ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado
Ingeniería Informática — Ingeniería del Software

WASM2010: WebAssembly port of ASM2010

Realizado por
Guillermo Diz Gil

Dirigido por
José Antonio Pérez Castellanos

Departamento
Lenguajes y Sistemas Informáticos

Sevilla, septiembre de 2022

Resumen

WASM2010 es un conjunto de herramientas que proporciona al usuario un entorno de experimentación para el CS2010, una arquitectura hardware diseñada por la Escuela Técnica Superior de Ingeniería Informática de la Universidad de Sevilla con propósitos académicos. Compuesto en su base de un ensamblador y un emulador, brinda al usuario la posibilidad de ensamblar programas para, posteriormente, cargarlos en una máquina CS2010 virtual y comprender su funcionamiento mediante la depuración paso a paso o la ejecución en tiempo real, así como de interaccionar con el ordenador usando dispositivos de entrada y salida.

Mientras que la interacción con el usuario se origina en una aplicación web, su núcleo se encuentra aislado en un módulo WebAssembly —un *bytecode* portable de bajo nivel— escrito en C, permitiendo así su reutilización en proyectos web y de escritorio que deseen incorporar o extender su funcionalidad.

Palabras clave: CS2010, ensamblador, emulador, WebAssembly

Abstract

WASM2010 is a set of tools that provides the user with an experimental environment for CS2010, a hardware architecture designed by the Higher Technical School of Computer Engineering of the University of Seville for academic purposes. Composed at its core of an assembler and an emulator, it allows the user to assemble programs, load them into a virtual CS2010 machine, and understand their operation through step-by-step debugging or real-time execution, as well as to interact with the computer using input and output devices.

While the user interaction originates in a web application, its core is isolated in a WebAssembly module —a low-level portable bytecode— written in C, thus allowing its reuse in web and desktop projects that wish to incorporate or extend its functionality.

Keywords: CS2010, assembler, emulator, WebAssembly

Agradecimientos

No puedo agradecer lo suficiente a mis padres y mi familia por criarme, cuidarme y orientarme a lo largo de toda mi vida. A mi tutor, por ser un referente y alimentar mi sed de saber con su experiencia. A mis amigos por ayudarme, animarme y alegrar mis días, y también a todos los que han contribuido a este proyecto de alguna manera.

Acknowledgements

I cannot express enough how thankful I am to my parents and family for raising me, taking care of me and guiding me throughout my life. Thanks to my tutor for becoming a reference and feeding my thirst for knowledge with his own. Finally, thanks to my friends for keeping my spirits high and brightening my days, and to everyone who contributed to this project somehow.

Contents

List of Figures

List of Tables

LIST OF EQUATIONS

LIST OF CODE

1 Introduction

1.1 Motivation

In modern informatics, computers have grown significantly in complexity and continue growing over the years. Current instruction sets count on refined macroinstructions to efficiently solve common problems that modern CPUs must handle, and they keep evolving to provide better support for new technologies. However, this also makes it difficult to understand how things work under the hood. This struggle becomes prominent for ISAs based on a CISC design, like today's standard in personal computers, x86-64. The rise of RISC ISAs like ARM or RISC-V does not relieve the situation, though they are indeed simpler, as they focus on being performant to carry out real-life operations.

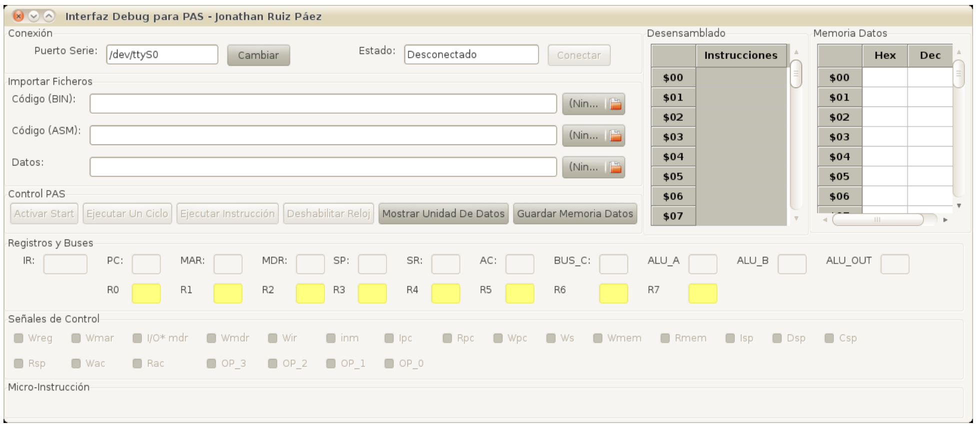
Historically, colleges have used simpler ISAs on academic grounds, allowing students to understand how the data and control flow is happening inside the computer at the expense of performance optimisations. CS2010 (in Spanish, *Computador Simple 2010*) is the ISA proposed by the Higher Technical School of Computer Engineering of the University of Seville as the final step of a series of simple register-based academic computers that show the student the basics of computer architecture.

Despite being simple enough for educational purposes, keeping track of all the details during a program execution still entails an effort not worth it for a person, even for simple programs of a few lines of source assembly code, thus rendering the ISA less valuable. The need for a toolkit providing this functionality comes to light.

1.2 State of the art

A deep search on the Internet reveals that there has been little work on this matter. In 2012, Jonathan Ruiz Páez implemented a hardware and software framework for a simple academic processor, *Interfaz Debug para PAS* [1]. While this processor seems to be a clone of CS2010, some drawbacks prevent it from being functional. The simulation is carried out on a Digilent Basys 2 prototyping board, already discontinued and no longer for sale [2]. Therefore, a user who does not have this board cannot use the toolkit. It seems impossible to download the software, and its source code seems inaccessible too.

Although it could be helpful for other purposes, those drawbacks render this software unusable in most environments, leaving the users with no opportunity to run a CS2010 simulation.



Puerto Serie: /dev/ttyS0

Estado: Desconectado

Figure 1-1. Interfaz Debug para PAS's main view

1.3 Goals

This project's primary goal is to successfully design and develop a software toolkit capable of assembling and executing CS2010 programs. The toolkit's kernel should be a native library that can be reused in other software and run on almost every modern host platform, even in environments with limited resources. Apart from the native library, a front-end application must be developed to let the users use the toolkit conveniently. This front-end application will run in the browser as a web application in order to maximize compatibility and ease access to the toolkit.

We expect the front-end application to use the library as any other software would, ensuring that the provided API is enough for any program to work with the CS2010 and providing an actual implementation for the library's use cases. To communicate the native library with the web application, we will investigate and use the WebAssembly capabilities that modern browsers ship. As it is difficult for a web application to use the library directly, an adapter will be needed to interact with it effectively. This adapter will be part of the development and must be generic enough to be reusable in future works, so it must provide an interface to ease the use of native libraries and programs from the web.

The toolkit must be able to parse well-formed CS2010 assembly and transform it into CS2010 machine code according to the official specifications so that the machine can execute it step by step, allowing the user to observe the machine's state at every point. It should be easy to keep track of the changes after each instruction. Even if we expect the user to have some knowledge of CS2010 before using the toolkit, it should be straightforward to use and self-explainable.

1.4 Structure

This document is divided into chapters. The first chapter introduces the main subject to the reader, while each of the following chapters focuses on a different topic.

In Chapter 2, we will provide an in-depth definition of the CS2010 architecture, following the official documents of the University of Seville. It will serve as a basis for understanding the project requirements.

Chapter 3 covers some formal aspects of the project, such as the methodology used throughout the development, estimation and planning of tasks and cost analysis.

Chapter 4 introduces the project's analysis and the elicited requirements of the final system and each of its parts, including functional and non-functional requirements.

Chapter 5 discusses the design and implementation followed to achieve the final product and the acceptance tests to verify its correct functioning.

Chapter 6 comprises a set of guides on how to build and deploy the project.

Chapter 7 provides a user guide for using the final product.

Eventually, Chapter 8 summarises the conclusions and presents some ideas for future work.

On the other hand, appendices will provide definitions, explanations or clarifications regarding the topics covered in the main chapters.

Appendix A will introduce some basics of digital electronics systems, such as the definition of the register transfer notation used numerous times throughout the document.

2 CS2010 ARCHITECTURE IMPLEMENTATION

In this chapter, we will describe the specifications of the target architecture by using an actual implementation. We will comprehensively define the digital system using the RT language and discuss implementation-specific details.

2.1 Overview

The CS2010 is a general-purpose machine based on the Harvard architecture and inspired by the 8-bit Atmega328P, though being way more superficial than the latter. Its instruction set is based on a small subset of the AVR 8-bit instructions [3]. Its assembly can produce AVR-compatible programs with little to no modifications in the source code. However, both architectures are incompatible at the machine code level, meaning that no assembled program for the CS2010 could run in the AVR and vice versa.

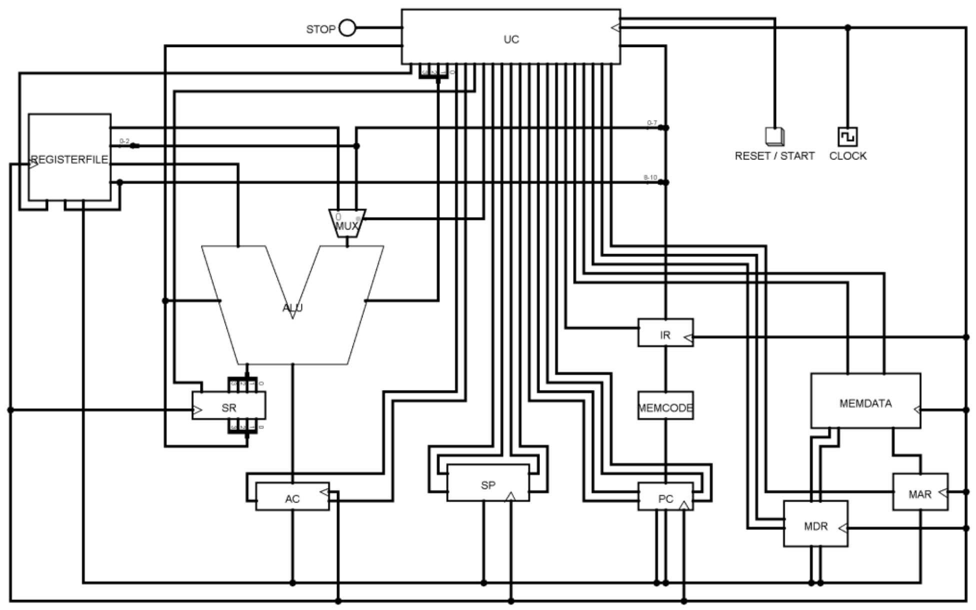


Figure 2-1. Outermost view of a complete CS2010 implementation

2.2 Memories

Following Harvard architecture, two separate memories coexist within the computer.

2.2.1 Random-access memory

The random-access memory (RAM) features 8-bit wide 256 words, allowing reading and writing at any address with no time penalty, thus rendering access-related hazards inexistent. The memory provides an 8-bit channel to provide an address and a shared 8-bit channel to output the content of the matching cell or to receive the content that will be stored. Every address is mapped directly to its corresponding physical cell; the computer does not provide complex addressing mechanisms such as virtual memory, pagination or segmentation.

W	**R**	$\mathbf{A}_{/8}$	$\mathbf{{}^*D}_{/8}$ **:=**	$\mathbf{RAM[A]}_{/8} \leftarrow$
0	0	-	HI	-
0	1	-	RAM[A]	-
1	0	-	HI	D

Table 2-1. RTL description of RAM

2.2.2 Read-only memory

In contrast to the RAM and following the instruction format —covered in future sections— the read-only memory (ROM) features 256 words of 16 bits, each containing a complete machine instruction that the machine can address and execute.

$\mathbf{A}_{/8}$	$\mathbf{D}_{/16}$ **:=**
-	ROM[A]

Table 2-2. RTL description of ROM

2.3 Registers

CS2010 features a set of registers whose content can be used as operands for the main arithmetic and logic operations, either explicitly or implicitly. Since CS2010 is a register-based computer, registers also store the result of any computation. Except for the instruction register (IR) and the state register (SR), which are 16 and 4-bit wide, all registers are 8-bit wide, following the architecture's data path width and making it easier to transfer values through the data pipeline.

2.3.1 General-purpose registers

Eight general-purpose registers are visible to the user, all named RX, with X being an integer number ranging from 0 to 7. The user is free to change the content of these registers at will, as the architecture does not give them any internal use. Since they are not the implicit target of any operation, it is safe to assume that their content will remain permanent unless the user performs an operation that explicitly targets them.

W	**SA**$_{/3}$	**SB**$_{/3}$	**SW**$_{/3}$	**IN**$_{/8}$	**A**$_{/8}$ **:=**	**B**$_{/8}$ **:=**	**RF[SW]**$_{/8}$ $\leftarrow$
0	-	-	-	-	RF[SA]	RF[SB]	-
1	-	-	-	-	RF[SA]	RF[SB]	IN

Table 2-3. RTL description of the general-purpose register file

2.3.2 Specific-purpose registers

Some registers are specific to the machine's state and cannot be used as the target of instructions, but their content can still be mutated as the side effect of some instructions' execution.

- **Program counter (PC).** It contains the address of the next instruction to execute. It is always initialised to zero when the machine starts and increases by one after each instruction. After it reaches the maximum value, it wraps to zero. Arbitrary addresses can also be loaded when a branching instruction executes.
- **Status register (SR).** It is a 4-bit wide register where each bit represents whether some condition has been met after the last logical-arithmetic operation. These flags will help decide whether the machine should take a conditional branch, as we will see in later sections.
    - **Bit 0 (C flag, carry).** Register's least significant bit. It indicates whether an arithmetic carry or borrow has been generated or stores the shifted bit when a shift operation occurs.
    - **Bit 1 (Z flag, zero).** The result has all its bits to zero.
    - **Bit 2 (N flag, negative).** When represented in two's complement, it signals a negative result.
    - **Bit 3 (V flag, overflow).** It is impossible to represent the result in two's complement due to overflow, i.e., the

sum of two positive numbers has resulted in a negative number or vice versa, or a shift operation has generated a result with a different sign than the source.

- **Stack pointer (SP).** It works as the basis for implementing a stack in the machine. As with most implementations, the stack grows towards lower memory addresses, reducing the odds of colliding with the rest of the user-managed memory (the heap). It points at the highest address possible in memory (i.e., 0xFF) and decreases in one as the effect of calling a subroutine. The SP will always point to a free memory address where a new stack frame can be stored.

  Note that SP can decrease indefinitely and will wrap to the highest address after reaching zero. Thus, if SP reaches the heap or a programmer inadvertently overwrites the stack, it is likely to cause unexpected behaviour in the user program. Overwriting the stack frame might cause the SP to point to an arbitrary address, breaking the program flow completely. While some defensive programming techniques aim to avoid these problems, it is reasonable to think that these hazards are unlikely to happen due to the low complexity of CS2010's programs.

I	**W**	**R**	**$OUT_{/8}$:=**	**$*IB_{/8}$:=**	**$PC_{/8} \leftarrow$**
-	-	0	PC	HI	-
-	-	1	PC	PC	-
0	0	-	-	-	-
0	1	-	-	-	IB
1	0	-	-	-	PC + 1

Table 2-4. RTL description of PC

W	**V_{in}**	**N_{in}**	**Z_{in}**	**C_{in}**	**$SR_3 \leftarrow$**	**$SR_2 \leftarrow$**	**$SR_1 \leftarrow$**	**$SR_0 \leftarrow$**
0	-	-	-	-	-	-	-	-
1	-	-	-	-	V_{in}	N_{in}	Z_{in}	C_{in}

Table 2-5. RTL description of SR (inputs)

W	**V_{out} :=**	**N_{out} :=**	**Z_{out} :=**	**C_{out} :=**
-	SR_3	SR_2	SR_1	SR_0

Table 2-6. RTL description of SR (outputs)

I	D	C	R	$*IB_{/8}$:=	$SP_{/8} \leftarrow$
-	-	-	0	HI	-
-	-	-	1	SP	-
0	0	0	-	-	-
0	0	1	-	-	0xFF
0	1	0	-	-	SP - 1
1	0	0	-	-	SP + 1

Table 2-7. RTL description of SP

2.3.3 Hidden registers

Similar to specific-purpose registers, all registers that fall under this category are not manipulable by the user. Their use is restricted to the machine's internal operation, and their work is transparent from the user's perspective.

- **Instruction register (IR).** It is a 16-bit register containing the instruction being executed. At the same time that a given instruction executes its last cycle, the instruction pointed out by the PC is loaded in this register.
- **Memory data register (MDR).** It is an 8-bit register that serves as an intermediary buffer between the main memory and the data path, allowing the exchange of data between the RAM and the rest of the computer.
- **Memory address register (MAR).** It is an 8-bit register that addresses the RAM. Before reading or writing a given cell in the main memory, its address must be loaded in this register.
- **Accumulator register (AC).** It is an 8-bit wide, temporary register where the result of the operations carried out by the ALU is stored, preventing the data from being spread directly into the data path.

W	$IN_{/8}$	$OUT_{/8}$:=	$IR_{/8}$
0	-	IR	-
1	-	IR	IN

Table 2-8. RTL description of IR

W	**I/O**	**$^*\mathrm{IB}_{/8}$:=**	**$^*\mathrm{EB}_{/8}$:=**	**$\mathrm{MDR}_{/8} \leftarrow$**
0	0	HI	MDR	-
0	1	MDR	MDR	-
1	0	HI	MDR	IB
1	1	HI	HI	EB

Table 2-9. RTL description of MDR

W	**$\mathrm{IN}_{/8}$**	**$\mathrm{OUT}_{/8}$:=**	**$\mathrm{MAR}_{/8} \leftarrow$**
0	-	MAR	-
1	-	MAR	IN

Table 2-10. RTL description of MAR

W	**R**	**$\mathrm{IN}_{/8}$**	**$^*\mathrm{IB}_{/8}$:=**	**$\mathrm{AC}_{/8} \leftarrow$**
0	0	-	HI	-
-	1	-	AC	-
1	-	-	-	IN

Table 2-11. RTL description of AC

2.4 Arithmetic logic unit

The CS2010's arithmetic logic unit (ALU) is a fundamental block of the architecture, providing the principal operations the computer can carry out. It supplies several working modes that can be switched using a specific bit signal. Most of these modes use two inputs (also called operands) to produce the result, while others might use only one input. Some of these modes do not produce a valuable result, but they update flags in the status register, which is directly connected to the ALU. For this case, we will define the set S_{out} in Equation 2-1 as the collection of ALU's status-related outputs so it is easier to describe the behaviour of some instructions.

$$S_{out} = \{V_{out}, N_{out}, Z_{out}, C_{out}\}$$

Equation 2-1. Set of ALU's status outputs

Analogously, in Equation 2-2, we will define S_{in} as the set containing all the ALU's status-related input flags.

$$S_{in} = \{V_{in}, N_{in}, Z_{in}, C_{in}\}$$

Equation 2-2. Set of ALU's status inputs

OP$_{3\text{-}0}$				**A$_{/8}$**	**B$_{/8}$**	**S$_{in/4}$**	**R$_{/8}$:=**
0	0	-	0	-	-	-	-
0	0	1	1	-	-	-	-
0	1	0	0	-	-	-	$SHR(A, C_{in})$
0	1	0	1	-	-	-	$SHL(A, C_{in})$
0	1	1	-	-	-	-	A
1	0	0	-	-	-	-	$(A + B)\ mod\ 2^8$
1	0	1	-	-	-	-	$(A - B)\ mod\ 2^8$
1	1	-	-	-	-	-	B

Table 2-12. RTL description of ALU (results)

OP$_{3\text{-}0}$				**V$_{out}$:=**	**N$_{out}$=**	**Z$_{out}$:=**	**C$_{out}$:=**
0	0	-	0	V_{in}	N_{in}	Z_{in}	0
0	0	1	1	$\mathrm{V_{in}}$	N_{in}	$\mathrm{Z_{in}}$	1
0	1	0	0	$C_{in} \oplus A_7$	R_7	$\wedge_{e=0}^{7} \overline{R_e}$	A_0
0	1	0	1	$A_7 \oplus A_6$	R_7	$\wedge_{e=0}^{7} \overline{R_e}$	A_7
0	1	1	-	-	-	-	-
1	0	0	-	$SVF(A, B)$	R_7	$\wedge_{e=0}^{7} \overline{R_e}$	$UVF(A, B)$
1	0	1	-	$SVF(A, -B)$	R_7	$\wedge_{e=0}^{7} \overline{R_e}$	$\overline{\mathrm{UVF(A, -B)}}$
1	1	-	-	-	-	-	-

Table 2-13. RTL description of ALU (flags)

Where SVF can be defined as a function such that:

$$SVF(A_{/n}, B_{/n}) = \begin{cases} 1 & if\, sgn(A) = sgn(B) \wedge sgn(A) \neq sgn(A + B) \\ 0 & otherwise \end{cases}$$

Equation 2-3. Signed addition overflow function

Note how SVF represents an overflow after two signed numbers have been added, i.e., the impossibility of representing the result using the same number of bits as the operands.

Given that A and B are two signed binary numbers represented using two's complement, we can define the sign function as:

$$sgn(X_{/n}) = X_{n-1}$$

Equation 2-4. Two's complement sign function

A signed overflow can only occur when both operands share the same sign, as the result of adding two numbers with different signs will always be in the range formed by the operands. Consequently, taking Equation 2-4 into Equation 2-3 and expanding it using Boolean algebra, we can get the following definition of SVF:

$$\mathrm{SVF}(\mathrm{A}_{/\mathrm{n}}, \mathrm{B}_{/\mathrm{n}}) = \mathrm{A}_{\mathrm{n}-1} \wedge \mathrm{B}_{\mathrm{n}-1} \wedge \overline{(\mathrm{A}+\mathrm{B})_{\mathrm{n}-1}} \vee \overline{\mathrm{A}_{\mathrm{n}-1}} \wedge \overline{B_{n-1}} \wedge (\mathrm{A}+\mathrm{B})_{\mathrm{n}-1}$$

Equation 2-5. Two's complement signed addition overflow function

Equation 2-5 is probably the simplest way to implement this flag in logic circuits and modern programming languages. On the other hand, UVF is a function defined as follows:

$$\mathrm{UVF}(A_{/n}, B_{/n}) = \begin{cases} 1 & if\ 2^n \leq A+B \\ 0 & otherwise \end{cases}$$

Equation 2-6. Unsigned addition overflow function

However, sometimes the result of adding two unsigned n-bit words wraps around, i.e., the result exhibits modulo behaviour, as the C programming language defines. In that case, we can provide an equivalent definition using modular arithmetics:

$$\mathrm{UVF}(A_{/n}, B_{/n}) = \begin{cases} 1 & if\ (A+B)\ mod\ 2^n < A\ mod\ 2^n \\ 0 & otherwise \end{cases}$$

Equation 2-7. Modular unsigned addition overflow function

This is, in fact, a carry function. In real implementations, it is typical to see this function as the carry output of a ripple carry adder circuit. This circuit comprises up to n full-adder circuits cascaded in parallel, with n being the word's width to add.

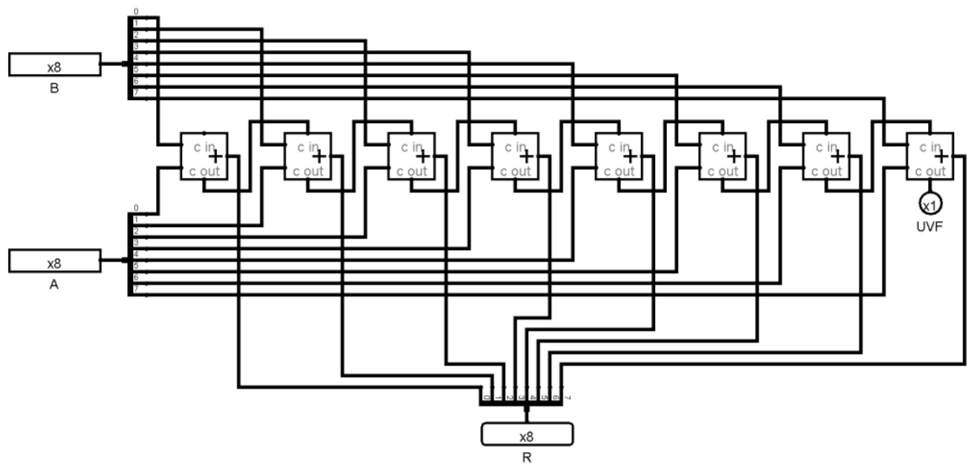


Figure 2-2. Typical 8-bit ripple carry adder

Taking this idea into account, we can define UVF as a recursive function:

$$\mathrm{UVF}(X_{/n}, Y_{/n}) = \begin{cases} X_{n-1} \wedge Y_{n-1} & if\, n = 1 \\ X_{n-1} \wedge Y_{n-1} & otherwise \\ \vee\, UVF(X_{/n-1}, Y_{/n-1}) \wedge (X_{n-1} \oplus Y_{n-1}) & \end{cases}$$

Equation 2-8. Recursive unsigned addition overflow function

Note that while the ALU uses UVF for adding and subtracting, the function needs to be negated when it is used for subtractions. This is a convention used in many platforms, such as x86 [4], that consists of setting the carry flag when a borrow occurs in a subtraction (i.e. A and B are subtracted and $A < B$) and clear it otherwise. Other platforms, such as ARM, PowerPC and System/360, use the result of UVF for substractions as they do for additions (without negating it), which means that the carry flag will be set when A and B are subtracted and $A \geq B$.

2.5 Buses

Buses are one of the essential parts of the data path. They connect all the functional units, allowing the data to flow through the processor. In the previous sections, we provided the RTL definition of all the units in the data path. However, the definition would be incomplete without knowing which inputs and outputs are connected. We will summarise the data path in

Figure 2-3 and provide a formal description of all the connections in two tables.

