Wireworld Project

De Wiki LOGre
< Projet Wireworld

Language:	**Français • English**

Sommaire

Wireworld (https://en.wikipedia.org/wiki/Wireworld) is a Cellular automaton (https://en.wikipedia.org/wiki/Cellular_automaton) with few simples rules but that is Turing-complete (https://en.wikipedia.org/wiki/Turing-complete) and that allow to simulate electornic logic elements. You can find more details on its Wireworld wikipedia page (https://en.wikipedia.org/wiki/Wireworld)

Vocabulary

- **Generation** : iteration number of automaton. First generation is generation 0.
- **Period** : in case of repetitive thing period define the number of generation

needed to come back in the same stategenerations necessaires pour revenir au meme etat
- **n microns technology** : n is the period between 2 electrons in an electron burst

Simulators

I wrote my own simulators for Wireworld automaton :

- A version whose simulator core is C++ based : Repo (https://github.com/quicky2000/P_wireworld/tree/master/sources/wireworld)
- A version whose simulator core is [1] (https://en.wikipedia.org/wiki/SystemC) based : Repo (https://github.com/quicky2000/P_wireworld/tree/master/sources/wireworld_systemc)

I use additional libraries like lib SDL 1.2 (https://www.libsdl.org/) for graphical display and xmlParser (http://www.applied-mathematics.net/tools/xmlParser.html) for XML file parsing

Ressources sharing

Everything that is independant from simulator's core is located in a common package (https://github.com/quicky2000/wireworld_common/) containing:

- Parser used to read automaton description
- Analyser which compute design partition and neigborhood to determine parts that will be simulated
- Generic configuration XML parser that allow design parametrisation
- Configuration parser which define design parameters

File formats

Input file

Automaton is described in a text file where each cell is represented by a character:

- **.** : empty cell
- **#** : copper cell
- **E** : electron head cell
- **Q** : electron tail cell

By example :

```
.....................................................................#.......#.#....................
.....................................................................#.......#.#....................
.....................................................................#........##....................
.....................................................................#.........#....................
```

```
...#################..###.......###.......###.....#.###.......###.....#.###.....#.###.......###......
..E.................###.##.##.###.##.##.###.##.##.###.##.##.###.##.##.###.##.##.###.##.##.###.##.##.#
...Q################..#.#.#..#..#.#.#..#..#.#.#..#..#.#.#..#..#.#.#..#..#.#.#..#..#.#.#..#..#.#.#..#.
......................###.#.###.###.#.###.###.#.###.###.#.###.###.#.###.###.#.###.###.#.###.###.#.###
.......................#..#..#...#..#..#...#..#..#...#..#..#...#..#..#...#..#..#...#..#..#...#..#..#.
.......................#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#.
.......................#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#...#.#.#.#.
........................#..#.#....#..#.#....#..#.#....#..#.#....#..#.#....#..#.#....#..#.#....#..#.#.
..........................#..#......#..#......#..#......#..#......#..#......#..#......#..#......#..#.
..........................#..#......#..#......#..#......#..#......#..#......#..#......#..#......#..#.
..........................#..#......#..#......#..#......#..#......#..#......#..#......#..#......#..#.
...........................##........##........##........##........##........##........##........##..
```

Generic configuration file

The file format is XML and it defines items associated to cell coordinates that will be electron head or tail.
Coordinates defined in items are relative to coordinates defined at the beginning of file.
When an item is configured then associated cells will be electron head or tail, else it initial state will be the one defined in input file

```
<?xml version="1.0" encoding="UTF-8"?>
<generic_definition version ="1.0">
  <origin coord="178,182"/>
  <item_list>
    <item name="init_pulse_1" e_head="-8,11"/>
    <item name="init_pulse_2" e_tail="-15,11"/>
    <item name="shift_1" e_head="-167,13" e_tail="-168,13"/>
    <item name="shift_2" e_head="-173,13" e_tail="-174,12"/>
  </item_list>
</generic_definition>
```

Configuration file

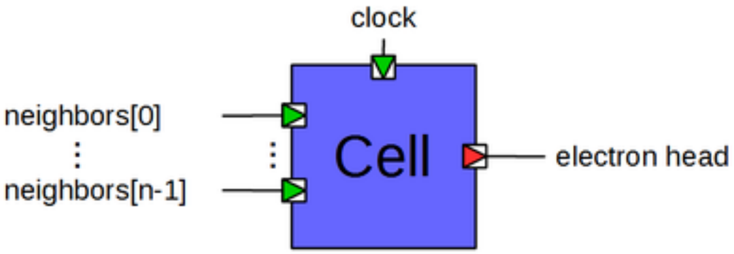
It complete the generic configuration file by defining if items are active or not.
File format is basic : **item_name:[0|1]**
Here is an example :

```
# Test file
init_pulse_1:1
init_pulse_2:1
shift_1:0
shift_2:1
#EOF
```

SystemC simulator

Each automaton's cell is represented by a SystemC module containing

- An input boolean port for clock signal
- An array of input boolean port whose size depends on neighbor number
- A boolean output port indicating if cell is in electron head state



SystemC module contains a SystemC process sensitive on clock that will do the following depending on cell's interal state:

- Count number of boolean inputs whose value is 1
- Update cell's internal state
- Update boolean output depending on cell's internal state

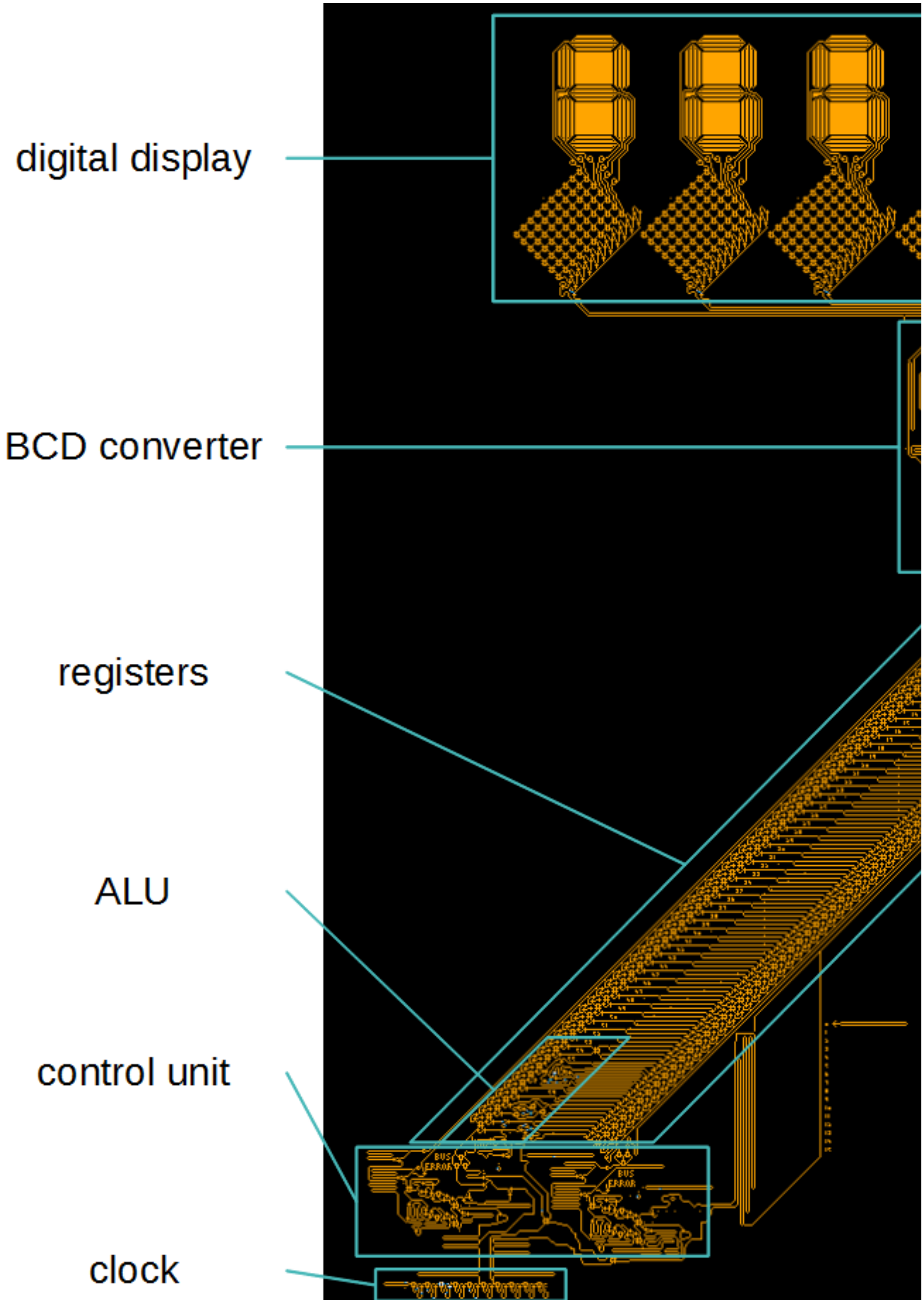
Output of neighbor cells are bound to cell's inputs.
In order to improve performances, SystemC module is templated on number of neighbor cells and the correct module type is instanciated by analyzer depending on neighbor number.

Wireworld Computer Reverse Engineering

Wireworld computer (http://www.quinapalus.com/wi-index.html) is a design based on wireworld cellular automaton. It defines an URISC (https://en.wikipedia.org/wiki/One_instruction_set_computer#urisc) processor with TTA (https://en.wikipedia.org/wiki/Transport_triggered_architecture) architecture containing 64 registers of 16 bits
This design has been realised between 1990 and 1992 by David Moore, Mark Owen and some other people and can be considered as a precursor of what some people do today with Minecraft and its Redstone extension
A Turing Machine based on Game of life had been designed several years before Wireworld Computer but is programming was far les suser friendly compared to the one of wireworld computer
When a discovered this design and saw it simulated I was immediately fascinated. The web site explain its global operating but major part of the design is not explain in details so I was interested in understanding the following points:

- how the design operate
- how complexity emerge from very few simple rules
- how authors succeeeded to overcome difficulties raised by this automaton: propagation constraints, spatial constraints due to 2D universe etc

Before reading the following it is interesting to read the few pages of wireworld computer website (http://www.quinapalus.com/wi-index.html) to understand general operating principles of Wireworld computer


digital display
BCD converter
registers
ALU
control unit
clock
BUS ERROR
BUS ERROR

Methodology

- I use my C++ simulator to run the design in performance mode
- I use my SystemC simulator to run the desing in debug mode :
  - Evolution of automaton cell states is recorded in VCD format (https://en.wikipedia.org/wiki/Value_change_dump)
  - I use GTKwave (http://gtkwave.sourceforge.net/) to open them
- For some part of the design I use Logisim (http://www.cburch.com/logisim/) to simulate them as logic gates. It allows to have a more 'understandable' view

Clock system

A simple way to generate a period P clocks is to draw a loop with size P containing a single electron.
This approach works well for small values of P but become space exepensive with huge values.
The clock system address this issue with a compact and elegant design that allow to geerate clocks with large periods in a size-contained space

It is

composed of

- A clock injector
- A chain of clock dividers

Clock injector

This is a simple loop with inject an electron in clock dividers chain with a period of 36
In the remaining part of Wireworld Computer reverse engineering the leftmost copper cell will be considered as time origin and coordinates origin for the other cells

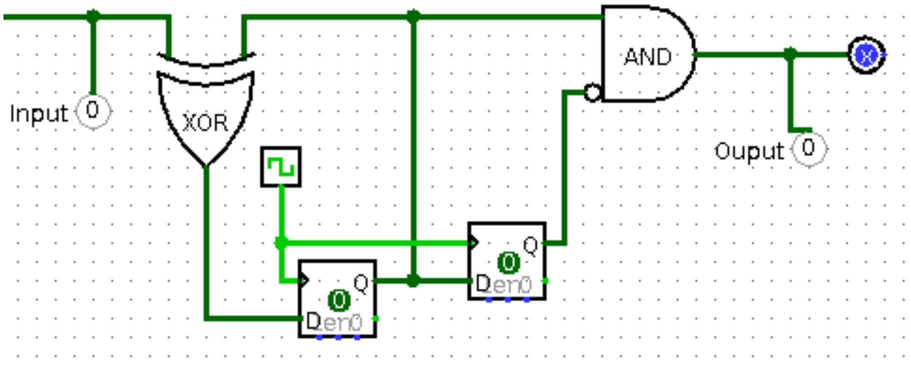
Clock divider

It allows to divide clock frequency by 2.
It is composed of 2 logic gates (XOR and A & /B), a loop and a delay path.
The loop is used as electron generator and has period of 12.

