MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION

FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION

“NOVOSIBIRSK NATIONAL RESEARCH UNIVERSITY STATE UNIVERSITY”

(NOVOSIBIRSK STATE UNIVERSITY, NSU)

15.03.06 - Mechatronics and Robotics

Focus (profile): Artificial intelligence

TERM PAPER

Authors:

Dondokov Artem

Kuznetsov Gleb

Medvedeva Daria

Job topic:

“Frogger”

Novosibirsk, 2023

TABLE OF CONTENTS

Introduction

The game "Frogger" was selected for our project. Our task was to make a copy of the famous game of the 1980s. We made a large colorful display of many matrices in Logisim, linked the control to the keyboard and wrote code on cdm8 for the game statistics.

Problem statement

Our tasks are announced in the "Technical specifications" file. It was important to make the game as similar as possible to the arcade games of the last century. It was also necessary to use the program Logisim to implement it, in which schemes had to be constructed and the playing field itself had to be made. Finally, the code had to be written in macro-assembler and connected correctly to the hardware.

Analogues

Our game is a copy of the original Frogger from 1981. The graphics are similar to Frogger on the Atari 2600. A player controls a frog that has one goal in life - to cross a road and a stream of water and get to his swamp. The game starts on the ground, where nothing can harm a frog, but over time it dries out and if the frog is out of the swamp for too long it will die. So you need to cross the road. The first track is a school bus, the next line is a car, the third is a bolide F1, after it rides an SUV, and the last one is a truck. Over the road, you come to the second ground field, where you can have a little rest, but don’t forget about the time. Then you have to overcome the river flow. Next to the shore are three turtles, but they really don’t like being stepped on, and I don't think anyone likes it. So they can sink into the water and you have to hurry up to move to the small log, then the big log, after it two turtles and a middle size log. Wow, you have finished. Right? So you must help the other 3 members of the Frogger's family. By the way, I think frogs are bad at arithmetics so.. Am.. Maybe they don’t mind if two frogs die. That’s finish, isn't it?

It turns out that you need to repeat the last two actions twice. But every time the traffic becomes faster.

Hardware

On the game board we have:

1. A display with fields, cars, logs and turtles and a frog.
2. A timer that shows how much time remains before a frog dies.
3. A current score.
4. The highest score.
5. A keyboard to control a toad.
6. The number of lives.

Game engine and its elements

The display resolution is 120x130 pixels. It contains 30x10 52 matrices in the *background layer* and the same quantity for *player layer*, where:

Display - union of 104 matrices;

Background layer - the lower layer of the display, there are drawing entities, like cars, turtles and logs.

Player layer - highest layer of the display, there is a drawing player (frog), without background (all turned off pixels are transparent).

Block - place in the display size 10x10 pixels, it's a basic element of the display, all models were made on this basis. For example, the frog model has a size of 10x10, every turtle also has this size.

Display shift - value of horizontal shift of entity on the display, it's a 2 bit value which takes values from 0-3 0b(00-11). It determines which of 4 matrix entities will be drawn.

Matrix shift - value of horizontal shift of entity on the matrix, it's a 5 bit value which takes values from 0-29 0b(00000-11101). It determines the distance between the first column of an entity and the beginning of a matrix.

Two things defined higher are the same for the *background layer* and *player layer*.

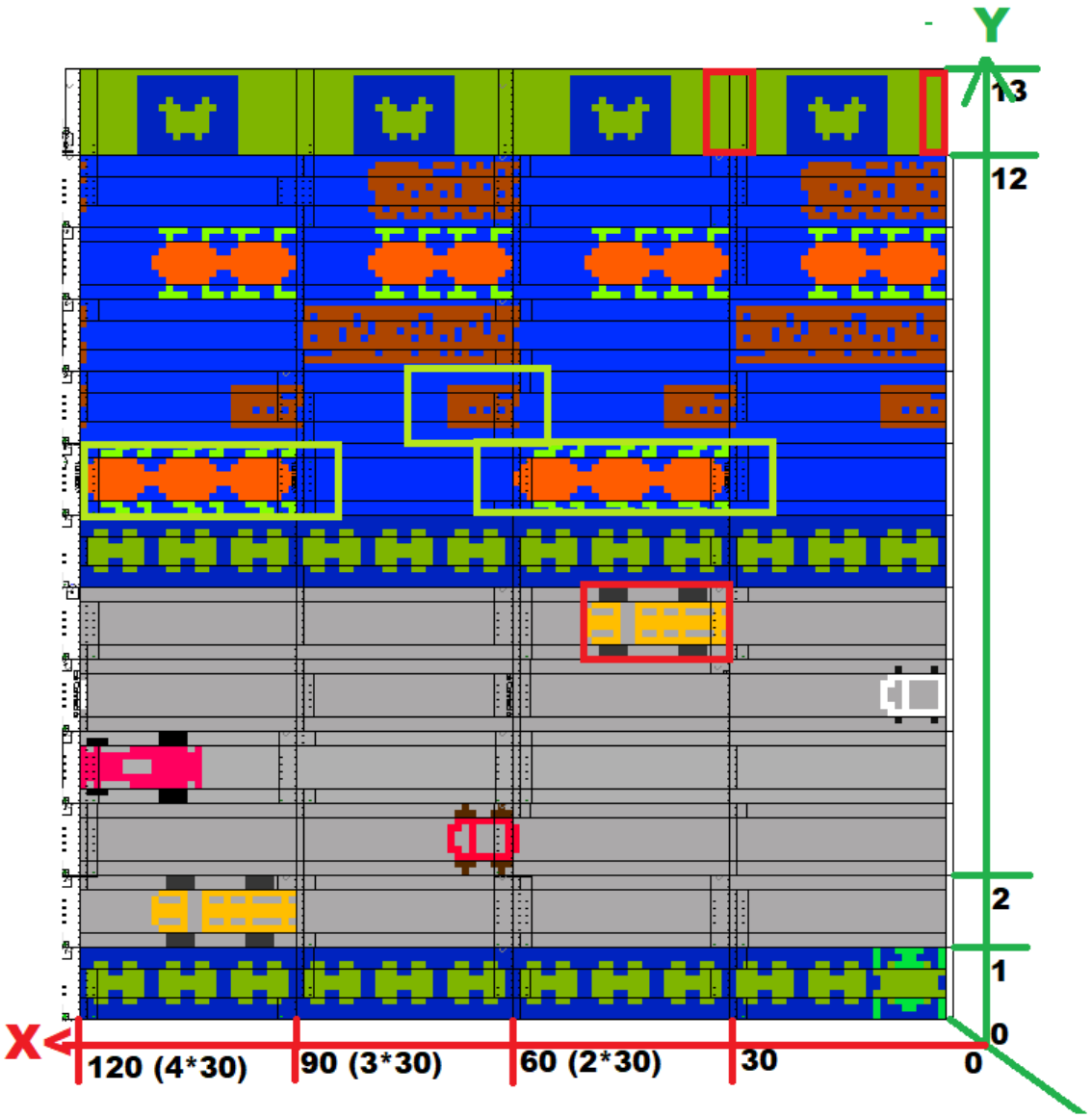
Matrix - matrix from the standard Logisim I/O library, in our case it has 30x10 pixels, or 1 row and 3 columns of blocks.

Display string - part of the display that consists of 1 row and 4 columns of matrices, it has resolution 120x10.

Main trick for so many colors in the display

First of all, the many colors of the background layer. Every string can contain some sub-matrices with the same width and the same sum of heights, but everything has a different shiny color and the same background color.

Secondly, the frog which is placed over the background - on the player layer. Every player layer's string is shifted right by 1 pixel, because we can't place matrices in one column because their pins will intersect.



X coordinates = display shift * matrix shift. Green boxes are examples of zones where a player can step and stay alive. Red boxes are examples of fields where a player will die.

Matrix driver - element which renders the picture by defining data. In the project there are 12 different drivers, but all of them differ from the other ones by the entity which is loaded in it and some more possibilities for water drivers. Because we have 102 matrices, we have 102 connectors, 8 in every row where 4 for the background layer and 4 for the player layer. The names of all drivers are busDriver, bolidDriver, carDriver, suvDriver, logDriver_1, logDriver_2,

logDriver_3, turtleDriver_2, logDriver_3, swampDriver, finishDriver. Below are tables with the driver pin specification and description of differences between all these drivers.

1. busDriver, bolidDriver, carDriver, suvDriver are the same, one difference is the model which is loaded with them. It is a plain driver which takes all shifts and makes a picture on the certain matrix.
2. logDriver_1, logDriver_2, logDriver_3 - render logs. logDriver_1 doesn't consider the display shift, because on the display are 4 logs, one a matrix. Also, it moves the player with the same speed, when the player is standing on them.
3. turtle_3x_1, turtle_2x_1 - the most complex drivers, because they have functions like logs and to this add animation. For three turtles, it is an animation of a paw moving and for two turtles it is an animation of sinking in the water. For these drivers we have a particular intersect computer which has a mod with an animation state counter. For sinking turtles, it returns a four bit string ([0;1] U [13;15] - turtles are in the shallow end, [2;5] U [9;12] they are on the sinking or pop up process, [6;8] - turtles are underwater). For moving three turtles, it returns one bit signal.
4. swampDriver - the complaints circuit, which draws a static background and has a classic frog mapping on the player layer.
5. finishDriver - shows squares where frogs came in.

Below you can see the specifications for the drivers:

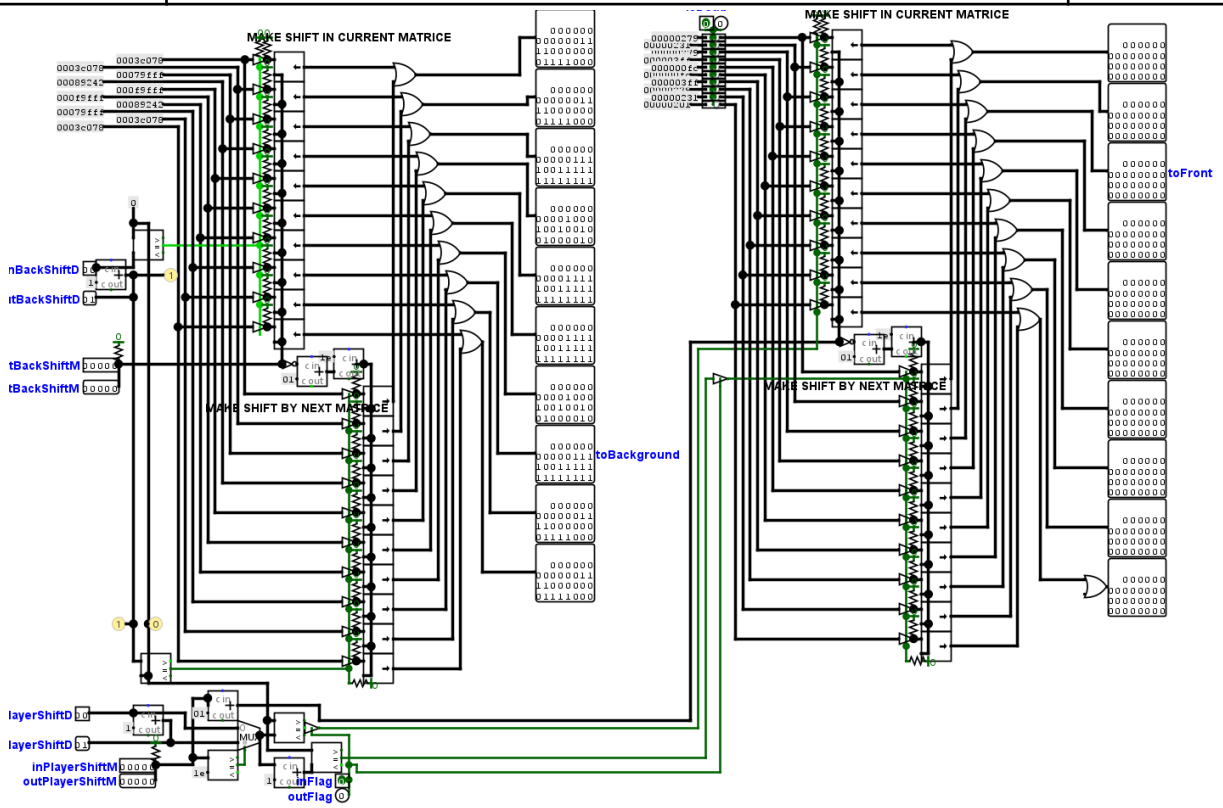
6 input pins

Name	Description	Values
inBackShiftD	*input data display shift* for the *background layer*	2 bits [0b00; 0b11]
inPlayerShiftD	*input data display shift* for *player layer*	2 bits [0b00; 0b11]
inputBackShiftM	*input data matrix shift* for *background layer*	5 bits [0b00000; 0b11101]
inPlayerShiftM	*input data matrix shift* for *player layer*	5 bits [0b00000; 0b11101]
inFlag	the driver will render picture if this value is 1, otherwise it will do nothing	1 bit [0b0; 0b1]
isDead	if this value is equal to 1, a frog will be red colored	1 bit [0b0; 0b1]

26 output pins

Name	Description	Values
toBackground (10 pieces)	every one of the ten pins is connected to a matrix string and determines which pixels are shined in the matrix on the *background layer*	30 bits [0x0; 0x3fffffff]
toFront (10 pieces)	every one of ten pins is connected to a string of matrix and determines which pixels are shined in a matrix on the player *layer*	30 bits [0x0; 0x3fffffff]
outputBackShiftM	original value taken from *inputBackShiftM, it gives a M shift for the next driver if that exists*	5 bits [0x0; 0x1e]
outPlayerShiftM	original value taken from *inPlayerShiftM, it gives an M shift for the next driver if that exists*	5 bits [0x0; 0x1e]
outBackShiftD	incremented value from *inBackShiftD, it gives a D shift for the next driver if that exists*	2 bits [0x0; 0x3]
outPlayerShiftD	original value taken from *inPlayerShiftD, it gives a D shift for the next driver if that exists*	2 bits [0x0; 0x3]
inFlag	takes value from input *inFlag* and gives it to the next driver in a string if that exists	1 bit [0b0; 0b1]

isDead	takes value from input *isDead* and gives it to the next driver in a string if that exists	1 bit [0b0; 0b1]



There are parts:

Shifter - make a matrix and display shifts for every string. It takes a tact and speed mode (0/1/2) (1 shift a tact/ 2 shifts a tact / 4 shift a tact). On the board there are right and left shifters.

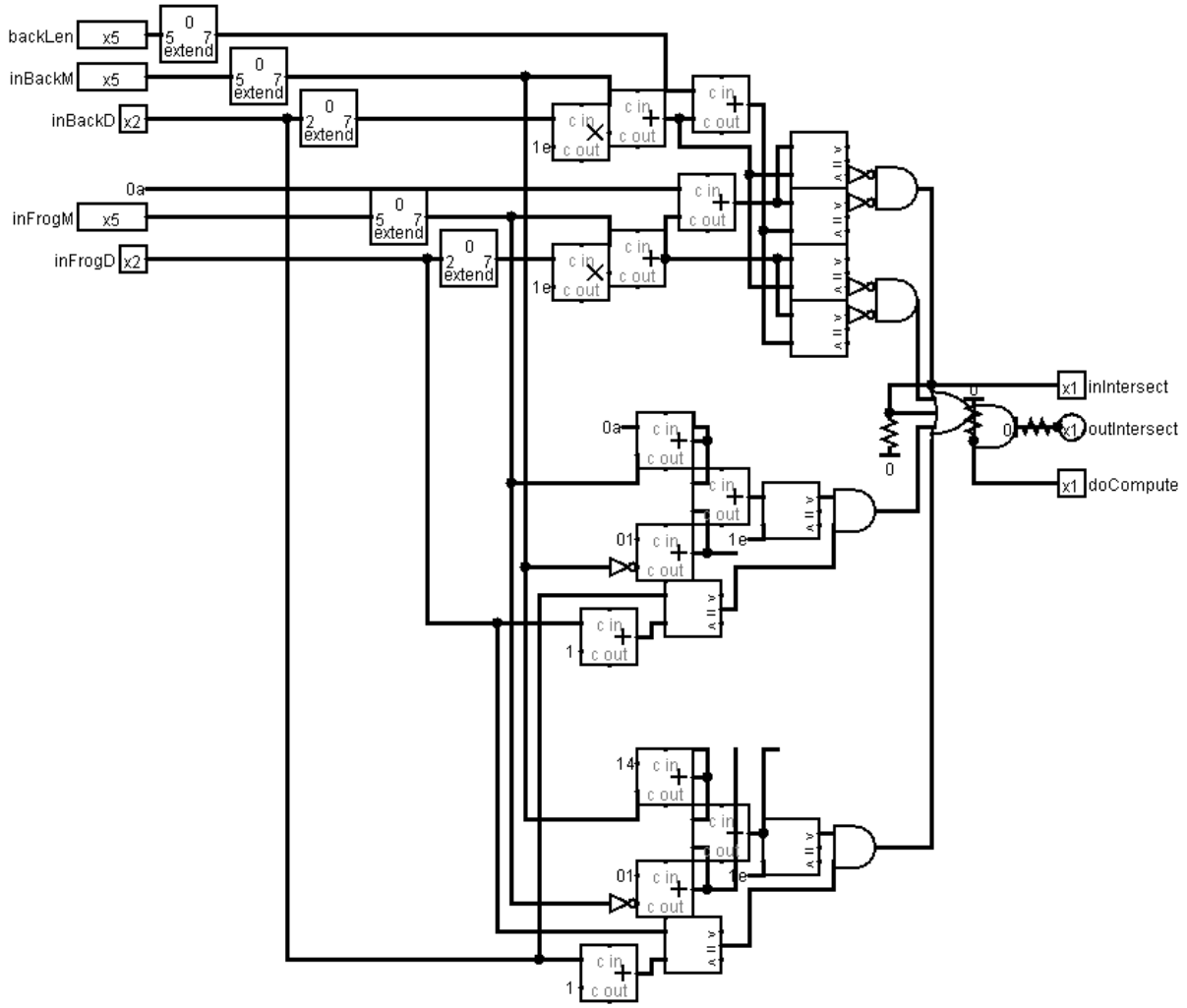
Intersection computer - it takes all shifts, takes an entity length and return status of intersection.