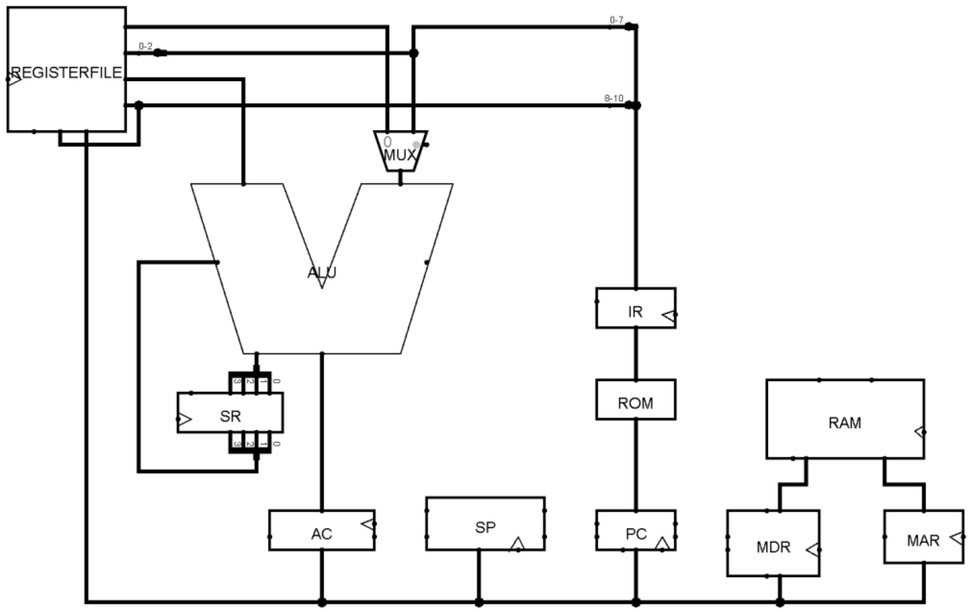


Figure 2-3. Functional units connected to their corresponding buses

Note how every signal from the control unit (CU), including the clock signal, has been excluded from Figure 2-3.

Two different types of buses can be observed along the data path:

- **Two-state buses (directed buses).** This type of buses have, at maximum, one output pin connected. Consequently, they always carry an active value. The output pin is also named *emitter*, while every input pin connected to the bus is named *receiver*. Sometimes, it is useful to switch from different output pins emitting simultaneously, in which case a multiplexer is used to select which output will be present on the bus. The signal for the multiplexer usually comes from the control unit and is not directly controlled by the functional units in the data path.
- **Three-state buses (shared buses).** One of the most significant limitations of two-state buses is that they do not allow values from multiple output pins without multiplexing. Here, three-state buses are handy as they add a new state, high impedance (HI), which is the state used to represent that no value is present in the bus. Every output pin connected to this bus must be able to stay in a high impedance state, in which case it disconnects from the circuit. Only one output pin can write a value simultaneously (i.e. not being in the HI state), preventing bus contention from happening. However, input pins can still read the bus at any time.

Name	**Connection**	**Range**	**Mux**
RAM addressing	MAR(OUT) – RAM(A)	0 – 7	-
ROM addressing	PC(OUT) – ROM(A)	0 – 7	-
Instruction bus	ROM(D) – IR(IN)	0 – 15	-
Immediate value	IR(OUT) – ALU(B)	0 – 7	INM = 1
Register B value	RF(B) – ALU(B)	0 – 7	INM = 0
Register A value	RF(A) – ALU(A)	0 – 7	-
Register B selector	IR(OUT) – RF(SB)	0 – 2	-
Register A selector	IR(OUT) – RF(SA)	8 – 10	-
Write register selector	IR(OUT) – RF(SW)	8 – 10	-
Accumulator bus	ALU(R) – AC(IN)	0 – 7	-
Current C flag	SR(C_{out}) – ALU(C_{in})	0	-
Current Z flag	SR(Z_{in}) – ALU(Z_{in})	0	-
Current N flag	SR(N_{in}) – ALU(N_{in})	0	-
Current V flag	SR(V_{in}) – ALU(V_{in})	0	-
New C flag	ALU(C_{out}) – SR(C_{in})	0	-
New Z flag	ALU(Z_{out}) – SR(Z_{in})	0	-
New N flag	ALU(N_{out}) – SR(N_{in})	0	-
New V flag	ALU(V_{out}) – SR(V_{in})	0	-

Table 2-14. List of two-state buses

Alias	**Connections**	**Range**

Global bus	MAR(IN)
	MDR(*IB)
	PC(*IB)
	SP(*IB)
	AC(*IB)
	RF(IN)
		0 – 7

RAM content	RAM(*D)
	MDR(*EB)
		0 – 7

Table 2-15. List of three-state buses

2.6 Instruction set

We will describe the current instruction set available for the computer and the side effects each instruction has after being executed. How the computer must behave to perform instructions is out of the scope and will be treated in the following sections.

2.6.1 Instruction format

As seen previously, all CS2010 instructions have a fixed width of 16 bits. While there are many instruction formats, the five most significant bits of the instruction are reserved for the operation code (opcode), which identifies the operation to be performed. The rest of the bits have different meanings depending on the instruction format, as detailed in Table 2-16.

Format	**15 — 11**	**10 — 8**	**7 — 3**	**2 — 0**
A	Operation code	Reg_a	—	Reg_b
B	Operation code	Reg_a	Immediate / address	
C	Operation code	Condition	Address	
D	Operation code	—		

Table 2-16. Instruction formats

In general, the most significant byte of the instruction contains metadata; in contrast, the least significant byte contains the actual payload of the instruction in most cases, if applicable. We will thoroughly explain each of the existing formats and their differences.

Format A, namely *register addressing* format, uses two numerical operands that point to general-purpose registers. Some unary instructions might not use the second operand, namely Reg_b. In general, Reg_a is the target of the action, whereas Reg_b is the source of the data.

Format B follows two different schemas: while the operand in the high byte is always a numerical value pointing to a register, its second operand can be either an immediate value or a memory address. Therefore, it is named *immediate and direct addressing* format as the type of the last operand is up to the instruction.

Format C is used only for branching instructions; thus, we will call it *immediate branching* format. As with format A, some instructions that perform unconditional branching might not use the condition.

Format D is the simplest, as it does not use any operand. It is called *implied format*, as the target of the operation is already implicit in the instruction itself.

2.6.2 Data-transfer instructions

Opcode	Name	Format	Syntax	Effect
00000	ST	A	ST (Ra), Rb	RAM[[Ra]] ← Rb
00001	LD	A	LD Ra, (Rb)	Ra ← RAM[[Rb]]
00010	STS	B	STS Addr, Rb	RAM[Addr] ← Rb
00011	LDS	B	LDS Ra, Addr	Ra ← RAM[Addr]
01111	MOV	A	MOV Ra, Rb	Ra ← Rb
11111	LDI	B	LDI Ra, Im	Ra ← Im

Table 2-17. Description of data transfer instructions

2.6.3 Arithmetic and logical instructions

Opcode	Name	Format	Syntax	Effect

01000	ADD	A	ADD Ra, Rb	Ra ← Ra + Rb
				SR ← ALU(S_{out})

01010	SUB	A	SUB Ra, Rb	Ra ← Ra – Rb
				SR ← ALU(S_{out})

01011	CP	A	CP Ra, Rb	Ra - Rb SR ← ALU(S_{out})
10010	CLC	D	CLC	SR(C) ← 0
10011	SEC	D	SEC	SR(C) ← 1

10100	ROR	A	ROR Ra	Ra ← SHR(Ra, SR(C_{out}))
				SR ← ALU(S_{out})

10101	ROL	A	ROL Ra	Ra ← SHR(Ra, SR(C_{out}))
				SR ← ALU(S_{out})

11000	ADDI	B	ADDI Ra, Im	Ra ← Ra + Im
				SR ← ALU(S_{out})

11010	SUBI	B	SUBI Ra, Im	Ra ← Ra - Im
				SR ← ALU(S_{out})

11011	CPI	B	CPI Ra, Im	Ra - Im SR ← ALU(S_{out})

Table 2-18. Description of arithmetic and logical instructions

2.6.4 Branching instructions

Opcode	Name	Format	Syntax	Effect
00100	CALL	C	CALL Addr	MEM[[SP]] 🡐 PC SP 🡐 SP – 1 PC 🡐 Addr
00101	RET	D	RET	SP 🡐 SP + 1 PC 🡐 MEM[[SP]]
00110	BRcc	C	BRcc Addr	cc: PC 🡐 Addr
00111	JMP	C	JMP Addr	PC 🡐 Addr

Table 2-19. Branching instructions

2.6.5 Special instructions

Opcode	Name	Format	Syntax	Effect
10111	STOP	D	STOP	—

Table 2-20. Special instructions

2.6.6 Conditional jumping

The instruction *BRcc* defined in Table 2-19 represents a conditional branching where *cc* can be one of the conditions presented in Table 2-21. Each condition is computed exclusively using the content of the status register; the value column shows how every condition is computed. If the chosen condition does not hold when the instruction is executed, the execution will have no side effects apart from fetching the next instruction.

Bit encoding	Condition name (cc)	Value
000	EQ	$SR(Z)$
001	CS / LO	$SR(C)$
010	VS	$SR(V)$
011	LT	$SR(N) \oplus SR(V)$

Table 2-21. Conditions for *BRxx*

Understanding when these conditions hold can be tricky, as a condition depends on the programmer's intentions to be meaningful. For example, the subtraction of two integers will trigger certain flags and make some conditions hold, independently of the signedness. But whether the programmer considers the operands signed or unsigned can give a different meaning to these conditions. We will provide a comprehensive list of the usual meanings of each condition after specific operations.

The condition *EQ* holds if:

- After subtraction, the result is 0 (i.e., operands are equal).
- After addition, the result is 0 (i.e., operands are complementary).
- After a shift, the result is 0.

The condition *CS / LO* holds if:

- After subtraction, the minuend is lower than the subtrahend, assuming unsigned operands (i.e., a borrow occurs).
- After addition, the result wraps around (i.e., a carry/unsigned overflow occurs).
- After a shift, the shifted bit is 1.
- The instruction *SEC* is executed.

The condition *VS* holds if:

- After subtraction or addition, the result is not representable in two's complement (i.e., a signed overflow occurs).

The condition *LT* holds if:

- After subtraction, A is lower than B, assuming signed operands (i.e., a borrow occurs).

2.7 Control unit

The control unit (CU) manages the computer state by receiving the output signals of all the functional units in the data path and signalling their inputs. In section 2.6, we described the instruction set and the behaviour that each instruction exhibits, although we did not describe how the computer could achieve the desired side effects.

For each instruction, a series of micro-operations will perform. Each micro-operation corresponds to a clock cycle, thus making some instructions take longer to execute than others. The last cycle of each instruction is also used to fetch the following instruction independently from the executing micro-operation, although this might not hold in micro-operations that use the PC. In this case, an extra clock cycle is added to fetch the following instruction.

While instructions can be implemented in multiple ways, this document will stick to the definition given in [5].

2.7.1 Implementation of data-transfer instructions

Cycle	Micro-operation	Flags
1	AC 🡐 Rb	AC(W), ALU(OP_3), ALU(OP_2)

2	MAR 🡐 AC
	AC 🡐 Ra
		MAR(W), AC(R)
		AC(W), ALU(OP_2), ALU(OP_1)

3	MDR 🡐 AC	MDR(W), AC(R)
4	RAM[[MAR]] 🡐 MDR	RAM(W)

Table 2-22. Micro-operations for an ST instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Rb	AC(W), ALU(OP_3), ALU(OP_2)

2	MAR 🡐 AC
	AC 🡐 Ra
		MAR(W), AC(R)
		AC(W), ALU(OP_2), ALU(OP_1)

3	MDR 🡐 RAM[[MAR]]	MDR(W), MDR(I/O), RAM(R)
4	Ra 🡐 MDR	RF(W), MDR(I/O)

Table 2-23. Micro-operations for an LD instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Im	AC(W), ALU(OP_3), ALU(OP_2) INM

2	MAR 🡐 AC
	AC 🡐 Ra
		MAR(W), AC(R)
		AC(W), ALU(OP_2), ALU(OP_1)

3	MDR 🡐 AC	MDR(W), AC(R)
4	RAM[[MAR]] 🡐 MDR	RAM(W)

Table 2-24. Micro-operations for an STS instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Im	AC(W), ALU(OP_3), ALU(OP_2) INM

2	MAR 🡐 AC
	AC 🡐 Ra
		MAR(W), AC(R)
		AC(W), ALU(OP_2), ALU(OP_1)

3	MDR 🡐 RAM[[MAR]]	MDR(W), AC(R)
4	Ra 🡐 MDR	RF(W), MDR(I/O)

Table 2-25. Micro-operations for an LDS instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Rb	AC(W), ALU(OP_3), ALU(OP_2)
2	Ra 🡐 AC	AC(R), RF(W)

Table 2-26. Micro-operations for a MOV instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Im	AC(W), ALU(OP_3), ALU(OP_2) INM
2	Ra 🡐 AC	AC(R), RF(W)

Table 2-27. Micro-operations for an LDI instruction

2.7.2 Implementation of branching instructions

Cycle	Micro-operation	Flags
1	MDR 🡐 PC AC 🡐 Im	MDR(W), PC(R) AC(W), ALU(OP_3), ALU(OP_2), INM

2	MAR 🡐 SP
	SP 🡐 SP – 1
		MAR(W), SP(R)
		SP(D)

3	PC 🡐 AC
	RAM[[MAR]] 🡐 MDR
		PC(W), AC(R)
		RAM(W)

Table 2-28. Micro-operations for a CALL instruction

Cycle	Micro-operation	Flags
1	SP 🡐 SP + 1	SP(I)
2	MAR 🡐 SP	MAR(W), SP(R)
3	MDR 🡐 RAM[[MAR]]	MDR(W), MDR(I/O), RAM(R)
4	PC 🡐 MDR	PC(W), MDR(I/O)

Table 2-29. Micro-operations for a RET instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Im	AC(W), ALU(OP_3), ALU(OP_2), INM
2	cc: PC 🡐 AC	cc: PC(W), AC(R)

Table 2-30. Micro-operations for a BRcc instruction

Cycle	Micro-operation	Flags
1	AC 🡐 Im	AC(W), ALU(OP_3), ALU(OP_2), INM
2	PC 🡐 AC	PC(W), AC(R)

Table 2-31. Micro-operations for a JUMP instruction

2.7.3 Implementation of arithmetic and logical instructions

Cycle	Micro-operation	Flags

1	AC 🡐 Ra + Rb mod 2^8
	SR 🡐 ALU(S_{out})
		AC(W), ALU(OP_3)
		SR(W)

2	Ra 🡐 AC	RF(W), AC(R)

Table 2-32. Micro-operations for an ADD instruction

Cycle	Micro-operation	Flags

1	AC 🡐 Ra - Rb mod 2^8
	SR 🡐 ALU(S_{out})
		AC(W), ALU(OP_3), ALU(OP_1)
		SR(W)

2	Ra 🡐 AC	RF(W), AC(R)

Table 2-33. Micro-operations for a SUB instruction

Cycle	Micro-operation	Flags

1	AC ← Ra - Rb mod 2^8
	SR ← ALU(S_{out})
		AC(W), ALU(OP_3), ALU(OP_1)
		SR(W)

Table 2-34. Micro-operations for a CP instruction

Cycle	Micro-operation	Flags
1	SR(C) ← 0	SR(W)

Table 2-35. Micro-operations for a CLC instruction

Cycle	Micro-operation	Flags
1	SR(C) ← 1	SR(W), ALU(OP_1), ALU(OP_0)

Table 2-36. Micro-operations for an SEC instruction

Cycle	Micro-operation	Flags

1	AC ← SHR(Ra, SR(C))
	SR ← ALU(S_{out})
		AC(W), ALU(OP_2), INM
		SR(W)

2	Ra ← AC	RF(W), AC(R)

Table 2-37. Micro-operations for a ROR instruction

Cycle	Micro-operation	Flags

1	AC ← SHL(Ra, SR(C))
	SR ← ALU(S_{out})
		AC(W), ALU(OP_2), ALU(OP_0), INM
		SR(W)

2	Ra ← AC	RF(W), AC(R)

Table 2-38. Micro-operations for a ROL instruction

Cycle	Micro-operation	Flags

1	AC ← Ra + Im mod 2^8
	SR ← ALU(S_{out})
		AC(W), ALU(OP_3), INM
		SR(W)

2	Ra ← AC	RF(W), AC(R)

Table 2-39. Micro-operations for an ADDI instruction

Cycle	Micro-operation	Flags
1	AC ← Ra - Im mod 2^8	AC(W), ALU(OP_3), ALU(OP_1), INM
	SR ← ALU(S_{out})	SR(W)
2	Ra ← AC	RF(W), AC(R)

Table 2-40. Micro-operations for a SUBI instruction

Cycle	Micro-operation	Flags
1	AC ← Ra - Im mod 2^8	AC(W), ALU(OP_3), ALU(OP_1), INM

Table 2-41. Micro-operations for a CPI instruction

2.7.4 Implementation of special instructions

Cycle	Micro-operation	Flags
1	—	—

Table 2-42. Micro-operations for a STOP instruction

Note that the STOP instruction is the only exception to the rule described in section 2.7. This instruction does not only present any side effects but does not trigger any instruction fetching, leading the CU to a sleep state which can only leave if the user sends a reset signal. This signal is expected to perform a soft reset on the computer, i.e., reset the PC and the SP registers, so the execution is restarted. However, a hard reset can also occur, wiping out the content of every memory and register in the computer.

Deciding which reset should occur is up to the implementation. For instance, a sample implementation of a soft reset might consist of setting the PC to zero and the SP to the highest address possible (i.e., 0xFF). These operations would allow the computer to fetch the first instruction in the ROM while wiping out any stack frame. Note that the memory content persists; it is up to the programmer not to trust the memory being filled with zeroes at the program's start.

2.8 Input and output

While there are no special instructions in the instruction set to interface with external devices, there is a possibility to map the external devices directly into the CS2010's RAM. The user does not need to be aware of how the external devices operate, as they can be written and read as other regular addresses in the memory.

This scheme has several implications. Foremost, we will need a device capable of determining whether an address belongs to the RAM or any external device. This device will be called the IO controller and act as a facade for both the RAM and the external devices, receiving read and write requests and deciding which device should handle the request.

Whether the IO controller and the external devices share their buses does not directly impact the computer's high-level design; thus is up to the implementation, as it is any design beyond the IO controller. Sharing buses might prevent bus contention since external devices do not share their data bus and do not need to support a high impedance state in their outputs. However, it would also be more expensive to implement due to the need for different buses.

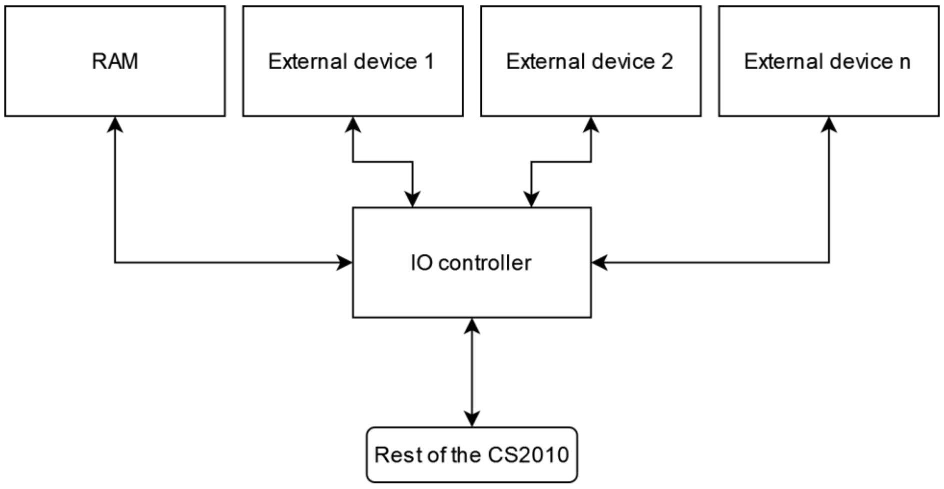


Figure 2-4. Input/output circuit scheme

Theoretically, the IO controller can map up to 256 external devices onto the main memory. The main drawback of this approach is that the user loses access to some addresses in the RAM. However, we can consider that the trade-off between decreasing available memory and providing input/output to the architecture is not relevant in the scope of most programs due to their simple nature.

3 Project planning

In this chapter, we will cover some formal aspects of the project. We will begin by explaining the methodology used throughout all the project stages, following up with an analysis of the planned tasks over time. Eventually, we will provide a cost analysis, including the total budget and foreseen variances.

3.1 Methodology

As stated in chapter 1.3, the final result of this project comprises several products. On the one hand, the project requires a core library that will provide emulation capabilities to recreate the CS2010 computer architecture. On the other hand, it also requires a front-end application so the end user can access these capabilities and, in turn, can use the library conveniently. In the same scope, developing a middleware that allows the front-end application to communicate with the library maximises decoupling and guarantees reusability, especially since the library and the application are built using different technologies.

Dividing the original project into three different subprojects imply that a special effort must be put into the design phase, as every subproject must offer an effective programming interface while maintaining the separation of concerns and being well decoupled. However, it is challenging to foresee the difficulties that each project will go through and which features will be missing without extensive prototyping, as requirements will need refinement and evolve as soon as any part of the project is developed.

3.1.1 Enough Design Up Front

The idea of using Enough Design Up Front (EDUF) arises after combining the need for having a solid high-level detailed design with the need for an agile development method. EDUF is a software development approach that states that each software project, depending on its size and its goals, has different design needs. In contrast with its siblings, Big Design Up Front (BDUF) and No Design Up Front (NDUF), it realizes that the need for changing requirements might arise as long as the software is still used. Therefore, it is only necessary to do the right amount of design and leave it flexible for future improvements and changes. [6]

While we might consider the toolkit done at some point, it should be flexible enough to allow future features and improvements to be implemented quickly, such as supporting a new version of the CS architecture series. This way, we can study the features that each CS version might share and produce a solid design that minimizes the number of changes needed to support both of them. Therefore, other minor elements, such as those related to the design of the front-end toolkit's user interface, will require less design effort and will be more flexible in their implementation.

3.1.2 Iterative and incremental development

While many other methodologies and frameworks exist, they are likely to be excessively bloated for a single-person team. Together with the previously defined EDUF and following the project's needs, a personal methodology based on iterative and incremental development will be used during every project phase. Some of the critical points of this methodology are:

- **Iterative cycles do not need to have a fixed duration.** Instead, the cycle duration will match the time it takes to do the proper research, implement a feature, test, clear up technical debt and update the documentation. However, the duration of a cycle must be adapted to deadlines. In this sense, milestones can help approach the correct timing.
- **Documentation must always be in sync with the actual design.** As stated previously, updating the documentation is essential to every cycle. Consequently, the documentation must reflect every addition or removal to any interface before the development is resumed. This practice avoids making too many breaking changes in the code, leaving the documentation outdated.
- **Finding difficulties signals the need for a redesign.** When an iterative cycle is stuck or might be in the foreseeable future, we must halt the development and put all the focus on redesigning. We can also find difficulties during the design: if we fail to see a proper strategy, it might be helpful to reconsider previous steps to understand the root of the problem.
- **Diagrams must reflect every design decision.** Diagrams should be enough to understand how every system is designed, i.e. which features it offers and how it can interoperate with others.

3.2 Task planning

In this section, we will provide a list of the tasks we must accomplish to consider the project finished, their scheduled and actual duration and their deviation according to the gathered data.

No.	Task	Planned duration	Actual duration	Deviation
1	CS2010 research	20 h	10 h	-10 h
2	WebAssembly research	15 h	20 h	5 h
3	Requirements elicitation	20 h	24 h	4 h
4	ASM2010 development	70 h	80 h	10 h
5	Jasm development	50 h	45 h	-5 h
6	WASM2010 development	60 h	59 h	-1 h
7	Testing	20 h	20 h	0 h
8	Deployment	10 h	12 h	2 h
9	Final report	35 h	40 h	5 h
	Total	**300 h**	**310 h**	**10 h**

Table 3-1. Planned tasks, duration and deviation

According to Table 2-1, the project was planned to be executed in 300 hours, distributed between researching, designing, developing, testing, deploying and the final report, but the actual duration of the project exceeded 10 hours of the planned time.

3.3 Cost analysis

In this section, we will provide a brief breakdown of the project's total cost, including direct and indirect costs. Also, we will compare the planned budget to the actual cost to determine deviations.

3.3.1 Direct costs

Any expense related to this project falls under this category, such as labour, software or tools. As any software used throughout the design and development is free, we will only count the wage a developer would have been paid according to salary statistics.

According to Glassdoor, the average annual gross income of a software engineer in Spain is close to 32000,00 € as of 2022 [7], so we will approximate a monthly gross salary of 2666,00 €. We will compute the gross cost of a single worker by supposing a standard eight-hour workday, twenty-two labour days a month, which makes a cost per hour of 16,70 € after rounding.

Task	Planned cost	Actual cost	Deviation
CS2010 research	334,00 €	167,00 €	-167,00 €
WebAssembly research	250,50 €	334,00 €	83,50 €
Requirements elicitation	334,00 €	400,80 €	66,80 €
ASM2010 development	1169,00 €	1336,00 €	167,00 €
Jasm development	835,00 €	751,00 €	-83,50 €
WASM2010 development	1002,00 €	985,30 €	-16,70 €
Testing	334,00 €	334,00 €	0,00 €
Deployment	167,00 €	200,40 €	33,40 €
Final report	584,50 €	668,00 €	83,50 €
Total	**5010,00 €**	**5176,50 €**	**167,00 €**

Table 3-2. Planned and actual direct costs and deviation

3.3.2 Indirect costs

In this section, we will account for those expenses not directly related to the project, such as the equipment used. However, a personal computer is the only equipment used to execute this project. As this is a long-duration active asset, we will calculate a simple amortization according to its initial cost and expected lifetime.

Supposing an initial cost of 750,00 € and an expected lifetime of 7 years, we can assume a monthly amortization of 8,90 €. Furthermore, if we suppose a regular 30-day month that lasts 720 hours, we can compute the total amortization during the project's lifetime, as shown in Table 2-1.

