# GateCPU

First edition

*Prepared by*

**Mugilan Baskaran**

Chennai, India

*January 10, 2024*

# Motivation

I've always wondered how computers *think* at the very fundamental level. I understand that not everyone would agree with the use of the word *think* in this context, but I'm at a loss for a better word; excuse me. To differentiate, I will italicize it whenever I can.

Physically, almost all parts of the traditional computers are static. There are exceptions, for example, the actuators of Hard Disk Drives (HDDs), Compact Disks (CDs), and several components used for mitigating excessive thermal energy usually have moving parts. Of course, many Input/Output (I/O) devices generally aren't static either.

But the Central Processing Unit (CPU) does not have any dynamically moving components, at least, not as intended. To put it in a rather oversimplified manner, the only parameters that change within a CPU Die over time are Voltage levels. How can such a thing control machines and react to external *stimuli* in a very complex way? How can a **preordained** circuit really seem to *think*?

I've also wondered quite a bit about how computers work in general. Questions like "Why do you need a clock?", "How are Logic gates even relevant?", "What happens under the hood when you execute a program?" were also puzzling. I discovered the answers to these questions by myself, when trying to answer the grand question.

There is a popular survival game called *Minecraft*. Those of you who are familiar with it might know about Redstones. I was absolutely blown away by how people made computers *within* Minecraft. This was my initial inspiration. This is why I've come this far in this topic. My initial efforts were in making a Redstone "Adder". When I tried to build increasingly complex devices, I realized that the fundamental *flaw* with Redstone - needing Repeaters for every 8 blocks of signal, becomes too cumbersome to work with.

This made me switch to Digital Circuit simulators. There is a well-known Open source simulator called *Logisim*. However, this was discontinued in 2014. A fork of it called *Logisim-evolution* exists currently, which was last updated in 2022. There is

another piece of software called *Digital*, which was last updated in 2023. It was built from the ground-up, inspired from logisim. By no means did I compare *Logisim-evolution* and *Digital* in a comprehensive way, I just chose *Digital* and stuck with it. I greatly appreciate Helmut Neemann for developing *Digital* and for keeping it open-source. I've chosen not to embed components into packages while building this because I want to be able to view everything all at once, and the full extent of it. I like it this way. I understand that makes readability considerably more difficult, but that is also why I've written this document in detail, to compensate for that.

# A note to the reader

I wrote this with a rather broad audience in mind. Several friends from varying backgrounds requested an explanation of GateCPU, so I thought it would be better to compile everything into a single document.

My understanding of contemporary computer hardware on a circuit level is almost nil. I do not have formal or structured learning in the advanced topics of this subject. The architecture I've arrived at may or may not be typical. I've had some amount of very scattered learning in this subject, and have come across Assembly language a few times. I've glanced at it, but never really wrote code in it. The terminology I've adopted is relatively standard, though.

I tried to look into the 8085 instruction set (A common microprocessor students generally start learning with) to get an inspiration, but it has more than 200 instructions! Something of that scale would take me months to design. I've only spent about 50 hours designing GateCPU and another 30-40 to fix errors/test programs. Maybe another 30 hours to write the first edition of this document. So, I've really gone with what I thought 'would work'. Considering the exploratory nature of this project (as evident in the previous section), I thought this was sufficient.

I've tried to structure this document in a bottom-up manner. I introduce the most basic and simple elements first, and then build on top of it. If you want to run these, visit the linked repository, download the relevant circuit files and run them in the latest version of *Digital*, which can be found **here**.

If you have a knowledge of gates, latches and digital circuits you can skip over most of the theory. Just read the section on simulator constraints and move on to section 6.

# Contents

# 1 Technical Specifications of GateCPU:

- **16-bit Data bus:** Each word consists of 16 bits, so this could be called as a 16-bit CPU.
- **4-bit Opcode bus:** Each operation is coded by a 4 bit binary number, so there are 15 possible operations this CPU can do. However, I only have definitions for 11 instructions as of now.
- **6-bit Address bus:** Each address is coded by a 6-bit binary number, so there are 63 possible addressable registers.
  - **Working memory:** There are 24 words' worth of Working memory available.
  - **Read-Only memory (ROM):** There are 16 words' worth of ROM available.
- **6-bit Program counter:** Each program can only have 63 lines of operations.
- **Clock:** There is a customizable clock, but the programs have only been tested at 1 Hz clock rate, program counter at 0.5 Hz.

As of 4 January, there are about 1,302 AND Gates*, 1,384 Drivers, 487 J-K Flip-flops, 83 R-S Flip-flops, 48 XOR Gates, 25 XNOR Gates and 1 NAND Gate in the circuit. It would take an estimated 10k+ transistors to make this circuit.

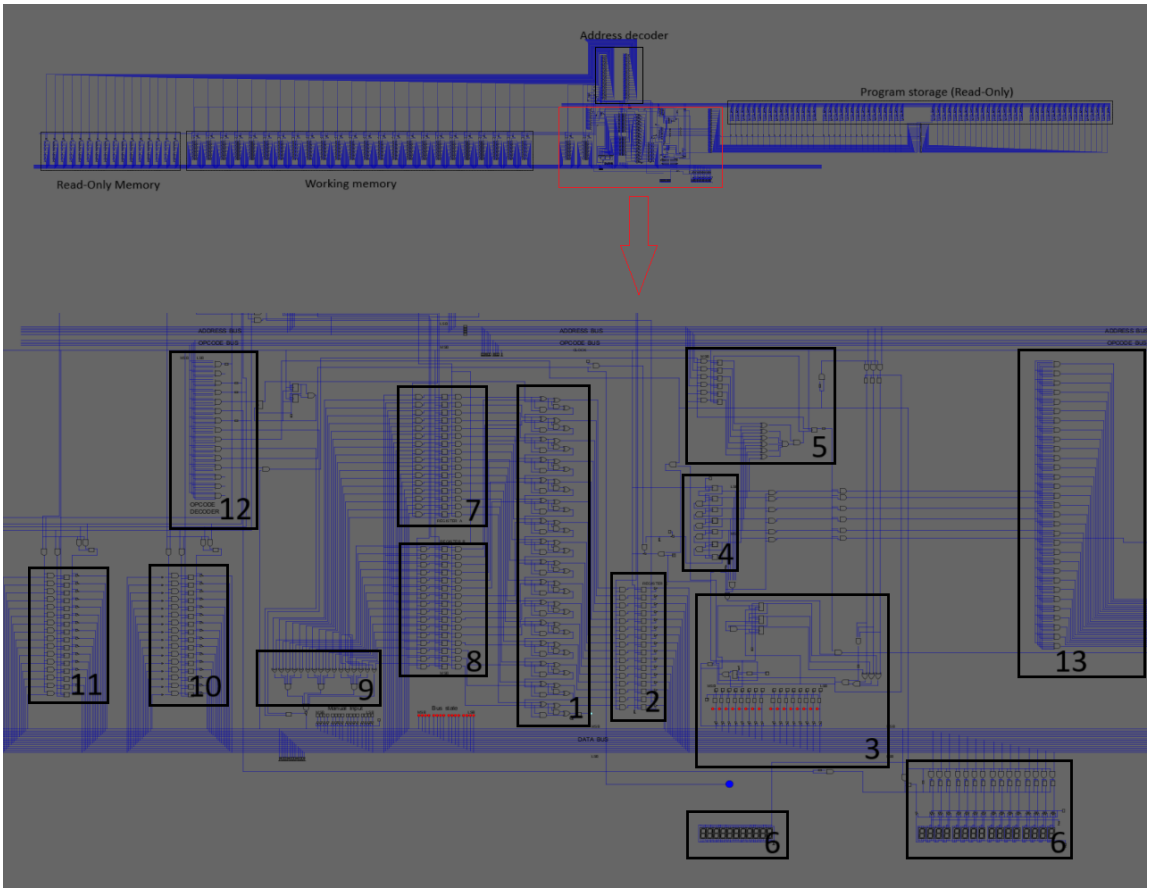


Figure 1: GateCPU Schematic

## 2 Structure

The parts are as follows:

1. 16-bit adder
2. Register C
3. User input handler
4. Program counter
5. GOTO handler

6. Display/output
7. Register A
8. Register B
9. Comparator
10. Register 4/Register NOT
11. Register 5/Register bitreverse
12. Opcode decoder
13. Program counter decoder

# 3 Introduction

This is a very basic 16-bit processor. It acts only on 16-bit data at all times and manipulates them according to the program. The program has to be written in binary. Documentation is in section 7.
The relevant Git repository is located here. The *.dig file is what you need to download to run the CPU on your end.
Alternatively, If you just want to view the circuit, it is here. Please zoom out and use the scroll bars to look around. As this is rendering an SVG, detail will not be lost when zoomed in. This can be viewed on many platforms, including Android. The current version holds the add or subtract program loaded (Program details are at section 8.3).

# 4 Theory

This section contains the relevant concepts that you need to learn to understand this circuit. Most of this section is about basic boolean ideas, except for section 4.5.

## 4.1 Digital Circuits

What are digital circuits? First and foremost, what is a circuit? In electronics, a circuit is a network of connections connecting several electronic components in a specific manner, and behaves in a predictable fashion. A **Digital** circuit is one where the signal or power levels (can be used interchangeably in this context) are either HIGH, LOW or Not defined (High-impedance state). That means that, any piece of wire in the circuit will be in one of these three states, and in our context, the 'Not defined' state is not desirable.

- **HIGH:** The wire is ON. There is power/signal flowing through, it will power any component connected to it directly.

- **LOW:** The wire is OFF. There is NO power flowing through. Connected components will not receive a signal.
- **High-Z/Not defined:** When a wire is in High-Z state, then at present, that wire does not have any impact on the circuit. It is basically the equivalent of not having a wire. Any devices that function on only the two states (HIGH and LOW) will not work when connected to this wire, unless this state of this wire is changed.

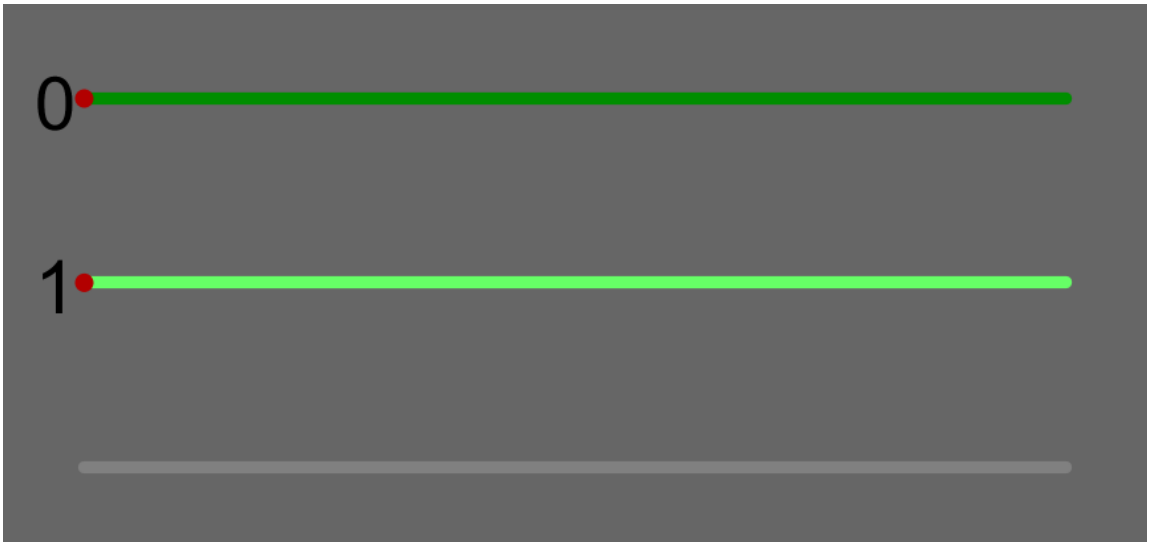


Figure 2: All possible states of a wire in Digital. Note that the 1 and 0 on the left are signal sources. From top to bottom, the wires are in LOW, HIGH and high-Z states respectively.

