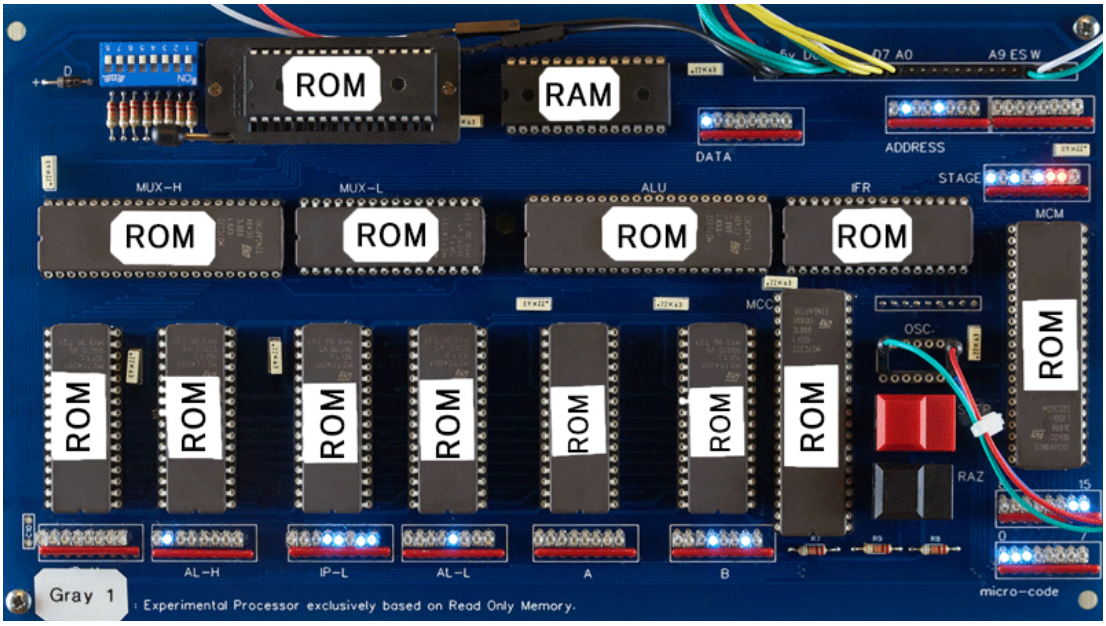
THE GRAY-1, A HOMEBREW CPU EXCLUSIVELY COMPOSED OF MEMORY

Par Olivier Bailleux

ROM + ROM + … + ROM = CPU



This is a complete and functional computer central unit almost entirely made of memory.

This article presents a very original computer. Its CPU is only made of ROM, but it can execute complex programs, such as finding all the solutions of the 8-queens problem. A video is available here.

- No microprocessor!
- The only active components are memory chips.
- Gray code both positive edge and negative edge triggered program counter.
- Ultra-RISC architecture with 8 instructions.
- No stack.
- Neither indirect nor indexed addressing mode.
- Turing complete: can compute everything that is computable by a conventional computer.

Motivations

Since the 1970s, computers are essentially composed of microprocessors, memory, and peripherals such as keyboard, mouse, display, network interface… The microprocessor plays a central role of running the programs located in the working memory. Most people, even among computer experts, see the microprocessor as a black box whose operation is too complex to be understood. Creating a microprocessor from very simple components is a good way to deeply understand how a computer works.

The four main contributions of this project are the followings:

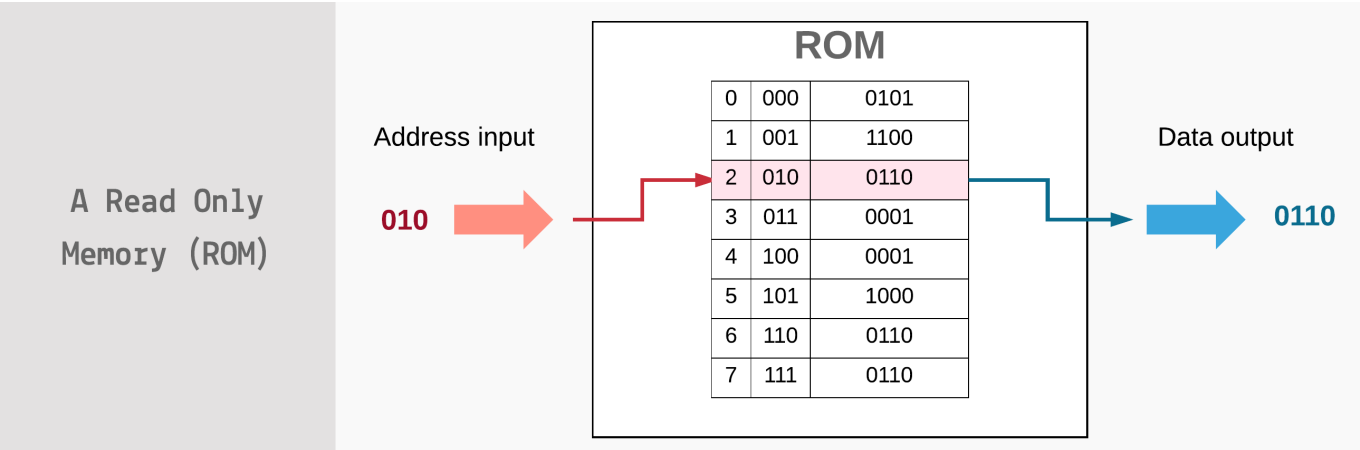
1. **Pedagogical aspect**: going to the essence by creating a computer as simple as possible.

2. **Science popularization**: show that a simple building block (whose behavior can be described in one sentence) is enough to make a machine that can perform the most complex tractable tasks.
3. **Engineering aspect**: show how a complex system can be divided into simpler parts**.**

Fundamentals

What is a memory?

A memory chip is a device that stores binary words. Each storage location is designated by a binary word called address. In read mode, each time an address is put on the address input, the data stored at that address appears on the data output. In write mode (if applicable) the data input is stored at the address placed on the address input. The Gray-1 is essentially composed of ROM (Read Only Memory), more precisely EPROMs (Erasable and Programmable Read Only Memory). An EPROM retains its data until it is erased by a specific ultra violet light source. It also uses RAM (Random Access Memory) to store the running program and its variables.

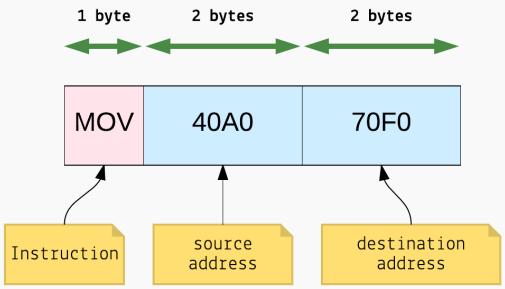


What does a CPU do?

A CPU (Central Process Unit) executes a program consisting of a sequence of instructions and operands. These instructions and operands are encoded as binary words and located into the working memory of a computer. The instructions are simple commands such as adding two values, reading a value at a source address and writing it to a destination address, or jumping to a given address at which the program will continue to run. A Microprocessor is a single chip CPU. There are mainly two kind of microprocessor architecture: RISC (Reduced Instruction Set Computers) and CISC (Complex Instruction Set Computers). RISC microprocessors typically have a few dozen instructions, while CISC ones can have hundreds of instructions. With 8 instructions, the Gray-1 can be considered as an ultra-RISC computer.

The figure below shows one of the instructions of the Gray-1, including the instruction code and the two operands, which are memory addresses. In the memory, this instruction is encoded as 5 successive bytes (8 bits wide binary words), namely 00000001, 01000000, 10100000, 01110000, and 11110000. But for sake of readability the instruction code is represented as a mnemonic and the subsequent operands are represented in hexadecimal.

The MOV instruction and its two operands



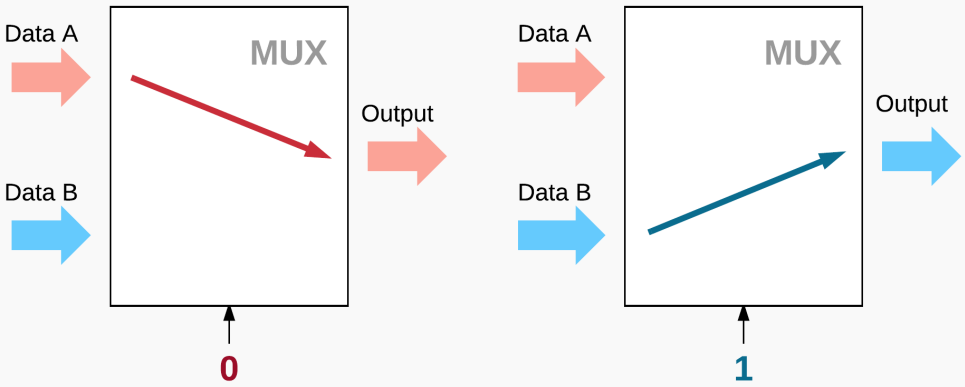
How does a CPU work?

