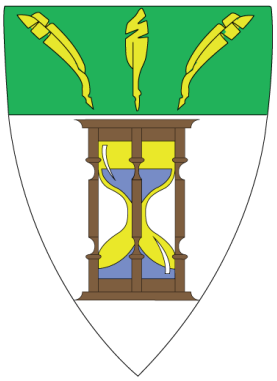
Les travaux personnels du Lycée Ermesinde Mersch



The Anatomy of a Computer

How computers work, explained with Computer Logic

Jan Masanas Amer

Classe : 4CLA3

Tuteur : Ken Vallender

Avril 2023

Table of contents

The Anatomy of a Computer

Introduction

Computers have come a long way since their inception, and today they play an integral role in virtually every aspect of our lives. From social media and online shopping to complex scientific simulations and medical diagnoses, computers are the engines that drive our modern world. Yet, despite their ubiquitous presence, many of us still have only a limited understanding of how they work.

What lies beneath the glossy exterior of our laptops and smartphones? How do they process the vast amounts of information we feed them every day, and what makes them capable of performing such complex tasks with seemingly effortless ease? In this TraPe, we will embark on a journey of discovery, exploring the inner workings of a computer on the binary level. We will examine the basic building blocks of a computer system, including the core Processors (CPU/GPU), memory (RAM), and the ways they communicate with each other.
We will delve into the intricacies of binary, the language of computers, and understand how it is used to transmit and process information. We will also look at some concrete diagrams and circuits, to fully understand how it works.

This deep dive into the workings of a computer is intended for anyone who wants to gain a better understanding of how these machines operate. The following explanations assume you have a basic understanding of how computer logic works (logic gates). Whether you are a student, a software developer, or simply someone who is curious about the technology that powers our lives, this project will provide you with a comprehensive and in-depth look at the mechanisms that make computers run.

This TraPe can be read in no particular order, and I highly recommend you read the explanations to each individual circuit, when they are mentioned in the TraPe, in the chapter titled *Explanations of specific circuits.*

RAM (Random Access Memory)

RAM, also known as Random Access memory, serves the purpose of a temporary memory. this memory is used to store important data that the computer needs to access quickly, like for example a document that is being edited or your clipboard.

Data is stored in RAM using a system of addresses and registers. Each register holds a certain amount of data, typically 8 bits, and is assigned a unique address. You can think of a register like a box, where you can store a number between 0 and 255, and the address is like a label with a number written on it.

When the computer needs to access or modify data in the RAM, it first sends the address of the desired register through the bus, which consists of several pins labelled A0 to A3 on this diagram. Next, it sends a command to either write new data to the register or read the data stored in the register. This command is sent through the CS and WE pin, which stand for Chip Select and Write Enable, respectively.

When the RAM receives a Read operation, it retrieves and outputs the data stored at the specified address through the output pins O1 to O4.

On the other hand, if the RAM receives a Write operation, the computer needs to provide the data to be written to the input pins D1 to D4 along with the target address. The RAM then replaces the existing data in the register at that address with the provided data.

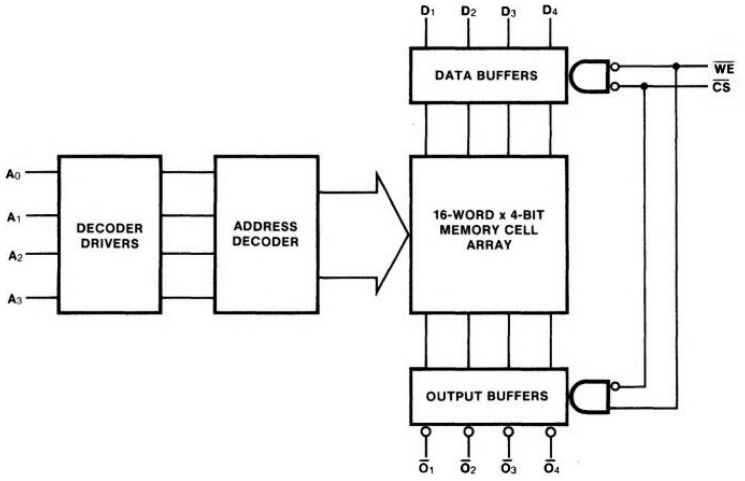
The 5 main components of RAM are the Address decoder, Decoder Driver, Memory Cell array, Data Buffer and Output Buffer.

FUNCTION TABLE

INPUTS $\overline{CS}$	INPUTS $\overline{WE}$	OPERATION	CONDITION OF OUTPUTS
L	L	Write	High Impedance
L	H	Read	Complement of Stored Data
H	X	Inhibit	High Impedance

H = HIGH Voltage Level
L = LOW Voltage Level
X = Immaterial

LOGIC DIAGRAM

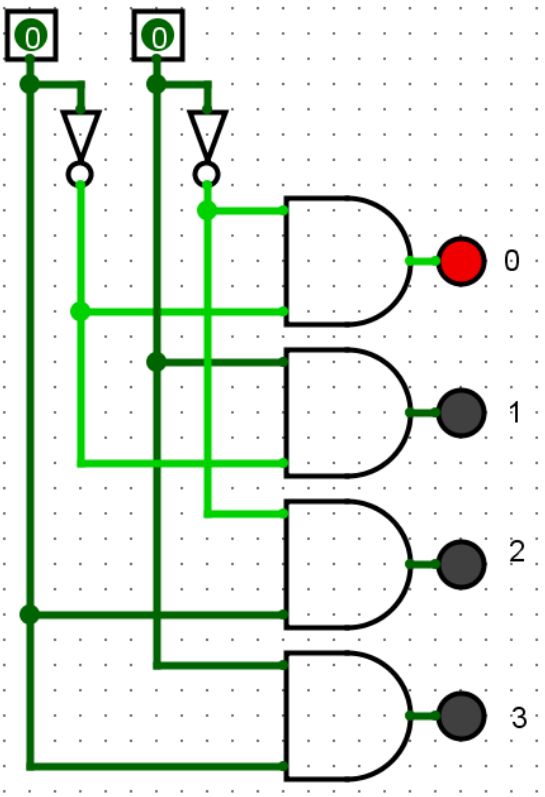


ADDRESS DECODER

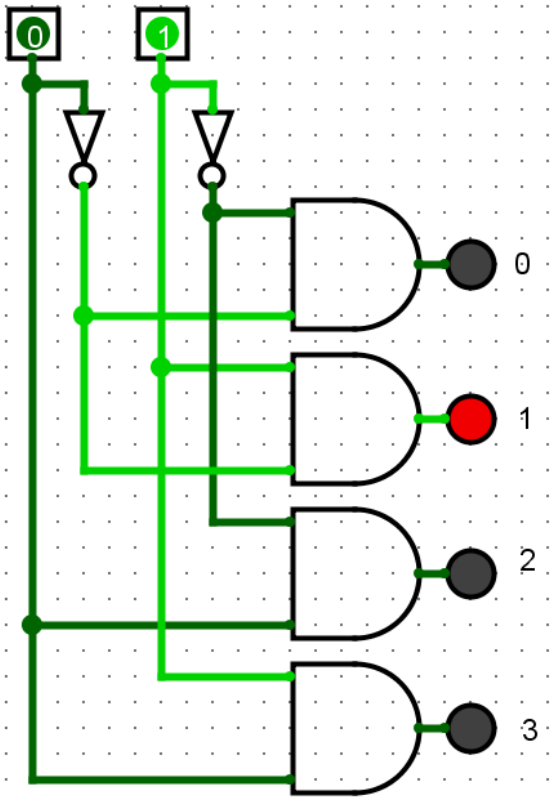
The Address decoder decodes a binary output so that one wire corresponds to one address.
For example, 00 in binary (=0 decimal) corresponds to output 0, 01 (=1 decimal) to output 1, 10 (=2 decimal) to the third and so on.

These outputs are then sent to the memory cell array, where the actual data is stored and processed.

This example uses a 2-bit Address decoder for the sake of simplicity, while most modern computers use 32 bits (allowing around 4 billion addresses and with 1 byte per address is 4GB of RAM,



instead of the 4 bytes in this example).



