<franzageek>

Heza 1 processor

Custom Processor Architecture

User Manual

Disclaimer

General Disclaimer

Heza 1 is a fully working CPU (Central Processing Unit) design made in Logisim, a powerful software that allows the design of integrated logic circuits and their testing in a visual environment. This means that the Heza 1 processor can be tested and exists only in a Logisim environment, and is not a real thing. The realization of an actual chip to use in an actual board is currently planned, however its realization is not a priority at the moment.

Document Disclaimer

Revision History

Each entry in the following table represents a change to this document from its previous version.

Date	Rev. Level	Description	Page
Aug 16, 2023	0	First release of the document	All
Aug 24, 2023	1	Correct some words, add missing punctuation marks	ii, 6, 8, 10, 11, 14, 15

Table of Contents

Table of Figures

Architecture Overview

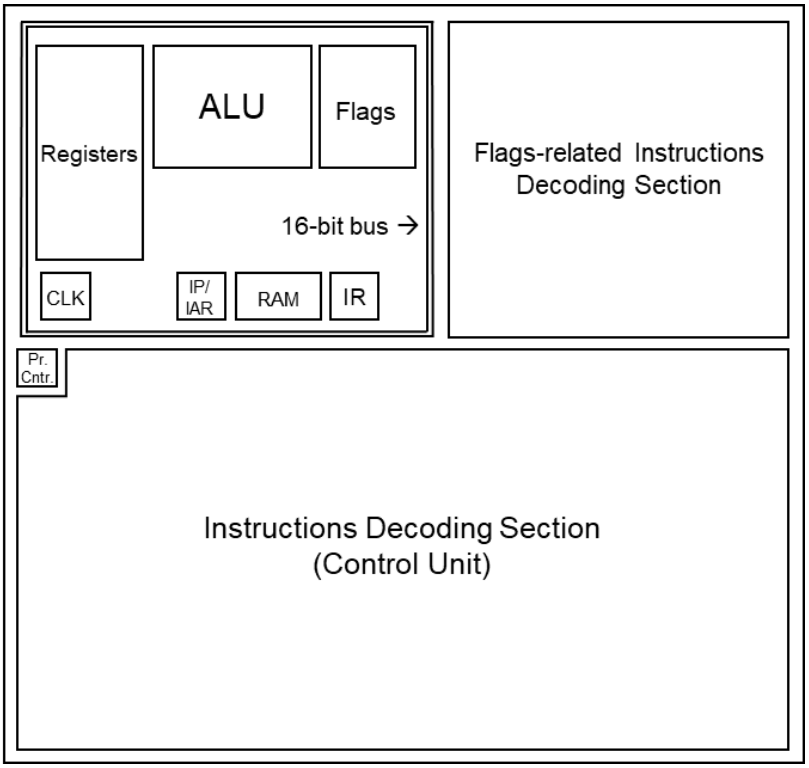
Heza 1 is a 16-bit CUPRA (CUstom PRocessor Architecture) RIS (Reduced Instruction Set) microprocessor designed in Logisim.
Logisim is a powerful free and open-source software available as an executable file for Windows and MacOS X in their respective formats, and as a Java .jar executable in its cross-platform version.
Logisim can be used to design and test integrated logic designs in a visual environment. It is available at http://www.cburch.com/logisim/.

