PROJECT	EP1
TOPIC CODE	FI1AMP175
STUDENT NAME	David Jøssang
SP1 TOPIC / PROJECT NAME	**Simple CPU Architecture**

Table of Contents

1. Introduction **2**
2. Main Part **2**
2.a Assumptions 2
2.1 Planning 3
2.1.1 General Structure 3
2.1.2 Microinstruction Planning 4
2.1.2.1 Microinstructions 4
2.1.2.1.1 Arithmetic Instructions 5
2.1.2.1.2 Read Instructions 5
2.1.2.1.3 Write Instructions 5
2.1.2.1.4 Special Instructions 5
2.1.2.2 Microinstruction Structure 6
2.1.2.3 Microprogram Address Structure 7
2.1.3 Assembly Instructions 7
2.2 Creation and Simulation 9
2.2.1 Logic Gate Fundamentals 9
2.2.2 Clock 10
2.2.3 Data Bus 10
2.2.4 Registers 11
2.2.4.1 General Purpose Registers 11
2.2.4.2 Zero Flag Register 12
2.2.4.3 Instruction Register 12
2.2.4.4 Address Register 13
2.2.5 Memory 13
2.2.5.1 Random Access Memory 15
2.2.5.2 Microprogram Read-Only Memory 16
2.2.5.3 Program Read-Only Memory 17
2.2.6 Counters 17
2.2.6.1 Program Counter 18
2.2.6.2 Microprogram Instruction Counter 19
2.2.7 Arithmetic Logic Unit 20
2.2.7.1 Addition 21
2.2.7.2 Subtraction 21
2.2.7.3 Multiplication 22
2.2.7.4 Logical Bit Shifting 23
2.2.8 Microprogram Control Unit 23
2.2.9 Programming the Assemblers 24
2.2.9.1 Microprogram Assembler 24
2.2.9.2 Assembler 25
3. Conclusion **26**
Acknowledgements: **26**
Sources **26**
Addendum **27**
Microassembler python program: 27
Manas-CPU - Assembler python program: 31
Logic gate circuits made with transistors: 34
Manas-CPU - Logisim-Evolution .circ file: 35

Creation of a Simple CPU Architecture

1. Introduction

This project will cover creating and simulating a simple *central processing unit (CPU)* and its architecture. This project is being done because the CPU is one of humanity's greatest inventions, revolutionising computing; as such, shedding some light on its inner workings is this project's purpose. This project aims to cover the CPU's planning and creation stage. It will cover the planning of the processor's general structure, the microinstructions to be included, their structure, and the microprogram address structure. Then it will cover the assembly instructions to be included and the structure of their microinstructions before moving over to the creation stage.

The creation stage will cover everything from the fundamentals of logic gates to the programming of the assemblers used in this project. It starts with logic gate fundamentals before moving to the clock and the registers needed. Afterwards, the project covers the memory and the counters before moving over to the arithmetic logic unit and the microprogram control unit. Finally, the last section covers the assemblers made in Python. It is expected to go fine; timing issues might be a problem; otherwise, this project will take much time to complete.

2. Main Part

The planning and creation stage will be described and shown in this section. The different subsections will detail the various stages of this project, such as showing the different instructions during the planning stage or the different registers or counters in the creation stage. Creation and simulation will happen mainly in the open-source logic simulation software *Logisim-Evolution*. The main part will end with some programming in the Python programming language, specifically showcasing the making of the two different assemblers required for this project.

2.a Assumptions

- It is assumed that a copy of the open-source logic and circuit simulation software, "*Logisim-Evolution*" v. 3.8.0 or newer, has been installed on the host operating system from the official GitHub page. (Kluter, *Logisim-Evolution* 2022)
- It is assumed that a sufficient rudimentary knowledge of the Python programming language has been acquired. Also, a copy of the Python programming language and its libraries has been installed on the host operating system from the official Python webpage, version 3.10 or newer. (Python, 2019)

- It is assumed that knowledge of different number bases is at a sufficient level.

2.1 Planning

This section will go over the different stages of planning that were required to do this project. It will cover everything from the general structure, such as how many bits the instructions in the instruction set should be made of. To the specifics, such as what microinstructions should be implemented in the control unit, how they are structured and what microinstructions will be making what assembly instructions. The planning stage is crucial in understanding the goal and how it should be implemented during the creation stage.

2.1.1 General Structure

Establishing the foundations of the project first is essential. Here it will be described what size the standard data unit should be, what components the central processing unit (CPU) should be made of, such as how many registers are needed and what their purpose is, the counters and what their purpose is, and more. All these things are highly suggested to plan out during an endeavour such as this project to make the future work far more manageable.

First, it is essential to specify the standard data size the processor should use. It is essential to ensure all the individual parts do not constantly mix data sizes, as that can lead to a host of different bugs and problems unless the creator is entirely sure of what they are doing. In this project, it has been decided that the central processing unit should operate on words, which are typically data sizes of 16-bits, twice the amount of an 8-bit byte. This decision was made to balance the amount of data and simplicity.

Secondly, specifying what registers are needed is also convenient. In this project, this CPU will need six, five 16-bit registers and one 1-bit register. There are the general purpose registers A, B, and C. The instruction and address registers are vital as well. The final register will be the flag register; however, seeing as only the zero flag will be implemented, it will be referred to as the zero flag register. It will be a simple one-bit register for storing whether or not an arithmetic operation’s result is zero, which is needed to implement a conditional jump instruction.

The general purpose registers A, B, and C are crucial. A will act as the accumulator, the register where the result of arithmetic operations will be stored, and a general purpose register where the user can read or write from as wanted. A and B will be the constant inputs to the Arithmetic Logic Unit (ALU) for simplicity and ease of design. C will be an additional register for operations, seeing as two are not enough; it will also enable operations like switching the contents of A and B, where C will act as intermediary storage to facilitate such an operation, or switching between A and C, where B will take that role.

Essential are also the instruction and address registers. The instruction register is where the current instruction being executed will be stored (Hennessy, Patterson and Krste Asanović,

2012), and its contents will split off to be processed further. The address register is where the address currently being accessed in memory will be stored. (Hennessy, Patterson and Krste Asanović, 2012) Together they are both indispensable to ensuring the functionality of the CPU.
Counters are vital in computing, and as such, they will also be required in this project. Counters simply increment or decrement a binary number. (Hennessy, Patterson and Krste Asanović, 2012) In this project, only two counters, the program counter and the microprogram instruction counter, will be needed. Firstly, the program counter will be where the address of the next instruction in memory will be stored, the address in the counter will be incremented after every instruction, and the address will be loaded into the address register. Secondly, the microprogram instruction counter will increment through the microprogram read-only memory (ROM) during the execution of each instruction, which contains the microinstructions for the instructions.

Three pieces of data storage will be needed for this project, and a way to connect the data of all these components. There will be two pieces of read-only memory, the microprogram ROM and the program ROM. The microprogram ROM will contain the microprogram of the CPU, the instructions for the CPU's instructions, or microinstructions for short; it is essentially the lowest level firmware for the processor. (Belzer and Kent, 1993) As such, it will act as the lookup table for executing instructions. The program ROM will store the program to be loaded into random access memory (RAM). In essence, it is where a user will store their assembled assembly programs in order to run them on the device. Finally, one piece of random access memory (RAM) is also required; it is where the machine code of the current program will be copied to be executed. This ability to access and modify the code as it is being executed can also allow a user to modify the currently running code as it is running, allowing for a good amount of creativity but also requiring the user to be observant of which memory addresses they use. A data bus is needed to connect all these components; in this case, it will be a 16-bit parallel connection multidrop bus, meaning it can send and receive 16-bits at once, and the microcode will specify which components are sending or receiving data at any one time. (Carl Stephen Clifton, 1987)

2.1.2 Microinstruction Planning

This section will discuss everything about the planning that went into the microinstructions and their code. Listing of all the different microinstructions and their functions. The structure of the microinstructions themselves, their code, in other words. Lastly, the composition of the microprogram address structure will be shown.

2.1.2.1 Microinstructions

This section will list all the different microinstructions and their functions. The arithmetic, read, write, and special microinstructions. Planning the microinstructions to be implemented is critical because microinstructions are the constituent parts of the higher-level assembly instructions.

2.1.2.1.1 Arithmetic Instructions

The arithmetic microinstructions are all related to the Arithmetic Logic Unit (ALU), which will be in addition mode by default; it is necessary not to forget microprogramming the instructions for addition. These four instructions are the ALU microinstructions:

- ***SU*** - Enables subtraction mode in the ALU.
- ***MU*** - Enables multiplication mode in the ALU.
- ***SR*** - Short for “shift right”, will enable the logical bit shift right operation of the ALU.
- ***SL*** - Short for “shift left”, will enable the logical bit shift left operation of the ALU.

2.1.2.1.2 Read Instructions

The read microinstructions are, as they sound, all about reading data from one of the different components. These six microinstructions will specify which component is allowed to send data onto the data bus at any one time, in order they are:

- ***RA*** - Read from the A register onto the data bus.
- ***RB*** - Read from the B register onto the data bus.
- ***RC*** - Read from the C register onto the data bus.
- ***RM*** - Read from RAM at the address specified in the address register onto the data bus.
- ***IR*** - Read the instruction register from the lowest 11 bits onto the data bus.
- ***CR*** - Read the address currently in the program counter onto the data bus.

2.1.2.1.3 Write Instructions

The write microinstructions are the opposite of the read microinstructions in that they specify which component the value on the data bus should be written to at any one time. There are a total of seven write microinstructions; they are as follows:

- ***WA*** - Write from the data bus into the A register.
- ***WB*** - Write from the data bus into the B register.
- ***WC*** - Write from the data bus into the C register.
- ***WM*** - Write from the data bus into RAM at the address specified in the address register.
- ***AW*** - Write from the data bus into the address register.
- ***IW*** - Write from the data bus into the instruction register.
- ***CW*** - Write from the data bus into the program counter. This acts like a jump instruction by changing the next instruction to be executed.

2.1.2.1.4 Special Instructions

The special microinstructions are the microinstructions that execute a particular function, which either does not fit into any of the other categories or, due to the structure of the

microinstructions, need to be categorized for themselves. There are a total of seven special microinstructions; they are:

- ***EI*** - End instruction, marks the end of an instruction and tells the microprogram instruction counter to reset to zero.
- ***CI*** - Counter increment, this microinstruction increments the program counter.
- ***EO*** - Enable or evaluate operations; it evaluates the instruction specified or adds the A and B registers and writes the output to the A register.
- ***ST*** - Stops the processor clock.
- ***JZ*** - Jump if zero flag set; writes to program counter from data bus if the zero flag is high.
- ***Cpy_ROM*** - Copies from program ROM to RAM by enabling output from the program ROM and direct access from the program counter to the address register.
- ***Chk_Prg_End*** - Checks for a program end instruction in the program ROM, stops copying and starts executing if found; if not found, it will keep copying.

2.1.2.2 Microinstruction Structure

This section will show the planned structure of the microinstructions. That being how many bits the microinstructions will have and what each bit will represent. Without this, the CPU, as it is designed, would not function.

Each entry in the microprogram will be made of 16-bits, one word. In this word, each bit will have a specified purpose, something they will signify.

- 0000000000000000 | 16-bits, one word # Bit 0-15
- 0—---------- | *Cpy_ROM*, will show if Cpy_ROM is currently active. | Bit 15
- –0—---------- | *Chk_Prg_End* | Bit 14
- —0—---------- | Jump if zero (JZ) | Bit 13
- ——-000---------- | ALU operations; Add, SU, MU, SR, SL |Bit 12-10
- ———-0—----------| EO | Bit 9
- ———-000---------- | Read microinstructions. | Bit 8-6
- ———-000------| Write microinstructions. | Bit 5-3
- ———-0----| Program Counter Increment/Enable (CI) | Bit 2
- ———-0--| End Instruction (EI) | Bit 1
- ———-0| Halt the processor clock (ST) | Bit 0

Planning this will make the future creation process much more straightforward. Putting related instructions, except the special microinstructions, in the same groups condenses the data so that it can represent as much information in as little space as possible. Something not feasible with the special microinstructions, as they need to be enabled or disabled individually.

2.1.2.3 Microprogram Address Structure

This section will show the structure of the microprogram address. The structure of the address will relate to the assembly instructions, which are covered in the next section; however, the composition of the microprogram address structure is straightforward.
The microprogram address is 8-bits long, or one byte. It is composed of two individual sections, the instruction opcode section and the microprogram instruction counter (MIC) section. The instruction opcode section is 5-bits long, the length of this CPU's instruction opcodes. Therefore the second section, the MIC section, can only be 3-bits long, which is just enough for this project.

- 11111222 | 1 = Instruction opcode; 5 bits | 2 = Counter; 3 bits

The microprogram address will point to an individual word in the microprogram ROM. As an example, *00000000* will point to the first word in the ROM of the first instruction, *00000001* will point to the second word of the first instruction, and so on until a max of *00000111* which will point to the eight word of the first instruction unless EI is met first. The counter will continue to increment until it hits the eight word before it repeats or until it hits EI first. *00001000* will point to the first word of the second instruction, *00010000* will point to the first word of the third instruction, and so on. This way, each assembly instruction will have eight words worth of microinstruction data available, plenty to implement each assembly instruction.

2.1.3 Assembly Instructions

This section will show all the different assembly instructions to be implemented and their purpose. Instruction opcode will be five bits long, the five most significant bits, allowing for a total of 32 instructions. However, the first and last opcode are reserved for special instructions, leaving the possibility of 30 custom instructions by mixing the microinstructions into sections representing words of the microprogram ROM and individual clock cycles. Since each word in the microprogram ROM will only take one clock cycle to complete, the sections mean both. The eleven least significant bits are reserved for the instruction argument data, such as what value to load into register A in the LDIA <value> instruction. The assembly instructions are as follows:

- Instruction <argument> :opcode: | Description
  - word/cycle: microinstructions needed to do this instruction

- FETCH | Fetches the next instruction
  - 0: CR, AW | 1: RM, IW | 2: CI, EI
- Copy_Prg_ROM :*00000*: | Copies the program ROM, used by default at 00000
  - 0: Copy_ROM, AW | 1: Copy_ROM, Check_prg_end | 2: Copy_ROM, CI, WM | 3: EI | 4: FETCH, to get instructions started, jumps here if prg_end detected

- LDA <addr> *00001* | Load data from RAM to A register
  - 1: IR, AW | 2: RM, WA | 3: FETCH
- LDB <addr> *00010* | Load data from RAM to B register

    - 1: IR, AW | 2: RM, WB | 3: FETCH
- LDC <addr> *00011* | Load data from RAM to C register
    - 1: IR, AW | 2: RM, WC | 3: FETCH

- LDIA <value> *00100* | Immediately load <*value*> into register A
    - 1: IR, WA | 2: EI
- LDIB <value> *00101* | Immediately load <*value*> into register B
    - 1: IR, WB | 2: EI
- LDIC <value> *00110* | Immediately load <*value*> into register C
    - 1: IR, WC | 2: EI

- STA <addr> *00111* | Store A into memory <addr>
    - 1: IR, AW | 2: RA, WM | 3: EI
- STB <addr> *01000* | Store B into memory <addr>
    - 1: IR, AW | 2: RB, WM | 3: EI
- STC <addr> *01001* | Store C into memory <addr>
    - 1: IR, AW | 2: RC, WM | 3: EI

- ADD *01010* | Add register B to A, and store the result in register A
    - 1: EO, WA | 2: EI
- SUB *01011* | Subtract register B from A, and store the result in A
    - 1: SU, EO, WA | 2: EI
- MULT *01100* | Multiply register B with A, store result in A
    - 1: MU, EO, WA | 2: EI
- SHL *01101* | Shift value in A register by value in B amount to the left
    - 1: SL, EO, WA | 2: EI
- SHR *01110* | Shift value in A register by value in B amount to the right
    - 1: SR, EO, WA | 2: EI

- JMP <addr> *01111* | Change program counter to <addr>, changes next instruction
    - 1: IR, CW | 2: EI
- JMPZ <addr> *10000* | Jump if zero flag set
    - 1: IR, JZ | 2: EI

- LDAIN *10001* | Use A as mem addr, then load from RAM to A register
    - 1: RA, AW | 2: RM, WA | 3: EI
- STAOUT *10010* | Use register A as mem addr, then load B to memory address
    - 1: RA, AW | 2: RB, WM | 3: EI

- SWP *10011* | Swap contents in register A and B, overwrites C
    - 1: RA, WC | 2: RB, WA | 3: RC, WB | 4: EI
- SWPC *10100* | Swap reg A and C, overwrites B
    - 1: RA, WB | 2: RC, WA | 3: RB, WC | 4: EI

- HLT *10101* | Stop the clock
  - 1: ST
- NOP *10110* | No operation | Fetch the next instruction
  - 1: FETCH
- Program_END *11111* | Signifies end of the program in ROM, stops copying here; hex: F800 | Reserved for the assembler, not something the user inputs.

2.2 Creation and Simulation

In this section, the creation and simulation of the central processing unit will be showcased and explained. It will start with a short showcase of the basic logic gates. Then it will show the clock, the data bus, the registers, the data storage, the counters, the arithmetic logic unit (ALU), and then it will show how the microprogram control unit (CU) was designed, and finally the making of the assemblers.

2.2.1 Logic Gate Fundamentals

This section will quickly explain the basic logic gates, their shape, and their function. The gates that will be explained are shown in *Figure 1* as follows:

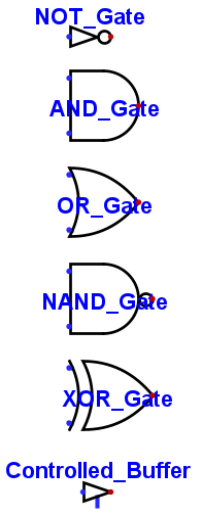

Figure 1: The Logic gates

- The NOT gate performs a logical NOT operation on the input, which inverts it. So, if the input is one, the output will be zero.
- The AND gate performs a logical AND operation on the input, only turning the output on if both inputs are high. So, if both inputs are one, the output will also be one; otherwise, it will be zero.
- The OR gate outputs one if any of its inputs are one; zero if they are all zero.
- The NAND gate performs a logical AND operation; the only difference is that the output is inverted, in other words, passed through a NOT gate; the only time the output will be "false", or zero, is when both inputs are true.
- The XOR gate, or "*Exclusive OR*", is different from the OR gate in that it will only output one if exclusively *one* of its inputs are one. So, if it has two inputs and both are one, it will output zero; however, if only one of the inputs is one, it will output one.
- The controlled buffer is a *tri-state* logic gate; Meaning it has three states; high (1), low (0) and high impedance (Z). It acts like a logical AND gate in terms of logic, only allowing the input to pass through if the enable pin is on; so, if both inputs are one, it will output one. However, it is more accurate to say it is like a switch, only connecting the circuit if enabled. It is used a lot in this project regarding the data bus so that the output of the components will not interfere with each other.

