

Mano CPU Emulator Generator

Software Engineering B.Sc. Final Project Write-up

Yuval Tzur | Supervisor: Dr. Yigal Hoffner

12/10/2013

The system is a CPU emulator generator, capable of generating different CPU emulators based on the definition of the assembly language instructions at the micro-code level and the definition of the relevant instruction formats. The system can be used in different ways by students who wish to understand the inner workings of a CPU and learn to program it: both at the assembly language level – by programming the CPU in assembly language and executing the assembled code, and the micro-code level – by defining the assembly level instructions and their format. The generated CPU emulator allows the programmer to see the execution of a program at the level of the assembly instructions as viewed through the programmer interface. In addition, the emulator enables the programmer to see the instructions being executed at the micro-code level, thereby exposing the components that are hidden from the assembly language programmer. In addition to the above, the system also generates the specific Assembler that translates an assembly language program, written in the language defined by the user, to the pseudo-binary code that can then be executed by the generated CPU emulator. The resulting system provides a powerful tool for teaching computer science, software and electrical engineering students.

Table of Contents

1	Introduction	9
1.1	Project Goal	9
1.2	Problem	9
1.3	Solution	9
1.4	Incentive	9
1.4.1	Speed and Scale	9
1.4.2	Data Representation	9
1.4.3	Physical Limitations	10
1.5	Description	10
1.6	Future Development.....	10
1.7	Audience	10
1.8	System Usage Stages	11
1.8.1	Emulator Generation	11
1.8.2	Program Assembling	11
1.8.3	Program Execution	11
1.9	System Users and Roles	11
1.9.1	User Types	11
1.9.2	User Roles	12
1.10	Terminology	14
1.10.1	Emulator	14
1.10.2	Component.....	14
1.10.3	ALU.....	14
1.10.4	Bus	14
1.10.5	Register	14
1.10.6	Flag.....	14
1.10.7	Instruction Timer	14
1.10.8	CPU Architecture	14
1.10.9	M. Morris Mano Architecture	14
1.10.10	CPU & Memory	14
1.10.11	System State	14
1.10.12	Assembly Instruction Set	15

1.10.13	Assembly Instruction Format	15
1.10.14	Micro-Operation	15
1.10.15	Assembler.....	15
1.10.16	Instruction Set Generator (Compiler)	15
1.10.17	Assembly Program Editor	15
1.10.18	Instruction Set Template File	15
1.10.19	Instruction Set Template Editor.....	15
1.10.20	User Control Panel.....	15
1.10.21	Pseudo-Binary.....	15
1.10.22	Program Execution	15
1.10.23	Executable File	15
1.11	Cross-Platform Portability	16
2	Literature	17
2.1	Existing Emulators	17
2.1.1	Basic Computer Simulator – Laurens Rodriguez.....	17
2.1.2	Computer Simulator – Dr. Nicholas Duchon	17
2.2	Field-Programmable Gate Array (FPGA)	17
2.2.1	Cost	17
2.2.2	Setup	17
2.2.3	Availability.....	17
3	Architecture	18
3.1	Abstraction.....	18
3.1.1	Hardware Abstraction	18
3.1.2	Data Abstraction	18
3.2	Specifically Designed Syntax	18
3.3	Modularity	19
4	Implementation	20
4.1	System Structure.....	20
4.1.1	Emulator	20
4.1.2	Assembler.....	28
4.1.3	Parser	30
4.1.4	GUI	34
4.2	File System Structure	35

4.2.1	AppData	35
4.2.2	Resources	35
4.2.3	Programs	35
4.2.4	Templates	35
4.2.5	Machines	35
4.3	System Design	36
4.3.1	DFD 0	36
4.3.2	DFD 1	37
4.3.3	Emulator	38
4.3.4	Assembler	47
4.3.5	Parser	47
4.3.6	GUI	47
4.4	Compilers	48
4.4.1	Instruction Set Compiler	48
4.4.2	Assembler	53
5	Development Environment	55
5.1	Programming Paradigm	55
5.2	Programming Language	55
5.2.1	Platform Portability	55
5.2.2	Dynamic Class Loading	55
5.3	System Limitations	55
5.3.1	Architectural Restrictions	55
5.3.2	User Interface	56
5.4	Tools	56
5.4.1	IDE	56
5.4.2	Diagrams	56
5.4.3	Compiler Generators	56
5.5	Testing	56
6	User Guide	57
6.1	Error Handling	57
6.1.1	Syntax Errors	57
6.1.2	System Crash	57
6.1.3	Infinite Loops	57

6.1.4	Instruction Set-Assembler Mismatch	57
6.1.5	Missing Instruction Set	57
6.2	Workflow Continuity.....	57
6.3	Environment Requirements	58
6.4	Mano Basic Architecture	58
6.4.1	Architecture Components.....	58
6.4.2	Architecture Structure	59
6.4.3	Basic Syntax and Commands	60
6.5	Instruction Set Template Guide.....	62
6.5.1	Format Syntax.....	62
6.5.2	Code Syntax	64
6.6	User Control Panel.....	69
6.6.1	Memory Panel.....	69
6.6.2	Variables Panel.....	69
6.6.3	System State Panel	69
6.6.4	Command Panel	70
6.6.5	Run Button	70
6.6.6	Step Button	70
6.6.7	uCode Checkbox.....	70
6.6.8	Reset Button	70
6.6.9	Exit Button.....	70
6.6.10	Load Program Button	70
6.6.11	Assemble Button	70
6.6.12	Default Template Button	70
6.6.13	Current Instruction Set Panel	70
6.6.14	New Template Button	70
6.6.15	Compile Button	70
6.6.16	Instruction Set List	70
6.6.17	Reload Template Button.....	71
6.7	Program Editor.....	71
6.7.1	Program Panel	71
6.7.2	Use Button	71
6.7.3	Load Button	71

6.7.4	Save Button	72
6.7.5	Cancel Button	72
6.8	Instruction Set Template Editor.....	72
6.8.1	Template Name.....	72
6.8.2	Template Panel	72
6.8.3	Use Button	72
6.8.4	Load Button	73
6.8.5	Save Button	73
6.8.6	Cancel Button	73
6.9	Timeout Dialog.....	73
6.9.1	Stop Button	73
6.9.2	Cancel Button	73
7	Summary.....	74
7.1	Main Focal Points	74
7.1.1	CPU Architecture and Functionality	74
7.1.2	Software-Hardware Modeling	74
7.1.3	Scalability	74
7.2	Conclusions	74
7.2.1	Good Planning is Key for Success	74
7.2.2	Good Tools can Make the Difference	74
7.3	Future Work	75
7.3.1	Multithreaded Implementation.....	75
7.3.2	Editable Architecture	75
7.3.3	Move to the Cloud.....	75
7.3.4	Instruction Set Generation Tool.....	75
8	References	76
8.1	Research	76
8.2	Technical References	76
9	Appendix A.....	77
9.1	Template for the Basic Mano Instruction Set	77
10	Appendix B	80
10.1	Useful Links	80
10.1.1	Java CUP.....	80

10.1.2 IntelliJ IDEA.....	80
10.1.3 Visio.....	80
10.1.4 Sublime Text	80
10.1.5 WinMerge	80

Table of Figures

Figure 1-1: The different types of users and the parts of the system that they use	12
Figure 1-2: System work sequence	13
Figure 1-3: The different types of users and the processes which they can control	13
Figure 4-1: DFD 0	36
Figure 4-2: DFD 1	37
Figure 4-3: Emulator package	38
Figure 4-4: Global package	39
Figure 4-5: Components package.....	43
Figure 4-6: Assembler package	47
Figure 4-7: Parser package	47
Figure 4-8: GUI package.....	47
Figure 6-1: The M. Morris Mano CPU architecture	59
Figure 6-2: User control panel	69
Figure 6-3: Program editor	71
Figure 6-4: Instruction set template editor.....	72
Figure 6-5: Timeout dialog alert.....	73

1 Introduction

1.1 Project Goal

The project's goal is to provide a tool for software and electrical engineering students. The system will allow students to learn and understand the inner workings of a simple CPU and the assembly language it implements:

- **Novice students** can use the tool to learn the assembly language while observing the changes in [memory and main registers](#).
- **Intermediate students** can use the tool to learn the [micro-code](#) and binary representations used by the CPU that implement the assembly commands.
- **Advanced students** will have the ability to add, remove and rearrange micro-operations to add new assembly commands or edit/remove existing ones, as well as redefining the [instruction format](#) that may be necessary to accommodate the changes.

1.2 Problem

While currently there are tools that students can use to learn the assembly and micro-code of a CPU, none of them offers an editable [instruction set](#). All the lectures and home assignments which require changes to the instruction set are done on paper, and every change to the instruction set can only be checked manually following each and every micro-operation, noting all the changes it generated in the CPU.

1.3 Solution

In order to facilitate both teaching and learning the effects a change of the instruction set will have on the CPU, users can generate a customized instruction set.

1.4 Incentive

An obvious assumption would be that learning about the functionality of a CPU using a real, functioning, CPU will have better results, but such an assumption is wrong for several reasons:

1.4.1 Speed and Scale

Hardware components are designed for speed, and to achieve that, they are designed as small possible. The high speed performance would prevent real time observation, as the slowest general purpose CPU performs at a rate of 100k operations each second. A modern CPU's inner circuits cannot be seen with an optical microscope.

1.4.2 Data Representation

Hardware processors do not represent the data as simple 'ones and zeroes' as shown on abstract models. The real representation of the data is defined as different ranges of voltage that vary from one processor to the other. Furthermore, the same values can be represented using different voltage ranges within the same chip. To be used for observation purposes, the values need to be measured and translated into a readable state.

1.4.3 Physical Limitations

As opposed to software, hardware has a physical dimension. The CPU's functionality is defined by physical circuits. Changing a processor's functionality is very hard, and cannot be done as simply as changing the functionality of a software program.

1.5 Description

The system is a software based generator that is capable of generating CPU emulators, based on the definition of the assembly instructions provided by the user. The resulting CPU emulators provide the functionality of a basic central processing unit of a digital computer.

The use of the system can be viewed from different points of view according to the expertise of the user:

- Initially, a specifically generated CPU emulator provides a working platform for basic programming, using a low-level program language (Assembly language). After writing a program, the student can use the system to run it and watch for the program's result.
- After a student has reached a higher level as a programmer, she can use the same platform in order to learn how a CPU operates when following the instructions of the program. To do so, the system can run on a step-by-step mode, executing one assembly command at a time, allowing the student to observe the changes made to the [data and other values](#) by the command. A more advanced mode can show not only each assembly command, but also the flow of data between the different CPU [components](#) while evaluating the outcome of that command.
- Finally, an advanced student can delete, edit or create assembly instructions by defining the data flow steps that take place within the CPU whenever a command is processed. After redefining the commands, a new assembly language is created, consisting of the new [set of instructions](#) the student defined. The new language can now be used to create new programs with the customized commands.

1.6 Future Development

The emulator, as currently implemented, allows for the instruction set to be edited, but it is still limited by its [basic architecture](#). While not supported yet, the emulator was designed in a way that allows the ability to configure the [architecture](#) to be added in the future. A closer resemblance to real hardware can also be accomplished by allowing some of the functionality to run in parallel, like real electrical circuits do.

In addition, the [GUI](#) is not final. While allowing the user to use most of the system, some features were not implemented (I/O devices, for example). A web-based GUI would allow the users to use the emulator without downloading it, removing the need for a Java SDK installation and improving platform portability.

1.7 Audience

The system is designed for academic purposes, to ease the learning process of the CPU's behavior. The system can be used by the professor while lecturing or while grading home assignments, or by the students when practicing the course material.

1.8 System Usage Stages

The system has three usage stages that allow the user to generate, and then use, a CPU [emulator](#) (Figure 1-1, Figure 1-2).

1.8.1 Emulator Generation

This is the first stage. In this stage, the [instruction set](#) is defined by creating an [instruction set template file](#) that defines the instruction format and the assembly micro-code. The template is used to generate a new emulator and [assembler](#).

1.8.2 Program Assembling

The second stage uses the previously generated assembler to convert programs written by the user, using the newly defined assembly language, into [pseudo-binary code](#) that can be [executed](#) by the emulator.

1.8.3 Program Execution

In the last stage, a pseudo-binary code can be loaded into the emulator's memory. The code will be executed by the emulator. Using the emulator's various interfaces, the user can follow the program's execution.

1.9 System Users and Roles

The system has four different types of users and two different roles. There is not any formal limitation on what type of user can perform each of the roles, but the skill level of the user might prevent them from assuming the advanced role of [instruction set editor](#).

1.9.1 User Types

The user types differ mostly by their knowledge and experience levels. As a user's skill level increases he will be able to use the more advanced capabilities of the system (Figure 1-1).

1.9.1.1 Novice

A novice student is a student not familiar with assembly programming. This type of student is expected to write simple programs and [execute](#) them. To better understand the assembly language, this type of student can use the step-by-step command execution, allowing them to understand the effect each command has on the [system](#).

1.9.1.2 Intermediate

An intermediate student has a higher skill level and should be comfortable writing complex programs. To further understand the effect each assembly command has on the CPU, this type of student can use the step-by-step [micro-operation](#) execution, which splits each assembly command to its micro-operations and performs them separately.

1.9.1.3 Advanced

The advanced student's skill level goes beyond the programming skills. This type of student understands the processor's micro-operations and the way they affect each [component](#), and the system as a whole at the same time. An advanced student has the amount of skill and knowledge to assume the role of [instruction set editor](#).

1.9.2 User Roles

The user roles are defined by the usage of different parts in the system and the way they are being used. A user may change roles while using the system (Figure 1-3).

1.9.2.1 Professor

The professor can use the system when teaching by using it in front of the class to showcase an example or while assessing or grading a student's performance by reloading a program or an [instruction set template](#) made by the student.

1.9.2.2 Programmer

A programmer is a user that uses the system to write assembly programs and executes them. A programmer's skill level can vary, but even expert programmers are bound by the limit of the assembly language.

1.9.2.3 Instruction Set Editor

An instruction set editor is an expert programmer that reached a skill level high enough to fully understand the assembly language and the micro-operations implementing it. Using their expertise, instruction set editors can expand or change the boundaries of the assembly language by redefining it.

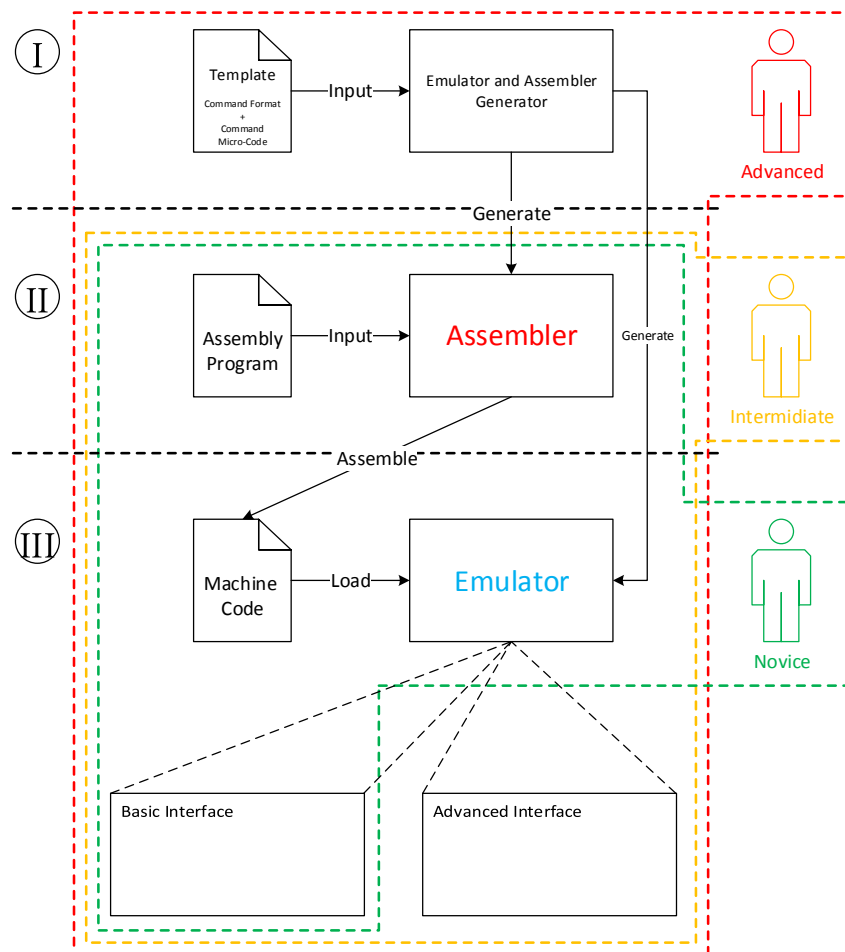


Figure 1-1: The different types of users and the parts of the system that they use

October 12, 2013

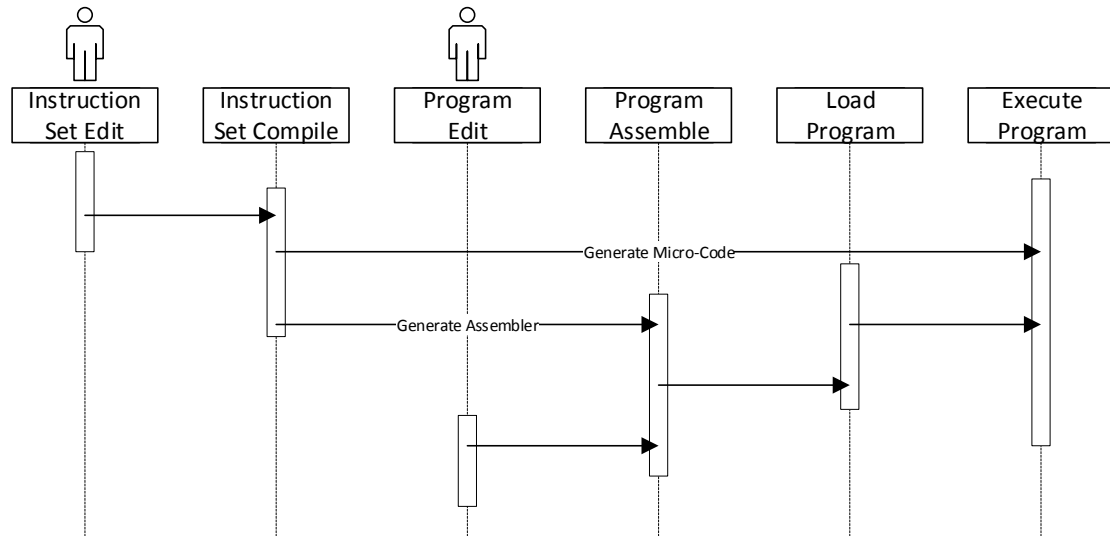


Figure 1-2: System work sequence

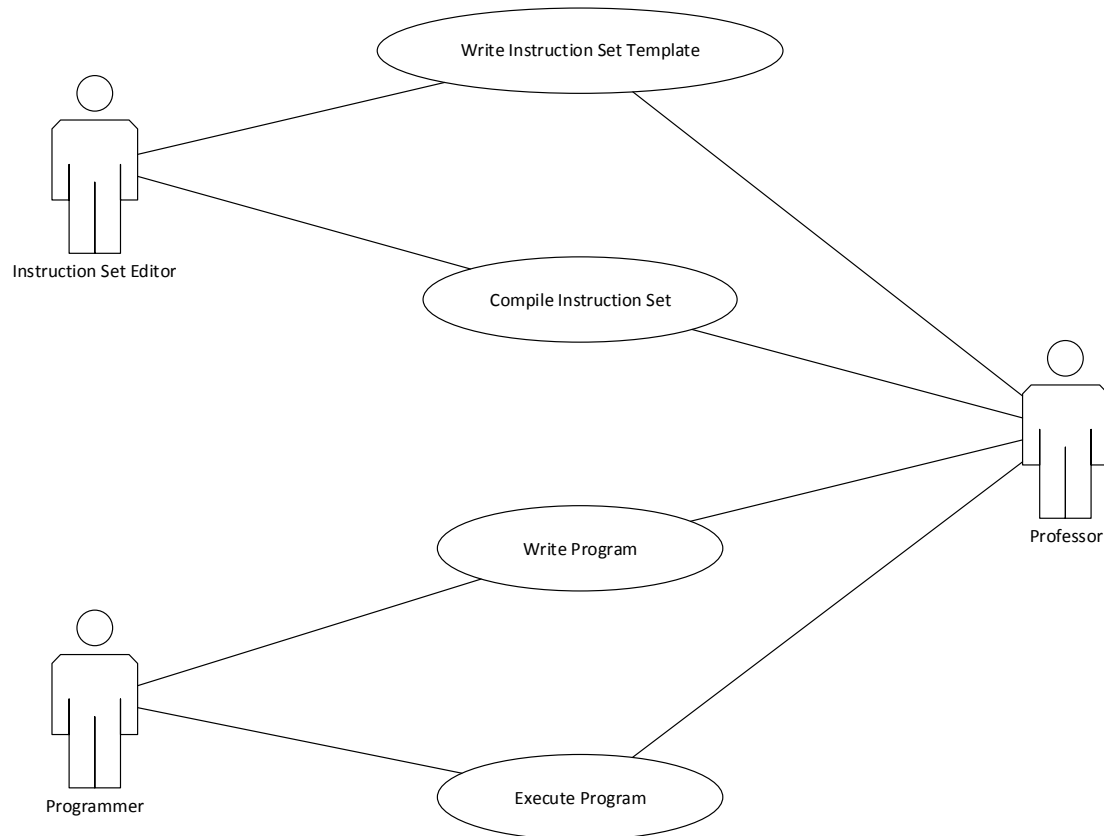


Figure 1-3: The different types of users and the processes which they can control

1.10 Terminology

1.10.1 Emulator

The emulator is the system as a whole. The emulator contains an emulated [CPU and memory](#), an [assembler](#), an [instruction set generator \(Compiler\)](#), a [program editor](#), an [instruction set template editor](#) and a [user control panel](#).

