**Table of contents**

1. [**Memory**](#Memory) **..................................................................... 2**

**1.1.** [**Program**](#Program) **......................................................................... 3**

**1.1.1.** [**Program**](#Program) **.................................................................. 3  
1.1.2.** [**MemoryChip128B**](#MemoryChip128B) **................................................... 6  
1.1.3.** [**ByteCell**](#ByteCell) **…….……………………………………………………….. 11**

**1.2.** [**MyRegisters**](#MyRegister) **................................................................. 12**

1. [**Computing**](#Computing) **…………………………………………………………… 13**
   1. [**ALU**](#ALU) **............................................................................ 14**

**2.1.1** [**ALU**](#ALU) **…………………………………………………………………. 14**

**2.1.2** [**Full8BitAdder & Full8BitSubtractor2sC**](#AdderSubtractor) **…………….. 16**

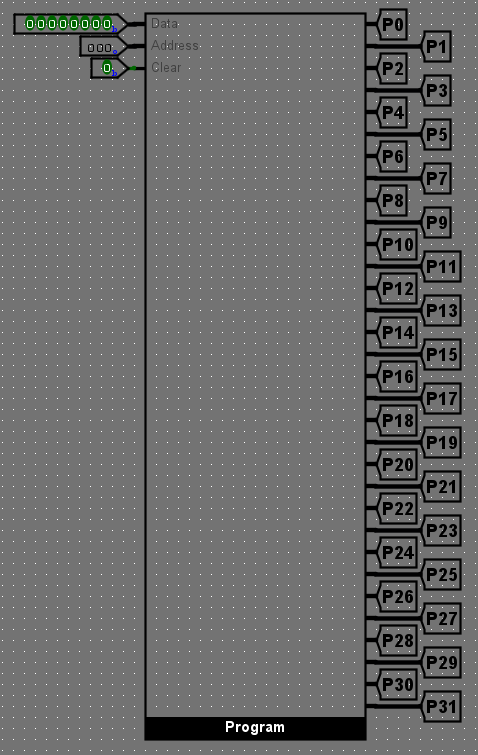
**2.1.3** [**FullAdder**](#FullAdder) **………………………………………………………… 18**

1. **Code**
   1. [**Instructions**](#Instructions) **……………………………………………………………... 20**
   2. **Code parsing .............................................................. 2**

**Memory**

**Program and Registers**

**Program**

****

The “Program” circuit represents the main memory of the computer. Its space is 256B and allows for the storage of a code made out of up to 256 instructions. It has 3 inputs, “Data”, “Address” and “Clear”.

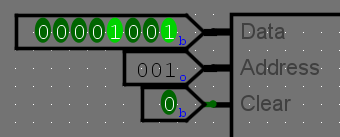
The “Data” input is used for entering the instruction code.  
 The “Address” input is used for entering the address in the memory where the current instruction will be stored.   
 The “Clear” input is used for wiping the entire memory clear, resetting each bit memory cell to 0.

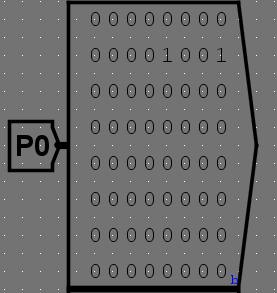
All of the instructions stored in the memory are synchronously displayed at all times.

**Instruction feeding**

The instructions are fed into the memory by first selecting the address at which the instruction should be stored by entering the address in the “Address” input in octal, followed by then typing the instruction code in the “Data” input in binary.

Example: Storing the instruction “00001001” at address “0018” (110)



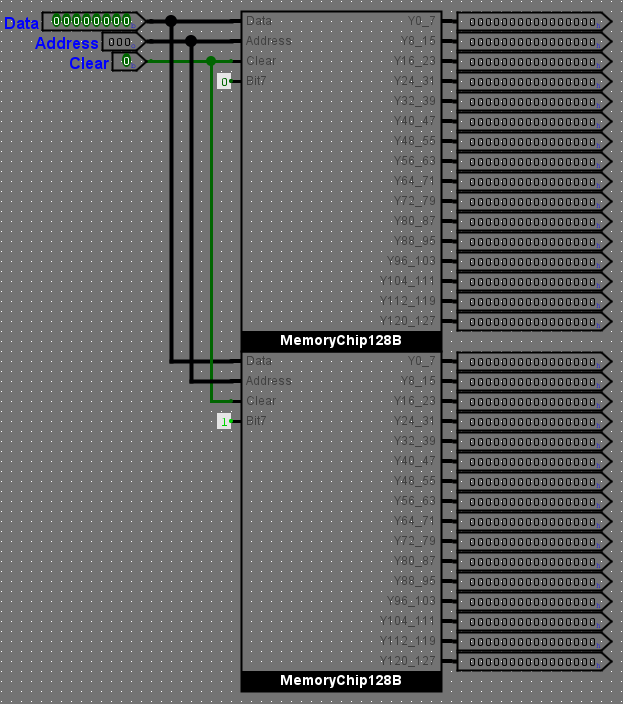


The instructions are automatically written into the memory, so modifying the address first is crucial so as to not overwrite the instruction present at the currently accessed address. The reason as for choosing an octal numbering system for inputting the address is explained in the “Display” part of the manual.

The first address in the memory is 0008 (010), and the final address is 3778 (25510).

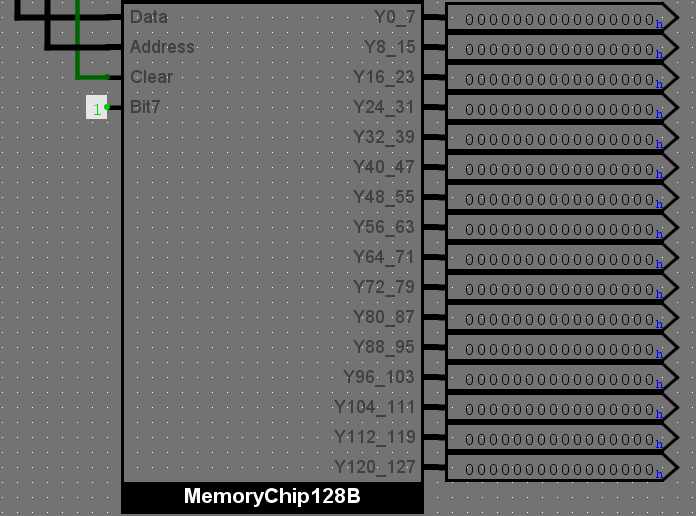
**Internal structure**

Internally, the “Program” circuit is composed of two “MemoryChip128B” circuits, its data and address buses are both 8bit and its clear bus is 1bit.



For details regarding the number of outputs, see [OBSERVATION 1](#OBSERVATION1).

**MemoryChip128B**



The “MemoryChip128B” circuit represents one of the two memory chips of the computer. Its space is 128B. It has 4 inputs, “Data”, “Address”, “Clear” and “Bit7”.

* The “Data” input is used for entering the instruction code.
* The “Address” input is used for entering the address in the memory where the current

instruction will be stored.

* The “Clear” input is used for wiping the entire memory chip clear, resetting each bit memory cell to 0.
* The “Bit7” input is used for indexing the chips and addressing.

