Software Requirements Specification

For

Mano CPU Architecture Emulator with Customable Instruction Set

Version 2.1

Created by:
Yuval Tzur

Table of Contents

Revision History

Name	**Date**	**Reason For Changes**	**Version**
Yuval Tzur	25/10/12	First draft	1.0
Yuval Tzur	10/2/13	Adapted design to actual implementation	2.0
Yuval Tzur	22/2/13	Adapted architecture to actual implementation	2.1

1. Introduction

1.1 Purpose

This SRS describes the software functional and nonfunctional requirements for release 1.0 of the Mano CPU Architecture Emulator with a Customable Instruction Set. This document is intended to be used by the project developer that will implement and verify the correct functioning of the system. Unless otherwise noted, all requirements specified here are of high priority and committed for release 1.0.

1.2 Project Scope and Product Features

The Mano CPU Emulator will be a learning tool that will allow students to emulate the execution of programs written in assembly code under the Mano CPU architecture the same way the currently used emulator enables. In addition, the new emulator will introduce a compiler that creates new instruction sets when provided an instruction set specification file.

2. Overall Description

2.1 Product Perspective

The Mano CPU Emulator will introduce new features that add capabilities not covered by the currently used emulator. The ability to create new instruction sets as well as writing and executing assembly code that uses the new commands will help students with more advanced courses that currently have no interactive learning tools. When fully implemented, the emulator will help teach the thought process needed when creating an instruction set, and the importance of key elements such as micro-code timing and order, correct command layout and user (Programmer) usability.

2.2 User Classes and Characteristics

The Mano CPU Emulator will have three main user classes:

Professor	The course professor will provide the students with the emulator, as well as an instruction set. The professor can create several interchangeable instruction sets, each with a different purpose (Basic instruction set, advanced instruction set, SISC instruction set, and more).
Programmer	The programmer is a student that uses the emulator to write and execute programs written in assembly code. The emulator interface will allow the programmer to follow the program execution by showing the state of the CPU elements, the memory content and the system state in real time, on a micro-instruction by micro-instruction basis.
Instruction Set Writer	The instruction set writer is a more advanced student that understands the inner workings of the CPU architecture. The instruction set writer will create an instruction set specification file that defines all the instructions, their op-code and the set of micro-instructions executed by each instruction. The micro-instructions will be defined by a set of limited actions applicable to the different elements in the CPU. By extent, the instruction set will also define the instruction format.

2.3 Operating Environment

The Mano CPU Emulator will work in three main environments:

Professor's PC	The professor will have a copy of the emulator on his personal computer. The professor will use his copy to prepare for his lectures as well as executing home assignments for grading purposes.
Student's PC	The students will receive a copy each. Students will use the emulator to aid them with the home assignments. The emulator will also be used for the end of semester project.
Classroom	The emulator will be available in some of the classrooms. Using the classroom copies will allow the students to execute programs during class and ask for the professor's help in real time.

2.4 Design and Implementation Constraints

HW-SW Compatibility	The emulator is implemented in software, but it emulates hardware components. While software is more versatile and more structured, it should enforce some of the hardware constraints. This impacts the way data is transferred between the different components of the CPU, the range of supported micro-operations, synchronous and asynchronous signal handling, etc.

Specification File Template	The emulator requires a compiler that converts specification files into Java code. The specification file template should be clearly and carefully defined to allow maximum usability while adhering to the strict limitations of hardware component capabilities.
Mano Emulator Compatibility	The emulator should be able to fully implement the original Mano instruction set. There should be full backwards compatibility with all programs written using the currently used emulator. All programs should be executed exactly as they would in the current emulator version as far as the programmer is concerned.
Mano Architecture Compatibility	The emulator should run under the Mano architecture. No components can be added or removed, and all connectivity between components must be preserved. Adding or removing any component, as well as altering the order in which the components are connected to one another will change the functionality of the emulated CPU.
Easily Exchangeable Source Files	Each instruction set is converted into Java code that uses a special API created for the CPU. Whenever a new instruction set is introduced, a new Java source file is created. A system that replaces the source file and updates the executable file seamlessly is a must.
Conventional Platforms	While the emulator will use some extended tools such as Java Cup (A LEX/YACC tool for Java), the final executable must use standard packages and libraries only. Although the professor has direct control over his personal computer and can request for special installation in the classrooms, the students should not be expected to install third party applications and cannot be bound to a specific OS.

2.5 User Documentation

Programmer Interface	The emulator will have a complete and detailed help document that fully explains the programmer interface. The help document will cover register and flag content, memory content, system output screen, system on-screen keyboard, the micro-operation box and the built-in editor.
CPU API	A CPU API document will list the different components and the usability of each component (Purpose and functionality).
Specification File Template	A document will be provided that describes the specification file template for new instruction sets. The document will describe the operations available for each component type and the correct syntax for that operation, as well as global instruction definitions.
Original Mano Instruction Set	The original Mano instruction set will be the system's default, and a guide will be provided so new programmers will be able to start writing assembly programs without a full understanding of instruction set customization.

2.6 Assumptions and Dependencies

JVM	The emulator will be written in Java and the assumption is that anyone using it will have an up-to-date JVM installed.
Assembly Knowledge	It is assumed that anyone using the emulator is familiar with assembly language. The emulator is used to write and execute assembly programs, but it can't be used as the main source for assembly language learning.
Mano CPU Knowledge	It is assumed that whoever uses the Mano CPU emulator is familiar with the Mano architecture. No highly detailed document about the architecture itself will be provided.

Program Scale	It's assumed that as a learning tool, no big scale programs will run on the emulator. The purpose of the emulator is to allow the programmer to observe the actions and operations made by the CPU, not to emulate full-scale programs. The overhead might be too high to run big chunks of code.
Boolean Arithmetic Knowledge	As any "real" CPU, this emulator performs its operations on a bit-wise basis. It is assumed that any programmer or instruction set writer is comfortable with Boolean arithmetic and bit-wise operations such as bit-wise AND, bit-wise OR, complement, etc.

3. System Features

3.1 Assemble a Program

3.1.1 Description and Priority

Once an assembly program is loaded, the emulator should assemble it into machine code. The emulator will use the user defined instruction set and instruction structure to parse the program given in the editor, assemble it, and store the resulting machine code in the emulated machine's memory.
Priority: High.

3.1.2 Stimulus/Response Sequences

Stimulus:	A programmer selects to assemble a given program.
Response:	The emulator assembles the code and stores it in the emulated machine's memory.
Stimulus:	A programmer wishes to edit a program.
Response:	The editor will open and allow the assembly code to be edited.
Stimulus:	A programmer wishes to delete a program.
Response:	The emulated machine's memory will be cleared and the editor will show no assembly.

