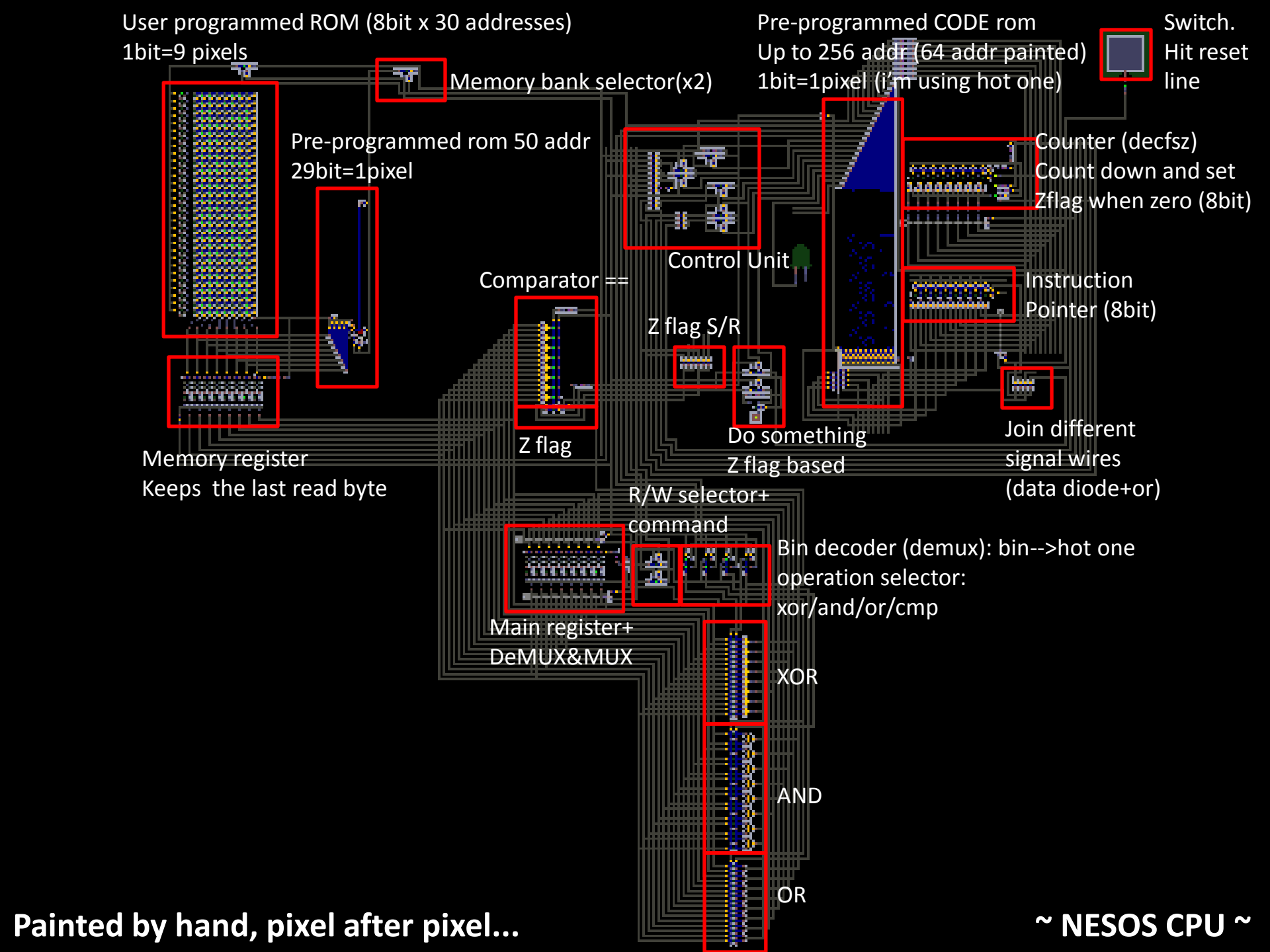

User programmed ROM (8bit x 30 addresses)
1bit=9 pixels
Memory bank selector(x2)
Pre-programmed CODE rom
Up to 256 addr (64 addr painted)
1bit=1pixel (i'm using hot one)
Switch.
Hit reset
line
Pre-programmed rom 50 addr
29bit=1pixel
Counter (decfsz)
Count down and set
Zflag when zero (8bit)
Control Unit
Comparator ==
Instruction
Pointer (8bit)
Z flag S/R
Memory register
Keeps the last read byte
Z flag
Do something
Z flag based
Join different
signal wires
(data diode+or)
R/W selector+
command
Bin decoder (demux): bin-->hot one
operation selector:
xor/and/or/cmp
Main register+
DeMUX&MUX
XOR
AND
OR
Painted by hand, pixel after pixel...
~ NESOS CPU ~

Initial idea and things to note:

My initial idea was to build a simple finite state automata and not a complete computer to keep things simple and solvable in decent time also to new users. In fact it doesn't have many essential components like ram or addressable memory.

Turns out that many of the problems i have encountered could be solved by making the things in the way computers are build. Building a pc is something really worth to try; you might think that they are not very complex, after all there are a bunch of components activated by instruction bits and each component does a task; but keeping precise timings and ensure that everything works when interconnected is not so simple as i initially thought.

You also need to ensure that everything is in a known state for each clock cycle and that on "boot" it starts in a known state; i have solved this by using a reset line connected to each component.

Another *amazing* thing to notice is that my first computer was a pentium iii with windows 98 had 64MB of ram and 3 gb of hdd (today a dvd is bigger and becoming useless because is too small).

My new pc is much faster, has 8 gb ram 250 gb hdd. Higher numbers means it is better.

But these are just *numbers* you don't really see any difference, my old pc needed some time to boot and the new one too, i had programs and games on the old one and same on the new one.

Behind these numbers there are people trying hard to make things faster and smaller. Building a pc in powder toy allows you to be such a person and understand how many tricks you have to do to make it work "fast". Having a limited space (and speed) to be used means that you can't limit yourself to make something that "just works". You have to optimize things to gain some frames of speed, optimize the space used or you will run out of space. And many people in this game are trying really hard in this optimization. For example the multiplexer/ deviator switch used near main register is made with piston and inst and it doesn't matter where current flow, it doesn't lose any frame; it was very important to not lose any frame or everything goes out of sync and i couldnt reinvent half pc to fix that. Someone invented a full pc based on this trick.

Anothr example take the three memory used here:

-the left one needs 3x3=9 pixels to store 1 bit of data (is slow on reading and has no constant time)

-the right one needs 1 pixel to store 1 bit (and seems the best possible)

-but the center one can store 29 bits in one pixel!

Think also about the speed, wires can be sparked every 8 frames

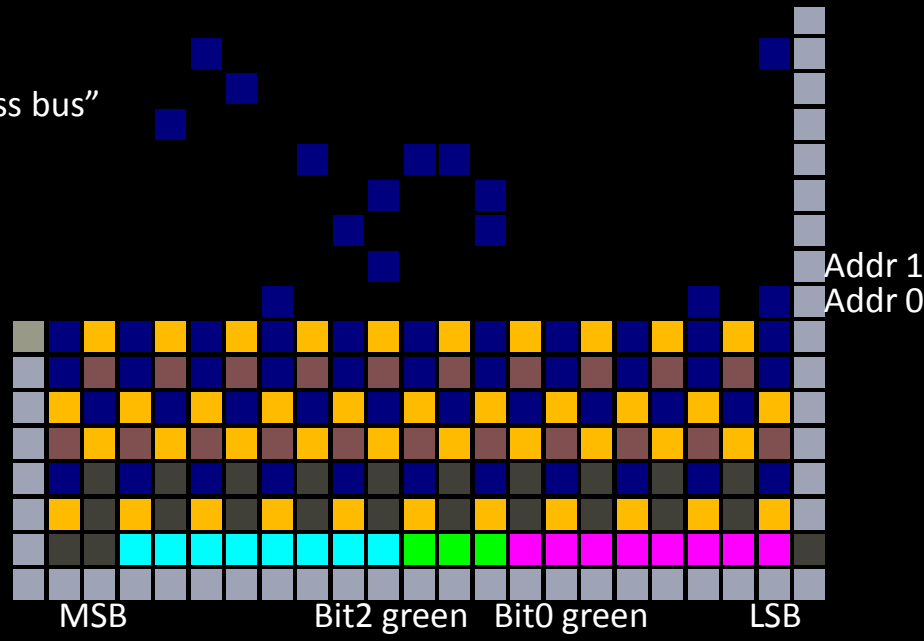
So you might thing that this is the maximum speed possible, like the pentium 4; you couldn't go above that speed or everything melts down due to power dissipation problems.

But someone found a solution: multicore in real pc and exploiting game mechanic to run at 60fps in powder toy.

Some computers worth of note (much better than mine): id:1599945; id: 975033; id: 1761441; id:1932845

ISA (Instruction Set Architecture):

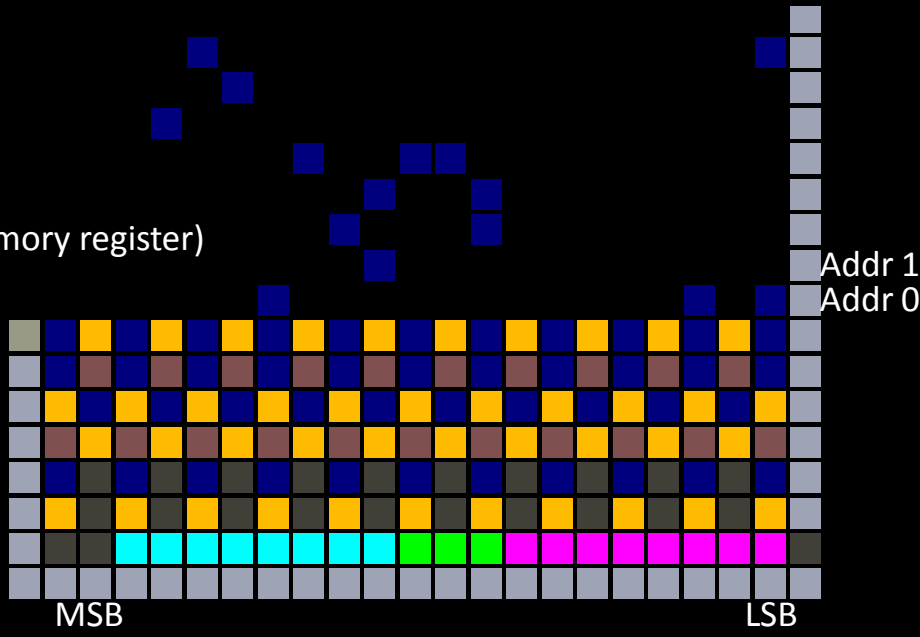
Most of the instructions are “hot one” to avoid encodings.
The pink painted part of the memory is connected to the “address bus”
And is connected to:
-the decfsz counter (to set its initial number)
-IP/program counter in case of JMP
The green part is used to select between options and is used to:
-set read/write mode of the main register (bit0 green)
0=read; 1=write
-set the memory bank to be read (bit0 green)
0=left bank; 1=right bank
-set the operation (bit 2 and bit1 green)
Xor / and / or / cmp
0,0 / 0,1 / 1,0 / 1,1



The light blue part are real instructions:
Bit 0 light blue = read memory (bank from bit0 green), result goes to memory register
Bit1 = activate memory register and main register(read/write based on bit0 green); if set to write this is mov mem,mainreg
Bit2 = activate “ALU” (operation selected by bit2 and bit 1 green)
Internally connected to read memory register wire + activate (default read) main register + multiplexer switch of main register (connect it’s input/output to logic gates instead of data bus).
After a small delay it also set write mode on main register and enable it (to store the operation result; except for cmp)
Bit3 = set decfsz counter value (value stored on pink bits)
Bit4 = decfsz: decrement the counter by one and skip one instruction if zero (by increasing IP/program counter)
Bit5 = JMP: unconditional jump, destination on pink bits
Bit6 = HCFNE, Halt and Catch Fire if Not Equal; explode/autodestruct if not zero. Thanks to this i don’t need to implement conditional jumps and is also lots nicer ehehe :)
Bit7 = turn on led (there is no connection to cpu parts, this just turn on led)
The two remaining grey bits are not connected/used

PROGRAM:

The memory low address are down and increase going up
Program entry point is at address 0
The initial part of the program is explained here:
-Set counter = 5 (prepare a loop)
-read bank 0, left one, the one you have to fill (result goes to memory register)
-mov memory register, main register (copy to main reg)
-read bank 1 (the fixed one, right)
-compare mem reg and main reg
-self destroy if not equal
-decfsz (when 0 jmp will be skipped and program continue)
-jmp address 1 (do everything again)



All this is used to compare left and right memory
and check if it contains “flag{“

The second part read your char, read the bank1 xor your char with the bank1 value, read again bank1 and compare.
So its like if (flag[i]^constVal[i1]==otherConstVal[i2])

Third part does bank1[i1] AND bank1[i2] OR bank1[i3] == flag[i4]

Last part read flag char from bank0 XOR it with value from bank1 and compare if equal to next flag char (bank0).
This has multiple solutions based on the last flag char. Logic says that last char must be } so only one correct solution exist.
But to ensure that led doesn’t light up in case of wrong solution there is one last cmp that check if last char is }.

```
Flag{d0nt_you-like-f.s.au7oma}

BLU = confronta se uguale   ← Cmp user flag==programmed flag
VERDE = cmp flag XOR const1 == const2
GIALLO = cmp flag  == (c1 AND c2 OR c3)
ROSSO = for each char: cmp (flag[i] XOR c1 ) == flag[i+1]
EXTRA cmp ultimo char == }
```

