Developer's Guide

MicroCISC

Introduction **2**
Purpose 2
Scope 2
Feature list 2
Architecture **3**
Hardware Overview 3
Software Overview 4
Application modules overview 4
UI 4
Assembler 5
CPU 6
MainMemory 7
Requirements 8
UI 8
Assembler 8
CPU 8
MainMemory 8
Development Setup **9**
Visual Studio **9**
DrawIO or PlantUML **9**
Github Actions `act` tool **9**
Testing **10**
Local testing 10
Automatic testing 11

Introduction

Purpose

The purpose of this document is to offer the necessary information needed to understand, develop and maintain the MicroCISC project.

Scope

MicroCISC is an application that comes bundled with the “Computer Architecture” course from the “Calculatoare și Inginerie Electrică” department of University of Lucian Blaga of Sibiu (ULBS), meant to facilitate an easier understanding of how a basic Complex Instruction Set Computer (CISC) is structured and functions.

Feature list

The applications offers the following features:

- integrated text editor for writing, saving and loading assembly code
- intuitive processor diagram for easy visualization
- hex viewer for main memory content
- dynamic micro-program memory viewer
- debugging facility at instruction, micro-instruction and micro-command levels
- windows layout manager
- editable interrupts that can be triggered by the user

Architecture

Hardware Overview

The didactic processor (fig. 1) represent a simple CISC machine that features an Arithmetic Logic Unit (ALU), 16 bit general purpose registers (R0 through R15), an interrupt controller (which uses the register IVT - Interrupt Vector Register) with support up to four internal interrupts (called also exceptions) and four external interrupts, a stack pointer, a programmable control unit (BGC block - Bloc Generator de Comenzi) and three buses used for transferring data across the processor (Source Bus, Destination Bus and Result Bus). Together with the processor there is a 64KB of main memory (or simply RAM) available for computer interrupts and user programs. For a more detailed description please refer to the ISA and Reference Manual that are available together with this document in the project repository.