The execution of each instruction is divided into several steps. Most CPUs include a program counter, which successively points the current instruction, optionally one of the subsequent operands, then the next instruction to be executed. The first step consists in reading the instruction code and store it in a dedicated register. The next steps depend on the running instruction and can include numerous operations such that read one of the operands, read the data designated by one of the operands, perform a calculation, store a value at the address designated by one of the operands, or modify the current value of program counter. The details of these operations depend on the CPU elements and how they are connected, namely its *architecture*. There are two kinds of elements into a CPU: combinational circuits and sequential circuits.

Combinational circuits

The current output values of a combinational circuit depend only of its current input values. The behavior of such a circuit can be characterized by a truth table. In a CPU, combinational circuits can have various functions such as data multiplexing (see figure below) and performing basic arithmetic operations.

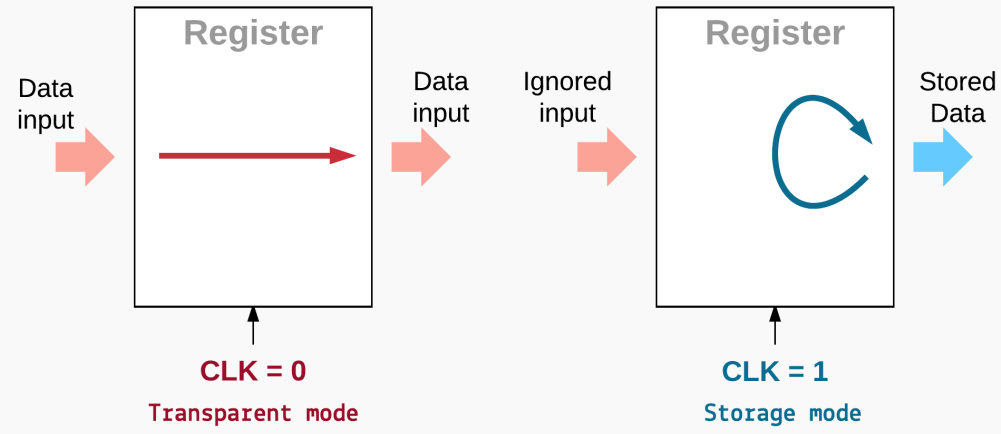
The two modes of a multiplexer



Sequential circuits

The current output values of a sequential circuit depend both on the current and the previous values of its inputs. In other terms, a sequential circuit do have memory. A typical example of sequential circuit used in CPUs is a device called a *register*, which can memorize a binary word. The figure below shows the two modes of a register : the transparent mode, where the output has the same value as the input, and the memorization mode, where the output does not change, regardless of the input value. The working mode is controlled by the CLK (clock) input. Some registers are transparent when their clock is set to 0, other when it is 1, and others are positive or negative edge triggered.

The two modes of a register



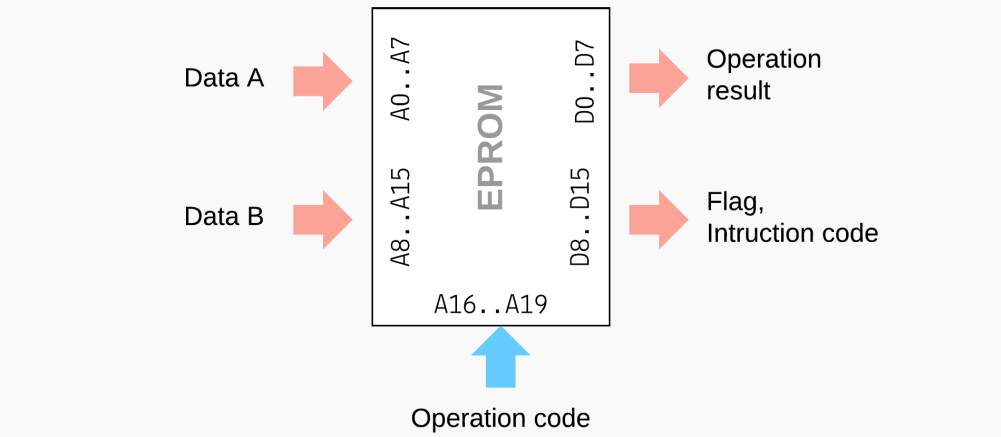
As another example, counters are useful sequential devices which produce sequences of binary words. They can be negative or positive edge triggered. Loadable counters, which can be preset to any of these possible output values, can be used as program counters in CPUs.

Building blocs

Implementing combinatorial circuits with ROMs

Any combinatorial circuit can be easily implemented by putting its truth table into a ROM. For example, the ALU (Arithmetic and Logic Unit) of the Gray-1 is implemented in a vintage 27C322 EPROM that has 21 address inputs (only 19 are used) and 16 data outputs. On the figure below, the data A and data B inputs represent two byte encoded input values. According to the 4 bits operation code, the operation result can be for example A+B, A nand B, A – B, A / 2, A, B… One of the other outputs are used to set a flag that indicates whether an arithmetic carry or borrow has been generated or whether the result of some operation is 0. The remaining outputs are used to route the instruction codes to the instruction register.

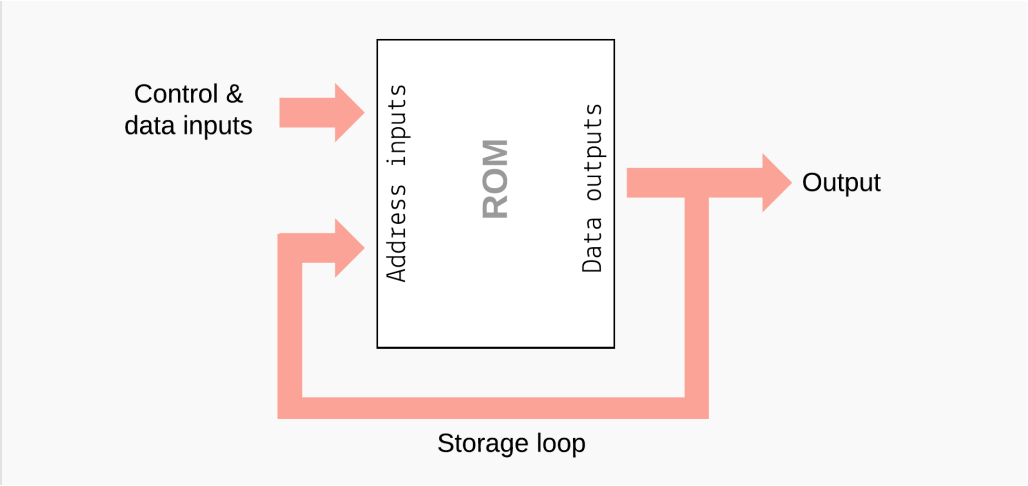
Implementation of an Arithmetic and Logic Unit in a 27C322 EPROM



Implementing sequential circuits with ROMs

The easiest and safest way to implement a sequential circuit with a ROM involves the use of an edge-triggered register. But the challenge I gave myself was not to use existing registers, but only ROM memory chips. This requires turning a ROM into a register. The figure below shows how a ROM can be turned into a sequential circuit by connecting some data outputs on some address inputs.

Turning a ROM into a sequential circuit



This principle relates to the art of asynchronous sequential circuits synthesis. Some rules have to be followed. They will be explained bellow. To this end, let us introduce the following notations :

- Given any binary word A of length n the values of the bits of A are denoted A[0], A[1], …, A[n-1].
- Given any binary word A with length the number of address inputs of a given ROM (or EPROM), f(A) denotes the data word memorized at the address A.
- Given two binary words A and B of length n, T(A,B) denotes the set of binary words of length n such that for any C in T(A,B) and any i in 0..n-1, if A[i] = B[i] then C[i] = A[i] = B[i].