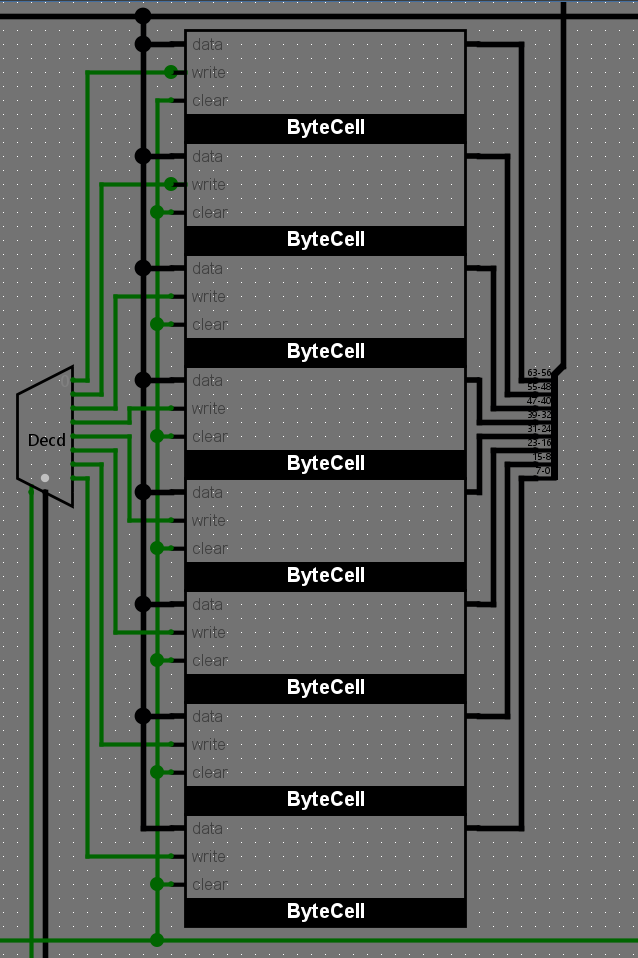
For the first chip, “Bit7” is 0. For the second chip, “Bit7” is 1.

If the 7th bit of the address has the same logical value as the “Bit7” input, then the value from the “Data” input will be stored it that particular chip.

**Internal structure**



The *MemoryChip128B* is comprised of 16 groups of 8 *ByteCell* circuits.

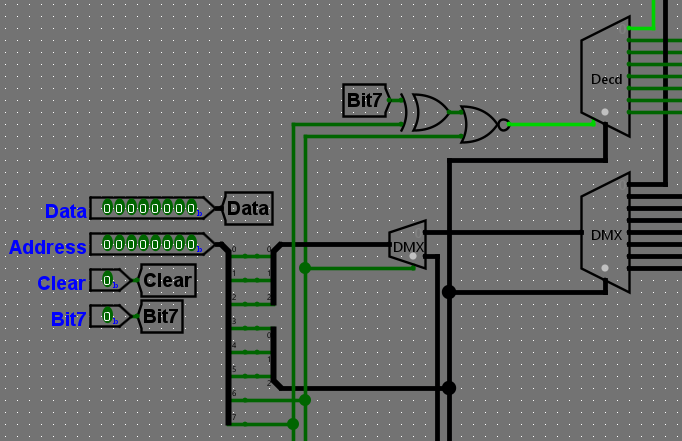


**The Data bus**

|  |  |
| --- | --- |
|  |  |

The data bus is connected directly to every *ByteCell*. The data is stored only if the write input is enabled.

**The Address bus**



The address bus is used for directing the signal which enables writing to the right *ByteCell*. The write enable signal is transmitted through 3 layers of demultiplexing.

The first step is determining if the address corresponds to the current chip. This is achieved by comparing the 7th bit in the Address with the value the chip receives from Bit7.



Using a XOR gate, we obtain the parity of 1s, outputting 0 when they are identical and 1 when they are different. By negating the output, it will now output 1 when it’s inputs are identical.

If the output of the XNOR is 1, it means that the current chip is the correct one.

The first layer is comprised of 8bit logical gates with the 6th and 7th bit as inputs to determine the right row of *ByteCell* groups.

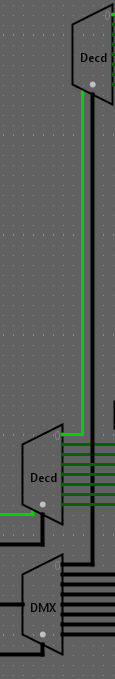
|  |  |
| --- | --- |
|  |  |
|  |  |

When the XOR = 0 and 6th bit = 0, the NOR = 1 and a positive enable is sent to the 1st row of *ByteCell* groups.  
 When the XNOR = 1 and 6th bit = 1, the AND = 1 and a positive enable is sent to the 2nd row of *ByteCell* groups.

The second layer is comprised of a 3bit demultiplexer and a 3bit decoder, which use the 3rd, 4th and 5th address bits to determine the right group of *ByteCell* circuits in a row. The demultiplexer transmits the last 3 bits of the address (0, 1, 2) and the decoder transmits and enable signal to that group’s decoder.

The final layer is comprised of a 3bit decoder, which uses the 0th, 1st and 2nd address bits to determine the right *ByteCell* in a group and transmits the enable signal to the write input of the *ByteCell*.

3,4,5 0,1,2 3,4,5 1st layer



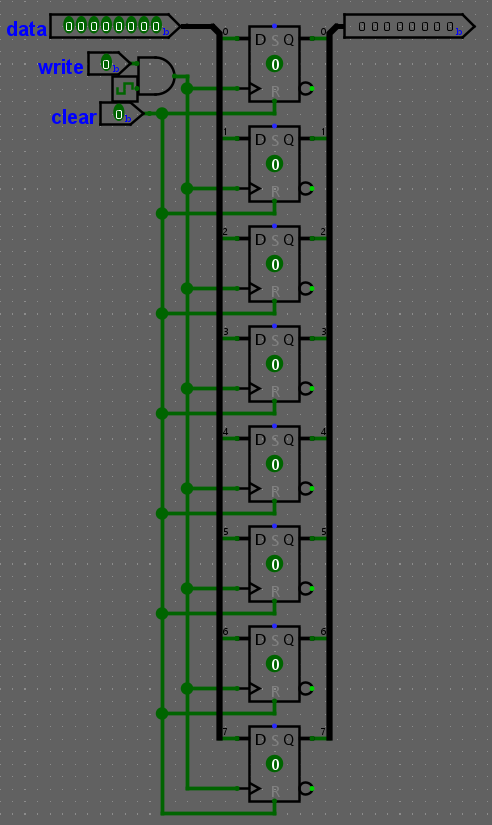
**Clear input**

The clear cable is directly connected to the clear inputs of every *ByteCell*.

**Output**

All the data stored in the *ByteCell*s is outputted at all times, sent through a splitter and grouped into 16 x 64bit data busses, connected to 16 x 64bit outputs.