There are also NOR and XNOR. They can be made by inverting their respective gates with a NOT gate. It is helpful to look at the logic tables of the gates for further reading, in order to understand how they work and affect each other intuitively.

2.2.2 Clock

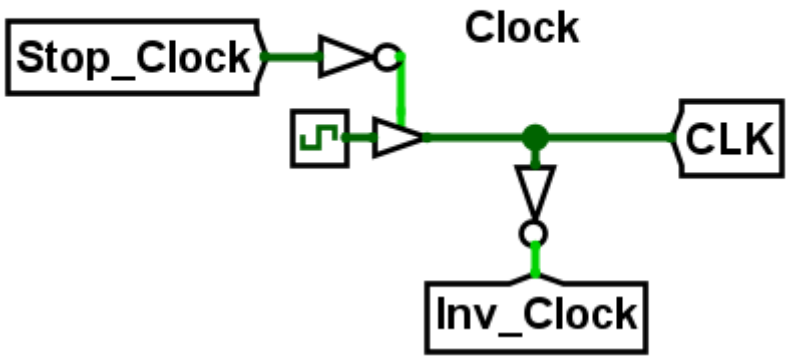

Figure 2: The clock design of the CPU

As seen in *Figure 2,* the clock is simple in design. The clock is a square wave clock, as seen in *Figure 3*; it has a high duration of one tick and a low duration of one tick, meaning it will output a one for one unit of time and zero for one unit of time, together, those two units of time make up one clock cycle. Its output goes through a controlled buffer to make stopping the clock through code possible. This controlled buffer's output is only enabled if there is no "*Stop_Clock*" signal; the text boxes are tunnels, essentially hidden wires. They can be placed anywhere and helps with the organisation of the design. The clock outputs to both a "*CLK*" tunnel to be placed wherever needed, and an inverted "*Inv_Clock*" tunnel useful for the execution of the microprogram.

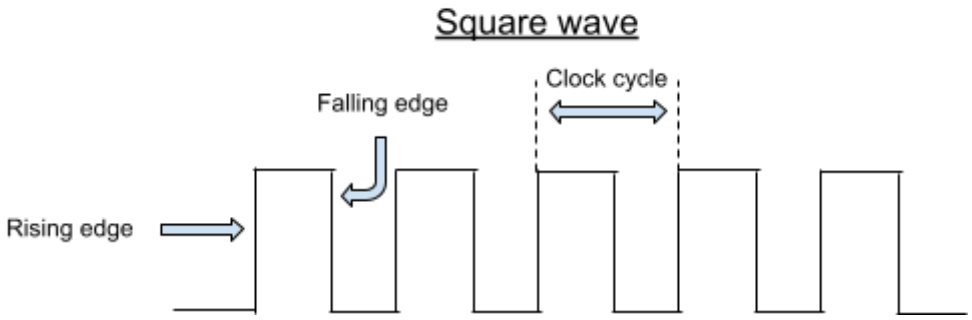

Figure 3: Drawing of a square wave, showing the rising, falling edge and clock cycle

2.2.3 Data Bus

The data bus is a multidrop parallel connection bus. (Carl Stephen Clifton, 1987) In this project, it is simply a collection of wires sending 16-bits of data in parallel to whatever component is reading. The write and read permissions are not handled by the bus itself; however, all components that handle data are connected and can be written to or read from at will. All data handled by the CPU goes onto this collection of wires.

2.2.4 Registers

Registers are essential in computing. They are generally made up of an array of flip-flops, logic circuits of 1-bit storage usually triggered by a clock's rising or falling edge, connected in parallel as seen in Figure 4, which is again built up usually of SR latches, 1-bit storage with a set and reset pin, made of logic gates as seen in Figure 5. (Texas Instruments, 1988) Registers are quick storage for low amounts of data, used in most processor operations. There are many types of registers, but in essence, they are quick data storage for x amount of bits for a processor.

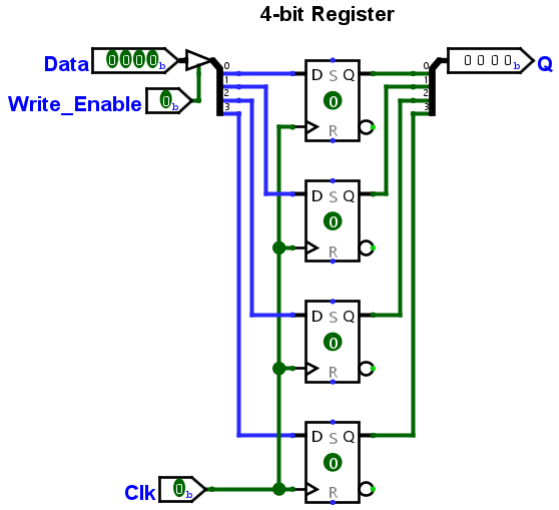

Figure 4: 4-bit Register of D-flip flops

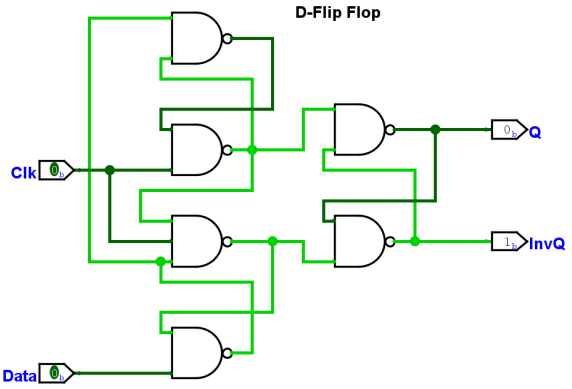

Figure 5: D-flip flop of SR NAND latches

This project has six registers accessed by or connected to the CPU's operation. Three general purpose registers, one zero flag register, an instruction register, and an address register.

2.2.4.1 General Purpose Registers

This section will list out the general purpose registers of this processor. There are three general-purpose registers A, B, and C. They are 16-bit registers, with A acting as the accumulator, as detailed in *2.1.1 General Structure*.

A clock edge does not trigger these registers. Instead, they are *high* enabled. Instead of only writing on the rising or falling edge of the clock cycle, it will constantly write whatever is on the data bus to the register if the clock signal is high. The outputs of A and B are constantly transferred to the ALU's A and B inputs. If the corresponding register receives a *Read_Reg* signal, it will enable output to the data bus.

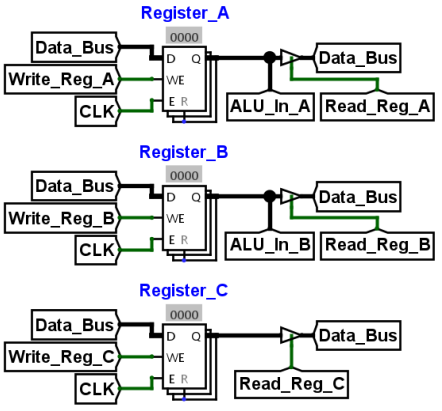

Figure 6: Layout of general purpose registers

2.2.4.2 Zero Flag Register

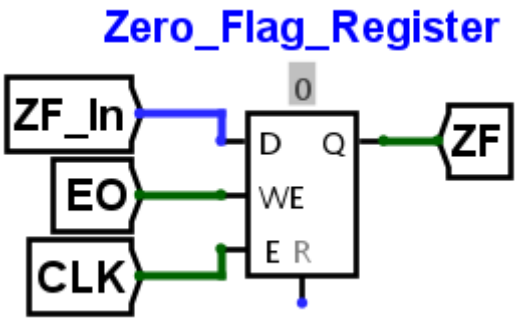

Figure 7: Zero flag register

The zero flag register is a simple 1-bit register; see *Figure 7*. It is high enabled like the general purpose registers, constantly writing from the "*ZF_In*" tunnel if *the microprogram specified EO, Evaluate Operation*. The "*ZF_In*" tunnel reads from the ALU; it specifies whether or not the current operation returns zero as the answer. It constantly outputs to the "*ZF*" tunnel, which goes to the program counter and is used to conditionally write to it in order to implement the *jump if zero (JZ)* microinstruction, which itself is used to implement the "*JMPZ*" assembly instruction, and is to facilitate conditional branching in the code.

2.2.4.3 Instruction Register

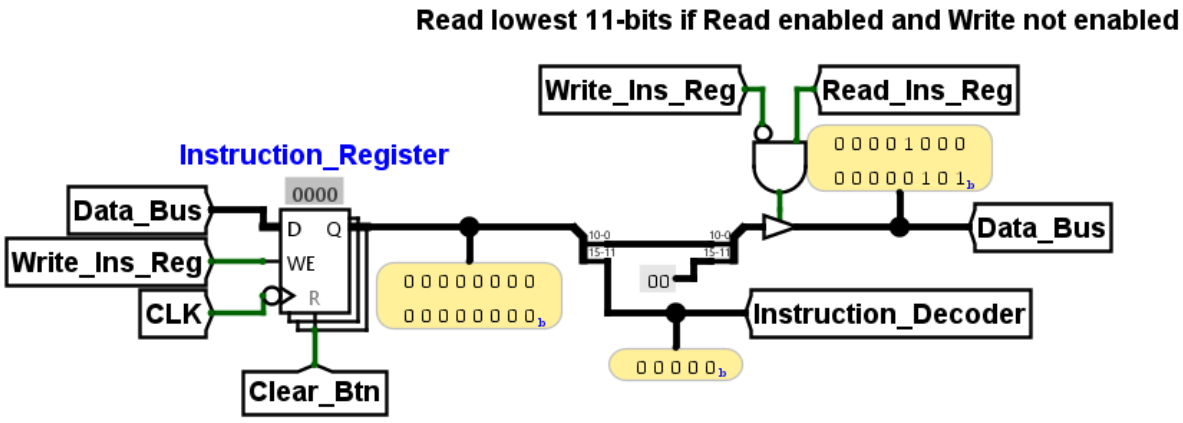

Figure 8: Design of instruction register and its related circuitry

The instruction register is one of the most essential components in the processor. It is a 16-bit register that holds the machine code of the current instruction being executed. The falling edge of the clock cycle triggers this register. Its write function is enabled if specified through the microprogram. The register's output is split into two, the most significant five bits and the least significant eleven bits. Its most significant five bits contain the instruction opcode, which is sent to the instruction decoder as part of the microprogram address, used to look up the microinstructions of the current instruction being executed. The least significant eleven bits get five zeroes inserted at the front to make the total data sixteen bits,

then sent through the controlled buffer to the data bus if "*Read_Ins_Reg*" and not "*Write_Ins_Reg*".

2.2.4.4 Address Register

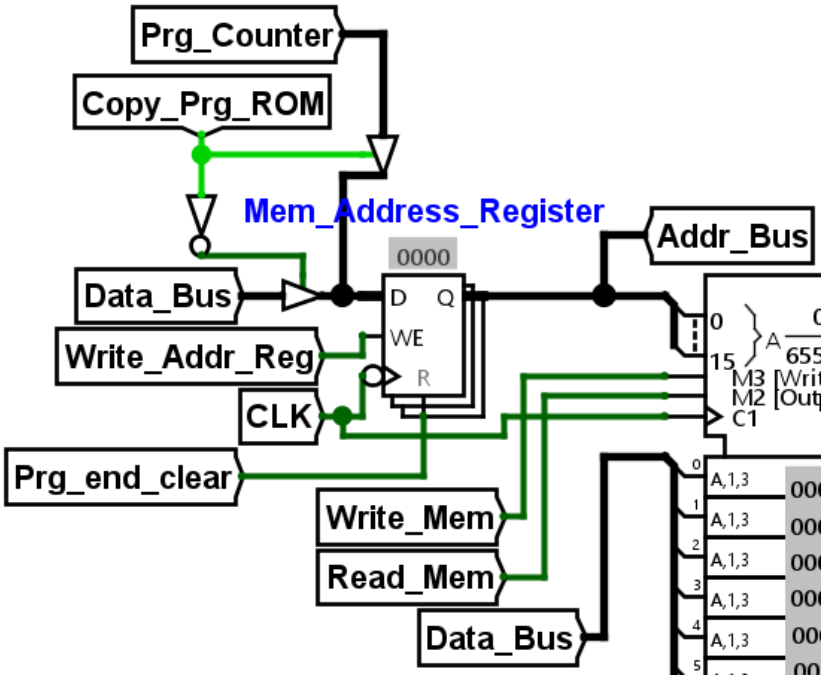

Figure 9: Design of the memory address register and its connection to RAM on the right.

This section will detail the address register and its design choices. The memory address register is a 16-bit register that holds the address currently accessed in memory. The falling edge cycle triggers this register to time it properly and not have any issues. It can be written to directly from the program counter or from the data bus. This direct connection to the program counter only happens during the initialisation phase, when the CPU needs to copy the program ROM to RAM. It is then reset once it hits the end of the program in the program ROM. The address register's output is connected to the RAM directly, and the program ROM through the tunnel "*Addr_Bus*", which is used to synchronise the current memory location being addressed in RAM and the program ROM.

2.2.5 Memory

In this section, the different RAM and ROM components will be detailed. There are a total of three RAM and ROM components, one RAM acting as the primary external memory for the execution of the programs, and two ROM components. One program ROM which holds the machine code of the assembled program to be executed, and one microprogram ROM, where the microprogram firmware is stored. Memory is usually built up of an array of memory cells where an address input and a demultiplexer will cycle through the memory cells, consisting of an array of flip-flops with read and write enable pins and a clock pin for triggering the write on the rising or falling edge. (IBM, 2017) See *Figure 10-12* for designs.

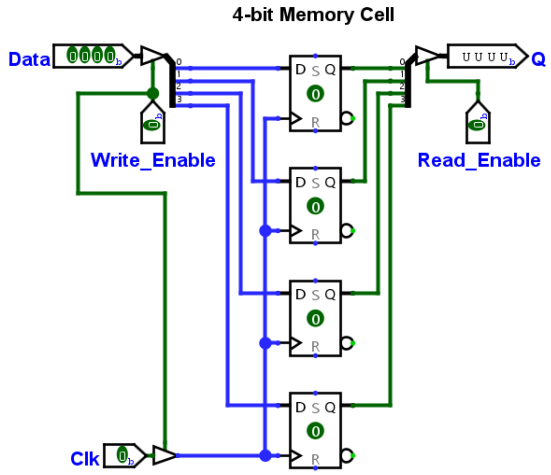

Figure 10: Design of a 4-bit memory cell

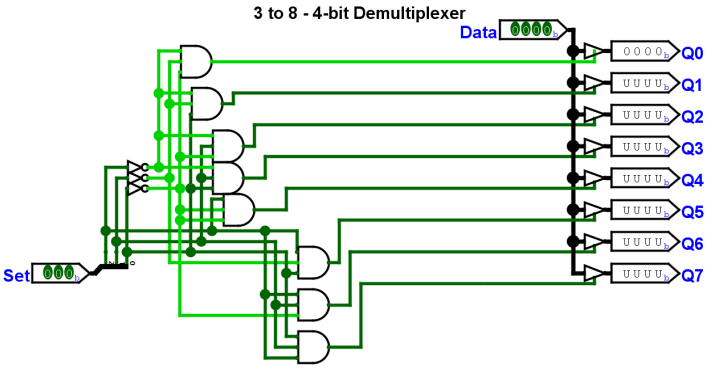

Figure 11: Design of a 3 to 8 demultiplexer

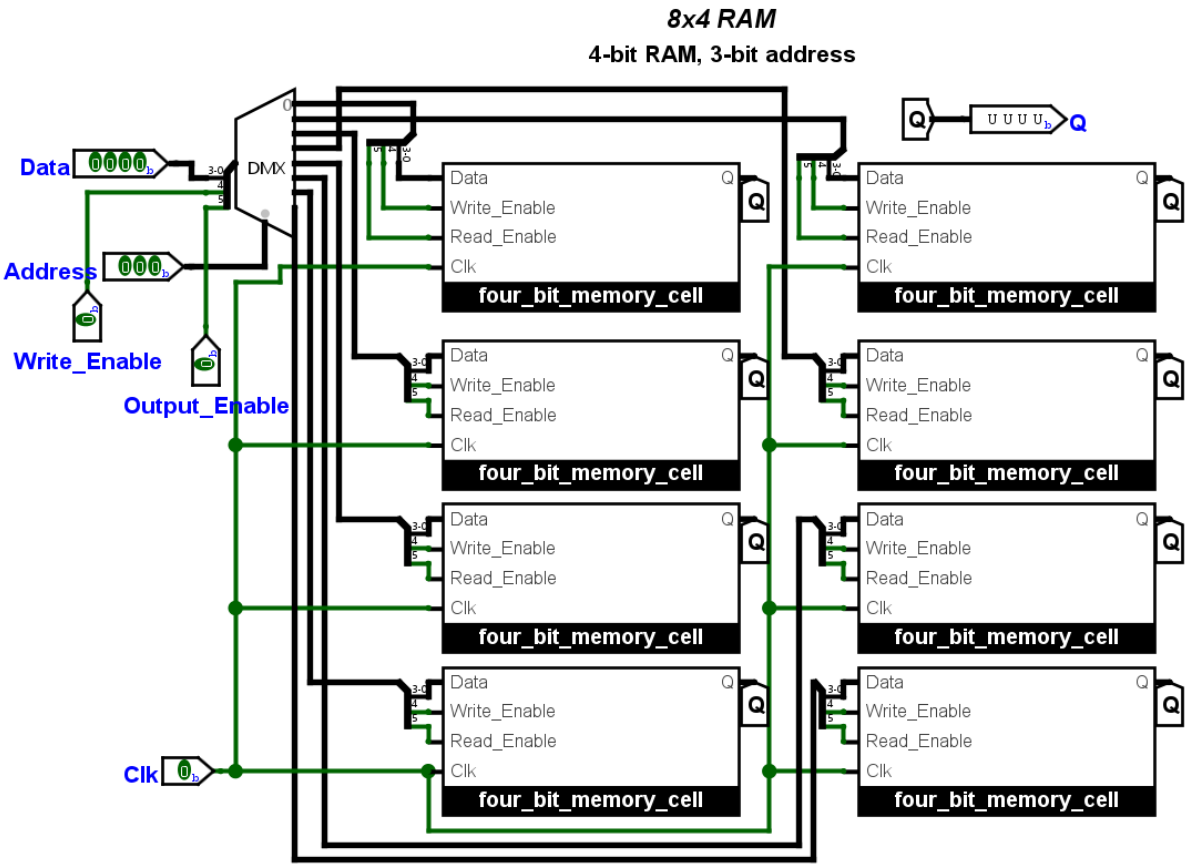

Figure 12: Design of a 4-bit RAM with a 3-bit addressing scheme

2.2.5.1 Random Access Memory

This section will show the layout of this CPU's RAM. It stores the program that will be copied over from the program read-only memory before it is executed; any other data specified by the executed program can also be stored here. Meaning the program can be modified as it is being executed by modifying the contents of the RAM in the memory addresses of the program itself.