## 4.2 Binary

Please note that the term "HIGH" is equivalent to "1" and "ON". The term "LOW" is equivalent to "0" and "OFF". These equivalent terms will be used interchangeably throughout this document.

1 and 0 as given above is what essentially makes the individual digits of a binary number. A Bit stands for **B**inary dig**it**. In a conventional base-10 system, each digit can accommodate 10 values ranging from 0 to 9. In binary, only 0 and 1 can be accommodated. You might ask, is this enough to describe all whole numbers? Yes! Table 1 shows an example of binary numbers and their base-10 equivalents.

| Binary representation | Base-10 |
|---|---|
| 00 | 0 |
| 01 | 1 |
| 10 | 2 |
| 11 | 3 |

Table 1: Binary representation

As more binary digits are added, larger base-10 numbers can be counted. Each digit carries the weight of $base^{digit-index}$. For example, take a binary number 0101: The Least Significant Bit (LSB) is the 1 on the rightmost. It is the equivalent of the "unit digit". This digit carres a weight of $2^0$. The digit that has the next higher significance is the 0 to the left of it, which is the equivalent of the "tens digit". It has a weight of $2^1$. You might ask, what do these weights do? Well, you can use these weights to convert the number to base-10. (More technically, you use the weights to actually express the number in terms of its digits). How do I do that? Simple - Multiply the digit by its weight. So 0101 would be $(2^0 \times 1) + (2^1 \times 0) + (2^2 \times 1) + (2^3 \times 0) = 1 + 4 = 5$

To represent *negative* integers, there are two methods to do it:

### 4.2.1 Ones' complement

The ones' complement form is easy: The leading bit (MSB or Most significant bit) is used as a *sign bit*. A sign bit only denotes the sign of the number, and has no value. It is 1 if the number is negative. Take a number 0101. The leading bit is 0, which is

a sign bit. Flip all the zeroes to ones and ones to zeroes. The number is now 1010, which represents a minus 5. To switch it back, do this:

1. Is the Sign Bit (MSB) 0? If so, then this is a positive number, you just need to evaluate the number using its weights.

2. If not, then flip all the bits - 1s to 0s and vice-versa.

3. Now evaluate the number using the weights, and add a negative sign to it. Voila!

### 4.2.2 Twos' complement

In twos' complement form, you take the negative number, convert it into ones' complement form, and add one to it. An example: 0101 - minus five. Ones' complement is 1010. Add one: This becomes now 1010 + 1 = 1011. To convert back, apply the same steps. Flip all the bits and add one. 1011 becomes 0100 becomes 0101.
But why twos' complement form? This is used because to add two negative numbers, it follows the same algorithm of addition (explained later), regardless of whether the number is negative or not. Such is not true for ones' complement, where special logic is required to ensure negative numbers are added correctly.

## 4.3 Logic Gates

Logic gates are small digital devices that act upon binary data and produce an output binary value in a predictable manner. It may have any number of inputs, but it is most commonly 1 or 2, and one output. Different logic gates produce different outputs when acted upon by the same inputs.

## 4.4 Truth Tables

A truth table is an exhaustive list of answers to "What output will I get if I provide this specific input?" for varying inputs, to a specific logic gate. It basically *describes*

the logic gate. When the inputs are given, you can lookup this table and get to know the output. If there are 2 inputs and 1 output, then the truth table will have 3 columns - one for each input, and one for the output. Table 2 is a template for a truth table with 2 inputs and 1 output. The specific values for $?_1, ?_2, ?_3, ?_4$ uniquely determine a logic gate.

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | $?_1$ |
| 0 | 1 | $?_2$ |
| 1 | 0 | $?_3$ |
| 1 | 1 | $?_4$ |

Table 2: A template for a truth table

## 4.5 Simulator constraints

The simulator has several unavoidable constraints, that are to be kept in mind when making a circuit. In particular, these two are very notorious.

### 4.5.1 Gate delay

When a signal enters a gate, there is a fixed amount of time which is taken by the software to process the output from the gate. It is to note that, to quote, this delay is the same regardless of the complexity of the gate. When there are a large number of components involved, gate delays may reach a significant fraction of the clock pulse time. This occurs in GateCPU. When this happens, important **timing considerations** have to be taken into account when designing components.
If there are two signals that arrive at different times, it may produce unexpected behaviour. One common example is part of the previous instruction arriving late enough to influence the execution of the present instruction, thereby causing an error. Gate delay is, in my opinion, the primary design flaw of GateCPU. When building, this was not considered at all. It was during testing that this came into

picture and signals have been synchronized to a good extent. However, I cannot guarantee, as I've only tested three programs (the individual components have been tested individually of course), of which only one uses all the available instructions.

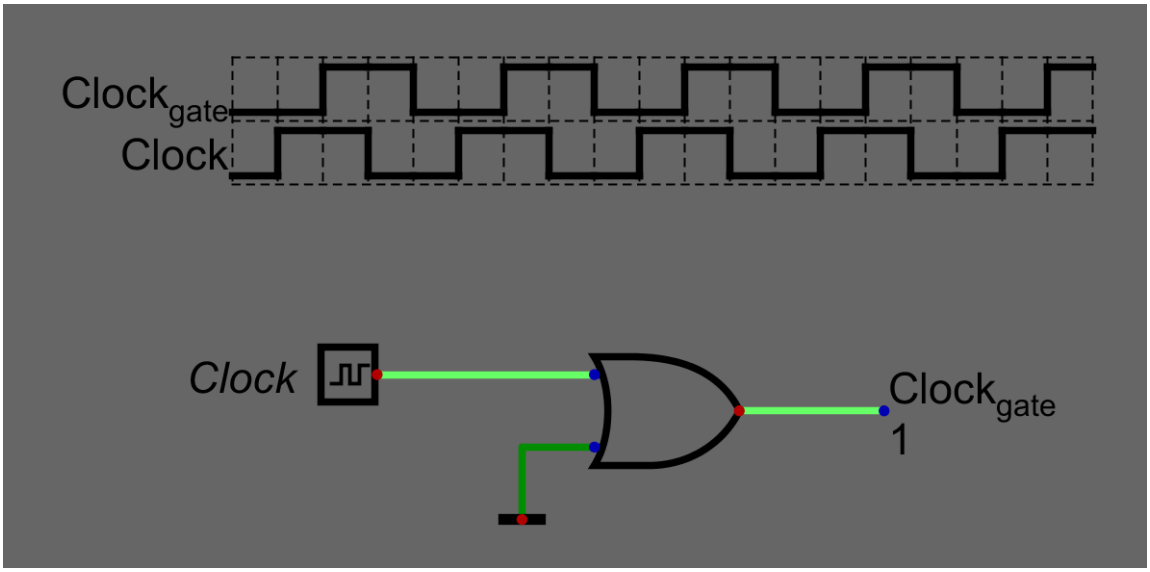


Figure 3: The clock signal is delayed by 1 unit of gate delay after it passes through a gate. In the graph, the x axis is time, increasing toward the right. Please note that gate delay is usually much smaller than a single clock cycle, especially at low frequencies.

### 4.5.2 Multiple inputs

When there are two opposite signal sources feeding into a piece of wire at the same time, the simulation crashes, as it is not possible to decide which signal source to use for the wire's state. To work around this, one can use a driver, which isolates the two parts of the circuit until it gets a signal.

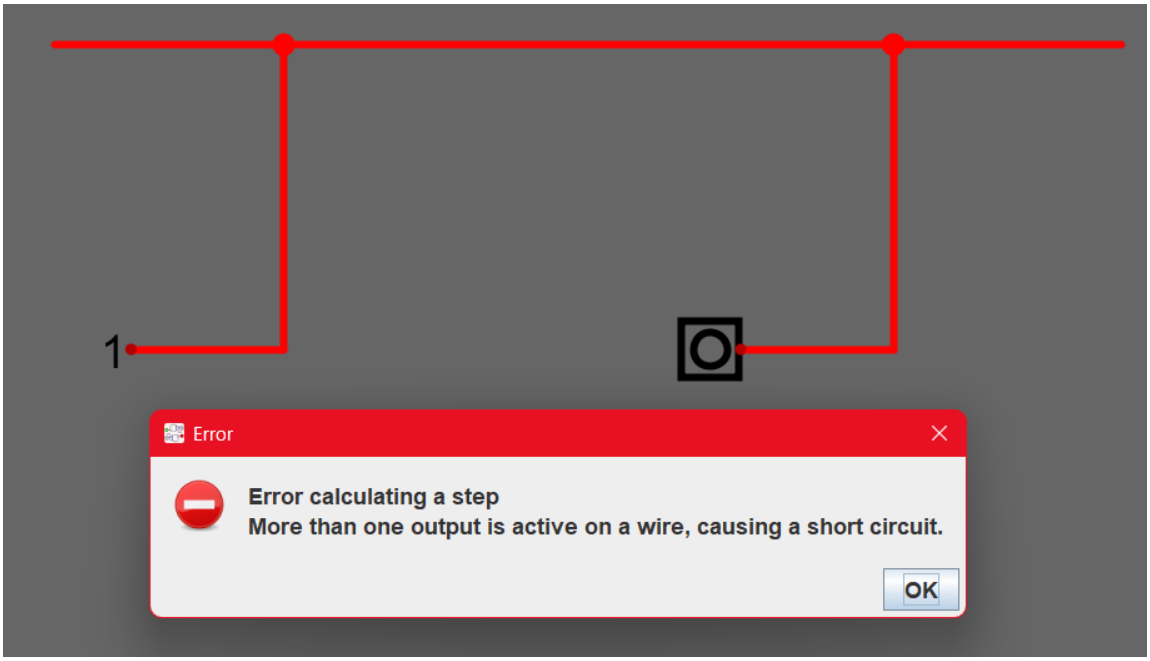


Figure 4: The Input device on the right was OFF by default. On starting the simulation, two signal sources providing different states to the wire caused this error. The simulator does NOT add the signal state implicitly to 1.

# 5 Components

The following sections contain detailed descriptions/explanations of the components used in the circuit.

## 5.1 Bus

A bus is just a set of wires that run in parallel. It is used for communicating data between two parts of the Processing unit. Think of it like individual lanes in a highway connecting several cities. In a bus, many 'lanes' are used because the data that is being sent is longer than 1-bit. If a 4-bit message is to be sent at a time, then you need 4 wires to send it, thereby, making a 4-bit bus. In GateCPU, three buses exist:

- **Opcode bus:** This bus carries information about what OPCODE is to be executed at that instant. This is 4-bits wide.
- **Address bus:** This bus carries information about what part (register, to be precise) of the Processing unit should be activated. This is 6-bits wide.
- **Data bus:** This bus carries the binary datum that is being worked on currently. This is 16-bits wide.

