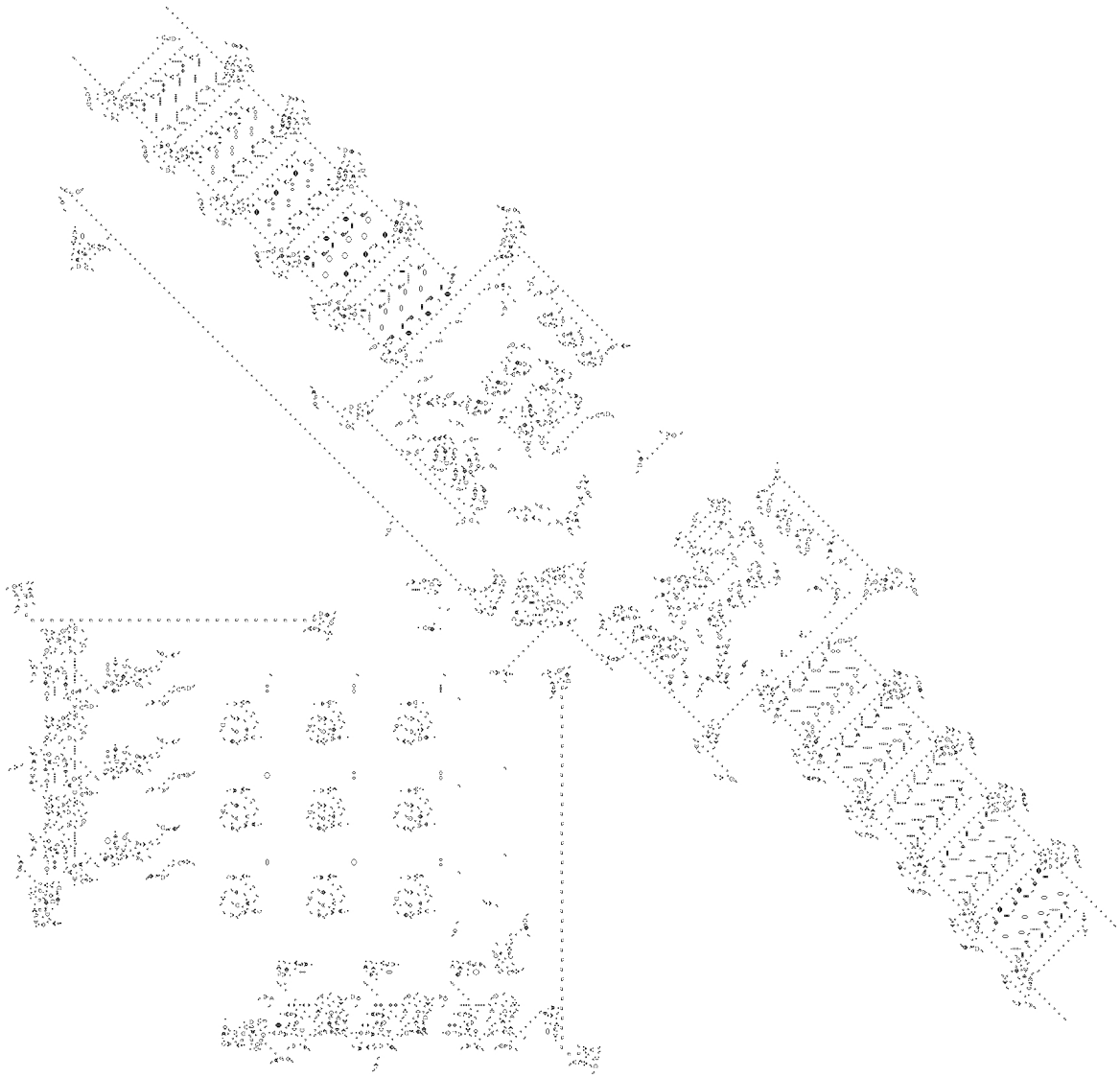
ERD modular eurorack series 2017 ERD/SIR plague reader

micro_research

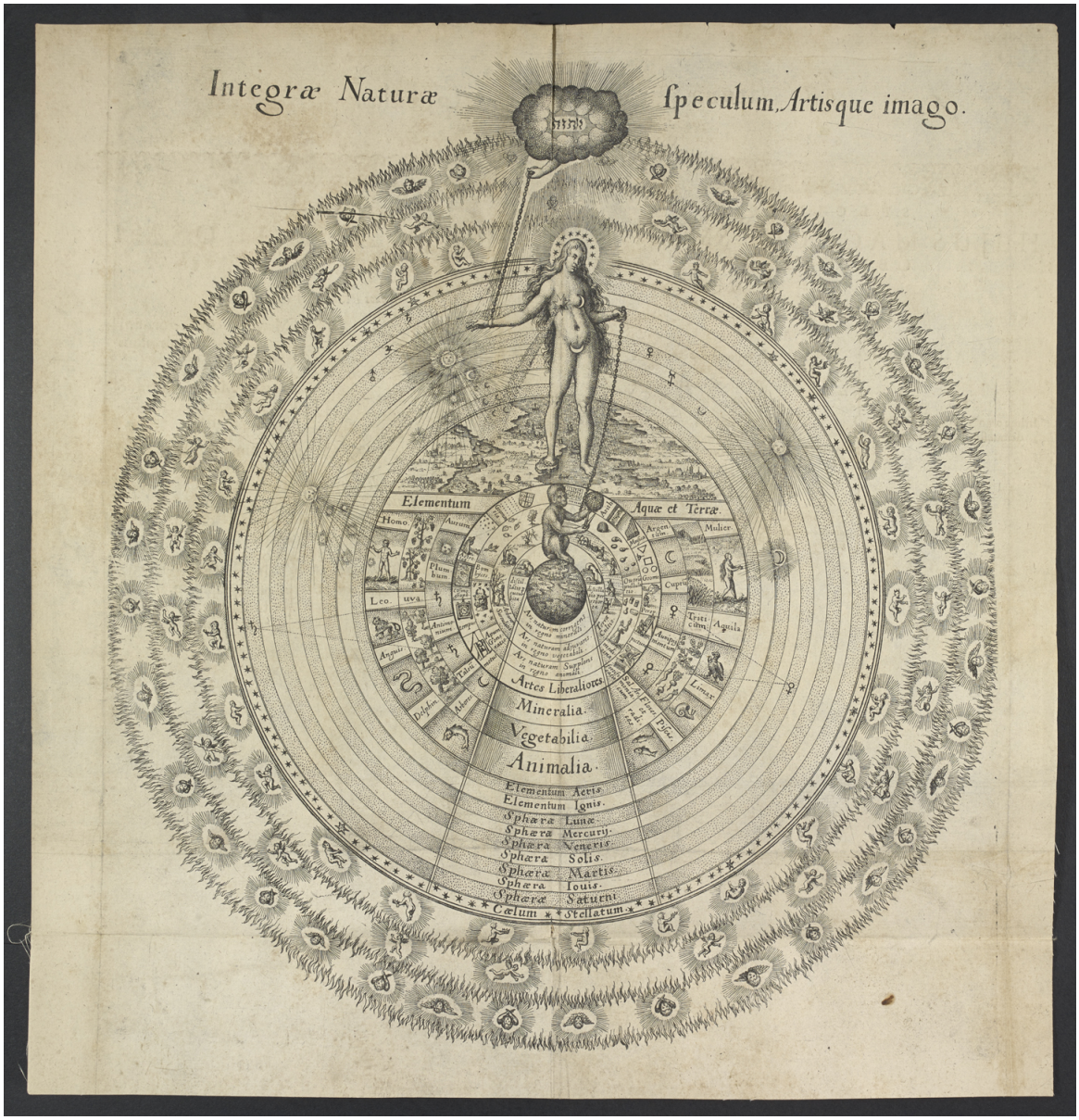
March 31, 2017



This reader expands on the plague CPU simulations and settings which are accessed from the top right knob and lower left CV input.

As the knob is turned clockwise the user accesses 16 plague simulations modes and 16 instruction sets for plague processor units running from the inbuilt villagers, tape or memory (accessed by middle instruction knob/lower right CV). The plague simulations

repeat sequentially for each of the macro processor modes which divide up the knob into 16 grosser segments.



This reader presents key texts for essential modes and lists each mode with a short description as follows:

Modes

Plague modes

- The first mode selects a plague mode depending on the instruction encountered // mutation also
- SIR - epidemiological model – susceptible (S), infected (I), and resistant (R)
- Hodgepodge - A.K. Dewdney. The hodgepodge machine makes waves. Scientific American. August 1988. p. 104
- Cellular automata
- The Game of life
- Simple increment
- Shift instructions
- Chunk instructions
- Wireworld cellular automata: Turing-complete, simulates electronic logic elements.
- Hodgepodge - differing implementation
- Forest fire model
- Krum unknown cellular automata model
- Hodgepodge - differing implementation
- The Game of life - differing implementation
- Cellular automata - differing implementation
- Mutation

Processor Units

- True Turing Machine - machine description input on instruction knob
- Plague instruction set - a simulated processor model
- Brainfuck - esoteric programming language
- SIR - - epidemiological model – susceptible (S), infected (I), and resistant (R)

- Redcode/Corewars - Core War is a 1984 programming game created by D. G. Jones and A. K. Dewdney
- Simply inputs instructions into the tape/memory
- Masque of the Red Death - Edgar Allen Poe simulation
- Biota - an esoteric programming language in two dimensions
- Leaking stack instruction set - a simulated processor model
- Worm movements
- Befunge - a two-dimensional fungeoidal (in fact, the original fungeoid) esoteric programming language
- Langton's ant - a kind of Turmite
- Turmite - a turmite is a Turing machine which has an orientation
- Ant - a singular ant
- Simple Turing complete instruction set
- More Corewars

Some texts and code

SIR and contagion

It is the simple SIR epidemic without births or deaths.

```
#!/usr/bin/env python

#########################################################################
###    This is the PYTHON version of program 2.1 from page 19 of   #
### "Modeling Infectious Disease in humans and animals"            #
### by Keeling & Rohani.                                            #
###                                                                 #
### It is the simple SIR epidemic without births or deaths.        #
#########################################################################

#########################################################################
```

```
### Copyright (C) <2008> Ilias Soumpasis                                    #
### ilias.soumpasis@deductivethinking.com                                   #
### ilias.soumpasis@gmail.com                                               #
###                                                                         #
### This program is free software: you can redistribute it and/or modify #
### it under the terms of the GNU General Public License as published by #
### the Free Software Foundation, version 3.                                #
###                                                                         #
### This program is distributed in the hope that it will be useful,      #
### but WITHOUT ANY WARRANTY; without even the implied warranty of       #
### MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        #
### GNU General Public License for more details.                            #
###                                                                         #
### You should find a copy of the GNU General Public License at          #
###the Copyrights section or, see http://www.gnu.org/licenses.            #
#############################################################################


import scipy.integrate as spi
import numpy as np
import pylab as pl

#beta=1.2247 #transmission rate
beta=0.8
247 #transmission rate
gamma=0.14286 # recovery
TS=1.0
ND=46.0 # steps
S0=1-1e-6 # proportion susc
I0=1e-6 #proportion inf
INPUT = (S0, I0, 0.0)


def diff_eqs(INP,t):
'''The main set of equations'''
Y=np.zeros((3))
V = INP
Y[0] = - beta * V[0] * V[1]
Y[1] = beta * V[0] * V[1] - gamma * V[1]
Y[2] = gamma * V[1]
return Y   # For odeint
```

```
t_start = 0.0; t_end = ND; t_inc = TS
t_range = np.arange(t_start, t_end+t_inc, t_inc)
RES = spi.odeint(diff_eqs,INPUT,t_range)


# for ress in RES[:,1]:
#       #print int(ress*22000)
#       deltax=0
#       deltay=0
#       rot=0
#       print "%d %d %d %d %d" % (ress*22000,ress*22000,deltax,deltay,rot)

for res,ress,resss in RES:
print "%d %d %d %d %d" %
(500+resss*3000.0,500+resss*3000.0,ress*1500.0,ress*1500.0,res*36000.0)


# #Ploting
# pl.subplot(211)
# pl.plot(RES[:,0], '-g', label='Susceptibles')
# pl.plot(RES[:,2], '-k', label='Recovereds')
# pl.legend(loc=0)
# pl.title('Program_2_1.py')
# pl.xlabel('Time')
# pl.ylabel('Susceptibles and Recovereds')
# pl.subplot(212)
# pl.plot(RES[:,1], '-r', label='Infectious')
# pl.xlabel('Time')
# pl.ylabel('Infectious')
# pl.show()
```

Applied Mathematical Sciences, Vol. 3, 2009, no. 20, 959 - 968

Using Cellular Automata to Simulate Epidemic Diseases

S. Hoya White

Department of Applied Mathematics
Universidad de Salamanca
sarahw@usal.es

A. Martín del Rey

Department of Applied Mathematics
Universidad de Salamanca
delrey@usal.es

G. Rodríguez Sánchez

Department of Applied Mathematics
Universidad de Salamanca
gerardo@usal.es

Abstract

In this work, a novel model to simulate epidemic spreading is introduced. It is based on the use of two-dimensional cellular automata, where each cell stand for a square portion of the environment. It is suppose that the distribution of the population is homogeneous, that is, all cells have the same population. The laboratory simulations obtained seem to be in agreement with the real behavior of epidemic spreading.

Mathematics Subject Classification: 68Q80, 92D30

Keywords: Cellular automata, epidemic spreading, SIS model

1 Introduction

The public health issues have a lot of importance in our society, particularly viral spread through populated areas. Epidemics refer to a disease that spreads rapidly and extensively by infection and affecting many individuals in an area

at the same time. In this way, the most recent worrying epidemic was the Severe Acute Respiratoy Syndrome (SARS) outbreak in Asia. Consequently, since the publication of the first mathematical epidemic models (see [2, 5]), several mathematical models to study the dynamics of epidemics have been apperared in the literature. Many of them are based on differential equations, which neglect spatial aspects of the epidemic process. As a consequence, this can lead to very unrealistic results, such as, for example, endemic patterns relaying on very small densities of individuals, which are called "atto-foxes" or "nano-hawks" (see [4]). Other mathematical models are based on discrete systems: stochastic interacting particle models, cellular automata models, etc. (see, for example [1, 3, 6, 7]). These simple models eliminate the last mentioned shortcomings, and are specially suitable for computer simulations.

The main goal of this work is to introduce a new cellular automaton model to simulate the spread of a general epidemic. As is mentioned above, cellular automata (CA for short) are simple models of computation capable to simulate complex physical, biological or environmental phenomena (see [8]). Specifically, a two-dimensional CA is formed by a two-dimensional array of identical objects called cells, which are endowed with a state that change in discrete steps of time according to a specific rule. As the CA evolves, the updated function (whose variables are the states of the neighbors) determines how local interactions can influence the global behaviour of the system.

The rest of the paper is organized as follows: In Section 2 a review of bidimensional cellular automata is given; the proposed model is introduced in Section 3; some graphical simulations are shown in Section 4; and, finally, the conclusions and the future work are presented in Section 5.

2 An overview on cellular automata

Two-dimensional cellular automata are discrete dynamical systems formed by a finite number of identical objects called cells, which are arranged uniformly in a two-dimensional space. Each cell is endowed with a state, belonging to a finite state set, that changes at every discrete step of time according to a rule, called local transtition function. More precisely, a CA can be defined as a 4-uplet, $\mathcal{A} = (C, S, V, f)$, where C is the cellular space formed by a two-dimensional array of $r \times c$ cells (see Figure 1-(a)): $C = \{(a, b), 1 \leq a \leq r, 1 \leq b \leq c\}$.

The state of each cell is an element of a finite state set, S, in such a way that the state of the cell (a, b) at time t is denoted by $s_{ab}^{(t)} \in S$. The matrix $C^{(t)} = \left(s_{ij}^{(t)}\right)$ is called configuration of the CA at time t. Moreover, $C^{(0)}$ is the initial configuration of the CA.

The neighborhood of a cell (a, b) is the set of cells whose states at time t determine the state of the cell (a, b) at time $t + 1$, by means of the local

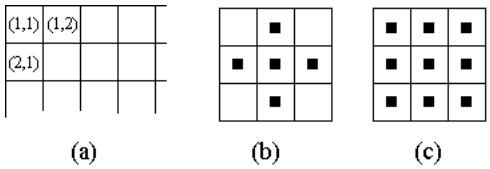


Figure 1: (a) Cellular space (b) Von Neuman neighborhood (c) Moore neighborhood

transition function. Depending on the process to be modelled, one can choose an appropiate neighborhood. Nevertheless, the traditional neighborhoods considered are the Von Neumann neighborhood (see Figure 1-(b)), and the Moore neighborhood (see Figure 1-(c)). Note that the main cell is also considered in its neighborhood. A neighborhood is defined by means of a finite set of indices $V = \{(\alpha_i, \beta_i), 1 \leq i \leq m\} \subset \mathbb{Z} \times \mathbb{Z}$, such that for every cell (a, b), its neighborhood, $V_{(a,b)}$, is the set of m cells given by

$$V_{(a,b)} = \{(a + \alpha_1, b + \beta_1), \ldots, (a + \alpha_m, b + \beta_m) : (\alpha_k, \beta_k) \in V\}.$$

Note that for Moore neighborhood, we have

$$V = \{(0,0), (-1,0), (-1,1), (0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1)\}.$$

Moreover, we will denote by V^* the set of indices defining the neighborhood of a cell (a, b) in which the main cell is not considered, that is, $V^* = V - \{(0,0)\}$.

As is mentioned above, the CA evolves deterministically in discrete time steps, changing the states of all cells according to a local transition function $f\colon S^m \to S$. The updated state of the cell (a, b) depends on the m variables of the local transition function, which are the previous states of the cells constituting its neighborhood, that is,

$$s_{ab}^{(t+1)} = f\left(s_{a+\alpha_1,b+\beta_1}^{(t)}, \ldots, s_{a+\alpha_m,b+\beta_m}^{(t)}\right).$$

As the cellular space is considered to be finite, boundary conditions must be taken into account in order to assure the well-defined dynamics of the CA. Several boundary conditions can be chosen depending on the phenomenon to be simulated. In this work, we will consider null boundary conditions, that is: $s_{ab}^{(t)} = 0$ if $a < 1$ or $a > r$, or if $b < 1$ or $b > c$.

A very important type of CA, suitable for simulating some ecological and biological systems, are those whose local transition functions are as follows:

$$s_{ab}^{(t+1)} = g\left(\sum_{(\alpha,\beta)\in V} \mu_{\alpha\beta}^{(a,b)} s_{a+\alpha,b+\beta}^{(t)}\right), \tag{1}$$

where $g\colon \mathbb{R} \to S$ is a suitable discretization function, and $\mu_{\alpha\beta}^{(a,b)} \in \mathbb{R}$ are the specific parameters of the system to be simulated.

3 The model

The proposed model is based on the use of a two-dimensional cellular endowed with a local transition function of the form (1). Each cell stands for a square area of the land in which the epidemic is spreading. Moreover, it is assumed that the population distribution is homogeneous, *i.e.*, all cells have the same population at every step of time.

The state of each cell at each time step is obtained from the fraction of the number of individuals of the cell which are infected by the epidemic, that is, $s_{ab}^{(t)}$ is a suitable discretization of

$$\frac{\text{infected population of } (a,b)}{\text{total population of } (a,b)}. \tag{2}$$

At time $t = 0$, the last expression is exactly the state for each cell. Consequently, $s_{ab}^{(t)} \in [0,1]$ for every t. As $s_{ab}^{(t)}$ is a real number and the state set is finite, we must discretize such value in order to obtain an element of S. In this work a state set with 11 elements will be considered:

$$S = \{s_0 = 0, s_1 = 0.1, \ldots, s_9 = 0.9, s_{10} = 1\}. \tag{3}$$

Furthermore, it is supposed that the state of each cell at a particular time step depends on the states of its eight nearest cells and the cell itself at the previous time step. Hence, the neighborhood considered is the Moore neighborhood.

Taking into account the definition of S, we will consider the following discretization function:

$$g\colon \mathbb{R} \to S, \quad x \mapsto g(x) = \begin{cases} 0, & \text{if } x < 0 \\ \frac{[10x]}{10}, & \text{if } 0 \le x \le 1 \\ 1, & \text{if } x > 1 \end{cases} \tag{4}$$

where $[x]$ stands for the $round(x)$ function.

These states change according to the following local transition function:

$$s_{ab}^{(t+1)} = g\left((1 - P(t))\, s_{ab}^{(t)} + \left(1 - s_{ab}^{(t)}\right) \left[\varepsilon s_{ab}^{(t)} + \sum_{(\alpha,\beta)\in V^*} \mu_{\alpha\beta}^{(a,b)} s_{a+\alpha,b+\beta}^{(t)} \right] \right),$$

where g is the discretization function given in (4).

The function $P(t)$ stands for the recovering process of the infected cells, *i.e.*, it measures the population of the cell that has recovered from the disease after

a time step. The explicit expression of the function $P(t)$ must be determined according to the epidemic to be modeled. For the sake of simplicity, in our work we suppose that this function is a polynomial of degree n.

Moreover, the real parameters ε and $\mu_{\alpha\beta}^{(a,b)}$ represent the main characteristics of the epidemic and the environment. Obviously, ε and $\mu_{\alpha\beta}^{(a,b)}$ are determined by the epidemic to be modeled.

Specifically, the real parameter $\mu_{\alpha\beta}^{(a,b)}$ involves three factors: The connection factor, $c_{\alpha\beta}^{(a,b)}$, the movement factor, $m_{\alpha\beta}^{(a,b)}$, between the cell (a,b) and its neighbor cell $(a+\alpha, b+\beta)$, and the virulence of the epidemic, $v \in [0,1]$. As a consequence, $\mu_{\alpha\beta}^{(a,b)} = c_{\alpha\beta}^{(a,b)} \cdot m_{\alpha\beta}^{(a,b)} \cdot v$, for every cell (a,b) and every $(\alpha,\beta) \in V^*$.

As is mentioned above, it is suppose that the way on infection is the contact between two individuals. Then, the non-infected individuals located at the cell (a,b) can be infected by the infected individuals of the main cell (a,b), or by the infected individuals located at a neighbor cell, $(a+\alpha, b+\beta)$, that have travelled to the cell (a,b).

In the first situation, that is, when all individuals considered belong to the cell (a,b), the infection process is given by the parameter $\varepsilon \in [0,1]$, which represents the portion the non-infected individuals of the cell at time t infected by the infected population of the cell at the same time.

In the second situation, that is, when the non-infected individuals of (a,b) are infected by the infected individuals of the neighbor cells, some type of connection (by airplane, by train, by car, etc.) between two neighbor cells must be considered in order to permit the epidemic propagation from a neighbor cell to the main cell. This connection is given by the coefficients $c_{\alpha\beta}^{(a,b)}$, such that if there is some connection between $(a+\alpha, b+\beta)$ and (a,b), then $c_{\alpha\beta}^{(a,b)} = 1$, and if there is not connection between these two cells, then $c_{\alpha\beta}^{(a,b)} = 0$.

The parameter $m_{\alpha\beta}^{(a,b)}$ gives the probability of an infected individual belongs to the cell $(a+\alpha, b+\beta)$ to be moved to the cell (a,b). As a consequence, $m_{\alpha\beta}^{(a,b)} \in [0,1]$.

4 Simulations

The cellular space in the next simulations will be formed by a two-dimensional array of 40×40 cells and, as is mentioned above the state set is formed by 11 elements (see (3)). To represent the state of each cell, a color code is used and it is shown in Figure 2. Moreover, we also suppose that $\varepsilon = 0.4$, $m_{\alpha\beta}^{(a,b)} = 0.4$ for all (a,b) and $(\alpha,\beta) \in V_{(a,b)}$, $v = 0.4$, and $P(t) = 0.2t + 0.2$. Note that for the sake of simplicity, these parameters are artificially chosen.

Moreover, the size of the time step must be considered according to the main characteristic of the epidemic and the environment.

Figure 2: Color codes

First of all, let us consider the case in which each cell is connected with all of its neighbor cells, that is $c_{\alpha\beta}^{(a,b)} = 1$ for all (a, b) and $(\alpha, \beta) \in V^*$. If the initial configuration is formed by all cells with state 0 except for three cells which are infected as follows: $s_{10,10}^{(0)} = 0.8, s_{20,20}^{(0)} = 1, s_{30,30}^{(0)} = 0.5$, then, the evolution of the epidemic spreading obtained from the CA is shown in Figure 3 (the evolution goes from left to right, and from top to botton). Note that in this case, after 11 iterations, the epidemic disappeared.

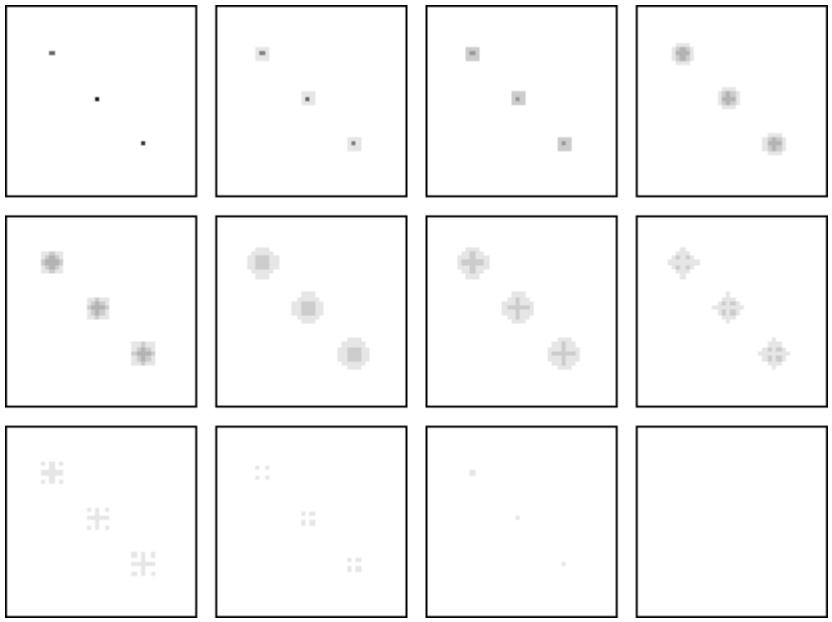


Figure 3: Simulation with all neighbor cells connected

Furthermore, suppose that the state of each cell at time $t = 0$ is randomly chosen, then the evolution is shown in Figure 4. In this case, the epidemic disappeared after 20 iterations. Moreover, the frequency of the different states in the evolution of the CA is shown in the following table, and the percentage of infected population evolves as is shown in Figure 5.

On the other hand, let us suppose that the conections between the cells are given by the following graph (see Figure 6), where each vertix stands for a cell and each edge between two vertices stands for a conection between these two cells. Note that in this case, the cellular space is of 13×13. If the parameters

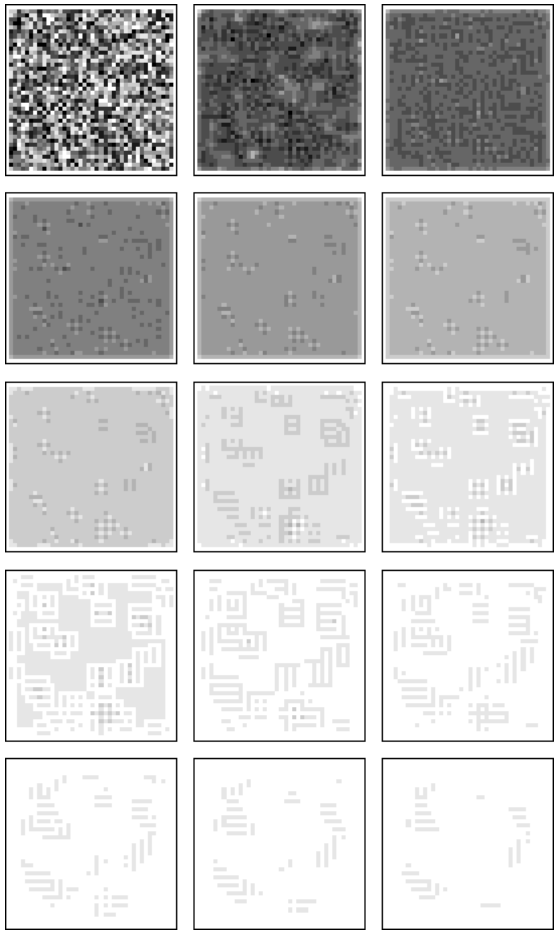


Figure 4: Simulation with a random initial configuration: Configurations $C^{(0)}$–$C^{(14)}$

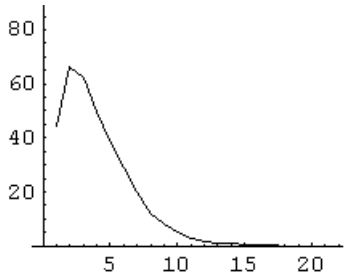


Figure 5: Evolution of the percentage of infected population

of the CA are $P(t) = 0.1t + 0.1$, $\varepsilon = 0.4$, $m_{\alpha\beta}^{(a,b)} = 0.8$, $\forall (a,b), \forall (\alpha,\beta) \in V^*$ and $v = 0.6$, then the evolution of the epidemic spreading is given in Figure 7.

t	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
0	82	174	156	164	184	150	151	143	166	151	79
1	0	0	1	3	27	105	557	628	238	37	4
2	0	0	0	1	7	171	933	488	0	0	0
3	0	0	0	4	205	1251	136	4	0	0	0
4	0	0	4	197	1329	70	0	0	0	0	0
5	0	0	209	1334	57	0	0	0	0	0	0
6	0	77	1444	79	0	0	0	0	0	0	0
7	27	1316	253	4	0	0	0	0	0	0	0
8	349	1186	64	1	0	0	0	0	0	0	0
9	762	806	32	0	0	0	0	0	0	0	0
10	1203	392	5	0	0	0	0	0	0	0	0
11	1336	264	0	0	0	0	0	0	0	0	0
12	1424	176	0	0	0	0	0	0	0	0	0
13	1477	123	0	0	0	0	0	0	0	0	0
14	1516	84	0	0	0	0	0	0	0	0	0
15	1550	50	0	0	0	0	0	0	0	0	0
16	1572	28	0	0	0	0	0	0	0	0	0
17	1586	14	0	0	0	0	0	0	0	0	0
18	1593	7	0	0	0	0	0	0	0	0	0
19	1598	2	0	0	0	0	0	0	0	0	0
20	1600	0	0	0	0	0	0	0	0	0	0

Table 1: Frequency of the states

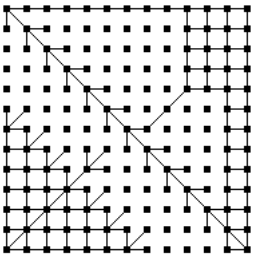


Figure 6: Connections between the cells

5 Conclusions and future work

In this work a new mathematical model to simulate the spreading of an epidemic is introduced. It is based on the use of two-dimensional cellular automata endowed with a suitable local transition function. The state of each cell is considered to be the portion of its population which is infected at each time step. The laboratory simulations obtained seem to be in agreement with the expected behaviour of a real epidemic.