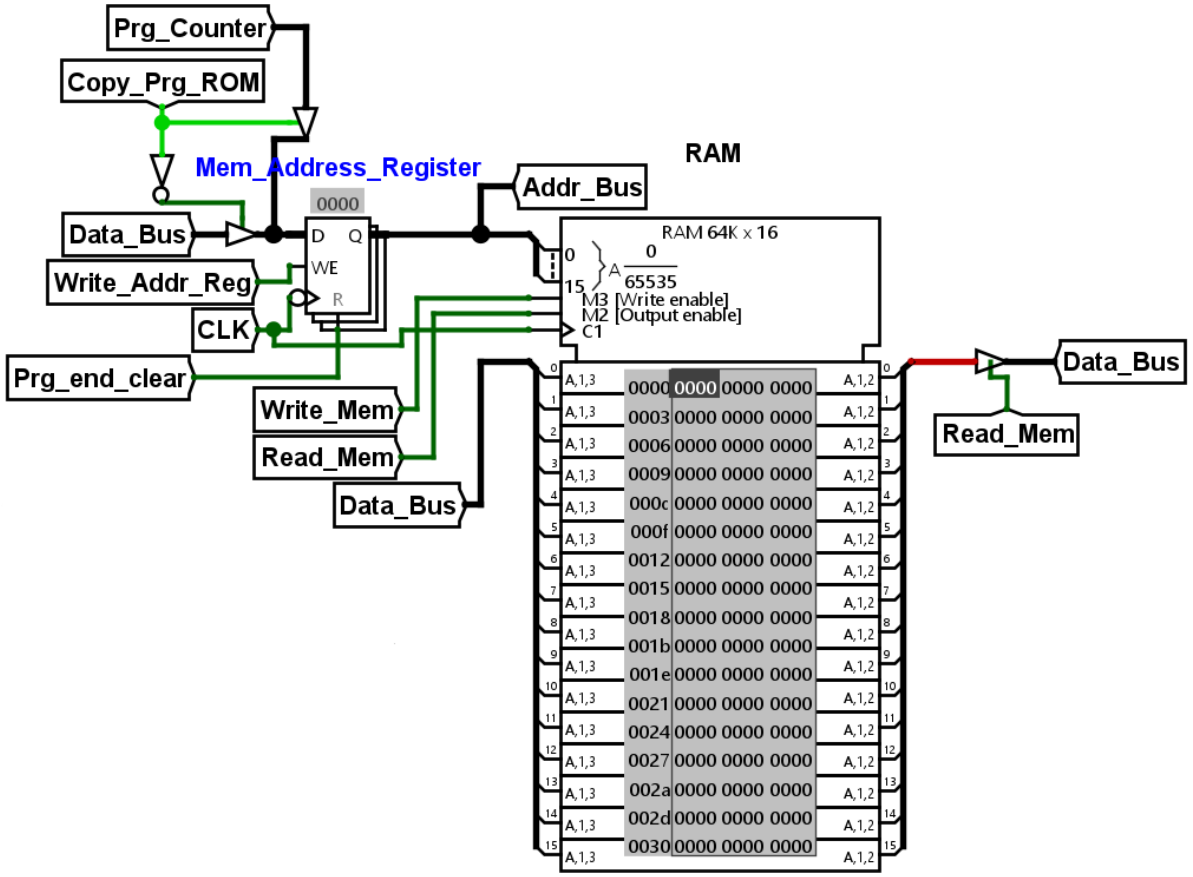

Figure 13: Layout and connection of RAM and the address register.

This RAM component comprises 65536 16-bit memory cells. It uses a 16-bit address, as sixteen bits can address a total of 65536, where the highest number is 65535 and the lowest number is 0. Like all other components, the RAM component is controlled by the microprogram, which can enable writing to individual memory cells or reading from them; it is also directly connected to the memory address register, as shown in *Figure 13*.

2.2.5.2 Microprogram Read-Only Memory

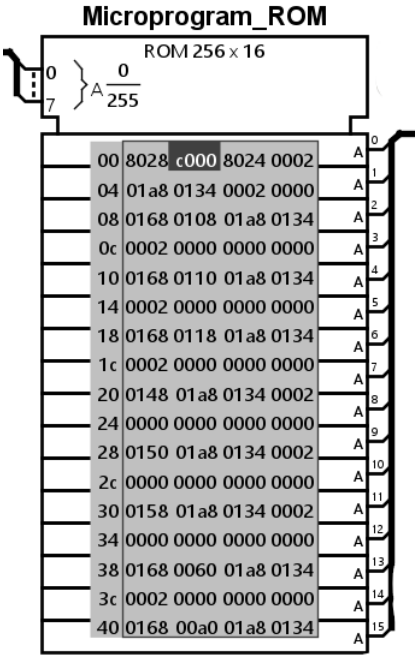

Figure 14: The Microprogram ROM.

The microprogram ROM is a simple and small ROM component. It is made up of 256 16-bit memory cells and uses an 8-bit address. Every eight memory address starting from zero is the start of the microinstructions for a new assembly instruction, meaning each instruction can be made up of up to eight microinstructions. It is only one part of the microprogram control unit (CU).

2.2.5.3 Program Read-Only Memory

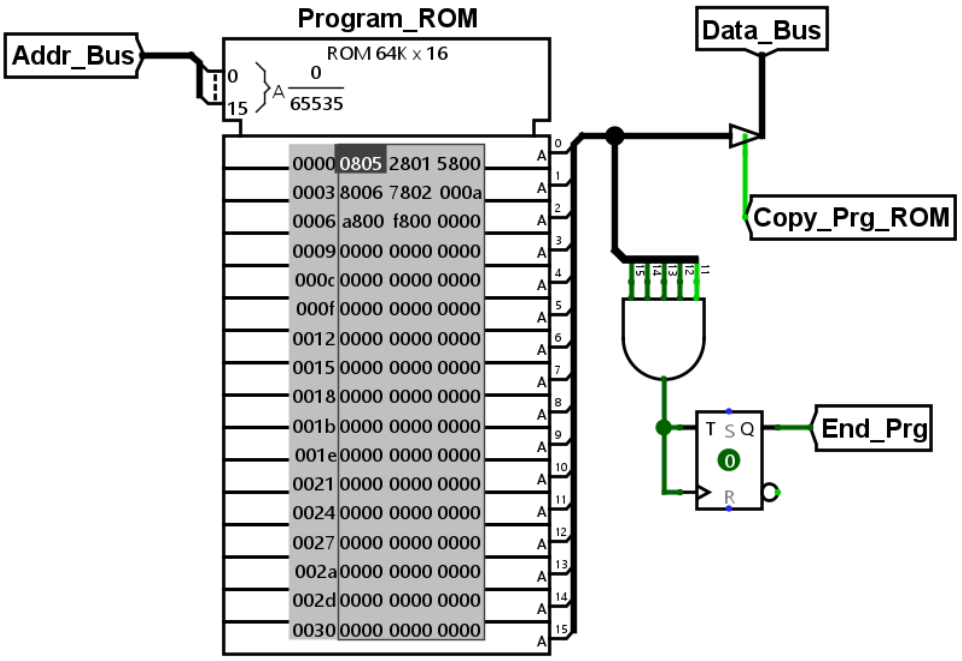

Figure 15: The Program Read-Only Memory and its surrounding circuit.

The program ROM is a ROM component the same size as the RAM. The differences between them are that there is no write function that can be accessed through the logic circuit, and the ROM is non-volatile memory, meaning its contents get stored even if the program gets shut down, or in the case of an actual circuit, it would be the circuit itself getting turned off. Its address input is directly connected to the output of the address register through the "*Addr_Bus*" tunnel. The output's five most significant bits, where the opcode is, get checked for *11111,* representing the program's end. At that point, it stores a high signal in a small 1-bit register to store the end of program signal and uses that further to clear the address register and the program counter. Otherwise, it keeps outputting whatever is in the memory cell the address register is pointing to so that it can be copied to RAM.

2.2.6 Counters

This section will detail the two counters used in this project and their purpose. They are the program counter and the microprogram instruction counter. The purpose of counters is to simply increment or decrement a given number or the number that is currently stored. (Arun Kumar Singh, 2006) In this case, it is used to access places in memory, both RAM and the microprogram ROM.

A counter can be made in many ways; however, a simple design, is the ripple counter. The ripple counter is made of flip-flops where the standard output of one flip-flop goes into the clock input of the next, in that way the signal ripples through the different flip-flops, hence why it is called the ripple counter. The ripple counter shown in *Figure 16* is made of D-flip-flops, see *Figure 5*, where the inverted output is used as the data input of the same flip-flop, and the standard output as the clock input of the next.

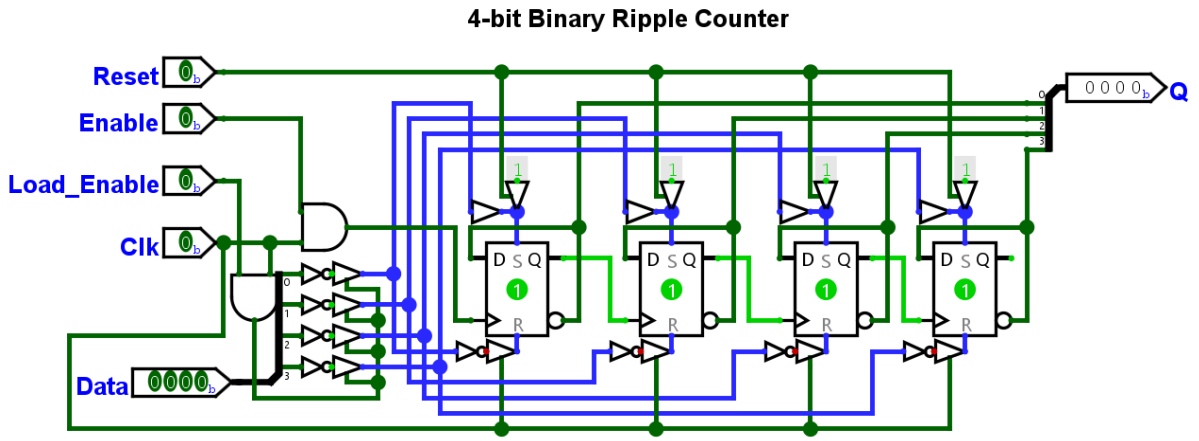

Figure 16: 4-bit binary ripple counter with load and reset functionality.

2.2.6.1 Program Counter

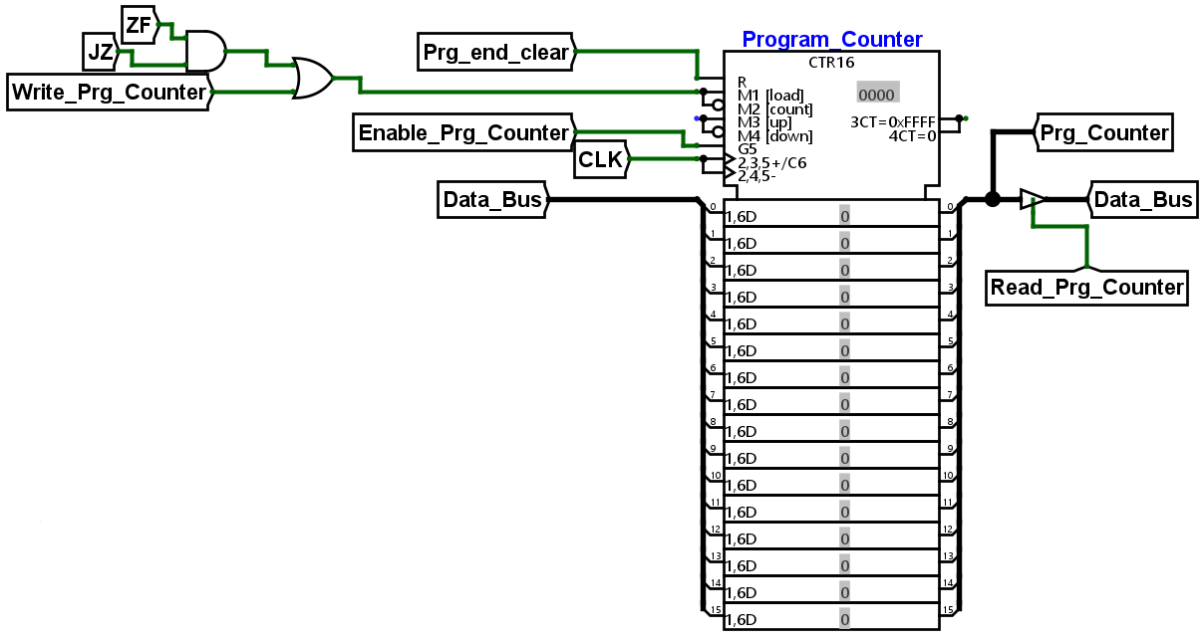

Figure 17: Program counter and surrounding circuitry

The program counter is where the address of the next instruction to be executed is stored. It is a 16-bit counter and can therefore increment through all of RAM when used as an address. It gets reset to zero when it overflows or when the *program end clear* signal is sent. The microprogram controls it, and it can be written to from the data bus, either when the microprogram sends the *write program counter* signal or when both *Jump Zero* (*JZ)* and *Zero Flag (ZF)* are high. The rising edge of the clock cycle triggers the program counter.

2.2.6.2 Microprogram Instruction Counter

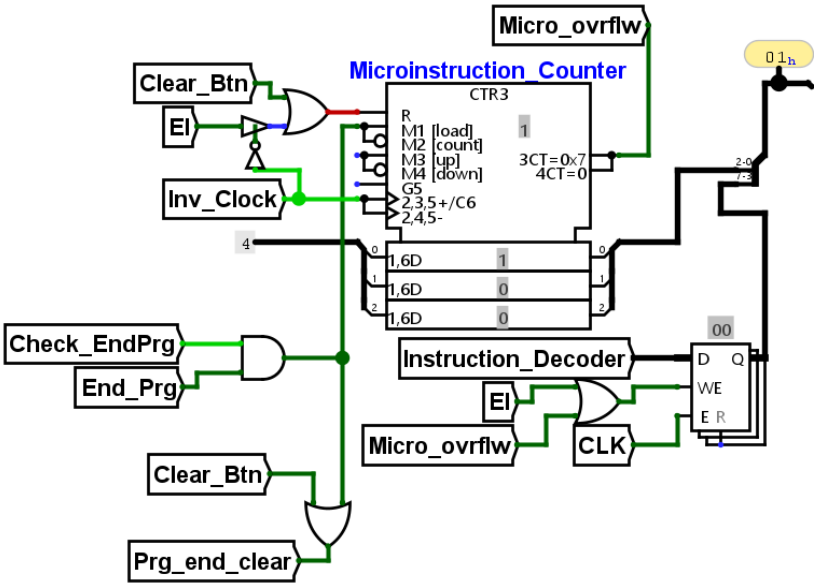

Figure 18: Microprogram instruction counter and circuitry.

The microprogram instruction counter increments through the max eight microinstructions of an instruction, until it hits *End Instruction (EI)* and the inverse clock is not high. It is an 8-bit counter to facilitate this. Since the microinstructions of copying the program ROM take up the first three memory cells of the microprogram ROM, the number four will be loaded into this counter if the processor is currently checking for the end of the program and the end of the program is found, and the "*Prg_end_clear*" signal is sent to clear the program counter and address register. Its output combines with the currently stored opcode in a 5-bit register and thus becomes the three least significant bits of the combined output.

2.2.7 Arithmetic Logic Unit

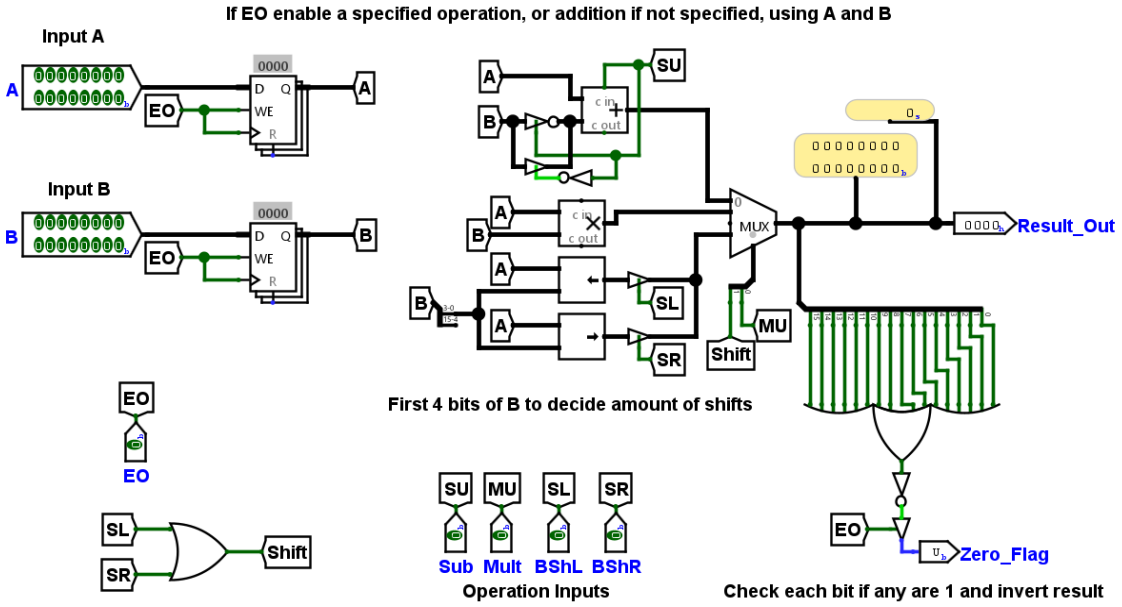

Figure 19: Internal design of the Arithmetic Logic Unit (ALU).

The Arithmetic Logic Unit (ALU) is the component in charge of managing all the arithmetic operations of the CPU. This implementation has five arithmetic operations: *addition, subtraction, multiplication, bit-shift left, and bit-shift right*. *Figure 19* shows six inputs, two 16-bit inputs directly connected to Register A and B, as seen in *Figure 20*, and four 1-bit operation inputs controlled by the microprogram that chooses which operation to output. These operations are chosen using a multiplexer, a component which chooses which data input to output via a combination of selection inputs. In this case, the data inputs are *addition or subtraction*, *multiplication*, and *shifting either left or right*; three data inputs. It has two selection bits, the multiplication signal (MU) and a shift signal, which is a combination of either *shift left (SL)* or *shift right (SR)*. If neither of those signals are on, then it can be presumed that the operation wanted is addition or subtraction, which both use a 16-bit full adder for their operation. All output bits are constantly monitored using an inverted 16-bit OR gate, a *NOR* gate, to check if all of the bits are low, and will output the zero flag to the zero flag register if *EO* is high. If EO is high, it will also output its result to the data bus.