3.1.3 Functional Requirements

Editor – Open	Anyone using the emulator as a programmer can open the editor and read/write/edit/delete its content.
Editor – Save	When finished, the contents of the editor can be saved.
Editor – Cancel	If needed, the editor can be closed without saving any changes made to the content.
Assembler – Assemble	Once the programmer is finished, he or she can assemble the contents of the editor.
Assembler – Build Labels Table	The assembler uses the two pass assembly method. During the first pass, the assembler will create a table that holds all labels used by the program. A label can be used as a variable name, a subroutine name or a marker for a JMP statement. The table will hold the name and memory address for each label.
Assembler – Fail	The assembler will fail if the assembly code does not follow the instruction set syntax, a tag appears more than once or if a nonexistent command is used.
Assemble – Pass	The assembler will pass and assemble the program if it was correctly written.
Assembler – Error Message	When assembly fails, the assembler will produce an error message to notify the user.
Memory – Init Memory	Once the code has been assembled, the assembler will initialize the CPU's memory with the assembled code.

3.2 Execute a Program

3.2.1 Description and Priority

Once a program has been assembled and loaded to the memory, the emulator allows the programmer to emulate a full program execution. During the execution, the emulator will calculate the CPU's state on a micro-operation basis, and will update the programmer's interface accordingly.
Priority: High.

3.2.2 Stimulus/Response Sequences

Stimulus:	A programmer presses the "Start" button.
Response:	The emulator will start executing the program.
Stimulus:	A programmer changes execution speed.
Response:	The emulator increases delay between micro-operations execution.
Stimulus:	A programmer presses the "Step" button.
Response:	The emulator will execute one micro-operation and then will stop until further instructions from the programmer.

3.2.3 Functional Requirements

Emulator – Start	The emulator will start execution of micro-operations in a loop, and will continue until reaching the micro-operation that indicates the execution termination.
Emulator – Step	The emulator will execute one micro-operation and will update the CPU's state.
Emulator – Pause	The emulator will finish executing the micro-operation that is being executed and will not proceed to the next micro-operation until further instructions from the programmer.
Emulator – Stop	The emulator will reset the CPU's state to its defaults. All data from the stopped execution will be lost.
Emulator – Set Execution Speed	The emulator will change the inter-operation delay value according to the selected execution speed. The higher the delay, the slower is the execution.
CPU – Execute	The CPU will execute one micro-operation and change its state accordingly.
CPU – Stop	Once a micro-operation sets the "Stop" flag to true, the CPU will no longer execute any operations until the emulator resets.
CPU – Reset	Resets all the CPU's elements to their default values.
Interface – Update State	When the emulator completes the execution of a micro-operation, the programmer's interface will be updated with the new CPU state.

3.3 Compile a Template File

3.3.1 Description and Priority

When given a template file, the system will compile it into two source files:

- A source file that implements the instruction set emulation on the CPU, which will be compiled later on with the CPU source file to create the emulator.
- A source file that implements an assembler to be used by the emulator, which matches the new instruction set.

Priority: High.

3.3.2 Stimulus/Response Sequences

Stimulus:	An instruction set writer presses the "Create" button.
Response:	The system will compile the assembler, the instruction set and the emulator.

3.3.3 Functional Requirements

Compiler – Compile Instruction Set	One of the compiler's is the source code that implements the instruction set's operations. The new code will implement each operation by a set of micro-operations. The new code will be arranged by time cycles.
Compiler – Compile Assembler	In addition to the instruction set's source code, an assembler source code will be generated. The new code will define the hexadecimal values of the different assembly instructions and elements.

3.4 True Hardware Emulation

3.4.1 Description and Priority

When executing a program, the emulator will emulate a true hardware behavior. If a template file defines an instruction that sets the value of a register from different sources at the same time or more than one register sends data through the same bus at the same time, the emulator will use the data from one of the sources randomly as if a real decoder was used by the hardware to select a source.

Priority: Low.

3.4.2 Stimulus/Response Sequences

Stimulus:	A register is set with two different values at the same time.
Response:	Only one value will be set, ignoring the other value.

3.4.3 Functional Requirements

Emulator – True Hardware Setter	Each value transmission will be declared separately. If more than one transmission is made to the same component, one of them will override the rest and will be the only value to be set.

4. External Interface Requirements

4.1 User Interfaces

Assembler Editor: The assembler editor will allow the user to edit assembly programs. The editor will have the following options:

- Load – Loads a text file containing an assembly program.
- Save – Saves an assembly program into a file.
- Assemble – Assembles the assembly program.
- Cancel – Closes the editor ignoring all changes.
- Edit – The editor allows the programmer to edit the program's content.

Programmer UI: The programmer UI will visualize the CPU state and the memory at any given time. The basic UI will show only relevant registers, the current instruction and the memory state. The advanced UI will also include the current micro-operation and the remaining registers. The UI will also include the following options:

- Edit Program – Opens the Assembler Editor.
- Start – Starts program execution.
- Stop – Resets program execution.
- Pause/Run – Toggles program execution between paused and running.
- Speed Select – Changes the delay time between executed instructions.

Instruction Editor: The instruction editor allows the instruction set writer to edit the instruction set. The editor will have the following options:

- Load – Loads a text file containing an instruction set template.
- Save – Saves an instruction set template into a file.
- Compile – Compiles the instruction set template.
- Cancel – Closes the editor ignoring all changes.
- Edit – The editor allows the instruction set writer to edit the instruction set's content.

4.2 Hardware Interfaces

No hardware interfaces needed.

4.3 Software Interfaces

CPU API: The CPU API defines the operations the emulator can execute on the emulated CPU. The API will be used by the compiler, which will compile an instruction set template into a set of API calls.

4.4 Communications Interfaces

No communication interfaces needed.

5. Other Nonfunctional Requirements

5.1 Performance Requirements

No critical performance requirements.

5.2 Safety Requirements

No safety requirements.

5.3 Security Requirements

No security requirements.

5.4 Software Quality Attributes

- The emulator must be compatible with the basic Mano CPU architecture and instruction set.
- The compiled assembler and instruction set must implement the exact same instruction set as defined by the instruction set template.