Future work aimed at designing a more complete CA-based epidemic model involving additional effects such as the population movement (that is, a non-homogeneous population environment), or the effect of vaccination of the population, virus mutation, etc. Furthermore, it is also interesting to consider non-constant connections factors.

ACKNOWLEDGEMENTS. This work has been supported by the Consejería de Sanidad of Junta de Castilla y León (Spain) under grant SAN673 / SA23 / 08.

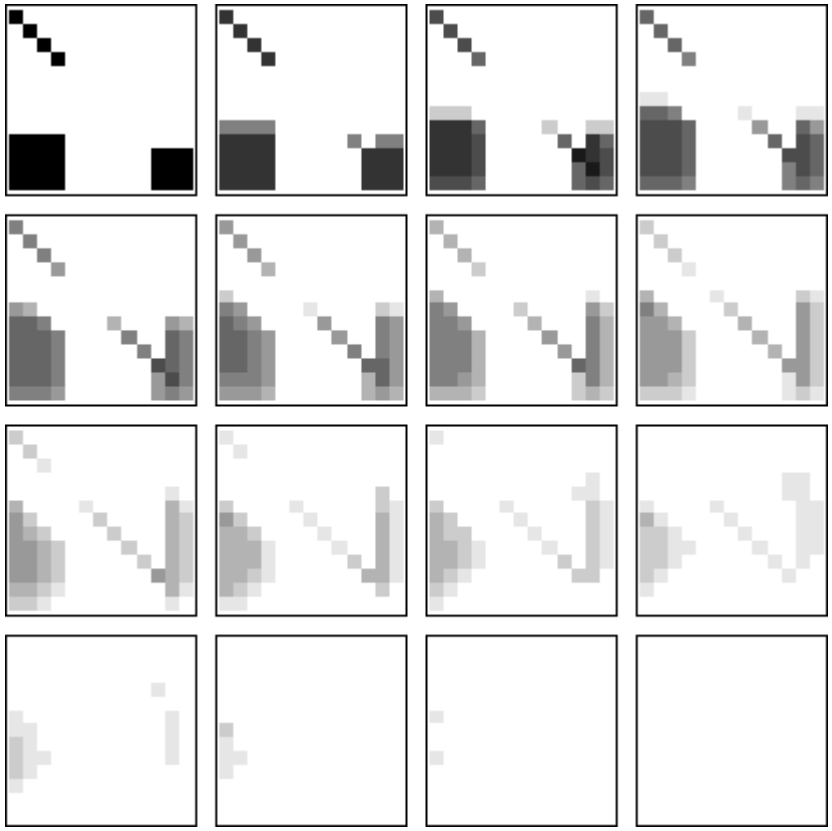


Figure 7: Simulation with variable connections

References

[1] C. Beauchemin, J. Samuel and J. Tuszynski, A simple cellular automaton model for influenza A viral infections, *J. Theor. Biol.,* **232** (2005), 223 - 234.

[2] W. O. Kermack and A. G. McKendrick, Contributions to the mathematical theory of epidemics, part I, *Proc. R. Soc. Edin. A,* **115** (1927), 700 - 721.

[3] M. L. Martins, G. Ceotto, S. G. Alves, C. C. B. Bufon, J. M. Silva and F.,F. Laranjeira, A cellular automata model for citrus variegated chlorosis, *Physica A,* **295** (2001), 42 - 48.

[4] D. Molisson, The dependence of epidemic and population velocities on basic parameters, *Math. Biosci.,* **107** (1991), 255-287.

[5] R. Ross, *The prevention of malaria,* 2nd edition, Murray, London, 1911.

[6] J. Satsuma, R. Willox, A. Ramani, B. Grammaticos and A. S. Carstea, Extending the SIR epidemic model, *Physica A,* **336** (2004), 369–375.

[7] G. Ch. Sirakoulis, I. Karafyllidis and A. Thanailakis, A cellular automaton model for the effects of population movement and vaccination on epidemic propagation, *Ecol. Model.,* **133** (2000), 209-223.

[8] T. Toffoli and N. Margolus, *Cellular Automata Machines. A New Environment for Modeling,* MIT Press, Cambridge, MA, 1987.

Received: November, 2008

In: Infectious Disease Modelling Research Progress ISBN 978-1-60741-347-9
Editors: J.M. Tchuenche and C. Chiyaka, pp. 133-150

Chapter 4

WHEN ZOMBIES ATTACK!: MATHEMATICAL MODELLING OF AN OUTBREAK OF ZOMBIE INFECTION

***Philip Munz*[1]*, *Ioan Hudea*[1]†, *Joe Imad*[2]‡, *Robert J. Smith?*[3]§**
[1]School of Mathematics and Statistics, Carleton University,
1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada
[2]Department of Mathematics, The University of Ottawa,
585 King Edward Ave, Ottawa ON K1N 6N5, Canada
[2]Department of Mathematics and Faculty of Medicine, The University of Ottawa,
585 King Edward Ave, Ottawa ON K1N 6N5, Canada

Abstract

Zombies are a popular figure in pop culture/entertainment and they are usually portrayed as being brought about through an outbreak or epidemic. Consequently, we model a zombie attack, using biological assumptions based on popular zombie movies. We introduce a basic model for zombie infection, determine equilibria and their stability, and illustrate the outcome with numerical solutions. We then refine the model to introduce a latent period of zombification, whereby humans are infected, but not infectious, before becoming undead. We then modify the model to include the effects of possible quarantine or a cure. Finally, we examine the impact of regular, impulsive reductions in the number of zombies and derive conditions under which eradication can occur. We show that only quick, aggressive attacks can stave off the doomsday scenario: the collapse of society as zombies overtake us all.

1. Introduction

A zombie is a reanimated human corpse that feeds on living human flesh [1]. Stories about zombies originated in the Afro-Caribbean spiritual belief system of Vodou (anglicised

*E-mail address: pmunz@connect.carleton.ca
†E-mail address: iahudea@connect.carleton.ca
‡E-mail address: jimad050@uottawa.ca
§E-mail address: rsmith43@uottawa.ca. Corresponding author.

voodoo). These stories described people as being controlled by a powerful sorcerer. The walking dead became popular in the modern horror fiction mainly because of the success of George A. Romero's 1968 film, Night of the Living Dead [2]. There are several possible etymologies of the word zombie. One of the possible origins is *jumbie*, which comes from the Carribean term for ghost. Another possible origin is the word *nzambi* which in Kongo means 'spirit of a dead person'. According to the Merriam-Webster dictionary, the word zombie originates from the word *zonbi*, used in the Louisiana Creole or the Haitian Creole. According to the Creole culture, a zonbi represents a person who died and was then brought to life without speech or free will.

The followers of Vodou believe that a dead person can be revived by a sorcerer [3]. After being revived, the zombies remain under the control of the sorcerer because they have no will of their own. Zombi is also another name for a Voodoo snake god. It is said that the sorcerer uses a 'zombie powder' for the zombification. This powder contains an extremely powerful neurotoxin that temporarily paralyzes the human nervous system and it creates a state of hibernation. The main organs, such as the heart and lungs, and all of the bodily functions, operate at minimal levels during this state of hibernation. What turns these human beings into zombies is the lack of oxygen to the brain. As a result of this, they suffer from brain damage.

A popular belief in the Middle Ages was that the souls of the dead could return to earth one day and haunt the living [4]. In France, during the Middle Ages, they believed that the dead would usually awaken to avenge some sort of crime committed against them during their life. These awakened dead took the form of an emaciated corpse and they wandered around graveyards at night. The idea of the zombie also appears in several other cultures, such as China, Japan, the Pacific, India, Persia, the Arabs and the Americas.

Modern zombies (the ones illustrated in books, films and games [1, 5]) are very different from the voodoo and the folklore zombies. Modern zombies follow a standard, as set in the movie Night of the Living Dead [2]. The ghouls are portrayed as being mindless monsters who do not feel pain and who have an immense appetite for human flesh. Their aim is to kill, eat or infect people. The 'undead' move in small, irregular steps, and show signs of physical decomposition such as rotting flesh, discoloured eyes and open wounds. Modern zombies are often related to an apocalypse, where civilization could collapse due to a plague of the undead. The background stories behind zombie movies, video games etc, are purposefully vague and inconsistent in explaining how the zombies came about in the first place. Some ideas include radiation (Night of the Living Dead [2]), exposure to airborne viruses (Resident Evil [6]), mutated diseases carried by various vectors (Dead Rising [7] claimed it was from bee stings of genetically altered bees). Shaun of the Dead [8] made fun of this by not allowing the viewer to determine what actually happened.

When a susceptible individual is bitten by a zombie, it leaves an open wound. The wound created by the zombie has the zombie's saliva in and around it. This bodily fluid mixes with the blood, thus infecting the (previously susceptible) individual.

The zombie that we chose to model was characterised best by the popular-culture zombie. The basic assumptions help to form some guidelines as to the specific type of zombie we seek to model (which will be presented in the next section). The model zombie is of the classical pop-culture zombie: slow moving, cannibalistic and undead. There are other 'types' of zombies, characterised by some movies like 28 Days Later [9] and the 2004 re-

make of Dawn of the Dead [10]. These 'zombies' can move faster, are more independent and much smarter than their classical counterparts. While we are trying to be as broad as possible in modelling zombies – especially since there are many varieties – we have decided not to consider these individuals.

2. The Basic Model

For the basic model, we consider three basic classes:

- Susceptible (S).
- Zombie (Z).
- Removed (R).

Susceptibles can become deceased through 'natural' causes, i.e., non-zombie-related death (parameter δ). The removed class consists of individuals who have died, either through attack or natural causes. Humans in the removed class can resurrect and become a zombie (parameter ζ). Susceptibles can become zombies through transmission via an encounter with a zombie (transmission parameter β). Only humans can become infected through contact with zombies, and zombies only have a craving for human flesh so we do not consider any other life forms in the model. New zombies can only come from two sources:

- The resurrected from the newly deceased (removed group).
- Susceptibles who have 'lost' an encounter with a zombie.

In addition, we assume the birth rate is a constant, Π. Zombies move to the removed class upon being 'defeated'. This can be done by removing the head or destroying the brain of the zombie (parameter α). We also assume that zombies do not attack/defeat other zombies.

Thus, the basic model is given by

$$\begin{aligned} S' &= \Pi - \beta SZ - \delta S \\ Z' &= \beta SZ + \zeta R - \alpha SZ \\ R' &= \delta S + \alpha SZ - \zeta R\,. \end{aligned}$$

This model is illustrated in Figure 1.

This model is slightly more complicated than the basic SIR models that usually characterise infectious diseases [11], because this model has two mass-action transmissions, which leads to having more than one nonlinear term in the model. Mass-action incidence specifies that an average member of the population makes contact sufficient to transmit infection with βN others per unit time, where N is the total population without infection. In this case, the infection is zombification. The probability that a random contact by a zombie is made with a susceptible is S/N; thus, the number of new zombies through this transmission process in unit time per zombie is:

$$(\beta N)(S/N)Z = \beta SZ\,.$$

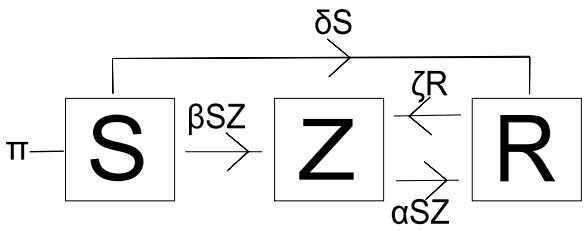


Figure 1. The basic model.

We assume that a susceptible can avoid zombification through an altercation with a zombie by defeating the zombie during their contact, and each susceptible is capable of resisting infection (becoming a zombie) at a rate α. So, using the same idea as above with the probability Z/N of random contact of a susceptible with a zombie (not the probability of a zombie attacking a susceptible), the number of zombies destroyed through this process per unit time per susceptible is:

$$(\alpha N)(Z/N)S \quad = \quad \alpha SZ\,.$$

The ODEs satisfy

$$S' + Z' + R' \quad = \quad \Pi$$

and hence

$$S + Z + R \quad \to \quad \infty$$

as $t \to \infty$, if $\Pi \neq 0$. Clearly $S \not\to \infty$, so this results in a 'doomsday' scenario: an outbreak of zombies will lead to the collapse of civilisation, as large numbers of people are either zombified or dead.

If we assume that the outbreak happens over a short timescale, then we can ignore birth and background death rates. Thus, we set $\Pi = \delta = 0$.

Setting the differential equations equal to 0 gives

$$\begin{aligned} -\beta SZ &= 0 \\ \beta SZ + \zeta R - \alpha SZ &= 0 \\ \alpha SZ - \zeta R &= 0\,. \end{aligned}$$

From the first equation, we have either $S = 0$ or $Z = 0$. Thus, it follows from $S = 0$ that we get the 'doomsday' equilibrium

$$(\bar{S}, \bar{Z}, \bar{R}) \quad = \quad (0, \bar{Z}, 0)\,.$$

When $Z = 0$, we have the disease-free equilibrium

$$(\bar{S}, \bar{Z}, \bar{R}) \quad = \quad (N, 0, 0)\,.$$

These equilibrium points show that, regardless of their stability, human-zombie coexistence is impossible.

The Jacobian is then

$$J = \begin{bmatrix} -\beta Z & -\beta S & 0 \\ \beta Z - \alpha Z & \beta S - \alpha S & \zeta \\ \alpha Z & \alpha S & -\zeta \end{bmatrix}.$$

The Jacobian at the disease-free equilibrium is

$$J(N,0,0) = \begin{bmatrix} 0 & -\beta N & 0 \\ 0 & \beta N - \alpha N & \zeta \\ 0 & \alpha N & -\zeta \end{bmatrix}.$$

We have

$$\det(J - \lambda I) = -\lambda\{\lambda^2 + [\zeta - (\beta - \alpha)N]\lambda - \beta\zeta N\}.$$

It follows that the characteristic equation always has a root with positive real part. Hence, the disease-free equilibrium is always unstable.

Next, we have

$$J(0,\bar{Z},0) = \begin{bmatrix} -\beta\bar{Z} & 0 & 0 \\ \beta\bar{Z} - \alpha\bar{Z} & 0 & \zeta \\ \alpha\bar{Z} & 0 & -\zeta \end{bmatrix}.$$

Thus,

$$\det(J - \lambda I) = -\lambda(-\beta\bar{Z} - \lambda)(-\zeta - \lambda).$$

Since all eigenvalues of the doomsday equilibrium are negative, it is asymptotically stable. It follows that, in a short outbreak, zombies will likely infect everyone.

In the following figures, the curves show the interaction between susceptibles and zombies over a period of time. We used Euler's method to solve the ODE's. While Euler's method is not the most stable numerical solution for ODE's, it is the easiest and least time-consuming. See Figures 2 and 3 for these results. The MATLAB code is given at the end of this chapter. Values used in Figure 3 were $\alpha = 0.005$, $\beta = 0.0095$, $\zeta = 0.0001$ and $\delta = 0.0001$.

3. The Model with Latent Infection

We now revise the model to include a latent class of infected individuals. As discussed in Brooks [1], there is a period of time (approximately 24 hours) after the human susceptible gets bitten before they succumb to their wound and become a zombie.

We thus extend the basic model to include the (more 'realistic') possibility that a susceptible individual becomes infected before succumbing to zombification. This is what is seen quite often in pop-culture representations of zombies ([2, 6, 8]).

Changes to the basic model include:

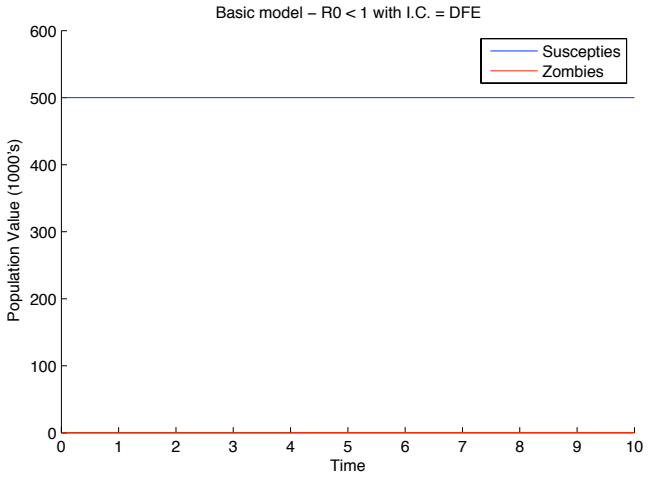


Figure 2. The case of no zombies. However, this equilibrium is unstable.

- Susceptibles first move to an infected class once infected and remain there for some period of time.
- Infected individuals can still die a 'natural' death before becoming a zombie; otherwise, they become a zombie.

We shall refer to this as the SIZR model. The model is given by

$$\begin{aligned} S' &= \Pi - \beta SZ - \delta S \\ I' &= \beta SZ - \rho I - \delta I \\ Z' &= \rho I + \zeta R - \alpha SZ \\ R' &= \delta S + \delta I + \alpha SZ - \zeta R \end{aligned}$$

The SIZR model is illustrated in Figure 4

As before, if $\Pi \neq 0$, then the infection overwhelms the population. Consequently, we shall again assume a short timescale and hence $\Pi = \delta = 0$. Thus, when we set the above equations to 0, we get either $S = 0$ or $Z = 0$ from the first equation. It follows again from our basic model analysis that we get the equilibria:

$$\begin{aligned} Z = 0 \implies (\bar{S}, \bar{I}, \bar{Z}, \bar{R}) &= (N, 0, 0, 0) \\ S = 0 \implies (\bar{S}, \bar{I}, \bar{Z}, \bar{R}) &= (0, 0, \bar{Z}, 0) \end{aligned}$$

Thus, coexistence between humans and zombies/infected is again not possible.

In this case, the Jacobian is

$$J = \begin{bmatrix} -\beta Z & 0 & -\beta S & 0 \\ \beta Z & -\rho & \beta S & 0 \\ -\alpha Z & \rho & -\alpha S & \zeta \\ \alpha Z & 0 & \alpha S & -\zeta \end{bmatrix}.$$

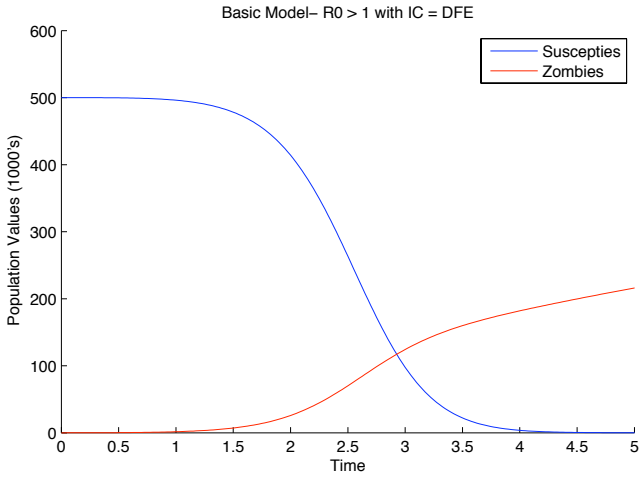


Figure 3. Basic model outbreak scenario. Susceptibles are quickly eradicated and zombies take over, infecting everyone.

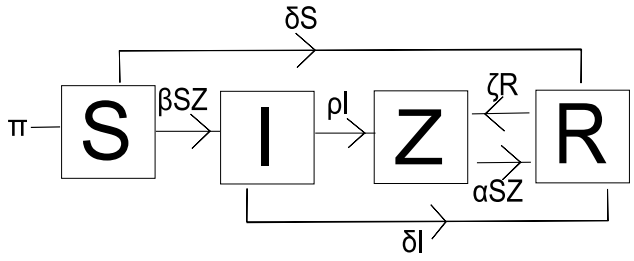


Figure 4. The SIZR model flowchart: the basic model with latent infection.

First, we have

$$\begin{aligned}
\det(J(N,0,0,0)-\lambda I) &= \det\begin{bmatrix} -\lambda & 0 & -\beta N & 0 \\ 0 & -\rho-\lambda & \beta N & 0 \\ 0 & \rho & -\alpha N-\lambda & \zeta \\ 0 & 0 & \alpha N & -\zeta-\lambda \end{bmatrix} \\
&= -\lambda\det\begin{bmatrix} -\rho-\lambda & \beta N & 0 \\ \rho & -\alpha N-\lambda & \zeta \\ 0 & \alpha N & -\zeta-\lambda \end{bmatrix} \\
&= -\lambda\left[-\lambda^3-(\rho+\zeta+\alpha N)\lambda^2-(\rho\alpha N+\rho\zeta-\rho\beta N)\lambda\right. \\
&\quad \left.+\rho\zeta\beta N\right].
\end{aligned}$$

Since $\rho\zeta\beta N > 0$, it follows that $\det(J(N,0,0,0)-\lambda I)$ has an eigenvalue with positive real part. Hence, the disease-free equilibrium is unstable.

Next, we have

$$\det(J(0,0,\bar{Z},0)-\lambda I) \quad = \quad \det\begin{bmatrix} -\beta\bar{Z}-\lambda & 0 & 0 & 0 \\ \beta\bar{Z} & -\rho-\lambda & 0 & 0 \\ -\alpha\bar{Z} & \rho & -\lambda & \zeta \\ \alpha\bar{Z} & 0 & 0 & -\zeta-\lambda \end{bmatrix}.$$

The eigenvalues are thus $\lambda = 0, -\beta\bar{Z}, -\rho, -\zeta$. Since all eigenvalues are nonpositive, it follows that the doomsday equilibrium is stable. Thus, even with a latent period of infection, zombies will again take over the population.

We plotted numerical results from the data again using Euler's method for solving the ODEs in the model. The parameters are the same as in the basic model, with $\rho = 0.005$. See Figure 5. In this case, zombies still take over, but it takes approximately twice as long.

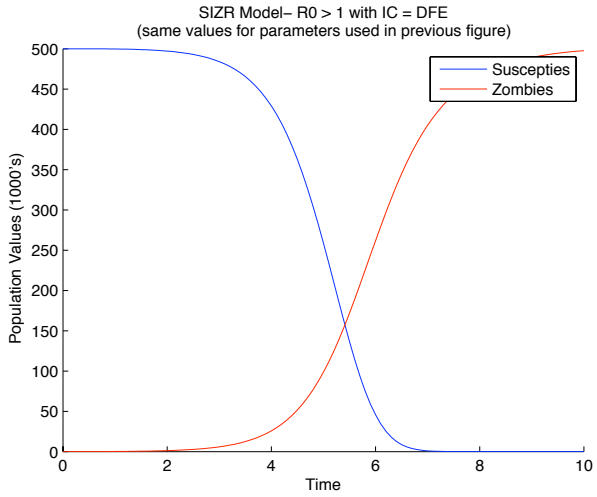


Figure 5. An outbreak with latent infection.

4. The Model with Quarantine

In order to contain the outbreak, we decided to model the effects of partial quarantine of zombies. In this model, we assume that quarantined individuals are removed from the population and cannot infect new individuals while they remain quarantined. Thus, the changes to the previous model include:

- The quarantined area only contains members of the infected or zombie populations (entering at rates κ and σ, respectively).
- There is a chance some members will try to escape, but any that tried to would be killed before finding their 'freedom' (parameter γ).
- These killed individuals enter the removed class and may later become reanimated as 'free' zombies.

The model equations are:

$$\begin{aligned} S' &= \Pi - \beta SZ - \delta S \\ I' &= \beta SZ - \rho I - \delta I - \kappa I \\ Z' &= \rho I + \zeta R - \alpha SZ - \sigma Z \\ R' &= \delta S + \delta I + \alpha SZ - \zeta R + \gamma Q \\ Q' &= \kappa I + \sigma Z - \gamma Q \,. \end{aligned}$$

The model is illustrated in Figure 6.

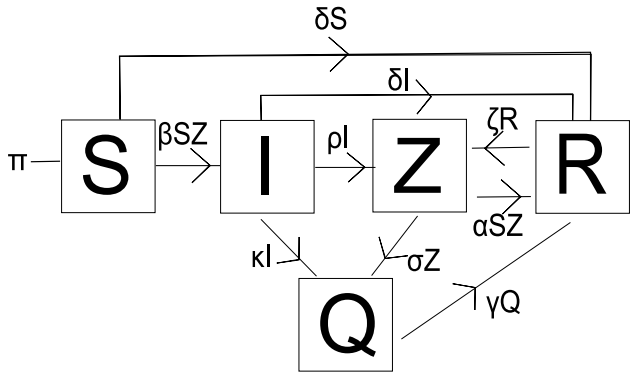


Figure 6. Model flow diagram for the Quarantine model.

For a short outbreak ($\Pi = \delta = 0$), we have two equilibria,

$$(\bar{S}, \bar{I}, \bar{Z}, \bar{R}, \bar{Q}) = (N, 0, 0, 0, 0), (0, 0, \bar{Z}, \bar{R}, \bar{Q})\,.$$

In this case, in order to analyse stability, we determined the basic reproductive ratio, R_0 [12] using the next-generation method [13]. The basic reproductive ratio has the property that if $R_0 > 1$, then the outbreak will persist, whereas if $R_0 < 1$, then the outbreak will be eradicated.

If we were to determine the Jacobian and evaluate it at the disease-free equilibrium, we would have to evaluate a nontrivial 5 by 5 system and a characteristic polynomial of degree of at least 3. With the next-generation method, we only need to consider the infective differential equations I', Z' and Q'. Here, F is the matrix of new infections and V is the matrix of transfers between compartments, evaluated at the disease-free equilibrium.

$$F = \begin{bmatrix} 0 & \beta N & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} \rho + \kappa & 0 & 0 \\ -\rho & \alpha N + \sigma & 0 \\ -\kappa & -\sigma & \gamma \end{bmatrix}$$

$$\begin{aligned} V^{-1} &= \frac{1}{\gamma(\rho+\kappa)(\alpha N+\sigma)} \begin{bmatrix} \gamma(\alpha N+\sigma) & 0 & 0 \\ \rho\gamma & \gamma(\rho+\kappa) & 0 \\ \rho\sigma + \kappa(\alpha N+\sigma) & \sigma(\rho+\kappa) & (\rho+\kappa)(\alpha N+\sigma) \end{bmatrix} \\ FV^{-1} &= \frac{1}{\gamma(\rho+\kappa)(\alpha N+\sigma)} \begin{bmatrix} \beta N\rho\gamma & \beta N\gamma(\rho+\kappa) & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \end{aligned}$$

This gives us

$$R_0 = \frac{\beta N \rho}{(\rho+\kappa)(\alpha N+\sigma)}.$$

It follows that the disease-free equilibrium is only stable if $R_0 < 1$. This can be achieved by increasing κ or σ, the rates of quarantining infected and zombified individuals, respectively. If the population is large, then

$$R_0 \approx \frac{\beta \rho}{(\rho+\kappa)\alpha}.$$

If $\beta > \alpha$ (zombies infect humans faster than humans can kill them, which we expect), then eradication depends critically on quarantining those in the primary stages of infection. This may be particularly difficult to do, if identifying such individuals is not obvious [8].

However, we expect that quarantining a large percentage of infected individuals is unrealistic, due to infrastructure limitations. Thus, we do not expect large values of κ or σ, in practice. Consequently, we expect $R_0 > 1$.

As before, we illustrate using Euler's method. The parameters were the same as those used in the previous models. We varied κ, σ, γ to satisfy $R_0 > 1$. The results are illustrated in Figure 7. In this case, the effect of quarantine is to slightly delay the time to eradication of humans.

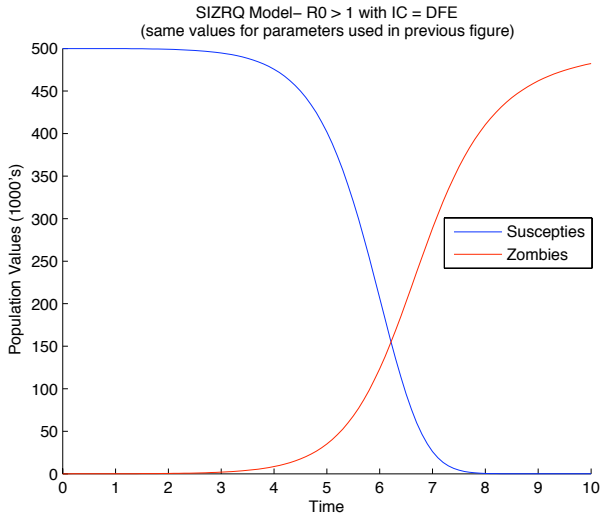


Figure 7. An outbreak with quarantine.

The fact that those individuals in Q were destroyed made little difference overall to the analysis as our intervention (i.e., destroying the zombies) did not have a major impact to the system (we were not using Q to eradicate zombies). It should also be noted that we still expect only two outcomes: either zombies are eradicated, or they take over completely.

Notice that, in Figure 7 at $t = 10$, there are fewer zombies than in the Figure 5 at $t = 10$. This is explained by the fact that the numerics assume that the Quarantine class continues

to exist, and there must still be zombies in that class. The zombies measured by the curve in the figure are considered the 'free' zombies: the ones in the Z class and not in Q.