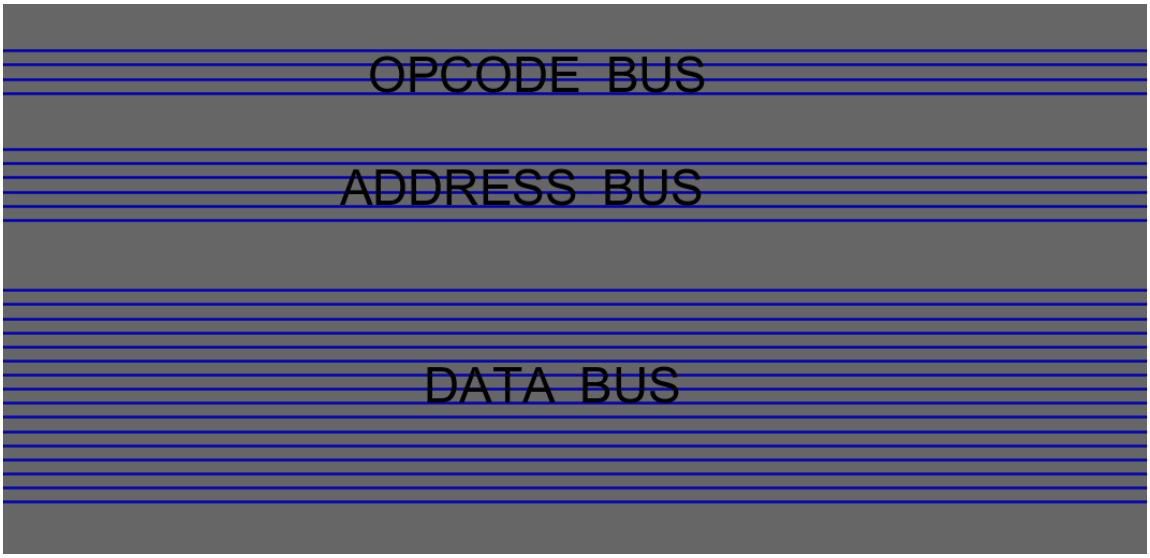


Figure 5: The three buses that are present in GateCPU.

## 5.2 Pull-up/Pull-down Resistors

When the state of a wire is Not defined or High-Z, then using these resistors one can set a value to it. When a Pull-**up** resistor is used, then that resistor will ensure the signal/state of the wire is HIGH when there is nothing else acting on the wire otherwise, i.e., when it is in a high-z state. Similarly, a Pull-**Down** resistor ensures the signal is 0 when the wire is in such a state.
In cases where there is already a resistor and there is a signal incoming, the resistor

will not do anything in this simulation, as when there is a signal the wire will no longer be in an undefined state.

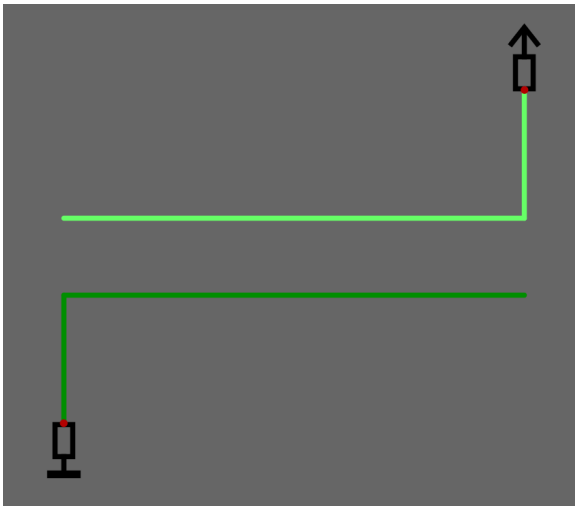

Figure 6: Pull-up and Pull-down resistors

## 5.3 Common Logic gates

These are the common logic gates that are used in digital computing. Note that more logic gates are possible, however, they are not very useful. It is also interesting to note that any of these gates can be constructed by a sufficient number of either NAND or NOR gates when wired up correctly. There are no other gates with this property of universality.

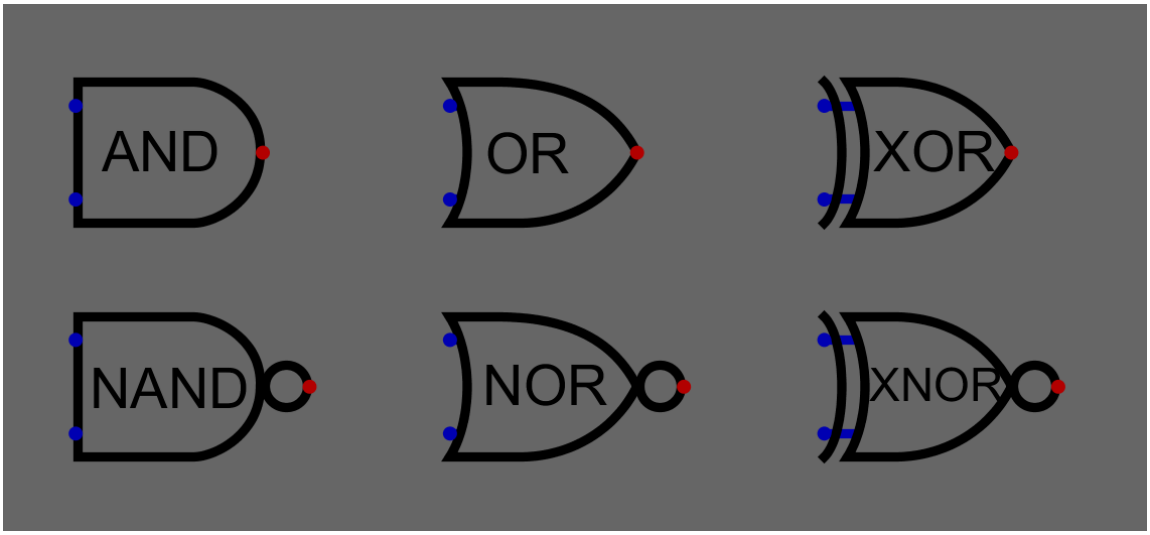


Figure 7: Six of the seven common logic gates used in digital circuits. These are unconnected. According to Digital, these are IEEE 91-1984 shapes.

### 5.3.1 The NOT Gate

The NOT gate has one input and one output. It inverts the value of the input and sends it to the output. This is the equivalent of asking "What is **NOT** A?". By observing the truth table, it is clear that one application of this is in finding the Ones' complement form of a number.

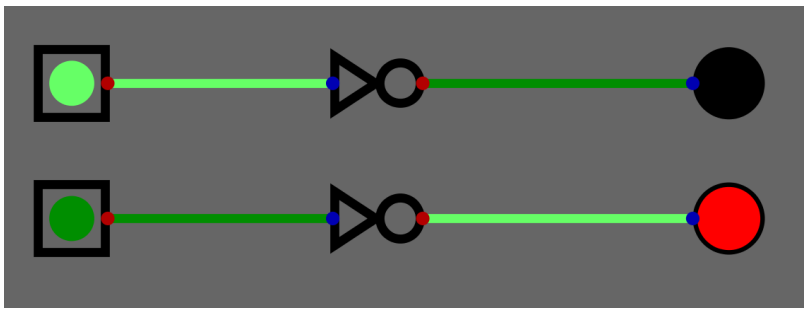

Figure 8: The NOT gate.

| Input | Output |
|---|---|
| 0 | 1 |
| 1 | 0 |

Table 3: NOT Truth table

### 5.3.2 The OR Gate

The OR gate has two inputs and one output. The output is 0 only if **both** the inputs are 0. The output is 1 in all the other cases. This gate is the equivalent of asking "Is either A **OR** B On?"

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 4: OR Truth table

### 5.3.3 The NOR Gate

The NOR gate has two inputs and one output. The output is 1 only if **both** the inputs are 0. The output is 0 in all the other cases. This is the same as connecting a NOT gate to an OR gate's output, and finding the final result. It is the same as asking "Is neither A **NOR** B On?"

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Table 5: NOR Truth table

### 5.3.4 The AND Gate

The AND gate has two inputs and one output. The output is 1 only if **both** the inputs are 1. The output is 0 in all the other cases. This is the equivalent of asking "Are A **AND** B both On?". This was the most used gate in the circuit, by a very large margin. In the circuit, multiple-input AND gates are used - upto 5. n-input AND gates output 1 IF and ONLY IF ALL the inputs are high.

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Table 6: AND Truth table

### 5.3.5 The NAND Gate

The NAND gate has two inputs and one output. The output is 0 only if **both** the inputs are 1. In all the other cases the output is 1. This is the same as attaching a NOT to the output of an AND and considering the final output. Unfortunately, NAND is not a word in English.

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 7: NAND Truth table

### 5.3.6 The XOR Gate

The XOR (Exclusive OR) gate has two inputs and one output. The output is 1 when the values of the input differ. The output is 0 if the inputs are equal.

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 8: XOR Truth table

### 5.3.7 The XNOR Gate

The XNOR (Exclusive NOR) gate has two inputs and one output. The output is 1 when the values of the input are equal. The output is 0 if the inputs are different. This can be used in comparing two binary numbers, to check if they're equal.

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Table 9: XNOR Truth table

In the simulator, there is an added feature where you can invert inputs of any gates. Inverted inputs are marked with a little circle. This means that the actual wire that comes in will be inverted when used for calculating the output. One example is an OR gate with one of the inputs inverted.

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Table 10: OR' Truth table

## 5.4 Clock

A clock in this context, is an oscillator. It provides a signal which turns ON and OFF at fixed intervals. The oscillation frequency used in GateCPU is 1 Hertz (Hz). This means that there is one wave per second. One wave comprises of being ON for half the time and being OFF for half the time. So, at 1 Hz, the clock signal spends 0.5 seconds being ON and 0.5 seconds being OFF. This process keeps repeating until the clock is disabled.

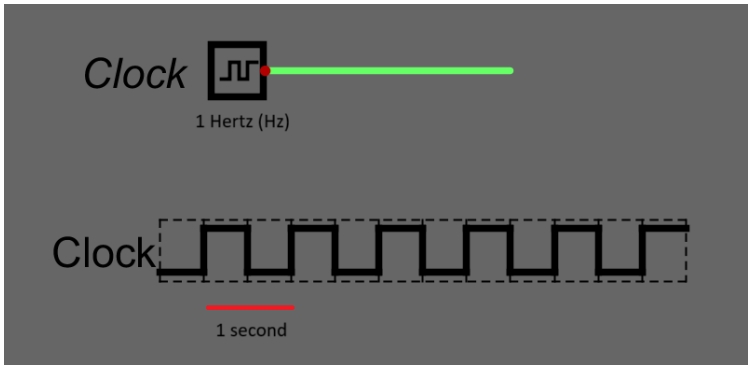


Figure 9: Signal of a clock that was set to oscillate at 1Hz

## 5.5 Driver

A driver is *logically* an AND gate. It has two inputs, if both are ON it will output HIGH. However, the inputs are not identical. One of the inputs is called the driving input. When this input is HIGH, the Output is directly connected to the other input. When this input is LOW, the Output is considered to be disconnected from the other input. In which case, the output wire will be in a high-Z state.
This is to ensure there is only one active output acting on a wire - each output that is connected to a piece of wire will be through a driver: When a specific output is to be enabled, a driving signal should be provided. This is to overcome the problem stated in section 4.5.