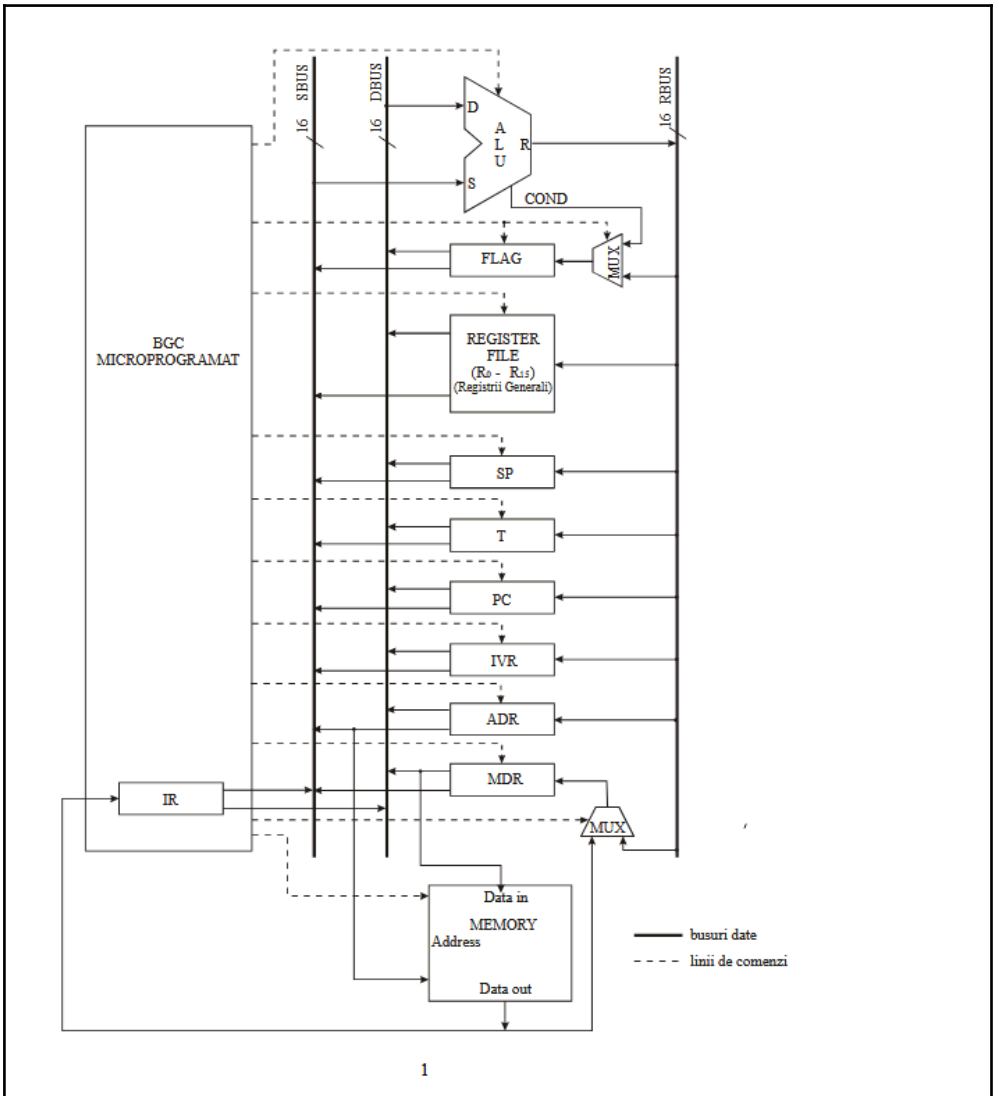

Fig. 1 Didactic processor

Software Overview

Application modules overview

In order to model the real hardware present prior, the application has been structured into modules presented down below (fig. 2):