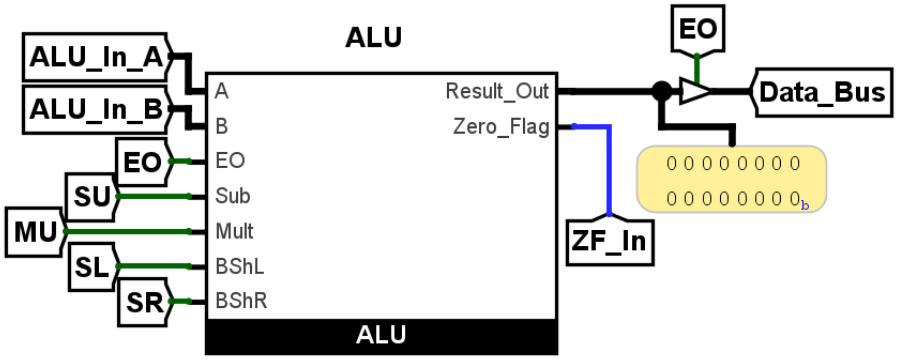

Figure 20: The ALU component and its connections.

2.2.7.1 Addition

Binary addition is pretty simple. At its most basic, it uses half-adders to add two bits; if one of the bits are high, then output high; if both are high, then output high to the carry output, as shown in *Figure 21*. In *Figure 22,* a 1-bit full-adder is shown. Here two half-adders are put together, and the output of one goes into the input of the next half-adder; the third input allows for a carry-in, meaning three 1-bit values can be added together. While in *Figure 23,* a 2-bit full-adder is shown, meaning two 2-bit values and a 1-bit carry-in can be added together. This ALU uses a 16-bit full-adder for its addition.

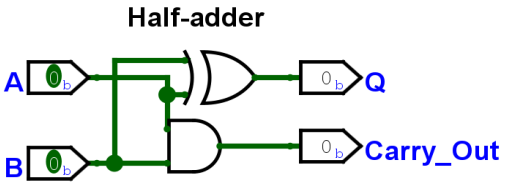

Figure 21: Half-adder design

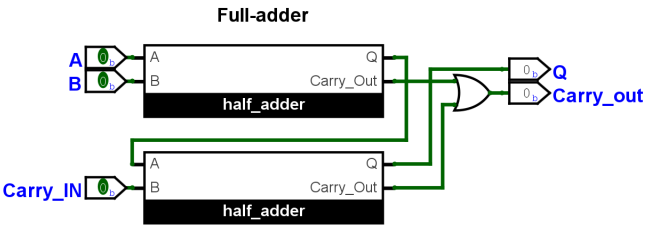

Figure 22: 1-bit full-adder design.

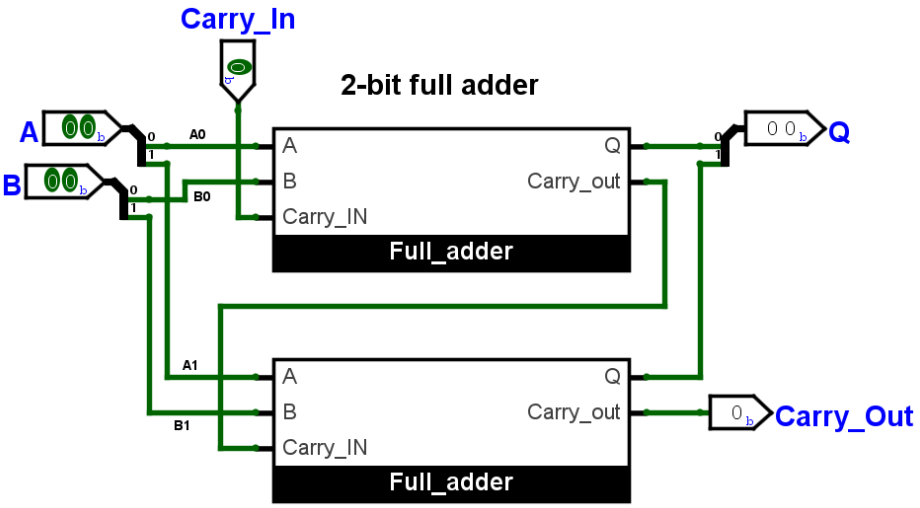

Figure 23: 2-bit full-adder design

2.2.7.2 Subtraction

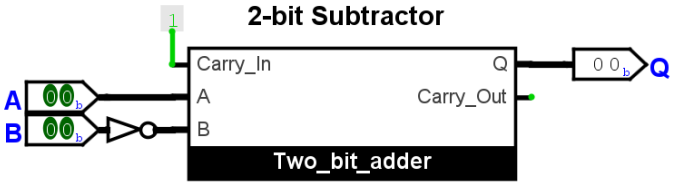

Figure 24: 2-bit subtraction using two's complement, with a 2-bit adder.

Subtraction with binary numbers is different, but still, it uses full-adders and addition. It works using two's complement. (Lilja and Sapatnekar, 2004) With two's complement, a binary number can be converted to its signed negative representation by inverting all bits and adding one, ignoring overflow, as seen in *Figure 24*. (Lilja and Sapatnekar, 2004) The

leftmost bit represents if it is a positive or negative number. Here are some examples of converting an unsigned 3-bit integer to a two's complement integer:

$$001 = 1 \rightarrow 110 + 1 = 111 = -1$$
$$010 = 2 \rightarrow 101 + 1 = 110 = -2$$
$$011 = 3 \rightarrow 100 + 1 = 101 = -3$$
$$100 = 4 \rightarrow 011 + 1 = 100 = -4$$

Due to two's complement using the final bit as a sign of negative or positive, the highest number represented by a signed 3-bit integer can only be three. At the same time, the lowest can be negative four. Giving a range of *-4 to 3*, still eight different numbers. Using this principle, one can add a regular 3-bit integer and a signed two's complement integer to get subtraction using only addition.

2.2.7.3 Multiplication

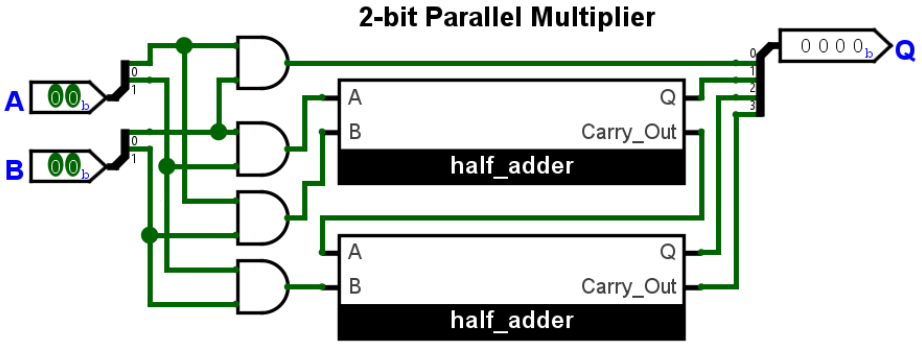

Figure 25: 2-bit parallel multiplier design using half adders and AND gates

Multiplication with binary numbers can be done in two main ways. Either using bit-shifts and adding the partial products or generalising the multiplication and doing each bit in parallel. (Lilja and Sapatnekar, 2004) Bit-shifting for multiplication is cumbersome and takes several clock cycles; however, doing it in parallel can be done almost instantly. The output will always be $2 \times n$ where *n* is the number of bits in the input, so if each input is two bits, there will be four bits in the output, as seen in *Figure 25*. It works because binary multiplication with only one bit only requires an AND gate.

$$0 \times 0 = 0$$
$$1 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 1 = 1$$

1-bit binary multiplication.

As seen above, the results look exactly like the truth table of an AND gate. As such, it is possible to complete the operation in parallel. There are two inputs of two bits A and B, and an output, *x*, of four bits. The formula will look like so:

$$x_0 = (A_0 \wedge B_0)$$

$$x_1 = (A_1 \wedge B_0) + (A_0 \wedge B_1)$$
$$x_2 = \left[\left((x_1 > 1) \Rightarrow 1\right) \vee 0\right] + (A_1 \wedge B_1)$$
$$x_3 = \left[\left((x_2 > 1) \Rightarrow 1\right) \vee 0\right]$$

Meaning x_0 will be one if A_0 *and* B_0 are one, ∧ is the mathematical symbol for the logical AND operation, ∨ is for the logical *OR* operation, and ⇒ can be thought of like an *if* statement, if (statement) true then x. x_1 can be, in binary, either *01* or *10*; in the final result the overflow of each statement will be ignored, so anything that equals *10* will return 0. x_2 uses the value of x_1; if x_1 is greater than one, then add 1 to $A_1 \times B_1$, otherwise add 0 to $A_1 \times B_1$. While the fourth bit, x_3, is equal to one if there is an overflow from x_2 meaning it is greater than 1; otherwise, it is 0.

2.2.7.4 Logical Bit Shifting

Bit shifting is simply moving the values of the bits to the left or right. So, 0001 bit-shifted once to the left would become 0010, doubling in value. Going the other way, it will act like a floor division, dividing it by two floored to the nearest integer. (Lilja and Sapatnekar, 2004) In *Figure 19,* it is possible to see that this ALU uses the four least significant bits of the B register to determine how many times the 16-bit number of the A register gets shifted in the direction specified by the microprogram control unit. It can be done with bit-shift registers, barrel shifting and many other methods.

2.2.8 Microprogram Control Unit

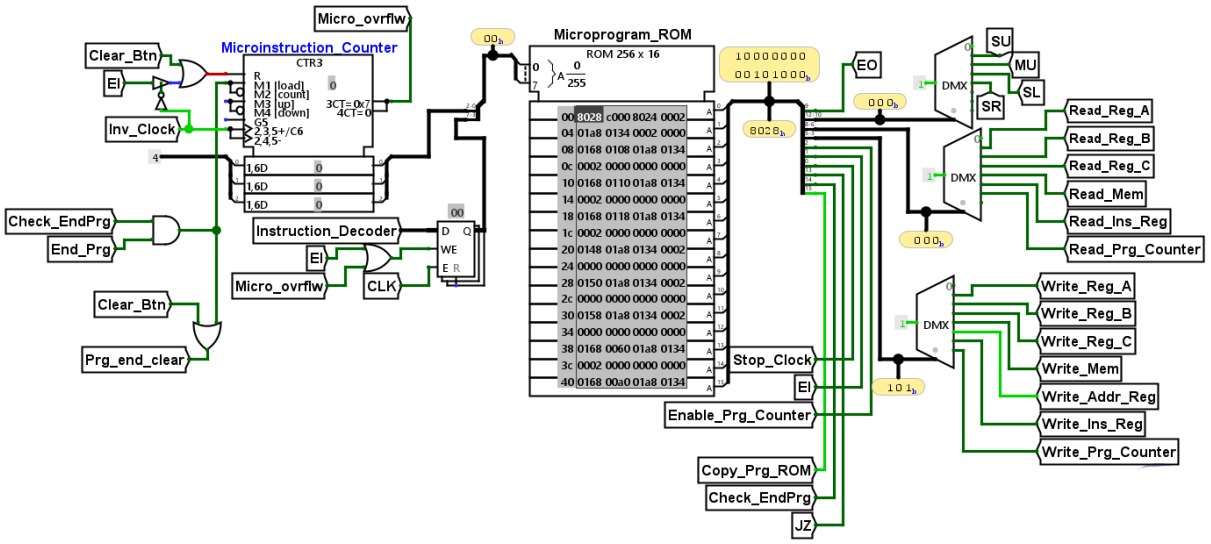

Figure 26: The Microprogram Control Unit and its surrounding circuitry and components.

The microprogram control unit is the brain of the processor. It is where the machine code instructions get decoded and executed. It is composed of the microprogram ROM, the microprogram instruction counter, and a host of demultiplexers for choosing which signal to send, depending on the value in the current microprogram ROM memory cell. The structure

of the microinstructions in the ROM has been explained in the planning stage *2.1.2.2 Microinstruction Structure*.

2.2.9 Programming the Assemblers

This section will detail the making of the two assemblers required to make the logic circuit useful, the microprogram assembler and the assembly assembler. They were made in *Visual Studio Code* using *Python 3.11.0* for simplicity.

2.2.9.1 Microprogram Assembler

```
def printInstructions():
    lineLength = 8
    line = 0
    with open("microprogram.txt", "w") as f:
        print("v3.0 hex words addressed", file=f)
        for instruction in Assembly_Instructions.values():
            remainingInsLength = lineLength - len(instruction)
            for y in range(remainingInsLength):
                instruction.append(f"{0:04x}")
            print(f"{line*8:02x}:", *instruction, file=f)
            line = line + 1
```

Figure 27: Main print instructions function of the microassembler.

The microprogram assembler assembles the microprogram that will control the microprogram control unit, and the processor's function when executing. It outputs all the microinstructions of the instructions in an addressed hex format to a *microprogram.txt* file that it creates or overwrites if it already exists, as shown in *Figure 27*. The *JMP* instruction's structure as shown in *Figure 28*:

```
"JMP": [ # JMP <addr>    01111 : Change program counter to <addr>, changes next instruction
    f'{(Read_Instructions["IR"]):04x}',
    f'{(Read_Instructions["IR"] | Write_Instructions["CW"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
],
```

Figure 28: Structure of the JMP instruction in the microprogram assembler.

Each instruction is placed in a dictionary where the value is an array of formatted four-character long hex strings. The JMP instruction consists of the microinstructions *IR*, in the 0th position of the array, representing the first memory cell at address 01111000 of the microprogram ROM; the next memory cell is found by doing a logical OR operation with IR and CW, then the fetch instruction follows in the next three memory cells. Section *2.1.3 Assembly instructions* will show the same in a slightly different format; one can also find the *FETCH* instruction's microinstructions there. Optionally, the entire code for the microassembler is shown in the *Addendum* section under the section *Microassembler python program*.

2.2.9.2 Assembler

The assembler is responsible for looking through the provided assembly code file and converting it to machine code, readable by the microprogram control unit. It outputs its output to an addressed hex format file specified in the command line as the second input, or defaults to writing to a file called *output.manasbin*. *Figure 29* shows an example assembly program, while *Figure 30* shows its assembled output where the final word f800 is the end_program signal.

prg01-example.manas

```
; Program counts down from 10 and ends.

main:                       ; Program will always start at 0x0000, which is the start of the program ROM
LDA data                    ; so try to keep main: label at the top, unless you're comfortable changing it
LDIB 0x1

loop:
SUB                         ; Subtracts B register from A register, here subtracts 1 until A is zero
JMPZ end
JMP loop

data:                       ; Label for memory address where a 16-bit word with value 10 is located
db 0x0a

end:
HLT                         ; Stops the processor clock

testvar: 6009               ; Variable which will not be assembled, but can be used anywhere in this file.
                            ; Converted from label by having a value after label to turn it into variable.
                            ; Labels store Memory addresses, while variables store values.

```

Figure 29: Example assembly program that uses registers A and B to count down from ten.

prg01-example.manasbin

```
v3.0 hex words addressed
0000: 0805 2801 5800 8006 7802 000a a800 f800
```

Figure 30: Example assembly program's machine code output in addressed hex format.

The assembler works by going through each line, converting the first one or two clusters of characters into its machine code in hex format, and addressing it by having a maximum of sixteen 16-bit words on each line, each line being sixteen higher than the last. One can make labels, which are temporarily stored variables containing either the memory address of the next instruction or a value if specified after the colon, using a colon at the end of a word and use that wherever needed in the code; this enables the use of loops by combining jump instructions and labels, or directly writing the memory address. The labels themselves are not assembled into the program itself, only stored for use when assembling; however, using *db*, one can define a 16-bit word that will be stored in the machine code of the assembled program itself. Numbers can be in three bases, *hex, binary,* or *decimal*. To write hex numbers, one only needs to prefix the number with *0x*; the prefix *0b* is for binary, while no prefix means the number will be in decimal base. The complete code and notes are in the *Addendum* section under the section *Manas-CPU - Assembler python program*.

3. Conclusion

The project was a success. All goals were met; referring back to the introduction, all project stages were covered. The stages of both planning and creation. No significant problems were met; however, a lot of testing was required to determine the timing of the individual parts, as presumed. There are ways to optimise the circuitry and code and functions to add, like a dedicated division function in the ALU instead of only bit-shifting right for a floor division. Through this report, it is hoped that some light has been shed on the inner workings of a simple but functional CPU; to better appreciate the brilliant minds of humanity has invented.

Acknowledgements:

Thanks and acknowledgement of the efforts of every developer that has worked on both *Logisim* and its successor *Logisim-Evolution,* are needed. Also, acknowledgements are in order for all the brilliant researchers, scientists and mathematicians who publish their work so everyone can learn. Sincerely, thanks to everyone involved.

Sources

- Kluter, Prof.Dr.T. (2022) *Logisim-Evolution*, *Releases Logisim-Evolution*. Available at: https://github.com/logisim-evolution/logisim-evolution/releases (Accessed: 08 June 2023).
- Python (2019). *Welcome to Python.org*. [online] Python.org. Available at: https://www.python.org/. (Accessed: 08 June 2023).
- Hennessy, J.L., Patterson, D.A. and Krste Asanović (2012). *Computer architecture : a quantitative approach*. Waltham, Ma: Morgan Kaufmann.
- Belzer, J. and Kent, A. (1993). *Encyclopedia of computer science and technology Vol. 28 Supplement 13*. New York, Ny [U.A.] Dekker.
- Carl Stephen Clifton (1987). *What every engineer should know about data communications*. New York ; Basel: Marcel Dekker.
- Texas Instruments (1988). Dual D-Type Positive-Edge -Triggered Flip-Flops With Preset And Clear datasheet. [online] Texas Instruments. Available at: https://www.ti.com/lit/ds/symlink/sn74s74.pdf [Accessed 10 Jun. 2023].
- IBM (2017). DRAM - The Invention of On-Demand Data. [online] IBM. Available at: https://www.ibm.com/ibm/history/ibm100/us/en/icons/dram/ [Accessed 11 Jun. 2023].
- Arun Kumar Singh (2006). Digital principles foundation of circuit design and application. New Age International.
- Lilja, D.J. and Sapatnekar, S.S. (2004). Designing Digital Computer Systems with Verilog. Leiden: Cambridge University Press.