Register synthesis

The registers of the Gray-1 are implemented by using vintage 27C4001 EPROMs, but any ROM with at least 17 address inputs and 8 data outputs may be suitable. The storage loop connects the 8 data outputs to the address inputs A0..A7. The data inputs of the register are the address inputs A8..A15. The address input A16 acts as the clock of the register, which is transparent when this input is set to 0. The ROM is programmed in such a way that if A16=0 (transparent mode) then D0..D7 = A8..A15 regardless of the values of A0..A7, else D0..D7 = A0..A7 regardless of the values of A8..A15 (memorization mode).

```
IF A16=0 THEN
  % Transparent mode
  D0..D7 <- A8..A15
ELSE
  % Storage mode
  D0..D7 <- A0..A7
ENDIF
```

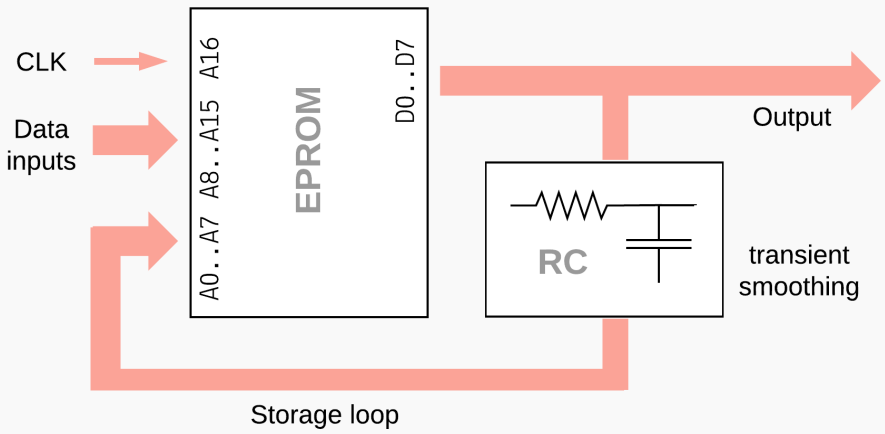
The two following conditions are required for proper operation:

1. The data input A8..A15 must be stable when the clock changes from 0 to 1.
2. Let A and B be two addresses such that for any C in T(A,B), f(C) = X, where X is any value. If the address inputs change from A to B, the data output must remain equal to X.

Unfortunately, it seems that the second condition is not strictly respected by the 27C4001 EPROMs. Some observations have led me to assume that when the address input changes from a value A to a value B, even if the same value X is memorized in all the addresses in T(A,B), a transient state can appear on the data outputs during a few nanoseconds. This transient phenomenon can be enough to “break” the storage loop. I worked around this problem by putting a 1nf capacitor on each data output of the EPROMs used to implements registers. This solution gives no guarantee of reproducibility. It could be improved by using resistor capacitor circuits in the storage loop

instead of single capacitors. Without this hack, the computer crashes after running a few tens of thousands of instructions.

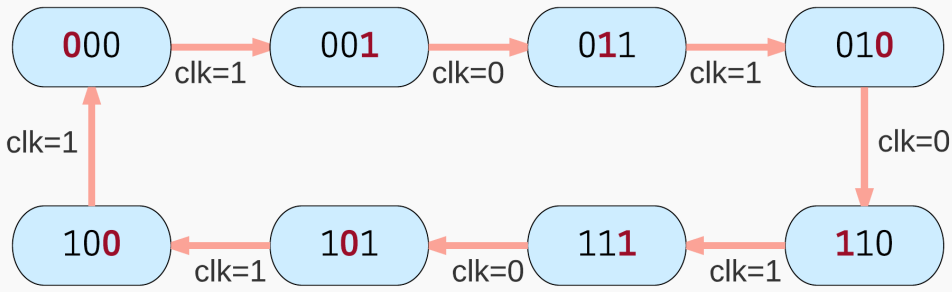
Turning a 27C4001 EPROM into a register



Counter synthesis

A 8 bit counter can be implemented using any ROM with at least 10 address input and 8 data output in the following way: the data outputs D0..D7 are connected to the address inputs A0..A7, A8 acts as the clock input, and A9 acts as the clear input. To ensure proper transition between two successive stable states, only one bit must change. To meet this requirement, the count will be in Gray code. As another unusual feature, our counter is both positive and negative edge triggered.

Counting in Gray code : example with 3 bits



The 27C4001 EPROM is encoded as follow, where int2Gray is a function which translates the binary representation of any integer into the related Gray code, and Gray2int is the inverse function:

```
IF A9=0 THEN
  % Clear mode
  D0..D7 <- 00000000
ELSE
  % Counting mode
  IF Gray2int(A0..A7) is even THEN
      IF CLK=0 THEN
        D0..D7 <- A0..A7
      ELSE
        D0..D7 <- int2Gay( Gray2int(A0..A7) + 1 )
      ENDIF
  ELSE
      IF CLK=1 THEN
        D0..D7 <- A0..A7
      ELSE
        D0..D7 <- int2Gay( Gray2int(A0..A7) + 1 )
```

```
        ENDIF
      ENDIF
    ENDIF
```

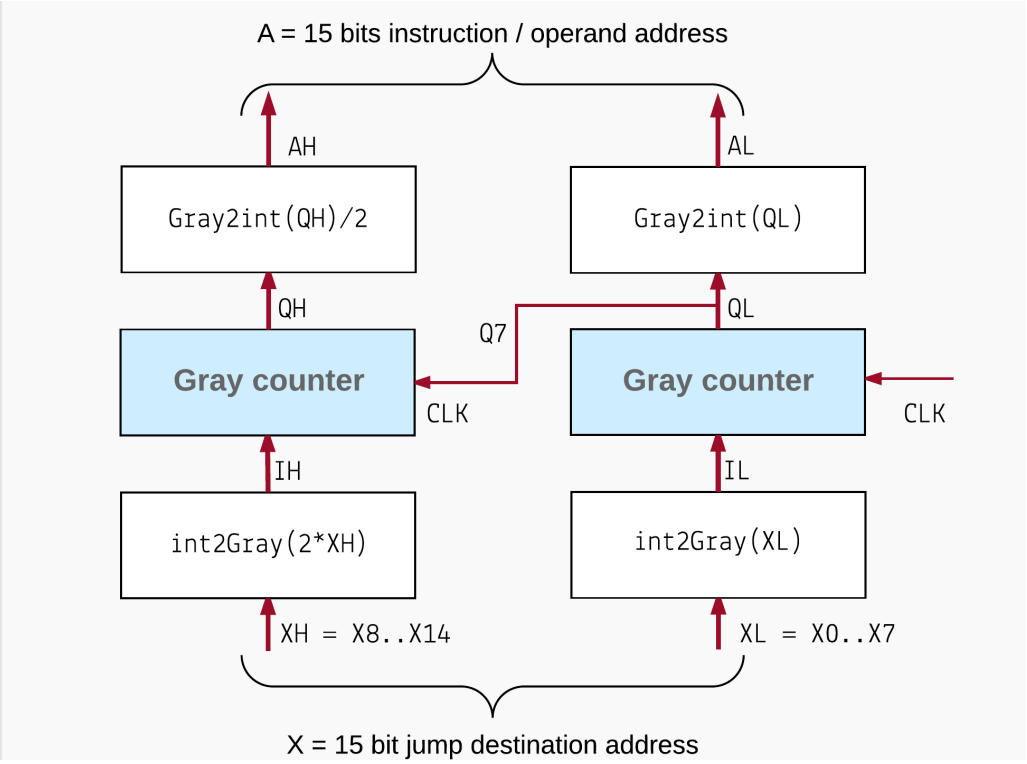
From my experiments, it appears that this design does not require any transient smoothing system.

A 15-bits program counter

A loadable counter can be implemented by combining the two previously described designs. The resulting device has 8 bit data input and 3 command inputs, namely a “clock” input, a “clear” input, and a “load” input. The latter one allows to load the counter with the data input. The counting then continues from this value.

Two 8 bits binary counters can easily be chained so as to achieve a 16 bits counter. However, chaining two Gray counters is less obvious, especially when these counters are both positive and negative edge triggered! The program counter of the Gray-1 is a 15-bit loadable binary counter made with two 8 bits loadable Gray counters as shown in the figure below. The required Gray2int and int2Gray functions are implemented into the EPROMs dedicated to address multiplexing and arithmetic and logic unit respectively (see next section).