1. When a player and the entity are in the same matrix, it checks for one of two events - matrix shift of frog bigger than entity shift and less then entity shift + entity length. Or matrix shift of frog + frog's length bigger than entity shift and less then entity shift + entity length have happened.

2. backD + 1 is equal to playerD, so if the above condition is true for backM - 30 then intersection has happened.

3. backD - 1 is equal to playerD, so if the above condition is true for playerM - 30 then intersection has happened.

If an intersection has happened on a roadblock, it returns 1 and means death, else if it's happened at the water block it means that a player is placed on a moving platform and staying alive. There are three varieties of this circle: original for the road block, for water and for water with managing the turtle's animation.



Height flag - plain circle that compares Y coordinate of a player and Y position of display string.

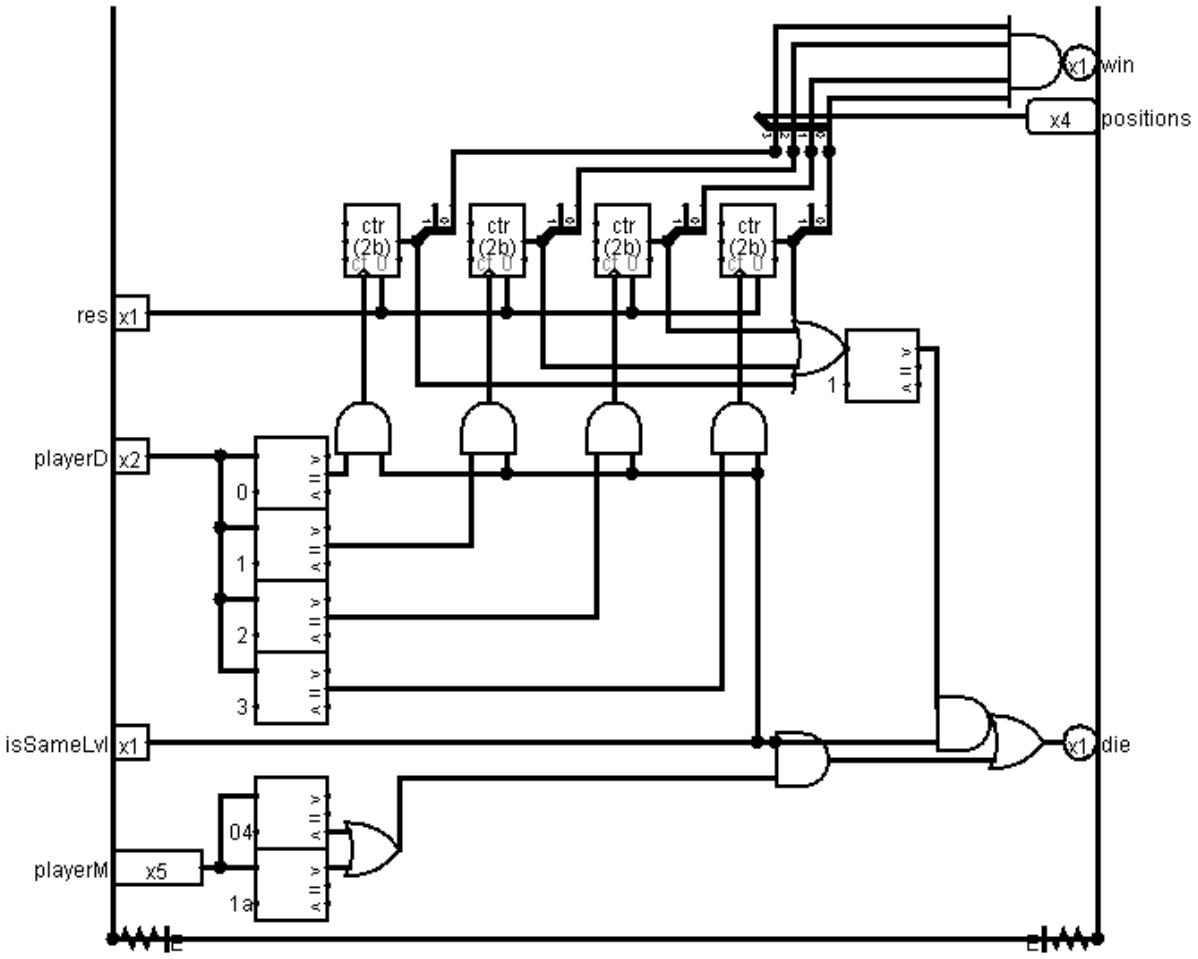
Finish computer - circle with 4 counters which manage the finish line. Start states of counters are zero, if a player steps into the finish matrix and the distance between it and matrix border more than five pixels counter of this matrix is incrementing else schema return signal that player's died. If the counter state was one and the player stepped onto the matrix with this counter again, then the frog died.

Input values

Name	Description	Value
res	takes the signal of the restart and erase all data on the scheme	1b[0x0;0x1]
playerM	the same quantity as in the driver	5b[0x0;0x1e]
playerD	the same quantity as in the driver	5b[0x0;0x1e]
isSameLvl	if 1 we can add count or kill a player (comes from heightFlag)	1b[0x0;0x1]

Output values

Name	Description	Value
win	1 if all squares are filled	1b[0x0;0x1]
positions	bit string where every digit determines the state of the square	4b[0x0;0xf]
die	becomes 1 if player has bumped into a border or tried to fill already filled cell	1b[0x0;0x1]
ok	1 if a frog stepped into correct position	1b[0x0;0x1]



Keyboard - a gaming controller. Two modules were created for it. The first module counts the frog's steps on the vertical and using a comparator we define which key was pressed. If a player presses the ‘w’ key, the module adds 1 to the 4 bit counter. And if he presses the ‘s’ key, the module subtracts 1 from the counter. And the second works the same as the first one, but there are two counters. One of them is a 5 bit counter and another is a 2 bit counter. When a player presses the ‘a’, we add 10 to the 5 bit counter (determine which part of the matrix a frog is in). And when it is 30, we reset it to zero and add 1 to the 2 bit counter (switch to another matrix). If a player presses the ‘d’ key, the module subtracts 10 from the 5 bit counter. And if it is zero, the module subtracts 1 from the 2 bit counter and the 5 bit counter becomes 20. At the same time, it is impossible to go beyond the minimum and maximum values in the both counters. The keyboard also implements frog shifting on turtles and logs.

Software part

There are 2 external devices:

- **scores_counter** displays the current score and the record score and stores them at the addresses **0xF6** and **0xF7**, respectively.
- **lives_counter** displays the number of remaining lives and the number of saved frogs and stores them at the addresses **0xF8** and **0xF9**, respectively.

```
1          # External devices:
2       asect 0xF6
3   scores_counter:
4       current_score : ds 1
5       record_score : ds 1
6
7       asect 0xF8
8   lives_counter:
9       number_of_remaining_lives : ds 1
10      number_of_saved_frogs : ds 1
```

It was necessary to implement functions for three events:

- A frog died, or in other words, a **bump** happened.
- A frog was **rescue**d.
- A frog made a **good jump**.

Interrupts were used for this:

```
12         # Interrupts:
13      asect 0xF0
14  bump : dc bump_ISR                  # the current score - 10,
15  bump_flags_combo : dc 0x00          # the number of remaining lives - 1;
16
17  rescue : dc rescue_ISR              # the current score + 16,
18  rescue_flags_combo : dc 0x00        # the number of saved frogs + 1;
19
20  good_jump : dc good_jump_ISR        # the current score + 1.
21  good_jump_flags_combo : dc 0x00
```

Each of these cases has its own interrupt vector, which is two memory cells that store a pointer to its ISR and the contents of the CPU status that must be set in the PS register before control is passed to the ISR. But ISRs do not need to use this register because they have no conditional constructs, so these memory cells store combinations of flags instead of just one flag signaling that an interrupt has occurred. This reduces the number of missed interrupts to almost zero.

The ISRs are located at the following addresses:

- **0xF0** - stores the ISR for the frog **bump** interrupt.
- **0xF2** - stores the ISR for the frog **rescue** interrupt.
- **0xF4** - stores the ISR for the **good** frog **jump** interrupt.

The start of the code is stored at **0x00**. **Start_game** is the first routine, which initializes the number of lives and starts the main program.

```
23          # Routines:
24      asect 0x00
25  start_game:
26      ei                          # enabling the interrupts
27      ldi r0, number_of_remaining_lives
28      ldi r1, 3                   # there are 3 lives at the beginning of the game
29      st r0, r1                   # initializing the number of remaining lives
30      br check_for_interrupts
```

All other routines can be divided into 3 semantic groups:

- Checks.
- Updates.
- Interrupt Service Routines.

Checks:

```
90  check_number_of_hearts:
91      ldi r0, number_of_remaining_lives
92      ld r0, r1
93      if
94          tst r1
95      is eq                          # the game is lost
96          di                         # disabling interrupts
97          halt
98      else
99          br check_for_interrupts # continuation
100     fi
```

```
102 check_number_of_survivors:
103     ldi r0, number_of_saved_frogs
104     ld r0, r1
105     ldi r0, 4
106     if
107         cmp r1, r0
108     is eq                           # the game is won
109         di                          # disabling interrupts
110         halt
111     else
112         br check_for_interrupts # continuation
113     fi
```

These routines are quite similar. The number of remaining lives or the number of saved frogs is loaded into the register r1. Then they check if the game is over. It ends in two cases:

- ❖ There are no lives left. This is a loss.
- ❖ All 4 frogs have been saved. This is a win.

Check_for_interrupts is the main routine.

```
32      # Checks:
33  check_for_interrupts:
34      if
35          ldi r0, bump_flags_combo
36          ld r0, r1
37          tst r1
38      is z
39          if                      # the bump_interrupt has not occurred yet
40              ldi r0, rescue_flags_combo
41              ld r0, r1
42              tst r1
43          is z
44              if                  # and the rescue_interrupt has not occurred yet
45                  ldi r0, good_jump_flags_combo
46                  ld r0, r1
47                  tst r1
48              is z
49                  wait            # and the good_jump_interrupt has not occurred yet
50                  br check_for_interrupts
51              else                # the good_jump_interrupt has already occurred
52                  jsr update_current_score
53                  jsr update_record_score
54                  clr r1          # clearing the good_jump_flags_combo
55                  ldi r0, good_jump_flags_combo
56                  st r0, r1
57                  br check_for_interrupts
58              fi
59          else                    # the rescue_interrupt has already occurred
60              ld r0, r0
61              while
62                  dec r0
63              stays ge
64                  ldi r1, 16          # adding 16 points
65                  jsr update_current_score
66              wend
67              jsr update_record_score
68              jsr update_frog_survivors
69              clr r1              # clearing the rescue_flags_combo
70              ldi r0, rescue_flags_combo
71              st r0, r1
72              br check_number_of_survivors
73          fi
74      else                        # the bump_interrupt has already occurred
75          ld r0, r0
76          while
77              dec r0
78          stays ge
79              ldi r1, -10             # removing 10 points
80              jsr update_current_score
81          wend
82          jsr update_record_score
83          jsr update_frog_lives
84          clr r1                  # clearing the bump_flags_combo
85          ldi r0, bump_flags_combo
86          st r0, r1
87          br check_number_of_hearts
88      fi
```

There are 3 checks to determine if an appropriate interrupt has occurred. If an interrupt occurs, processing begins. This includes updating the current

score and the record score, and there may also be a check to see if the game is over. If there are no interrupts, the processor is in wait mode. This improves the efficiency of the game.

You can see the cycles of updating the current score. Since it is the combinations of flags that are stored at the addresses, the program handles as many interrupts as possible. And each of these interrupts adds or subtracts points from the current score. This is done outside the IRS to speed it up. You can also notice that only registers r0 and r1 are used. This avoids pushing all the registers on the stack at the beginning and taking them back at the end, which would take another 2 bytes of instructions for each ISR. This results in the fastest Interrupt Service Routines.

Updates:

```
115     # Updates:
116 update_current_score:         # r1 must contain the number of points to be added
117     ldi r0, current_score
118     ld r0, r0
119     if
120         add r0, r1
121     is lt                     # if the current score becomes negative,
122         clr r1                # then it becomes zero
123         ldi r0, current_score
124         st r0, r1
125     else
126         ldi r0, current_score
127         st r0, r1
128     fi
129     rts
```

```
131 update_record_score:          # r1 must contain the current score
132     ldi r0, record_score
133     ld r0, r0
134     if
135         cmp r0, r1
136     is lt                     # if the record score is less than the current one,
137         ldi r0, record_score  # then the record score is updated
138         st r0, r1
139     fi
140     rts
```

On processing of each of the interrupts, the current score is updated, followed by the record score. To make these routines run faster, each of them assumes that register r1 stores the required data. This is observed in the code because before each call to **update_current_score**, the number of points to be added is written to register r1 (if they need to be removed, it is a negative number), followed by a call to **update_record_score**, and the current score is stored in register r1 after the **update_current_score** routine.

When the current score is updated, a check is made to see if the score has become negative. Since the score cannot be negative, it is canceled in this case. Otherwise, the new score value is stored.

When the record score is updated, a simple comparison is made to the current score. If the current score is greater than the record score, the record score is updated to equal the current score.

```
142 update_frog_lives:
143     ldi r1, bump_flags_combo
144     ld r1, r1
145     ldi r0, number_of_remaining_lives
146     ld r0, r0
147     if
148         sub r0, r1
149     is lt                    # if the number of remaining lives becomes negative,
150         clr r1               # then it becomes zero
151         ldi r0, number_of_remaining_lives
152         st r0, r1
153     else
154         ldi r0, number_of_remaining_lives
155         st r0, r1
156     fi
157     rts
```

```
159 update_frog_survivors:
160     ldi r1, rescue_flags_combo
161     ld r1, r1
162     ldi r0, number_of_saved_frogs
163     ld r0, r0
164     add r0, r1                # increasing the number of saved frogs
165     ldi r0, number_of_saved_frogs
166     st r0, r1
167     rts
```

There are a couple of other updates, namely updating the number of remaining lives and the number of surviving frogs, which involve working with combinations of flags.

The **update_frog_lives** routine reduces the number of remaining lives by the number of bump interrupts that have occurred, equal to the number of flags in the combination. But the number of remaining lives can become negative, and this should not be the result. So there is a check to see if the number has become negative. If it has, it must return to zero.

The procedure **update_frog_survivors** increases the number of saved frogs by the number of rescue interrupts that occurred, equal to the number of flags in the combination.

ISRs:

```
169     # Interrupt Service Routines:
170 bump_ISR:
171     ldi r2, bump_flags_combo
172     ld r2, r3
173     inc r3                  # increasing the combination of bump_flags
174     st r2, r3
175     rti
176
177 rescue_ISR:
178     ldi r2, rescue_flags_combo
179     ld r2, r3
180     inc r3                  # increasing the combination of rescue_flags
181     st r2, r3
182     rti
183
184 good_jump_ISR:
185     ldi r2, good_jump_flags_combo
186     ld r2, r3
187     inc r3                  # increasing the combination of good_jump_flags
188     st r2, r3
189     rti
190
    end
```

Each of them takes only 6 bytes of instructions for maximum execution speed. They only involve loading a value, incrementing it and storing it back. Other functions required for a specific event are executed outside of the Service Routine.

The **bump_ISR** is executed in three cases:

- A frog crashes into a vehicle on the road, which is the first challenge for it.
- A frog sinks into a swamp, which is the second challenge.
- A frog drying out in the sun.

The **rescue_ISR** is called when another frog has been rescued, i.e. brought home.

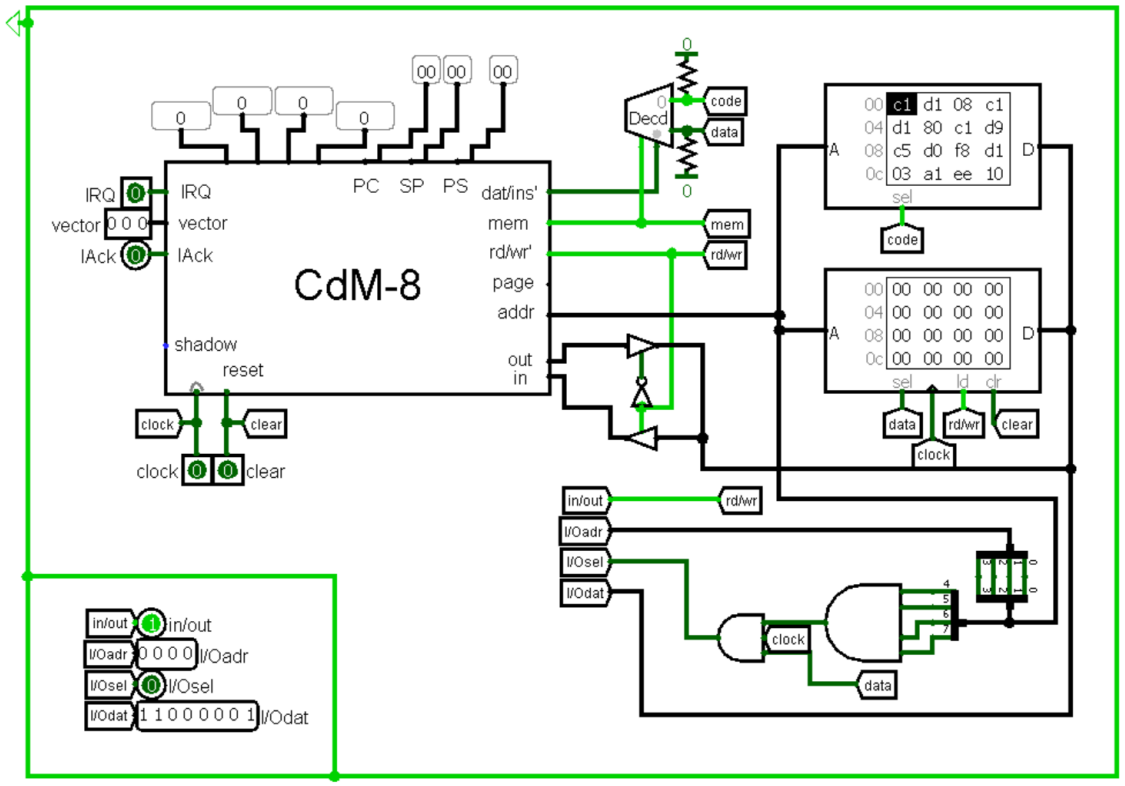
The **good_jump_ISR** is activated when a frog jumps over a new line for it.