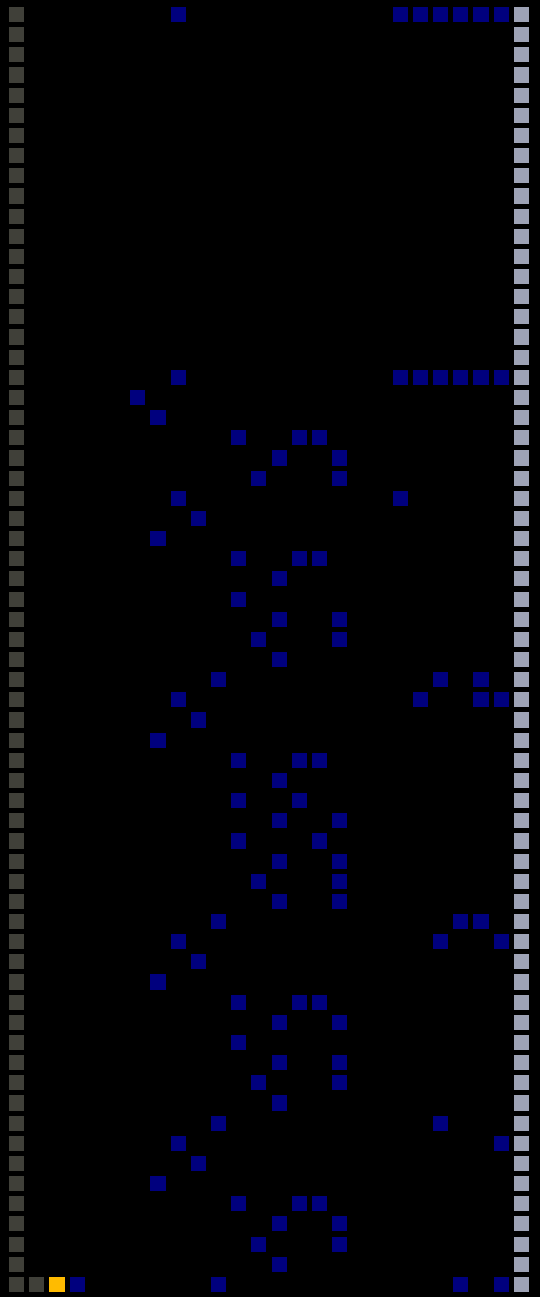
Check last char == } to be sure that there is only one solution

PROGRAM:

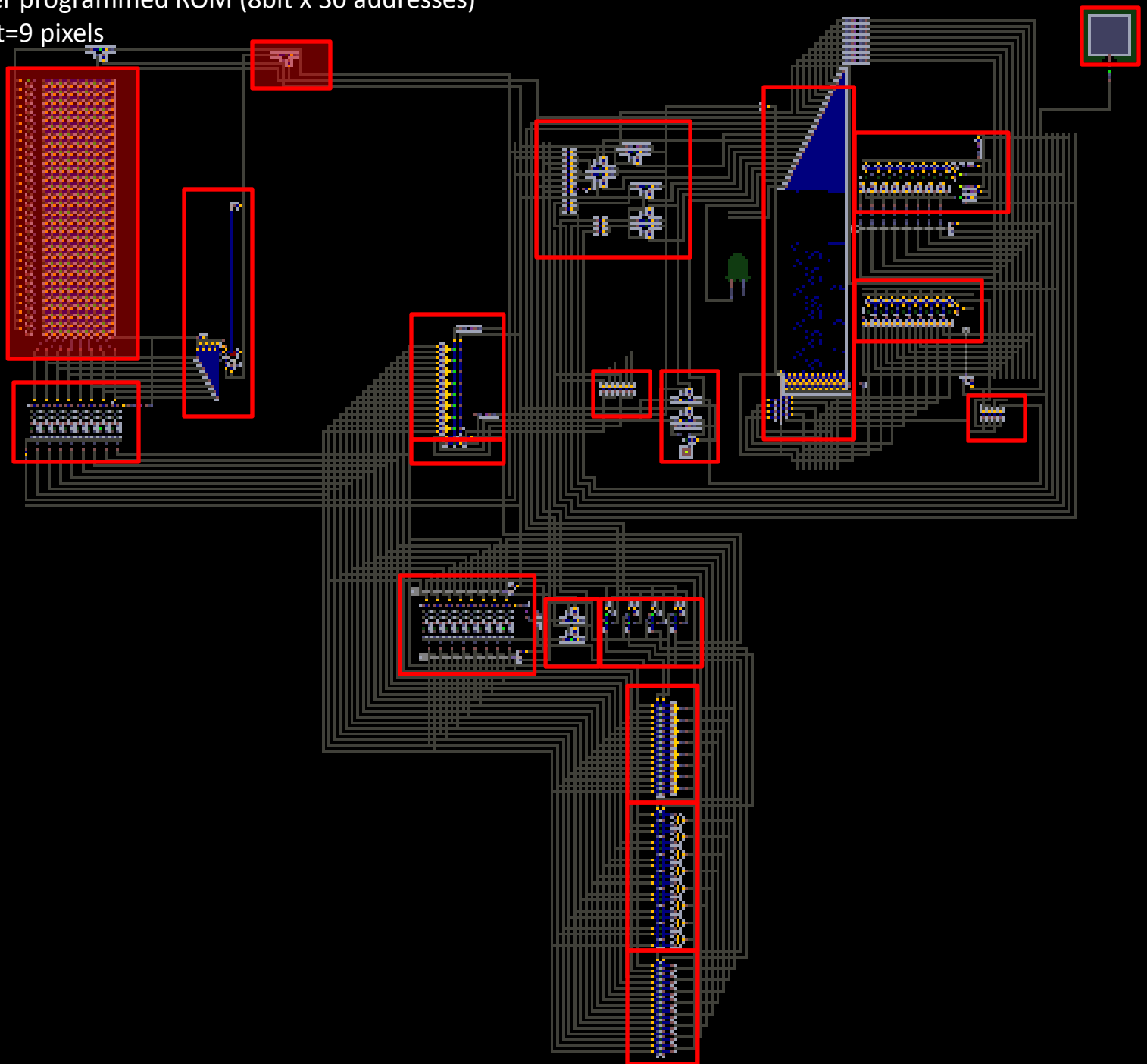
Pseudo assembly program used.

Must be read from down to up (the way memory is read) and from left to right because it doesn't fit in one page;)

```
jmp giù3        jmp self
decfsz          fine:
bom nz          jmp fine
cmp mw          on led
rdm 0           bom nz
or mw           cmp mw
rdm 1           rdm 1
and mw          mov mw
rdm 1           jmp giù4
mov mw          decfsz
rdm 1           bom nz
giù3: (19)      cmp mw
setc 6          rdm 0
                xor mv
jmp giù         rdm 1
decfsz          mov mw
bom nz          giù4: (32)
cmp mw          rdm 0
rdm 1           setc 10
mov mw
rdm 0           jmp giù3
giù: (1)        decfsz
setc 5          bom nz
                cmp mw
jmp giù         rdm 0
decfsz          or mw
bom nz          rdm 1
cmp mw          and mw
rdm 1           rdm 1
mov mw          mov mw
rdm 0           rdm 1
giù: (1)        giù3: (19)
setc 5          setc 6
```

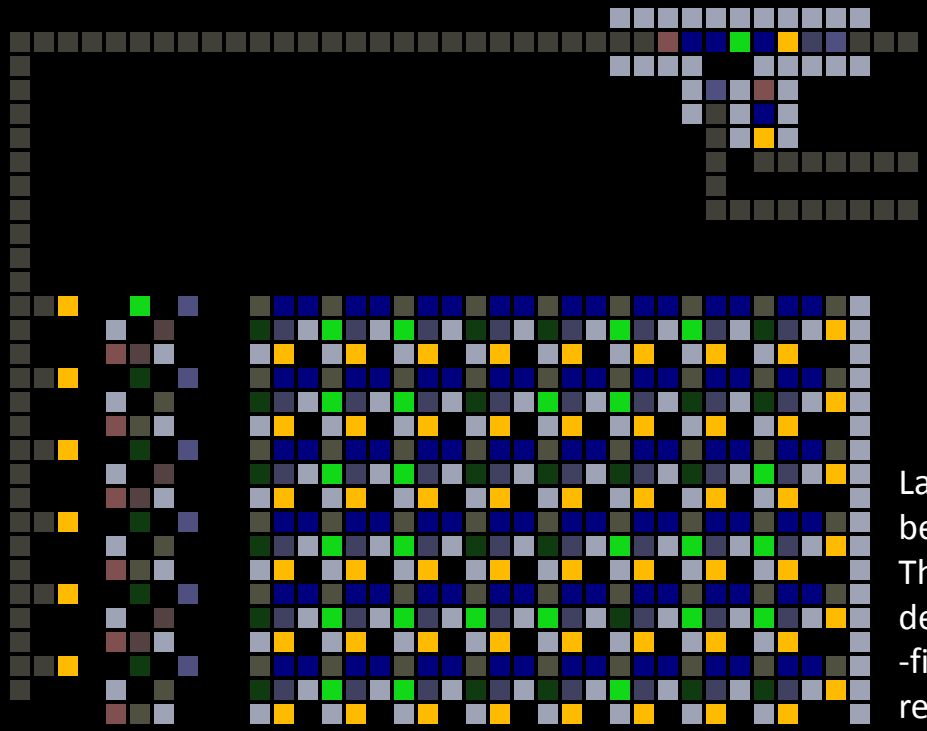


User programmed ROM (8bit x 30 addresses)
1bit=9 pixels



Sequential memory, after each read the address is incremented by one

This switch is used in many places (especially in control unit) can work with signals in the same frame:
Command+ on/off in the same frame works good



Read memory instruction

Bank Selector:
One wire to reset line, other to bank bit
On the other memory the wires are swapped

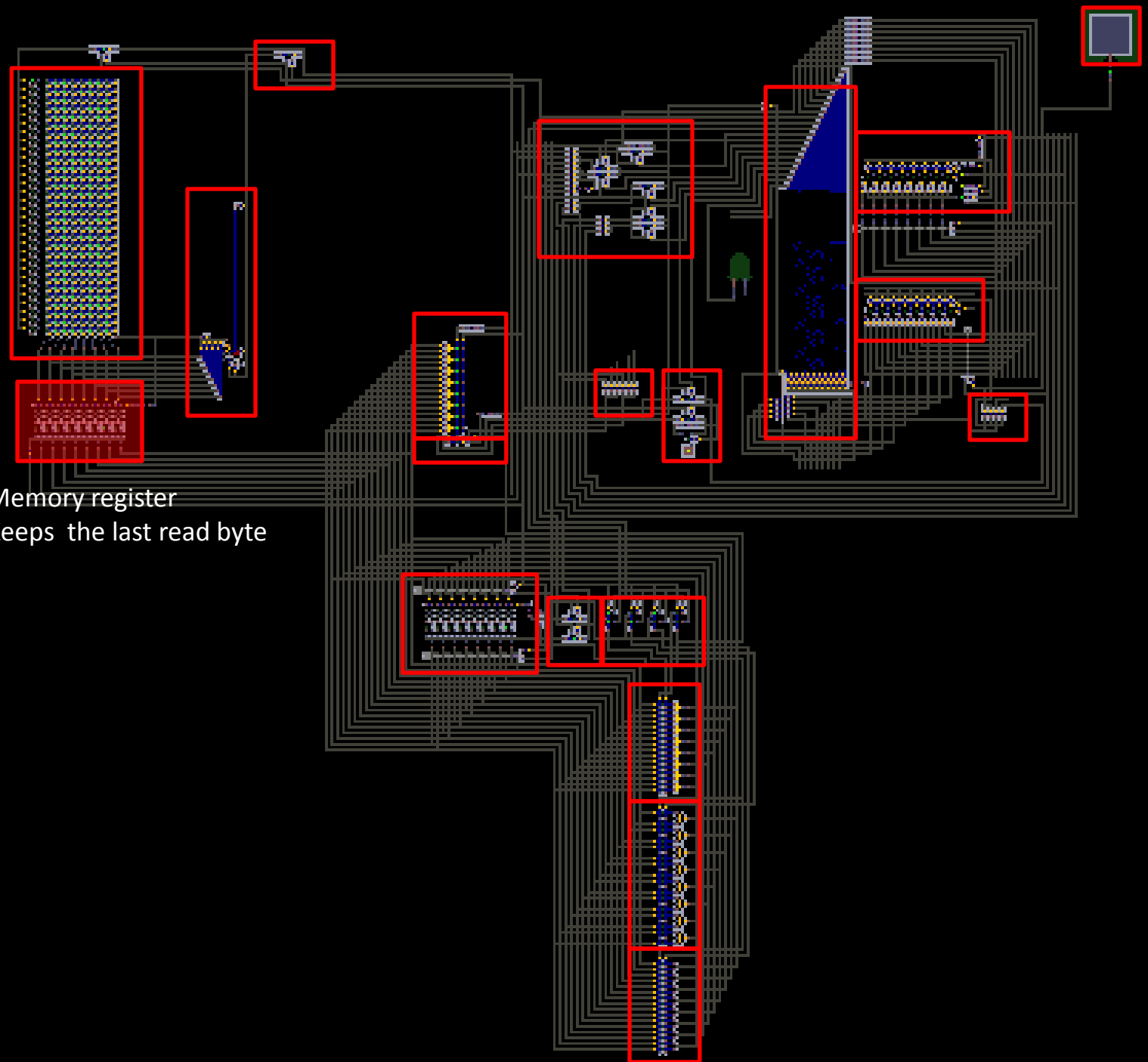
Some kind of shift register:
-Bray pass
-switch self turn off
-spark turn on next switch
(incremental memory reading)

Last bit indicate data-ready to be written in register;
This memory is time dependent:
-first address is the slowest reading
-last one is the fastest

-cpu clock has been selected to be compatible with first read.