Appendix A: Design

1. CPU Data Structure

1.1 Constants

TYPE	NAME	VALUE	Comments
Boolean	_0	false	0 doesn't represent false in Java by default
Boolean	_1	true	1 doesn't represent true in Java by default
Integer	DATA_REGISTER_SIZE	16	16 bit data
Integer	ADDR_REGISTER_SIZE	12	12 bit address
Integer	IO_REGISTER_SIZE	8	8 bit characters
Integer	MEMORY_SIZE	$2^{\text{ADDR\_REGISTER\_SIZE}}$	Memory size defined by the max address value
Integer	BUS_SIZE	DATA_REGISTER_SIZE	Bus size is equal to the size of the data
Integer	DATA_COMPONENTS	22	Number of components
Integer	DATA_TABLE_SIZE	15	Number of components that hold data
Integer	TIMER_LIMIT	16	Number of max cycles per instruction
Integer	ALU	0	
Integer	ALU_IN0	1	
Integer	ALU_IN1	2	
Integer	ALU_OUT	3	
Integer	M	4	
Integer	BUS	5	
Integer	AR	6	
Integer	PC	7	
Integer	DR	8	
Integer	AC	9	
Integer	IR	10	
Integer	TR	11	
Integer	TR0	TR	TR0 is TR for backward compatibility
Integer	TR1	12	
Integer	INPR	13	
Integer	OUTR	14	
Integer	E	15	
Integer	R	16	
Integer	S	17	
Integer	I	18	
Integer	IEN	19	
Integer	FGI	20	
Integer	FGO	21	
Integer	TIMER	22	
Integer	UNREACHABLE	-99	Data transfer to target component is impossible
Integer	TARGET_REACHED	-1	Data transfer completed

1.2 Value

1.2.1 Overview

This class defines a binary value. To do so, the class uses a Boolean array that represents a set of Boolean digits. A value can be represented as a decimal value (Integer), a binary value (Boolean array or String) or an hexadecimal value (String).

1.2.2 Fields

- _size : int
- _content : boolean[]

1.2.3 Methods

- Value(size : int)
  Constructor. Defines the number of bits.
- Value(boolean[] value)
  Constructor. Sets a value from a Boolean array.
- Value(size : int, value : int)
  Constructor. Sets number of bits and the value from an integer.
- Value(value : String)
  Constructor. Sets the value from a hexadecimal value.
- set_content(boolean[] value) : void
  Sets a value from a Boolean array.
- set_content(value : int) : void
  Sets number of bits and the value from an integer.
- set_content(value : String) : void
  Sets the value from a hexadecimal value.
- set_content(value : Value) : void
  Copies the content of another Value.
- get_size() : int
  Gets the number of bits.
- boolean[] get_content()
  Gets the content as a binary value.
- get_decimal() : int
  Gets the content as a decimal value.
- get_hexadecimal() : String
  Gets the content as a hexadecimal value.
- boolean[] toBinary(size : int, value : int)
  Transforms a decimal value to a binary value.
- boolean[] toBinary(value : String)
  Transforms a hexadecimal value to a binary value.
- toDecimal(boolean[] value) : int
  Transforms a binary value to a decimal value.
- toDecimal(value : String) : int
  Transforms a hexadecimal value to a decimal value.
- toHexadecimal(boolean[] value) : String
  Transforms a binary value to a hexadecimal value.
- toHexadecimal(size : int, value : int) : String
  Transforms a decimal value to a hexadecimal value.

- toHexadecimal(value : String) : String
  Transforms a binary string to a hexadecimal string.
- toString() : String
  Gets the binary value as a string.

1.3 Component

1.3.1 Overview

This class defines a hardware component. Each class that emulates a hardware component will inherit from this class.
Abstract.

1.3.2 Fields

- _id : int
- _name : string

1.3.3 Methods

- setId(id : int) : void
  Sets the component's ID.
- setName(name : string) : void
  Sets the component's name.
- getId() : int
  Gets the component's ID.
- getName() : string
  Gets the component's name.

1.4 DataComponent

1.4.1 Overview

This class acts as an interface to all the components that can hold data. The common interface is needed to allow a components array to be used in the System class. The array holds all the data-holding components in the CPU and allows interaction with those components by using those component IDs as the array indices. This interface will be implemented in all data-holding components, but some of them will have no effect.
Interface.

1.4.2 Methods

- get_input0() : DataComponent
- get_input1() : DataComponent
- get_output() : DataComponent
- evaluateAsBoolean() : boolean
- evaluateAsBoolean(bitIndex : int) : boolean
- evaluateAsBoolean(bitStart : int, bitEnd : int) : boolean
- isWritable() : boolean
- isWritable0() : boolean
- isWritable1() : boolean
- changed() : boolean
- get_value() : Value
- get_value(index : int) : Value
- get_value(start : int, end : int) : Value
- get_row(address : int) : Value
- set_value(value : Value) : void

- set_value(index : int, value : Value) : void
- set_value(start : int, end : int, value : Value) : void
- enableWrite() : void
- enableWrite0() : void
- enableWrite1() : void
- disableWrite() : void
- disableWrite0() : void
- disableWrite1() : void
- update() : void
- clear() : void
- increment() : void
- set() : void
- passThrough0() : void
- passThrough1() : void
- sum() : void
- subtract() : void
- multiply() : void
- divide() : void
- modulo() : void
- and() : void
- or() : void
- xor() : void
- equal() : void
- notEqual() : void
- greaterThan() : void
- lessThan() : void
- greaterOrEqual() : void
- lessOrEqual() : void
- complement() : void
- complement0() : void
- complement1() : void
- shiftLeft0(numOfBits : int, filler : boolean) : void
- shiftLeft1(numOfBits : int, filler : boolean) : void
- shiftRight0(numOfBits : int, filler : boolean) : void
- shiftRight1(numOfBits : int, filler : boolean) : void
- get_decimal() : int
- get_decimal(bitIndex : int) : int
- get_decimal(bitStart : int, bitEnd : int) : int

1.5 Bus

1.5.1 Overview

This class defines a bus. Most of the components will be connected to one another through a bus. The bus size is defined by the data register size.
Inherits from Component, implements DataComponent.