Software and Hardware interaction

The game uses the algorithm in the **statistics.asm** file and the **statistics.circ** library to implement the statistics. The following has been added to the display:

- ❖ Time left for a frog not to dry up.
- ❖ The number of lives left.
- ❖ The number of members of the frog's family that has reached home.
- ❖ The current and record scores.

The interaction between software and hardware is made possible by the **CdM-8** processor, based on the Harvard architecture (ROM and RAM are used separately):



It has 4 input pins and 5 output pins:

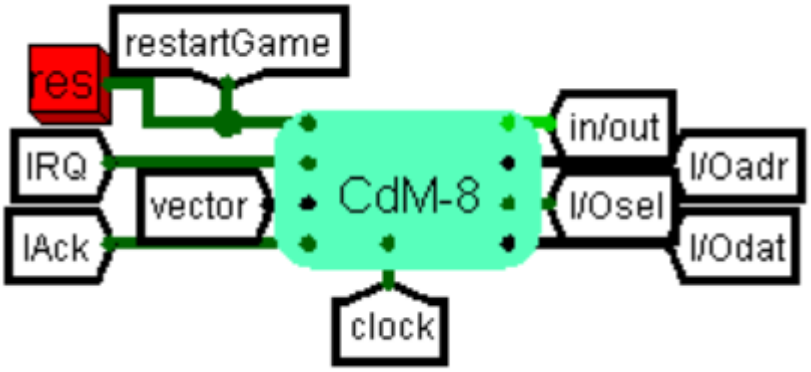
The input pins:

Name	Description	Value
clock	Cadence	1b[0x0;0x1]
clear	If 1, it resets the CdM8 processor and clears the ROM	1b[0x0;0x1]
IRQ	If 1, it tries to interrupt the main program	1b[0x0;0x1]
vector	Interrupt vector, it determines which interrupt should occur	3b[0x0;0x7]

The output pins:

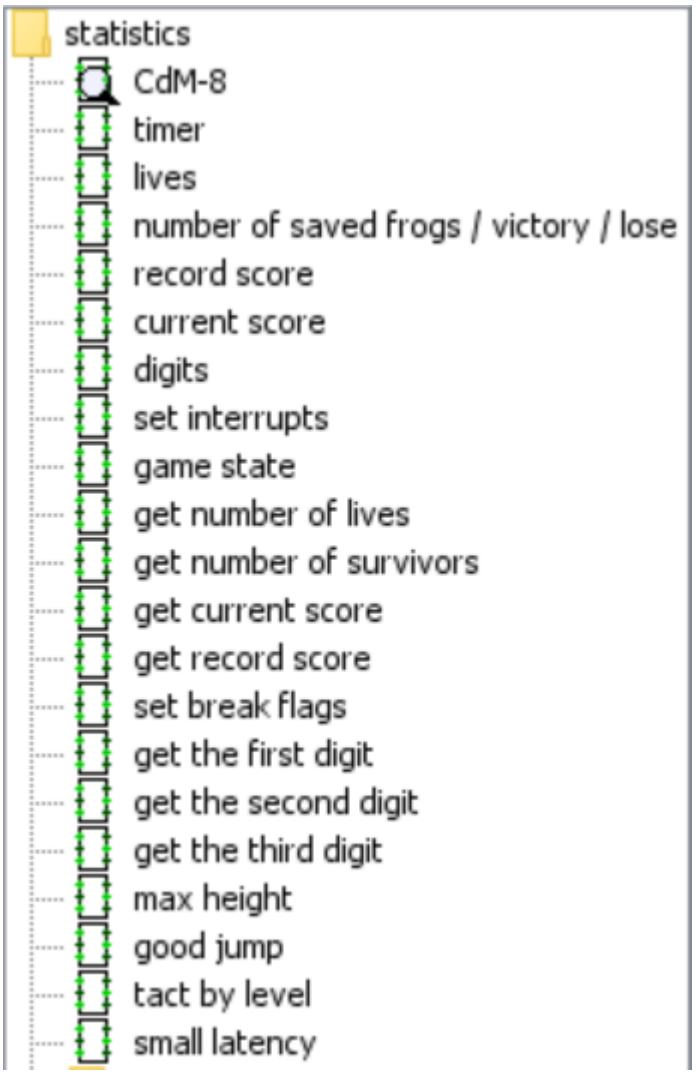
Name	Description	Value
IAck	Interrupt acknowledgement	1b[0x0;0x1]
in/out	If 1, the data are read in the program; if 0, the data are written in Logisim	1b[0x0;0x1]
I/Oadr	Current address	4b[0x0;0xF]
I/Osel	1 if an external device is used	1b[0x0;0x1]
I/Odat	Actual data	8b[0x0;0xFF]

On top of the processor there are detectors that show the contents of registers r0...r3 as well as PC, SP and PS. Aesthetically, the CdM-8 looks like this on the main schematic in Logisim:



The red res button is used to start a new game.

This is the contents of the **statistics** library:



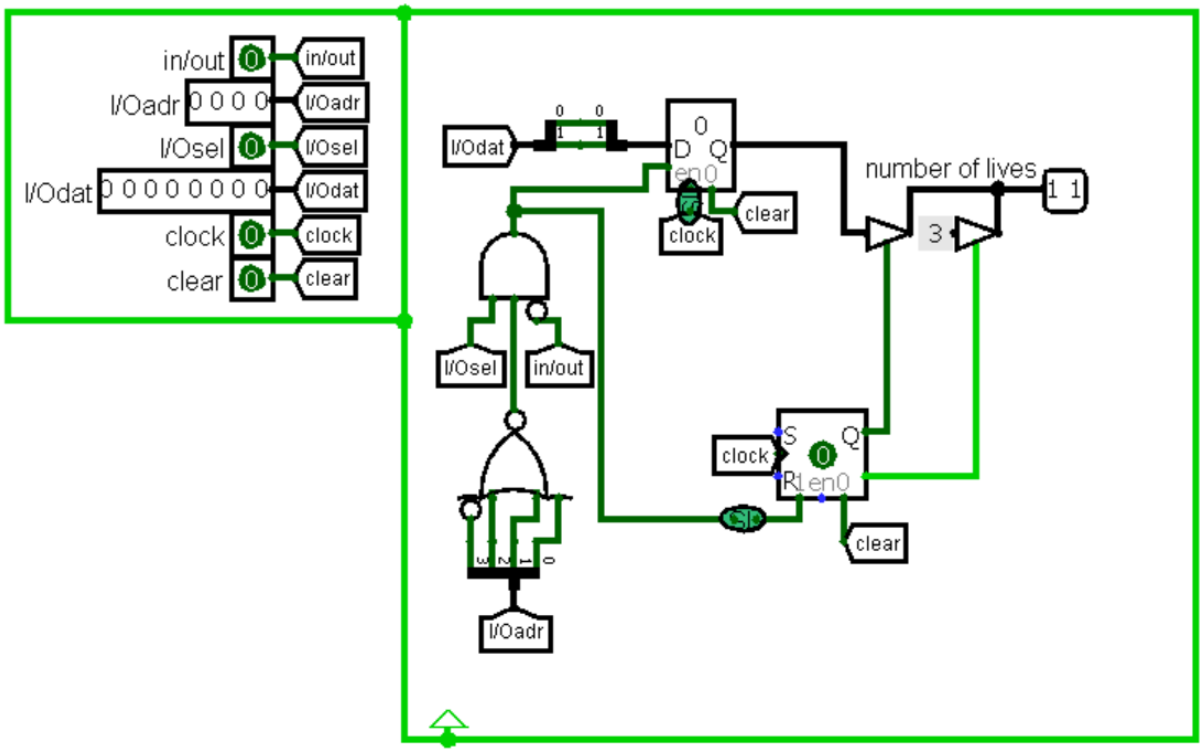
It contains the following:

- The circuit with the **CdM-8** processor, which handles the interrupts.
- The schemes for **timer**, **lives**, the **number of saved frogs / victory / loss**, **record score**, **current score** and **digits**, with which all important information is displayed on the screen.
- The **set interrupt** scheme that informs the processor of events that have occurred and need to be processed.
- The **game state** scheme, which is used to set the state of the game, namely victory, loss, and continuation of the game.

- The schemes **get number of lives**, **get number of survivors**, **get current score** and **get record score**, which take data from the ROM, which are loaded by the program.
- The auxiliary circuit **set break flags**, which sets a flag for each small heart that it has been broken.
- The **get the first digit**, **get the second digit**, and **get the third digit** auxiliary circuits, which are needed to display each digit of a score separately.
- The auxiliary circuits **max height** and **good jump** to set the good jump interrupt.
- The **tact by level** scheme, which is needed to implement the difficulty levels of the game.
- The **small latency** scheme, which is very useful for working with registers. You need to delay the clock signal before writing data to the register, so that the data has time to reach its destination.

The following circuits are used to read the data:

❖ Reading the number of remaining lives:



This circuit has 6 input pins on the left side and one output contact. Many of the circuits in this library have exactly this set of input pins. These are the data provided by the CdM-8 processor, as well as the **clock** and **clear** signals. Their description can be found in the circuit description of the CdM-8.

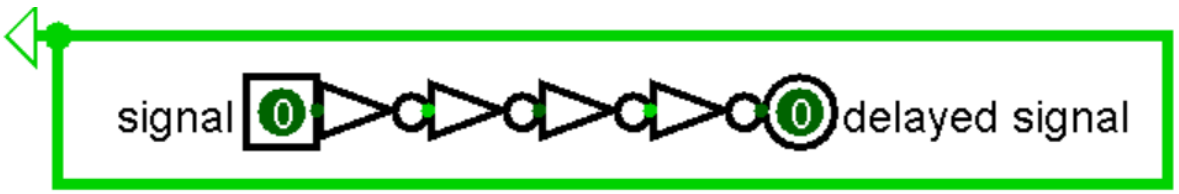
The data is read into the register, but first three signals are checked:

1. The **I/Osel** signal must be high. This means that an external device is being handled and the first 4 bits of the address are 0xF.
2. The last 4 bits of the address must be 0b1000. This is the address where the external device **lives_counter** stores the **number_of_remaining_lives**.
3. The **In/out** signal must be 0 because the data is being written to Logisim.

As the number of lives does not exceed 3, only the first 2 bits are taken from the **I/Odat** tunnel.

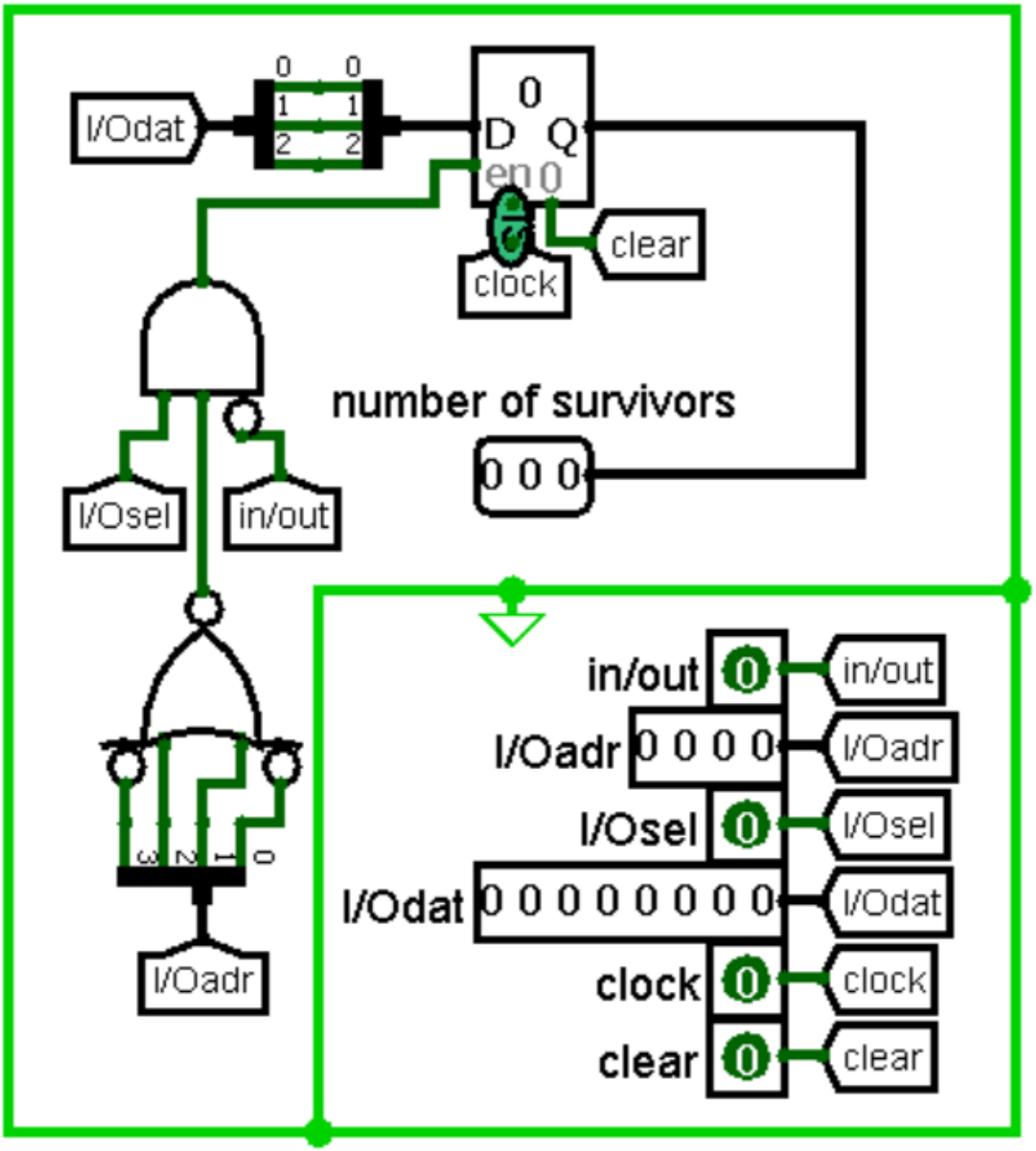
The initialisation value, i.e. 3, should be output as long as no trigger events have occurred. The S-R trigger is used for this purpose. The first time the 3-signal check is performed, the signals are switched to output the value of the register.

To ensure that all the signals have time to get to the right place, there is a **s**mall **l**atency in some places, these are the green ovals in the circuit. This is what it looks like inside:



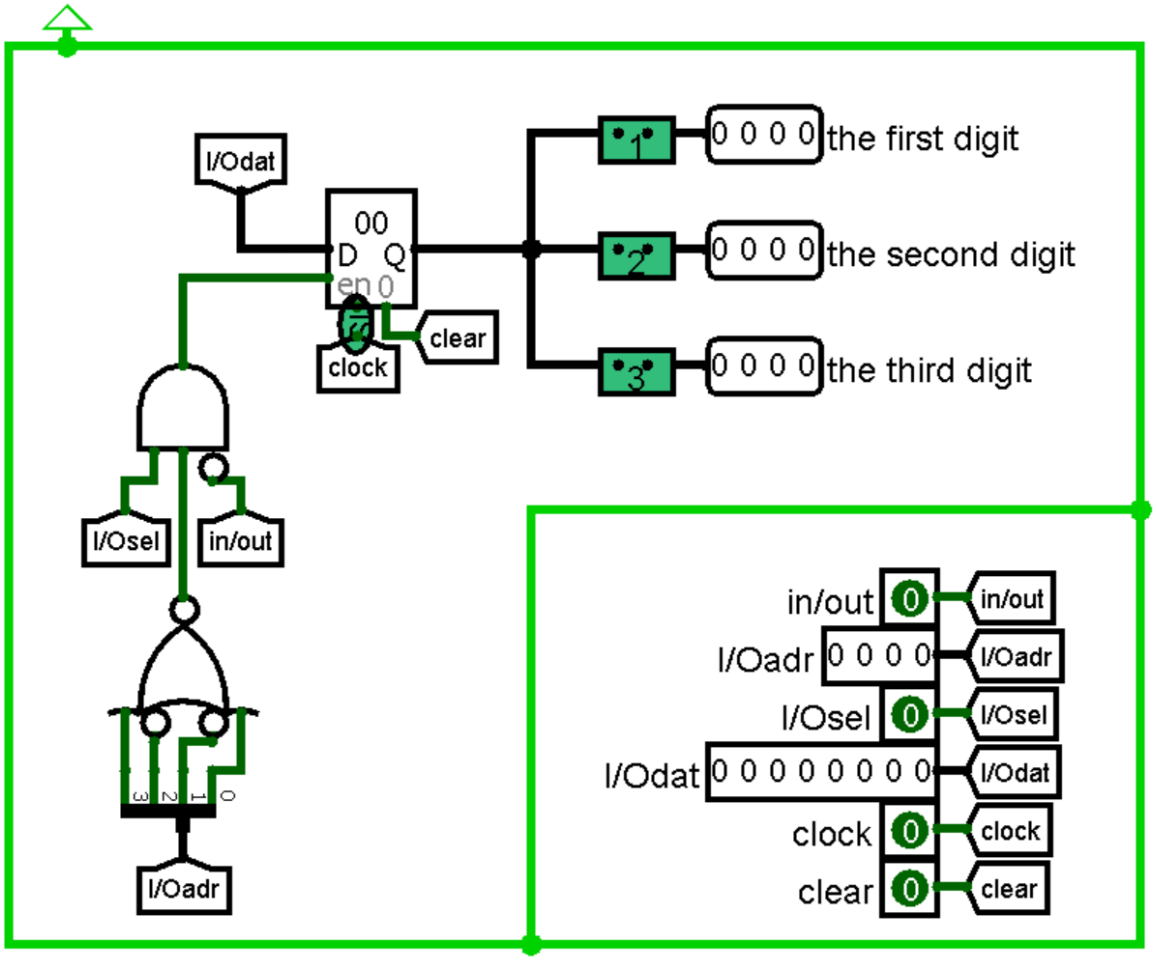
A signal is received, it goes through 4 inverters and it doesn't change its value, but it is delayed a bit.