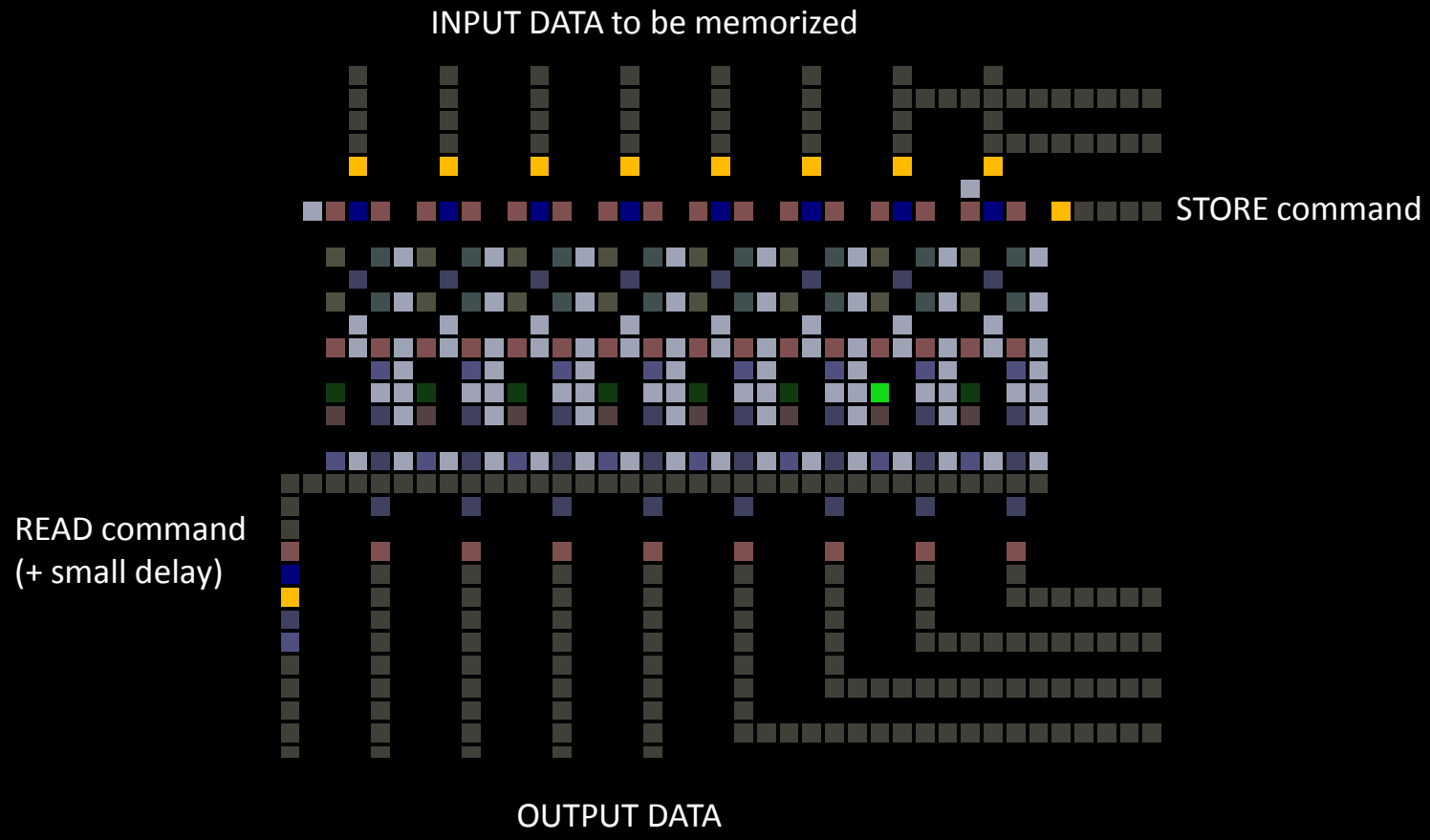
Single bit rom example

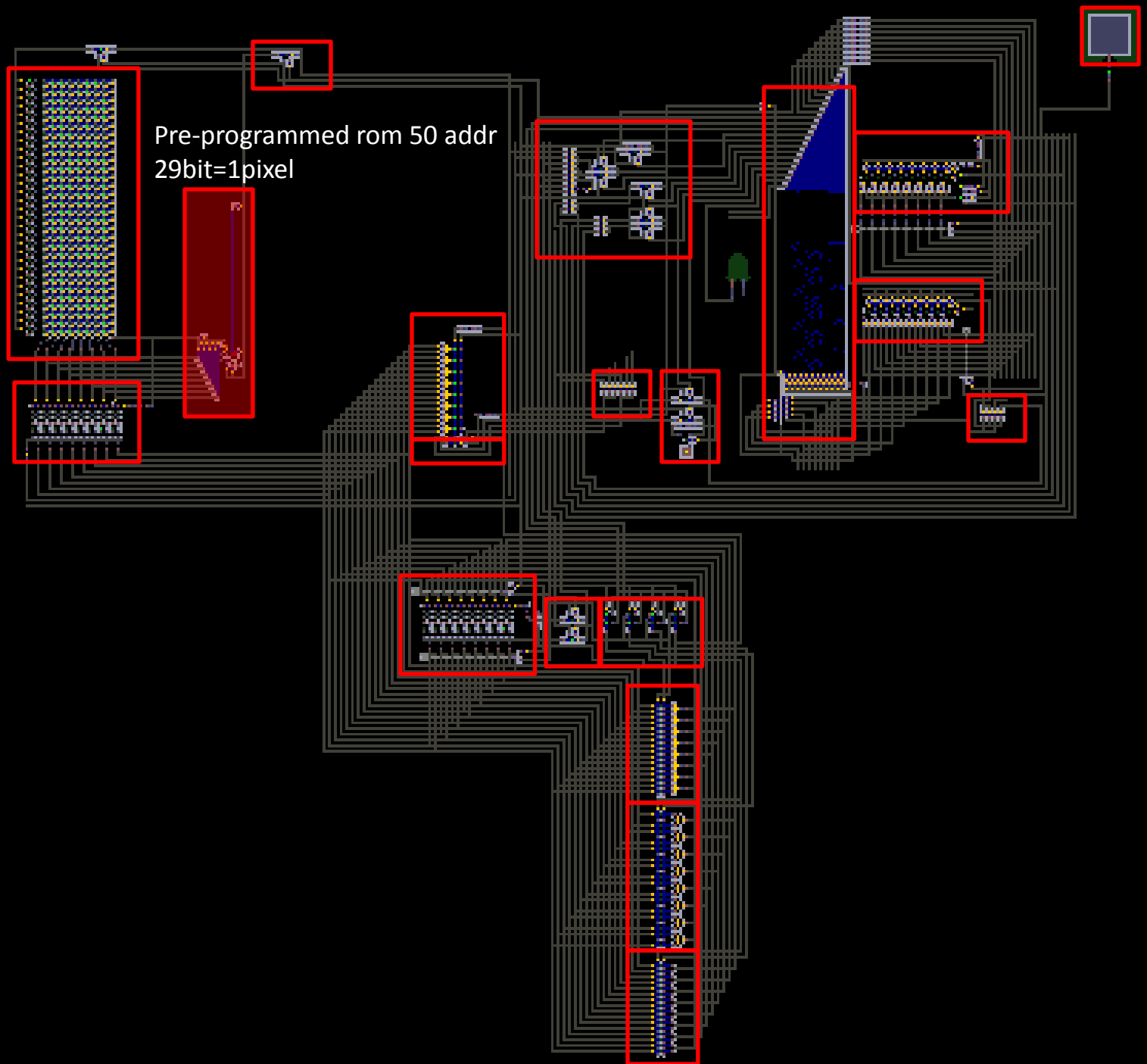
3 bit one line/address example



Memory register
Keeps the last read byte

Quite old register, stackable found online i started with this and planned to replace it with my/better version but ended keeping this due to timings problem in replacing it.
Uses PTCT / NTCT so to work correctly ambient heat must be OFF.
Input data bray heats (or not) the METL in center.
Store command sparks the PSCN and spark goes to left or right side based on the temperature.
One side switch on SWCH other switch it off.




Pre-programmed rom 50 addr
29bit=1pixel

To register store data command

DATA (encoded as ctype in FILT)

READ command (also activate PSTN to go to the next byte)

Line of FILT (get spectra)

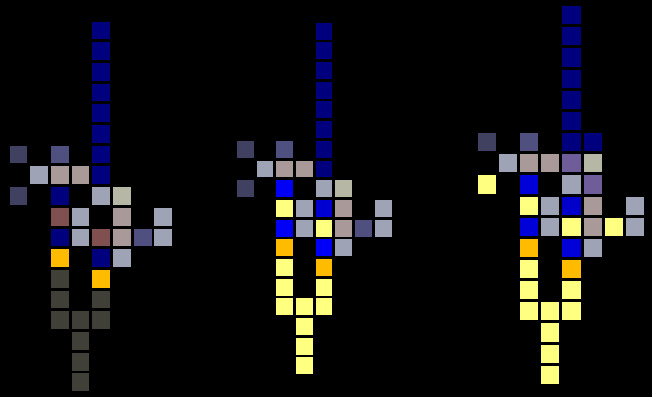
Line of FILT (AND)

1 FILT (no effect)

1 DTEC (BRAY)

red FILT =dummy

Pistons to move right and than up the filter; Uses subframe



On file open pixel are placed and gets incremental ID:

```
Int id=0;
for each row
{
  for each column
  {
    PlacePixel(x, y, id);
    id++;
  }
}
```

Pixels are processed from lower to higher ID:

```
for each pixel_ID
{
  if (SomethingToDo()==true)
    DoSomehting();
}
```

Subframe: when more than one thing move/ gets sparked in a frame but is processed in different moments due to internal game behaviour. in the same frame happens this:

1-the upper piston (not visible here) move data column (FILT) down by one

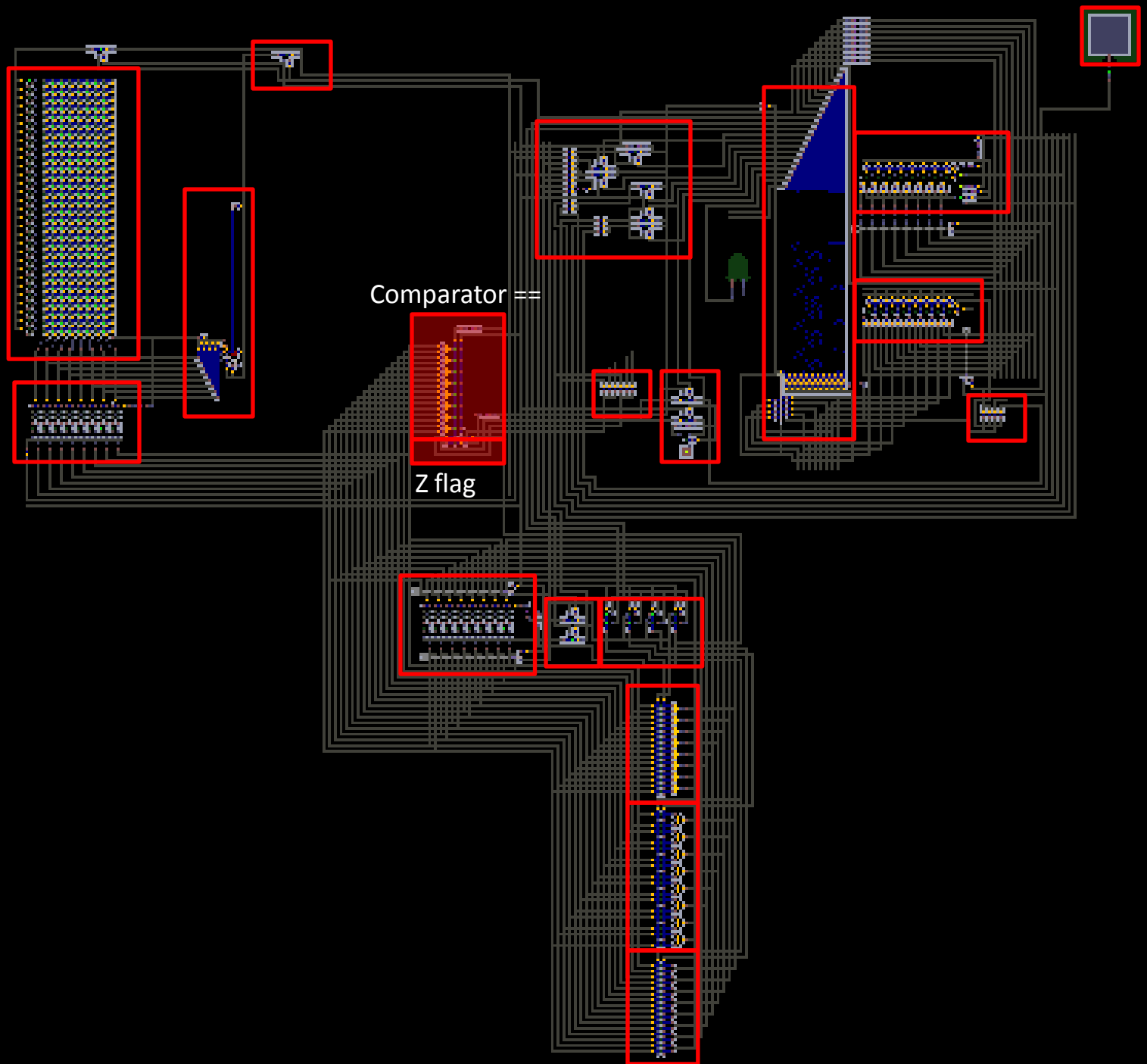
2-since the horizzontal piston is in a higher row than the vertical one it will be processed first and so the filt is moved right

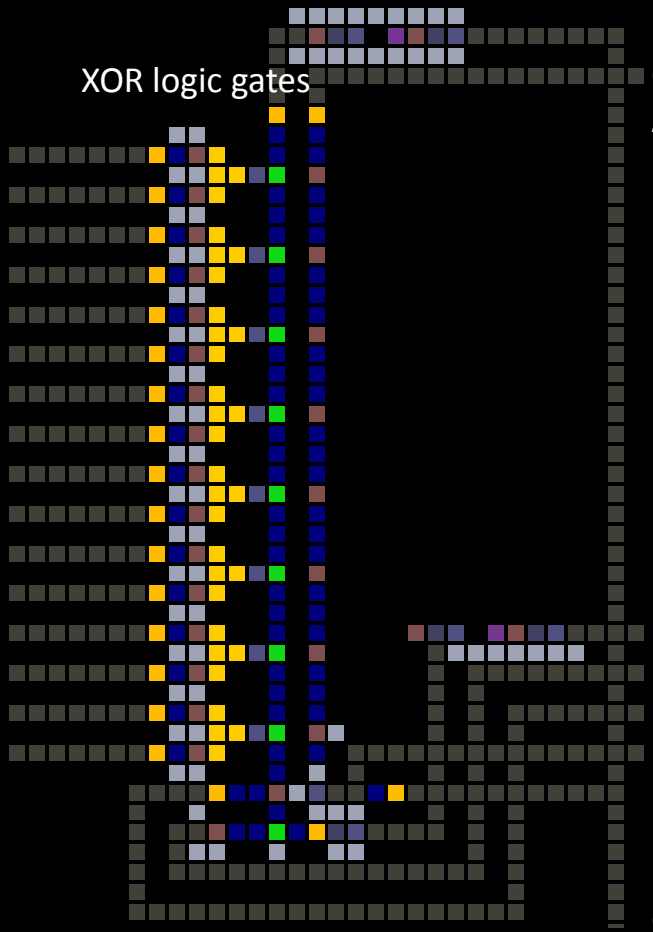
3-now is vertical piston turn that push it up

All in the same frame! (if you rotate this structure it will not work)

Three pistons push in different directions (down->right->up) in the same frame and everything works only because game process theese in different moment (from lower pixel ID to higher).

