Abterra I

Thomas van Maaren

January 2, 2021

1 Introduction

The Abterra I is a computer built in minecraft without the use of any command blocks or mods. The computer can be downloaded here: https://www.kznarsk.xyz/abterra/. This manual shall explain the inner workings of this machine, so you will be able to program or modify it. I would advise you to learn binary and the logical operations before you read this article. This is a good source: https://simple.wikipedia.org/wiki/Binary_number.

Maybe you are confused why this would count as a computer, because it does not fit in the stereotypical image of a computer. It does not have a keyboard or a screen for example. What you don't understand is that a screen or a keyboard are just ways for the computer to communicate with the user and are not essential to run a program.

This computer differs from other minecraft computers in a few ways:

- Small and simple instruction set.
- Instructions and data stored in the same place.
- Programs can be written from the Control room.

It being a minecraft computer you shouldn't expect it to run too complex software. Here are the specs:

- 256 addresses of ram, which in this case is 2816 bits
- 8 instructions
- Cpu of 48 milliherz
- Two clock cycles per instruction

This computer is eaven way slower than computers from the 1940's and can't really be used for any practical applications.

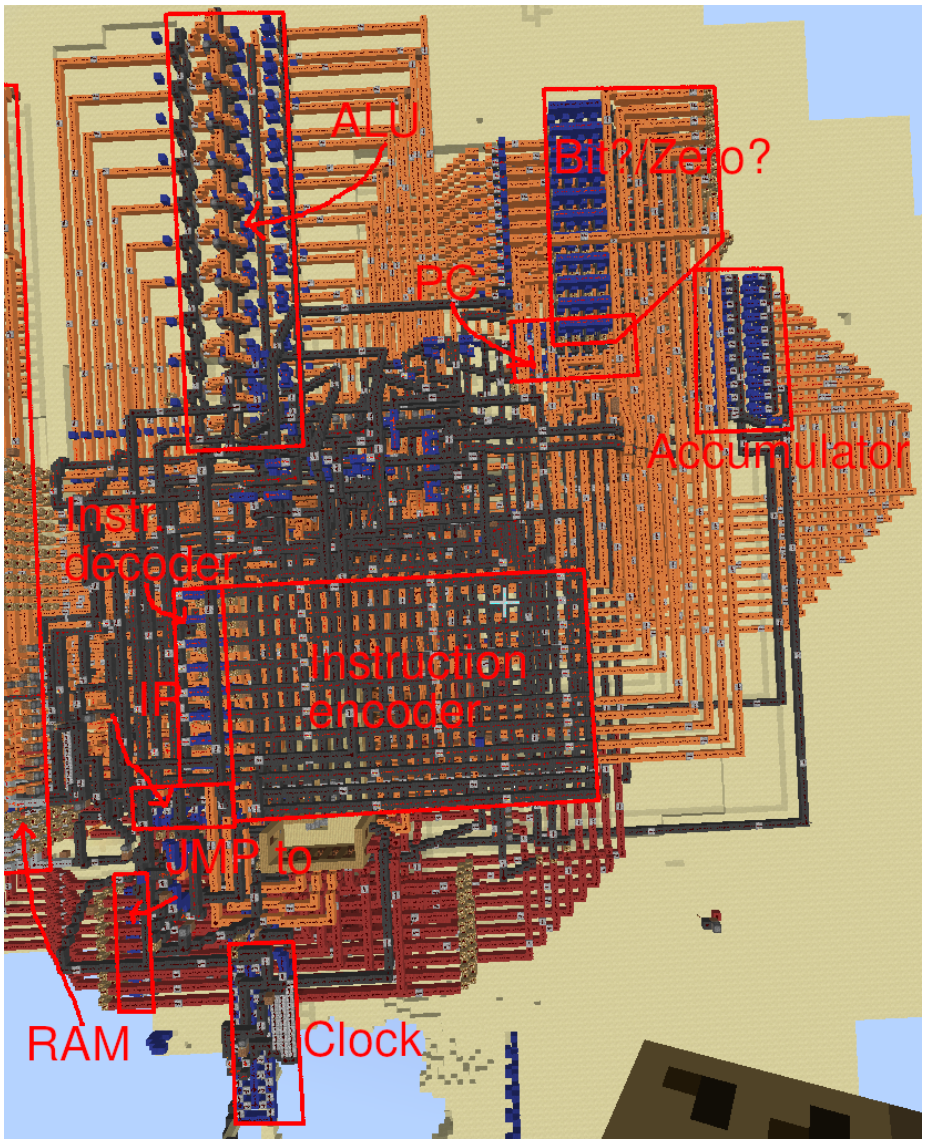


Figure 1: Map of Abterra I

2 RAM

RAM stands for Random Access Memory. It is the place where data is stored. What is special about this minecraft computer is that the program that the computer is currently running is in the same place as the data that the program uses. a downside of this is that the computer is bigger, because the RAM has to be big enough to store a reasonably big program. The advantages of having the data and program in the same place is that the program can change itself. See section 7.5.2. It also features an alternative way of writing the code into the computer. See section 7.4.

Figure 2: Shows the bussses going into RAM

The RAM is comprised of 256 addresses which can be individually read or be written too by the CPU. To select which address it wants to read or write the CPU uses the address bus. This bus has eight bits which makes it capable to address all the 256 addresses. The RAM is enabled when the "Enable RAM" line is turned on. If it is enabled the contents of the specified address will then be put on the data out bus and if read/write is turned off and RAM is enabled the value on the "data in" bus shall be stored in the specified address.