❖ Reading the number of saved frogs:



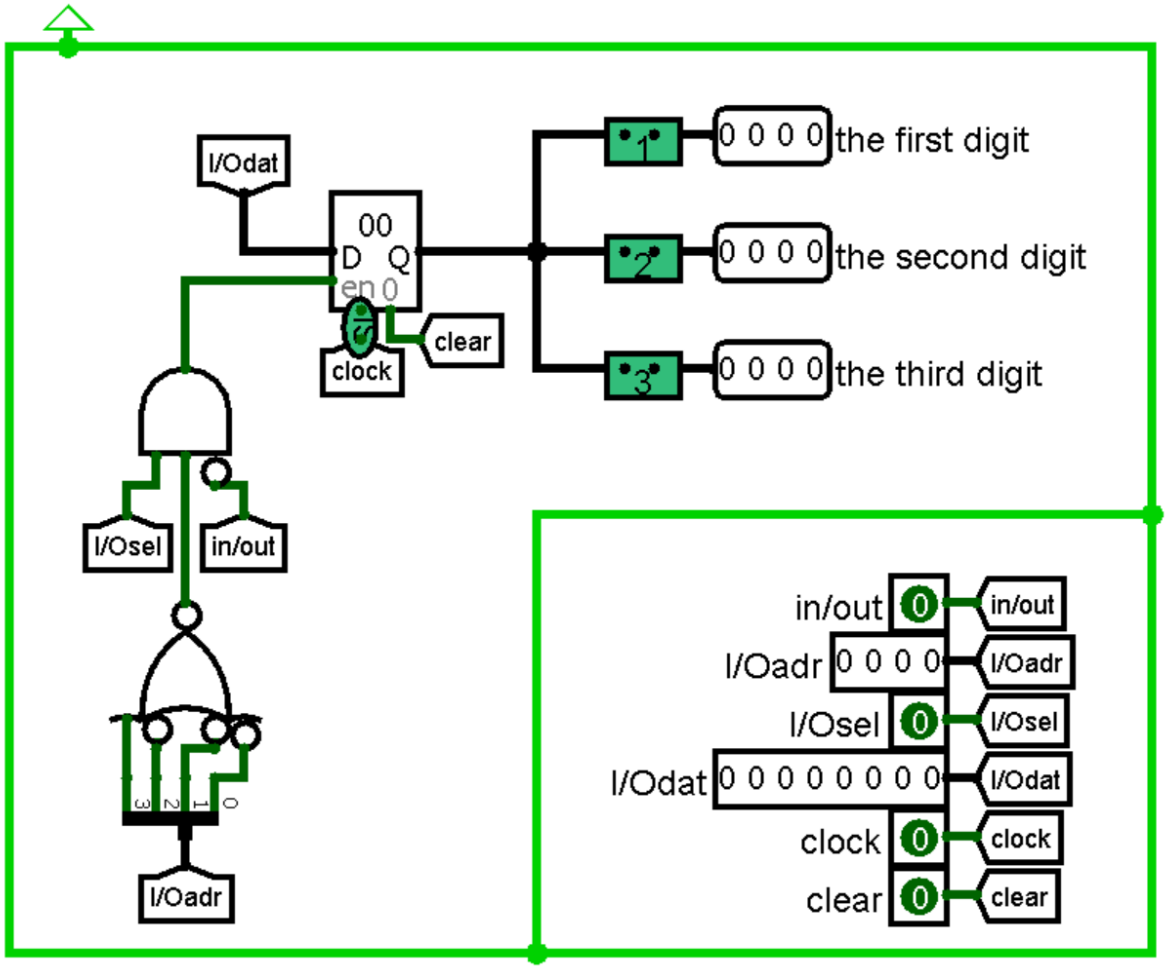
This uses the same check as the previous scheme. The only difference is the address to check. The external device **lives_counter** stores the **number_of_saved_frogs** at address 0xF9. So the last 4 bits of the address are checked to be 0b1001. And this time 3 bits are taken from the **I/Odat**, because the maximum number of frogs is 4.

❖ Reading the current score:



This scheme is not very different from the previous two. The difference is the address to check and the number of output pins. The external device **scores_counter** stores the **current_score** at address 0xF6, so the last 4 bits must be 0b0110. Now all 8 bits of **I/Odat** are taken, as the score can be quite high. And with additional sub-circuits, described later, it is divided into three decimal digits to make it easier to display them separately.

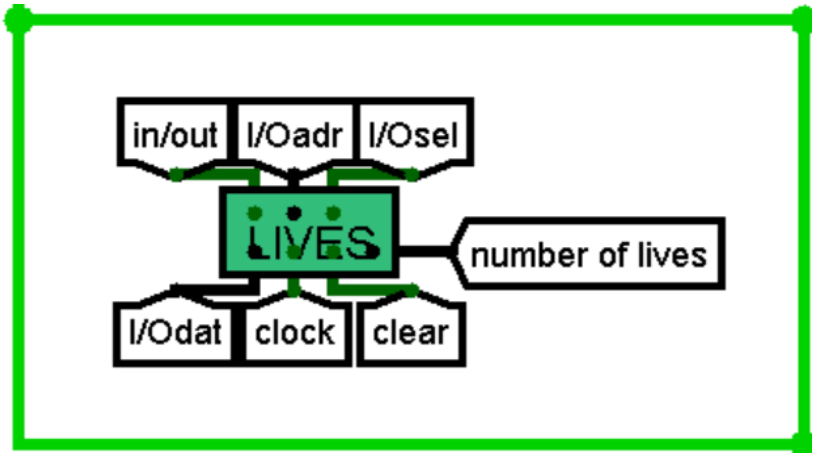
❖ Reading the record score:



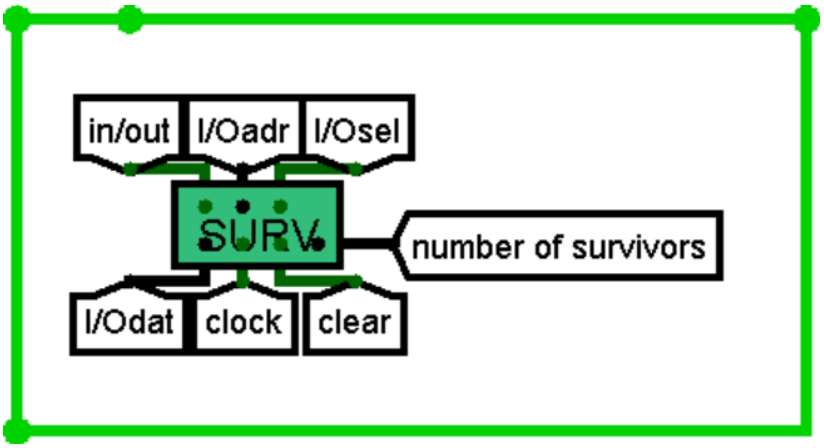
The circuit is very similar to the previous one, with one small difference. The external **scores_counter** stores **record_score** at address 0xF7, so the last 4 bits of the address must be 0b111.

The circuits described above for reading data are as follows:

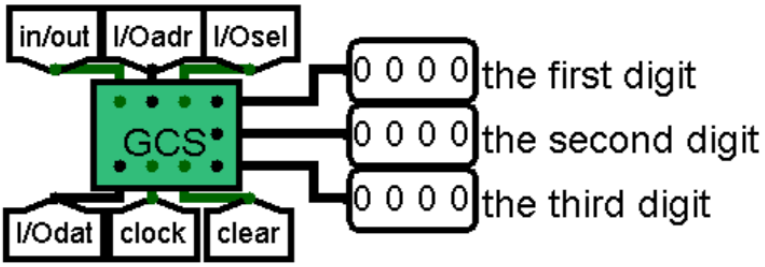
❖ Reading the number of remaining **lives**:



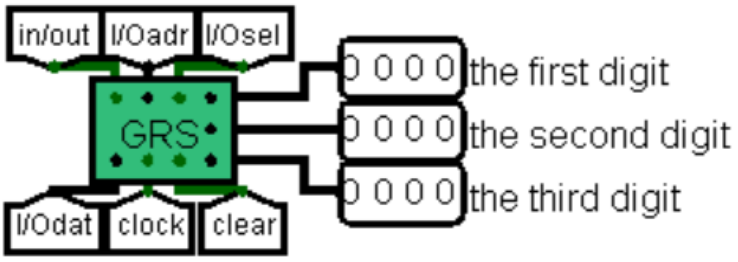
❖ Reading the number of saved frogs, i.e. **surv**ivors:



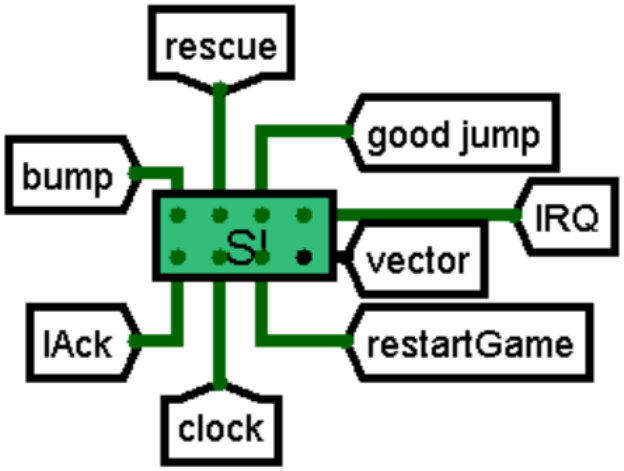
❖ Reading the current score, i.e. **g**etting a **c**urrent **s**core:



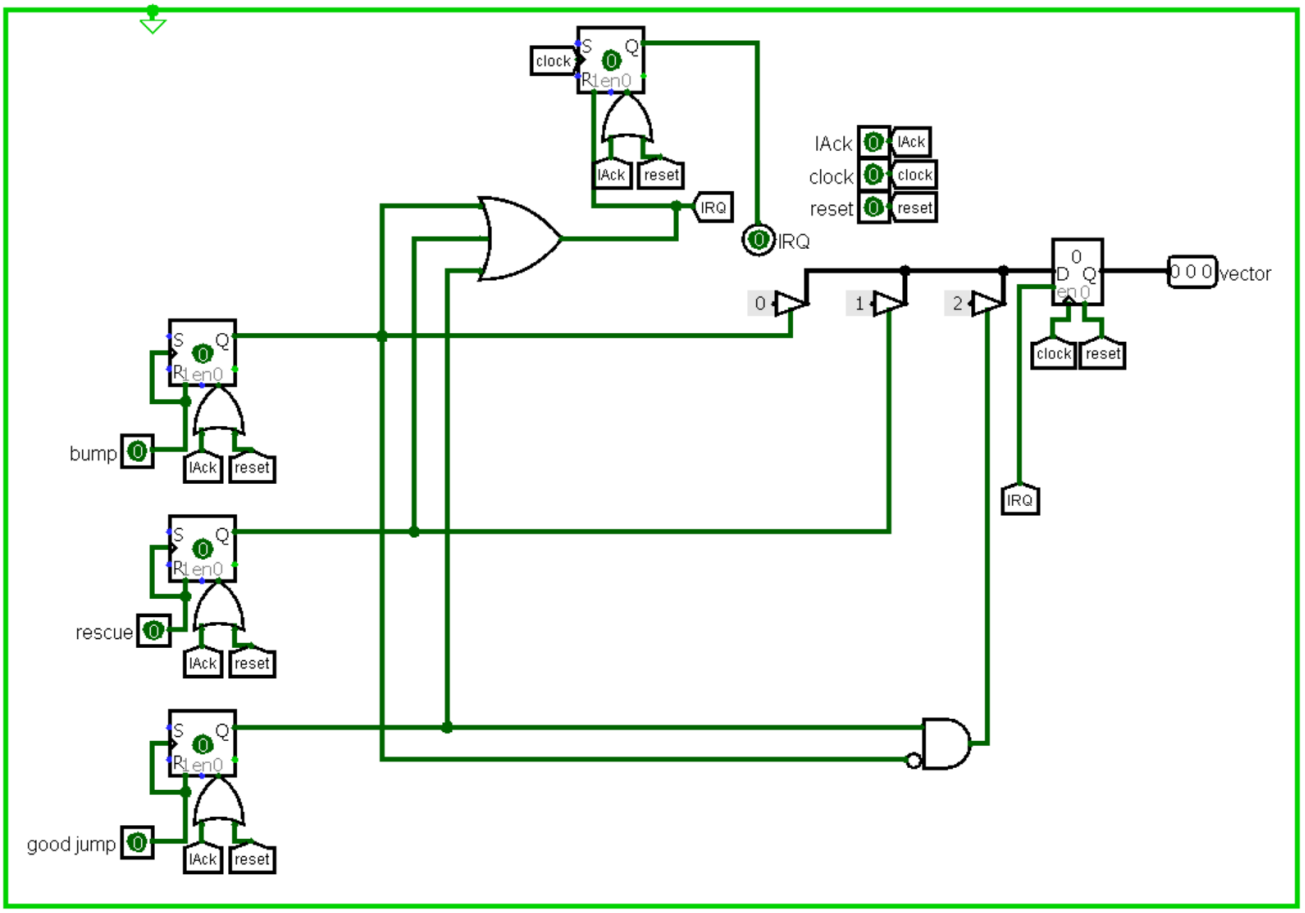
❖ Reading the record score, i.e. **g**etting a **r**ecord **s**core:



The following scheme is used to signal that a program needs to be interrupted (to **s**et **i**nterrupts):



The internal structure is as follows:



There are 6 input pins and 2 output pins.

The 6 input pins:

Name	Description	Value
bump	If 1, a frog is dead	1b[0x0;0x1]
rescue	If 1, a frog is rescued	1b[0x0;0x1]
good jump	If 1, a frog has made a good jump	1b[0x0;0x1]
IAck	If 1, an interrupt has been acknowledged by the processor	1b[0x0;0x1]

	and S-R triggers are cleared. No need to clear the register	
clock	Cadence	1b[0x0;0x1]
reset	If 1, the register and S-R triggers are cleared	1b[0x0;0x1]

The 2 output pins:

Name	Description	Value
IRQ	When an interrupt occurs, it rises to signal this to the processor	1b[0x0;0x1]
vector	Interrupt vector, it determines which interrupt should occur	3b[0x0;0x7]

The **IRQ** signal is lifted if at least one of the interrupt signals is active. It is also held by the S-R trigger until it receives an **IAck** signal from the processor. In this case it is allowed to write to the registry.

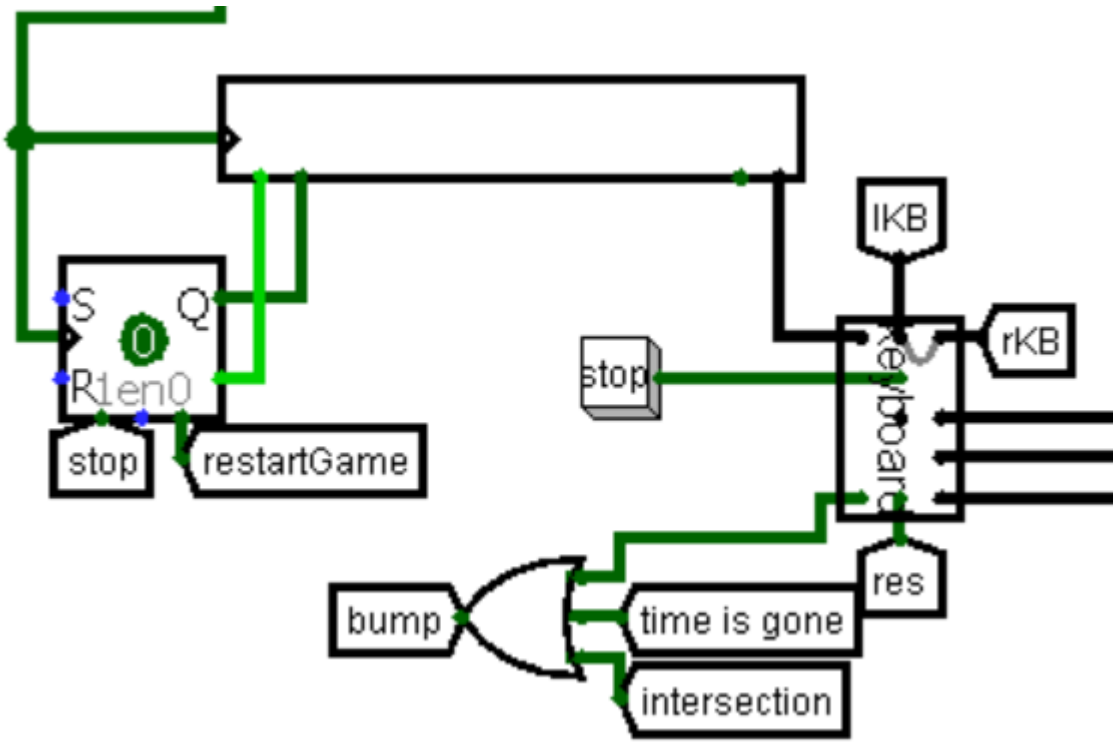
Controlled buffers determine which interrupt vector the processor must use. Several events can occur in a short period of time, but the **bump**, **rescue**, and **good jump** signals can never be raised at the same time. Moreover, it is impossible for both **bump** and **rescue** signals to be given (since the **rescue** signal is only given when a frog has reached the finish line and is no longer in danger), as well as **rescue** and **good jump** signals (**good jump** is not counted when a frog is on the penultimate line). However, it is possible to give the **bump** and **good jump** signals at the same time if the frog crosses a new line but dies immediately because of one of the 3 defined cases. Therefore, in the lower right part of the circuit for the **good jump** signal, there is an extra check that the

signal is not raised. If it is high, the **good jump** signal is ignored and then canceled by the **IAck** signal from the **bump** interrupt acknowledgement.

If the **good jump** signal is counted before the **bump** signal rises, this means that the **IAck** signal has already come from the processor. This means that the **good jump** interrupt has started to be processed and the **good jump** signal has already been dropped. But the **bump** signal has just been raised. In this case, S-R triggers are used to hold the fleeting event signals for further processing.