36 being a multiple of 12 and the loop being powered by clock injector, when the loop will be full an electron will arrive on the XOR gate at the same time than an electron coming from clock injector
An electron spend 4 generations to go from cell common to loop and delay path to A input of gate A & /B
An electron spend 16 générations to go from cell common to loop and delay path to B input of gate A & /B
It means the delay is 12 générations which is the period of the loop so nth electron will arrive on A & /B gate at the same time than electron (n - 1)th.
Only the first electron will arrive alone on the gate and will succeed to go ahead
In schematic below loop period and delay path are modelledf by a D flip-flop.

- At startup loop is empty. The first arriving electron will fill it. Thanks to delay introduced by the delay path the first electron go through gate A &/B but not the following electron.
- The next electron that will arrive on XOR gate will empty the loop but will not reach the output due to the lock of A& /B gate

By this way only one electron of two reach the output of clock divider



Implemented feature

Clock injector period being 36 and each clock divider dividing clock frequency by 2 (ie multiplying period by 2) the clock system allow to generate in a compact design several clock frequencies.
it is composed of 10 clock dividers and has outputs after 5h and 6h division unit which allows to have respective output frenquencies :

- After clock divider 5 : 36 * 2^5 = 36 * 32 = **1152**
- After clock divider 6 : 36 * 2^6 = 36 * 64 = 1152 *2 = **2304**

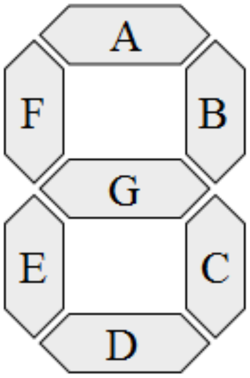
Digit display

Display of numbers in Wireworld Computer is realised by 5 digits displays
Each digit display is composed of :

- 1 7 segments display

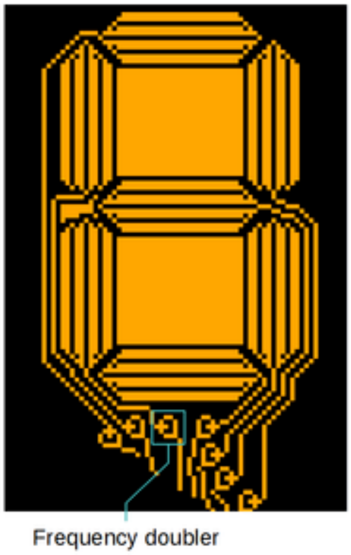
- 1 ROM with 10 inputs and 7 outputd
- 1 ROM controller

7 segments display



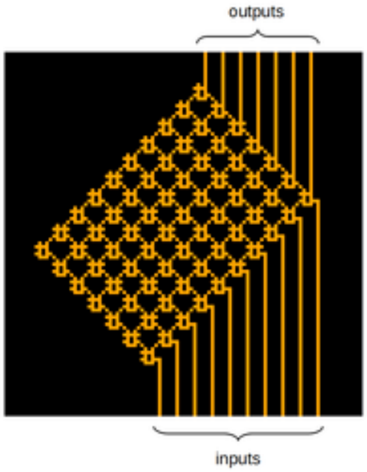
It consist of wireworld cells assembled to represent a 7 segments display
Each segment is supplied by a wire that is distributed between several wires
Wires filled with electrons represent enlighted segments.
To maximise the "shine" of a wire electrons must be very closed eacg other wich consist of using 3 micron Afin de maximiser la "brillance" d un fil il faut que les électrons soient très rapprochés, c est a dire utiliser la technologie 3 microns technonlogy which is the thinnest allowed by Wireworld The other parts of digit display are designed in 6 microns technology so each input of 7 segments display has a frequency doubler:

- Each incoming electron (period 6) is duplicated and delay of 3 generations before being reintroduced in an OR gate which generates 2 electronds with period 3



ROM

It codes the correspondancy between a digit and its represention on 7 segments display
Its operating is not the same than standard ROM. Indeed in a standard rom inputs biary code the address of the ROM that has to be read.
In this case each correspond to a single address so there are as much address as inputs which means that only one input should be active at a time : input n code for digit n.

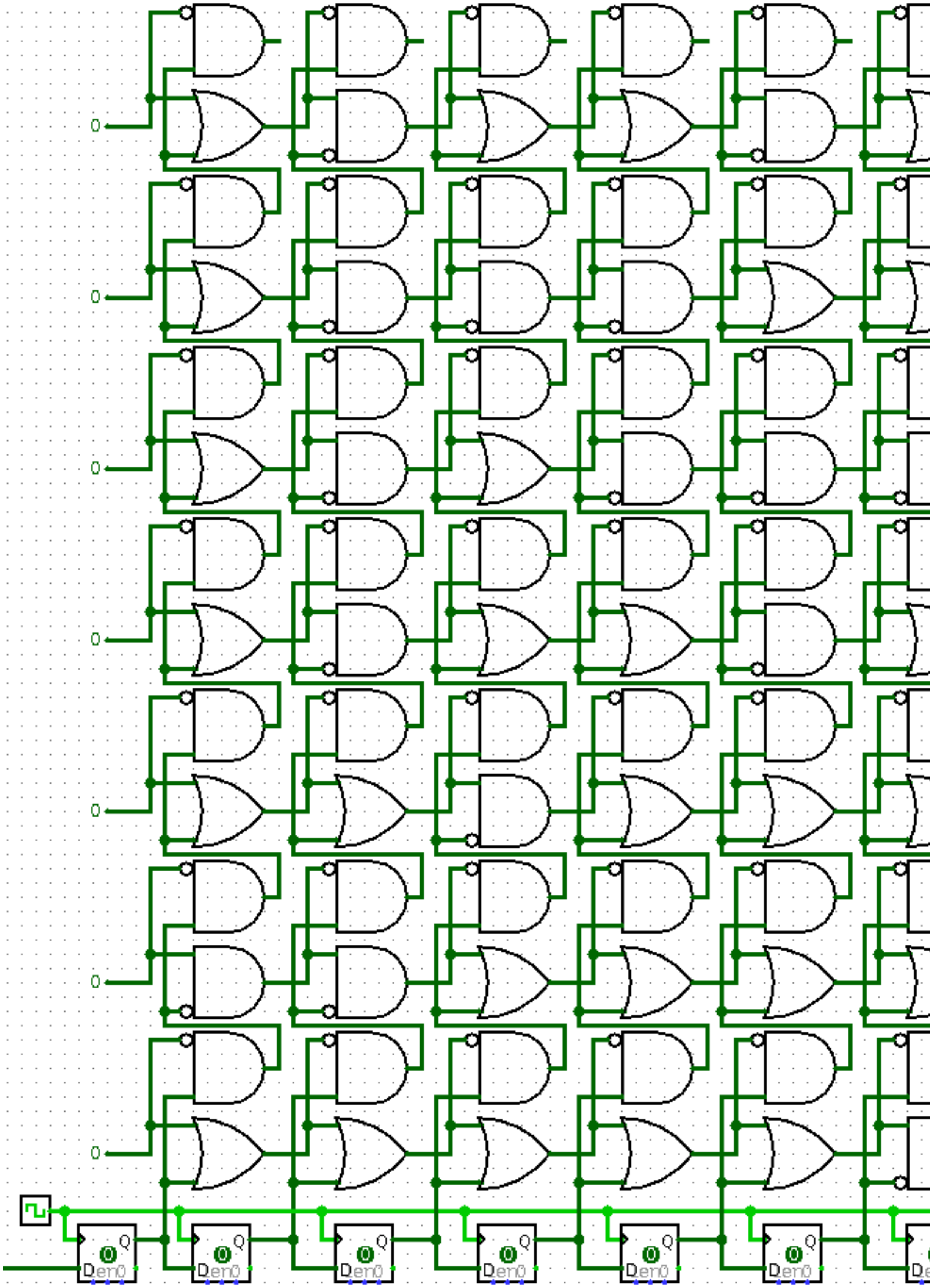


For one electron coming on a ROM input, one electron will be generated on each ouput of the ROM coding 1.
To maintain segments lighted the ROM input correspoding to the digit must be continuously supplied.
This feature is implemented by ROM controller

ROM model

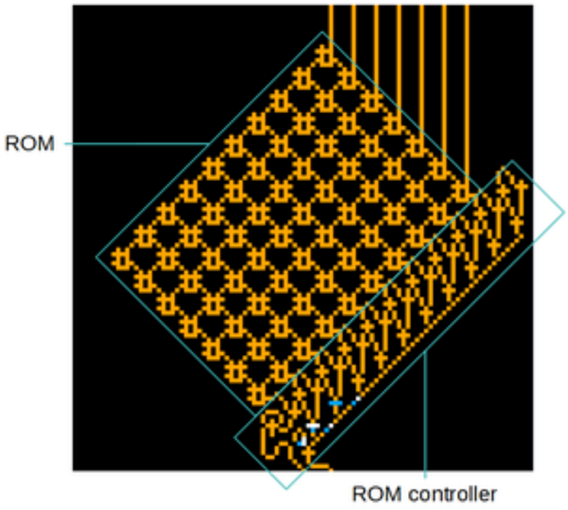
For this component electron propagation time is not meaningfull functionnally so they are not modelled.
The ROM is partially modelled: only digits 0 to 6 are managed

- In a first time circuit is empty, an electron is supplied to light the zero
- At each cycle the active column/input will be shifted to light up next digit



Contrôleur de ROM

Its role is to continuously supply ROM and to select whcih ROM input will be supplied with electrons



Each ROM input is bound to a 6 period loop going through an OR gate, to introduce an electron in the loop, and an A & /B gate, to empty the loop.
Each input loops are bound in the following way :

- A wire going out from **loop n-1** go into A input of A & /B gate, controlling loop transfer, while output of this gate is bound to input of OR gate responsible of electon introduction in **loop n**
- A wire going out from **loop n + 1** go into B input of A & /B gate controlling the **loop n** clean

By this way in case **loop n** is supplied and B input of n to n+1 transfer gate is supplied than n remain active.
In case input B is no more supplied than electron of loop n will be duplicated in loop n+1 which will clean loop n.
Thanks to this mechanism there is only one loop active at at time so there is only one ROM input active at a time.

B inputs of transfer gates are all bound on the same wire itself bound on on a A & /B gate output whose A input is driven by an electron generator of period 6.
By controling B input it is possible to stop the supply of transfer gates.
A loop transfer needs 10 generations to be performed so for each electron locked the active input of ROM is shifted by one.

Remark:

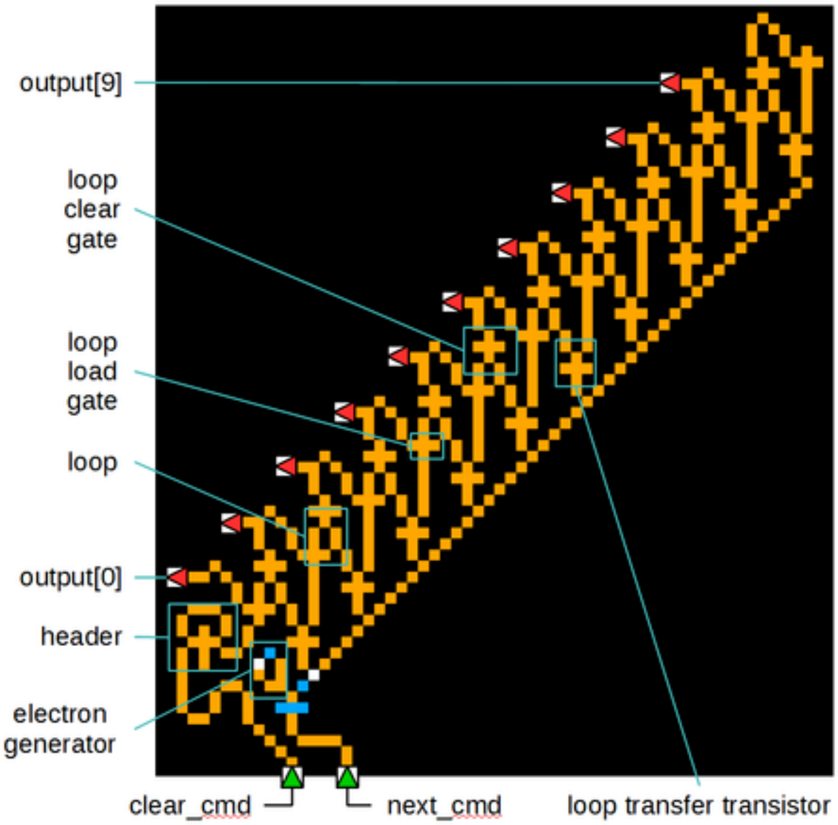
- To make the ROM controller work properly the electron stored in the loop need to arrive on transfer gate input at the same time than electron produced by electorn generator
- There is a 10th loop that allow to clean the 9th loop
- The logic gate OR controllinh the electron introduction in loop 0 is supplied by a wire going through a "tailer", a logic gate of type n & !(n+1) for 6 microns technology, which means that if n electrons arrive only ce qui signifie qui si l on envoie n electrons, only the n-1eme will go through the gate.