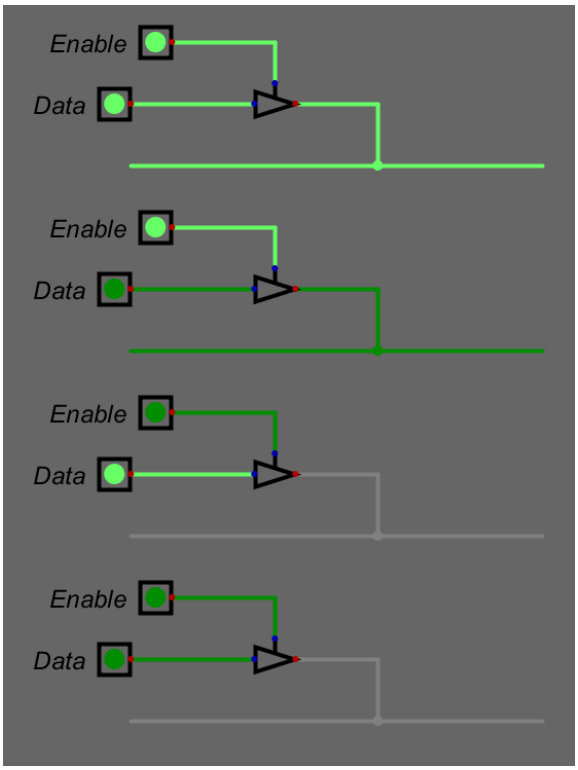


Figure 10: The four possible states of a driver and how it affects the downstream wire.

## 5.6 Edge detector

When a wire changes state, this is a device that can detect the *change* be used to send a short pulse indicating a change has happened. Have a look at figure XXXX. There are three probes. The input probe shows the signal level as given by the clock. The *rising edge* probe shows the pulse emitted by the rising edge detector. The *falling edge* probe shows the pulse emitted by the falling edge detector. There are 2 AND gates with 1 inverted input each. There are 3 gate delays. A gate delay is used to delay the transmission of the incoming signal by a small amount (Technically, it delays by exactly one unit of gate delay).
Let us consider the Rising edge detector. There are 2 inputs. When the lower input (B) is HIGH and the other input (A) is LOW, this will turn on. But both inputs are connected together, except, A has a single gate delay. Input B has the most recent signal. Input A has a slightly older information about the signal - the *past* state. During the instant where the main Input rises to HIGH (from LOW), Input A will still retain LOW, and Input B will hold HIGH. Thus, the output turns on. But after one gate delay's worth of time has passed, input A will update to to HIGH. Thus, both inputs become HIGH, and the AND gate's condition fails, turning the output LOW. Try to explore how falling edge detector works. It uses the same logic.

The input probe is behind a gate delay to synchronize the graph. The graph's X-axis is time, with t = 0 to the left and increasing toward the right. The clock is to the left.

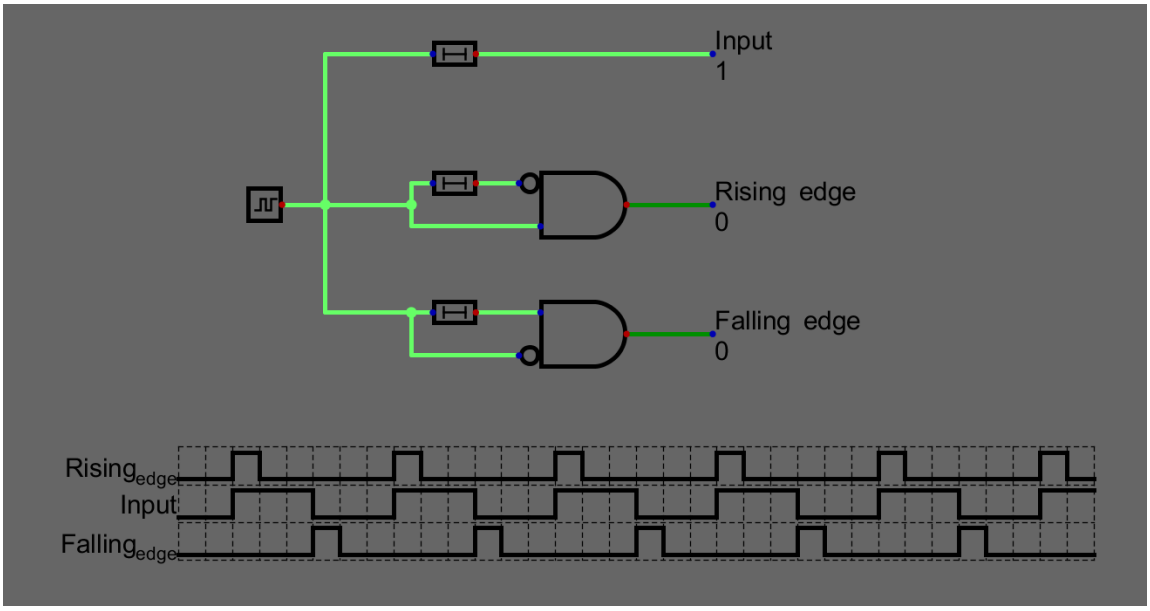


Figure 11: Edge detector circuit

## 5.7 Memory cells

These are the most fundamental parts of the circuit which are capable of storing a value, acting as memory. There are several types of them, I've used these two in my circuit.

### 5.7.1 The S-R Latch

The S-R Latch is the simplest form of memory. As the name implies, it is the equivalent of a door latch: when you *lock* a door, the latch stays locked. You cannot *further lock* the door. The only change that can happen is when you *unlock* the door. When that happens, the latch slides back to the initial unlocked position.
Consider the following figure 12. This is the digital equivalent of a door latch. Assume the off state to be this. As you can see, all gate conditions satisfy, so this is

a valid state. This is equivalent to having the door latch open, or off. Now, let us consider what happens when we RESET. This should try to un-latch the door. Since the door is already unlatched, nothing should happen. Let us evaluate this. Upon RESET, the lower NOR gate stays off. No changes occur.
When we try to SET, or latch the door, something should happen. Let us evaluate. When SET is ON, the upper NOR gate turns OFF because it now has an input. As a result, the lower NOR gate turns ON. This leads to two things - the lamp turning ON and the other input to the upper NOR gate turning ON. The processing has finished, all of this happens within ONE gate delay. The state now is referred to by figure 13.

When the SET button is no longer held, one of the inputs to the upper NOR gate turns OFF. But since the other input is ON, nothing else changes in the circuit. The circuit is now considered to be latched, or holding the ON position. This is given by figure 14

If one tries to reset the circuit, or unlatch the door, let us see what happens. Upon RESET going HIGH, the lower NOR gate turns OFF because one of its inputs are now HIGH. This leads to the upper NOR gate turning ON because both of the inputs are LOW. As a result, the output turns OFF, and the circuit is now in a latched-OFF state. Further RESETs do not affect the circuit. This is shown in figure 15

There is one disadvantage to building an S-R latch with gates like this. When the simulation is started, the circuit has an equal probability of being in an ON position or in an OFF position. But in either case, the state will be valid. To solve this, once needs to RESET every latch in the start-up procedure to ensure all latches are in a known state (OFF). I had not thought of a start-up procedure for GateCPU, therefore I went with the ready-made version of the S-R latch as given in *Digital*.

This has two outputs - Q (Output) and $\bar{Q}$ (inverse of Q). It also has the same 2 inputs - S for Set and R for Reset. This latch behaves predictably every time - when both inputs are 0 at the starting of the simulation, the output is 0.

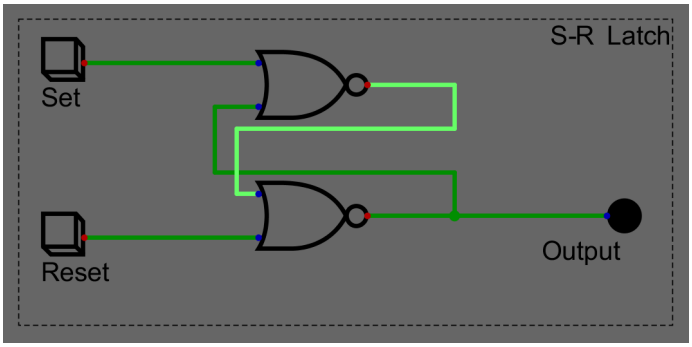


Figure 12: The S-R Latch in the OFF condition/starting condition.

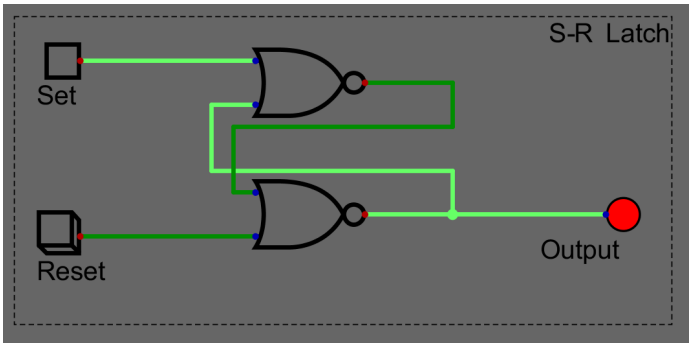


Figure 13: The S-R Latch when it is transition to ON.

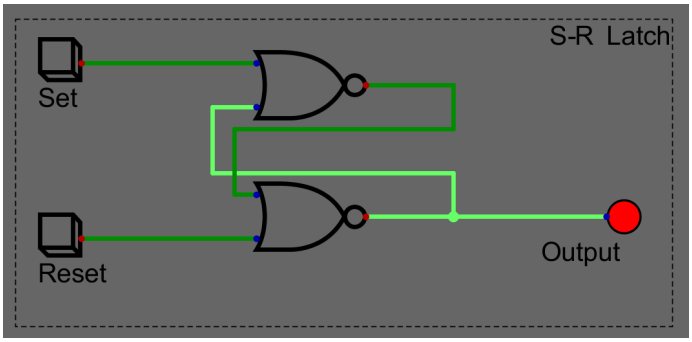


Figure 14: The S-R Latch in the ON condition.

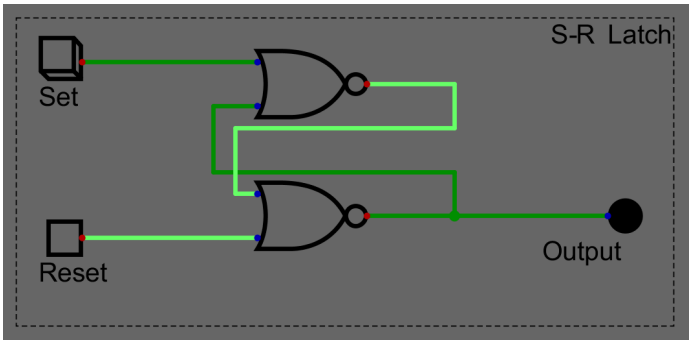


Figure 15: The S-R Latch in the transition from ON to OFF.

| Set | Reset | Q |
|---|---|---|
| 0 | 0 | Retain previous |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

Table 11: S-R Latch truth table

### 5.7.2 The J-K Flip-flop

The J-K Flip flop is an upgraded version of the S-R latch. It has three inputs J, K and Clock. It has two outputs Q and $\bar{Q}$. The are two differences from an S-R latch. One being, the states will only change when the Clock input goes HIGH - To accomplish this, a rising edge detector is used. This means that the state changes are synchronized with the rising edge of the clock signals. The other difference is, when J = K = 1, the state TOGGLES, rather than staying 0.

| J | K | Q |
|---|---|---|
| 0 | 0 | Retain previous |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | Toggle |

Table 12: J-K Flip-flop truth table

There is also an Asynchronous version of J-K flip-flop used. The advantage to that is there are the usual J, K, clock inputs as well as Set, Reset inputs in it. The J and K inputs change in synchrony with the clock, while the Set-Reset inputs can be used to change anytime - even when not in sync with clock.