The following is used to set the signal that an interrupt has occurred:

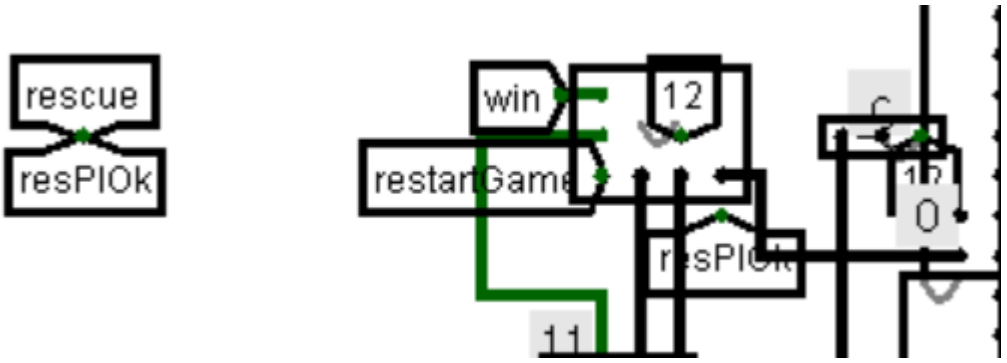
- A frog is dead.



If it is detected that a frog has **bump**ed into a car or has sunk (this is a signal from the **intersection** tunnel), or time has run out, or a frog is outside the bounds of the playing area (this is a signal from the keyboard sub-circuit), the **bump** signal is set.

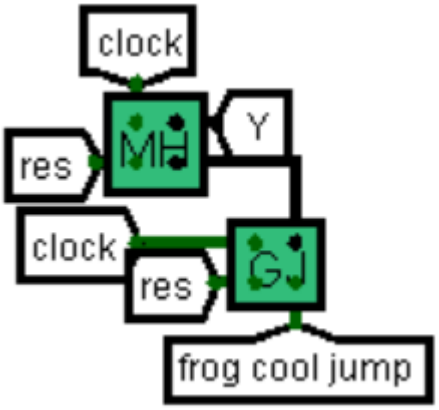
When the game ends, the keyboard is cleared and stops responding to keys. This is done with an S-R trigger. When the game starts again, characters can be written to the keyboard buffer and frog control becomes possible.

❖ A frog is saved.



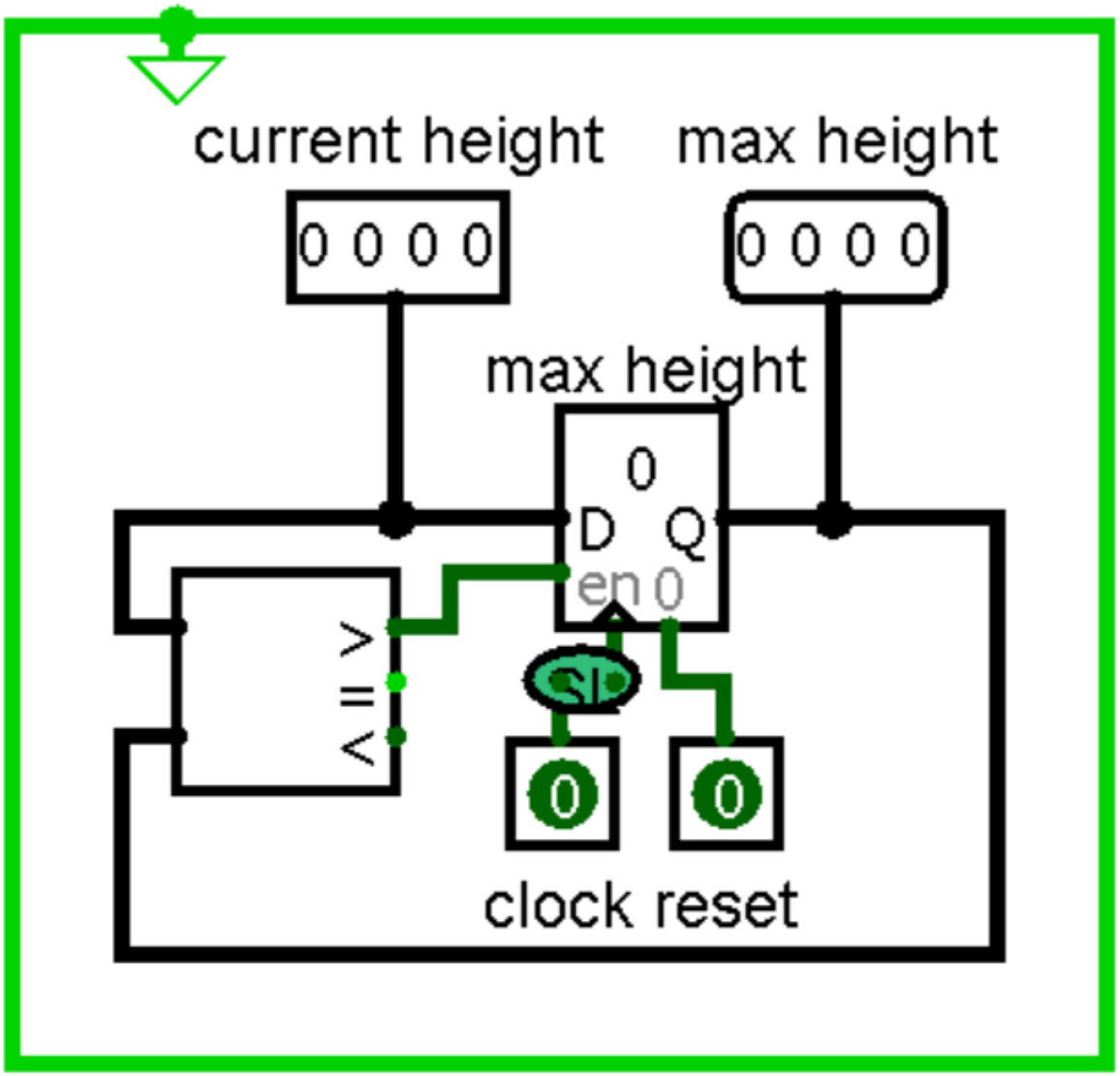
If a frog enters its home, it will survive. This is indicated by the **resPIOk** signal from the **finishComputer** circuit. Then the **rescue** signal rises.

❖ A frog jumps well.



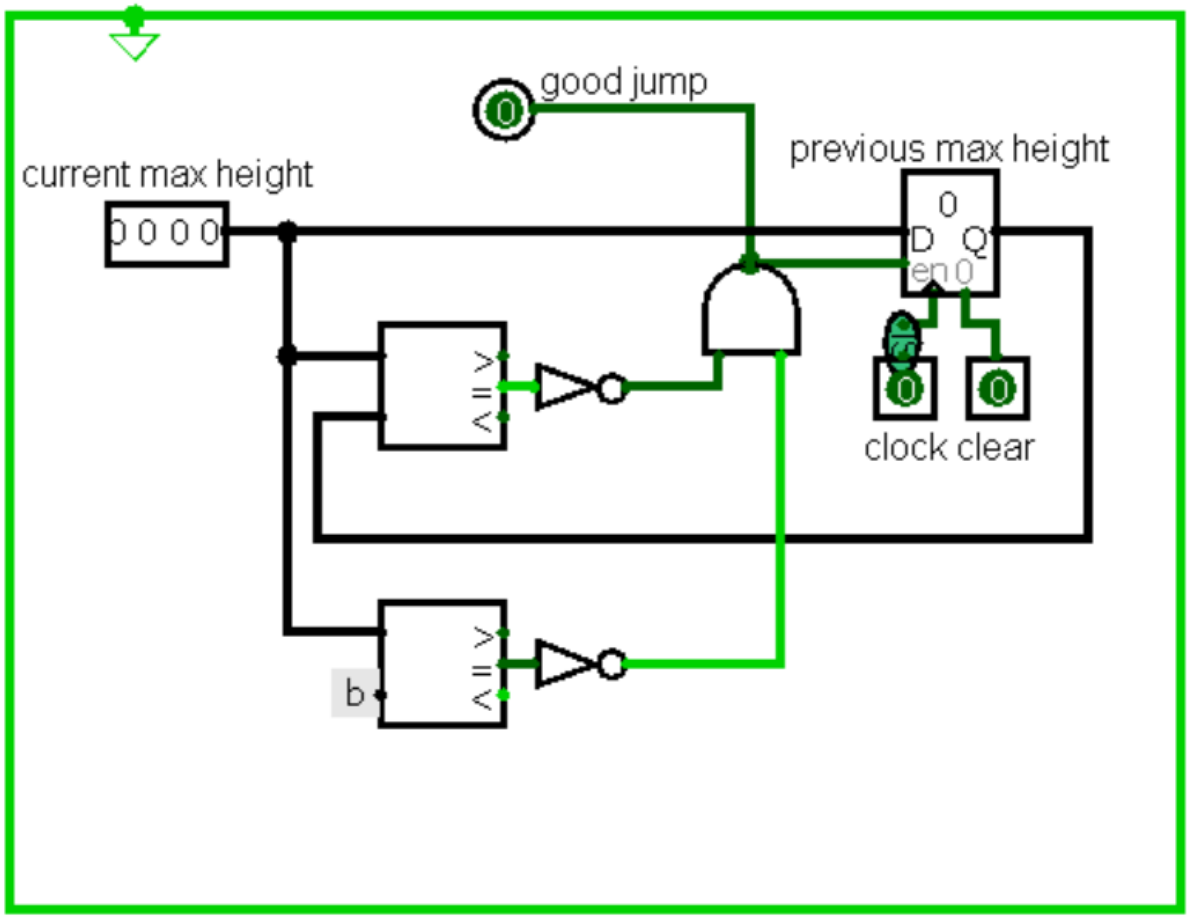
This event is calculated using two sub-circuits, the **max height** sub-circuit and the **good jump** sub-circuit. The Y signal comes from the keyboard sub-circuit.

The first sub-circuit from the inside looks like this:



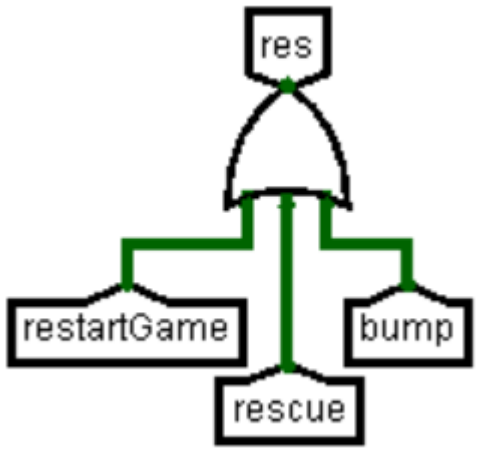
It has 3 input pins, namely **clock**, **reset** and **current height**. A register is used to store the maximum height and a comparator to compare the current and maximum heights. When the **current height** becomes greater than the maximum height, it is allowed to write to the register to update the maximum height. The **reset** signal clears the contents of the register. This circuit has only one output pin, which is the value of the maximum height.

The second sub-circuit from the inside looks like this:

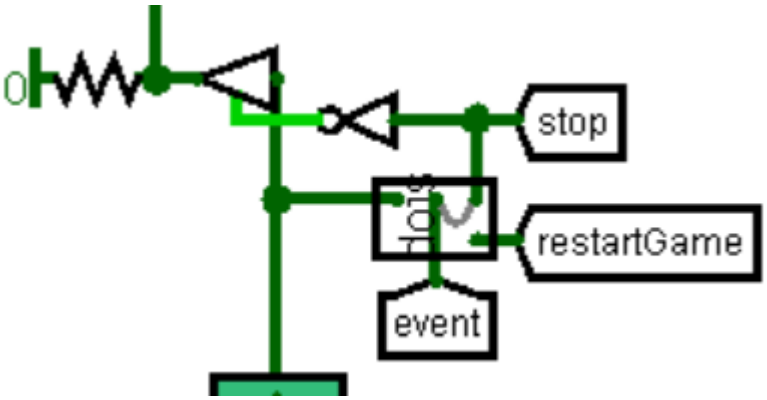


There are also 3 input contacts, they are **clock**, **clear** and **current max height**. **Clock** and **clear** have the same function as the **clock** and **reset** in the previous circuit. A register is used to store the previous maximum height, and two comparators are used to compare the current maximum height with the previous maximum height and to ensure that the current maximum height is not equal to the penultimate line, as a **good jump** is not counted separately on it. If they are not the same, it means that a frog has crossed a new line, which is not the penultimate one, and has made a **good jump**. In this case, the previous maximum height is updated to the current height and a **good jump** signal is given.

A new round starts in one of 3 cases: there has been a bump, a frog has been saved or it has been decided to start again.

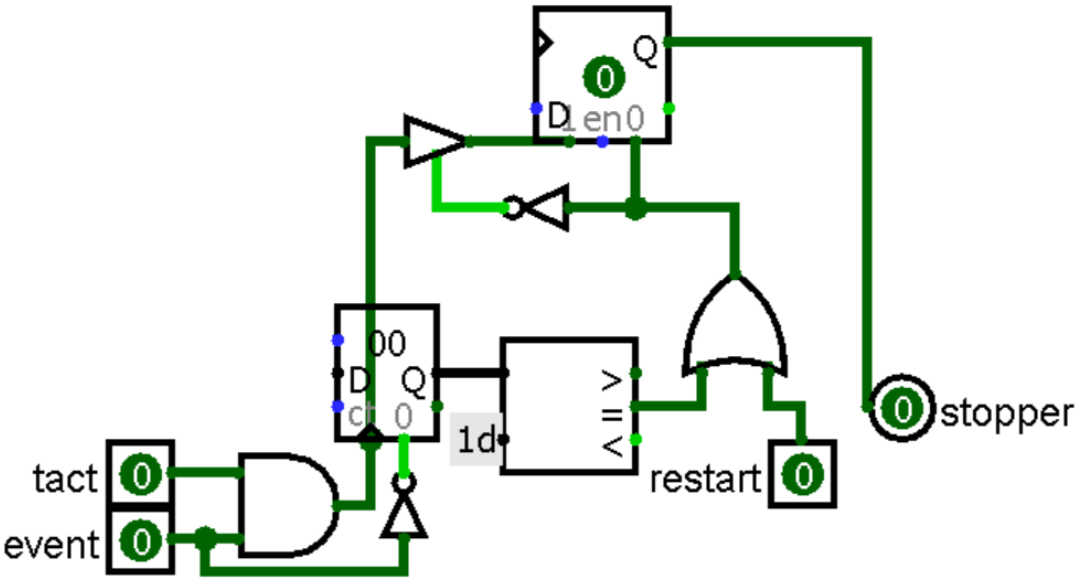


When an event occurs, it means that the game is either won or lost. In any case, at that moment, all actions in the background must be slowed down. This function is performed by the **stopGame** sub-circuit.



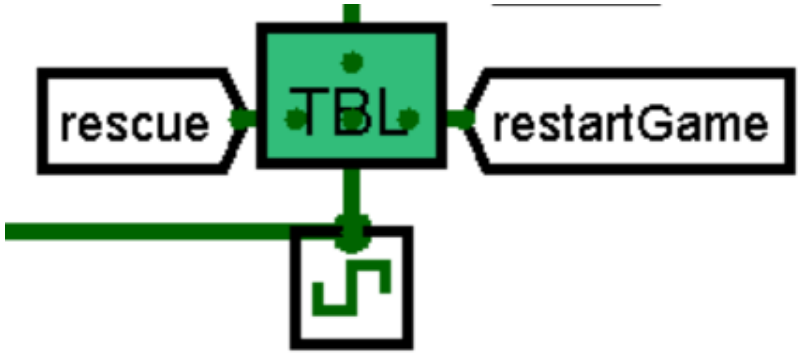
The signal going to the top left corner is the **clock**, which stops when an **event** occurs. There are 3 input signals and 1 output signal in the top right corner of the board.

From the inside, it looks like this:



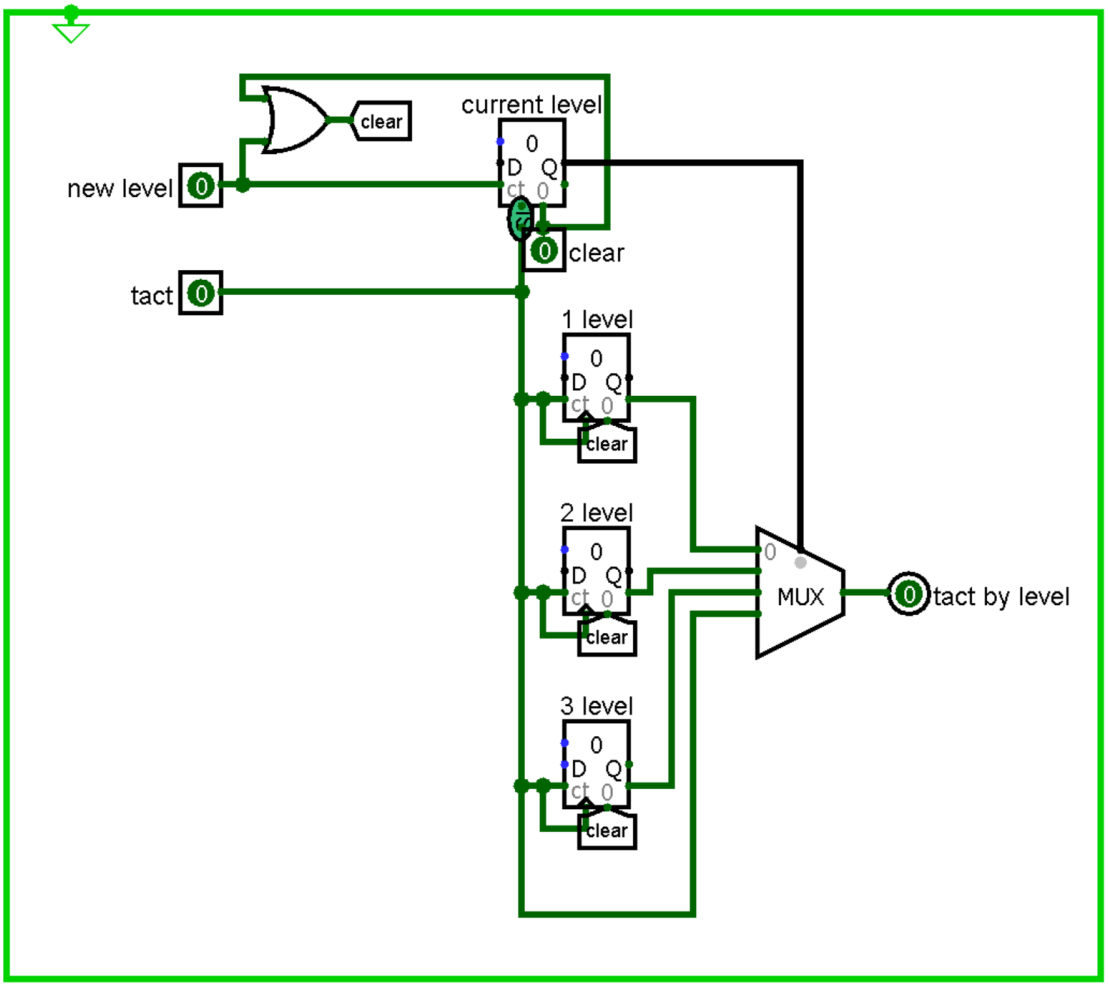
There is a counter and an S-R trigger. If the **event** signal is low, nothing happens. But if it is high, the signal of the **stopper** is raised by the S-R trigger and with each new **tact** the value in the counter is incremented until it reaches 1d in hexadecimal notation. Then the S-R trigger is cleared and the signal of the **stopper** is dropped. However, if the **event** signal is still high, the signal of the **stopper** will also rise rapidly. This is repeated over and over again. This delays the clock signal so that everything in the background is slow.