Also if you paint this by hand it will probably not work due to different ID, but if you save and reload it will work.


Comparator ==
Z flag



XOR logic gates

Data bus line 1
Data bus line 2
(both bit 0)

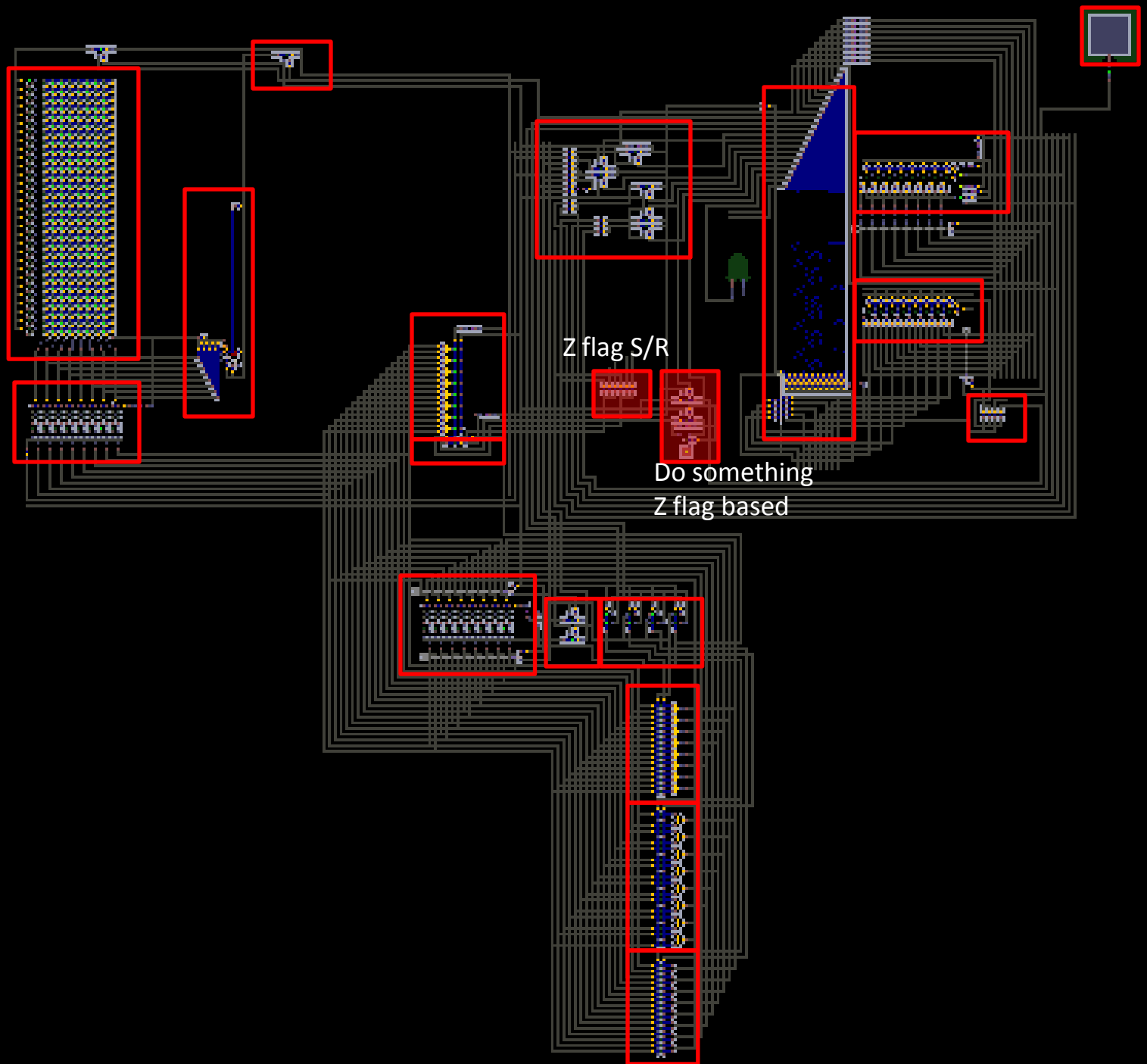
INPUT data
Two data lines
8 bit each

Connected to RESET line;
Allows a new compare operation

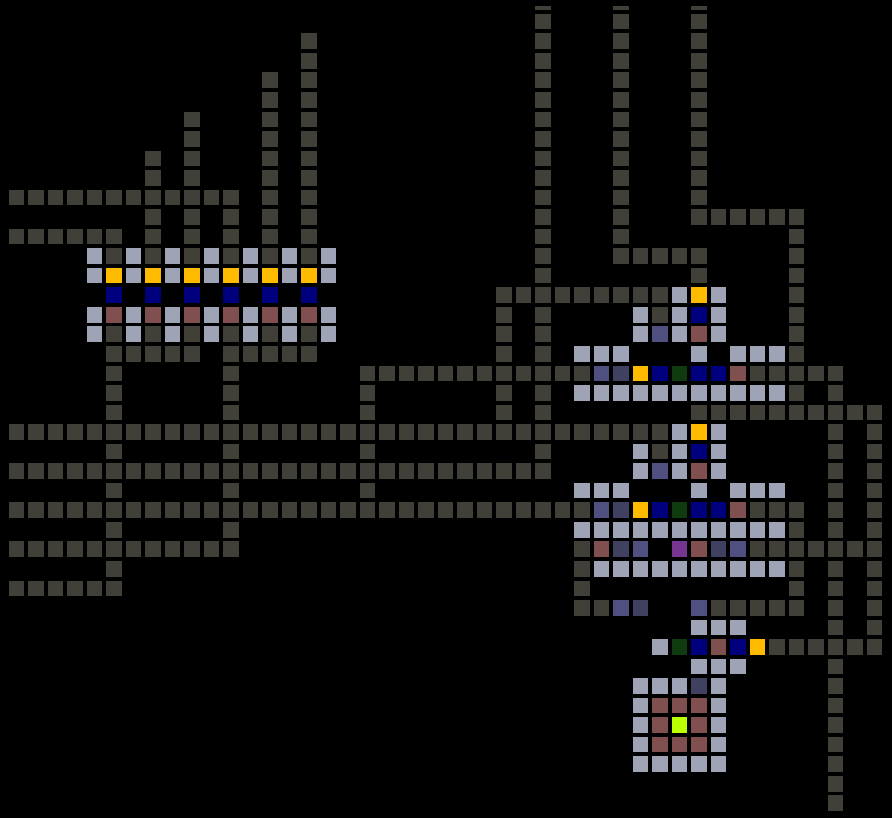
READ Zflag signal+delay (connected to CPU Clock)
Zflag output/status (after read)
SET Zflag (from other commands
CLEAR Zflag (from other commands

COMPARE signal:
-reset Z flag (SWCH down)
-spark ray up
If all swch are ON data is equal and ray
gows down and turn on the last SWCH
If data is different it pass in the logic gate
and enable the NSCN that turn off a SWCH
so Z flag will not be sparked

READ Zflag has a delay so that it is
read AFTER all computations are done;
in this way we read it after it has been
set/cleared.


Z flag S/R
Do something
Z flag based

(half unused) signal joiner: it allows multiple input to single output without interference (something like a diode array)
It was intended to be used to set and reset Z flag from different commands
but so far only one has been used: decfsz (cmp has its own pair of wires)
That instruction decrement a counter (set with other instruction) and set Zflag if the result is zero. Otherwise clear zflag