For selecting the right address the ram uses decoders. The ram has eight layers which all have two sides which both have sixteen addresses. The address uses the format lllsaaaa. Bit 7, 6, 5 of the address bus selects which layer the address is, bit 4 Selects in which side of the layer the address is and bit 3, 2, 1, 0 selects the address in the side in the layer. For this it uses three types of decoders.

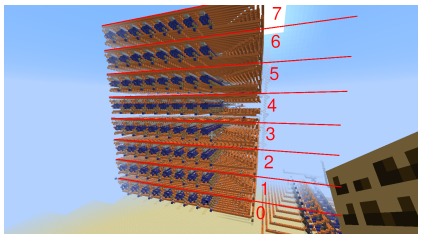


Figure 3: 8 layers

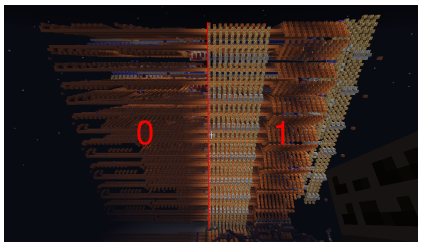


Figure 4: 2 sides per layer

One to select the layer, one to select the side in the layer and one to select the address in the side in the layer. This is more efficient than having one big decoder, because the decoders can decode simultaneously.

The design of the RAM addresses comes from this video: https://www.youtube.com/watch?v=dy_sBWzFRtk&lis

3 ALU

The ALU has the responsibillity to perform arrithmetic and logical operations on values. There are four operations:

- add
- neg
- NOT
- AND

To add the ALU uses adders. Watch this video to understand how an adder works: https://www.youtube.com/watch?v=qZauC0OicRM. In binary a number can be negated by performing a NOT operation on it and then adding 1 to it. When neg is being performed it will set In1 to 0x001 which is equal to one. The NOT will be enabled wich will perform the logical operation NOT on In2. This value shall then be added with In1 wich is as previously described one. Thus the negative of In2 shall come out of the ALU. NOT works the same as Neg, except that In1 is equal to 0x000 or zero. If you understand how an adder works which you do if you have watched the video. You would know that the carry of every half adder is the same as performing the logical operation AND on the two inputs. Thus we can get an output of AND by connecting the carry bit to the ALU out. The carry lines between the half adders will be disconnected, as the different bits shouldn't have effect on each other.

In this design there are many operations that are missing, but this is not a big problem as with these operations al other operations can be performed in software.

Figure 5: ALU

3.1 carry

If the ALU is performing the operation add there is a possibility that the output value is to big to display with 11 bits. This is called overflow. If overflow occures the carry register shall be turned to one. If there is no overflow during addition the carry register shall be put to zero. This register is used for conditional branching (see section 5.2).

4 Instructions

In this section we will talk about de different instructions and how they are executed and loaded.