5. A Model with Treatment

Suppose we are able to quickly produce a cure for 'zombie-ism'. Our treatment would be able to allow the zombie individual to return to their human form again. Once human, however, the new human would again be susceptible to becoming a zombie; thus, our cure does not provide immunity. Those zombies who resurrected from the dead and who were given the cure were also able to return to life and live again as they did before entering the R class.

Things that need to be considered now include:

- Since we have treatment, we no longer need the quarantine.
- The cure will allow zombies to return to their original human form regardless of how they became zombies in the first place.
- Any cured zombies become susceptible again; the cure does not provide immunity.

Thus, the model with treatment is given by

$$\begin{aligned} S' &= \Pi - \beta SZ - \delta S + cZ \\ I' &= \beta SZ - \rho I - \delta I \\ Z' &= \rho I + \zeta R - \alpha SZ - cZ \\ R' &= \delta S + \delta I + \alpha SZ - \zeta R\,. \end{aligned}$$

The model is illustrated in Figure 8.

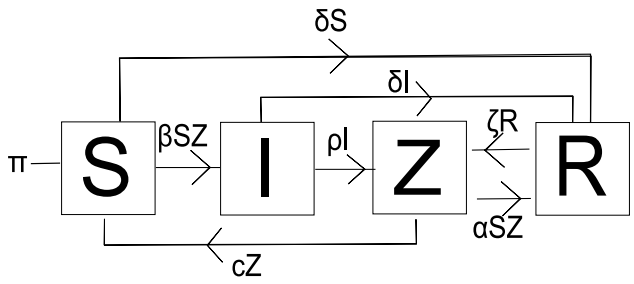


Figure 8. Model flowchart for the SIZR model with cure.

As in all other models, if $\Pi \neq 0$, then $S + I + Z + R \to \infty$, so we set $\Pi = \delta = 0$. When $Z = 0$, we get our usual disease-free equilibrium,

$$(\bar{S}, \bar{I}, \bar{Z}, \bar{R}) = (N, 0, 0, 0)\,.$$

However, because of the cZ term in the first equation, we now have the possibility of an endemic equilibrium $(\bar{S}, \bar{I}, \bar{Z}, \bar{R})$ satisfying

$$\begin{aligned} -\beta\bar{S}\bar{Z} + c\bar{Z} &= 0 \\ \beta\bar{S}\bar{Z} - \rho\bar{I} &= 0 \\ \rho\bar{I} + \zeta\bar{R} - \alpha\bar{S}\bar{Z} - c\bar{Z} &= 0 \\ \alpha\bar{S}\bar{Z} - \zeta\bar{R} &= 0\,. \end{aligned}$$

Thus, the equilibrium is

$$(\bar{S}, \bar{I}, \bar{Z}, \bar{R}) = \left(\frac{c}{\beta}, \frac{c}{\rho}\bar{Z}, \bar{Z}, \frac{\alpha c}{\zeta\beta}\bar{Z}\right).$$

The Jacobian is

$$J = \begin{bmatrix} \beta Z & 0 & -\beta S + c & 0 \\ \beta Z & -\rho & \beta S & 0 \\ -\alpha Z & \rho & -\alpha S - c & \zeta \\ \alpha Z & 0 & \alpha S & -\zeta \end{bmatrix}.$$

We thus have

$$\begin{aligned} \det(J(\bar{S}, \bar{I}, \bar{Z}, \bar{R}) - \lambda I) &= \det \begin{bmatrix} \beta\bar{Z} & 0 & 0 & 0 \\ \beta\bar{Z} & -\rho & c & 0 \\ -\alpha\bar{Z} & \rho & -\frac{\alpha c}{\beta} - c & \zeta \\ \alpha\bar{Z} & 0 & \frac{\alpha c}{\beta} & -\zeta \end{bmatrix} \\ &= -(\beta\bar{Z} - \lambda) \det \begin{bmatrix} -\rho & c & 0 \\ \rho & -\frac{\alpha c}{\beta} - c & \zeta \\ 0 & \frac{\alpha c}{\beta} & -\zeta \end{bmatrix} \\ &= -(\beta\bar{Z} - \lambda) \left\{ -\lambda \left[\lambda^2 + \left(\rho + \frac{\alpha c}{\beta} + c + \zeta \right) \lambda \right. \right. \\ &\quad \left. \left. + \frac{\zeta\alpha c}{\beta} + \frac{\rho\alpha c}{\beta} + \rho\zeta + c\zeta \right] \right\}. \end{aligned}$$

Since the quadratic expression has only positive coefficients, it follows that there are no positive eigenvalues. Hence, the coexistence equilibrium is stable.

The results are illustrated in Figure 9. In this case, humans are not eradicated, but only exist in low numbers.

6. Impulsive Eradication

Finally, we attempted to control the zombie population by strategically destroying them at such times that our resources permit (as suggested in [14]). It was assumed that it would be difficult to have the resources and coordination, so we would need to attack more than once, and with each attack, try and destroy more zombies. This results in an impulsive effect [15, 16, 17, 18].

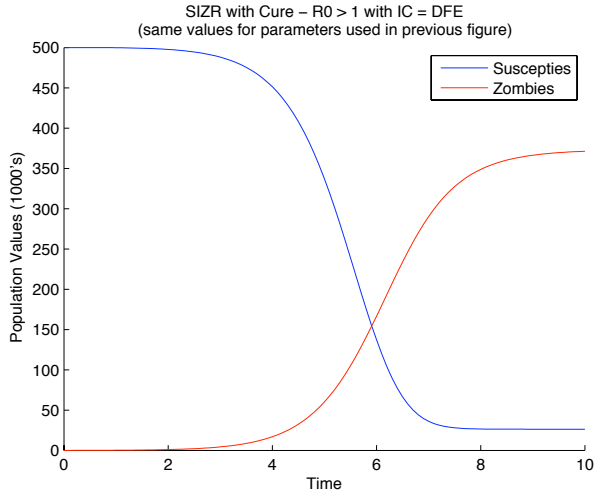


Figure 9. The model with treatment, using the same parameter values as the basic model.

Here, we returned to the basic model and added the impulsive criteria:

$$\begin{aligned} S' &= \Pi - \beta SZ - \delta S & t &\neq t_n \\ Z' &= \beta SZ + \zeta R - \alpha SZ & t &\neq t_n \\ R' &= \delta S + \alpha SZ - \zeta R & t &\neq t_n \\ \Delta Z &= -knZ & t &= t_n\,, \end{aligned}$$

where $k \in (0, 1]$ is the kill ratio and n denotes the number of attacks required until $kn > 1$. The results are illustrated in Figure 10.

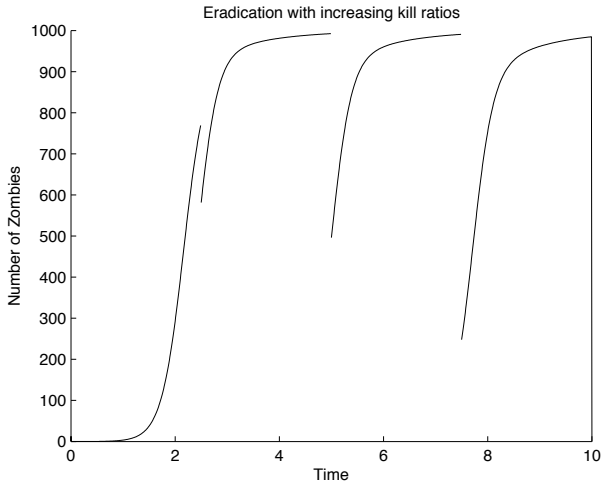


Figure 10. Zombie eradication using impulsive attacks.

In Figure 10, we used $k = 0.25$ and the values of the remaining parameters were

$(\alpha, \beta, \zeta, \delta) = (0.0075, 0.0055, 0.09, 0.0001)$. Thus, after 2.5 days, 25% of zombies are destroyed; after 5 days, 50% of zombies are destroyed; after 7.5 days, 75% of remaining zombies are destroyed; after 10 days, 100% of zombies are destroyed.

7. Discussion

An outbreak of zombies infecting humans is likely to be disastrous, unless extremely aggressive tactics are employed against the undead. While aggressive quarantine may eradicate the infection, this is unlikely to happen in practice. A cure would only result in some humans surviving the outbreak, although they will still coexist with zombies. Only sufficiently frequent attacks, with increasing force, will result in eradication, assuming the available resources can be mustered in time.

Furthermore, these results assumed that the timescale of the outbreak was short, so that the natural birth and death rates could be ignored. If the timescale of the outbreak increases, then the result is the doomsday scenario: an outbreak of zombies will result in the collapse of civilisation, with every human infected, or dead. This is because human births and deaths will provide the undead with a limitless supply of new bodies to infect, resurrect and convert. Thus, if zombies arrive, we must act quickly and decisively to eradicate them before they eradicate us.

The key difference between the models presented here and other models of infectious disease is that the dead can come back to life. Clearly, this is an unlikely scenario if taken literally, but possible real-life applications may include allegiance to political parties, or diseases with a dormant infection.

This is, perhaps unsurprisingly, the first mathematical analysis of an outbreak of zombie infection. While the scenarios considered are obviously not realistic, it is nevertheless instructive to develop mathematical models for an unusual outbreak. This demonstrates the flexibility of mathematical modelling and shows how modelling can respond to a wide variety of challenges in 'biology'.

In summary, a zombie outbreak is likely to lead to the collapse of civilisation, unless it is dealt with quickly. While aggressive quarantine may contain the epidemic, or a cure may lead to coexistence of humans and zombies, the most effective way to contain the rise of the undead is to hit hard and hit often. As seen in the movies, it is imperative that zombies are dealt with quickly, or else we are all in a great deal of trouble.

Acknowledgements

We thank Shoshana Magnet, Andy Foster and Shannon Sullivan for useful discussions. RJS? is supported by an NSERC Discovery grant, an Ontario Early Researcher Award and funding from MITACS.

```
function [ ] = zombies(a,b,ze,d,T,dt)
% This function will solve the system of ODE's for the basic model used in
% the Zombie Dynamics project for MAT 5187.  It will then plot the curve of
% the zombie population based on time.
% Function Inputs: a - alpha value in model: "zombie destruction" rate
%                  b - beta value in model: "new zombie" rate
```

```
%                   ze - zeta value in model: zombie resurrection rate
%                   d - delta value in model: background death rate
%                   T - Stopping time
%                   dt - time step for numerical solutions
% Created by Philip Munz, November 12, 2008

%Initial set up of solution vectors and an initial condition
N = 500;        %N is the population
n = T/dt;
t = zeros(1,n+1);
s = zeros(1,n+1);
z = zeros(1,n+1);
r = zeros(1,n+1);

s(1) = N;
z(1) = 0;
r(1) = 0;
t = 0:dt:T;

% Define the ODE's of the model and solve numerically by Euler's method:
for i = 1:n
    s(i+1) = s(i) + dt*(-b*s(i)*z(i));  %here we assume birth rate
     = background deathrate, so only term is -b term
    z(i+1) = z(i) + dt*(b*s(i)*z(i) -a*s(i)*z(i) +ze*r(i));
    r(i+1) = r(i) + dt*(a*s(i)*z(i) +d*s(i) - ze*r(i));
    if s(i)<0 || s(i) >N
        break
    end
    if z(i) > N || z(i) < 0
         break
    end
    if r(i) <0 || r(i) >N
        break
    end
end
hold on
plot(t,s,'b');
plot(t,z,'r');
legend('Suscepties','Zombies')

------------

function [z] = eradode(a,b,ze,d,Ti,dt,s1,z1,r1)
% This function will take as inputs, the initial value of the 3 classes.
% It will then apply Eulers method to the problem and churn out a vector of
% solutions over a predetermined period of time (the other input).
% Function Inputs: s1, z1, r1 - initial value of each ODE, either the
%                               actual initial value or the value after the
%                               impulse.
%                  Ti - Amount of time between inpulses and dt is time step
% Created by Philip Munz, November 21, 2008
k = Ti/dt;
%s = zeros(1,n+1);
%z = zeros(1,n+1);
%r = zeros(1,n+1);
```

```
%t = 0:dt:Ti;
s(1) = s1;
z(1) = z1;
r(1) = r1;
for i=1:k
    s(i+1) = s(i) + dt*(-b*s(i)*z(i));  %here we assume birth rate
     = background deathrate, so only term is -b term
    z(i+1) = z(i) + dt*(b*s(i)*z(i) -a*s(i)*z(i) +ze*r(i));
    r(i+1) = r(i) + dt*(a*s(i)*z(i) +d*s(i) - ze*r(i));
end

%plot(t,z)


------------

function [] = erad(a,b,ze,d,k,T,dt)
% This is the main function in our numerical impulse analysis, used in
% conjunction with eradode.m, which will simulate the eradication of
% zombies.  The impulses represent a coordinated attack against zombiekind
% at specified times.
% Function Inputs: a - alpha value in model: "zombie destruction" rate
%                  b - beta value in model: "new zombie" rate
%                  ze - zeta value in model: zombie resurrection rate
%                  d - delta value in model: background death rate
%                  k - "kill" rate, used in the impulse
%                  T - Stopping time
%                  dt - time step for numerical solutions
% Created by Philip Munz, November 21, 2008

N = 1000;
Ti = T/4; %We plan to break the solution into 4 parts with 4 impulses
n = Ti/dt;
m = T/dt;
s = zeros(1,n+1);
z = zeros(1,n+1);
r = zeros(1,n+1);
sol = zeros(1,m+1);  %The solution vector for all zombie impulses and such
t = zeros(1,m+1);
s1 = N;
z1 = 0;
r1 = 0;
%i=0;    %i is the intensity factor for the current impulse
%for j=1:n:T/dt
%    i = i+1;
%    t(j:j+n) = Ti*(i-1):dt:i*Ti;
%    sol(j:j+n) = eradode(a,b,ze,d,Ti,dt,s1,z1,r1);
%    sol(j+n) = sol(j+n)-i*k*sol(j+n);
%    s1 = N-sol(j+n);
%    z1 = sol(j+n+1);
%    r1 = 0;
%end

sol1 = eradode(a,b,ze,d,Ti,dt,s1,z1,r1);
sol1(n+1) = sol1(n+1)-1*k*sol1(n+1);    %347.7975;
```

```
s1 = N-sol1(n+1);
z1 = sol1(n+1);
r1 = 0;
sol2 = eradode(a,b,ze,d,Ti,dt,s1,z1,r1);
sol2(n+1) = sol2(n+1)-2*k*sol2(n+1);
s1 = N-sol2(n+1);
z1 = sol2(n+1);
r1 = 0;
sol3 = eradode(a,b,ze,d,Ti,dt,s1,z1,r1);
sol3(n+1) = sol3(n+1)-3*k*sol3(n+1);
s1 = N-sol3(n+1);
z1 = sol3(n+1);
r1 = 0;
sol4 = eradode(a,b,ze,d,Ti,dt,s1,z1,r1);
sol4(n+1) = sol4(n+1)-4*k*sol4(n+1);
s1 = N-sol4(n+1);
z1 = sol4(n+1);
r1 = 0;
sol=[sol1(1:n),sol2(1:n),sol3(1:n),sol4];
t = 0:dt:T;
t1 = 0:dt:Ti;
t2 = Ti:dt:2*Ti;
t3 = 2*Ti:dt:3*Ti;
t4 = 3*Ti:dt:4*Ti;
%plot(t,sol)
hold on
plot(t1(1:n),sol1(1:n),'k')
plot(t2(1:n),sol2(1:n),'k')
plot(t3(1:n),sol3(1:n),'k')
plot(t4,sol4,'k')
hold off
```

References

[1] Brooks, Max, 2003 *The Zombie Survival Guide - Complete Protection from the Living Dead*, Three Rivers Press, pp. 2-23.

[2] Romero, George A. (writer, director), 1968 *Night of the Living Dead*.

[3] Davis, Wade, 1988 *Passage of Darkness - The Ethnobiology of the Haitian Zombie*, Simon and Schuster pp. 14, 60-62.

[4] Davis, Wade, 1985 *The Serpent and the Rainbow*, Simon and Schuster pp. 17-20, 24, 32.

[5] Williams, Tony, 2003 *Knight of the Living Dead - The Cinema of George A. Romero*, Wallflower Press pp.12-14.

[6] Capcom, Shinji Mikami (creator), 1996-2007 *Resident Evil*.

[7] Capcom, Keiji Inafune (creator), 2006 *Dead Rising*.

[8] Pegg, Simon (writer, creator, actor), 2002 *Shaun of the Dead*.

[9] Boyle, Danny (director), 2003 *28 Days Later*.

[10] Snyder, Zack (director), 2004 *Dawn of the Dead*.

[11] Brauer, F. Compartmental Models in Epidemiology. In: Brauer, F., van den Driessche, P., Wu, J. (eds). *Mathematical Epidemiology.* Springer Berlin 2008.

[12] Heffernan, J.M., Smith, R.J., Wahl, L.M. (2005). Perspectives on the Basic Reproductive Ratio. *J R Soc Interface* **2**(4), 281-293.

[13] van den Driessche, P., Watmough, J. (2002) Reproduction numbers and sub-threshold endemic equilibria for compartmental models of disease transmission. *Math. Biosci.* **180**, 29-48.

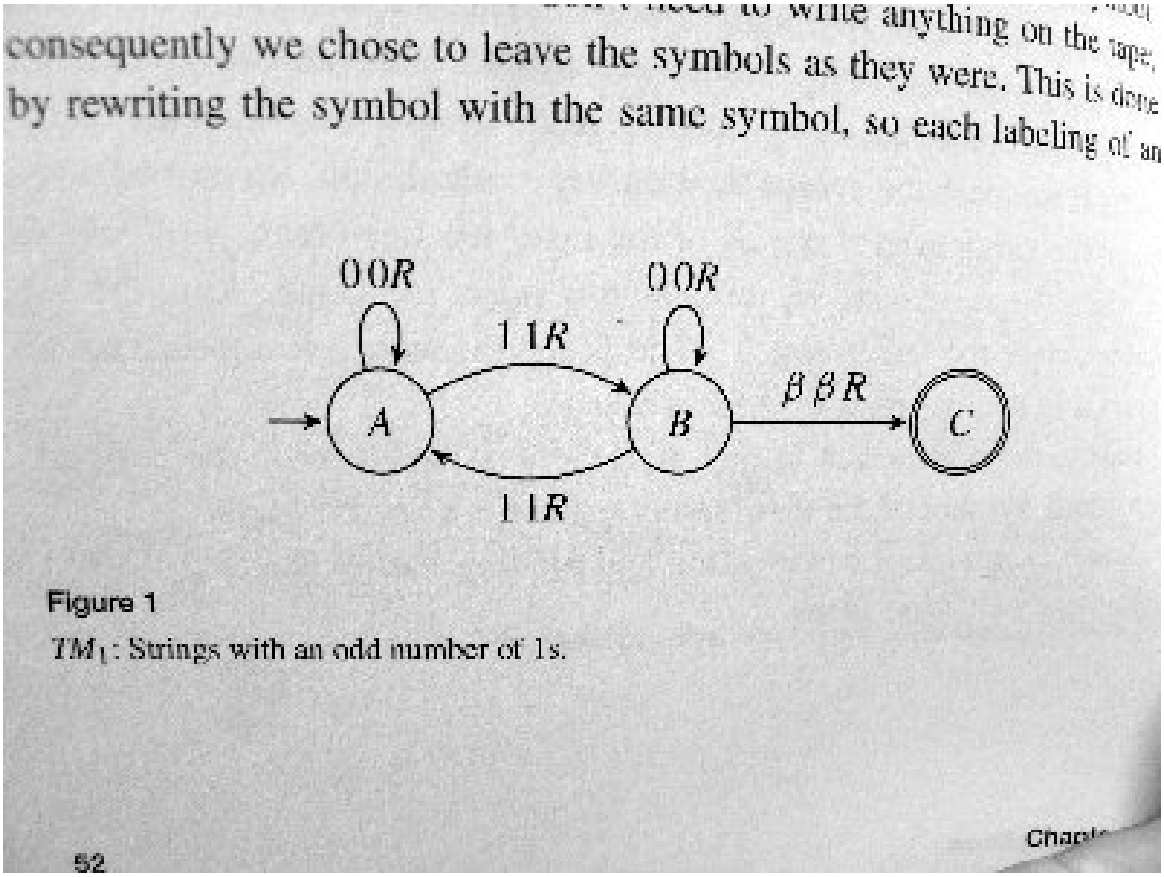
[14] Brooks, Max, 2006 *World War Z - An Oral History of the Zombie War*, Three Rivers Press.

[15] Bainov, D.D. & Simeonov, P.S. *Systems with Impulsive Effect.* Ellis Horwood Ltd, Chichester (1989).

[16] Bainov, D.D. & Simeonov, P.S. *Impulsive differential equations: periodic solutions and applications*. Longman Scientific and Technical, Burnt Mill (1993).

[17] Bainov, D.D. & Simeonov, P.S. *Impulsive Differential Equations: Asymptotic Properties of the Solutions.* World Scientific, Singapore (1995).

[18] Lakshmikantham, V., Bainov, D.D. & Simeonov, P.S. *Theory of Impulsive Differential Equations*. World Scientific, Singapore (1989).


consequently we chose to leave the symbols as they were. This is done
by rewriting the symbol with the same symbol, so each labeling of an
0 0 R
0 0 R
1 1 R
β β R
A
B
C
1 1 R
Figure 1
TM1: Strings with an odd number of 1s.
52

Turing Machines with Two Letters and Two States

Maurice Margenstern

Université Paul Verlaine — Metz, LITA, EA 3097
UFR MIM, Campus du Saulcy, 57045 METZ Cédex, France
margens@univ-metz.fr

In this paper we provide a survey of the technique that allows giving a simple proof that all Turing machines with two letters and two states have a decidable halting problem. The result was proved by L. Pavlotskaya in 1973.

1. Introduction

The notion of Turing machines appeared in 1936 in the famous paper by Alan M. Turing [1]. Turing's notion has since given rise to computer science. A few months later a paper by Emil Post appeared, describing the same object. Post's paper gives a very precise and simple description of the Turing machine which is more or less what is used today.

In this paper we follow the classical notion of a Turing machine. It is a device consisting of an infinite tape of squares indexed by $\mathbb{Z}$, a head that looks at the *scanned* square and which is in a *state* belonging to a fixed finite set of states. The index x of a square is called its *address* and we shall also say the *square* x for the *square with address* x. Each square contains a symbol belonging to a fixed finite set of *letters* also called the alphabet. Among these letters, a symbol is distinguished and called the *blank*. The device also contains a finite sequence of *instructions* described as a quintuple: two data constitute the *input* of the instruction, the scanned symbol, and the state of the head; three data constitute the *output*, the letter written by the head in place of the scanned letter, the new state that replaces the current state of the head, and the *move* performed by the head. After executing the instruction, the next cell to be scanned is to the left or right of the previously scanned cell, or is the same cell. A Turing machine with ℓ letters and s states is called an *$s\times\ell$-machine*. Note that classical Turing machines are *deterministic*, meaning that the input of two distinct instructions are different. The symbols ℓ and s have this meaning throughout this paper. The *computation* of a Turing machine is defined by the sequence of successive configurations obtained from an initial configuration, where all but a finite number of cells are blank.

The computation continues until a possible final configuration which occurs, in Turing's definition, after a halting state was called. In Turing's definition too, the result of the computation is what is written on the tape once the machine halted.

In [1] the description is more sophisticated and the basic properties of the set of Turing machines are given: the existence of universal machines, which, by definition, are able to simulate any Turing machine and the existence of a limit to the model as a problem that cannot be solved by any Turing machine. This problem is now known as the *halting problem*. It is an essential feature of Turing machines that their computation may halt or not and that to determine whether this is the case or not, which is the halting problem, turns out to be undecidable: there is no algorithm to solve it.

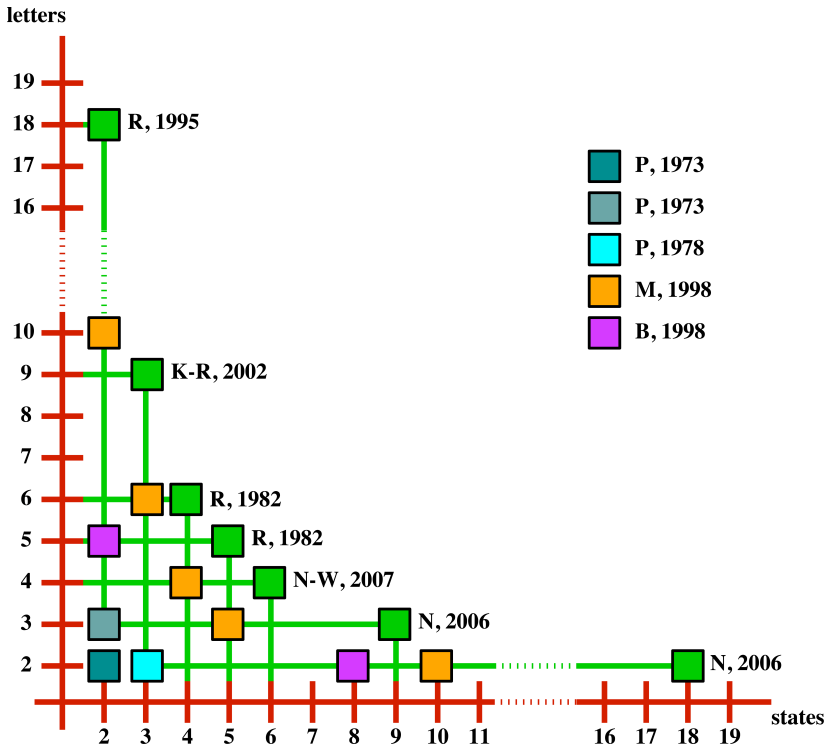


Figure 1. The small universal Turing machines and those with a decidable halting problem (in blue). The figure indicates the best known results only. The machines indicated with an orange or a purple square simulate the iterations of the $3x + 1$ function.

Later, in the 1950s, Claude Shannon raised the problem of what is now called the *descriptional complexity* of Turing machines: how many states and letters are needed in order to get universal machines? A race ensued to find the smallest Turing machine that was stopped by Yurii Rogozhin's result in 1982 [2]. Seven universal Turing machines were given, one in each of the following sets of machines: 2×21, 3×10, 4×6, 5×5, 7×4, 11×3, and 24×2 (see Figure 1). Nothing changed during the next 10 years. In 1992, Rogozhin improved

his 11×3 universal machine into a 10×3. In 1995, he proved that there are universal 2×18-machines. After an exchange of mails with the author, who had found a 2×21-machine, in 1998 Rogozhin found a 22×2 universal machine. In 2001, Claudio Baiocchi found a universal 19×2-machine. Then, in 2002, Rogozhin and Manfred Kudlek found a universal 3×9-machine. Recently, in 2006, Turlough Neary found a universal 18×2-machine and in 2007, Neary and Damien Woods found a universal 6×4-machine. Note that all of the machines mentioned from 1995 onwards were found at the occasion of a forthcoming edition of Machines, Computations, and Universality (MCU) conferences organized by the author.

Remember that all of these machines are universal in the sense that they simulate any Turing machine starting from a finite configuration and that when their computation stops the halting instruction is not taken into account when counting the instructions.

Turing machines on infinite configurations were also studied. The immortality problem, first studied by Philipp K. Hooper in 1966, consists of finding an initial infinite configuration on which the Turing machine never halts, whatever the initial state [3]. Other models of discrete computations were studied in this regard, in particular cellular automata and planar Turing machines. There, it turned out that by using initial infinite configurations it was possible to reduce the number of states and letters in order to obtain a universal device.

Now, we have to be careful about universality in this context: what does it mean? The reason is that if we allow arbitrary initial infinite configurations, then the halting problem becomes solvable. It is enough to encode the characteristic function of the set of all n for which the n^{th} Turing machine with input n halts on the tape of the Turing machine! This is why, during a certain time, initial infinite configurations were required to be ultimately periodic. This means that outside some finite interval, what remains of the tape on the left- and on the right-hand side is periodic, the periods being possibly different on each side of the tape. The rest of the simulation is the same as in the case of a classical Turing machine. Note that the classical situation is a particular case of this definition: the period is 1 and the content of the square is given; it must be the blank. This extended definition of universality is called *weak universality*. Although this generalization is very natural, there is a sharp difference from the classical case. The results indicated later also point to this difference.

Not everybody makes use of the term weak universality. Many a researcher does not think it that important to make a distinction on properties of the initial configuration leading to universal computations and, as an example, calls rule 110 universal.

In this context, the works of Stephen Wolfram on cellular automata inspired research that reached an important result: the weak universality of rule 110, [4, 5] an elementary cellular automaton. The corollary was the construction of very small weakly universal Turing machines, already announced in 2002, with significantly fewer instruc-

tions than the machines in Figure 1: eight instructions in 2005 [5] and five instructions in 2007 [5, 6]. Another difference is that the halting of these very small machines is not obtained by a halting instruction. This point about the way of halting was already raised in [7] where a universal planar Turing machine with eight instructions is constructed that does not halt on a specific instruction. It was also raised in the construction of reversible computations, first of cellular automata and then of Turing machines, which forced people to slightly change the notion of halting: in this frame, it could no more be characterized by a unique configuration. In 2003 the author, in a joint work with Ludmila Pavlotskaya, proved that a Turing machine with four instructions, even coupled with a finite automaton, has a decidable halting problem [8]. In the same paper, the authors proved that there is a Turing machine with five instructions and a particular finite automaton such that the resulting couple is universal. This can be compared with the result in [6] established after the well-known challenge launched by Wolfram. The result in [6] is stronger than that of [8] as in [6] the tape of the Turing machine is initially fixed. Its initial configuration is not exactly periodic, but it is "regular" in the sense that the infinite word written on the tape can be generated by a Muller finite automaton.

In this paper, we are interested by the decidability side of the question, about which very little is known [9]. Marvin Minsky mentions an unpublished proof by him and one of his students in [10] as unreadable because it involves a huge number of cases. The first readable proof was published by Pavlotskaya and states the following.

Theorem 1. (Pavlotskaya [11]) The halting problem is decidable for any 2×2-Turing machine.