1.5.2 Fields

- _value : Value

1.5.3 Methods

- Bus(id : int, name : String)
  Constructor.
- set_value(value : Value) : void
  Moves data to the bus.
- get_value() : Value
  Gets the value currently in the bus.
- get_row(address : int) : Value
  Gets the value currently in the bus. Address is ignored.
- get_value(start : int, end : int) : Value
  Gets the value currently in the bus by a given range.
- get_value(index : int) : Value
  Gets the value of a bit currently in the bus.
- evaluateAsBoolean() : boolean
  Evaluates if the bus' value is 0 or not.
- evaluateAsBoolean(bitIndex : int) : boolean
  Evaluates if the value of a bit in the bus.
- evaluateAsBoolean(bitStart : int, bitEnd : int) : boolean
  Evaluates if the range's value is 0 or not.
- get_decimal() : int
  Gets the decimal value of the bus.
- get_decimal(bitIndex : int) : int
  Gets the decimal value of a bit in the bus.
- get_decimal(bitStart : int, bitEnd : int) : int
  Gets the decimal value of a range in the bus.

1.6 Register

1.6.1 Overview

This interface defines the Register interface. A register can hold data, memory addresses or I/O data. A class will be defined for each of these registers types, using this interface.
Abstract, inherits from Component, implements DataComponent.

1.6.2 Fields

- _value : Value
- _buffer : Value
- writeEnable : boolean

1.6.3 Methods

- set_value(value : Value) : void
  Sets the register value.
- get_value() : Value
  Gets the register value.
- evaluateAsBoolean() : boolean
  Evaluates if the registers' value is 0 or not.
- evaluateAsBoolean(bitIndex : int) : boolean
  Evaluates if the bits' value is 0 or not.
- evaluateAsBoolean(bitStart : int, bitEnd : int) : boolean
  Evaluates if the ranges' value is 0 or not.
- get_decimal() : int
  Gets the decimal value of the register.
- get_decimal(bitIndex : int) : int
  Gets the decimal value of the bit.
- get_decimal(bitStart : int, bitEnd : int) : int
  Gets the decimal value of the range.
- enableWrite() : void
  Enables writing.
- disableWrite() : void
  Disables writing.
- isWritable() : boolean
  Checks if writing is enabled.
- clear() : void
  Sets the register's value to 0.
- increment() : void
  Increases the register's value by 1.
- update() : void
  Updates the register's value to the value in the buffer.
- set_value(index : int, value : Value) : void
  Sets the value of a bit in the register.
- get_value(index : int) : Value
  Gets the value of a bit in the register.
- set_value(start : int, end : int, value : Value) : void
  Sets the value of a range in the register.
- get_value(start : int, end : int) : Value
  Gets the value of a range in the register.

1.7 DataRegister

1.7.1 Overview

This class defines registers for data. Data can be an instruction or a value.
Inherits from Register.

1.7.2 Fields

Defined by Register.

1.7.3 Methods

Defined by Register.

- DataRegister(id : int, name : String)
  Constructor. Sets the value and buffer lengths to DATA_REGISTER_SIZE

1.8 AddressRegister

1.8.1 Overview

This class defines registers for memory addresses.
Inherits from Register.

1.8.2 Fields

Defined by Register.

1.8.3 Methods

Defined by Register.

- AddressRegister(id : int, name : String)
  Constructor. Sets the value and buffer lengths to ADDR_REGISTER_SIZE

1.9 IORegister

1.9.1 Overview

This class defines registers for input and output data.
Inherits from Register.

1.9.2 Fields

Defined by Register.

1.9.3 Methods

Defined by Register.

- IORegister(id : int, name : String)
  Constructor. Sets the value and buffer lengths to IO_REGISTER_SIZE

1.10 Flag

1.10.1 Overview

This class defines a flag. A flag will contain a 1-bit value of TRUE or FALSE. To allow a flag to be used for interrupts, it will also have a field that notifies if the value was set to TRUE since the last time it was checked.
Inherits from Component, implements DataComponent.

1.10.2 Fields

- _value : boolean
- _buffer : boolean
- _changed : boolean

1.10.3 Methods

- Flag(id : int, name : String)
  Constructor.
- set_value(value : Value) : void
  Sets the flag to to bit 0 of the value. Raises the 'changed flag'.
- get_value() : Value
  Gets the flag's value. Clears the 'changed flag'.
- evaluateAsBoolean() : boolean
  Evaluates if the flag value is true or false.
- get_decimal() : int
  Gets the decimal value of the flag (True = 1 and false = 0).
- set() : void
  Sets the flag to true. Raises the 'changed flag'.
- clear() : void
  Sets the flag to false. Clears the 'changed flag'.
- complement() : void
  Reverses the flag's value.
- changed() : boolean
  Checks if the 'changed flag' is raised.
- update() : void
  Updates the flag's value to the value in the buffer.

1.11 Memory

1.11.1 Overview

This class defines the system's memory. The memory is composed by Registers and holds all the instructions and variable or constant values the program needs to be executed. The memory size is defined as $2^{ADDR\_REG\_SIZE}$.
Inherits from Component, implements DataComponent.

1.11.2 Fields

- _memory : Value[MEMORY_SIZE]
- _buffer : Value[MEMORY_SIZE]
- _address : AddressRegister
- _writeEnable : boolean

1.11.3 Methods

- Memory(id : int, name : String, addressRegister : AddressRegister)
  Constructor. Sets a pointer to an address register that serves as the memory's index.
- set_value(value : Value) : void
  Sets the value of the memory slot indicated by the current value of the address register.
- get_value() : Value
  Gets the value of the memory slot indicated by the current value of the address register.
- get_row(address : int) : Value
  Gets the value of the memory slot indicated by the given value of the address register.
- get_value(start : int, end : int) : Value
  Gets the value of a range in the memory slot indicated by the current value of the address register.
- get_value(index : int) : Value
  Gets the value of a bit in the memory slot indicated by the current value of the address register.
- get_decimal() : int
  Gets the decimal value of the memory slot indicated by the current value of the address register.
- get_decimal(bitIndex : int) : int
  Gets the decimal value of a bit in the memory slot indicated by the current value of the address register.
- get_decimal(bitStart : int, bitEnd : int) : int
  Gets the decimal value of a range in the memory slot indicated by the current value of the address register.
- evaluateAsBoolean() : boolean
  Evaluates if the value of the memory slot indicated by the current value of the address register is 0.
- evaluateAsBoolean(bitIndex : int) : boolean
  Evaluates if the value of a bit in the memory slot indicated by the current value of the address register is 0.
- evaluateAsBoolean(bitStart : int, bitEnd : int) : boolean
  Evaluates if the value of a range in the memory slot indicated by the current value of the address register is 0.
- enableWrite() : void
  Enables writing.
- disableWrite() : void
  Disables writing.
- isWritable() : boolean
  Checks if writing is enabled.
- update() : void
  Updates the memory's value to the value in the buffer.