Equipment	Initial cost	Lifetime	Usage duration	Amortization
Personal computer	750,00 €	7 years	310 h	3,83 €
Total				**3,83 €**

Table 3-3. Equipment's amortization

4 Project analysis and requirements

This chapter will analyze the project's goals to elicit its requirements. We will first understand who the users will be and their expectations expressed through goals. Next, we will elicit the requirements needed to satisfy these needs, starting from a high-level view of the project and stepping into more detailed requirements.

4.1 Actors

As stated in 1.3, the goal is to develop a toolkit composed of a native library and web application. Therefore, it is reasonable to think that at least two actors, represented by physical users, will be interested in using each part of the toolkit.

However, the web application's need to use the native library from the browser conveniently brought the need for an adapter to light. Since every part of the project must be reusable —including the adapter— we will add a third actor to guarantee that this adapter remains generic enough so it can be reused in other external software.

We will list the series of actors and their goals to understand their needs better. Written from the user's perspective, they will help us understand which features the final software should have to fulfil their desires, thus easing the requirement elicitation process in the following sections.

Code	GL-001
Actor	Application user
Description	Write a CS2010 assembly program in an editor, assemble and run it interactively

Table 4-1. Application user's goals

Code	GL-002
Actor	Web developer
Description	Use the capabilities of native libraries/programs from web applications using WebAssembly

Table 4-2. Web developer's goals

Code	GL-003
Actor	Desktop developer
Description	Assemble CS2010 assembly code and emulate CS2010 in a native program

Table 4-3. Desktop developer's goals

4.2 General requirements

Now that we have described the actors' goals, we will provide general requirements for each component to consider the system's acceptance. These provide an overview of the functionality they must provide, which we will use to extract functional and non-functional requirements in the following sections.

Code	GR-001
Component	Web application
Description	It must provide an in-browser assembly editor and the ability to show the state of the CS2010 emulation provided by the native library

Table 4-4. General requirement 1

Code	GR-002
Component	Adapter
Description	It must provide the web application with a better mechanism to read and write the memory of the WebAssembly virtual machine where the native library is executed so they can share the emulation's state effectively

Table 4-5. General requirement 2

Code	GR-003
Component	Native library
Description	It must provide assembly and emulation capabilities accessible from an interface that can be used by other native programs or WebAssembly-powered browsers

Table 4-6. General requirement 3

4.1 Architectural requirements

We will provide a high-level overview of the architecture of the final system and requirements to understand how the main interactions between each component will occur, according to Figure 4-1.

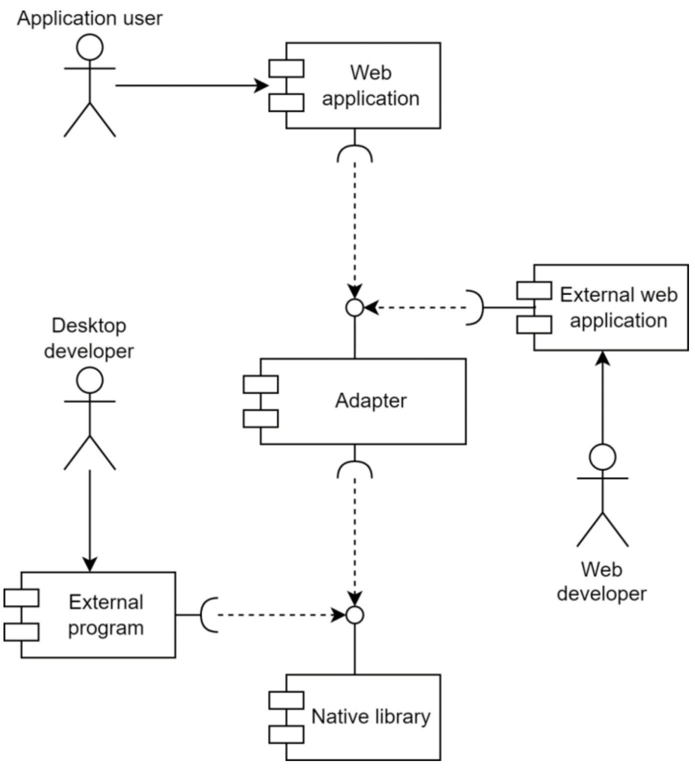


Figure 4-1. Overview of the system's architecture

Code	AR-001
Component	Web application
Description	It must use the adapter to achieve an efficient communication channel with the native library

Table 4-7. Architectural requirement 1

Code	AR-002
Component	Adapter
Description	It must offer a generic interface to every web application

Table 4-8. Architectural requirement 2

Code	AR-003
Component	Adapter
Description	It must be able to interact with any native library using their interface

Table 4-9. Architectural requirement 3

Code	AR-004
Component	Native library
Description	It must offer a valid interface for external programs and the adapter to use it

Table 4-10. Architectural requirement 4

4.2 Functional requirements

This section will break down each general requirement into a series of fine-grained requirements. Each requirement specifies a single behaviour the respective component must expose to be satisfied. This list of behaviours details the functionality of the system.

We will follow a bottom-up approach during the elicitation, following the system's architecture described in Figure 4-1. To provide a clearer view, we will use different sections to separate them by components.

4.2.1 Native library

Code	FR-001
Title	Assemble CS2010 assembly code
Component	Native library
Description	It must provide a way to assemble compliant CS2010 source assembly into machine code

Table 4-11. Functional requirement 1

Code	FR-002
Title	Retrieve assembly status
Component	Native library
Description	It must provide a way to retrieve the status of the assembly process, such as errors and warnings

Table 4-12. Functional requirement 2

Code	FR-003
Title	Disassemble CS2010 machine code
Component	Native library
Description	It must provide a way to disassemble CS2010 machine code into compliant source assembly

Table 4-13. Functional requirement 3

Code	FR-004
Title	Instantiate emulation sandboxes
Component	Native library
Description	It must provide a way to ìnstantiate multiple sandboxed CS2010 emulation instances

Table 4-14. Functional requirement 4

Code	FR-005
Title	Load CS2010 programs
Component	Native library
Description	It must provide a way to load a compliant CS2010 program provided as raw machine code onto a given emulation instance, setting everything up so the execution of the new program can begin immediately after it

Table 4-15. Functional requirement 5

Code	FR-006
Title	Handle mapped I/O devices
Component	Native library
Description	It must provide a way to map and unmap external devices to an emulation instance dynamically on a given emulation instance, i.e. letting the user control which addresses are mapped to I/O, as well as giving complete control over how the external devices will behave during read-and-write operations

Table 4-16. Functional requirement 6

Code	FR-007
Title	Perform a soft and hard reset
Component	Native library
Description	It must provide a way to perform a hard reset on a given emulation instance, i.e. clearing all memories and setting registers to default values, as well as a soft reset, which will only reset the necessary registers to start over the execution

Table 4-17. Functional requirement 7

Code	FR-008
Title	Clear memory
Component	Native library
Description	It must provide a way to clear each memory on a given emulation instance

Table 4-18. Functional requirement 8

Code	FR-009
Title	Reset register
Component	Native library
Description	It must provide a way to reset each register on a given emulation instance

Table 4-19. Functional requirement 9

Code	FR-010
Title	Step microinstruction (clock cycle)
Component	Native library
Description	It must provide a way to forward the execution by a single microinstruction (clock cycle) in a given emulation instance

Table 4-20. Functional requirement 10

Code	FR-011
Title	Step instruction
Component	Native library
Description	It must provide a way to forward the execution by a single instruction in a given emulation instance

Table 4-21. Functional requirement 11

Code	FR-012
Title	Step block of instructions
Component	Native library
Description	• It must provide a way to forward the execution by a block of instructions in a given emulation instance, i.e. steps instructions indefinitely until one of the following conditions is met: o The computer fetches a branching instruction. o The computer halts the execution due to a STOP instruction. o The computer reaches the limit of executed instructions before halting.

Table 4-22. Functional requirement 12

4.2.2 Adapter

Code	FR-013
Title	Encode/decode basic types
Component	Adapter
Description	It must provide a way to read and write the memory of the WebAssembly virtual machine, converting native C types to JavaScript types and vice versa, according to the official WebAssembly C ABI [8]

Table 4-23. Functional requirement 13

Code	FR-014
Title	Encode/decode strings
Component	Adapter
Description	It must provide a way to read and write ASCII-encoded strings from and to the WebAssembly virtual machine, according to the official WebAssembly C ABI [8]

Table 4-24. Functional requirement 14

Code	FR-015
Title	Compute aggregate structures
Component	Adapter
Description	It must provide a way to compute the alignment of C structures and unions, according to the official WebAssembly C ABI [8]

Table 4-25. Functional requirement 15

Code	FR-016
Title	Encode/decode aggregate structures
Component	Adapter
Description	It must provide a way to read and write C structures and unions from and to the WebAssembly virtual machine, according to the schema computed in FR-017

Table 4-26. Functional requirement 16

Code	FR-017
Title	Handle pointers
Component	Adapter
Description	It must provide a way to operate with pointers, providing common mechanisms such as pointer indirection

Table 4-27. Functional requirement 17

4.2.3 Web application

Code	FR-018
Title	Assembly text editor
Component	Web application
Description	It must provide a built-in assembly editor where the user can perform basic text operations

Table 4-28. Functional requirement 18

Code	FR-019
Title	File-related operations
Component	Web application
Description	It must provide a way to create, load and save source assembly files in the user's hard drive

Table 4-29. Functional requirement 19

Code	FR-020
Title	See output machine code
Component	Web application
Description	It must show the output machine code after assembling the source code, using a customizable number radix

Table 4-30. Functional requirement 20

Code	FR-021
Title	Track output errors
Component	Web application
Description	It must show errors and warnings after assembling the source code, indicating which line produced the error

Table 4-31. Functional requirement 21

Code	FR-022
Title	Start/stop the emulation
Component	Web application
Description	It must provide a way to start a new emulation of the output machine code once it is already valid, as well as a way to stop and wipe it

Table 4-32. Functional requirement 22

Code	FR-023
Title	Track source code execution
Component	Web application
Description	It must provide a way to track which line of the source code is being executed at a given time

Table 4-33. Functional requirement 23

Code	FR-024
Title	Inspect CS2010's ROM
Component	Web application
Description	It must provide a view to inspect the content of the CS2010 emulator's ROM, using a customizable number radix and marking the address pointed by the PC. The content of each address must be shown as source assembly code too

Table 4-34. Functional requirement 24

Code	FR-025
Title	Inspect CS2010's RAM
Component	Web application
Description	It must provide a view to inspect the content of the CS2010 emulator's RAM, using a customizable number radix and marking the addresses pointed by SP and MAR

Table 4-35. Functional requirement 25

Code	FR-026
Title	Inspect CS2010's registers
Component	Web application
Description	It must provide a view to inspect the content of every register in the CS2010 emulator, using a customizable number radix

Table 4-36. Functional requirement 26

Code	FR-027
Title	Inspect CS2010's registers
Component	Web application
Description	It must provide a view to inspect the content of every register in the CS2010 emulator, using a customizable number radix

Table 4-37. Functional requirement 27

Code	FR-028
Title	Inspect CS2010's signals
Component	Web application
Description	It must provide a view to inspect the status of every signal in the CS2010 emulator

Table 4-38. Functional requirement 28

Code	FR-029
Title	Interact with I/O devices
Component	Web application
Description	It must provide a way to interact (providing input and seeing the output) with simple I/O devices connected to the emulator, whose mappings in memory can be customized by the user

Table 4-39. Functional requirement 29

Code	FR-030
Title	Reset the CS2010
Component	Web application
Description	It must provide a way to perform both a soft and hard reset on the emulator

Table 4-40. Functional requirement 30

Code	FR-031
Title	Manually control the emulation
Component	Web application
Description	It must provide a way to forward the emulation by microinstructions (clock cycles), instructions or blocks of instructions (as described in Functional requirement 1)

Table 4-41. Functional requirement 31

Code	FR-032
Title	Start/stop the CS2010's internal clock
Component	Web application
Description	It must provide a way to start/stop the CS2010's internal clock, which will run the emulation at a fixed frequency customizable by the user

Table 4-42. Functional requirement 32

4.3 Non-functional requirements

In this section, we will list the requirements that do not directly describe the component's behaviour but place constraints on how it behaves. Most of these requirements are related to performance and usability. Again, we will follow a bottom-up approach.

Code	NFR-001
Title	C99-compliant and platform independent
Component	Native library
Description	It must be C99-compliant, according to the ISO/IEC 9899:1999 [9], as well as multiplatform (Windows, Unix-like, WASI)

Table 4-43. Non-functional requirement 1

Code	NFR-002
Title	Zero-dependency, lightweight, high-performance
Component	Native library
Description	The source code and binary library must be lightweight and perform well in performance terms. The library must not link with any other dependencies except for the C standard library

Table 4-44. Non-functional requirement 2

Code	NFR-003
Title	Minimal performance overhead at runtime
Component	Adapter
Description	It must reduce the performance overhead to the minimum, particularly on read-and-write operations (runtime operations), even when this implies increasing the space complexity in a tradeoff for a minor time complexity

Table 4-45. Non-functional requirement 3

Code	NFR-004
Title	Browser compatibility
Component	Web application
Description	The web application must run in any modern standard browser, such as Mozilla Firefox, Google Chrome, Opera, Safari or Microsoft Edge released since 2018

Table 4-46. Non-functional requirement 4

Code	NFR-005
Title	Simple and minimalist user interface
Component	Web application
Description	It must provide a simple and minimalist user interface, preferring black-and-white tones over vibrant colours that might distract the user. The information displayed must always remain precise and quick to seek, lowering the number of clicks needed to perform any action

Table 4-47. Non-functional requirement 5

Code	NFR-006
Title	Responsive user interface
Component	Web application
Description	It must provide a responsive user interface so that it adapts to tablet-sized and desktop-sized screens—mobile-sized screens do not need to be supported

Table 4-48. Non-functional requirement 6

4.4 Use cases

We will provide an overview of the system's use cases using UML diagrams. Each diagram will display every use case in a given component.

Additionally, we will provide a textual description of each use case, describing the series of actions that a given actor must take to achieve a given goal.

As with section 4.2, we will separate every use case diagram and textual definition into categories according to their component.

4.4.1 Native library

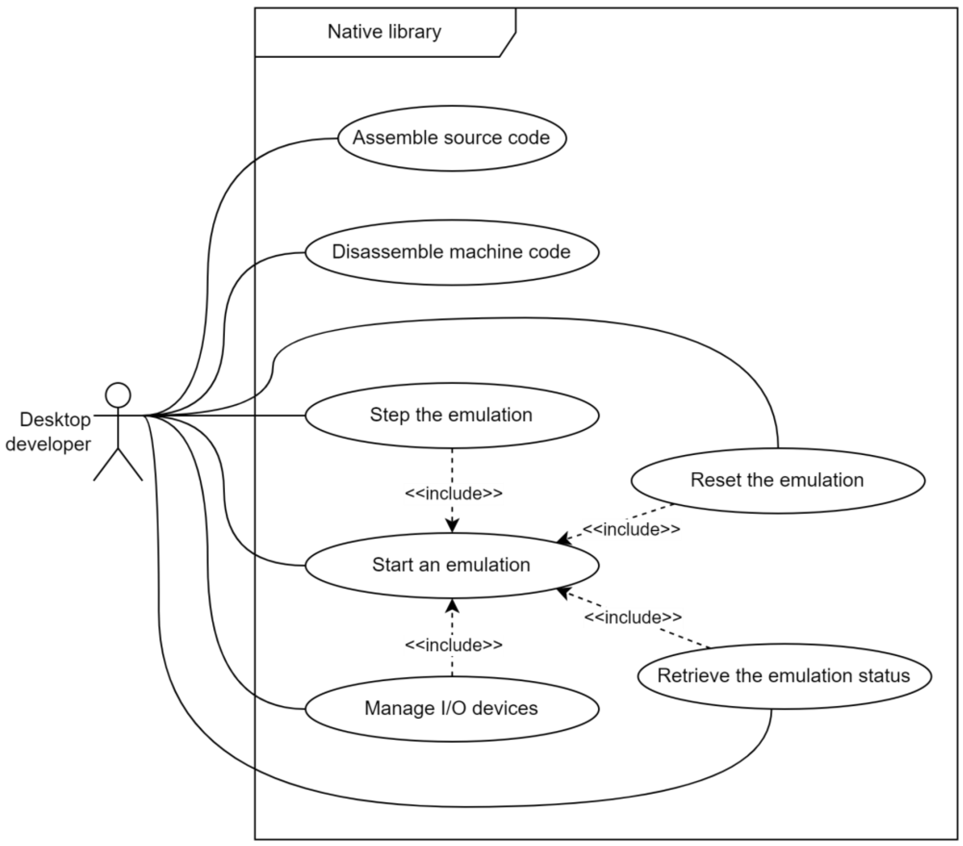


Figure 4-2. Native library's use cases

Code	UC-001
Title	Assemble source code
Component	Native library
Pre-condition	The user provides CS2010 assembly code
Path	1. A new assembly unit is created 2. The assembly unit parses each source line independently 3. Valid assembly code lines are assembled into machine code
Post-condition	The assembly unit contains valid machine code and a log containing errors and warnings that occurred during the assembly The assembly unit returns a status code

Table 4-49. Use case 1

Code	UC-002
Title	Disassemble machine code
Component	Native library
Pre-condition	The user provides a CS2010 machine code instruction (result of UC-001)
Path	1. The machine code instruction is translated to its correspondent assembly line
Post-condition	The user is provided with a valid assembly line

Table 4-50. Use case 2

Code	UC-003
Title	Start an emulation
Component	Native library
Pre-condition	The user provides a series of CS2010 machine code instructions (the result of UC-001)
Path	1. A new emulation instance is created 2. Memories and registers inside the instance are cleared and set up for a proper emulation 3. The machine code is loaded onto the ROM
Post-condition	The emulation instance is ready to begin the execution

Table 4-51. Use case 3

Code	UC-004
Title	Step the emulation
Component	Native library
Pre-condition	UC-003 has been completed
Path	1. The user forwards either: • A clock cycle (microinstruction) • An instruction • A block of instructions 2. The emulation instance executes the corresponding operations
Post-condition	The machine's state has been changed accordingly

Table 4-52. Use case 4

Code	UC-005
Title	Reset the emulation
Component	Native library
Pre-condition	UC-003 has been completed
Path	1. The user performs either: • A soft reset • A hard reset 2. The emulation instance clears and resets the corresponding memories and registers
Post-condition	The machine's state has been changed accordingly

Table 4-53. Use case 5

Code	UC-006
Title	Retrieve the emulation status
Component	Native library
Pre-condition	UC-003 has been completed
Path	1. The user consults the state of any part of the emulation instance
Post-condition	The user retrieves valid and consistent information

Table 4-54. Use case 6

Code	UC-007
Title	Manage I/O devices
Component	Native library
Pre-condition	UC-003 has been completed
Path	1. The user tells the emulation instance to use a given I/O handler where the external device logic is implemented 2. The emulation instance changes its internal I/O handler
Post-condition	The emulation instance's I/O handler has been changed

Table 4-55. Use case 7

4.4.2 Adapter

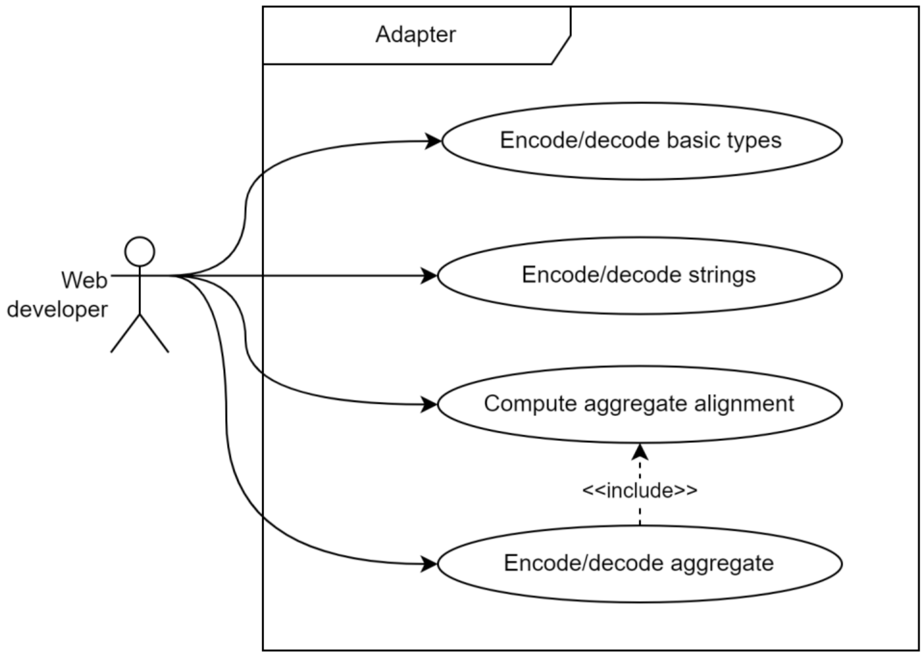


Figure 4-3. Adapter's use cases

Code	UC-008
Title	Encode/decode basic types
Component	Adapter
Pre-condition	A valid WebAssembly instance has been instantiated An adapter instance has been instantiated
Path	1. The user tells the adapter to read/write the content at a given address 2. The adapter performs a conversion between JavaScript types and C types 3. The adapter reads/writes the content at a given address
Post-condition	The content at the given address is read or written

Table 4-56. Use case 8

Code	UC-009
Title	Encode/decode strings
Component	Adapter
Pre-condition	A valid WebAssembly instance has been instantiated An adapter instance has been instantiated
Path	1. The user tells the adapter to read/write a string at a given address 2. The adapter computes the length of the string to read/write 3. The adapter reads/writes the content at the starting address until the string is completely read/written
Post-condition	The string at the given address is read or written

Table 4-57. Use case 9

Code	UC-010
Title	Compute aggregate alignment
Component	Adapter
Pre-condition	An adapter instance has been instantiated
Path	1. The user tells the adapter to compute the alignments of a given aggregate 2. The adapter computes the alignment and padding for the given aggregate according to the WebAssembly C ABI 3. If the aggregate contains any other aggregate whose alignments have not been computed yet, it will compute the alignment of the child aggregate recursively
Post-condition	The alignments of the aggregate and all its members are computed The aggregate and all its members are ready to be encoded/decoded

Table 4-58. Use case 10

Code	UC-011
Title	Encode/decode aggregate
Component	Adapter
Pre-condition	An adapter instance has been instantiated UC-010 has been completed
Path	1. The user tells the adapter to read/write an aggregate at a given address 2. The adapter reads/writes the content of each member in the aggregate recursively 3. If the user performs a reading operation, the adapter will build up a new JavaScript object following the aggregate's structure and will place the content in each member
Post-condition	The content of the aggregate is read or written

Table 4-59. Use case 11

4.4.3 Web application

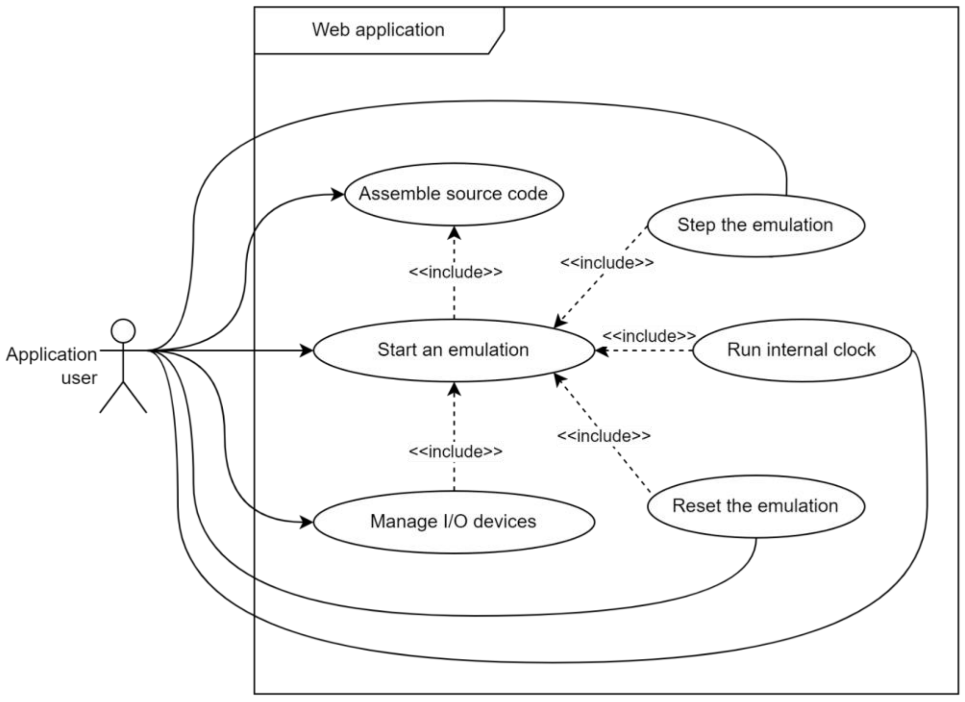


Figure 4-4. Web application's use cases

Code	UC-012
Title	Assemble source code
Component	Web application
Pre-condition	There is no current emulation running
Path	1. The user writes assembly code in the built-in text editor 2. After a few seconds, a new assembly unit is created 3. The source assembly code is parsed and assembled in the assembly unit 4. The output machine code is displayed together with errors and warnings
Post-condition	The assembly unit contains valid machine code If the source assembly contains errors, the emulation capabilities are blocked

Table 4-60. Use case 12

Code	UC-013
Title	Start an emulation
Component	Web application
Pre-condition	UC-012 has been completed successfully
Path	1. The user clicks on the button to start emulating the assembled machine code 2. The assembly editor view becomes read-only until the emulation is finished 3. A new emulation instance is created and set up, and the program is loaded on the ROM
Post-condition	The emulation instance is ready to begin the execution The emulation view is presented to the user

Table 4-61. Use case 13

Code	UC-014
Title	Step the emulation
Component	Web application
Pre-condition	UC-013 has been completed successfully The machine’s internal clock is not running
Path	1. The user clicks on the button to forward either: • A clock cycle (microinstruction) • An instruction • A block of instructions 2. The emulation instance executes the corresponding operations
Post-condition	The machine’s state has been changed accordingly The user interface refreshes the changes immediately

Table 4-62. Use case 14

Code	UC-015
Title	Run internal clock
Component	Web application
Pre-condition	UC-013 has been completed successfully
Path	1. The user sets the desired internal clock’s frequency 2. The user clicks on the button to start the internal clock 3. Manual stepping and other features are disabled while the internal clock is running 4. The emulation instance starts the internal clock and keeps it running until the user stops it
Post-condition	The machine’s internal clock is started Some parts of the user interface are disabled

Table 4-63. Use case 15

Code	UC-016
Title	Reset the emulation
Component	Web application
Pre-condition	UC-013 has been completed successfully The machine's internal clock is not running
Path	1. The user performs either: • A soft reset • A hard reset 2. The emulation instance clears and resets the corresponding memories and registers
Post-condition	The machine's state has been changed accordingly The user interface refreshes the changes immediately

Table 4-64. Use case 16

Code	UC-017
Title	Manage I/O devices
Component	Web application
Pre-condition	UC-013 has been completed successfully The machine's internal clock is not running
Path	1. The user enters the settings panel 2. The user maps/unmaps an I/O device at a given address 3. The user confirms the changes
Post-condition	The device is mapped/unmapped immediately The user interface refreshes the changes immediately

Table 4-65. Use case 17

5 Design and implementation

In this chapter, we will comprehensively analyse the design and implementation of each component according to the system architecture provided in the project's requirements.