Operating

- In case 10 electrons are sent on this wire and on loop transfer control wire then then 10 loops bound on ROM inputd are clean and one electron is introduced in loop 0 meaning that 0 wil be displayd on 7 segments

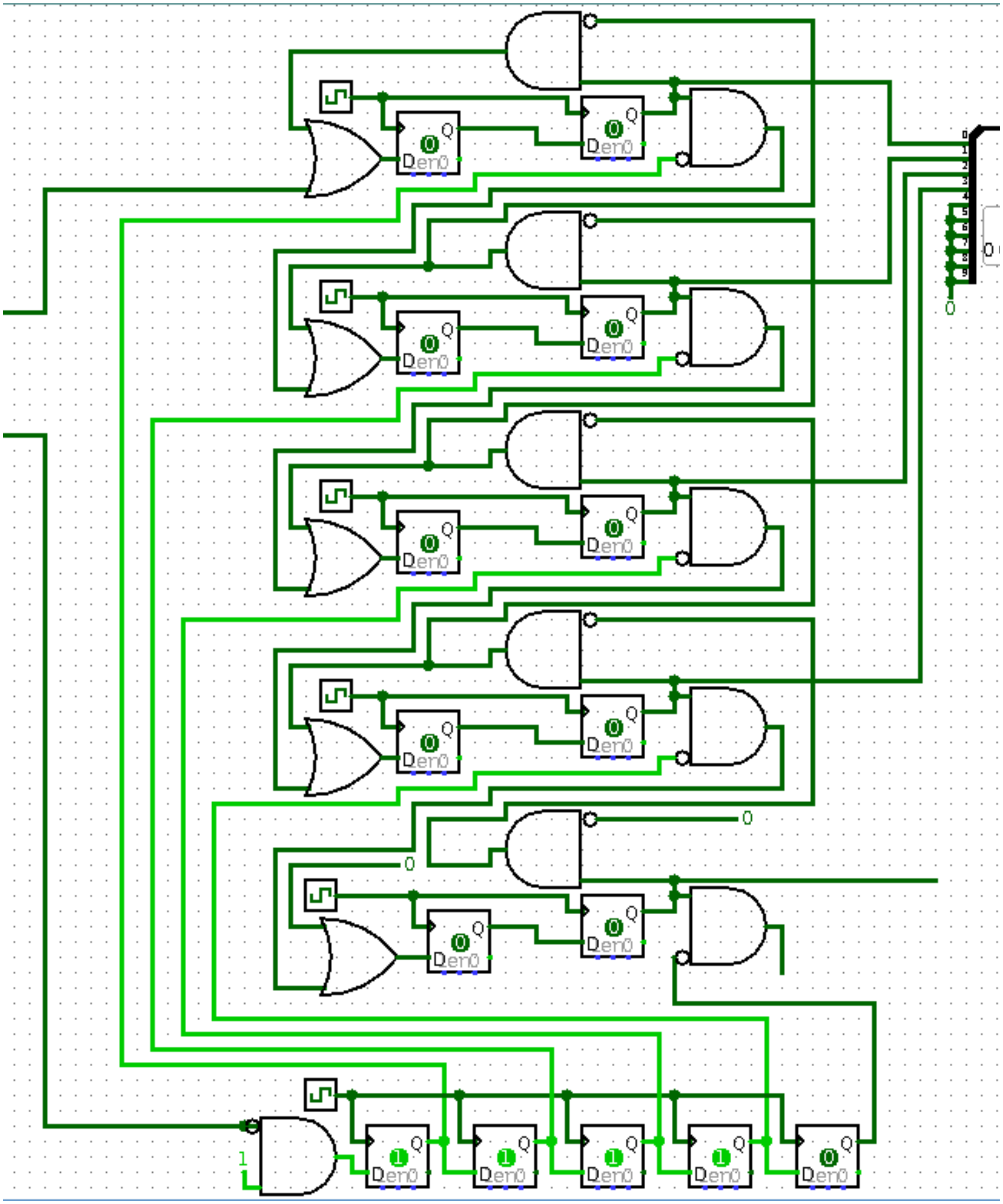
  display.
- If we want to display 5 and the displayed digit is zerp then it is needed to send 5 electrons on loop transfer control wire to activate loop 5

ROM controller model



In this example D flip-flop are there only to model electrons' propagation delay
Only loop 0 to 3 and clean loop have been modelled
To keep a reasonable model's size I use a standard ROM where only addresse having only one bit at 1 are used

- In a first time circuit is empty, an electron is introduced to display 0
- After few cycles 3 electrons are sent on B input to display 3
- After few cycles a burst of electron is sent to clean loops and reset display



Data latch

This module control :

- When data coming from register 0 is tkane in account and sent to binary/BCD converter
- Reset of digit display

Inputs/Outputs

Tt has 3 inputs:

- Data coming from register 0 (picture bottom left)
- Write command indicating that a new value has been written in register 0 (picture bottom right)
- Set command to indicate that data should be loaded in adder of binary/BCD converter (picture top right)

The Set command is sent by bottom megaloop of binary/BCD converter

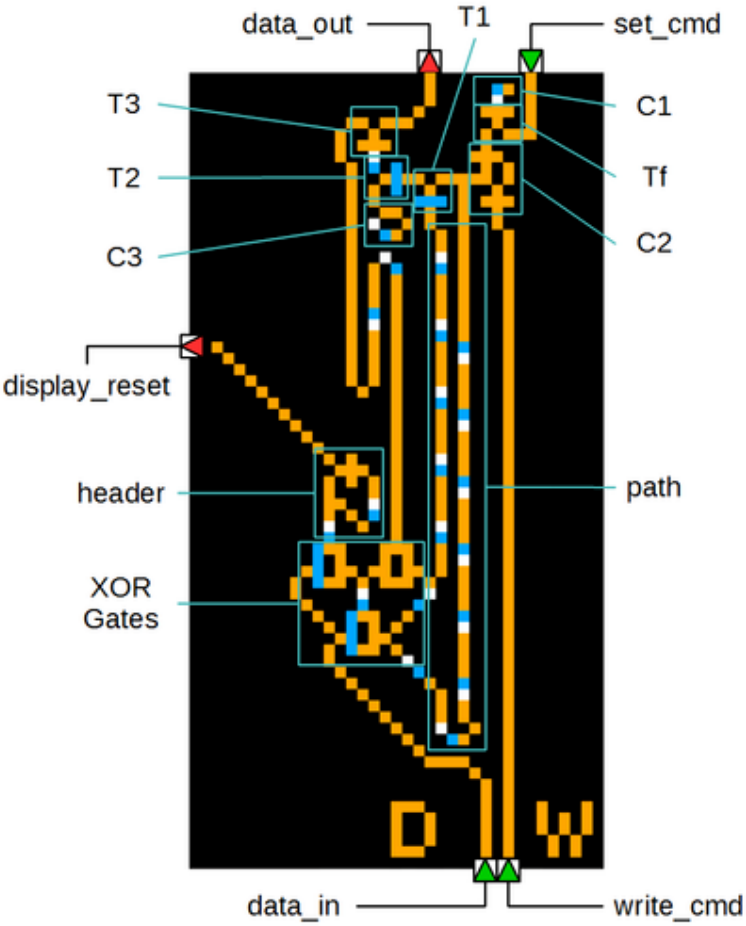
It has 2 outputs:

- Digit display reset that will make all 7 segments displays display zero (picture top left)
- Data to load in adder of binary/BCD converter (picture top center)

Internal architecture

It is made up of :

- 3 microns clock always active (**C1**) + 1 transistor (**Tf**)
- 6 microns clock with set and reset (**C2**)
- 6 microns clock de 6 microns always active (**C3**)
- 3 transistors (**T1**,**T2**,**T3**)
- 3 XOR gates to implement a wire crossing
- 1 path that can contain 16 electrons in 6-microns
- 1 "header" for 6 microns burst that allow only first electron to go through



Operating

Bottom megaloop of Binary/BCD converter, bound to **Set input** of Data Latch, contains an electron that is the **Set command** and a list of aritmetic values in 6 microns.

These values should not generate **Set command** so they must be filtered
Filetering is done by the **3 microns clock C1** which drives a **transistor Tf** controlling if electron coming from Megaloop continue in Data latch or not.
Arithmetic values of megaloop are coded in 6 microns and are in phase which C1 meaning that each electron of arithmetic value reach **transistor Tf** input at the time electron generated by **clock C1** disable **transistor Tf** so megaloop electrons don't go throught transistor
Command electron is slightly out of phase compared to **clock C1** which allows it to go trough **Tf** and to generate **Set command**.

Set command trigger the introduction of an electron inside the loop of **clock C2** that will generate burt of electron flow with a 6 microns period.
The **Write command** empty this loop.
This command is made up of 16 electrons in 6 microns but only the first one has an effect, indeed the following electrons will empty an already empty loop.
The electron flow generated by **C2 clock** go along 2 different wires :

- A wire going to **transistor T1** input and then on control input of **transistor T2**
- A central wire that can contain 16 electrons in 6 microns and that go into control input of **transistor T1**

The central wire cross the **Data input** wire thanks to 3 XOR gates and supply too a header that will only let the first electron go to **Reset output** controlling digit display reset

When central wire is full than its electrons disable **T1 transistor** so electrons coming from **clock C2** cannot pass.
These electrons disable **T2 transistor** which prevents electrons coming from **clock C3** to reach **transistor T3** control input
Transistor T3 control transfer from **Data input** to **Data output**

- During normal operating:
  - **C2 clock** is active and central wire is full
  - **T1 transistor** is disabled so electrons from **C2** cannot reach control input of **T2 transistor**
  - **T2 transistor** is enabled so electrons from **C3** reach control input of **transistor T3**
  - **T3 transistor** is disable so **Data input** cannot go to **Data output**.

- When **Write command** arrive:
  - Loop of **clock C2** become empty. There are no more electrons sent to central wire
  - Central wire become empty
  - **T1** become enable but **C2** loop is empty so no electron go through **T1** so **T2** remains enabled
  - Transistor **T3** remains disabled so **Data input** cannot go to **Data output**.

- When **Set command** arrive:
  - An electron is introduced **C2** loop which restart to generate electrons

    every 6 microns
  - Central wire is filling
  - Until central wire is full **T1** remains enabled so the first 16 1er electrons coming from **C2** reach **T2** control input
  - **T2** is disabled by par 16 consecutive electrons so 16 electrons from **C3 clock** doesn't reach control input of **T3**
  - **T3** is enabled and let the 16 electrons go from **Data input** to **Data output**

Once central path is full operating come back to normal mode

Remark :In order Data Latch work properly the following timing constraints must been respected:

- **Write command** should be synchronised with **C2 loop** to empty it
- **Set command** should arrive with the good timing so that Data 16 electrons arrive on **T3** input when it is enabled

Binary/BCD converter

Kind reminder, BCD (https://en.wikipedia.org/wiki/Binary-coded_decimal) is a way to code decimal representation of binary numbers using 4 bits to represent each digit
By example, BCD representation of **125** is **0001 0010 0101**.

The Binary/BCD converter is one of the most complex part of Wireworld Computer, which is reflexed by the number of cells and the area it represents in the full design.