CPU Features

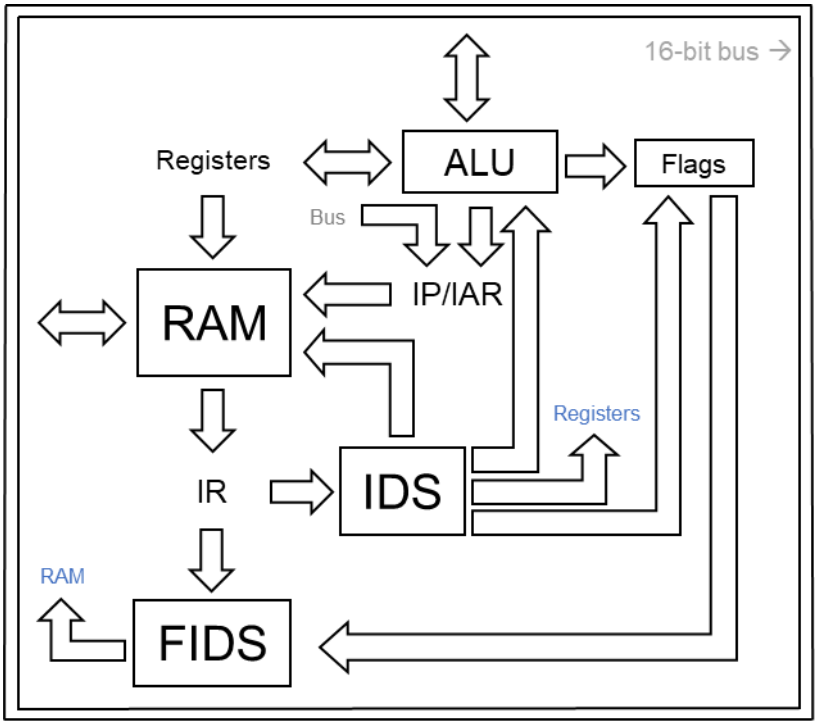
Heza 1's custom architecture features a speed between 1Hz and 4.1KHz when running in Logisim Classic. When running in Logisim Evolution, it can reach speeds up to 2MHz (achievable with a reasonably powerful system). Unfortunately, Heza 1's port to Logisim Evolution is not completed yet, and may result in some issues due to some base-level differences in the default components between Logisim Classic and Logisim Evolution. The fact that it may result unstable is the reason why the port version is currently not public.

The CPU features 12 registers (4 general purpose registers, 5 flag registers, 1 instruction pointer/instruction address register, 1 instruction register and 1 4-steps program counter register), a 16-bit ALU capable of adding, subtracting, multiplying, dividing, shifting, NOTing, ANDing, ORing, XORing and comparing two numbers. It's got 128 kilobytes of RAM, a 4-steps program counter, a 30-bit control bus to control memory locations, ALU and all the devices of the CPU.

The below figure shows the internal layout of the processor:



The below figure shows a representation of the connections between various parts of the CPU:



For more clarity on abbreviations and the meaning of colors, see the following page.

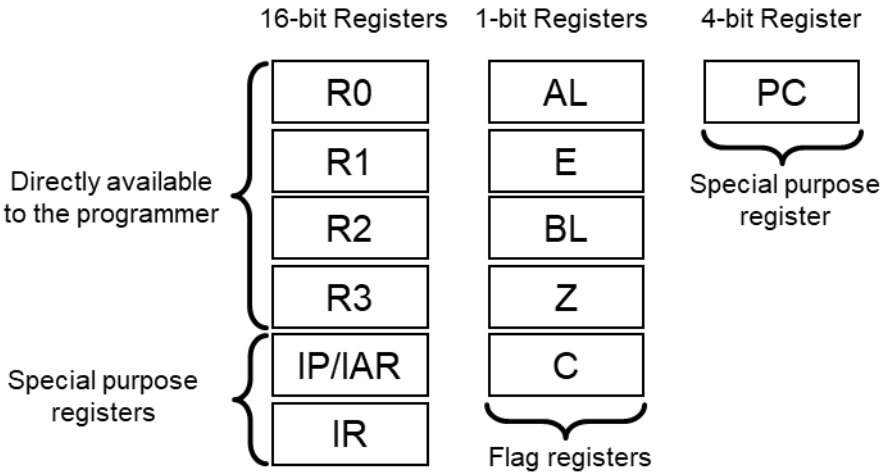
Abbreviations and colors clarification

CLK: Clock
IP/IAR: Instruction Pointer/Instruction Address Register
IR: Instruction Register
PC/Pr. Cntr.: Program Counter
IDS: Instruction Decoding Section (Control Unit)
FIDS: Flags-related Instruction Decoding Section