1.12 ALU

1.12.1 Overview

This class defines the ALU. The ALU will perform all the calculations and operations needed for the program to be executed. The ALU has two inputs for up to two operands needed for the operations. The outcome will be saved as the output.
Inherits from Component, implements DataComponent.

1.12.2 Input

This is an inner class that defines an input value to the ALU. It implements the same methods as a register, but it has no buffer and no master-slave functionality. Its size is the same as a data register.
Inherits from Component, implements DataComponent.

1.12.3 Output

This is an inner class that defines an output value of the ALU. It implements the same methods as a register, but it has no buffer and no master-slave functionality. Its size is the same as a data register. It is always writeable.
Inherits from Component, implements DataComponent.

1.12.4 Fields

- _input0 : Input
- _input1 : Input
- _output : Output
- _endCarry : Flag

1.12.5 Methods

- ALU(id : int, name : String, endCarry : Flag)
  Constructor. Sets a pointer to a flag that holds the end carry of relevant operations.
- get_value() : Value
  Gets the value of the output.
- get_input0() : DataComponent
  Gets a pointer to _input0.
- get_input1() : DataComponent
- Gets a pointer to _input1.
- get_output() : DataComponent
- Gets a pointer to _output.
- evaluateAsBoolean() : boolean
  Evaluates if the value of the output is 0.
- evaluateAsBoolean(bitIndex : int) : boolean
  Evaluates if the value of a bit in the output is 0.
- evaluateAsBoolean(bitStart : int, bitEnd : int) : boolean
  Evaluates if the value of a range in the output is 0.
- get_value(start : int, end : int) : Value
  Gets the value of a range in the output.
- get_value(index : int) : Value
  Gets the value of a bit in the output.

- get_decimal() : int
  Gets the decimal value of a range in the output.
- get_decimal(bitIndex : int) : int
  Gets the decimal value of a bit in the output.
- get_decimal(bitStart : int, bitEnd : int) : int
  Gets the decimal value of a range in the output.
- enableWrite0() : void
  Enables writing to _input0.
- enableWrite1() : void
  Enables writing to _input1.
- disableWrite0() : void
  Disables writing to _input0.
- disableWrite1() : void
  Disables writing to _input1.
- isWritable0() : boolean
  Checks if writing is enabled in _input0.
- isWritable1() : boolean
  Checks if writing is enabled in _input1.
- passThrough0() : void
  Moves the data from _input0 to the output.
- passThrough1() : void
  Moves the data from _input1 to the output.
- sum() : void
  Sums _input0 and _input1. The result is put in _output. Updates _ endCarry if needed.
- subtract() : void
  Subtracts _input1 from _input0. The result is put in _output.
- multiply() : void
  Multiplies _input0 by _input1. The result is put in _output.
- divide() : void
  Divides _input0 by _input1. The result is put in _output.
- modulo() : void
  Multiplies _input0 by _input1. The remainder is put in _output.
- and() : void
  Performs a bitwise AND between _input0 and _input1. The result is put in _output.
- or() : void
  Performs a bitwise OR between _input0 and _input1. The result is put in _output.
- xor() : void
  Performs a bitwise XOR between _input0 and _input1. The result is put in _output.
- complement0() : void
  Performs a bitwise NOT on _input0. The result is put in _output.
- complement1() : void
  Performs a bitwise NOT on _input1. The result is put in _output.
- equal() : void
  Checks if _input0 is equal to _input1. The result (0 or 1) is put in _output.
- notEqual() : void
  Checks if _input0 is not equal to _input1. The result (0 or 1) is put in _output.
- greaterThan() : void
  Checks if _input0 is greater than _input1. The result (0 or 1) is put in _output.
- lessThan() : void
  Checks if _input0 is less than _input1. The result (0 or 1) is put in _output.
- greaterOrEqual() : void
  Checks if _input0 is greater than or equal to _input1. The result (0 or 1) is put in _output.
- lessOrEqual() : void
  Checks if _input0 is less than or equal to _input1. The result (0 or 1) is put in _output.

- shiftLeft0(numOfBits : int, filler : boolean) : void
  Shifts the bits in _input0 to the left by the given number of bits. Empty bits are filled according to the given value. The result is put in _output.
- shiftLeft1(numOfBits : int, filler : boolean) : void
  Shifts the bits in _input1 to the left by the given number of bits. Empty bits are filled according to the given value. The result is put in _output.
- shiftRight0(numOfBits : int, filler : boolean) : void
  Shifts the bits in _input0 to the right by the given number of bits. Empty bits are filled according to the given value. The result is put in _output.
- shiftRight1(numOfBits : int, filler : boolean) : void
  Shifts the bits in _input1 to the right by the given number of bits. Empty bits are filled according to the given value. The result is put in _output.

1.13 InstructionTimer

1.13.1 Overview

This class defines the micro code timer that times the micro-instructions within each instruction.
Inherits from Component.

1.13.2 Fields

- _currentCycle : int
- _reset : boolean

1.13.3 Methods

- InstructionTimer(id : int, name : String)
  Constructor.
- get_currentCycle() : int
  Gets the current cicle.
- pulse() : void
  If the timer wasn't reset since the last cycle, goes to the next cycle. Goes to 0 after the number of cycles defined by TIMER_LIMIT.
- reset() : void
  Restarts the cycle count.

1.14 ProgramLine

1.14.1 Overview

This class defines one line of a program. Each line consists of the line's address and the line's content.

1.14.2 Fields

- _address : Value
- _content : Value

1.14.3 Methods

- ProgramLine(address : Value, content : Value)
  Constructor.
- get_address() : Value
  Gets the line's address.
- get_content() : Value
  Gets the line's content.

1.15 Program

1.15.1 Overview

This class defines a program as a set of program lines to be loaded to the memory.

1.15.2 Fields

- _lines : List<ProgramLine>

1.15.3 Methods

- addLine(programLine : ProgramLine) : void
  Adds a program line to the program.
- getLines() : ProgramLine[]
  Gets an array containing all the program lines.
- clear() : void
  Clears the program by deleting all lines.

1.16 DataTransferMap

1.16.1 Overview

The data transfer map is a table that defines the next step needed in order to transfer data from the current component into a target component. Each row will define the current component, the column will define the target component and their intersection will define the component to which the data will be transferred in the next cycle. Rows' and columns' numbers are used as components' IDs. Each entry will contain one of the following values: The next component ID, UNREACHABLE (If the data can't be transferred to the target component) or TARGET_REACHED (When a component intersects with itself, the current and target components are the same, therefore the data has reached the target). Some component names are aliases for other components. These components have several names with the same ID and occupy only one slot in the transfer map (For example, TR and TR0 are two names for the same register and have both the same ID). On other occasions, one name can identify more than one target. In that case, the transfer map entry intersections differ according to the location (For example, all rows marked as ALU refer to the ALU's output, while columns marked as ALU refer to either input 0 or input 1, depending on the source component being DR or AC/INPR).

