Université Grenoble Alpes

POLYTECH GRENOBLE

RETROCOMPUTING

Coupling of the Digital software with an MC6809 microprocessor emulator programmed in Python.

January – April 2021

Sami ELHADJI TCHIAMBOU

Corentin HUMBERT

Mathis MUTEL

Olivier RICHARD

I. Table of content

II. Acknowledgements

First, we would like to thank our teacher Mr. Olivier RICHARD for giving us the opportunity to work on this project. Throughout this project, we were able to use knowledge acquired from our first year at Polytech when we studied computer hardware.

We would also like to address our thanks to Helmut Neeman, creator of Digital and Jens Diemer, creator of the MC6809 simulator for helping us during the realization of this project by answering our questions very rapidly.

III. Introduction

We are a team of three fourth-year students in computer science at Polytech Grenoble. As part of our fourth year, we had to choose a project to work on.

The purpose of this document is to showcase the work done on our Retrocomputing project that lasted for three months starting in January 2021 and ending in March 2021. This project consisted in coupling two existing technologies: The Digital software created by Helmut Neeman and the MC6809 microprocessor emulator by Jens Diemer. Both technologies are licensed under the GNU General Public License v3.0 and are freely available on GitHub.

Throughout this document, we will talk about how both technologies work and explain how we managed to couple them together during our project.

More information about this project can be consulted on the official Wiki AIR page with the following link: https://air.imag.fr/index.php/Retrocompute_simulateur

IV. Project context

1. Let us talk about Retrocomputing

Figure 1: The 1977 Apple II computer

Our project is about Retrocomputing. It is important that we first define what Retrocomputing about. Retrocomputing is an activity that consists in the use of old computer hardware and software in modern times. Because of how fast technology is evolving, computers that were made decades ago have been clearly outperformed by nowadays computers. That is why we are “easily” able to re-create older computers with our current computing power. By “easily” we mean in terms of hardware (CPU, memory, etc.), because even computers made decades ago were so cleverly put together that trying to re-create them from scratch can prove to be very challenging. And that is essentially what Retrocomputing is about.

Retrocomputing mainly exists for educational purposes. There is no reason to re-create older machines in terms of practical usage (since it is essentially slower than the machine it runs on) but it can be quite satisfying to achieve building one from scratch and this allows us to better understand how computers work in the first place. Because, while hardware and software have both drastically evolved throughout the years, nowadays computers still work the same as older ones. For those who are old enough to have possessed some of these old machines at one point, Retrocomputing brings a certain nostalgia and allows them to relive long forgotten experiences using today’s technologies.

2. What are we trying to achieve exactly?

The goal of this project is simple in appearance: Coupling an emulator of an old 8-bit microprocessor (MC6809) with a software that allows to design electrical circuits (Digital).

Since both software have been developed outside the scope of this project, we had to tweak them a bit to achieve our goal. For the MC6809 emulator, we set up a server and adapted the working environment of the processor. For the Digital software, we designed a custom component that can connect to the MC6809 emulator and perform various operations on the emulated CPU.

Without further ado, let us talk more about the MC6809 emulator and the Digital software.

V. Technologies

1. MC6809 emulator

The MC6809 CPU emulator is an Open-source emulator for the 6809 CPU. The Motorola microprocessor 6809 or commonly called MC6809 is an 8-bit microprocessor used in old home computer such as Dragon32/64 and Tandy TRS-80 Color Computer (CoCo) built in the 1980s. The emulator is entirely coded in Python (version 3.6+) and aims to re-create and run a fully functional virtual version of the MC6809 on a personal computer. More in-depth information about the MC6809 can be found here.

Figure 2: Motorola microprocessor 6809

2. Digital software

Digital is a digital logic designer and circuit simulator designed for educational purposes. This software has many functionalities and allows to design anything from simple logic gates to more complex logic structures such as memories, ALU or even actual computers. You can get a glimpse at the GUI of the Digital software as well as example circuits in the image below:

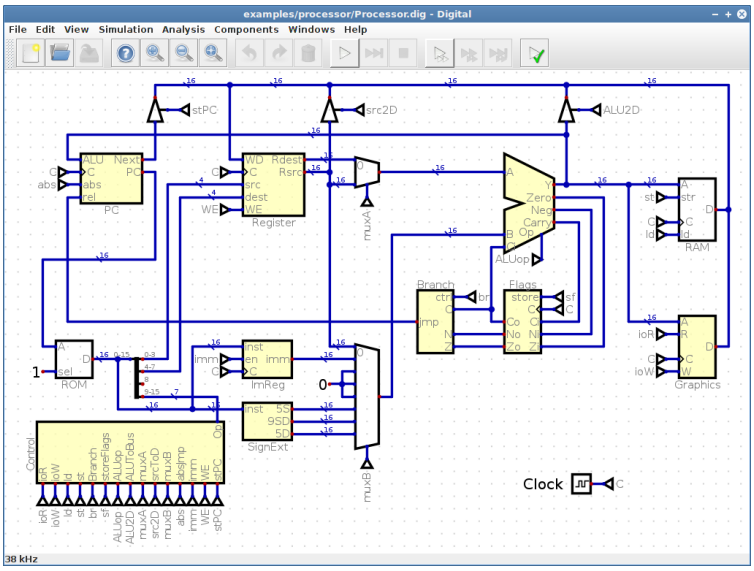


Figure 3: Window of a custom digital circuit

Digital is written using the Java programming language and because of how modular it is, it offers the possibility to create our own custom components in Java. We will use this asset to create our own component which will interact with the MC6809. Digital also provides a set of built-in components such as memory components. We will use these components when designing a circuit that uses our custom component to see the coupling with the MC6809 emulator in effect.