Black label: component
Blue label: links to a component

CPU Memory

The Heza 1 CPU contains 64 bits of read/write memory that are directly available to the programmer and 40 bits of memory that are automatically controlled by the CPU. The below figure shows how these 104 bits of memory are configured to six 16-bit registers, five 1-bit registers and a single 3-bit register:



Special-Purpose Registers

Instruction Pointer (IP): The Instruction Pointer is a special type of register that increments its value by one at each clock cycle. It is able to count from 0 to 65,535 and it's connected to the RAM's Address input. Its job is to make the RAM point at the next address during the clock cycle. When a JUMP instruction is executed, the new value is automatically placed in the Program Counter, overriding the increment.

Program Counter (PC): It is a 3-bit register whose output is fed into the input of a 3x8 decoder. The PC increments its value by one at each clock cycle, thus making the decoder enable the next output wire. The enabled output wire changes whenever a clock pulse is applied, thus allowing to split the work of the Instruction Decoding Section into 4 different phases. Phase 5 is utilized to reset the PC. Phases 6 & 7 are disabled.

Instruction Register (IR): It's a regular 16-bit falling edge-triggered register which is connected to the Data Output Bus of the RAM. Its purpose is to store the current instruction OPCODE and pass it to the Instruction Decoding Section. It gets set during Phase 1, the phase which determines which instruction needs to be executed. The IR holds the OPCODE during Phases 2, 3 & 4, phases while RAM would normally output other values needed to execute the instruction.

Flag Registers

Flag registers are a group of five 1-bit registers that are used to store specific attributes of the previous operation, such as if the first input was greater than the second, or if the result of the operation was zero.

A LARGER: when 1, the first input is greater than the second.
EQUAL: when 1, the two inputs are equal.
B LARGER: when 1, the second input is greater than the first.
ZERO: when 1, the result of the operation is zero.
CARRY (OUT): when 1, the operation has a carry out.

General-Purpose Registers

Heza 1 CPU offers a group of four general-purpose 16-bit registers that are directly controllable by the programmer. They can store up to 64 bits of data altogether. These 16-bit registers are *REG_00*, *REG_01*, *REG_02* and *REG_03* (also called *R0*, *R1*, *R2* and *R3*).
These registers do not differ in speed or efficiency from each other, they all run at the same speed and can read and write data simultaneously and independently.

Random Access Memory

The processor features 128KB of built-in RAM, capable of addressing to up to 65,535 different memory locations. It's got one bi-directional data bus, which can operate in both Input and Output mode. The RAM is connected to the Internal Data Bus, and can write data based on its value and output data or instruction OPCODEs onto it.
The RAM allows you to manually access its content and edit every single memory location's value with complete ease. You can also fill out the memory by specifying an existing image: the RAM will fill up with the values inside the image. This is especially useful when loading a program, as it's the quickest way to do it.

Instruction Register and CPU Control

Once an instruction is fetched from memory, its OPCODE is set in the Instruction Register. The IR then passes it to the Instruction Decoding Section or Control Unit. The Control Unit then identifies the OPCODE and enables the necessary control signals needed to read or write data from or to the registers, control the ALU or access the RAM. The Heza 1 processor features a 30-bit Control Bus, which bundles all existing control signals and makes them accessible to the Instruction Decoding Section and its Flags-related part.