Inputs/Outputs

Binary/BCD converters has 1 input and 5 outputs
Its input receive binary data coded on 16 bits LSB first The 5 outputs are :

- a pair of wire for each digit display

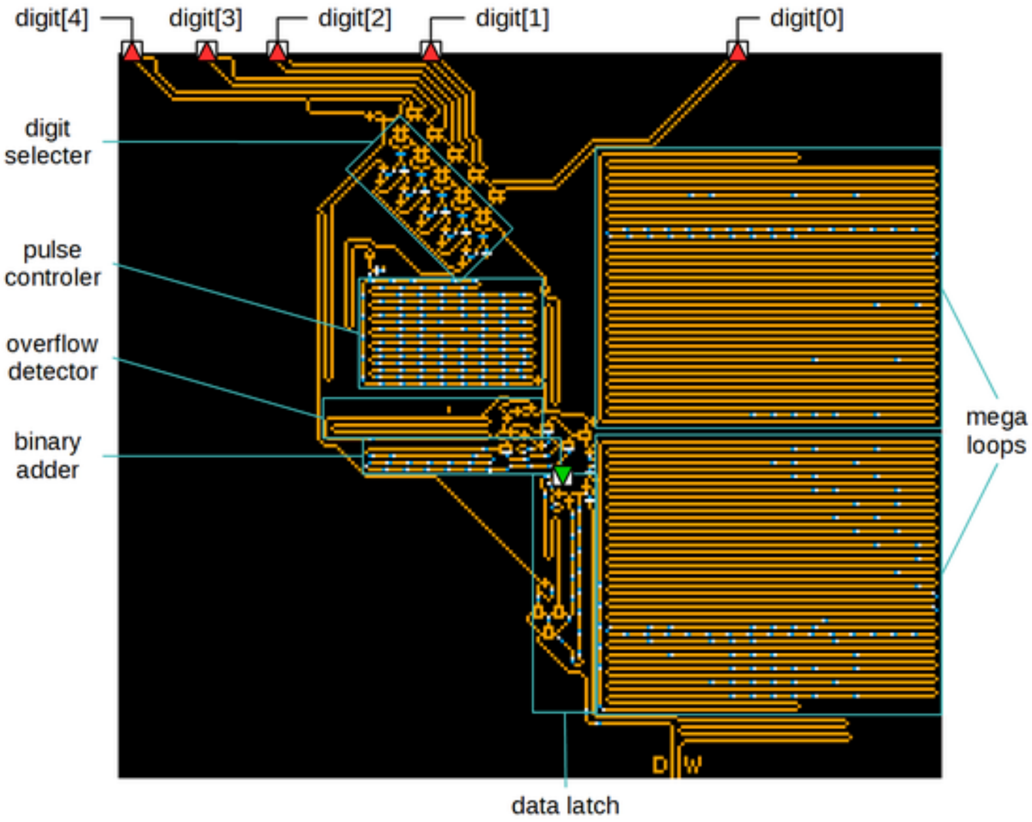
Internal architecture

Binary/BCD converter is composed of several parts:

- Binary adder
- Digit selecter
- Overflow detection loop associated to a pulse generator
- Pulse controler for digit display
- 2 mega loops

Binary adder

This is a standard serial binary adder whose output is bound on one of its



input. Loop period is 193 générations which allow him to operate on numbers coded on 32 bits in 6 microns.

Digit selecter

Inputs/Outputs

It has 2 inputs:

- **reset** input driven by **Digit display reset** output coming from **Data Latch**
- **Digit change** input driven by pulse controler for digit display
- **Data** input which receive pulses send by pulse generator and destinated to digit displays

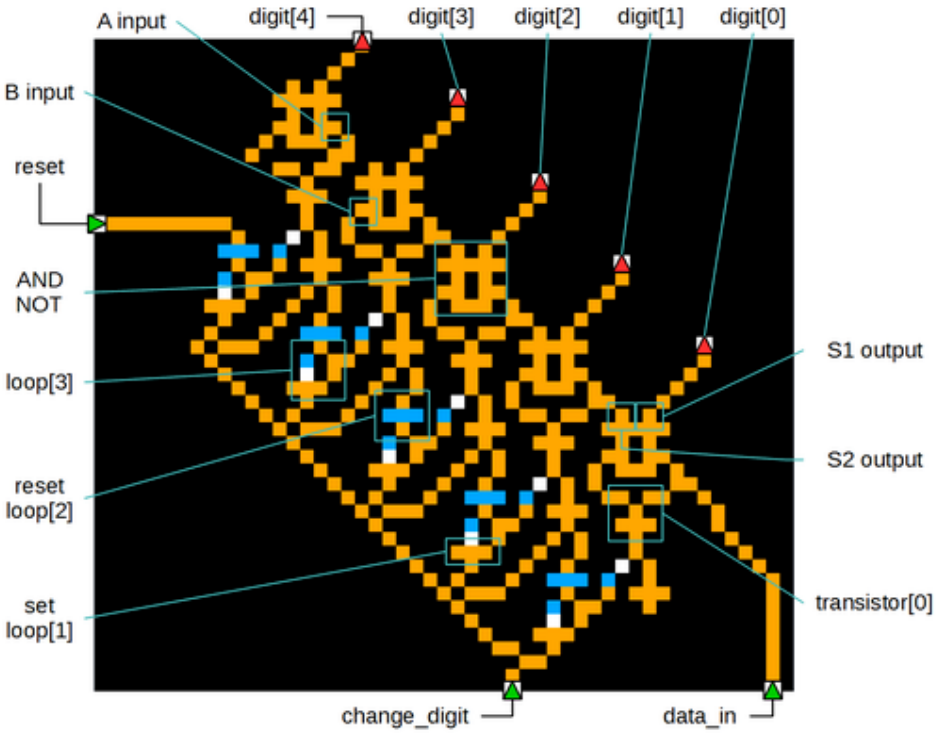
and 5 outputs:

- One output, made up a pair of wire, per digit display

Internal architecture

The way its control part operates is very similar to the way displays's controler is working

- It is made up of a series of 5 electron generation loops with a **set** and a **reset** that are bound each other



- **Digit_display[n]** is driven by **loop[n]**, **Digit_display[0]** control unit digit
- **loop[n+1]** drive **reset** of **loop[n]** so that if **loop[n+1]** is active then **reset** command of loop **boucle[n]** is intercepted
- Every **set** are driven by the **Digit change** input

The loop outputs are bound on transistors controlling replication of **B input** of double A AND NOT gates to their **A input**
These logic gates have 2 outputs **S1** and **S2** respectively bound to **control input** of digit display and to **B input** of next gate. The outputs implement the following equations:

- S1 = /A && B
- S2 = B && /A

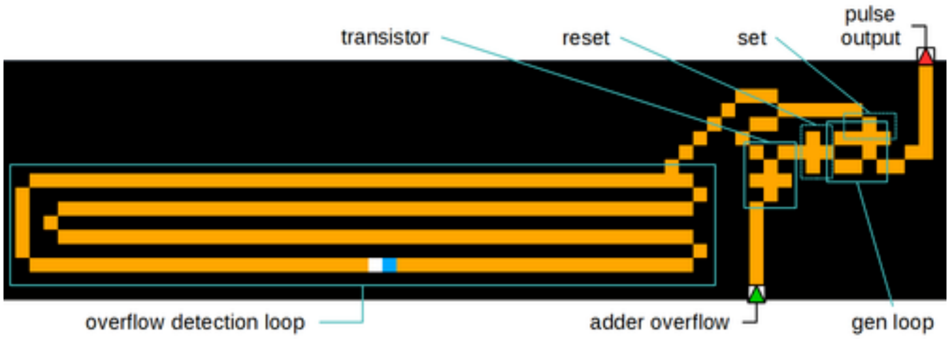
gate[0] has its **B input** bound to **pulse generator output** while **gates[n]** have their **B input** boudn to **S2 output** of **gate[n-1]**
By this way when **loop[n]** is active it locks **transistor[n]** so **gate[n]** has its **A input** at 0 so **S2 = B** making electrons coming from pulse generator go to **gate[n+1]**
On the other hand when **loop[n]** is empty then **transistor[n]** is enable so electron coming from pulse generator is replicated on **B input**.
Due to propagation delay **A** went down to 0 making **S1** change to 1 so electron is send to digit display

Operating

- By default all loops are active
- When **Data Latch** send its **reset** it makes **loop[4]** empty

  - When pulse controller send a **digit change** command then as loop correspondig to active digit display[n] is empty then **reset** of **loop[n-1]** is not intercepted so **loop[n-1]** while loop[n] is refilled
  - When **loop[0]** is reactivated then other loops are active too so we come back to default state

Overflow detection loop and pulse generator

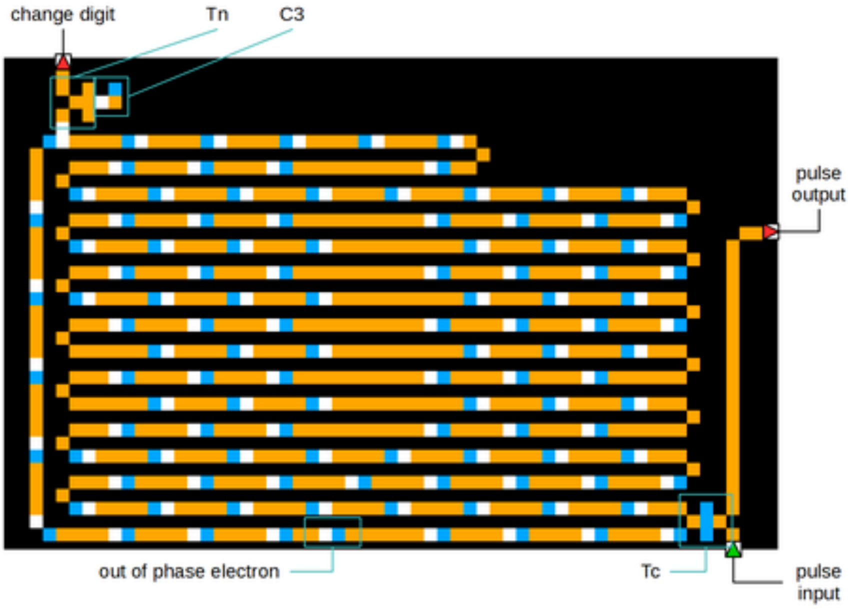


The period of this loop is 193, it stores one electron and is synchronised with the binary adder loop The electron control activation of **set** and **reset** of a period 6 loop (**gen loop**) in pulse generator:

- If there is no overflow in binary adder then **reset** is done just after **set** no pulse is generated
- If there is an overflow in adder than transistor is disabled which inhibit **reset** so pulses are generated by period 6 loop

Due to the length of the overflow detection loop the pulse generator produce 32 electrons pulses before loop 6 period reset is performed by detection loop

Pulse controller



It has 1 input:

- It receives pulses sent by pulse generator

and 2 outputs :

- Pulse output that will send only pulses needed to display the correct digit
- Digit change that will emit an electro to indicate to digit selector that it

should select an other digit

Pulse controller is mainly made up of a loop with period 769 which allow it to store 4 values coded on 32 bits in 6 microns
Values stored in pulse controller loop:

Index	Hexadecimal value	Binary value	Number of bits with value 0
0	0x7FFFFFFF	0111 1111 1111 1111 1111 1111 1111 1111	1
1	0x77777777	0111 0111 0111 0111 0111 0111 0111 0111	8
2	0x7F7F7F7F	0111 1111 0111 1111 0111 1111 0111 1111	4
3	0x7FFF7FFF	0111 1111 1111 1111 0111 1111 1111 1111	2

In addition to these values pulse controller loop contain an out-of-phase electron
On the top-left part of pulse controller there is a transistor **Tn** driven by a clock **C3** of périod e 3 microns which filters electrons coding values in the loop and prevent them to be sent on output wire driving digit change
On the other hand out-of-phase electron is not filtered and is able to go on output wire which allow to indicate to digit selecter to select an other digit
To work properply the pulse controler need to be well synchronised with pulse generator so that generated pulses for the current digit go through digit selecter before digit change command
Values stored in the loop are synchronised in such a way that they control **Tc** located at bottom-right wihich allow to control if pulses sent by pulse generator reach the output or not

- If value bit is 1 then **Tc** is locked and electron sent by pulse generator does not reach output.
- If value bit is 0 then **Tc** is unlocked and electron sent by pulse generator reach output.