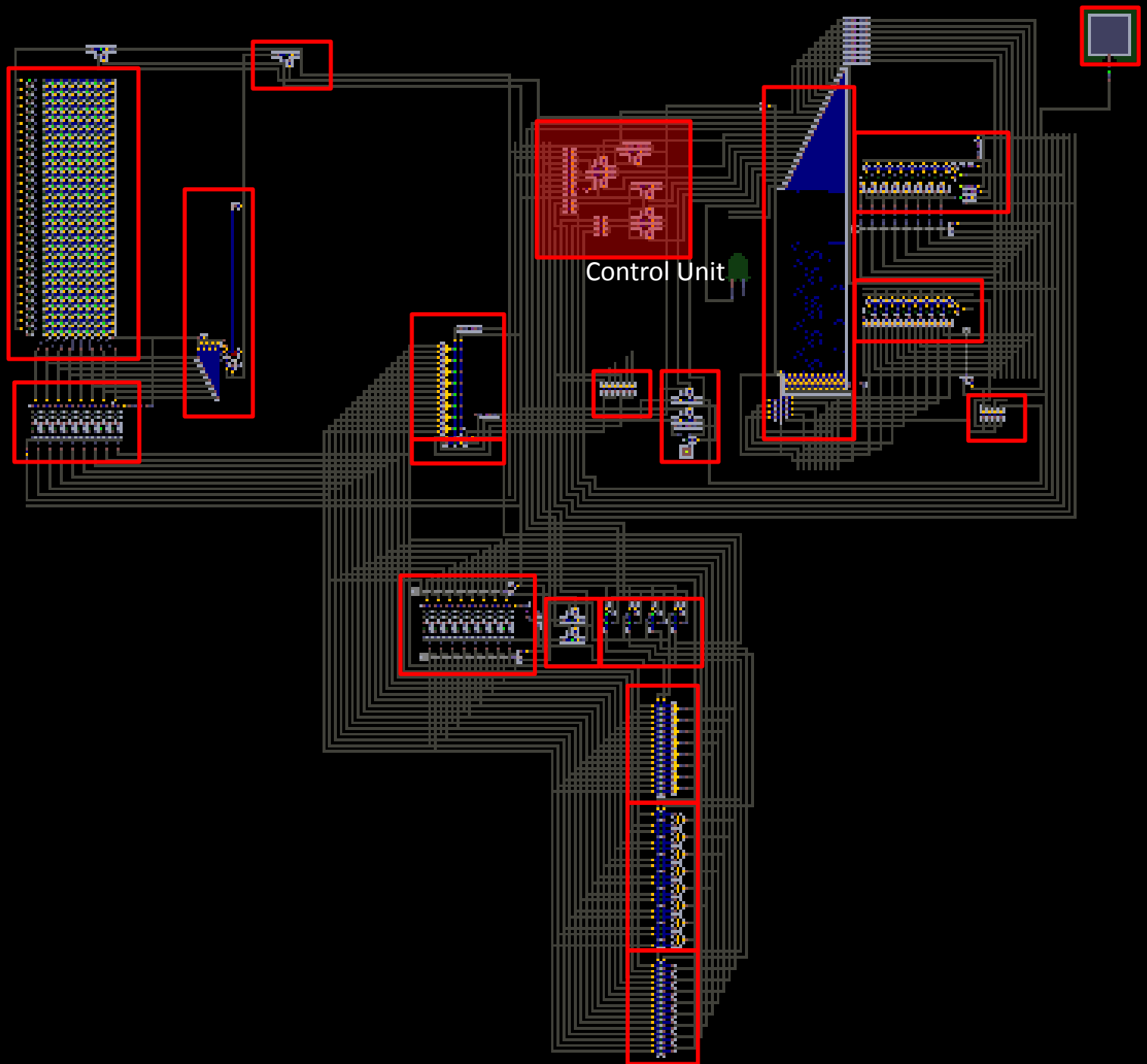


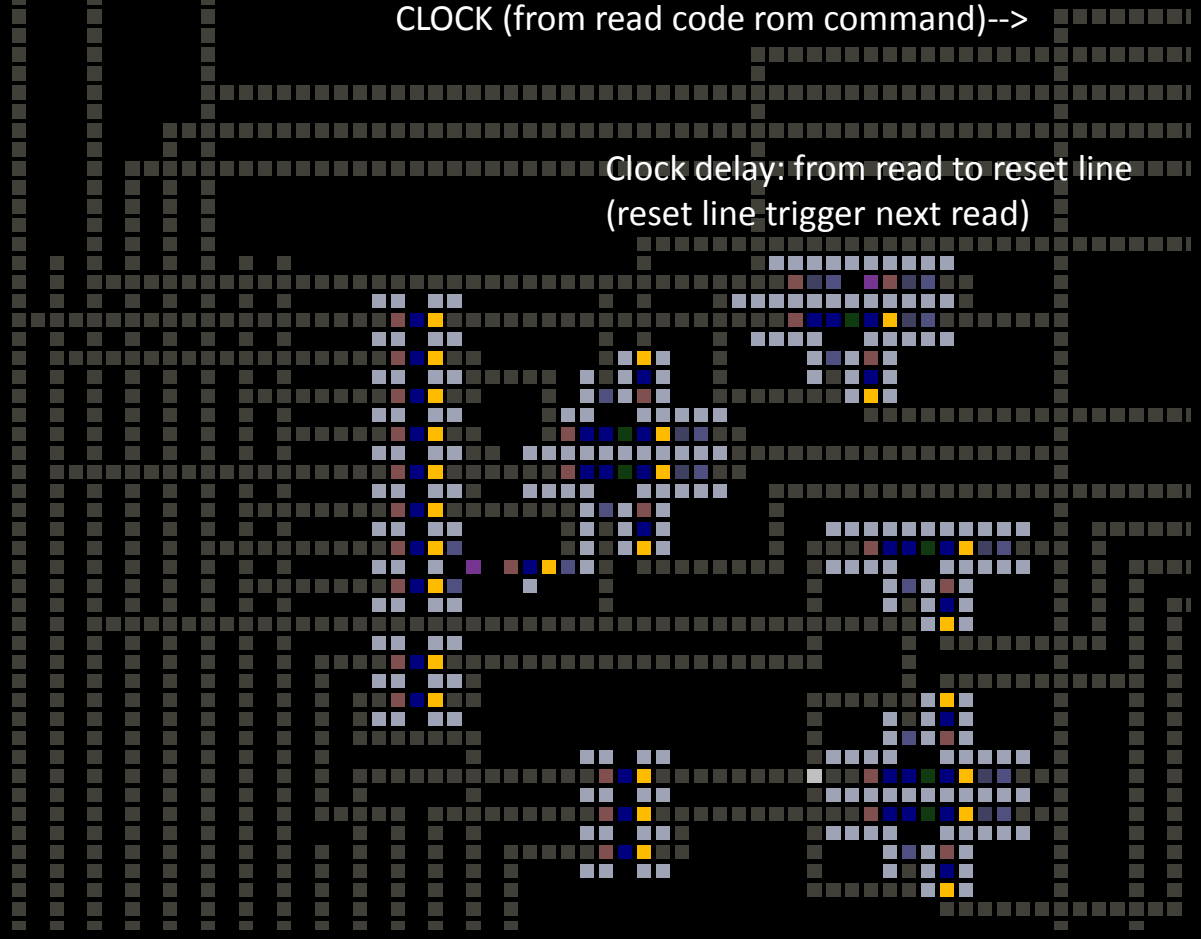
Small part of control unit designed to do something based on the Zflag:
-first one increment IP if zflag is set:
Switch is turned on by decfsz instruction (up) and zflag result goes from left to right.
If zflag was set signal will be present and will also pass due to correct instruction.

-Halt and catch fire if not equal
Same as above:
-“port” Switch turned on by instruction
-self destruction signal delay is activated
-self destruction switch is turned on

if zflag is not set noone will clear the self destruct switch and when delay finish the delay signal will destroy the pc: CRAY emit 8 singularity particles (distant) that will eat the pc.

If zflag is present it will win the race condition and clear the self destruct switch before the delay expire.

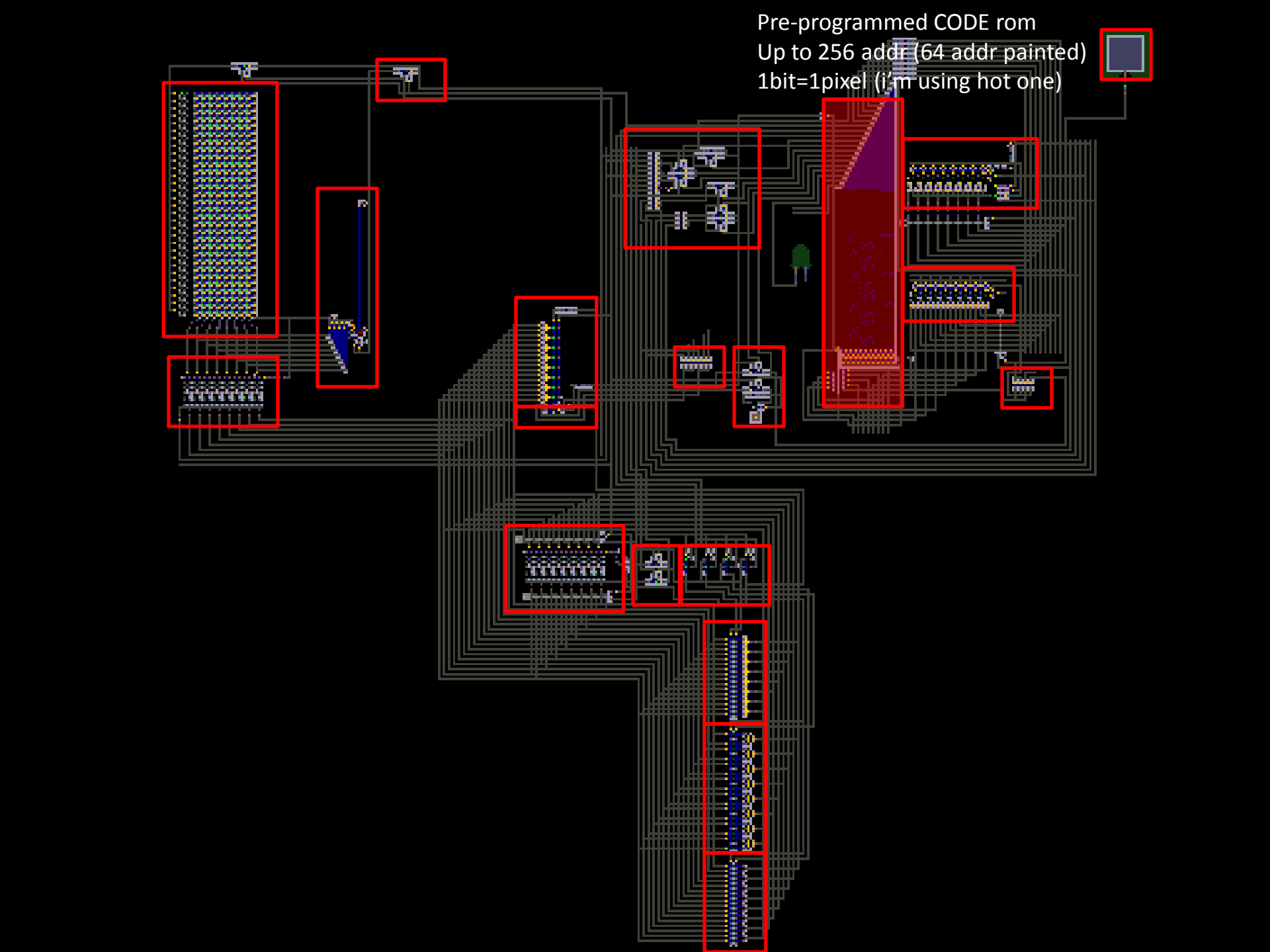

Control Unit

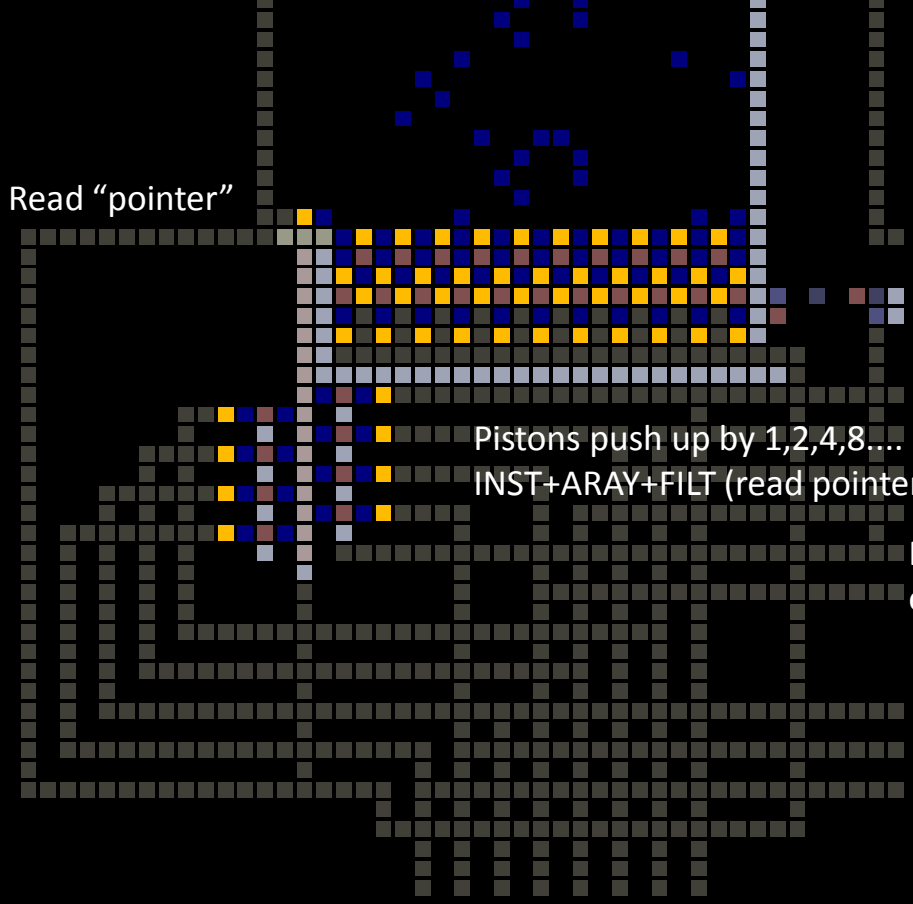


Clock delay: from read to reset line
(reset line trigger next read)

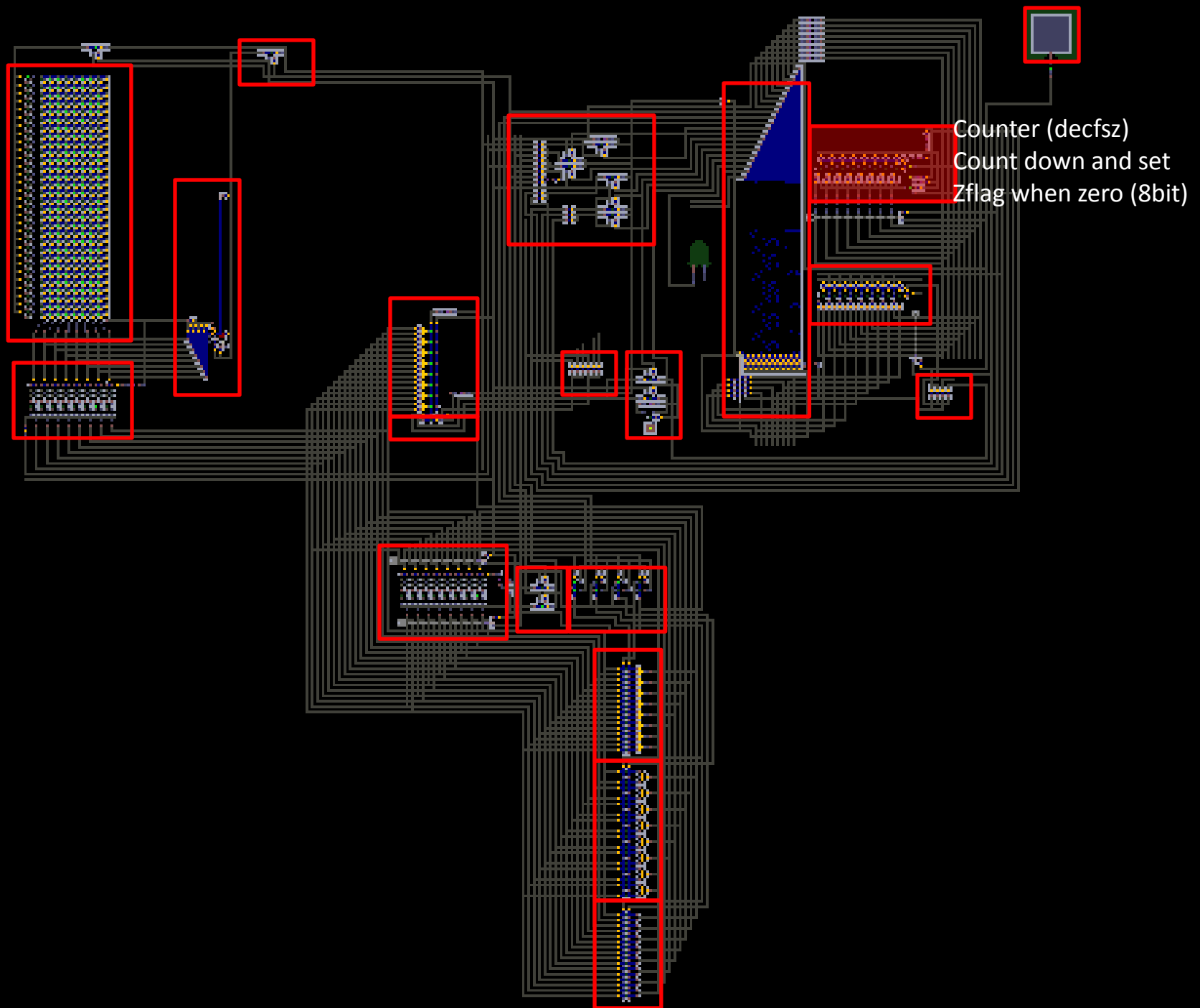
From code ROM (instructions)
Some wires are directly connected to the command bus, others pass through the control unit.
Directly connected (faster):
-Reset Line
-main register R/W mode shared with memory bank selector
-bin decoder (de mux) mode (operation selector)
-turn on led (there is not much to do)
In Control unit:
-read memory
-mem register to main register
-ALU instruction
-set counter(for decfsz)
-decfsz (decrement file skip if zero, inspired from PIC microcontroller)
-jmp (unconditional)
-HCFNE halt and catch fire if not equal (port near Zflag zone)