Arithmetic and Logic Unit

The 16-bit arithmetic and logical instructions of the CPU are executed in the Arithmetic and Logic Unit (ALU). The ALU is connected to the registers, to the Instruction Pointer and to the Internal Data Bus. The operations that can be performed by the ALU include:

- Addition
- Subtraction
- Multiplication
- Division
- Logical Shift Left
- Logical Shift Right
- Logical NOT
- Logical AND
- Logical OR
- Logical XOR
- Comparison of two numbers

Timing

The processor executes instructions by stepping through a set of basic operations. This set includes:

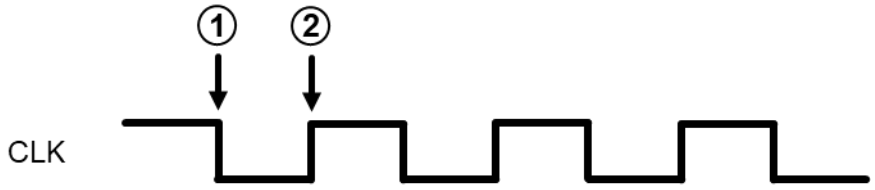
- Reading or writing from or to the registers
- Reading or writing from or to the RAM
- Controlling the ALU
- Incrementing the Instruction Pointer
- Resetting the Program Counter

The processor can run at speeds up to 4.1KHz, with an average speed of 3.5KHz on a discrete PC. Each instruction can take from two to four clock cycles to complete, since all of these basic operations have to be broken down into multiple steps.

Instruction Fetch

The next figure shows the timing diagram of each Instruction Fetch Cycle.

Right after the RESET signal goes low, the RAM loads its data to the output.
Since the clock signal is high (it always starts in a high state thanks to some logic, regardless of the state of the CLK item) and since the Instruction Register is a falling edge-triggered register (while the rest of the circuit is rising edge-triggered), no instruction will be executed until the clock signal goes low (*instruction OPCODE gets placed in the IR and passed to the Control Unit*) and high again (*all the rising edge-triggered devices get clocked*). Only then will the first instruction be executed.



(1): The instruction OPCODE is placed in the Instruction Register and the instruction gets identified;

(2): The instruction begins to execute.

Heza 1 CPU Instructions

The Heza 1 Processor can execute 23 different instructions, each one with its own sub-instructions and combinations. They all fall in these major groups:

- Move
- Arithmetic and Logical
- Shift
- Bit manipulation (Set, Reset, End)
- Jump

Instruction Types

“Move” instructions move data among CPU registers or between CPU registers and system memory. Each of these instructions need a source location (from which the data is to be moved) and a destination location to be specified. The source location is not altered by a “Move” instruction, except for the “Overwrite” instruction, which is for testing purposes only: it is unstable and can be executed only as a standalone instruction (separated from the rest of the program) as it would mess up with the data in the registers.
Examples of “Move” instructions include moves between any of the general-purpose registers such as “move Register A to Register B”. This group also include moves of immediate values into any general-purpose register or to any system memory location. Other types of “Move” instruction allow transfers between CPU registers and system memory

The “Arithmetic and Logical” instructions operate on data stored in general-purpose registers. The appropriate flags are set according to the result of the operation.
An example of “Arithmetic” operation can be every 16-bit operation, such as adding the contents of two registers, or dividing one by the other.
A “Logical” operation can instead be represented by binary ORing the contents of two registers, or by NOTing a register’s value.

The “Shift” group includes instructions that allow to shift the value of a register to the left or to the right by one digit, with or without a carry.

The “Bit Manipulation” instructions allow any general-purpose register or any special-purpose register to be reset in any moment. To this group belongs also the “End” instruction, which stops all upcoming instructions from being executed.