Mega loops

Mega loops are huge size circular buffers containing some numeric values.
Their period is 3841 générations which allow them to store 640 bits which represents 20 values coded on 32 bits in 6 microns
Compared to electron move direction in the loop values are stored LSB first

Values stored in megaloop:

Index	Top Megaloop	Bottom Megaloop
0	0xFFFFFF80	0xFFFF63C0
1	0x0	0x4E20
2	0x0	0x2710
3	0x1890	0x7D0
4	0x0	0xFA0
5	0x0	0x7D0
6	0x0	0x3E8
7	0x3E8	0xC8
8	0x0	0x190
9	0x0	0xC8
10	0x0	0x64
11	0x44	0x14
12	0x0	0x28
13	0x0	0x14
14	0x0	0xA
15	0xA	0x2
16	0x0	0x4
17	0x0	0x2
18	0x0	0x1
19	0x1	0x1

Bottom megaloop contains too an out-of-phase electron compared to values electrons.
This is this particular electron that will not be filtered by transistor **Tf** of data latch and that will be the **Set** command
Electrons coding values are in phase with **Tf** lock and will be filtered

Operating

Principle

Base principle of Binary/BCD converter is overflow detection
The value to be displayed will be added with a series of predefined values stored in bottom megaloop which generate or not some overflows
When an overflow is detected then pulse generator is activated and generates pulses composed of 32 electrons
These electrons go on input of a transistor drivent by pulse controller
Depending on current value of pulse controler at this time only 1,2,4 ou 8 electrons will succeed to pass through the transistor and reach data input of digit selecter that will route them to the correct digit display

To display a digit d you need to decompose it in a sum of 1, 2 4, 8 which is equivalent to code it in binary
As we are in base ten to display a number n with need to decompose it in a sum of (1, 2 4, 8) * 10 exponent(digit_position)

Remark : The 32 electrons burst generated in case of overflow is re-injected in adder carry to disable it. Values contained in top megaloop are computed in a way that prevent carry to propagate between 2 decimal digits that's why we can remark that megaloop values are not null for indexes corresponding to stronger bit of each digit coded in BCD

Arithmetic

Operations performed in Binary/BCD converter are done on 32 bits, consequently the maximum representable value is 0xFFFFFFFF
If first bottom megaloop value is substracted to this maximum value the result is the following :

- 0xFFFFFFFF - 0xFFFF63C0 = 39999

meaning that every number >= 40000 added to 0xFFFF63C0 will generate an overflow. By repeating this process on other values of bottom megaloop we obtain the following array :

Index	Previous value	Current value	V[n-1] + V[n]	Substraction from Vmax	Overflow limit
0	0xFFFF63C0	0x4E20	0xFFFFB1E0	0xFFFFFFFF – 0xFFFFB1E0	19999
1	0xFFFFB1E0	0x2710	0xFFFFD8F0	0xFFFFFFFF – 0xFFFFD8F0	9999
2	0xFFFFD8F0	0x7D0	0xFFFFE0C0	0xFFFFFFFF – 0xFFFFE0C0	7999
3	0xFFFFE0C0	0xFA0	0xFFFFF060	0xFFFFFFFF – 0xFFFFF060	3999
4	0xFFFFF060	0x7D0	0xFFFFF830	0xFFFFFFFF – 0xFFFFF830	1999
5	0xFFFFF830	0x3E8	0xFFFFFC18	0xFFFFFFFF – 0xFFFFFC18	999
6	0xFFFFFC18	0xC8	0xFFFFFCE0	0xFFFFFFFF – 0xFFFFFCE0	799
7	0xFFFFFCE0	0x190	0xFFFFFE70	0xFFFFFFFF – 0xFFFFFE70	399
8	0xFFFFFE70	0xC8	0xFFFFFF38	0xFFFFFFFF – 0xFFFFFF38	199
9	0xFFFFFF38	0x64	0xFFFFFF9C	0xFFFFFFFF – 0xFFFFFF9C	99
10	0xFFFFFF9C	0x14	0xFFFFFFB0	0xFFFFFFFF – 0xFFFFFFB0	79
11	0xFFFFFFB0	0x28	0xFFFFFFD8	0xFFFFFFFF - 0xFFFFFFD8	39
12	0xFFFFFFD8	0x14	0xFFFFFFEC	0xFFFFFFFF – 0xFFFFFFEC	19
13	0xFFFFFFEC	0xA	0xFFFFFFF6	0xFFFFFFFF – 0xFFFFFFF6	9
14	0xFFFFFFF6	0x2	0xFFFFFFF8	0xFFFFFFFF – 0xFFFFFFF8	7
15	0xFFFFFFF8	0x4	0xFFFFFFFC	0xFFFFFFFF – 0xFFFFFFFC	3
16	0xFFFFFFFC	0x2	0xFFFFFFFE	0xFFFFFFFF – 0xFFFFFFFE	1
17	0xFFFFFFFE	0x1	0xFFFFFFFF		
18	0xFFFFFFFF	0x1	0		

It is obvious that overflow limits correspond to code with 1,2,4,8 and 10 powers

Simulation

C++ code below implement arithmetic principle of Binary/BCD converter and display internal states to illustate its operating

```
#include <iostream>
#include <stdint.h>
#include <stdlib.h>
#include <iomanip>

using namespace std;

int main(int argc, char ** argv)
{
  if(argc != 2)
    {
      cout << "Usage is binary2bcd <number>" << endl ;
      exit(-1);
    }
  uint64_t l_number = strtoll(argv[1],NULL,0);
  cout << "Input number is " << l_number << endl ;

  uint32_t l_bottom_loop[] = {
    0xFFFF63C0,
    0x4E20,
    0x2710,
    0x7D0,
    0xFA0,
    0x7D0,
    0x3E8,
    0xC8,
    0x190,
    0xC8,
    0x64,
    0x14,
    0x28,
    0x14,
    0xA,
    0x2,
    0x4,
    0x2,
    0x1,
    0x1
  };

  uint32_t l_top_loop[] = {
    0xFFFFFF80,
    0x0,
    0x0,
    0x1890,
    0x0,
    0x0,
    0x0,
    0x3E8,
    0x0,
    0x0,
    0x0,
    0x44,
    0x0,
    0x0,
    0x0,
    0xA,
    0x0,
    0x0,
    0x0,
    0x1,
  };

  uint64_t l_adder_content = l_number;
  uint64_t l_adder_full = 0xFFFFFFFF;
  uint32_t l_display[5] = {0,0,0,0,0};
  uint32_t l_display_index = 0;
  uint32_t l_power_index = 2;
```

```
  uint32_t l_carry = 0;

  //std::cout << "carry : 0x" << setw(8) << setfill('0') << hex << l_carry << dec << endl ;
  for(uint32_t l_index = 0; l_index < 20 ; ++l_index)
    {
      if(((l_index + 1) % 4) == 0)
        {
          std::cout << "-------------------------------------------------------";
          std::cout << "-------------------------------------------------------";
          std::cout << "-------------------------------------------------------";
          std::cout << "--------------------------------------------------" << std::endl ;
        }
      std::cout << "Step[" << setfill(' ') << setw(2) << l_index << "]: ";
      cout << "carry : 0x" << setw(8) << setfill('0') << hex << l_carry << dec << " & ~(" ;
      cout << "Top_loop[" << setfill(' ') << setw(2) << l_index << "] : 0x" << setw(8) << setfill('0'
      l_carry = l_carry & ( ~ (l_top_loop[l_index]));
      cout << "Adjusted carry 0x" << setw(8) << setfill('0') << hex << l_carry << dec << " ^ " ;
      cout << "Bot_loop[" << setfill(' ') << setw(2) << l_index << "] : 0x" << setw(8) << setfill('0'
      uint32_t l_to_add = l_carry ^ l_bottom_loop[l_index];
      cout << "To add 0x" << setw(8) << setfill('0') << hex << l_to_add <<dec << " + " ;
      cout << "Adder content : " << setw(8) << setfill('0') << hex << l_adder_content << dec << " | "
      l_adder_content +=  l_to_add;
      cout << "==> Adder content : " << setw(8) << setfill('0') << hex << l_adder_content << dec ;
      if(l_adder_content > l_adder_full)
        {
          cout << "\tOverflow !" ;
          l_adder_content = (l_adder_content & 0xFFFFFFFF) + 1;
          l_carry = 0xFFFFFFFF;

          // Part implemented by Pulse controler
          l_display[l_display_index]+= 1 << l_power_index;
        }
      else
        {
          l_carry = 0;
        }
      // Part implemented by Pulse controler
      l_power_index = (l_power_index > 0 ? l_power_index - 1 : 3);
      // Part implemented by Pulse controler and Digit display controler
      if(l_power_index == 3)
        {
          ++l_display_index;
        }
      cout << endl;
    }

  // Display Results
  for(uint32_t l_index = 0 ; l_index < 5; ++l_index)
    {
      cout << "|" << l_display[l_index] ;
    }
  cout << "|" << endl ;
}
```

To compile it use the following command

```
g++ -Wall -ansi -pedantic -g -std=c++11 -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -D__STDC_CONSTAN
```

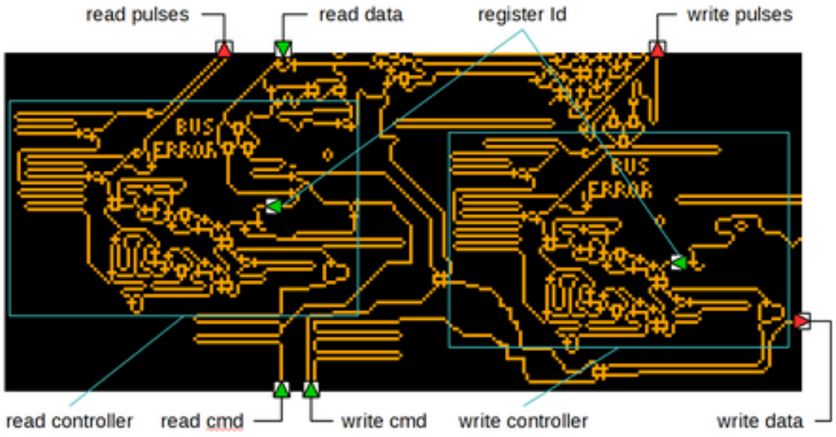
Here is an execution example:

```
$ ./binary2bcd.exe 0x100
Input number is 256
Step[ 0]: carry : 0x00000000 & ~(Top_loop[ 0] : 0xffffff80) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 1]: carry : 0x00000000 & ~(Top_loop[ 1] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 2]: carry : 0x00000000 & ~(Top_loop[ 2] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
-----------------------------------------------------------------------------------------------------
Step[ 3]: carry : 0x00000000 & ~(Top_loop[ 3] : 0x00001890) => Adjusted carry 0x00000000 ^ Bot_loop[
```

```
Step[ 4]: carry : 0x00000000 & ~(Top_loop[ 4] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 5]: carry : 0x00000000 & ~(Top_loop[ 5] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 6]: carry : 0x00000000 & ~(Top_loop[ 6] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
-----------------------------------------------------------------------------------------------------
Step[ 7]: carry : 0x00000000 & ~(Top_loop[ 7] : 0x000003e8) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 8]: carry : 0x00000000 & ~(Top_loop[ 8] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 9]: carry : 0x00000000 & ~(Top_loop[ 9] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[10]: carry : 0xffffffff & ~(Top_loop[10] : 0x00000000) => Adjusted carry 0xffffffff ^ Bot_loop[1
-----------------------------------------------------------------------------------------------------
Step[11]: carry : 0x00000000 & ~(Top_loop[11] : 0x00000044) => Adjusted carry 0x00000000 ^ Bot_loop[1
Step[12]: carry : 0x00000000 & ~(Top_loop[12] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[1
Step[13]: carry : 0xffffffff & ~(Top_loop[13] : 0x00000000) => Adjusted carry 0xffffffff ^ Bot_loop[1
Step[14]: carry : 0x00000000 & ~(Top_loop[14] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[1
-----------------------------------------------------------------------------------------------------
Step[15]: carry : 0xffffffff & ~(Top_loop[15] : 0x0000000a) => Adjusted carry 0xfffffff5 ^ Bot_loop[1
Step[16]: carry : 0x00000000 & ~(Top_loop[16] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[1
Step[17]: carry : 0xffffffff & ~(Top_loop[17] : 0x00000000) => Adjusted carry 0xffffffff ^ Bot_loop[1
Step[18]: carry : 0xffffffff & ~(Top_loop[18] : 0x00000000) => Adjusted carry 0xffffffff ^ Bot_loop[1
-----------------------------------------------------------------------------------------------------
Step[19]: carry : 0x00000000 & ~(Top_loop[19] : 0x00000001) => Adjusted carry 0x00000000 ^ Bot_loop[1
|0|0|2|5|6|
```

The **Overflow** represent bits with 1 value in BCD code with first bit being the MSB

Registers access controller

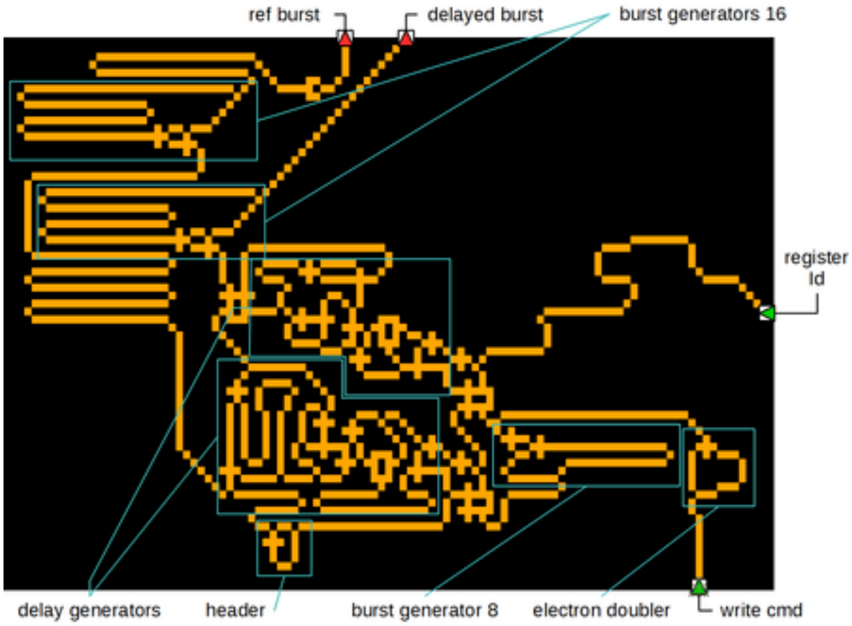


Wireworld Computer contains 64 registers 16 bits width, registers access controllers allow to select register to write in or to read from
There is a write access controller and a read access controller
Register selection is done by sending two burst of 16 electrons in 6 microns, one burst going in direction of register bank top and one going in direction of register bank bottom.
Register located at point where the tow burst will met up become reachable.
The falling burst is emitted with a fix period whereas the rising burst is emitted with a controllable delay
Controlling this delay allow to control the location where the 2 bursts will met up and consequently the accessed register.
The register access controller generate the rising burst with a delay depending on Id of registers that needed to be accessed.