Input 1	Input 2		Output 0	Output 1	Output 2	Output 3
0	0		1	0	0	0
0	1		0	1	0	0
1	0		0	0	1	0
1	1		0	0	0	1

MEMORY CELL ARRAY, DATA BUFFER & OUTPUT BUFFER

The memory cell array is the main bulk of RAM. It is where all the data is stored and consists of an array of so called “cells”, which hold 1 bit of data each.

A single cell contains 3 main parts: SR-Latch, the SR controller, and the Output controller.

The SR-Latch (Blue region on diagram) stores the bit of information (more on this circuit in “List of reoccurring circuits”).

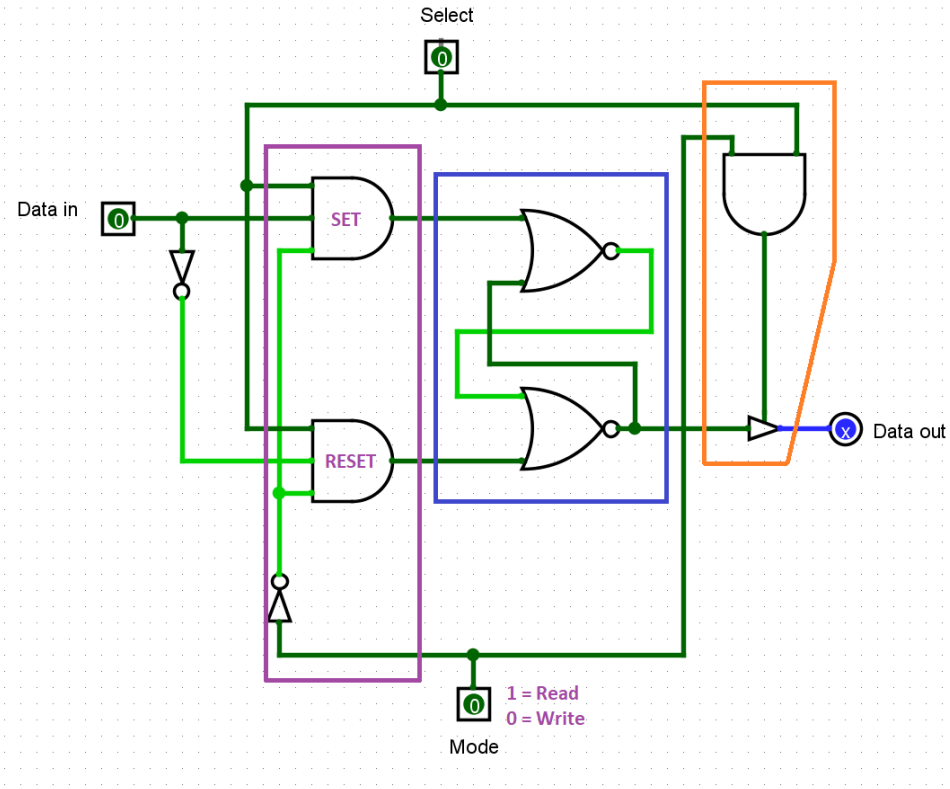
The SR controller (Purple region on diagram) controls the state of the SR-Latch. It only SETs it if:

Select = 1 (the Address decoder has selected it)
Mode = 0 (mode 0 is Write & mode 1 is Read)
Data in = 1

The SR controller only RESETs if:

Select = 1
Mode = 0
Data in = 0

The Output controller (Orange region on diagram) controls when the data should be outputted to Data Out. It does this using an AND gate and a Controlled buffer. It only opens the controlled buffer if both Select and Mode (Mode 1 is read) are 1, outputting the data.



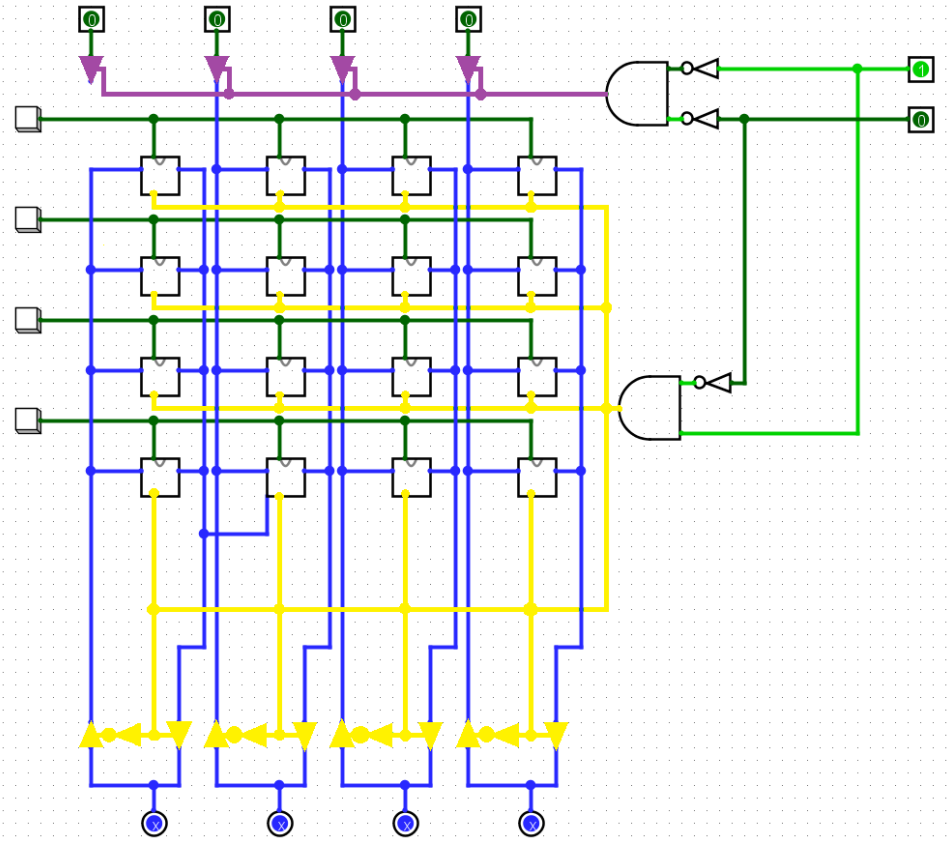
An 8 GB RAM's cell array holds about 64.000.000.000 of these cells, but in the following example, the array will be just 4 by 4

The following diagram contains the data buffer (purple circuit), the Output buffer (yellow circuit) and the cell array (the rest of the diagram). The green circuits are connected to each individual Output on the address decoder.

When the Output buffer is on (WE = 0 & CS = 1), it sets the "Mode" pin on each individual cell to 1, meaning they enter "Read" mode. It also outputs the selected cells' "Data out" pins to the output bus.

When one of the addresses (Dark green cables) is selected, all the connected cells' "Select" Inputs are set to 1.

When the Input buffer is on (WE = 0 & CS = 0), it sets the selected cells' "Data" pins to whatever is in the input bus.



As such, the CPU can access the RAM at any time extremely efficiently by just inputting the address in binary to the address decoder, the operation (Read/Write) and (if it is writing,) the Input.

CPU

CPUs (Central processing Unit) in their early days consisted of 3 core components: The Arithmetic and logic unit (ALU) and the Control unit (CU) Memory/Storage Unit. The memory Unit is in our case the RAM (Explained in RAM chapter). It is technically a part of the CPU but most regard it as a component of its own.
More modern CPUs have many more components, but for the sake of simplicity, I will explain how their simpler ancestors worked.