## 5.8 Input/Output

In this section, components that are used for taking input from the user or displaying output from GateCPU are described.

### 5.8.1 Buttons

There are two types of buttons in *Digital*. One of them is a push button, which will output HIGH as long as the user is holding it with the mouse button. When left, it

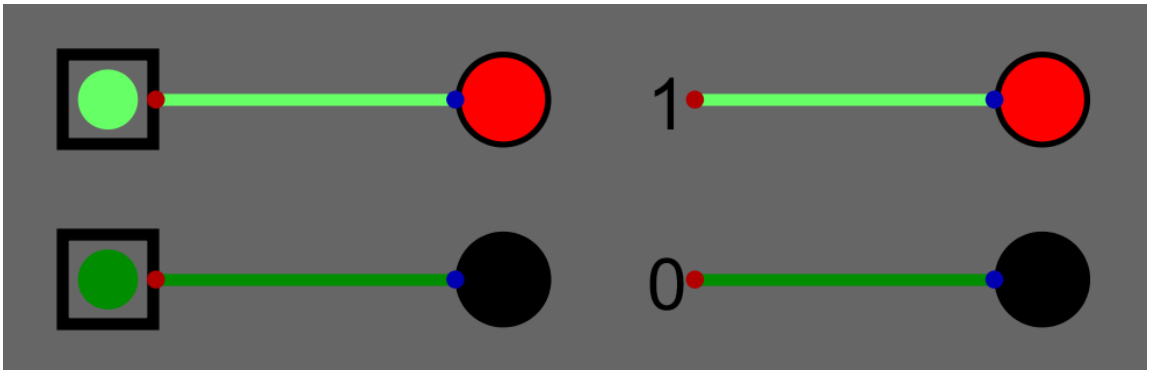


Figure 16: Buttons, lamps and constants.

will output LOW. The other button is a toggle button, it will toggle output states on press.

### 5.8.2 Constant

The constant output is a component which can be set to output either a HIGH or a LOW to the wire that is connected to it. This value cannot be changed during the simulation. This is used in Read-Only Memory, a type of memory which is not changed during the simulation. It can be changed when altering the circuit elements.

### 5.8.3 Lamp

It's a simple lamp. Glows when input is HIGH and turns black when input is LOW.

### 5.8.4 7-segment display

There are 8 inputs to this. Each one controls a particular segment, and the final one controls the decimal indicator. As each input goes HIGH, the appropriate segment glows. It turns grey when the input is LOW.

# 6 Devices

This section attempts to explain the *devices* within GateCPU. These are structures at one level of abstraction higher than the components that were previously discussed.

## 6.1 Register

A register is a series of memory cells that hold a block of information when asked to. Upon the right activation signal, it will write said information to the wires that are connected to its output. In GateCPU, registers are 16 bits long, that is, they hold 16 bits' worth of information. This is because all data is 16 bits wide. In GateCPU, they are comprised of J-K Flip-flops. Consider figure 17.

As you can see, there are 4 J-K Flip-flops connected to a 4-bit data bus. The K pins of all flip-flops are connected together to the Reset button. All the clock inputs are connected to a Clock. Now, if I want to store a value, then I need to hold the appropriate input buttons. The next time clock rises to HIGH, the values will be stored in the flip-flops. If I want to delete/erase it, then I need to hold the RESET button. The next time clock rises to HIGH, the values are erased from the flip-flops. This is a simple 4-bit register.

However, there are flaws with this model. Firstly, consider the outputs. The outputs cannot be used anywhere, or transferred back to the bus if needed. Why not, you may ask. Let us try connecting the outputs to the bus, and writing a value. An error pops up. This is because of the limitation mentioned earlier (hyperref). So, we need to add an output driver to each output that is writing to the bus. Check figure 18

Now, there are 4 drivers, whose enable pins are all connected together to the *Enable write* button. When this button is held, the values of the J-K flip-flops are

sent to the bus. There is also the *Enable input* button which is to enable writing the button inputs to the bus. But now, since by default neither outputs are enabled, the BUS wires are essentially unconnected. So we need to add pull-down resistors to ensure correct behaviour at the start. As long as we ensure both enables are not ON at the same time, this simulation should be fine.

There is one more flaw with this model. This register ALWAYS reads from the bus. We don't want this behavior. It is necessary that any register should read only when instructed upon. Therefore, to solve this, we can either add a driver (and pull-down resistors) or an AND gate at the input. I've opted for the latter solution. The circuit is shown in figure 19. This will read from the bus if READ is HIGH and clock rises, it will reset the register if RESET is HIGH and clock rises. Finally, it will write to the bus if WRITE is HIGH. The registers in GateCPU have the same exact configuration as this but with 16 flip-flops with one extra addition: an S-R Latch to latch the WRITE activation. It is latched on by a Start WRITE command, and latched off by a STOP WRITE command.

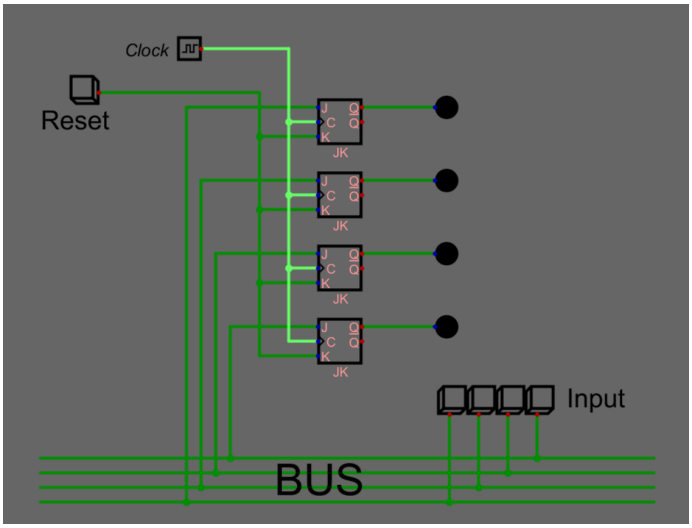


Figure 17: First iteration of the register

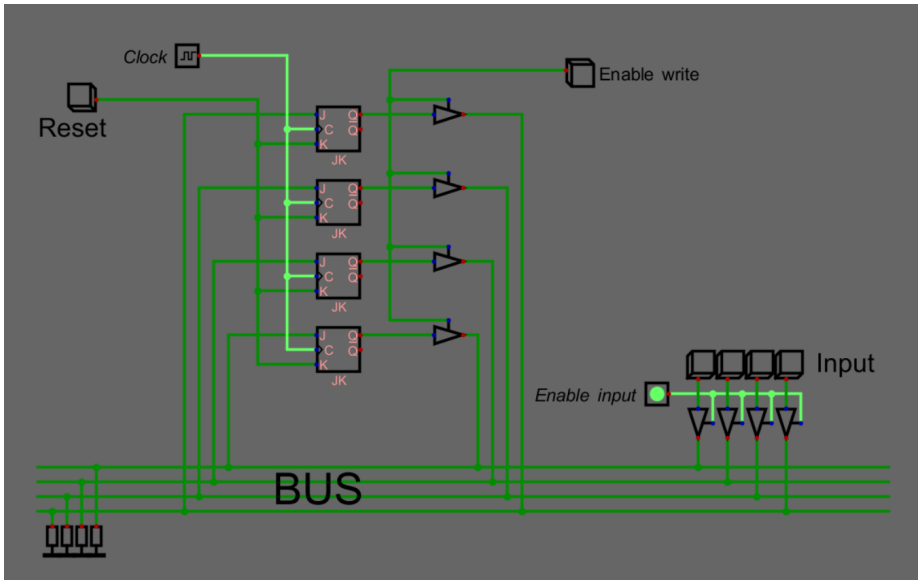


Figure 18: Second iteration of the register, including drivers for the output

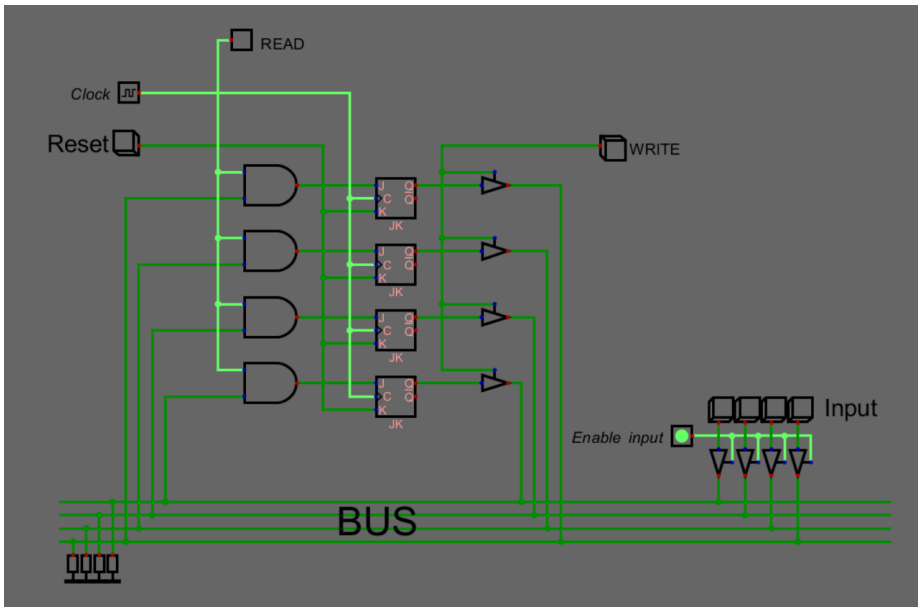


Figure 19: Third iteration of the register, with drivers for both output AND input

## 6.2 Adder

An adder is a circuit that can add 2 or 3 bits. It is considered a half-adder if it can add 2 bits, and a full adder if 3. When 2 bits are added, the truth table is given in table XXX. Note that the LSB of the output is basically an XOR of the inputs. The MSB is an AND of the inputs. This is shown in the figure 20.

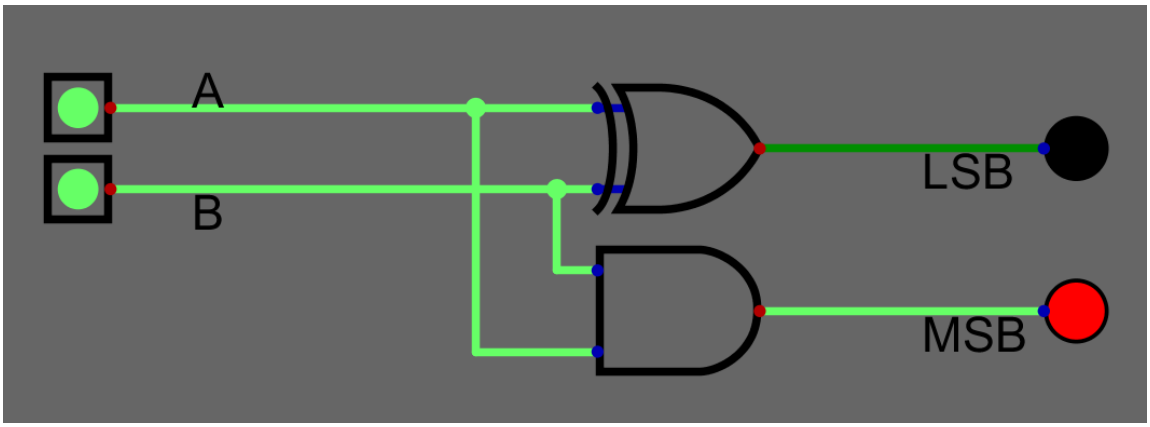


Figure 20: Half-adder

| A | B | Output MSB | Output LSB |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Table 13: Half-adder truth table

Now, to add two binary numbers, one would need several of these adders in parallel, one for each digit. But is that enough? What happens when there is a carry over? This circuit is not enough as it cannot add a carry over. One would need a Full adder for that. The circuit in fig 21 is a full adder circuit, it takes a carry-in along with A and B to produce a 2 bit output - an output number an a carry-out.