4.1 Program counter

The program counter has the task to keep track of which instruction is currently being executed or loaded. When the computer is reset, the program counter will be equal to zero. This is true for all registers.

4.2 Instruction cycle and clock

The program that the computer runs is comprised of instructions that are executed sequentially. There are two fases in executing instructions. First is the fetch fase where it will increment the program counter by one and then load the

instruction from that address and store it in to the instruction register. If it is loading the first instruction that gets executed, the program counter wont be incremented as it should load from address 0x000.

After the fetch fase comes the execute fase which is when the instruction in the instruction register will be executed. The fetch/exec register keeps track in which fase the computer is. When it is in the execute fase it will hold the value one and if it is in the fetch fase it will hold the value zero. The clock goes on and off in a constant frequency. The clock does this to make sure that the different parts of the computer are synchronised. a Fetch/ex period is equal to two clock periods. A clock period is equal to one fase and a Fetch/ex period is equal to one loaded and executed instruction.

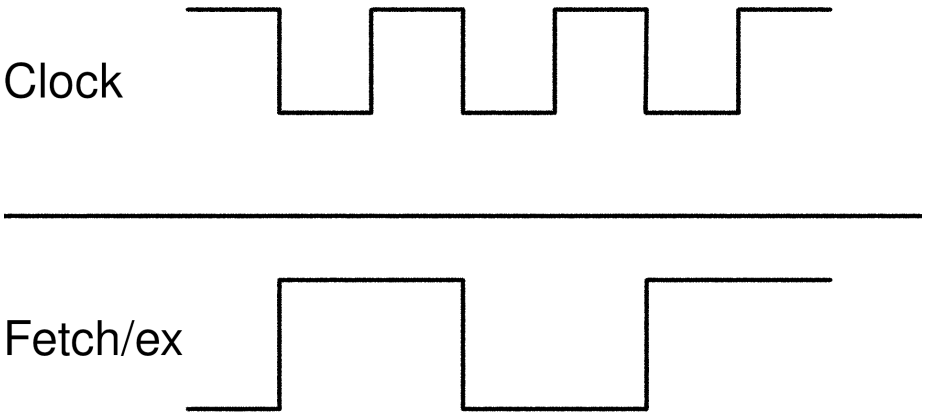


Figure 6: shows the period of Fetch/ex compared to clock

4.3 Accumulator

The accumulator is a register which functons as a temporary storage location. This is necassary when there has to be an operation done with multiple values. Only one value can be loaded from RAM at a time. Thus it should load one value and store it in the accumulator and then perform the operation.

4.4 Load

Instruction:000 aaaa aaaa
Loads from an address in RAM and stores it in the accumulator. The a bits should be replaced with the address in RAM that it should load from.

4.5 Store

Instruction:001 aaaa aaaa
Loads from the accumulator and stores it in an address in RAM. The a bits should be replaced with the address in RAM it should write too.

4.6 Add

Instruction:100 aaaa aaaa
Loads from the accumulator and load from an address in RAM and add those values. This value is then stored back in the accumulator. Replace the a bits with the address it should load from.

4.7 Neg

Instruction:101 aaaa aaaa
Load from an address in RAM, negate [1] this value and store it in the accumulator. Replace the a bits whith the address that it should load from.

4.8 NOT

Instruction:011 aaaa aaaa
Load from an address in RAM, perform the logical operation NOT on it and store it in the accumulator. Replace the a bits whith the address that it should load from.

4.9 AND

Instruction:100 aaaa aaaa
Loads from the accumulator and load from an address in RAM and perform the logical operation AND to them. This value is then stored back in the accumulator. Replace the a bits whith the address that it should load from.

4.10 Decoding and encoding

As said in section 4.2 the current intruction is stored in the instruction register. Bit 7, 6 and 5 which specify the instruction are then send to the instruction decoder which connects with the instuction encoder. The instruction encoder describes which actions should be taken given the instrution. It is composed of of 11 lines. Eight correspond with an instruction and three correspond with a fetch operation (see section 5) If the fetch/ex register is turned of which is equivalent to saying that the CPU is in the fetch fase it will turn of one of the three fetch lines. If it is in the execute fase it will Every instruction has its own line that can be turned of.