Finally, the "Jump" group of instructions includes all the instruction that allow to jump from a RAM location to another one, by changing the address stored in the Instruction Pointer. By breaking the normal routine of incrementing the Instruction Pointer at every clock cycle, and by loading a custom address into it, the RAM will not follow the normal execution of the program (one address after another), and will instead skip every address until the specified one is reached. Then, the RAM will resume processing addresses one-by-one, one after the other.
There are two types of jumps that can be performed:
Conditional Jump: The Jump is executed only if the specified condition turns out to be true. There are several examples: *jump if equal* (jumps only if the two inputs of the previous operation were equal), *jump if A larger* (jumps only if the first input of the previous operation was larger than the second one), *jump if B larger* (jumps only when the second input of the previous operation was greater than the first), *jump if zero* (jumps only when the result of the previous operation was zero) or *jump if carry* (which jumps only if the previous operation left a carry out).
Unconditional Jump: The Jump is executed whatsoever. No matter what, when an Unconditional Jump instruction is executed, the Instruction Pointer will be loaded with a custom value, thus making the RAM point at the specified address.
There are also two different ways to specify the address to jump to:
Jump to Register Value: This instruction needs a source register to be specified. The value in that register will be taken as an address to jump to. This is particularly useful to save memory space: the jump-to-register-value instruction can be easily embedded in a large program (since it takes up only 2 bytes) for which every byte is essential. This is also useful when the destination address needs to be specified by an arithmetic or logical operation (the result of which would be saved in a register that could later be specified as the destination address of the jump instruction).
Jump to Immediate Value: With this instruction there's no need to specify a source register from which to take the destination address. This instruction is not particularly memory space-friendly as it takes 4 bytes of space in RAM, two for the 16-bit opcode and two for the address. The big advantage of this instruction is that the destination address of the jump is stored in the next memory location, making it more directly accessible as it can be chosen right when coding the program. It can also be set according to a memory address corresponding to a *:label* in the code, which can be used to divide the program into parts or to create loops.

Addressing Modes

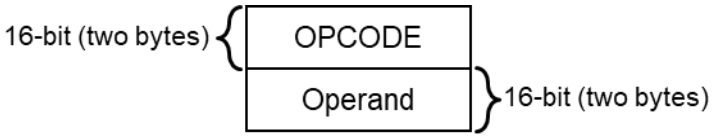
Most of the Heza 1 instructions operate on data stored in internal CPU registers or in system memory. Addressing refers to how the address of this data is generated in each instruction. This section is a brief summary of the types of addressing used in the Heza 1 CPU.

Register Addressing

Many of the Heza 1 OPCODEs contain bits which specify what registers to consider for a certain operation. All the parameters in a Register Addressing instruction need to refer to a register. An example of Register Addressing instruction is moving the value stored in Register A to Register B.

Immediate Addressing

In the Immediate Addressing mode, the memory location next to the opcode contains the actual operand, as shown in the figure below.

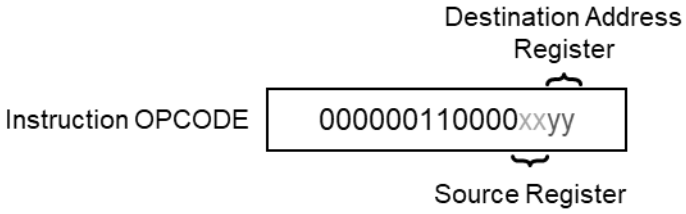


An example of this addressing mode is loading a value into a register: first comes the OPCODE which contains all the needed parameters (such as the selected register), then comes the value to load into that register.

Indirect Addressing

The Indirect Addressing mode is the mode of addressing where the instruction contains the address of the location where the target address is stored.
For example, the “move the value of Register A into the RAM address stored in Register B” instruction (of OPCODE = ‘000000110000xxyy’, where the four last bits are the source register and the register which contains the destination address) is an Indirect Addressing instruction, as the location of the target address is already specified in the OPCODE itself.

The figure below shows a graphic representation of the example. For further information about the instruction itself, see the *Instruction OPCODEs* chapter.

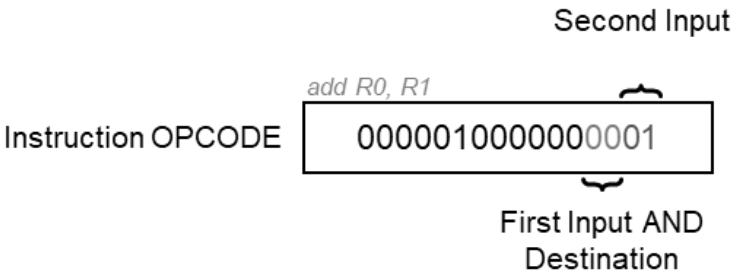


Bit Order-Sensitive Addressing (pattern)