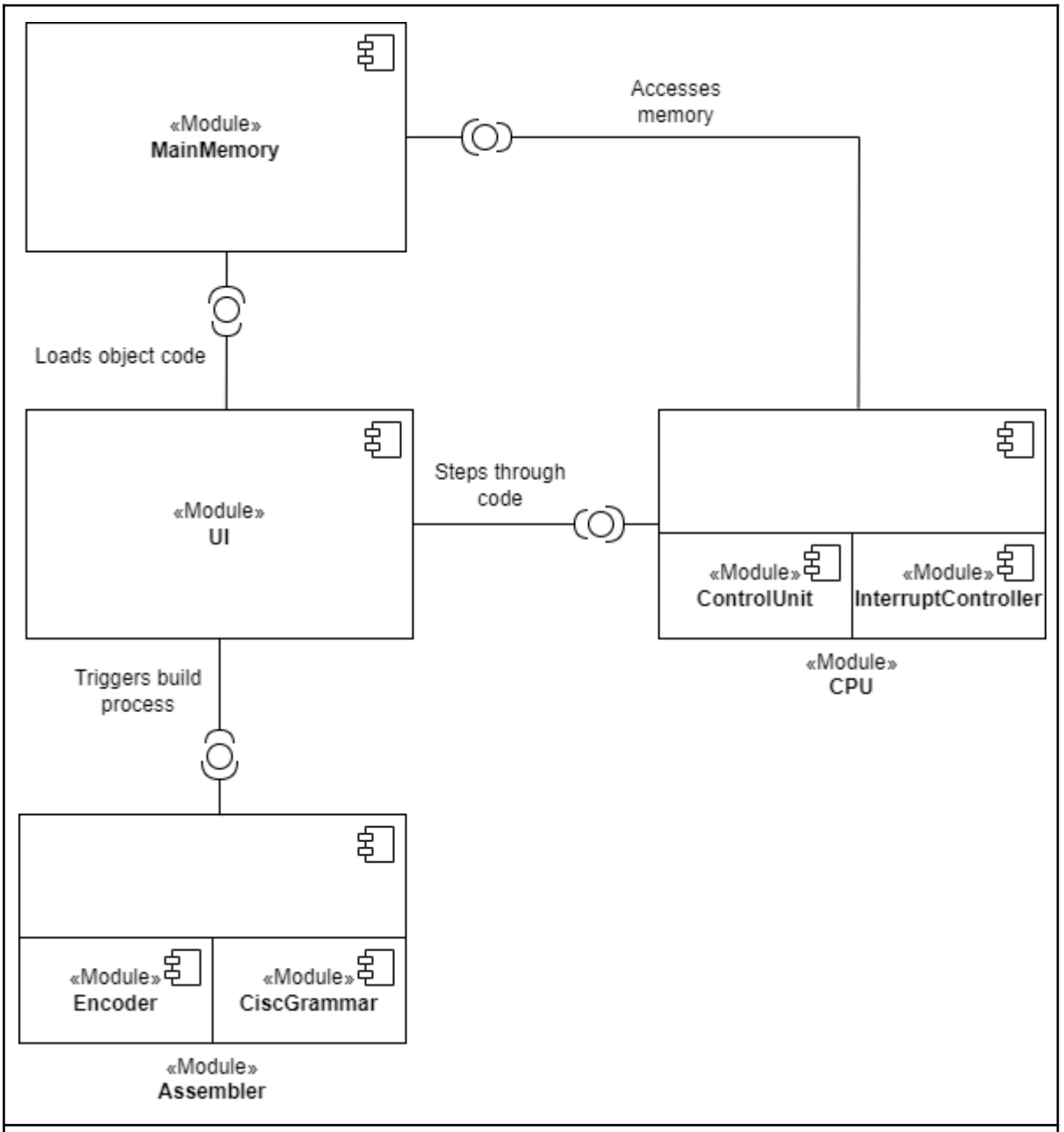

Fig. 2 Application modules

UI

- responsible for managing the visual and functional aspects of the user interface such as button action and graphical presentation
- structured based on Model-View-Viewmodel architecture

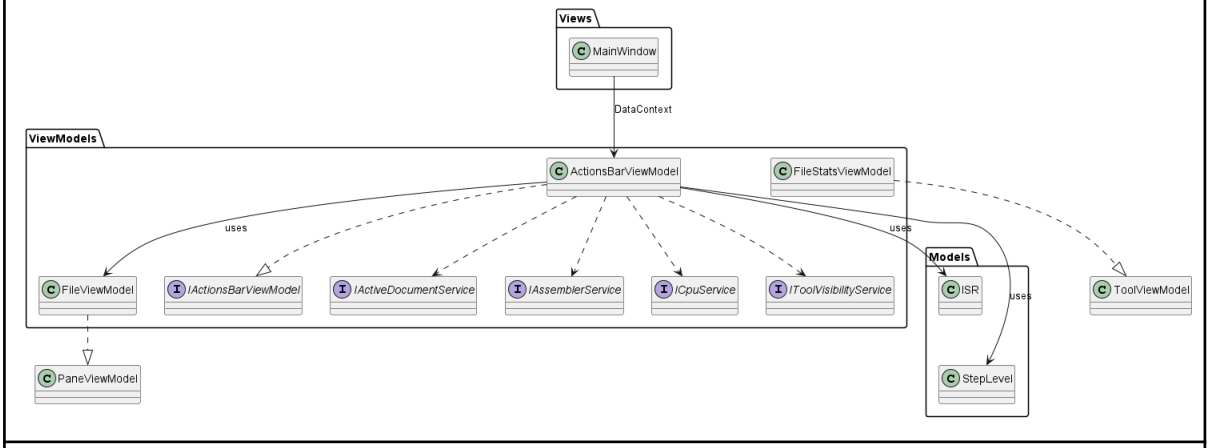

Fig.3 UI Model-View-Viewmodel architecture

Assembler

- responsible for taking the user code (files with the extension '.s`) and generating the appropriate machine code
- uses 'Encoder' and 'CiscGrammar' submodules in order to easily parse the given assembly code (for more information please refer to 'Irony' C# library)
- has integrated syntactic error detection