[1] Making a positive number from a negative number and making a negative number from a positive number

Figure 7: Shows the different types of lines that the encoder has.

4.11 Control lines

There are in total 22 control lines which are connected between the instruction encoder and the different parts of the machine. Here is a table of all of them.

name	num	LOAD	STORE	AND	NOT	add	neg	SKIP
ALUb=JUMP to	21	0	0	0	0	0	0	0
en IR	20	0	0	0	0	0	0	0
addr.=instr.	19	1	1	1	1	1	1	0
en RAM	18	1	1	1	1	1	1	0
ALUb=ALU	17	0	0	1	1	1	1	0
ld	16	1	0	1	1	1	1	0
jmp	15	0	0	0	0	0	0	0
skip	14	0	0	0	0	0	0	1
Too In2.1	13	0	0	0	0	0	0	0
Too In2.0	12	0	0	0	0	0	0	0
addr=PC	11	0	0	0	0	0	0	0
In2=CU	10	0	0	0	0	0	0	0
In2=RAM	9	0	0	1	1	1	1	0
neg	8	0	0	0	0	0	1	0
en acc	7	1	0	1	1	1	1	0
add	6	0	0	0	0	1	0	0
AND	5	0	0	1	0	0	0	0
NOT	4	0	0	0	1	0	0	0
In1=pc	3	0	0	0	0	0	0	0
In1=acc	2	0	0	1	0	1	0	0
en PC	1	0	0	0	0	0	0	0
ALUb=RAMb	0	1	0	0	0	0	0	0

continued:

name	JMP	-norm	-jmp	-skip	Description
ALUb=JUMP to	0	0	1	0	The contents of the "JMP to" register shall be put on the ALUb
en IR	0	1	1	1	Change the value of the instruction register.
addr.=ins	0	0	0	0	Bit 7 untill 0 of the instruction goes too the address bus.
en RAM	0	1	1	1	Enable ram
ALUb=ALU	0	1	0	1	Put the output of the ALU on the ALU bus
ld	0	1	1	1	Read from RAM, otherwise it will write to RAM
jmp	1	0	0	0	Enable JMP register.
skip	0	0	0	0	Enable the skip register
In2.1	0	0	0	1	If "In2=CU" is also enabled it decides if bit 1 of In2 should be 1 or 0
In2.0	0	1	0	0	If "In2=CU" is also enabled it decides if bit 2 of In2 should be 1 or 0
addr=ALUb	0	1	1	1	Output of the ALU goes to the address bus of RAM
In2=CU	0	1	0	1	Bit 2-10 of In2 will al be 0 and 0-1 are decided by In2.1 and In2.0
In2=RAM	0	0	0	0	The output of ram will be put in In2.
neg	0	0	0	0	Output of ALU is negative of In2
en acc	0	0	0	0	Change the value of accumulator
add	0	1	0	1	Output of ALU is sum of In2, In1
AND	0	0	0	0	Output of ALU is AND on In2, In1
NOT	0	0	0	0	Output of ALU is NOT on In2
In1=pc	0	1	0	1	In1 is the contents of PC
In1=acc	0	0	0	0	In1 is the contents of accumulator
en PC	0	1	1	1	Change the value of PC
ALUb=RAM	0	0	0	0	Puts output of RAM on ALU bus

5 Branching

5.1 JMP

Instruction:111iiiiiiii
When the JMP instruction is executed the JMP register shall be turned to one and bits i bit of the instruction (bit 0-7) shall be stored in the "JMP to" register. Because the JMP register is set to one, the instruction -jmp shall be executed instead of -norm during the next fetch fase. What this means is that it won't

go to the next instruction as would normally happen, but to the instruction specified by the jmp instruction.

5.2 Skip

Instruction:110bbbbzcsx
When the Skip instruction is executed the Skip register shall be turned to one if a certain condition is met. If the skip register is turned on the PC register shall be incremented by two instead of one in the next fetch fase, thus skipping an instruction. There are three posible conditions:

- Zero
- Carry
- Bit

If you want the zero condition you should turn on bit z (bit 3). If this condition is selected the skip register shall be turned to one when the value stored in the accumulator is zero.