VI. Project realization

The project is divided into three parts: The implementation of a simple server application that waits for clients to connect and runs an emulation of the MC6809 for each client, the Digital client that takes the form of a custom Digital component that can connect to a server and the communication channel which allows the client and the server to communicate. We will treat these three parts in the stated order. Finally, we will end by showcasing a hand-made circuit in Digital that allows us to execute code on the MC6809 emulator.

1. Modifications on the MC6809 emulator

The execution of emulations of the MC6809 processor is done using several Python classes with each class having been designed for a specific aspect of the emulation. We will see all these classes in this part.

a. A server to handle connections

The server is implemented in a file called server.py. The code for the server itself is short and only focus on initiating a server that waits for clients to connect. The server creates a socket on port 6809 and listens for clients. Once a client connects to the listening socket, the server starts a new thread for the client to control an instance of the 6809 processor. This means that each client has its own environment where it can execute an emulation of the MC6809.

b. Remote instances of the CPU

As we said in the previous part, once a client connects to the server, a new thread is started. This thread creates and runs an instance of the DigitalCore class. This class runs an instance of the MC6809 CPU on the TCP socket. To be able to run an instance of the 6809 on a TCP socket, we first had to take our time and study how the processor was implemented in Python. Upon studying the implementation, we learned that when created, the CPU had its own array of bytes in which machine code as well as data was stored. In terms of mechanical components, the memory of the processor is a separate component operating outside of the MC6809 chip. This meant that we could not use the native memory used in the MC6809 emulator.

That is why we created our own memory class named DigitalMemory. The code of this class is essentially the same as the one found in the code file for the original Memory class used by the emulator. The only things that change are the four following functions: **read_byte**, **read_word**, **write_byte** and **write_word**. These functions directly interact with the byte array in which data and machine code is stored. Instead of simply retrieving bytes from the array and writing them to the array, the DigitalMemory calls two methods from the DigitalCore depending on the operation to

perform. If we want to read a byte or a word, the method **await_read_byte** will be called. If we want to write a byte or a word, the method **await_write_byte** will be called.

Reading a byte from Digital (await_read_byte)

To read a byte from the memory stored in the Digital client, the processor must first send a message through the TCP channel telling the Digital client at which address it wants to read. Once the message is sent, the processor will then wait for an incoming message containing the byte/word stored at the sent address. It will then proceed to the next instruction by advancing its program counter.

Writing a byte to Digital (await_write_byte)

To write a byte to the memory stored in the Digital client, we must simply send a message to the Digital client containing both the data we want to write and the address where it must be written at. The processor does not wait for any feedback from the Digital client, so once the processor executes a write operation on the Digital memory, it will assume that the operation succeeded and will continue its execution by incrementing its program counter.

Note: The entire process regarding the messages sent over the TCP socket will be detailed in a later section where we talk about the TCP socket. For now, we will assume that the messages travelling over the TCP socket contain all the information the processor needs.

Now that we have covered most of the changes that were made on the Python emulator, it is time to talk about how we implemented a custom component in Digital.

2. Modifications on Digital

The custom component that we implemented in Digital is a coupler component that can interact with the MC6809 through a TCP socket. It is equipped with forty pins, each corresponding to a specific pin in the original MC6809 chip (An image showing the pin arrangement on the actual chip can be found in the Annex). For example, the "!Reset" pin is used to reset the processor. If it is set to high voltage, the processor will be unable to increment its program counter and will remain in a frozen state until the pin is set back to low voltage.

On top of that, the component also has an input pin corresponding to the clock that the original MC6809 chip does not have. Instead, the real chip has two output pins named "Q" and "E" that monitor the cycles of instructions. This tweak in the original design allows to make the interaction between Digital and the MC6809 emulator easier. Another design tweak was setting the Vcc pin as an input pin. This pin is mandatory for the communication to work because it allows to turn the component on. In terms of actual logic, turning on the Vcc pin will attempt to connect to a remote MC6809 emulator.

In terms of Java implementation, we have two lists of **ObservableValue** (Java class that corresponds to both input and output pins). The fist list is for the input pins and the second one for the output pins. Since input pins and outputs behave differently, it raises a question: When looking at the datasheet of the MC6809 microprocessor, we can observe that there are eight pins working both as input and outputs. These eight pins correspond to the data bus. So how do we implement bidirectional pins?

a. How we implemented the data bus in Digital

In a CPU, the data bus is often bidirectional. This means that the data bus behaves both as an input and an output bus. This is the case in the MC6809. Since we had very few knowledges of the Java code of Digital at the start, we implemented this bidirectional bus by having the data pins both as input and output pins. Because we were not satisfied with this solution, we decided to ask the question directly to Helmut Neeman, the creator of Digital. He answered to our question very quickly and pointed out what we needed to do to make the bidirectional bus work. To avoid duplicating pins and to make the code more digest, we switched to this new solution. In terms of code, we simply had to set the pins to bidirectional and set them to High Z mode whenever we wanted them to be treated as outputs.

b. How does the custom component communicate with the server?