Now, if one wants to add two 4-bit numbers, then all you need to do is to have 4 full adders in parallel as shown in fig 22, with the carry-over of the lesser bit feeding into the carry-in of the higher bit. In GateCPU, there is a 16-bit full adder connected in this manner. The final carry-out bit is ignored. The carry-in bit of the LSB adder is used in twos' complement; one can set that specific $C_{in}$ to HIGH to add 1 to the otherwise usual total. This will be explained in the opcodes section later.

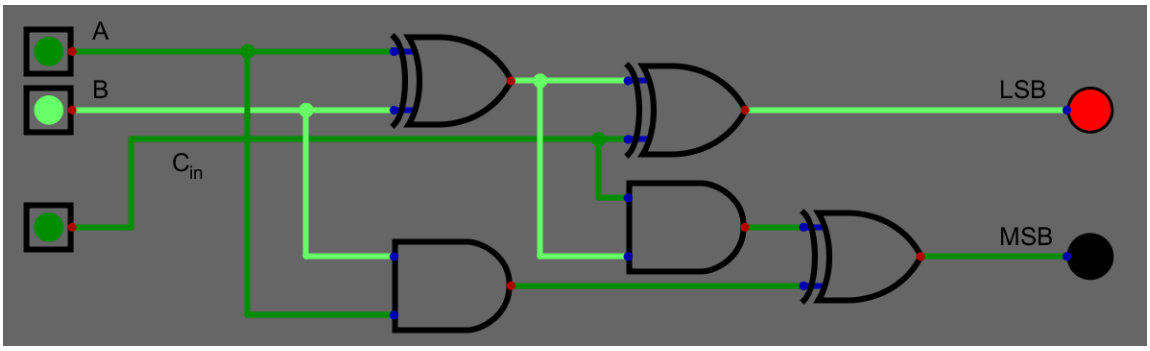


Figure 21: Full-adder

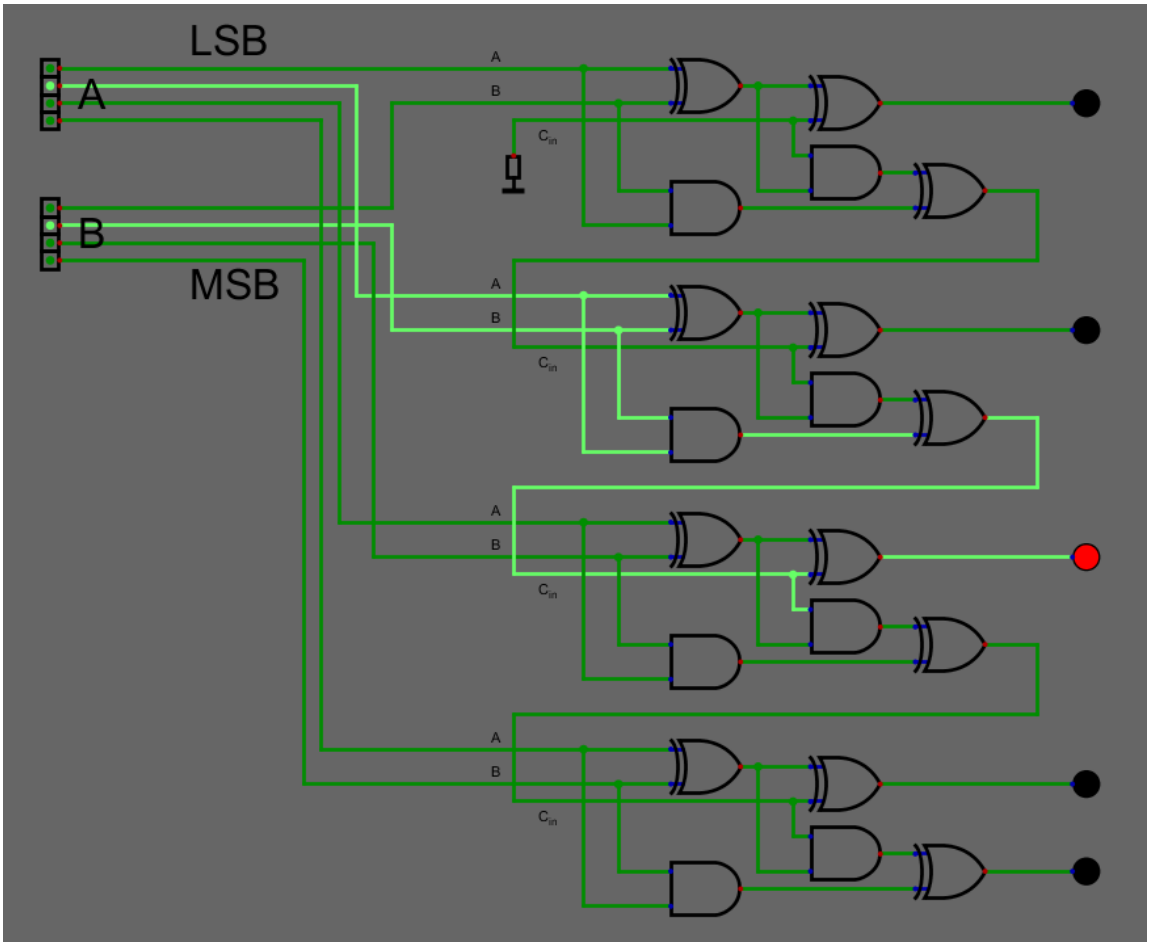


Figure 22: 4-bit adder

## 6.3 Counter

A counter is a circuit consisting majorly of J-K Flip-flops that *counts* the number of pulses inputted to it. There are two ways of thinking about it: One, it outputs a binary number which is a direct count of the number of pulses. Two, every bit in the number oscillates at $2^{-n}$ times the frequency of the clock, where n is the significance of the bit considered, starting with 1. That is, if the clock runs at 2048 Hz, the LSB oscillates at 1024 Hz, tens bit oscillates at 512 Hz, hundreds bit at 256 Hz, thousands bit at 128 Hz, and so on. It is limited only by the number of bits that have been wired. When there is a 4-bit counter and it has reached state 1111, the next pulse will bring it back to 0000.
A counter is used for many purposes. In GateCPU, it is used to *advance* the program. As our program has many lines, to execute them sequentially, we use a clock and and a counter. There is another component called as an activator which activates the specific line of code as given by the counter.

To design the unit bit of the counter, use a J-K flip-flop to ensure the output stays activated until the next clock cycle. Upon the next clock cycle, it should toggle OFF. This is given by diagram 23. The toggling behavior is enabled by setting J = K = 1, which means at the rising edge of every clock input it will toggle. The tens' bit has to turn ON only if the CLOCK input is rising AND the ones' bit is ON. When this condition is true, it should be activated. This can be done using an AND gate. Every subsequent bit can be realized in the same way: It should be turned ON on a rising clock cycle as well as when ALL the previous bits are ON. This is given in the final circuit diagram of the counter in fig 24