If you want the carry condition you schould turn on bit c (bit 2). If this condition is selected the skip register shall be turned to one if the carry register is on (see 3.1).

If you want the Bit condition you should turn on bit s (bit 1). This will check a specified bit in the accumulator. You specify the bit of the accumulator you want to check with the b bits (bit 4-7). If the specified bit is one it will turn the skip register on.

If the skip register is on the -skip instruction shall be performed in the next fetch fase, otherwise it will perform the -norm instruction as usual.

6 Control Room

The Control Room has two spaces: The debugging room and the read/writing room.
As the name suggests the debugging room is the place where one would debug the machine. There are indicator lights which display the contents of the different registers. In figure 13a, 9b and 10 you can see which registers are displayed.

If you want to debug the machine it would be nice if you could pause the computer. The leaver in figure 10 can be used to switch the computer between running and paused. When the system is paused you can use the button on the left to change the clock from an on to an off state or vice versa.

(a) Debugging room

(b) Read/Writing room

(a) The PC register and accumulator

(b) The flag registers

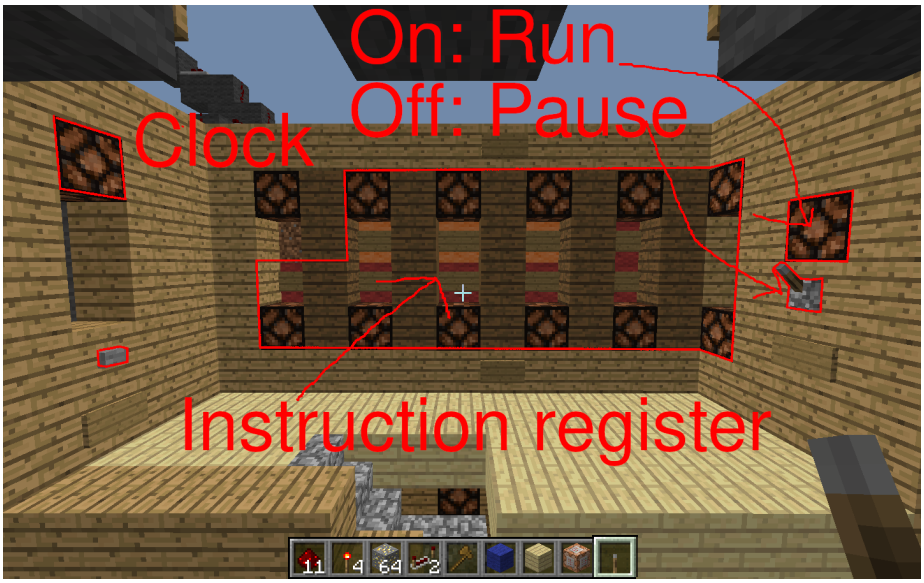


Figure 10: Instruction register and clock

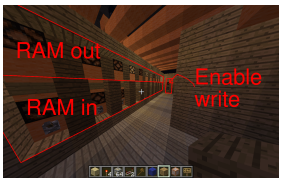
In figure 11a you see two switches. The reset switch and the programming/run

switch. The reset switch sets all registers to zero. It will in other words restart the machine. It will not reset the RAM though. The other switch is usefull when you would like to edit the contents of RAM. This will disconnect the RAM from the CPU it will also open the door to the write door.

In the write room you can read or edit the contents of RAM. Before you go into the write room make sure you have paused the machine and switched the programming/run switch to programming. This will open the door to the write room. As previously mentioned you can edit the contents of RAM in this room. To do this you should specify the RAM address you want to edit: figure 11b. You can see what is currently in that address in the RAM out and can store something at that address with the switches of the RAM in. Additionally you have to switch the enable write switch and you should see see the value you've put in the RAM come in the RAM out after a while. When this is the case you know that you have succesfully written to RAM. Make sure to put switches to an off state when you leave the room, otherwise the signal would interfere with that of the CPU which will cause unpredictable behaviour.

(a) Door to the write room.

(b) Here you specify the address.



(c) Here you write to RAM.