We mentioned earlier that the Vcc pin was the one responsible for issuing a communication between the server and the client. What we did not talk about is how the component is able to know when it receives messages from the server. This is very simple. Once the Vcc pin is set to high voltage (it must remain that way throughout the entire simulation), it attempts to connect to the server. If the connection succeeds, the client stops and wait until it receives a message from the server.

On the server-side, a thread is created, and the CPU emulation starts. The fist action is performed by the remote MC6809 and it consists in reading the byte at the address corresponding to its program

counter. Since the memory is externalized into our Digital client, upon attempting to read a byte, the MC6809 will send a message through the TCP socket to the Digital client.

Once the message arrives to our client, it will process it and perform actions depending on what was written in the message. For example, if it is a read operation from the MC6809, the client will update the address pins accordingly and send whatever it stored on its data bus to the server. If we connect both the address bus and the data bus to a built-in memory component in Digital, we will be able to store machine code in the memory and send it to the server to advance the simulation.

To make things easier on our end, we implemented a simple automaton to keep track of how the client is doing (see the picture below):

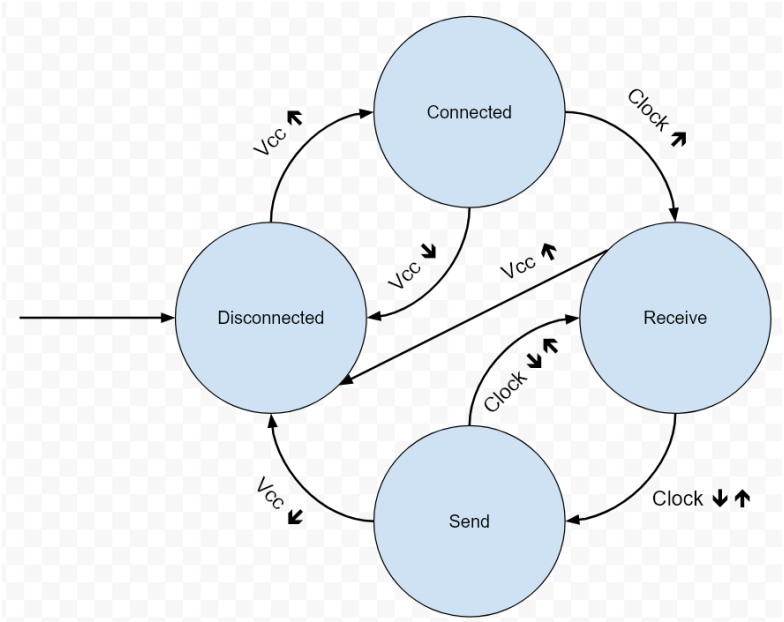


Figure 4: TCP communication automaton of the Digital client

Now that we have talked about how the Digital client exchanged with the server, it is time to see what kind of information they exchange and how it is done.

3. Communication channel

We have been hinting this part for some time and it is finally time to talk about the information that travels through the TCP socket. We took some time thinking about the information we would send over the TCP socket and how we would encode it. In the end, we established the following protocol: When the Digital client is running, we observe the Vcc pin of the custom component. If its state changes to high voltage, the client on Digital sends a connection request to the MC6809 server, which instantiates a new MC6809 CPU.

Once the emulator has been instantiated, a rising edge on the Clock pin will advances the Digital communication automaton. Depending on the current state the automaton is in, it will either send or receive. Regardless of whether it is sending or receiving, the message format remains the same. The

message is a UTF-8 encoded string with a length of exactly forty-two characters. Each character can be either '0' or '1'. The first two bits are control bits used to configure the connection. The Digital client can disconnect from the server by sending a message containing the following control bits: "01". The sending of disconnection messages happens independently from the clock. This can be easily visualized in **figure 4** as we can see that it is a failing edge on the Vcc pin that allows the automaton to jump to the disconnected state. The strength of these control bits is that they will be processed by the server regardless of the current state of the exchange. This means that the client can decide to stop the simulation whenever they will it, even if the CPU is currently trying to interact with the memory. Once a disconnection message is received by the server, the socket will be closed, therefore, ending the simulation.