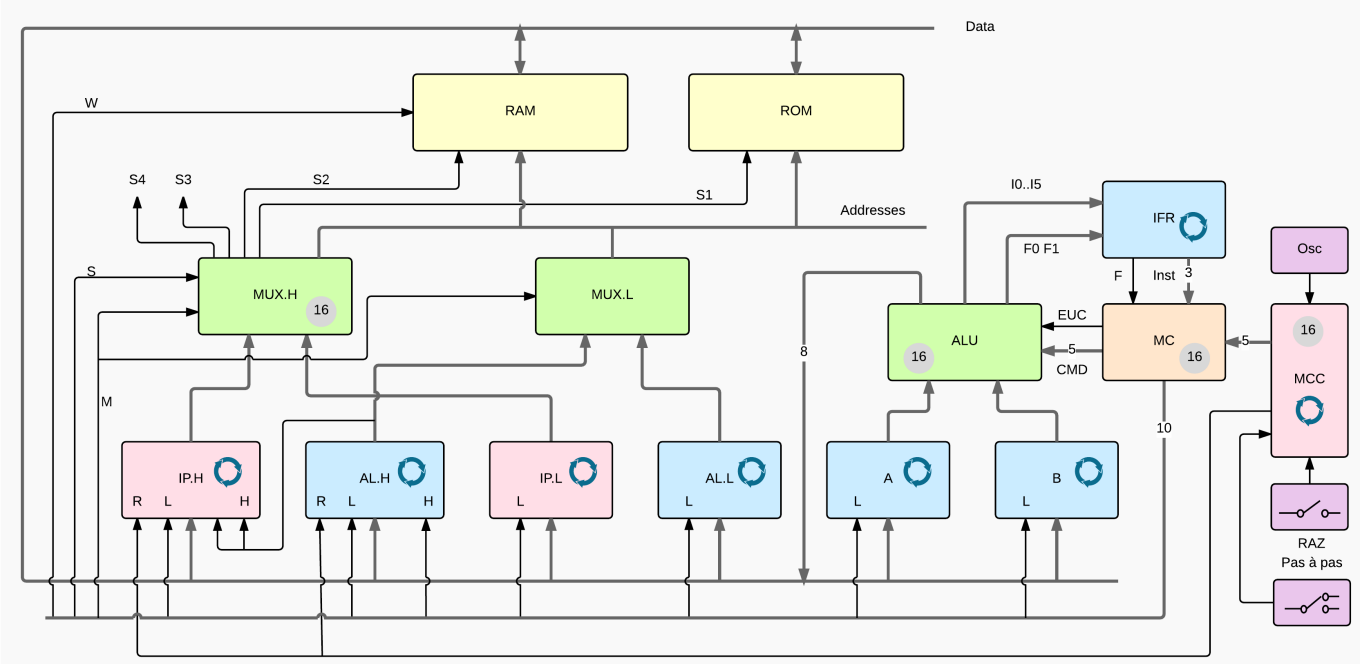
Building a 15-bit program counter with two loadable 8-bit Gray counters



Architecture

The following figure shows the GRAY-1 architecture. Each of the twelve colored blocs is implemented by an EPROM.

A **CPU** made of **Read Only Memory**



The 3 pink blocks are **counters**.

- IP.H and IP.L (for Instruction Pointer Hight and Low) constitute the program counter.
- MCC is the microcode counter. It clocks the stages of execution of instructions.

The 5 blue blocks are **registers**.

- AL.H and AL.L (for Address Latch Hight and Low) store the address where the current data is read or written.
- A and B store the last read data.
- IFR store the current instruction code and the current value of the flag.

The 3 green blocks and the orange one are **combinational functions**.

- MUX.H and MUX.L constitute the multiplexer which forwards either the value of the program counter or the value of the address latch to the address bus. They also provide the translation of the program counter output from Gray into base 2.
- ALU is the Arithmetic and Logic Unit performs the calculations required by the execution of the instructions, including the translation of the jump destination addresses from base 2 into Gray code.
- MC is the Microcode memory. It encodes all the actions required to execute each of the instructions supported by the CPU.

Instruction set

The instruction set can be easily modified by reprogramming the microcode memory. Each instruction is encoded into an odd number of bytes. The first byte is the instruction code. Its value is between 0 and 7. The next bytes encodes the operands. If necessary, an additional alignment byte is added so that the next instruction is located at an odd address. There are two kinds of operands: 1-byte constants and 2-byte addresses. The target addresses of jump instruction must be odd.

The current version consists in the 8 following instructions. Given an address addr, [addr] denotes the value located at addr.

```
MOV src dest      % [src] -> [dest], flag set if [src] = 0
STR const dest    % const -> [dest]
```

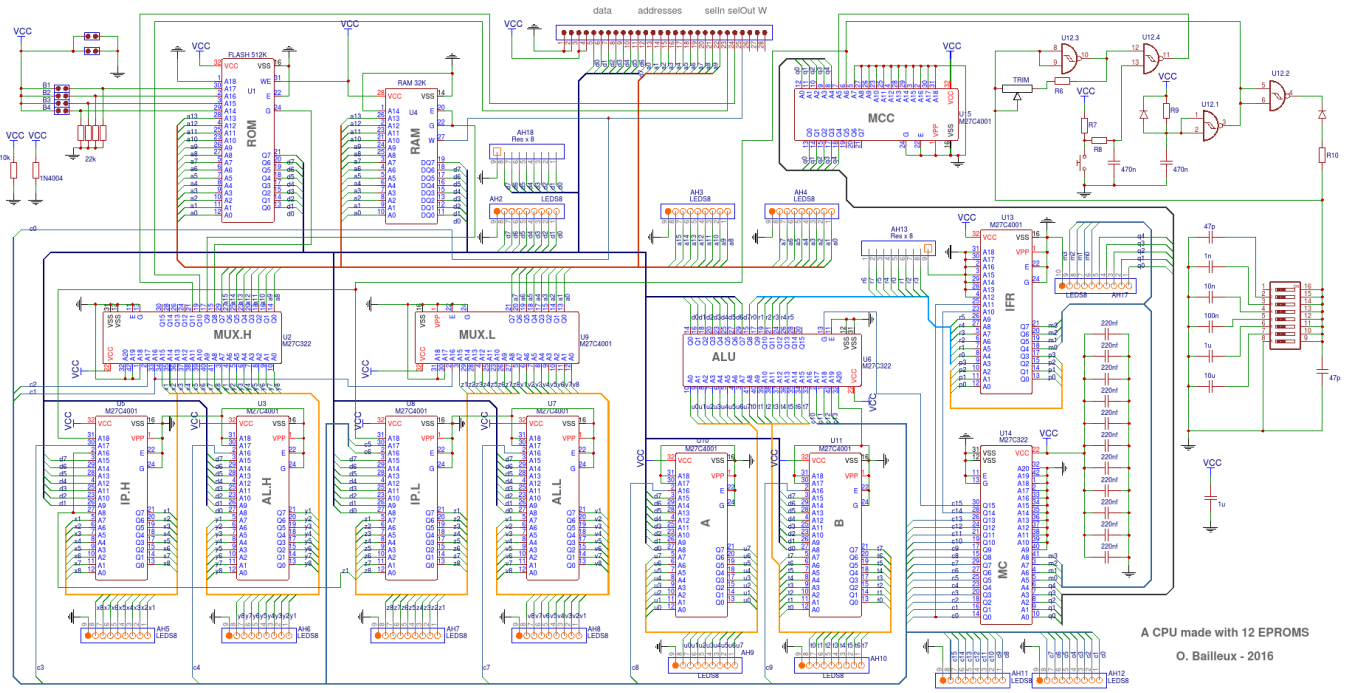
```
ADD src dest      % [src] + [dest] -> [dest], flag set if carry
SUB src dest      % [src] - [dest] -> [dest], flag set if borrow
NAND src dest     % [src] nand [dest], flag set if result = 0
DIV2 src dest     % [src] / 2 -> dest, flag set if result = 0
JF target         % jump to target if flag = 1
JMP target        % jump to target
```

These 8 instructions are more than enough to make the Gray-1 a Turing-complete computer. In fact, only two of them – for example NAND and JF – would be sufficient, but the programs would be longer.

There are two addressing modes: immediate, only available with STR, and direct memory, available with MOV, ADD, NAND, and DIV2. In immediate mode, the data is the operand. In direct mode, the operand is the address where the data is located. Indirect and indexed addressing modes can be simulated by modifying the operand of an instruction during program execution (the program modifies its own code).

Hardware details

Here is the schematic of the Gray-1.