1.10.2 Component

In the scope of this project, a component is a software class that emulates the structure and behavior of a hardware component. The components emulated are: ALU, Bus, Register, Flag, Instruction Timer and Memory.

1.10.3 ALU

An ALU is an arithmetic and logic unit. The ALU is the main [component](#) in the CPU, whose function is to perform arithmetical or logical operations on a given input.

1.10.4 Bus

A bus emulates a data channel, used to transfer data from one [component](#) to another.

1.10.5 Register

A register is a collection of bits that store data.

1.10.6 Flag

A flag is a 1-bit [component](#), usually used to note the occurrence of an event or a state of the system.

1.10.7 Instruction Timer

The instruction timer's function is to count the cycles of an assembly instruction, performing the relevant [micro-operations](#) in each cycle.

1.10.8 CPU Architecture

CPU architecture is a set of [components](#) and the relation between those components that defines the structure of the CPU.

1.10.9 M. Morris Mano Architecture

The M. Morris Mano architecture (Figure 6-1) is a basic [CPU architecture](#) designed by Mano in his book 'Computer System Architecture'. This architecture is very basic and was designed for pedagogical purposes.

1.10.10 CPU & Memory

Both the CPU and the memory are software models of the respective hardware. In the scope of this project, the memory is treated as if it was a component within the CPU. The memory stores all the instructions and commands as [pseudo-binary](#) code. The CPU emulates the [execution](#) of the program on the [M. Morris Mano architecture](#).

1.10.11 System State

The system state is the overall value of each [component](#) at a given time during [program execution](#).

1.10.12 Assembly Instruction Set

An instruction set is a set of assembly instructions that are known to the CPU. The instruction set defines any assembly instruction as a group of [micro-operations](#) that implement the given instruction.

1.10.13 Assembly Instruction Format

The instruction format is the description of what the bits represent and how they are arranged within a command.

1.10.14 Micro-Operation

A micro operation is the basic operation performed by the CPU, usually consisting of one operation performed on a [component](#), or transferring data between components.

1.10.15 Assembler

The assembler is a basic compiler that translates assembly code into a [pseudo-binary executable file](#). The assembly code is provided to the [emulator](#) using the [program editor](#).

1.10.16 Instruction Set Generator (Compiler)

The instruction set generator is a compiler that translates a specially designed [template file](#) into a working [instruction set](#). A new [assembler](#), updated according to added or removed instructions, is generated as well.

1.10.17 Assembly Program Editor

The program editor is used by the user to write or edit assembly programs to be [executed](#).

1.10.18 Instruction Set Template File

An instruction set template file is a text file that defines assembly instructions and the way they translate into [micro-operations](#), as well the way they should be translated by the [assembler](#). A template file should follow a specifically designed syntax.

1.10.19 Instruction Set Template Editor

The instruction set template editor is used by the user to write or edit [template files](#).

1.10.20 User Control Panel

The user control panel allows the user to monitor the [system state](#) during [program execution](#) and load new programs or [template files](#).

1.10.21 Pseudo-Binary

A Pseudo-binary representation is a representation of content in a binary form, using a predetermined format instead of real bits. The representation of bits is done using 'true' and 'false' values in software, or the '1' and '0' characters in text.

1.10.22 Program Execution

The program execution is the process of moving from one [system state](#) to another according to the program given, in order to reach the program's result.

1.10.23 Executable File

An executable file works like a real executable, using [pseudo-binary](#) content. The CPU uses this file the same way a real CPU uses a binary executable.

1.11 Cross-Platform Portability

Designed for the use of students, the system cannot be bound to a specific platform. To improve portability, the system was designed and implemented using the Java programming language, which is known for its cross-platform capabilities. Although Java was used, some features use functionality given by the OS when high level functionality, such as code compiling, is needed. The use of these system calls was reduced to a minimum and the calls themselves were made as general as possible, but the system's performance in non-Microsoft Windows environments has yet to be proven.

Another possible solution for this problem would be to use cloud computing, running the system on a supporting server and providing a web-based user interface for the users.

2 Literature

There are several existing tools that can be used to teach the mechanics of a CPU, but none of them have the full usability of the emulator created in this project.

2.1 Existing Emulators

These are two examples of existing emulators. Both examples, as well as other emulators that were tested, have no capability to customize the instruction set. Most simulators and emulators contain a bug inherited from the book's implementation of the [SPA](#) command. By checking the MSb (sign bit) only, the command treats zero as a positive number.

2.1.1 [Basic Computer Simulator – Laurens Rodriguez](#)

The Basic Computer Simulator is a basic, web-based, implementation of the Mano CPU architecture. It provides a graphic user interface for the [system state](#), a small editor and basic specifications for the registers and assembly language. The GUI shows only 16 memory slots at any given time, requiring the user to move the memory table's starting point several times when executing a program with more than 16 lines of code. The Basic Computer Simulator's step-by-step option works on [micro-operations](#) and does not have the ability to [execute](#) a full assembly command on each step.

2.1.2 [Computer Simulator – Dr. Nicholas Duchon](#)

This simulator provides a platform that allows execution of assembly programs according to Mano's assembly instruction set. The GUI is somewhat uncomfortable and not intuitive. When given a label to an address, the program editor will translate it to a hexadecimal value if possible, which is not compatible with the assembly language described in the book, and is not mentioned in the tool's documentation. The simulator does not ignore whitespaces (not mentioned in the documentation either) and the workflow is very slow.

2.2 Field-Programmable Gate Array (FPGA)

An FPGA is a programmable hardware component that can be used to emulate integrated circuits. The use of FPGAs has several disadvantages:

2.2.1 Cost

While software can be distributed with no additional costs, FPGAs are physical hardware components that need to be purchased.

2.2.2 Setup

The use of an FPGA requires the full circuit implementation of all the components on each FPGA. To monitor the system's state, each FPGA needs a connection to a computer or any other external output device. All values in the FPGA are represented by voltage differences and need to be translated to a bitwise representation.

2.2.3 Availability

Because of the cost and the setup requirements, making the FPGAs available for students outside of a dedicated lab is impractical.

3 Architecture

3.1 Abstraction

The purpose of the system is to facilitate teaching and learning of the functionality of the CPU. To do so, the system uses several layers of abstraction:

3.1.1 Hardware Abstraction

As the system revolves around imitating a CPU's functionality, it should contain the various parts that a CPU is composed of, as well as the way they are connected to one another. To create all those components and their interactions, a software model of each component was created. Any one of those models is defined by its purpose and behavior, and imitates, as closely as possible, the real hardware counterpart. The interaction between components is performed by following logical rules that are based on real electrical functionality (for example, master-slave flip-flop functionality is simulated by writing all data to temporal buffers until all data read operations are fulfilled).

3.1.2 Data Abstraction

In real hardware systems, data is defined by electrical or magnetic bits. Those bits use voltage or magnetic field differences to denote the bit's value. To make the data accessible to the system users, the data is represented using various representations:

- Within the system, bit values are represented using Boolean values. In addition to the trivial transformation from physical to logical values, using Boolean variables allow the system to easily perform checks on the data by using basic conditional statements available in any programming language.
- When interacting with the user, the data is represented textually. The textual representation can use the binary notation, using the '0' and '1' characters to denote the value per bit, or the hexadecimal notation that allows for better readability when handling large amounts of data.

The system offers a built in mechanism that performs conversions from any of those representations to any other.

3.2 Specifically Designed Syntax

Generating a new customized emulator affects both the assembly commands and the way those commands are processed by the CPU. For the new assembly language to take effect, changes are required in the code implementing the assembler and the CPU. The changes are not always intuitive and may require various modifications in different locations throughout the code, making manual configuration impractical.

To overcome this, a high level syntax was developed. The syntax uses a set of symbols to represent the different actions the CPU can perform, allowing the user to write relatively simple expressions without ever needing to understand the actual implementation of the system. The translation from the user-friendly syntax into code implementing both the new assembly language and its assembler is done using a subsystem developed specifically for that purpose.

3.3 Modularity

By implementing each component separately and connecting them to create the system as a whole, a high level of modularity was attained. Adding and removing components, as well as changing the connectivity of the different components, can be done easily, allowing the system to emulate different CPU architectures simply by replacing one set of components by another. One of the future objectives is to allow this kind of changes to be done automatically, using methods similar to those used currently for new assembly language generation.

4 Implementation

4.1 System Structure

This is a high level overview of the system. A more detailed view can be found in the [system design](#) section.

The system is composed of four main packages:

4.1.1 Emulator

The Emulator package is the core of the system, providing most of the functionality. The package contains all the [components](#) and defines the [architecture](#) of the emulated CPU. The Emulator package is composed of three smaller packages (Figure 4-3):

4.1.1.1 Global

The Global package implements classes that generate functionality relevant to the system as a whole, and are not related to a specific component (Figure 4-4).

4.1.1.1.1 Input Files

The [CPU emulator](#) uses two input files, both of them used by classes within the global package.

4.1.1.1.1.1 DataTransferMap.csv

The DataTransferMap.csv file serves as a configuration file to define connectivity between all the different components that can hold data. The transfer map is based on the same principles as a routing table. Each line consists of three fields:

- Current component: Indicates the current component holding the data that needs to be transferred.
- Target component: Indicates the component that the data should be transferred to.
- Next component: Indicates the next component in the data route that leads to the target component. In addition to component names, this field can also contain status values such as TARGET_REACHED or UNREACHABLE.

4.1.1.1.1.2 Program.csv

The Program.csv file represents a [pseudo-binary](#) executable file. Each line in this file defines one line of the program, as stored in the memory. Each line contains three fields:

- Memory address (Hexadecimal representation): Defines the memory address in which the content is held. This allows for memory gaps without saving all the empty memory slots between program blocks.
- Memory content (Hexadecimal representation): Defines the contents of the given memory address.
- Assembly command (Text): This field is not used by the emulator directly. The purpose of this field is to provide the assembly command that generated the line, so it can be shown by the UI.

4.1.1.1.2 Classes

4.1.1.1.2.1 Loader

Extends the ClassLoader class. The Loader class was implemented to allow loading classes using a custom path instead of the default path defined by the package hierarchical system. This allows for all the classes of a specific instruction set to be placed in the same directory.

4.1.1.1.2.2 Constants (Interface)

The Constants interface defines a set of constant values used throughout the system.

Type	Name	Value	Comments
Boolean	_0	false	Custom notation
Boolean	_1	true	Custom notation
Integer	TIMEOUT	10,000	Commands before timeout
Integer	DATA_REGISTER_SIZE	16	16-bit data
Integer	ADDR_REGISTER_SIZE	12	12-bit address
Integer	IO_REGISTER_SIZE	8	8-bit ASCII encoding
Integer	MEMORY_SIZE	$2^{\text{ADDR_REGISTER_SIZE}}$	Maximal address value
Integer	BUS_SIZE	DATA_REGISTER_SIZE	Same as biggest storing unit
Integer	DATA_COMPONENTS	22	Total number of components
Integer	DATA_TABLE_SIZE	15	Number of data components
Integer	TIMER_LIMIT	16	Max cycles per command
Integer	ALU	0	Component ID
Integer	ALU_IN0	1	Component ID
Integer	ALU_IN1	2	Component ID
Integer	ALU_OUT	3	Component ID
Integer	M	4	Component ID
Integer	BUS	5	Component ID
Integer	AR	6	Component ID
Integer	PC	7	Component ID
Integer	DR	8	Component ID
Integer	AC	9	Component ID
Integer	IR	10	Component ID
Integer	TR	11	Component ID
Integer	TR0	TR	Backwards compatibility
Integer	TR1	12	Component ID
Integer	INPR	13	Component ID
Integer	OUTR	14	Component ID
Integer	E	15	Component ID
Integer	R	16	Component ID
Integer	S	17	Component ID
Integer	I	18	Component ID
Integer	IEN	19	Component ID
Integer	FGI	20	Component ID
Integer	FGO	21	Component ID
Integer	TIMER	22	Component ID
Integer	UNREACHABLE	-99	Impossible data transfer
Integer	TARGET_REACHED	-1	Data transfer complete

Table 4-1: System constants and values

4.1.1.1.2.3 *Value*

Implements the [Constants](#) interface. An object of the Value class represents a numerical value. The numerical value is stored in a binary representation, implemented as an array of Boolean variables. The binary value has a configurable number of bits. Smaller numbers do not reduce the number of bits. They instead fill the free bits with leading zeroes. A value can be set or accessed using decimal (integer) or hexadecimal (string) representations. In its binary representation, parts of the value can be accessed by providing a specific bit or a range of bits. Parts of the value can be set or accessed in decimal or hexadecimal representations. The Value class can be used as a representation converter, and it supports negative values. In its binary representation, a negative value is defined using 2's complement, calculated using the specified amount of bits.

4.1.1.1.2.4 *DataTransferMap*

Implements the [Constants](#) interface. The data transfer map is a 2-dimensional array that defines the route used when transferring data from one component to another. The rows and columns represent the components, where any component is represented by the index matching its ID. The rows represent the components currently holding the data, the columns represent the components the data is transferred to and the intersections represent the next component in the data route. In addition, any intersection can hold the status values of TARGET_REACHED, if the current component is the same as the target component, or UNREACHABLE if no route exist to transfer the data from the current component to the target. Some components cannot hold data and therefore cannot be used as targets. The transfer map is filled using a two-step algorithm. At first, all intersections of a row and column with the same index are marked as TARGET_REACHED and the rest are marked as UNREACHABLE. In the second step, the [DataTransferMap.csv](#) file is loaded and used to update the map. Only intersections different from the default are required to be written in the DataTransferMap.csv file, but no error occurs if a line in the file is the same as an existing intersection value.

To facilitate use, the [ALU](#) is defined as a single component in addition to the definition of each of his inputs and outputs. Whenever the ALU is set as the target, the transfer map will automatically route towards the correct input of the ALU. If the ALU is set as the current component, the transfer map will refer to it as if it was the ALU output.

For backwards compatibility, TR and TR0 share the same ID, and are represented by the same entrance in the transfer map.

The default transfer map, supporting the [M. Morris Mano architecture](#), appears in the next page (Table 4-2).

Target Current	ALU	ALU_IN0	ALU_IN1	ALU_OUT	M	BUS	AR	PC	DR	AC	IR	TR0	TR1	INPR	OUTR
ALU	Green	Red	Red	Green	Red	Red	Red	Red	Red	AC	Red	Red	Red	Red	Red
ALU_IN0	Green	Green	Red	ALU_OUT	Red	Red	Red	Red	Red	ALU_OUT	Red	Red	Red	Red	Red
ALU_IN1	Green	Red	Green	ALU_OUT	Red	Red	Red	Red	Red	ALU_OUT	Red	Red	Red	Red	Red
ALU_OUT	Red	Red	Red	Green	Red	Red	Red	Red	Red	AC	Red	Red	Red	Red	Red
M	Red	Red	Red	Red	Green	Red	BUS	BUS	BUS	Red	BUS	BUS	BUS	Red	BUS
BUS	Red	Red	Red	Red	M	Red	AR	PC	DR	Red	IR	TR0	TR1	Red	OUTR
AR	Red	Red	Red	Red	BUS	Red	Green	BUS	BUS	Red	BUS	BUS	BUS	Red	BUS
PC	Red	Red	Red	Red	BUS	Red	BUS	Green	BUS	Red	BUS	BUS	BUS	Red	BUS
DR	ALU_IN0	ALU_IN0	Red	ALU_IN0	BUS	Red	BUS	BUS	Green	ALU_IN0	BUS	BUS	BUS	Red	BUS
AC	ALU_IN1	Red	ALU_IN1	ALU_IN1	BUS	Red	BUS	BUS	BUS	Green	BUS	BUS	BUS	Red	BUS
IR	Red	Red	Red	Red	BUS	Red	BUS	BUS	BUS	Red	Green	BUS	BUS	Red	BUS
TR0	Red	Red	Red	Red	BUS	Red	BUS	BUS	BUS	Red	BUS	Green	BUS	Red	BUS
TR1	Red	Red	Red	Red	BUS	Red	BUS	BUS	BUS	Red	BUS	BUS	Green	Red	BUS
INPR	ALU_IN1	Red	ALU_IN1	ALU_IN1	Red	Red	Red	Red	Red	ALU_IN1	Red	Red	Red	Green	Red
OUTR	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Red	Green

Table 4-2: Data transfer map

- Green – Target reached
- Red – Unreachable

4.1.1.1.2.5 *ProgramLine*

Implements the [Constants](#) interface. A program line object stores one assembly command. The command is represented by the binary representation of the command, the binary representation of the command's memory address and a string containing the assembly command as it appeared in the [Program.csv](#) file.

4.1.1.1.2.6 *Program*

A program object stores a complete program as a list of [program lines](#). The objective of this class is to serve as a list with no set size. Once all the program lines were loaded, the Program can convert the list into a workable array with a well defined length and indices.

4.1.1.1.2.7 *Processor*

Implements the [Constants](#) interface. The Processor class implements the actual [CPU architecture](#). It contains all the necessary [components](#) and the [data transfer map](#). The processor defines the type, name and ID for each component used by the CPU. During [program execution](#), the processor holds the [system state](#) at all times. Every [micro-operation](#) execution will affect the values of the processor's components, moving it to the next system state.

In order to obtain an easy access to all the different components, an array was added to the processor. Each array element points to a component. The component's ID defines the index of the element in the array pointing to it.

4.1.1.1.2.8 *iInstructionsUCode (Interface)*

The iInstructionsUCode interface defines the methods any InstructionsUCode class needs to implement. All the instruction sets, including the default one and the auto-generated ones, will implement this interface.

4.1.1.1.2.9 *InstructionsUCode*

Implements the [iInstructionsUCode](#) and [Constants](#) interfaces. This is the default system's [instruction set](#). Its purpose is to perform [micro-operations](#) on the [processor](#), according to the currently [executed](#) assembly command. The InstructionsUCode class implements a method for each [timer cycle](#). When executing a micro-operation, the method corresponding to the current cycle is invoked. The method will go through all possible actions for that cycle, looking for an applicable action. If the [system's state](#) allows for an action to occur, that action is executed on the CPU.

Whenever a new [instruction set is generated](#), a copy of this class is generated as well, adapted to the newly defined instruction set. The new copy's name includes the new instruction set's name as a prefix.

4.1.1.1.2.10 *Emulator*

Implements the [Constants](#) interface. The Emulator class combines the [processor](#), [instruction set](#) and [assembler](#) into a working emulated computer. The emulator can load a [program](#), assemble it, load the resulting [pseudo-binary](#) into the system's [memory](#) and use the instruction set to execute it on the processor. The emulator can also provide component values to be used by [other parts of the system](#).