CPUs work in Cycles. Each cycle completes exactly one instruction. A cycle consists of these 4 phases:

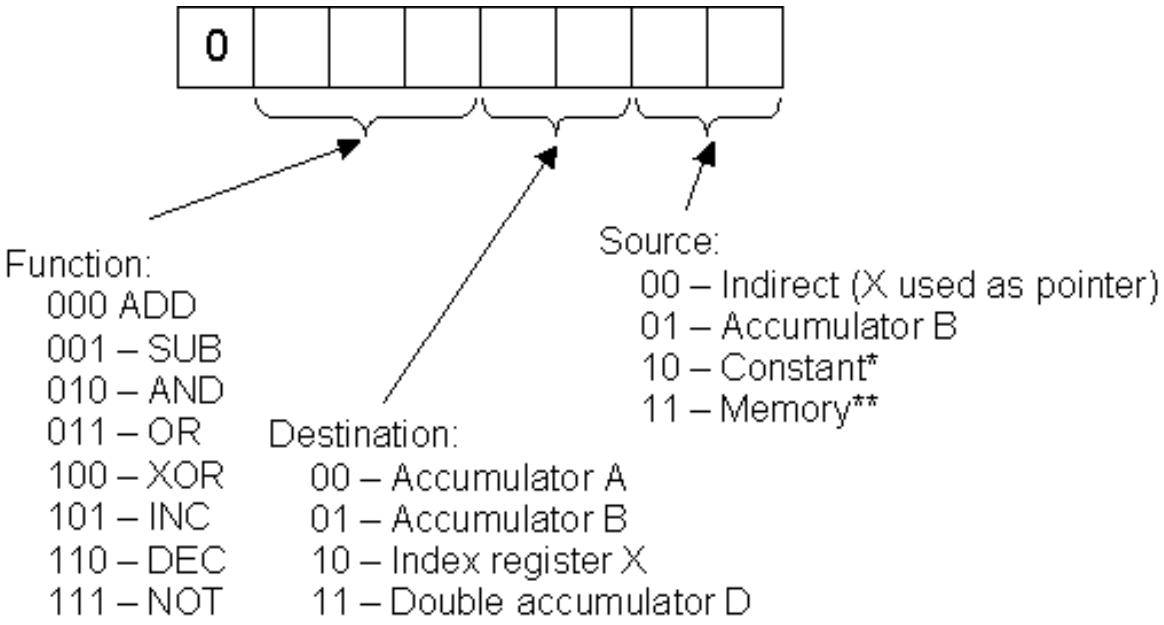
1. Fetch: The CPU goes to fetch the next instruction and data from the RAM (Stored in OPCODE).
2. Decode: The CU decodes the OPCODE into instructions the ALU can understand.
3. Execute: The ALU executes the desired action.
4. Store: Stores the outputted data back into the RAM

Now comes the question: What is OPCODE?

OPCODE

OPCODE is the most basic, low-down level of programming. Each instruction consists of (in the case of 8-bit computers) only 8 bits of information (more for more capable computers).

This is how a standard 8-bit OPCODE instruction is structured.



A line of OPCODE contains 1 word of OPCODE as well as 1 word of OPERAND. The OPERAND is where all the addresses are stored for A: the address of where to store the result, B: the address of the 1st argument of the operation and C: the address of the 2nd argument of the operation.

CONTROL UNIT (CU)

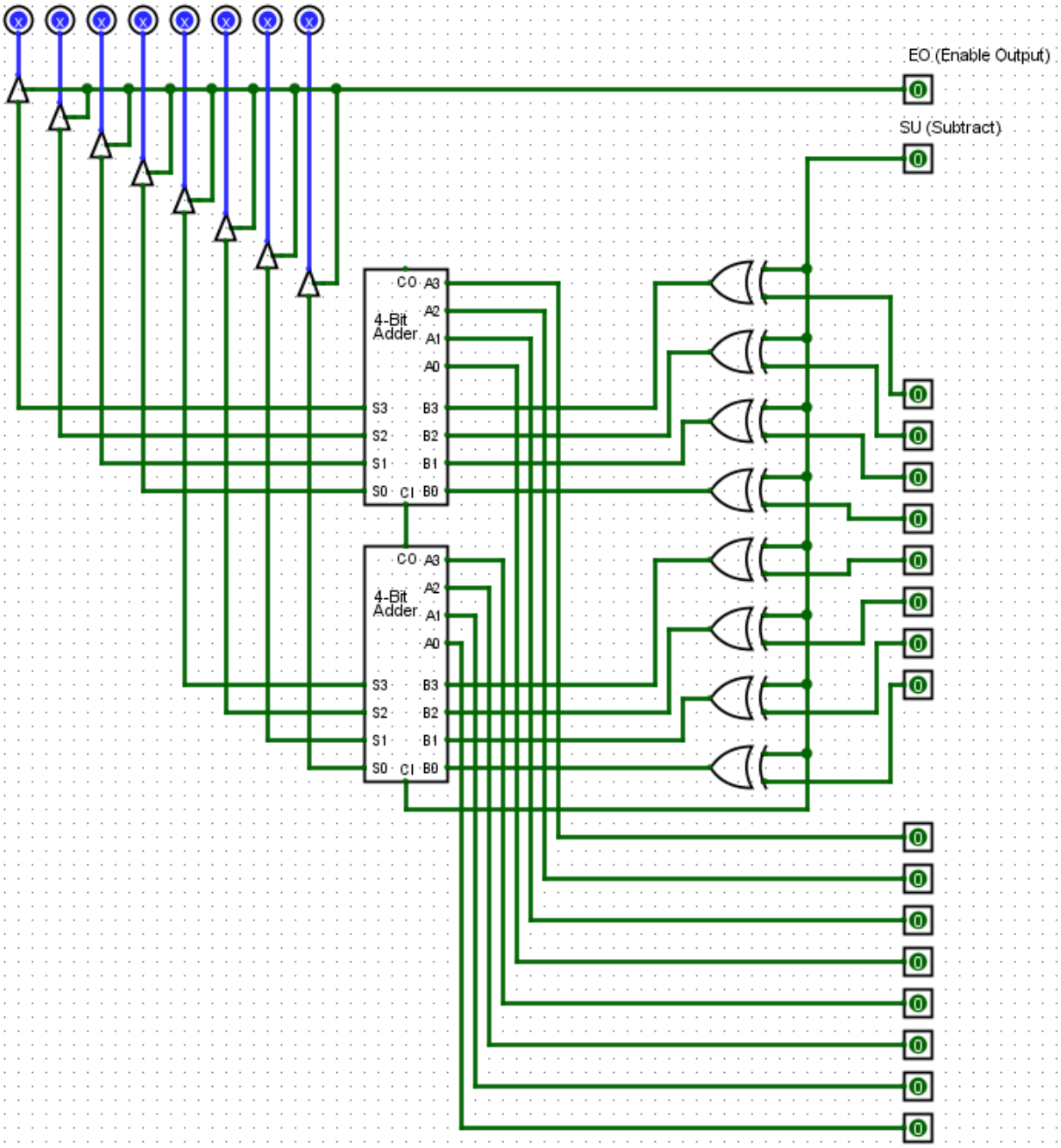
The Control unit is responsible for the 1st, 2nd and 4th phases. It serves as a sort of gateway between the rest of the computer and the CPU. It is responsible for taking the OPCODE from the RAM and decoding it for the other components.

This process is extremely extensive and will not be showcased in this TraPe, as it could alone constitute a whole

ARITHMETIC AND LOGIC UNIT (ALU)

The ALU is the core of the processor. It does every arithmetic and logical Calculation on the computer, getting orders from the CU and sending the result back to it.

Here is a boiled down 8-bit ALU. Keep in mind, 8-bit Computers do not have multiplication or division functions integrated in the OPCODE, and as such neither does the ALU. In more modern Computers, the ALU also contains a multiplier and divider. Multiplication is of course still possible, but it takes much more processing power.



GPU

Modern GPUs are renowned for their immense processing power and are as such used for crypto mining and other processor-heavy tasks, acting as a powered-up CPU. This however was not always the case, GPUs used to only serve to turn data from the CPU into data the screen can understand.

Nowadays, we mostly use HDMI cables, but let us look at how older GPUS handled VGA cables.

VGA cables use 15 Pins. Most these are not needed for displaying images. Many pins are used for communication of information such as the resolution of the screen and are as such optional. As such, we only need 5 of these pins (pins 1, 2, 3, 13 and 14).