Addendum

Microassembler python program:

```
"""
Manas-CPU microassembler for assembling the microprogram
   - Microinstruction 16-bits
   - For custom 16-bit Manas-CPU architecture
"""
"""
Copyright (C) 2023 David Jøssang

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundarer version 3 of the License, or
(at ion) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
"""

NOTHING = 0b0000000000000000

Special_Instructions = {
   "Cpy_ROM":     0b1000000000000000, # Copy rom and enable direct access from counter to address register
| bit 16
   "Chk_Prg_End": 0b0100000000000000, # Check for program end in ROM, jump to 3 in microinstruction
counter | bit 15
   "JZ":          0b0010000000000000, # Jump to address if zero flag set | bit 14
   "EO":          0b1000000000,       # Evaluate ALU operation, addition if not specified otherwise | bit 10
   "CI":          0b100,              # Increment/Enable program counter | bit 3
   "EI":          0b010,              # End instruction                  | bit 2
   "ST":          0b001               # Stop clock                       | bit 1
}

ALU_Instructions = { # bits 13-11
   "SU":0b0010000000000, # Enable subtract mode
   "MU":0b0100000000000, # Enable multiplication mode
   "SL":0b0110000000000, # Enable bitshift left mode
   "SR":0b1000000000000  # Enable bitshift right mode
}

Read_Instructions = { # bits 9-7
```

```
    "RA":0b001000000, #Read A register to bus
    "RB":0b010000000, #Read B register to bus
    "RC":0b011000000, #Read C register to bus
    "RM":0b100000000, #Read from memory at address specified in address register to bus
    "IR":0b101000000, #Read lowest 11bits of instruction register to bus
    "CR":0b110000000  #Read program counter to bus
}

Write_Instructions = { # bits 6-4
    "WA":0b001000, # Write from bus to A register
    "WB":0b010000, # Write from bus to B register
    "WC":0b011000, # Write from bus to C register
    "WM":0b100000, # Write from bus to memory at address specified in address register
    "AW":0b101000, # Write from bus to address register
    "IW":0b110000, # Write from bus to instruction register
    "CW":0b111000  # Write from bus to program counter | acts as a jump instruction
}

Microprogram = [[hex(0x0000) for i in range(16)] for i in range(0x1f)]
# Have them 8 bits for each list, because address and counter just look
FETCH = (f'{(Read_Instructions["CR"] | Write_Instructions["AW"]):04x}',
         f'{(Read_Instructions["RM"] | Write_Instructions["IW"] | Special_Instructions["CI"]):04x}',
         f'{(Special_Instructions["EI"]):04x}')
Assembly_Instructions = {
    "Copy Program ROM": [
        f'{(Special_Instructions["Cpy_ROM"] | Write_Instructions["AW"]):04x}',
        f'{(Special_Instructions["Cpy_ROM"] | Special_Instructions["Chk_Prg_End"]):04x}',
        f'{(Special_Instructions["Cpy_ROM"] | Special_Instructions["CI"] | Write_Instructions["WM"]):04x}',
        f'{(Special_Instructions["EI"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "LDA": [ # Load data from memory to A register | 00001 | LDA <addr> 11-bits
        f'{(Read_Instructions["IR"] | Write_Instructions["AW"]):04x}',
        f'{(Read_Instructions["RM"] | Write_Instructions["WA"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "LDB": [ # Load data from memory to B register | 00010 | LDB <addr> 11-bits
        f'{(Read_Instructions["IR"] | Write_Instructions["AW"]):04x}',
        f'{(Read_Instructions["RM"] | Write_Instructions["WB"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "LDC": [ # Load data from memory to C register | 00011 | LDC <addr> 11-bits
        f'{(Read_Instructions["IR"] | Write_Instructions["AW"]):04x}',
        f'{(Read_Instructions["RM"] | Write_Instructions["WC"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "LDIA": [ # Load data immediately from instruction to A register | 00100 | LDIA <value> 11-bits
        f'{(Read_Instructions["IR"] | Write_Instructions["WA"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "LDIB": [ # Load data immediately from instruction to B register | 00101 | LDIB <value> 11-bits
        f'{(Read_Instructions["IR"] | Write_Instructions["WB"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "LDIC": [ # Load data immediately from instruction to C register | 00110 | LDIC <value> 11-bits
        f'{(Read_Instructions["IR"] | Write_Instructions["WC"]):04x}',
```

```
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "STA": [ # Store A reg into <addr> in memory | 00111
    f'{(Read_Instructions["IR"] | Write_Instructions["AW"]):04x}',
    f'{(Read_Instructions["RA"] | Write_Instructions["WM"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "STB": [ # Store B reg into <addr> in memory | 01000
    f'{(Read_Instructions["IR"] | Write_Instructions["AW"]):04x}',
    f'{(Read_Instructions["RB"] | Write_Instructions["WM"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "STC": [ # Store C reg into <addr> in memory | 01001
    f'{(Read_Instructions["IR"] | Write_Instructions["AW"]):04x}',
    f'{(Read_Instructions["RC"] | Write_Instructions["WM"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],

  "ADD": [ # Add register B to A, and store result in register A | 01010
    f'{(Special_Instructions["EO"] | Write_Instructions["WA"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "SUB": [ # 01011 : Subtract register B from A, and store result in A
    f'{(ALU_Instructions["SU"] | Special_Instructions["EO"] | Write_Instructions["WA"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "MULT": [ # 01100 : Multiply register B with A, store result in A
    f'{(ALU_Instructions["MU"] | Special_Instructions["EO"] | Write_Instructions["WA"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "SHL": [ # 01101 : Shift value in A register by value in B amount to the left
    f'{(ALU_Instructions["SL"] | Special_Instructions["EO"] | Write_Instructions["WA"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "SHR": [ # 01110 : Shift value in A register by value in B amount to the right
    f'{(ALU_Instructions["SR"] | Special_Instructions["EO"] | Write_Instructions["WA"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],

  "JMP": [ # JMP <addr>    01111 : Change program counter to <addr>, changes next instruction
    f'{(Read_Instructions["IR"]):04x}',
    f'{(Read_Instructions["IR"] | Write_Instructions["CW"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
  "JMPZ": [ # JMPZ <addr>   10000 : Jump if zero flag set
    f'{(Read_Instructions["IR"]):04x}',
    f'{(Read_Instructions["IR"] | Special_Instructions["JZ"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],

  #LDAIN and STAOUT allows for 16-bit mem address
  "LDAIN": [ # LDAIN                  10001 : Use A as mem addr then load from memory to A register
    f'{(Read_Instructions["RA"] | Write_Instructions["AW"]):04x}',
    f'{(Read_Instructions["RM"] | Write_Instructions["WA"]):04x}',
    FETCH[0], FETCH[1], FETCH[2]
  ],
```

```
    "STAOUT": [ # STAOUT  10010 : Use register A as mem addr, then load B to memory address
        f'{(Read_Instructions["RA"] | Write_Instructions["AW"]):04x}',
        f'{(Read_Instructions["RB"] | Write_Instructions["WM"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],

    "SWP": [ # SWP                                  10011 : Swap contents in register A and B, overwrites C
        f'{(Read_Instructions["RA"] | Write_Instructions["WC"]):04x}',
        f'{(Read_Instructions["RB"] | Write_Instructions["WA"]):04x}',
        f'{(Read_Instructions["RC"] | Write_Instructions["WB"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],
    "SWPC": [ # SWPC                    10100 : Swap contents in register A and C, overwrites B
        f'{(Read_Instructions["RA"] | Write_Instructions["WB"]):04x}',
        f'{(Read_Instructions["RC"] | Write_Instructions["WA"]):04x}',
        f'{(Read_Instructions["RB"] | Write_Instructions["WC"]):04x}',
        FETCH[0], FETCH[1], FETCH[2]
    ],

    "HLT": [ # HLT                          10101 : Stop the clock
        f'{(Special_Instructions["ST"]):04x}',
        FETCH[0], FETCH[1], FETCH[2] #just add in case something goes wrong
    ],
    "NOP": [ # NOP                            10110 : No operation | just fetch next instruction
        FETCH[0], FETCH[1], FETCH[2]
    ]

}

def printInstructions():
    lineLength = 8
    line = 0
    with open("microprogram.txt", "w") as f:
        print("v3.0 hex words addressed", file=f)
        for instruction in Assembly_Instructions.values():
            remainingInsLength = lineLength - len(instruction)
            for y in range(remainingInsLength):
                instruction.append(f"{0:04x}")
            print(f"{line*8:02x}:", *instruction, file=f)
            line = line + 1

printInstructions()
```

Manas-CPU - Assembler python program:

```
"""
Assembler for the 16-bit Manas-CPU
   - Highest value of instruction argument is 11-bits, 5 bits for instruction code, so 0x7FF.
   - Provide .manas assembly file name or path.
   - Can also include output file name or path as the third argument; however, this is optional, it will default to
"output.manasbin".

Functions:
   - strToInt(number:str) -> int
   - ReadInstructions(file: str) -> list[list]
   - Assemble(instructionList: list[list], outfile: str) -> None
   - run() -> None

Constants:
   - PRG_END

Dictionaries:
   - Instruction_codes -> Contains all the instruction opcodes for the assembly instructions.
"""
"""
Copyright (C) 2023 David Jøssang

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
"""

import sys

#Explanations are in the microassembler
PRG_END = 0b1111100000000000
Instruction_codes = {
   "LDA":     0b0000100000000000,
   "LDB":     0b0001000000000000,
   "LDC":     0b0001100000000000,
   "LDIA":    0b0010000000000000,
   "LDIB":    0b0010100000000000,
   "LDIC":    0b0011000000000000,
   "STA":     0b0011100000000000,
   "STB":     0b0100000000000000,
   "STC":     0b0100100000000000,

   "ADD":     0b0101000000000000,
   "SUB":     0b0101100000000000,
   "MULT":    0b0110000000000000,
   "SHL":     0b0110100000000000,
   "SHR":     0b0111000000000000,
```

```
    "JMP":      0b0111100000000000,
    "JMPZ":     0b1000000000000000,

    "LDAIN":    0b1000100000000000,
    "STAOUT":   0b1001000000000000,

    "SWP":      0b1001100000000000,
    "SWPC":     0b1010000000000000,

    "HLT":      0b1010100000000000,
    "NOP":      0b1011000000000000
}

def strToInt(number: str) -> int:
    "Checks if number in str form is appended by 0x or 0b and converts it into appropriate base integer"
    if (len(number) > 1):
        if (number[0:2] == "0x"):
            return int(number[2:],base=16)
        elif (number[0:2] == "0b"):
            return int(number[2:],base=2)
        else:
            return int(number)
    else:
        return int(number)

def ReadInstructions(file: str) -> list[list]:
    """
    Reads provided .manas assembly file, removes unnecessary whitespace characters and returns list of
    instructions and labels.

    Attributes:
        - file: Name or path to .manas assembly file to read.
    """
    readInstructions = []
    with open(file, "r") as f:
        for line in f:
            instructions = line.rstrip()
            instructions = " ".join(instructions.split()) # Remove all whitespace characters and seperate instructions by
only one space
            instructions = instructions.split(" ")
            if (instructions[0] != "" and instructions[0][0] != ";"):
                readInstructions.append(instructions)
    return readInstructions

def Assemble(instructionList: list[list], outfile: str) -> None:
    """
    Reads provided list of instructions and labels and assembles it into hex addressed machine code for the
    Manas-CPU.

    Attributes:
        - instructionList: List of instructions and labels to assemble.
        - outfile: Name or path to file the function will write to.
    """
    assembledInstructions = []
    lineLength = 16
```

```
    labels = {}
    currentPosition = -1
    adjustment = 0
    #print(instructionList)
    for instruction in instructionList:
        currentPosition += 1
        if (instruction[0][-1] == ":"): # do labels and define bytes
            if (len(instruction) > 1 and instruction[1][0] != ";"): # if there is a value after label, then treat label as a
variable
                labels[instruction[0][0:-1]] = strToInt(instruction[1])
            else: # If label only by itself, treat as address
                labels[instruction[0][0:-1]] = currentPosition - adjustment # Adjust address for amount of labels
            adjustment += 1
    #print(labels)
    currentPosition = -1
    for instruction in instructionList:
        currentPosition += 1
        if (instruction[0][0:-1] in labels): continue
        elif (instruction[0] == "db"): # Define one word of data, placed at current position in program memory
            assembledInstructions.append(f'{strToInt(instruction[1]):04x}')
        elif (instruction[0] in Instruction_codes): # Check if current instruction in opcodes
            if (len(instruction) == 1 or instruction[1][0] == ";"):
                assembledInstructions.append(f'{Instruction_codes[instruction[0]]:04x}')
            elif (instruction[1] in labels): # Check if argument is label, can be address or variable
                assembledInstructions.append(f'{Instruction_codes[instruction[0]] | labels[instruction[1]]:04x}')
            else:
                assembledInstructions.append(f'{Instruction_codes[instruction[0]] | strToInt(instruction[1]):04x}')
        else:
            print(f"Error, cannot assemble instruction: {instruction}!")
    assembledInstructions.append(f'{PRG_END:04x}')
    with open(outfile, "w") as f:
        line = 0
        currentPosition = 0
        print("v3.0 hex words addressed", file=f)
        for i in range(len(assembledInstructions) // lineLength + 1):
            print(f'{(line*lineLength):04x}:', file=f, end=" ")
            for y in range(lineLength):
                if (currentPosition > len(assembledInstructions)-1):
                    break
                print(assembledInstructions[currentPosition], file=f, end=" ")
                currentPosition += 1
    #print(assembledInstructions)

def run() -> None:
    if (len(sys.argv) < 2):
        print(__doc__)
    elif (len(sys.argv) == 2):
        manasAssemblyfile = sys.argv[1]
        outputfile = "output.manasbin"
        insList = ReadInstructions(manasAssemblyfile)
        Assemble(insList, outputfile)
    else:
        manasAssemblyfile = sys.argv[1]
        outputfile = sys.argv[2]
        insList = ReadInstructions(manasAssemblyfile)
        Assemble(insList, outputfile)
```

```
if __name__ == "__main__":
    run()
```

Logic gate circuits made with transistors:

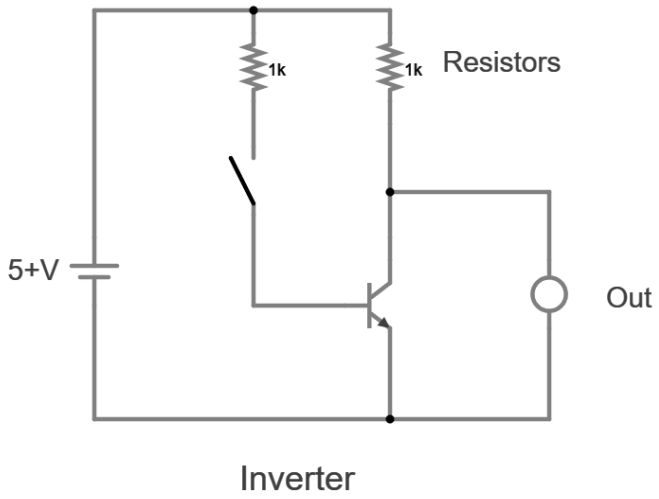

Figure 31: 5V transistor inverter circuit, representing a NOT gate.

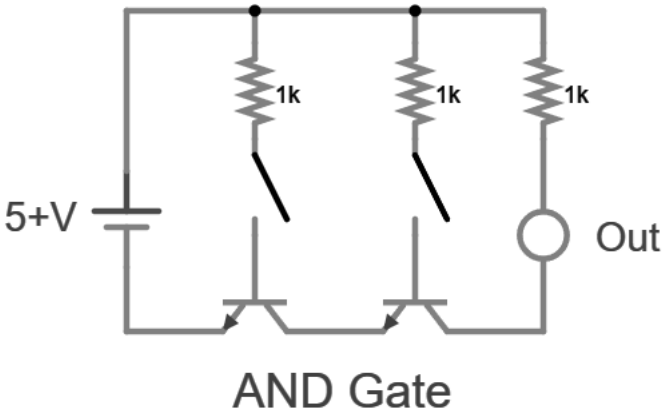

Figure 32: Transistor AND gate circuit.

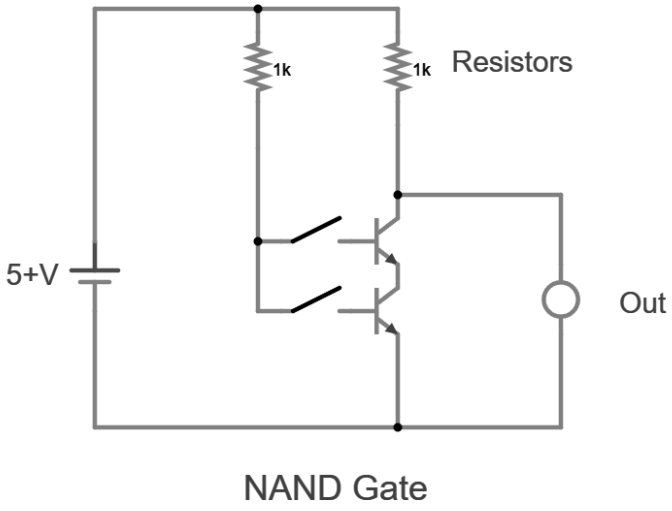

Figure 33: Transistor circuit of NAND gate

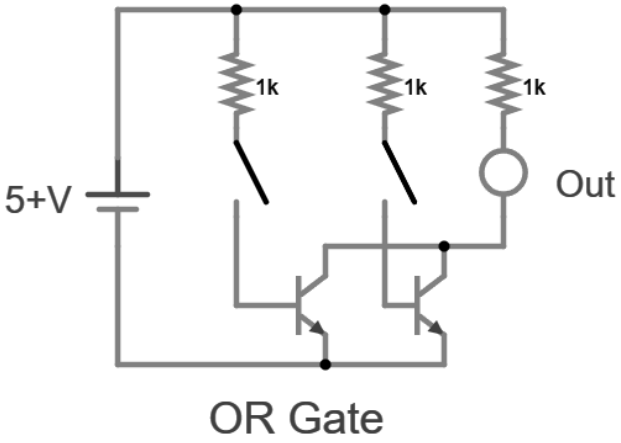

Figure 34: Transistor circuit of OR gate

Manas-CPU - Logisim-Evolution .circ file:

Can copy this into a file and open the file in Logisim-Evolution to view the CPU

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project source="3.8.0" version="1.0">
 This file is intended to be loaded by Logisim-evolution v3.8.0(https://github.com/logisim-evolution/).