The game also has difficulty levels. With each new saved frog, the drivers and the flow speed up and the sun gets hotter. The following scheme is used to implement them:



It has 3 input contacts and 1 output contact, it's a level tact.

This is what it looks like from the inside:

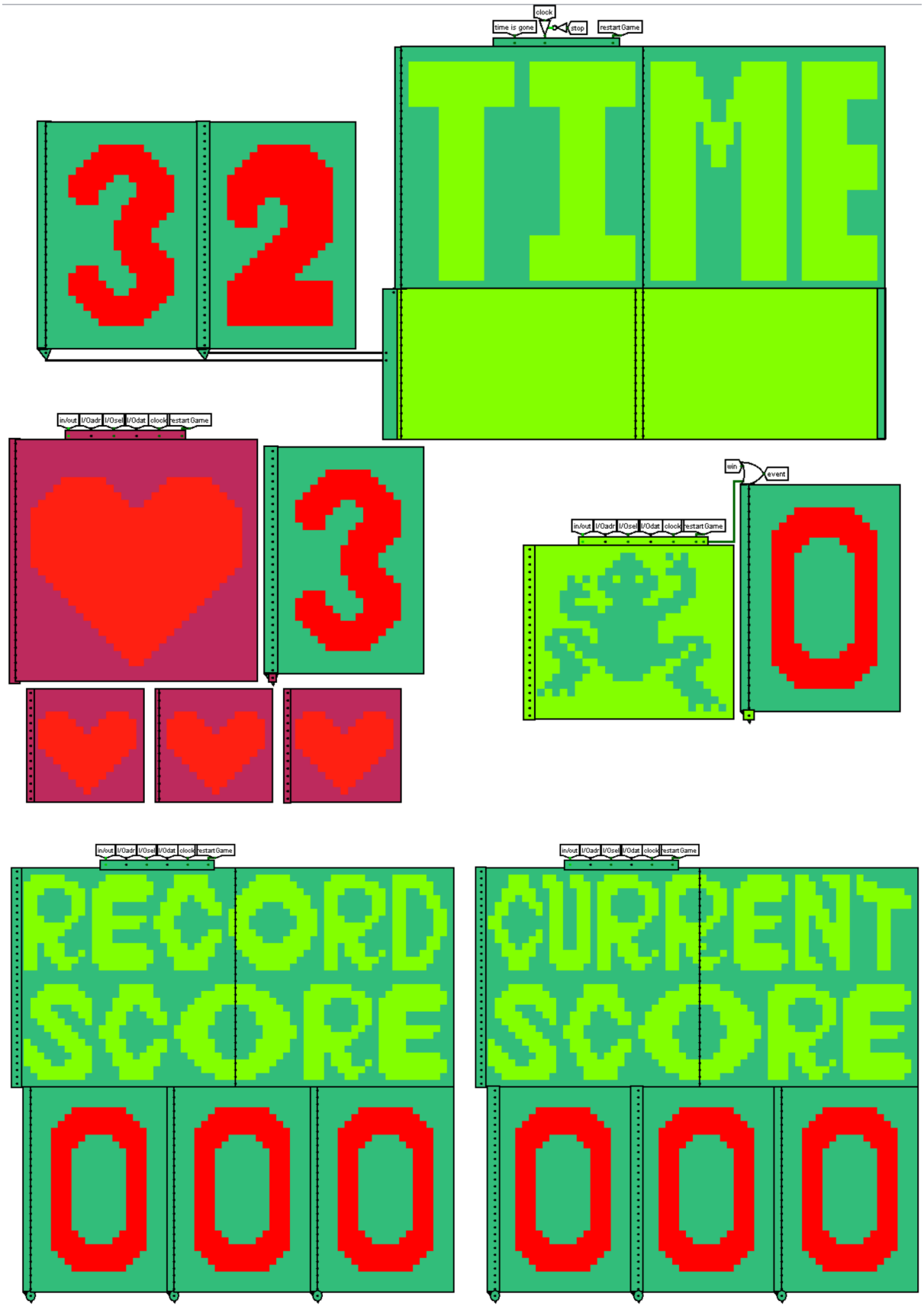


The **tact** signal and the **new level** signal are received, indicating that it is time to increase the difficulty of the game. There are 4 counters, one of which stores the current difficulty level, and the other 3 give the **t**acts **b**y **l**evel. The multiplexer decides which clock to output. There are also two **clear** signals. The first is sent to the circuit and clears the counter that stores the current difficulty level, and the second is received when the difficulty level is increased or the **clear** signal is sent to the circuit.

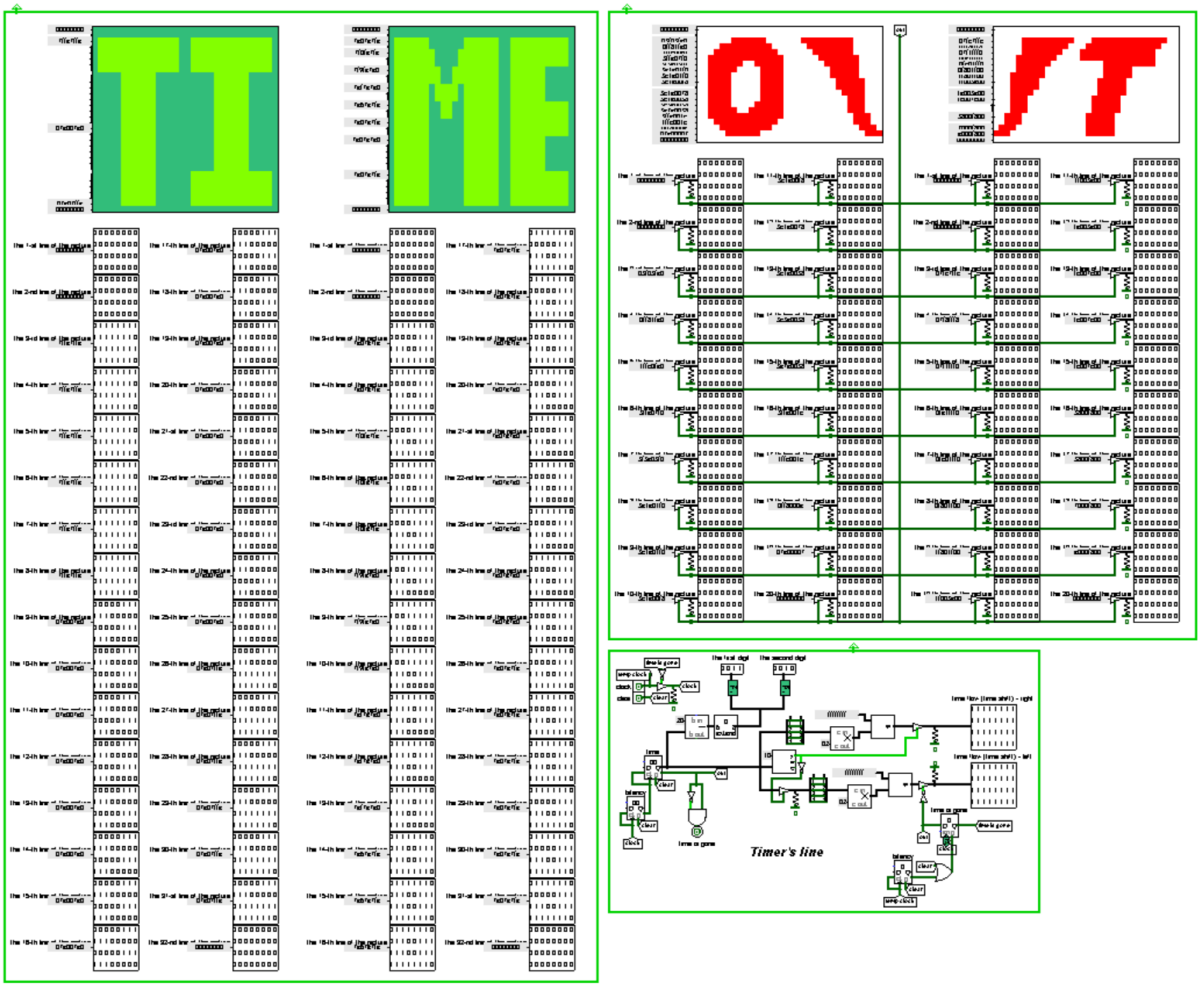
The maximum difficulty level is four. Therefore, the counter that stores the current difficulty level is reset when it reaches 0x3 (the count starts at 0). Each level also has its own reset limit. The **first** level is the slowest, so it takes four attempts to get through it. Each next level requires one less attempt than the previous one. This means that at the **second** level every third attempt is successful, at the **third** level every second attempt is successful and at the **fourth** level every attempt sent to the circuit is successful.

Thus, at the hardest level, every **tact** that changes the state of the game is passed, which increases the speed of events, and at the easiest level, on the contrary, there is a slowing down of tacts, which means that a **tact** passes through time.

This is what the statistics look like:

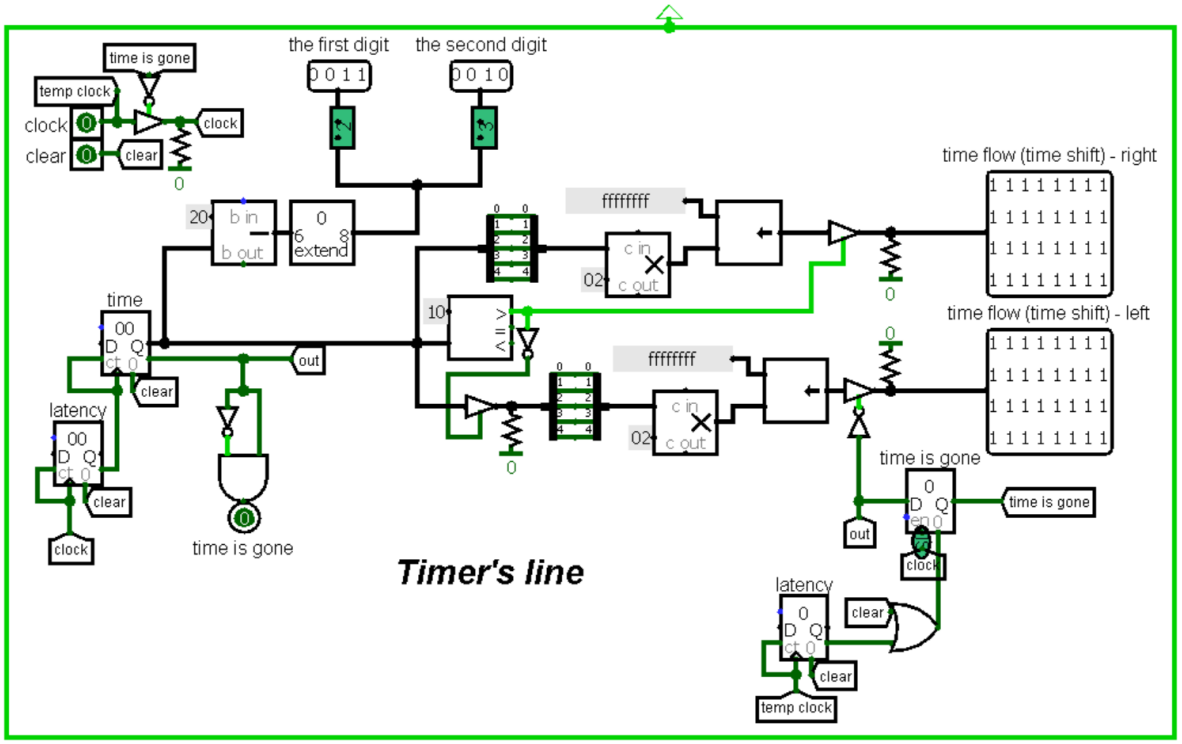


This is what the circuit for displaying of the time looks like:



The majority of the circuit is made up of constants, whose bit representation, using a zero and a one, indicates which of the two colors the pixels in the picture should light up. There are 64 output contacts to display the TIME text and 40 output contacts to display the OUT text when the time is gone. If there is still time, OUT should not be displayed. Controlled buffers and terminating resistors are used to achieve this. If the **out** signal is low, this means that there is still time and the controlled buffers will not output anything, but the terminating resistor will set 0 on each wire leading to the OUT text output pin.

This part of the circuit is where the main action takes place:



There are 2 input pins and 5 output pins. The **clock** and **clear** input pins have the same meaning as before. The difference is that a controlled buffer is used which does not pass the **clock** signal to delay the display of the OUT text when the time has elapsed. As a result, the **clock** signal may be undefined, so there is a terminating resistor to set the undefined value to zero. It also uses the **temp clock** signal to count the tacts and, having reached a limit on how long the OUT text should be displayed at each elapsed time, allows the **clock** signal to pass again to start counting the time again.

There are 3 counters and 1 register. The elapsed time is stored in the 6-bit counter labeled **time**. The **latency counter** next to it delays the elapsed time so that 32 seconds do not elapse in one instant. When the **time** counter reaches 0x20, the **time has gone** signal is raised for a moment. Above 0x20 (or 32 in decimal notation), the elapsed time is subtracted to obtain the remaining time.

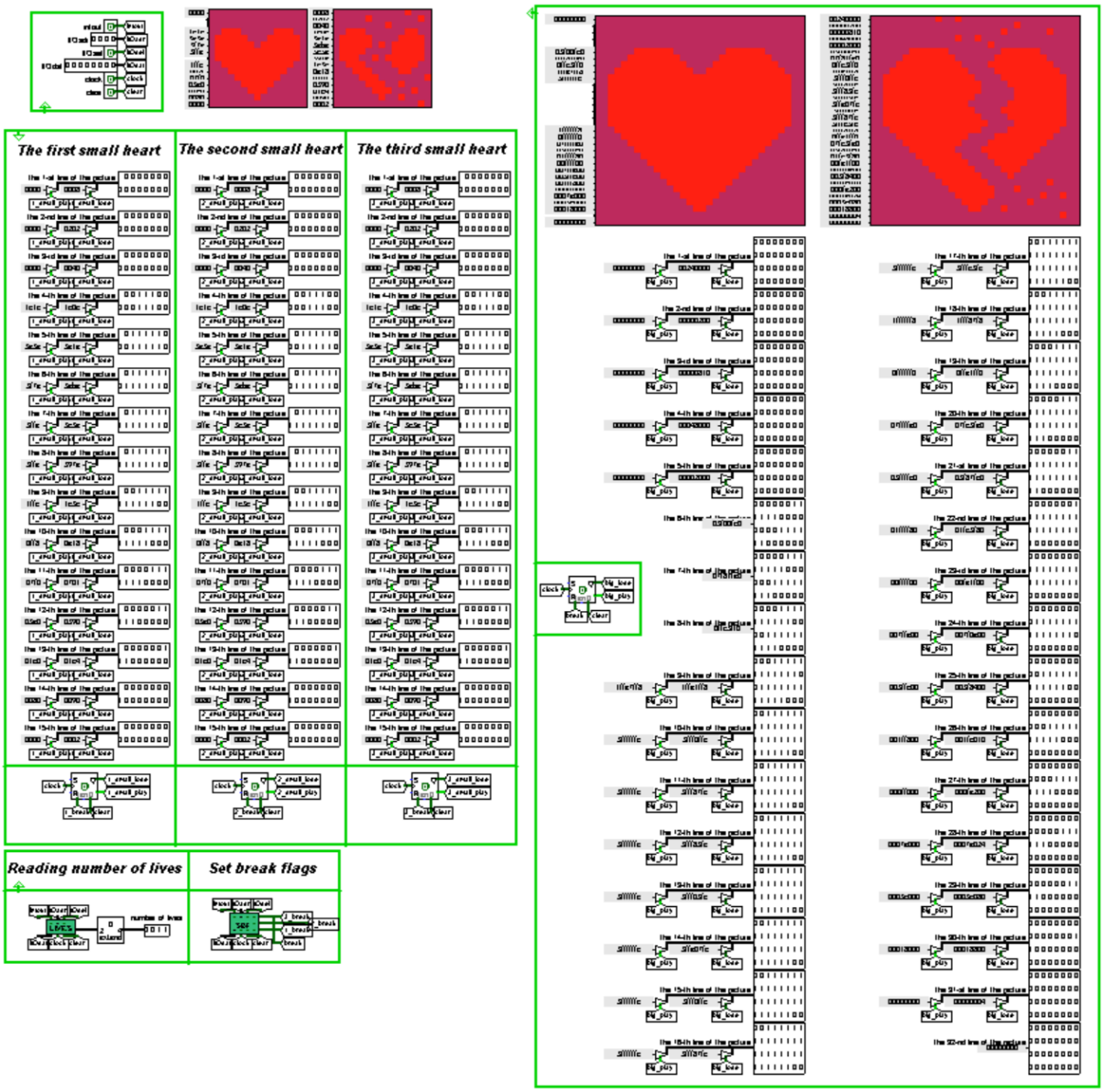
This is then expanded to 8 bits to be entered into sub-circuits that take the last two digits of a three-digit number. These digits are displayed.