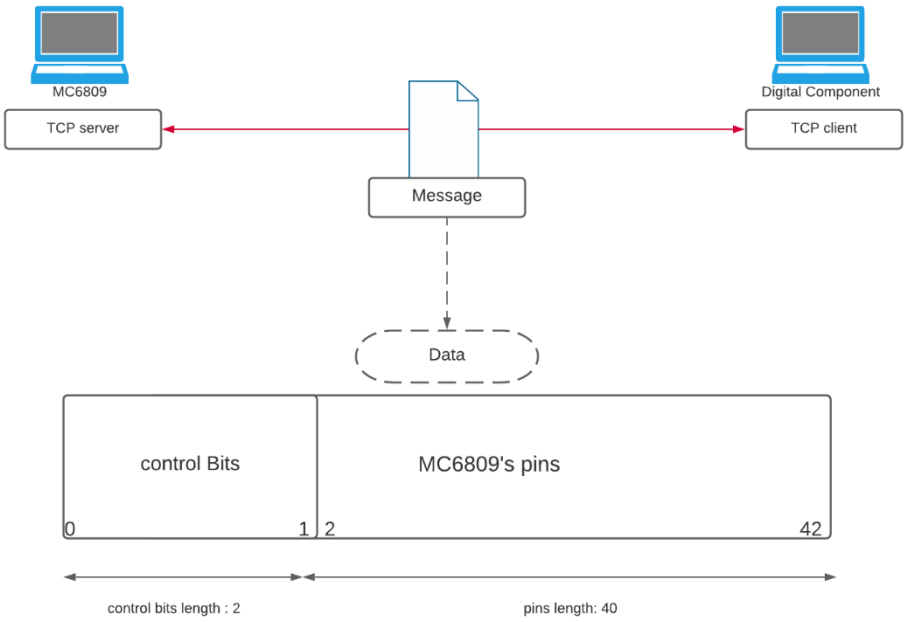


Figure 5: Communication between the client and the server

Note: Despite having the message encoded as an UTF-8 string, we only write sequences of binary values in the message. This means that we are currently sending over 336 bits even though we could have reduced the length to only 42 bits. We decided to go with it because at this scale, the difference does not really matter and using UTF-8 encoded string made the process of treating messages easier.

As for the forty remaining bits, they correspond to the forty pins of the MC6809. Whether it is the Digital client or the MC6809 emulator, the messages will be decoded in a similar way. The only difference is how the data from the message is used. In the MC6809 emulator, the bits received will be processed as either instructions or arguments. In the Digital client, the bits received will be used to set the value of the input pins accordingly.

Note: Since the input Clock pin was not in the original MC6809 architecture, it is not being sent to the MC6809 server. This pin is only meaningful to the Digital client to control its automaton.

When the Vcc pin in Digital is set to low voltage, the client sends a message with the control bits to close the connection. The server then closes the instance linked to this connection, and the client closes the connection. Thus, switching the Vcc pin from high to low voltage and then from low to high voltage will have the effect of resetting the microprocessor.

4. Trying out our implementation

a. Designing a simple circuit

Both the server and the client are operational. Messages can be exchanged between the two of them to advance the simulation. All that is left now is to design a simple circuit to test out our implementation. The designed circuit can be found at Digital/customCircuits/WorkingSimulation.dig and is viewable in the image below:

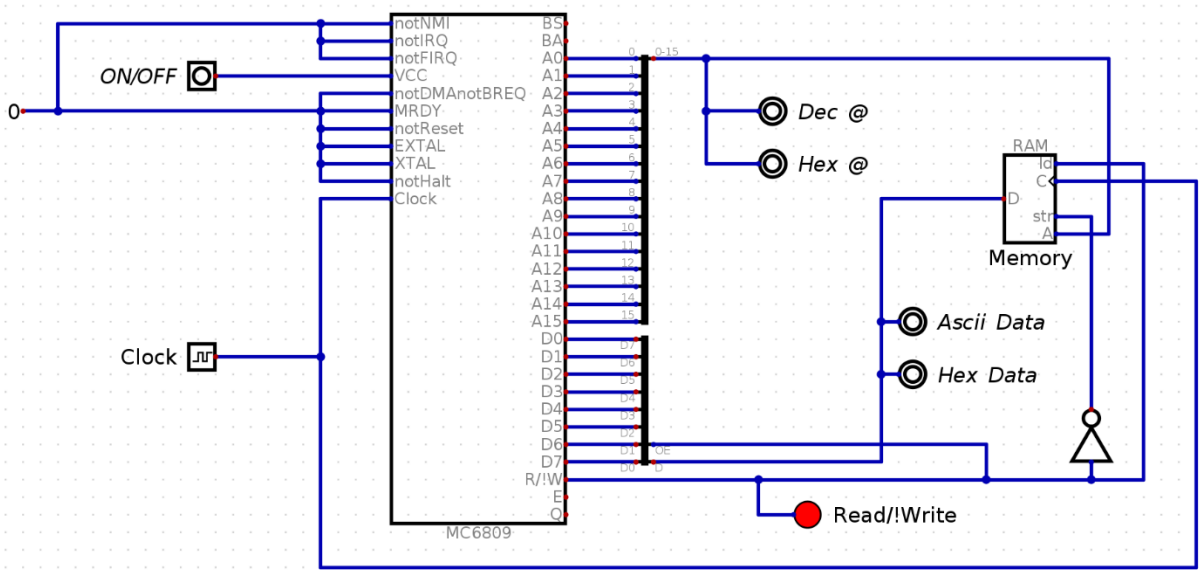


Figure 6: Working circuit that uses the built-in RAM component

You can see our custom component on the left-hand side of the image. This component is the star of the circuit, it the one that communicates with the remote emulator.

You can see several smaller components on the left such as an input that allows to set the voltage of the Vcc pin or a clock input linked to the Digital clock.