Name	Dir	Description
1	RED	Red Video (75 ohm, 0.7 V p-p)
2	GREEN	Green Video (75 ohm, 0.7 V p-p)
3	BLUE	Blue Video (75 ohm, 0.7 V p-p)
4	RES	Reserved
5	GND	Ground
6	RGND	Red Ground
7	GGND	Green Ground
8	BGND	Blue Ground

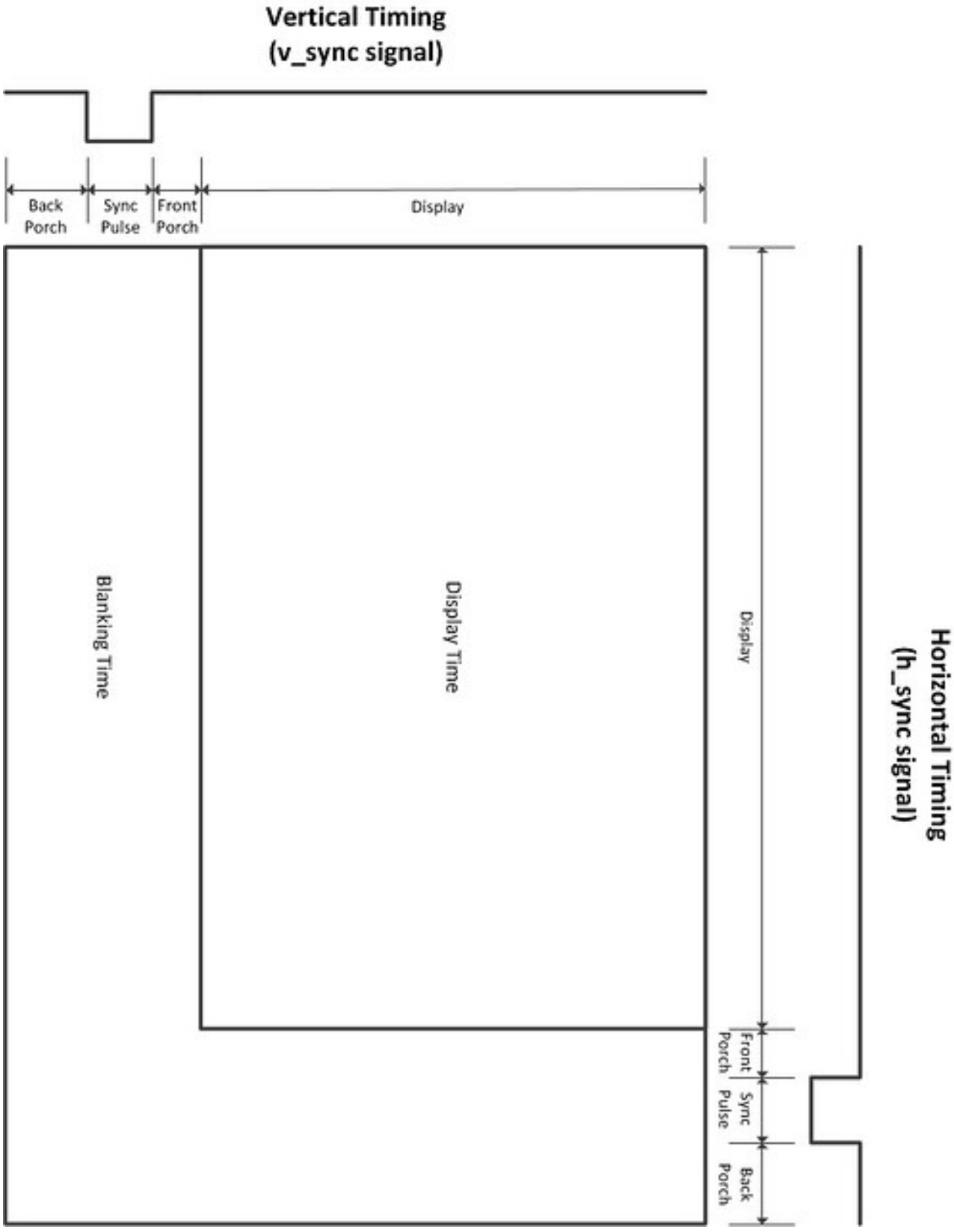
9	+5V	+5 VDC
10	SGND	Sync Ground
11	ID0	Monitor ID Bit 0 (optional)
12	SDA	DDC Serial Data Line
13	HSYNC or CSYNC	Horizontal Sync (or Composite Sync)
14	VSYNC	Vertical Sync
15	SCL	DDC Data Clock Line

Monitors, when using VGA Inputs draw the image starting from the top left corner and working its way down, left to right and top to bottom. The timing needed to write these are the same as they were with CRT monitors, around 100 years ago (first commercialised in 1922).
CRT monitors used to work by using an electron gun, with electromagnets aiming the electrons. The timing of that beam created a baseline for timing, which is still used with VGA cables.

As the beam took time to move from the right to the left of the monitor, the original inventors had to leave a certain amount of time after each row of pixels that was drawn on the screen, as well as a buffer on the left and right of the screen (Front and Back Porch). For the screen to understand when the beam of electrons should be moved back to the left or top of the screen, we use the VSYNC (Vertical Sync) and HSYNC (Horizontal Sync) pins. This timing was maintained for VGA signals, even though there was no good reason anymore to do so apart from backwards compatibility.

What is important here, is that for each resolution, the timing must be correct, so it does not try to write data outside the screen or offset it in any way.

All the timings are conveniently listed online here:
http://www.tinyvga.com/vga-timing


Vertical Timing
(v_sync signal)
Back Porch
Sync Pulse
Front Porch
Display
Blanking Time
Display Time
Horizontal Timing
(h_sync signal)
Display
Front Porch
Sync Pulse
Back Porch

Here are the timings we will be using:

SVGA Signal 800 x 600 @ 60 Hz timing

General timing

Screen refresh rate	60 Hz
Vertical refresh	37.878787878788 kHz
Pixel freq.	40.0 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is positive.

Scanline part	Pixels	Time [µs]
Visible area	800	20
Front porch	40	1
Sync pulse	128	3.2
Back porch	88	2.2
Whole line	1056	26.4

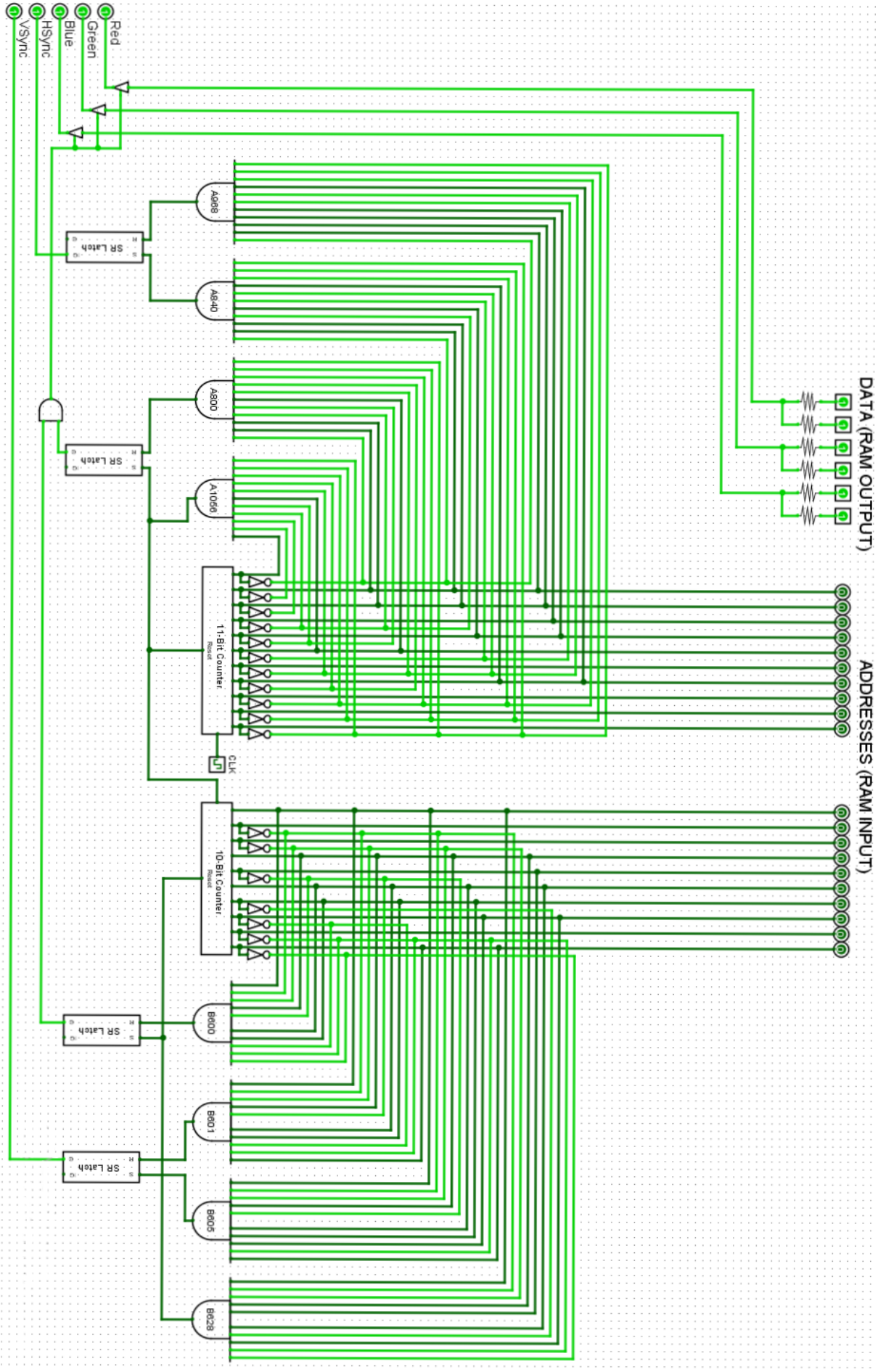
Vertical timing (frame)