4.1.1.2 Components

The Component package implements the classes used as the system's [components](#). Each class emulates the class emulates the functionality of a specific hardware component (

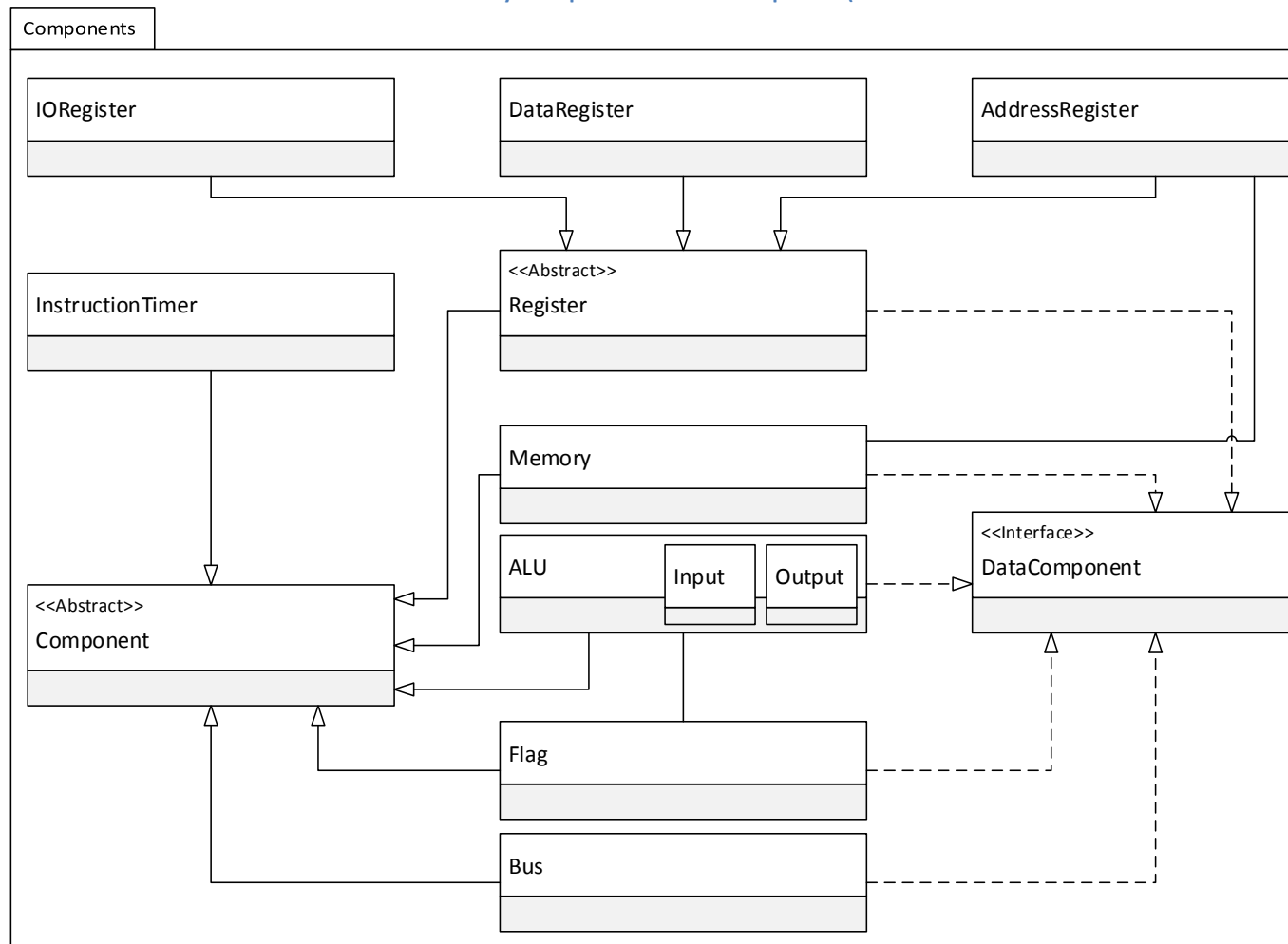


Figure 4-5).

4.1.1.2.1 Classes

4.1.1.2.1.1 Component (Abstract)

The Component class defines the basic information any component should have. The basic information includes a component ID and a component name. The ID is the same as described in the [constants](#) table and the name is a more descriptive string. A component ID must be a positive integer, as those IDs are later used as indices for arrays.

4.1.1.2.1.2 InstructionTimer

Extends the [Component](#) class. The instruction timer's function is to count the cycles within each assembly command. The maximum number of cycles allowed for each command is defined by the `TIMER_LIMIT` [constant](#). After reaching the limit, the timer resets to zero. An assembly command can be implemented using less than the maximum number of cycles. If that is the case, the instruction timer will do nothing on the remaining cycles unless given the reset command. The reset command sets the counter to zero without reaching the maximum boundary. A [micro-operation](#) is performed in one instruction cycle, but more than

one micro-operation can be performed in the same cycle. Performing more than one micro-operation in the same cycle should be done carefully, as the [system's state](#) is updated only at the end of each cycle. The result of changing the value in a given component more than once before the state was updated is not defined, and may result in data loss or corruption.

4.1.1.2.1.3 DataComponent (Interface)

The DataComponent interface defines all the methods used by the [components](#) that can store data. Each data-storing component implements the relevant methods, and returns a default value while doing nothing for irrelevant methods. While most components use a small subset of the DataComponent interface, the interface was needed to allow for a polymorphic behavior when handling data.

4.1.1.2.1.4 Bus

Extends the [Component](#) class and implements the [DataComponent](#) interface. The Bus class emulates the functionality of a data bus, connecting [components](#) to one another, allowing data flow between them. The bus is used to move data, but is not considered a data-storing component. The value on the bus is the last value moved through it and it has no meaning by itself. The bus will accept new data without the need for the processor to enable it for writing. Data written to the bus overrides the previous data immediately. Moving more than one value through the bus in the scope of a single [cycle](#) may result in data loss or corruption.

4.1.1.2.1.5 Flag

Extends the [Component](#) class and implements the [DataComponent](#) interface. This class emulates the functionality of a 1-bit flag. The flag simulates the master-slave functionality of a real hardware flag, allowing its old value to be read while updating its value in the same [cycle](#). A flag object can only hold a true or false value, not a numerical value. In addition, the flag notes if its value was set to true since the last time its value was read.

4.1.1.2.1.6 Register (Abstract)

Extends the [Component](#) class and implements the [DataComponent](#) interface. The Register class emulates the functionality of a hardware [register](#). The register stores a numeric [value](#) in its binary representation as a series of bits. A register simulates the master-slave functionality of a real hardware register, allowing its old value to be read while updating its value in the same [cycle](#). For the value of a register to be changed, the system must enable it for writing. A register has two special operations, clear and increment, that set the value to zero or increase the value by one, respectively. The system uses three different kinds of registers:

- [Data registers](#): Used to store data or commands.
- [Address registers](#): Used to store memory addresses.
- [I/O registers](#): Used to store data used by input and output devices.

The register types differ only by the number of bits used to store their value.

4.1.1.2.1.7 DataRegister

Extends the [Register](#) class. Data [registers](#) are used to store data or commands. Their length is defined by the DATA_REGISTER_SIZE [constant](#).

4.1.1.2.1.8 *AddressRegister*

Extends the [Register](#) class. Address [registers](#) are used to store [memory](#) addresses. Their length is defined by the ADDR_REGISTER_SIZE [constant](#).

4.1.1.2.1.9 *IORegister*

Extends the [Register](#) class. I/O [registers](#) are used to store data used by input and output devices. Their length is defined by the IO_REGISTER_SIZE [constant](#).

4.1.1.2.1.10 *Memory*

Extends the [Component](#) class and implements the [DataComponent](#) interface. The Memory class emulates the system's memory as an array of [values](#). The memory points to an [address register](#) whose value represents the memory address to be written or read. Whenever a memory slot is accessed, the value of the address register is used as the array's index. The memory simulates the master-slave functionality of real hardware, allowing an old value to be read while updating the value in the same memory slot within the same [cycle](#). Accessing different slots at the same time is not possible, as all slots share the same address register, which can hold only one value at any given time. Each slot can contain one command or one data value represented as a numerical value in a binary representation. The number of bits used to represent each value is defined by the DATA_REGISTER_SIZE [constant](#). The total memory size is defined by the MEMORY_SIZE constant, which equals to $2^{\text{ADDR_REGISTER_SIZE}}$. This allows for one memory slot for each possible value of the address register.

An additional array of strings, the same size as the memory, was added to the memory. Each element in the string array holds the assembly command corresponding to the memory slot in the respective index. While this extra data is not required for the emulator's functionality, it allows the commands to be mapped to their assembly origins without decompiling the [pseudo-binary](#) code.

4.1.1.2.1.11 *ALU*

Extends the [Component](#) class and implements the [DataComponent](#) interface. The ALU class emulates the functionality of an [arithmetic and logic unit](#). The ALU is composed of two inner classes. An ALU has two [input](#) and one [output](#) objects. Binary operations use both inputs as the two operands and the output to store the result. Unary operations require the input used to be stated explicitly. Logical operations will set the output value to 1 if the result is 'true' or 0 if the result is 'false'. Arithmetical operations will set the output to the resulting numerical value. The ALU points to an end carry [flag](#), which it sets to true if the arithmetical calculation generates an end carry. Data can be lost if the result exceeds the number of bits used to represent the output's value. Both inputs and the output have their number of bits defined by the DATA_REGISTER_SIZE [constant](#). The ALU, both inputs and the output have an ID each.

4.1.1.2.1.11.1 *Inner Classes*

4.1.1.2.1.11.1.1 *Input*

Extends the [Component](#) class and implements the [DataComponent](#) interface. The input's number of bits is defined by the DATA_REGISTER_SIZE [constant](#) and it requires the system to enable it for writing.

4.1.1.2.1.11.1.2 Output

Extends the [Component](#) class and implements the [DataComponent](#) interface. The output's number of bits is defined by the DATA_REGISTER_SIZE [constant](#) and it does not require the system to enable it for writing. The output's value should not be set in any way other than by the ALU's functionality.

4.1.1.2.1.11.2 Operations

The ALU supports a variety of operations. Operations using both inputs as operands are binary operations, and operations using only one of the inputs are called unary operations.

4.1.1.2.1.11.2.1 Binary Operations

The binary operations can be divided into two categories:

4.1.1.2.1.11.2.1.1 Arithmetical Operations

These operations perform some kind of mathematical or a bitwise calculation and store the result in the ALU's output. The arithmetical operations are:

- Sum: Sums the values of both inputs.
- Subtract: Subtracts the value of 'Input 1' from 'Input 0'.
- Multiply: Multiplies the values of both inputs.
- Divide: Divides 'Input 0' by 'Input 1'.
- Modulo: Calculates the remainder of the division of 'Input 0' by 'Input 1'.
- AND: Performs a bitwise AND operation on both inputs.
- OR: Performs a bitwise OR operation on both inputs.
- XOR: Performs a bitwise XOR operation on both inputs.

4.1.1.2.1.11.2.1.2 Logical Operations

These operations perform a logical evaluation on the inputs and set the output value to 1 if the evaluation returned a 'true' result or to 0 if the evaluation returned a 'false' result. The logical operations are:

- Equal: Checks if 'Input 0' is equal to 'Input 1'.
- Not equal: Checks if 'Input 0' is not equal to 'Input 1'.
- Greater than: Checks if 'Input 0' is greater than 'Input 1'.
- Less than: Checks if 'Input 0' is less than 'Input 1'.
- Greater or equal: Checks if 'Input 0' is greater than, or equal to, 'Input 1'.
- Less than or equal: Checks if 'Input 0' is less than, or equal to, 'Input 1'.

4.1.1.2.1.11.2.2 Unary Operations

The unary operations operate on a single input, thus requiring the input used to be announced explicitly. The result of the operation is stored in the ALU's output. Each of the unary operations exists for both the inputs. The unary operations are:

- Pass through: Moves the value of the specified input to the output, without changing it.
- Complement: Performs a bitwise NOT operation on the specified input.

- Shift left: Moves the bits of the specified input to the left. The offset and the value used to fill the missing bits are given as parameters.
- Shift right: Moves the bits of the specified input to the right. The offset and the value used to fill the missing bits are given as parameters.

4.1.1.3 Exceptions

The Exceptions package defines custom exceptions used by the system.

4.1.1.3.1 NegativeIdException

This exception is thrown whenever a [component](#) is given a negative ID number.

4.1.2 Assembler

The Assembler package functions as the system's [assembler](#), whose function is to compile assembly programs into a [pseudo-binary](#) file (Figure 4-6).

4.1.2.1 Input / Output

The assembler uses one input file and generates two output files.

4.1.2.1.1 Program.dat

Program.dat is the input file for the assembler. It contains the program exactly as it was written by the [programmer](#). The content of this file is loaded into the assembler, and is then compiled into the [Program.csv](#) and [VarTable.csv](#) files.

4.1.2.1.2 Program.csv

The [Program.csv](#) file is one of the outputs of the assembler.

4.1.2.1.3 VarTable.csv

The VarTable.csv file is also created by assembler. This file maps the labels used in the program and the address in memory each occupy. It is used by the UI. The two fields in every line are:

- Label name (Text): This is the name of the label. A label can be used to identify a variable, a subroutine or any part of the code that the program can branch to.
- Memory address (Hexadecimal representation): Similar to the Program.csv file, this field represents the address of the corresponding label.

4.1.2.2 Classes

4.1.2.2.1 iAssembler (Interface)

The iAssembler interface defines the methods any assembler class needs to implement. All assemblers, the default one and the auto-generated ones, will implement this interface.

4.1.2.2.2 Assembler

Implements the [iAssembler](#) interface. This is the default system's assembler. Its purpose is to load the [Program.dat](#) file and feed it to the assembler's [parser](#).

Whenever a new [instruction set is generated](#), a copy of this class is generated as well, adapted to invoke the appropriate parser. The new copy's name includes the new instruction set's name as a prefix.

4.1.2.2.3 `sym`

This class defines the constant symbols that represent the tokens known to the [lexer](#). The known tokens are:

- **NUMBER:** A numerical value in a decimal, hexadecimal or binary representation.
- **NEGATIVE:** The negative sign ('-').
- **EOF:** End of File.
- **ORG:** Short for "organize" – A command that instructs the assembler to insert new content to a specific memory address.
- **BIN:** Notifies the assembler that the following numerical value has a binary representation.
- **DEC:** Notifies the assembler that the following numerical value has a decimal representation.
- **HEX:** Notifies the assembler that the following numerical value has a hexadecimal representation.
- **ID:** Defines an identifier token. An identifier can be any name given to a label, or any command.
- **END:** The keyword 'END' represents the end of the program.
- **COMMA:** The comma sign (',').
- **NEWLINE:** A character representing the end of the current line.
- **error:** Defines a token that does not match any known pattern.

4.1.2.2.4 `Lexer`

The Lexer is a lexical analyzer. It analyzes the program file, returning the type, and when applicable, the value of each of the tokens it encounters. The types are defined in the [sym](#) class.

4.1.2.2.5 `parser`

The parser is a syntactical analyzer and it serves two functions:

- **Syntax validation:** The parser validates that the program is syntactically correct.
- **Compilation:** The parser translates the [Program.dat](#) file into a [Program.csv](#) file. In addition, it generates a [VarTable.csv](#) file that corresponds to the program being assembled.

Whenever a new instruction set is generated, a copy of this class is generated as well, adapted to recognize the new instruction set and the way it should be translated into a [pseudo-binary](#) file. The new copy's name includes the new instruction set's name as a prefix.

4.1.3 `Parser`

The Parser package is used to generate new instruction sets according to given [template files](#). This package implements the [instruction set generator](#) (Figure 4-7).

4.1.3.1 *Input / Output*

The parser uses one input file and generates five output files. The Java code files are compiled into class files and deleted afterwards by the [Compiler](#) class.

4.1.3.1.1 Template.dat

The Template.dat file is the input file for the parser. It contains the [instruction set template](#) as it was written by the [instruction set editor](#). The content of this file is loaded into the compiler, and is then compiled to create the [assembler](#) and [instruction set](#).

4.1.3.1.2 Assembler.cup

This file is used by the [Compiler](#) class to generate the [sym](#), [Parser](#) and [Assembler](#) Java classes. The file defines the syntactic rules for the [assembler's syntactical analyzer](#).

4.1.3.1.3 Assembler.java

The Assembler.java file implements a new [assembler class](#). The class is generated using the name of the new instruction set as a prefix for the newly generated class name.

4.1.3.1.4 sym.java

This file is generated automatically by Java CUP, but it is not needed, as it is identical to the [sym class](#) used by the default assembler.

4.1.3.1.5 Parser.java

The Parser.java file implements the [assembler's parser class](#). The class is generated using the name of the new instruction set as a prefix for the newly generated class name.

4.1.3.1.6 UCode.java

The UCode.java file implements the new [instruction set](#). The class is generated using the name of the new instruction set as a prefix for the newly generated class name.

4.1.3.2 Classes

4.1.3.2.1 Compiler

The Compiler class is used to load the [Template.dat](#) file and feed it to the compiler's [parser](#). After generating all the Java code needed for a new [assembler](#) and [instruction set](#), the compiler compiles those classes into Java Bytecode. In addition, the compiler deletes the source code. The assembler's [Lexer class](#) Bytecode is copied from the 'Resources' directory, as it is the same as the default's assembler and does not need to be compiled again.

4.1.3.2.2 sym

This class defines the constant symbols that represent the tokens known to the [lexer](#). The known tokens are divided into three states:

4.1.3.2.2.1 Regular Tokens

Regular tokens have no special state. They can be divided into different classifications:

4.1.3.2.2.1.1 Keywords

The keywords are words used for a specific functionality.

- **FORMAT:** Indicates the start of the [instruction format](#) section.
- **ACCESSMODES:** Indicates that the following code defines the access modes known to the assembler.

- TAG: When defining the format of an instruction, indicates the location a label should be placed when assembled.
- OPCODE: Indicates that the following value should be translated into an OpCode when assembled.
- AM: When defining the format of an instruction, indicates the location the access mode should be placed when assembled.
- CODE: Indicates the start of the instruction set's code section.
- END: Indicates the end of an assembly instruction.
- HLT: Indicates the end of [execution](#).

4.1.3.2.2.1.2 Component Instructions

These tokens represent commands performed on a [component](#) directly.

- COMPLEMENT: Indicates a bitwise NOT should be performed.
- INCREMENT: Indicates the value of the component should be increased by one.
- CHANGED: Indicates a [flag](#) change status needs to be checked.
- CLEAR: Indicates that the component's value should be set to zero if it is a [register](#), or to false if it is a flag.
- SET: Indicates that a flag should be set to true.

4.1.3.2.2.1.3 Assignment Operators

These are operators that define different types of assignments.

- A_ASSIGN: Assignment between [components](#).
- E_ASSIGN: Assignment of a numerical value to a variable in the system.
- F_ASSIGN: Assignment of an [instruction format](#) to an instruction.

4.1.3.2.2.1.4 Instruction Set Structure Commands

These tokens dictate the format of the [instruction set class](#).

- CYCLE: Indicates the [cycle number](#) for a group of [micro-operations](#).
- DECIMAL: Indicates that the value should be used as an integer by the system.
- IF: Indicates a Boolean condition check.
- AND: Indicates an AND operator used in an 'if' clause.
- OR: Indicates an OR operator used in an 'if' clause.
- NOT: Indicates a NOT operator used in an 'if' clause.

4.1.3.2.2.1.5 Brackets

These tokens represent different kinds of brackets.

- L_TRIANGULAR, R_TRIANGULAR: '<' and '>' respectively.
- L_CURLY, R_CURLY: '{' and '}' respectively.
- L_SQUARE, R_SQUARE: '[' and ']' respectively.
- R_BRACKET, L_BRACKET: '(' and ')' respectively.

4.1.3.2.2.1.6 Punctuation Marks

These tokens represent the different punctuation marks used.

- COLON: The colon sign (':').
- SEMICOLON: The semicolon sign(';').
- COMMA: The comma sign(',').
- HYPHEN: The hyphen sign('-').

4.1.3.2.2.1.7 Identifiers

These tokens are used to identify actions and values.