7 Writing software

7.1 Writing techniques

The most straitforward way of writing software for this machine might seem to just put in the program with the switches in the read/writing room and then debugging it and edit it as you go. This is inefficient, as it is easy to loose the big picture as it can be hard to read the code in binary. Debugging the program on the minecraft computer is a verry slow process including the fact that when you forgot to write one instruction say in address 123 you must move up all the instructions that come after it which all takes a long time. In this section we will discuss how we can program the machine more efficiently.

7.2 Assembler

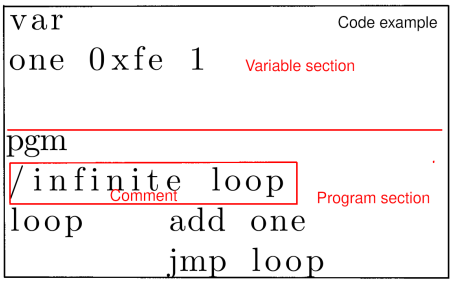
An assembler converts assembly language into machine code (binary instructions). Assembly language is a way of making machine code more human-readable. Instead of binary it uses names for the different instructions. It also

uses variables to make it easier to read the specified addresses. This is usefull, because if you change the position of a certain variable or instruction. You don't have to look through all the instructions and change everything manually, but you can just change the position of the variable and your finished.

For this purpose I wrote an assembler which you can download from my website at https://www.kznarsk.xyz/abterra/assembler.py. If you use Mac os, linux or any other unix based operating you should go to the directory of the assembler in your terminal and execute the commnand: "python assembler.py". On windows it could be a bit more complicated. Read this article to learn how to run a python file on windows: https://www.wikihow.com/Use-Windows-Command-Prompt-to-Run-a-Python-File. If you use any other operating system you are probably smart enough to figure it out yourselve.

Before you can assemble your first program you first have to write the assembly code. You can write this in any text editor like vim, emacs or notepad++. The first thing you need to understand is that you have different sections in the code. One is named "var" and another is called "pgm". You start a section by making a line with the word "pgm" or "var" on it. Every line under the section specifier "pgm" or "var" whill be part of that type of section. Untill another specifier is found.

Commments are usefull to explain your code, so you or someone else can understand how your code works. You make a comment by placing a '/' a comment by writing a '/'. All text that is on the same line and comes after the '/' shall then be ignored.



In the pgm section the user makes a list of instructions. The uppermost instruction shall be mapped with address zero and the second uppermost shall be mapped with address one and so on. If you have to many instructions the assembler shall give an error message as there are only 256 addresses in the RAM of the computer. You can only write one instruction per line, but you are allowed to keep lines empty. In most cases you must start your instruction with the name of the instruction. The possible instruction names are:

- load
- store
- add
- neg
- NOT
- AND
- jmp
- skip

These obviously correspond with the 8 instructions that the computer has. For the instruction load, store, add, neg, NOT and AND You have to specify an address. There are several ways of specifying an address. You can give the name of variable specified in the "var" section or an instruction label specified in the "pgm" section. It is important that the variable is defined before the instruction, but the instruction label doesn't need to be defined before the instruction. The other option is to specify the address. You would mostly specify the address in decimal. You can also specify it in hexadecimal by starting the value with "0x" or in binary by starting with "0b".

```
1  /code example
2  var
3  variable 255 12
4
5  pgm
6  begin    load variable/a variable
7           add 255/a decimal number
8           store 0xff/a hexadecimal number
9           neg 0b11111111/a binary number
10          jmp begin/an instruction label
```

The instruction jmp also requires an address to be specified, but this would mostly be an instruction instead of a variable. To do this efficiently you can use instruction labels. These are similar to variables as they specify an instruction, but different because they are linked to an instruction instead of a variable. To make an instruction label you can start an instruction with the name of the label instead of an instruction. It is required that the label doesn't have the name of any of the instructions.

The instruction skip is a more special instruction. You specify the letter that corresponds to the condition. Here is the list of all conditions with the corresponding letter.