Polarity of vertical sync pulse is positive.

Frame part	Lines	Time [ms]
Visible area	600	15.84
Front porch	1	0.0264
Sync pulse	4	0.1056
Back porch	23	0.6072
Whole frame	628	16.5792

We will be using a 40 MHz Clock hooked up to an 11-bit binary counter (11 bits is the minimum to be able to reach 1056, which is the size of one line). We will use AND gates to make a short pulse every time a certain threshold is crossed (E.G: 800 pixels to know when to stop drawing). By hooking these up to SR-Latches, we can define when the drawing zone begins and ends (0-800) as well as when the HSync should be 1 (840-968) and when it should reset to 0 (1056). This is what's happening on the left side of the circuit. We can copy a similar thing to the right, except we change the timing.
By feeding the count to a RAM or other storage device as an address, every pixel would have its own address which can be accessed when needed by the GPU. We only enable the RAM's Output when we are both in the vertical and horizontal display time. Trying to display pixels when not in display time could damage CRT monitors, but newer monitors just stop trying to display anything instead.

This is what a complete (yet very bad) GPU looks like:

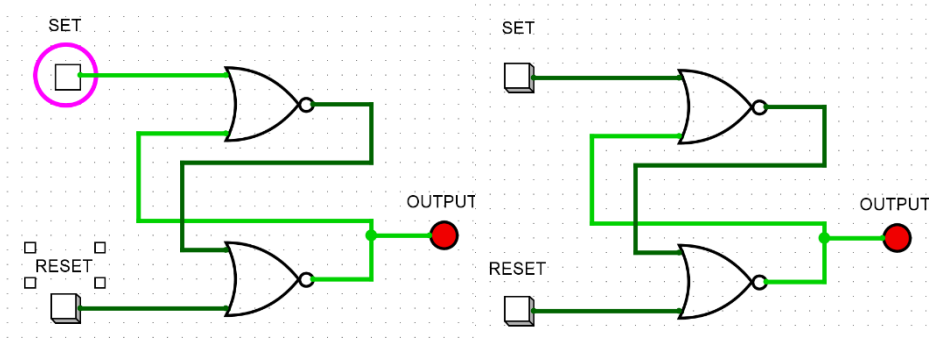

VSync
HSync
Blue
Green
Red
A968
A840
A800
A1056
SR Latch
11-Bit Counter
Reset
CLK
10-Bit Counter
Reset
B600
B601
B605
B628
DATA (RAM OUTPUT)
ADDRESSES (RAM INPUT)

Explanations of specific circuits

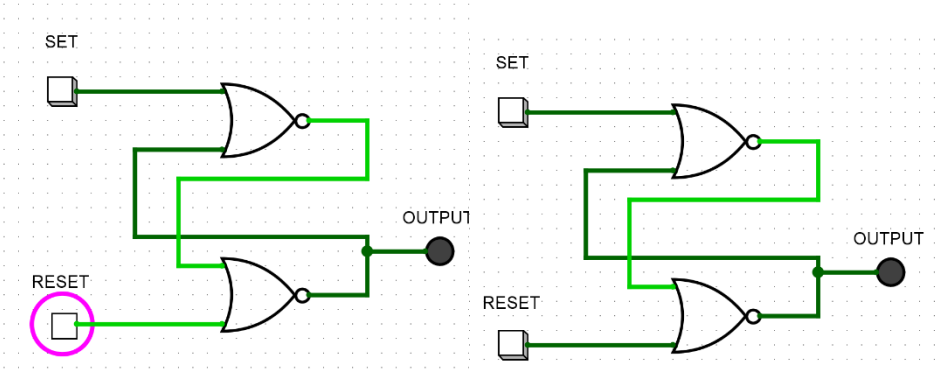
Data Storage

SR-Latch:

A pulse on ”SET” sets the output to 1 and a pulse on ”RESET” sets it to 0. The Output stays until another pulse comes in.



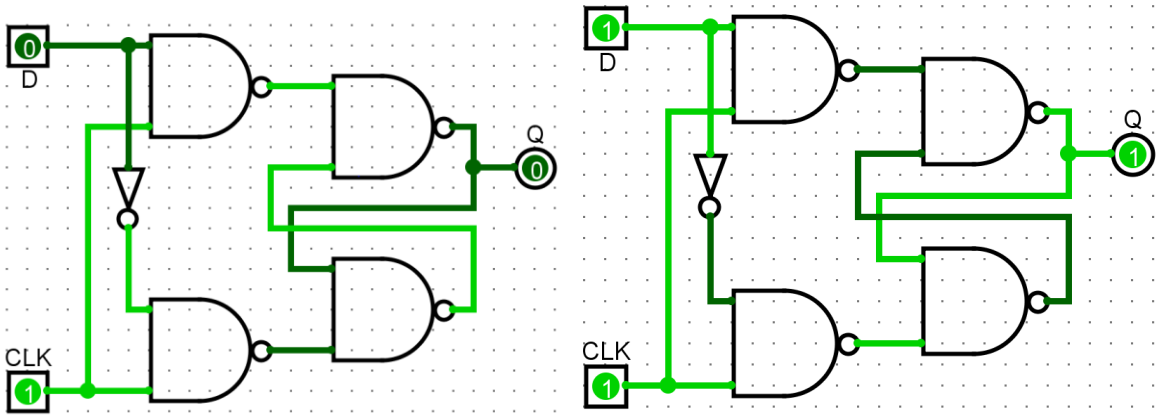
SET is pressed -> OUTPUT is on SET is released -> OUTPUT stays on



RESET is pressed -> OUTPUT is off RESET is released -> OUTPUT stays off

D-Flip Flop:

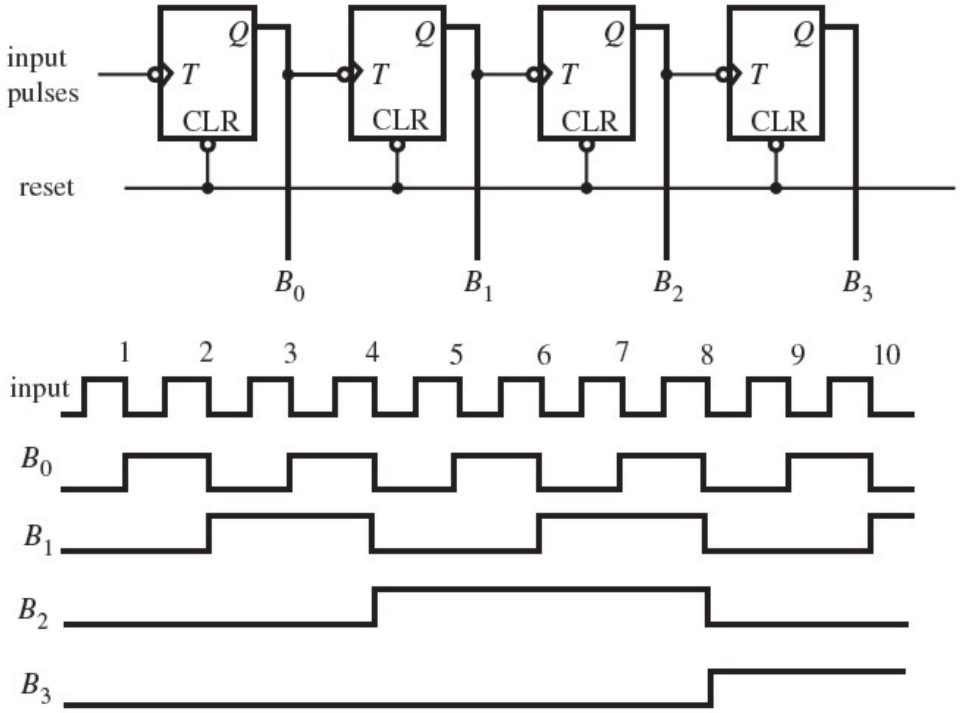
The D-Flip Flop stores the Data input (D) if CLK is on, and outputs the stored state to the Output (Q).



Inputs			Outputs
D (Data)	CLK (Clock)		Q
1	1		1
0	1		0
1	0		Q (Previous output)
0	0		Q (Previous output)

Binary counter

Binary counters serve the simple purpose of adding 1 to a number each time it receives a pulse. This could be done with a binary adder but would be extremely inefficient and expensive. Instead, we count using this:

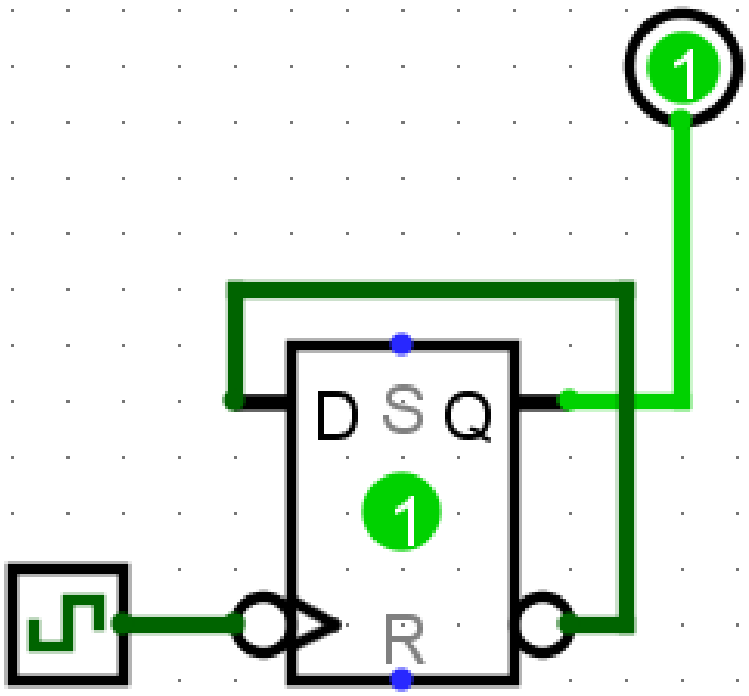


Every time the Input goes from 1 to 0, B0 toggles, every time B0 goes from 1 to 0, B1 toggles, and so on.

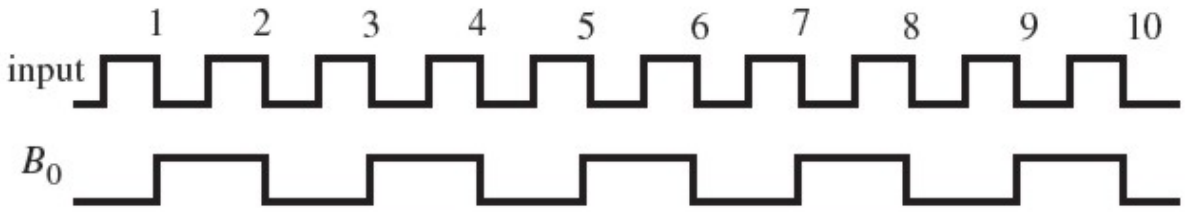
Notice how each frequency is exactly half of the previous frequency. The circuits responsible for this are called Frequency dividers. By chaining these up, we have a circuit that counts from 0000 to 1111, adding 1 every time the clock ticks.

Frequency Divider

Frequency Dividers can be made with any flip flop latch (we will use a D-Flip flop). It is Important that the CLK Input is **falling edge.** Logisim Evolution does not allow you to make your own falling edge D-Flip Flop from scratch, so we will be using the built-in circuit.



By hooking up !Q to D, we set Q to !Q every time the clock switches from ON to OFF, effectively outputting the Input frequency divided by 2 as the output.



Arithmetic Operations

Adder

Adders add 2 binary numbers. Here is how computers add numbers:

Input A	1	0	1	1	+		0	1	1	1	+
Input B	0	0	1	0	=		0	0	1	1	=
Carry		1					1	1	1		
Output	1	1	0	1			1	0	1	0	

The computer calculates the numbers from the right to the left, as we do too.

For each outputted number:

If there is only 1 positive input (Input A, B and Carry), the output is 1.

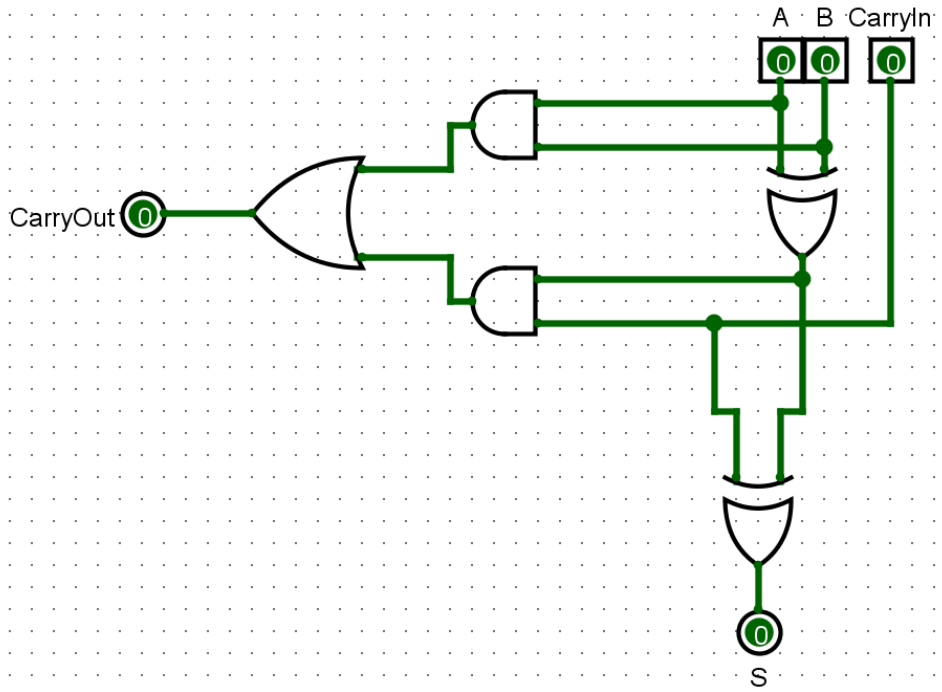
If there are 2 positive inputs, the output is 0 and we carry a 1 to the next digit.

If there are 3 positive inputs, the output is 1 and we carry a 1 to the next digit.