- ID: Represents a set of alphanumerical values. An ID can reference a command or a label.
- NUMBER: Represents a numerical value. Numbers can be represented in decimal, hexadecimal or binary representations.

4.1.3.2.2.1.8 System Tokens

These tokens are used by the [parser](#) while analyzing the [template file](#).

- EOF: End of File.
- error: Defines a token that does not match any know pattern.

4.1.3.2.2.2 ALU Tokens

These tokens represent operations performed by the [ALU](#).

- ALU_SUM: Perform the 'sum' operation.
- ALU_SUB: Perform the 'subtract' operation.
- ALU_MULT: Perform the 'multiply' operation.
- ALU_DIV: Perform the 'divide' operation.
- ALU_MOD: Perform the 'modulo' operation.
- ALU_EQ: Perform the 'equal' operation.
- ALU_NE: Perform the 'not equal' operation.
- ALU_GR: Perform the 'greater' operation.
- ALU_LS: Perform the 'less than' operation.
- ALU_GE: Perform the 'greater or equal' operation.
- ALU_LE: Perform the 'less than or equal' operation.
- ALU_NOT: Perform the 'not' operation.
- ALU_AND: Perform the 'and' operation.
- ALU_OR: Perform the 'or' operation.
- ALU_XOR: Perform the 'xor' operation.
- ALU_L_SHIFT: Perform the 'shift left' operation.
- ALU_R_SHIFT: Perform the 'shift right' operation.
- ALU_FILL_ZERO: Use 0 as a filler in a shift operation.
- ALU_FILL_ONE: Use 1 as a filler in a shift operation.

4.1.3.2.2.3 String Tokens

The STRINGVAL token represents a string of alphanumerical characters encased by two double quote (") signs.

4.1.3.2.3 Lexer

The Lexer class is a lexical analyzer. It analyzes the template file, returning the type, and when applicable, the value of each of the tokens it encounters. The types are defined in the [sym](#) class.

4.1.3.2.4 parser

The parser is a syntactical analyzer and it serves two functions:

- Syntax validation: The parser validates that the [template file](#) is syntactically correct.
- Compilation: The parser scans the [Template.dat](#) and generates the [Assembler.cup](#), [Assembler.java](#) and [UCode.java](#) files.

4.1.4 GUI

The GUI package implements all the GUI elements needed to create the user interface. The GUI consists of a main user interface and two file editors. The editors can be launched from the main window (Figure 4-8).

4.1.4.1 *Input / Output*

The GUI uses two files:

4.1.4.1.1 Metadata.dat

This file stores the name of the currently used [instruction set](#). At startup, this file is checked, and the last instruction set is reloaded. If the instruction set changes, this file is updated.

4.1.4.1.2 VarTable.csv

This file is created by the [assembler](#), and is used to create a table of contents of the different variables and labels for the user. This table contains the name of each label and its memory address.

4.1.4.2 *Classes*

4.1.4.2.1 mainWindow

The mainWindow class generates the main user interface. This window is the [user's control panel](#). The main window can be used by the user to monitor the [system's state](#) during the [program's execution](#), or load programs and [instruction set templates](#) using the editors. The execution of a program can be set to run from start to finish, or to run in a step-by-step mode. The step-by-step mode can be toggled between the assembly state, performing a full assembly command every step, and the advanced [micro-operation](#) state, that performs one micro-operation at a time.

4.1.4.2.2 ProgramEditor

The program editor is a simple text editor used by the [programmer](#) to load and edit programs. The editor enables the programmer to load text files containing previously written programs and to save the program to a file. It is also used to load a program to the emulator (loaded programs are not assembled automatically). The default program shown in the editor is the last program loaded. The editor loads and edits the [Program.dat](#) file.

4.1.4.2.3 TemplateEditor

The template editor is a simple text editor used by the [instruction set editor](#) to load and edit new instruction set templates. The template editor enables the instruction set editor to load text files containing previously written instruction set templates and to save the instruction set template to a file. It is also used to load an instruction set template to the emulator (loaded instruction set templates are not compiled automatically). The default template shown in the editor is the last template loaded. The editor loads and edits the [Template.dat](#) file. An additional field in the editor sets the instruction set's name. This name will be used to define the instruction set and all related classes and files. Whenever a template is loaded from an existing file, the file's name becomes the name of the instruction set (it can be edited later).

4.1.4.2.4 TimeOutDialog

To prevent badly written programs from running indefinitely, causing the user to kill the process forcefully using the operation system, a time out dialog will appear if the program exceeded a predetermined timeout limit. The timeout defines the upper limit of operations [executed](#) consecutively without reaching the end of the program. When the timeout dialog appears, the user may choose to terminate the program, or to reset the timeout counter and resume execution. The timeout dialog will reappear if the timeout limit was reached again. The upper limit is defined by the TIMEOUT [constant](#).

4.2 File System Structure

The system uses five directories:

4.2.1 AppData

This directory holds all the files used during runtime:

- [DataTransferMap.csv](#)
- [Program.csv](#)
- [VarTable.csv](#)
- [Metadata.dat](#)
- [Program.dat](#)
- [Template.dat](#)

4.2.2 Resources

This directory holds resources needed for the system to function. It contains three files:

- java-cup-11a.jar: Used when compiling [Assembler.cup](#) into a working assembler.
- [Lexer.class](#): Is copied into the instruction set's directory whenever a new instruction set is generated.
- ManoCPU.jar: Is used to hold class definitions used in compilations of new instruction sets.

4.2.3 Programs

This directory can hold program text files to be loaded into the system. Programs can be loaded from anywhere, but this is the default directory.

4.2.4 Templates

This directory can hold text files containing instruction set templates to be loaded into the system. Templates can be loaded from anywhere, but this is the default directory.

4.2.5 Machines

Each instruction set, once compiled, is stored in a directory with that instruction set's name under the "Machines" directory.

4.3 System Design

This is a detailed technical description of the system's design. A high level overview can be found in the system [structure section](#).

4.3.1 DFD 0

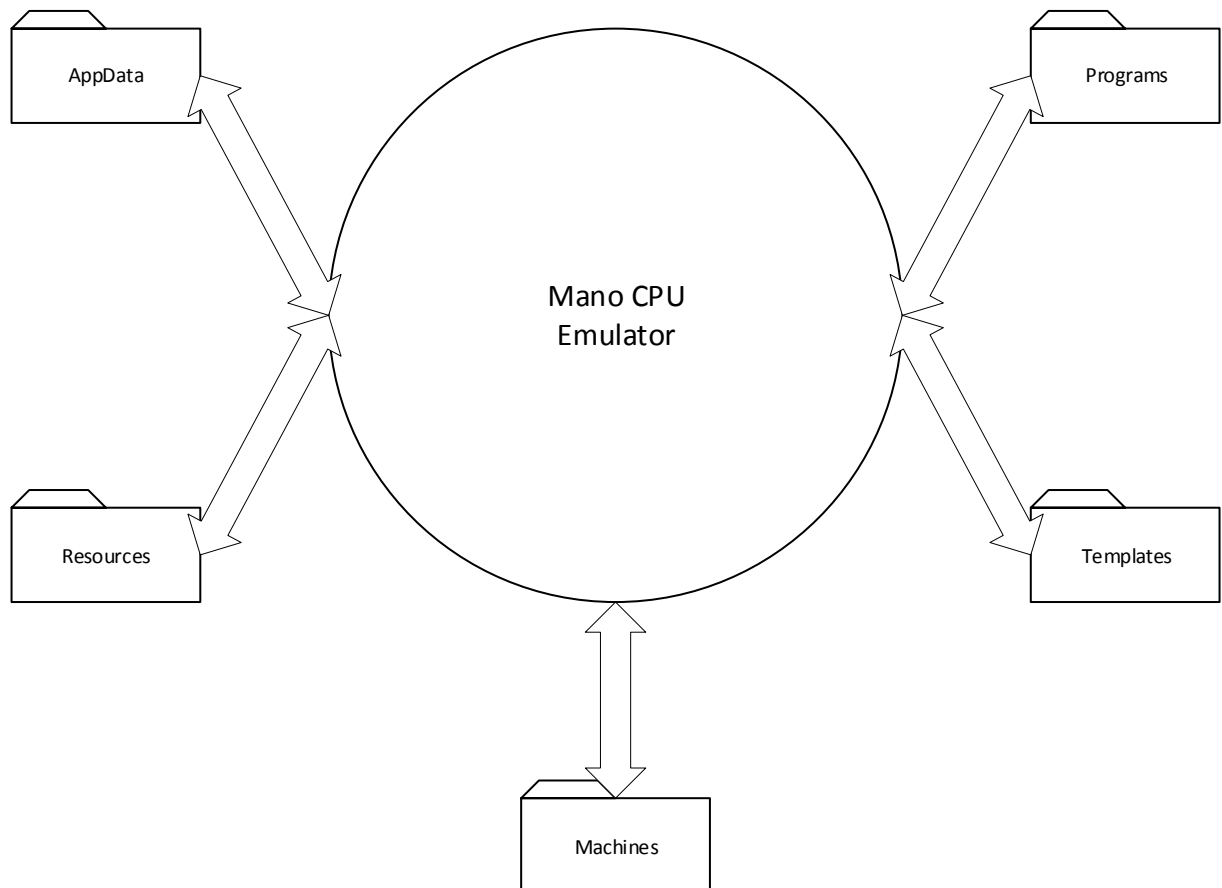


Figure 4-1: DFD 0

4.3.2 DFD 1

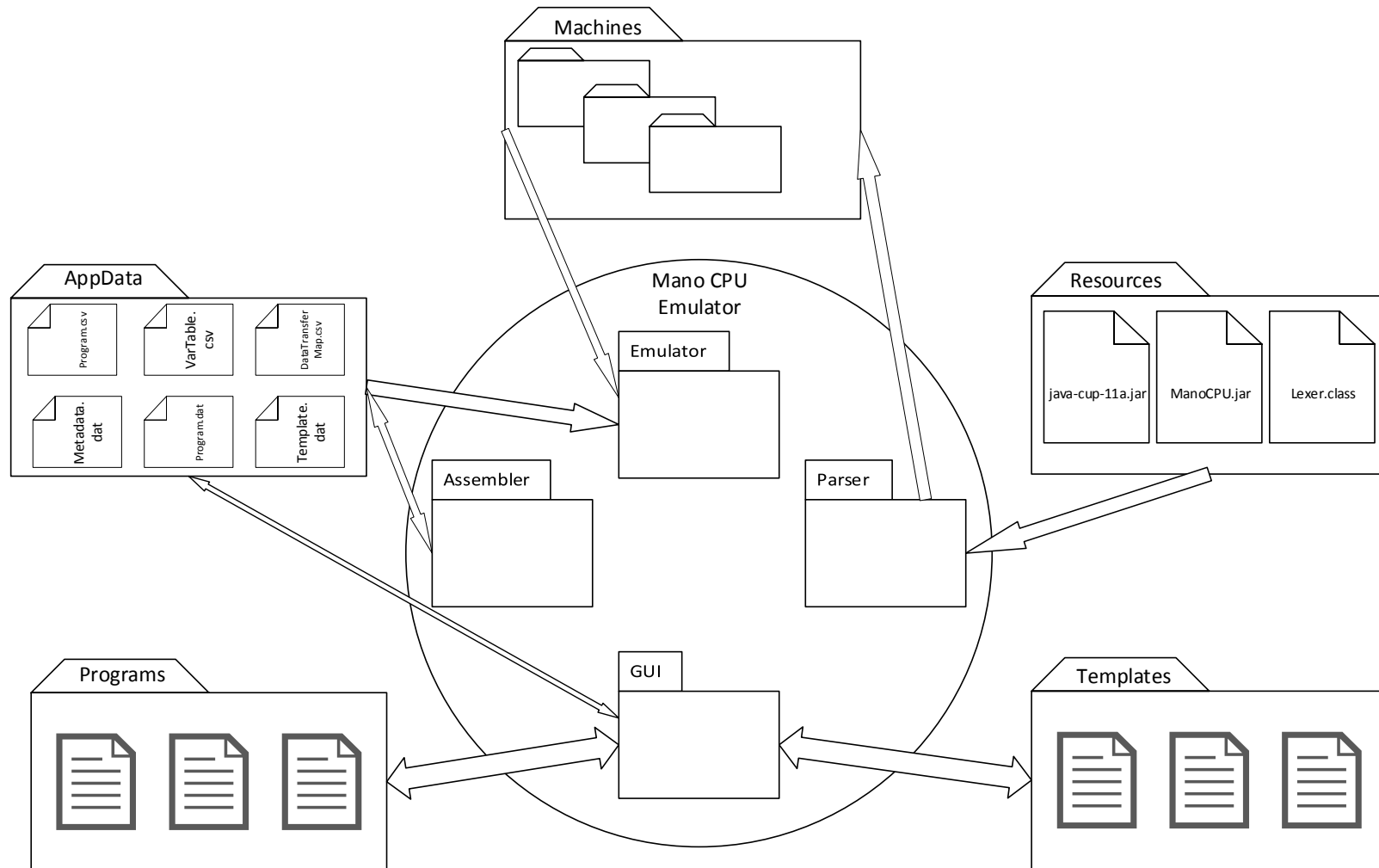


Figure 4-2: DFD 1

4.3.3 Emulator

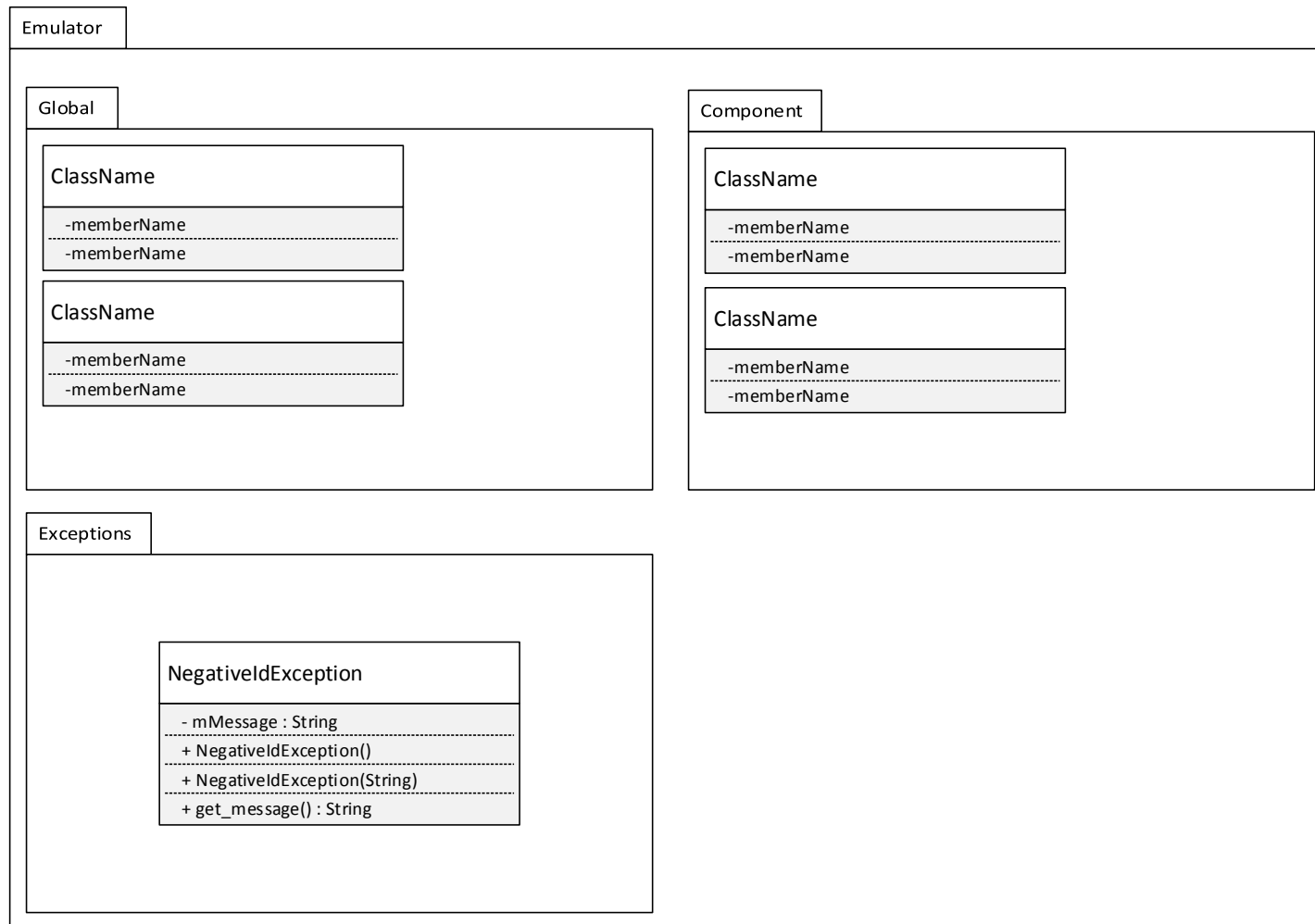


Figure 4-3: Emulator package

4.3.3.1 Global

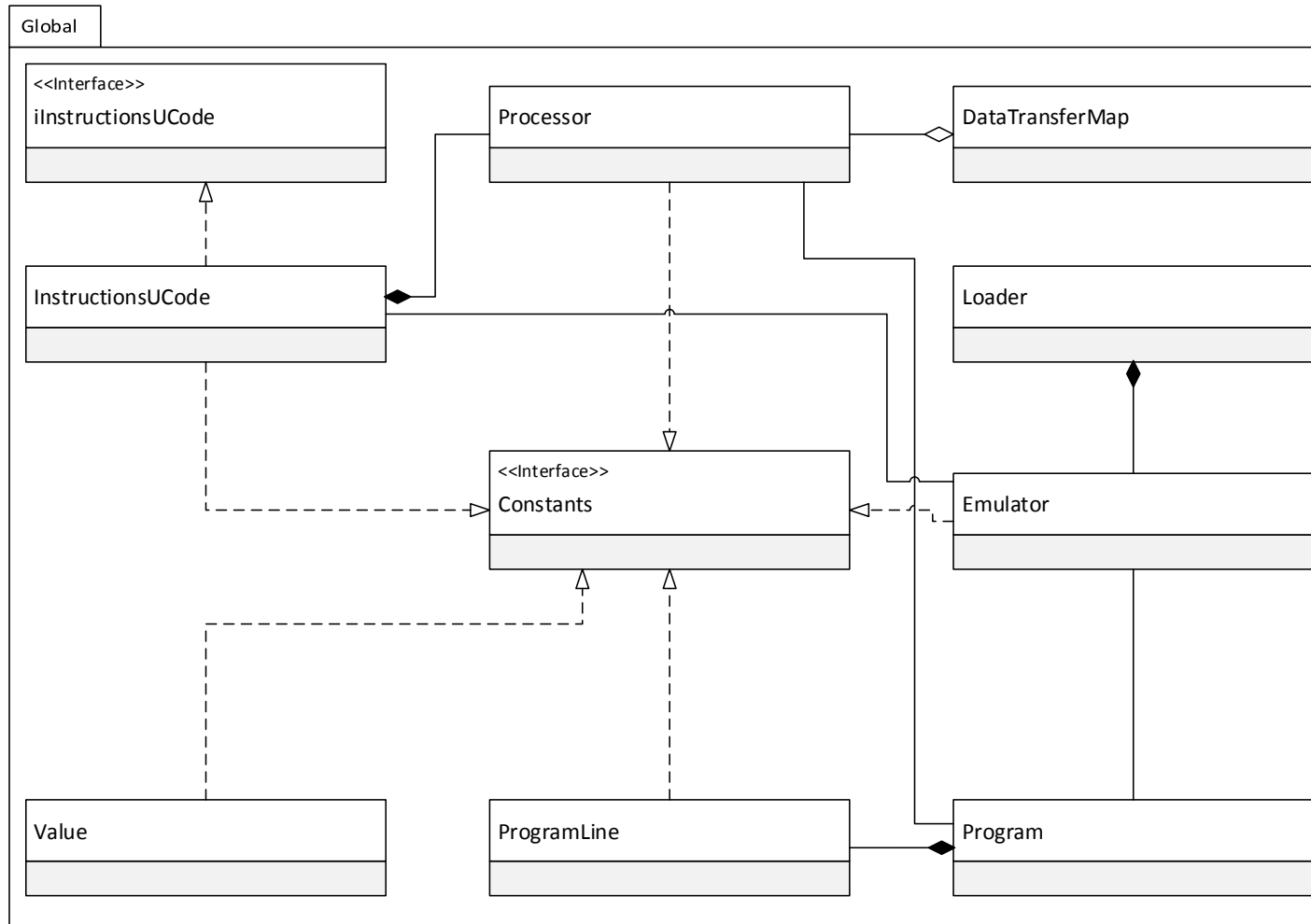


Figure 4-4: Global package

4.3.3.1.1 Loader

Loader
- mTemplateName : String
+ Loader(String)
+ loadClass(String) : Class<?>
+ findClass(String) : Class<?>
- loadClassData(String) : byte[]

4.3.3.1.2 Constants

The members of the Constants interface are detailed in the [system structure](#) section. It has no methods.