Clock goes to the input of each port; ports are enabled by code rom bits; each port represent an instruction;
ports output goes to data diode that turn on necessary pieces for that instruction.
first port (top) is read memory and turn on only read memory signal (bank select is connected directly to code rom).
While second turn on both read mem reg and (generic) turn on main register; write mode is set directly from rom code.
Probably everything works well also without this kind of ports and the clock concept; just a small delay to the data diodes.
ALU port does this: main reg read + mem reg read + set operation signal (demux was set before directly); wreg write (after operation is completed.
Set counter: clear counter+enable write (value comes directly from code rom)
Decfsz: enable the counter to count down
Jmp: clear IP + enable write (value comes directly from code rom)
HCFNE: send destroy signal delayed and turn on destroy component, component is turned off by Zflag signal (if present)

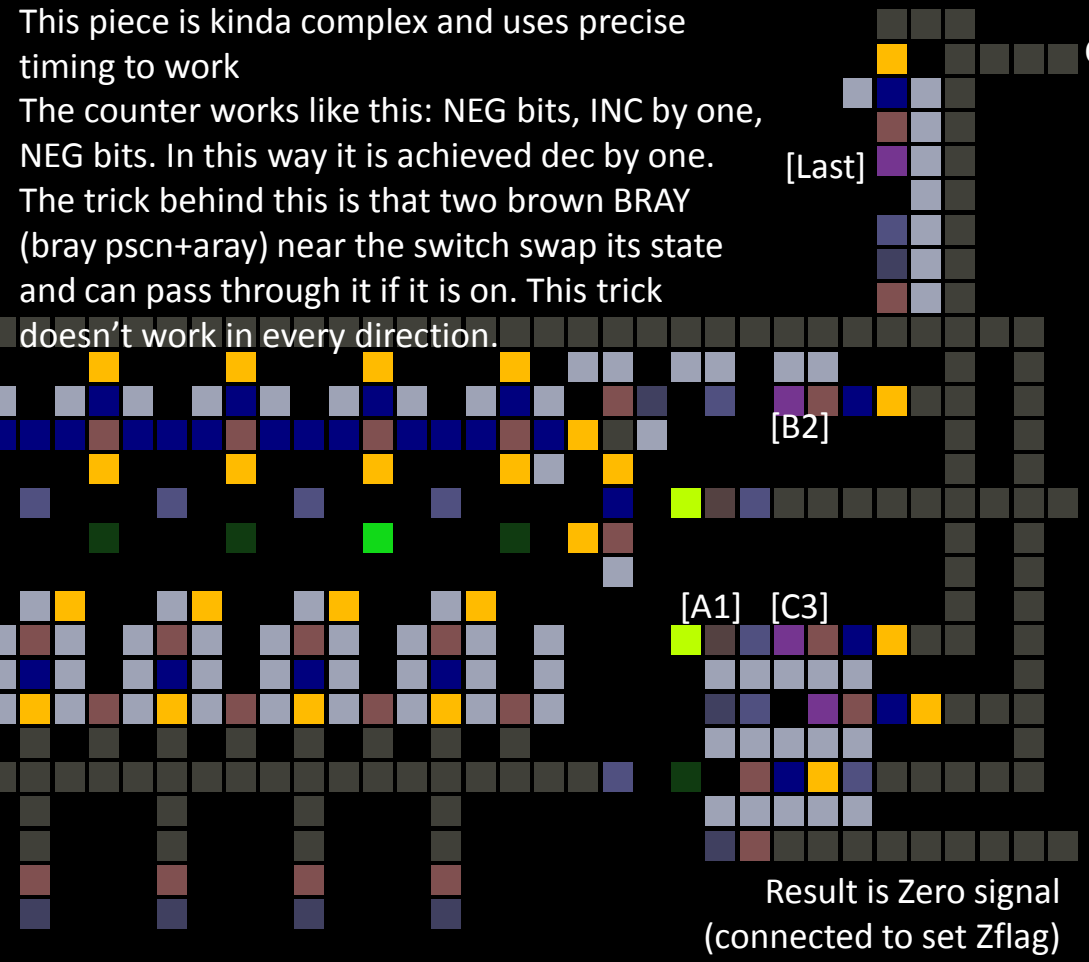

Pre-programmed CODE rom
Up to 256 addr (64 addr painted)
1bit=1pixel (i'm using hot one)



Also this memory uses subframe:
Read pointer and aray block are sparked and must emit a BRAY but since read pointer is in a higher position than the aray block is processed first and emit a ray.
The ray will block vertical rays (except where there is a FILT, in that point BRAY doesn’t interfere so vertical ray can pass.
In the same frame (but after the read pointer) the aray block emit a vertical BRAY in the whole memory but it can pass only where filter is present because of read pointer bray that block rays.
Still in the same frame the piston reset (connected to read pointer and aray block) is sparked and the pointer goes down again). Up there are PSCN that catch the rays.

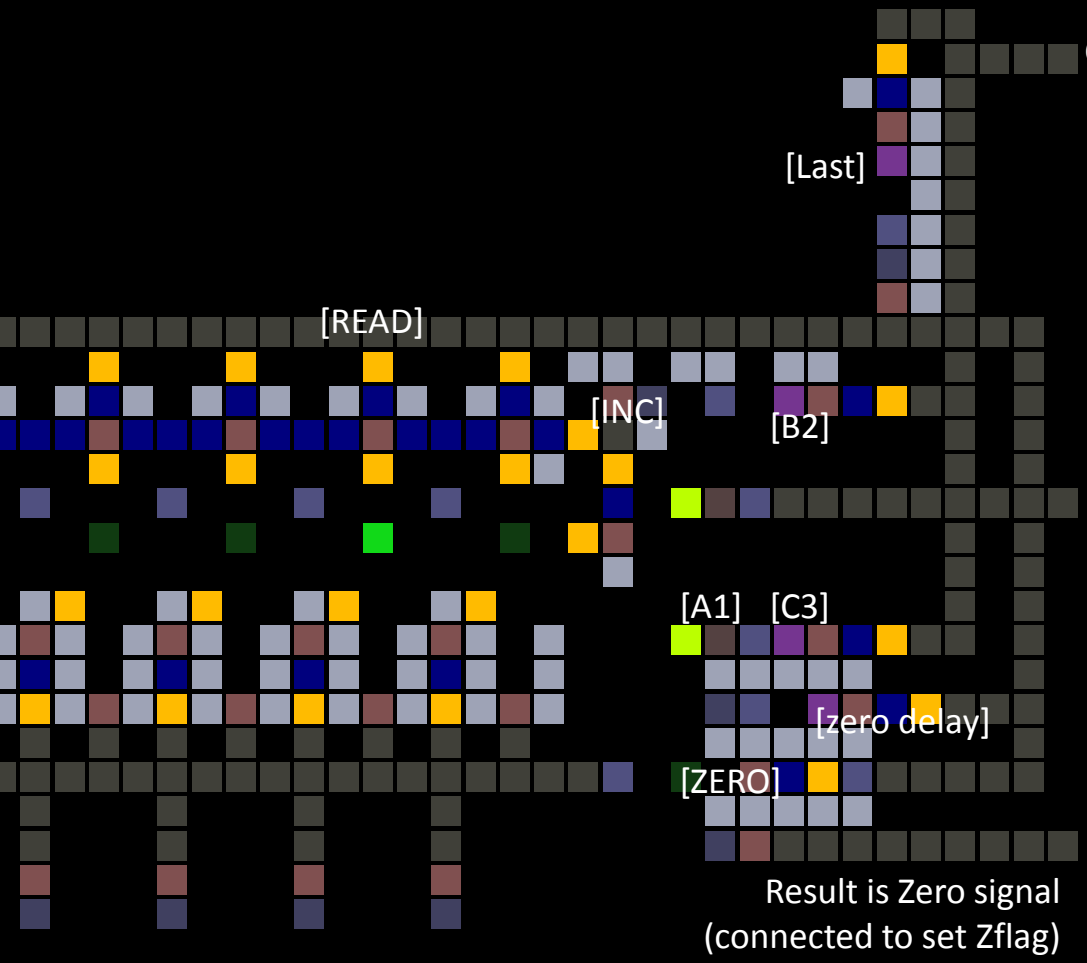

Counter (decfsz)
Count down and set
Zflag when zero (8bit)

This piece is kinda complex and uses precise timing to work
The counter works like this: NEG bits, INC by one, NEG bits. In this way it is achieved dec by one.
The trick behind this is that two brown BRAY (bray pscn+aray) near the switch swap its state and can pass through it if it is on. This trick doesn't work in every direction.



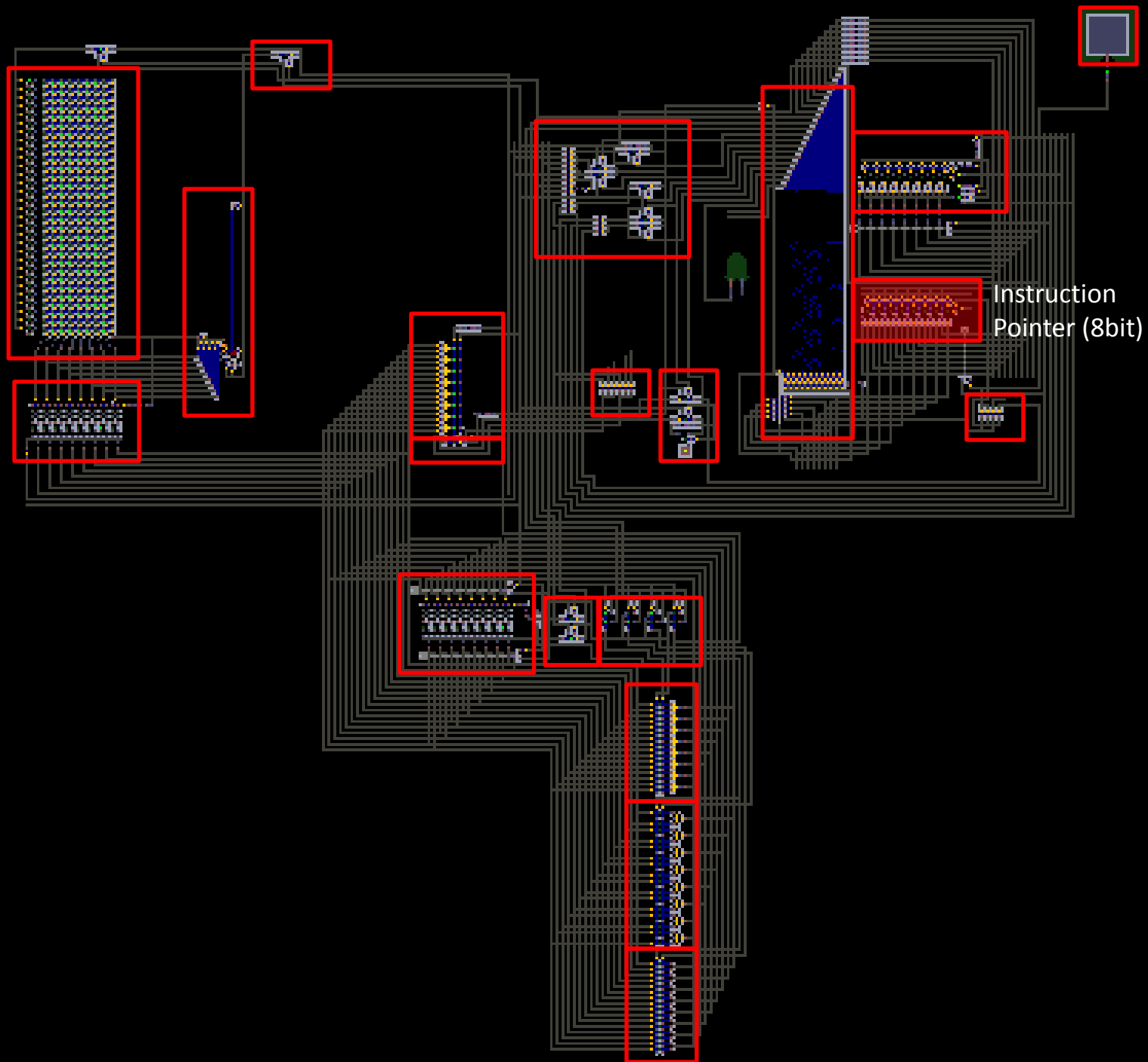
Set custom value bits

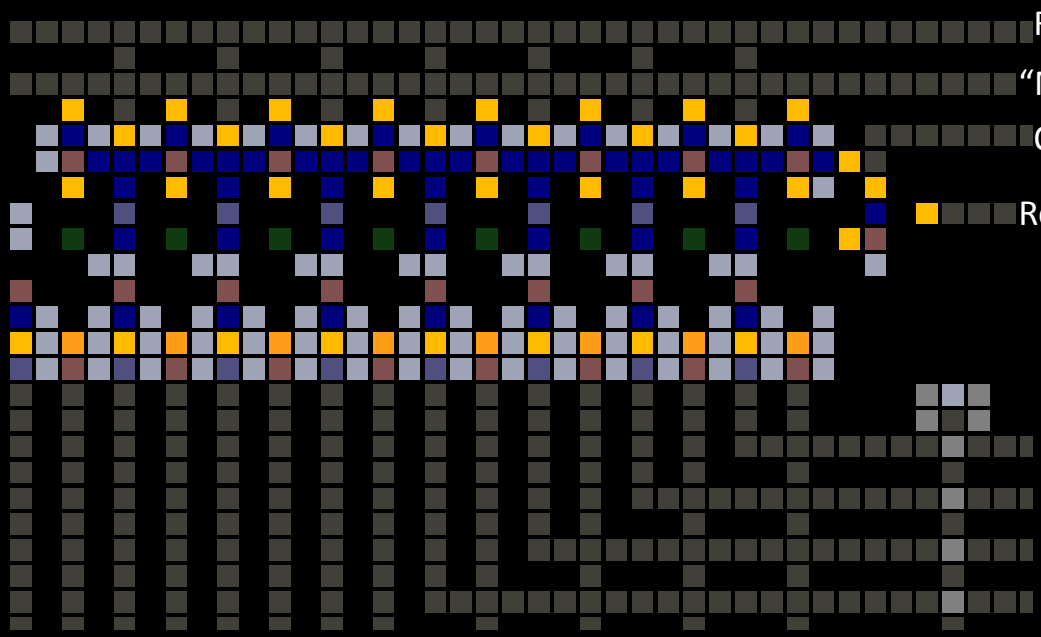
when you spark count down the first things that are activated are:
-the up delay (long) ; this is the last thing that will be activated [Last]
-since the pscn and nscn are spaced only by one signal can pass over the delay (but delay has been sparked and will work later, so there will be double spark on this line) [A1]
-increment by one line is sparked (short delay) [B2]
-the second delay is sparked (medium delay) [C3]
The [A1] spark will invert all bits (NEG), this happens because CRAY(spark) can pass through every material and will spark pscn that will make 45° brown BRAY that will invert swich state. (the same line is used, after a reset, to set a custom value to the counter).