 <lib desc="#Wiring" name="0">
  <tool name="Splitter">
   <a name="facing" val="south"/>
  </tool>
  <tool name="Pin">
   <a name="appearance" val="classic"/>
  </tool>
```

```
  <tool name="Probe">
    <a name="appearance" val="classic"/>
    <a name="facing" val="south"/>
  </tool>
  <tool name="Tunnel">
    <a name="facing" val="east"/>
  </tool>
</lib>
<lib desc="#Gates" name="1">
  <tool name="NOT Gate">
    <a name="facing" val="south"/>
  </tool>
  <tool name="Buffer">
    <a name="facing" val="south"/>
  </tool>
  <tool name="AND Gate">
    <a name="facing" val="south"/>
  </tool>
</lib>
<lib desc="#Plexers" name="2"/>
<lib desc="#Arithmetic" name="3"/>
<lib desc="#Memory" name="4"/>
<lib desc="#I/O" name="5"/>
<lib desc="#TTL" name="6"/>
<lib desc="#TCL" name="7"/>
<lib desc="#Base" name="8"/>
<lib desc="#BFH-Praktika" name="9"/>
<lib desc="#Input/Output-Extra" name="10"/>
<lib desc="#Soc" name="11"/>
<main name="main"/>
<options>
  <a name="gateUndefined" val="ignore"/>
  <a name="simlimit" val="1000"/>
  <a name="simrand" val="0"/>
</options>
<mappings>
  <tool lib="8" map="Button2" name="Poke Tool"/>
  <tool lib="8" map="Button3" name="Menu Tool"/>
  <tool lib="8" map="Ctrl Button1" name="Menu Tool"/>
</mappings>
<toolbar>
  <tool lib="8" name="Poke Tool"/>
  <tool lib="8" name="Edit Tool"/>
  <tool lib="8" name="Wiring Tool"/>
  <tool lib="8" name="Text Tool"/>
  <sep/>
  <tool lib="0" name="Pin">
    <a name="facing" val="north"/>
  </tool>
  <tool lib="0" name="Pin">
    <a name="facing" val="north"/>
    <a name="output" val="true"/>
  </tool>
  <sep/>
  <tool lib="1" name="NOT Gate">
    <a name="facing" val="west"/>
  </tool>
  <tool lib="1" name="AND Gate">
    <a name="facing" val="south"/>
  </tool>
  <tool lib="1" name="OR Gate"/>
  <tool lib="1" name="XOR Gate"/>
  <tool lib="1" name="NAND Gate"/>
  <tool lib="1" name="NOR Gate"/>
```

```
  <sep/>
  <tool lib="4" name="D Flip-Flop"/>
  <tool lib="4" name="Register"/>
 </toolbar>
 <circuit name="main">
  <a name="appearance" val="logisim_evolution"/>
  <a name="circuit" val="main"/>
  <a name="circuitnamedboxfixedsize" val="true"/>
  <a name="clabelfont" val="SansSerif bold 14"/>
  <a name="simulationFrequency" val="64.0"/>
  <comp lib="0" loc="(1020,310)" name="Tunnel">
   <a name="facing" val="east"/>
   <a name="label" val="Data_Bus"/>
   <a name="width" val="16"/>
  </comp>
  <comp lib="0" loc="(1020,340)" name="Tunnel">
   <a name="facing" val="east"/>
   <a name="label" val="Write_Ins_Reg"/>
  </comp>
  <comp lib="0" loc="(1020,370)" name="Tunnel">
   <a name="facing" val="east"/>
   <a name="label" val="CLK"/>
  </comp>
  <comp lib="0" loc="(1080,400)" name="Tunnel">
   <a name="facing" val="north"/>
   <a name="label" val="Clear_Btn"/>
  </comp>
  <comp lib="0" loc="(110,170)" name="Tunnel">
   <a name="facing" val="east"/>
   <a name="label" val="Stop_Clock"/>
  </comp>
  <comp lib="0" loc="(1170,340)" name="Probe">
   <a name="appearance" val="NewPins"/>
   <a name="facing" val="north"/>
  </comp>
  <comp lib="0" loc="(1230,320)" name="Splitter">
   <a name="appear" val="right"/>
   <a name="bit1" val="0"/>
   <a name="bit10" val="0"/>
   <a name="bit11" val="1"/>
   <a name="bit12" val="1"/>
   <a name="bit13" val="1"/>
   <a name="bit14" val="1"/>
   <a name="bit15" val="1"/>
   <a name="bit2" val="0"/>
   <a name="bit3" val="0"/>
   <a name="bit4" val="0"/>
   <a name="bit5" val="0"/>
   <a name="bit6" val="0"/>
   <a name="bit7" val="0"/>
   <a name="bit8" val="0"/>
   <a name="bit9" val="0"/>
   <a name="incoming" val="16"/>
  </comp>
  <comp lib="0" loc="(1240,580)" name="Tunnel">
   <a name="facing" val="north"/>
   <a name="label" val="ZF_In"/>
  </comp>
  <comp lib="0" loc="(1240,790)" name="Probe">
   <a name="appearance" val="NewPins"/>
   <a name="facing" val="south"/>
  </comp>
  <comp lib="0" loc="(1240,830)" name="Probe">
   <a name="appearance" val="NewPins"/>
```

```
  <a name="facing" val="north"/>
  <a name="radix" val="16"/>
 </comp>
 <comp lib="0" loc="(1270,810)" name="Splitter">
  <a name="appear" val="right"/>
  <a name="bit0" val="6"/>
  <a name="bit1" val="5"/>
  <a name="bit10" val="1"/>
  <a name="bit11" val="1"/>
  <a name="bit12" val="1"/>
  <a name="bit13" val="7"/>
  <a name="bit14" val="8"/>
  <a name="bit15" val="9"/>
  <a name="bit2" val="4"/>
  <a name="bit4" val="3"/>
  <a name="bit5" val="3"/>
  <a name="bit6" val="2"/>
  <a name="bit7" val="2"/>
  <a name="bit8" val="2"/>
  <a name="bit9" val="0"/>
  <a name="fanout" val="10"/>
  <a name="incoming" val="16"/>
 </comp>
 <comp lib="0" loc="(1290,1090)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Stop_Clock"/>
 </comp>
 <comp lib="0" loc="(1290,1120)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EI"/>
 </comp>
 <comp lib="0" loc="(1290,1150)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Enable_Prg_Counter"/>
 </comp>
 <comp lib="0" loc="(1290,1210)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Copy_Prg_ROM"/>
 </comp>
 <comp lib="0" loc="(1290,1240)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Check_EndPrg"/>
 </comp>
 <comp lib="0" loc="(1290,1270)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="JZ"/>
 </comp>
 <comp lib="0" loc="(1290,350)" name="Constant">
  <a name="value" val="0x0"/>
  <a name="width" val="5"/>
 </comp>
 <comp lib="0" loc="(1290,390)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
 </comp>
 <comp lib="0" loc="(1290,490)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="EO"/>
 </comp>
 <comp lib="0" loc="(1310,540)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
 </comp>
 <comp lib="0" loc="(1320,510)" name="Tunnel">
```

```
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="0" loc="(1330,240)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Ins_Reg"/>
 </comp>
 <comp lib="0" loc="(1330,320)" name="Splitter">
  <a name="bit1" val="0"/>
  <a name="bit10" val="0"/>
  <a name="bit11" val="1"/>
  <a name="bit12" val="1"/>
  <a name="bit13" val="1"/>
  <a name="bit14" val="1"/>
  <a name="bit15" val="1"/>
  <a name="bit2" val="0"/>
  <a name="bit3" val="0"/>
  <a name="bit4" val="0"/>
  <a name="bit5" val="0"/>
  <a name="bit6" val="0"/>
  <a name="bit7" val="0"/>
  <a name="bit8" val="0"/>
  <a name="bit9" val="0"/>
  <a name="facing" val="west"/>
  <a name="incoming" val="16"/>
 </comp>
 <comp lib="0" loc="(1330,370)" name="Tunnel">
  <a name="label" val="Instruction_Decoder"/>
  <a name="width" val="5"/>
 </comp>
 <comp lib="0" loc="(1330,770)" name="Tunnel">
  <a name="label" val="EO"/>
 </comp>
 <comp lib="0" loc="(1360,820)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="south"/>
 </comp>
 <comp lib="0" loc="(1370,240)" name="Tunnel">
  <a name="label" val="Read_Ins_Reg"/>
 </comp>
 <comp lib="0" loc="(1380,1110)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
 </comp>
 <comp lib="0" loc="(1390,1050)" name="Constant"/>
 <comp lib="0" loc="(140,200)" name="Clock"/>
 <comp lib="0" loc="(1410,790)" name="Constant"/>
 <comp lib="0" loc="(1410,950)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
 </comp>
 <comp lib="0" loc="(1420,300)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="south"/>
 </comp>
 <comp lib="0" loc="(1420,890)" name="Constant"/>
 <comp lib="0" loc="(1460,320)" name="Tunnel">
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="0" loc="(1480,800)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="SR"/>
 </comp>
```

```
<comp lib="0" loc="(1490,760)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="SU"/>
</comp>
<comp lib="0" loc="(1500,770)" name="Tunnel">
  <a name="label" val="MU"/>
</comp>
<comp lib="0" loc="(1510,470)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="Copy_Prg_ROM"/>
</comp>
<comp lib="0" loc="(1510,800)" name="Tunnel">
  <a name="label" val="SL"/>
</comp>
<comp lib="0" loc="(1530,1010)" name="Tunnel">
  <a name="label" val="Write_Reg_A"/>
</comp>
<comp lib="0" loc="(1530,1040)" name="Tunnel">
  <a name="label" val="Write_Reg_B"/>
</comp>
<comp lib="0" loc="(1530,1070)" name="Tunnel">
  <a name="label" val="Write_Reg_C"/>
</comp>
<comp lib="0" loc="(1530,1100)" name="Tunnel">
  <a name="label" val="Write_Mem"/>
</comp>
<comp lib="0" loc="(1530,1130)" name="Tunnel">
  <a name="label" val="Write_Addr_Reg"/>
</comp>
<comp lib="0" loc="(1530,1160)" name="Tunnel">
  <a name="label" val="Write_Ins_Reg"/>
</comp>
<comp lib="0" loc="(1530,1190)" name="Tunnel">
  <a name="label" val="Write_Prg_Counter"/>
</comp>
<comp lib="0" loc="(1530,550)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(1530,640)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Prg_end_clear"/>
</comp>
<comp lib="0" loc="(1560,820)" name="Tunnel">
  <a name="label" val="Read_Reg_A"/>
  <a name="labelfont" val="SansSerif bold 14"/>
</comp>
<comp lib="0" loc="(1560,850)" name="Tunnel">
  <a name="label" val="Read_Reg_B"/>
  <a name="labelfont" val="SansSerif bold 14"/>
</comp>
<comp lib="0" loc="(1560,880)" name="Tunnel">
  <a name="label" val="Read_Reg_C"/>
  <a name="labelfont" val="SansSerif bold 14"/>
</comp>
<comp lib="0" loc="(1560,910)" name="Tunnel">
  <a name="label" val="Read_Mem"/>
</comp>
<comp lib="0" loc="(1560,940)" name="Tunnel">
  <a name="label" val="Read_Ins_Reg"/>
</comp>
<comp lib="0" loc="(1560,970)" name="Tunnel">
  <a name="label" val="Read_Prg_Counter"/>
```

```
</comp>
<comp lib="0" loc="(1570,580)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Addr_Reg"/>
</comp>
<comp lib="0" loc="(1570,610)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(1580,420)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Prg_Counter"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(1660,100)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="JZ"/>
</comp>
<comp lib="0" loc="(1660,660)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Mem"/>
</comp>
<comp lib="0" loc="(1660,690)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Read_Mem"/>
</comp>
<comp lib="0" loc="(1690,80)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="ZF"/>
</comp>
<comp lib="0" loc="(1700,720)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(1730,130)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Prg_Counter"/>
</comp>
<comp lib="0" loc="(1730,520)" name="Tunnel">
  <a name="label" val="Addr_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(1980,210)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(2000,100)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Prg_end_clear"/>
</comp>
<comp lib="0" loc="(2000,160)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Enable_Prg_Counter"/>
</comp>
<comp lib="0" loc="(2040,180)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(2040,660)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Read_Mem"/>
</comp>
```

```
<comp lib="0" loc="(2060,1010)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Addr_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(2070,630)" name="Tunnel">
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(210,250)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Inv_Clock"/>
</comp>
<comp lib="0" loc="(2340,160)" name="Tunnel">
  <a name="label" val="Prg_Counter"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(2350,1130)" name="Splitter">
  <a name="bit0" val="none"/>
  <a name="bit1" val="none"/>
  <a name="bit10" val="none"/>
  <a name="bit11" val="0"/>
  <a name="bit12" val="1"/>
  <a name="bit13" val="2"/>
  <a name="bit14" val="3"/>
  <a name="bit15" val="4"/>
  <a name="bit2" val="none"/>
  <a name="bit3" val="none"/>
  <a name="bit4" val="none"/>
  <a name="bit5" val="none"/>
  <a name="bit6" val="none"/>
  <a name="bit7" val="none"/>
  <a name="bit8" val="none"/>
  <a name="bit9" val="none"/>
  <a name="facing" val="south"/>
  <a name="fanout" val="5"/>
  <a name="incoming" val="16"/>
</comp>
<comp lib="0" loc="(2380,210)" name="Tunnel">
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(2380,260)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Read_Prg_Counter"/>
</comp>
<comp lib="0" loc="(2430,1110)" name="Tunnel">
  <a name="label" val="Copy_Prg_ROM"/>
</comp>
<comp lib="0" loc="(2440,1010)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(2470,1240)" name="Tunnel">
  <a name="label" val="End_Prg"/>
</comp>
<comp lib="0" loc="(260,200)" name="Tunnel">
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(330,820)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="ZF_In"/>
</comp>
```

```
<comp lib="0" loc="(330,850)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EO"/>
</comp>
<comp lib="0" loc="(330,880)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(380,360)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(380,390)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Reg_A"/>
</comp>
<comp lib="0" loc="(380,420)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(380,490)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(380,520)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Reg_B"/>
</comp>
<comp lib="0" loc="(380,550)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(380,620)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(380,650)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Write_Reg_C"/>
</comp>
<comp lib="0" loc="(380,680)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(440,830)" name="Tunnel">
  <a name="label" val="ZF"/>
</comp>
<comp lib="0" loc="(500,200)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(520,400)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="ALU_In_A"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(520,530)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="ALU_In_B"/>
  <a name="width" val="16"/>
```

```
</comp>
<comp lib="0" loc="(530,200)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="west"/>
</comp>
<comp lib="0" loc="(530,940)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Check_EndPrg"/>
</comp>
<comp lib="0" loc="(530,970)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="End_Prg"/>
</comp>
<comp lib="0" loc="(540,660)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Read_Reg_C"/>
</comp>
<comp lib="0" loc="(540,790)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EI"/>
</comp>
<comp lib="0" loc="(560,370)" name="Tunnel">
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(560,500)" name="Tunnel">
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(560,630)" name="Tunnel">
  <a name="label" val="Data_Bus"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(580,760)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Clear_Btn"/>
</comp>
<comp lib="0" loc="(580,870)" name="Constant">
  <a name="value" val="0x4"/>
  <a name="width" val="3"/>
</comp>
<comp lib="0" loc="(590,1030)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Clear_Btn"/>
</comp>
<comp lib="0" loc="(600,840)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Inv_Clock"/>
</comp>
<comp lib="0" loc="(610,1090)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Prg_end_clear"/>
</comp>
<comp lib="0" loc="(620,400)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Read_Reg_A"/>
</comp>
<comp lib="0" loc="(620,530)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Read_Reg_B"/>
</comp>
<comp lib="0" loc="(760,1010)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Micro_ovrflw"/>
```

```
</comp>
<comp lib="0" loc="(760,1070)" name="Splitter">
  <a name="appear" val="right"/>
  <a name="facing" val="south"/>
  <a name="fanout" val="3"/>
  <a name="incoming" val="3"/>
</comp>
<comp lib="0" loc="(760,980)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EI"/>
</comp>
<comp lib="0" loc="(770,200)" name="Tunnel">
  <a name="label" val="Clear_Btn"/>
</comp>
<comp lib="0" loc="(840,1010)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(840,950)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Instruction_Decoder"/>
  <a name="width" val="5"/>
</comp>
<comp lib="0" loc="(850,720)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="Micro_ovrflw"/>
</comp>
<comp lib="0" loc="(930,590)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="MU"/>
</comp>
<comp lib="0" loc="(930,810)" name="Splitter">
  <a name="bit1" val="0"/>
  <a name="bit2" val="0"/>
  <a name="bit3" val="1"/>
  <a name="bit4" val="1"/>
  <a name="bit5" val="1"/>
  <a name="bit6" val="1"/>
  <a name="bit7" val="1"/>
  <a name="facing" val="west"/>
  <a name="incoming" val="8"/>
</comp>
<comp lib="0" loc="(940,750)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="south"/>
  <a name="radix" val="16"/>
</comp>
<comp lib="0" loc="(960,570)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="SU"/>
</comp>
<comp lib="0" loc="(960,610)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="SL"/>
</comp>
<comp lib="0" loc="(960,650)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(990,490)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="ALU_In_A"/>
  <a name="width" val="16"/>
</comp>
```

```
<comp lib="0" loc="(990,520)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="ALU_In_B"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(990,550)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EO"/>
</comp>
<comp lib="0" loc="(990,630)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="SR"/>
</comp>
<comp lib="1" loc="(1300,510)" name="Controlled Buffer">
  <a name="control" val="left"/>
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(1350,300)" name="AND Gate">
  <a name="facing" val="south"/>
  <a name="negate0" val="true"/>
  <a name="size" val="30"/>
</comp>
<comp lib="1" loc="(1360,320)" name="Controlled Buffer">
  <a name="control" val="left"/>
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(1510,530)" name="NOT Gate">
  <a name="facing" val="south"/>
</comp>
<comp lib="1" loc="(1570,550)" name="Controlled Buffer">
  <a name="control" val="left"/>
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(160,170)" name="NOT Gate"/>
<comp lib="1" loc="(1600,490)" name="Controlled Buffer">
  <a name="facing" val="south"/>
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(170,200)" name="Controlled Buffer">
  <a name="control" val="left"/>
</comp>
<comp lib="1" loc="(1740,100)" name="AND Gate">
  <a name="size" val="30"/>
</comp>
<comp lib="1" loc="(1820,120)" name="OR Gate">
  <a name="size" val="30"/>
</comp>
<comp lib="1" loc="(2040,630)" name="Controlled Buffer">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(210,240)" name="NOT Gate">
  <a name="facing" val="south"/>
</comp>
<comp lib="1" loc="(2350,210)" name="Controlled Buffer">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(2380,1210)" name="AND Gate">
  <a name="facing" val="south"/>
  <a name="inputs" val="5"/>
</comp>
<comp lib="1" loc="(2440,1060)" name="Controlled Buffer">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(550,370)" name="Controlled Buffer">
```

```
  <a name="width" val="16"/>
 </comp>
 <comp lib="1" loc="(550,500)" name="Controlled Buffer">
  <a name="width" val="16"/>
 </comp>
 <comp lib="1" loc="(550,630)" name="Controlled Buffer">
  <a name="width" val="16"/>
 </comp>
 <comp lib="1" loc="(580,800)" name="NOT Gate">
  <a name="facing" val="north"/>
  <a name="size" val="20"/>
 </comp>
 <comp lib="1" loc="(590,790)" name="Controlled Buffer"/>
 <comp lib="1" loc="(600,950)" name="AND Gate">
  <a name="size" val="30"/>
 </comp>
 <comp lib="1" loc="(630,1080)" name="OR Gate">
  <a name="facing" val="south"/>
  <a name="size" val="30"/>
 </comp>
 <comp lib="1" loc="(630,780)" name="OR Gate">
  <a name="size" val="30"/>
 </comp>
 <comp lib="1" loc="(830,980)" name="OR Gate">
  <a name="size" val="30"/>
 </comp>
 <comp lib="2" loc="(1400,1050)" name="Demultiplexer">
  <a name="select" val="3"/>
 </comp>
 <comp lib="2" loc="(1420,790)" name="Demultiplexer">
  <a name="select" val="3"/>
 </comp>
 <comp lib="2" loc="(1430,890)" name="Demultiplexer">
  <a name="select" val="3"/>
 </comp>
 <comp lib="4" loc="(1050,290)" name="Register">
  <a name="appearance" val="logisim_evolution"/>
  <a name="label" val="Instruction_Register"/>
  <a name="showInTab" val="true"/>
  <a name="trigger" val="falling"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="4" loc="(1590,520)" name="Register">
  <a name="appearance" val="logisim_evolution"/>
  <a name="label" val="Mem_Address_Register"/>
  <a name="showInTab" val="true"/>
  <a name="trigger" val="falling"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="4" loc="(1740,540)" name="RAM">
  <a name="addrWidth" val="16"/>
  <a name="appearance" val="logisim_evolution"/>
  <a name="dataWidth" val="16"/>
  <a name="label" val="Memory"/>
 </comp>
 <comp lib="4" loc="(2090,100)" name="Counter">
  <a name="appearance" val="logisim_evolution"/>
  <a name="label" val="Program_Counter"/>
  <a name="max" val="0xffff"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="4" loc="(2090,1000)" name="ROM">
  <a name="addrWidth" val="16"/>
  <a name="appearance" val="logisim_evolution"/>
```