<<Interface>>
Constants

4.3.3.1.3 DataTransferMap

DataTransferMap
- mMap : int[][]
+ DataTransferMap(String)
+ nextInRoute(int, int) : int

4.3.3.1.4 ProgramLine

ProgramLine
- mAddress : Value
- mContent : Value
- mMetaCommand : String
+ ProgramLine(Value, Value, String)
+ get_address() : Value
+ get_content() : Value
+ get_metaCommand() : String

4.3.3.1.5 Program

Program
- mLines : ArrayList<ProgramLine>
+ addLine(ProgramLine) : void
+ getLines() : ProgramLine[]
+ clear() : void

4.3.3.1.6 Value

Value
- mSize : int
- mContent : boolean[]
+ Value(int)
+ Value(boolean[])
+ Value(int, int)
+ Value(String)
+ set_content(boolean[]) : void
+ set_content(int) : void
+ set_content(String) : void
+ set_content(Value) : void
+ get_size() : int
+ get_content() : boolean[]
+ get_decimal() : int
+ get_hexadecimal() : String
_toBinary(int, int) : boolean[]
_toBinary(String) : boolean[]
_toDecimal(boolean[]) : int
_toDecimal(String) : int
_toHexadecimal(boolean[]) : String
_toHexadecimal(int, int) : String
_toHexadecimal(String) : String
+ toString() : String
+ complement() : void
+ increment() : void

4.3.3.1.7 Emulator

Emulator
- mManoCPU : Processor
- mUCode : iInstructionsUCode
- mAssembler : iAssembler
- mCycleDescription : String
- mAssemblyCommand : String
+ run(boolean) : boolean
+ assemble() : void
+ getSystemComponent(int) : String
+ getCycleDescription() : String
+ getAssemblyCommand() : String
+ getSystemMemory() : String[][]
+ reset() : void
+ stopProgram() : void
+ notifyInput() : void
+ notifyOutput() : void
+ getProgram() : void
+ loadTemplate(String) : void
+ loadDefaultTemplate() : void

4.3.3.1.8 iInstructionsUCode

<<Interface>> iInstructionsUCode
+ setCPU(Processor) : void
+ t0() : String
+ t1() : String
+ t2() : String
+ t3() : String
+ t4() : String
+ t5() : String
+ t6() : String
+ t7() : String
+ t8() : String
+ t9() : String
+ t10() : String
+ t11() : String
+ t12() : String
+ t13() : String
+ t14() : String
+ t15() : String

4.3.3.1.9 InstructionsUCode

InstructionsUCode
- mCPU : Processor
- mCycleDescription : String
+ InstructionsUCode()
/+ setCPU(Processor) : void
/+ t0() : String
/+ t1() : String
/+ t2() : String
/+ t3() : String
/+ t4() : String
/+ t5() : String
/+ t6() : String
/+ t7() : String
/+ t8() : String
/+ t9() : String
/+ t10() : String
/+ t11() : String
/+ t12() : String
/+ t13() : String
/+ t14() : String
/+ t15() : String

4.3.3.1.10 Processor

Processor
_ constantTable : HashMap<String, Integer>
+ mComponentsList : DataComponent[]
- mInstructionTimer : InstructionTimer
- mTransferMap : DataTransferMap
- mOpCode : int
- e : Flag
- r : Flag
- s : Flag
- i : Flag
- ien : Flag
- fgi : Flag
- fgo : Flag
- ar : AddressRegister
- pc : AddressRegister
- dr : DataRegister
- ac : DataRegister
- ir : DataRegister
- tr0 : DataRegister
- tr1 : DataRegister
- inpr : IORegister
- outr : IORegister
- bus : Bus
- memory : Memory
- alu : ALU
+ Processor()
+ set_opCode(int) : void
+ get_opCode() : int
+ get_cycleNum() : int
+ loadProgram(Program) : void
+ resetTimer() : void
+ halt : void
+ checkOpCode(int) : boolean
+ nextCycle() : void
+ moveData(int, int) : void
+ moveData(Value, int) : void

4.3.3.2 Components

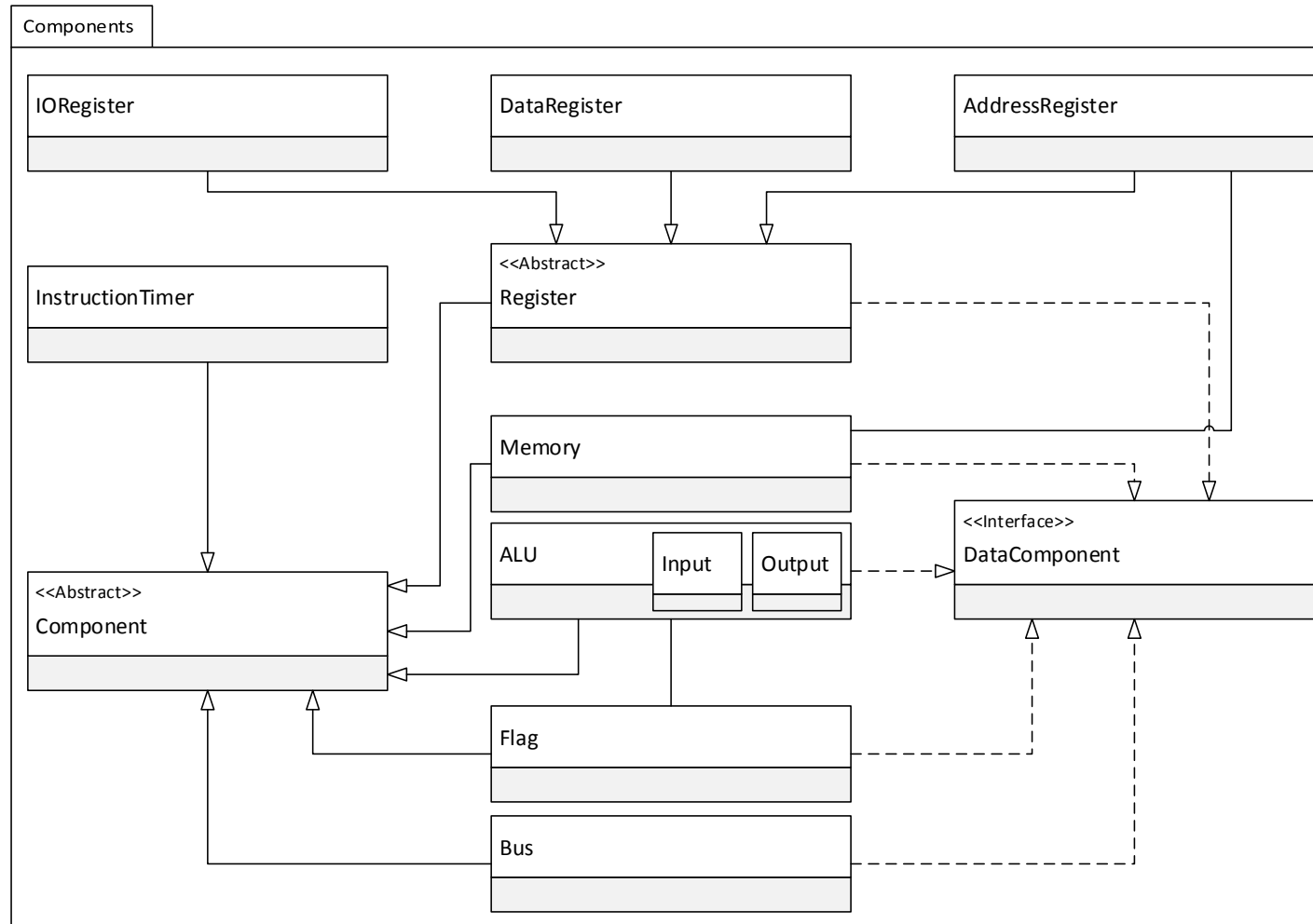


Figure 4-5: Components package

4.3.3.2.1 Component

<<Abstract>>	
Component	
- mId : int	
- mName : String	
+ set_id(int) : void	
+ set_name(String) : void	
+ get_id() : int	
+ get_name() : String	

4.3.3.2.2 InstructionTimer

InstructionTimer	
- mCurrentCycle : int	
- mReset : boolean	
+ InstructionTimer(int, String)	
+ get_currentCycle() : int	
+ pulse() : void	
+ reset() : void	

4.3.3.2.3 DataComponent

This class has no members and all methods are implemented in other classes.

<<Interface>>	
DataComponent	

4.3.3.2.4 Bus

Bus	
- mValue : Value	
+ Bus(int, String)	
/+ set_value(Value) : void	
/+ get_value() : Value	
/+ get_row(int) : Value	
/+ get_value(int, int) : Value	
/+ get_value(int) : Value	
/+ evaluateAsBoolean() : boolean	
/+ evaluateAsBoolean(int) : boolean	
/+ evaluateAsBoolean(int, int) : boolean	
/+ get_decimal() : int	
/+ get_decimal(int) : int	
/+ get_decimal(int, int) : int	

4.3.3.2.5 Flag

Flag	
- mValue : boolean	
- mBuffer : boolean	
- mChanged : boolean	
+ Flag(int, String)	
/+ set_value(Value) : void	
/+ get_value() : Value	
/+ evaluateAsBoolean() : boolean	
/+ get_decimal() : int	
/+ set() : void	
/+ clear() : void	
/+ complement() : void	
/+ changed() : boolean	
/+ update() : void	

4.3.3.2.6 Register

<<Abstract>>	
Register	
- mValue : Value	
- mBuffer : Value	
- mWriteEnable : boolean	
/+ set_value(Value) : void	
/+ set_value(int, Value) : void	
/+ set_value(int, int, Value) : void	
/+ get_value() : Value	
/+ get_value(int) : Value	
/+ get_value(int, int) : Value	
/+ evaluateAsBoolean() : boolean	
/+ evaluateAsBoolean(int) : boolean	
/+ evaluateAsBoolean(int, int) : boolean	
/+ get_decimal() : int	
/+ get_decimal(int) : int	
/+ get_decimal(int, int) : int	
/+ enableWrite() : void	
/+ disableWrite() : void	
/+ isWritable() : boolean	
/+ clear() : void	
/+ increment() : void	
/+ update() : void	

4.3.3.2.7 DataRegister

DataRegister
+ DataRegister(int, String)

4.3.3.2.8 AddressRegister

AddressRegister
+ AddressRegister(int, String)

4.3.3.2.9 IORegister

IORegister
+ IORegister(int, String)

4.3.3.2.10 Memory

Memory
- mMemory : Value[]
- mBuffer : Value[]
- mMetaCommand : String[]
- mAddress : AddressRegister
- mWriteEnable : boolean
+ Memory(int, String, AddressRegister)
/+ set_value(Value) : void
/+ set_metaCommand(String) : void
/+ get_value() : Value
/+ get_metaCommand() : String
/+ get_row(int) : Value
/+ get_value(int, int) : Value
/+ get_value(int) : Value
/+ evaluateAsBoolean() : boolean
/+ evaluateAsBoolean(int) : boolean
/+ evaluateAsBoolean(int, int) : boolean
/+ get_decimal() : int
/+ get_decimal(int) : int
/+ get_decimal(int, int) : int
/+ enableWrite() : void
/+ disableWrite() : void
/+ isWritable() : boolean
/+ update() : void

4.3.3.2.11 ALU

ALU
- mInput0 : Input
- mInput1 : Input
- mBuffer : Value[]
- mOutput : Output
- mEndCarry : Flag
+ ALU(int, String, Flag)
/* get_value() : Value
/* get_value(int) : Value
/* get_value(int, int) : Value
/* get_input0() : DataComponent
/* get_input1() : DataComponent
/* get_output() : DataComponent
/* evaluateAsBoolean() : boolean
/* evaluateAsBoolean(int) : boolean
/* evaluateAsBoolean(int, int) : boolean
/* get_decimal() : int
/* get_decimal(int) : int
/* get_decimal(int, int) : int
/* enableWrite0() : void
/* enableWrite1() : void
/* disableWrite0() : void
/* disableWrite1() : void
/* isWritable0() : boolean
/* isWritable1() : boolean
/* passThrough0() : void
/* passThrough1() : void
/* sum() : void
/* subtract() : void
/* multiply() : void
/* divide() : void
/* modulo() : void
/* and() : void
/* or() : void
/* xor() : void
/* complement0() : void
/* complement1() : void
/* equal() : void
/* notEqual() : void
/* greaterThan() : void
/* lessThan() : void
/* greaterOrEqual() : void
/* lessOrEqual() : void
/* shiftLeft0() : void
/* shiftLeft1() : void
/* shiftRight0() : void
/* shiftRight1() : void

4.3.3.2.11.1 Input

Input
- mValue : Value
- mWriteEnable : boolean
+ Input(int, String)
/* set_value(Value) : void
/* get_value() : Value
/* get_value(int) : Value
/* get_value(int, int) : Value
/* evaluateAsBoolean() : boolean
/* evaluateAsBoolean(int) : boolean
/* evaluateAsBoolean(int, int) : boolean
/* get_decimal() : int
/* get_decimal(int) : int
/* get_decimal(int, int) : int
/* enableWrite() : void
/* disableWrite() : void
/* isWritable() : boolean

4.3.3.2.11.2 Output

Output
- mValue : Value
+ Output(int, String)
/* set_value(Value) : void
/* get_value() : Value
/* get_value(int) : Value
/* get_value(int, int) : Value
/* evaluateAsBoolean() : boolean
/* evaluateAsBoolean(int) : boolean
/* evaluateAsBoolean(int, int) : boolean
/* get_decimal() : int
/* get_decimal(int) : int
/* get_decimal(int, int) : int

4.3.4 Assembler

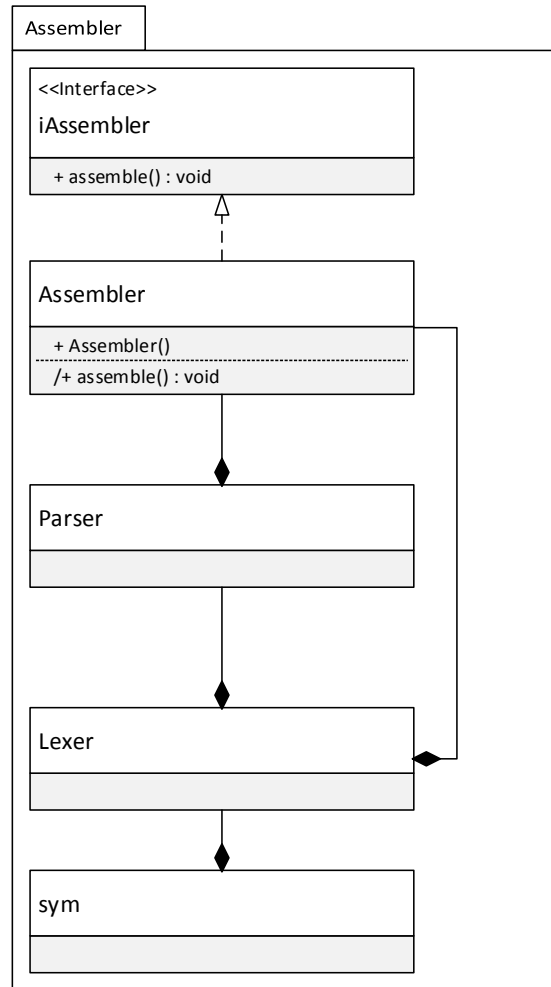


Figure 4-6: Assembler package

4.3.5 Parser

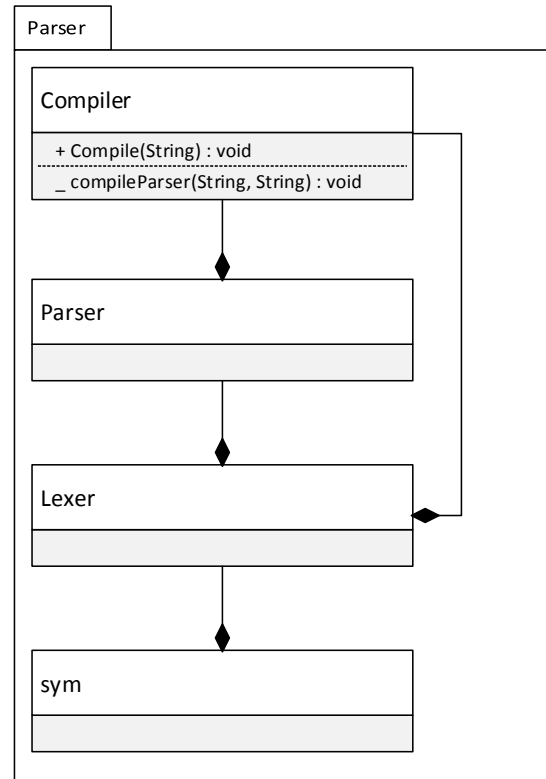


Figure 4-7: Parser package

4.3.6 GUI

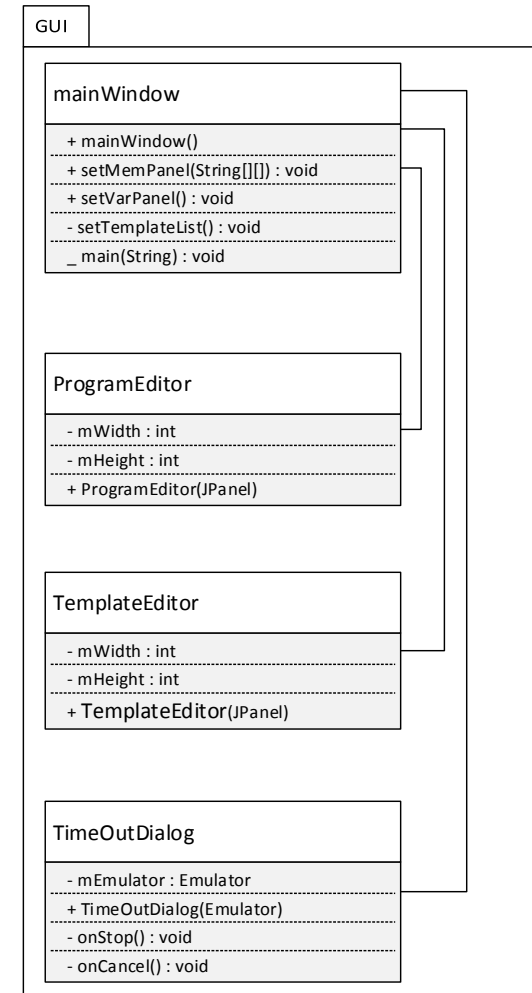


Figure 4-8: GUI package

4.4 Compilers

The system makes use of two compilers. One serves as the [instruction set](#) compiler and the other is the [assembler](#). The assembler is auto-generated by the instruction set compiler.

4.4.1 Instruction Set Compiler

The [instruction set](#) compiler has two functions:

- Generate an [assembler](#).
- Generate an [instruction set implementation](#).

To do that, the template file has two sections:

- Format: Defines the instruction format the assembler should follow.
- Code: Describes the code implementing the instruction set.

The template consists of a format followed by the code or vice versa.

4.4.1.1 Symbols

While parsing, the template is divided into symbols (tokens). Each symbol has a definition. Some symbols have values.