1.16.2 Fields

- _map : int[][]

1.16.3 Methods

- DataTransferMap(String filename)
  Constructor. Builds a transfer map from a configuration file.
- nextInRoute(currID : int, targeted : int) : int
  Gets the source and target components and returns the components the data should be transferred to.

1.17 InstructionsUCode

1.17.1 Overview

This class defines the instruction set's micro operations. Each cycle has its own function. The function defines the actions to be taken by the system according to the system's state. The Java file implementing this class is generated by the compiler according to the template file. The current Mano CPU architecture supports 16-cycle operations, but the number of cycle functions is not limited and is defined by the instruction set writer.
Auto-Generated.

1.17.2 Fields

- _CPU : System
- _cycleDescription : String

1.17.3 Methods

- InstructionsUCode(System CPU)
  Constructor. Sets a pointer to the system used.
- t0 : void
  Defines actions to be taken by the system in cycle 0.
- t1 : void
  Defines actions to be taken by the system in cycle 1.
- t2 : void
  Defines actions to be taken by the system in cycle 2.
- t3 : void
  Defines actions to be taken by the system in cycle 3.
- t4 : void
  Defines actions to be taken by the system in cycle 4.
- t5 : void
  Defines actions to be taken by the system in cycle 5.
- t6 : void
  Defines actions to be taken by the system in cycle 6.
- t7 : void
  Defines actions to be taken by the system in cycle 7.
- t8 : void
  Defines actions to be taken by the system in cycle 8.
- t9 : void
  Defines actions to be taken by the system in cycle 9.
- t10 : void
  Defines actions to be taken by the system in cycle 10.
- t11 : void
  Defines actions to be taken by the system in cycle 11.
- t12 : void
  Defines actions to be taken by the system in cycle 12.
- t13 : void
  Defines actions to be taken by the system in cycle 13.
- t14 : void
  Defines actions to be taken by the system in cycle 14.
- t15 : void
  Defines actions to be taken by the system in cycle 15.

1.18 System

1.18.1 Overview

This class defines the system. The system will act as the CPU. The system will contain all the components and the connectivity between them. It will also provide the API used by the instruction set. The system provides the CPU's state, and has no direct effect on the micro operations executed in each cycle or their order.

1.18.2 Fields

- constantTable : HashMap<String, Integer>
- _instructionTimer : InstructionTimer
- _componentsList : DataComponent[]
- _transferMap : DataTransferMap
- _opCode : 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

1.18.3 Methods

- System()
  Constructor.
- set_opCode(opCode : int) : void
  Sets the current Op-Code
- loadProgram(program : Program) : void
  Loads a given program to the memory.
- resetTimer() : void
  Resets the cycle timer.
- halt() : void
  Ends the execution.
- checkOpCode(opCode : int) : boolean
  Checks if a given Op-Code is the current Op-Code.
- nextCycle() : void
  Moves the instruction timer to the next cycle.

- moveData(fromID : int, toID : int) : void
  Moves data from one component to another iteratively according to the transfer map.
- moveData(value : Value, toID : int) : void
  Sets the value of a certain component to the value given.

1.19 Emulator

1.19.1 Overview

This class acts as the emulator. It calls the cycle functions as long as the system is in execution mode (The S flag is on). The emulator also provides data to the GUI.

1.19.2 Fields

- _ManoCPU : System
- _uCode : InstructionsUCode
- _cycleDescription : String

1.19.3 Methods

- main(args : String[]) : void
  Main function. Runs the emulation.
- getProgram() : Program
  Loads a memory image that has been assembled from an assembly program.

2. Compilers

2.1 Template Compiler

The Template compiler is used to compile the template file into a working instruction set. The template file is composed of two parts, each with its own grammar: Format-Template Compiler and Micro-Code-Template Compiler.

2.2 Format-Template Compiler

2.2.1 Overview

The Format-Template compiler will compile the assembler according to the instruction set template file. The assembler needs to be compiled with every new instruction set, or else there would be no compatibility between the instruction set format and the assembled code format.

2.2.2 Format-Template Compiler Grammar

format → "format" "{" access_modes command_format_list "}"
access_modes → "access_modes" "<" access_modes_list ">"
access_modes_list → access_modes_list "," access_mode | access_mode
access_mode → "[" ID "]" "=" STRINGVAL | "[" "]" = STRINGVAL
command_format_list → command_format_list command_format | command_format
command_format → ID "=>" element_listl | ID "=>" STRINGVAL
element_list → element_listl element | element
element ::= "opcode" "<" NUMBER ">" "(" NUMBER ")" | TAG | AM

2.3 Micro-Code-Template Compiler

2.3.1 Overview

The Micro-Code-Template compiler will compile the emulator functionality (InstructionsUCode class implementation when running a program based on the instruction set.

2.3.2 System-Template Compiler Grammar

```
u_code → "code" "{" command_list "}"
command_list → command_list command | command
command → "T" NUMBER "(" condition_listl ")" ":" u_op_list ";"
condition_list → "(" condition_list ")" | condition_list "&&" condition_list | condition_list "||" condition_list |
                 "!" condition_list:cl | condition:c
condition → ID range | ID ".chn" | "opcode" "(" NUMBER ")"
u_op_list → u_op_list "," u_op:uo | u_op:uo
u_op → ID component_command | "if" "(" condition_list ")" "{" u_op_list "}" | alu_command | move:m |
       asign_op_code | "end" | "hlt"
component_command → ".cmp" | ".inc" | ".clr" | ".set"
move → ID "[" NUMBER "-" NUMBER "]" "<-" ID range | ID "[" NUMBER "]" "<-" ID range |
       ID "<-" ID "[" NUMBER "-" NUMBER "]" | ID "<-" ID "[" NUMBER "]" | ID "<-" ID | ID "<-" alu_command
asign_op_code → "opcode" "=" "#" ID range
alu_command → "<" ID ":" ID alu_b_op ID ">" | "<" ID:alu ":" NUMBER ":" ID alu_o_op ">" |
              "<" ID ":" NUMBER ":" alu_o_op ID ">" |
              "<" ID ":" NUMBER ":" ID alu_shift_side alu_shift_filler NUMBER ">"
alu_b_op → "`==`" | "`!=`" | "`>`" | "`<`" | "`>=`" | "`<=`" | "`&`" | "`|`" | "`^`" | "`+`" | "`-`" | "`*`" | "`/`" | "`%`"
alu_o_op → "`~`"
alu_shift_side → "`<<" | "`>>"
alu_shift_filler → "(0)`" | "(1)`"
range → "[" NUMBER "-" NUMBER "]" | "[" NUMBER "]" | ε
```

2.4 Assembler

2.4.1 Overview

The assembler will go through the assembly code written by the programmer, assembling it into machine code. The output is a binary (Hexadecimal) set of instructions that can be loaded into the system's memory. The assembler uses the two-pass method to assemble a program. The first pass will create a list of all tags used for variables and brunching as well as their addresses, and will assemble the code, leaving placeholders whenever a tag is encountered. The second pass will replace the placeholders with the memory addresses of the tags. The assembler grammar definition is generated by the Format-Template compiler.