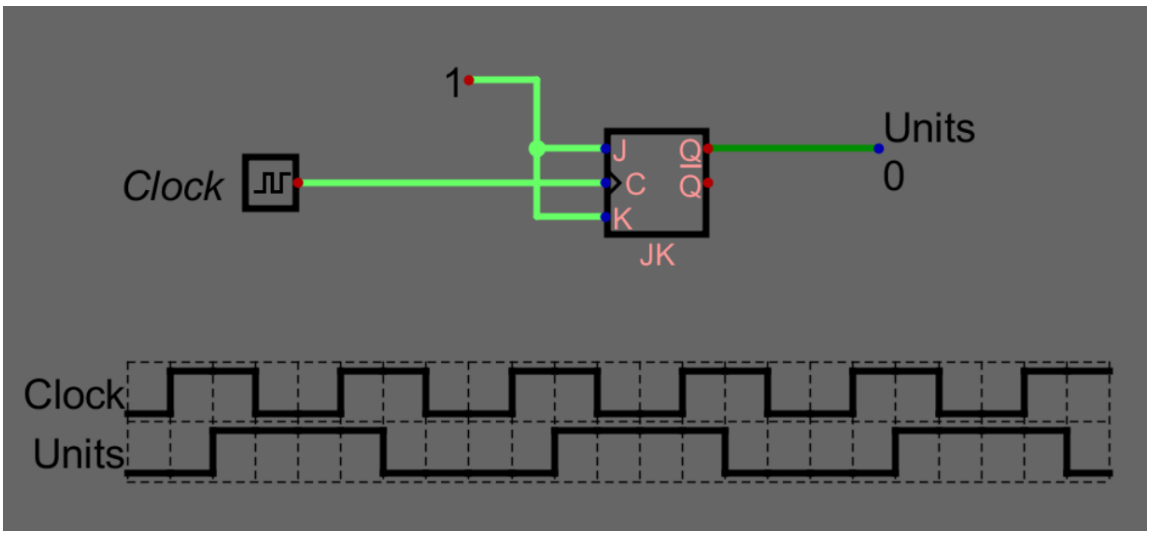


Figure 23: Single bit counter. Runs at half the clock rate

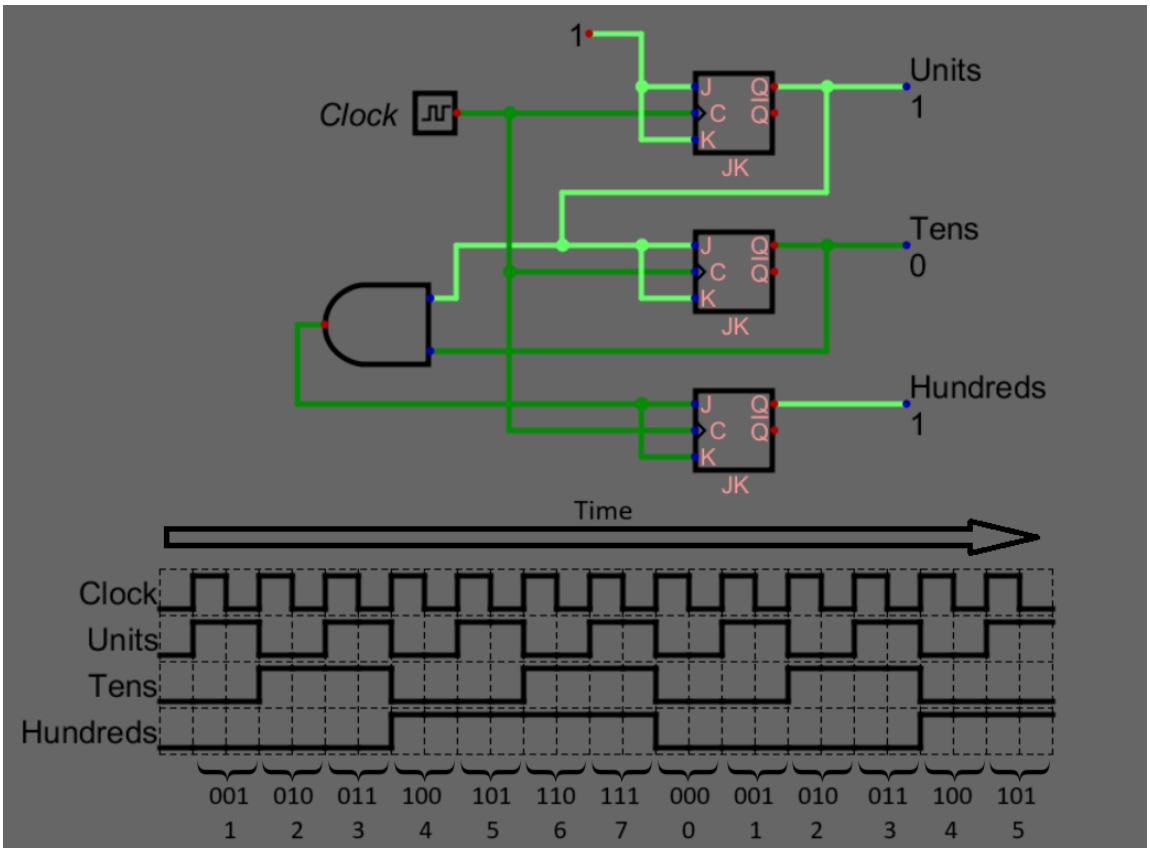


Figure 24: 3-bit counter. Can count from 0 to 7

## 6.4 Decoder/Enabler

This is one of the most simple devices, but also the most used. This is basically an AND gate with an appropriate number of inputs, with the appropriate inputs inverted. The objective of this device is to encode an *if* condition. *If* the input bits are, say, 10111, *then* output 1, else 0. Almost *all* component activation is based on this concept. A 'circuit diagram' for the above example, along with another example of an activation of 01100 is shown in figure 25:

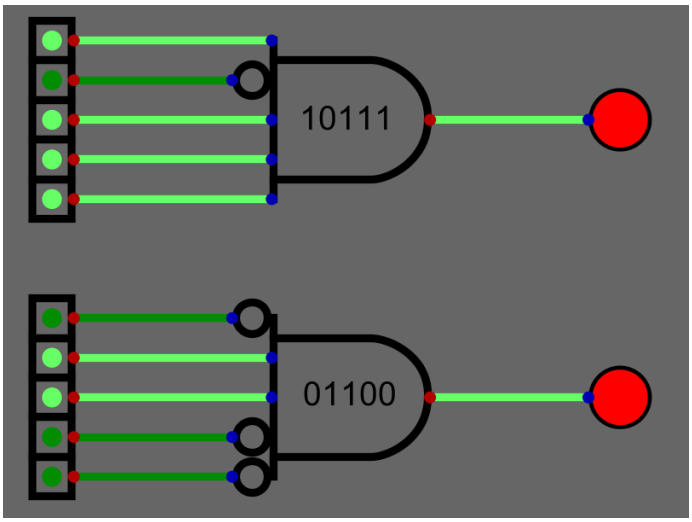


Figure 25: Activator/enabler. The top gate activates when the input is 10111 and the bottom gate activates when the input is 01100

## 6.5 Comparator

Typically, a comparator compares two binary numbers A and B. It outputs a signal in either of three wires. A signal on wire X means that A is larger than B. A signal on wire Y means A and B are equal. A signal on wire Z means B is larger than A. However, the circuit to make this for a 16-bit comparator is huge, and I felt it was unnecessary.

The comparator in GateCPU is a device that outputs HIGH if both the inputs A and B are equal. It is an equality comparison only. Logically, it is the equivalent of "Is A == B?". This can be achieved by having 16 XNOR gates in parallel. Each XNOR gate will be HIGH if and only if that specific bit is same in both A and B. For the entire number A to be equal to B, all the bits have to be same, so ALL XNOR gates should output HIGH. To achieve this, one can use a 16-bit wide AND gate and connect all the XNOR gates' output to the AND's input. However *Digital* does not have 16-bit input gates, so I have to use multiple AND gates to satisfy this.

## 6.6 Input

There are two kinds of input present in GateCPU. They are Manual input and Programmed input.

### 6.6.1 Manual input

This is a set of input buttons whose activation is directly controlled by a button that sits right next to it. This method of input is to be used with **CAUTION**, as data flow is **NOT** interrupted upon activation. It may result in a simulation error.

### 6.6.2 Programmed Input

While this method of input also has a set of input buttons, it is not directly connected to the bus. Instead, it is connected to a latch which stores the user input. As you may notice from figure XXX, the buttons used are push buttons instead of toggle buttons like in Manual input. The lamps indicate the value stored in the latches, which will be written to data bus upon tapping ENTER.
Upon activation, which is from a specific Opcode, this device will get activated. When this occurs three things happen simultaneously. One, the display below it will glow and read "Enter input". Two,, The clock is stopped*. It will resume only if the gate that controls the clock outputs the appropriate value. Three, the last three bits

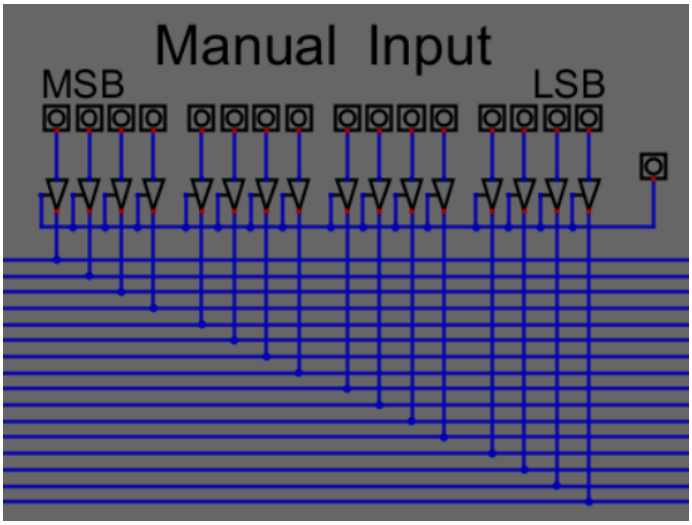


Figure 26: Manual input

of the *address* part of the program instruction gets copied into a register. This will determine the number of rising edge clock pulses for which the inputted data will be written to the data bus.
The user should enter the appropriate number. If there is an error when entering, the user can tap RESET and enter it again. When the user is ready to pass the number to GateCPU, they should tap ENTER. The program will wait indefinitely until ENTER is pressed.

When ENTER is pressed, the driver latch is activated, thereby allowing the latches to write to the data bus. A counter is started. It will keep counting until it gets a master RESET signal. Meanwhile, a comparator compares the counter's output with the number stored in the register mentioned above. This value is met when a sufficient number of clock cycles has passed according to the instruction. When this happens, the master RESET signal is triggered, which resets several things - the register, the counter and the driver latch. This means that the driver stops working, thereby disconnecting the latches from the bus.

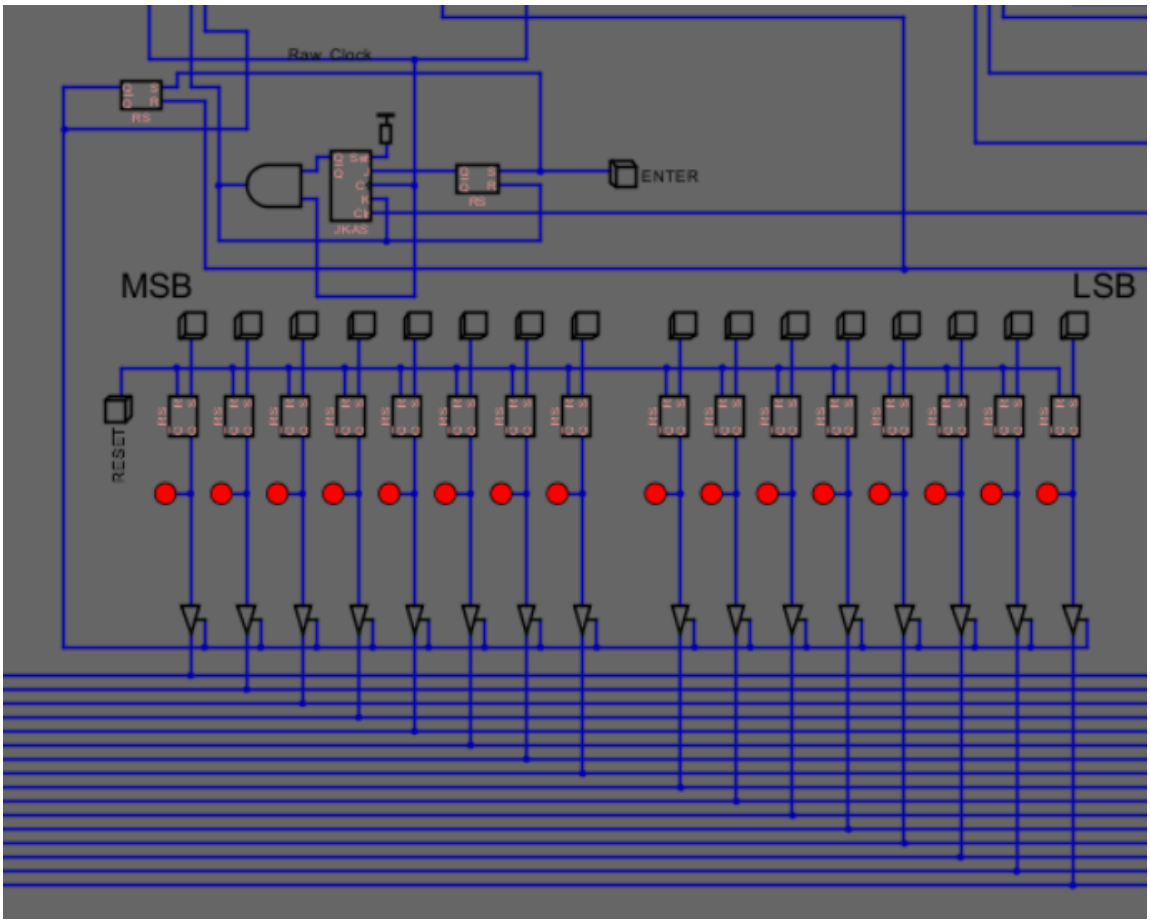


Figure 27: A part of the programmed input device.

## 6.7 Display

There are 2 displays in GateCPU. One of them reads "Enter input" when activated. The other one shows the (15 + 1) bit binary number that is in the bus, upon activation. It is activated by a specific Opcode. There is an S-R register connected to the bus, which feeds to the display. When the PRINT operation is initiated, 3 things happen sequentially. One, the register is reset. Two, the values from data bus is copied to the register, Three, this value is printed indefinitely. This means that the display will show the last printed number until the next print command arrives, which clears the screen automatically and prints the new value. The display is wired up accordingly to print 1 if there's an incoming signal and 0 if there is none.

## 6.8 GOTO

The intentions of the if/else/GOTO device is to evaluate an IF-ELSE condition, and if it fails, skip to a particular line in the program. The target line can also be *before* the line where GOTO is. The full process goes on like this:

1. If LSB in data bus is 1 AND the instruction to initiate IF/ELSE GOTO command is present in the opcode bus, then the GOTO device is activated.

2. When the GOTO device is activated, all the 6 bits in the address bus gets copied to a register. This represents the target line number of the program that is to be jumped to.