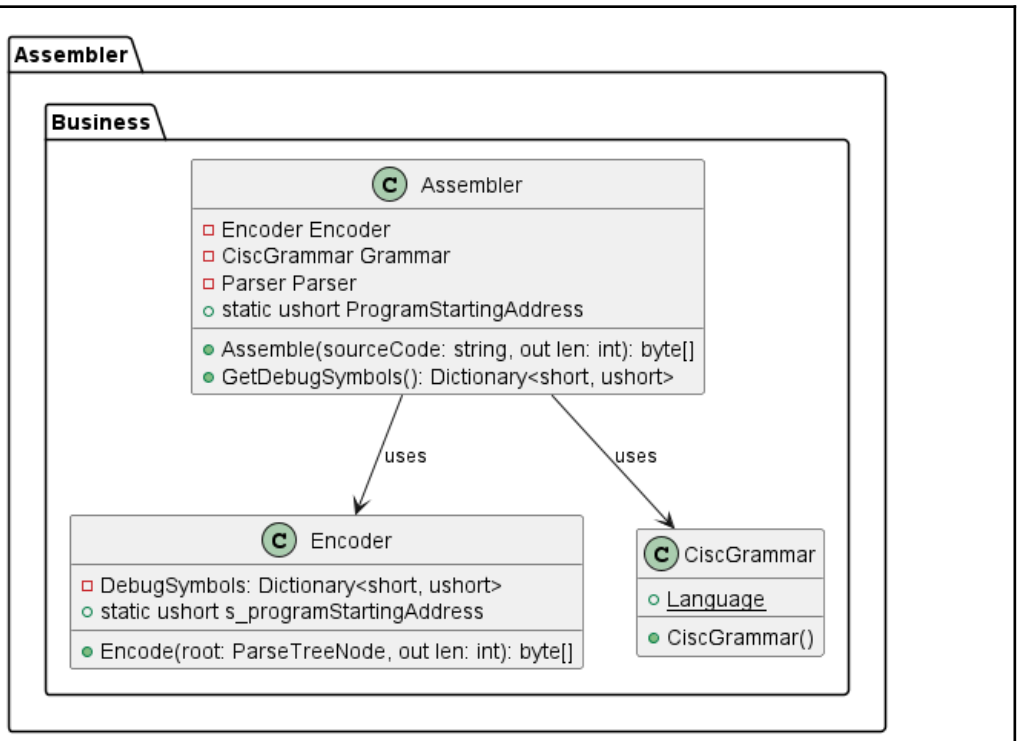

Fig. 4 Assembler business logic structure

CPU

- responsible for simulating the generated object code
- can step through user code at instruction, micro-instruction or micro-command level
- is subdivided into ‘ControlUnit’ and ‘InterruptController’ submodules, responsible for controlling the execution flow of the code and interrupt detection respectively