2.4.2 Assembler Grammar

```
program → command_list "END"
command_list → command_list command_line | command_line
command_line → tag_declaration operation "\n" | tag_declaration var_declaration "\n" | operation "\n" |
               "ORG" NUMBER "\n"
tag_declaration → ID ","
var_declaration → "HEX" NUMBER | "DEC' NUMBER | "BIN" NUMBER
access_mode → ID | ε
operation → ID ID access_mode | ID
```

Appendix B: System Architecture

1. System Components

1.1 ALU (ALU)

1.1.1 Metadata

Type: ALU
ID: 0 (ALU_IN0 = 1; ALU_IN1 = 2; ALU_OUT = 3)
Size: ALU_IN0 = 16 bit; ALU_IN1 = 16 bit; ALU_OUT = 16 bit

1.1.2 Inputs

- Data Register (ALU_IN0)
- Accumulator (ALU_IN1)
- Input Register (ALU_IN1)

1.1.3 Outputs

- E flag (Direct)
- Accumulator (ALU_OUT)

1.2 Memory (M)

1.2.1 Metadata

Type: Memory
ID: 4
Size: 4096 × 16 bit

1.2.2 Inputs

- Bus
- Address Register (Address Index)

1.2.3 Outputs

- Bus

1.3 Bus (BUS)

1.3.1 Metadata

Type: Bus
ID: 5
Size: 16 bit

1.3.2 Inputs

- Memory
- Address Register
- Program Counter
- Data Register
- Accumulator
- Instruction Register
- Temporary Register 0
- Temporary Register 1

1.3.3 Outputs

- Memory
- Address Register
- Program Counter
- Data Register
- Instruction Register
- Temporary Register 0
- Temporary Register 1
- Output Register