4.4.1.1.1 Keywords

- FORMAT: "format"
- ACCESSMODES: "access_modes"
- TAG: "<TAG>", "<LABEL>" or "<VAR>"
- OPCODE: "opcode"
- AM: " <AM>" or " <A_MODE>"
- CODE: "code"
- END: "end"
- HLT: "hlt"

4.4.1.1.2 Component Instructions

- COMPLEMENT: ".cmp"
- INCREMENT: ".inc"
- CHANGED: ".chn"
- CLEAR: ".clr"
- SET: ".set"

4.4.1.1.3 Assignment Operators

- A_ASSIGN: "<-"
- E_ASSIGN: "="
- F_ASSIGN: "=>"

4.4.1.1.4 String Tokens

- STRINGVAL: A set of characters encased in double-quotes (").

4.4.1.1.5 Instruction Set Structure Commands

- CYCLE: An upper case letter 'T', only if followed by a number.
- DECIMAL: '#'
- IF: "if"
- AND: "&&"
- OR: "||"
- NOT: '!'

4.4.1.1.6 Brackets

- L_TRIANGULAR: '<'
- R_TRIANGULAR: '>'
- L_CURLY: '{'
- R_CURLY: '}'
- L_SQUARE: '['
- R_SQUARE: ']'
- R_BRACKET: '('
- L_BRACKET: ')'

4.4.1.1.7 Punctuation Marks

- COLON: ':'
- SEMICOLON: ';'
- COMMA: ','
- HYPHEN: '-'

4.4.1.1.8 ALU Tokens

- ALU_SUM: '+'
- ALU_SUB: '-'
- ALU_MULT: '*'
- ALU_DIV: '/'
- ALU_MOD: '%'
- ALU_EQ: "=="
- ALU_NE: "!="
- ALU_GR: '>'
- ALU_LS: '<'
- ALU_GE: ">="
- ALU_LE: "<="
- ALU_NOT: '~'
- ALU_AND: '&'
- ALU_OR: '|'
- ALU_XOR: '^'
- ALU_L_SHIFT: "<<"
- ALU_R_SHIFT: ">>"
- ALU_FILL_ZERO: "(0)"
- ALU_FILL_ONE: "(1)"

4.4.1.1.9 Identifiers

Identifiers have values.

- ID: Uppercase letter followed by any number of uppercase letters and/or digits.
- NUMBER: A set of at least one digit, followed by any number of digits.

4.4.1.1.10 System Tokens

- EOF: The end of the file was reached.
- error: A token did not match any of the known tokens.

4.4.1.2 Format Grammar

format \rightarrow "format" "{" access_modes command_format_list "}"

access_modes \rightarrow "access_modes" "<" access_modes_list ">"

access_modes_list \rightarrow access_modes_list "," access_mode

access_modes_list \rightarrow access_mode

access_mode \rightarrow "[" ID "]" "=" STRINGVAL

access_mode \rightarrow "[" "]" "=" STRINGVAL

command_format_list \rightarrow command_format_list command_format

command_format_list \rightarrow command_format

command_format \rightarrow ID "=>" element_list

command_format \rightarrow ID "=>" STRINGVAL

element_list \rightarrow element_list element

element_list \rightarrow element

element \rightarrow "opcode" "<" NUMBER ">" "(" NUMBER ")"

element \rightarrow TAG

element \rightarrow AM

4.4.1.3 Code Grammar

epsilon \rightarrow ϵ

u_code \rightarrow "code" "{" command_list "}"

command_list \rightarrow command_list command

command_list \rightarrow command

command \rightarrow "T" NUMBER "(" condition_list ")" : u_op_list ";"

condition_list → "(" condition_list ")"

condition_list → condition_list "&&" condition_list

condition_list → condition_list "||" condition_list

condition_list → "!" condition_list

condition_list → condition

condition → ID range

condition → ID ".chn"

condition → "opcode" "(" NUMBER ")"

u_op_list → u_op_list "," u_op

u_op_list → u_op

u_op → ID component_command

u_op → "if" "(" condition_list ")" "{" u_op_list "}"

u_op → alu_command

u_op → move

u_op → assign_op_code

u_op → "end"

u_op → "hlt"

component_command → ".cmp"

component_command → ".inc"

component_command → ".clr"

component_command → ".set"

move → ID "[" NUMBER "-" NUMBER "]" "<-" ID range

move → ID "[" NUMBER "]" "<-" ID range

move → ID "<-" ID "[" NUMBER "-" NUMBER "]"

move → ID "<-" ID "[" NUMBER "]"

move → ID "<-" ID

move → ID "<-" alu_command

assign_op_code → "opcode" "=" "#" ID range

alu_command → "<" ID ":" ID alu_b_op ID ">"

alu_command → "<" ID ":" NUMBER ":" ID alu_o_op ">"

alu_command → "<" ID ":" NUMBER ":" alu_o_op ID ">"

alu_command → "<" ID ":" NUMBER ":" ID alu_shift_side alu_shift_filler NUMBER ">"

alu_b_op → "=="

alu_b_op → "!="

alu_b_op → ">"

alu_b_op → "<"

alu_b_op → ">="

alu_b_op → "<="

alu_b_op → "&"

alu_b_op → "|"

alu_b_op → "^"

alu_b_op → "+"

alu_b_op → "-"

alu_b_op → "*"

alu_b_op → "/"

alu_b_op → "%"

alu_o_op → "~"

alu_shift_side → "<<"

alu_shift_side → ">>"

alu_shift_filler → (0)

alu_shift_filler → (1)

range → "[" NUMBER "-" NUMBER "]"

range → "[" NUMBER "]"

range → epsilon

4.4.2 Assembler

The [assembler](#) is used to translate assembly programs into [pseudo-binary executables](#). During parsing, the assembler generates a map of labels to their addresses, and translates the commands into pseudo-binary. If a command uses a label, that command's is saved as a pseudo-binary OpCode with a special label identifier. Once the whole program has been parsed, all label identifiers are translated into their pseudo-binary addresses. Parts of the assembler are auto-generated by the [instruction set compiler](#).

4.4.2.1 Symbols

While parsing, the template is divided into symbols (tokens). Each symbol has a definition. Some symbols have values.

4.4.2.1.1 Keywords

- ORG: "ORG"
- END: "END"
- HEX: "HEX"
- DEC: "DEC"
- BIN: "BIN"

4.4.2.1.2 Punctuation Marks

- COMMA: ','
- NEGATIVE: '-'
- NEWLINE: '\n', '\r' or '\r\n'

4.4.2.1.3 Identifiers

Identifiers have values.

- ID: A letter or underscore followed by any number of letters, digits and/or underscores.
- NUMBER: There are three definitions for a number
 - Decimal: A set of at least one digit, followed by any number of digits.
 - Binary: A set of one or more zeroes and ones.
 - Hexadecimal: A zero followed by one or more digits and/or uppercase letters in the range of 'A' to 'F'.

4.4.2.1.4 System Tokens

- EOF: The end of the file was reached.
- error: A token did not match any of the known tokens.

4.4.2.2 *Constant Grammar*

Constant grammar remains the same for any given [instruction set](#).

epsilon $\rightarrow \epsilon$

program \rightarrow command_list "END"

command_list \rightarrow command_list command_line

command_list \rightarrow command_line

command_line \rightarrow tag_declaration operation NEWLINE

command_line \rightarrow tag_declaration var_declaration NEWLINE

command_line \rightarrow var_declaration NEWLINE

command_line \rightarrow operation NEWLINE

command_line \rightarrow "ORG" NUMBER NEWLINE

command_line \rightarrow NEWLINE

tag_declaration \rightarrow ID ", "

var_declaration \rightarrow number

number \rightarrow "HEX" NUMBER

number \rightarrow "HEX" "-" NUMBER

number \rightarrow "DEC" NUMBER

number \rightarrow "DEC" "-" NUMBER

number \rightarrow "BIN" NUMBER

4.4.2.3 *Generated Grammar*

This grammar changes whenever an instruction set is [generated](#). For each instruction set, a new list defining legal IDs and their values is generated.

access_mode \rightarrow ID

access_mode \rightarrow epsilon

operation \rightarrow ID ID access_mode

operation \rightarrow ID

5 Development Environment

5.1 Programming Paradigm

The system is built using Object-Oriented Programming. The system is composed of various classes, each with its own purpose. OOP was chosen for its ability to generate objects with specific behavior. The lowest level of implemented classes mimics the behavior of real hardware [components](#), and are then used together to construct more complex objects (such as a [processor](#)). The system simulates real connections between the different components within the CPU.

The division into different packages allows for changes to be done relatively easily. The GUI, for example, can be completely redesigned and reattached to the system. A simpler system can be generated by disconnecting the [instruction set generator](#). In that case, the default [instruction set](#) and [assembler](#) will be used.

5.2 Programming Language

The system was built using the Java programming language for two main reasons:

5.2.1 Platform Portability

The system is designed for students. As such, it may need to run on different platforms and operating systems. Both of the alternatives that were considered, C++ and C#, are much less portable. C++ for the need of a different compilation for each platform and C# for not having a well adapted framework for operation systems other than Windows.

5.2.2 Dynamic Class Loading

C++ operates much closer to the hardware level and has less abstraction levels. While that makes hardware simulation much easier, C++ is a fully compiled language, thus forcing the system to restart after every change done to the code. Since one of the core capabilities of the system is the ability to change its behavior during runtime, the inability to reload classes dynamically becomes very problematic.

5.3 System Limitations

The system, like any other, has some limitations. Some of them due to the way the system was designed and built, and some are inherent from the software domain.

5.3.1 Architectural Restrictions

As currently implemented, the system is restricted by its architecture. There are three main restrictions:

5.3.1.1 Static Architecture

While the [instruction set](#) can be customized, it is still bound by the [components](#) used in the basic [Mano architecture](#) (with an added temporary register). Adding new components cannot be done easily and requires editing the Java code of the emulator.

5.3.1.2 Available Component Types

Some components were not modeled because they were not needed to emulate the Mano CPU architecture. One example is the 'Partial Register', which is a register that represents a segment of a larger register (used in the x86 and other architecture designs). Other examples can be decoders and multiplexers.

5.3.1.3 No Advanced Features

Features like branch prediction, command pipeline, out of order execution and other advanced features commonly used in modern processors cannot be implemented because of the restrictive architecture.

5.3.2 User Interface

At this point, the user interface supports most of the system's functionality, but it has some known flaws:

- There is no I/O device emulation in the GUI (Supported by the system).
- The GUI is not completely multi-threaded, preventing some features from working in parallel at the same time.
- The GUI was designed for functionality purposes, and does not provide the best user experience possible.

5.4 Tools

5.4.1 IDE

The whole system was implemented, built and tested using the Community Edition of JetBrains' IntelliJ IDEA v12. It was chosen because it integrates code auto-completion, static error notification and linting, automatic module import functionality, basic module format generation, GUI generator, debugger and a JUnit testing module.

5.4.2 Diagrams

All diagrams were created using Microsoft's Visio 2013 Professional.

5.4.3 Compiler Generators

All compilers ([instruction set compiler](#) and [assembler](#)), pre-generated or auto-generated, are created using JFlex for the lexical analyzer and Java CUP for the syntactical analyzer. JFlex and Java CUP are Java implementations of the Lex/YACC compiler generation software.

5.5 Testing

Unit testing was done using JUnit. Full system testing was done by generating programs and instruction sets. In both cases legal input was introduced to prove correct functionality and illegal input was provided to assure error reporting and system recovery were functioning correctly.

6 User Guide

6.1 Error Handling

6.1.1 Syntax Errors

Like any other software language, the assembly language and the [template file](#) have their own syntax. If not followed perfectly, the text cannot be parsed into a functioning program or [instruction set](#). To address this problem, both the compiler and the assembler check the syntax and notify the user if any syntax errors were found. In that case, the operation is aborted and the system remains in the last functioning configuration.

6.1.2 System Crash

System crashes cannot be prevented if the source of the failure is external. To prevent loss of work, the system allows the user to save the work done so far. In addition, the system saves the last legal [memory content](#), program and template, and reloads them whenever they are needed.

6.1.3 Infinite Loops

A programmer who is not careful might create an infinite loop. After a set amount of cycles, if the program did not end, a notification will let the user decide whether to stop or keep running. This notification can trigger more than once.

6.1.4 Instruction Set-Assembler Mismatch

When changing the configuration of an instruction set, the user is not bound by the existing assembly language. Commands can be added, renamed or deleted. An assembly command unknown to the assembler cannot be used, even if correctly defined, because it cannot be added to the [executable file](#). To solve this problem, an [instruction set editor](#) can change both the instruction set and the assembler. A mismatch can still occur if the memory is not reloaded, leaving old assembly command representations, but avoiding these errors is under the responsibility of the user.

6.1.5 Missing Instruction Set

If the creation of an instruction set fails, the action is aborted and the last working instruction set remains active. If the instruction set is missing when trying to reload a previously used instruction set, the system will load a default configuration. In both cases a notification will be given to the user.

6.2 Workflow Continuity

When [generating a new instruction set](#), actual Java code is generated and compiled. To load the newly compiled code, the system needs to be restarted, which has a bad effect on the continuity of the user's workflow. To address this problem, the system uses Java's ability to dynamically load new parts of the code that were not loaded when the system was initialized. In this case, some of the dynamically loaded code did not exist before the system generated it during that same run.

When editing an existing instruction set, a system restart might still be needed. In that case, the user will be notified.

6.3 Environment Requirements

The system was implemented using the Java programming language, thus requiring a JVM to be installed on the user's environment. Furthermore, once a user has reached the skill level allowing them to become an instruction set editor, the use of the instruction set generator will generate and compile new Java code. To make that possible, the user will need a workable JDK, and an operating system (OS) configured to allow that JDK to be accessed directly from the command prompt or terminal.

6.4 Mano Basic Architecture

The system, as currently implemented, simulates the [architecture](#) described by M. Morris Mano in his book 'Computer System Architecture', with one added TR register. The new register is named TR1, and the name of the temporary register was changed from TR to TR 0. To allow for backwards compatibility, both TR0 and TR address the same register.

6.4.1 Architecture Components

6.4.1.1 ALU

The [ALU](#) is the arithmetic and logic unit. The ALU performs arithmetical or logical operations on a given input.

6.4.1.2 Memory

The [memory](#) is the storage unit. It holds both data and commands.

6.4.1.3 Bus

The [bus](#) is a data channel, used to transfer data from one [component](#) to another.

6.4.1.4 Flags

[Flags](#) are 1-bit [components](#), usually used to note the occurrence of an event or a state of the system. The flags are:

- E: "End Carry" flag. Indicates if the operation performed by the ALU had a carry.
- R: "Interrupt Request" flag. Indicates that an interrupt is waiting to be handled.
- S: "CPU Start" flag. Indicates that the CPU is [executing](#) a program.
- I: "Indirect" flag. Indicates that a given label's value should be used as a pointer.
- IEN: "Interrupts Enable" flag. Indicates that the system handles interrupts.
- FGI: "Flag Input". Indicates that new input was introduced to the CPU.
- FGO: "Flag Output". Indicates that the output device is ready to receive output.

6.4.1.5 Registers

[Registers](#) are collections of bits that store data. The registers are:

- AC (16 bits): "Accumulator". The accumulator holds the results of the [ALU](#). It is the only register controlled directly by the user. It holds the second operand of the ALU.
- DR (16 bits): "Data Register". The data register holds the first operand of the ALU.
- IR (16 bits): "Instruction Register". Holds the command while it is being decoded.
- TR0 (16 bits): "Temporary Register 0". Used for miscellaneous actions. Same as TR.
- TR1 (16 bits): "Temporary Register 1". Used for miscellaneous actions.

- PC (12 bits): "Program counter". Holds the address of the next command to be performed.
- AR (12 bits): "Address Register". Used as the index to access the memory.
- INPR (8 bits): "Input Register". Holds one byte of data received from an input device.
- OUTF (8 bits): "Output Register". Holds one byte of data to be sent to an output device.

6.4.2 Architecture Structure

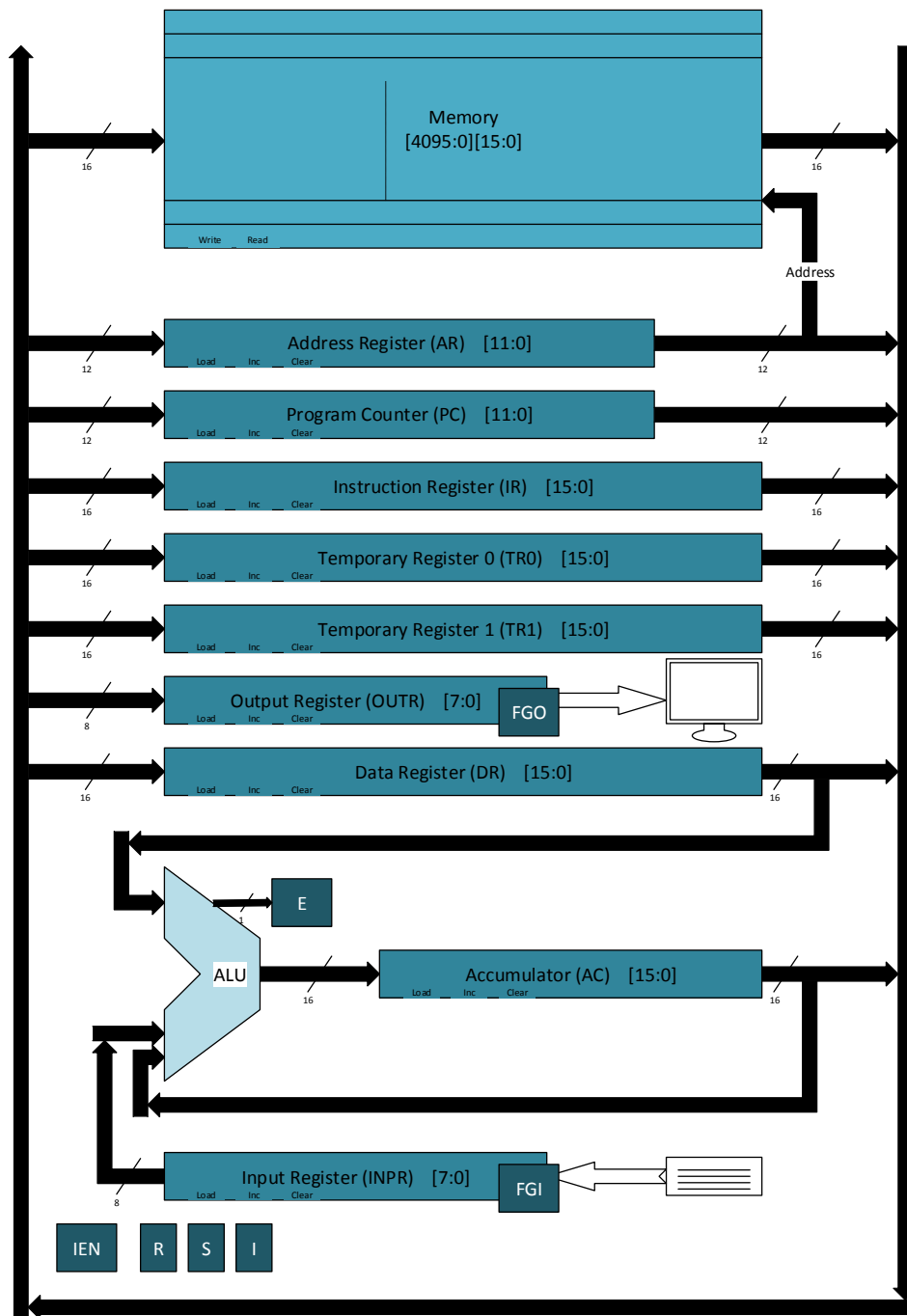


Figure 6-1: The M. Morris Mano CPU architecture

6.4.3 Basic Syntax and Commands

Like most programming and scripting languages, whitespaces are ignored.

6.4.3.1 Memory-Referencing Instructions

Memory-referencing instructions are commands that access the [memory](#). These commands use a label to declare the address accessed. An MRI command can be followed by the 'Indirect' option. In that case, the label's value will be used as a pointer.

6.4.3.1.1 AND <LABEL> [I]

Performs a bitwise AND operation between a value from the memory and the accumulator. The result is stored in the accumulator.

6.4.3.1.2 ADD <LABEL> [I]

Sums a value from the memory into the accumulator.

6.4.3.1.3 LDA <LABEL> [I]

Load accumulator. Loads a value from the memory to the accumulator.

6.4.3.1.4 STA <LABEL> [I]

Store accumulator. Stores the value of the accumulator in the memory. Does not affect the accumulator.