Using these rules, we can deduce:

The output is 1 if there is an odd number of inputs. (Otherwise, it is 0)

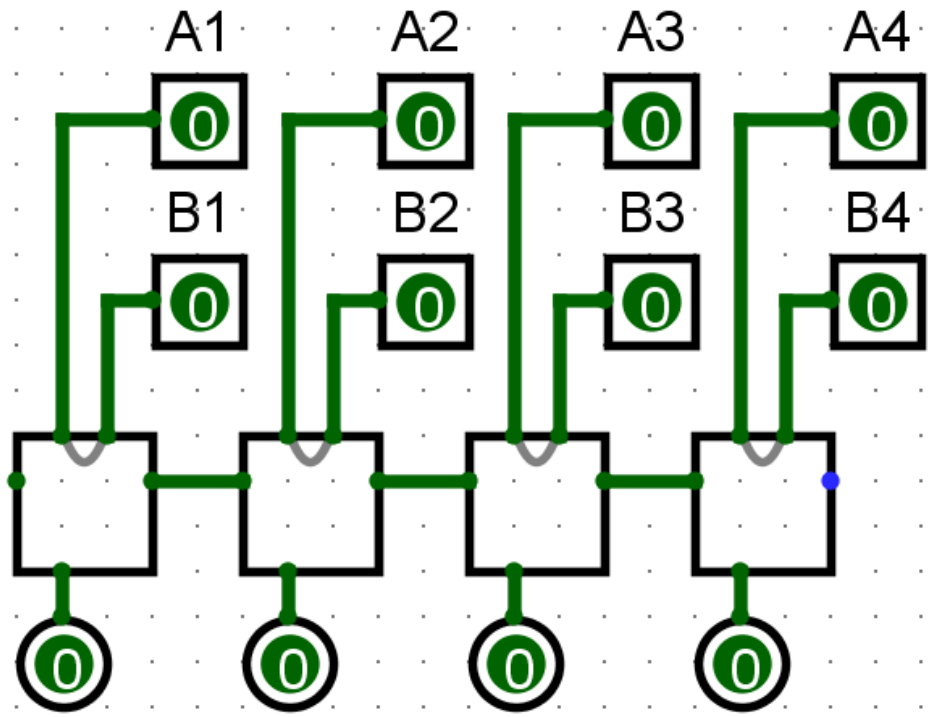
The carry is 1 if there are more than 2 positive inputs. (Otherwise, it is 0)



Here is the truth table for this design (Known as the 1-bit full adder)

Inputs				Outputs	
A	B	Carry In		Carry Out	Sum
0	0	0		0	0
0	0	1		0	1
0	1	0		0	1
0	1	1		1	0
1	0	0		0	1
1	0	1		1	0
1	1	0		1	0
1	1	1		1	1

When arranged so that the Carry Out connects to the next 1-bit full adder's Carry In, it looks like this:

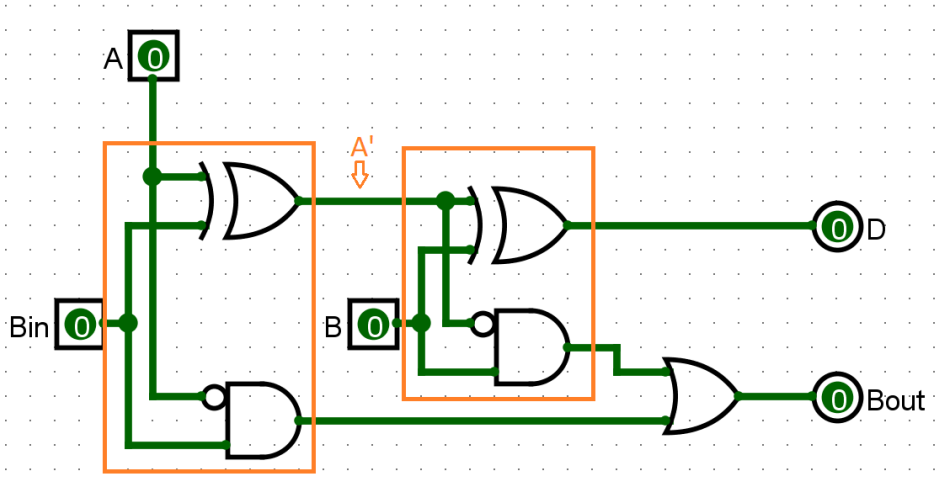


This design can be expanded infinitely relatively easily if needed.

Subtracter (Specialized)

You can use a similar method to addition for subtraction

Input A	1	0	1	1	-		1	1	1	0	-
Input B	0	0	1	0	=		0	0_1	1_1	1	=
Borrow											
Output	1	0	0	1			1	0	1	1	



The 1-bit full subtractor consists of 2 partial subtractors (Orange zones), which each subtract just one pair of numbers (A and Bin, and A' and B).

Here is the truth table for a partial subtractor:

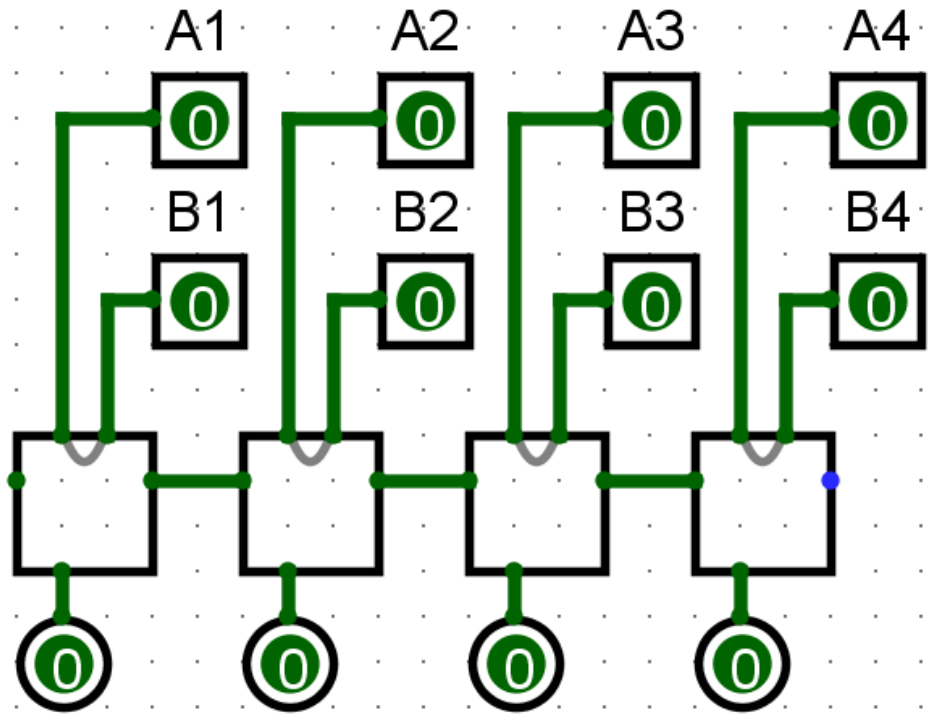
Inputs			Outputs	
Minuend (A)	Subtrahend (B)		Difference (D)	Borrow out (Bout)
0	0		0	0
0	1		1	1
1	0		1	0
1	1		0	0

The 1st partial subtractor subtracts Bin from A (A-Bin) and the 2nd one subtracts the B from output of the 1st one ([A-Bin]-B). The truth table for the 1-bit full subtractor is as follows:

Inputs				Outputs	
A	B	Borrow In (Bin)		Borrow out (Bout)	Difference (D)

0	0	0		0	0
0	0	1		1	1
0	1	0		1	1
0	1	1		1	0
1	0	0		0	1
1	0	1		0	0
1	1	0		0	0
1	1	1		1	1

When arranged so that the Borrow Out connects to the next 1-bit full subtractor's Borrow In, it looks like this:



Much like the adder, this one can also be expanded infinitely relatively easily

Subtractor (Generalized)