On the right-hand side of the image, we can see a more complex layout with all the data and address buses going into a merger component. The merger component is a simple built-in component that simply put several wires together into one bus. Both the address and data bus are connected to an output displaying both their Ascii and Hex value. This allows us to visually see what is happening in the simulation.

Finally, we have a simple RAM memory on the right. The address bus of the MC6809 goes to the address input of the RAM. The data bus of the MC6809 goes into the data input of the RAM. An important thing to note is that the data input of the specific RAM we used is bidirectional. Therefore, the data bus is also bidirectional and that is why we could not use a simple merger component but had to use a bidirectional merger for this specific case.

We can see a pin on the MC6809 that we have not talked about yet. It is the “R/!W”. If set to low voltage, it means that the MC6809 wants to write in the memory, otherwise, it means the processor wants to read from the memory. You can see that the wire coming out of this pin is also inputted in

the RAM component. The only difference from the other inputs is that the “R/!W” wire goes to both the “ld” and “str” input pins of the RAM. In the case of “str”, it is being inverted. The purpose of this is to tell the memory whether we want to write or read. The last input is the clock input which is linked to the clock on the left.

b. Executing a simple program

All that is left to do to test the circuit is to start the simulation on Digital, input machine codes into the RAM and turn on the MC6809 component through the Vcc pin. The remote emulator will then perform operations on the RAM component by exchanging messages on the TCP channel. For you to quickly start experimenting with the coupling, we provide the following machine code:

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107	0x108	0x109
0xF6	0x10	0x00	0xC0	0x30	0xF7	0x10	0x01	0x00	0x00

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	0x1008	0x1009
0x34	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Each array represents a part of the RAM memory which you can easily edit by right clicking on it during run time. The first line (in black) represents the address, and the second line is the value stored at the specific address. The given machine code is explained in the following table:

Machine code	Instruction	Description
0xF6 0x10 0x00	LDB (extended) 0x1000	Loads the value stored at address 0x1000 into the B register.
0xC0 0x30	SUBB (immediate) 0x30	Subtracts 0x30 (48 in decimal) from the B register.
0xF7 0x10 0x01	STB (extended) 0x1001	Stores the value of the B register at the address 0x1001.

The above program is a very basic implementation of the atoi function. This only works with a single digit. In our case we used the hexadecimal value 0x34 which corresponds to the Ascii character ‘4’. Executing the above code will eventually put 0x04 at the address 0x1001.

Note: By default, the emulated processors all start with their program counter value set to 0x100. Therefore, machine instructions must be written starting at the address 0x100 in the Digital RAM component. The default value of the PC is hardcoded in the server but can be easily changed.

VII. Future improvements

A project laid out in the span of three months is very limited in terms of work done but, in the end, we have managed to make a working implementation of the project by coupling the MC6809 emulator with the Digital software. However, as it stands right now, both the client and the servers lack some functionalities. For example, only a few pins from the original MC6809 chip are treated by the Python emulator. The emulator can adapt its execution depending on the values set on the address buses and data buses. It is also able to reset whenever the !Reset pin is set to true. But many pins such as ones used to create interruptions have not been implemented.

The emulator could also demand feedback from the Digital client when trying to perform a write operation in the memory to react to eventual software failures on the Digital client.

Finally, we could try messing with the encoding of the messages sent over the TCP channel. Since we know that only forty-bits of data are needed, we could try and improve the current encoding and see if it has any effect performance-wise.

VIII. Conclusion

We have a working coupling between the MC6809 emulator and the Digital client. We can launch the server and then the client to start a simulation. We created a circuit in Digital using a RAM component along with our custom component to test our implementation. It works and we can use some simple programs to read and write into the RAM.

Overall, this project was both an entertaining and enriching experience. We have had some difficulties along the way, there were times when we felt lost but once we managed to execute an actual program stored in the Digital memory, it was really rewarding. It was only three months, but we all agree in saying that this was a fun project, and we understand why people would try and re-create old computers with nowadays technologies. I mean, if I were to tell you that tomorrow you could be able to play old games from the 1980s by inputting code in a RAM in Digital, wouldn't you want to go back in time for a moment and learn about the computers from the 1980s?

We are proud of the work we have done, and we have all learned lots of things thanks to this project. We feel at ease with the Digital software now. More than teaching us about Retrocomputing, this project also allowed us to use our current knowledge and discover new things.