5.1 System architecture

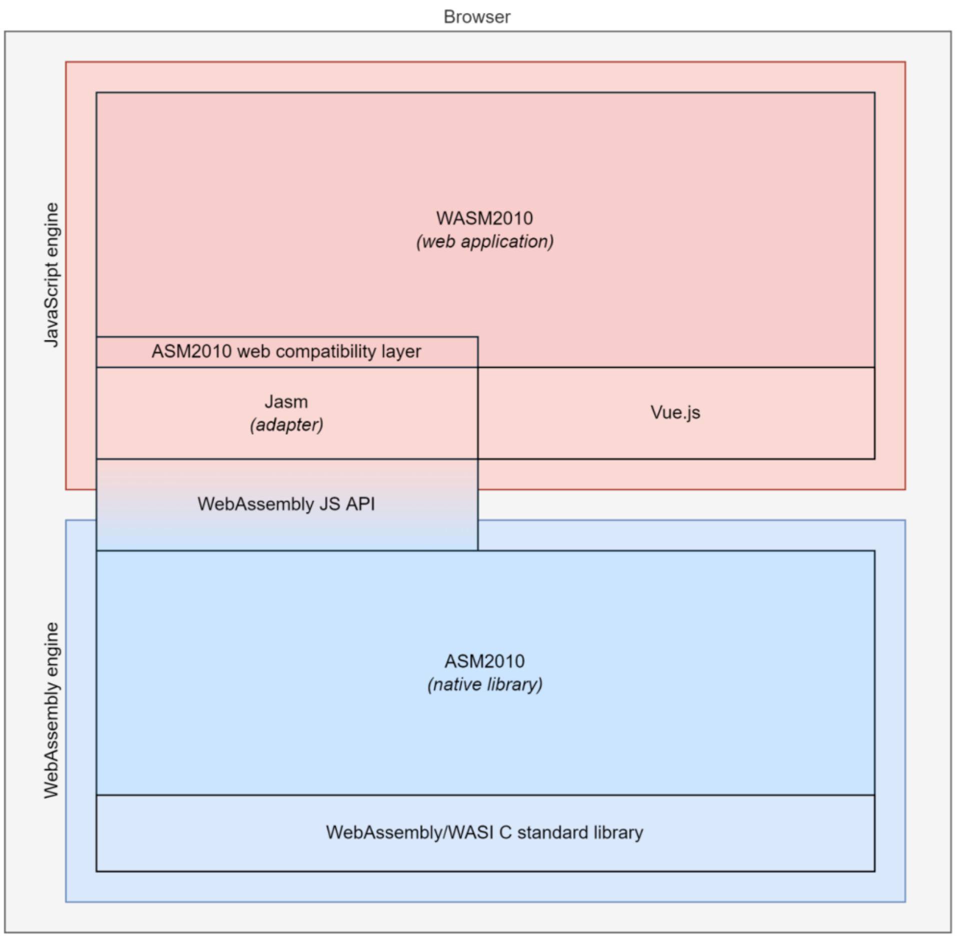


Figure 5-1. System architecture overview

As observed in Figure 5-1, the system's environment is a web browser, representing a modern version of any standard major browser, such as Mozilla Firefox, Google Chrome, Opera or Microsoft Edge, which provides support for the ECMAScript 5 specification [10] and the WebAssembly 1.0 specification [11]. Any version of these browsers released since late 2017 fulfils these requirements. It is important to note that, as the system does not perform any network request to a back-end application, it can be converted into a progressive web application that can be downloaded and run without any Internet connection.

Once inside the browser, the system is split into two parts, each running into a different engine: the JavaScript engine and the WebAssembly engine. Note that some browsers design their engines to run JavaScript and WebAssembly simultaneously to optimize performance —as in the case of Mozilla's SpiderMonkey [12] and Google's V8 [13]—. However, both environments will always be sandboxed and can be considered separated.

Two of the three developed components —the web application, WASM2010, and the adapter, Jasm— run entirely inside the JavaScript engine, while the native library, ASM2010, runs entirely in the WebAssembly engine. The only communication bridge between both engines is the WebAssembly JS API, which allows the JavaScript engine to command and manage the WebAssembly engine.

As stated in the requirements, WASM2010 and ASM2010 can not talk to each other directly. However, they have to use Jasm as a middle layer, allowing WASM2010 to decouple from almost all the logic related to the JavaScript—WebAssembly communication. Nevertheless, one middle step between WASM2010 and Jasm, the ASM2010 web compatibility layer, implements the necessary logic for WASM2010 to interface with ASM2010, easily ignoring every step in between.

This layered architecture, together with the effort put into separating components' concerns, allows every part of the system to be conveniently reused in other software projects. It also eases debugging the system, as errors can be isolated in a single layer, helping to find and fix them quickly.

In the following sections, we will closely examine each component, their internal architecture and runtime dependencies. The previous will be followed with a description of their communication paths and a comprehensive description of their application programming interface (API), completing the picture and clearly understanding how the system works hand in hand. Again, we will follow a bottom-up approach.

5.2 ASM2010: the native library

ASM2010 is the name given to the native C library developed in the project's scope. This library provides all the necessary functionality and business logic related to CS2010.

5.2.1 Architecture and implementation

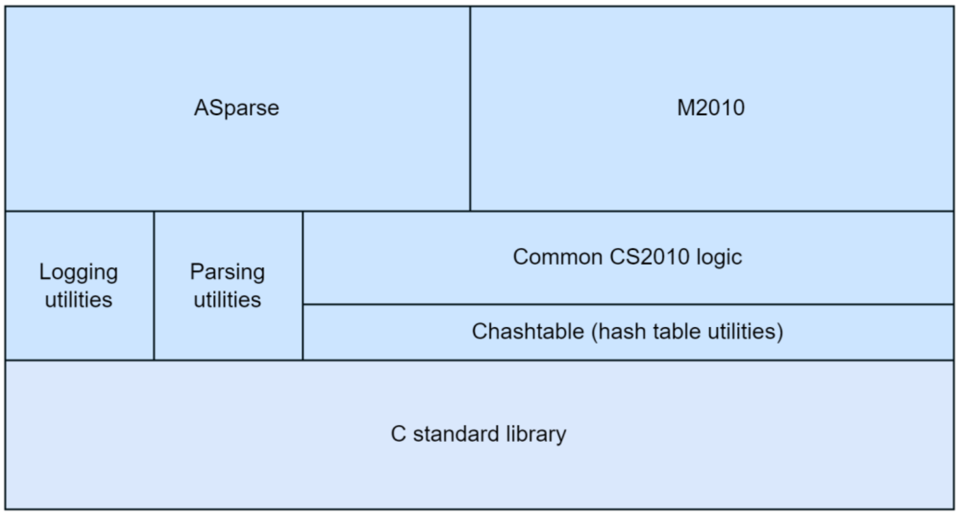


Figure 5-2. ASM2010 architecture

The entire library is built upon the C standard library and has no other external dependency, allowing the library to target virtually any platform with a compliant C implementation. Moreover, ASM2010 only uses a minimal set of the functionality provided by the C library, such as typical type definitions, memory, string and formatting utilities. Avoiding some features, such as I/O or file handling, allows it to be easily ported to a web browser, where these operations have to be implemented on the JavaScript side via WASI system calls [14].

```
const imports = {
    wasi_snapshot_preview1: {
        fd_close: () => {
            /* Stub method */
        },
        fd_seek: () => {
            /* Stub method */
        },
        fd_write: () => {
            /* Stub method */
        },
    },
};
```

Code 5-1. ASM2010's required WASI system calls

Building ASM2010 against the standard WASI C library allows us to see that only three system calls appear throughout the binary file: *fd_close*, *fd_seek* and *fd_write*. All these calls are related to file-handling features; however, they are never called from the actual implementation. Since these calls appeared somewhere in the C standard library, replacing them with stub functions is safe, as shown in Code 5-1.

On top of the C library, we find some utilities that support the main functionality. It is essential to mention Chashtable, a statically linked library that provides dynamic hash tables based on linked lists. This data structure is fundamental for parsing assembly source code, as it is needed for searching CS2010 operation codes, amongst other tasks. While it has been developed in the project's scope, it remains generic enough to be used as a library for other C projects. This way, the source code and the documentation can be found on its official home page [15].

```c
/** @file hash_table.h */

#ifndef HASH_TABLE_H
#define HASH_TABLE_H

#include <stddef.h>

#define CHT_MIN_LOAD_FACTOR 0.15
#define CHT_MAX_LOAD_FACTOR 0.75

/** @brief Struct representing the hash table */
struct hash_table {
  /** @brief Size index of the array of pointers to entries
*/
  size_t size_index;
  /** @brief Number of entries in the hash table */
  size_t entry_count;
  /** @brief Array of pointers to entries */
  struct hash_table_entry **entries;
};
typedef struct hash_table hash_table;
```

Code 5-2. Overview of the hash table interface

On the other hand, parsing and logging utilities are considered parts of ASM2010 exclusively, even when they could also be used in other software due to their genericity and ease of use, as can be seen in Code 5-3. Note that these utilities are used exclusively by the ASM2010's assembly parsing features and are not required to emulate the CS2010. A small extract of some utility functions is shown in Code 5-3.

```
/**
 * @brief Checks whether two unsigned will wrap when added
 * @param a First addend
 * @param b Second addend
 * @return true if addition wraps, false otherwise
 */
bool check_unsigned_add_wrap(size_t a, size_t b);
```

Code 5-3. Overview of ASM2010's utilities

Eventually, we reach the top layer of ASM2010. ASM2010's main functionality is provided by this layer, formed by two extensive modules built on top of some CS2010 common logic. ASM2010 is named after these blocks: Asparse (stylized as ASparse), which is the module in charge of assembling CS2010 source code, and M2010, the module accountable for performing emulations.

If we take a deeper look into both modules, we identify some shared files, such as those that describe the internal CS2010's structure: memories, registers and assembly language. Asparse and M2010 modules are implemented using this shared logic, avoiding code duplication. However, only M2010 implements the logic associated with the CS2010 microcode, as shown in Figure 5-3.

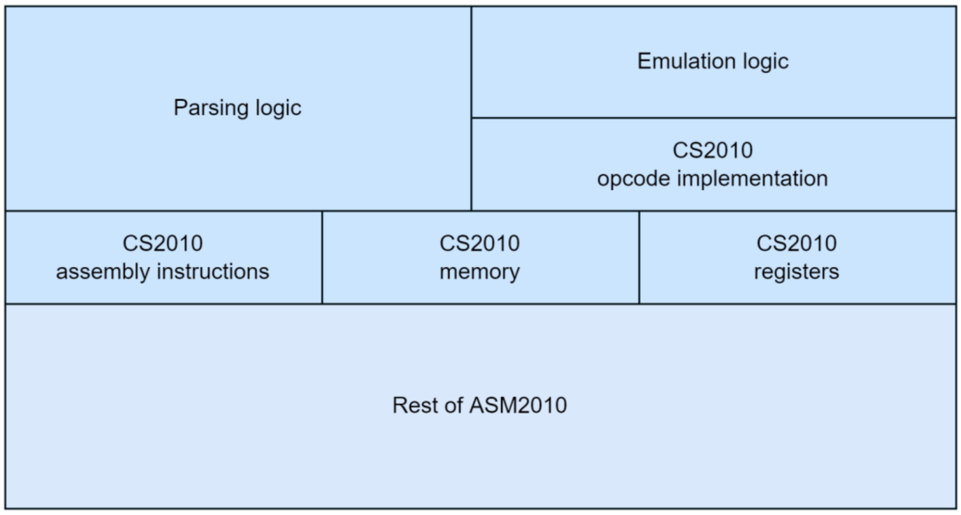
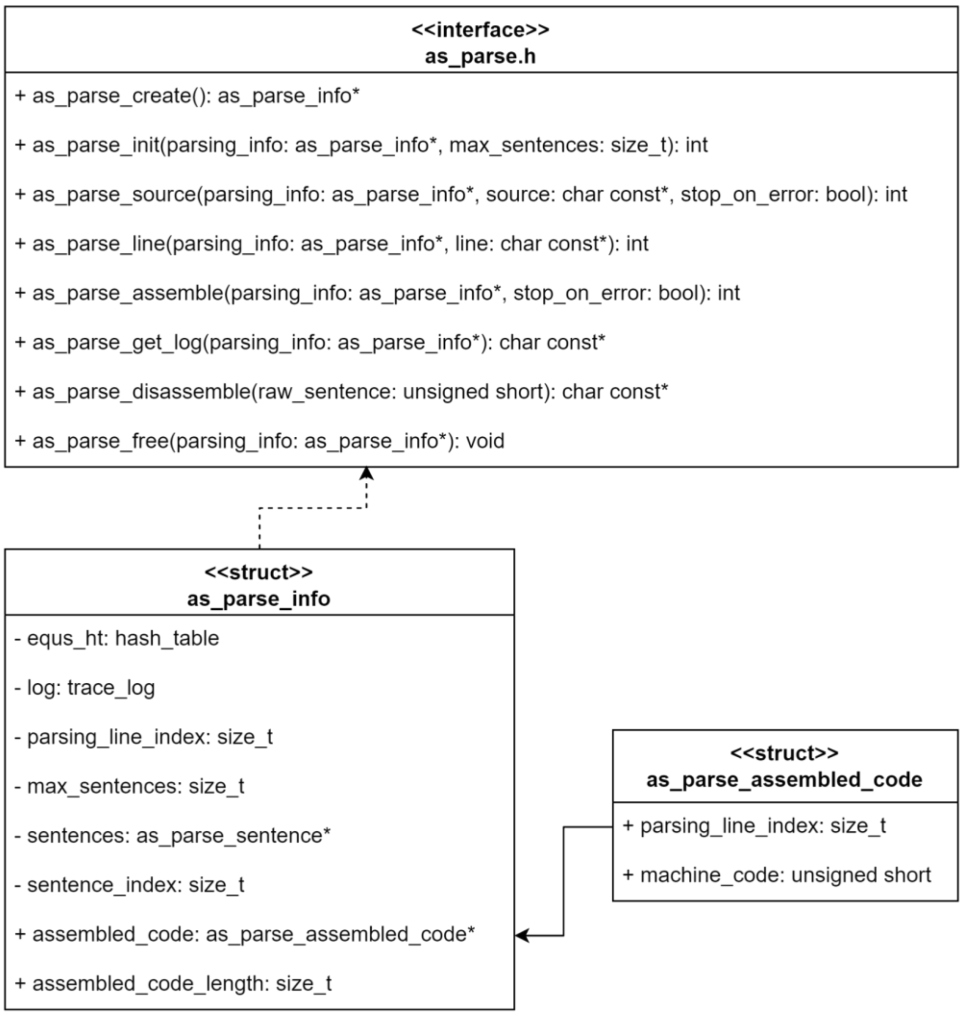


Figure 5-3. Asparse and M2010 internal architecture

5.2.2 Application programming interface

ASM2010 provides its API through Asparse and M2010, whose headers and source can be included and statically linked in any C/C++ program.



<<interface>>
as_parse.h
+ as_parse_create(): as_parse_info*
+ as_parse_init(parsing_info: as_parse_info*, max_sentences: size_t): int
+ as_parse_source(parsing_info: as_parse_info*, source: char const*, stop_on_error: bool): int
+ as_parse_line(parsing_info: as_parse_info*, line: char const*): int
+ as_parse_assemble(parsing_info: as_parse_info*, stop_on_error: bool): int
+ as_parse_get_log(parsing_info: as_parse_info*): char const*
+ as_parse_disassemble(raw_sentence: unsigned short): char const*
+ as_parse_free(parsing_info: as_parse_info*): void
<<struct>>
as_parse_info
- equs_ht: hash_table
- log: trace_log
- parsing_line_index: size_t
- max_sentences: size_t
- sentences: as_parse_sentence*
- sentence_index: size_t
+ assembled_code: as_parse_assembled_code*
+ assembled_code_length: size_t
<<struct>>
as_parse_assembled_code
+ parsing_line_index: size_t
+ machine_code: unsigned short

Figure 5-4. ASM2010's Asparse API definition

As observed in Figure 5-4, Asparse's API is straightforward. The assembly process starts with creating a new assembly unit (*as_parse_info* structure). This structure contains all the necessary information to track the ongoing assembly and the output machine code after the process is finished. It also contains a log filled with all the information produced during the assembly, which can be read using the *as_parse_get_log* function.

When the assembly is over, the generated machine code will be saved as an array of *as_parse_assembled_code* in the *assembled_code* member, while *assembled_code_length* allows knowing the array bounds. Each entry contains a single CS2010 machine code operation (*machine_code*), together with the index of the source assembly line that generated the associated machine code (*parsing_line_index*).

The disassembly process does not require a parsing unit. It can be called at any time by passing a single machine code as an unsigned short integer, where it is expected to fit according to the C99 standard type specification, which reserves at least 16 bits for a short value.

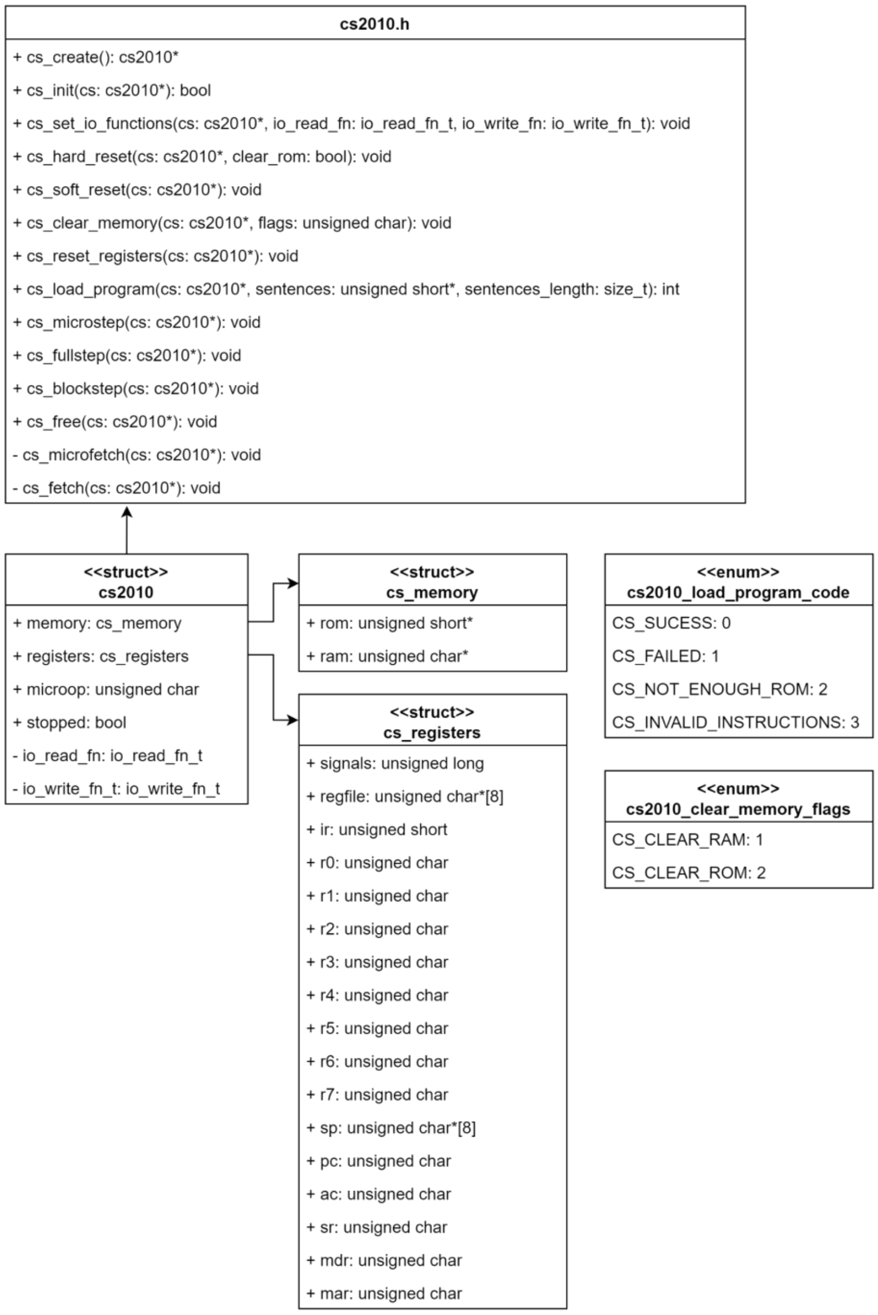

cs2010.h
+ cs_create(): cs2010*
+ cs_init(cs: cs2010*): bool
+ cs_set_io_functions(cs: cs2010*, io_read_fn: io_read_fn_t, io_write_fn: io_write_fn_t): void
+ cs_hard_reset(cs: cs2010*, clear_rom: bool): void
+ cs_soft_reset(cs: cs2010*): void
+ cs_clear_memory(cs: cs2010*, flags: unsigned char): void
+ cs_reset_registers(cs: cs2010*): void
+ cs_load_program(cs: cs2010*, sentences: unsigned short*, sentences_length: size_t): int
+ cs_microstep(cs: cs2010*): void
+ cs_fullstep(cs: cs2010*): void
+ cs_blockstep(cs: cs2010*): void
+ cs_free(cs: cs2010*): void
- cs_microfetch(cs: cs2010*): void
- cs_fetch(cs: cs2010*): void
<<struct>>
cs2010
+ memory: cs_memory
+ registers: cs_registers
+ microop: unsigned char
+ stopped: bool
- io_read_fn: io_read_fn_t
- io_write_fn_t: io_write_fn_t
<<struct>>
cs_memory
+ rom: unsigned short*
+ ram: unsigned char*
<<struct>>
cs_registers
+ signals: unsigned long
+ regfile: unsigned char*[8]
+ ir: unsigned short
+ r0: unsigned char
+ r1: unsigned char
+ r2: unsigned char
+ r3: unsigned char
+ r4: unsigned char
+ r5: unsigned char
+ r6: unsigned char
+ r7: unsigned char
+ sp: unsigned char*[8]
+ pc: unsigned char
+ ac: unsigned char
+ sr: unsigned char
+ mdr: unsigned char
+ mar: unsigned char
<<enum>>
cs2010_load_program_code
CS_SUCESS: 0
CS_FAILED: 1
CS_NOT_ENOUGH_ROM: 2
CS_INVALID_INSTRUCTIONS: 3
<<enum>>
cs2010_clear_memory_flags
CS_CLEAR_RAM: 1
CS_CLEAR_ROM: 2

Figure 5-5. ASM2010's M2010 API definition

Note how M2010's API follows a similar schema. First, a new emulation instance must be created and initiated, virtually creating a new CS2010 computer. The machine code can be loaded into the ROM using the *cs_load_program* function, which not only writes the memory but also hard resets the computer and checks the machine code's correctness before loading it. This function returns several status codes, allowing the programmer to know the state of the loading process.

The programmer can call any function after initiating a given emulation instance, thus accessing the full capabilities of M2010. The current status of the CS2010 computer can be directly read from the structure at any point, as it will always be consistent.

In order to allow I/O mapping, emulation instances invoke two internal handlers when a read-write operation takes place. Their result is critical to know whether a given address is mapped to an external I/O device and, in this case, to carry out the requested operation. The signatures for these handlers are defined as follows:

```
size_t (*io_read_fn_t)(size_t address);
bool (*io_write_fn_t)(size_t address, unsigned char
content);
```

Figure 5-6. M2010 I/O handling types

Where *io_read_fn_t* represents the signature of a read I/O handler, and *io_write_fn_t* is the signature of a write I/O handler.

The read I/O handler will be called whenever a read operation takes place, with the requested memory address passed as the first argument in the call. The handler must respond by returning a fixed constant (*CS_IO_READ_NOT_CONTROLLED*) in case the address is not mapped to any external device, which redirects the request to the main memory instead, or with the value that the external device mapped at that address returns as a result of the read operation. Technically, returning any value that exceeds the upper limit of an immediate value (this is, $2^8 - 1$), such as -1, will be considered a negative answer from the read handler, but using a fixed constant is encouraged as it clarifies intentions and avoids the magic number problem [16].

The write I/O handler follows a similar logic. Whenever a write operation occurs, it will be called and passed the address at which the operation takes place as the first argument and the immediate value to be written as the second argument. The handler must then return a boolean value indicating whether external devices handle this request (returns constant *CS_IO_WRITE_CONTROLLED*) or the main memory should handle it (returns constant *CS_IO_WRITE_NOT_CONTROLLED*), which are in turn truthy and falsy values, respectively.

We can depict both processes using communication diagrams with a synchronous flow. The reading process is analyzed in the diagram shown in Figure 5-7.