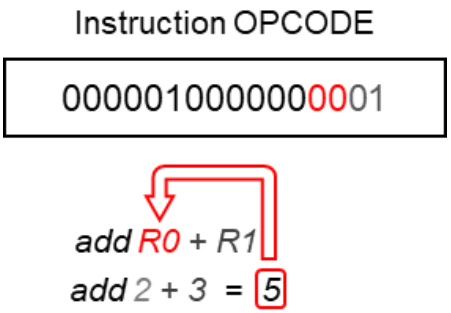
This addressing pattern is widely used when talking about arithmetic and logic operations, which require the OPCODE parameters to be positioned in a specific way.

More precisely, this applies to the last four bits of the OPCODE, which determine the two input registers of the operation.
The first two bits of those four specify the first input of the operation, but also the register which will hold the result of the operation.

To explain this addressing mode, here's a simple example.
Let's say that we've loaded 2 into R0 and 3 into R1, and we want to add R0 and R1:



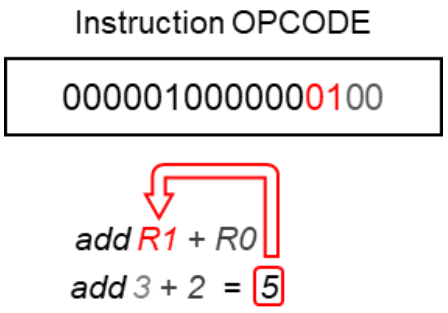
By executing the above instruction, we would get a result of 5:



The key thing here is that the result of the operation, in this case the number *5*, will be saved in the *register specified by the top two bits* of the last four bits of the OPCODE.

After executing this instruction, we will get 5 as the result. The number 5 will then be saved in R0. So, if we now look at the values in our registers, we will see that R0 contains now *5* and R1 still contains the *3* we previously loaded.

But what if we load again a 2 into R0 and execute the instruction again, but swapping the first input with the second (therefore adding *R1 with R0*)?



We would still get a result of 5, but this time it was saved in R1 instead of R0. Meanwhile, R0 remains set to 2.

This addressing mode applies to all two inputs-arithmetic operations and also to two inputs-logical operations such as AND, OR and XOR. Shifts and NOTs do not use Bit Order-Sensitive Addressing.

Instruction OPCODEs

This section describes each of the Heza 1 instruction OPCODEs. All instruction OPCODEs are listed in hexadecimal notation, along with a decimal conversion.

➤ **Note:** This user manual **will not cover** each instruction's description, behavior, parameters, usage and mnemonics. For further details on that, see the ***Instruction Table*** that comes with the circuit files and the documentation.

Each instruction is divided in two parts: the *Instruction OPCODE* and the *parameters*.

The **Instruction OPCODE** is represented by the top eight bits of the instruction. It is the instruction identifier: the combination of ones and zeros that determines the instruction to execute. The OPCODE gets saved into the IR at Phase 1, and passed to the Control Unit to control which of the individual components needs to be used.

The **Instruction Parameters** are specified by the last eight bits of the instruction. They include a bunch of settings that can be decided by the programmer, and which may change the outcome of the instruction. For example, a parameter is represented by the code of the arithmetic operation to execute, or by the code of the registers that are used as the two inputs of the operation.

This section will only go through the OPCODEs of the instructions: to know more about their behavior or about the supported parameters, see the ***Instruction Table*** included with the circuit files and the documentation.

Move Between Registers/Move Immediate

The OPCODE for these instructions is **00**xx – **00000000**xxxxxxxx.
These include:

- Move the value stored in a register into another
- Move an immediate value into a register

Move Indirect Value into Register

The OPCODE of this instruction is **02**xx – **00000010**xxxxxxxx.
This OPCODE is valid for the instruction that loads the value of the RAM address specified by a register INTO another register.