6.4.3.1.5 BUN <LABEL> [I]

Branch unconditional. Changes the address of the next command to be performed. Does not affect the accumulator.

6.4.3.1.6 BSA <LABEL> [I]

Branch and store address. Saves the value of the program counter to the memory and branches to the address following the given address. Does not affect the accumulator.

6.4.3.1.7 ISZ <LABEL> [I]

Increment and skip if zero. Increments a value in the memory and skips the next command if the updated value is equal to zero. Does not affect the accumulator.

6.4.3.2 Non-Memory-Referencing Instructions

Non-memory-referencing instructions affect the [system state](#) without accessing the memory.

6.4.3.2.1 CLA

Clear accumulator. Sets the value in the accumulator to zero.

6.4.3.2.2 CLE

Clear end carry. Sets the value of the end carry flag to zero.

6.4.3.2.3 CMA

Complement accumulator. Performs a bitwise NOT operation on the accumulator.

6.4.3.2.4 CME

Complement end carry. Performs a NOT operation on the end carry flag.

6.4.3.2.5 CIR

Circulate right. Moves the value of each bit in the accumulator to the bit located at its right. The rightmost bit is moved to the end carry flag and the value of the end carry flag is moved to the leftmost bit of the accumulator.

6.4.3.2.6 CIL

Circulate left. Moves the value of each bit in the accumulator to the bit located at its left. The leftmost bit is moved to the end carry flag and the value of the end carry flag is moved to the rightmost bit of the accumulator.

6.4.3.2.7 INC

Increment. Increases the value of the accumulator by one.

6.4.3.2.8 SPA

Skip if positive accumulator. Skips the next command if the value in the accumulator is positive.

6.4.3.2.9 SNA

Skip if negative accumulator. Skips the next command if the value in the accumulator is negative. Defined by 2's complement.

6.4.3.2.10 SZA

Skip if zero accumulator. Skips the next command if the value in the accumulator is zero.

6.4.3.2.11 SZE

Skip if zero end carry. Skips the next command if the end carry flag is set to zero.

6.4.3.2.12 HLT

Halt. Stops the [program execution](#).

6.4.3.2.13 INP

Input. Moves one byte of data from an input device into the accumulator.

6.4.3.2.14 OUT

Output. Moves one byte of data (LSB) from the accumulator to an output device.

6.4.3.2.15 SKI

Skip if input. Skips the next command if the input flag indicates that data from an input device was sent to the CPU.

6.4.3.2.16 SKO

Skip if output. Skips the next command if the output flag indicates that an output device is ready to receive data.

6.4.3.2.17 ION

Interrupts on. Turns on the interrupts enable flag.

6.4.3.2.18 IOF

Interrupts off. Turns off the interrupts enable flag.

6.5 Instruction Set Template Guide

The [instruction set template](#) is used to generate new [instruction sets](#). The template was created to bypass the need for users to write Java code whenever they wish to change the instruction set. The use of the template enables full usability of the system for users unfamiliar with Java on one hand, and prevents code exposure on the other.

Whenever an instruction set changes, the [assembler](#) needs to be changed as well. If the assembler remained the same, new commands would not be assembled and edited or removed commands would be assembled into wrong [pseudo-binary](#) representations, causing unexpected results. To address this problem, the template file is divided into two sections: [Format](#) and [Code](#). The format section defines the way a command should be represented in its pseudo-binary form. The code section defines the [micro-code](#) that implements the instruction set. The order in which the sections appear in the template is not important, but both sections must be written.

Like most programming and scripting languages, whitespaces are ignored.

6.5.1 Format Syntax

The format section is coded inside the format block. The format block is identified by the keyword "format" followed by a block encased in curly brackets ({ }):

```
format{  
    .  
    .  
    .  
}
```

The first part of the format section describes the different existing access modes. The access modes description is represented by the "access_modes" keyword, followed by a triangular brackets (< >). Within the triangular brackets each access mode is defined by its name, encased in square brackets, the equals sign and a string representing its value. Empty square brackets represent the default state, where no access mode is used. Different access modes are separated by commas.

```
format{  
    access_modes < [I] = "1", [] = "0" >  
    .  
    .  
    .  
}
```

In this example, an access mode represented by the symbol "I" will be evaluated as "1", and no access mode will be evaluated to "0".

The commands' format can be represented in two ways. Each format belongs to a command type.

6.5.1.1 Memory-Referencing Instructions

[Memory-referencing instructions](#) are commands that access the [memory](#). These commands use a label to declare the address accessed. An MRI command can be followed by an access mode. In that case, the label's value will be used according to the declared access mode.

The assembly format is always the same: <COMMAND> <LABEL> [ACCESS_MODE].

The MRI command format lets us decide the order in which those fields will be joined to create the pseudo-binary representation of the given instruction. The command will be replaced by the operation code. The label and access mode will use special tags.

An operation code is represented by the keyword "opcode", the number of bits it occupies in triangular brackets and its value in parentheses. A label is represented by one of these tags: "<LABEL>", "<TAG>" or "<VAR>". The tags are interchangeable for the user's convenience and have no syntactical difference. An access mode is represented by the "<AM>" or "<A_MODE>" tags. These, too, are interchangeable.

```
format{
    .
    .
    LDA => <AM> opcode<3>(2) <LABEL>
    .
    .
}
```

This example shows how the [LDA](#) command is declared. The label and access mode assembly syntax is always the same, so writing them next to the command name is redundant. This line indicates that if the assembler encounters a command named LDA, it should put the access mode as the first bit, then put the OpCode value 2, translated into a 3-bit binary value, and then add the address of the label. If the program contains the line: [LDA X I](#), and the address of X is 011101001101, it will be translated to: [1010011101001101](#).

6.5.1.2 Non-Memory-Referencing Instructions

[Non-memory-referencing](#) instructions affect the [system](#) without accessing the memory, so there is no need for a label or an access method, forcing the translation to always be the same. To take advantage of this, non-MRI command can be defined using a direct translation from the command name to its value. For easier usability, the translation is written using a string with the hexadecimal value of the command.

```
format{
    .
    .
    CMA => "7200"
    .
    .
}
```

In this example we see how the [CMA](#) command is translated directly to 7200₁₆, which has the binary value of 0111001000000000.

6.5.2 Code Syntax

Like the [format](#) section, the code section is coded inside the code block. The code block is identified by the keyword "code" followed by a block encased in curly brackets ({ }):

```
code{
    .
    .
    .
}
```

Each line in the code block defines a group of [micro-operations](#) done in a specific [instruction cycle](#), given a set of conditions are met. The conditions in each line should be mutually exclusive, so no two groups are [executed](#) on the same cycle. Failing to do so is not defined and may result in unexpected results, as the order in which the micro-operations are executed is not deterministic. All the micro-operations within a given group are considered to be executed in parallel. The order of the lines' positions is not important, as all the lines should be mutually exclusive. If a line is identical for several assembly commands, it should be written only once. The lines are not tied directly to any specific command. Grouping lines belonging to a specific command can be done for convenience, but it has no bearing on the resulting [instruction set implementation](#).

Each line starts by declaring the instruction cycle and the conditions for it to be activated. The instruction cycle is defined by an uppercase 'T' followed by a cycle number in the range of 0-15. The conditions appear in parentheses. Several conditions may appear with logical relations between them. A logical NOT can also be used. A condition can check if the currently executed OpCode matches a specific value by writing the keyword "opcode" followed by the numerical value in parentheses, check if a [flag](#) was set to '1' since the last time it was checked by writing the flag's name followed by ".chn" or check if the value of a [component](#) evaluates to "true" by writing that component's name. The value evaluates to "false" if it is zero, or to "true" in any other case. An evaluation of a specific bit or a range of bits can be done by adding the bit number or bits range (lower bit-higher bit) in square brackets.

```
code{
    .
    .
    T4(opcode(2)) ...
    T3(opcode(7) && !I && IR[5]) ...
    T12(IEN && (FGI.chn || FGO.chn)) ...
    .
    .
}
```

In this example, the first row is checked on instruction cycle number 4, and it checks if the OpCode is equal to 2. The second row is checked on instruction cycle number 3, and it checks if the OpCode is 7, the indirect flag is set to '0' and bit number 5 of the instruction

register is set to '1'. The last row is checked on instruction cycle number 12, and it checks if the IEN flag is set to '1' and one of the I/O flags was set to '1' since it was last checked.

The second part of the line defines the actions to be taken if the conditions are met. If more than one action should be executed, the actions are separated by a coma. The two parts are separated by a colon and the line should end with a semicolon. There are six different action types:

6.5.2.1 System Commands

System commands are commands for the CPU. There are two system commands:

- end: Ends the execution of a command. Resets the instruction timer back to 0 for the next command.
- hlt: Sets the CPU start flag to 0, thus terminating the program execution.

6.5.2.2 OpCode Assignment

This action decodes the bits representing the OpCode. It is done by writing the keyword "opcode", an assignment sign ('='), the "translate to decimal" sign ('#') and the name of the component holding the OpCode. A bit or a bits range can be added.

```
code{
    .
    .
    T2(!R): opcode = #IR[12-14] ...
    .
    .
}
```

This row sets the OpCode to the decimal value of bits 12-14 of the instruction register if the interrupt request flag is set to '0'.

6.5.2.3 Component Commands

These commands affect a specific component by writing its name followed by the desired action. The four component commands are:

- .cmp: Reverses the value of a flag. Has no effect on a register.
- .set: Sets the value of a flag to '1'. Has no effect on a register.
- .clr: Sets the value of the component to zero.
- .inc: Increases the value of a register by one. Has no effect on a flag.

```
code{
    .
    .
    T3(opcode(7) && !I && IR[8]): E.cmp, end;
    T2(R): PC.inc, IEN.clr, R.clr, end;
    .
    .
}
```

6.5.2.4 Move Commands

These commands move data from one component to another. This is done by applying the "move" operator ("<-") on two components. The data will move from the right hand operand to the left hand operand. The action can be performed on a single bit or a range of bits as well. If fewer bits than a component's capacity are moved, the extra bits will be filled with leading zeroes. If the component receiving the data has fewer bits than needed, the value will be truncated, losing the amount of extra bits (starting at the MSb).

```
code{
    .
    .
    T2(!R): opcode = #IR[12-14], AR<-IR[0-11], I<-IR[15];
    T3(opcode(7) && I && IR[11]): AC[0-7] <- INPR ...
    T4(opcode(2)): DR <- M;
    .
    .
}
```

Moving data from different components at the same time using the same bus might corrupt the data and should be avoided. A warning will be issued whenever two or more move commands appear in the same instruction cycle.

6.5.2.5 Conditional Actions

Some commands perform actions only if a condition is met. To implement commands such as [SZE](#) or [SPA](#), a conditional execution is needed. Adding a conditional execution of the micro-operations, is possible using the "if" operation. The "if" operation includes a condition clause and a commands block. The condition clause follows the same rules as the conditions for the instruction cycle, and the commands block follow the same rules as the instruction cycle actions. The commands block is encased in curly brackets ({ }).

```
code{
    .
    .
    T3( ... ): if(!E) { PC.inc }, end;
    T3( ... ): if(!AC[15] && AC[0-14]) { PC.inc }, end;
    .
    .
}
```

There is no "else" block. Such functionality can be achieved using an "if" operation with the opposite condition or by using the "end" command within the commands block.

```
// If positive accumulator do nothing, else skip.
T3( ... ): if(!AC[15] && AC[0-14]) { end };
T4( ... ): PC.inc, end;
```

6.5.2.6 ALU Commands

ALU commands are commands performed by the [ALU](#). These commands use one or both inputs of the ALU, perform an arithmetic or logic operation and store the result in the ALU's output. If the command is given the names of components that are connected to the ALU inputs, data is moved automatically from those components to the ALU. The result is a value that can be used as a right hand operand of a ["move" command](#). If not moved to the accumulator or any other component, the result cannot be used, and it will be overwritten when another ALU command is performed. Different ALU commands may differ in syntax, depending on the number of operands used and the operation performed. The ALU commands are encased in triangular brackets (< >) and the operation appears between two back quotes (''). The first parameter of an ALU command is the ALU's ID followed by a colon.

6.5.2.6.1 Binary Commands

[Binary commands](#) use both operands. These commands appear after the ID. They are composed of two IDs, declaring the operands, with an operation sign in between them.

```
code{
    .
    .
    T5(opcode(0)): AC <- <ALU:AC `&` DR>, end;
    T5(opcode(1)): AC <- <ALU:AC `+` DR>, end;
    .
    .
}
```

The first line in this code example performs a bitwise AND operation between the data register and the accumulator, and moves the result to the accumulator. The second line sums the values of the data register and the accumulator in the same way.

6.5.2.6.2 Unary Commands

[Unary commands](#) use only one operand. To specify which of the inputs should be used, the second parameter is the input number (can be '0' or '1') followed by a colon as well. The operation is declared as an operation sign followed by the component ID on which the operation should be performed. While syntactically correct, naming a component that does not match the declared input will result in undefined behavior, returning a wrong result.

```
code{
    .
    .
    T3( ... ): AC <- <ALU:1: `~` AC>, end;
    .
    .
}
```

This line performs a bitwise NOT on 'Input 1' after moving the data from the accumulator.

6.5.2.6.3 Shift Commands

Shift commands are a special kind of unary commands. While they use one operand like any other unary command, they require extra parameters. The syntax for the shift commands is similar to that of binary commands with two main differences:

- Instead of a second operand, there is a number that defines the offset amount.
- Instead of an operation sign there is an operation expression that defines the side to which the bits will be shifted and the value used to fill missing bits. The value appears in parentheses and the whole expression is encased in back quotes.

```
code{
    .
    .
    T3( ... ): <ALU:1:AC `<<(0)` 1>, ...
    .
    .
}
```

The line in the example uses data from the accumulator through 'Input 1'. It shifts all bits to the left by an offset of 1, filling the missing bits with zeroes.

```
// Signed shift-right:
    T3( ... ): if(AC[15]) { <ALU:1:AC `>>(1)` 1>, end };
    T4( ... ): <ALU:1:AC `>>(0)` 1>, end;
```

The example above implements a signed shift-right (not part of Mano's instruction set). At instruction cycle number 3, if the sign bit is '1', a shift-right is performed using '1' as the filler, and the command is ended. If the sign bit is '0', cycle number 3 does nothing and cycle number 4 performs a shift-right filling the missing bits with a '0'.

6.6 User Control Panel

The [user control panel](#) is the graphic user interface that the user uses to monitor the program during [execution](#) or to load programs and [instruction sets](#).

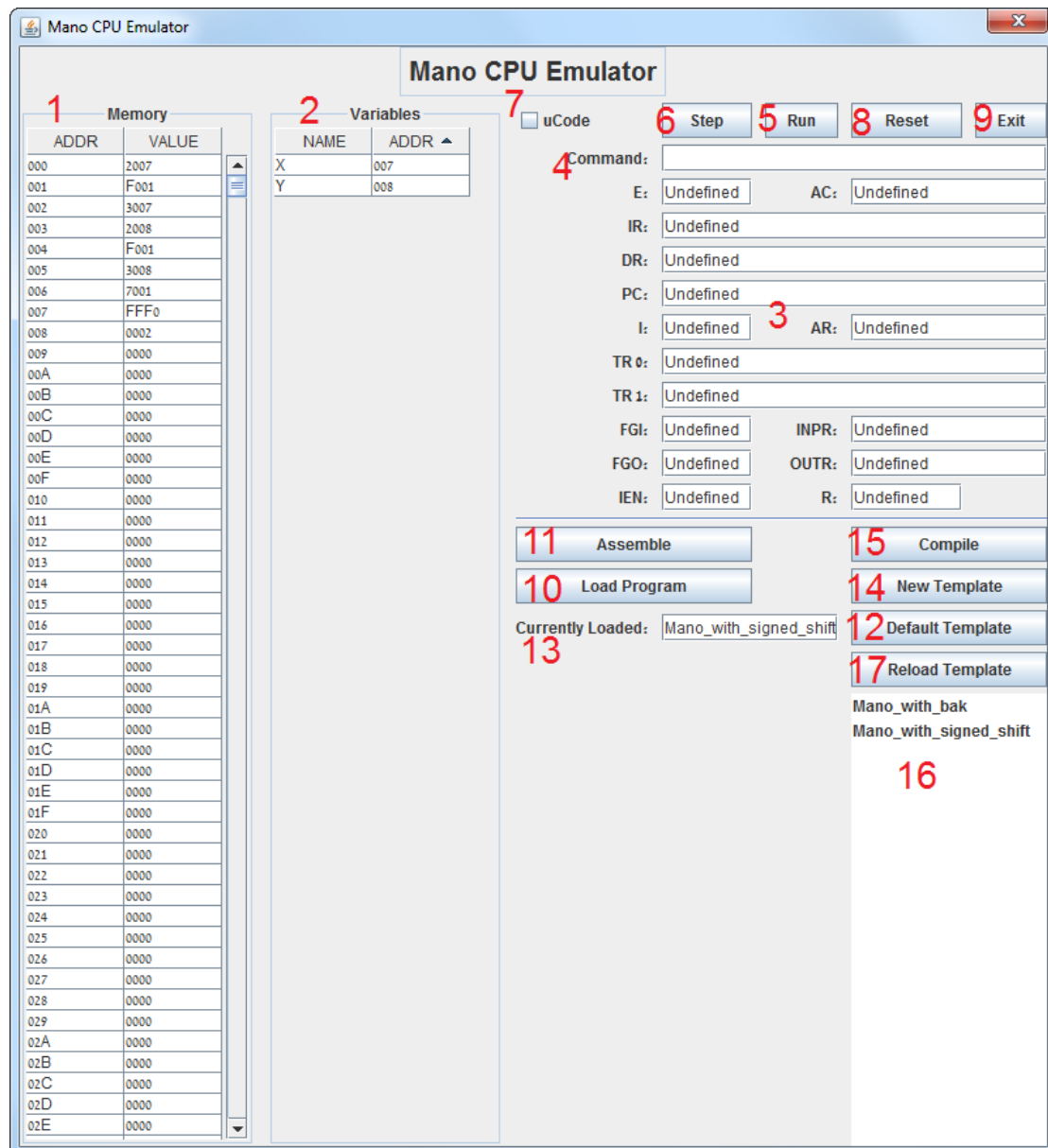


Figure 6-2: User control panel

6.6.1 Memory Panel

The memory panel shows the content of the memory in real time.

6.6.2 Variables Panel

The variables panel shows all the labels with their memory address.

6.6.3 System State Panel

The system state panel shows the [system state](#) in real time.

6.6.4 Command Panel

The command panel shows the command being [executed](#).

6.6.5 Run Button

The "Run" button executes the program.

6.6.6 Step Button

The "Step" button executes the next command.

6.6.7 uCode Checkbox

When checked, the "Step" button executes each [micro-operation](#) instead of each assembly command. The command panel changes accordingly.

6.6.8 Reset Button

The "Reset" button resets the system to its default values without unloading the program or the [instruction set](#).

6.6.9 Exit Button

The "Exit" button exits the system, closing all opened windows.

6.6.10 Load Program Button

The "Load Program" button opens the [program editor](#).

6.6.11 Assemble Button

The "Assemble" button assembles the program and loads it to the system's memory. It updates the variables panels and resets the system's state. If a syntax error is encountered, the memory and variables remain as they were.

6.6.12 Default Template Button

The "Default Template" button loads the default [instruction set](#) and [assembler](#) instead of the ones currently loaded.

6.6.13 Current Instruction Set Panel

The panel shows the name of the currently used instruction set.

6.6.14 New Template Button

The "New Template" button opens the [instruction set template editor](#).

6.6.15 Compile Button

The "Compile" button compiles the last [instruction set template](#) loaded. After compilation, a new instruction set implementation and assembler are generated. If an assembly command was removed from the instruction set, and that command was used in the last program loaded, trying to [assemble](#) the program will fail. If that command was previously assembled, it will remain in the system's memory. That command will either do nothing, or perform a different command if it fits its binary representation.

6.6.16 Instruction Set List

The instruction set list shows previously compiled instruction sets.

6.6.17 Reload Template Button

The "Reload Template" button reloads an instruction set and assembler selected from the instruction set list.

6.7 Program Editor

The [program editor](#) is used to load, save and edit programs.