Later in [12], Kudlek proved the same result in a very different way, classifying the machines according to what the computations produce, thus including machines that never halt. It is interesting to note that all computations are more or less trivial except one case, putting aside the trivial permutations and symmetries on states and letters. This case was also found in [8] where it was proved to have an exponential time computation on a sequence of patterns of the form 1^n.

In this paper, we give a simple proof of Theorem 1 that is based on an analysis of the motion of the Turing machine head on its tape. Section 2 deals with this analysis. In Section 3 we prove Theorem 1.

2. Motion of the Turing Machine Head on Its Tape

In this section, we fix notions and notations for the rest of the paper. We denote by t the current *time* of execution, t being a non-negative integer. Usually, the initial time is denoted by t_0 and, most often,

$t_0 = 0$. The current instruction is performed at time t and we get the result at time $t+1$ when the next instruction is performed.

2.1 Two Position Lemmas

Let $h(t)$ be the position of the head on the tape at time t. We denote by $\eta(t)$ the state of the head at time t and by $\sigma(t, x)$ the content of the square x at time t. By definition, $\sigma(t, x) \ldots \sigma(t, x+L)$ is the word whose letters consist of the contents of the squares with addresses from x to $x+L$ at time t. This word will also be called the interval $[x, x+L]$ of the tape at time t. We define ℓ_0 and r_0 to be the left- and right-hand side ends of the smallest interval that contains all the nonblank squares of the tape together with the square scanned by the head at time 0, the initial time. We define two functions ℓ and r to indicate the limits of the current configuration at time t as

$$\begin{aligned} \ell(0) &= \ell_0, \\ r(0) &= r_0, \\ \ell(t+1) &= \min(h(t+1), \ell(t)), \\ r(t+1) &= \max(h(t+1), r(t)). \end{aligned}$$

In other terms, $\ell(t+1) < \ell(t)$ if and only if $h(t) = \ell(t)$ and the machine performs an instruction with a move to the left at time t. Symmetrically, $r(t+1) > r(t)$ if and only if $h(t) = r(t)$ and the machine performs an instruction with a move to the right at time t. The configuration at time t is denoted by C_t.

The functions ℓ and r allow us to define the notion of the head *exiting* the limits of the current configuration. Define the times of exit E as

$$\begin{aligned} E(0) &= 0 \\ E(i+1) &= \min\{t \mid h(t) < \ell(E(i)) \vee h(t) > r(E(i))\}. \end{aligned}$$

From this definition, when $E(i) \le t < E(i+1)$, we have:

$$[\ell(t), r(t)] \subseteq [\ell(E(i)), r(E(i))]$$

and, by *abus de langage*, we call $E(i)$ the i^{th} exit.

Now, such an exit is called a *left-* or *right-hand side* exit, depending on whether $\ell(E(i)) \neq \ell(E(i-1))$ or $r(E(i)) \neq r(E(i-1))$. This allows us to define functions LE and RE to denote the i^{th} left-hand or i^{th} right-hand side exit as

$$\begin{aligned} LE(0) &= 0, \\ LE(i+1) &= \min\{t \mid h(t) < \ell(LE(i))\}, \\ RE(0) &= 0, \\ RE(i+1) &= \min\{t \mid h(t) < r(RE(i))\}. \end{aligned}$$

The motion of the Turing machine head on its tape consists of a sequence of consecutive runs over an interval during which the head moves in the same direction each time. Let us call such a run *sweeping* and note that a sweeping may be finite or infinite. Also note that in the case of an infinite sweeping the motion of the machine is ultimately periodic. After a certain time the head encounters an infinite interval of blank cells and, because the move at each step is constant, it always scans a blank and the only changing parameter is its state. As the number of states is finite, there must be a repetition and this interval between two occurrences of the same state is a period of the motion.

Any sweeping has at least one *half-turn*, that is, a time and a position such that the next move is in the opposite direction of the previous move. The half-turn is called a *left-* or *right-hand side* half-turn, depending on whether it occurs on the left- or right-hand side of the sweeping. Now, we say that an exit is *extremal* if and only if the new limit of the configuration which it defines is a half-turn. We now define the functions LEE and REE to denote the left- and right-hand side extremal exits, respectively:

$$\begin{aligned} LEE(0) &= 0, \\ LEE(i+1) &= \min\{t \mid h(t) < \ell(LEE(i)) \wedge h(t+1) > h(t)\}, \\ REE(0) &= 0, \\ REE(i+1) &= \min\{t \mid h(t) > r(REE(i)) \wedge h(t-1) < h(t)\}. \end{aligned}$$

We now have the following first property: a finite interval $[a, b]$ of the tape is a *trap zone* for the machine starting from a time t_1 if and only if $a \le h(t) \le b$ for all t with $t \ge t_1$. We say that $[a, b]$ is a trap zone for the machine if there is a time t_1 starting from which it is a trap zone. Here is an easy lemma to test whether a given finite interval is a trap zone for the machine.

Lemma 1. Let $[a, b]$ be a finite interval of the tape of the machine M. Then, we know whether $[a, b]$ is a trap zone starting from a given time t_1 after at most $n^{b-a+1}(b-a+1)s+1$, where n is the size of the alphabet of the machine and s is the number of its states.

The obvious proof is left to the reader.

From Lemma 1, we have the following corollary, whose trivial proof is also left to the reader.

Corollary 1. The functions E, ℓ, and r are recursive and the finiteness of the domain of definition of E is recursively enumerable.

Now, we turn to an important lemma that relies on the same idea as Lemma 1.

Lemma 2. (Margenstern [13, 14]) Let M be a Turing machine and assume that there are two right-hand side exits at times t_1 and t_2, with $t_1 < t_2$ and that there is an address a such that:

i. $\forall\ t \in [t_1, t_2], h(t) \geq a$

ii. let $\delta = h(t_2) - h(t_1) \geq 0$; then: $\forall\ x \in [a, h(t_1)], \sigma(t_1, x) = \sigma(t_2, x + \delta)$

iii. $\eta(t_1) = \eta(t_2)$.

Then, the sequence of instructions on the time interval $[t_1, t_2 - 1]$ is an execution pattern that is endlessly repeated and we say that the motion of the machine is *ultimately periodic*.

Proof. The conditions of the lemma are illustrated by Figure 2. The statement assumes that between times t_1 and t_2 the head never goes to the left of square a, that at times t_1 and t_2 the head of the machine is under the same state **u**, and that the words of the interval $[a, h(t_1)]$ at time t_1 and of the interval $[a + \delta, h(t_2)]$ at time t_2 are the same.

Let C_{t_1} be the configuration at time t_1. Now, imagine that at time t_1 we replace the interval $]-\infty, a-1]$ by the same interval with all squares filled up with the blank. Let C'_1 be this new configuration. Then, the motion of the Turing machine on the tape between t_1 and t_2 is the same, whether it starts from C_{t_1} or C'_1. Now, we clearly can repeat the same for C_{t_2} being replaced by C'_2 where all squares on the left of $a + \delta$ are replaced by the blank for times t_2 and $t_2 + t_2 - t_1$. And so, the same motion is repeated during the time interval $[t_2, 2\,t_2 - t_1]$. And this can be repeated endlessly by an easy induction left to the reader. □

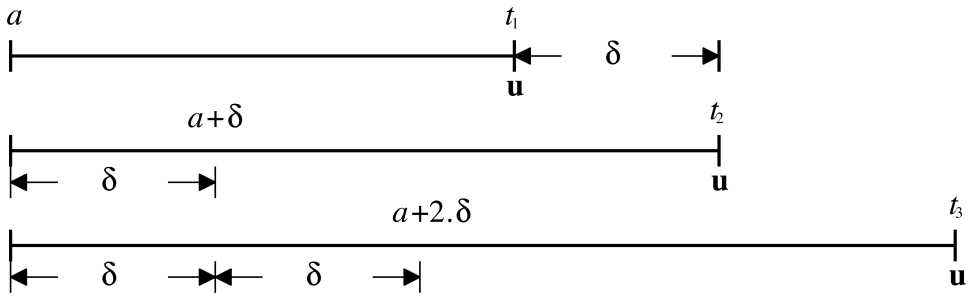


Figure 2. Illustrating the assumptions of Lemma 2 and its conclusion.

Note that the conditions of the assumption of Lemma 2 are recursively enumerable.

2.2 Additional Tools

The occurrence of a halting instruction is always recursively enumerable. And so, to prove that the halting problem is decidable, it is enough to focus on the nonhalting situations and prove that the general nonhalting situation is also recursively enumerable. This happens each time we have an algorithmic way to decide, after a certain time which may depend on the considered instance of the problem, that the machine will not halt. Later on, we shall only consider the nonhalting situations.

If the machine does not halt but instead remains within a finite interval, Lemma 1 indicates that the motion of the machine is ultimately periodic: the same finite sequence of instructions is endlessly repeated.

From now on, we assume that the Turing machine head traverses an infinite interval, which implies, of course, that the machine does not halt. If the motion involves an interval of time $[t, t+s]$ such that $t+i$, for $i \in \{0 \,..\, s\}$ is an instance of a right-hand side exit, then there will be two times $t+h$ and $t+k$, with $h < k$, such that the head of the machine is under the same state at these times. As a right-hand side instruction is performed at these times and because the head always scans a blank due to the exit, the same sequence of instructions performed between $t+h$ and $t+k-1$ will be repeated endlessly, involving a motion which is ultimately periodic. Of course, we can perform a similar argument for successive $s+1$ times at which a left-hand side exit occurs.

If we do not have this situation, necessarily, there are infinitely many extremal exits. In general, we have three cases.

i. There are infinitely many right-hand side extremal exits and finitely many left-hand side ones.

ii. There are infinitely many left-hand side extremal exits and finitely many right-hand side ones.

iii. There are infinitely many left-hand side extremal exits and infinitely many right-hand side ones.

Consider case i. We may assume that, after a time t_1, there are no more left-hand side exits. Denote by $lmp(j)$ the leftmost position of the machine head between $REE(j)$ and $REE(j{+}1)$. Call *lmp-time* for $lmp(j)$ the first time after $REE(j)$ that the machine head scans the square $lmp(j)$. Let a be the address of the last left-hand side exit. Then, $lmp(j) \geq a$ for all j. Let $\lambda = \liminf_{j\to\infty} lmp(j)$. Of course $\lambda \geq a$ whether λ is finite or not. Let $a_m = \inf_{j\geq m} lmp(j)$. Because $a_m \geq a$, and as a_m is an integer, a_m is reached: there is an integer $n_m \geq j$ such that $lmp(n_m) = a_m$ and so, $lmp(j) \geq lpm(n_m)$ for all $j \geq n_m$. Note that the position $lmp(n_m)$ is absolute in the following sense: after $REE(n_m)$, the machine never goes to the left of the square $lmp(n_m)$.

The sequence of the $lmp(n_m)$ is nondecreasing and we may assume that the sequence of the n_m is increasing.

Note that case ii can be dealt with symmetrically. We can define $rmp(j)$, the rightmost position between two consecutive left-hand side extremal exits and *rmp*-time, the first time that $rmp(j)$ is reached after $LEE(j)$.

2.3 Laterality and Color of an Instruction

Consider an instruction of the program of the Turing machine M. The instruction can be written $\mathbf{i}xyM\mathbf{j}$ where x is the scanned symbol at the current time, the current state of the machine is $\mathbf{i}$, and y is the symbol written by the machine head in place of x. Then, $\mathbf{j}$ is the new state taken by the machine head and M is the move performed by the head just after writing y: M is L, D, or S depending on whether the move makes the head go one square to the left, to the right, or to stay on the same square. Call the *color* of an instruction $\mathbf{i}xyM\mathbf{j}$ the triple xMy and call *laterality* of the instruction the value of M. Note that when we have a stationary instruction I, that is, $M = S$, the next instruction is either stationary or not. Repeating this argument, either there is a cycle of stationary instructions, or a halting or, after a certain sequence of consecutive stationary instructions, the next instruction is not stationary. If we are in the first or second case, we say that I is a *blocking* instruction. Consequently, if I is not blocking, it is ultimately followed by a nonstationary instruction J: by definition, the laterality of I is that of J. We say that a machine is *unilateral* if and only if all of its instructions have the same laterality: either L or R. By this we mean that when a stationary instruction is not blocking, its laterality is defined by one of the other instructions. We also consider that a halting instruction is a particular case of a blocking instruction.

Lemma 3. The halting problem is decidable for any unilateral Turing machine.

Proof. We may assume that all of the instructions are right-hand side. Because the occurrence of a blocking instruction is recursively enumerable it is enough to wait for its possible execution and we may assume that the machine head always goes to the right. It eventually exits from the right-hand side limit of the initial configuration and, later on, the machine head only sees a blank in the scanned cell. If after s+1 steps the machine does not find a blocking instruction, it will fall under two identical states and the sequence of instructions performed between the times of two consecutive occurrences of this state will be endlessly repeated meaning that the machine will not halt. □

However, note that if the machine has two heads, Lemma 3 is no longer true when stationary instructions are allowed [15].

Say that a Turing machine M is *erasing to the left* if all its left-hand side instructions write a blank. We have the following lemma.

Lemma 4. The halting problem is decidable for a Turing machine that is erasing to the left.

Note that if M is a unilateral machine whose laterality is L, we obtain another unilateral machine by changing L to R in all of the instructions. This operation is called *lateral symmetry*. In particular, Lemma 2 has a left-hand side version which is obviously obtained by lateral symmetry. We shall use this version only without further justification.

Proof of Lemma 4. We consider that the Turing machine M is erasing to the left and that it does not halt. We may assume that we are not in a trap zone nor in the situation of a sequence of times $t, \dots, t+s$ of exit on the same side. Accordingly, we are in one of the cases defined in Section 2.2.

Case i. We may assume that after a time t_1, all exits are right-hand side.

Consider the sequence of absolute *lmp* defined in Section 2.2, which, here also, we denote by $lmp(n_m)$, with increasing n_m.

Consider the configurations $C_{t_{n_1}}, \dots, C_{t_{n_s}}$ where t_{n_m} is the first time between $REE(n_m)$ and $REE(n_m+1)$ that $lmp(n_m)$ is reached by the machine head. For two of these configurations, the machine head will be under the same state. By definition of $C_{t_{n_m}}$, the head is on the $lmp(n_m)$ at its *lmp*-time. Now, because the machine writes blanks only when it performs a left-hand side instruction, there are only blanks on the right-hand side of the $lmp(n_m)$ at this time, from the address $lmp(n_m)+1$ up to infinity. Let h and k be the indices for which the state of the head is the same at t_{n_h} and t_{n_k}, with $h < k$. Because the $lmp(n_m)$ are absolute, the machine never goes to the left of $lmp(n_h)$ between t_{n_h} and t_{n_k}. And so, the sequence of instructions between these two times is repeated endlessly. Now, the occurrence of two configurations with these conditions is also recursively enumerable.

Case ii. For this case we can prove a stronger result and distinguish it as Sublemma 1.

Sublemma 1. Assume that for a Turing machine M, there are infinitely many left-hand side extremal exits and finitely many right-hand side ones and that all of the left-hand side instructions write the same letter y. Then the halting problem of M is decidable.

Proof of Sublemma 1. We assume that we are at a time after t_1, starting from which there are only left-hand side exits. Under these assumptions, the configuration of the i^{th} left-hand side exit after t_1 is

$$\mathbf{u}_i\, y^{n_i}\, W_i$$

where $\mathbf{u}_i$ is the state of the head at the exit, n_i is the number of consecutive y written by the head before the exit, and W_i is a word on the alphabet of the machine that does not start with y when not empty. Because there are no more right-hand side exits, the right-hand side limit of the configuration is always $r(t_1)$.

Denote by a_i the address of the rightmost y in the word y^{n_i}. We get that $a_i \leq r(t_1)$ for all i. As all the left-hand side instructions write y on the tape, it is plain that $a_{i+1} \geq a_i$. Let $a = \limsup_{i\to\infty} a_i$. We have that $a \leq r(t_1)$ and a is reached, because the a_i are all integers. If a is reached at a time t_2, then, after $s+1$ exits after t_2, we can see two exits which satisfy the assumptions of the left-hand side version of Lemma 2. Now, the occurrence of two such exits is recursively enumerable.

Case iii. Because there are infinitely many left- and right-hand side exits, there are infinitely many extremal right-hand side exits such that the next exit is on the left-hand side. And so, there is an increasing sequence of n_m such that $E(n_m)$ is a right-hand side exit and that $E(n_m+1)$ is a left-hand side one. Because the left-hand side instructions write the blank only, at time $E(n_m+1)$, the tape of the machine contains blanks only. Now, looking at the configurations at the times $E(n_1+1)$, ... , $E\left(n_{s+1}+1\right)$, two of them are identical: the tape is empty and the head is under the same state. This induces a sequence of endlessly repeating instructions and the occurrence of two such configurations is recursively enumerable. The motion of the machine is ultimately periodic because if there is no shift in the position of the machine head on the two detected configurations with an empty tape, the motion remains trapped in a finite interval. If there is a shift, the motion goes infinitely on one side of the tape only. And so, in all three situations, there cannot be infinitely many left- and right-hand side exits.

This means that Case iii does not occur for the considered machines.

Accordingly, the nonhalting of M is recursively enumerable in both possible cases of the motion of the machine head, which proves the lemma. □

3. Proof of Theorem 1

The proof relies on the following property.

Lemma 5. (Pavlotskaya [11]) Let M be a Turing machine on the alphabet $\{0, 1\}$ such that M has a single instruction whose laterality is L. Then the halting problem of M is decidable.

Theorem 1 is an immediate corollary of Lemma 5. Indeed, if a 2×2-machine has no halting instruction, it never halts. So, it has at least one halting instruction. On the others, it has at most one instruction whose laterality is not shared by the others. By lateral symmetry, we may assume that the laterality with a unique instruction is L.

Proof of Lemma 5. From Lemma 4, we may rule out the colors $x\,L\,0$ for the unique left-hand side instruction. And so, we remain with the colors $x\,L\,1$ with $x = 0$ or $x = 1$.

Color $1\,L\,1$. In this case, if the machine head reads a blank, it does not move to the left. Consider the case of a stationary instruction which is unique: otherwise we have a unilateral machine for which Lemma 3 applies. Then, if the color of the instruction is $0\,S\,0$, this blank square is a trap zone or it calls a right-hand side instruction. If the stationary instruction has the color $0\,S\,1$, and because we assumed that the machine has a single instruction with the laterality L, the stationary instruction calls the right-hand side one and so the machine goes to the right.

In all cases that the machine head reads a blank, if it does not halt or if it is not stuck in the same place, it goes to the right. In particular, if there is a right-hand side exit, the machine head goes to the right forever. Accordingly, we may assume that all exits are on the left-hand side. But now we are under the assumptions of Sublemma 1 as all left-hand side instructions write a 1. And so, we know that in this case, the nonhalting is recursively enumerable.

Color $0\,L\,1$. This time, if the machine reads a 1, it goes to the right.

Indeed, it cannot go to the left because one of the remaining nonhalting instructions is on the right-hand side and the other is either on the right-hand side or stationary. If the instruction is stationary, it is of the form $1\,S\,1$ or $1\,S\,0$. If it is stationary, and because we assume that there is a single instruction with the laterality L, a stationary instruction of the color $1\,S\,1$ reading a 1 will either keep the head on this square or call the right-hand side instruction. Now, if the stationary instruction is of the color $1\,S\,0$, and because we assume that there is a single instruction with the laterality L, it also necessarily calls the right-hand side instruction as the machine is not assumed to be unilateral.

If the new state of an instruction is the same as its current one we say that the instruction is *stable*. Otherwise, we call it *unstable*.

First, assume that the left-hand side instruction is unstable. Then, if it scans the square a at time t, it scans the square $a+1$ at time at most $t+4$, unless it is blocked in between.

Indeed, if a right-hand side instruction is performed, then $a+1$ is reached at time $t+3$. Otherwise, we have several cases. If the head performs a stationary instruction, either the head is blocked or it calls a right-hand side instruction again and $a+1$ is reached at time $t+2$. If the head performs the left-hand side instruction at time t, we have

the configuration $\epsilon_1 \bullet 0\, \epsilon_2$, where $\bullet$ represents the position of the head at this time. Then we have

$$(\nabla) \qquad \epsilon_1 \bullet 0\, \epsilon_2 \to \bullet \epsilon_1\, 1\, \epsilon_2 \to^{\alpha} \epsilon_1' \bullet 1\, \epsilon_2 \text{ with } \alpha \in \{1, 2\},$$

with α indicating the number of instructions applied by the machine. From the given analysis, scanning 1 in the square a, if the machine is not blocked in between, its head reaches a+1 at most two steps later.

Accordingly, if the machine does not halt, it goes forever to the right. Now we want to find out if that motion is predictable. From Section 2.2 we may assume that there are infinitely many extremal right exits. At an extremal exit the head arrives on a blank on which it performs a half-turn, meaning that the sequence (∇) is repeated, with $\epsilon_2 = 0$. Now, it is plain that considering $REE(1)$, $REE(2)$, and $REE(3)$, we can find two exits among them for which ϵ_1 is the same. Accordingly, in between the two times, we have a sequence of instructions that is endlessly repeated. Note that there are at most s steps between $REE(i)$ and $REE(i+1)$ but s may be very large.

At last, we remain with the case of a stable left-hand side instruction.

Note that if there is a left-hand side exit, it is performed by the left-hand side instruction and, since it is stable, the head goes infinitely to the left.

And so, we may assume that there are no left-hand side exits. Again, from the study of Section 2.2, we may assume that we have infinitely many extremal right-hand side exits. From the stability of the left-hand side instruction, we conclude that $lmp(i)$ is the position of the rightmost 1 of the tape at $REE(i)$. Because we assume there are no left-hand side exits, there is an address a such that $lmp(i) \geq a$ for all i.

Consider the sequence of absolute lmp defined by $lmp(n_m)$ with increasing n_m. From the previous remark about the stability of the left-hand side instruction, there must be a 1 in the interval of the tape $\left[lmp(n_k), h(REE(n_k + 1))\right]$.

Now we consider $\lambda_i = h(REE(i)) - lmp(i)$. We know that $\lambda_k > 0$ for all k. Let $\lambda = \liminf_{k \to \infty} \lambda_{n_k}$. There are two cases, depending on whether $\lambda < +\infty$ or $\lambda = +\infty$. In the latter case, λ is a natural number.

When $\lambda < +\infty$, considering $REE(n_1)$, ... , $REE(n_{\lambda+1})$, we find two indices h and k such that $\lambda_h = \lambda_k$. Now, at the times j_{n_α}, with $\alpha \in \{h, k\}$, of lmp-time at $lmp(n_\alpha)$ we have the same configuration from the address $lmp(n_\alpha)$ and to its right-hand side at time j_{n_α}, also because the state at time j_{n_α} is always the same for both values of α. Indeed, this latter property follows from the fact that we have a single left-hand side instruction. Accordingly, assuming $h < k$, the same sequence of instructions between $REE(n_h)$ and $lmp(n_k)$ is endlessly repeated.

Now we consider the case when $\lambda = +\infty$. In this case we assume that λ_{n_k} is big enough. On a large interval of ones, the machine is unilateral and because we assume that it does not halt, its motion is ultimately periodic. Since the state at the time of an $lmp(j)$ is always the same, the sequence of instructions on an interval of ones is the beginning of the same infinite sequence ω which is periodic starting from a certain rank. We may assume that λ_{n_k} is big enough to contain at least one complete period of this sequence. Define $m_i = h(REE(i+1)) - h(REE(i))$ as the length of the interval of zeros traversed by the head starting from the first exit to the right after $h(REE(i))$ until the next right-hand side half-turn. If a 1 is written during the period of the motion on ones or on the m_i zeros, we have infinitely many situations when $\lambda_i \leq 2s$, because the period cannot be greater than the number of states. Accordingly, $\liminf_{k\to\infty} \lambda_{n_k} < +\infty$, a contradiction with our assumption and the 1 written after the lmp-time of $lmp(n_k)$ is written by the head in the aperiodic part of its motion on the interval of ones. By possibly taking a subsequence of the λ_{n_k}, we may assume that $\lambda_{n_k} < \lambda_{n_{k+1}}$.

Now let w be the length of the smallest period of ω and consider the times when the head reaches $lmp(n_k)$ at its lmp-time for $k \in [1\,..\,w+1]$. For two of them, say j_1 and j_2, the state under which the head reaches $h(REE(n_{j_\alpha}))$, $\alpha \in \{1, 2\}$ while coming from $lmp(n_{j_\alpha})$ after its lmp-time is the same, say **u**, and it has the same place in the period of ω. Because $\lambda_{n_{j_\alpha}} > w$, and assuming $j_1 < j_2$, we may write $\lambda_{j_2} = \lambda_{j_1} + b.w$, for some integer b.

Now, the sequence of right-hand side half-turns in the tape interval $[REE(j_1+1), REE(j_2+1)[$ is endlessly repeated. The words of the tape defined by the intervals $[lmp(j_\alpha), h(REE(j_\alpha))]$ at the time of the first right-hand side exit after $REE(j_\alpha)$ have a common prefix and a common suffix, the rest being a word of the form W^{h_α} with $h_2 = h_1 + b$.

Because this situation is recursively enumerable, it is also the case when we have $\liminf_{k\to\infty} \lambda_{n_k} = +\infty$ to complete the proof. □

Acknowledgment

The author is very much in debt to Stephen Wolfram from asking him to write this paper. He is also very much in debt to the anonymous referees whose precious remarks allowed him to improve the paper.

References

[1] A. M. Turing, "On Computable Real Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society,* **42**(2), 1936 pp. 230-265.

[2] Y. V. Rogozhin, "Sem' universal'nykh mashin T'juringa," *Matematicheskie Issledovaniya*, **69**, 1982 pp. 76-90 (Seven Universal Turing Machines) (in Russian).

[3] P. K. Hooper, "The Undecidability of the Turing Machine Immortality Problem," *Journal of Symbolic Logic*, **31**(2), 1966 pp. 219-234.

[4] M. Cook, "Universality in Elementary Cellular Automata," *Complex Systems*, **15**(1), 2004 pp. 1-40.

[5] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.

[6] A. Smith, "Wolfram's 2,3 Turing Machine Is Universal," *Complex Systems*, to appear. (Aug 12, 2010)
http://www.wolframscience.com/prizes/tm23/solved.html

[7] L. Priese, "Towards a Precise Characterization of the Complexity of Universal and Nonuniversal Turing Machines," *SIAM Journal of Computation*, **8**(4), 1979 pp. 508-523.

[8] M. Margenstern and L. Pavlotskaya, "On the Optimal Number of Instructions for Universal Turing Machines Connected with a Finite Automaton," *International Journal of Algebra and Computation*, **13**(2), 2003 pp. 133-202.

[9] M. Margenstern, "Frontier between Decidability and Undecidability: A Survey," *Theoretical Computer Science*, **231**(2), 2000 pp. 217-251.

[10] M. L. Minsky, *Computation: Finite and Infinite Machines*, Englewood Cliffs, NJ: Prentice Hall, 1967.

[11] L. M. Pavlotskaya, "Razreshimost' problemy ostanovki dlja nekotorykh klassov mashin T'juringa," *Matematicheskie Zametki*, **13**(6), 1973 pp. 899-909, (translation: "Solvability of the Halting Problem for Certain Classes of Turing Machines," *Mathematical Notes of the Academy of Sciences of the USSR*, **13**(6), Nov. 1973 pp. 537-541).

[12] M. Kudlek, "Small Deterministic Turing Machines," *Theoretical Computer Science*, **168**(2), 1996 pp. 241-255.

[13] M. Margenstern, "Non Erasing Turing Machines: A Frontier between a Decidable Halting Problem and Universality," in *Lecture Notes in Computer Science: Proceedings of the 9th International Symposium on Fundamentals of Computation Theory (FCT 1993)*, Szeged, Hungary (Zoltán Éstik, ed.), Berlin: Springer-Verlag, 1993 pp. 375-385.

[14] M. Margenstern, "The Laterality Problem for Non-Erasing Turing Machines on {0, 1} Is Completely Solved," *Informatique théorique et Applications/Theoretical Informatics and Applications,* **31**(2), 1997 pp. 159-204.

[15] M. Margenstern, "On Quasi-Unilateral Universal Turing Machines," *Theoretical Computer Science*, **257**(1-2), 2001 pp. 153-166.

Corewars

COMPUTER RECREATIONS: In the game called Core War hostile programs engage in a battle of bits.

by A. K. Dewdney

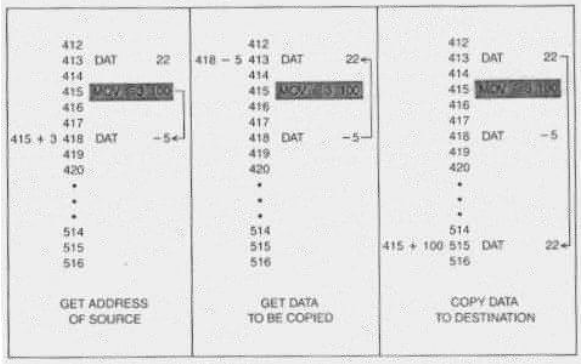
INSTRUCTION	MNEMONIC	CODE	ARGUMENTS	EXPLANATION
Move	MOV	1	A B	Move contents of address *A* to address *B*.
Add	ADD	2	A B	Add contents of address *A* to address *B*.
Subtract	SUB	3	A B	Subtract contents of address *A* from address *B*.
Jump	JMP	4	A	Transfer control to address *A*.
Jump if zero	JMZ	5	A B	Transfer control to address *A* if contents of address *B* are zero.
Jump if greater	JMG	6	A B	Transfer control to address *A* if contents of *B* are greater than zero.
Decrement, jump if zero	DJZ	7	A B	Subtract 1 from contents of address *B* and transfer control to address *A* if contents of address *B* are then zero.
Compare	CMP	8	A B	Compare contents of addresses *A* and *B*; if they are unequal, skip the next instruction.
Data statement	DAT	0	B	A nonexecutable statement; *B* is the data value.