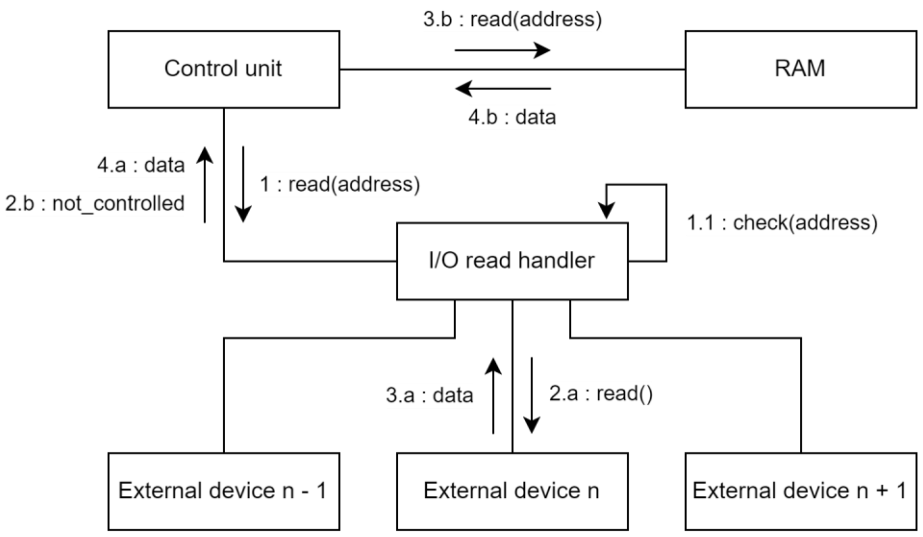


Figure 5-7. Memory read process

Similarly, if we depict the writing process, we get the diagram shown in Figure 5-8. Note how writing presents slightly lower latency than reading.

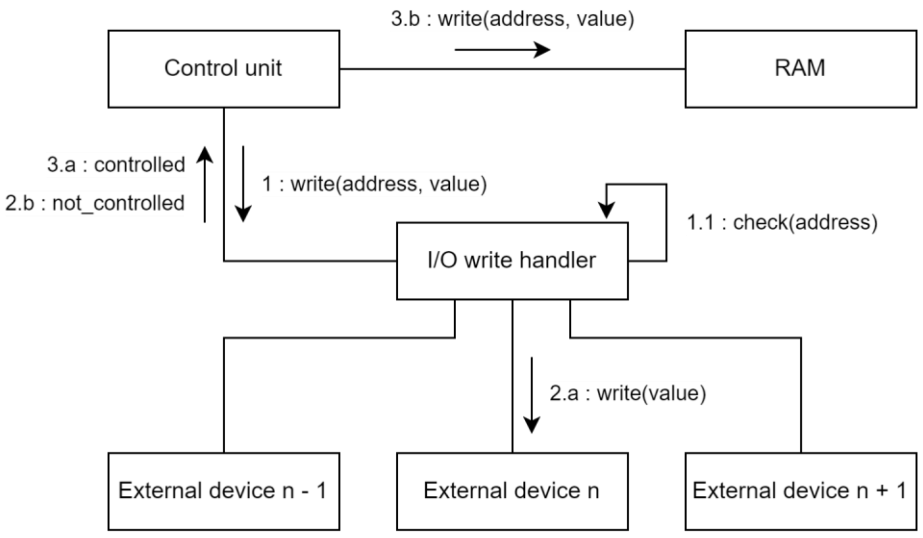


Figure 5-8. Memory write process

In practice, how these handlers decide which value should return is up to the programmer. This fact allows for modifying the behaviour of devices and even the device mapping itself at runtime. However, it is essential to remark that these handlers operate in a synchronous thread so that any delay on them will hang the entire execution as a consequence. Any significant computation must be performed in a different thread to avoid performance overhead.

5.3 Jasm: the adapter

Jasm is a TypeScript library developed in the project's scope. This library provides functionality that facilitates the interoperation between a JavaScript module and a C program, easing access to the C memory model according to the WebAssembly C ABI.

5.3.1 Architecture and implementation

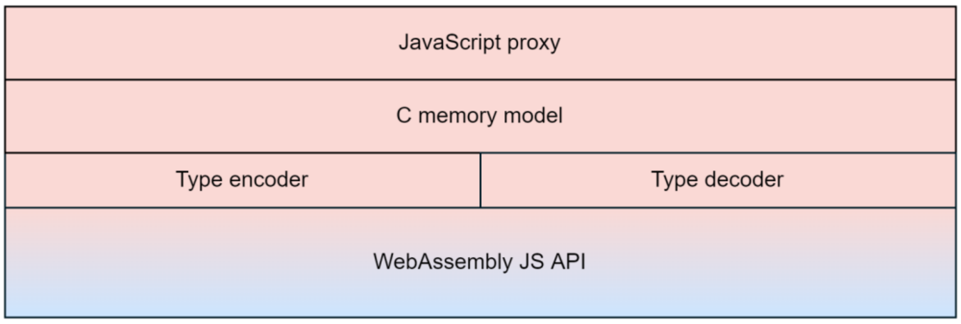


Figure 5-9. Jasm architecture

Jasm is built on top of the WebAssembly JS API and follows a layered architecture where each layer gets closer to WebAssembly as we step down.

At the bottom layer, type encoders and decoders have their place. These components help solve issues that arise when WebAssembly memory is accessed directly, including endianness and buffer detaching problems. Most implementations of the WebAssembly JS API expose a raw buffer to access the memory. However, this buffer detaches from the memory when the memory grows, either because JavaScript requests it or because a WebAssembly grow instruction is executed. As long as it is challenging to solve this issue, encoders and decoders provide safe access anytime.

On top of this layer, we find an implementation of the C memory model described by the WebAssembly C ABI. This layer contains all the information about how C types are represented inside the WebAssembly memory and their matching JavaScript types. It also carries out complex computations to determine how C aggregates (structures and unions) are internally arranged in memory, thus relieving the user from manually calculating alignments and paddings.

The top layer exposes the functionality accessible from JavaScript. The programmer can create JavaScript objects and bind them to a single instance of a data structure in the WebAssembly memory, thus giving access to the memory and allowing it to perform read-write operations transparently from JavaScript.

5.3.2 Application programming interface

Jasm provides a TypeScript API over a single UMD JavaScript module, which means it can be used in any environment that runs JavaScript, even in projects that do not use bundlers.

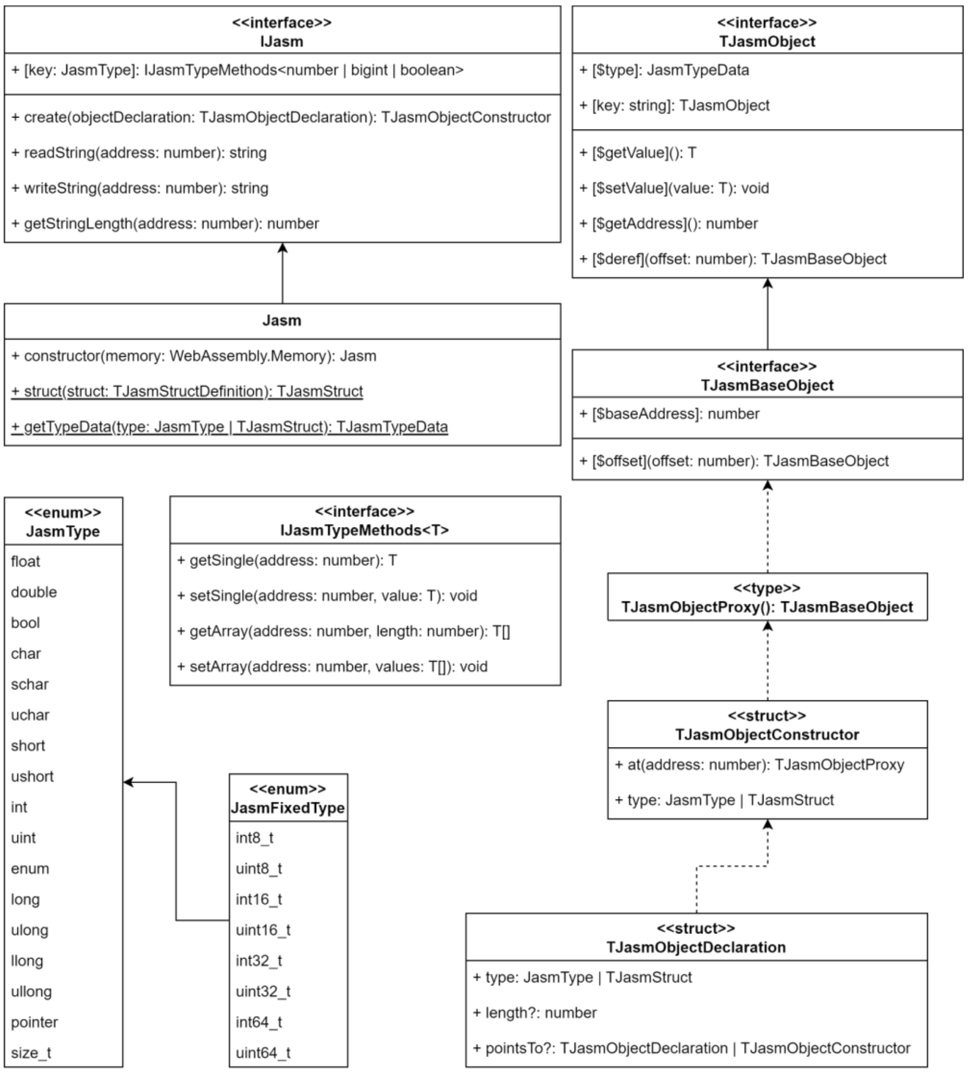

<<interface>>
IJasm
+ [key: JasmType]: IJasmTypeMethods<number | bigint | boolean>
+ create(objectDeclaration: TJasmObjectDeclaration): TJasmObjectConstructor
+ readString(address: number): string
+ writeString(address: number): string
+ getStringLength(address: number): number
<<interface>>
TJasmObject
+ [$type]: JasmTypeData
+ [key: string]: TJasmObject
+ [$getValue](): T
+ [$setValue](value: T): void
+ [$getAddress](): number
+ [$deref](offset: number): TJasmBaseObject
Jasm
+ constructor(memory: WebAssembly.Memory): Jasm
+ struct(struct: TJasmStructDefinition): TJasmStruct
+ getTypeData(type: JasmType | TJasmStruct): TJasmTypeData
<<interface>>
TJasmBaseObject
+ [$baseAddress]: number
+ [$offset](offset: number): TJasmBaseObject
<<enum>>
JasmType
float
double
bool
char
schar
uchar
short
ushort
int
uint
enum
long
ulong
llong
ullong
pointer
size_t
<<interface>>
IJasmTypeMethods<T>
+ getSingle(address: number): T
+ setSingle(address: number, value: T): void
+ getArray(address: number, length: number): T[]
+ setArray(address: number, values: T[]): void
<<type>>
TJasmObjectProxy(): TJasmBaseObject
<<struct>>
TJasmObjectConstructor
+ at(address: number): TJasmObjectProxy
+ type: JasmType | TJasmStruct
<<enum>>
JasmFixedType
int8_t
uint8_t
int16_t
uint16_t
int32_t
uint32_t
int64_t
uint64_t
<<struct>>
TJasmObjectDeclaration
+ type: JasmType | TJasmStruct
+ length?: number
+ pointsTo?: TJasmObjectDeclaration | TJasmObjectConstructor

Figure 5-10. Jasm API definition

The Jasm API starts in the homonym class *Jasm*, which provides the entry point for every feature. The class itself contains two static methods, *struct* and *getTypeData*. The first allows the programmer to define an aggregate (struct or union) so Jasm can compute its alignment and padding. The second allows getting the type information, such as type size or alignment, from a given native type (defined in JasmType) or from an aggregate already parsed by the *struct* method. These are the only operations performed without attaching Jasm to a WebAssembly memory.

```
const cs2010 = Jasm.struct({
  members: [
    { name: 'memory', type: cs_memory },
    { name: 'registers', type: cs_registers },
    { name: 'microop', type: JasmType.uchar },
    { name: 'is_stopped', type: JasmType.bool },
  ],
});
```

Code 5-4. Computing a structure with Jasm

Jasm needs to be instantiated using its constructor and passing a WebAssembly memory instance to start diving into the rest of the functionality. During the initialization, the memory will be bound to the Jasm instance, allowing it to keep track of the memory's buffer. This binding has no side effects on the memory's content, as Jasm does not store any information. Instead, it relieves the user from passing the memory as an argument every time a method is called.

Once Jasm has been instantiated, the programmer can read and write the memory using native types and strings. The support for native types is given through the *IJasmTypeMethods* provided for every native type, while strings can be manipulated using the methods *readString* and *copyString*, respectively.

```
  const finalSourceAssembly = `${sourceAssembly}\n`;
  const src_code_ptr = malloc(finalSourceAssembly.length);
  jasm.copyString(src_code_ptr, finalSourceAssembly);
```

Code 5-5. Allocating and copying a string using Jasm

However, all these methods require passing an address every time they are called, and they do not support more complex types, such as aggregates. The *create* method, available on the Jasm instance, allows the creation of an arbitrarily complex type. The *TJasmObjectDeclaration* interface allows any native type or aggregate to be used as a custom type. This declaration will be transformed into a *TJasmObjectConstructor*, which can be seen as a type definition in the C programming language. It provides the ability to declare instances of a given type and bind them to any address, avoiding recomputing the type metadata every time.

Rather than creating a *TJasmObject* instance directly, *TJasmObjectConstructor* returns a proxy function that, in turn, returns the actual object instance. While this might seem confusing at first glance, the purpose of this proxy is to avoid recreating a new TJasmObject every time a new object instance is created, as this is an expensive task in terms of performance. Instead, an internal object instance is reused every time the proxy is called, thus adding almost zero overhead to functions commonly invoked at run-time.

When a given TJasmObject represents an aggregate type, its elements can be accessed directly as JavaScript properties inside it. Whenever the programmer wants to access the value of an object, its address or any other method described in the TJasmObject definition, it is necessary to use symbols instead. Using JavaScript symbols avoids naming conflicts between the members of an aggregate and the actual names of the methods of an object, such as *$setValue* or *$getValue*.

```
export function csGetStatus(): TCsStatus {
  const registers = cs().registers[$getValue]();
  return {
    ram: jasm.uchar.getArray(cs().memory.ram[$getValue](),
CS_RAM_SIZE),
    registers,
    signals: Object.entries(CsSignalOffset).reduce(
      (acc, [signalName, signalOffset]) => ({
        ...acc,
        [signalName]: registers.signals & signalOffset,
      }),
      {}
    ) as TCsSignals,
    microop: cs().microop[$getValue](),
    isStopped: cs().is_stopped[$getValue](),
  };
}
```

Code 5-6. Typical use of JasmObjects and their methods

It is important to remark that the dereference operator, namely *$deref* —note it represents a JavaScript symbol and not a literal property— only exists in those objects whose type is a pointer. Trying to invoke otherwise will throw an undefined exception.

5.4 WASM2010: the web application

WASM2010 is a web application that provides access to ASM2010's features, such as parsing and emulation. Note that the project is named after this component, too, as it is the culmination of many software layers.

5.4.1 Architecture and implementation

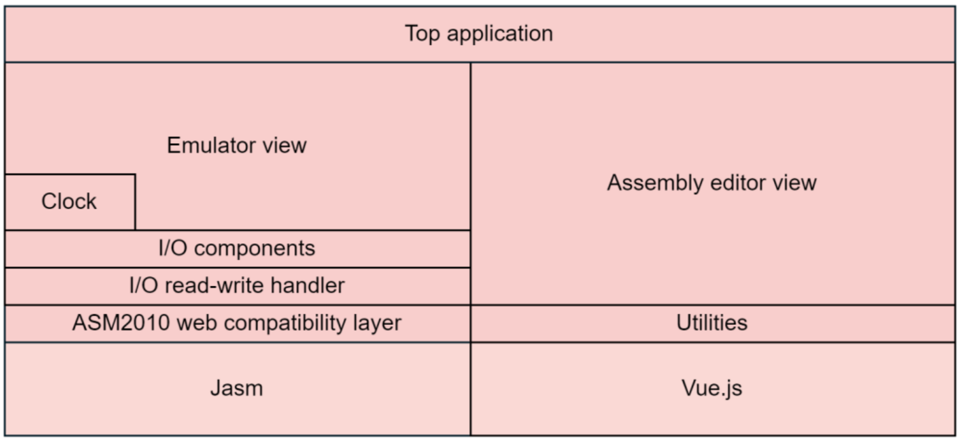


Figure 5-11. WASM2010 architecture

WASM2010 is built on top of Vue.js, an open-source framework created by Evan You for building web user interfaces approaching the reactive paradigm [17]. The entry point consists of a Vue application with two different views. The first one is built around the assembly capabilities of ASM2010, presenting a source assembly editor where the user can create his own CS2010 programs. On the other hand, the emulator view introduces a full-featured emulation sandbox where the user can run and debug CS2010 assembly code.

The ASM2010 web compatibility layer and some general-purpose utilities support both views. This layer allows the views to interact with the ASM2010 native library seamlessly. It uses Jasm to implement essential data structures such as ASM2010's interfaces and types, providing an ASM2010-like interface that can be used directly from the web application, thus abstracting it from any lower-level detail.

Thanks to the I/O extension mechanism described in ASM2010 and the joint efforts of Jasm and the web compatibility layer, WASM2010 can implement its own fully compatible I/O devices (also called components) by providing custom I/O handlers from the JavaScript side. In addition, WASM2010 provides a clock on its side that can be used to run the emulation at a fixed frequency by sending periodical signals to ASM2010.

As this is the entry point of the user interaction, WASM2010 does not expose any API. However, it remains an interoperable piece of software that could be used as a front-end application for other emulators.

5.5 Testing

Due to the project's complexity and extension, manually performing horizontal end-to-end testing (also known as E2E) has been considered the best approach. This way, two central systems have been defined as testing targets: the assembly and emulator subsystems.

During the development, these tests have been applied with different data, including edge cases, to verify the system's acceptance according to the requirements. These manual E2E tests have also been applied to every developed component to guarantee they work as expected in every development phase. As describing these tests formally is challenging, we will give some examples in the following list.

Code	ETE-001
Title	Users can assemble a valid CS2010 program
Target	Assembly subsystem
Pre-condition	The page is loaded successfully
Path	1. The user writes a CS2010 program without errors 2. The user changes the output radix to binary 3. The user hides the output radix 4. The user saves the source as a disk file 5. The user creates a new source file 6. The user writes a CS2010 program with errors 7. The user loads the previously saved disk file 8. The user changes the output radix to hexadecimal
Post-condition	The user must see the output hexadecimal machine code matching the first assembly program

Table 5-1. E2E test 1

Code	ETE-002
Title	Users can run a CS2010 program manually
Target	Emulator subsystem
Pre-condition	ETE-001 has been completed successfully
Path	1. The user clicks on Start emulation button to switch to the emulator view 2. The user changes the RAM radix to binary 3. The user changes the ROM radix to decimal 4. The user changes the registers radix to hexadecimal 5. The user unmaps every I/O device 6. The user clicks on Step microop button 7. The user clicks on Step op button 8. The user clicks on Step block button until the program is finished
Post-condition	The user is notified that the execution is finished The user can see the result of the execution successfully

Table 5-2. E2E test 2

Code	ETE-003
Title	Users can run a CS2010 program automatically
Target	Emulator subsystem
Pre-condition	ETE-001 has been completed successfully
Path	1. The user clicks on Start emulation button to switch to the emulator view 2. The user changes the clock frequency to 60Hz 3. The user clicks on Start clock button 4. The user waits until the execution is finished
Post-condition	The user is notified that the execution is finished The clock is stopped The user can see the result of the execution successfully

Table 5-3. E2E test 3

6 Building and deploying

In this chapter, we will discuss how to build and deploy the software following two different approaches. In the first one, we describe the entire manual building process to understand how the final bundle is generated. The second uses Docker to automatize this process for the user.

6.1 Manually building and deploying the project

The guide will cover the steps to follow to build the entire toolkit. While it is a complex software project, the build process has been simplified thanks to external libraries, such as Jasm, which is already built and uploaded to the NPM global registry [18], where the toolkit will fetch it from as specified in the package dependencies. For simplicity, the core C library, ASM2010, has also been prebuilt for the wasm32 target, meaning that the build process can be entirely skipped, no matter the user's host operating system.

6.1.1 Prerequisites for building ASM2010

A WebAssembly port of ASM2010 is provided in the *src/asm2010* directory of the WASM2010 source code. The requirements to build the project are listed below:

- An LLVM installation (12.0 or later) with proper support for the *wasm32-unknown* target. It is required to install the LLVM builtins library for wasm32, which can be compiled from the source or obtained from the WASI SDK.
- The WASI libc, which provides a standard C library on top of the WASI syscalls.
- A Make-like tool to build the library.

While obtaining a WASI-enabled LLVM installation is not difficult, using the WASI SDK can ease this process. The WASI SDK is a toolchain comprising a compliant Clang compiler with out-of-the-box support for building WebAssembly/WASI binaries. Due to its simplicity, we will be using this SDK from now on. It can be downloaded free of charge from its home source page [19]. It is available for any modern operating system today, including Microsoft Windows, GNU/Linux and macOS, and it can be assumed that it is safe to use the latest version available.

The WASI SDK does not feature a Make-like tool. While building the project without this tool is still possible, using it will make the process easier. On GNU/Linux and macOS, Make comes together with the standard developer tools (the build-essentials package and the Apple Developer Tools, respectively). On Microsoft Windows, the best option is likely to use MinGW32-make, which is a valuable replacement.

6.1.2 Building ASM2010 for WebAssembly/WASI

After getting a copy of the WASI SDK and *make*, the process is straightforward and does not depend on the host operating system. We must take the following steps:

1. Clone the WASM2010 Git repository using the command prompt. Any of the methods shown in Code 6-1 can be used.

```
# Using HTTPS
git clone https://github.com/GuilleX7/WASM2010.git

# Using SSH
git clone git@github.com:GuilleX7/WASM2010.git
```

Code 6-1. Cloning the WASM2010 repository

2. Navigate to *src/asm2010*.
3. Open a command prompt in the *src/asm2010* directory, where the Makefile is found.
4. Build the binary using the commands in Code 6-2. Please note that we are passing an argument to Make (*WASI_SDK*) to tell it where is our WASI SDK located, as we have seen in step 3. The user must change this argument to point to the downloaded WASI SDK installation path. If the user does not pass any argument, Make will try to get it from the system's environment variables.

```
# Replace with your WASI SDK path
make WASI_SDK=<YOUR_WASI_SDK_PATH> all
make clean
```

Code 6-2. Building ASM2010 using the WASI SDK

An example of a proper replacement for the WASI_SDK argument would be the one shown in Code 6-3. Note how the path must end without any slash.

```
make WASI_SDK=/home/guillex7/wasi-sdk all
```

Code 6-3. Passing a proper WASI SDK path to Make

However, some people might prefer not to use the WASI SDK but a previous installation of LLVM that can build WebAssembly binaries. In that case, we will need to pass two different variables to the Makefile instead of *WASI_SDK*, as shown in Code 6-4.

```
make CC=<CLANG_PATH> WASI_SYSROOT=<WASI_SYSROOT_PATH> all
```

Code 6-4. Building ASM2010 without the WASI SDK

Make sure that the *make* executable is in the *PATH* environment variable so it can be used directly from the terminal. Executing any of Code 6-3 or Code 6-4 will build the entire project, which in turn consists of a binary target, *build/asm2010.wasm*.

6.1.3 Explaining the ASM2010 building process

We will explain the building process under the hood to understand better how the final library is generated. The main logic of the Makefile is shown in Code 6-5, placed after some initial configuration. These initial declarations allow the user to pass the WASI SDK path as an argument or retrieve it from the environment variables. While this step could have been further automated using advanced tools such as CMake, autotools or similar toolchains, it does not seem to be a worthy investment of time, as it still requires little input from the user.

Once inside the Makefile, we can see some targets by the end of the file. Note that the target named *all* is a synonym for *OUTPUT_LIB*, which generates the output library. The process resembles a regular C program compilation, where each compilation unit is translated to an object file, and all object files are linked into the final executable. In the compilation stage, some flags are passed to the compiler to enable aggressive speed optimizations (-*O3*), turn on all warnings during the compilation (-*Wall*) and choose the target platform of the object files (--*target*).

However, we will look closer at the flags used during the linking stage that appears in Code 6-5, declared altogether in the variable *LDFLAGS*.

```
CFLAGS=-O3 -Wall --target=wasm32-wasi
LDFLAGS=--sysroot=$(WASI_SYSROOT) -nostartfiles
-fvisibility=hidden -Wl,--no-entry -Wl,--export-dynamic
-Wl,--export=malloc -Wl,--export=free -Wl,--import-undefined

DEPS=src/about.h src/as_parse.h src/cs_instructions.h
src/cs_memory.h src/cs_op.h src/cs_registers.h src/cs2010.h
src/hash_table.h src/parse.h src/trace_log.h src/utils.h
src/wasm.h

OBJ=build/as_parse.o build/cs_instructions.o build/cs_op.o
build/cs2010.o build/hash_table.o build/parse.o
build/trace_log.o build/utils.o build/wasm.o

OUTPUT_LIB=build/asm2010.wasm

build/%.o: src/%.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

$(OUTPUT_LIB): $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

all: $(OUTPUT_LIB)

clean:
    rm build/*.o

.PHONY: clean
```

Code 6-5. Main logic of the ASM2010 Makefile

- *--sysroot*. Chooses the *sysroot* path, i.e., the root directory where the headers and libraries reside. This argument points to the WASI SDK directory containing the C standard library.
- *-nostartfiles*. It tells the linker not to put any code in our binary that prepares the environment before *main()* is called, as the code does not contain any *main()* function.
- *-Wl,--no-entry*. We need to tell the linker not to look for any start point in the code, like *main()* or *_start()*. As ASM2010 is built and used as a library, exported functions will be called directly from JavaScript when needed.
- *-fvisibility*. This option marks every symbol as an internal symbol by default. Only those marked as visible will be exported to the

symbol table in the linking stage, reducing the binary size considerably, as the linker can decide not to put unused symbols into the output library. It also allows the library to hide unintended internal functions for external use and provide a public API free of implementation details.