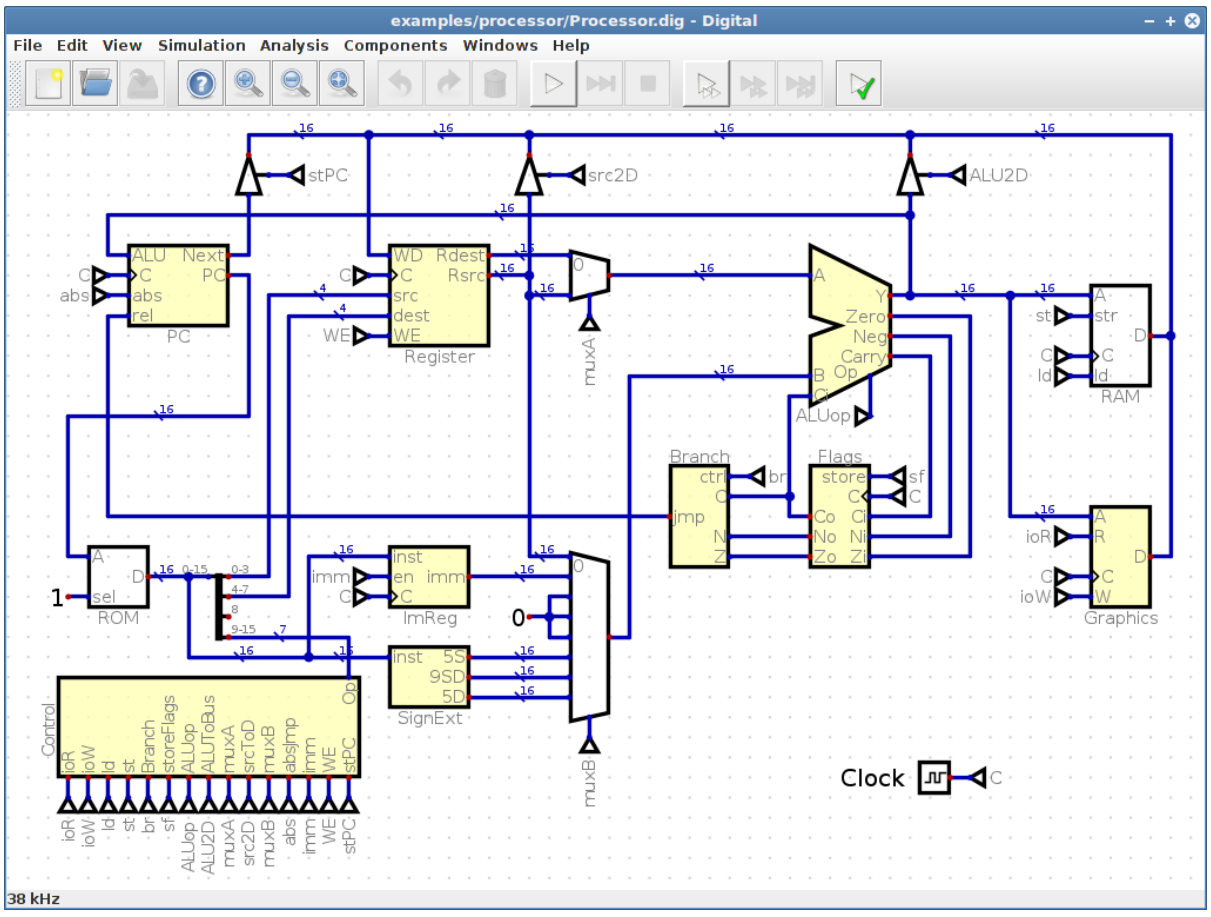
IX. Glossary

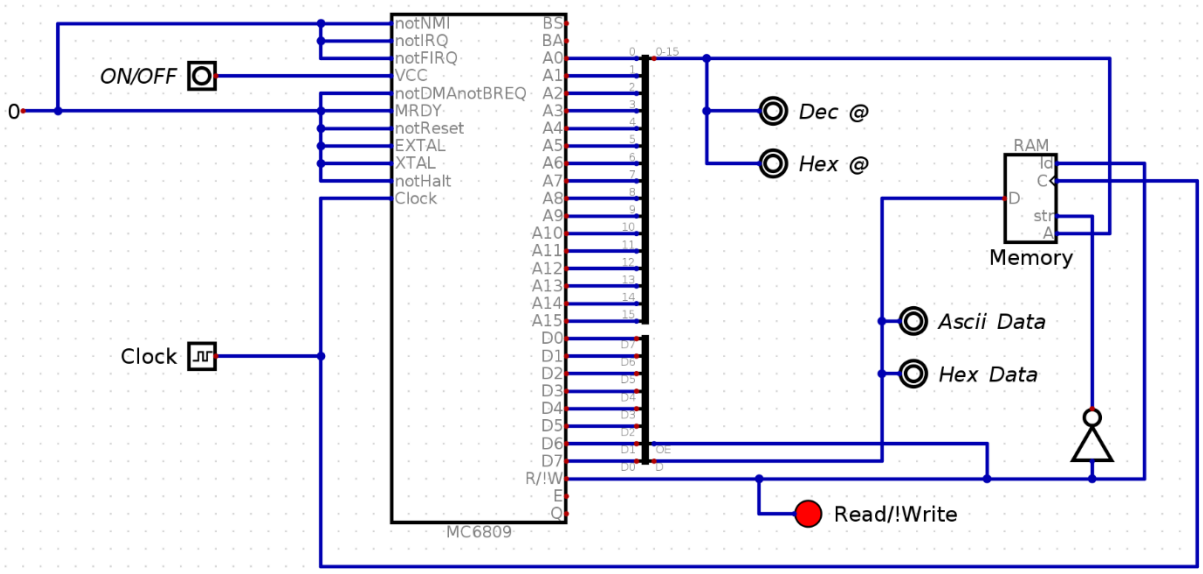
MC6809: The Motorola 6809 is an 8-bit microprocessor CPU used in some computers back in the 1980s.