The instruction set of Redcode, an assembly language for Core War

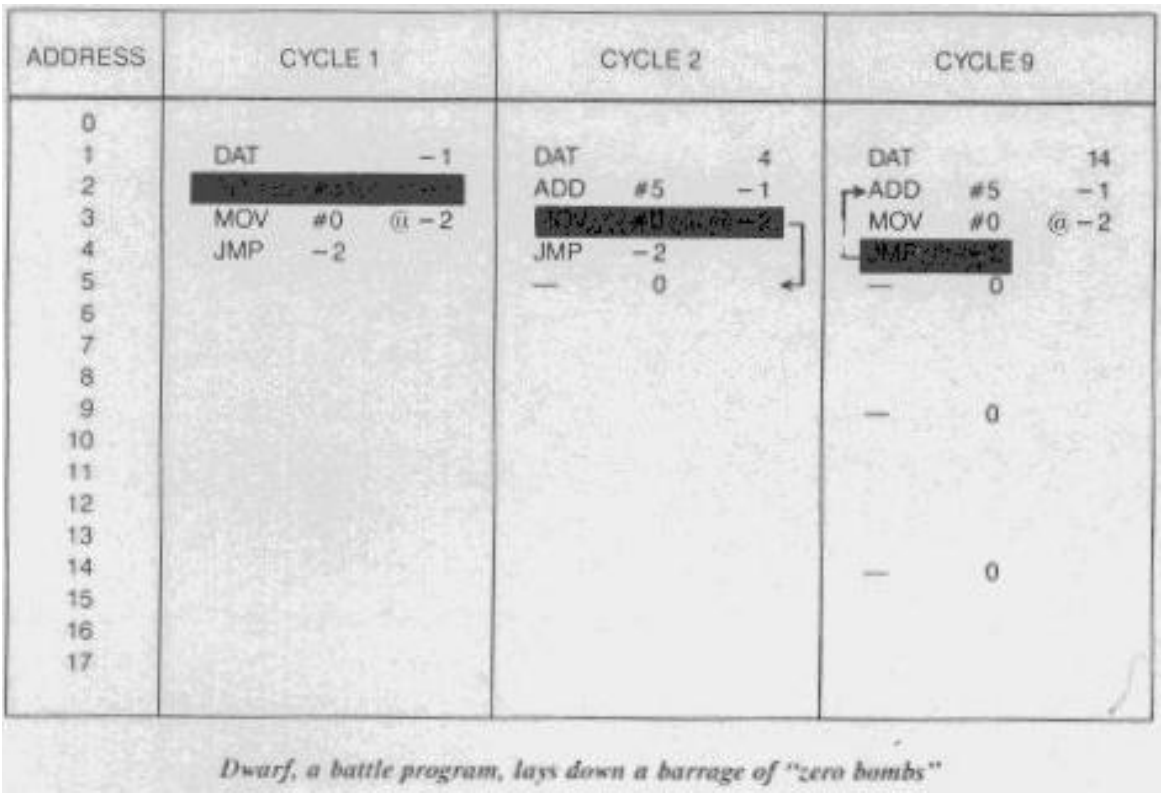
MNEMONIC	ARGUMENT A	ARGUMENT B	OPERATION CODE	MODE DIGIT ARGUMENT A	MODE DIGIT ARGUMENT B	ARGUMENT A	ARGUMENT B
DAT		−1	0	0	0	0000	7999
ADD	#5	−1	2	0	1	0005	7999
MOV	#0	@−2	1	0	2	0000	7998
JMP	−2		4	1	0	7998	0000

ADDRESSING MODES: IMMEDIATE # 0
DIRECT 1
INDIRECT @ 2

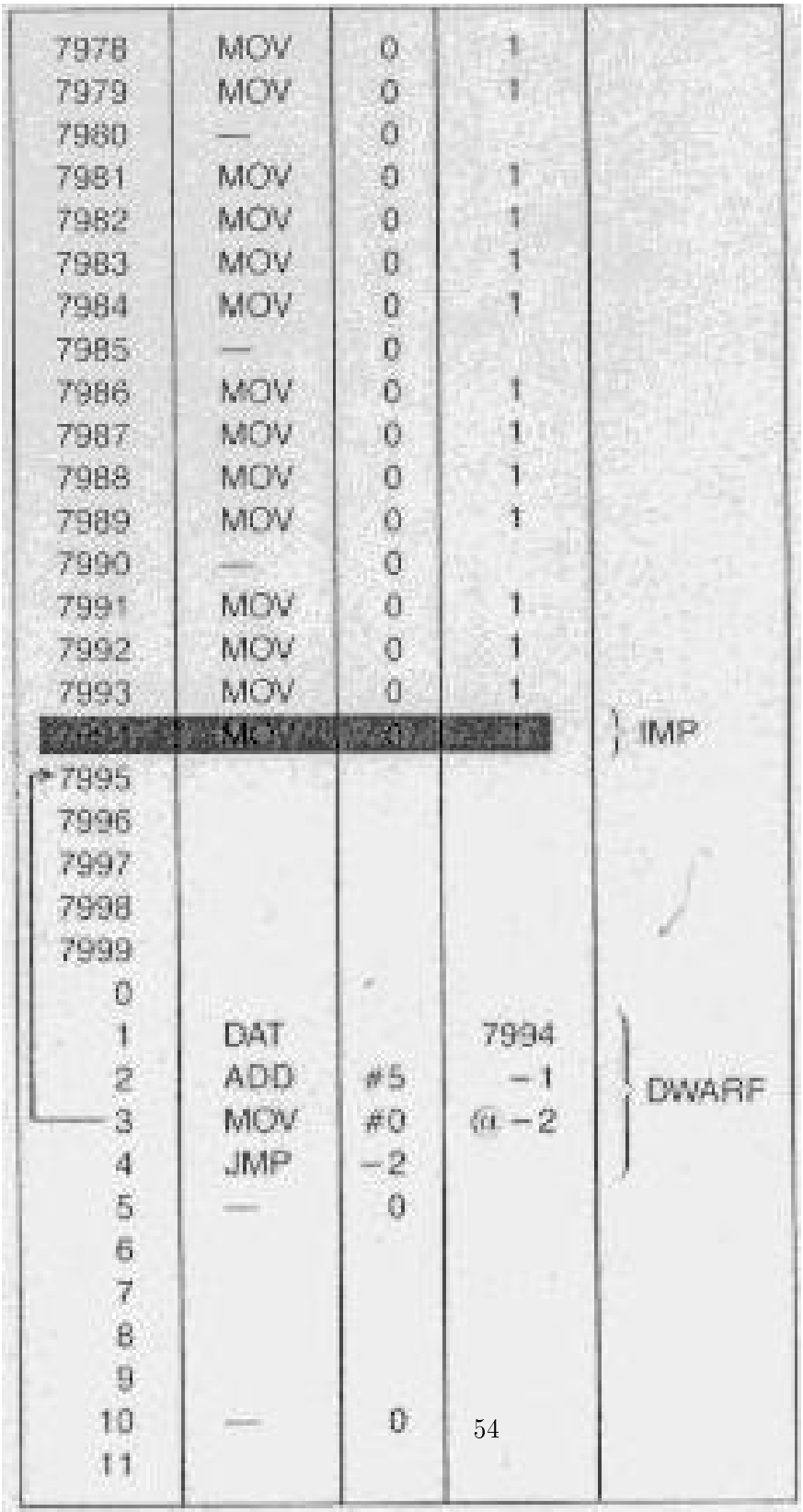
The encoding of Redcode instructions as decimal integers



The three-step mechanism of indirect relative addressing



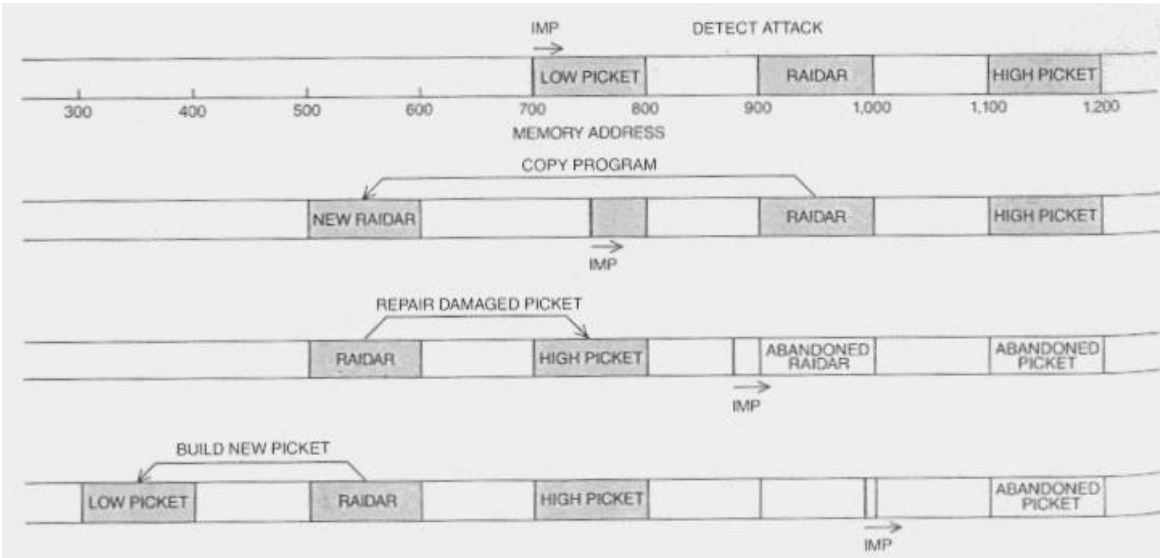
Dwarf, a battle program, lays down a barrage of "zero bombs"



Imp v. Dwarf: Who wins?

	DAT	0	/pointer to source address
	DAT	99	/pointer to destination address
LOOP	MOV	@ −2	@ −1 /copy source to destination
	CMP	−3	#9 /if all 10 lines have been copied . . .
	JMP	4	/. . . then leave the loop;
	ADD	#1	−5 /otherwise, increment the source address . . .
	ADD	#1	−5 /. . . and the destination address . . .
	JMP	−5	/. . . and return to the loop
	MOV	#99	93 /restore the starting destination address
	JMP	93	/jump to the new copy

Gemini, a program that copies itself to a new position in the memory array



Raidar, a sophisticated battle program, eludes the simpler Imp in the memory array of Core War

Two computer programs in their native habitat – the memory chips of a digital computer – stalk each other from address to address. Sometimes they go scouting for the enemy; sometimes they lay down a barrage of numeric bombs; sometimes they copy themselves out of danger or stop to repair damage. This is the game I call Core War. It is unlike almost all other computer games in that people do not play at all! The contending programs are written by people, of course, but once a battle is under way the creator of a program can do nothing but watch helplessly as the product of hours spent in design and implementation either lives or dies on the screen. The outcome depends entirely on which program is hit first in a vulnerable area.

The term Core War originates in an outmoded memory technology. In the 1950's and

1960's the memory system of a computer was built out of thousands of ferromagnetic cores, or rings, strung on a meshwork of fine wires. Each core could retain the value of one bit, or binary digit, the fundamental unit of information. Nowadays memory elements are fabricated on semiconductor chips, but the active part of the memory system where a program is kept while it is being executed, is still often referred to as core memory, or simply core.

Battle programs in Core War are written in a specialized language I have named Redcode, closely related to the class of programming languages called assembly languages. Most computer programs today are written in a high-level language such as Pascal, Fortran or BASIC; in these languages a single statement can specify an entire sequence of machine instructions. Moreover, the statements are easy for the programmer to read and to understand. For a program to be executed, however, it must first be translated into "machine language," where each instruction is represented by a long string of binary digits. Writing a program in this form is tedious at best.

Assembly languages occupy an intermediate position between high-level languages and machine code. In an assembly-language program each statement generally corresponds to a single instruction and hence to a particular string of binary digits. Rather than writing the binary numbers, however, the programmer represents them by short words or abbreviations called mnemonics (because they are easier to remember than numbers). The translation into machine code is done by a program called an assembler.

Comparatively little programming is done in assembly languages because the resulting programs are longer and harder to understand or modify than their high-level counterparts. There are some tasks, however, for which an assembly language is ideal. When a program must occupy as little space as possible or be made to run as fast as possible, it is generally written in assembly language. Furthermore, some things can be done in an assembly language that are all but impossible in a high-level language. For example, an assembly-language program can be made to modify its own instructions or to move itself to a new position in memory.

Core War was inspired by a story I heard some years ago about a mischievous programmer at a large corporate research laboratory I shall designate X. The programmer wrote an assembly-language program called Creeper that would duplicate itself every time it was run. It could also spread from one computer to another in the network of the X corporation. The program had no function other than to perpetuate itself. Before long there were so many copies of Creeper that more useful programs and data were being crowded out. The growing infestation was not brought under control until someone thought of fighting fire with fire. A second self-duplicating program called Reaper was written. Its purpose was to destroy copies of Creeper until it could find no more and then to destroy itself. Reaper did its job, and things were soon back to normal at the X lab.

In spite of fairly obvious holes in the story, I believed it, perhaps because I wanted to. It took some time to track down the real events that lay behind this item of folklore. (I

shall give an account of them below.) For now it is sufficient to note that my desire to believe rested squarely on the intriguing idea of two programs doing battle in the dark and noiseless corridors of core.

Last year I decided that even if the story turned out not to be true, something like it could be made to happen. I set up an initial version of Core War and, assisted by David Jones, a student in my department at the University of Western Ontario, got it working. Since then we have developed the game to a fairly interesting level.

Core War has four main components: a memory array of 8,000 addresses, the assembly language Redcode, an executive program called MARS (an acronym for Memory Array Redcode Simulator) and the set of contending battle programs. Two battle programs are entered into the memory array at randomly chosen positions; neither program knows where the other one is. MARS executes the programs in a simple version of time-sharing, a technique for allocation the resources of a computer among numerous users. The two programs take turns: a single instruction of the first program is executed, then a single instruction of the second, and so on.

What a battle program does during the execution cycles allotted to it is entirely up to the programmer. The aim, of course, is to destroy the other program by ruining its instructions. A defensive strategy is also possible: a program might undertake to repair any damage it has received or to move out of the way when it comes under attack. The battle ends when MARS comes to an instruction in one of the programs that cannot be executed. The program with the faulty instruction – which presumably is a casualty of war – is declared the loser.

Much can be learned about a battle program merely by analyzing its actions mentally or with pencil and paper. To put the program to the test of experience, however, one needs access to a computer and a version of MARS. The programs could be made to operate on a personal computer, and Jones and I have prepared brief guidelines for those who would like to set up a Core War battlefield of their own. (For a copy of the guidelines send your name and address and $2 for postage and handling to Core War, Scientific American, 415 Madison Avenue, New York, N.Y., 10017. Delivery may take a few weeks.)

Before describing Redcode and introducing some simple battle programs, I should say more about the memory array. Although I have noted that it consists of 8,000 addresses, there is nothing magical about this number; a smaller array would work quite well. The memory array differs from most computer memories in its circular configuration; it is a sequence of addresses numbered from 0 to 7999 but it thereupon rejoins itself, so that address 8000 is equivalent to address 0. MARS always reduces an address greater than 7999 by taking the remainder after division by

1. Thus if a battle program orders a hit at address 9378, MARS

interprets the address as 1378.

Redcode is a simplified, special-purpose assembly-style language. It has instructions

to move the contents of one address in memory to another address, to alter the contents arithmetically and to transfer control forward or backward within a program. Whereas the output of a real assembler consists of binary codes, the mnemonic form of a Redcode instruction is translated by MARS into a large decimal integer, which is then stored in the memory array; each address in the array can hold one such integer. It is also MARS that interprets the integers as instructions and carries out the indicated operations.

A list of the elementary Redcode instructions is given in the top illustration on page 19. With each instruction the programmer is required to supply at least one argument, or value, and most of the instructions take two arguments. For example, in the instruction JMP -7 the mnemonic JMP (for "jump") is followed by the single argument -7. The instruction tells MARS to transfer control to the memory address seven places before the current one, that is, seven places before the JMP -7 instruction itself. If the instruction happened to be at address 3715, execution of the program would jump back to address 3708.

This method of calculating a position in memory is called relative addressing, and it is the only method employed in Redcode. There is no way for a battle program to know its own absolute position in the memory array.

The instruction MOV 3 100 tells MARS to go forward three addresses, copy what it finds there and deliver it 100 addresses beyond the MOV instruction, overwriting whatever was there. The arguments in this instruction are given in "direct" mode, meaning they are to be interpreted as addresses to be acted on directly. Two other modes are allowed. Preceding an argument with an @ sign makes it "indirect." In the instruction MOV @3 100 the integer to be delivered to relative address 100 is not the one found at relative address 3 but rather the one found at the address specified by the contents of relative address

1. (The bottom illustration on page 19 gives more detail on the

process of indirect addressing.) A # sign makes an argument "immediate," so that it is treated not as an address but as an integer. The instruction MOV #3 100 causes the integer 3 to be moved to relative address 100.

Most of the other instructions need no further explanation, but the data statement (DAT) requires some comment. It can serve as a work space to hold information a program may need to refer to. Strictly speaking, however, it is not an instruction; indeed, any memory location with a zero in its first decimal position can be regarded as a DAT statement and as such is not executable. If MARS should be asked to execute such an "instruction," it will not be able to and will declare that program the loser.

The decimal integer that encodes a Redcode instruction has several fields, or functional areas [see middle illustration on page 19]. The first digit represents the mnemonic itself, and two more digits identify the addressing mode (direct, indirect or immediate). In addition four digits are set aside for each argument. Negative arguments are stored in complement form: -1 would be represented as 7999, since in the circular memory array adding 7999 has the same effect as subtracting 1.

The instructions making up a simple battle program called Dwarf are listed in the illustration on page 20. Dwarf is a very stupid but very dangerous program that works its way through the memory array bombarding every fifth address with a zero. Zero is the integer signifying a nonexecutable data statement, and so a zero dropped into an enemy program can bring it to a halt.

Assume that Dwarf occupies absolute addresses 1 through 4. Address 1 initially contains DAT -1, but execution begins with the next instruction. ADD #5 -1. The effect of the instruction is to add 5 to the contents of the preceding address, namely the DAT -1 statement, thereby transforming it into DAT 4. Next Dwarf executes the instruction at absolute address 3, MOV #0 @-2. Here the integer to be moved is 0, specified as an immediate value. The target address is calculated indirectly in the following way. First MARS counts back two addresses from address 3, arriving at address 1. It then examines the data value there, namely 4, and interprets it as an address relative to the current position; in other words, it counts four places forward from address 1 and hence deposits a 0 at address 5.

The final instruction in Dwarf, JMP -2, creates an endless loop. It directs execution back to absolute address 2, which again increments the DAT statement by 5, making its new value DAT 9. In the next execution cycle a 0 is therefore delivered to absolute address 10. Subsequent 0 bombs will fall on addresses 15, 20, 25 and so on. The program itself is immobile but its artillery threatens the entire array. Eventually Dwarf works its way around to addresses 7990, 7995 and then 8000. As far as MARS is concerned, 8000 is equal to 0, and so Dwarf has narrowly avoided committing suicide. Its next missile again lands on address 5.

It is sobering to realize that no stationary battle program that has more than four instructions can avoid taking a hit from Dwarf. The opposing program has only three options: to move about and thereby elude the bombardment, to absorb hits and repair the damage or to get Dwarf first. To succeed through the last strategy the program may have to be lucky: it can have no idea where Dwarf is in the memory array, and on the average it has about 1,600 execution cycles before a hit is received. If the second program is also a Dwarf, each program wins 30 percent of the time; in 40 percent of the contests neither program scores a fatal hit.

Before taking up the other two strategies, I should like to introduce a curious one-line battle program we call Imp. Here it is:

```
MOV 0 1
```

Imp is the simplest example of a Redcode program that is able to relocate itself in the memory array. It copies the contents of relative address 0 (namely MOV 0 1) to relative address 1, the next address. As the program is executed it moves through the array at a speed of one address per cycle, leaving behind a trail of MOV 0 1 instructions.

What happens if we pit Imp against Dwarf? The barrage of zeros laid down by Dwarf moves through the memory array faster than Imp moves, but it does not necessarily follow that Dwarf has the advantage. The question is: Will Dwarf hit Imp even if the barrage does catch up?

If Imp reaches Dwarf first, Imp will in all probability plow right through Dwarf's code. When Dwarf's JMP -2 instruction transfers execution back two steps, the instruction found there will be Imp's MOV 0 1. As a result Dwarf will be subverted and become a second Imp endlessly chasing the first one around the array. Under the rules of Core War the battle is a draw. (Note that this is the outcome to be expected "in all probability." Readers are invited to analyze other possibilities and perhaps discover the bizarre result of one of them.)

Both Imp and Dwarf represent a class of programs that can be characterized as small and aggressive but not intelligent. At the next level are programs that are larger and somewhat less aggressive but smart enough to deal with programs in the lower class. The smarter programs have the ability to dodge an attack by copying themselves out of trouble. Each such program includes a segment of code somewhat like the one named Gemini, shown in the lower illustration on page 22. Gemini is not intended to be a complete battle program. Its only function is to make a copy of itself 100 addresses beyond its present position and then transfer execution to the new copy.

The Gemini program has three main parts. Two data statements at the beginning serve as pointers; they indicate the next instruction to be copied and its destination. A loop in the middle of the program does the actual copying, moving each instruction in turn to an address 100 places beyond its current position. On each transit through the loop both pointers are incremented by 1, thereby designating a new source and destination address. A compare instruction (CMP) within the loop tests the value of the first data statement; when it has been incremented nine times, the entire program has been copied, and so an exit from the loop is taken. One final adjustment remains to be made. The destination address is the second statement in the program and it has an initial value of DAT 99; by the time it is copied, however, it has already been incremented once, so that in the new version of the program it reads DAT

1. This transcription error is corrected (by the instruction MOV #99

2. and then execution is transferred to the new copy.

By modifying Gemini it is possible to create an entire class of battle programs. One of these, Juggernaut, copies itself 10 locations ahead instead of 100. Like Imp, it tries to roll through all its opposition. It wins far more often than Imp, however, and leads to fewer draws, because an overwritten program is less likely to be able to execute fragments of Juggernaut's code. Bigfoot, another program employing the Gemini mechanism, makes the interval between copies a large prime number. Bigfoot is hard to catch and has the same devastating effect on enemy code as Juggernaut does.

Neither Bigfoot nor Juggernaut is very intelligent. So far we have written only two battle programs that qualify for the second level of sophistication. They are too long to reproduce here. One of them, which we call Raidar, maintains two "pickets" surrounding the program itself [see illustration on page 18]. Each picket consists of 100 consecutive addresses filled with 1's and is separated from the program by a buffer zone of 100 empty addresses. Raidar divides its time between systematically attacking distant areas of the memory array and checking its picket addresses. If one of the pickets is found to be altered, Raidar interprets the change as evidence of an attack by Dwarf, Imp or some other unintelligent program. Raidar than copies itself to the other side of the damaged picket, restores it, constructs a new picket on its unprotected side and resumes normal operation.

In addition to copying itself a battle program can be given the ability to repair itself. Jones has written a self-repairing program that can survive some attacks, although not all of them. Called Scanner, it maintains two copies of itself but ordinarily executes only one of them. The copy that is currently running periodically scans the other copy to see if any of its instructions have been altered by an attack. Changes are detected by comparing the two copies, always assuming that the executing copy is correct. If any bad instructions are found, they are replaced and control is transferred to the other copy, which then begins to scan the first one.

So far Scanner remains a purely defensive program. It is able to survive attacks by Dwarf, Imp, Juggernaut and similar slow-moving aggressors – at least if the attack comes from the right direction. Jones is currently working on a self-repair program that keeps three copies of itself.

I am curious to see whether readers can design other kinds of self-repairing programs. For example, one might think about maintaing two or more copies of a program even though only one copy is ever executed. The program might include a repair section that would refer to an extra copy when restoring damaged instructions. The repair section could even repair itself, but it might still be vulnerable to damage at some positions. One measure of vulnerability assumes that a single instruction has been hit; on the average, how many such instruction, if they are hit, ultimately cause the program to die? By this measure, what is the least vulnerable self-repairing program that can be written?

Only if reasonably robust programs can be developed will Core War reach the level of an exciting game, where the emphasis is shifted from defense to offense. Battle programs will then have to seek out and identify enemy code and mount an intensive attack wherever it is found.

I may have given the impression that Redcode and the entire MARS system are fixed. They are not. In spare moments we have been experimenting with new ideas and are certainly open to suggestions. Indeed, we have been experimenting so much with new programs and new features that some battles remain to be fought in our own system.

One idea we have been playing with is to include an extra instruction that would make self-repair or self-protection a little easier. The instruction PCT A would protect the

instruction at address A from alteration until it is next executed. How much could the vulnerability of a program be reduced by exploiting an instruction of this kind?

In the guidelines offered above we describe not only the rules of Core War but also how to set up a memory array and write a MARS system in various high-level languages. We also suggest how to display the results of a Core War battle. For now the following rules define the game with enough precision to enable pencil-and-paper players to begin designing programs:

1. The two battle programs are loaded into the memory array at random

positions but initally are no closer than 1,000 addresses.

1. MARS alternates in executing one instruction from each program until

it reaches an instruction that cannot be executed. The program with the erroneous instruction loses.

1. Programs can be attacked with any available weapon. A "bomb" can be a

0 or any other integer, including a valid Redcode instruction.

1. A time limit is put on each contest, determined by the speed of the

computer. If the limit is reached and both programs are still running, the contest is a draw.

The story of Creeper and Reaper seems to be based on a compounding of two actual programs. One program was a computer game called Darwin, invented by M. Douglas McIlroy of AT&T Bell Laboratories. The other was called Worm and was written by John F. Shoch of the Xerox Palo Alto Research Center. Both programs are some years old, allowing ample time for rumors to blossom. (Darwin was described in Software: Practice and Experience, Volume 2, pages 93-96, 1972. A vague description of what appears to be the same game is also given in the 1978 edition of Computer Lib.)

In Darwin each player submits a number of assembly-language programs called organisms, which inhabit core memory along with the organisms of other players. The organisms created by one player (and thus belonging to the same "species") attempt to kill those of other species and occupy their space. The winner of the game is the player whose organisms are most abundant when time is called. McIlroy invented an unkillable organism, although it won only "a few games." It was immortal but apparently not very aggressive.

Worm was an experimental program designed to make the fullest use possible of minicomputers linked in a network at Xerox. Worm was loaded into quiescent machines by a supervisory program. Its purpose was to take control of the machine and, in coordination

with Worms inhabiting other quiescent machines, run large applications programs in the resulting multiprocessor system. Worm was designed so that anyone who wanted to use one of the occupied machines could readily reclaim it without interfering with the larger job.

One can see elements of both Darwin and Worm in the story of Creeper and Reaper. In Core War, Reaper has becom reality.

A Core War bestiary of viruses, worms and other threats to computer memories

by A. K. Dewdney

```
1   IF PEEK (104) = 134 GOTO 10
2   POKE 104, 134: POKE 134 * 256,0
3   PRINT CHR$(4) "RUN APPLE WORM"
10  HOME : POKE - 16302,0: POKE - 16304,0: POKE 1023,160
20  FOR I = 0 TO 94: READ D: POKE 1024 + I, D: NEXT I
30  POKE - 16368,0
40  IF PEEK ( - 16384)<128 GOTO 40
50  CALL 1024
100 DATA 160,225,200,185,255,3,153,127,4,192,95,208,245,
    160,18,190,76,4,24,189,128,4,105,128,157,128,4,189,129,
    4,105,0,157,129,4,192,13,208,18,238,23,4,173,23,4
200 DATA 141,151,4,206,31,4,173,31,4,141,159,4,136,208,211,
    173,167,4,72,173,176,4,141,167,4,104,141,176,4,76,128,
    4,7,20,25,28,33,46,55,61,65,68,72,75,4,16,40,43,49,52
```

A worm that inhabits Apples

```
DAT                    0       /pointer to address being protected
ADD     #1            —1       /increment protection address
PCT     @—2                    /protect the address
CMP     #102          —3       /if all 102 instructions protected...
JMP     2                      /...then leave loop
JMP     —4                     /otherwise re-enter it.
                .
                .
                .
        BODY OF MAIN BATTLE PROGRAM
                .
                .
                .
```

This loop protects combatants from stray bombs

INSTRUCTION	MNEMONIC	CODE	ARGUMENTS		EXPLANATION
Move	MOV	1	A	B	Move contents of address *A* to address *B*.
Add	ADD	2	A	B	Add contents of address *A* to address *B*.
Subtract	SUB	3	A	B	Subtract contents of address *A* from address *B*.
Jump	JMP	4	A		Transfer control to address *A*.
Jump if zero	JMZ	5	A	B	Transfer control to address *A* if contents of address *B* are zero.
Jump if greater	JMG	6	A	B	Transfer control to address *A* if contents of *B* are greater than zero.
Decrement: jump if zero	DJZ	7	A	B	Subtract 1 from contents of address *B* and transfer control to address *A* if contents of address *B* are then zero.
Compare	CMP	8	A	B	Compare contents of addresses *A* and *B*; if they are unequal, skip the next instruction.
Split	SPL	9	A		Split execution into next instruction and the instruction at *A*.
Data statement	DAT	0		B	A nonexecutable statement; *B* is the data value

The list of instructions for Core War programs

When the column about Core War appeared last May, it had not occurred to me how serious a topic I was raising. My descriptions of machine-language programs, moving about in memory and trying to destroy each other, struck a resonant chord. According to many readers, whose stories I shall tell, there are abundant examples of worms, viruses and other software creatures living in every conceivable computing environment. Some of the possibilities are so horrifying that I hesitate to set them down at all.

The French spy thriller *Softwar: La Guerre Douce* (English translation to be published by Holt, Rinehart & Windston) provides a geopolitical fantasy of this type. Authors Thierry Breton and Denis Beheich spin a chilling yarn about the purchase by the Soviet Union of an American supercomputer. Instead of blocking the sale, American authorities, displaying studied reluctance, agree to the transaction. The computer has been secretly programmed with a "software bomb." Ostensibly bought to help with weather forecasting over the vast territory of the Soviet Union, the machine, or rather its software, contains a hidden trigger; as soon as the U.S. National Weather Service reports a certain temperature at St. Thomas in the Virgin Islands, the program proceeds to subvert and destroy every piece of software it can find in the Soviet network. To the extent that such scenarios represent real possibilities, I am tempted to say, "If we must have war, by all means let it be soft." On the other hand, the possibility of an accident due to the intimate connection between military software and weapons-control systems gives me pause.