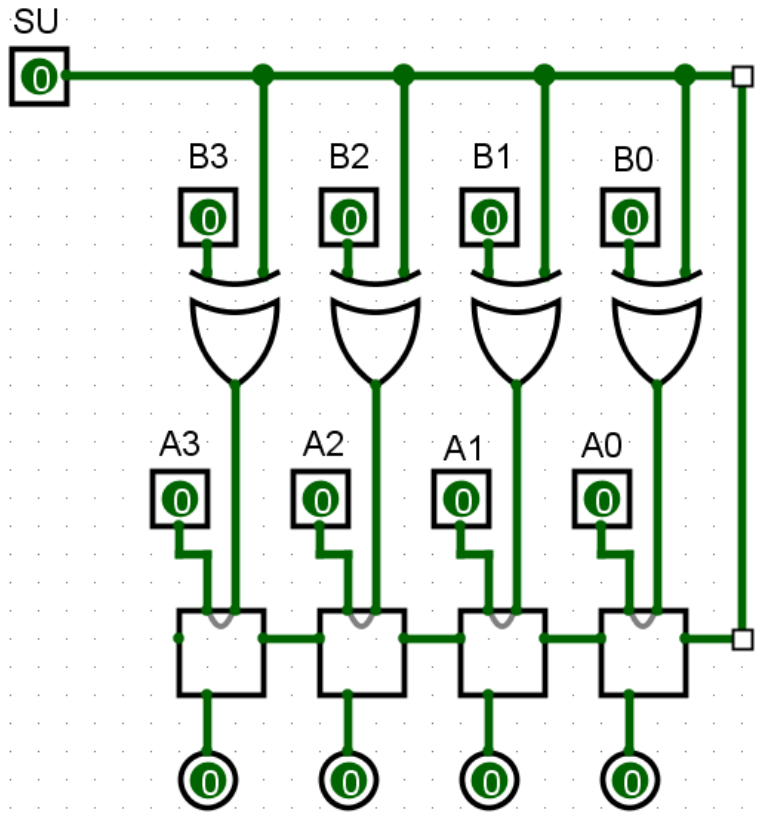
Binary subtraction can also be fulfilled in an indirect way by using an adder instead of making a whole new circuit. It works a bit like this:

Let us say A = 1011 and B = 0010

									1	+	
Input A	1	0	1	0	-	Input A		1	0	1	0	+
Input B	0	0	0	1	=	Input $\overline{B}$		1	1	1	0	=
Borrow			1			carry		1				
Output	1	0	0	1		Output	1	1	0	0	1	

As you can see, by inverting Input B, using a 4-bit *adder,* adding 1 and ignoring the overflow, we get the same result as if we had used a subtractor.

This is useful so we do not have to use 2 separate circuits for addition and subtraction.
By adding a SU toggle to our adder, we get an adder/subtracter circuit.



When SU (subtract mode) = 1, Input B is inversed, and the carry in of the 1st 1-bit full adder is set to 1, effectively adding 1 to the whole thing.

Multiplier

Broken down to its main components, for

Decimal Multiplication looks like this:

Input A			2	3	*
Input B			9	5	=
Partial Product 1			27	15	+
Partial Product 2		18	10		=
Output	2	1	8	5	

We can apply the same principle

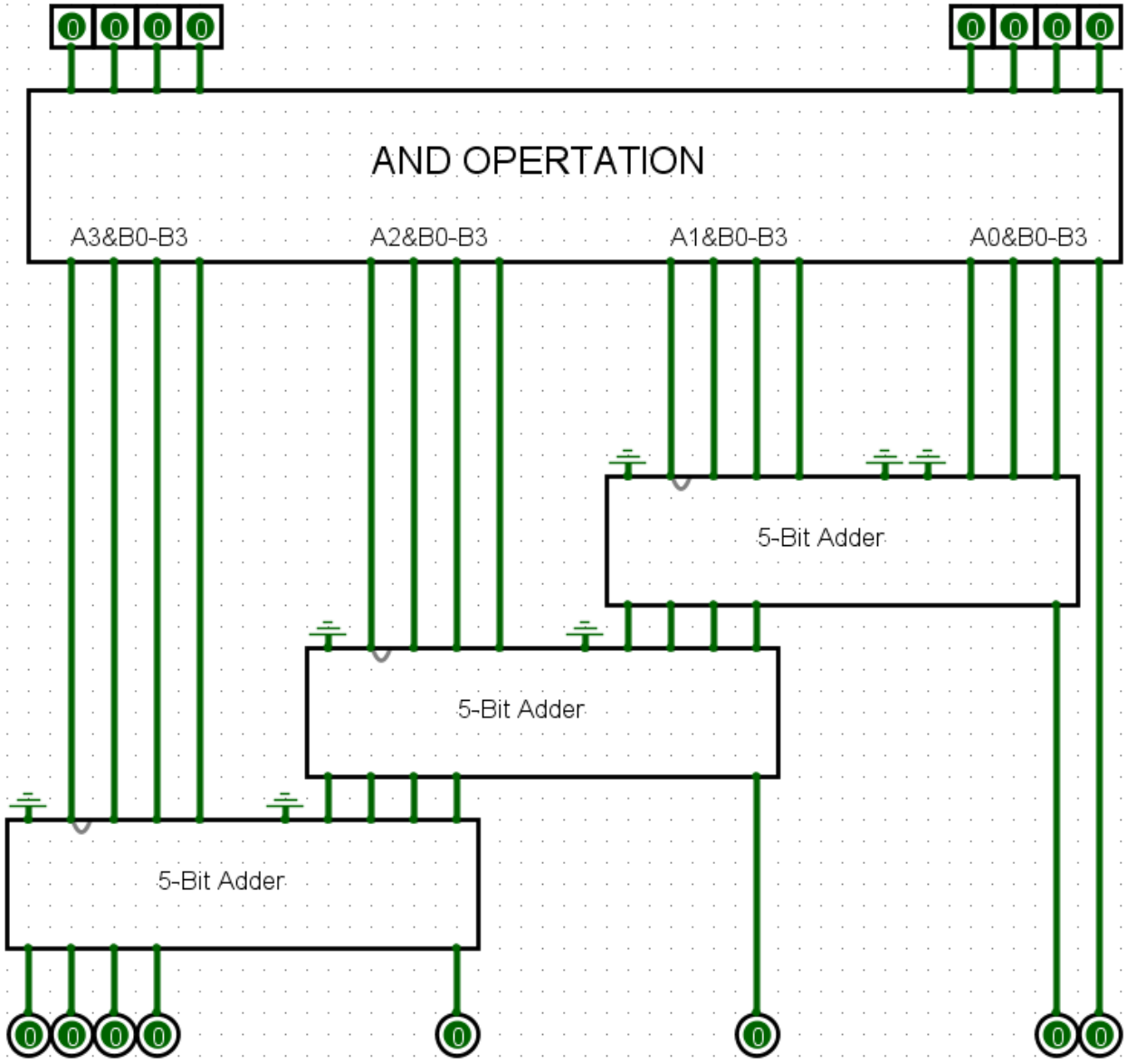
Binary Multiplication:

Input A			1	0	0	1	*
Input B			1	0	1	1	=
Partial Product 1			1	0	0	1	+
Partial Product 2		1	0	0	1		+
Partial Product 3	0	0	0	0			+
Partial Product 4	1	0	0				=
Output	1	0	1	1	0	1	1

If we look at it from a computer logic standpoint, it would be this:

Input A				A3	A2	A1	A0 *
Input B				B3	B2	B1	B0 =
Partial Product 1	0	0	0	A0*B3	A0*B2	A0*B1	A0*B0 +
Partial Product 2	0	0	A1*B3	A1*B2	A1*B1	A1*B0	0 +
Partial Product 3	0	A2*B3	A2*B2	A2*B1	A2*B0	0	0 +
Partial Product 4	A3*B3	A3*B2	A3*B1	A3*B0	0	0	0 =
Output	Partial Product 1 + Partial Product 2 + Partial product 3 + Partial Product 4						

If we compile this to a diagram:



Note, that we use 5-bit Adders to account for overflow, so calculations of higher numbers do not mess up. You could also instead use 4-bit Adders and connect the Cout to where the 5-bit adder's last output goes.

Sources:

Digital Computer Electronics (Albert Paul Malvino)

https://eater.net

https://eater.net/VGA

https://eater.net/8bit/

https://eater.net/8bit/alu

http://www.tinyvga.com/vga-timing/800x600@60Hz

http://www.tinyvga.com/vga-timing

https://pinoutguide.com/Video/VGAVesaDdc_pinout.shtml

Original Research (Trial and error with Logisim Evolution) (Source of almost every diagram)