Move Register Value into Indirect Address

The OPCODE of this instruction is **03**xx – **00000011**xxxxxxxx.
This OPCODE is the identifier of the instruction that loads the value of a register INTO the RAM address specified by another register.

ALU Operations

The operations performed by the ALU have all the same OPCODE:
04xx – **00000100**xxxxxxxx.

These operations include:

- Addition
- Subtraction
- Multiplication
- Division
- Logical Shift Left
- Logical Shift Right
- Logical NOT
- Logical AND
- Logical OR
- Logical XOR

Jump Instructions

Jump instructions allow to jump to a specific RAM address if the specified condition turns out to be true.
They can be identified with the OPCODE **05**xx – **00000101**xxxxxxxx.
There are different types of Jumps that can be performed:

- Jump if greater
- Jump if equal
- Jump if lesser
- Jump if zero
- Jump if carry
- Unconditional Jump

Clear Registers

This instruction allows to reset the value of a register.
Its OPCODE is **07**xx – **00000111**xxxxxxxx.

END Instruction

This instruction stops the CPU from processing instructions.
However, it is not divided in two parts (as previously stated), one being the OPCODE and one containing the parameters, but it is considered as a whole.
Its identifier is **aaaa – 1010101010101010**.

➤ **Note:** Each instruction has its own parameters and mnemonics. To discover them, see the ***Instruction Table***.

Heza 1 CPU Assembly Language

An assembly language allows the user to write a program without concern for memory addresses or machine instruction formats. It uses symbolic addresses to identify memory locations and mnemonic codes (OPCODEs and operands) to represent the instructions. Labels (symbols) are assigned to a particular instruction step in a source program to identify that step as an entry point for use in subsequent instructions. Operands following each instruction represent storage locations, registers, or constant values.
To code a program, see the ***Instruction Table*** Excel file. It contains all the information that you may want to know when coding a program for the Heza 1 CPU.

Assembler Program

FranzaGeek provides an assembler program written in Batch, to make it light and easy to use.
It's got a simple Command Line Interface which allows the user to see the current assembly phase and status in real time.
It's currently stable, but it doesn't support certain instructions (such as Jump instructions and the Overwrite instruction) and it's slightly bugged, but it works, and that's the important thing.
Important updates with bug fixes and missing instructions-implementation will be published, making the assembler program more and more advanced each time.

To compile a program, first you need to save your source program in a TXT format. Then, open a Command Prompt and run this command:

```
> assembler.bat YourProgram.txt
```

The compiling process takes less than 1 second. You will see a bunch of text lines, and soon after your program will be compiled.

Running a program

Let's assume that you're done coding your source program and you've assembled it. To make the Heza 1 CPU execute your code, follow these steps:

1. Open the Heza 1 circuit file ("*heza-cpu.circ"*).
2. Scroll until you find the RAM component.
3. Right click on the RAM item and choose "*Edit Contents...*" from the menu. You will be able to see the full content of the RAM.
4. Click the "*Open…*" button at the very bottom of the window (you might need to resize the window to see it).
5. From the file selection dialog, choose the HEZ program that you want to load.
6. Reset the CPU a couple of times by clicking the "*r*" button.
7. Choose your clock speed from the topbar menu (*Simulate* > *Tick Frequency* > *...*).
8. Reset the CPU one more time (it's a good practice to do so).
9. Press **Ctrl+K** to start the simulation.
10. Once the CPU is done executing the program, press **Ctrl+K** again to stop the simulation.
11. Repeat from Point 3 to execute a different program, repeat from Point 8 to execute the same program again.
12. When you're done, save the project and quit Logisim.

Support

If you have problems when executing a program or when using the assembler, please let me know by filling an Issue Report at https://github.com/franzageek/heza-cpu/issues.
You can see other projects from FranzaGeek at https://github.com/franzageek/.
You can download the assembler at https://github.com/franzageek/heza-cpu-assembler.
If you need more information, visit the repo at https://github.com/franzageek/heza-cpu.