**ByteCell**

****

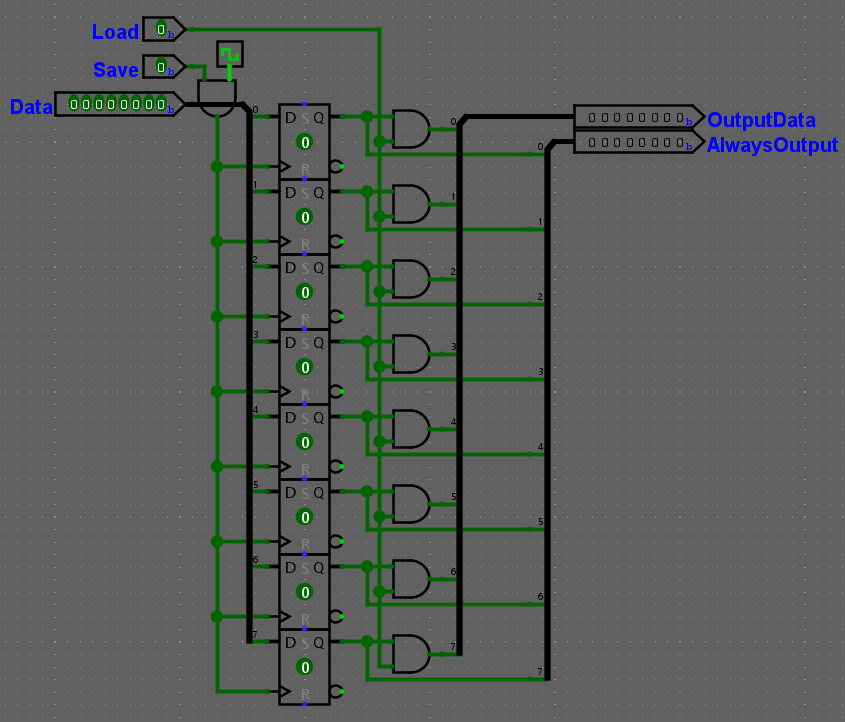
**The *ByteCell* is made out of 8 D Flip-Flops. The data bus goes through a splitter and enters the D input of the Flip Flops.**

**The clock is connected to an internal clock, which goes through an AND gate with the write signal. The AND gate functions as a switch. When *write* is on, the clock is allowed to pass and reach the Flip Flops, updating the values stored in them.**

**The clear input is connected to the Reset of every Flip Flop.**

**The data stored in them is outputted without stop.**

**MyRegister**

****

**The *ByteCell* is made out of 8 D Flip-Flops. The data bus goes through a splitter and enters the D input of the Flip Flops.**

**The clock is connected to an internal clock, which goes through an AND gate with the write signal. The AND gate functions as a switch. When *write* is on, the clock is allowed to pass and reach the Flip Flops, updating the values stored in them.**

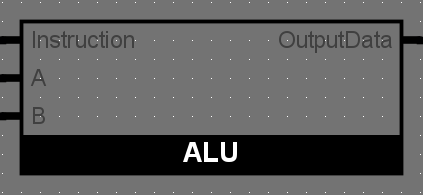
**The clear input is connected to the Reset of every Flip Flop.**

**The data stored in them is outputted without stop though the *AlwaysOutput* and through *OutputData* only when the *Load* input is on. The AND gates function as switches.**

**Computing**

**ALU**

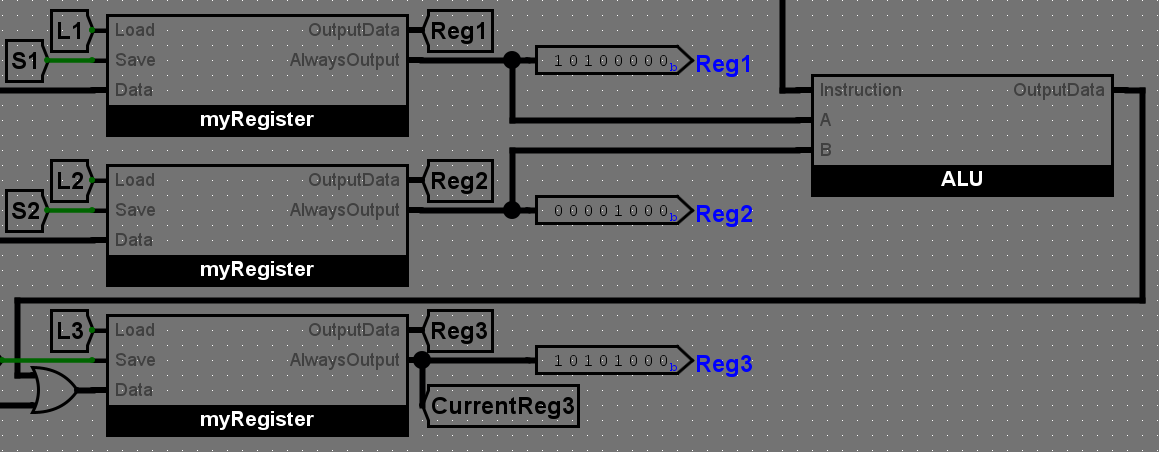
**ALU**

**

*ALU* stands for Arithmetic Logic Unit and it is used for OPERATIONAL instructions. Its purpose is to perform basic arithmetical and logical operations on two given values (in this computer, the ones stored in *Register1* and *Register2*), without modifying the given data and sending the result to *Register3*.

**Operating mode**

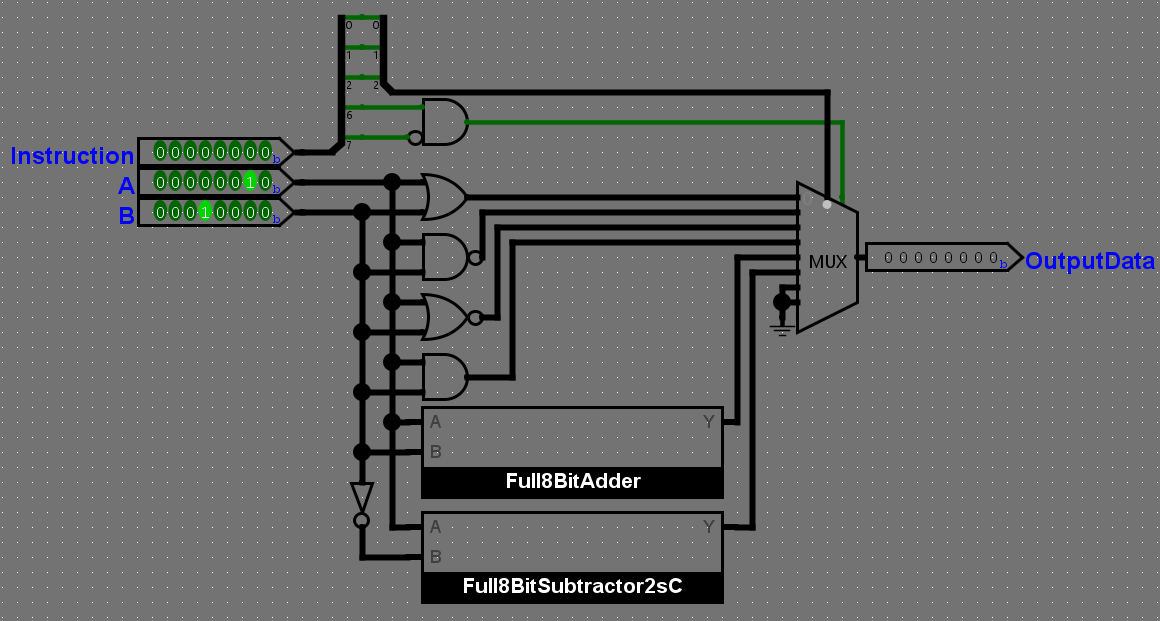
The *ALU* receives The *Instruction Decoder* interprets the OPCODE and sends the information to the *ALU*. The *ALU* takes the values from *Register1*’s and *Register2*’s *AlwaysOutput* outputs, performs *Addition* on the values, uploads 000001112 (710) to the data bus and turns on the *Save* input for *Register3*, allowing the result to be stored into the register.