```
    <a name="contents">addr/data: 16 16
805 2801 5800 8006 7802 a a800 f800
</a>
    <a name="dataWidth" val="16"/>
    <a name="label" val="Program_ROM"/>
    <a name="labelvisible" val="true"/>
   </comp>
   <comp lib="4" loc="(2400,1230)" name="T Flip-Flop">
    <a name="appearance" val="logisim_evolution"/>
   </comp>
   <comp lib="4" loc="(360,800)" name="Register">
    <a name="appearance" val="logisim_evolution"/>
    <a name="label" val="Zero_Flag_Register"/>
    <a name="showInTab" val="true"/>
    <a name="trigger" val="high"/>
    <a name="width" val="1"/>
   </comp>
   <comp lib="4" loc="(410,340)" name="Register">
    <a name="appearance" val="logisim_evolution"/>
    <a name="label" val="Register_A"/>
    <a name="showInTab" val="true"/>
    <a name="trigger" val="high"/>
    <a name="width" val="16"/>
   </comp>
   <comp lib="4" loc="(410,470)" name="Register">
    <a name="appearance" val="logisim_evolution"/>
    <a name="label" val="Register_B"/>
    <a name="showInTab" val="true"/>
    <a name="trigger" val="high"/>
    <a name="width" val="16"/>
   </comp>
   <comp lib="4" loc="(410,600)" name="Register">
    <a name="appearance" val="logisim_evolution"/>
    <a name="label" val="Register_C"/>
    <a name="showInTab" val="true"/>
    <a name="trigger" val="high"/>
    <a name="width" val="16"/>
   </comp>
   <comp lib="4" loc="(650,760)" name="Counter">
    <a name="appearance" val="logisim_evolution"/>
    <a name="label" val="Microinstruction_Counter"/>
    <a name="max" val="0x7"/>
    <a name="width" val="3"/>
   </comp>
   <comp lib="4" loc="(870,920)" name="Register">
    <a name="appearance" val="logisim_evolution"/>
    <a name="trigger" val="high"/>
    <a name="width" val="5"/>
   </comp>
   <comp lib="4" loc="(960,750)" name="ROM">
    <a name="appearance" val="logisim_evolution"/>
    <a name="contents">addr/data: 8 16
8028 c000 8024 2 1a8 134 2 0
168 108 1a8 134 2 0 0 0
168 110 1a8 134 2 0 0 0
168 118 1a8 134 2 0 0 0
148 1a8 134 2 4*0 150 1a8 134
2 4*0 158 1a8 134 2 4*0 168
60 1a8 134 2 0 0 0 168
a0 1a8 134 2 0 0 0 168
e0 1a8 134 2 0 0 0 208
1a8 134 2 4*0 608 1a8 134 2
4*0 a08 1a8 134 2 4*0 e08 1a8
134 2 4*0 1208 1a8 134 2 4*0
```

```
140 178 1a8 134 2 0 0 0
140 2140 1a8 134 2 0 0 0
68 108 1a8 134 2 0 0 0
68 a0 1a8 134 2 0 0 0
58 88 d0 1a8 134 2 0 0
50 c8 98 1a8 134 2 0 0
1 1a8 134 2 4*0 1a8 134 2
</a>
    <a name="dataWidth" val="16"/>
    <a name="label" val="Microprogram_ROM"/>
    <a name="labelvisible" val="true"/>
  </comp>
  <comp lib="5" loc="(720,1090)" name="DipSwitch">
    <a name="number" val="3"/>
  </comp>
  <comp lib="5" loc="(740,200)" name="Button"/>
  <comp lib="8" loc="(1105,485)" name="Text">
    <a name="text" val="ALU"/>
  </comp>
  <comp lib="8" loc="(1360,205)" name="Text">
    <a name="text" val="Read lowest 11-bits if Read enabled and Write not enabled"/>
  </comp>
  <comp lib="8" loc="(1855,505)" name="Text">
    <a name="text" val="RAM"/>
  </comp>
  <comp lib="8" loc="(205,165)" name="Text">
    <a name="text" val="Clock"/>
  </comp>
  <comp lib="8" loc="(435,300)" name="Text">
    <a name="text" val="General registers"/>
  </comp>
  <comp loc="(1220,510)" name="ALU"/>
  <wire from="(1000,490)" to="(1000,510)"/>
  <wire from="(1000,520)" to="(1000,530)"/>
  <wire from="(1020,310)" to="(1050,310)"/>
  <wire from="(1020,340)" to="(1050,340)"/>
  <wire from="(1020,370)" to="(1050,370)"/>
  <wire from="(1050,310)" to="(1050,320)"/>
  <wire from="(1050,360)" to="(1050,370)"/>
  <wire from="(1080,380)" to="(1080,400)"/>
  <wire from="(110,170)" to="(130,170)"/>
  <wire from="(1110,320)" to="(1170,320)"/>
  <wire from="(1170,320)" to="(1170,340)"/>
  <wire from="(1170,320)" to="(1230,320)"/>
  <wire from="(1200,810)" to="(1240,810)"/>
  <wire from="(1220,510)" to="(1270,510)"/>
  <wire from="(1220,530)" to="(1240,530)"/>
  <wire from="(1240,530)" to="(1240,580)"/>
  <wire from="(1240,790)" to="(1240,810)"/>
  <wire from="(1240,810)" to="(1240,830)"/>
  <wire from="(1240,810)" to="(1270,810)"/>
  <wire from="(1250,330)" to="(1310,330)"/>
  <wire from="(1250,340)" to="(1250,370)"/>
  <wire from="(1250,370)" to="(1290,370)"/>
  <wire from="(1270,510)" to="(1270,530)"/>
  <wire from="(1270,510)" to="(1280,510)"/>
  <wire from="(1270,530)" to="(1310,530)"/>
  <wire from="(1290,1090)" to="(1330,1090)"/>
  <wire from="(1290,1120)" to="(1340,1120)"/>
  <wire from="(1290,1150)" to="(1350,1150)"/>
  <wire from="(1290,1210)" to="(1300,1210)"/>
  <wire from="(1290,1240)" to="(1310,1240)"/>
  <wire from="(1290,1270)" to="(1320,1270)"/>
  <wire from="(1290,350)" to="(1300,350)"/>
```

```
<wire from="(1290,370)" to="(1290,390)"/>
<wire from="(1290,370)" to="(1330,370)"/>
<wire from="(1290,490)" to="(1290,500)"/>
<wire from="(1290,820)" to="(1320,820)"/>
<wire from="(1290,830)" to="(1360,830)"/>
<wire from="(1290,840)" to="(1390,840)"/>
<wire from="(1290,850)" to="(1360,850)"/>
<wire from="(1290,860)" to="(1350,860)"/>
<wire from="(1290,870)" to="(1340,870)"/>
<wire from="(1290,880)" to="(1330,880)"/>
<wire from="(1290,890)" to="(1320,890)"/>
<wire from="(1290,900)" to="(1310,900)"/>
<wire from="(1290,910)" to="(1300,910)"/>
<wire from="(1300,340)" to="(1300,350)"/>
<wire from="(1300,340)" to="(1310,340)"/>
<wire from="(1300,510)" to="(1320,510)"/>
<wire from="(1300,910)" to="(1300,1210)"/>
<wire from="(1310,530)" to="(1310,540)"/>
<wire from="(1310,900)" to="(1310,1240)"/>
<wire from="(1320,770)" to="(1320,820)"/>
<wire from="(1320,770)" to="(1330,770)"/>
<wire from="(1320,890)" to="(1320,1270)"/>
<wire from="(1330,240)" to="(1340,240)"/>
<wire from="(1330,320)" to="(1340,320)"/>
<wire from="(1330,880)" to="(1330,1090)"/>
<wire from="(1340,240)" to="(1340,260)"/>
<wire from="(1340,870)" to="(1340,1120)"/>
<wire from="(1350,300)" to="(1350,310)"/>
<wire from="(1350,860)" to="(1350,1150)"/>
<wire from="(1360,1090)" to="(1380,1090)"/>
<wire from="(1360,240)" to="(1360,270)"/>
<wire from="(1360,240)" to="(1370,240)"/>
<wire from="(1360,320)" to="(1420,320)"/>
<wire from="(1360,820)" to="(1360,830)"/>
<wire from="(1360,830)" to="(1440,830)"/>
<wire from="(1360,850)" to="(1360,1090)"/>
<wire from="(1380,1090)" to="(1380,1110)"/>
<wire from="(1380,1090)" to="(1420,1090)"/>
<wire from="(1390,1050)" to="(1400,1050)"/>
<wire from="(1390,840)" to="(1390,930)"/>
<wire from="(1390,930)" to="(1410,930)"/>
<wire from="(140,200)" to="(150,200)"/>
<wire from="(1410,790)" to="(1420,790)"/>
<wire from="(1410,930)" to="(1410,950)"/>
<wire from="(1410,930)" to="(1450,930)"/>
<wire from="(1420,300)" to="(1420,320)"/>
<wire from="(1420,320)" to="(1460,320)"/>
<wire from="(1420,890)" to="(1430,890)"/>
<wire from="(1440,1020)" to="(1460,1020)"/>
<wire from="(1440,1030)" to="(1520,1030)"/>
<wire from="(1440,1040)" to="(1510,1040)"/>
<wire from="(1440,1050)" to="(1500,1050)"/>
<wire from="(1440,1060)" to="(1490,1060)"/>
<wire from="(1440,1070)" to="(1470,1070)"/>
<wire from="(1440,1080)" to="(1450,1080)"/>
<wire from="(1450,1080)" to="(1450,1190)"/>
<wire from="(1450,1190)" to="(1530,1190)"/>
<wire from="(1460,1010)" to="(1460,1020)"/>
<wire from="(1460,1010)" to="(1530,1010)"/>
<wire from="(1460,760)" to="(1490,760)"/>
<wire from="(1460,770)" to="(1500,770)"/>
<wire from="(1460,780)" to="(1500,780)"/>
<wire from="(1460,790)" to="(1480,790)"/>
<wire from="(1470,1070)" to="(1470,1160)"/>
```

```
<wire from="(1470,1160)" to="(1530,1160)"/>
<wire from="(1470,860)" to="(1480,860)"/>
<wire from="(1470,870)" to="(1540,870)"/>
<wire from="(1470,880)" to="(1560,880)"/>
<wire from="(1470,890)" to="(1540,890)"/>
<wire from="(1470,900)" to="(1530,900)"/>
<wire from="(1470,910)" to="(1520,910)"/>
<wire from="(1480,790)" to="(1480,800)"/>
<wire from="(1480,840)" to="(1480,860)"/>
<wire from="(1480,840)" to="(1540,840)"/>
<wire from="(1490,1060)" to="(1490,1130)"/>
<wire from="(1490,1130)" to="(1530,1130)"/>
<wire from="(1500,1050)" to="(1500,1100)"/>
<wire from="(1500,1100)" to="(1530,1100)"/>
<wire from="(1500,780)" to="(1500,800)"/>
<wire from="(1500,800)" to="(1510,800)"/>
<wire from="(1510,1040)" to="(1510,1070)"/>
<wire from="(1510,1070)" to="(1530,1070)"/>
<wire from="(1510,470)" to="(1510,480)"/>
<wire from="(1510,480)" to="(1510,500)"/>
<wire from="(1510,480)" to="(1590,480)"/>
<wire from="(1510,530)" to="(1560,530)"/>
<wire from="(1520,1030)" to="(1520,1040)"/>
<wire from="(1520,1040)" to="(1530,1040)"/>
<wire from="(1520,910)" to="(1520,970)"/>
<wire from="(1520,970)" to="(1560,970)"/>
<wire from="(1530,550)" to="(1550,550)"/>
<wire from="(1530,640)" to="(1620,640)"/>
<wire from="(1530,900)" to="(1530,940)"/>
<wire from="(1530,940)" to="(1560,940)"/>
<wire from="(1540,820)" to="(1540,840)"/>
<wire from="(1540,820)" to="(1560,820)"/>
<wire from="(1540,850)" to="(1540,870)"/>
<wire from="(1540,850)" to="(1560,850)"/>
<wire from="(1540,890)" to="(1540,910)"/>
<wire from="(1540,910)" to="(1560,910)"/>
<wire from="(1560,530)" to="(1560,540)"/>
<wire from="(1570,550)" to="(1580,550)"/>
<wire from="(1570,580)" to="(1590,580)"/>
<wire from="(1570,610)" to="(1580,610)"/>
<wire from="(1580,420)" to="(1600,420)"/>
<wire from="(1580,490)" to="(1580,550)"/>
<wire from="(1580,490)" to="(1600,490)"/>
<wire from="(1580,550)" to="(1590,550)"/>
<wire from="(1580,610)" to="(1580,620)"/>
<wire from="(1580,610)" to="(1590,610)"/>
<wire from="(1580,620)" to="(1660,620)"/>
<wire from="(1590,570)" to="(1590,580)"/>
<wire from="(1590,590)" to="(1590,610)"/>
<wire from="(160,170)" to="(160,190)"/>
<wire from="(1600,420)" to="(1600,470)"/>
<wire from="(1620,610)" to="(1620,640)"/>
<wire from="(1650,550)" to="(1710,550)"/>
<wire from="(1660,100)" to="(1670,100)"/>
<wire from="(1660,610)" to="(1660,620)"/>
<wire from="(1660,610)" to="(1740,610)"/>
<wire from="(1660,660)" to="(1670,660)"/>
<wire from="(1660,690)" to="(1680,690)"/>
<wire from="(1670,100)" to="(1670,110)"/>
<wire from="(1670,110)" to="(1710,110)"/>
<wire from="(1670,590)" to="(1670,660)"/>
<wire from="(1670,590)" to="(1740,590)"/>
<wire from="(1680,600)" to="(1680,690)"/>
<wire from="(1680,600)" to="(1740,600)"/>
```

```
<wire from="(1690,80)" to="(1700,80)"/>
<wire from="(170,200)" to="(210,200)"/>
<wire from="(1700,720)" to="(1710,720)"/>
<wire from="(1700,80)" to="(1700,90)"/>
<wire from="(1700,90)" to="(1710,90)"/>
<wire from="(1710,520)" to="(1710,550)"/>
<wire from="(1710,520)" to="(1730,520)"/>
<wire from="(1710,550)" to="(1740,550)"/>
<wire from="(1710,630)" to="(1710,720)"/>
<wire from="(1710,630)" to="(1740,630)"/>
<wire from="(1730,130)" to="(1790,130)"/>
<wire from="(1740,100)" to="(1770,100)"/>
<wire from="(1770,100)" to="(1770,110)"/>
<wire from="(1770,110)" to="(1790,110)"/>
<wire from="(1820,120)" to="(1840,120)"/>
<wire from="(1840,120)" to="(1840,130)"/>
<wire from="(1840,130)" to="(2090,130)"/>
<wire from="(1980,210)" to="(2090,210)"/>
<wire from="(1980,630)" to="(2020,630)"/>
<wire from="(2000,100)" to="(2090,100)"/>
<wire from="(2000,160)" to="(2080,160)"/>
<wire from="(2030,640)" to="(2040,640)"/>
<wire from="(2040,180)" to="(2090,180)"/>
<wire from="(2040,630)" to="(2070,630)"/>
<wire from="(2040,640)" to="(2040,660)"/>
<wire from="(2060,1010)" to="(2090,1010)"/>
<wire from="(2080,160)" to="(2080,170)"/>
<wire from="(2080,170)" to="(2090,170)"/>
<wire from="(2090,100)" to="(2090,120)"/>
<wire from="(210,200)" to="(210,210)"/>
<wire from="(210,200)" to="(260,200)"/>
<wire from="(210,240)" to="(210,250)"/>
<wire from="(2290,210)" to="(2310,210)"/>
<wire from="(2310,160)" to="(2310,210)"/>
<wire from="(2310,160)" to="(2340,160)"/>
<wire from="(2310,210)" to="(2330,210)"/>
<wire from="(2330,1060)" to="(2350,1060)"/>
<wire from="(2340,220)" to="(2340,230)"/>
<wire from="(2340,230)" to="(2380,230)"/>
<wire from="(2350,1060)" to="(2350,1130)"/>
<wire from="(2350,1060)" to="(2420,1060)"/>
<wire from="(2350,210)" to="(2380,210)"/>
<wire from="(2360,1150)" to="(2360,1160)"/>
<wire from="(2370,1150)" to="(2370,1160)"/>
<wire from="(2380,1150)" to="(2380,1160)"/>
<wire from="(2380,1210)" to="(2380,1240)"/>
<wire from="(2380,1240)" to="(2380,1280)"/>
<wire from="(2380,1240)" to="(2390,1240)"/>
<wire from="(2380,1280)" to="(2390,1280)"/>
<wire from="(2380,230)" to="(2380,260)"/>
<wire from="(2390,1150)" to="(2390,1160)"/>
<wire from="(2400,1150)" to="(2400,1160)"/>
<wire from="(2430,1070)" to="(2430,1110)"/>
<wire from="(2440,1010)" to="(2440,1060)"/>
<wire from="(2450,1240)" to="(2470,1240)"/>
<wire from="(330,820)" to="(360,820)"/>
<wire from="(330,850)" to="(360,850)"/>
<wire from="(330,880)" to="(360,880)"/>
<wire from="(360,820)" to="(360,830)"/>
<wire from="(360,870)" to="(360,880)"/>
<wire from="(380,360)" to="(410,360)"/>
<wire from="(380,390)" to="(410,390)"/>
<wire from="(380,420)" to="(410,420)"/>
<wire from="(380,490)" to="(410,490)"/>
```