3. The counter is disconnected from the program code via the use of an AND gate. The clock still runs, but program code is not activated.

4. A separate comparator actively checks if the output from the aforementioned register and the counter are equal.

5. If they are equal, then the program should resume from this point, so a master RESET signal connects the counter to the program and resets the register.

If the target line is *before* the present line, then this will still work because the program counter loops from the starting once it reaches the last line of code. So for example, if line 38 is a GOTO statement asking to jump to 20, the program counter will count to 63 and then from 0 till 20. When it reaches 20, program execution is resumed.

# 7 Technical reference

## 7.1 Programming

Program lines are written in the top-right "Program memory" section of GateCPU. Each line consists of a 4-bit opcode and optionally a 6-bit address. For most operations, an address is required for meaningful execution. 'Write' here implies that the circuit is changed before simulation is started - the 1's and 0's of the Program memory have to be changed.

## 7.2 OPCODES

There are 15 possible opcodes, of which 11 are defined currently. Starred ones do not require an address. These are:

- **0001 - READ:** This operation commands the register that is pointed by the address to READ the value in the Data bus for the duration of a clock cycle. It has no effect on the register when the next line of program is being executed.
- **0010 - UNUSED**
- **0011 - Start WRITE:** This operation sets the specified register to WRITE mode. The output of this register will be connected to the data bus for an

indefinite duration, until a Stop WRITE operation has been called. Make sure to not WRITE two registers into the bus at the same time or the simulation will halt.

- **0100 - Stop WRITE:** This operation stops the specified register from writing to the data bus.
- **0101 - SET $C_in = 1$*:** This operation sets the carry-in bit of the LSB in the adder to HIGH for 2 full cycles.
- **0110 - PRINT*:** Prints the number that is currently in the data bus to the display
- **0111 - RESET Register:** This commands the specified register to reset.
- **1000 - Start If/WRITE*:** If the values at register A and B are equal, then start writing bus MSB = 1.
- **1001 - UInput:** This instruction asks the user for input and pauses the program until the user has inputted a number and pressed ENTER. The last 3 (LSB side) bits determine the number of ticks for which the number stays on the bus.
- **1010 - Stop If/WRITE*:** This instruction stops If/WRITE from writing bus MSB.
- **1011 - HALT*:** Upon activating this command, the entire process halts and the clock is disabled. No further lines of code will be processed.
- **1100 - If/Else GOTO:** This is the If/Else GOTO command mentioned here XXXX. When this command is activated, it checks if the bus MSB == 1. If this condition is satisfied, then execution continues as is. If not, it will skip to the program line number one more than the number mentioned in the address bus. For example, GOTO 48 will skip to line 49.

Do note that number of cycles as mentioned above includes the command's cycle. So if you have a command $C_in = 1$ at line 41, it will stay only until line 42. However, I encourage you to test it out before writing your program.

## 7.3 Addresses

Addresses range from 000 001 to 111 111. However, many of these addresses remain unused. The used ranges are shown in table XXXX

| | |
|---|---|
| 000 001 (1) | Register A |
| 000 010 (2) | Register B |
| 000 011 (3) | Register C |
| 000 100 (4) | Register NOT |
| 000 101 (5) | Register bitreverse |
| 001 000 (8) to 011 111 (31) | Working memory |
| 100 000 (32) to 101 111 (47) | Read Only Memory |

Table 14: Special addresses

Register A and B are used for the adder. They have separate READ address but they have a common WRITE address. They both WRITE into the adder when WRITE A is called upon. Register C holds the value of the output from the adder. Register 4 is the NOT register, which stores the complement of the number in the data bus when reading. Register 5 is bitreverse, it reverses the order of bits when writing back to bus.

## 7.4 GateCPU settings

- **Enable CLK:** Toggle switch which is ON by default. Turn this OFF if you want to disable the clock altogether.

- **Enable Autostop:** Toggle switch which is OFF by default. Turn this ON if you want the program to halt after line 63. Keep in mind this may break GOTO-previous functionality.
- **Initiate program:** Toggle switch which is ON by default. Set this to OFF by default if you don't want the program to start. The clock still runs, though. May cause unexpected behavior when turned OFF amidst a program.
- **Display ON:** Toggle switch which is ON by default. Set this to OFF if you do not want the display to function.

## 7.5 Limitations

Will be updated...

# 8 Tested programs

These three programs have been tested to work on GateCPU.

## 8.1 Adding

This program obtains 2 numbers from the user and displays their sum.

| Line number | Program | Translation |
|---|---|---|
| 1 | 1001 000 010 | INPUT 2 |
| 2 | 0001 000 001 | READ A |
| 3 | 1001 000 010 | INPUT 2 |
| 4 | 0001 000 010 | READ B |
| 5 | 0011 000 001 | START WRITE A, B |
| 6 | 0001 000 011 | READ C |
| 7 | 0100 000 001 | STOP WRITE A, B |
| 8 | 0011 000 011 | START WRITE C |
| 9 | 0110 000 000 | PRINT |
| 10 | 0100 000 011 | STOP WRITE C |
| 11 | 1011 000 000 | HALT |

## 8.2 Subtracting

This program obtains 2 numbers from the user and subtracts the latter from the former. The result is in twos' complement form, so it may not be directly readable.

| Line number | Program | Translation |
|---|---|---|
| 1 | 1001 000 010 | INPUT 2 |
| 2 | 0001 000 001 | READ A |
| 3 | 1001 000 010 | INPUT 2 |
| 4 | 0001 000 100 | READ 4 |
| 5 | 0011 000 100 | START WRITE 4 |
| 6 | 0001 000 010 | READ B |
| 7 | 0100 000 100 | STOP WRITE 4 |
| 8 | 0011 000 001 | START WRITE A |
| 9 | 0101 000 000 | ENABLE C-in |
| 10 | 0001 000 011 | READ C |
| 11 | 0100 000 001 | STOP WRITE A |
| 12 | 0011 000 011 | START WRITE C |
| 13 | 0110 000 000 | PRINT |
| 14 | 0100 000 011 | STOP WRITE C |
| 15 | 1011 000 000 | HALT |

## 8.3 Add or subtract

The following program obtains 2 numbers A and B from the user, in that order. Then it requests for a third number. If the MSB of the third number is 1, then the program prints A+B. Else, it prints A-B, but not in twos' complement form. It prints in a readable form: it prints the number without sign first, then adds the sign before it. For example, 0 101 = +5 and 1 101 = -5, where the leading bit is sign bit.

| Line number | Program | Translation |
|---|---|---|
| 1 | 1001 000 010 | INPUT 2 |
| 2 | 0001 000 001 | READ A |
| 3 | 1001 000 010 | INPUT 2 |
| 4 | 0001 001 000 | READ 8 |
| 5 | 1001 000 010 | INPUT 2 |
| 6 | 1100 010 000 | IF/ELSE GOTO 15 |
| 7 | 0011 001 000 | START WRITE 8 |
| 8 | 0001 000 010 | READ B |
| 9 | 0100 001 000 | STOP WRITE 8 |
| 10 | 0011 000 001 | START WRITE A |
| 11 | 0001 000 011 | READ C |
| 12 | 0100 000 001 | STOP WRITE A |
| 13 | 0011 000 011 | START WRITE C |
| 14 | 0110 000 000 | PRINT |
| 15 | 1011 000 000 | HALT |
| 16 | 0011 001 000 | START WRITE 8 |
| 17 | 0001 000 010 | READ 4 |
| 18 | 0100 001 000 | STOP WRITE 8 |
| 19 | 0011 000 100 | START WRITE 4 |
| 20 | 0001 000 010 | READ B |
| 21 | 0100 000 100 | STOP WRITE 4 |
| 22 | 0011 000 001 | START WRITE A |
| 23 | 0101 000 000 | SET C-in |
| 24 | 0001 000 011 | READ C |
| 25 | 0100 000 001 | STOP WRITE A |

| Line number | Program | Translation |
|---|---|---|
| 26 | 0011 000 011 | START WRITE C |
| 27 | 1100 101 001 | IF/ELSE GOTO 43 |
| 28 | 0111 000 100 | RESET 4 |
| 29 | 0001 000 100 | READ 4 |
| 30 | 0100 000 011 | STOP WRITE C |
| 31 | 0111 000 010 | RESET B |
| 32 | 0111 000 001 | RESET A |
| 33 | 0011 000 100 | START WRITE 4 |
| 34 | 0001 000 010 | READ B |
| 35 | 0100 000 100 | STOP WRITE 4 |
| 36 | 0011 100 000 | START WRITE 32 |
| 37 | 0001 000 001 | READ A |
| 38 | 0100 100 000 | STOP WRITE 32 |
| 39 | 01111 000 011 | RESET C |
| 40 | 0011 000 001 | START WRITE A |
| 41 | 0001 000 011 | READ C |
| 42 | 0100 000 001 | STOP WRITE A |
| 43 | 0011 000 011 | START WRITE C |
| 44 | 0110 000 000 | PRINT |
| 45 | 1011 000 000 | HALT |

# 9 Everything together

This section explains in detail about how the add program in section 8.1 works.

1. **INPUT 2:** The input device is activated. Once enter is pressed, the input will stay on the data bus for the full duration of the *next* tick.
2. **READ A:** The A register reads from the data bus, which currently holds the

value inputted from Line 1.

3. **INPUT 2:** The input device is activated again.
4. **READ B:** The B register reads from the data bus, which currently holds the value inputted from Line 2.
5. **START WRITE A, B:** As A and B have the same address for WRITE operations (000 001), this command initiates writing of A and B register into the adder. It will keep feeding into the adder until a STOP command is encountered.
6. **READ C:** The C registers reads from the adder. Essentially, it reads the output of the adder, which is A + B.
7. **STOP WRITE A, B:** The start write command from Line 5 is now stopped, and A and B no longer write to the adder.
8. **START WRITE C:** The C registers starts writing to the data bus.
9. **PRINT:** The print statement prints the value in the data bus into the output display.
10. **STOP WRITE C:** This command stops C from writing into the data bus, which happened in line 8. This does not affect printing because the print device copies the value from the data bus when printing.
11. **HALT:** This command halts the program.

# 10 Further work

1. **Multiplication** A program to multiply two integers is yet to be written. With the availability of a GOTO command, this is possible. However, it is not clear whether it is possible to write this under 63 lines.

2. **Fixed point arithmetic** Currently, I have not explored decimal computing. Fixed point decimal arithmetic is not very different from integral computing, so this may be possible without significant change in circuitry.

3. **Division** Dividing two numbers may be possible when Fixed point arithmetic has been implemented.

4. **Python integration** Programming in GateCPU is done by changing the 1's and 0's in the circuitry manually. It can be made much simpler if there were a python program that can directly read the .dig file and change the lines accordingly.

   A python program can also be written to translate between binary and assembly/human readable code. Both of these features can be integrated together to smoothen the process of programming on the GateCPU to a great extent.

5. **Dynamic programming** GateCPU at present cannot change the executed code when the simulation is running. It can only ask the user for input. It only really works with the data bus. If there is another module or device that works with the program-line registers, one can enter code during runtime, acting just like a contemporary computer. However, this may require a major overhaul because currently program-line registers are Read-Only

6. **Hitting limits?** When new components are added, it becomes cumbersome to check for gate delays everywhere. It may be best to build a newer version "GateCPU v2" from scratch, considering gate delays at each and every wire.

# 11 Thanks!

Thanks for making it this far. Any comments/suggestions/feedback is appreciated. Please reach me at **mugilan.an@gmail.com** regarding the same.