Inputs/Outputs

Register access controllers have 2 inputs and 2 outputs. The 2 inputs are the following:

- Access command that will activate controller
- Data containing Id of register we want to access to



The outputs are the following:

- Burst of 16 electrons in 6 microns generated with a fixed delay after access command
- Burst of 16 electrons in 6 microns generated with a variable delay after access command depending on register Id

Internal architecture

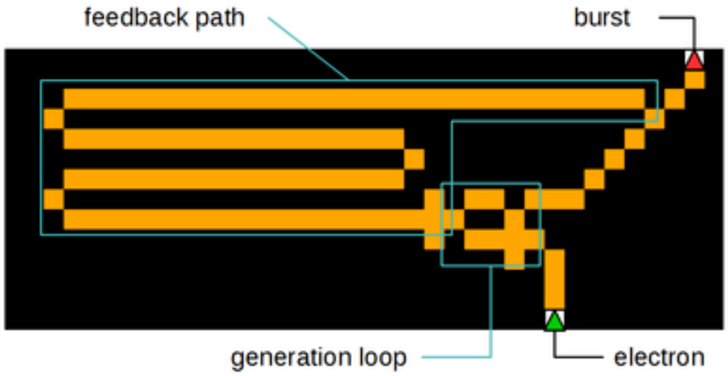
Each access controller is made up of the following elements:

- 2 burst generators of 16 electrons 6 microns : **GSup** and **GSdown'**
- 1 burst generator of 8 electrons 6 microns
- 1 electron doubler in 6 microns
- 2 delay generators in 6 microns
- 1 header that let pass only first electron of a burst

Burst generator of n electrons in 6 microns

They receive an electron as input and generate electron burst as output. Operating principle is always the same :

- An electron activate a loop whose period is 6 microns
- Electrons generated by the loop go to the output and a feedback wire that control the reste of the period 6 loop.



The length of feedback wire determine how many electrons will be emitted before the first one make the loop empty.

Electron doubler in 6 microns

Electron coming at input is sent on 2 wires going on a OR gate
One wire is longer by 6 micron compared to the other which make that the output will receive one electron + one electron 6 generation later

Described using algorithm we obtain the following code for **n=8**:

```
// this code start to be executed at activation
bool B = true;
bool Bc[8]
do
{
  for(int I = 0 ; i < 8 ; ++i)
  {
    bool output_bit = B ^ Bc[I];
    if(Bc[I] == true)
    {
      B = false;
    }
    B[I] = output_bit
  }
} while(!B)
// Send electron to output
```

If we apply this code to a numeric value like **0011011** (LSB first) so **108**

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Bc[Index] in input	0	0	1	1	0	1	1
B value	1	1	1	0	0	0	0
Bc[Index] in output	1	1	0	1	0	1	1

So an output value of 107 what was expected

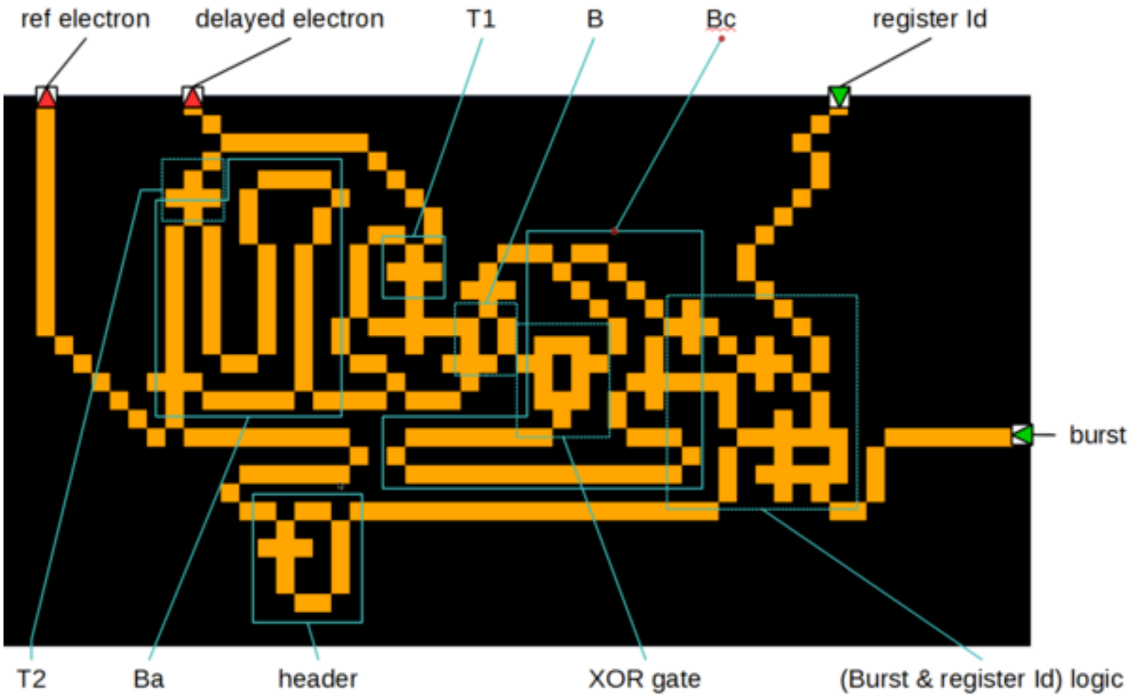
Delay generator in 6 microns

They are chacterized by their period. Generated delay will be a multiple of their **période** which is itself a multiple of 6 microns.
n is the factor (**période** / 6)
In input they receive:

- a value **V** coded on n bits
- a burst of n electrons considered as a value Vbis with n bits at 1

They generate the following outputs :

- a reference electron at time **t**
- an electron at time **t`** so thaht **t` = t + constant + V * periode**



Core of delay generator is a substracter working on values coded on **n** bits.
It is composed of :

- 2 loops: **Ba** the activation loop and **Bc** the computation loop. They have the same **périod** than generator and can store **n** bits
- a **XOR** gate
- 2 transistors **T1** and **T2**
- A loop **B** of period 6 controllable (set/reset) which can store one bit

Activation electron is inserted in **Ba** which drive set of loop **B**.
Result of **V AND Vbis** is inserted in computation loop when delay generator is activated.
Bc go through **XOR** gate whose other input is the value contained in **B**.
B reset is directly bound to **Bc** so the first non zero bit in computation loop empty **B**
Ba is bound too to a wire in direction of output with a duplicator to transistor **T1** which make it cancel itself unless **B** contains an electron in which case the duplicated activated electron doesn`t reach the input inhibiting transitor **T1**.
When substracter contains value zero, there are nore more bits at 1 to clean B so activation electron will reach the output and clean **Ba** via transistor **T2**.
Output is generated by the substracter underflow

Operating principle

We saw that register controler contains 2 delay generators:

- One with a period 48 and so based on a 8 bits substracter: **Gd48**
- One with a period 12 and so based on a 2 bits substracter: **Gd12**

Both generators are chained so that output electron from period 48 generator activate period 12 generator
Command electron from register controler go inside electron doubler, the 2 generated electrons are sent:

- on **Vbis** input of **Gd12** at the same time the 2 LSB electrons of **register Id** reach its **V** input
- on input of 8 electrons burst generator

The 8 electron burst is sent on **Vbis** input of **Gd48** at the same time that the 6 MSB electrons of **Register Id** reach **V** input
The first of the 8 electrons is sent too on **GSdown** command while output of **Gd12** command **GSsup**

The use of 2 generators delays with different periods allow to generate huge delays (**Gd48**) with a thin granularity of 12 generations (**Gd12**) which finally give a delay **d** between activation of **GSup** and **GSdown** defined as the following:

```
d = 48 * (RegisterId[7:2] >> 2) + 12 * RegisterId[1:0]
```

ignoring propagation constants

Control Unit

Operating principle

This the unit that drive the execution of Wireworld Computer by managing instruction cycle (https://en.wikipedia.org/wiki/Instruction_cycle):

- *Fetch(1/2)* : Read **Program Counter** from **Register[63]**
- *Fetch(2/2)* : Read instruction **MOV Rs Rt** contained in **Register[Program Counter]**
- *Decode(1/2)* : copy value **Rs** on **Register Id** input of register read access controller
- *Decode(2/2)* : copy value **Rt** on **Register Id** input of register write access controller
- *Execute(1/2)* : Read value **V** contained in **Register[Rs]**
- *Execute(2/2)* : Write value **V** contained in **Register[Rt]**
- Incrementation of **Program Counter** to go to next instruction

To improve execution speed of Wireworld Computer all *Fetch/Decode/Execute* steps are done in parallel
By example step 1 of Fetch is done via a dedicated wire which allow to perform *Execute(1/2)* at the same time
Finally it leads to do 2 read operations for one write Operation
This is visible in the design by an output wire after 5th frewuency clock divier that command register read access controler and an output wire after register write access controler

The following array summarise ordering of the different operations

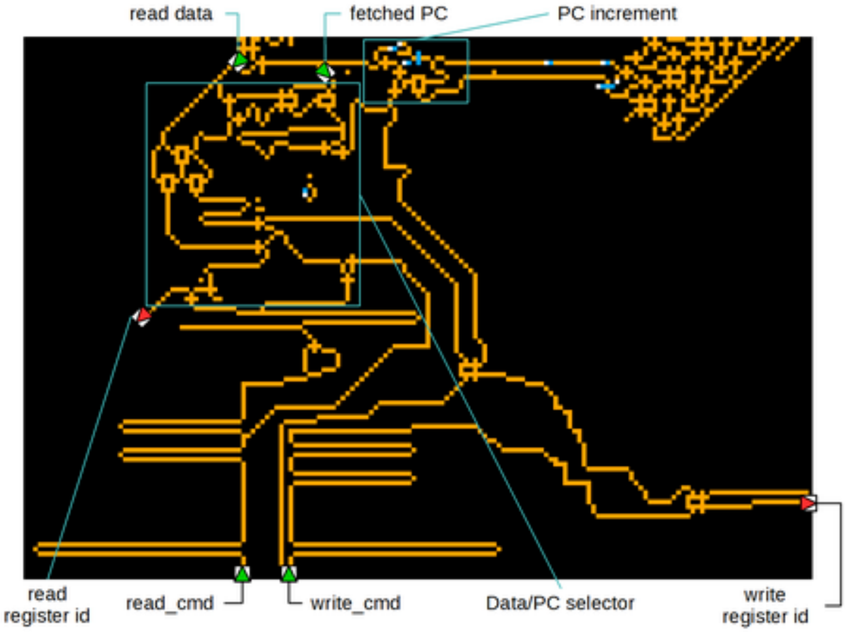
Operation 1	Operation 2	Operation 3	Read Register Id	Write Register Id	Data read	Data to write
Fetch(1/2)	Execute(1/2)	Decode(2/2)	PC value	Result from previous Fetch(2/2) = Rt	V = Content Register Rs	
Fetch(2/2)	Execute(2/2)	Decode(1/2)	Rs		Instruction pointed by PC	V

Implementation

Inputs/Outputs

The control unit has the following inputs:

- Data **D** coming from register bank
- **Program Counter** value or **PC**
- Read command **Cr**
- Double electron from register read access controller
- Write command/**PC** read command



The control unit has the 2 following outputs:

- Read register Id
- Write Register Id/ Data to write

Internal architecture

Unit control is made up of the following elements:

- Incrementer
- XOR gate
- Double A AND NOT B **L1** gate

- OR gate
- Transistor **Tpc**
- Triple XOR gate
- Transistor **Tr**
- Transistor **Tw**

Operating

Read command electron start a burst generator whose stop is driven by electrons generated by register read access controller electron doubler which generates a 16 electrons burst
Burst electrons go to the input of transistors **Tr** and **Tw**.
In case transistors are not locked burst electrons are sent:

- By **Tr** on Register Id input of register read access controler
- By **Tw** on Register Id input of register write access controler and on Data wire of register bank write side