The capacitors of 1nf that have been added to the data output of IP.H, IP.L, AL.H, AL.L, A, B, and IFR for sake of stability are not represented. The LED bars are made with hight luminosity LEDs and 22k resistor networks. MUX.H, ALU, MC are 16-bit EPROMs 27C322. IP.H, IP.L, AL.H, AL.L, A, B, IFR and MCC are 8 bits EPROMs 27C4001. The RAM is a 62256 and the ROM is a 39SF040 Flash memory. Address decoding is ensured by MUX.H with the following address map:

```
Address ranges     Role

0000..3FFF         ROM
4000..7FFF         RAM
8000..CFFF         outputs
D000..FFFF         inputs
```

Because the 15-bits program counter, the space available for the programs is limited to 32ko (16k ROM + 16k RAM). The remaining 32k of the 16-bits addressing space can be only used for data storage and inputs / outputs.

Microcode

The microcode is a critical part of the CPU. It specifies all the actions required for executing each instruction. The execution of each instruction is broken down into 32 steps. At each step, the microcode memory assigns a value to each of the following signals:

```
Signal       Effect when active

clock        clock of the program counter
mem          memory or input/output selection (read or wite)
calc         ALU output is low impedance
mux          address latch output -> address bus
write        write to the memory or outputs, set flag
ipL          ipHload data to program counter
laL          laHload data to address latches
a b          load data into A or B register
instr        load instruction code
ula          ALU operation code
```

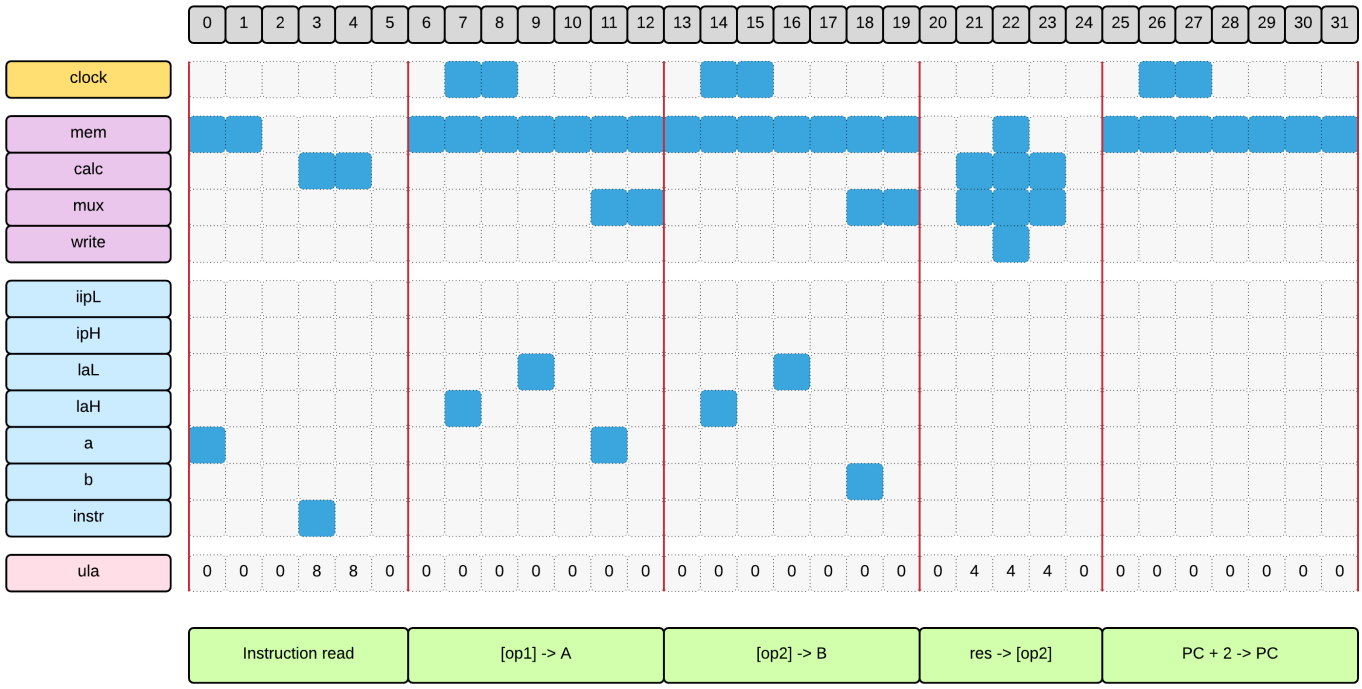
The outputs of the ALU EPROM are used in the following way:

- D0..D7 is the result of the current calculation, connected to the data bus when the ULA EPROM output is low impedance. In the following, it will be designated by result.
- D8..D10 is the instruction code, connected to the instruction code inputs of the instruction and flag register IFR. It will be designated by code.
- D13 is the flag value, connected to the flag input of the instruction and flag register IFR. It will be designated by flag.

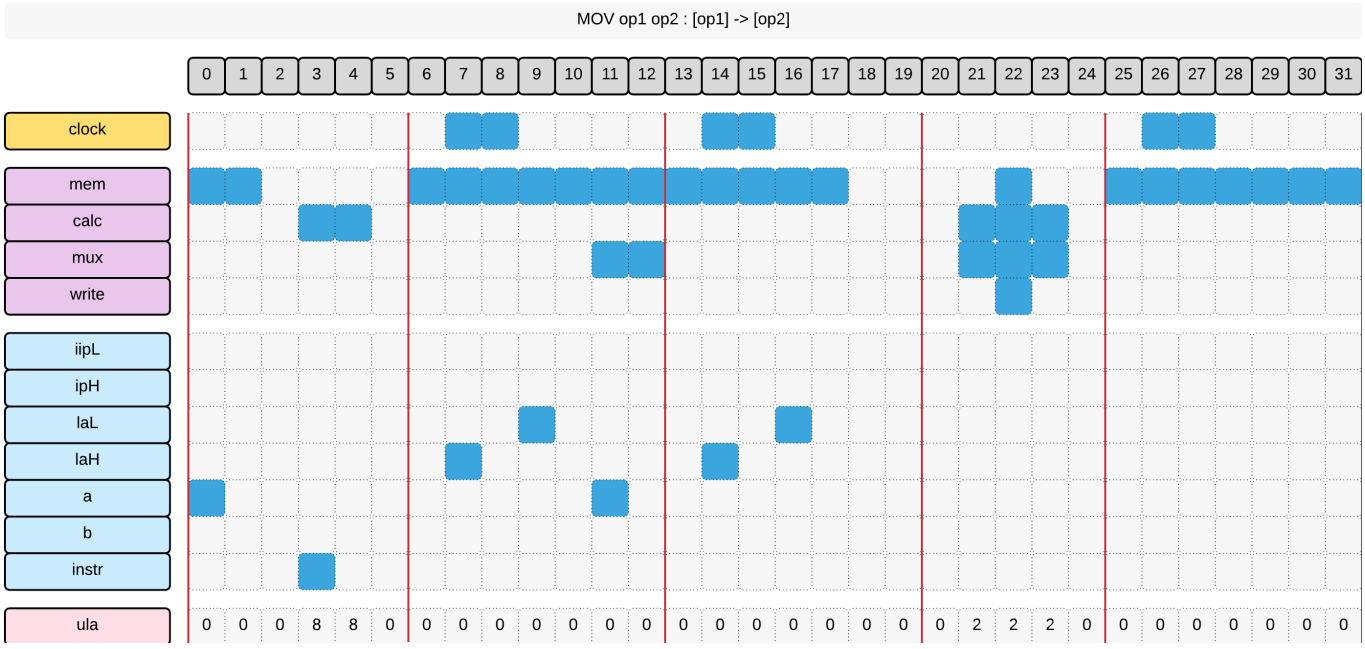
The ALU operation code indicates the operation performed by the Arithmetic and Logic Unit. It should not be confused with the instruction code. The possible values are the following:

```
Value        Effect

0            A -> result
1            B -> result
2            A -> result, flag = 1 if A = 0
3            A - B -> result, borrow -> flag
4            A + B -> result, carry -> flag
5            A nand B -> result, flag = 1 if 0
6            A * 2 -> result
7            A / 2 -> result, flag = 1 if 0
8            0 -> result, A -> code
```