Before I describe the experiences of various readers with hostile programs it would be worthwhile to summarize Core War for those who missed the May 1984 column:

Two players write one program each in a low-level language called REDCODE. The programs are placed in a vast, circular arena called Core. In reality Core is simply an array of several thousand locations whose last address is contiguous to the first. Each battle-program instruction occupies one location in Core. A Memory Array Redcode Simulator executive program (MARS for short) runs the battle programs by alternately executing one instruction of each, in the manner of a simple time-sharing system: the two programs attack each other and seek in turn to avoid damage or to repair it. A simple mode of attack can be executed by means of MOV instructions. For example,

```
MOV #0 1000
```

causes the number 0 to be placed in the location whose address lies 1,000 locations beyond this instruction. The previous contents of that location are thereby erased. If the 0 were placed on top of an enemy instruction, it too would be wiped out and the program would no longer be executable. The enemy would lose the game.

Since no computer, whether personal or mainframe, comes equipped with REDCODE and a convenient battle array, such features must be simulated. Guidelines for writing a simulation program were and still are available from the offices of *Scientific American* at a cost of $2 to cover postage and handling. Please address your request to Core War, *Scientific*

American, 415 Madison Avenue, New York, N.Y. 10017. Last year several hundred readers obtained the guidelines and a large percentage of them wrote Core War game programs.

Inspired by a June 1959 *Scientific American* article on self-reproducing mechanisms by L. S. Penrose, Frederick G. Stahl of Chesterfield, Mo., created a miniature linear universe in which humble creatures lived, moved and (after a fashion) lived out their destinies. Stahl writes:

"Like Core War, I set aside a closed, linear segment of main memory in which a creature was simulated by modified machine language. The machine was an IBM Type 650 with drum memory. The creature was programmed to crawl through its universe eating food (nonzero words) and creating a duplicate of itself when enough food was accumulated. Like Core War, I had an executive program which kept track of who was alive and allocated execution time among the living creatures. I called it the 'Left Hand of God.'" Stahl goes on to discuss his program's ability to reproduce. He also describes an interesting mutation mechanism; a program being copied might experience a small number of random changes in its code. However, Stahl reports, "I abandoned this line of work after one production run in which a sterile mutant ate and killed the only fertile creature in the universe. It was apparent that extraordinarily large memories and long computer runs would be needed to achieve any interesting results."

A similar story concerns a game called Animal in which a program tries to determine what animal a human is thinking of by playing a form of Twenty Questions. David D. Clark of the Massachusetts Institute of Technology Laboratory for Computer Science writes that the employees of a certain company devotedly played Animal. While it resembles neither a battle program nor even Stahl's simple creatures, Animal achieved the ability to reproduce itself in the corridors of core through the efforts of a programmer to enhance a key feature of the game: when the program guesses incorrectly what animal the human has in mind, it asks the human to suggest a question it might ask to improve its future performance. This feature, Clark continues, led the programmer to invent a certain trick for making sure that everyone always had the same version of Animal.

"On a very early computer system, which lacked any shared directory structure, but also lacked any protection tools, a programmer invented a very novel way of making the game available to several users. A version of the game existed in one user's directory. Whenever he played the game, the program made a copy of itself into another directory. If that directory had previously contained a copy of the game, then the old version was overwritten, which made the behavior of the game change unexpectedly to the player. If that directory had previously had no version of Animal, the game had been offered to yet another user."

Clark recalls that Animal was such a popular game that eventually every directory in the company system contained a copy. "Furthermore, as employees of the company were transferred to other divisions...they took Animal as well, and thus it spread from machine to machine within the company." The situation would never have become serious had it

not been for the fact that all those copies of this otherwise innocuous game began to clog the disk memory. Only when someone devised a more "virulent" version of the game was the situation brought under control. When the new version of Animal was played, it copied itself into other directories not once but twice. Given enough time, it was thought, this program would eventaully overwrite all the old versions of Animal. After a year had passed, a certain date triggered each copy of the new Animal program. "Instead of replicating itself twice whenever it was invoked, it now played one final game, wished the user 'goodbye' and then deleted itself. And thus Animal was purged from the system."

Ruth Lewart of Homdel, N.J., once created a monster (of sorts) without even writing a program. Working on her company's time-sharing system, she was preparing a demonstration version of a teaching program when she decided to make a backup copy on another time-sharing system. When the original system began to seem sluggish, she "switched to the backup system, which was very responsive – for all of three minutes, by which time there was no response and utter chaos on the screen of my graphics terminal. It was not possible for any user to log on or to log off from the system. The conclusion was inescapable – my program was somehow at fault! Despite my panic, I suddenly realized that I had specified an ampersand as the terminal's field separator character. But the ampersand was also the character used by the computer system to spawn a background process! The first time the computer read from the screen, it must have intercepted the ampersands meant for the terminal, and spawned a number of processes, which in turn each spawned more processes, ad infinitum." A frantic long-distance call informed a system administrator of the source of the disease and the mainframe computer was then shut down and restarted. Needless to say, Lewart changed the ampersand to a less dangerous character. Her program "has been humming happily ever since."

Even though Core War programs are not spawned in this manner, additional copies can enhance their survival. Several readers suggested three copies of a program be made so that the copy currently executing could use the other two copies to determine whether any of its instructions were wrong. The executing program could even replace a faulty instruction with a viable one. A similar idea lies behind Scavenger, a program designed to protect mass-storage files from error when backup copies are made on magnetic tape. Arthur Hudson, who lives in Newton, Mass. (and works for yet another unnamed company), writes: "Anyone who used much magnetic tape found himself beset by an alien force called the Law of Joint Probability." Hudson goes on to cite various errors connected with the handling of tapes and shows that, although each kind of error has a relatively small chance of happening, the probability or at least one of them occurring is uncomfortably large. He continues:

"Fear not, Scavenger is with you; If you place a mass-storage file in its care, it will copy the file on three magnetic tapes without bothering you with housekeeping details. Even if the computer crashes logically (as it did several times per day), the run backlog usually will not be destroyed; when the computer comes back up, whatever Scavenger worms are

in the backlog will run in their turn. Each tape is written by a separate run scheduled from a master runstream."

Owners of Apple computers should beware a mean little program called Apple Worm, created by Jim Hauser and William R. Buckley of California Polytechnic State University at San Luis Obispo. Written for the Apple II in 6502 assembly language, this species of worm replicates itself on a merry little journey through the host Apple. Initially one loads a special BASIC program [/see illustration on preceding page /] that in turn loads the worm into low memory (the part with low addresses). The BASIC program, on the other hand, occupies high memory.

"Because the Worm is loaded into one of the graphics areas of the machine, you can watch the Worm as it begins it headlong (actually, tail-long) dash into high memory.... After the Worm leaves the graphics window... you can wait until the Worm erases all of high memory (including the BASIC program) and crashes into the system ROMS."

Hauser and Buckley plan to publish a collection of worms in the not too distant future. They have designed a Worm Operating System and have even written a video game with Worm as one of its main characters.

Another software threat has been propounded by Roberto Cerruti and Marco Morocutti of Brescia, Italy. Inspired by the translation of the column on Core War in the Italian edition of *Scientific American, Le Scienze*, the two sought a way of infecting the Apple II computer, not with a worm but with a virus. Reports Erruti:

"Marco thought to write a program capable of passing from one computer to the other, like a virus, and 'infecting' in this way other Apples. But we were not able to conceive it unil I realized that the program had to 'infect' floppy disks, and use the computers only like a media from a disk to the other. So our virus began to take shape.

"As you know every Apple diskette contains a copy of the Disk Operating System, which is bootstrapped by the computer at power on. The virus was an alteration in this DOS, which at every write operation checked his presence on the disk and, if not, would modify in the same way the DOS on the disk, thus copying itself on every diskette put in the drive after the first power up. We thought that installing such a DOS on some disks used in the biggest computer shop in out city, Brescia, would cause an epidemic to spread in the city.

"But was it a real epidemic, of such unharming viruses? No, our virus should be malignant! So we decided that after 16 self-reproduction cycles, counted on the disk itself, the program should decide to re-initialize the disk immediately after bootstrap. Now the awful evil of our idea was clear, and we decided neither to carry it out, nor to speak to anybody about our idea."

That was kind of Cerruti and Morocutti. In a personal computer the disk operating system is the ultimate arbiter of the fate of programs, data and all else. In the scheme just described the infected disk operating system erases the disk whence it came and can therefore never be loaded again except from a new disk. The diseased DOS could even

cause an irritating message to be displayed periodically:

IS YOUR DISK SLIPPING? It's time you got DOS DOCTOR available on disk at a computer store near you

The viral infection just described has already happened on a small scale. Richard Skrenta, Jr., a high school student in Pittsburgh, wrote such a program. Instead of wiping disks or displaying advertisements, this form of infection caused subtle errors to appear throughout the operating system.

"All of this seems pretty juvenile now," writes Skrenta, but "Oh woe to me! I have never been able to get rid of my electronic plague. It infested all of my disks, and all of my friends' disks. It even managed to get onto my math teacher's graphing disks." Skrenta devised a program to destroy the virus, but it was never as effective as the virus itself had been.

There is a good problem implicit here and I would be both unimaginative and irresponsible for not posing it; In one page or less describe DOS DOCTOR, a program on disk that somehow stamps out such electronic epidemics. Many disks used by a personal computer contain copies of its DOS. When started up, the computer obtains its copy of the DOS from the disk. This DOS will still be in charge when other disks, also containing copies of the DOS, are run. If it is infected, the DOS currently in charge may alter the other copies of the DOS or even replace them with copies of itself. But how to counteract such virulence?

In the initial version of Core War the main challenge was to enable battle program A to protect itself from stray hits generated by battle program B. If such protection could be more or less guaranteed, then evolution of the game was to proceed to the next level, where programs would have been forced to seek each other out and develop concentrated attacks.

In an effort to guarantee such protection, I suggested in the May column the instruction

```
PCT A
```

where A is the relative address (either direct or indirect) of an instruction that is to be protected. A single attempt to change the contents of that address would be prevented by MARS, the game's supervisory system. The next attempt, however, would succeed. It seemed to me that by employing a simple loop, any battle program could protect all its own instructions from stray bombs long enough to be able to launch an undistracted probe for the other program. The illustration on this page displays such a self-protecting program in schematic form. The protection loop consists of six instructions, four of them executed at each cycle through the loop. Thus a battle program of n instructions (including the loop) would require nearly 4 x n executions to have complete protection from a single hit. This salutary shielding is hardly proof against a dwarf program that hurls two shots at each location.

There is another use of this instruction, unforeseen in the earlier Core War article. Stephen Peters of Timaru, New Zealand, and Mark A. Durham of Winston-Salem, N.C.,

independently thought of using PCT offensively. A program called TRAP-DWARF lays down a barrage of zeros in the usual way but then protects each deposit against overwriting. This means that an unwary enemy program may fall into one of these traps in the course of writing inself into a new area. The instruction meant for the location occupied by a protected zero would of course have no effect on that location. Later, when the new program's execution reaches that address, it dies because 0 is not an executable instruction. PCT may be worthy of inclusion in some future version of Core War but I shall shelve it for now in the interest of simplicity, the game designer's touchstone.

Other reader ideas varied from the two-dimensional Core array suggested by Robert Norton of Madison, Wis., to the range-limitation rule suggested by William J. Mitchell of the mathematics department at Pennsylvania State University. Norton's idea is largely self-explanatory but Mitchell's suggestion requires some elaboration. Allow each battle program to alter the contents of any location up to a distance of some fixed number of addresses. Such a rule automatically keeps DWARF from doing any damage outside this neighborhood. The rule has many other effects as well, including a strong emphasis on movement. How else can a battle program get within range of an opponent? The rule has much merit and I hope that some of the many readers with a Core War system of their own will give it the further exploration it deserves.

Norton also suggests that each side in a Core War battle be allowed more than one execution. The same idea occurred to many other readers. Indeed, I have decided to adopt the suggestion. Core War now assumes a previously lacking wide-open character.

The change is made by adding the following instruction, called a split, to the official Core War list [/see illustration on page 14/].

```
SPL A
```

When execution reaches this point, it splits into two parts, namely the instruction following SPL and the one A addresses away. Because this immediately allows each Core War player to have several programs running at once, it is necessary to define how MARS will allocate such execution. Two possibilities exist.

To illustrate them suppose one player has programs A1, A2 and A3, whereas the other player has programs B1 and B2. One alternative is to have all the first player's programs run, followed by those of the second player. The order of execution would thus be A1, A2, A3 and then B1 and B2. The cycle would repeat indefinitely. The second alternative is to have the programs of the two players alternate. In this case the sequence would be A1, B1, A2, B2, A3, B1 and so on. The two schemes are quite different in effect. The first scheme puts great emphasis on unlimited proliferation and seems thereby to limit the role of intelligence in the game. The second scheme, however, implies that the more programs either player has running, the less often each will be executed. A low of diminishing returns seems appropriate in this context and I have therefore adopted the second scheme. The purpose of the game, in any event, is to bring all enemy programs to a halt.

The new instruction is rife with creative possibilities. As an illustration of the humblest issue possible, there is a battle program called IMP GUN:

```
SPL 2
JMP -1
MOV 0 1
```

Consider what happens when execution first arrives at the top of this program. The instruction SPL 2 means there will be two executions allotted to this program later: both JMP -1 and MOV 0 1 will be performed. The first instruction causes the program to recycle and the second sets an IMP in motion. The IMP will move down, of course, since the target of the MOV command will always be the next address, as indicated by they (positive) 1. The IMP's run pattering through Core bent on the destruction or subversion of hostile programs. At first glance it may seem that no defense is possible against such an army of IMP's, but in fact one is. Enter IMP PIT, an even simpler program set in motion by an SPL command in some larger assembly of instructions wishing to protect its upper flank:

```
MOV #0 -1
JMP -1
```

At each execution, IMP PIT places a zero just above itself in the hope of zapping an oncoming IMP. Here the execution-allocation rule is critical. If IMP GUN belongs to A and IMP PIT belongs to B, then A needs n turns to execute n IMP's; only one IMP can arrive at the location just above the IMP PIT. Other things being equal, B has to execute IMP PIT only once to terminate an arriving IMP.

In the expanded Core War game, one imagines each side generating and deploying small armies of programs individually shaped to detect, attack, protect and even repair. Many subtleties such as the one suggested by John McLean of Washington, D.C., await further investigation. McLean imagines a specialized trap program that places JMP commands at various addresses throughout the Core array in the hope of landing a JMP command inside an enemy program. Each JMP so placed would transfer execution of the enemy program to the trap program, causing it to go over to the enemy, so to speak.

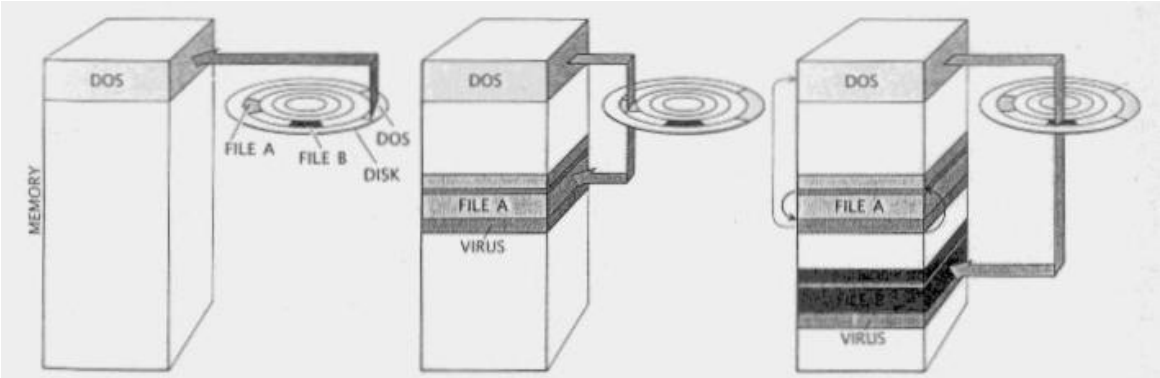
One major problem in need of resolution emerges from the melee of battle programs. What is to prevent a battle program for one side from attacking its colleagues? A recognition system appears to be necessary.

Among the many readers who have constructed Core War systems three deserve special mention: Chan Godfrey of Wilton, Conn., Graeme R. McRae of Monmouth Junction, N.J., and Make Rosing of Littleton, Colo., have taken special care in defining and documenting their projects. I should particularly like to make Rosing's documents available to readers, but there is another and better idea that includes this possibility and solves other communication problems as well. If any reader with a Core War system already running

will volunteer to act as the director of a Core War network, then documentation of various systems, rule suggestions, interesting programs and battles can be communicated to all participating Core War users. One volunteer will be chosen as director; the remaining volunteers might help with potential functions such as a newsletter, rules committee and so on, according to interest. In a future article I shall give the name and address of the network director.

Of worms, viruses and Core War

by A. K. Dewdney



Loading the DOS *into memory* (left) *allows a program to be read* (middle) *and a virus* (right) *to replicate itself*

INSTRUCTION	EXPLANATION
DAT B	A nonexecutable data statement; B is the data value.
MOV A B	Move contents of address A to address B.
ADD A B	Add contents of address A to address B.
SUB A B	Subtract contents of address A from address B.
JMP B	Transfer control to address B.
JMZ A B	Transfer control to address A if contents of address B are zero.
JMN A B	Transfer control to address A if contents of address B are not zero.
DJN A B	Subtract 1 from contents of address B and transfer control to address A if contents of address B are not zero.
CMP A B	Compare contents of addresses A and B; if they are equal, skip the next instruction.
SPL B	Split execution between next instruction and the instruction at address B.

A summary of Core War Instructions

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards – and even then I have my doubts." –Eugene H. Spafford

The knock on the door had a palpable urgency that brought the computer-center director's head up sharply from the pile of papers before him. He grunted loudly and the computer operator entered.

"Something's gone wrong. We have some very weird processes going on. We're running out of memory. I think we've got a virus in the system."

If the center had been equipped with claxons, the director would undoubtedly have set them off.

Such a scene has been played out in one form or another all too often in recent years, and

as a result computer viruses have been increasingly in the news. In fact, this department has been cited more than once in connection with the rash of virus outbreaks, probably because it is an instigator of Core War, a game in which computer programs are purposely designed to destroy one another. But, as we shall see, Core War has no direct connection with the infections.

To understand how a computer virus works one must first understand in great detail the system in which it operates. The same thing applies for understanding the operation of worms, logic bombs and other threats to computer security. This simple observation has two immediate implications. First, journalists are likely to misreport or distort virus news stories for quite innocent reasons: most reporters are more or less mystified by the internal workings of computers. Second, public descriptions of a computer virus – even fairly detailed ones – cannot be exploited to reconstruct the virus except by someone who has the requisite knowledge of the affected computer system to begin with. A knowledgeable "technopath" who is bent on destroying other people's programs and data hardly needs to read a magazine or newspaper article to begin imagining ways to construct a virus. There is consequently no harm in describing how viruses and other destructive programs work. (Indeed, such a description is probably constructive in that it might stimulate efforts to protect computer systems.)

One must distinguish from the start between the two commonest types of malignant programs. A virus rides piggyback on other programs; it cannot exist by itself. A worm, on the other hand, can pursue an independent existence, more in the manner of a bacterium. Both kinds of "infection," like all programs, depend on an operating system.

Most readers know that a running computer consists of both hardware and software. In front of me at the moment, for example, is a piece of hardware: an Apple IIc. Inside the machine's memory is software: a program called the Appleworks Word Processor. The program transfers the characters I type on the keyboard into a section of memory reserved by the program for text.

But the word-processing program is not able to run by itself. The program depends on an operating system that, among other things, translates it into a special machine language that enables the hardware to carry out the program's instructions. The operating system for a personal computer normally resides on a disk. To do anything on such a machine (from writing to playing games), the disk operating system (DOS) must first be loaded into the computer's hardware memory. In a home computer the DOS is usually loaded automatically from a disk, which may or may not contain the program one wants to run, as soon as one switches on the computer.

To run a particular program on my personal computer, I must type the name of the program into the computer. The computer's DOS then searches through the disk for a program with that name, loads it into memory and runs it – instruction by instruction – as is shown in the middle illustration on the opposite page.

In loading the program the DOS sets aside part of the hardware memory not only for

the program but also for the program's "work space." Here the program will store all the values assigned to its variables and its arrays, for example. In doing all of this the DOS is normally careful not to overwrite other programs or data areas, including whatever part of the DOS happens to be in memory. The DOS is equally careful in storing programs or data onto a disk.

Often a programmer may find it necessary to employ the commands the DOS itself uses, which can generally be found in the appropriate manual. Such commands make it possible to write a subprogram that can read files from disk into memory, alter the files and then write them back onto the disk – sometimes with malicious intent.

Here is a sample virus subprogram that does just that. It contains a mixture of pseudo-DOS commands and subroutines: small, internal programs (whose component instructions are kept separate from the subprogram's main body) that carry out specific missions whenever they are called.

```
/this/ := findfile
LOAD /(this)/
/loc/ := search /(this)/
insert /(loc)/
STORE /(this)/
```

The subroutine designated findfile opens the directory of executable files, or programs, on a disk, picks a random file name and assigns the name of that file to a variable called *this*. The next line of the program makes use of the pseudo-DOS command LOAD to copy the file into the computer's memory. Another subroutine called search then scans the program just loaded, looking for an instruction in it that can serve as a suitable insertion site for a virus. When it finds such an instruction, it determines the instruction's line number and assigns its value to the variable called *loc*.

At this point the virus subprogram is ready to infect the program it has randomly picked. The subroutine insert replaces the selected instruction with another instruction (such as a call for a subroutine) that transfers execution to a block of code containing the basic virus subprogram, which is appended to the end of the program. It then adds the program's original instruction to the end of the appended subprogram followed by a command that transfers execution to the instruction following the insert in the host program.

In this way when the virus subprogram is executed, it also executes its host program's missing instruction. The execution of the original program can then proceed as though nothing unusual had occurred. But in fact the virus subprogram has momentarily usurped control of the DOS to replicate itself into another program on the disk. The process is illustrated graphically at the right in the illustration below. When the newly infected program is subsequently loaded by the DOS into the computer's memory and run, it will in turn infect another program on the disk while appearing to run normally.

As early as 1984 Fred S. Cohen carried out controlled infection experiments at the University of Southern California that revealed – to his surprise – that viruses similar to the one I have just described could infect an entire network of computers in a matter of minutes. In order to explain the kinds of damage such viruses can do, I shall adapt Cohen's generic virus, writing it in a pseudolanguage.

1234567 main program:

1. infect
2. if trigger pulled, then do damage
3. go to host program

subroutine: infect

1. get random executable file
2. if first line of file = 1234567, then go to 1, else prepend virus to file

subroutine: trigger pulled subroutine: do damage

Cohen's generic virus is generic in all but its attachment site: instead of inserting itself in the middle or at the end of the host program, it attaches itself to the beginning. The first line of the virus program is the "recognition code" 1234567. The main program first calls up the subroutine infect, which randomly retrieves an executable file from a disk and checks whether the first line of that file happens to be

1. If it is, the program has already been infected and the

subroutine picks another program. If the subroutine happens to find an uninfected file, it "prepends" the entire virus program to the target program. This means simply that it places itself at the head of the program and arranges to get itself executed first before transferring control back to the infected program.

The next two subroutines call for a triggering condition and for some damage to be done. The triggering condition might be a certain date reached by the system clock, or perhaps the deletion of a certain employee's name from the payroll file. The damage to be done might be the erasure of all files or, more subtly, the random alteration of bits in just a few places. The possibilities are endless.

Triggering conditions and damage bring us to the edge of moral territory. I think there is no question that willfully perpetrating damage to computer files by means of a virus or other destructive program is a reprehensible act. Computer programs and data are not just impersonal strings of 0's and 1's but also miniature cities of thought. Why would anyone want to destroy the work of another?

Writers of virus programs do so for a variety of reasons. For example, a programmer in a large company might secretly harbor a grudge against the company's management. He might implant a virus for the day his employment is terminated. When that happens, his records will be deleted from the payroll file, triggering the virus. As a result, valuable company data and programs might either disappear or develop serious and costly errors. Replacing faulty data and programs with backups stored on other media might be of no avail, since if the backups were made recently, they too might be infected.

Of course, the kind of destruction just described would ordinarily take place in a multiple-user computer system. The operating system in this kind of computer environment is considerably more complex than a disk operating system for a personal computer. For one thing, the fact that so many users share the same facilities requires an operating system that protects users as much as possible from inadvertent or deliberate interference. Even here, however, viruses are possible, but they require much more sophistication. Usually they exploit a flaw in some part of the operating system – a bug, so to speak – as was evidenced by the "virus" (actually it was a worm) that spread thoughout the Internet last fall.

During the evening of November 2, 1988, someone ran a worm program on one of the several thousand North American computers interconnected through a data-communications network called the Internet. The network connects machines at universities, businesses, Government agencies such as the National Aeronautics and Space Administration and even some military installations. With frightening speed the worm spread to more than 1,000 machines during that evening and the next day. As copies of the worm proliferated, operators of individual systems noticed memory utilization soaring and machine response becoming sluggish. The worm did not attack files or other programs, however. It seemed content merely to proliferate throughout as many host machines as possible. Even so, the damage in lost time was immense.

As I mentioned above, a worm is a program that pursues an independent existence within a computer; it moves from computer to computer on its own, leaving duplicates of itself in each machine. The Internet worm consisted of two parts, a vector and a body. Starting from a given host computer, the worm would seek a new host by sending its vector to another computer. Once inside the machine, the vector would establish a communication link through which the worm's body could be sent. Details of this attack were revealed by Eugene H. Spafford of Purdue University in a 40-page document a few weeks after the event. One example of the worm's operation shows the cleverness of its creator.

UNIX, the operating system of choice on many of the Internet computers, allows processes to take place in the computer that are not associated with any particular user. Such independent processes are called demons. One demon, called fingerd (pronounced fingerdee), enables users to get information about other users. Such a service is desirable in a computing environment in which users must share programs and data for research and development purposes.

The worm in its current host computer would send a message to one of the other potential host computers on its list (which was obtained illegally). In requesting the services of the fingerd demon, the worm gave it some information, just as an ordinary user might. But the worm supplied so much information to the demon that the data filled the space reserved for it in the computer's memory and overflowed into a "forbidden" area.

The area that was thus overwritten was normally reserved for instructions that fingerd consulted in deciding what to do next. Once inside such an area the worm (whose body still inhabited the original host machine) invoked a so-called command interpreter of the new machine, effectively claiming a small piece of the UNIX operating system all to itself. After the command interpreter was at its disposal, the worm transmitted some 99 lines of source code constituting the vector. On the worm's command, the unwitting potential host then compiled and ran the vector's program, virtually guaranteeing infection.

The vector program hid itself in the system by changing its name and deleting all files created during its entry into the system. After doing that it established a new communication channel to the previous host and, using standard system protocols, copied over the files making up the main body of the worm.

Once inside a new host computer, the worm's main job was to discover the names and addresses of new host machines by breaking into areas reserved for legitimate users of the system. To do so it relied on an elaborate password-guessing scheme that, owing to the carelessness with which most users choose passwords, proved rather successful. When it had a legitimate user's password, the worm could pretend to be the user in order to read what he or she may have had in the computer's memory and to discover the names of other computers in the Internet that it could also infect.

According to Spafford, most of the UNIX features (or "misfeatures," as he calls them) that allowed the worm to function as it did have been fixed. Yet the fact has not allayed his worries about computer security, as the quotation at the beginning of this article reveals. Perhaps he was thinking of Cohen's theoretical investigation of viruses, which might apply to worms just as well.

If technopaths insist on vandalizing computer systems, it may be time to form a Center for Virus Control. During the Internet worm crisis teams at the University of California at Berkeley and a few other Internet stations were able to capture copies of the worm, analyze its code and determine how it worked. It would seem reasonable to establish a national agency that would combat computer viruses and worms whenever and wherever they break out – particularly if computer infections are destined to increase. Although the Internet experience hinted at the horrors that may still come, it also showed the efficacy of an organized resistance against them.

Cohen has established that it is impossible to write a computer program that will detect every conceivable virus, even though a defense can be constructed against any given virus. On the other hand, for any such defense there are other viruses that can get around it. According to Cohen, this ominous state of affairs might subject future computing

environments to a kind of evolution in which only the fittest programs would survive.

The situation is reminiscent of Core War, a computer game I have written about in previous comumns [see *SCIENTIFIC AMERICAN*, May, 1984, March, 1985, and January, 1987]. But a Core War program does not bully innocent systems. It picks on someone its own size: another Core War program. The two programs engage in subtle or blatant conflict in a specially reserved area of a computer's memory called the coliseum. There is no danger of a Core War program ever escaping to do damage in the real world, because no Core War program or anything like it would ever run effectively in a normal computing environment. Core War programs are written in a language called Redcode that is summarized in the table on the opposite page.

Perhaps a simple example of such a program will serve to introduce the game to readers not already familiar with it. Here is a program called DWARF that launches a 0-bomb into every fifth memory location:

```
DAT -1
ADD #5 -1
MOV #0 @-2
JMP -2
```

The memory coliseum that all Core War programs inhabit consists of several thousand addresses, or numbered memory cells, arranged in a long strip. The instructions that make up DWARF, for example, occupy four consecutive addresses in the coliseum, say 1001, 1002, 1003 and 1004.

The DAT statement serves to hold a value that will be used by the program (in this case -1) at the address 1001. The ADD statement adds the number 5 to the location that is -1 units away from the ADD statement. Since the ADD statement has address 1002, it adds 5 to the number stored at the previous address, namely 1001, changing the -1 to a

1. The MOV command moves the number 0 into the memory cell referred to

by @-2. Where is that? The address is found by referring to the DAT statement two lines in front of the MOV command. There one finds the address where the program will put the number 0. The final command, JMP, causes execution of the DWARF program to jump back two lines, to the ADD command. This begins the process all over again.