Set custom value bits

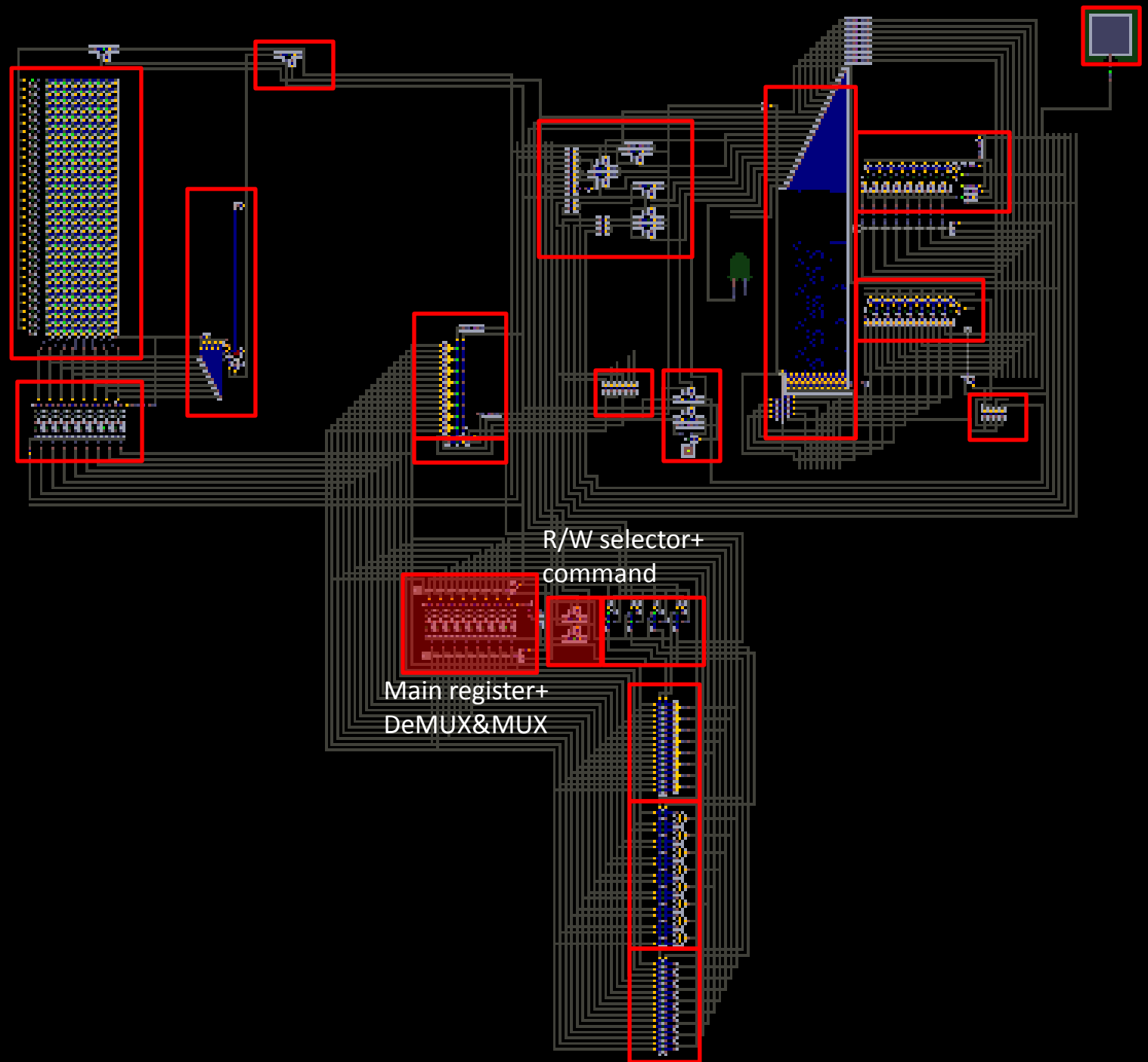
Right after this B2 line will be active and the spark will be at [INC] so inst will spark vertical and horizzontal pscn and they will spark aray that will cause inc by one. This happens because from top all switch are sparked but from right it will pass (and change state) through all ON switches and also changhe state to the most right inactive switch (binary counter).
After this [C3] delay expire and [A1] is sparked again so bit are NEGated again so the dec part is done.
[Last] delay expire and will spark [READ] line, like the code memory the double aray will read and clear ray.
At the same time [ZERO] switch is turned on and a [zero delay] above is sparked.
If the counter is not zero its output bits connected in OR will go to NSCN and turn off the switch.
If the counter is zero nothing will turn off the switch and when the delay expire it will spark the “result is zero signal”
In the same way as self destruct works. All this in 22 frames!

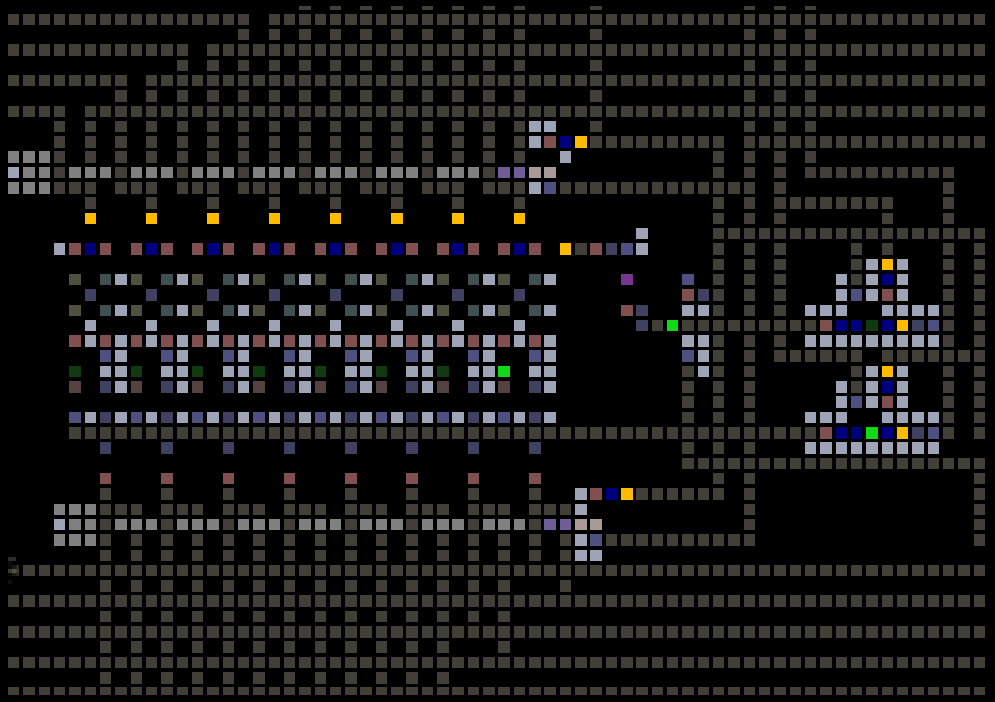

Instruction
Pointer (8bit)



Output data.

Data is read automatically when inc is sparked; this happens because near pscn there are DTEC(bray) that detect brown bray. To set manual read you can delete dtect and use read command. (and you can add dtect in counter)
This counter is similar to the decfsz one. It uses the brown bray trick to change switch state.