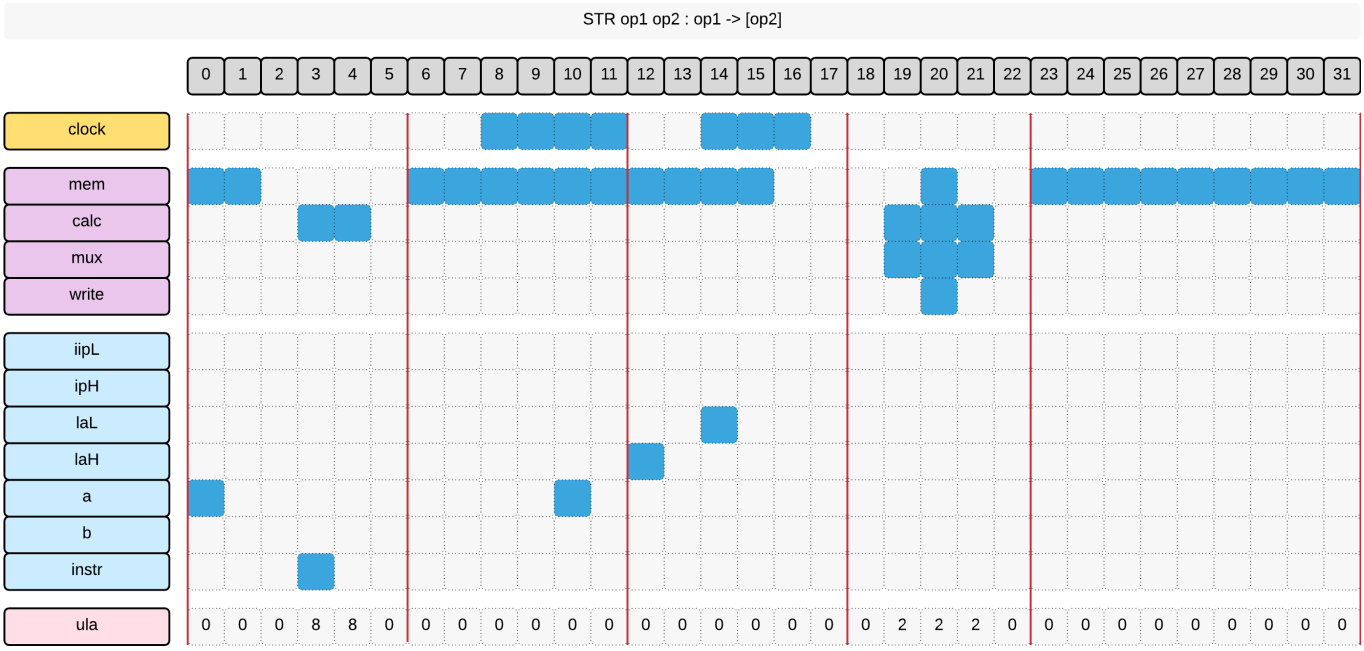
The figure below details the steps of executing the ADD instruction. The blue squares represent active signals. The execution of SUB, NAND, DIV2 are similar, except for the operation code.



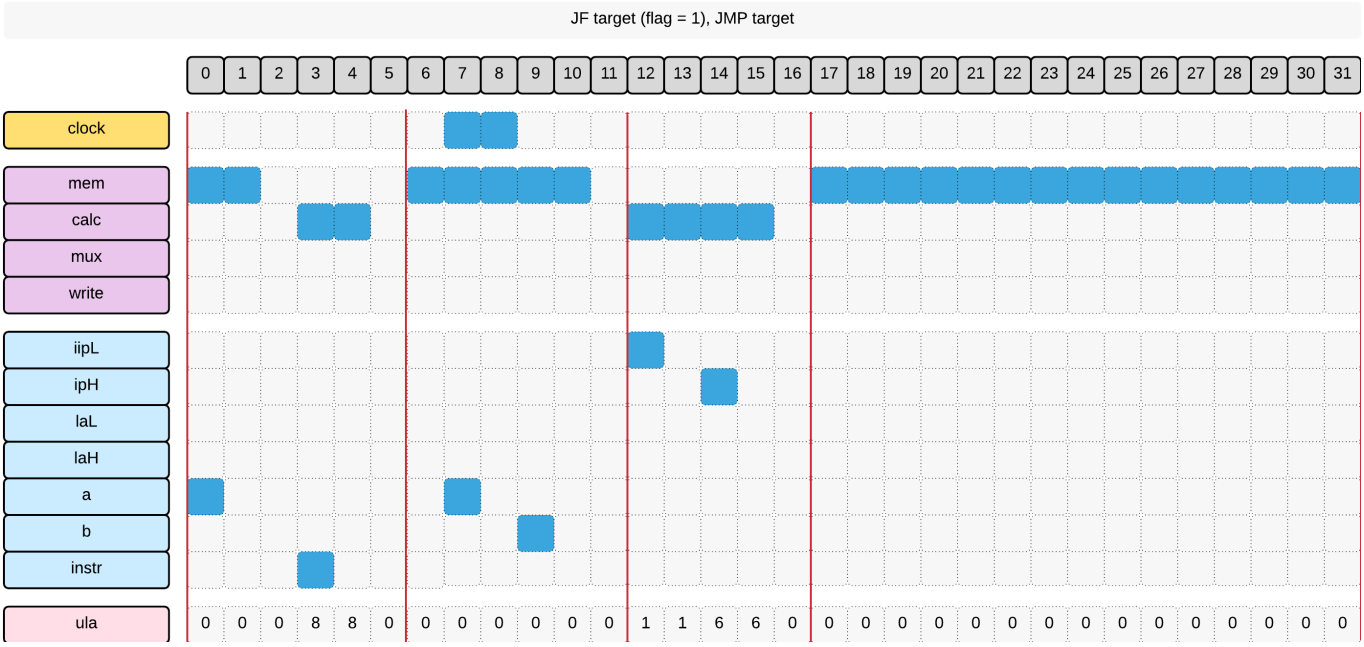
The figure below details the steps of executing MOV.



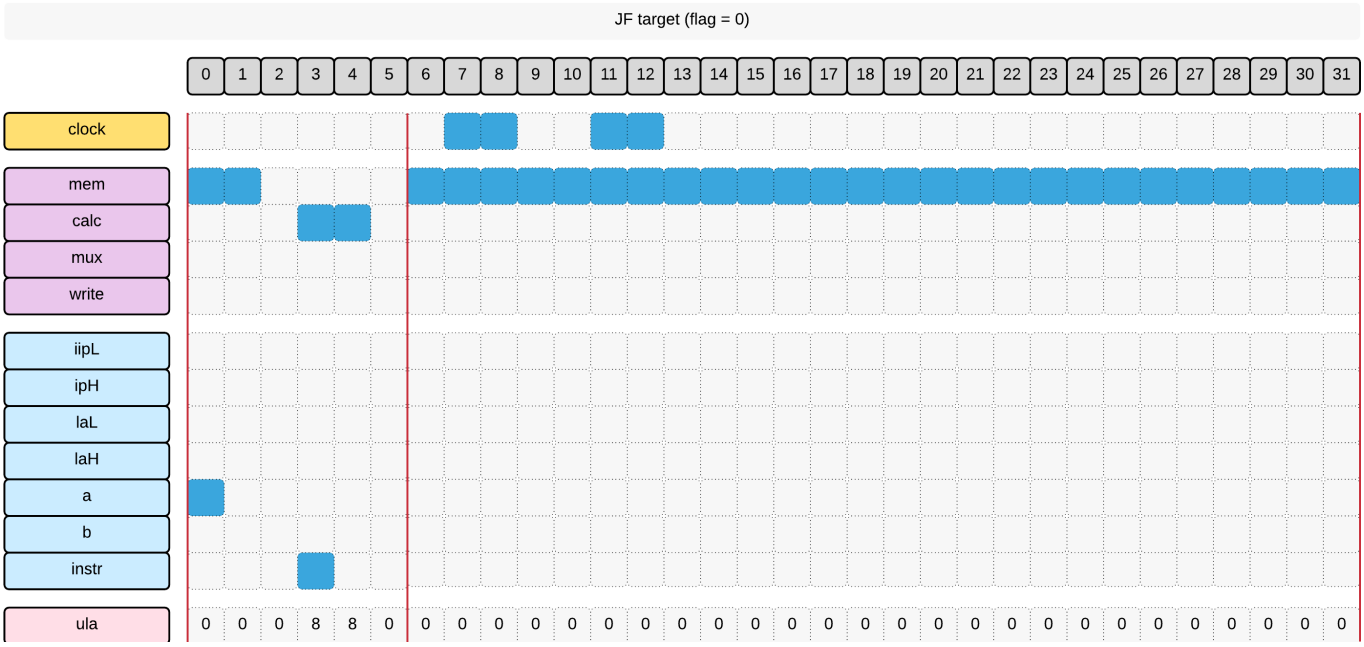
The figure below details the steps of executing STR.