The second time around, DWARF will change the contents of the DAT cell to 9 and then deliver a 0 to that memory address. If an enemy program happens to have an instruction at that address, it will be rendered inoperable and the program will perhaps "die" as a result.

In this manner DWARF goes on dropping 0-bombs on every fifth location until it reaches the end of memory – but memory never ends, because the last address is contiguous to the first. Consequently DWARF's bombs eventually begin to fall nearer and nearer to itself. Yet because DWARF is only four instructions long and the number of memory cells is

normally a multiple of 10, DWARF avoids hitting itself and lives to fight on – albeit blindly and rather stupidly.

Over the past few years Core War has evolved into a rather sophisticated game with numerous strategies and counterstrategies. There are programs that spawn copies of themselves, that launch hordes of mindless one-line battle programs and that even repair themselves when they are hit.

The International Core Wars Society, which currently has its headquarters in Long Beach, Calif., and branches in Italy, Japan, Poland, the Soviet Union and West Germany, organizes annual tournaments in which a programmer's skills are put to the test. Readers interested in joining a Core War chapter should contact William R. Buckley at 5712 Kern Drive, Huntington Beach, Calif. 92649.

In the 1987 tournament the Japanese entries gave the North American warrior programs a run for their money. The winner of the most recent tournament, held last December in Los Angeles, was a program from the Soviet Union called, oddly enough, COWBOY. Written by Eugene P. Lilitko of Pereslavl-Zalesky, a small city north-east of Moscow, COWBOY appeared to watch for "bombing runs" by enemy programs, to move itself out of harm's way and then to retaliate massively. Lilitko won the first prize of $250. The second prize of $100 went to Luca Crosara of Pistoia, Italy. The third prize of $50 was won by Douglas McDaniels of Alexandria, Va.

In closing I quote Spafford once again: "Writing and running a virus is not the act of a computer professional but a computer vandal." Let those who would even contemplate such an act try Core War instead.

I should like to thank Cohen, Spafford and John Carroll, a computer-security expert at the University of Western Ontario, for help with this article.

...

```
                              REDCODE REFERENCE

Simulator: pMARS
Version: 0.8.0
Standard: ICWS'94 draft (extended)

Language elements not currently in the draft are labeled with *, those
added in this version of pMARS are labeled with +.

----------------------------------------
Opcodes:

    DAT     terminate process
```

```
    MOV     move from A to B
    ADD     add A to B, store result in B
    SUB     subtract A from B, store result in B
    MUL     multiply A by B, store result in B
    DIV     divide B by A, store result in B if A <> 0, else terminate
    MOD     divide B by A, store remainder in B if A <> 0, else terminate
    JMP     transfer execution to A
    JMZ     transfer execution to A if B is zero
    JMN     transfer execution to A if B is non-zero
    DJN     decrement B, if B is non-zero, transfer execution to A
    SPL     split off process to A
    SLT     skip next instruction if A is less than B
    CMP     same as SEQ
    SEQ     (*) Skip next instruction if A is equal to B
    SNE     (*) Skip next instruction if A is not equal to B
    NOP     (*) No operation
    LDP     (+) Load P-space cell A into core address B
    STP     (+) Store A-number into P-space cell B

----------------------------------------
Pseudo opcodes:

    [labels] EQU text           replaces [labels] by text
            [EQU text]          (*) multi-line EQU: text continues

             ORG start          specifies execution start

             END [start]        end of assembly (optional execution start)

    [count]  FOR expression     (*) repeat enclosed instructions "expression"
             ROF                times, counter is incremented starting with 01

             PIN number         (+) P-space identification number, warriors
                                with same number share P-space

----------------------------------------
Modifiers:

    .A   Instructions read and write A-fields.

    .B   Instructions read and write B-fields.
```

```
    .AB  Instructions read the A-field of the A-instruction  and
         the B-field of the B-instruction and write to B-fields.

    .BA  Instructions read the B-field of the A-instruction  and
         the A-field of the B-instruction and write to A-fields.

    .F   Instructions read both A- and B-fields of  the  the  A-
         and  B-instruction and write to both A- and B-fields (A
         to A and B to B).

    .X   Instructions read both A- and B-fields of  the  the  A-
         and  B-instruction  and  write  to both A- and B-fields
         exchanging fields (A to B and B to A).

    .I   Instructions read and write entire instructions.

----------------------------------------
Addressing modes:

    #       immediate
    $       direct
    @       indirect using B-field
    <       predecrement indirect using B-field
    >       postincrement indirect using B-field
    *       (*) indirect using A-field
    {       (*) predecrement indirect using A-field
    }       (*) postincrement indirect using A-field


----------------------------------------
Directives:

    ;redcode                code follows, preceding text is ignored
    ;name                   name of warrior follows
    ;author                 name of author follows
    ;assert                 (*) expression that must evaluate to true
    ;trace [off]            (*) toggle trace bit for following instructions
    ;break                  (*) set trace bit for next instruction
    ;debug [static|off]     (*) enable/disable setting [static] trace bits

----------------------------------------
```

```
Predefined variables (*):

    CORESIZE          value of -s parameter (default: 8000)
    MAXPROCESSES      value of -p parameter (default: 8000)
    MAXCYCLES         value of -c parameter (default: 80000)
    MAXLENGTH         value of -l parameter (default: 100)
    MINDISTANCE       value of -d parameter (default: 100)
    ROUNDS            (+) value of -r parameter (default: 1)
    PSPACESIZE        (+) value of -S parameter (default: 1/16th CORESIZE)
    CURLINE           current line in generated assembly (starts with 0)
    VERSION           pMARS version ("60" is v0.6.0)
    WARRIORS          number of warriors specified on command line

----------------------------------------
Expression operators:

    Arithmetic:
        +   addition or unary plus
        -   subtraction or unary minus
        /   division
        %   modulo (remainder of division)

    Comparison (*):
        ==  equality
        !=  inequality
        <   less than
        >   greater than
        <=  less than or equal
        >=  greater than or equal

    Logical (*):
        &&  and
        ||  or
        !   unary negation

    Assignment (*):
    =   (to register variables a..z)

    Comparison and logical operators return 1 for true and 0 for false.
    Parentheses can be used to override this precedence order:
```

```
    1) ! - + (unary)
    2) * / %
    3) - + (binary)
    4) == != < > <= >=
    5) &&
    6) ||
    7) =


---------------------------------------
Redcode grammar:

statement_list :: statement statement_list | e ;
statement :: normal_stmt<1> |
             equ_stmt       |
             forrof_stmt    |
             comment_stmt   |
             substitution_stmt<2> ;

num       :: [0-9] ;
number    :: num number | num ;
alpha     :: [a-zA-Z] | "_" ;
alphanum  :: alpha | num ;
alphanums :: alphanum alphanums | e ;

label  :: alpha alphanums ;
label1 :: label label1 | label ;
labels :: label1 "\n" labels | label1 ;
stringization :: stringization"&"label | label ;

this_string :: (^\n)* ;
comment     :: ";" this_string "\n" | e ;

equ_stmt    :: labels equ_strings ;
equ_string  :: "equ" this_string comment "\n" ;
equ_strings :: equ_string "\n" equ_strings | equ_string ;

forrof_stmt :: labels index "for" expression<3> comment "\n"
               statement_list
               "rof" this_string "\n" ;
index       :: label
```

```
comment_stmt  :: info_comment | debug_comment | ignore ;
ignore        :: ";" this_string "\n" ;
info_comment  :: ";redcode" this_string "\n" |
                 ";" "name" this_string "\n" |
                 ";" "author" this_string "\n" |
                 ";" "date" this_string "\n" |
                 ";" "version" this_string "\n" |
                 ";" "assert" expression<4> "\n" ;
debug_comment :: ";" debug "\n" | ";" trace "\n" | ";" break "\n" ;
debug         :: "debug" | "debug" "off" | "debug" "static" ;
trace         :: "trace" | "trace" "off" ;
break         :: "break" ;
```

Note:
---
1. Normal statements are statements in the following form:
   opcode [address mode] operand [, [address mode] operand] [comment]
   More details about the grammar are given in the '88 or '94 proposal.

2. Substitution statements are labels that have been declared by EQU.
   If a label is declared this way: "IMP_instr equ imp mov imp, imp +
   1", the label name IMP_instr can be thought of as a statement.
   Therefore, whenf IMP_instr is used after its declaration, it will
   be replaced by "imp mov imp, imp + 1". It has effect of declaring
   a label 'imp' and inserting the statement 'mov imp, imp + 1'.

3. Valid expression for "FOR" statement is very close to C expression in
   which operator '()' has the highest precedence, followed by 'unary +,
   unary -', '*, / and %', 'binary + and binary -', '<, <=, >, >=',
   '==, !=', '&&', '||', and the lowest '='.
   Beside numbers, it can also has labels as its terms. All of its labels
   have to be declared before "FOR" statement is invoked.
4. Valid expression for "ASSERT" statement is the same as that for "FOR".

Case sensitivity

Opcode and pseudo-opcode names are case insensitive; labels are case
sensitive.

Label declaration

```
-----------------
```
Labels are declared in three ways:

- Using EQU. When a label name that first appears (has not been declared) is declared with EQU, all subsequent occurences of that label name will be replaced by the strings following the EQU.

  The following equates label THIS with "num + 1". THIS equ num + 1.

  If the string substituting the label contains other labels, those labels are also replaced by their substituting string. Recursive reference of labels are flagged as error.

  More than one statement can be declared as a label name. To achieve this, declare a blank label with EQU following the statement that declares as part of equation of a named EQU label. Thus, the declaration of the following:

```
core_clear equ spl 0
           equ mov 2, <-1
           equ jmp -1
```

  causes three statements are declared as strings of label core_clear.

- As an offset relative to the current normal statement. If some labels appear just before any '88 or '94 opcodes, they are automatically declared as constants that are relative to the current statement. For example:

```
first spl first                0000 spl 0
imp   mov imp, imp + 1  -----> 0001 mov 0, 1
```

- As an index belonging to FOR statement. Unlike other statements, labels declared with FOR consist of two ingredients: the last label serving as FOR index and the remaining labels serving as the same offset pointing at the first of the FOR statement. This allows the implementation of base[index] kind.

  This declaration:

```
base
index for 3
      mov base, base + index - 1
      rof
```

  translates into:

```
   0000 mov 0, 0
   0001 mov 0, 1
   0002 mov 0, 2
```

Stringization and declaration inside FOR/ROF:

```
FOR 5
imp mov imp, imp + 5
ROF
```

Declaration of the above example causes the assembler to complain for duplicating declarations. pMARS however offers the stringization feature to accomplish the same goal. Its syntax is: label"&"label"&"...
The first label can be any valid alphanums and it goes untranslated. The rest of the labels have to be a FOR index and it is to be substituted accordingly. Thus:

```
N FOR 5
  imp&N mov imp&N, imp&N + 1
  ROF
```

are expanded into:

```
imp01 mov imp01, imp01 + 5
imp02 mov imp02, imp02 + 5
imp03 mov imp03, imp03 + 5
imp04 mov imp04, imp04 + 5
imp05 mov imp05, imp05 + 5
```

It is then correctly compiled.

The following form is also valid:

```
prime01 equ 2
prime02 equ 3
prime03 equ 5
prime04 equ 7
prime05 equ 11

N FOR 5
  dat prime&N
```

```
  ROF

The conjunction of FOR statements and EQU statements:
-----------------------------------------------------

If labels that are not stringized are declared inside FOR/ROF statements,
the result is duplicating declaration. Although in the future it might be
allowed, it is well-advised that such labels are to be declared outside of
the FOR/ROF block.

If FOR statements are declared as strings of a label name using EQU such as:
THIS EQU N FOR 3
     EQU   mov 0, 3
     EQU   ROF

The expansion of such is feasible providing that both the FOR and ROF are
present in the same label name.

Therefore, the following will not work:
THIS EQU N FOR 3
THAT EQU   ROF

THIS
 mov 0, 3
THAT

or

TEST EQU N FOR 3
 mov 0, 3
 ROF

PMARS parsing background
------------------------
Parsing is done in three steps:
1. Reading from input
2. First pass assembly
3. Second pass assembly

- During reading: info_comment is parsed, comments to be ignored are removed,
  the remaining are copied into memory. If the first ';redcode' appears,
```

```
  all the current contents are erased from memory. Reading is continued
  until it encounters the next ';redcode'. Reading always stops when it
  encounters a line containing a word END or end-of-file.
- During the first pass, all labels are collected. All expansions and
  removals are done in this pass. All labels before opcode are substituted
  if they have been declared or declared otherwise. All labels after opcode
  are preserved for the second pass. This allows forward declaration
  mechanism for labels after opcode.
- During the second pass, no labels are declared and collected. There is
  no further statement expansions and removals. All labels that have not
  been substituted are substituted. Syntax checking to meet with '88 or '94
  requirement is done in this stage.

$Id: redcode.ref,v 1.5 1995/07/30 18:47:11 stst Exp stst $
```

Befunge

The Befunge FAQ v.4 - *Last updated November 4, 1997*

-1) History

The Befunge Mini-FAQ was first authored by Ben Olmstead, on May 21,

1. Then Pascal decided to mess with it a bit, and claim that he was

maintaining it or something. He also removed the 'Mini-'.

Many thanks are owed (in no particular order) to Chris Pressey, Ben Olmstead, Francis Irving, and Johannes Keukelaar for their contributions to this FAQ.

---

0) Table of Contents:

1. General Information 1.1) What is Befunge? 1.2) Oh. Wow. So, what is Befunge good for? 1.3) What versions of Befunge are there? 1.4) What Befunge resources are there? 1.5) OK. I'm hooked. What other nifty things can you point me at?

2. Befunge Programming 2.1) How does wrapping work? 2.2) How does time travel work?

3. WTF... 3.1) is WTF? 3.2) is a TM (Turing Machine)? 3.3) does TC (Turing-Complete) mean? 3.4) is the halting problem? 3.5) is a BEMD (BEM device)? 3.6) is LiS (Lost in Space)? 3.7) are Wierd, Cracked, etc.? 3.8) is YAPL?

1.1) What is Befunge?

Befunge is an interpreted, two-dimensional, stack-based programming language originally created in 1993 by Chris Pressey.

Basically, Befunge is the first language (to our knowledge) to allow the program counter to move not only forwards and back, but also sideways. The source code for the language looks like garbage. It has been described as "a cross between Forth and Lemmings."

Doesn't that make it all clear? Well, we'll start with a simple "Hello, world." program (all of you people who know what you're doing, I know that there are redundancies.):

```
0".dlrow ,olleH">v
                 ,:
                 ^_@
```

The **Instruction Pointer** starts at the upper-left-hand corner, heading right. It first encounters '0'. '0' means 'push the number 0'. Simple, right? OK. Next, it hits '"'. '"' toggles what is known as **stringmode**. In stringmode, the ASCII equivalent of each character gets pushed onto the stack as it is encountered. So, the '.' gets pushed first, then the 'd', and so on. It is important to remember that the stack is last in, first out, so the 'H', which gets pushed last, is going to come off the stack first. Because of this, essentially every string constant in Befunge appears backwards with respect to which direction the IP is moving.

So, the IP get finished pushing our message, and hits the second '"'. This time, the IP is already in stringmode, so it turns it off. The IP then encounters '>'. '>' tells the IP to start moving towards the right, but since it is already moving right, this doesn't do anything. The next instruction, 'v', tells the IP to start moving downward.

The IP begins moving downward and hits ':'. ':' is the 'duplicate' instruction: it takes whatever is at the top of the stack and makes a copy of it. So now there are two copies of 'H' on the top of the stack. The next instruction, '_' is called 'horizontal if'. It pops the top item off the stack, and if it is 0, tells the IP to go right, otherwise it tells the IP to go left. Since 'H' is obviously not 0, the IP goes to the left. Notice that, because we used ':', there is still a copy of 'H' on the stack.

The IP now reaches '^'. '^' is, guess what, 'up'. I think you know by now what that means. Next, it reaches ','. ',' means 'print character'. So, it pops the 'H' off the stack, and prints it out. The next item on the stack is now 'e'.

The IP now reaches that same 'right' command, and starts moving to the right, then down, then duplicates 'e', then goes left, then up, then prints 'e'. It does the same thing with each character, until it comes to the 0 at the bottom of the stack. When it reaches the 0, the '_' instruction tells the IP to go RIGHT, not left, and the IP reaches '@'. '@' is 'end program'.

That was a terribly quick and somewhat dirty look at Befunge. I glossed over a few points, and that particular "Hello, World" program is not the "best". (One usually determines the best Befunge program by how small it is.) However, it should be enough to give you a taste of the language.

1.2) Oh. Wow. So, what is Befunge good for?

Befunge is NOT good for:

- Writing an OS.
- Writing a reasonably-sized program.
- Speed-sensitive applications.
- Just about anything practical.

Befunge IS good for:

- A good laugh.
- Amazing your friends, amusing your colleagues.
- Whiling away bored hours when you really should be working.
- Expanding your horizons. (Infinitely, even.)

1.3) What versions of Befunge are there?

Befunge-93 is the original Befunge language created in '93 by Chris Pressey. It is set, it is standard, no one is likely to change it. (Nor use it much anymore, for that matter.)

Befunge-96 used to exist as a malformed half-standard, before January 1997 came around without anything happening to it. Don't bother implementing it or writing anything in it. The current standard is '97, below.

Befunge-97 is a working standard for 1997. It is defined and maintained by the people on the Befunge mailing list by general approval. There is no hard and fast rule for what is 'general approval', but a good rule of thumb is 'nobody (who is talking, at least) has any disagreements'. You can pretty safely assume that whatever is in Befunge-97 will stay.

The working '97 standard itself is stored at Chris Pressey's web page (See below). The expected date for solidification of the standard is about October 1997, but don't take this as a promise.

There are also several developmental concepts for the use of Befunge, including but not limited to:

- **BefBots** is similar to C-Robots in intent; Befunge programs control robots which attack each other.
- **BeGlad** (Befunge Gladiators) is more like Core Wars; Programs written in a modified version of Befunge-97 actually attack within Befunge-space itself, trying to force the other program to run out of processes and die.
- **Bevolve** is an attempt to modify the Befunge command set to better accomodate the evolution of Befunge programs in an open enviornment.
- **Bef***** is a general label applied to efforts to have a minimal instruction set for a two-dimensional language; The name derives from Befunge and Brainf***.

1.4) What Befunge resources are there?

So far, there are three web pages and a mailing list:

Got a nifty Befunge program or idea for the future of Befunge? Then submit it to the Befunge mailing list `mailto:befunge@cats-eye.com`, a little group of freaks who actually **like** toying with Befunge. It is currently responsible for maintaining the Befunge-97 Standard. To subscribe to the befunge mailing list, send a message with the subject "subscribe [email address]" to befunge-request@cats-eye.com `mailto:befunge-request@cats-eye.com`. To unsubscribe, send a message with the subject "unsubscribe [email address]" to the same place. All messages sent to befunge@cats-eye.com `mailto:befunge@cats-eye.com` will be distributed to everyone on the mailing list.

(Also, although it probably hasn't propagated very far yet, the newsgroup alt.lang.funge has purportedly been created for wide-ranging discussion on anything even remotely connected to *funges.)

Chris Pressey `mailto:cats-eye@cats-eye.com`, the original creator of Befunge, has a very nice Befunge web page. `https://web.archive.org/web/20010417044912/http://www.cats-eye.com/cet/soft/lang/befunge/`

On his site:

- the official Befunge-93 interpreter
- the Befunge-93 specification
- the working Befunge-97 standard
- a large number of Befunge sources
- information on how to subscribe to the Befunge mailing list

Jason Reed mailto:jreed@itis.com (alias Pascal) also has a page https://web.archive.org/web/20010417044912/http://www.loungelizard.com/pascal/befunge/ devoted to Befunge. On his site:

- the source for bp, a Befunge-97 interpereter written in perl
- a few sources that he has written
- this FAQ

Ben Olmstead mailto:bem@mad.scientist.com (alias Bug-eyed Monster or just BEM), a random person, has a site https://web.archive.org/web/20010417044912/http://www.geocities.com/SiliconValley/Way/3571/befunge.html devoted to Befunge, which is sometimes just a bit out of date... On his site:

- the semi-official Befunge-97 interpreter (in nearly-ANSI C),
- the semi-official Befunge-X interpreter (actually, uses the same source files),
- a port of Chris Pressey's Befunge-93 interpreter to Windows95 by Kevin Vigor (including a much-improved debugger, with source)
- a large number of Befunge sources, including all the ones from Chris' and Pascal's sites.

1.5) OK. I'm hooked. What other nifty things can you point me at?

If you like befunge, you may enjoy:

Orthogonal https://web.archive.org/web/20010417044912/http://www.muppetlabs.com/~breadbox/orth/home.html: updated Orthogonal interpreter. Orthogonal was one of the first 2-dimensional languages, but Befunge beats it by a year. Still, it is an independent look at 2-D programming not offered elsewhere. (And we can prove that it is independent because, though Befunge came first, it didn't arrive on the 'net until after Orthogonal.)

The Retrocomputing Museum https://web.archive.org/web/20010417044912/http://www.ccil.org/retro/retromuseum.html: home of the freaks, brain-damages, and ancients of the computing world. Includes a version of "hunt the wumpus" which is sickeningly true to the original BASIC version, emulators for OISC and URISC (RISCs taken to their logical extreme–single-instruction computers!), I believe an emulator for the PDP-11, Algol-60 (I think), BCPL and B, Foogol, and many others. Also includes out-of-date versions of Intercal and Orthogonal.

Intercal https://web.archive.org/web/20010417044912/http://www.ccil.org/intercal/: the language designed to be as completely unlike any other as possible while still being

Turing- complete. It has done well. The most recognizable operators are "xor/and/or with self, rolled one bit to the left". The character set is, to say the least, creative, including the rabbit-ears ("), spot (.), sqiggle (sic) (~), mesh (#), half-mesh (=), worm (-), spark (’), and hybrid (;). It has no "if". It has the COME FROM command. The simplest way to store the value 65536 to a 32-bit variable in Intercal is: :1<-#256~#0 The Intercal ROT-13 program makes a decent slowcat when two of them are stuck together, and the Intercal prime number generator took 17 hours on a SPARC to generate up to 65535. (A task which takes a decent C program 1.5 seconds on a 486/25...) This is not a language for the faint of heart, but the manual is far more amusing than anything in Befunge.

False `https://web.archive.org/web/20010417044912/http://www.cats-eye.com/cet/soft/lang/false/`: An interesting, simple, stack-based language. A few nifty features.

BrainF*** `https://web.archive.org/web/20010417044912/http://www.cats-eye.com/cet/soft/lang/bf/`: a language which is even more frightening than Intercal, and about as profane as its name. Intercal is a joke; BrainF*** is a perversion. There are eight commands, four of which are increments and decrements. Two are i/o. The other two are used for loops. If this does not frighten you, it should.

The Bad Languages Page `https://web.archive.org/web/20010417044912/http://www.cs.bc.edu/~connorbd/badlangs.html`: An entire page devoted to links to those wonderfully bad languages: Intercal, Orthogonal, Befunge, False, BrainF***, and whatever else can be dug up. Most of the sites listed above are linked from here.

Q-BAL `https://web.archive.org/web/20010417044912/http://lausd.k12.ca.us/~mshulman/q-bal/`: An unimplemented language almost as twisted as Befunge, but queue-based instead of stack-based.

2.1) How does wrapping work?

For various reasons, toroidal wrapping is problematic in Befunge-97. Instead, we use a special wrapping technique that has more consistant results in this new, more flexible environment. It is called *same-line wrapping*.

Same-line wrapping can be described in several ways, but the crucial idea it encompasses is this: unless the delta or position of the IP is changed by a command, the IP will always wrap such that it returns to the instruction it was on before it wrapped.

The mathematical description of same-line wrapping is known as *Lahey-space wrapping*, which defines a special topological space. It is generally of more interest to topologists and mathematicians than programmers. We won't cover it here, but it is included in Appendix A [or some other document that we may link to eventually] for completeness.

The algorithmic description of same-line wrapping can be described as *backtrack wrapping*. It is more of interest to Befunge Interpreter implementers than Befunge programmers. However, it does describe exactly how the wrapping *acts* in terms that a programmer can understand, so we will include it here.

When the IP attempts to travel into the whitespace between the code and the end of addressable space, it backtracks. This means that its delta is reflected 180 degrees and it ignores all instructions. Travelling thusly, it finds the other 'edge' of code in a similar way (when there is nothing but whitespace in front of it.) It is then reflected 180 degrees once more (to restore its original delta), stops ignoring, and executes the first available instruction.

For example, the following code:

```
v4
v >....@
>11X
    1
     2
      3
```

prints out "4 3 2 1 ". When the IP looks for the next instruction after the '3', it can't find one in the in the direction of the IP's delta ((1,1)), so the IP turns around, and finds the last command in opposite direction, ie (-1, -1), which happens to be the '4'. The IP turns around again, resuming its original delta, and executes the '4', then the '>', then the four '.'s, then the '@' and ends.

It is easy to see at this point that the IP remains on the same line: thus the name. (Also note that the entire wrapping process takes 0 ticks, as would be expected from any wrapping process.)

Same-line wrapping has the advantage of being backward-compatible with Befunge-93's toroidal wrapping. It also works safely both when the IP delta is non-cardinal, and when the size of the program changes.

2.2) How does time travel work?

Although the 't' (Time Travel) command has not yet been implemented, and is not planned for inclusion in *funge-97, it comes up often enough in discussion to deserve some explanation. Although no explanation has been offered as to *why* the concept exists, the following message by Chris Pressey (suggesting Chronefunge, a variant of Befunge including subject-singularity time travel) might help to explain how it works:

Chronefunge: Subject-singularity Time-Travel Befunge!

This can be done. It's a little horrendous... but it can emulate time-travel. Time is one of the "special" and largely uninterfered-with dimensions of Nefunge, and Chronefunge addresses this "problem".

First of all, you have to be aware of a fancy-sounding term I just made up, which applies to the science fiction of time travel: subject-singularity time travel.

Subject-singularity means that history is relative to the subject which is time-travelling.

If the subject jumps back in time 100 years, the last 100 years of history have been wiped out completely, and the "present time" "starts anew" in the year 1897. Another way to put this is that an act of backwards time-travel is always tantamount to destroying and rewriting history.

This gets around the Grandfather Paradox (although not elegantly); you go back in time, you kill your grandfather as a child, no problem - you're still in the past, alive, and now making a new history from that point on. You were never born but you're here now... and you know what might come next.

To get Befunge to do that, first, you have to store a history - a collection of program states including the stack, Befunge-space, and the IP list. This is a lot of data, but not like an unhandlable amount.

Then, declare a command like t "Time Travel" which takes one argument called tick-delta.

If tick-delta > 0, freeze this IP in "suspended animation" for tick-delta ticks. Note how simple it is to travel forward in time! "Suspended animation" is completely seperate from playfield and all other IP's: it's as if, to them, this IP didn't even exist for the time it's frozen.

If tick-delta $= 0$, do nothing ("time-travel to the present")

If tick-delta < 0, save the ip -> subject-ip. replace the ENTIRE PROGRAM STATE with the program state history of tick-delta ticks ago. Then add subject-ip back into the program. This IP is now "in the past!"

Oh what wonderful fun you can have in a language when you disregard entropy. Why, you could have the program display the result "before" it's been calculated!

A reasonable implementation of this might not even be required to provide a "time machine" (history taker) with an extreme range. Unreasonable arguments like

10/t ; try to go to the Restaurant at the End of the Universe ; 010/-t ; try to go to the Big Bang Burger Bar ;

...should be met with an error/exception if your interpreter only saves a smaller number of program states (like 256) in it's history. By no means is it expected to store MAXMEMSIZE frames of history. t should always make an error/exception if your interpreter doesn't support time-travel.

And if you travel before the start of the program? Well, it should probably bomb - or encounter empty-Befunge-space, and have the program just suddenly appear "when it should". Or another error/exception?

---

3) WTF...

3.1) is WTF?

The universal interrogative particle. Oh, you wanted to know WTF the point of this section is? Occasionally there arise certain abbrevations, concepts, and monstrosities on the befunge mailing list that would otherwise confuse the hell out of the casual lurker - this in addition to the capability of befunge itself to be confusing. Anyways, this section just tries to clear up some common terminology. Now you know.

3.2) is a TM (Turing Machine)?

(*This section needs revision!*)

3.3) does TC (Turing-Complete) mean?

A system, machine, programming language, tinkertoy set, etc. is considered TC, put simply, when it can simulate any TM. Every personal computer, every HP48GX, every Cray, no matter how 'powerful' in terms of memory or speed, is exactly TC, no more, no less.

3.4) is the halting problem?

Suppose you had a computer program (call it H) that could tell just by examining another program whether the latter would either a) eventually stop or b) run forever. This program would be said to **solve the halting problem**. Such a program would be extremely useful - you could feed it a program that would stop only when it found a counterexample to, say, Goldbach's Conjecture^* , and it could (in a finite amount of time) tell you if it had any counterexamples at all.

However, imagine another program, based on H, called U, which responds to the program you feed it in a certain way: if the input program stops, then U will enter an infinite loop, and if U determines that the input program doesn't ever stop, then U stops. Clearly, if H exists, then modifying it to become U is possible, even trivial.

Now the fun starts. Ask U, innocently, whether U itself stops. If U stops, then it doesn't, then it does, then it doesn't, etc. Thus, U can't exist, because it leads to a contradiction, and H can't exist either. **The Halting Problem Can Not Be Solved**

^* Goldbach's conjecture asserts that every sufficiently large even number can be expressed as the sum of two primes. To the editor's knowledge, it is still undecided.

3.5) is a BEMD (BEM device)?