As the line simulating the passage of time consists of two matrices, there are two output contacts and two controlled buffers for each of them. The matrix on the right is designed for the upper output contact, and the other matrix is designed for the lower output contact. The columns of the right matrix are only shifted if the elapsed time is less than 0x10, otherwise they are set to zero using a terminating resistor. Otherwise, the elapsed time is passed to the lower wire. If the **out** signal is low, then the left matrix columns are shifted.

To make the shift, the elapsed time is first shortened to 5 bits, as there are only 32 columns in the matrices. The elapsed time is then doubled to make the shift more noticeable. Finally, the value consisting of single ones is shifted by the number obtained in the previous step.

A register is used to store a flag indicating that the delay time for displaying the OUT text has elapsed. There is an auxiliary 2-bit counter next to it which is used to obtain the delay. The flag is raised until the counter reaches the maximum value or the **clear** signal is low.

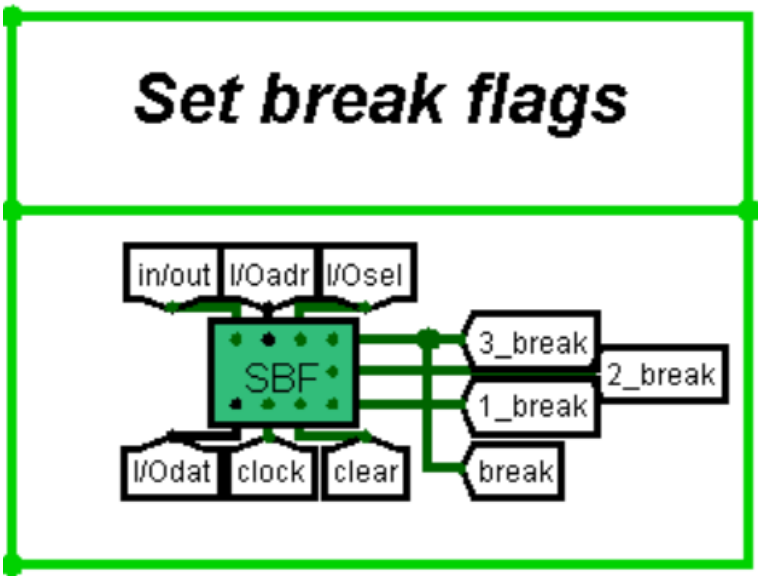
Here's what the remaining lives display looks like:



The main part of the circuit for displaying hearts. There are 32 output contacts for the large heart display, these are located on the right-hand side of the circuit. This heart represents the state of the game, i.e. if the game continues it is intact, but if a player loses it is broken. And the columns on the left are for displaying each small heart.

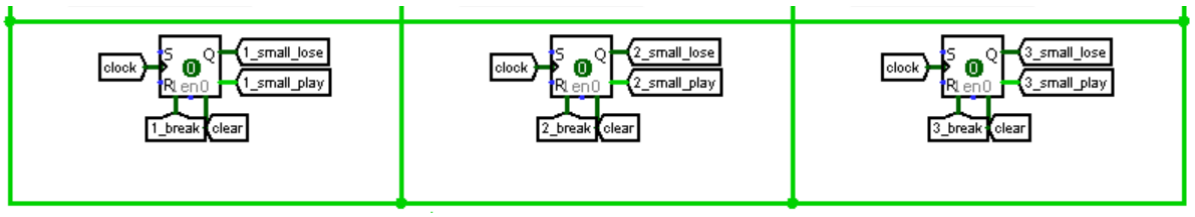
In the bottom left corner, the **number of remaining lives** is read out using the **get number of lives** sub-scheme.

To indicate that a heart has been broken, flags are used, which are set using the **s**et **b**reak **f**lags scheme. But it needs data from the processor, so there are 6 input contacts at the top left of the circuit.

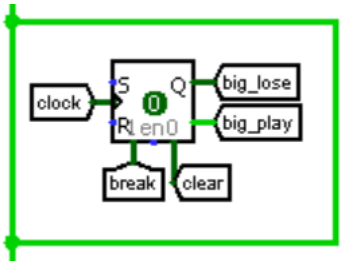


If the **break** signal is raised, it means that a big heart has been broken. If the **1_break**, **2_break**, **3_break** signals are raised, then the corresponding small hearts have been broken. S-R triggers are also used to set the flags. And controlled buffers are used to decide which image to display.

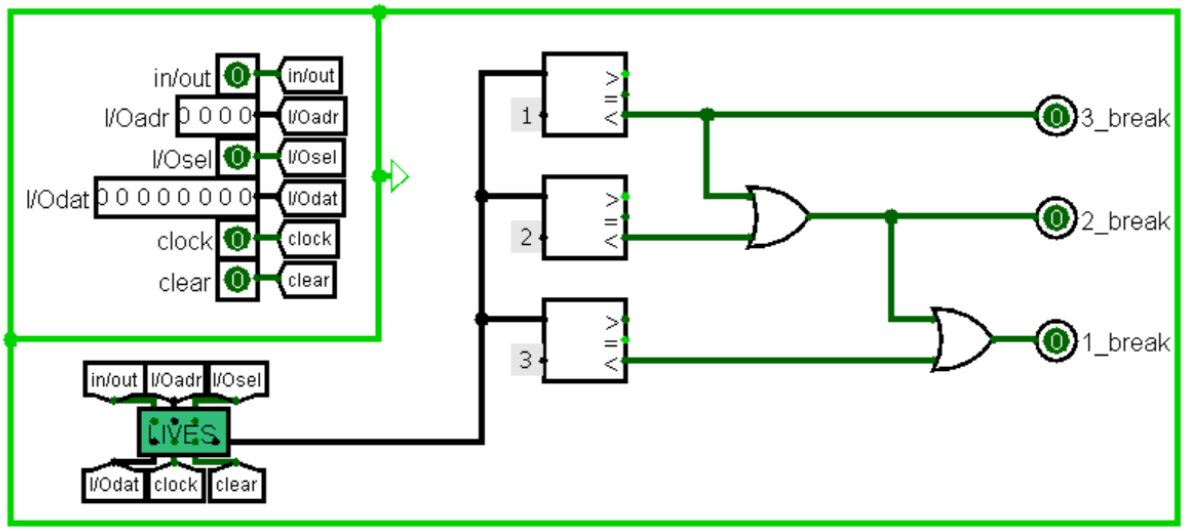
- For the small hearts:



- For the big heart:



This is the inside of the **set break flags** circuit:

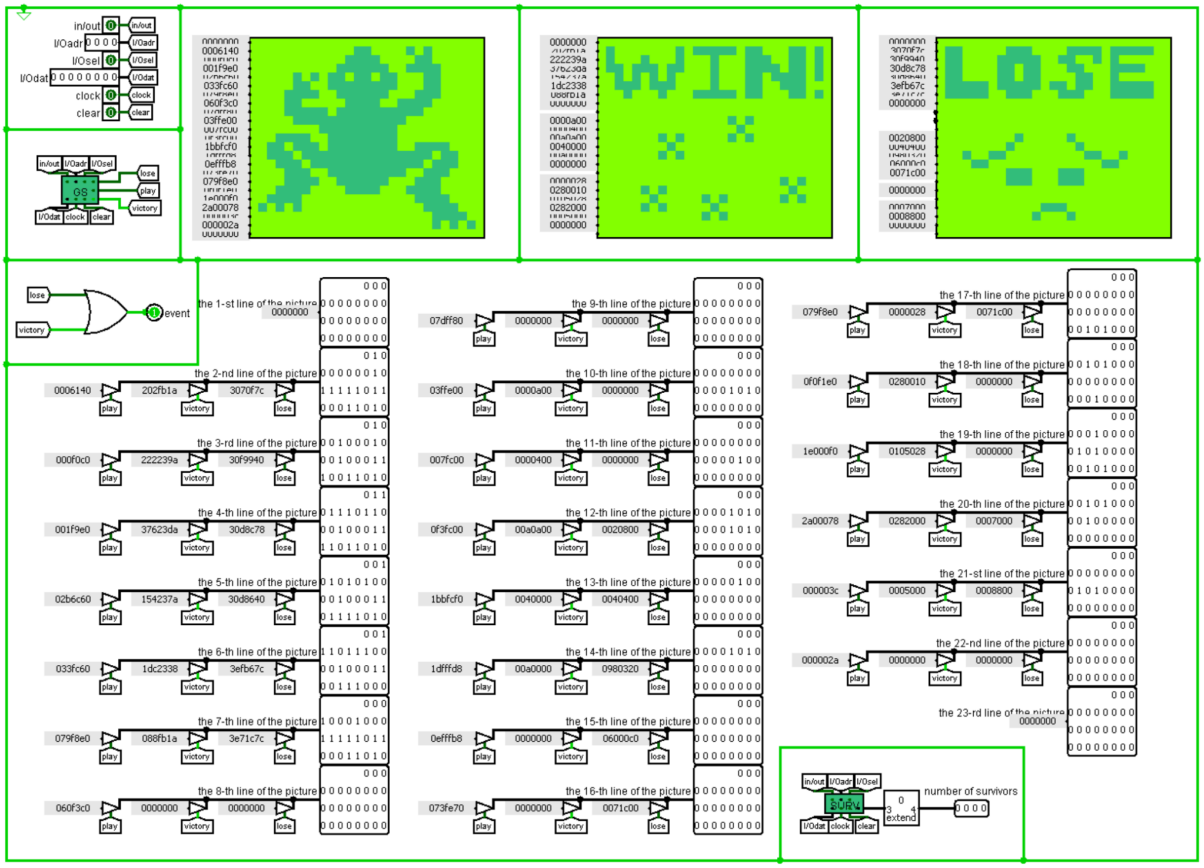


The **get number of lives** sub-circuit is used to get the **number of remaining lives**. Then the following checks are made:

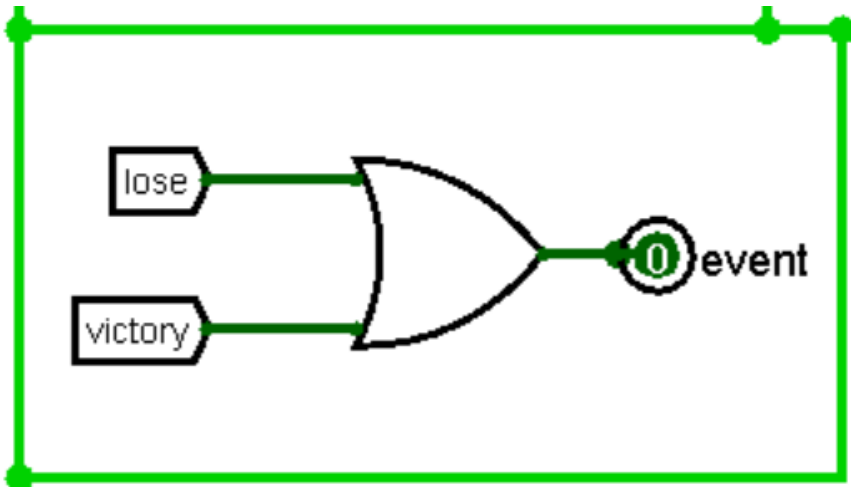
1. If there are less than 3 hearts, then the first small heart is broken.
2. If there are less than 2 hearts, the second small heart is broken and the first heart is also broken.
3. If there is less than 1 heart, then the third small heart is broken and the rests are also broken.

This is done by linking a current heart to a previous heart, using OR elements.

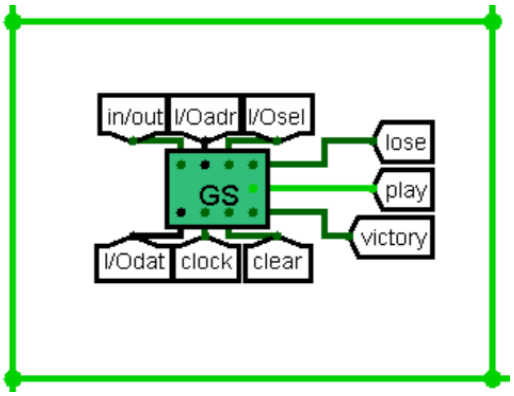
This is the circuit for displaying the **game status** and the **number of saved frogs**:



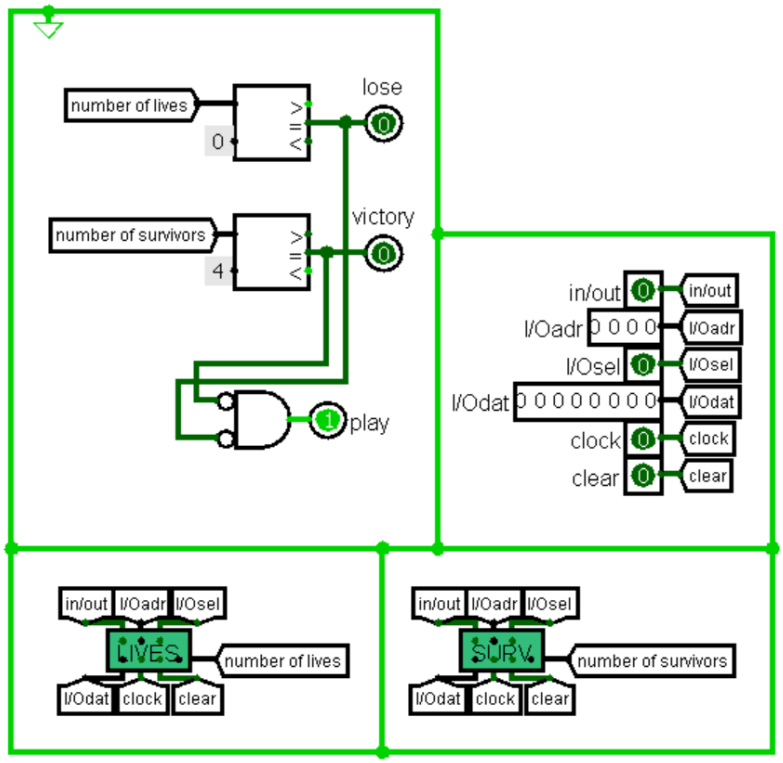
One of the three images is displayed using controlled buffers. The **number of saved frogs** is determined by the sub-scheme in the bottom right-hand corner. An **event** signal is used to slow down all background activity and disable keyboard input, and is triggered when a player wins or loses.



The following sub-scheme is used to set the **g**ame **s**tate:

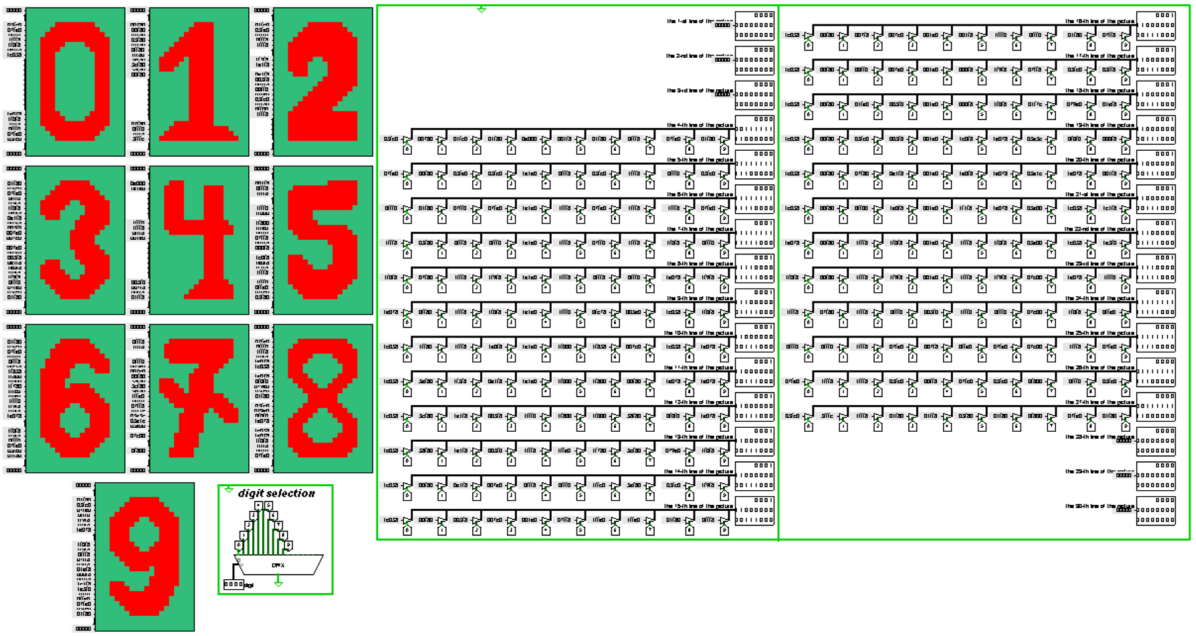


Inside, it looks like this:

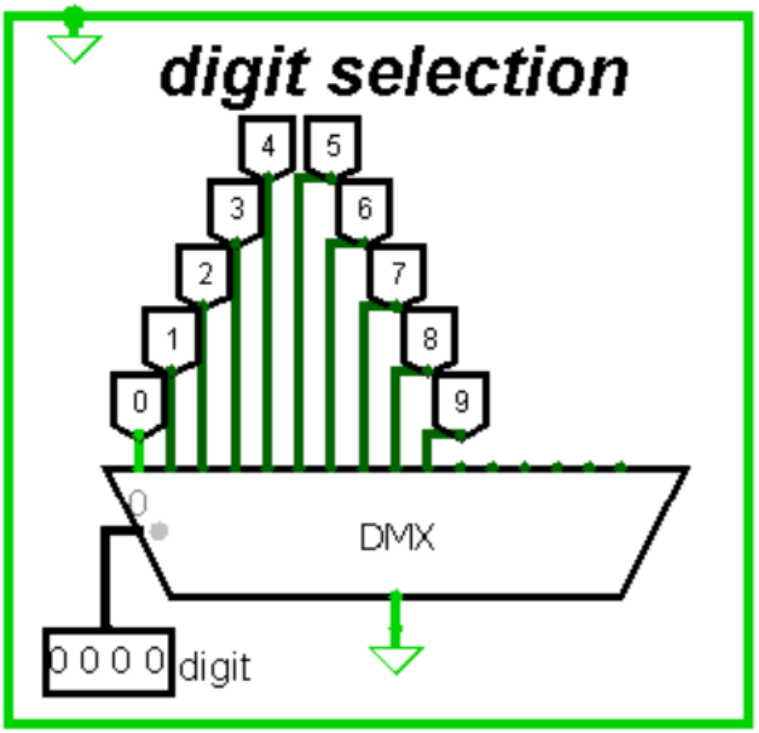


There is the usual set of input contacts and three output contacts to indicate the game status. The **number of remaining hearts** and **the number of saved frogs** are read from the sub-circuits at the bottom. If there are no hearts left, a **lose** signal is set. If **the number of saved frogs** is 4, the **victory** signal is set. If there are no **lose** and **victory** signals, the game continues and the **play** signal is raised.