The path of Data bus Data is very long so that Data arrives at register input at the it is activated by rising and falling burst of register write access as defined by *Decode(2/2)*'.
During *Decode(1/2)* value is on Data bus is not an instruction so timings are computed so that it arrives on Register Id input between 2 write command so it has no effects.
It was previously said that **Tr** and **Tw** are unlocked only when they are not driven so when sending a burst on input it means that **Output = NOT Input control**
During read operation on registers the output is NOT of register content and this result arrive on input control of **Tw** so the output of **Tw** is the value contained in register.
Read operation can be a register read or **PC** read in case of instruction fetch so there is some logics to manage both cases.
Write command electron is sent too to **PC** register
It starts PC incrementer and a burst generator that produce 8 electrons

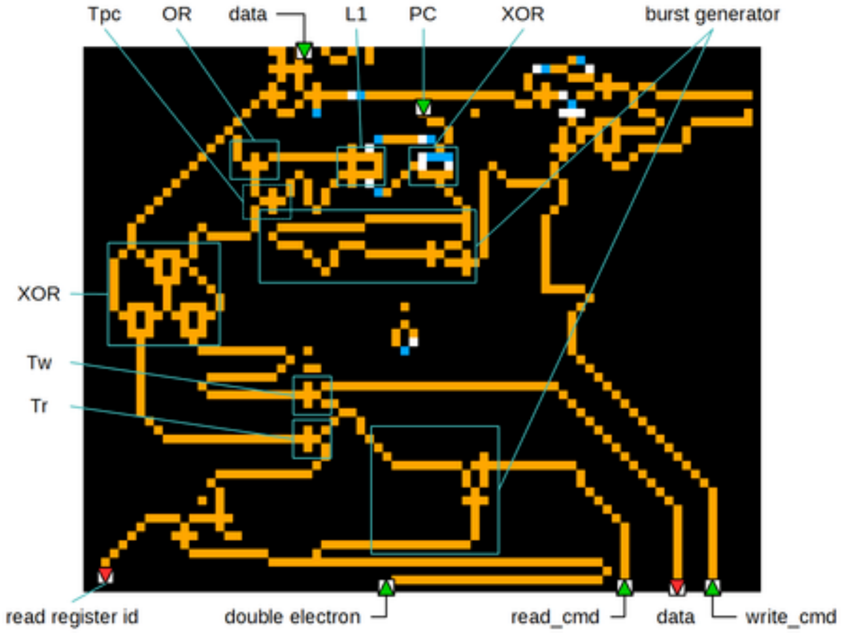
PC/Data register selecter

Generated burst is sent to a **XOR** gate whose other input receive **PC** value.
XOR gate output so produce **NOT(PC**) which arrive on double **A AND NOT B** gate **L1** whose other input receive **PC**. By this way both inputs never receive 1 at the same time so the gate behave as a wire cross and bottom output produce **PC** wherase top output produce **NOT(PC)**.
NOT(PC) output is bound on the input of **OR** gate whose other input receive data coming from registers.
OR gate output is bound on transistor **Tpc** driven by **PC** value.

In case of simple register read **XOR** gate has input at 0 so its output is **PC**.
L1 gate perform **AND NOT** of **PC** and **PC** so output remains 0 so **OR** gate output is value coming from registers, as **Tpc** is not driven value go in

direction of **Tr** transistor through wire crossing implemented by the triple XOR gate.

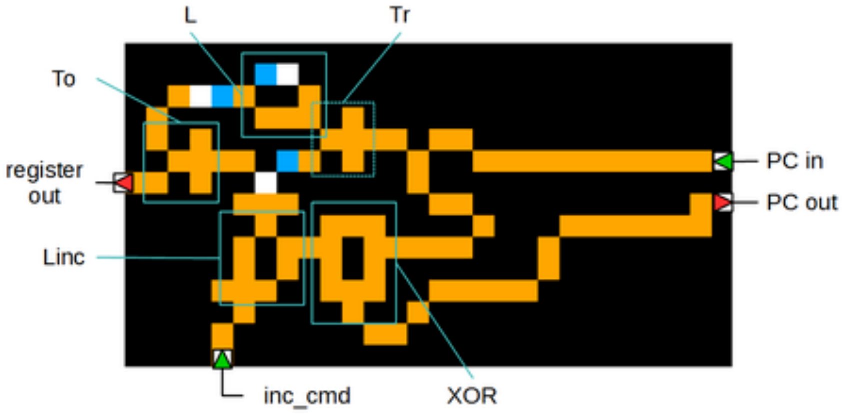
In case of **PC** fetch **OR** gate receive in input value coming from registers and **NOT(PC)**. **Tpc** transistor is driven by **PC** do transistor output is **((NOT(DATA)) OR (NOT(PC))) & NOT(PC)**.



- When **PC** bit is 1 output is zero
- When **PC** bit is zero output is **NOT(Data) OR NOT(PC)**, as **PC** bit is zero **OR** output is 1. **PC** drives **Tpc** so transitor output is **NOT(PC)**

PC incrementer

PC value go throught incrementer **XOR** gate whose other input come from a period 6 loop **Linc**. This loop is loaded by PC read command electron and the reset is driven by the output of a transistor **Tr** driven by **PC** bits and whose input is supplied by a period 6 clock **L**



If PC bit is at 1 then transistor **Tr** is locked so electron coming from clock **L** doesn't reset loop **Linc** and does not lock transistor **To** so electron emitted by **L** reach register output which allows to have PC value at register output despite incrementer logic
If PC bit is at 0 then transistor **Tr** is unlocked so electron coming from clock **L** reset loop **Linc** and lock transistor **To** so electron emitted by **L** does not reach register output which allows to have PC value at register output despite incrementer logic. Due to propagation delay **Linc** is clear only after value it contains has been XORed with PC bit so first PC bit with zero value is set to 1
From arithmetic point of view while PC bits are at 1 **Linc** keep value 1 so PC

bits are XORed to zero, the first zero bit will be set to 1 while **Linc** will be empty so next bits will remain unchanged. This is an increment.

Described in algorithmic way we obtain the following code for **n=4** wit n the number of bits coding PC value:

```
// this code start to be executed when reading PC
bool B = true;
// PC value
bool PC_in[8]
// PC value
bool PC_out[8]

for(int I = 0 ; i < 8 ; ++i)
{
    PC_out[I] = B ^PC_in[I];
    if(!PC_in[I])
    {
       B = false;
    }
}
```

If we apply this code to a numeric value like **b00000000** (LSB first) so **0**

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
PC_in[Index]	0	0	0	0	0	0	0
B value	1	0	0	0	0	0	0
PC_out[Index]	1	0	0	0	0	0	0

So an output value of **b10000000** (LSB first) so **1** what was expected

If we apply this code to a numeric value like **11100000** (LSB first) so **7**

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
PC_in[Index]	1	1	1	0	0	0	0
B value	1	1	1	1	0	0	0
PC_out[Index]	0	0	0	1	0	0	0

So an output value of **00010000** (LSB first) so **8** what was expected

Registers

There are 64 and are composed of:

- a loop of periode 96 used to store the data, a 16 bits word in 6 microns
- logic to write in register
- logic to read from register

Registers are serial registers meaning that they should written and read bit by bit

Register write

Operating principle

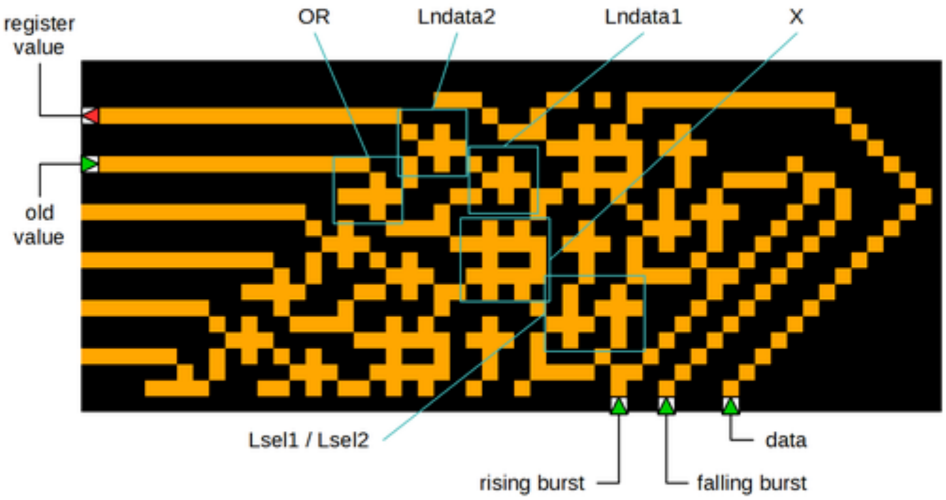
To write in a register it is needed to generate a write command and to present a data on register input
In wireworld computer data arrived from wire bound at register bank top right and will fall along register bank through a logic gate stream
Write command is generated at level of a single register and should be synchronised with the data so that they both arrive at the same time on register and data should be synchronised too with register loop so that first bit of data to be written is introduced in register at the location of the first bit stored in the loop
Write command will stop data propagation along bank register and will "deflect" it inside register
Write command is generated by register write access controller via 2 electron bursts

- One that raise along a wire located on the right of write logic mechanism
- One that fall between previous wire and data wire

So cells located in input of each register should implemnt the following features:

- Propagation of data to be written to bottom of register bank
- Propagation of rising and falling bursts along register bank
- Sending data to register selected by collision of rising and falling burst

Implementation



The write logic is composed of the following gates:

- 2 **AND NOT** gates called **Lsel1 Lsel2**
- 1 double **AND NOT** gate called **X**
- 2 **AND NOT** gates called '**Lndata1 Lndata2**
- 1 **OR** gate

Detection of collision is performed thanks to logic gates **Lsel1** et **Lsel2** which implement the following function:

- **f = (rising_burst AND NOT (rising_burst AND NOT falling_burst))**

By this way **f** value is 1 only when rising_burst and falling_burst are at 1 the same time which is the case during their collision

Gate **X** receive as input data to write and **f**, its bottom output is bound to top input of next register's **X** gate and income timings are computed in such a way that

- If **f** is 0 data to be writtend is propagated to the bottom of register bank
- If **f** is 1 data is not propagated to the bottom and top output of **X** is **f'=f**

Value contained in register go through **OR** gate via **A** input and **f`** is bound on **B** input so when register is selected previous value is replaced by a 16 electron burst, when register is not selected value remains the same.
During data propagation to bottom of register bank data arrive on **B** input of logic gate **Lndata1** whose **A** input receive **f'**.
Output of this gate is sent to **B** input of **Lndata2** gate whose **A** input receive **f`** and whose output go inside register.

- If **f'** is 0 then **Lndata1** receive register value on **A** input and 0 on **B** iput so register value remains the same
- If **f'** is 1 then **NOT(NOT(Data))** so **Data** arrive on register so data is loaded inside register

Register read

Operating principle

To read a register it is needed to generate a read command and to ge the data from register output
In wireworld computer data flows ou register on wire located at left of register bank and will fall along a stream of logic gates
Read command is generated at level of a signle register. She must be synchronised with register loop so that first bit of value contained in register be at register output at the same time than the first electron of read command
Read command is generated by register read access controler via 2 electron bursts

- One that raise along a wire located on the left of read logic
- One that fall along a wire located onthe left of previous wire

Cells located in output of each register should implement the following features:

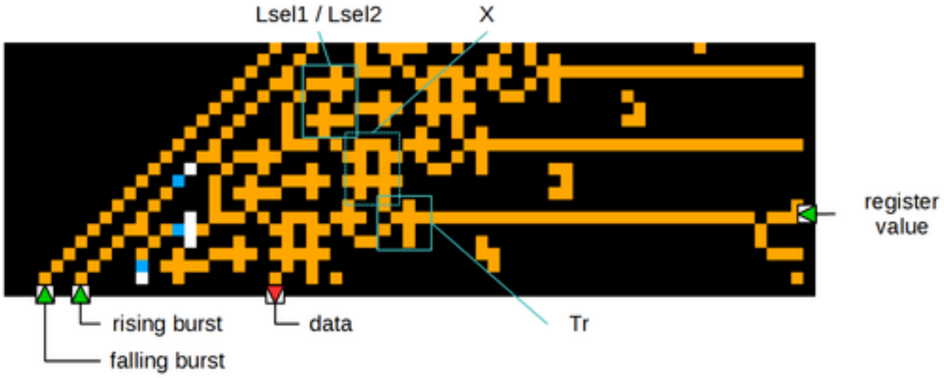
- Propagation of read data to the bottom of register bank
- Propagation of rising and falling burst along register bank
- Read data of register selected by collision of falling and rising electron

bursts

Implementation

Read logic is composed of following gates :

- 2 **AND NOT** gates called **Lsel1 Lsel2**
- A double **AND NOT** gate called **X**
- 1 transistor **Tr**



Read of register data is done thanks **Lsel1** and **Lsel2** logic gates which implement the following function:

- **f = (rising_burst AND NOT (rising_burst AND NOT falling_burst))**

By this way **f** is 1 only where there is the collision of the 2 bursts.

