accesses memory directly and therefore does not model any memory hierarchy. While this simplification is consistent with the original objectives of the project, adding cache support would allow the simulation of cache hits, cache misses, and replacement policies. Such an extension would increase the realism of the simulator and provide additional educational opportunities related to computer architecture.

Significant improvements could also be made to the code editor component. A custom editor could be developed to provide tighter integration with the assembler and simulator. Such an editor would ideally support syntax highlighting, accurate line numbering, real-time

parsing, incremental error detection, and direct visualization of compilation errors within the source code view.

Implementing such an editor would require introducing a more sophisticated text-processing pipeline capable of efficiently handling large files. In particular, incremental tokenization and streaming text analysis would be necessary to maintain good performance while providing immediate feedback to the user. Although this would considerably increase the complexity of the frontend architecture, it would also provide a more integrated and user-friendly development experience.

Finally, the current virtual file system could be extended to support user-created files and project organization features. This would allow larger assembly projects to be split across multiple source files and would further improve the scalability of the educational environment.

These perspectives highlight several directions for evolving the simulator beyond its current state. While the application already fulfills its primary objective of reproducing the functionality of BSim, these extensions could transform it into a more complete platform for teaching assembly programming and computer architecture. processor simulation.

Bibliography

https://api.flutter.dev/flutter/material/ThemeData/brightness.html
https://api.flutter.dev/flutter/material/SnackBar-class.html
https://pub.dev/packages/flutter_code_editor/install
https://pub.dev/packages/flutter_monaco
https://stackoverflow.com/questions/17306250/what-does-the-align-x86-assembler-directive-do-exactly
https://stackoverflow.com/questions/61863958/how-can-i-paint-a-widget-on-a-canvas-in-flutter
https://stackoverflow.com/questions/49388281/flutter-vertical-divider-and-horizontal-divider
https://stackoverflow.com/questions/53234825/what-is-the-difference-between-functions-and-classes-to-create-reusable-widgets
https://computationstructures.org/exercises/sandboxes/bsim.html
https://computationstructures.org/notes/pdfs/betainst.pdf
https://computationstructures.org/notes/pdfs/betadiagram.pdf
https://computationstructures.org/notes/pdfs/betadiagram.pdf