The figure below details the steps of executing JMP and JF when flag = 1.

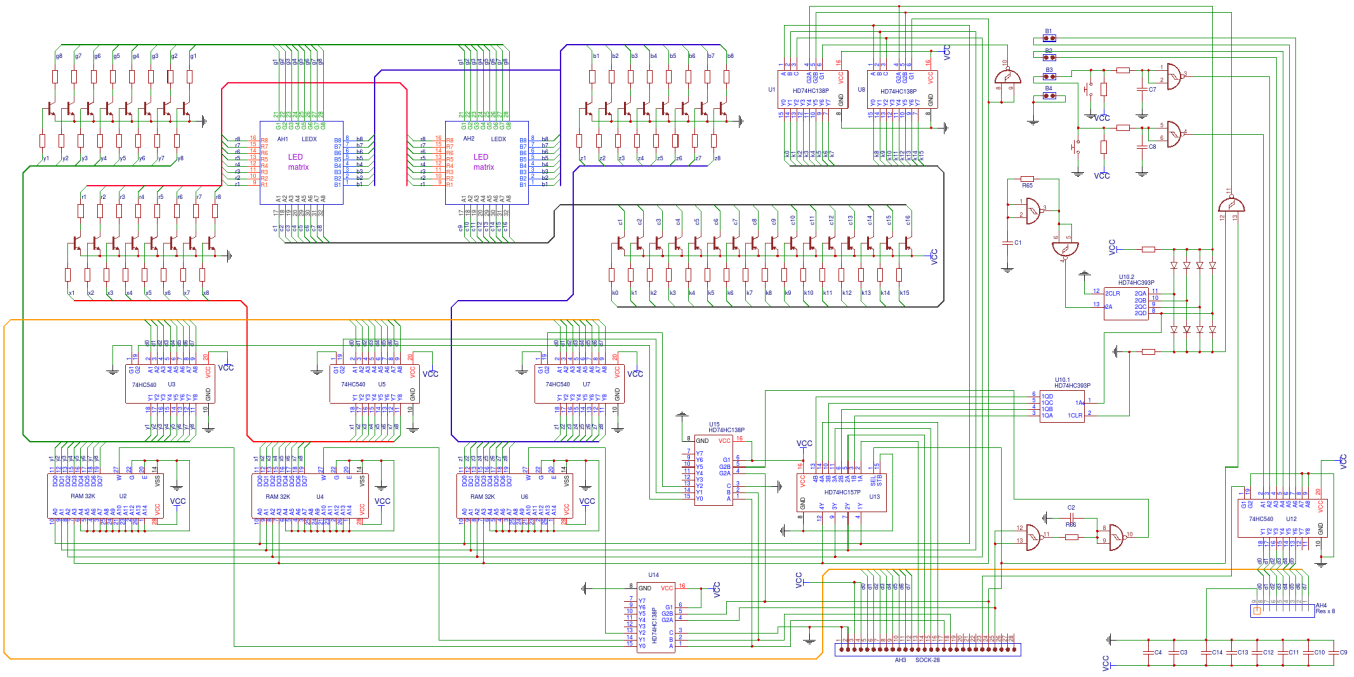


The figure below details the steps of executing JF when flag = 0.

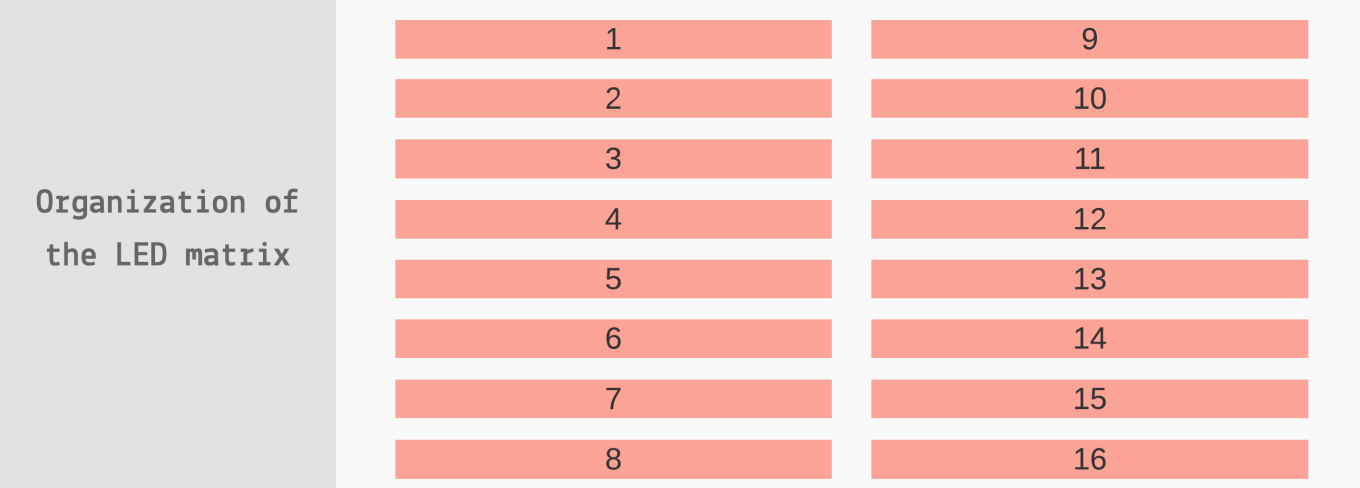


Display and keyboard

Thanks to its input / output connector, the “motherboard” of the Gray-1 can be extended with various display, output or input devices. I developed a display card driving a matrix of 128 (8 x 16) RGB LEDs. This card also has two push buttons and a connector for a possible keyboard. Below is the schematic of this display card.



The LED matrix is composed of 16 lines of 8 LEDs mapped as follows.



The 128 **green** LEDs are mapped to the addresses `C000` to `C01F`. For each byte, the most significant bit corresponds to the rightmost LED, and the last significant bit corresponds to the leftmost LED. The **red** LEDs are organized in the same way at the addresses `C020` to `C02F`. The **blue** ones are organized in the same way at the addresses `C030` to `C03F`. By acting on the 3 concerned bits, 8 colors can be obtained, namely dark, red, blue, green, magenta, cyan, yellow, and white.

Assembly language

The assembly language presented in this section allows for easier implementation of complex programs. It can be used thanks to an assembler written in Java.

Comments, labels and variables

```
@1000                   % Implantation address
.name                   % Variable (1 byte reserved)
.name const 70          % Initialized variable
!fill 00 ff 00 10       % Memory fill
_label                  % Jump <code>target
% Comment line
```

Basic instructions

```
ADD src dest            % Two symbolic operands
ADD @src @dest          % Other syntax for symbolic<code></code> operands
STR #52 dest            % Constant and symbolic operands
ADD #01 cpt             % Constant operand emulation
MOV src (7FE0)          % Symbolic and numeric address operands
JF _target              % Jump to a label
```

The third line of this example encodes the STR instruction on 4 bytes, where the second byte is a constant (immediate addressing mode). The fourth line simulates a constant source operand by encoding the ADD instruction on 6 bytes, where the second and third bytes represent an address where the constant is stored, namely the address of the alignment byte located in sixth position!

Indirect addressing mode simulation

The following syntactic convention identifies operands that can be modified during the execution of the program.

```
MOV [pointer] @dest      % pointer can be modified during execution
LEA stack @pointer       % copy effective address of stack into the
                         % two bytes of pointer representation
ADD #01 @pointer+1 %increments the second byte of pointer
```

Supposing that dest is located at address 0C12, the code produced by the first line is 01 00 00 0C 12 00. The values of the second and third bytes have to be set and modified by the program itself. They are designated by two automatically created variables, namely pointer and pointer+1. The second line encodes two STR instructions which store the two bytes of the effective implantation address of the variable stack into pointer and pointer+1 respectively. The third line gives an example of incrementation modulo 256 of the pointer.

Note that it is not possible to use the same pointer for reading and writing purposes.