X receive **f** on left input and data read from upper register on right input, its left output is bound on right input of below register gate **X** , its right output is bound to input of transistor **Tr** and income timings are computed so that:

- If **f** is 0 data read is propagated to bottom of register bank
- If **f** is 1 **X** right output is **f'=f**

f' is send to input of transistor **Tr** driven by value contained in register and its output is bound to right input of register below **X** gate
Transistor output is **f'** in case register data bit is 0, so we obtain **NOT(register data)** at transistor output

Special registers

They allow to perform other operations than simple read/write
Registers left/right SHIFT deal with leng of wire that ellectron follow to the register output to expose bit[1] or le bit[15] at the time bit[0] shoudl be exposed.
Registers NOT, AND NOT respectively use NOT, AND NOT gates to implement the feature

Adder register

It use a binary adder to compute sum of inputs
Remark : in case of overflow, RS flip-flop used to propagate carry will still be set when bit[0] of input registers will come back again inot adder inputs so result will be **R60 + R61 + 1**
This particularit is used in prime computation algorithm that work with 1 complement arithmetic. In this case **-1** is coded **0xFFFE** and **1** is coded **0x0001** which give **0xFFFF** when we sum them so **-0** in 1 complement arithmetic
To easily detect a null sum using conditional register program first execute and overflow operationlike by example **0xFFF6 + 0xFFFE** (-9 -1) which return 0xFFF5 (-10) with carry at 1 so when 10 is added the result is **0xA + 0xFFF5 + 1 = 0x0** which will be well detected by conditional register.

Conditional register

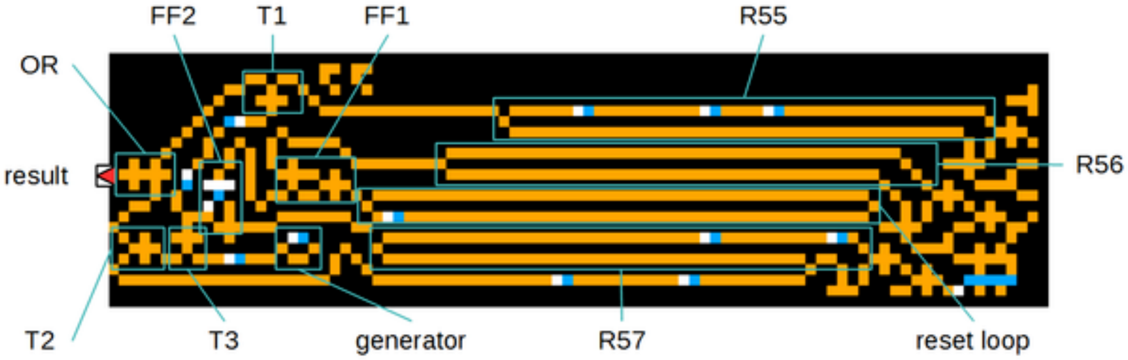
Read of Register R56 return R55 if R56 is not null else R57

Internal architecture

This feature is implemented using the following components:

- 2 RS Flip-Flops *FF1* and **FF2**
- 3 transistors **T1**,**T2**,**T3**
- An OR gate
- A reset loop whose period is the same than register storage loop
- An electron generator in 6 microns

Operating



Reset loop put flip-flop **FF1** to 0 and flip-flop **FF2** to 1

- Case R56 not null

Value stored in **R56** contains a bit ar 1 so **FF1** is set to 1 which set **FF2** to 0
As **FF2** is at 0 **T3** is unlocked so electrons emitted by generator lock **T2** so R57 value don't reach **OR** gate
FF2 is at 1 so **T1** is unlocked what let electrons stored in R55 reach **OR** gate

- Case R56 null

Value stored in **R56** does not contain bit at 1 so **FF1** is set at 0 and **FF2** remains at 1
As **FF2** is at 1 **T3** is locked so electrons emitted by generator don`t reach **T2**, consequently **T2** is unlocked so electrons stored in R57 reach **OR** gate
FF2 is at 1 so **T1** is locked so electrons stored in R55 don't reach **OR** gate

Register configuration

To make easier the execution of other software than the one provided with wireworld computer I wrote a generic configuration file for the desing that allow to define the content of each register at simulation startup via a configuration file.
By automatising generation if configuration file, corresponding to a software written in assembly, by the functionnal model of wireworld computer it become easy to execute them on the design

Wireworld computer

Functional model

Wireworld computer design is complex and execution of a single instruction take several hundred of generations
To te able to easily test and develop small programs for wireworld computer I developed a functional model in C++ language which reproduce execution pipeline and register behaviours
It generates too the configuration file corresponding to program to execute which will load registers with values necessary to execute program on Wireworld computer design.

Inputs/Outputs

Model receive the following parameters:

- Name of file containing program to execute written in assembly
- optional parameter **--detailled_display** to activate or not operating details of binary/BCD converter
- optional parameter **--instruction_delay** to define tempo duration (in ms) between each instruction
- optional parameter **--output_file** to define the name of configuration and generate it

```
Usage is :
        wireworld.exe [OPTIONS] <program_file>
OPTIONS : --<parameter_name>=<parameter_value>
        --detailled_display=...
        --instruction_delay=...
```

```
        --output_file=...
```

During its execution functional model display the following information:

- Cycle number
- PC value
- Instruction value
- Instruction mnemonic
- Source register => Read value => Destination register
- Value displayed in case of write in **R0** register.

Remark : functional model don't take in account delays needed to latch R0 value so in case of closer write functional model will indicate them as displayed whereas with real wireworld computer design first value would perhaps not had time to be be displayed before the second one be take in account.

```
***** Starting cycle 10*****
PC value :=> PC = 0xb
=> Instruction = 0x3b29
=> MOV R59, R41
R41 => 0x0 => R59
***** Starting cycle 11*****
PC value :=> PC = 0xc
=> Instruction = 0x30
=> MOV R0, R48
R48 => 0x0 => R0
-----------------------------------
** DISPLAY => 0
-----------------------------------
***** Starting cycle 12*****
PC value :=> PC = 0xd
=> Instruction = 0x3d3b
=> MOV R61, R59
R59 => 0x0 => R61
```

In case **detailled_display** has been activated the ouput will display details of internal operating of Binary/BCD converter

Assembly format

Assembly format used as input of functional simulator is very simple :

- Comments start with a `;` and end with line return
- There is one line per register with the following syntax

```
register_number : [<optional_label> :] (Value | Instruction )
```

By example:

```
; Register | Action on read                           | Action on write
;-----------------------------------------------------------------------------------
; R0       | Returns zero                             | Writes value to display module
; R1-R52   | Reads value from register                | Writes value to register
; R53      | Returns bitwise AND of R54 with NOT R53  | Writes value to register
; R54      | Returns bitwise AND of R53 with NOT R54  | Writes value to register
```

```
; R55      | Returns zero                          | Writes value to register
; R56      | Returns value in R55 if register R56 is  | Writes value to register
;          | non-zero, and the value in R57 otherwise |
; R57      | Returns zero                          | Writes value to register
; R58      | Returns R58 rotated right one place   | Writes value to register
; R59      | Returns R59 rotated left one place    | Writes value to register
; R60      | Reads value from register             | Writes value to register
; R61      | Returns sum of R60 and R61            | Writes value to register
; R62      | Reads NOT R62                         | Writes value to register
; R63      | Returns program counter value         | Causes branch to given target
;-------------------------------------------------------------------------------------
 0 :       UNUSED
 1 :       MOV R62, R42 ; Compute negative value of upper limit
 2 :       MOV R55, R47 ; Prepare branch if limit non reached
 3 :       MOV R57, R44 ; Prepare branch if limit reached
 4 :       MOV R61, R50 ; Load -1 in adder as second operand
 5 :       MOV R60, R62 ; negative (upper limit - 1) as first operand of adder
 6 :       MOV R60, R61 ; perform addition to set the Carry
 7 :       MOV R61, R41 ; current variable as second operand of adder
 8 :       MOV R56, R61 ; perform addition
 9 :       MOV R63, R56 ; Branch on addition result
10 :       MOV R59, R41 ; Prepare computation of 2 * V
11 :       MOV R0 , R48 ; Display square of variable
12 :       MOV R61, R59 ; Prepare computation of 2 * V + square(V) by setting 2 * V as second operand
13 :       MOV R60, R48 ; Prepare computation of 2 * V + square(V) by setting square(V) as first oper
14 :       MOV R61, R61 ; Compute addition of 2 * V + square(V) and set it as second operand of adder
15 :       MOV R60, R45 ; Preparing addition of increment by setting 1 as first operad of adder
16 :       MOV R48, R61 ; Compute the new square value and store it
17 :       MOV R61, R41 ; Prepare V + 1 by setting V as second operand of adder
18 :       MOV R63, R43 ; Preparing branch at the beginning of the loop
19 :       MOV R41, R61 ; Incrementing current variable
20 :       MOV R63, R44 ; Branching on end of the loop
21 :       MOV R0 , R46 ; End of loop
22 :       0x0000
23 :       0x0000
24 :       0x0000
25 :       0x0000
26 :       0x0000
27 :       0x0000
28 :       0x0000
29 :       0x0000
30 :       0x0000
31 :       0x0000
32 :       0x0000
33 :       0x0000
34 :       0x0000
35 :       0x0000
36 :       0x0000
37 :       0x0000
38 :       0x0000
39 :       0x0000
40 :       0x0000
41 : <V> : 0x0000       ; Initialisation running variable to 0
42 :       0x000a       ; Set upper limit - 1
43 :       0x0004       ; Branch value to restart the loop
44 :       0x0014       ; Branch value to end the loop
45 :       0x0001       ; Increment value
46 :       0xffff       ; Final value
47 :       0x000a       ; Branch value to continue the loop
48 : <SQ>: 0x0000       ; Current square value
49 : <DB>: 0x0000       ; Store double of current value
50 :       0xfffe       ;  -1
51 :       0x0000
52 :       0x0000
53 :       UNUSED
54 :       UNUSED
55 :       UNUSED
56 :       UNUSED
57 :       UNUSED
58 :       UNUSED
59 :       UNUSED
60 :       0x0000
61 :       UNUSED
62 :       UNUSED
```

```
63 : <PC>: 0x0001       ; Initial PC
```

Use

Functional model allowed me to test development of programs for wireworld computer.
By example a loop to display square of first ten integers. Interest of this program is to succeed to compute successive square despite the lack of multiplication.
It is based on the fact that **pow(n+1,2) = pow(n,2) + 2 * n + 1** which is 2 additions and one shift in case previous square value has been stored.
I wanted to use this principle to optimise the prime number computation program provided with wireworld computer assuming it is not necessary to search divisor greater than square_root(n) to determine if n is prime
As computing **square_root(n)** is not so easy I decided to start from minimum square and root: 1 and 1 and then to compute next square and root at each time new candidate is greater than current square.
The goal was to limit the number of time substraction loop is executed to simulate division of prime candidate **p** by divisor **q**
It is equivalent to divide the number of divisions of candidate **p** by 2 but increase code complexity:

- It is needed to perform a more complex test on p and q : comparison with square_root and square which imply substrction and result sign check
- It requires more test instruction with branch preparation it implies

Given the URISC (https://en.wikipedia.org/wiki/One_instruction_set_computer#urisc) architecture of wireworld computer if leads to an important code expansion which finally cancel the gain coming from reduce number of division and make alogrithm slightly slower than original one by using near the whole registers of processor.
The last point forced me to carrefully think the way to write the code and to correctly use register init values to succeed to make the wole program take place in registers

Conclusions

I'm still fascinated by this cellular automaton that allow to simulate such complex thinks like a small URISC processor with 64 registers of 16 bits depite its simple rules.
Wireworld Computer allowed me to discover architectures URISC (https://en.wikipedia.org/wiki/One_instruction_set_computer#urisc) and TTA (https://en.wikipedia.org/wiki/Transport_triggered_architecture) I was not aware of before and that I plan to reuse in my FPGA experimentations
The time spend to develop for Wireworld Computer make me also conscious of the limitations of this kind of architecture.
Reverse engineering of Wireworld Computer design increased my admiration

for those who designed it and I still find a kind of magic when seeing it in simulation despite I now understand its details

Récupérée de « https://www.logre.eu/mediawiki/index.php?title=Projet_Wireworld/en&oldid=11804 »

Catégories : Software | C++ | Projets

- Dernière modification de cette page le 22 janvier 2016 à 16:16.
-