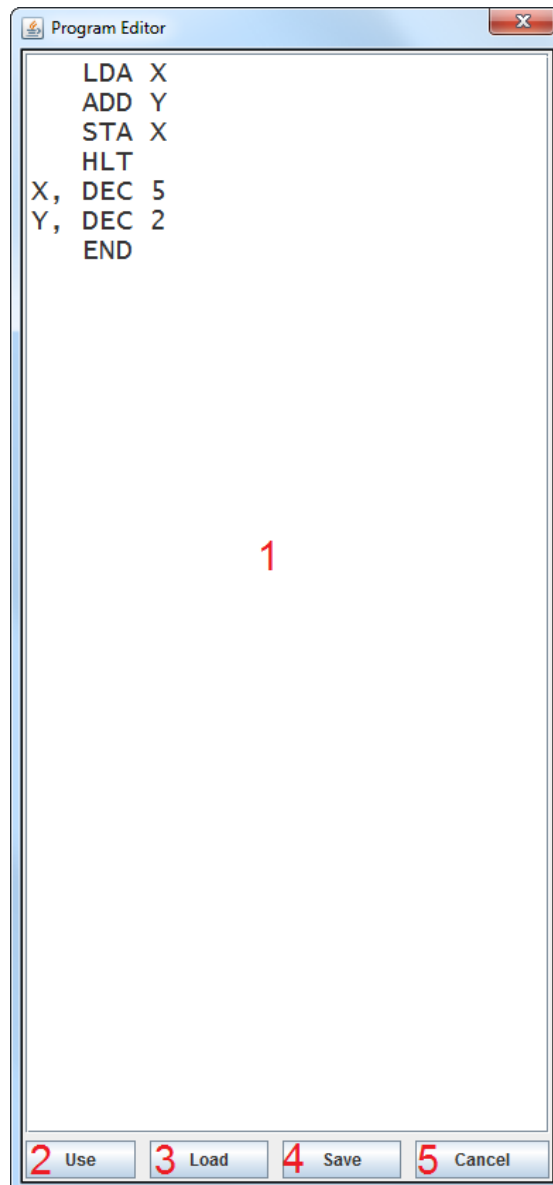


Figure 6-3: Program editor

6.7.1 Program Panel

The program panel is used to edit the program.

6.7.2 Use Button

The "Use" button closes the editor. The program is not assembled automatically.

6.7.3 Load Button

The "Load" button opens a loading dialog that allows for saved programs to be loaded.

6.7.4 Save Button

The "Save" button opens a saving dialog that allows the user to save programs.

6.7.5 Cancel Button

The "Cancel" button closes the editor without changing the program.

6.8 Instruction Set Template Editor

The [instruction set template editor](#) is used to load, save and edit instruction set templates.

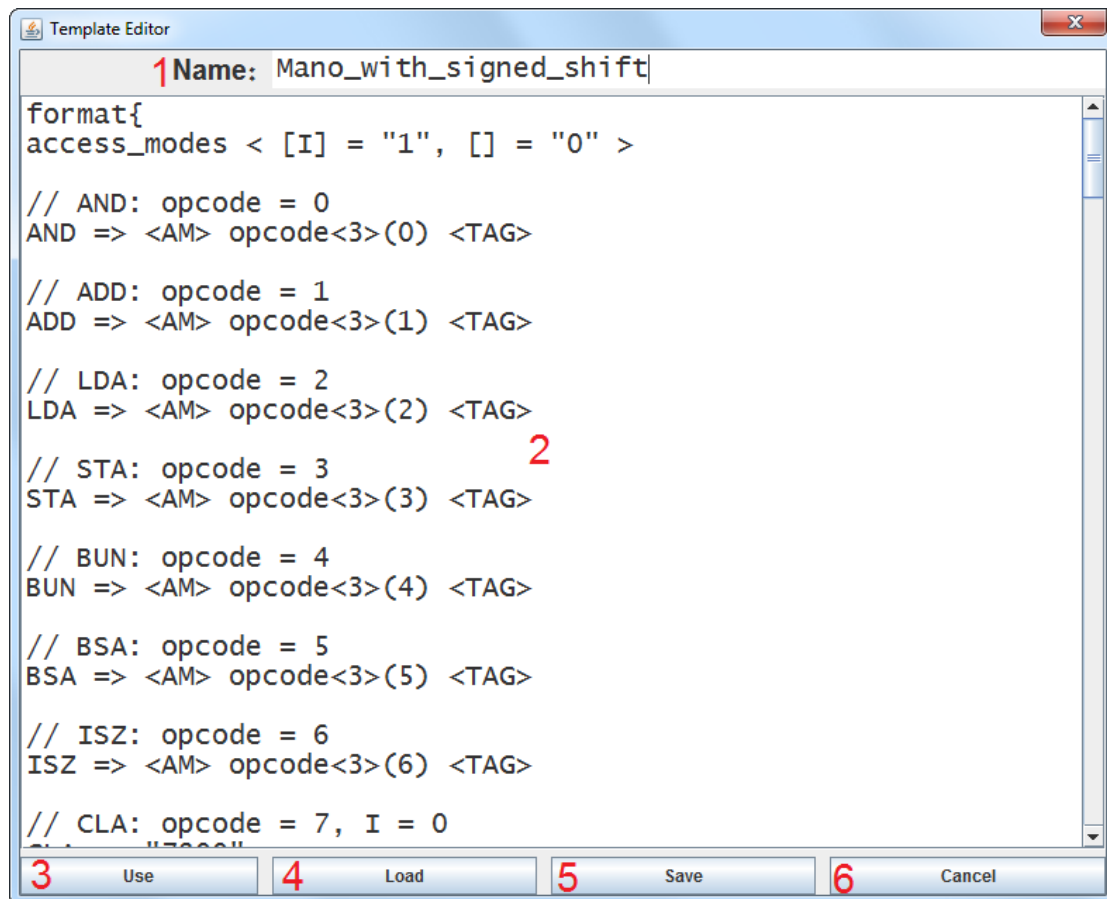


Figure 6-4: Instruction set template editor

6.8.1 Template Name

The template name is used to identify the different instruction sets. Once compiled, the name will be used to identify the files relevant for this instruction set. The name appears in the [instruction set list](#). When loading a template file, the file's name becomes the template name.

6.8.2 Template Panel

The template panel is used to edit the template.

6.8.3 Use Button

The "Use" button closes the editor. The template is not compiled automatically.

6.8.4 Load Button

The "Load" button opens a loading dialog that allows for saved templates to be loaded.

6.8.5 Save Button

The "Save" button opens a saving dialog that allows the user to save templates.

6.8.6 Cancel Button

The "Cancel" button closes the editor without changing the template.

6.9 Timeout Dialog

The timeout dialog appears if the program did not end within a specified amount of [executed](#) commands.

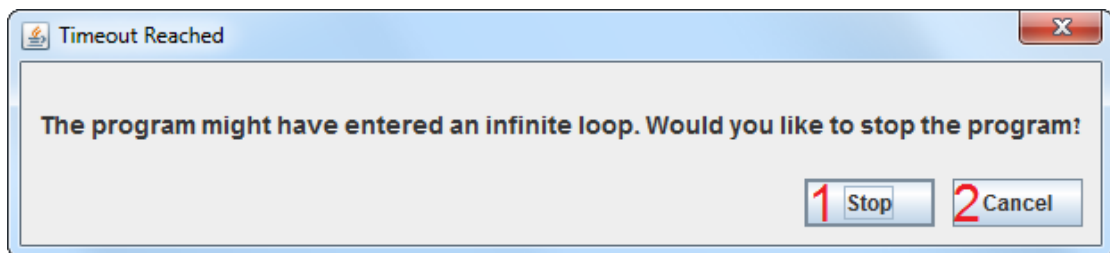


Figure 6-5: Timeout dialog alert

6.9.1 Stop Button

The "Stop" button terminates the program without waiting for it to finish properly.

6.9.2 Cancel Button

The "Cancel" button closes the timeout dialog and resumes the program. This should be used only if the programmer believes that the program should run for a long time by design. Cancelling the dialog will reset the timeout counter, resulting in the timeout dialog showing again once the timeout is reached for the second time. This dialog will keep popping until the program finishes properly or terminated.

7 Summary

7.1 Main Focal Points

The project revolves around three main focal points:

7.1.1 CPU Architecture and Functionality

The project's main objective is to give the students a better understanding of what [components](#) compose a CPU and how those different components interact with one another to create a functioning computer processor. This project allows the students to experiment beyond the basic CPU boundaries by enabling them to change the interactions between components and experience the results of those changes. On the other hand, the system teaches the students the limitations of the architecture and how to consider them carefully, as any change may upset the delicate balance within such a complex system.

7.1.2 Software-Hardware Modeling

To enable capabilities not present in existing systems, a more detailed modeling had to be accomplished. Existing systems use predetermined procedures to perform previously-known operations, rendering a fully modeled hardware unnecessary. Since this system is much more open ended, allowing the user to create new scenarios at a lower level, the modeling needs to remove some abstraction levels as well. Having registers as the lowest building blocks, for example, is not enough when allowing the user to edit the system at the bit level.

7.1.3 Scalability

Several good ideas were conceived during the brainstorming sessions at the early stages of the project's planning. Some of those ideas remained outside of the project's scope, but there are real intentions to develop them in the [future](#). To facilitate future development, the system's infrastructure was carefully developed. While not fully used, some tools and features were designed and integrated into the system for future utilization. An example for such a tool is the data transfer map. With unchangeable [architecture](#), the component connectivity could have been hardcoded, but that would cripple the system if a changeable architecture were to be implemented.

7.2 Conclusions

7.2.1 Good Planning is Key for Success

When developing the system, a considerable amount of time was used for the project's planning stage. Even though some unforeseen problems did arise during code development, most of the problems were solved during the planning phase, reducing the effort and time used dramatically.

7.2.2 Good Tools can Make the Difference

While implementing the system without using Java CUP, IntelliJ IDEA, JUnit or Visio could be done, the amount of effort would increase dramatically and the quality of the final result would be considerably lower.

7.3 Future Work

7.3.1 Multithreaded Implementation

As currently implemented, some functionality that should work using multiple threads in parallel does not do so. An example of this would be the [micro-operations execution](#). Multiple micro-operations executed in the same [instruction cycle](#) should be performed in parallel to better imitate the functionality of the hardware. Currently they are executed sequentially.

7.3.2 Editable Architecture

One of the initial goals of the system was to enable editable [architecture](#) in addition to an editable [instruction set](#). That goal proved to be greater than the outcome possible within the scope of this project and was postponed for future implementation. While this feature was not implemented, it is mostly supported by the system's infrastructure. Most of the work in the development of this feature will revolve around the implementation of a subsystem that generates a new CPU architecture. Such a subsystem would be very similar to the [instruction set generator](#).

7.3.3 Move to the Cloud

The system currently runs locally on the user's machine. This is problematic for several reasons:

- The system uses a file system that uses various files and directories, whose structure must be carefully maintained.
- The system generates and compiles Java code, forcing the user's operation system to support Java development.
- The system needs to be totally portable between different platforms.

The easiest solution to all those problems is to set up a server running the system with a web based user interface. Each user could use login credentials to access his or her work.

7.3.4 Instruction Set Generation Tool

A graphic tool that can be used to generate instruction sets would be much more user friendly than the currently used template based tool. Such a tool would prevent a significant amount of syntax and logic errors made by users. It could also introduce some default values, saving time in needless rewriting of common code.

8 References

8.1 Research

- [92] Mano M. M., "Computer System Architecture 3rd Edition", Prentice Hall, 1992.
- [10] Dr. Hoffner Y., "Computer Structure & Programming", Shenkar College, 2010.
- [10] Dr. Hoffner Y., "Computer Architecture", Shenkar College, 2010.
- [11] Dr. Hoffner Y., "CPU Design", Shenkar College, 2011.
- [13] Dr. Hoffner Y., "Mano_v.20.2 Simulator", Shenkar College, 2013.
- [13] "[Computer Architecture](#)", Wikipedia, 2013.
- [13] "[Mano Machine](#)", Wikipedia, 2013.
- [13] "[Hardware Description Language](#)", Wikipedia, 2013.
- [13] "[Verilog](#)", Wikipedia, 2013.
- [13] "[Register Transfer Level](#)", Wikipedia, 2013.
- [13] "[Register Transfer Language](#)", Wikipedia, 2013.
- [13] Intel Corporation, CMPG Dept, 2011-2013.

8.2 Technical References

- [10] Dr. Shichman M., "Object Oriented Programming", Shenkar College, 2010.
- [11] Dr. Shichman M., "Object Oriented Design", Shenkar College, 2011.
- [11] Michael H., "Java Programming", Shenkar College, 2011.
- [12] Pessach G., "Compilation Theory ", Shenkar College, 2012.
- [12] Nudler Y., "Computer Network and Telecommunication", Shenkar College, 2012.
- [13] [Stackoverflow.com](#), Stack Exchange, 2013.
- [13] Intel Corporation, CMPG Dept, 2011-2013.

9 Appendix A

9.1 Template for the Basic Mano Instruction Set

This [template](#) defines the [instruction set](#) as originally described by M. Morris Mano:

```
format {
// Access modes:
    access_modes < [I] = "1", [] = "0" >

// MRI format definitions:
    // AND: opcode = 0
    AND => <AM> opcode<3>(0) <VAR>
    // ADD: opcode = 1
    ADD => <AM> opcode<3>(1) <VAR>
    // LDA: opcode = 2
    LDA => <AM> opcode<3>(2) <VAR>
    // STA: opcode = 3
    STA => <AM> opcode<3>(3) <VAR>
    // BUN: opcode = 4
    BUN => <AM> opcode<3>(4) <LABEL>
    // BSA: opcode = 5
    BSA => <AM> opcode<3>(5) <LABEL>
    // ISZ: opcode = 6
    ISZ => <AM> opcode<3>(6) <VAR>

// Non-MRI format definitions:
    // CLA: opcode = 7, I = 0, Extended opcode bit = 11
    CLA => "7800"
    // CLE: opcode = 7, I = 0, Extended opcode bit = 10
    CLE => "7400"
    // CMA: opcode = 7, I = 0, Extended opcode bit = 9
    CMA => "7200"
    // CME: opcode = 7, I = 0, Extended opcode bit = 8
    CME => "7100"
    // CIR: opcode = 7, I = 0, Extended opcode bit = 7
    CIR => "7080"
    // CIL: opcode = 7, I = 0, Extended opcode bit = 6
    CIL => "7040"
    // INC: opcode = 7, I = 0, Extended opcode bit = 5
    INC => "7020"
    // SPA: opcode = 7, I = 0, Extended opcode bit = 4
    SPA => "7010"
    // SNA: opcode = 7, I = 0, Extended opcode bit = 3
    SNA => "7008"
    // SZA: opcode = 7, I = 0, Extended opcode bit = 2
    SZA => "7004"
    // SZE: opcode = 7, I = 0, Extended opcode bit = 1
    SZE => "7002"
    // HLT: opcode = 7, I = 0, Extended opcode bit = 0
    HLT => "7001"
    // INP: opcode = 7, I = 1, Extended opcode bit = 11
    INP => "F800"
    // OUT: opcode = 7, I = 1, Extended opcode bit = 10
    OUT => "F400"
    // SKI: opcode = 7, I = 1, Extended opcode bit = 9
    SKI => "F200"
    // SKO: opcode = 7, I = 1, Extended opcode bit = 8
    SKO => "F100"
    // ION: opcode = 7, I = 1, Extended opcode bit = 7
    ION => "F080"
    // IOF: opcode = 7, I = 1, Extended opcode bit = 6
    IOF => "F040"
}
```

```

code {
// Interrupts handling:
T0(R): AR.clr, TR <- PC;
T1(R): M <- TR, PC.clr;
T2(R): PC.inc, IEN.clr, R.clr, end;

T3(IEN && (FGI.chn || FGO.chn)): R.set;
T4(IEN && (FGI.chn || FGO.chn)): R.set;
T5(IEN && (FGI.chn || FGO.chn)): R.set;
T6(IEN && (FGI.chn || FGO.chn)): R.set;
T7(IEN && (FGI.chn || FGO.chn)): R.set;
T8(IEN && (FGI.chn || FGO.chn)): R.set;
T9(IEN && (FGI.chn || FGO.chn)): R.set;
T10(IEN && (FGI.chn || FGO.chn)): R.set;
T11(IEN && (FGI.chn || FGO.chn)): R.set;
T12(IEN && (FGI.chn || FGO.chn)): R.set;
T13(IEN && (FGI.chn || FGO.chn)): R.set;
T14(IEN && (FGI.chn || FGO.chn)): R.set;
T15(IEN && (FGI.chn || FGO.chn)): R.set;

// Fetch & decode:
T0(!R): AR <- PC;
T1(!R): IR <- M, PC.inc;
T2(!R): opcode = #IR[12-14], AR <- IR[0-11], I <- IR[15];
T3(!(opcode(7) && I): AR <- M;

// Memory-Reference code definitions:
// AND: Bitwise AND
T4(opcode(0)): DR <- M;
T5(opcode(0)): AC <- <ALU:AC `&` DR>, end;
// ADD: Sum
T4(opcode(1)): DR <- M;
T5(opcode(1)): AC <- <ALU:AC `+` DR>, end;
// LDA: Load to AC
T4(opcode(2)): DR <- M;
T5(opcode(2)): AC <- DR, end;
// STA: Store AC
T4(opcode(3)): M <- AC, end;
// BUN: Branch Unconditional
T4(opcode(4)): PC <- AR, end;
// BSA: Branch and Store Address
T4(opcode(5)): M <- PC, AR.inc;
T5(opcode(5)): PC <- AR, end;
// ISZ: Increment and Skip if Zero
T4(opcode(6)): DR <- M;
T5(opcode(6)): DR.inc;
T6(opcode(6)): M <- DR, if(!DR) { PC.inc }, end;

// Register-Reference code definitions:
// CLA: Clear AC
T3(opcode(7) && !I && IR[11]): AC.clr, end;
// CLE: Clear E
T3(opcode(7) && !I && IR[10]): E.clr, end;
// CMA: Complement AC
T3(opcode(7) && !I && IR[9]): AC <- <ALU:1:~`AC>, end;
// CME: Complement E
T3(opcode(7) && !I && IR[8]): E.cmp, end;
// CIR: Circulate Right
T3(opcode(7) && !I && IR[7]): <ALU:1:AC`>>(0)`1>,AC[15]<-E,E<-AC[0],end;
// CIL: Circulate Left
T3(opcode(7) && !I && IR[6]): <ALU:1:AC`<<(0)`1>,AC[0]<-E,E<-AC[15],end;
// INC: Increment
T3(opcode(7) && !I && IR[5]): AC.inc, end;
// SPA: Skip if Positive AC
T3(opcode(7) && !I && IR[4]): if(!AC[15] && AC[0-14]) { PC.inc }, end;

```

```
// SNA: Skip if Negative AC
T3(opcode(7) && !I && IR[3]): if(AC[15]) { PC.inc }, end;
// SZA: Skip if Zero AC
T3(opcode(7) && !I && IR[2]): if(!AC) { PC.inc }, end;
// SZE: Skip if Zero E
T3(opcode(7) && !I && IR[1]): if(!E) { PC.inc }, end;
// HLT: Halt
T3(opcode(7) && !I && IR[0]): hlt;

// Input-Output code definitions:
// INP: Input
T3(opcode(7) && I && IR[11]): AC[0-7] <- INPR, FGI.clr, end;
// OUT: Output
T3(opcode(7) && I && IR[10]): OUTR <- AC[0-7], FGO.clr, end;
// SKI: Skip if FGI
T3(opcode(7) && I && IR[9]): if(FGI) { PC.inc }, end;
// SKO: Skip if FGO
T3(opcode(7) && I && IR[8]): if(FGO) { PC.inc }, end;
// ION: Interrupts On
T3(opcode(7) && I && IR[7]): IEN.set, end;
// IOF: Interrupts Off
T3(opcode(7) && I && IR[6]): IEN.clr, end;
}
```


10 Appendix B

10.1 Useful Links

These are links to some of the tools used during development:

10.1.1 Java CUP

<http://www2.cs.tum.edu/projects/cup/>

10.1.2 IntelliJ IDEA

<http://www.jetbrains.com/idea/>

10.1.3 Visio

<http://office.microsoft.com/en-us/visio/>

10.1.4 Sublime Text

<http://www.sublimetext.com>

10.1.5 WinMerge

<http://winmerge.org>