Loop and procedure call macro-instructions

The following example shows how use the macro-instructions dedicated to program loops and procedures to write a procedure named refresh, which transfers the content of a display buffer to the display memory. This procedure can be called by the line `!call refresh`.

```
!PROC refresh
      lea display @src       %source address
      lea video @dest        %destination address
      str #30 @cpt           %number of bytes to transfer

      !loop [cpt]            %Repeat cpt times
      mov [src] [dest]       %1 byte transfer
            add #01 @src+1   %Increments source pointer
            add #01 @dest+1  %Increments destination pointer
      !endloop
!VARS                        %Local variables
      .cpt                   %loop counter
!ENDPROC
```

Each variable and label defined into a procedure can be used outside the procedure by prefixing its name with the procedure name following by the character ":". For example, the variable named cpt into the procedure refresh can be used outside this procedure with the name refresh:cpt. This prevents name conflicts and allows the use of local variables as parameters.

Booting process

The boot program starts running at address 0000, in ROM, but because indirect addressing mode can be emulated only in RAM, the first think it must do is to transfer the application program in RAM. Because this transfer process requires indirect addressing mode, the following little procedure must be first transferred in the RAM.

```
@7FE0
!proc copy
    mov [src] [dest]
!endproc
```

This transfer is made byte per byte in the following way.

```
@0000
_reset
    str #01 (7FE0)
    lea map (7FE1)
    lea video (7FE3)
    str #00 (7FE5)
    str #07 (7FE6)
    lea acc_e3 (7FE7)
```

```
        str #00 (7FE9)
```

The rest of the boot program uses this copy procedure to transfer all the bytes located at addresses 0200 to 3CFF (in the ROM) to the addresses 4000 to 7DEF (in the RAM). On the other hand, the assembly program put at addresses 0200 to 3CFF all the code implanted by the programmer at addresses 4000 to 7DEF.

The whole boot program is detailed below.

```
@01B0
%Welcome display
_map
!fill 00 89 89 8f 89 89 00 00
!fill 00 e0 a0 a0 a0 e3 00 00

!fill ff 00 00 00 00 00 ff 00
!fill ff 00 00 00 00 00 ff 00

!fill 00 70 10 70 10 70 00 00
!fill 00 04 04 04 04 1c 00 00

%Display memory
@C000
_video
@C007
_aff1
@C02f
_aff2

%Variables
@7E00
.trash
.size
.boot_nbseg
.boot_nboct

%Transfers 1 byte RAM
@7FE0
!proc copy
       mov [src] [dest]
!endproc

@0000
_reset

 %Initialization of the 1 byte transfer procedure
       str #01 (7FE0)
       lea map (7FE1)
       lea video (7FE3)
       str #00 (7FE5)

       str #07 (7FE6)
       lea acc_e3 (7FE7)
       str #00 (7FE9)

 %*************** Initializing welcome display. ***************

       str #30 @size
_acc_e1
```

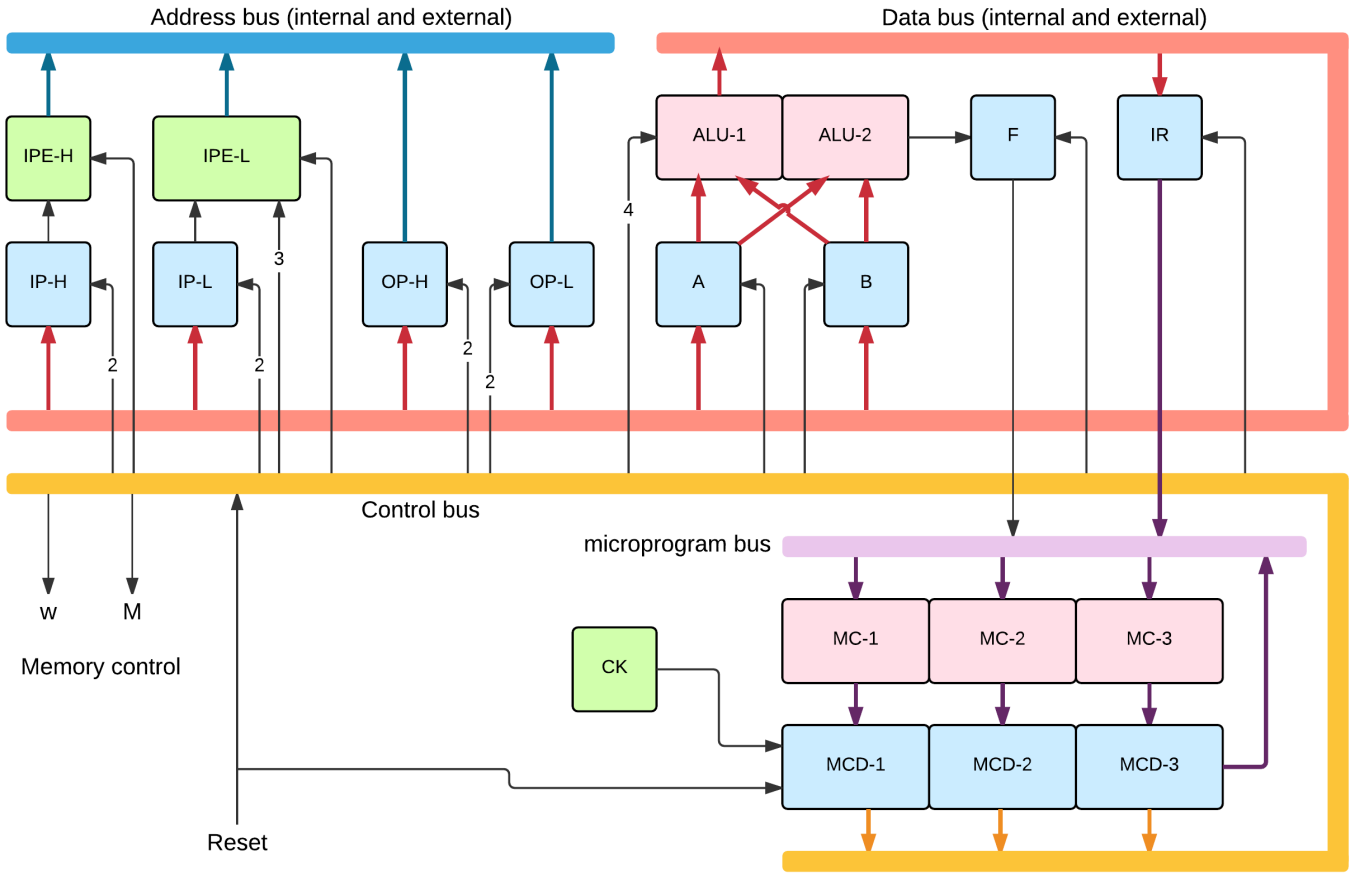
```
        mov @size @trash
        jf acc_e2

        add #ff @size
        jmp copy
_acc_e3
        add #01 @copy:src+1
        add #01 @copy:dest+1
        jmp acc_e1
_acc_e2

 %************** Transfers 0200..3CFF to 4000..7DFF. ***************

jmp boot_trz

%---------------------- Transfers 256 bytes ------------------------
_boot_trseg
        str #00 @boot_nboct
        str #00 @copy:src+1
        str #00 @copy:dest+1
_boot_trseg1
        jmp copy
_boot_trseg2
        add #01 @boot_nboct
        add #01 @copy:src+1
        add #01 @copy:dest+1
        mov @boot_nboct @trash
        jf boot_trz2
        jmp boot_trseg1

%--------------- Transfers 62 segments of 256 bytes. ---------------
_boot_trz
        str #3E @boot_nbseg
        str #02 @copy:src
        str #40 @copy:dest
        lea boot_trseg2 @copy:return
_boot_trz1
        jmp _boot_trseg
        add #01 @copy:src
        mov @copy:src @aff1
        add #01 @copy:dest
        mov @copy:dest @aff2
        add #ff @boot_nbseg
        mov @boot_nbseg @trash
        jf boot_fin
        jmp boot_trz1

_boot_fin
        jmp start_prog
```

Simulator

A simulator written in Java facilitates the development of programs before transferring them in the ROM. It allows step by step execution, breakpoints, memory spy and display simulation.

Future work

The next project in the pipeline is to create a CPU with 2 kinds of memory chips: ROM and edge triggered registers. The advantage of such a design is that the CPU will be fully reproducible, with easy component supply. And because a register is nothing but a 1-byte RAM, this CPU will be also exclusively made of memory. Below is the architecture of this new memory-based CPU.



Pink devices are 8-bits flash memory chips. Blue devices are edge triggered registers. There is no program counter, and each instruction must be located at an address multiple of 8. The two green devices IPE-H and IPE-L are tri-states buffers. There are required because have not found any D-Latch circuit with both a clear input and tri-states outputs.