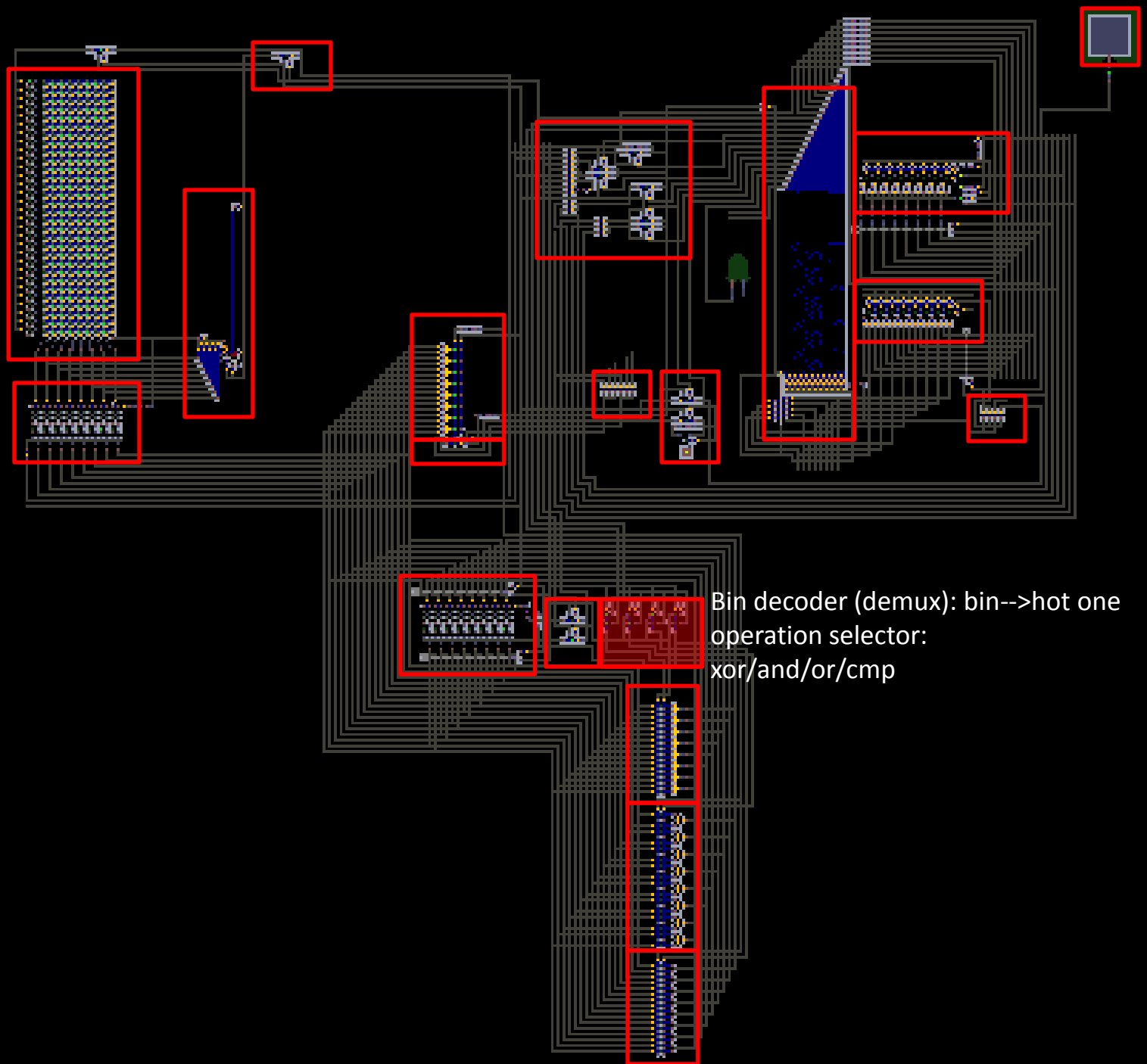

R/W selector+
command
Main register+
DeMUX&MUX

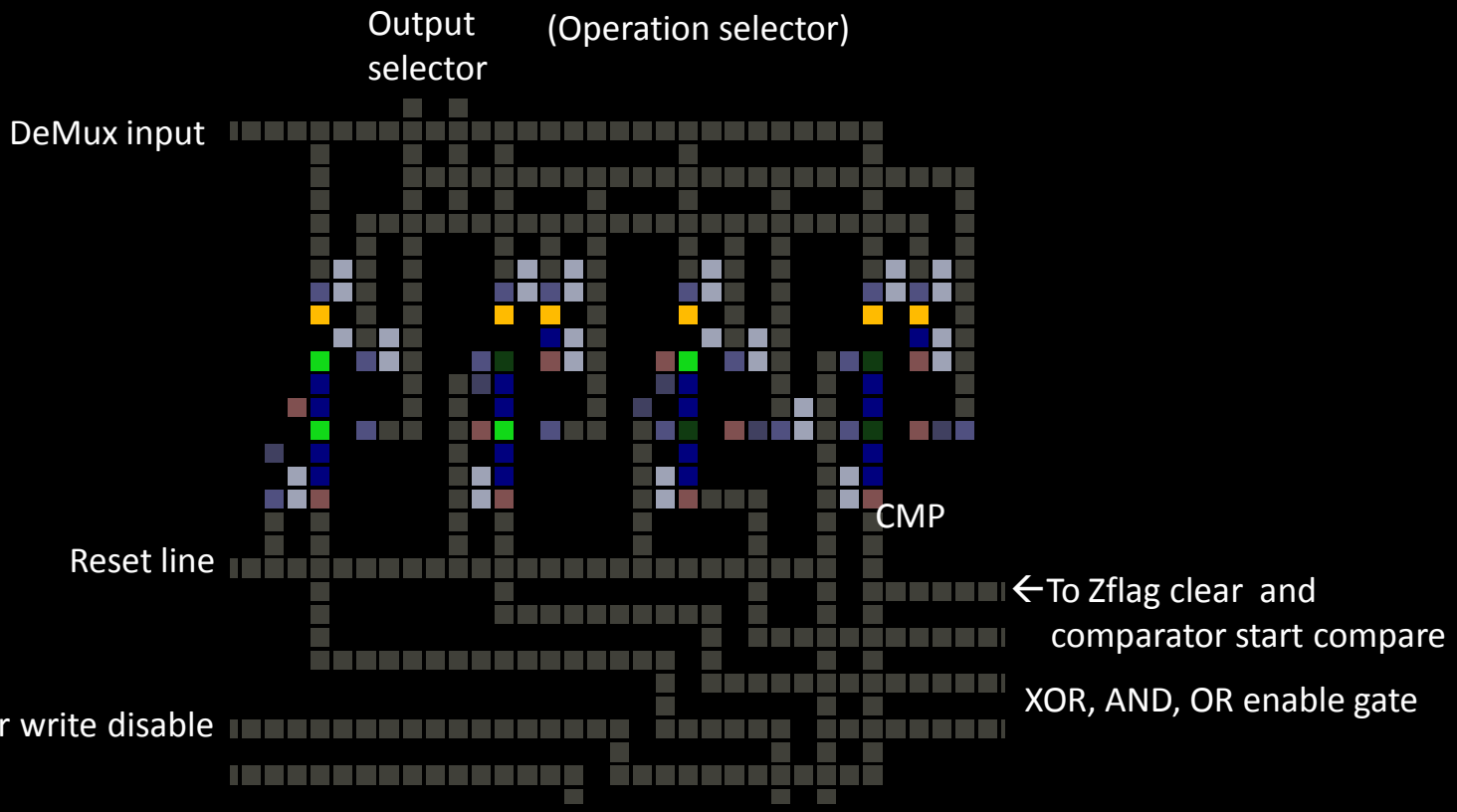


The main register is similar to the memory register. As extra it has two ports (on right) that select the mode of operation: read or write (1 bit). There is the execute/enable bit (that will read or write).
There is an extra switch connected to the reset line and cmp (so cmp will not write on main register).
Above and under the main register there are two piston based (0 delay) multiplexer.

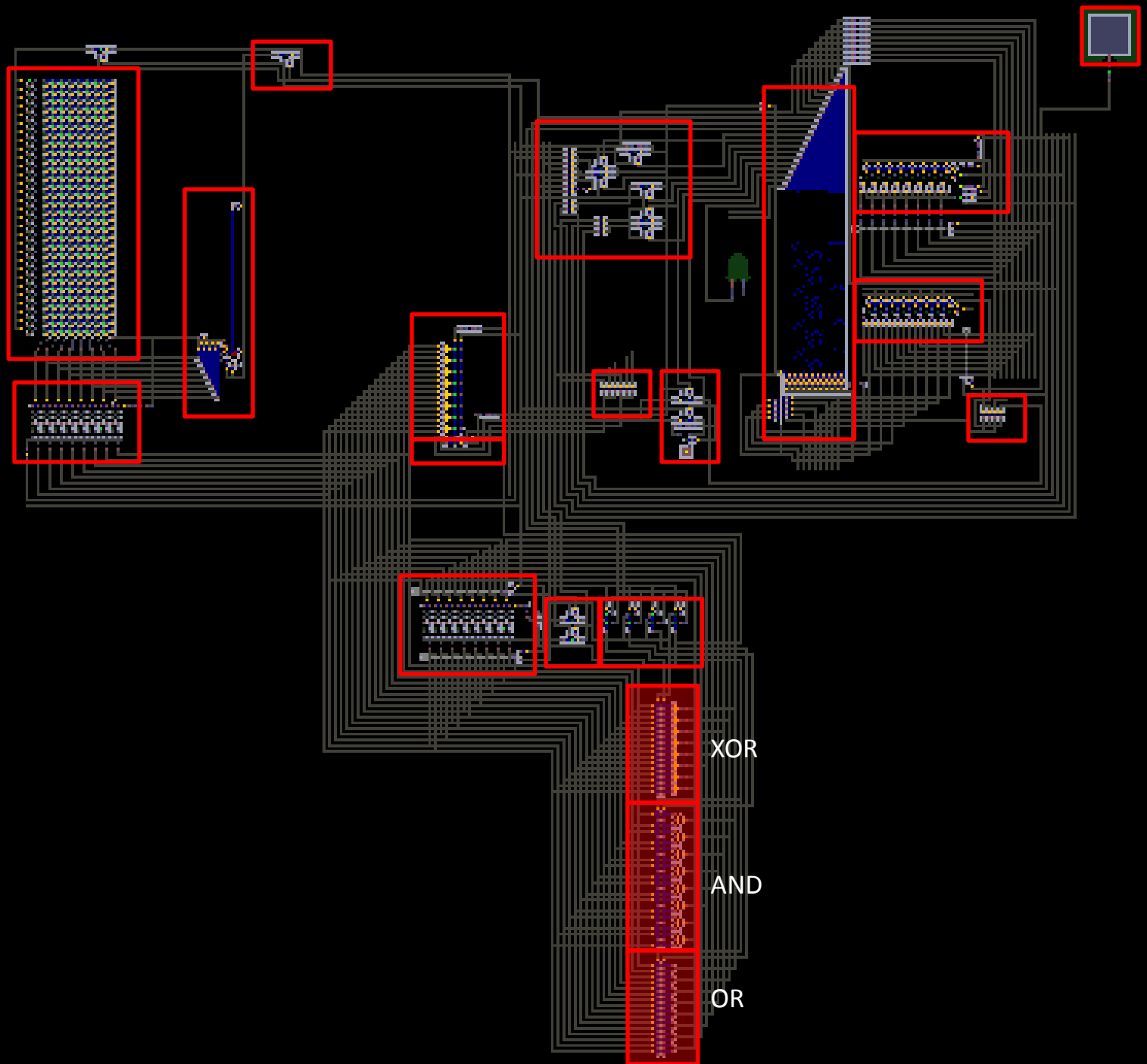
In case of normal operation input and output of the register are connected together and to the data bus.
In this way you can store in the register the data present on the bus (write mode+execute/enable).
Or read the data (useless since we have no ram or other registers.

When there is an operation “alu mode” is enabled and the multiplexers is switched on both sides (up and down).
Now the output is connected to inputB of logic gates (instead of inputA where memory register always is).
Input is connected to logic operations output.
Data is read from main and memory register; both goes to input gates (cmp/and/or/xor) using two separate buses.
And its output goes again to the main register which is overwritten with the new value. Precise timing allows this operation without interferences.


Bin decoder (demux): bin-->hot one
operation selector:
xor/and/or/cmp

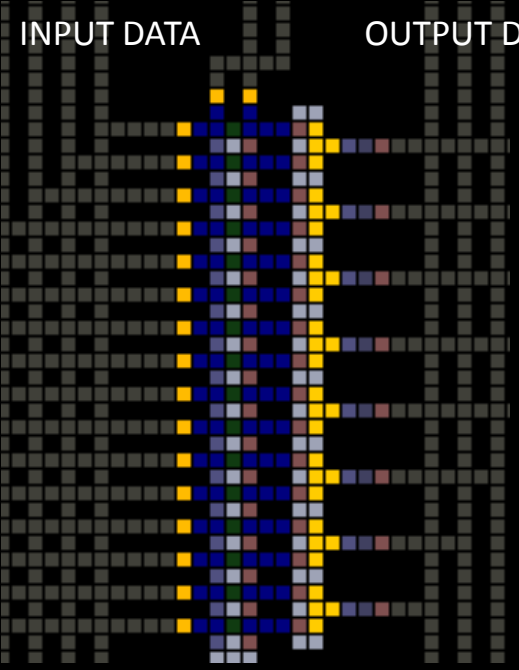


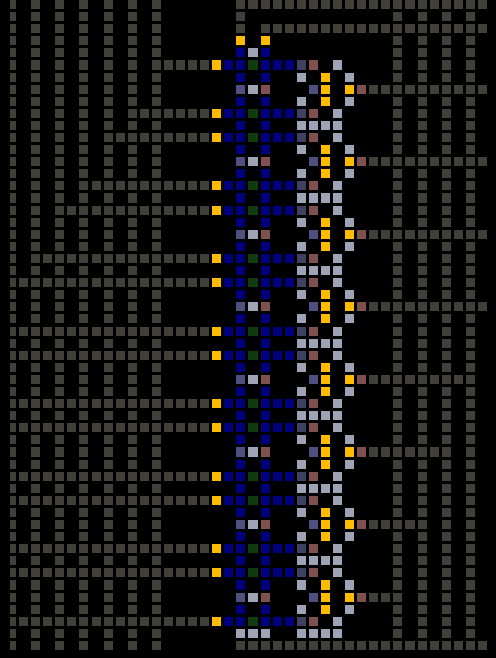
cmp doesn’t write but control unit will always issue write command. That wire prevent it

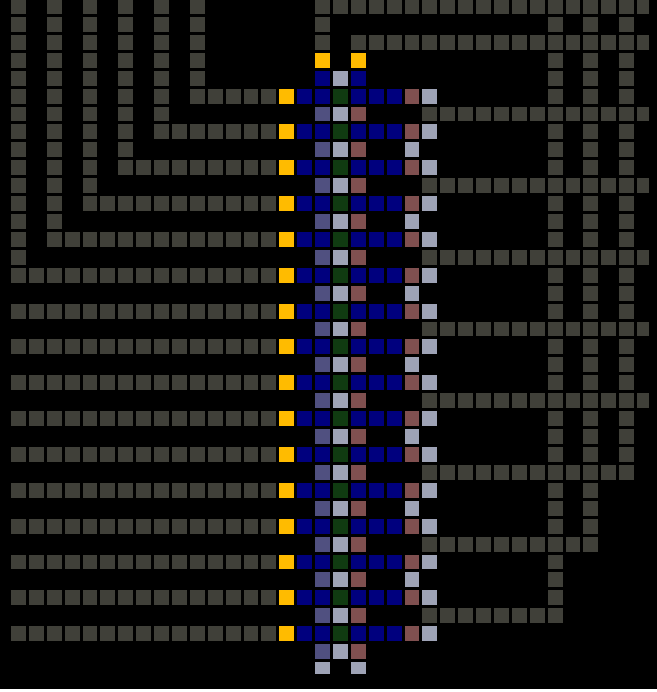

XOR
AND
OR

INPUT DATA

OUTPUT DATA





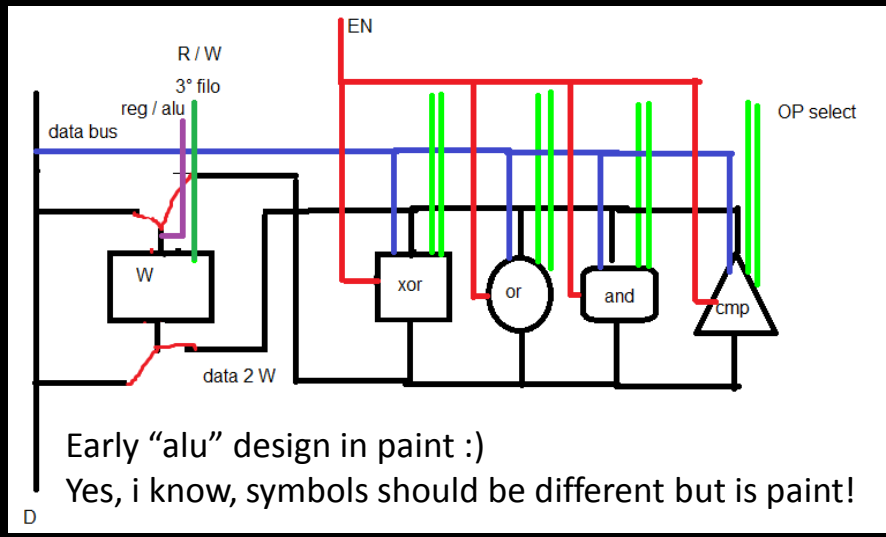


Data enable gate

XOR gate

There are 8 gates (one for each bit)

All three gates work in the same way:
Input data on left
Gate that allows (or not) the data to pass (from operation select)
Actual logic gate (xor/and/or)
Output data



Early “alu” design in paint :)
Yes, i know, symbols should be different but is paint!

Input data goes to all three ports (and comparator) but can pass only in one (the enabled one)
Input data comes from memory register and main register
Output data is common for all three ports and is connected to main register through multiplexer

flag{d0nt_you-l1ke-f.s.au7oma}

f	1100110=102			
l	1101100=108			
a	1100001=97			
g	1100111=103			
{	1111011=123			
		key:	result:	
d	1100100=100	h 104	12	
0	0110000=48	f 102	86	
n	1101110=110	t 116	26	
t	1110100=116	p 112	4	
_	1011111=95	t 116	43	
y	1111001=121	c 99	26	
o	1101111=111	p 112	31	
u	1110101=117	u 117	0 --> 256 (filter fix)	
		mem:	and:	or:
-	0101101=45	n 110	53	41
l	1101100=108	e 101	230	12
1	0110001=49	s 115	0	49 --> 256 (filter fix)
k	1101011=107	o 111	123	0 --> 256 (filter fix)
e	1100101=101	s 115	101	101
-	0101101=45	! 33	15	44
		key:		
f	1100110=102	72		
.	0101110=46	93		
s	1110011=115	93		
.	0101110=46	79		
a	1100001=97	20		
u	1110101=117	66		
7	0110111=55	88		
o	1101111=111	2		
m	1101101=109	12		
a	1100001=97	28		
}	1111101=125	[125]	--> final check to ensure only one solution;	
			also if should be obvious since flags also ends with }	