**

The *ALU* used for this computer is rudimentary. It performs 6 different 8bit operations at the same time. The results are outputted into a Multiplexor which further transmits only one of them to the Data bus, based on last 3 bits of the instruction:

* 000 -> OR ( *Register3*  = *Register1* OR *Register2* )
* 001 -> NAND ( *Register3*  = *Register1* NAND *Register2* )
* 010 -> NOR ( *Register3*  = *Register1*  NOR *Register2* )
* 011 -> AND ( *Register3*  = *Register1* AND *Register2* )
* 100 -> ADD ( *Register3*  = *Register1* + *Register2* )
* 101 -> SUB ( *Register3*  = *Register1* - *Register2* )

For OR, NAND, NOR, AND, 4 8bit logical gates are used.   
For ADD and SUB, 2 circuits, [*Full8BitAdder* and *Full8BitSubtractor2sC*](#AdderSubtractor) are used.



Numbering systems

This *ALU*’s result’s numbering system depends on the performed operation:

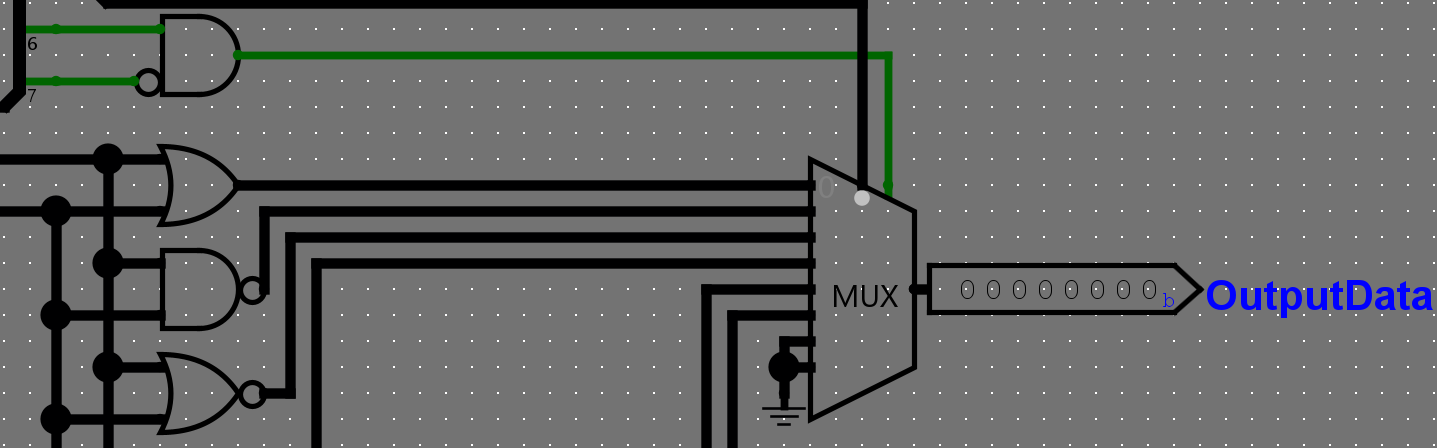
* OR, NAND, NOR, AND: the result can be interpreted in all numbering systems
* adding: the result can be interpreted in all numbering systems
* subtraction: the result must be interpreted in signed 2’s complement form only

Operating ranges

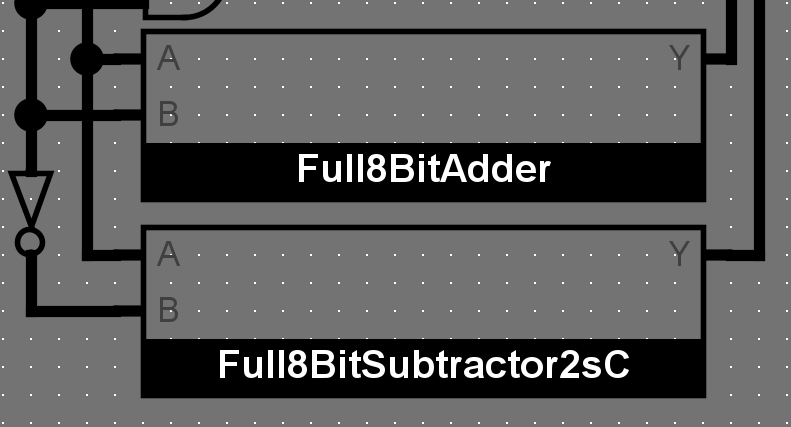
* Overflow when adding two numbers results in a number greater than 255
* Overflow when subtracting two numbers results in a number outside the range [-128, 127]

See [OBSERVATION 2](#OBSERVATION2)

While the computer is not performing any OPERATIONAL instruction, the ALU is not receiving any signal from the *Instruction Decoder*, meaning that the OPCODE of the instruction received by *ALU* is 00. In order not to influence the Data bus when the *ALU* should be inactive, a switch (AND gate) is used to enable/ disable the MUX which outputs the result. The enable is ON only when the OPCODE is 01.

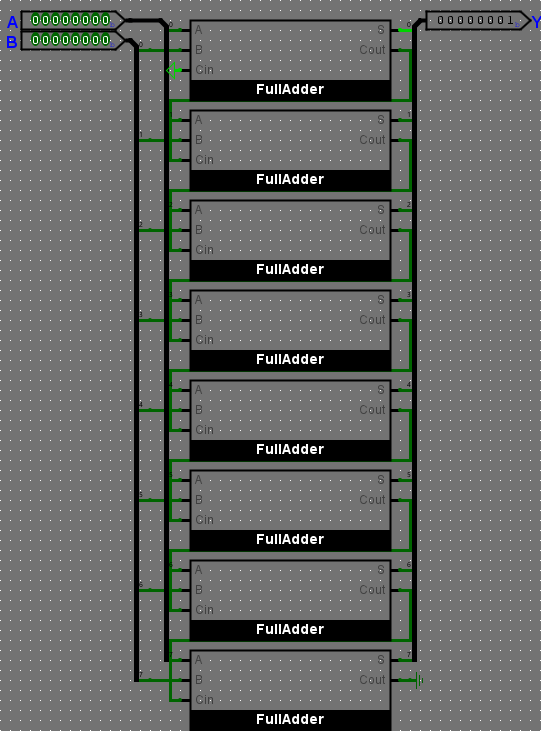
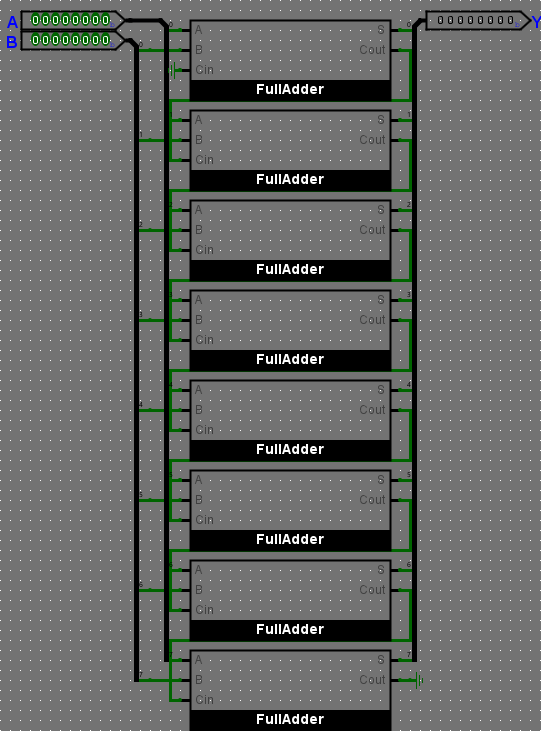


**Full8BitAdder & Full8BitSubtractor2sC**

****

Full8BitAdder and Full8BitSubtractor2sC have similar functions.