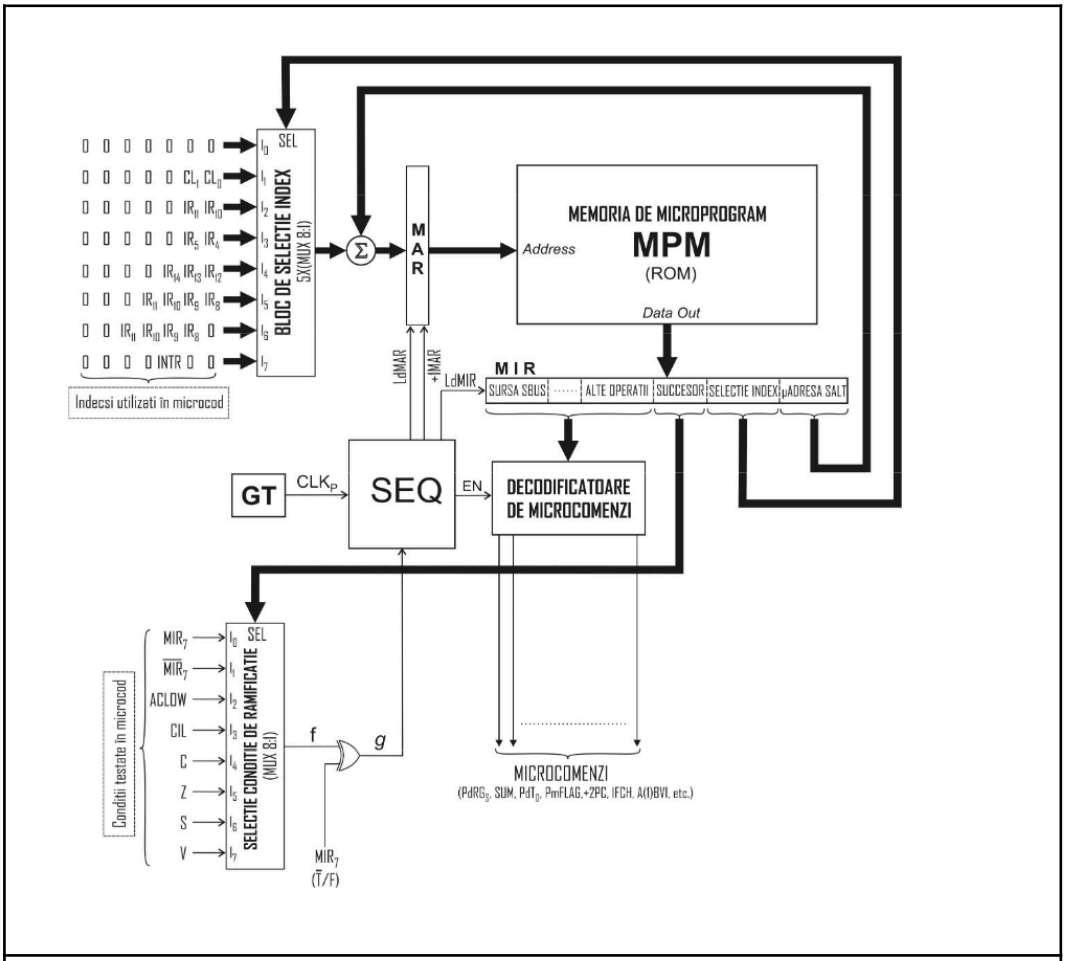

Fig. 5 Internal ControlUnit circuitry

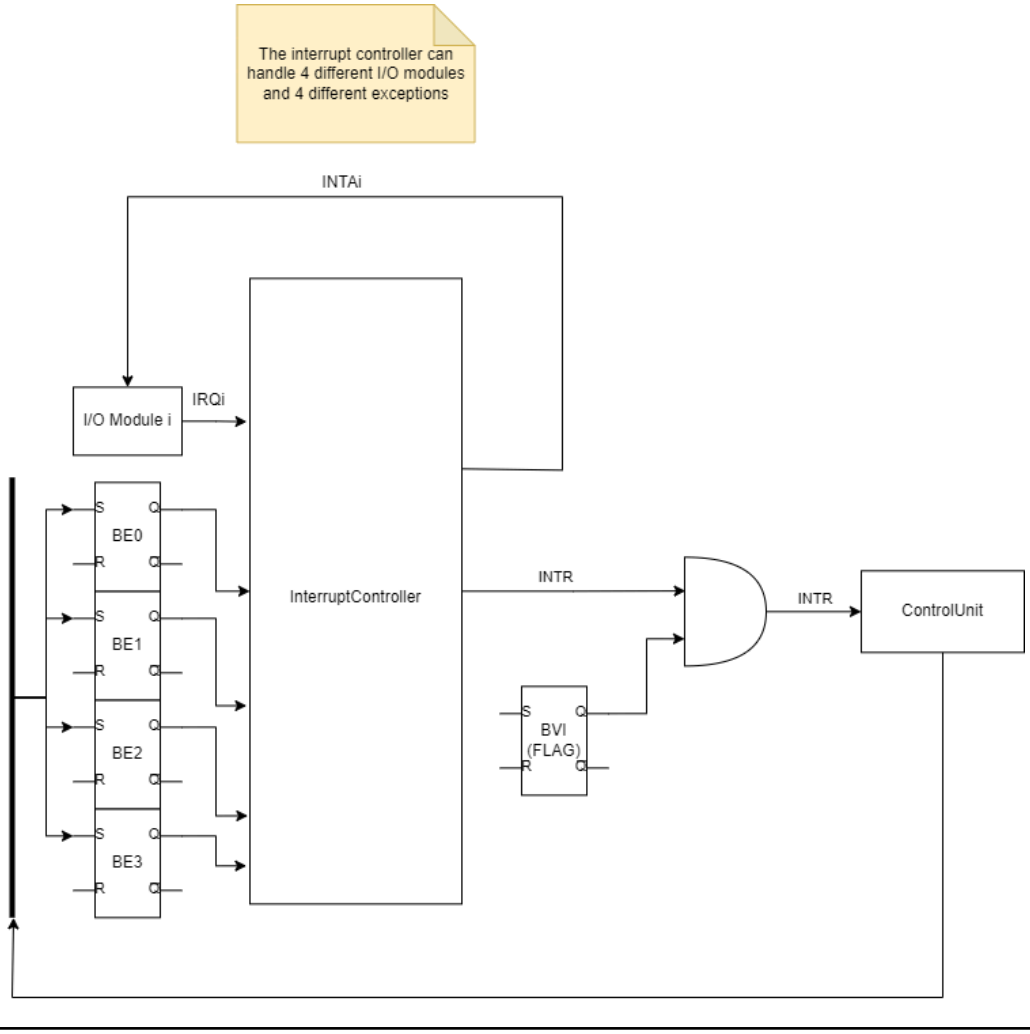

Fig.6 InterruptController internal circuitry

MainMemory

- responsible for simulating the behavior of a RAM device
- is segmented into the following areas: Interrupt Vector Table (IVT), User code, Interrupt Handles and Stack

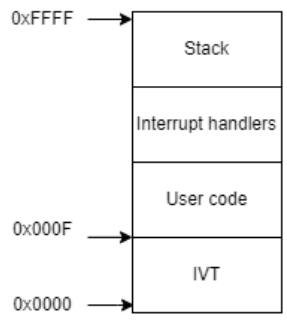

Fig. 7 MainMemory segments

Requirements

UI

1. The user shall be able to see the computer architecture diagram.
2. The code execution shall be reflected in the computer diagram via register and bus values.
3. The user shall be able to see the memory content of the computer into the following numerical representations: binary, decimal and hexadecimal.
4. The user shall have a `Help` section for architecture and development guides at his disposal.
5. The user shall be able to edit, save and load assembly code.
6. The user shall be able to trigger interrupts and exceptions during code debugging.

Assembler

1. The user shall be able to load the assemble code into `.s` files.
2. The assembler shall take as input the user generated `.s` file and generate the appropriate object code.
3. The assembler shall be able to detect any potential errors in code syntax.
4. The user shall be able to insert comment lines into the assembly code.
5. the user shall be able to insert labels into the assembly code.

CPU

1. The user shall be able to step through the code at instruction, micro-instruction or micro-command level.
2. The user shall be able to define and edit custom interrupts from the four available slots.
3. The user shall be able to update the content of the micro-program memory (MPM).

MainMemory

1. The user shall be able to set the dimension of the stack.
2. The user shall be able to edit the contents of the memory.