```
<wire from="(380,520)" to="(410,520)"/>
<wire from="(380,550)" to="(410,550)"/>
<wire from="(380,620)" to="(410,620)"/>
<wire from="(380,650)" to="(410,650)"/>
<wire from="(380,680)" to="(410,680)"/>
<wire from="(410,360)" to="(410,370)"/>
<wire from="(410,410)" to="(410,420)"/>
<wire from="(410,490)" to="(410,500)"/>
<wire from="(410,540)" to="(410,550)"/>
<wire from="(410,620)" to="(410,630)"/>
<wire from="(410,670)" to="(410,680)"/>
<wire from="(420,830)" to="(440,830)"/>
<wire from="(470,370)" to="(520,370)"/>
<wire from="(470,500)" to="(520,500)"/>
<wire from="(470,630)" to="(530,630)"/>
<wire from="(500,200)" to="(530,200)"/>
<wire from="(520,370)" to="(520,400)"/>
<wire from="(520,370)" to="(530,370)"/>
<wire from="(520,500)" to="(520,530)"/>
<wire from="(520,500)" to="(530,500)"/>
<wire from="(530,940)" to="(570,940)"/>
<wire from="(530,970)" to="(540,970)"/>
<wire from="(540,380)" to="(540,390)"/>
<wire from="(540,390)" to="(620,390)"/>
<wire from="(540,510)" to="(540,520)"/>
<wire from="(540,520)" to="(620,520)"/>
<wire from="(540,640)" to="(540,660)"/>
<wire from="(540,790)" to="(570,790)"/>
<wire from="(540,960)" to="(540,970)"/>
<wire from="(540,960)" to="(570,960)"/>
<wire from="(550,370)" to="(560,370)"/>
<wire from="(550,500)" to="(560,500)"/>
<wire from="(550,630)" to="(560,630)"/>
<wire from="(580,760)" to="(590,760)"/>
<wire from="(580,820)" to="(610,820)"/>
<wire from="(580,870)" to="(650,870)"/>
<wire from="(590,1030)" to="(620,1030)"/>
<wire from="(590,760)" to="(590,770)"/>
<wire from="(590,770)" to="(600,770)"/>
<wire from="(590,790)" to="(600,790)"/>
<wire from="(600,840)" to="(610,840)"/>
<wire from="(600,950)" to="(640,950)"/>
<wire from="(610,1090)" to="(630,1090)"/>
<wire from="(610,820)" to="(610,840)"/>
<wire from="(610,840)" to="(650,840)"/>
<wire from="(620,1030)" to="(620,1050)"/>
<wire from="(620,390)" to="(620,400)"/>
<wire from="(620,520)" to="(620,530)"/>
<wire from="(630,1080)" to="(630,1090)"/>
<wire from="(630,780)" to="(650,780)"/>
<wire from="(640,790)" to="(640,950)"/>
<wire from="(640,790)" to="(650,790)"/>
<wire from="(640,950)" to="(640,1050)"/>
<wire from="(740,200)" to="(770,200)"/>
<wire from="(760,1010)" to="(780,1010)"/>
<wire from="(760,980)" to="(770,980)"/>
<wire from="(770,970)" to="(770,980)"/>
<wire from="(770,970)" to="(800,970)"/>
<wire from="(780,990)" to="(780,1010)"/>
<wire from="(780,990)" to="(800,990)"/>
<wire from="(830,810)" to="(850,810)"/>
<wire from="(830,870)" to="(870,870)"/>
<wire from="(830,980)" to="(840,980)"/>
<wire from="(840,1010)" to="(870,1010)"/>
```

```
    <wire from="(840,950)" to="(870,950)"/>
    <wire from="(840,970)" to="(840,980)"/>
    <wire from="(840,970)" to="(870,970)"/>
    <wire from="(850,720)" to="(850,810)"/>
    <wire from="(870,820)" to="(870,870)"/>
    <wire from="(870,820)" to="(910,820)"/>
    <wire from="(870,990)" to="(870,1010)"/>
    <wire from="(900,830)" to="(900,850)"/>
    <wire from="(900,830)" to="(910,830)"/>
    <wire from="(900,850)" to="(930,850)"/>
    <wire from="(930,590)" to="(1000,590)"/>
    <wire from="(930,760)" to="(930,810)"/>
    <wire from="(930,760)" to="(940,760)"/>
    <wire from="(930,850)" to="(930,950)"/>
    <wire from="(940,750)" to="(940,760)"/>
    <wire from="(940,760)" to="(960,760)"/>
    <wire from="(960,570)" to="(1000,570)"/>
    <wire from="(960,610)" to="(1000,610)"/>
    <wire from="(960,650)" to="(1000,650)"/>
    <wire from="(990,490)" to="(1000,490)"/>
    <wire from="(990,520)" to="(1000,520)"/>
    <wire from="(990,550)" to="(1000,550)"/>
    <wire from="(990,630)" to="(1000,630)"/>
  </circuit>
  <circuit name="ALU">
    <a name="appearance" val="logisim_evolution"/>
    <a name="circuit" val="ALU"/>
    <a name="circuitnamedboxfixedsize" val="true"/>
    <a name="simulationFrequency" val="64.0"/>
    <appear>
      <rect fill="none" height="80" stroke="#000000" width="220" x="50" y="50"/>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="66" y="59">A</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="65" y="78">B</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="14" font-weight="bold"
text-anchor="middle" x="164" y="75">ALU</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="72" y="115">EO</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="110" y="115">SU</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="141" y="116">MU</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="170" y="115">SL</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="199" y="116">SR</text>
      <text dominant-baseline="central" font-family="SansSerif" font-size="12" text-anchor="middle"
x="250" y="59">Out</text>
      <circ-anchor facing="east" x="270" y="60"/>
      <circ-port dir="in" pin="140,430" x="70" y="130"/>
      <circ-port dir="in" pin="150,140" x="50" y="60"/>
      <circ-port dir="in" pin="150,270" x="50" y="80"/>
      <circ-port dir="in" pin="420,510" x="270" y="70"/>
      <circ-port dir="in" pin="460,510" x="140" y="130"/>
      <circ-port dir="in" pin="510,510" x="170" y="130"/>
      <circ-port dir="in" pin="560,510" x="200" y="130"/>
      <circ-port dir="in" pin="650,510" x="80" y="130"/>
      <circ-port dir="out" pin="820,550" x="210" y="130"/>
      <circ-port dir="out" pin="850,260" x="270" y="60"/>
    </appear>
    <comp lib="0" loc="(140,430)" name="Pin">
      <a name="appearance" val="NewPins"/>
      <a name="facing" val="north"/>
```

```
  <a name="label" val="EO"/>
 </comp>
 <comp lib="0" loc="(140,430)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="EO"/>
 </comp>
 <comp lib="0" loc="(140,530)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="SL"/>
 </comp>
 <comp lib="0" loc="(140,570)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="SR"/>
 </comp>
 <comp lib="0" loc="(150,140)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="label" val="A"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="0" loc="(150,270)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="label" val="B"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="0" loc="(180,160)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EO"/>
 </comp>
 <comp lib="0" loc="(180,190)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
 </comp>
 <comp lib="0" loc="(180,290)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EO"/>
 </comp>
 <comp lib="0" loc="(180,320)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="CLK"/>
 </comp>
 <comp lib="0" loc="(250,550)" name="Tunnel">
  <a name="label" val="Shift"/>
 </comp>
 <comp lib="0" loc="(330,140)" name="Tunnel">
  <a name="label" val="A"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="0" loc="(330,270)" name="Tunnel">
  <a name="label" val="B"/>
  <a name="width" val="16"/>
 </comp>
 <comp lib="0" loc="(400,330)" name="Splitter">
  <a name="appear" val="right"/>
  <a name="bit1" val="0"/>
  <a name="bit10" val="1"/>
  <a name="bit11" val="1"/>
  <a name="bit12" val="1"/>
  <a name="bit13" val="1"/>
  <a name="bit14" val="1"/>
  <a name="bit15" val="1"/>
  <a name="bit2" val="0"/>
  <a name="bit3" val="0"/>
  <a name="bit4" val="1"/>
  <a name="bit5" val="1"/>
```

```
  <a name="bit6" val="1"/>
  <a name="bit7" val="1"/>
  <a name="bit8" val="1"/>
  <a name="bit9" val="1"/>
  <a name="incoming" val="16"/>
</comp>
<comp lib="0" loc="(400,330)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="B"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(420,510)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
  <a name="label" val="Sub"/>
</comp>
<comp lib="0" loc="(420,510)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="SU"/>
</comp>
<comp lib="0" loc="(450,270)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="B"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(460,130)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="A"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(460,170)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="B"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(460,510)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
  <a name="label" val="Mult"/>
</comp>
<comp lib="0" loc="(460,510)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="MU"/>
</comp>
<comp lib="0" loc="(470,250)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="A"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(470,300)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="A"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(470,350)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="A"/>
  <a name="width" val="16"/>
</comp>
<comp lib="0" loc="(510,510)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
  <a name="label" val="BShL"/>
</comp>
```

```
<comp lib="0" loc="(510,510)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="SL"/>
</comp>
<comp lib="0" loc="(560,510)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
  <a name="label" val="BShR"/>
</comp>
<comp lib="0" loc="(560,510)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="SR"/>
</comp>
<comp lib="0" loc="(590,120)" name="Tunnel">
  <a name="label" val="SU"/>
</comp>
<comp lib="0" loc="(590,330)" name="Tunnel">
  <a name="label" val="SL"/>
</comp>
<comp lib="0" loc="(590,380)" name="Tunnel">
  <a name="label" val="SR"/>
</comp>
<comp lib="0" loc="(650,510)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="north"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(650,510)" name="Tunnel">
  <a name="facing" val="south"/>
  <a name="label" val="CLK"/>
</comp>
<comp lib="0" loc="(660,360)" name="Tunnel">
  <a name="facing" val="north"/>
  <a name="label" val="Shift"/>
</comp>
<comp lib="0" loc="(680,310)" name="Splitter">
  <a name="appear" val="right"/>
  <a name="facing" val="south"/>
</comp>
<comp lib="0" loc="(680,350)" name="Tunnel">
  <a name="label" val="MU"/>
</comp>
<comp lib="0" loc="(720,330)" name="Splitter">
  <a name="facing" val="south"/>
  <a name="fanout" val="16"/>
  <a name="incoming" val="16"/>
</comp>
<comp lib="0" loc="(770,220)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="south"/>
</comp>
<comp lib="0" loc="(780,530)" name="Tunnel">
  <a name="facing" val="east"/>
  <a name="label" val="EO"/>
</comp>
<comp lib="0" loc="(800,170)" name="Probe">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="south"/>
  <a name="radix" val="10signed"/>
</comp>
<comp lib="0" loc="(820,550)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="west"/>
  <a name="label" val="Zero_Flag"/>
```

```
  <a name="output" val="true"/>
</comp>
<comp lib="0" loc="(850,260)" name="Pin">
  <a name="appearance" val="NewPins"/>
  <a name="facing" val="west"/>
  <a name="label" val="Result_Out"/>
  <a name="output" val="true"/>
  <a name="radix" val="16"/>
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(220,550)" name="OR Gate"/>
<comp lib="1" loc="(500,210)" name="Controlled Buffer">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(510,170)" name="Controlled Inverter">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(510,220)" name="NOT Gate">
  <a name="facing" val="west"/>
</comp>
<comp lib="1" loc="(590,310)" name="Controlled Buffer">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(590,360)" name="Controlled Buffer">
  <a name="width" val="16"/>
</comp>
<comp lib="1" loc="(810,470)" name="OR Gate">
  <a name="facing" val="south"/>
  <a name="inputs" val="16"/>
</comp>
<comp lib="1" loc="(810,510)" name="NOT Gate">
  <a name="facing" val="south"/>
</comp>
<comp lib="1" loc="(810,540)" name="Controlled Buffer">
  <a name="facing" val="south"/>
</comp>
<comp lib="2" loc="(700,260)" name="Multiplexer">
  <a name="select" val="2"/>
  <a name="width" val="16"/>
</comp>
<comp lib="3" loc="(550,260)" name="Multiplier">
  <a name="width" val="16"/>
</comp>
<comp lib="3" loc="(550,310)" name="Shifter">
  <a name="width" val="16"/>
</comp>
<comp lib="3" loc="(550,360)" name="Shifter">
  <a name="shift" val="lr"/>
  <a name="width" val="16"/>
</comp>
<comp lib="3" loc="(570,160)" name="Adder">
  <a name="width" val="16"/>
</comp>
<comp lib="4" loc="(230,110)" name="Register">
  <a name="appearance" val="logisim_evolution"/>
  <a name="width" val="16"/>
</comp>
<comp lib="4" loc="(230,240)" name="Register">
  <a name="appearance" val="logisim_evolution"/>
  <a name="width" val="16"/>
</comp>
<comp lib="8" loc="(490,590)" name="Text">
  <a name="text" val="Operation Inputs"/>
</comp>
```

```
<comp lib="8" loc="(510,80)" name="Text">
  <a name="text" val="If EO enable a specified operation, or addition if not specified, using A and B"/>
</comp>
<comp lib="8" loc="(520,425)" name="Text">
  <a name="text" val="First 4 bits of B to decide amount of shifts"/>
</comp>
<comp lib="8" loc="(820,590)" name="Text">
  <a name="text" val="Check each bit if any are 1 and invert result"/>
</comp>
<comp lib="8" loc="(85,100)" name="Text">
  <a name="text" val="Input A"/>
</comp>
<comp lib="8" loc="(90,235)" name="Text">
  <a name="text" val="Input B"/>
</comp>
<wire from="(140,530)" to="(170,530)"/>
<wire from="(140,570)" to="(170,570)"/>
<wire from="(150,140)" to="(230,140)"/>
<wire from="(150,270)" to="(230,270)"/>
<wire from="(180,160)" to="(200,160)"/>
<wire from="(180,290)" to="(200,290)"/>
<wire from="(200,160)" to="(200,180)"/>
<wire from="(200,160)" to="(230,160)"/>
<wire from="(200,180)" to="(230,180)"/>
<wire from="(200,290)" to="(200,310)"/>
<wire from="(200,290)" to="(230,290)"/>
<wire from="(200,310)" to="(230,310)"/>
<wire from="(220,550)" to="(250,550)"/>
<wire from="(290,140)" to="(330,140)"/>
<wire from="(290,270)" to="(330,270)"/>
<wire from="(420,340)" to="(440,340)"/>
<wire from="(440,320)" to="(440,340)"/>
<wire from="(440,320)" to="(510,320)"/>
<wire from="(440,340)" to="(440,370)"/>
<wire from="(440,370)" to="(510,370)"/>
<wire from="(450,270)" to="(510,270)"/>
<wire from="(460,130)" to="(520,130)"/>
<wire from="(460,170)" to="(470,170)"/>
<wire from="(470,170)" to="(470,210)"/>
<wire from="(470,170)" to="(480,170)"/>
<wire from="(470,210)" to="(480,210)"/>
<wire from="(470,250)" to="(510,250)"/>
<wire from="(470,300)" to="(510,300)"/>
<wire from="(470,350)" to="(510,350)"/>
<wire from="(490,180)" to="(490,200)"/>
<wire from="(490,200)" to="(560,200)"/>
<wire from="(490,220)" to="(510,220)"/>
<wire from="(500,210)" to="(520,210)"/>
<wire from="(510,170)" to="(520,170)"/>
<wire from="(520,130)" to="(520,150)"/>
<wire from="(520,150)" to="(530,150)"/>
<wire from="(520,170)" to="(520,210)"/>
<wire from="(520,170)" to="(530,170)"/>
<wire from="(540,220)" to="(560,220)"/>
<wire from="(550,120)" to="(550,140)"/>
<wire from="(550,120)" to="(580,120)"/>
<wire from="(550,260)" to="(560,260)"/>
<wire from="(550,310)" to="(570,310)"/>
<wire from="(550,360)" to="(570,360)"/>
<wire from="(560,200)" to="(560,220)"/>
<wire from="(560,200)" to="(580,200)"/>
<wire from="(560,250)" to="(560,260)"/>
<wire from="(560,250)" to="(660,250)"/>
<wire from="(570,160)" to="(640,160)"/>
```

```
    <wire from="(580,120)" to="(580,200)"/>
    <wire from="(580,120)" to="(590,120)"/>
    <wire from="(580,320)" to="(580,330)"/>
    <wire from="(580,330)" to="(590,330)"/>
    <wire from="(580,370)" to="(580,380)"/>
    <wire from="(580,380)" to="(590,380)"/>
    <wire from="(590,310)" to="(630,310)"/>
    <wire from="(590,360)" to="(630,360)"/>
    <wire from="(630,270)" to="(630,310)"/>
    <wire from="(630,270)" to="(660,270)"/>
    <wire from="(630,310)" to="(630,360)"/>
    <wire from="(640,160)" to="(640,240)"/>
    <wire from="(640,240)" to="(660,240)"/>
    <wire from="(660,330)" to="(660,360)"/>
    <wire from="(670,330)" to="(670,350)"/>
    <wire from="(670,350)" to="(680,350)"/>
    <wire from="(680,280)" to="(680,310)"/>
    <wire from="(700,260)" to="(720,260)"/>
    <wire from="(720,260)" to="(720,330)"/>
    <wire from="(720,260)" to="(770,260)"/>
    <wire from="(730,350)" to="(730,420)"/>
    <wire from="(740,350)" to="(740,420)"/>
    <wire from="(750,350)" to="(750,420)"/>
    <wire from="(760,350)" to="(760,420)"/>
    <wire from="(770,220)" to="(770,260)"/>
    <wire from="(770,260)" to="(840,260)"/>
    <wire from="(770,350)" to="(770,420)"/>
    <wire from="(780,350)" to="(780,420)"/>
    <wire from="(780,530)" to="(800,530)"/>
    <wire from="(790,350)" to="(790,420)"/>
    <wire from="(800,170)" to="(840,170)"/>
    <wire from="(800,350)" to="(800,420)"/>
    <wire from="(810,350)" to="(810,420)"/>
    <wire from="(810,420)" to="(820,420)"/>
    <wire from="(810,470)" to="(810,480)"/>
    <wire from="(810,510)" to="(810,520)"/>
    <wire from="(810,540)" to="(810,550)"/>
    <wire from="(810,550)" to="(820,550)"/>
    <wire from="(820,350)" to="(820,410)"/>
    <wire from="(820,410)" to="(830,410)"/>
    <wire from="(830,350)" to="(830,400)"/>
    <wire from="(830,400)" to="(840,400)"/>
    <wire from="(830,410)" to="(830,420)"/>
    <wire from="(840,170)" to="(840,260)"/>
    <wire from="(840,260)" to="(850,260)"/>
    <wire from="(840,350)" to="(840,390)"/>
    <wire from="(840,390)" to="(850,390)"/>
    <wire from="(840,400)" to="(840,420)"/>
    <wire from="(850,350)" to="(850,380)"/>
    <wire from="(850,380)" to="(860,380)"/>
    <wire from="(850,390)" to="(850,420)"/>
    <wire from="(860,350)" to="(860,370)"/>
    <wire from="(860,370)" to="(870,370)"/>
    <wire from="(860,380)" to="(860,420)"/>
    <wire from="(870,350)" to="(870,360)"/>
    <wire from="(870,360)" to="(880,360)"/>
    <wire from="(870,370)" to="(870,420)"/>
    <wire from="(880,350)" to="(890,350)"/>
    <wire from="(880,360)" to="(880,420)"/>
    <wire from="(890,350)" to="(890,420)"/>
  </circuit>
</project>
```