Both circuits receive 2 8bit numbers and add them using 8 1bit *FullAdder*s.



|  |  |
| --- | --- |
| **Full8BitAdder** | **Full8BitSubtractor2sC** |

The *Full8BitAdder* simply adds the two values.

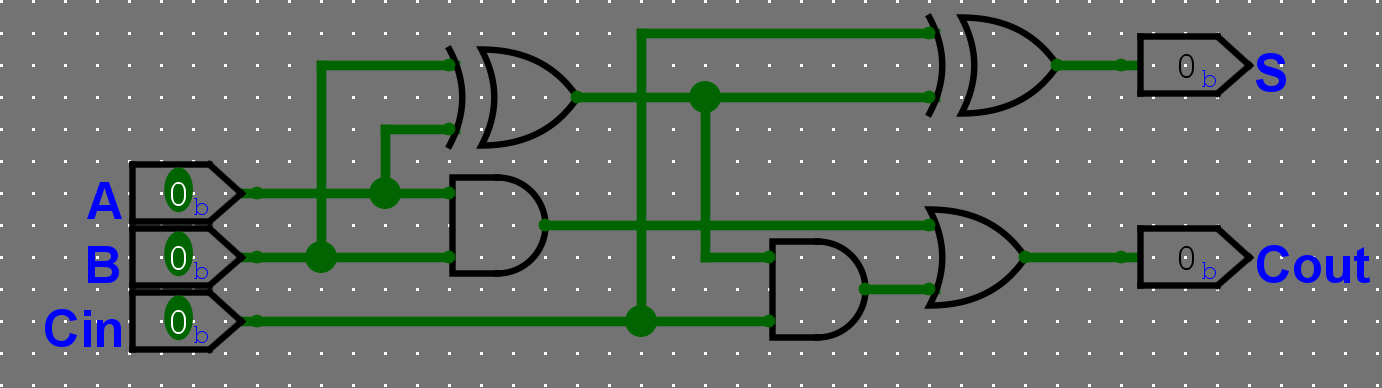
The *Full8BitSubtractor2sC* adds the first value with the reversed value of the second number (the value passes through an 8bit NOT gate in the ALU before reaching the input) and then adds 1 at the end (Cin of the LSB adder is connected to power).

Subtraction is achieved by adding the minuend and the 2’s complement form of the subtrahend.

( Difference = Minuend + 1’s complement form of Subtrahend + 1 )

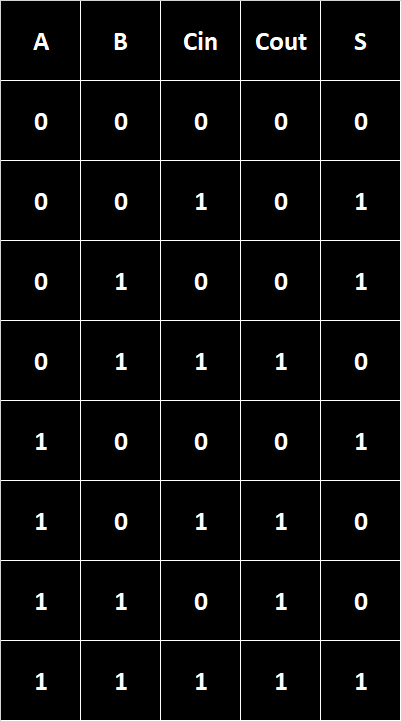
See [Operating Ranges](#OperatingRanges)

**FullAdder**

****

***FullAdder* is the standard 1bit Full Adder.**

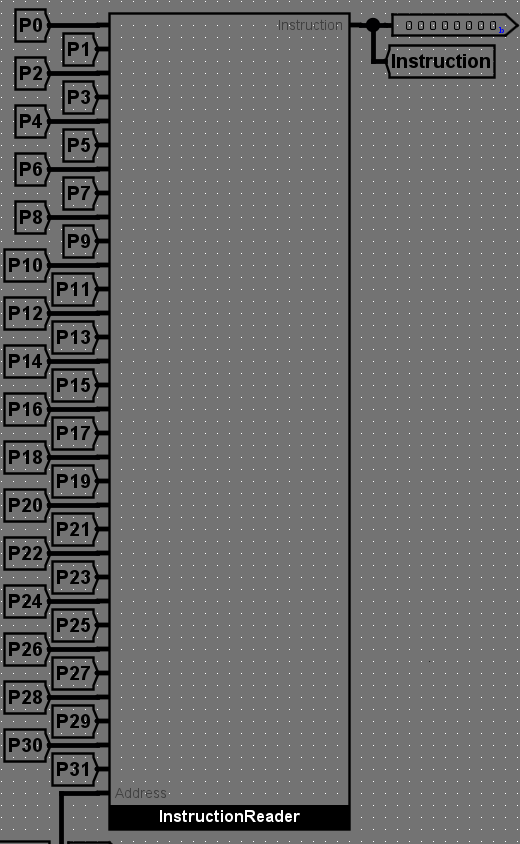
**Truth table**



**Code**

**Instructions, InstructionReader & IP**

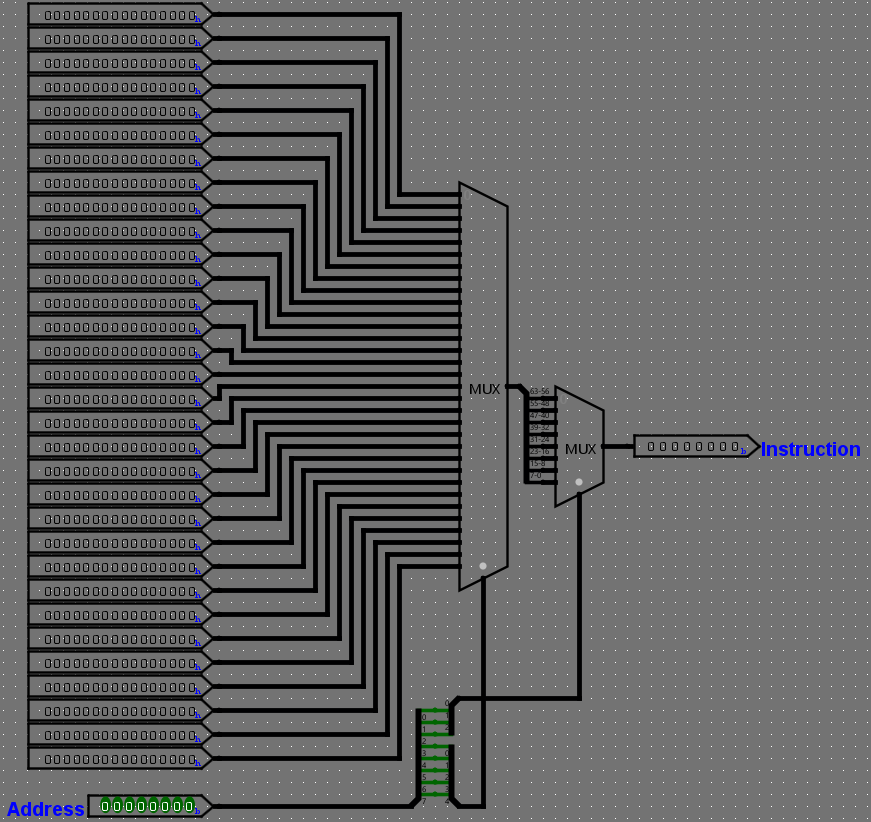
**InstructionReader**

****

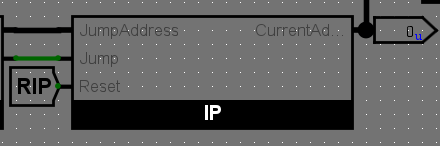
The *InstructionReader* and *Program* circuits would usually be a single circuit. The program should output a single data value, but as I wanted it to output every value at once for the display and readability of the lines of code for the user, they were split into two different components. The instruction reader has full access to every memory bit simultaneously and outputs only one of them at a time, depending on the Address received from the [IP](#IP).