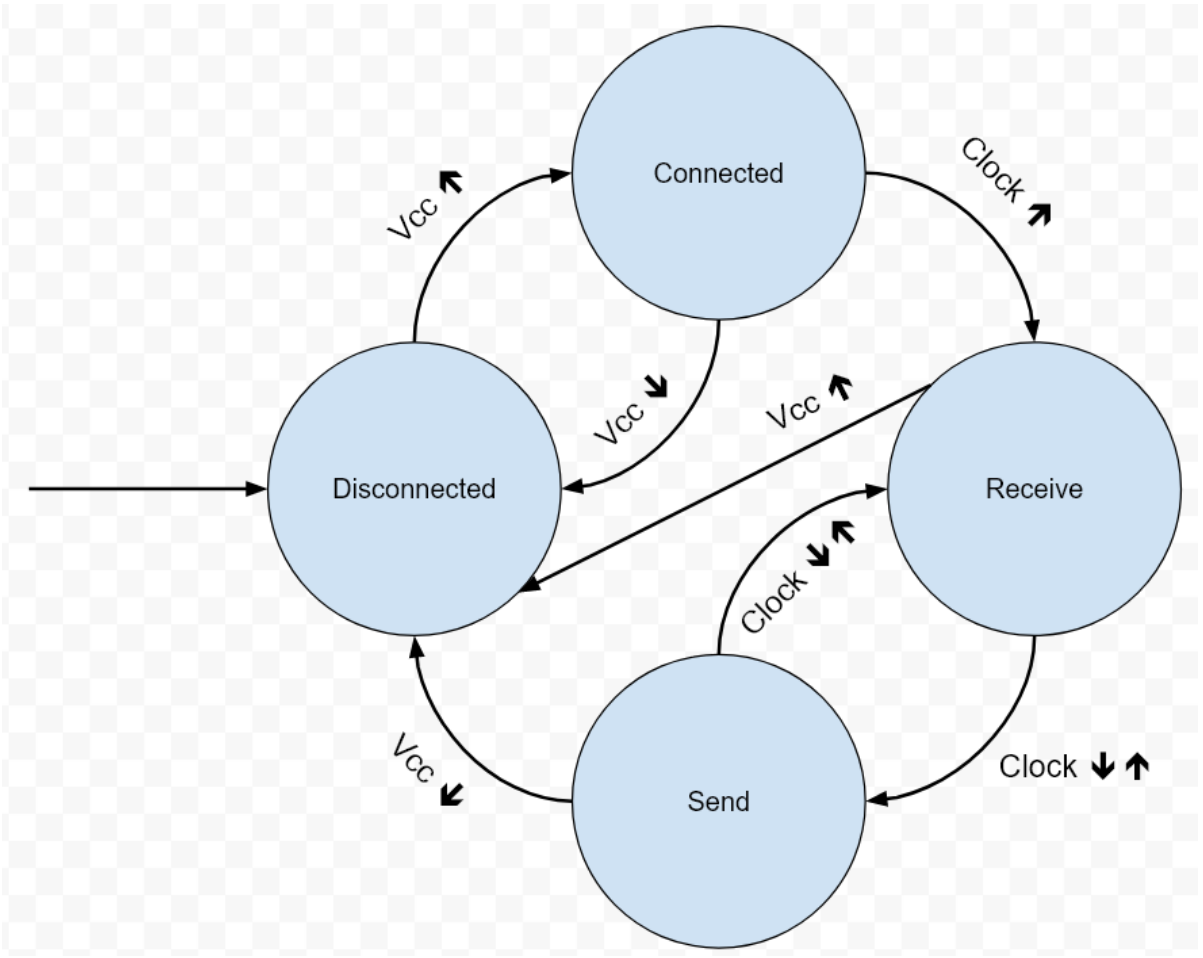
Digital: A software that focuses on the design and the execution of electrical and logic circuits.