1.4 Address Register (AR)

1.4.1 Metadata

Type: AddressRegister
ID: 6
Size: 12 bit

1.4.2 Inputs

- Bus

1.4.3 Outputs

- Bus

1.5 Program Counter (PC)

1.5.1 Metadata

Type: AddressRegister
ID: 7
Size: 12 bit

1.5.2 Inputs

- Bus

1.5.3 Outputs

- Bus

1.6 Data Register (DR)

1.6.1 Metadata

Type: DataRegister
ID: 8
Size: 16 bit

1.6.2 Inputs

- Bus

1.6.3 Outputs

- Bus
- ALU_IN0

1.7 Accumulator (AC)

1.7.1 Metadata

Type: DataRegister
ID: 9
Size: 16 bit

1.7.2 Inputs

- ALU_OUT

1.7.3 Outputs

- Bus
- ALU_IN1

1.8 Instruction Register (IR)

1.8.1 Metadata

Type: DataRegister
ID: 10
Size: 16 bit

1.8.2 Inputs

- Bus

1.8.3 Outputs

- Bus

1.9 Temporary Register 0 (TR0)

1.9.1 Metadata

Type: DataRegister
ID: 11 (TR = TR0)
Size: 16 bit

1.9.2 Inputs

- Bus

1.9.3 Outputs

- Bus

1.10 Temporary Register 1 (TR1)

1.10.1 Metadata

Type: DataRegister
ID: 12
Size: 16 bit

1.10.2 Inputs

- Bus

1.10.3 Outputs

- Bus

1.11 Input Register (INPR)

1.11.1 Metadata

Type: IORegister
ID: 13
Size: 8 bit

1.11.2 Inputs

- Keyboard

1.11.3 Outputs

- ALU_IN1

1.12 Output Register (OUTR)

1.12.1 Metadata

Type: IORegister
ID: 14
Size: 8 bit

1.12.2 Inputs

- Bus

1.12.3 Outputs

- Screen

1.13 End Carry (E)

1.13.1 Metadata

Type: Flag
ID: 15
Size 1 bit

1.14 Interrupt Request (R)

1.14.1 Metadata

Type: Flag
ID: 16
Size 1 bit

1.15 CPU Start (S)

1.15.1 Metadata

Type: Flag
ID: 17
Size 1 bit

1.16 Indirect (I)

1.16.1 Metadata

Type: Flag
ID: 18
Size 1 bit

1.17 Interrupt Enable (IEN)

1.17.1 Metadata

Type: Flag
ID: 19
Size 1 bit

1.18 Input Flag (FGI)

1.18.1 Metadata

Type: Flag
ID: 20
Size 1 bit

1.19 Output Flag (FGO)

1.19.1 Metadata

Type: Flag
ID: 21
Size 1 bit

2. Transfer Map

To / From	ALU	ALU_IN0	ALU_IN1	ALU_OUT	M	BUS	AR	PC	DR	AC	IR	TR0	TR1	INPR	OUTR
ALU	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	AC	🟥	🟥	🟥	🟥	🟥
ALU_IN0	🟩	🟩	🟥	ALU_OUT	🟥	🟥	🟥	🟥	🟥	ALU_OUT	🟥	🟥	🟥	🟥	🟥
ALU_IN1	🟩	🟥	🟩	ALU_OUT	🟥	🟥	🟥	🟥	🟥	ALU_OUT	🟥	🟥	🟥	🟥	🟥
ALU_OUT	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	AC	🟥	🟥	🟥	🟥	🟥
M	🟥	🟥	🟥	🟥	🟩	🟥	BUS	BUS	BUS	🟥	BUS	BUS	BUS	🟥	BUS
BUS	🟥	🟥	🟥	🟥	M	🟥	AR	PC	DR	🟥	IR	TR0	TR1	🟥	OUTR
AR	🟥	🟥	🟥	🟥	BUS	🟥	🟩	BUS	BUS	🟥	BUS	BUS	BUS	🟥	BUS
PC	🟥	🟥	🟥	🟥	BUS	🟥	BUS	🟩	BUS	🟥	BUS	BUS	BUS	🟥	BUS
DR	ALU_IN0	ALU_IN0	🟥	ALU_IN0	BUS	🟥	BUS	BUS	🟩	ALU_IN0	BUS	BUS	BUS	🟥	BUS
AC	ALU_IN1	🟥	ALU_IN1	ALU_IN1	BUS	🟥	BUS	BUS	BUS	🟩	BUS	BUS	BUS	🟥	BUS
IR	🟥	🟥	🟥	🟥	BUS	🟥	BUS	BUS	BUS	🟥	🟩	BUS	BUS	🟥	BUS
TR0	🟥	🟥	🟥	🟥	BUS	🟥	BUS	BUS	BUS	🟥	BUS	🟩	BUS	🟥	BUS
TR1	🟥	🟥	🟥	🟥	BUS	🟥	BUS	BUS	BUS	🟥	BUS	BUS	🟩	🟥	BUS
INPR	ALU_IN1	🟥	ALU_IN1	ALU_IN1	🟥	🟥	🟥	🟥	🟥	ALU_IN1	🟥	🟥	🟥	🟩	🟥
OUTR	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩

- Green: Target reached
- Red: Target is unreachable

3. ALU Commands

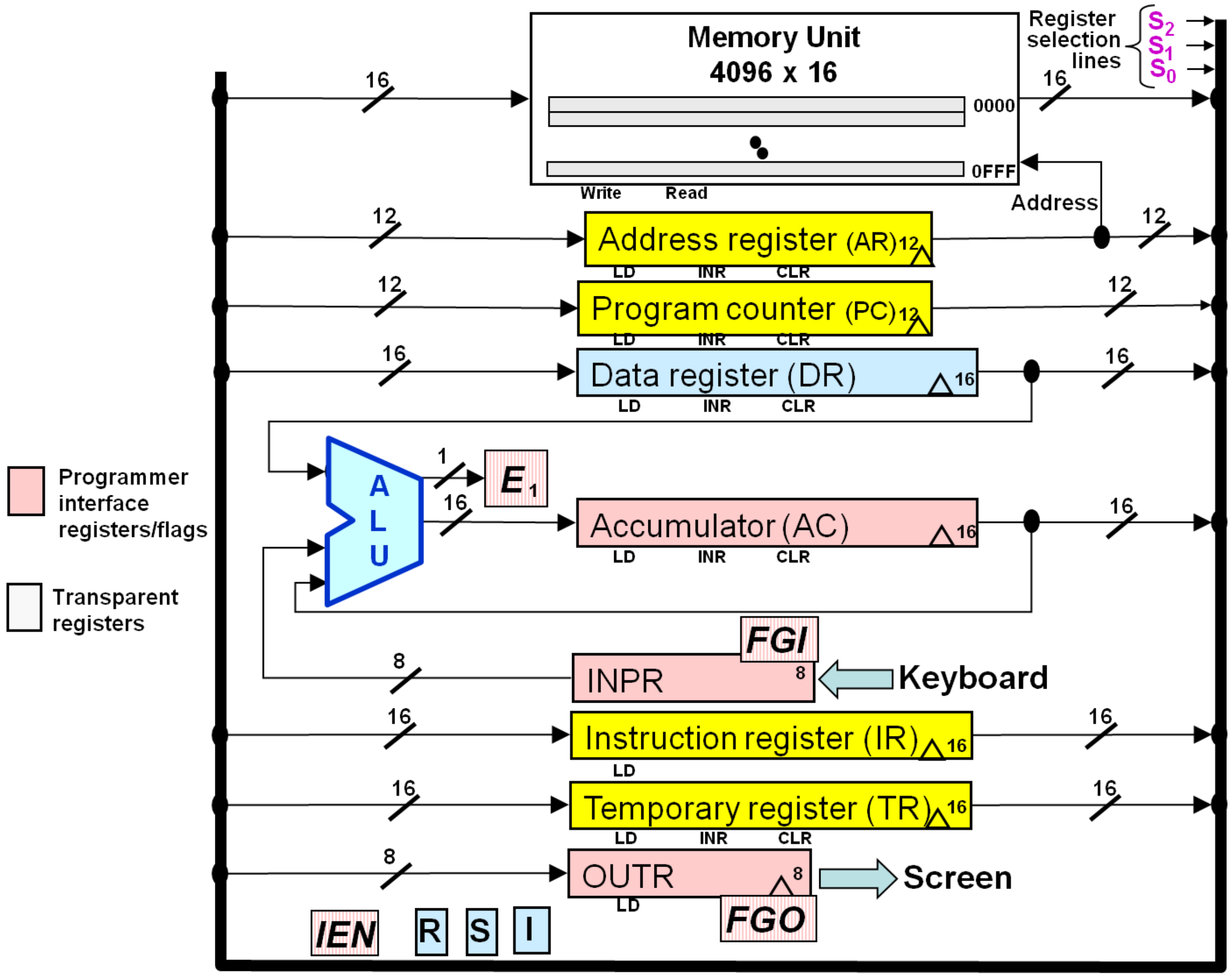
3.1 Unary Commands

- Complement: ~Operand

3.2 Binary Commands

- Sum: Operand + Operand
- Subtract: Operand - Operand
- Multiply: Operand * Operand
- Divide: Operand / Operand
- Modulo: Operand % Operand
- Bitwise And: Operand & Operand
- Bitwise Or: Operand | Operand
- Bitwise Xor: Operand ^ Operand
- Equal: Operand == Operand
- Not Equal: Operand != Operand
- Greater: Operand > Operand
- Lower: Operand < Operand
- Greater or Equal: Operand >= Operand
- Lower or Equal: Operand <= Operand
- Shift Left: Operand << <NUM_OF_BITS>
- Shift Right: Operand >> <NUM_OF_BITS>

4. CPU Architecture Diagram


Memory Unit
4096 x 16
0000
0FFF
Write
Read
Address
Register selection lines
S2 S1 S0
7
Address register (AR)
LD INR CLR
1
Program counter (PC)
2
Data register (DR)
3
A L U
E
Accumulator (AC)
4
Programmer interface registers/flags
Transparent registers
FGI
INPR
Keyboard
Instruction register (IR)
5
Temporary register (TR)
6
OUTR
Screen
FGO
IEN
R
S
I
16
12
8