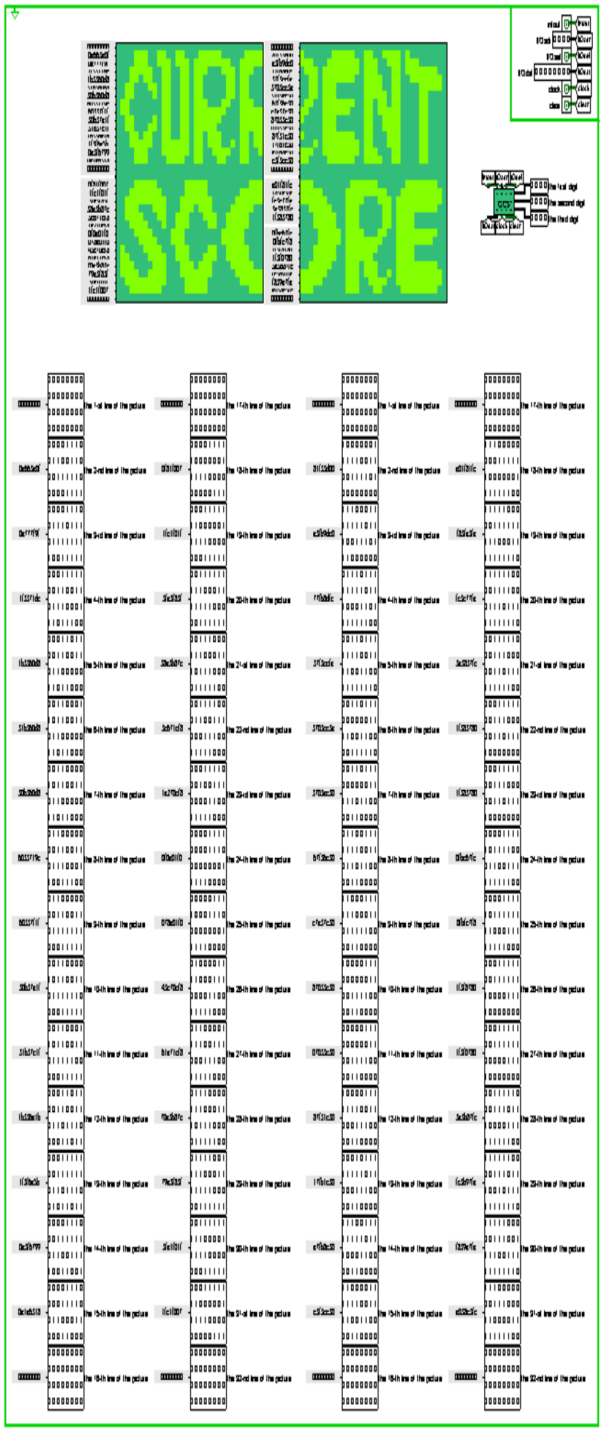
Digits are displayed according to the following scheme:

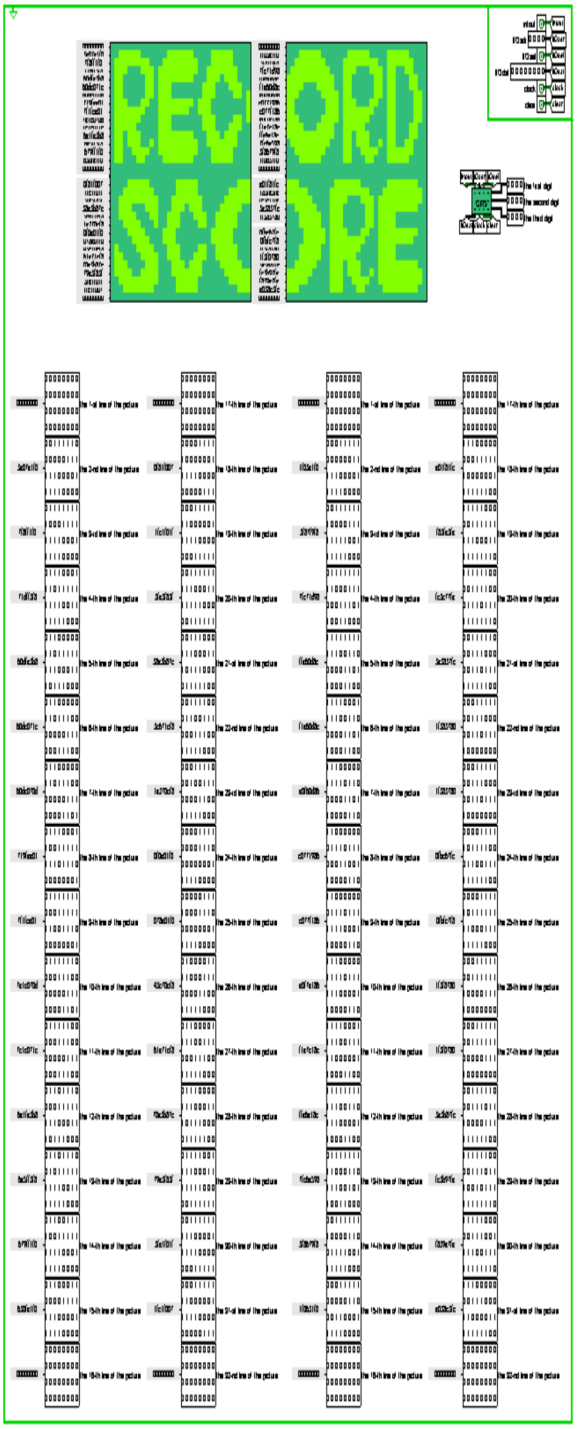


A **digit** between 0 and 9 is taken as input. A demultiplexer decides which digit to display. Then the constants required for the digit are sent to the output by means of controlled buffers.



The following circuits are used to display the **current score** and the **record score**:

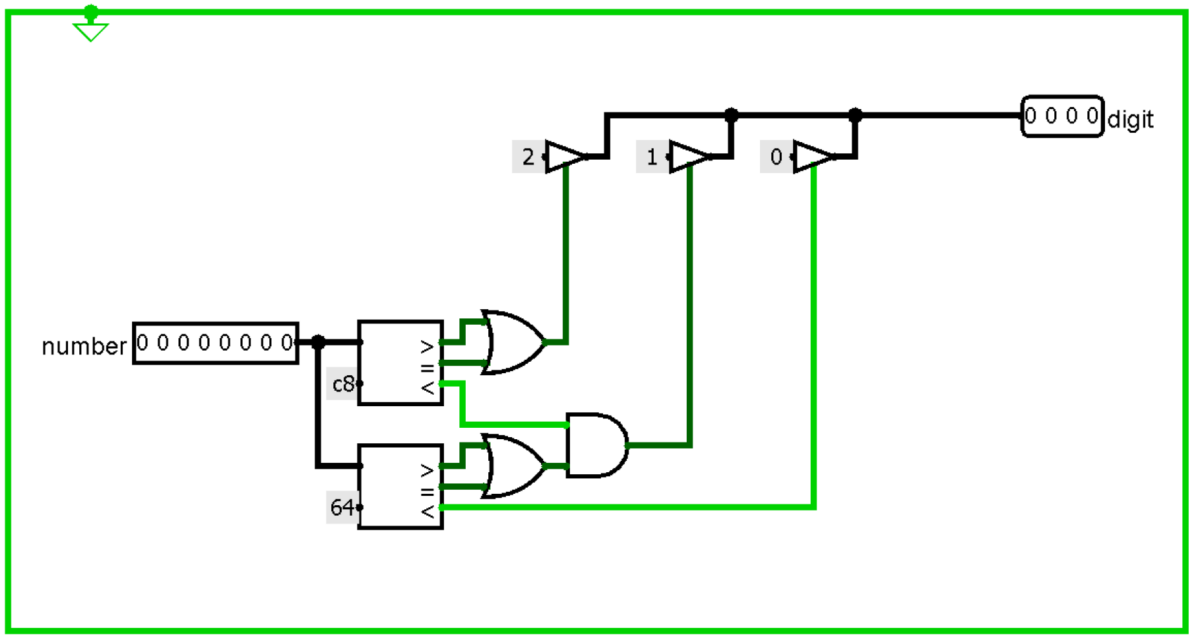




Each of these circuits has 64 output contacts for displaying text, and 3 output contacts for displaying the score digits separately. The scores are obtained by using the appropriate sub-circuits.

The **get current score** and **get record score** sub-circuits use the following sub-circuits:

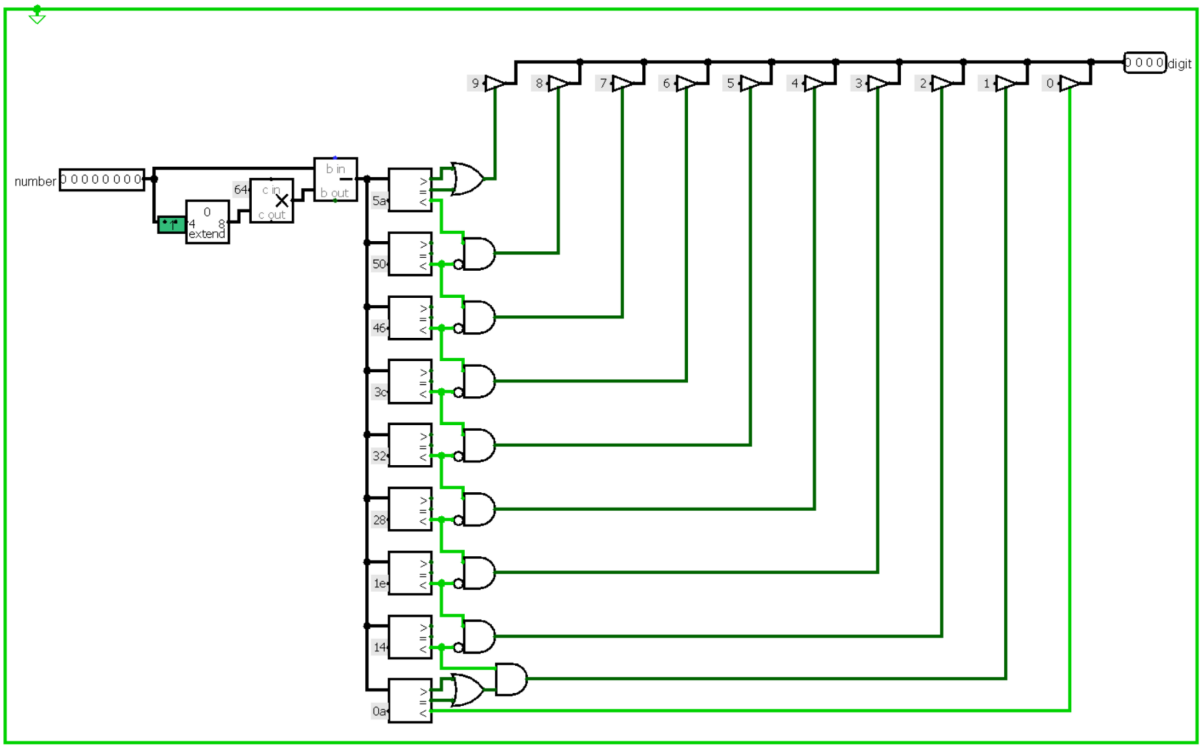
- **Get the first digit**:



The input is an 8-bit **number** not exceeding 255 in decimal notation. The next step is a conditional branching:

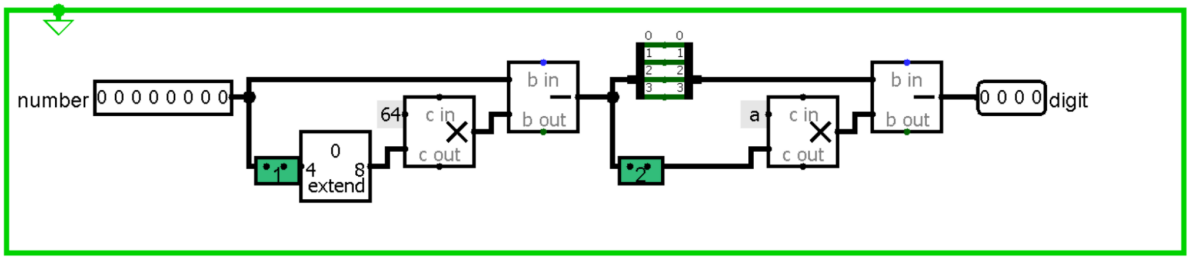
1. If the **number** is greater than or equal to 0xc8 (200 in decimal notation), the first **digit** is set to 2.
2. If the **number** is greater than or equal to 0x64 (100 in decimal notation) and less than 200, the first **digit** is set to 1.
3. If the **number** is less than 0x64, the first **digit** is set to 0.

❖ **Get the second digit**:



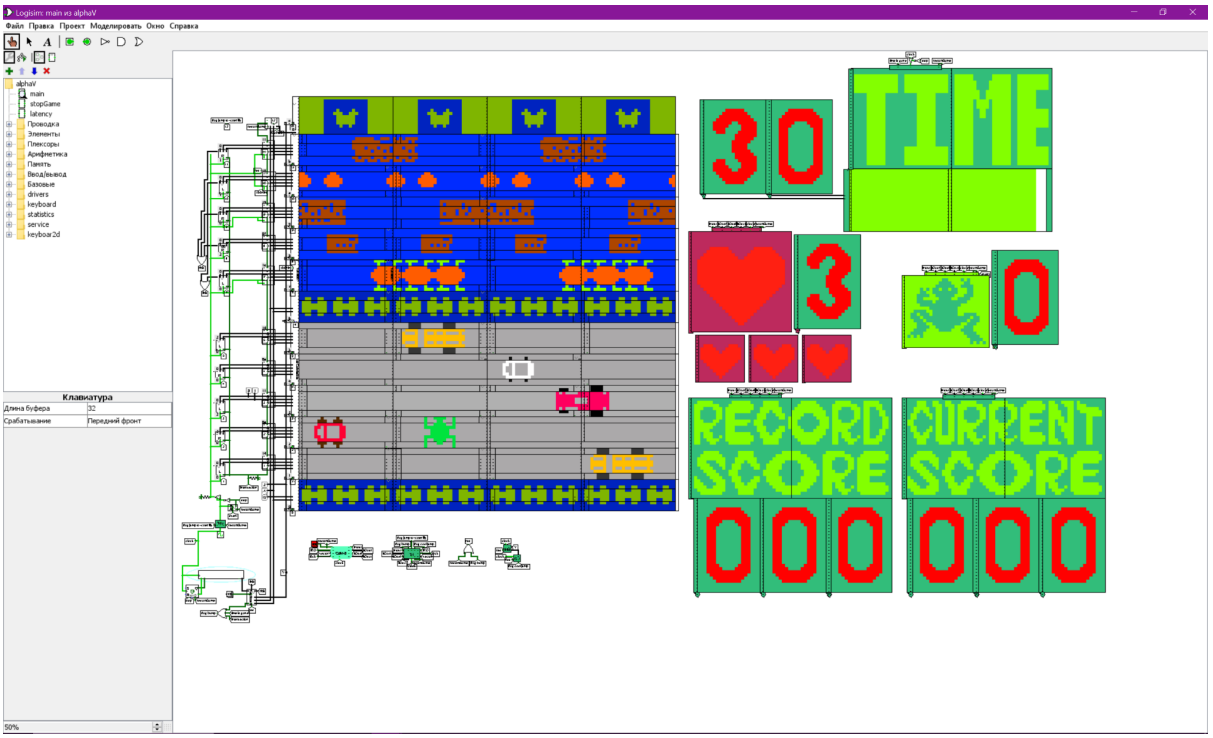
An 8-bit **number** is received as an input. Using the previous circuit, the first digit of the number is taken, expanded to 8 bits (the output pin of the previous circuit is 4 bits), then multiplied by 100, the result is subtracted from the original **number**, leaving only the last two digits. The second **digit** of this number is then determined using 9 comparators.

❖ **Get the third digit**:



The previous two sub-circuits subtract all hundreds and tens from the **number**, leaving the last **digit** to be output.

User Manual



A player needs to move the maximum number of frogs across the road and the swamp for 3 hearts. Each heart is broken if a frog dies. A frog dies if it is crushed by a car, if it floats away on a log or turtles over the edge of the screen, if it jumps into the water, if it runs out of time it will wither and die. A player can move the frog vertically by pressing 'w' up, 's' down, 'a' left, 'd' right. You can press any of the four buttons to make a frog take several steps at once. If the player has spent 3 hearts, he can try to go through the game again by pressing res.

Conclusion

We have done everything with reference to the "Technical specifications" file and also added some of our own. The cars and frog models, and the appearance of the playing field have been made to resemble the original "Frogger" game. We were creative in designing the look of the statistics. We tried to make it look like arcade games from the last century.

By working as a team on this project, we gained skills in time management, creating circuits and writing code in the low-level language Assembler, as well as experience in teamwork and the ability to distribute tasks properly. The result is a colorful and interesting game which was made in a very short time.