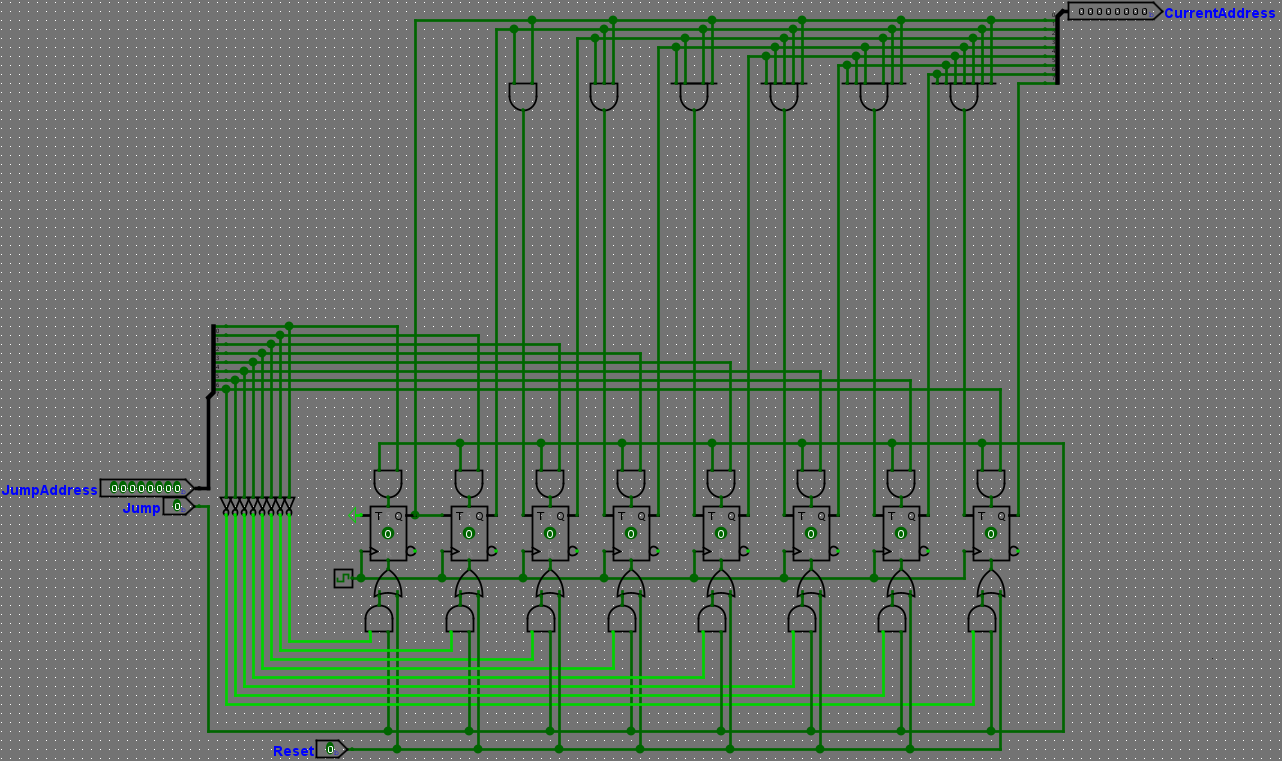
**Internal structure**

The *InstructionReader* is an 8-bit multiplexer. The first 5 bits of address (7-3) are used to multiplex the value from the correct *ByteCell* group and from the correct *MemoryChip128B*. The final 3 address bits (2-0) are used to select the data from right *ByteCell*, a single 8bit word.



**IP**

****

The *IP (Index Pointer)*  is a 8bit T Flip-Flop synchronous up counter. ****

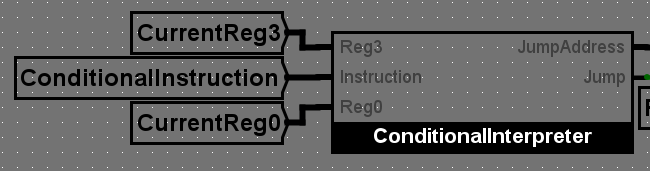
The JumpAddress input has the value currently stored in *Register0*. The value is sent to the Flip-Flops’ Set and Reset inputs. In order for the JumpAddress not to disturb the functionality of the up counter when a jump is not required, a second jump input, Jump, is used as an enable, along AND gates which act as switches. The current states of the Flip-Flops are outputted without restrictions.

The Jump and JumpAddress signals comes from the *ConditionalInterpreter*. If Jump is 1, the values from JumpAddress reach the Flip-Flops, otherwise, the values get stuck at the AND gates.

The purpose of the *IP* is to give the *InstructionReader* addresses to parse in an ascending order, starting from 0. While the Reset input transmits 1, the counter resets and gets stuck at 0.

**ConditionalInterpreter**

The *ConditionalInterpreter* circuit is used for lines jumping if a certain condition is met.

****

This circuit does not require an enable as the forwarding of the instruction is already set to 00000000 by the InstructionDecoder if the instruction does not begin with 11.

Based on the last 3 bits in the instruction, the comparations results are multiplexed and outputted as Jump, an enable for the *IP* to jump to a line given in JumpAddress.