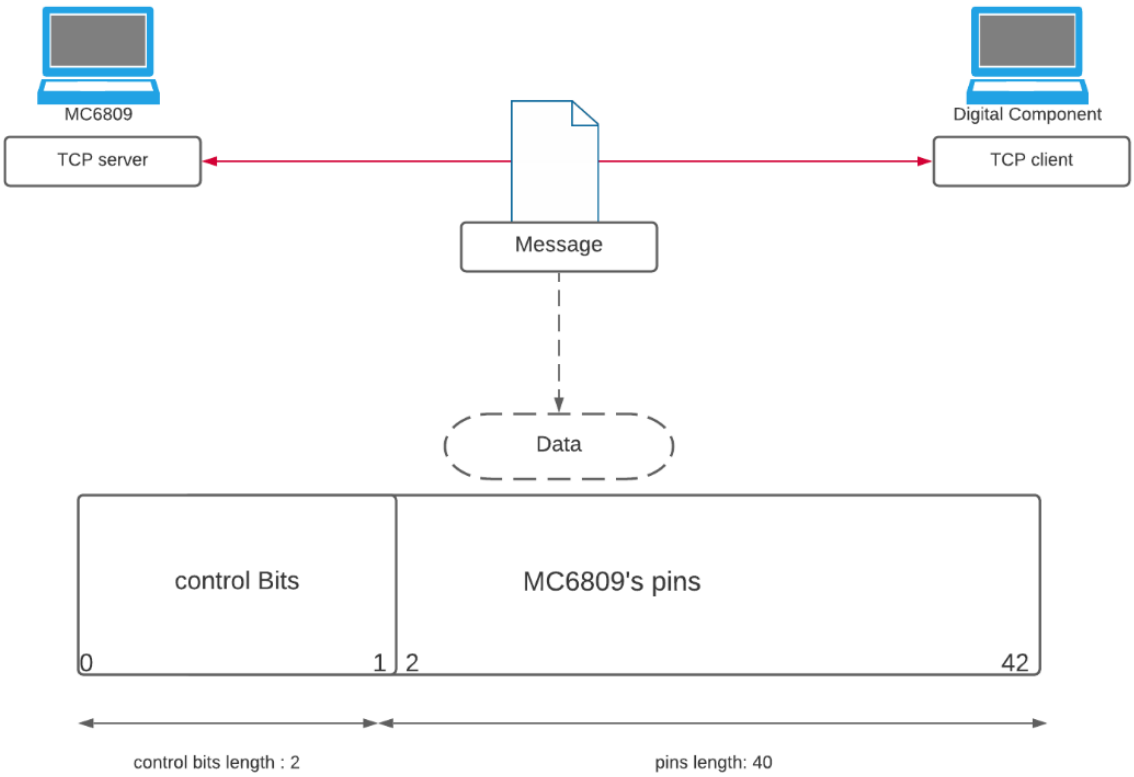
PC (Program counter): A specific register stored in the processor that keeps track of the progress. The current address pointed by the PC stores the instruction currently being executed by the processor.

X. Annex




ON/OFF
0
Clock
notNMI
notIRQ
notFIRQ
VCC
notDMAnotBREQ
MRDY
notReset
EXTAL
XTAL
notHalt
Clock
BS
BA
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15
D0
D1
D2
D3
D4
D5
D6
D7
R/!W
E
Q
MC6809
Dec @
Hex @
RAM
Memory
Ascii Data
Hex Data
Read/!Write


Connected
Clock
Vcc
Vcc
Vcc
Disconnected
Receive
Clock
Vcc
Clock
Send


MC6809
TCP server
Digital Component
TCP client
Message
Data
control Bits
MC6809's pins
0
1
2
42
control bits length : 2
pins length: 40

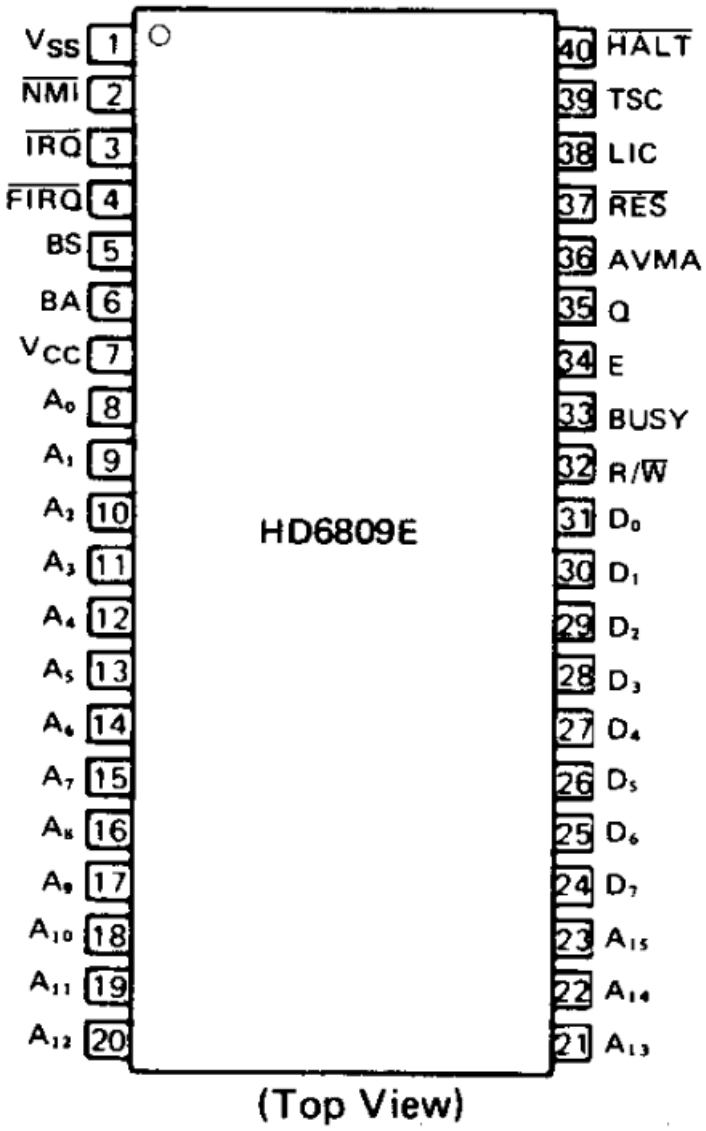


Figure 7: Pin arrangement on the MC6809 chip