- s bitselect
- z zero
- c carry

If you choose "s" or "bitselect" you must also choose which bit you want as condition. You have to do this in decimal by giving a value between 0 and 7.

```
1 /code example
2 pgm
3
4 begin   skip z/skip if accumulator is equal to zero
5         skip s 4/skip if bit 4 is 1
6         skip c/skip if carry register is 1
7         skip s 9/skip if bit 9 is 1
8         jmp begin/jmp to instruction 0
```

In the var section you give a list of variables. You can only create one variable per line. You start by giving the name of the variable. After this you give the position of the variable in RAM. You can specify this in binary, hexadecimal or decimal. Finally you should specify which value should be stored in that variable when the program is assembled which is called "the starting value".

```
1 /code example
2 var
3 /name          /position       starting value
4 Chickens       0xff            23
5 Hens           0xfe            0xa
6 Humans         253             0b11
```

When you run the assembler it will ask you to specify the file you want to assemble. It then asks you to specify the name of the file it will output to. It will assemble the specified file whereafter it will either give an error message which means you have to fix something or it will just stop running which means it has succesfully assembled. If it has succesfully assembled two new files should appear in the current directory. One with the specified output filename and one with the specified output filename and " logisim" appended to it. The first file is a list of the instructions in the program and the second one can be run in logisim (see section 7.3).

7.3 Logisim

Debugging the machine in minecraft is very time consuming. That is why I made a clone of this machine in logisim. Logisim is a digital logic simulator and the computer is a lot faster in logisim than in minecraft. I therefore advice you to debug your code in logisim. First you install logisim and then download

the logisim file here: https://www.kznarsk.xyz/abterra/Abterra%20I.circ. You open that file in logisim and you should see something like in figure 12.

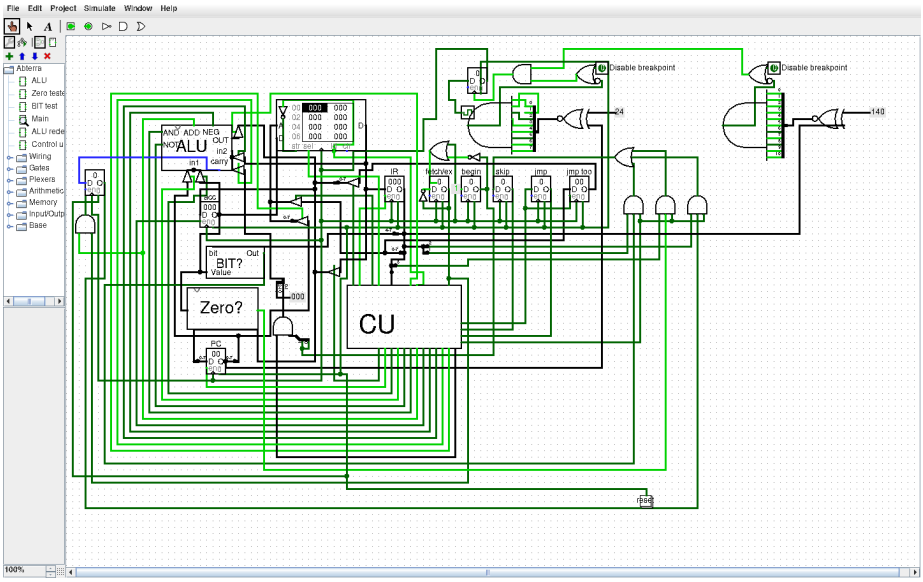
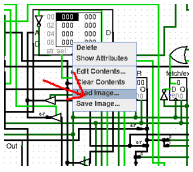


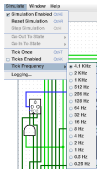
Figure 12: Abterra I in logisim

To load in the program you should right-click on the RAM wherafter a menu will pop up where you should to select “Load Image”. You should then select the assembled file with “ logisim” appended to it. To then run this program first make sure to click on the “reset” button at the bottemleft of your screen to make sure all the registers are set to zero. Then click on “simulate” on the top left of your screen wherafter you should click on “Ticks Enabled”. This will make the clock start ticking. You can change the frequency of the ticking in “Tick Frequency”. If you have ticks disabled you can manually tick the clock by pressing “Tick Once” By reading the values in the registers and enabling and disabling the clock you are able to debug the machine. If you have ever debugged a program before, you might have heard of the term breakpoint. a breakpoint is a place in code you define where the computer should stop executing. This is interesting when you want to debug a specific section section of the code, but you don’t want to stop the computer manually at a certain point. To make a breakpoint turn of the disable breakpoint pin by pressing on it. There are two two breakpoint sections in this computer. One is called “PC break” and will stop the clock when the machine is at a certain instruction. The other one can be connected to any register in the coputer, but it requires knowledge about logisim to use it properly so we won’t discuss it here. To set the address of the breakpoint press ctrl-2 to go into edit mode and click on the constant seen in figure 13c and change its value in the selection menu. If the pc register has that