- *-Wl,--export-dynamic.* Together with the previous flag, it allows the linker to export visible symbols.
- *-Wl,--export.* Apart from the visible symbols, two functions will be handy on the JavaScript side: malloc and free. They allow managing the heap of the WebAssembly virtual machine directly from JavaScript, which reduces the amount of boilerplate code needed to create and remove objects in the memory. As these symbols are declared in a system header, and we do not have access to them, we need to tell the linker to change their visibility using this flag.
- *-Wl,--import-undefined.* While the original ASM2010 library is self-contained and does not need to import anything from the environment, our current implementation needs two external symbols, *wasm_custom_io_read* and *wasm_custom_io_write*. JavaScript will provide these symbols as native JavaScript functions that will interoperate with ASM2010. Their purpose is to manage everything related to I/O, as they are called by ASM2010 when an I/O call is performed. As these functions are declared and used in the ASM2010 source code but not defined, the linker will put these undefined symbols in the import table. It is required for the host environment to provide these symbols when the WebAssembly module is loaded, which in this case, turns out to be the JavaScript runtime.

Eventually, the compiler will generate object files and the final library in the build directory. After the compilation, object files might be helpful to avoid compiling every part of the source code again. However, if this is not our case, we can remove all object files to clean up, for which we will use the *clean* phony target.

6.1.4 Prerequisites for building Jasm/WASM2010

Jasm and WASM2010 need a valid Node.js [20] installation, including NPM, the Node.js package manager. At least the NPM binary should be in the PATH environment variable. This environment allows conveniently installing JavaScript dependencies, relieving the user of the need to download and manage them separately.

While any modern version of Node (version 12 and later) should be capable of building the following projects, we recommend using Node 16 or later.

6.1.5 Building Jasm

Jasm is a TypeScript library that will be used by the front-end application to interact with ASM2010. Since the browser cannot execute TypeScript code directly, the library must be transcompiled into plain JavaScript. We will use Babel [21], a JavaScript transcompiler (source-to-source compiler), for this purpose. Babel takes modern JavaScript code and produces code compliant with older ECMAScript standards, allowing it to run in old browsers, and increasing compatibility. While Babel's primary focus is JavaScript, it can be extended to take TypeScript code as input. This way, Jasm benefits from the source-to-source conversion and can take advantage of these ECMAScript transformations.

While the code generated by Babel can be directly run in a browser, it is not size-optimized. For this reason, we will use Webpack [22], which is a module bundler, i.e., it takes modules of several types as input, including, but not limited to, JavaScript, and produces a bundle as output, which is a reduced number of compressed files that behave exactly as the input. This compression is achieved by minification and stripping unnecessary data, among other source compression techniques.

This bundle also benefits from improving its compatibility across different JavaScript environments due to the Universal Module Definition (UMD) format. This module format gives a JavaScript module the ability to understand the environment in which it is being executed, allowing it to be helpful outside the browser. The final result is a minimized universal JavaScript library that can be used directly from any other JavaScript application.

The user must take the following steps to build Jasm:

1. Clone the Jasm Git repository using the command prompt. Any of the methods shown in Code 6-6 can be used.

```
# Using HTTPS
git clone https://github.com/GuilleX7/Jasm.git

# Using SSH
git clone git@github.com:GuilleX7/Jasm.git
```

Code 6-6. Cloning the Jasm repository

2. Navigate to the root directory.
3. Install the necessary dependencies as shown in Code 6-7.

```
npm install
```

Code 6-7. Installing Jasm dependencies

4. Build the bundle as shown in Code 6-8.

```
npm run build
```

Code 6-8. Building the Jasm bundle

After the build completes, Webpack will generate a folder named *lib* in the root directory and place the bundle inside it.

We will need to use this library as a dependency for WASM2010. However, as NPM downloads every dependency from the official NPM registry, we need to tell NPM to use this local folder as the source of a dependency, as shown in Code 6-9.

```
npm link
```

Code 6-9. Linking the local bundle as a dependency

6.1.6 Building WASM2010

We will follow a similar build procedure to the previously mentioned, using Babel and Webpack to produce a minified bundle. This bundle not only comprises JavaScript code but also generates an HTML file together with every other asset that is not code, including but not limited to CSS files or the ASM2010 WebAssembly binary module, which the browser needs to load at runtime.

The user must take the following steps to build WASM2010:

1. If the WASM2010 repository has not been cloned yet in section 6.1.2, clone it using a command prompt. Otherwise, jump to the next step.
2. Navigate to the root directory.
3. As foreseen in Code 6-9, regularly installing the necessary dependencies would result in NPM fetching the Jasm sources from the global NPM repository. However, this is undesirable, as we want to use the local bundle we prepared during the steps followed

in 6.1.5. Use the command in Code 6-10 to link with the local Jasm and install every other dependency from the NPM registry.

```
npm link @guillex7/jasm
```

Code 6-10. Linking and installing WASM2010 dependencies

4. Build the bundle as shown in Code 6-11.

```
npm run build
```

Code 6-11. Building the WASM2010 bundle

After the build completes, Webpack will generate a folder named *dist* in the root directory and place the bundle inside.

6.1.7 Deploying WASM2010

While the generated bundle is understandable by a standalone browser, it is impossible to open it from a browser without using a web server, as most browsers forbid loading local resources for security reasons. To quickly open a web server with the development version of WASM2010, one can use the command shown in Code 6-12.

```
npm run dev
```

Code 6-12. Opening a local web server

After a short time, a web server should be opened at port 8090, accessible via localhost. In case this port is already used by another application, it will try to use the next available port after it. However, it is essential to remark that this web server is primarily intended for development, thus presenting some features that could be undesirable under production environments. Moreover, it lacks many settings and security features. A standalone web server is likely the best option in a real environment, such as Nginx, Apache or Lighttpd.

6.2 Automatically building and deploying the project

While following the steps to build and deploy the project manually is recommended to learn how things work under the hood, this might not be desirable in cases where the main goal is to have WASM2010 up and running in the minimum time possible.

Docker is a set of tools that will allow us to run containers, which are isolated boxes where software can run independently from other containers while still sharing the resources of the same operating system. This approach allows better performance and space optimization than virtual machines [12]. In contrast, as containers are not virtualized, they must run in the same operative system they were built. However, this concern is not an issue for this project, as popular images already have multiplatform support. For this reason, we will explain how to use Docker to quickly build and deploy every part of the project at once with a few commands.

6.2.1 Building and deploying with Docker

To build and deploy the entire project using Docker, one can take the following steps:

1. If the WASM2010 repository has not been cloned yet in section 6.1.2, clone it using the command prompt. Otherwise, jump to the next step.
2. Navigate to the root directory.
3. Build the docker image using the command shown in Code 6-13.

```
docker build -t wasm2010 .
```

Code 6-13. Building the WASM2010 docker image

4. Run a Docker container with the generated image using the command shown in Code 6-14. Note that the user should change *HOST_PORT* to the port that will be used to access the local web server via localhost. Usually, this port is 80, which is the default web port for the HTTP protocol.

```
docker run -p HOST_PORT:80 -d wasm2010
```

Code 6-14. Running a WASM2010 Docker container

The build process replicates almost the same procedure shown in 6.1, except for building Jasm from scratch. Instead, the Docker image installs the Jasm package directly from the NPM registry, simplifying the process. The ASM2010 library and WASM2010 are built using the provided source code, and the output bundle is put inside an Nginx web server [23], which will serve the static files in the given *HOST_PORT* port, thus allowing the browser to load the resources safely so the application can run locally.

It might be surprising that the final Docker image has a reduced size, even though the necessary tools to build the entire project require much more space. The use of multi-stage buildings explains this phenomenon during image creation. Instead of installing all the requirements in the same system containing the web server, we split the process into two stages that run in separate systems. The first is provided with the build toolchain and generates the final bundle. Then, the bundle is copied into this second stage, which contains only the bare minimum to run a web server that serves the bundle. As a result, only the second stage remains in the Docker image, notably reducing the final image size.

7 User guide

WASM2010 features a simple and minimalist user interface. As stated in 1.3, one of the project's key aspects is providing the user with an easy-to-use toolkit. In this chapter, we will provide the user with detailed instructions on how to use the toolkit.

7.1 Using the assembly editor

The assembly editor view is displayed when the user enters WASM2010, as shown in Figure 7-1.

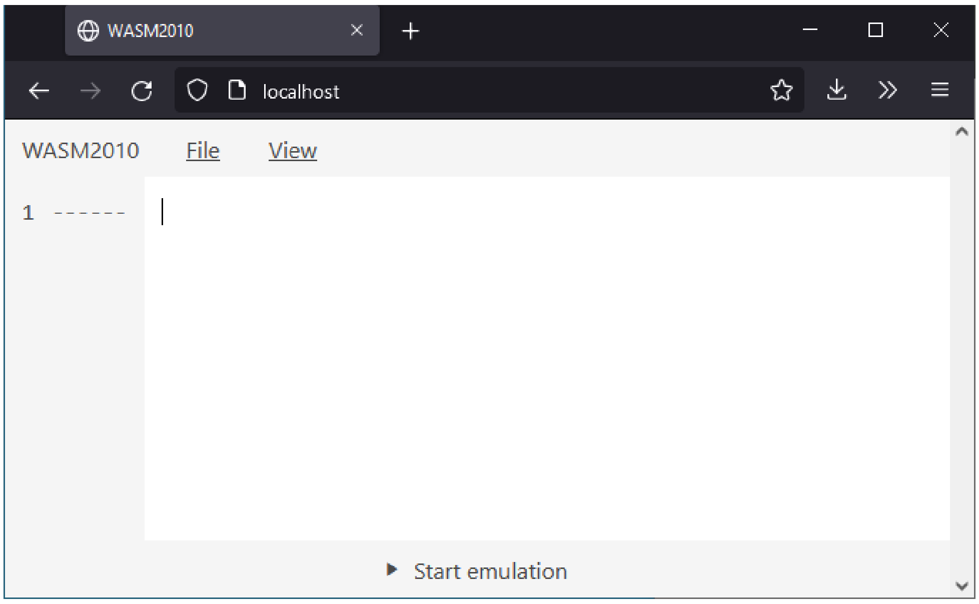
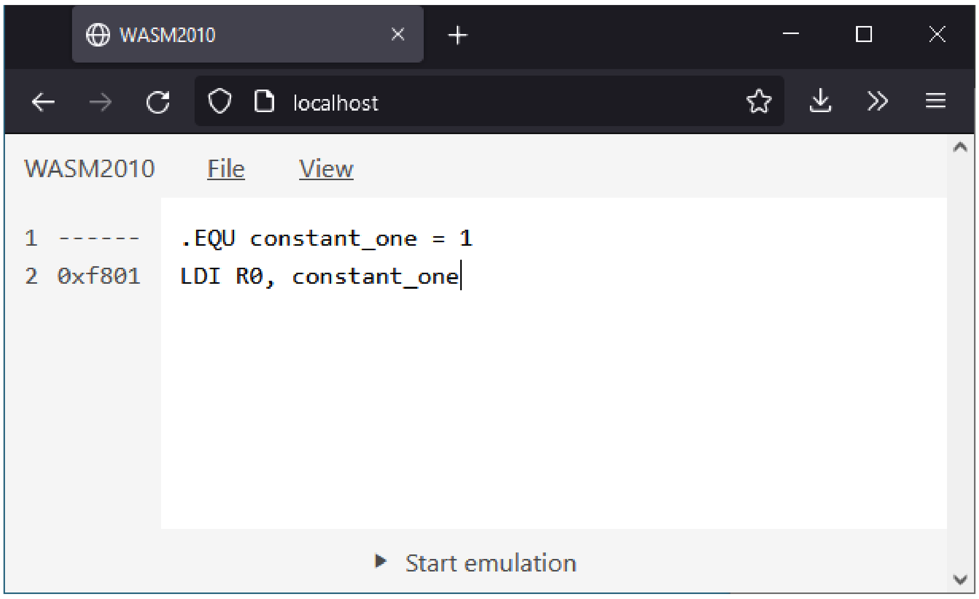


Figure 7-1. WASM2010's assembly editor view

The assembler view features a prominent source assembly editor that allows users to write their programs, with a menu at the top of the page and a large button to start the emulation. The editor provides basic unformatted text operations. Each line is numbered, starting from 1, at the left. A series of hyphens or a numerical value follows the numeration, representing the output machine code generated from the line, as seen in Figure 7-2.

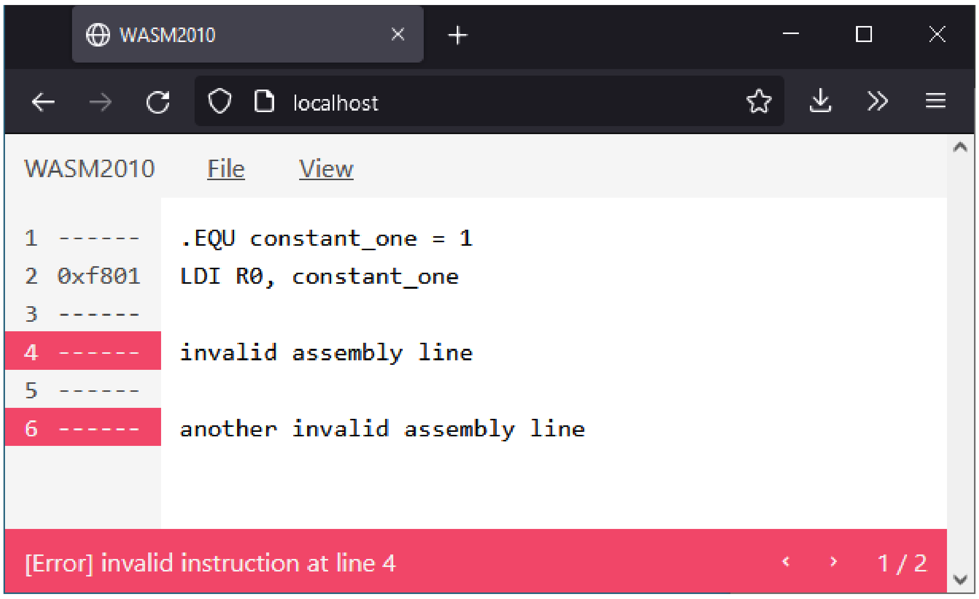
WASM2010 generates the machine code simultaneously when the user types or changes anything in the text editor.

The hyphens will appear whenever the line does not produce machine code, either because it is not a valid assembly line or it is an assembly directive. Otherwise, the editor will show the output machine code using the radix chosen by the user, which is hexadecimal by default.



```
1 ------   .EQU constant_one = 1
2 0xf801   LDI R0, constant_one
```

Figure 7-2. Line numbering and generated machine code



```
1 ------   .EQU constant_one = 1
2 0xf801   LDI R0, constant_one
3 ------
4 ------   invalid assembly line
5 ------
6 ------   another invalid assembly line
```

[Error] invalid instruction at line 4

Figure 7-3. Invalid assembly lines

If the user inputs one or multiple invalid assembly lines, the editor will highlight them in red. It also will replace the big button at the bottom of the page with a panel showing the errors that occurred during the assembly. This panel will always show the first error encountered, allowing the user to navigate the different error messages using two arrow buttons on the right side, followed by an error counter.

Any error present in the assembly code will prevent the user from starting the emulation. After the user fixes all the mistakes, the panel will disappear, and the editor will show the button to begin the emulation again.

Although the assembler will find and log any error present in the source assembly code, it will log only a fixed amount of errors until it halts the logging for performance reasons. If the assembly code contains more than a certain amount of invalid lines (around 30), the assembler will show only the first errors encountered. Only after the user fixes one of the present errors will the assembler continue displaying the remaining ones, if there are any.

In the same way, the assembler will not log more than one error in each line. If multiple errors exist in the same line, the assembler will stop after the first one encountered. When the user fixes this error, the following errors will be shown in order of appearance when it follows.

7.1.1 Managing source files and loading examples

Users can load local files to the assembly editor and save them later to their local drive. They can access these features from the top menubar, shown in Figure 7-4, after clicking in the *File* section.

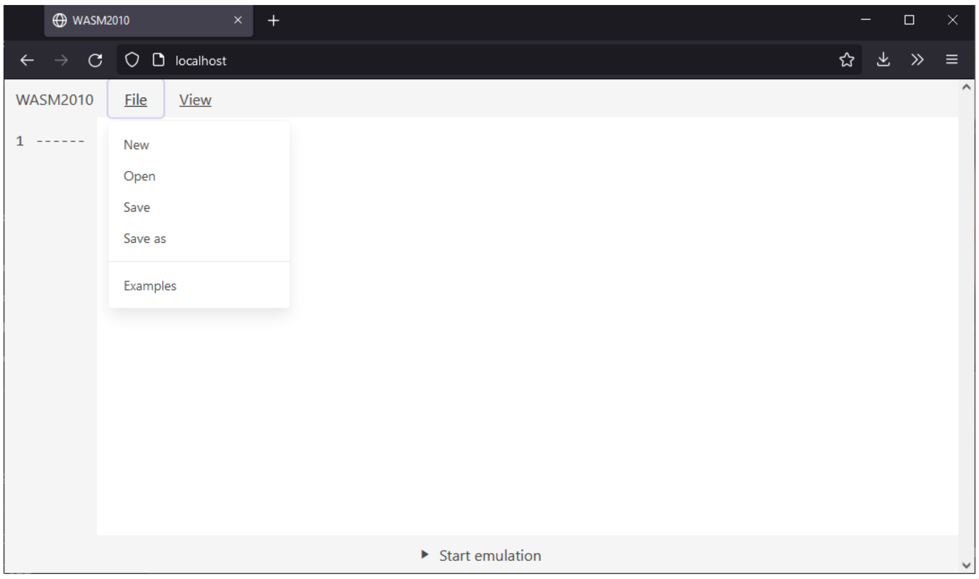


Figure 7-4. List of file-related features

Moreover, the web application features some example assembly programs that the user can load into the assembly editor. Clicking on the *Examples* button will open a sidebar, allowing the user to choose and load an assembly program, as shown in Figure 7-5.

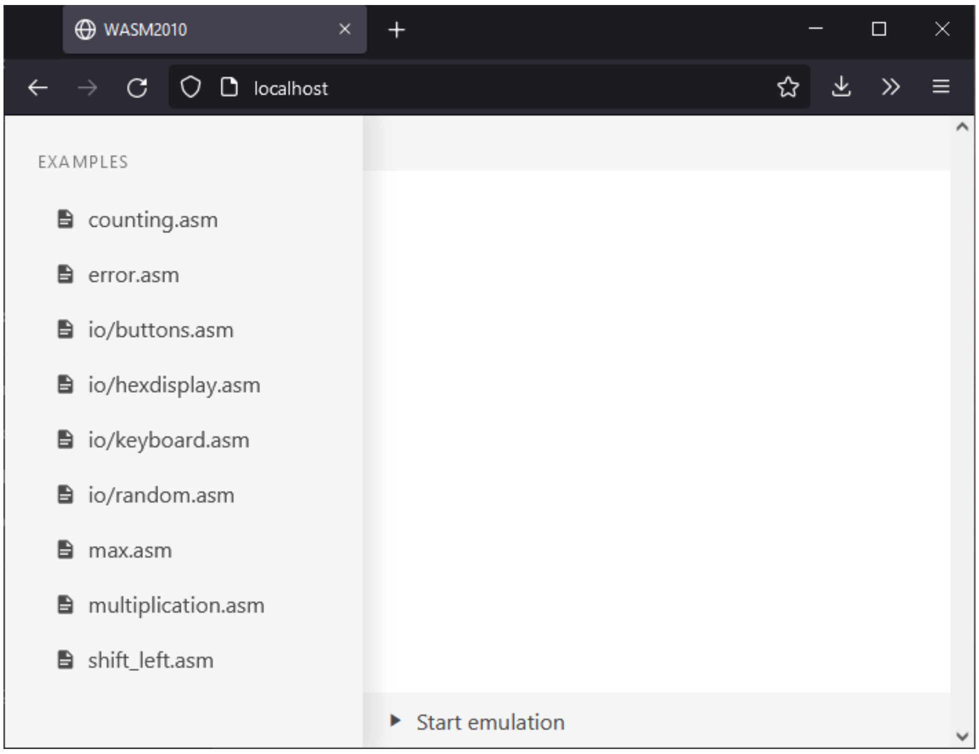
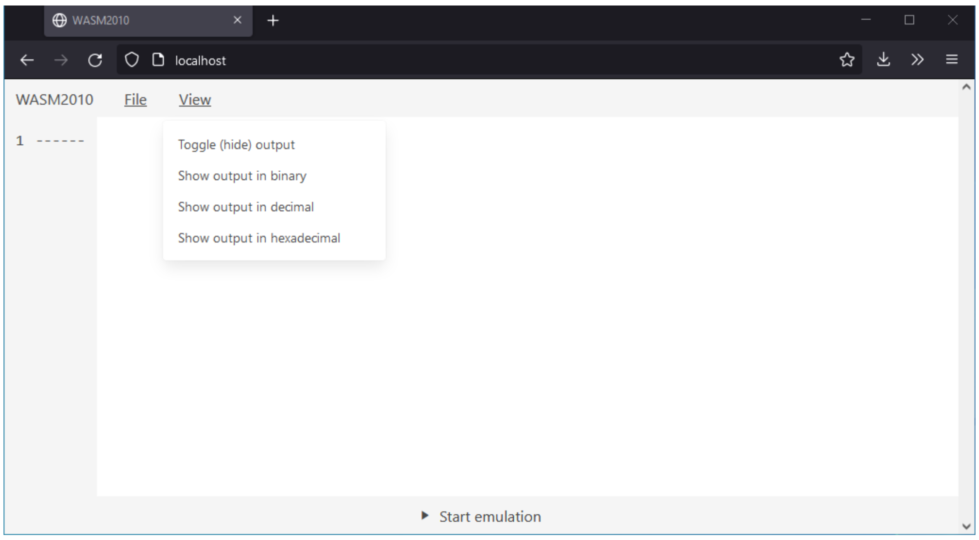


Figure 7-5. Example programs listed in the sidebar

7.1.2 Customizing the assembly editor

Even though the editor tries its best to be as simple as possible and prioritizes offering key features, the user can do some customisation. Clicking on the View section in the top menubar will display a list of options that can be used to customize the assembly editor appearance. These options can be seen in Figure 7-6.

It is possible to hide and show the output machine code for every assembly line using these options. Moreover, it is possible to choose the numerical radix that the editor will use to show the generated machine code. Binary and decimal radices do not prefix the output; therefore, the machine code will be shown as a simple number. The hexadecimal radix will prefix numbers with *0x*, similar to how hexadecimal values are commonly written. If the output machine code is hidden, but the radix is changed using any of these options, it will be automatically shown again.