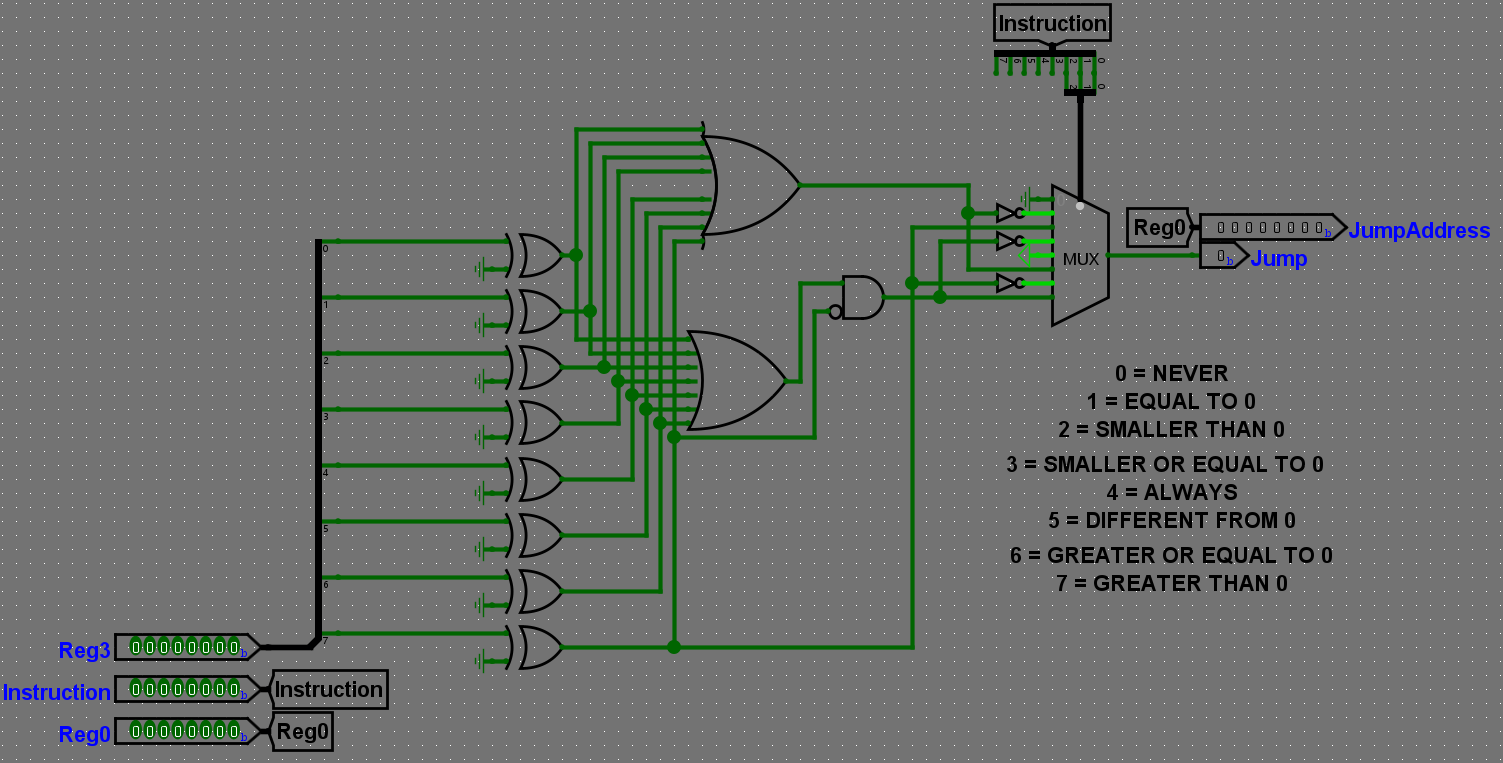
The JumpAddress is the value stored in Register0.   
 Values from 0 to 63 can be inputted directly into Register0 from the memory. Values included in the range 64-255 cannot be inputted directly from the memory. The user must copy them from another register into Register0. These values can be computed through addition or intentional overflow.

See [OBSERVATION 2](#OBSERVATION2)

The value present at that moment in *Register3* is compared to 0. Each bit of the register is first sent through and XOR gate along with a 0 bit. Initially, this had a purpose. Currently, it just is an unnecessary buffer, but it looks cool so I kept it.

**0. NEVER:** Ground connection. **ALWAYS 0**  
**1. EQUAL TO 0:** NOR of every bit. **CO of 5**  
**2. SMALLER THAN 0:** The 7th bit is 1. **CO of 6**  
**3. SMALLER OR EQUAL TO 0:** None of the 0-6 bits is 1 OR the 7th one is 1. **CO of 7**   
**4. ALWAYS:** Power connection. **Always 1**  
**5. DIFFERENT FROM 0:** At least 1 bit is 1. **CO of 1**  
**6. GREATER OR EQUAL TO 0:** The 7th bit is NOT 1. **CO of 2**  
**7. GREATER THAN 0:** At least 1 of the 0-6 bits is 1 AND the 7th one is NOT 1. **CO of** **3**

**CO of N** = “**C**omplete **O**pposite **of N**”

****

**Instructions**

The instructions which can be provided to the *Program* are stored in bytes. Each instruction is exactly 1 byte long. The first two bits contain the OPCODE of the function and the other 6 bits contain the parameters of the function, which can either be addresses or data, depending on the function type.

**Instruction structure**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| X7 | X6 | X5 | X4 | X3 | X2 | X1 | X0 |

The first 2 bits, X7 and X6, contain the opcode  
The last 6 bits, X5 – X0 contain the operands

**Instruction types**

Since the OPCODE is stored on 2 bits, there are 4 different OPCODEs that can be used:  
-00XXXXXX: IMMEDIATE mode  
-01XXXXXX: OPERATIONAL mode  
-10XXXXXX: COPY mode  
-11XXXXXX: CONDITIONAL mode

IMMEDIATE functions

An instruction starting with 00 will initiate the IMMEDIATE function, which will store the value represented by the other 6 bits ( 0 – 63 ) into the first register, Register0.

Example: 00010010 => Register0 will store the value 00010010 (18)

OPERATIONAL functions

An instruction starting with 01 will initiate the OPERATIONAL function, which will use the [*ALU*](#ALU) to apply one of 6 logical operations on the bits stored in Register1, Register2 and save the output in Register3.

OPERATIONAL functions:  
01XXX000 => Logical OR on Register1 and Register2, store in Register3  
01XXX001 => Logical NAND on Register1 and Register2, store in Register3  
01XXX010 => Logical NOR on Register1 and Register2, store in Register3  
01XXX011 => Logical AND on Register1 and Register2, store in Register3  
01XXX100 => ADD Register1 to Register2, store in Register3  
01XXX101 => SUBTRACT Register2 from Register1, store in Register3

This computer has only 6 functions for the OPERATIONAL mode and only the last 3 bits are used to determine the function used by the ALU. If the ALU receives an unknown operation (ending in 110 or 111), Register3 will store the value 00000000 (0).

Example: Suppose the values 000000112 (310) and 000001002 (410) are stored into *Register1* and *Register2*. The program receives the following instruction: 01000100.

The *ALU* performs *Addition* on the two values and stores 000001112 (710) in *Register3*.

COPY functions

An instruction starting with 10 will initiate the COPY function, which will move a value from a register to another, from the input, or to the output.

COPY functions:  
10000XXX => COPY from Register0 to ▮  
10001XXX => COPY from Register1 to ▮  
10010XXX => COPY from Register2 to ▮  
10011XXX => COPY from Register3 to ▮  
10100XXX => COPY from Register4 to ▮  
10101XXX => COPY from Register5 to ▮  
10110XXX => COPY from INPUT to ▮  
10XXX000 => COPY from ▮ to Register0  
10XXX001 => COPY from ▮ to Register1  
10XXX010 => COPY from ▮ to Register2  
10XXX011 => COPY from ▮ to Register3  
10XXX100 => COPY from ▮ to Register4  
10XXX101 => COPY from ▮ to Register5  
10XXX110 => COPY from ▮ to OUTPUT

Example: 10010110 => COPY from Register2 to OUTPUT

CONDITIONAL functions

An instruction starting with 11 will initiate the CONDITIONAL function, which takes the value stored in *Register3* and compares it to 0. If the condition is true, the *IP* takes the value currently stored in *Register0*.