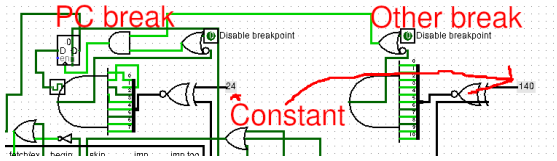
value the clock will be disconnected from the computer, to connnect it again disable the breakpoint or change the break address.



(a) The PC register and accumulator



(b) Tick settings



(c) Set breakpoint

7.4 Loading program

After you have written the program in assembly language, assembled it and debugged it in logisim it is time to load it in the minecraft computer. There is no way of doing this automaticly so you will have to plug in every instruction manually as described in section 6. You can use the output file of your program to easily see the binary of every instruction.

7.5 Example programs

7.5.1 Increment loop

This is a very simple program that just infinitely increments the accumulator.

```
1 var
2 one 0xfe 1
3
4 pgm
5 /infinite loop
6 loop    add one
7         jmp loop
```

7.5.2 Sorter

The next program is a more complex program that sorts a list of values. These values must be between 0x600 (same as -512) and 0x1ff (same as 511). You can make the list of values anywhere in RAM. You define the position of the list by setting address 0xff (same as 255) and 0xfd (same as 253). In address 0xff you set the position of the first element of the list and and in address 0xfd you set the address of the last element of the list. You know that the program has finished running when the program counter constantly stays at the same address.

There are more efficient sorting algorithms than this one it is just an example. This program starts at the first item in the list where it will find the smallest value in the list and move that too the first item. Then it will go too the second item where it will find the second smallest value and so forth untill it hits the end of the list every time having to look for less and less values as it knows that the smaller values are already behind it.

```
1  var
2
3  Beginlist        255     0
4  NEndlist         254     0/negative of endlist
5  Endlist          253     0
6  tmp              252     0
7  i                251     0/counter
8  ii               250     0/counter
9  one              249     1
10 Store            247     0x100
11 Neg              246     0x500
12 Add              245     0x400
13
14 pgm
15         load Beginlist
16         store i
17         neg Endlist
18         store NEndlist
19
20         /checks if i<NEndlist and ii=i+1
21         load i
22 loop1   add NEndlist
23         skip s 10
24         jmp end
25         load i
26         add one
27         store ii
28
29 /Change address of instructions to i
30         load Store
31         add i
32         store StoreI
33
34         load i
35         store LoadI
36
37         load Add
38         add i
```

```
39          store AddI
40
41          /checks if ii<NEndlist
42          load ii
43  loop2   add NEndlist
44          skip s 10/check if negative
45          jmp IncrI/ii>Endlist
46
47          /Change address of instructions to ii
48          load Store
49          add ii
50          store StoreII
51
52          load Neg
53          add ii
54          store NegII
55
56          load ii
57          store LoadII
58
59
60
61          /compare the value at i with the value at ii
62  NegII Skip2     neg 0/loads value at ii
63  AddI    add 0/add the value at i
64          skip s 10
65          jmp swap/*i>*ii swap
66  Return  load ii/*ii>*i /no action required
67          add one
68          store ii
69          jmp loop2
70
71  LoadII swap load 0/loads value at ii
72          store tmp
73  LoadI   load 0/loads value at i
74  StoreII store 0/stores value at ii
75          load tmp
76  StoreI  store 0/stores value at i
77          jmp Return
78
79  end     jmp end/infinite loop
80
81  IncrI   load i
82          add one
83          store i
84          jmp loop1
```

In the lines 30-39 and 48-57 we can see the advantages of having a program stored in RAM. The program reads from and writes too a list. Every time it wants to read or write from a different element from this it changes the address of the instruction. This is something that can't be done in computers where the program is stored in ROM. An alternative way of accessing elements of lists is by using a stack, but this would add more complexity to the machine which is something I wanted to avoid.