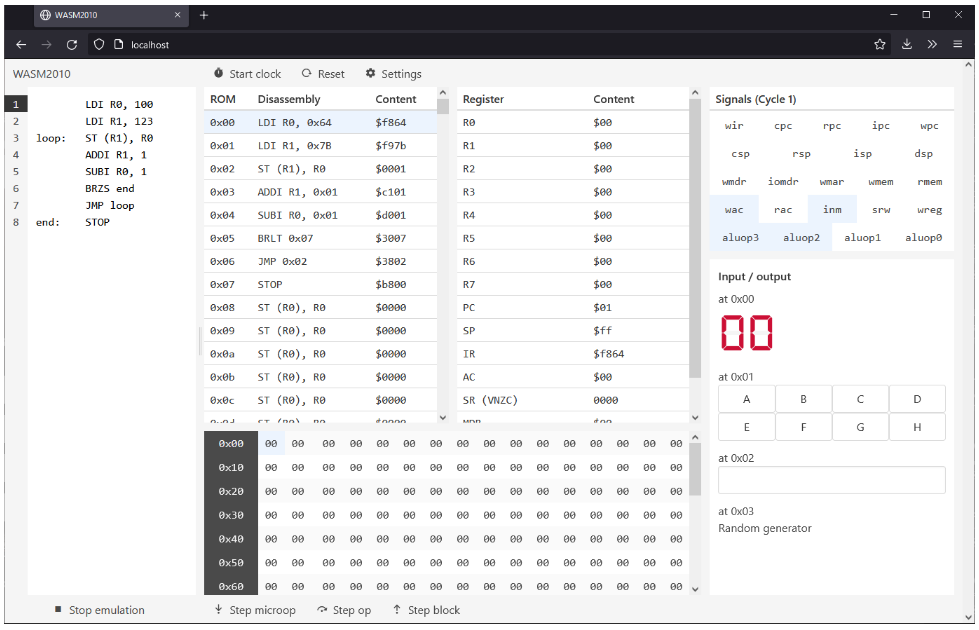


```
------
```

Figure 7-6. List of the editor's customization options

7.2 Using the emulator

After the source assembly is done, and if no errors have been encountered during the assembly stage, the *Start emulation* button will appear at the bottom of the page. After clicking on it, the user will enter the emulator view. The assembly editor and the Stop emulation button will be shrunk to the left side of the page together, allowing the user to stop the emulation and return to the assembly editor view again.



```
        LDI R0, 100
        LDI R1, 123
loop:   ST (R1), R0
        ADDI R1, 1
        SUBI R0, 1
        BRZS end
        JMP loop
end:    STOP
```

ROM	Disassembly	Content
0x00	LDI R0, 0x64	$f864
0x01	LDI R1, 0x7B	$f97b
0x02	ST (R1), R0	$0001
0x03	ADDI R1, 0x01	$c101
0x04	SUBI R0, 0x01	$d001
0x05	BRLT 0x07	$3007
0x06	JMP 0x02	$3802
0x07	STOP	$b800
0x08	ST (R0), R0	$0000
0x09	ST (R0), R0	$0000
0x0a	ST (R0), R0	$0000
0x0b	ST (R0), R0	$0000
0x0c	ST (R0), R0	$0000

Register	Content
R0	$00
R1	$00
R2	$00
R3	$00
R4	$00
R5	$00
R6	$00
R7	$00
PC	$01
SP	$ff
IR	$f864
AC	$00
SR (VNZC)	0000

```
0x00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

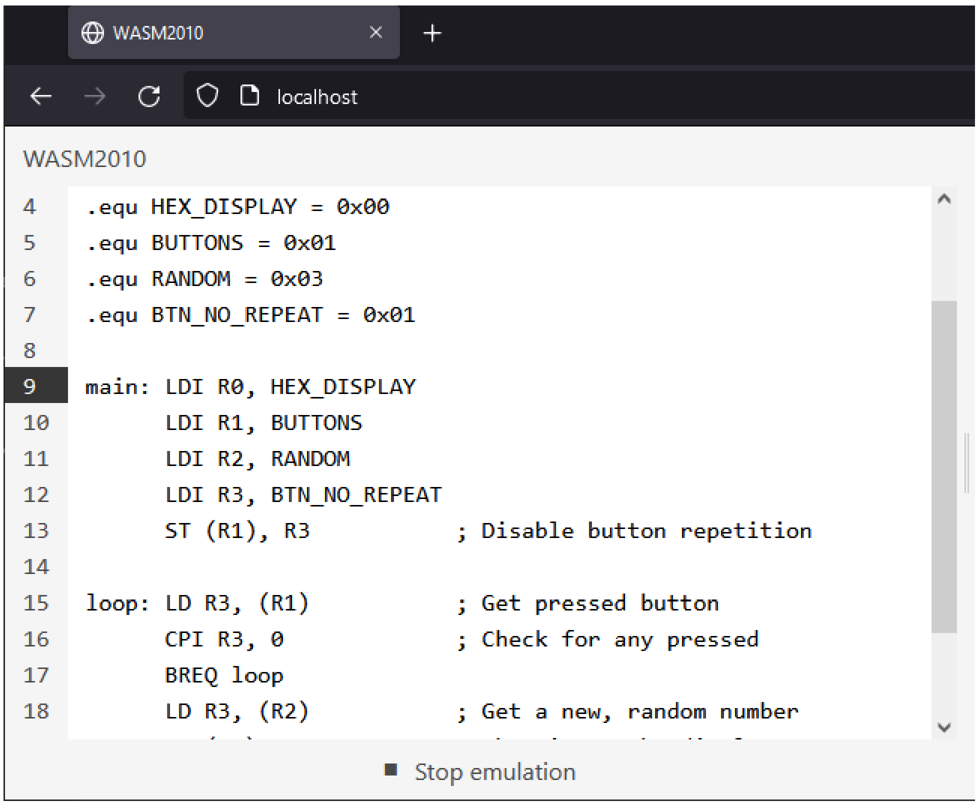
Figure 7-7. WASM2010's emulator view

As can be seen in Figure 7-7, the assembly editor has moved to the left side of the page, and the source cannot be edited during the emulation. At its immediate right, we can see the emulator with all its views. Enumerating them from the top-left corner to the right-bottom corner: the source view, the ROM view, the RAM view, the registers view, the signals view and the I/O view. Each view takes part in the emulation and is connected to the same CS2010 computer. We will study them carefully in the following sections.

7.2.1 Source view

The primary function of the source view is to show the source code currently running in the emulator. In addition, it highlights the line of source code that generated the machine code that is running at the moment, i.e., the content of the IR register. It is important not to mistake the highlighted line with the next instruction to fetch, which is pointed out by the PC register.

Whenever the current instruction changes, the source view will scroll automatically to fit the corresponding line of source code close to the view's centre, as shown in Figure 7-8. The view allows the user to expand and shrink it using the resizer at the right.

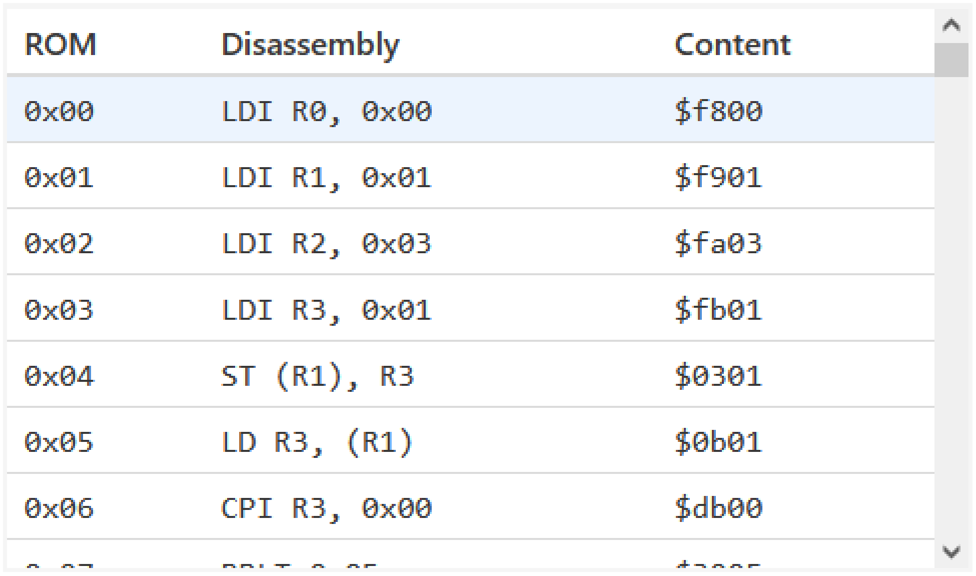


```
.equ HEX_DISPLAY = 0x00
.equ BUTTONS = 0x01
.equ RANDOM = 0x03
.equ BTN_NO_REPEAT = 0x01

main: LDI R0, HEX_DISPLAY
      LDI R1, BUTTONS
      LDI R2, RANDOM
      LDI R3, BTN_NO_REPEAT
      ST (R1), R3          ; Disable button repetition