Development Setup

In order for the user to be able to develop or maintain the MicroCISC project he shall need the following tools:

Visual Studio

Microsoft's official Integrated Development Environment that comes with the C# compiler, libraries and all of the necessary files for running Windows Presentation Foundation (WPF - the framework used by the UI component).
Link: https://visualstudio.microsoft.com/downloads/

DrawIO or PlantUML

Tools for modelling the behaviour of the application through UML diagrams (which were also used for this document). The main difference between the tools is that the former is GUI based tool that allows manual edition of the diagram, while the latter is script based meaning it is easier to edit later, detect changes using a version control software (e.g. Git) and offers a CLI for the user. It should be noted that both of the tools have integration with Visual Studio Code text editor.
DrawIO link: https://www.drawio.com/
PlantUML link: https://plantuml.com/download

Github Actions `act` tool

In order to be able to test the CI/CD jobs one could use `act` tool for running them on the local machine instead of waiting for Github's servers. Besides this CLI tool the user shall also need to install Docker for containerization.
act: https://github.com/nektos/act
Docker:https://www.docker.com/products/docker-desktop/

Testing

In order to ensure good software quality there must be a testing strategy defined. In the case of this project the testing strategy is to define and run unit tests on local machines and on a CI/CD job.

Local testing

At the moment of writing this document the application has unit tests for MainMemory and CPU modules. The latter ones are of interest due to the fact that every instruction class is tested by taking multiple combinations of addressing modes and flag values. Also they generate some artifacts called ‘snapshots’ which can be used to trace the execution of the code. Below an example shall be seen:

```
Trace log for test: Mov_AI_AI_Test
mov [r1], [r0]

Expected Path: IFCH_0 IFCH_1 B1_0 FOS_AI_0 FOSEND_0 FOD_AI_B1_0 MOV_0 WRD_2
Real Path:     IFCH_0 IFCH_1 B1_0 FOS_AI_0 FOSEND_0 FOD_AI_B1_0 MOV_0 WRD_2

IFCH_0
PdPCs NONE SBUS PmADR IFCH +2PC IF ACLOW JUMPI INDEX0 T PWFAIL

  R0: 0x0 -> 0x12
  PC: 0x10 -> 0x12
  ADR: 0x0 -> 0x10
  IR: 0x0 -> 0x821
  SBUS: 0x0 -> 0x10
  RBUS: 0x0 -> 0x10

IFCH_1
NONE NONE NONE NONE NONE NONE IF CIL JUMPI INDEX1 F B1

  SBUS: 0x10 -> 0x0

B1_0
NONE NONE NONE NONE NONE NONE JUMPI INDEX2 T FOS_AM
```

Fig. 8 Snapshot of B1 instruction test

Automatic testing

Due to human nature it is expected that some errors may slip through, which may give rise to the possibility of having an unstable `main` branch. In order to avoid this issue, a CI/CD job has been added for PR events that fetches and build the application together with the units test, thus ensuring that no critical error are merged.

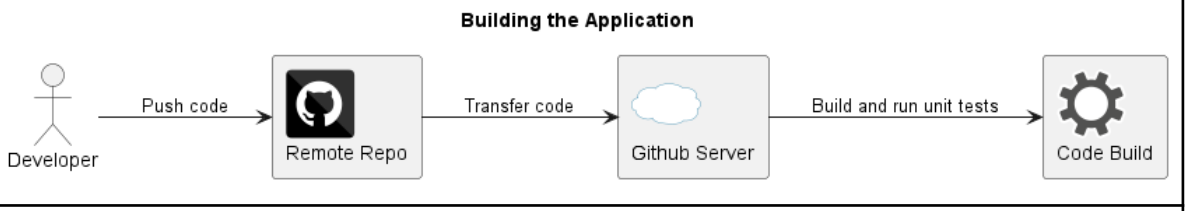

Fig. 9 Automatic testing