CONDITIONAL functions:   
10000000 => IF FALSE  
10000001 => IF Register3 IS EQUAL TO 0  
10000010 => IF Register3 IS SMALLER THAN 0  
10000011 => IF Register3 IS SMALLER OR EQUAL TO 0  
10000100 => IF TRUE  
10000101 => IF Register3 IS DIFFERENT FROM 0  
10000110 => IF Register3 IS GREATER OR EQUAL TO 0  
10000111 => IF Register3 IS GREATER THAN 0

CONDITIONAL functions are used for jumps.   
 Example: 00000100 => STORE 4 in *Register*0  
 10000100 => IF TRUE jump to line 4

**Code examples**

**Adding 2 numbers from the input and outputting the result:**

10110001 => COPY from INPUT to Register1  
 10110010 => COPY from INPUT to Register2  
 01000100 => ADD Register1 to Register2, store in Register3  
 10011110 => COPY from Register3 to OUTPUT

**Multiplying 2 numbers from input and outputting the result:**

10111010 => 00: COPY from INPUT to Register4

10100101 => 01: COPY from Register4 to Register5

10111110 => 02: COPY from INPUT to Register6

10111001 => 03: COPY from Register6 to Register1

00000001 => 04: STORE 1 into Register0

10000010 => 05: COPY from Register0 to Register2

01000101 => 06: SUBTRACT Register2 from Register1, store into Register3

00001111 => 07: STORE 16 into Register0

11000011 => 08: IF Register3 IS SMALLER OR EQUAL TO 0, jump to 16

11011111 => 09: COPY from Register3 to Register6

10101001 => 10: COPY from Register5 to Register1

10100101 => 11: COPY from Register4 to Register2

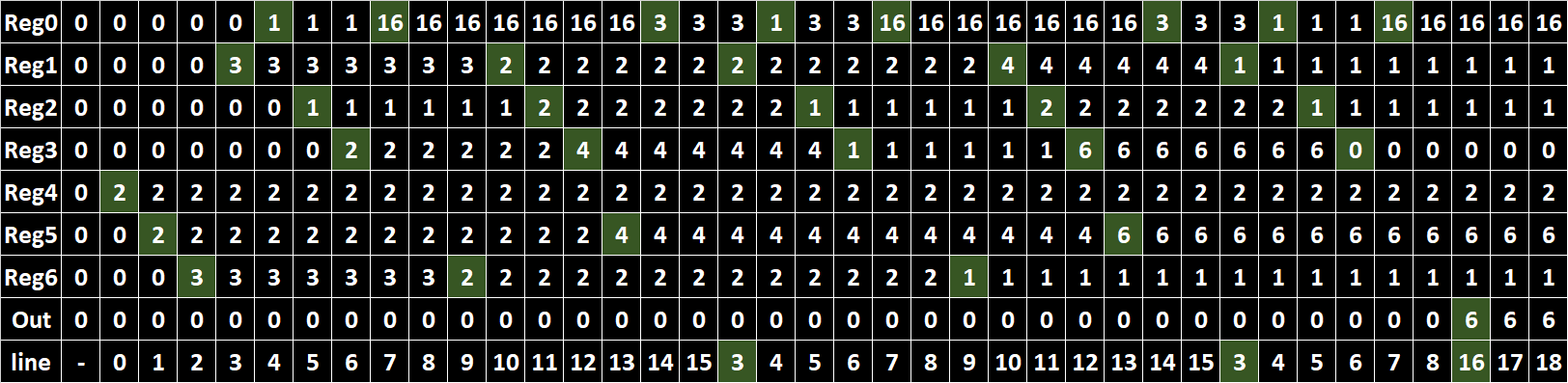
01000100 => 12: ADD Register1 to Register2, store into Register3

10011110 => 13: COPY from Register3 to Register 5

00000100 => 14: STORE 3 into Register0

11000100 => 15: IF ALWAYS jump to 3

10101110 => 16: COPY from Register5 to OUTPUT



2 x 3 = 2 + 2 + 2 = 6

**OBSERVATIONS**

OBSERVATION 1:

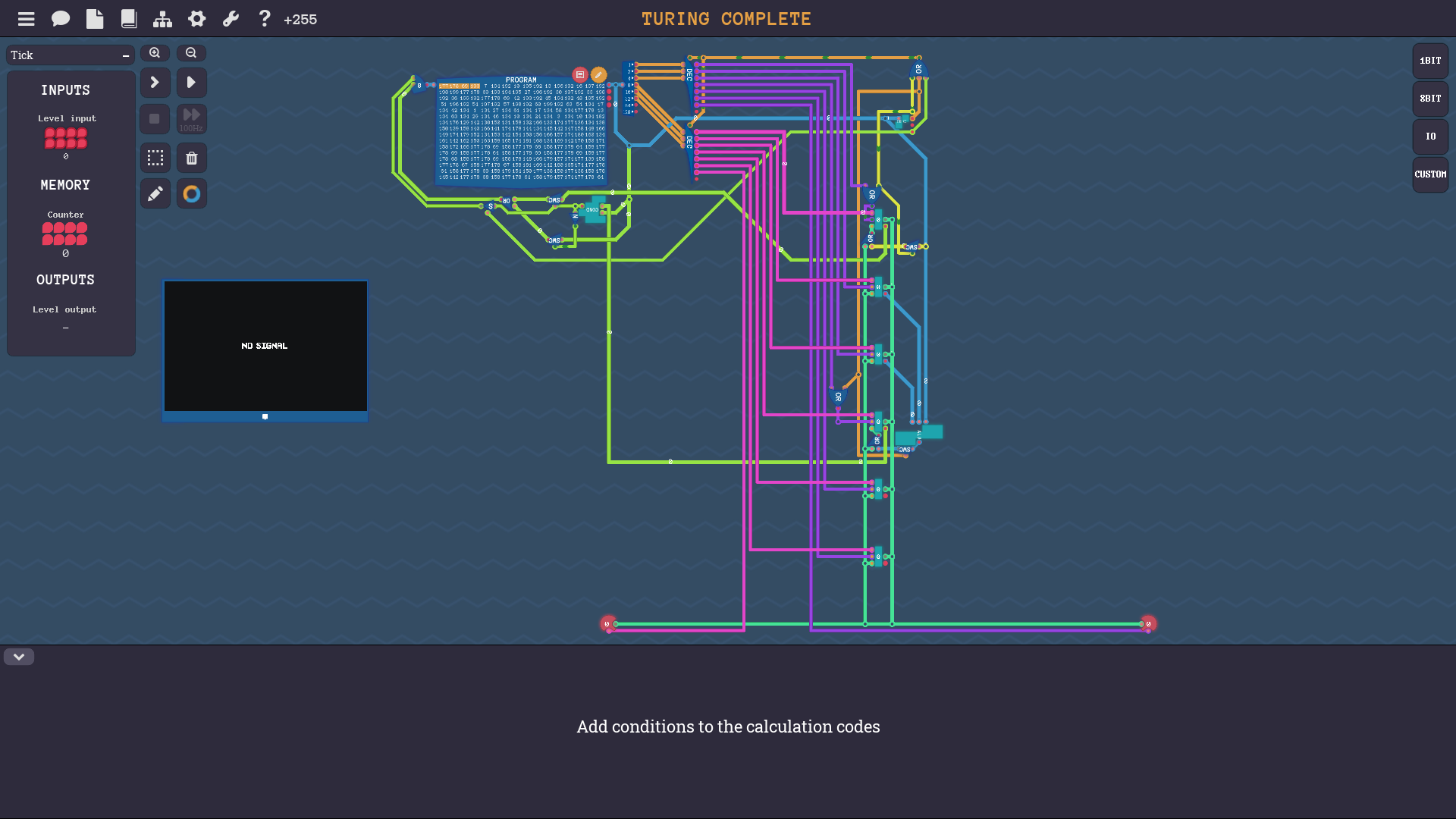
Due to Logisim-evolution v3.8.0 having a maximum number of 64 bits in a bus and the “Program” circuit having a space of 256B, 32 x 64b buses are required for a simultaneous output of all the instructions stored in the memory.

[Go back…](#OBSERVATION1goBack)

OBSERVATION 2:

This computer does not throw exceptions or halt in case of errors, such as overflows.   
[Go back…](#OperatingRanges)

OBSERVATION 3:



While this project is heavily inspired by a computer made in a virtual game, I chose to go forth and consider this circuit as just inspired and not copied due to the following:

* The memory part of the computer was not presented in the video game. It came as a prebuilt component. Only the structure of the memory (aligning flip flops on rows and columns) was inspired from a slide from the course.