loop: LD R3, (R1)          ; Get pressed button
      CPI R3, 0            ; Check for any pressed
      BREQ loop
      LD R3, (R2)          ; Get a new, random number
```

Figure 7-8. Emulator’s source view

7.2.2 ROM view

This view displays the content of the CS2010's ROM (see 2.2.2) in a table, as shown in Figure 7-9.



ROM	Disassembly	Content
0x00	LDI R0, 0x00	$f800
0x01	LDI R1, 0x01	$f901
0x02	LDI R2, 0x03	$fa03
0x03	LDI R3, 0x01	$fb01
0x04	ST (R1), R3	$0301
0x05	LD R3, (R1)	$0b01
0x06	CPI R3, 0x00	$db00

Figure 7-9. Emulator's ROM view

The first column corresponds to the address where each machine instruction is in the memory. The view always displays the entire memory, no matter how many addresses are filled with user machine instructions.

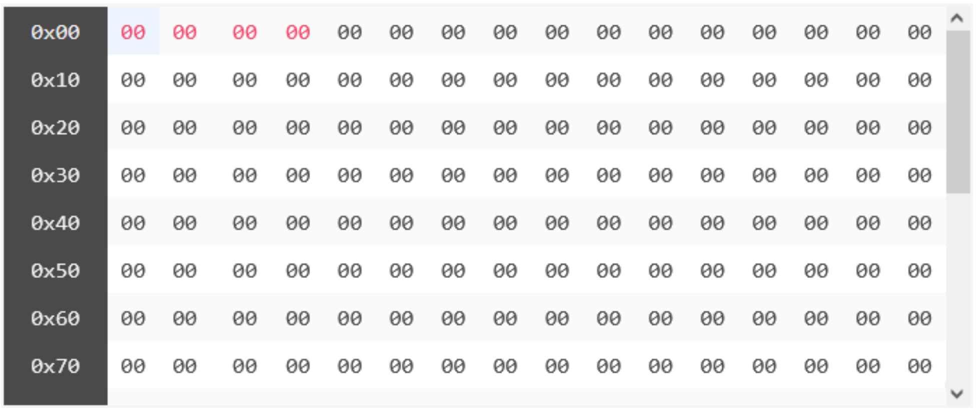
Following this column, we find the disassembly of the machine instruction, whose content resembles the matching line of source code in the source view. As the view does not distinguish between addresses that contain instructions coded by the user and addresses that contain garbage, the latter will be disassembled as indirect load instructions. Moreover, every assembly directive (such as EQUs) will be replaced with the corresponding numerical value, and the line will be free of comments, making it closer to what the CS2010 sees.

The last column depicts the content of each address. The radix used to show the content can be customized by the user from the emulator settings. Unlike other views, hexadecimal values are prefixed with a dollar ($) instead of the prefix 0x, as this helps to reduce the column's width.

Finally, similar to the behaviour of the source view, the currently executed instruction is highlighted. During the execution, it will be highlighted with a light blue tone, but it will turn to a light purple tone after the execution has finished.

7.2.3 RAM view

The view displays the content of each address in the CS2010's RAM (see 2.2.1), as seen in Figure 7-10. The content is displayed in a grid with 16 rows and 16 columns each. This way, each cell maps directly to one byte of the memory. The user can customize this layout from the emulator settings.

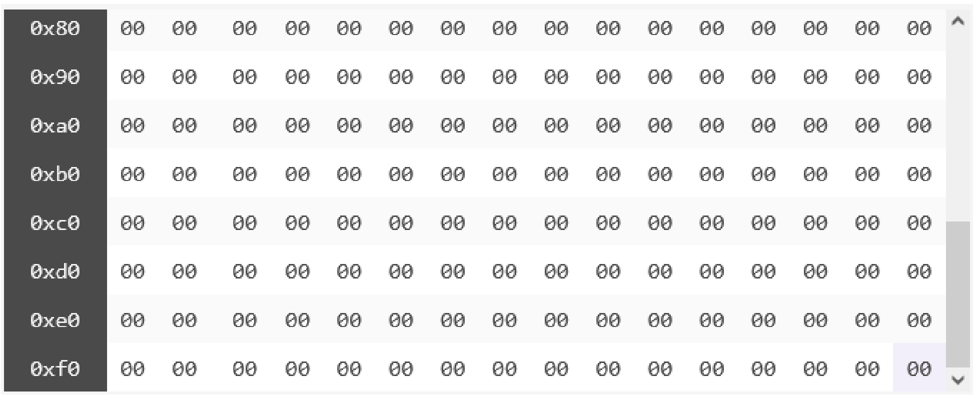


0x00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 7-10. Emulator's RAM view

Each cell represents the content of a given address using a numerical value. As with other views, the radix can be customized as well. In contrast, hexadecimal numbers are not prefixed at all this time.

Some cells have their background or text colour modified to point out some unique condition: a cell with a light blue background contains the address to which MAR points. A cell with light purple background indicates that it contains the address where SP points to, i.e., the first empty byte in the stack. A cell with red text means that its address is mapped to an input/output component; therefore, its content will not be displayed, even if the I/O component allows it to be read.

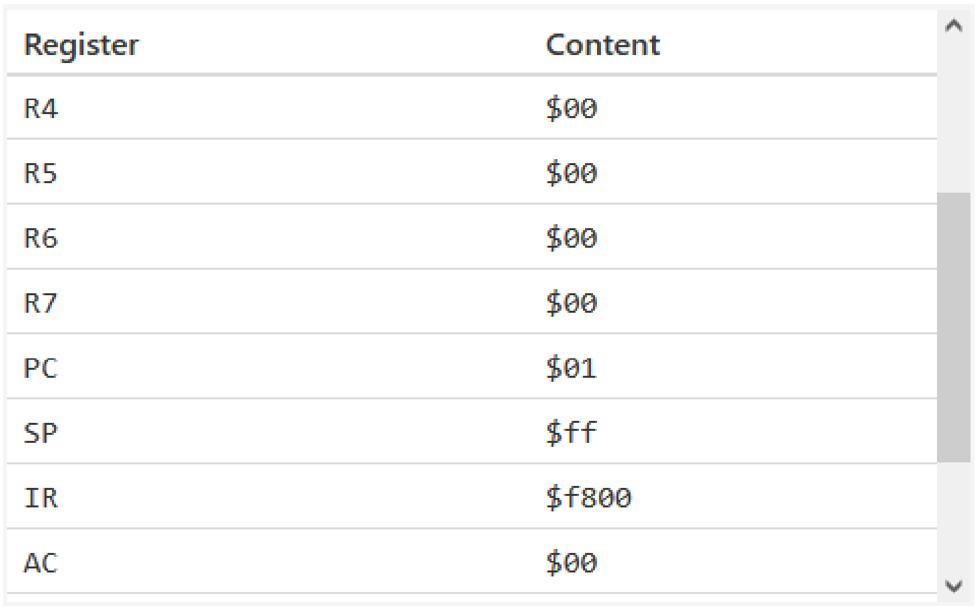


0x80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xa0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xb0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xc0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xd0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xe0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xf0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 7-11. Highlighted stack pointer in the RAM view

7.2.4 Registers view

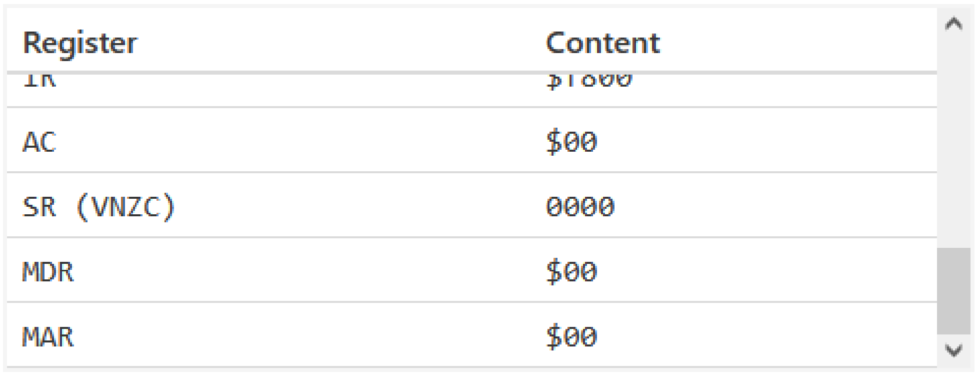
The registers view presents a table where each row contains the name and the value of a register (see 2.3), as seen in Figure 7-12.



Register	Content
R4	$00
R5	$00
R6	$00
R7	$00
PC	$01
SP	$ff
IR	$f800
AC	$00

Figure 7-12. Emulator's registers view

Content is displayed similarly to the ROM view, with hexadecimal values prefixed with a dollar sign. There is a single exception to this rule: the status register (described in 2.3.2) will always be represented using four individual bits using a binary radix. Each bit corresponds to the condition located at the same index inside the parenthesis. The user can customize the radix from the emulator's settings as with other views.

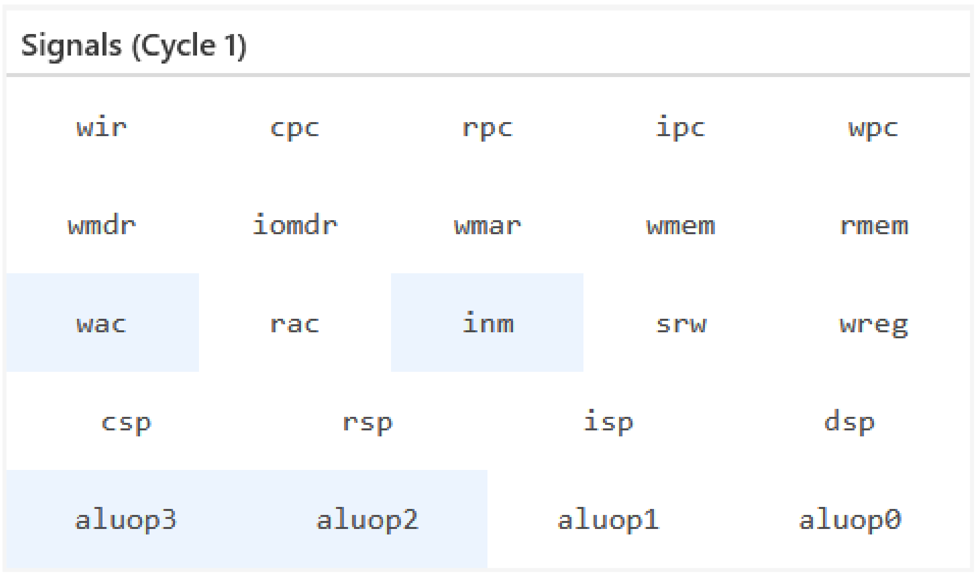


Register	Content
IR	$f800
AC	$00
SR (VNZC)	0000
MDR	$00
MAR	$00

Figure 7-13. Depiction of the status register

7.2.5 Signals view

The signals view presents the state of every control unit's signal in a grid (see 2.7). Each cell in the grid represents a single signal. The background colour changes depending on the state of the signal in a given clock cycle: it will become white when the signal is off and light blue when it is on, as shown in Figure 7-14.



Signals (Cycle 1)

wir	cpc	rpc	ipc	wpc
wmdr	iomdr	wmar	wmem	rmem
wac	rac	inm	srw	wreg
csp	rsp	isp	dsp	
aluop3	aluop2	aluop1	aluop0	

Figure 7-14. Emulator's signals view

Each grid row contains signals belonging to the same component to improve readability (see fourth row, grouping SP signals). In some cases, these signals belong to a similar part of the computer instead (the second row shows MDR, MAR and RAM signals; they all take part in RAM operations).

In addition, the view's title indicates which cycle of the current instruction is being executed at the given clock cycle. Together with the signals' state, this helps understand how instructions are performed in terms of microoperations.

7.2.6 I/O view

The ASM2010 kernel implements the mechanism described in the CS2010 architecture specification (see 2.8) to support external I/O devices. WASM2010, in turn, provides the user with some essential I/O devices that can be arbitrarily mapped to the memory. Each device defines its user interface as a web component that will be rendered inside this view and directly connected to the kernel, as shown in Figure 7-15. This view allows us to see and interact with these devices.

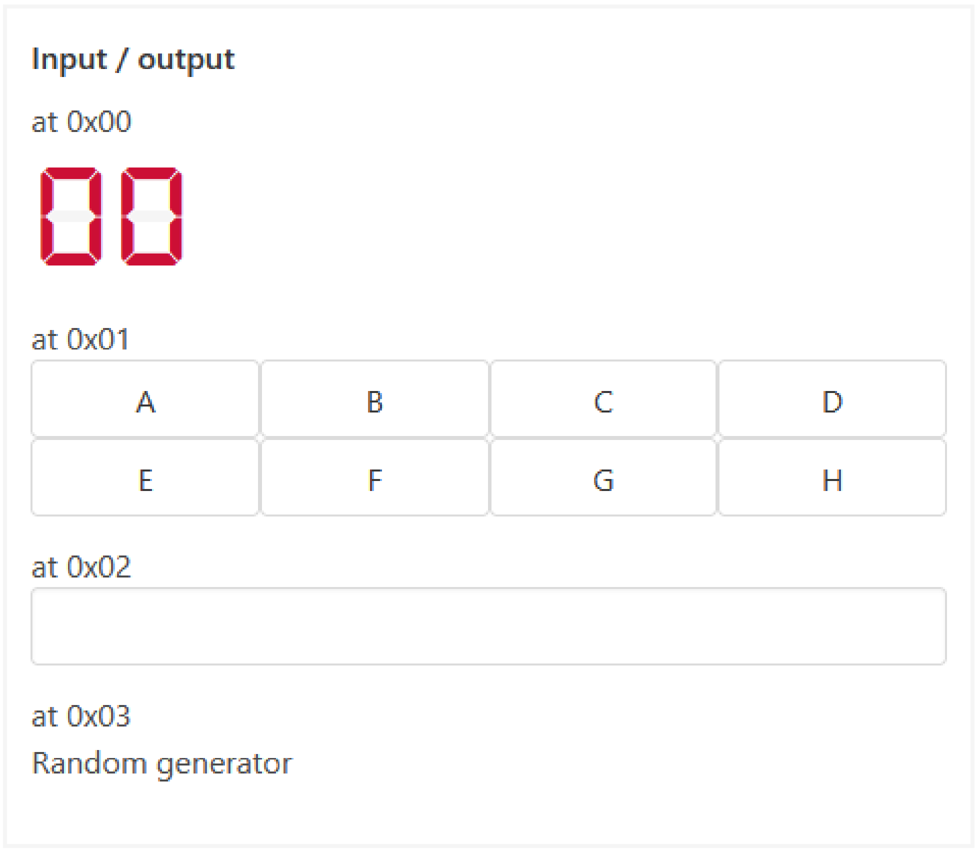
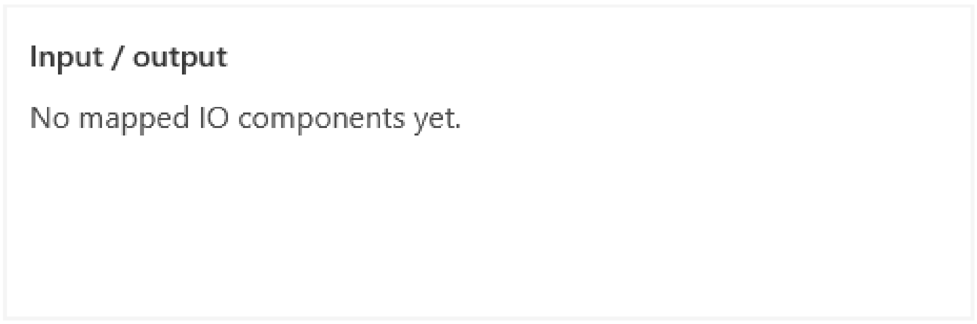


Figure 7-15. Emulator's input/output view

The view presents a list of the currently mapped components, ordered by their addresses in ascending order. Each component gets a small label immediately above to indicate which address it is mapped. The view will show a short message if no components are mapped, as seen in Figure 7-16. Input/output view with no mappings. When a component has no user interface (such as the random generator), its name will be shown instead. The following sections will discuss the functional description of each component and how the user can customize these mappings.


Input / output

No mapped IO components yet.

Figure 7-16. Input/output view with no mappings

7.2.7 Running the emulation manually

Once in the emulator view, the user can manually control the execution of each instruction in the emulation using the controls located in the bottom bar, as shown in Figure 7-17.

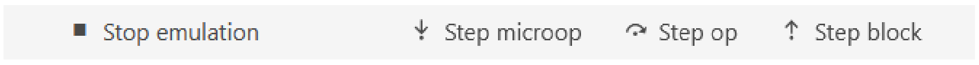


Figure 7-17. Emulator bottom bar controls

As discussed, the *Stop emulation* button allows the user to stop the emulation entirely and return to the assembly editor view, wiping out the emulator's state. The assembly code that the emulator was executing will become editable again. Even if the code has not changed, starting the emulation again will wipe the previous emulation’s state, as if the code is running on a new computer.

The following three buttons allow the user to forward the emulation in different ways:

- **Step microop.** Steps a single clock cycle, meaning only a microoperation of the current instruction will be performed. The clock cycle counter will increment unless the final microoperation has been reached, in which case the counter will reset, and the computer will fetch the following instruction.
- **Step op.** Steps a whole instruction, i.e., the number of clock cycles needed to execute the current instruction entirely and fetch the next instruction. Supposing the computer is in the middle of executing an instruction (i.e. the user has stepped a single clock cycle, but the instruction did not end), it will perform the pending microoperations for the current instruction and fetch the next one.
- **Step block.** Steps a block of instructions, i.e. steps instructions indefinitely until one of the following conditions is met:
    - The computer fetches a branching instruction.
    - The computer halts the execution due to a STOP instruction.
    - The computer reaches the limit of executed instructions before halting.

  The latter prevents the execution from entering an endless loop in case no branching nor STOP instructions appear in the code (note that the PC exhibits modulo behaviour). A notification will appear to notify the user if this limit is reached, as shown in Figure 7-18. The user can modify this limit from the emulator settings, which we will discuss subsequently.

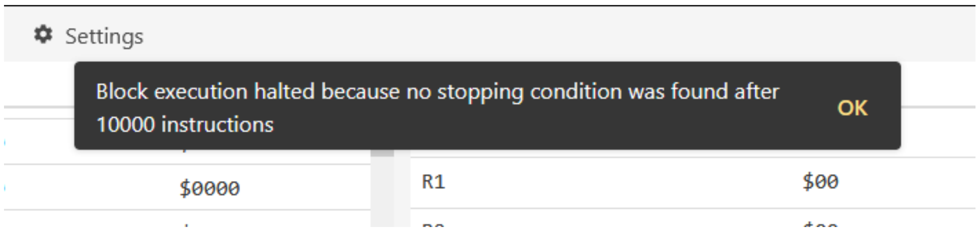

Block execution halted because no stopping condition was found after 10000 instructions

Figure 7-18. Block execution halting notification

7.2.8 Running the internal clock

While manually executing instructions can be the most helpful way to understand how a program works, sometimes it is worthwhile to see the output of a program to see how it behaves at a higher level. Input/output devices, such as hexadecimal displays, also require some high refresh rates to see them in action, which needs the execution to be performed according to a high-speed clock.

For this purpose, the user can click on the *Start clock* button to start the CS2010's internal clock, as shown in Figure 7-19. As soon as the clock starts, the button will be renamed to *Stop clock*, allowing the user to stop the emulation. Every other control will be disabled while the clock runs, except for the *Stop emulation* button, which will halt the clock and wipe out the simulation. Other parts of the UI, such as the views, are not affected by this condition and will behave as usual, allowing the user to interact, for example, with I/O devices in real time.

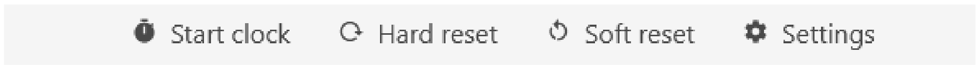
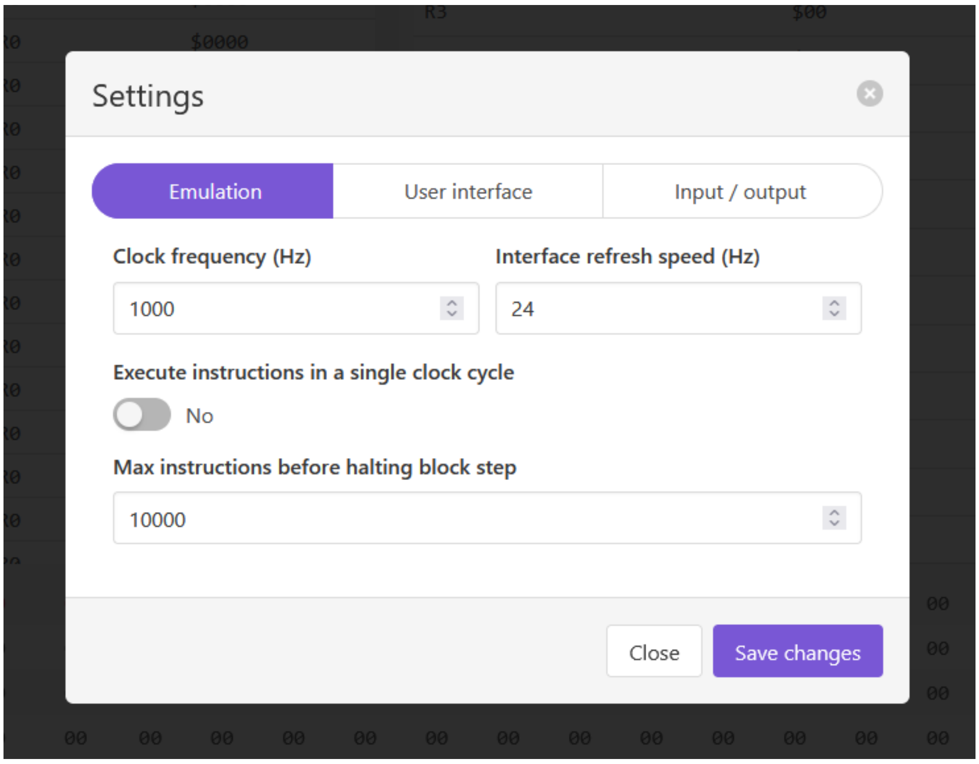


Figure 7-19. Emulator top bar controls

Apart from the clock, the emulator provides two different ways to restart the current simulation without quitting the current view. Clicking on the hard reset button clears the emulation completely, resetting every register and memory to its default value. In contrast, clicking on the soft reset button only clears the necessary registers to restart the emulation from the beginning. (See 2.7.4)

7.3 Customizing the emulator settings

As previously mentioned, the user can customize the emulator from the settings popup. To open it, the user must click the *Settings* button in Figure 7-19. The popup will appear, and the background will turn dim, blocking any other interaction of the user with the rest of the emulator, as shown in Figure 7-20.


Settings

Clock frequency (Hz): 1000

Interface refresh speed (Hz): 24

Execute instructions in a single clock cycle: No

Max instructions before halting block step: 10000

Figure 7-20. Emulator’s settings popup

Settings are divided into three categories: emulation, user interface and input/output. Users can navigate between categories using the tabs below the popup’s header. Note that settings cannot be opened while the clock of the CS2010 is running.

Every setting is given a default value when the user WASM2010 is loaded. Changes in settings will not be persisted across sessions, so reloading the page will load default values again.

All settings in all categories must be valid before the user can confirm the changes by clicking on the Save changes footer button. New settings will not be persisted until the changes are approved. Closing the popup when there are unconfirmed changes, either by clicking on the *Close* button, the small X or clicking outside the popup will result in discarding the unconfirmed changes.

Every validation condition must hold for a field to be considered valid. The required format or range for each field will be shown in an error message, usually between parenthesis, as shown in Figure 7-21. When a field does not match the required format or is out of range, the confirmation button will be disabled, and an error message will be shown below the corresponding field until the user fixes its value or the popup is closed.

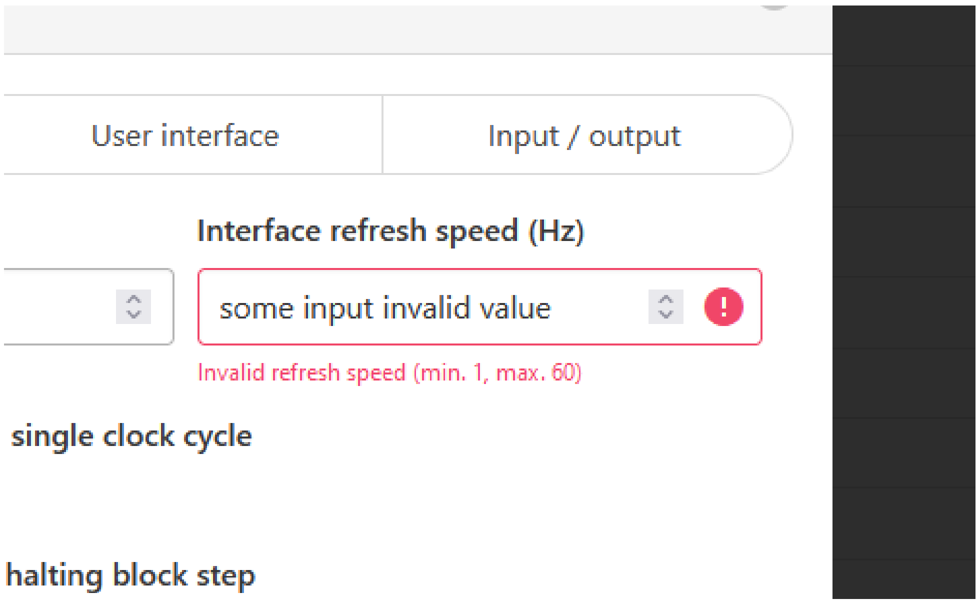


Figure 7-21. Example of an error message in a setting

We will provide a brief explanation of each category in the following sections.

7.3.1 Emulation settings

This category contains settings related to the emulation kernel.

- **Clock frequency (Hz).** The frequency at which the CS2010's internal clock will run. For instance, setting it to 60 Hz will result in 60 clock cycles being executed each second. The minimum frequency allowed is 1 Hz, and the maximum is 100,000 Hz.

  Higher frequencies will consume more CPU. It is not guaranteed that the emulation will run at this frequency all the time, especially when more CPU-demanding values are set (this depends heavily on how fast the host CPU is). However, it is guaranteed that it will not run at higher frequencies. The maximum possible frequency will be used when the requested frequency cannot be achieved.

- **UI refresh frequency (Hz).** The frequency at which the user interface (UI) will be refreshed. For instance, setting it to 30 Hz will result in 30 updates each second. The minimum frequency allowed is 1 Hz, and the maximum is 60 Hz.

  Refreshing the UI means updating every view connected to the CS2010, such as the memories, registers and signals. Therefore, this frequency is different from the one at which the browser re-

renders the application (which is not controllable), and therefore it does not affect the browser responsiveness.

Input/output devices are also affected by this speed. While user input will still be sent to the emulation kernel immediately, output devices' views will be refreshed together with other views. For instance, if a user sets a 1 Hz UI refresh frequency and clicks a button, the signal will be processed by the application and the emulation kernel instantly. However, any change this signal will cause in any part of the CS2010, including external output devices, will be observable only after the corresponding view is refreshed.

As with the clock frequency, it is not guaranteed that the interface will be refreshed at this frequency all the time, especially when more CPU-demanding values are set (this depends heavily on how fast the host CPU is). However, it is guaranteed that it will not be refreshed at higher frequencies. The maximum possible frequency will be used when the requested frequency cannot be achieved.

- **Execute instructions in a single clock cycle.** When this option is enabled, every instruction will take a single clock cycle to execute instead of a variable amount depending on the instruction. While the user can already forward microoperations and operations manually, this is useful when the CS2010's clock is running.
- **Max. instructions before halting block step.** The maximum number of instructions to be executed before a block step halts due to exhaustion, i.e. no stopping condition has been found during the execution. This behaviour is described in 7.2.7.

7.3.2 User interface settings

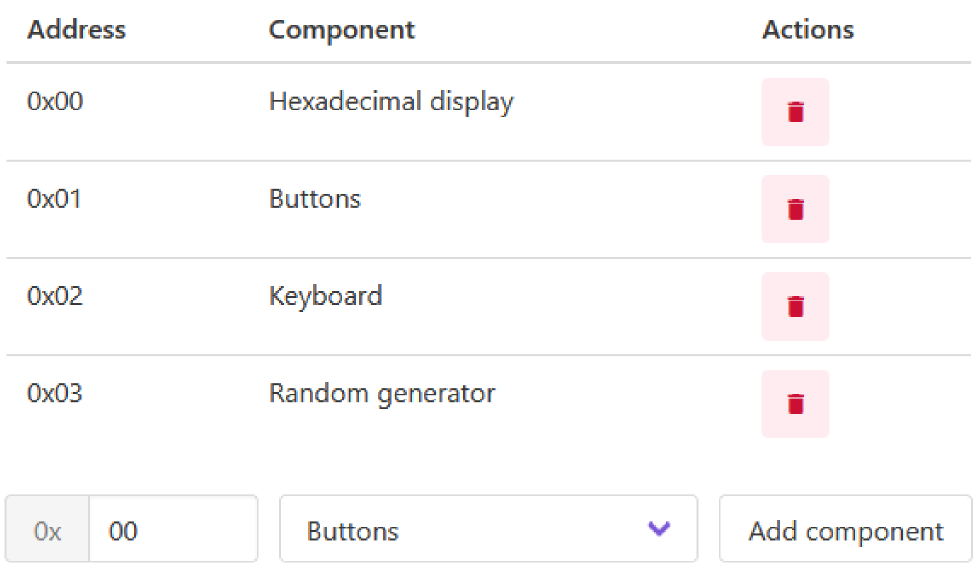
This category contains some settings that apply to the user interface itself and do not interfere with the emulation execution.

- **ROM content radix.** Sets the radix that will be used to display values in the content column of the ROM view.
- **RAM content radix.** Sets the radix that will be used to display values in each cell of the RAM view.
- **Register content radix.** Sets the radix that will be used to display values in the content column of the registers view.
- **RAM words per row.** Sets how many bytes (words) will be shown per row. Valid values range from 1 to 32 (inclusive). The amount of rows that will be displayed depends on this value, but it is guaranteed that the first row will always start with the byte 0, and the last row will contain the last byte in the memory. If

the last byte is not placed at the end of the last row, every subsequent byte will be filled with zeroes.

7.3.3 I/O settings

This category allows the user to manage the mapped I/O devices. The table will list each device and the address where it is mapped in the memory, as shown in Figure 7-22. Every device can be unmapped by clicking on the corresponding trash can button.



Address	Component	Actions
0x00	Hexadecimal display	🗑
0x01	Buttons	🗑
0x02	Keyboard	🗑
0x03	Random generator	🗑

Figure 7-22. I/O mapping list

To add a new component, the user must input the address where it will be mapped. The address must be written with up to two hexadecimal digits. A single hexadecimal digit will be considered the least significant nibble in the address. After selecting a new component and clicking the add component button, the new component will be registered. Choosing an address already in use will replace the current mapping with the new component. The user must confirm the changes for new devices to be mapped to the memory. Otherwise, no changes will be applied, and the previous devices will be restored.

7.4 Using I/O components

For the user to utilize the I/O components, it is crucial to understand how they map their values to the memory and which read and write operations are allowed. We will provide an in-details description of each device, its behaviour during reading and writing operations, and how the user can interact with them. Note that not all devices might allow performing specific operations on them. Performing these operations might result in the device having undefined behaviour, although it is guaranteed that this will not have side effects on the CS2010.

7.4.1 Hexadecimal display

This device allows displaying two hexadecimal digits using a single byte of memory. Each digit is mapped to each half of the byte: the leftmost digit (L) is mapped to the most significant nibble, while the rightmost digit (R) is mapped to the less significant nibble.

- **Reading.** The result is one byte containing both digits, as shown in Table 7-1.
- **Writing.** Updates the digits on the screen, according to Table 7-1.

	MSB							LSB
Bit	7	6	5	4	3	2	1	0
Content	L_3	L_2	L_1	L_0	R_3	R_2	R_1	R_0

Table 7-1. Hexadecimal display byte mapping

7.4.2 Buttons

This device provides two rows of four buttons each, labelled with a letter from A to H, making up eight buttons. Each button is mapped to a bit (signal) of the mapped byte. When a button is clicked, its corresponding signal is set. Only one signal can be set at a given time, so clicking other buttons will not have any effect until the last signal has been read and the previous button is released. If all signals are cleared, no button has been pressed since the last reading.

- **Reading.** The result is a byte containing all the signals, each mapped to a bit, as shown in Table 7-2. After the byte has been read, the current signal will be cleared.

	MSB							LSB
Bit	7	6	5	4	3	2	1	0
Content	H	G	F	E	D	C	B	A

Table 7-2. Buttons signal mapping

- **Writing.** Sets the device's reading mode. This mode determines how and when signals are cleared, according to Table 7-3. Bits marked with a hyphen can take any value.

	MSB							LSB
Bit	7	6	5	4	3	2	1	0
Content	-	-	-	-	-	-	-	M

Table 7-3. Buttons configuration mapping

M can take any value from Table 7-4.

M	Mode	Behaviour
0	Default mode	The current signal will be available for reading until the button is released and the last available signal has been read.
1	Non-repeat mode	The current signal will be available for reading once. After that, it will be cleared, even if the button is still pressed.

Table 7-4. Buttons reading modes

7.4.3 Keyboard

This device provides a textbox for the user to input data. The device has a first-in, first-out (FIFO) buffer with space for storing up to 256 characters encoded using ASCII. Each character (C) is mapped to a byte, with the most significant bit set to 0, as shown in Table 7-5. If the buffer is full, any new character will be discarded. Note that any input from the physical keyboard will be mapped to an ASCII character, including returns, backspaces or other special keys.

	MSB							LSB
Bit	7	6	5	4	3	2	1	0
Content	0	C_6	C_5	C_4	C_3	C_2	C_1	C_0

Table 7-5. Keyboard ASCII character mapping

- **Reading.** Depending on the configured reading mode, the result will be either a byte containing the first unread character or a byte with its less significant bit set to 1 if there is any unread character in the buffer. The first unread character is popped from the buffer in the first case.
- **Writing.** Allows to configure the device and to perform actions, depending on the input, as shown in Table 7-6. Bits marked with a hyphen can take any value.

	MSB							**LSB**
Bit	7	6	5	4	3	2	1	0
Content	-	-	-	-	-	-	M_1	M_0

Table 7-6. Keyboard configuration mapping

M_0 and M_1 can take any value from Table 7-7. Actions marked with a hyphen are not defined and will not have any side effects.

M_1	**M_0**	**Action**	**Behaviour**
0	0	Clear buffer	Clears the buffer, discarding any unread characters.
0	1	Set peek mode	Sets the peek mode. Next time the device is read, it will return a byte with its less significant bit set to 1 if there is any unread character in the buffer or to 0 otherwise. The device will be set to regular mode immediately after this.
1	0	Set regular mode	Sets the regular mode. Next time the device is read, it will return a byte containing the first unread character in the buffer.
1	1	-	-

Table 7-7. Keyboard actions

7.4.4 Random generator

This device provides a way to obtain pseudorandom numbers and does not provide a graphical interface. It will generate a new pseudorandom number (R) after each time it is read, as shown in Table 7-8.

	MSB							**LSB**
Bit	7	6	5	4	3	2	1	0
Content	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0

Table 7-8. Random generator mapping

- **Reading.** The result is a byte with all its bits set to 0 or 1 randomly, representing a random 1-byte number.
- **Writing.** This operation will not have any side effects.

8 Conclusions and future work

The development of a software toolkit capable of emulating the CS2010 architecture will be a great help to many students and teachers who are taking or teaching subjects related to computer architecture. WASM2010 covers many use cases that can arise in these subjects: a student could use it to test their knowledge of how the computer works or even to code assembly programs as homework. On the other hand, teachers can use it during their lessons to show every aspect of the computer in a practical manner.

The ease of use and the portability are two key points here: virtually, the toolkit can be run on almost any platform, even in those with meagre resources, thanks to its lightweight emulation kernel. Nowadays, the web is a stimulating environment; users typically prefer web applications that run directly from their browsers. Therefore, offering a web application is one of the keys to increasing the market share. Moreover, the WASM2010's user interface has been designed with the needs of all users in mind, providing a lean-and-mean experience that will help them to satisfy their needs.

Aside, WASM2010 performs a good exercise in increasing the number of open-source software projects. Not only can WASM2010 be used, modified and distributed freely, but Jasm has excellent potential as a stand-alone library that helps C programmers to port their programs to the web or any environment where JavaScript and WebAssembly coexist.

Similar projects to Jasm are Emscripten [24] and wasmbindgen [25], both being a complete toolchain to compile C/C++ and Rust projects, respectively, to WebAssembly. However, they aim at different targets: while they focus on providing Web APIs and other mappings directly to native programs, Jasm tries to give the JavaScript programmer minimal tools to help interoperate with programs written in C with little to no modifications.

8.1 Future work

As in any software project, there is always room for improvement. Both ASM2010, Jasm and WASM2010 can be improved in many different ways.

We will provide a comprehensive list of ideas that future work might consider to enhance these projects.

8.1.1 Ideas for ASM2010

- Porting the project to other languages, such as Rust, Golang, Python, and Java. Creating bindings for these languages is a good idea to reuse the existing code base.

8.1.2 Ideas for Jasm

- Jasm has not yet reached its first major version; therefore, it still lacks a stable public API. Further investigations into the structure of the project and its functionality would be a big step in this direction.
- Defining structures in JavaScript is tedious and error-prone. If definitions change on either side but not on the other, the code will break or behave unexpectedly. Generating definitions automatically from the C source would be a good step in improving its functionality.
- Jasm lacks support for future 64-bit WebAssembly [11]. It is likely that both versions, 32 and 64 bits, will have to coexist. Adding support for future WebAssembly specs would improve the code maintainability.
- Jasm lacks support for C++. While it might be out of scope, it could be helpful to investigate whether it is worth adding support for its structures and classes.

8.1.3 Ideas for WASM2010

- While the assembly editor is enough to write assembly, it lacks some additional features that more powerful source code editors feature, such as syntax highlighting or autocompletion. Implementing them would improve the user experience.
- Adding a visual representation of the CS2010's data and control paths would improve the user experience, allowing the user to observe how the data flows in a more visually appealing way.

Bibliography

[1] Ruiz, J., Guerrero, D., Gomez, I., & Viejo, J. (2012). Implementation of a hardware and software framework for a simple academic processor. *2012 Technologies Applied to Electronics Teaching (TAEE)*, 48–53. https://doi.org/10.1109/TAEE.2012.6235405

[2] *Basys 2—Digilent Reference*. (n.d.). Retrieved 10 August 2022, from https://digilent.com/reference/programmable-logic/basys-2/start

[3] Molina Cantero, A. J., Guerrero, D., Gómez, I., & Pérez, F. (2012, March). *EdC-T3-COMPUT-SIMPLE-2012_parte3_v1.pdf—Departamento de Tecnología Electrónica.* https://www.dte.us.es/docencia/etsii/gii-ti/edc/grupos/grupoAMC/material-adicional/tema-3-computador-simple/edc-t3-comput-simple-2012_parte3_v1.pdf/view

[4] Carry flag. (2021). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Carry_flag&oldid=1062800758

[5] Guerrero, D., & Gómez, I. (2013, May). *microoperaciones CS2010—Departamento de Tecnología Electrónica.* https://teclix.dte.us.es/docencia/etsii/gii-is/estructura-de-computadores/grupo-4-2018/mcs2010/view

[6] *SoftDevBigIdeas | Just Enough Design Up Front.* (n.d.). Retrieved 5 November 2022, from https://www.softdevbigideas.com/just-enough-design-up-front.html

[7] *Sueldo: Ingeniero Informático (Noviembre, 2022).* (n.d.). Retrieved 6 November 2022, from https://www.glassdoor.es/Sueldos/ingeniero-inform%C3%A1tico-sueldo-SRCH_KO0,21.htm

[8] *WebAssembly Tool Conventions.* (2022). WebAssembly. https://github.com/WebAssembly/tool-conventions/blob/79d4b069951edf50546ba8d2f2ecdef24301b7a1/BasicCABI.md (Original work published 2016)

[9] ISO Central Secretary. (1999). *Programming languages—C* (Standard ISO/IEC 9899:1999). International Organization for Standardization. https://www.iso.org/standard/29237.html

[10] ECMA International. (2011). *Standard ECMA-262—ECMAScript Language Specification* (5.1). http://www.ecma-international.org/publications/standards/Ecma-262.htm

[11] Rossberg, A. (2019). *WebAssembly Core Specification.* W3C. https://www.w3.org/TR/wasm-core-1/

[12] *SpiderMonkey—Firefox Source Docs documentation.* (n.d.). Retrieved 14 November 2022, from https://firefox-source-docs.mozilla.org/js/index.html

[13] V8 (intérprete de JavaScript). (2022). In *Wikipedia, la enciclopedia libre.* https://es.wikipedia.org/w/index.php?title=V8_(int%C3%A9rprete_de_JavaScript)&oldid=145406027

[14] WebAssembly Community Group. (2020). *WebAssembly System Interface.* https://github.com/WebAssembly/WASI/blob/d8b286c697364d8bc4daf1820b25a9159de364a3/phases/snapshot/docs.md

[15] Diz Gil, G. (2022). *Chashtable* [C]. https://github.com/GuilleX7/chashtable (Original work published 2020)

[16] Magic number (programming). (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Magic_number_(programming)&oldid=1116077509

[17] *Vue.js—The Progressive JavaScript Framework | Vue.js.* (n.d.). Retrieved 17 November 2022, from https://vuejs.org/

[18] *@guillex7/jasm.* (n.d.). Npm. Retrieved 27 October 2022, from https://www.npmjs.com/package/@guillex7/jasm

[19] *WebAssembly/wasi-sdk: WASI-enabled WebAssembly C/C++ toolchain.* (n.d.). Retrieved 27 October 2022, from https://github.com/WebAssembly/wasi-sdk

[20] Node.js. (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Node.js&oldid=1117561991

[21] Babel (transcompiler). (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Babel_(transcompiler)&oldid=1107446917

[22] Webpack. (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Webpack&oldid=1118395663

[23] Nginx. (2022). In *Wikipedia.* https://en.wikipedia.org/w/index.php?title=Nginx&oldid=1115025749

[24] *Main—Emscripten 3.1.25-git (dev) documentation.* (n.d.). Retrieved 4 November 2022, from https://emscripten.org/index.html

[25] *Introduction—The `wasm-bindgen` Guide*. (n.d.). Retrieved 4 November 2022, from https://rustwasm.github.io/docs/wasm-bindgen/introduction.html

[26] Register-transfer level. (2022). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Register-transfer_level&oldid=1088743222

Appendix A Register Transfer Language

In this appendix, we will define the abstract Register Transfer Notation (RTN) used throughout the document to formally describe how data is transferred between registers and memories in a digital synchronous circuit [26]. We will call this notation Register Transfer Language (RTL) henceforth.

A.1 Sequential circuits

The main symbols used in RTL are registers and memories, which fall into sequential circuits. We will name them following the same rule: their name must only be composed of a combination of letters, e.g. MDR, MAR or RAM are well-formed symbols. Although their writing and reading capabilities are similar, we will discuss their differences in the following sections.

Transferring data to either a register or memory is a persistent operation, i.e. the component must store the value permanently until another transfer occurs. When a data transfer occurs, there must always be a sequential circuit where data is written. However, the source of this data does not necessarily need to be a sequential circuit, as we can operate with immediate values or take values from other types of components that we will explain later on.

To represent the length of the word that registers and memories can store, we will add (preferably using subscript) a slash just after their name, followed by the bit-length expressed as a natural number.

A.1.1 Registers

Registers are one of the two types of sequential circuits we will discuss, and the simplest, as they can only contain one word of a fixed size. This fact follows that registers are non-addressable because they do not need to provide data addressing capabilities, i.e. selecting which word of the register will be written.

A.1.2 Memories

Memories share similarities with registers in the sense that they both store data. While a register can only store one word, memories can store many (usually a multiple of 2) words of the same size. Memories are addressable, as they must provide a way to select which word we want to read or write. We will usually use a number called the word address, which represents the position of the word inside the memory. To simplify this selection, we will consider that the address space of the memory is linear, i.e. the addresses start at a given offset (commonly 0) and grow up to the highest address possible, thus representing the address space as a row vector. Each entry in the row vector contains a single word, and the word address matches the entry's index.

To address a word in a memory, we will use the notation formed by the name of the memory, followed by a number between square brackets: RAM[23] points to the value of the RAM stored at address 23, starting from zero. We will call this way to access data direct addressing, as we provide the value of the address directly.

Another way to address data, named indirect addressing, consists of using the content of a register as the given address. In this case, we will use double square brackets around the name of the register: RAM[[MAR]], which means we point to the word stored at an address equal to the content of the register MAR.

Sometimes, registers can be combined into a single component_to increase the number of words stored without resorting to memories (e.g. a general register file). Even though this component is not formally a memory, in most cases, we will need to select the register we will be using, so it is likely to be addressable. In this case, the same convention applies.

A.2 Combinational circuits

In contrast with sequential circuits, combinational circuits cannot persist a value, and their output entirely depends on their inputs. ALUs and every other unit in the digital circuit that cannot be considered sequential fall under this category.

We cannot store data in these circuits, so they can never be the data transfer target. However, their outputs are still helpful as they can be a data source. As there are usually numerous inputs and outputs in a single combinational circuit, we will name them using the name of the circuit, followed by the name of the input/output in parentheses. This name must, in turn, be composed of letters and numbers (that may be in subscript): MAR(A), ALU(OP_1) or SR(C) are valid names. Outputs that admit the high impedance state shall start their name with an asterisk. The high impedance state shall be represented using the letters HI.

A.3 Immediate values

Immediate values represent the raw data that can be stored and transferred using RTL. Every data type must have a binary representation to be stored in a circuit. Unless explicitly stated, every data will be represented as an integer using the two's complement, even though other representations might be valid. Such is the case with real numbers that can be stored using standard floating-point representations.

The result of directly or indirectly addressing a memory, naming a register or naming a combinational output is an immediate value that can be used as the right-side operand in any operation.

A.4 Operations

Operations are events in which sequential, combinational circuits and immediate values take part. Every operation has an arity, i.e. the number of operands taken by the operation. Arity can go from one, in which case we say the operator is unary, to two (binary), three (ternary) or more (n-ary).

Every operation in RTL is binary, i.e. will have two operands, named left-side and right-side, by order of appearance. We will describe the possible operations and their outcome in Table 8-1.

Operation	Syntax	Outcome
Direct memory write	MEM[Addr] ← Im	Stores the immediate value Im at the address pointed by the immediate value Addr.
Indirect memory write	MEM[[REG]] ← Im	Stores the immediate value Im at the address pointed by the content of the register REG.
Combination al output	COM := Im	Emits the immediate value Im through the output COM.

Table 8-1. Description of RTL operations