Bem Olmstead, alias BEM, mentioned working on the idea of a program or dev

Brainfuck

I started to think about a BF interpreter written in BF, and because I did not want to write BF code directly, I started with writing a C program that could generate BF code for often

used constructs. After some experimentation, I decided to implement a direct execution mode (making use of a define), so that I didn't have to go through the generate-interpret cycle. This resulted in the BF interpreter in BF generation `http://www.iwriteiam.nl/Ha_genbfi_c.txt` program. If the macro symbol EXECUTE is not defined, this program when executed generates a BF interpreter in BF. This BF interpreter expects as input a BF program terminated with an exclamation mark, followed by the input for the program to be interpreted. I by no means claim that this BF interpreter in BF is the shortest possible. (Actually, NYYRIKKI wrote a much short one `http://www.iwriteiam.nl/Ha_vs_bf_inter.html` and Daniel B. Cristofani `http://www.hevanet.com/cristofd/` an even shorter one `http://www.hevanet.com/cristofd/dbfi.b`.)

The BF interpreter in BF (when filtered through a comment remover `http://www.iwriteiam.nl/Ha_bfrmcom_c.txt`) looks like:

```
>>>,[->+>+<<]>>[-<<+>>]>++++[<++++++++>-]<+<[->>+>>+<<<<]>>>>[-<<<<+>>
>]<<<[->>+>+<<<]>>>[-<<<+>>>]<<[>[->+<]<[-]]>[-]>[[-]<<<<->-<[->>+>>+
<<<<]>>>>[-<<<<+>>>>]<<<[->>+>+<<<]>>>[-<<<+>>>]<<[>[->+<]<[-]]>[-]>]<
<<[->>+<<]>[->+<]>[[-]<<<[->+>+<<]>>[-<<+>>]>++++++[<+++++++>-]<+<[->
>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<
[-]>>[[-]<<<<->-<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>
>]<[<[->>+<<]>[-]]<[-]>>]<<<<[->>>+<<<]>[->>+<<]>+>[<->[-]]<[<<<<+>>>
>[-]]<<<[->+>+<<]>>[-<<+>>]>+++++[<++++++++>-]<<[->>>+>+<<<<]>>>>[-<<
<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>[[-]<<<<->-<[
->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]
]<[-]>>]<<<<[->>>+<<<]>[->>+<<]>+>[<->[-]]<[<<<<++>>>>[-]]<<<[->+>+<<]
>>[-<<+>>]>++++++[<++++++++++>-]<<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+
>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>[[-]<<<<->-<[->>>+>+<<<<]>>>
>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>]<<<<[->
>>+<<<]>[->>+<<]>+>[<->[-]]<[<<<<+++>>>>[-]]<<<[->+>+<<]>>[-<<+>>]>+++
+++[<++++++++++>-]<+++<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-
<<<+>>>]<[<[->>+<<]>[-]]<[-]>>[[-]<<<<->-<[->>>+>+<<<<]>>>>[-<<<<+>>>>
]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>]<<<<[->>>+<<<]>[->>
+<<]>+>[<->[-]]<[<<<<++++>>>>[-]]<<<[->+>+<<]>>[-<<+>>]>+++++[<+++++++
++>-]<+<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->
>+<<]>[-]]<[-]>>[[-]<<<<->-<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<
]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>]<<<<[->>>+<<<]>[->>+<<]>+>[<->[-]
]<[<<<<++++>>>>[-]]<<<[->+>+<<]>>[-<<+>>]>++++[<++++++++++>-]<<[->>>
+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-
]>>[[-]<<<<->-<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>
]<[<[->>+<<]>[-]]<[-]>>]<<<<[->>>+<<<]>[->>+<<]>+>[<->[-]]<[<<<<++++++
>>>>[-]]<<<[->+>+<<]>>[-<<+>>]>+++++++[<++++++++++++>-]<<[->>>+>+<<<<
]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>[[-]
```

<<<<->-<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->
>+<<]>[-]]<[-]>>]<<<<[->>>+<<<]>[->>+<<]>+>[<->[-]]<[<<<++++++++>>>>[-
]]<<<[->+>+<<]>>[-<<+>>]>+++++++[<++++++++++++>-]<++<[->>>+>+<<<<]>>>
>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<]>[-]]<[-]>>[[-]<<<<
->-<[->>>+>+<<<<]>>>>[-<<<<+>>>>]<<<[->+>>+<<<]>>>[-<<<+>>>]<[<[->>+<<
]>[-]]<[-]>>]<<<<[->>>+<<<]>[->>+<<]>+>[<->[-]]<[<<<<++++++++>>>>[-]]<
<<<[->>+>+<<<]>>>[-<<<+>>>]<[<<<[->>>>>>>>>>+<+<<<<<<<<<]>>>>>>>>[-<<<<<
<<<+>>>>>>>>>]<<<<<<<[->>>>>>>>>>+<<+<<<<<<<]>>>>>>>[-<<<<<<<+>>>>>>>]>[
<[->>>>>+<<<<<]>[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>+>-]>>[-]<[->+<]<<[[-<
<<<<+>>>>>]<<<<<-]<<<<<<<<+>[-]>>[-]]<,[->+>+<<]>>[-<<+>>]>++++[<+++++
+++>-]<+<[->>+>>+<<<<]>>>>[-<<<<+>>>>]<<<[->>+>+<<<]>>>[-<<<+>>>]<<[>[
->+<]<[-]]>[-]>[[-]<<<<->-<[->>+>>+<<<<]>>>>[-<<<<+>>>>]<<<[->>+>+<<<]
>>>[-<<<+>>>]<<[>[->+<]<[-]]>[-]>]<<<<[->>+<<]>[->+<]>]<<<<<[-][->>>>>
>>>>+<<<<<<+<<<]>>>[-<<<+>>>]>>>>>>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>
+>-]>>[[-<+<+>>]<<[->>+<<]>[-<+>[<->[-]]]<[[-]<[->+>+<<]>>[-<<+>>]<<[[
-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]<<<<<<<<[-]>>>>>>>>>[-<<<<<<<<<+>>
>>>>>>]<<<<<<<<<<[->>>>>>>>>>+<+<<<<<<<<<]>>>>>>>>>[-<<<<<<<<<+>>>>>>
>>>]>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-]>>>+<<<<[[-<<<<<+>>>>>]<<<
<<-]<<<<<<<<<+[->>>>>>>>>>+<<<<<<+<<<]>>>[-<<<+>>>]>>>>>>[<[->>>>>+<<<<<
]>[->>>>>+<<<<<]>>>>+>-][-]]>>[-<+<+>>]<<[->>+<<]>[-[-<+>[<->[-]]]]<[[
-]<[->+>+<<]>>[-<<+>>]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]<<<<<<<<
[-]>>>>>>>>>[-<<<<<<<<<+>>>>>>>>>]<<<<<<<<<<[->>>>>>>>>>+<+<<<<<<<<<]>
>>>>>>>>[-<<<<<<<<<+>>>>>>>>>]>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-]
>>>-<<<<[[-<<<<<+>>>>>]<<<<<-]<<<<<<<<+[->>>>>>>>>>+<<<<<<+<<<]>>>[-<<<
+>>>]>>>>>>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-][-]]>>[-<+<+>>]<<[->
>+<<]>[-[-[-<+>[<->[-]]]]]<[[-]<[->+>+<<]>>[-<<+>>]<<[[-<<<<<+>>>>>]>[
-<<<<<+>>>>>]<<<<<<-]<<<<<<<<[-]>>>>>>>>>[-<<<<<<<<<+>>>>>>>>>]<<<<<<<
<<<->+[->>>>>>>>>>+<<<<<<+<<<]>>>[-<<<+>>>]>>>>>>[<[->>>>>+<<<<<]>[->>>
>>+<<<<<]>>>>+>-][-]]>>[-<+<+>>]<<[->>+<<]>[-[-[-[-<+>[<->[-]]]]]]<[[-
]<[->+>+<<]>>[-<<+>>]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]<<<<<<<<[
-]>>>>>>>>>[-<<<<<<<<<+>>>>>>>>>]<<<<<<<<<<+>+[->>>>>>>>>>+<<<<<<+<<<]>
>>[-<<<+>>>]>>>>>>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-][-]]>>[-<+<+>
>]<<[->>+<<]>[-[-[-[-[-<+>[<->[-]]]]]]]<[[-]<[->+>+<<]>>[-<<+>>]<<[[-<
<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]<<<<<<<<[-]>>>>>>>>>[-<<<<<<<<<+>>>>
>>>>>]<<<<<<<<<<[->>>>>>>>>>+<+<<<<<<<<<]>>>>>>>>>[-<<<<<<<<<+>>>>>>>>
>]>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-]>>>.<<<<[[-<<<<<+>>>>>]<<<<<
-]<<<<<<<<+[->>>>>>>>>>+<<<<<<+<<<]>>>[-<<<+>>>]>>>>>>[<[->>>>>+<<<<<]>
[->>>>>+<<<<<]>>>>+>-][-]]>>[-<+<+>>]<<[->>+<<]>[-[-[-[-[-[-<+>[<->[-]
]]]]]]]<[[-]<[->+>+<<]>>[-<<+>>]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<
-]<<<<<<<<[-]>>>>>>>>>[-<<<<<<<<<+>>>>>>>>>]<<<<<<<<<<[->>>>>>>>>>+<+<
<<<<<<<]>>>>>>>>>[-<<<<<<<<<+>>>>>>>>>]>[<[->>>>>+<<<<<]>[->>>>>+<<<<

```
<]>>>>+>-]>>>,<<<<[[-<<<<<+>>>>>]<<<<<-]<<<<<<<<<+[->>>>>>>>>+<<<<<<+<<
<]>>>[-<<<+>>>]>>>>>>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-][-]]>>[-<+
<+>>]<<[->>+<<]>[-[-[-[-[-[-[-[-<+>[<->[-]]]]]]]]]]<[[-]<[->+>+<<]>>[-<<+
>>]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]<<<<<<<<[-]>>>>>>>>>[-<<<<<
<<<<+>>>>>>>>>]<<<<<<<<<<[->>>>>>>>>>+<+<<<<<<<<<]>>>>>>>>>[-<<<<<<<<<
+>>>>>>>>>]>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-]>>>[-<<<+>+>>]<<[->
>+<<]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]>[-<<<<<<+>>>>>>]>+<<<<<<
<[>>>>>>>-<<<<<<<[-]]<<<[->>>>>>>>>>+<<<<<<+<<<]>>>[-<<<+>>>]>>>>>>[<[-
>>>>>+<<<<<]>[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>+>-]>[[-]<+[<[->>>>>+<<<<
<]>[->>>>>+<<<<<]>>>>+>>>[->>+<<<+>]<[->+<]>>>[-[-[-[-[-[-[-<<<+>>>[<<
<->>>[-]]]]]]]]]<<<[<+>[-]]>[->>+<<<+>]<[->+<]>>>[-[-[-[-[-[-[-[-<<<+>
>>[<<<->>>[-]]]]]]]]]]<<<[<->[-]]<]>[-]]<<[->>>>>+<<<<<]>>>>>+>[-]]>>[
-<+<+>>]<<[->>+<<]>[-[-[-[-[-[-[-[-<+>[<->[-]]]]]]]]]]<[[-][-]<[->+>+<
<]>>[-<<+>>]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]<<<<<<<<[-]>>>>>>>
>>[-<<<<<<<<<+>>>>>>>>>]<<<<<<<<<<[->>>>>>>>>>+<+<<<<<<<<<]>>>>>>>>>[-
<<<<<<<<<+>>>>>>>>>]>[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>>+>-]>>>[-<<<+>
+>>]<<[->>+<<]<<[[-<<<<<+>>>>>]>[-<<<<<+>>>>>]<<<<<<-]>[-<<<<<<+>>>>>>
]<<<<<<[->>>>>>>+<<<<<<<]<<<[->>>>>>>>>>+<<<<<<+<<<]>>>[-<<<+>>>]>>>>>>
[<[->>>>>+<<<<<]>[->>>>>+<<<<<]>[->>>>>+<<<<<]>>>+>-]>[[-]<+[<[-<<<<<+
>>>>>]>[-<<<<<+>>>>>]<<<<<<->>>[->>+<<<+>]<[->+<]>>>[-[-[-[-[-[-[-<<<+
>>>[<<<->>>[-]]]]]]]]]<<<[<->[-]]>[->>+<<<+>]<[->+<]>>>[-[-[-[-[-[-[-[
-<<<+>>>[<<<->>>[-]]]]]]]]]]<<<[<+>[-]]<]>[-]]<<[->>>>>+<<<<<]>>>>>+>[
-]]>>]
```

Next, I wrote an optimizing BF to C compiler <http://www.iwriteiam.nl/Ha_bf2c_c.txt> that eliminates all of the add-to loops. For example, '[<<+>+>-]' is compiled into 'p[2] += p[0]; p[1] += p[0]; p[0] = 0;'. It also removes all unnecessary 'p--' and 'p++' statements. The last project I embarked on, is a combined interpreter and compiler, which does even more optimizations. So far, I did not get very far, but you can have a look at it <http://www.iwriteiam.nl/Ha_bf_c.txt>. It supports the following command line options:

* -r: for interpreting a BF program
* -t: switching on trace mode during interpretation.
* -2c: for compiling to C.

One version of this program, I translated into JavaScript <http://www.iwriteiam.nl/JavaScript.html>, as part of the BF online <http://www.iwriteiam.nl/Ha_bf_online.html> page.

Brainf*** http://www.iwriteiam.nl/Ha_BF.html

BF http://www.iwriteiam.nl/Ha_BF.html is Turing-complete

Alan Turing http://www.wikipedia.org/wiki/Alan_Turing came up with the idea for the so called Turing Machine http://www.wikipedia.org/wiki/Turing_machine, a very simplistic computing model, which yet is powerfull enough to calculate all possible function which can be calculated. A Turing Machine consists of an endless tape of cells, which each can contain any of a given set of symbols, a pointer pointing to one of the cells, and a certain finite state control. Depending on the function to be calculated the finite state control determines how the tape is manipulated. It does this by reading the symbol under the pointer, and decides what to do: write a new symbol on the current cell, move the tape to the left, or to the right, and what will be the next state.

Each turing machine can be specified by the five elements:

- A finite set of symbols A, also called the alphabet.
- An initial symbol /a_init /, which each cell contains on initialization.
- A finite set of states S.
- An initial state /s_init /.
- A final state /s_final /.
- A function f of state and symbols onto a tupple consisting of a new state, a new symbol, and a movement. The movement can be one of **right** and **left**.

Many interesting properties of Turing Machines have been proven so far. For example, that each Turing Machine can be transformed into a Turing Machine with a tape that is cut-off on one side, and endless on the other side. Or that every Turing machine can be transformed into a Turing machine with set of symbols contains just two symbols.

A language is said to be 'Turing-complete' http://www.wikipedia.org/wiki/Turing_complete, if for each functions that can be calculated with a Turing Machine, it can be shown that there is a program in this language that performs the same function. There are basically three approaches to proof that a language is Turing-complete. These are:

1. Show <#map> there is some mapping from each possible Turing machine to a program in the language.
2. Show <#UTM> that there is a program in the language that emulates a Universal Turing Machine.

3. Show <#URM> that the language is equivalent with (or a superset of) a language that is known to be Turing-complete.

On the rest of the page, proofs of each of the above approaches will be given.

Translating Turing Machines to BF programs

We can proof that BF is Turing-complete, if we can show for every possible Turing machine, there is an equivalent BF program. Of course, we only need to show that a Turing Machine with a tape limited on one side can be simulated. What we need is an algorithm, which given a Turing machine, specified by the six elements given above, generates a BF program, which will simulate the Turing machine.

To achieve this, the set of symbols A is mapped onto the numbers 0 to n-1, where n equals the number of symbols, such that *a_ init / maps onto zero. Likewise, the set of states /S* is mapped onto the numbers 0 to m-1, where m equals the number of states, such that /s_init / maps onto zero, and /s_final / maps onto 1.

To simulate a Turing machine, we need an array with unlimited length to represent the tape, a index into that array representing the position of the head, and the current state. To program this in BF we will need to assign these to some memory locations. The position of the head will be stored in memory location *head*, and the current state in the memory location *state*. Of course, some additional temporary memory locations will be needed. Using the terminology introduced on the page An introduction to programming in BF `http://www.iwriteiam.nl/Ha_bf_intro.html`, we need a program of the following form:

```
/IsOne(state,t1)
Not(t1,t2,t3)
while(t2)/
    /getarray(base,1,0,head,cursymbol)/

    (some code which given the state and the cursymbol
    assigns a value to newstate, newsymbol, and newhead)

    /setarray(base,1,0,newhead,newsymbol)
    zero(cursymbol)
    zero(newsymbol)
    zero(head) move(newhead,head,t1)
    zero(state) move(newstate,state,t1) /

    /IsOne(state,t1)
    Not(t1,t2,t3)/
```

```
    /wend(t2)/
```

Now we only need to fill in the middle part of the loop. For each tupple in f we need to generate some code. This means that for each symbol /i/ in the range 0 to /n/-1, and each state /j/ in the range 0 to /m/-1, we have to insert a line:

```
   /map(state,head,cursymbol,
       i,j,k,l,m,
       newstate,newhead,newsymbol)/
```

Where, given the tupple /f(s_i ,a_j)/, /k/ is the number representing the new state, /l/ is the number representing the new symbol, and /m/ is equal to 1 if the movement is *right* and equal to 0 if the movement is *left*. The procedure /map/ is defined as follows:

```
  /map(state,head,cursymbol,
      astate,asymbol,anewstate,anewsymbol,movement,
      newstate,newhead,newsymbol)/
  = /
    copy(state,t1,t2)
    const(t2,astate)
    Equal(t1,t2,t3,t4,t5)
    if(t3)
      copy(cursymbol,t1,t2)
      const(t2,asymbol)
      Equal(t1,t2,t3,t4,t5)
      if(t3)
        const(newstate,anewstate)
        const(newsymbol,anewsymbol)
        copy(head,newhead,t1)
        ifelse(movement,t1)
          inc(newhead)
        else(movement,t1)
          dec(newhead)
        endelse(t1)
      endif(t3)
    endif(t3)/
```

In this code the expression /const(v,c)/ stands for the piece of code

that fills the memory location /v/ with the value /c/. Now, we only need to assign real values to the variable representing the memory locations to arrive at the final BF program executing the given Turing machine. Note that memory location assigned to /base/ should be larger than all other memory locations used.

Universal Turing Machine

The text of this section is based on
this fully documented BF program
<http://www.hevanet.com/cristofd/brainfuck/utm.b>
written by Daniel B. Cristofani
<http://www.hevanet.com/cristofd/brainfuck/>.

A Universal Turing Machine (UTM) is a Turing machine that can simulate some Turing-complete computational model. By giving a BF program which simulates a particular UTM, we proof that BF is Turing-complete. The UTM that we implement here is taken from from Yurii Rogozhin's article /Small universal Turing machines/, in Theoretical Computer Science, November 1996 (volume 168 pgs 215-240). This UTM simulate a Turing-complete class of tag-systems. A tag-system transforms strings over an alphabet A = {a[1],a[2],...,a[n], a[n+1]} as follows: a positive integer m is chosen, and so is a function P that maps each a[i] for 1<=i<=n to a string P(a[i]) over the alphabet A. Now:

1. if the string being transformed has fewer than m elements, the whole process stops now.
2. m elements are removed from the beginning of the string
3. call the first element removed a[k]; if k=n+1 the whole process stops now.
4. P(a[k]) is appended to the string.
5. steps 1-5 are repeated.

The particular class of tag-systems this Turing machine simulates is the class where m = 2, the initial string has length at least 2, and all P(a[i]) where 1<=i<=n are of the form a[n]a[n]B[i] where B[i] is some string over the alphabet A (B[i] is the empty string if and only if i=n).

The input for this Turing machine is mildly complex, and there is no error checking.

```
* The representation of a symbol a[i] from the alphabet A is a string
  of 1s which is one element longer than twice the combined length of
  all P(a[j]) where 1<=j<i.

* a value like P(a[i]) = a[n]a[n]a[w]a[x]...a[y]a[z] is represented as
  follows:
      b 1
      b 111...(as many as required to represent a[z] as described above) b
      b 111...(to represent a[y] as described above) b
      .
      .
      .
      b 111...(to represent a[x]) b
      b 111...(to represent a[w]) b
      b 111...(to represent a[n]) b
      b 111...(as many as for a[n] as described above, MINUS the
      number of 1s that represent a[i]; and no final b)

* The function P is represented by listing all its outputs in the
  order P(a[n]), P(a[n-1]),...,P(a[2]),P(a[1]). The representation of
  P(a[n+1])=STOP is done for you.

* The initial string a[q]a[r]...a[s]a[t] to be transformed by the
  tag-system is represented as
      111...(as many as required to represent a[q] as above) c
      111...(to represent a[r]) c
      .
      .
      .
      111...(to represent a[s]) c
      111...(to represent a[t]; note that there is no final c.)

* The input to this program is a function P as described above, then a
  single b, then the initial string to be transformed. Run all the 1s,
  bs, and cs together in one line with nothing between, followed by a
  return.

* The output format, if the program terminates, is the same as the
  input format for the initial string, and represents the final state
  of the transformed string, with a final a[n+1] appended due to a
```

peculiarity of the Turing machine's algorithm.

An example input string

We take as an example the tag-system over the alphabet A = {a[1], a[2], a[3], a[4]}, where m = 2 and:
P(a[1]) = a[3]a[3]a[2]a[1]a[4]
P(a[2]) = a[3]a[3]a[1]
P(a[3]) = a[3]a[3]
P(a[4]) = STOP

It meets the criteria above; and when applied to the initial string a[2]a[1]a[1] it gives:

```
a[2]a[1]a[1]
        a[1]a[3]a[3]a[1]
                a[3]a[1]a[3]a[3]a[2]a[1]a[4]
                        a[3]a[3]a[2]a[1]a[4]a[3]a[3]
                                a[2]a[1]a[4]a[3]a[3]a[3]a[3]
                                        a[4]a[3]a[3]a[3]a[3]a[3]a[3]a[1]
                                                a[3]a[3]a[3]a[3]a[3]a[1]
```

and then it's done.

Now, the encoding:

a[1] is 1
a[2] is 11111111111
a[3] is 11111111111111111
a[4] is 111111111111111111111

P(a[1]) is b1 b1111111111111111111111b b1b b11111111111b b11111111111111111b b1111111111111111
P(a[2]) is b1 b1b b11111111111111111b b111111
P(a[3]) is b1 b11111111111111111b b

the initial string is 11111111111c1c1

and so the whole input is

b1 b11111111111111111b b

```
b1 b1b b1111111111111111b b111111
b1 b11111111111111111111b b1b b11111111111b b1111111111111111b b1111111111111111
b
11111111111c1c1
```

which when run together for input to the Turing machine becomes

```
b1b1111111111111111bbb1b1bb1111111111111111bb111111b1b11111111111111111111bb1bb1111111111
```

The Universal Turing Machine

The Universal Turing Machine (UTM) implemented by the BF program is a four state, six symbol (namely: '1','b','<','>','0', and 'c') Turing machine. For those interested, the state table of the machine is given in the following table. Each cell contains a 5-tupple representing the current state, the symbol read, the symbol written, the action (L stands for move left, R stands for move right and H stands for halt), and the new state.

```
11<L1 210R2 311R3 410R4
1b>R1 2b>L3 3b<R4 4bcL2
1>bL1 2><R2 3>bR3 4><R4
1<0R1 2<>L2 3< H 4<
10<L1 201L2 30cR1 40cL2
1c0R4 2cbR2 3c1R1 4cbR4
```

The initial state is 1, tape cells are set as per the input but with the termination code P(a[n+1])=STOP represented as a <b at the left end and all other cells on both sides (actually, only the right is needed) holding symbol 0, and the head is initially at the first 1 in the code for the initial string.

The minimal test case b1b1bbb1c1c11111 represents the tag-system where P(a[1]) = a[1]a[1] and P(a[2]) = STOP, applied to the string a[1]a[1]a[2]. This runs for 518 steps of the Turing machine, exercising all 23 Turing machine instructions, before halting with the output string a[1].

BF program

Below follows the BF program which implements the above Universal Turing Machine. (For a detailed explaination of this program, read the original source file <http://www.hevanet.com/cristofd/brainfuck/utm.b> by Daniel Cristofani.)

```
+++>++>>>+[>>,[>+++++<[[->]<<]<[>]>]>-[<<+++++>>-[<<---->>-[->]<]]
<[<-<[<]+<+[>]<<+>->>>]<]<[<]>[-[>++++++<-]>[<+>-]+<<<+++>+>
  [-
    [<<+>->-
      [<<[-]>>-
        [<<++>+>-
          [<<-->->>+++<-
            [<<+>+>>--<-
              [<<->->-
                [<<++++>+>>+<-
                  [>-<-
                    [<<->->-
                      [<<->>-
                        [<<+++>>>-<-
                          [<<---->>>++<-
                            [<<++>>>+<-
                              [>[-]<-
                                [<<->>>+++<-
                                  [<<->>>--<-
                                    [<<++++>+>>+<-
                                      [<<[-]>->>++<-
                                        [<<+++++>+>>--<-
                                          [<->>++<
                                            [<<->>-
]]]]]]]]]]]]]]]]]]]]]]]]<[->>[<<+>>-]<<<[>>>+<<<-]<[>>>+<<<-]]>>]
>[-[---[-<]]>]>[+++[<+++++>--]>]+<++[[>+++++<-]<]>>[-.>]
```

URM is BF-complete

URM stands for Universal Register Machine. The statement "URM is BF-complete", simply means that all possible URM programs have an equivalent BF program, if each memory cell may contain an arbitrary

large number. Actually, the set of possible BF programs is much larger than the set of possible URM programs. You would hardly believe it, but URM is also Turing-complete, due to the fact that the registers can store infinite numbers, although the mapping from general Turing machines to URM is very, very complicated. Based on the ideas for a UTM in BF <#UTM>, Daniel Cristofani wrote a UTM for URM <http://www.hevanet.com/cristofd/brainfuck/urmutm.txt> using only five registers.) These fact combined, proof that BF is Turing-complete.

The Universal Register Machine is a machine with a fixed number of characters, and it supports the following commands:

  * a/n/ : increment register /n/
  * s/n/ : decrement register /n/
  * /x/; /y/ : execute command /x/ and then /y/
  * (/x/)/n/ : execute command /x/ while register /n/ is nonzero
  * . : ("dot blank") halt the machine.

Examples URM programs are:

  * Add register 3 to register 2:

        (a2;s3)3.

  * Multiply register 2 with register 3:

        (a4;a5;s2)2; ((a2;s4)4; s3; (a1;a4;s5)5; (a5;s1)1)3.

There is almost a one-to-one mapping from URM to BF. The a/n/ expression maps to +, the s/n/ expression maps to -, and the (/x/)/n/ expression maps to [/x/], of course preceded with the right amouth of > and < to move to the indicated memory location. The translation of the above two URM programs to BF are:

  * Add register 3 to register 2:

        >>>[<+>-]<<<

  * Multiply register 2 with register 3:

        >>[>>+>+<<<-]>[>[<<+>>-]<->>[<<<<+>>>+>-]<<<<[>>>>+<<<<-]>>]<<<

```
It is possible to make context free grammars for both BF and URM, such
that matching parse trees indicate equivalent programs. Such a grammar
is given below:

URM BF
/root/ = /S_1 /.  /root/ = /S_1 /
/S_n / = a/n/  /S_n / = +
/S_n / = s/n/  /S_n / = -
/S_n / = /S_n / ; /S_n /  /S_n / = /S_n / /S_n /
/S_n / = ( /S_n / )/n/  /S_n / = [ /S_n / ]
/S_n / = /S_n+1 /  /S_n / = > /S_n+1 / <
/S_n / = /S_n-1 /  /S_n / = < /S_n-1 / >

Using this mapping, one can create another UTM in BF
<http://www.hevanet.com/cristofd/brainfuck/urmutm.b> from the above
mentioned UTM in URM.
```

Still, I feel that BF is far more elegant than URM, because it gives you more with less symbols. BF only needs the symbols "+", "-", "<", ">", "[", and "]", whereas URM needs "a", "s", ";", "(", ")", ".", and "0" to "9". Although, I have to admit, that BF programs will in general be longer than the equivalent URM program, and in most cases also more cryptic.

```
;compliant version, non-commands are ignored, but 104 bytes long

[bits 16]
[org 0x100]
; assume bp=091e used
; assume di=fffe
; assume si=0100
; assume dx=cs (see here)
; assume cx=00ff
; assume bx=0000
; assume ax=0000 used (ah)
; assume sp=fffe
start:
        mov al, code_nothing - start
code_start:
        mov ch, 0x7f ; allow bigger programs
        mov bx, cx
```

```
        mov di, cx
        rep stosb
        mov bp, find_right + start - code_start ;cache loop head for smaller compiled progra
        jmp code_start_end
find_right:
        pop si
        dec si
        dec si ;point to loop head
        cmp [bx], cl
        jne loop_right_end
loop_right:
        lodsb
        cmp al, 0xD5 ; the "bp" part of "call bp" (because 0xFF is not unique, watch for add
        jne loop_left
        inc cx
loop_left:
        cmp al, 0xC3 ; ret (watch for ']')
        jne loop_right
        loop loop_right ;all brackets matched when cx==0
        db 0x3c ;cmp al, xx (mask push)
loop_right_end:
        push si
        lodsw ; skip "call" or dummy "dec" instruction, depending on context
        push si
code_sqright:
        ret
code_dec:
        dec byte [bx]
code_start_end:
        db '$' ;end DOS string, also "and al, xx"
code_inc:
        inc byte [bx]
        db '$'
code_right:
        inc bx ;al -> 2
code_nothing:
        db '$'
code_left:
        dec bx
        db '$'
code_sqleft:
```

```
        call bp
        db '$'
; create lookup table
real_start:
        inc byte [bx+'<'] ;point to code_left
        dec byte [bx+'>'] ;point to code_right
        mov byte [bx+'['], code_sqleft - start
        mov byte [bx+']'], code_sqright - start
        lea sp, [bx+45+2] ;'+' + 4 (2b='+', 2c=',', 2d='-', 2e='.')
        push (code_dec - start) + (code_dot - start) * 256
        push (code_inc - start) + (code_comma - start) * 256
pre_write:
        mov ah, code_start >> 8
        xchg dx, ax
; write
        mov ah, 9
        int 0x21
; read
code_comma:
        mov dl, 0xff
        db 0x3d ; cmp ax, xxxx (mask mov)
code_dot:
        mov dl, [bx]
        mov ah, 6
        int 0x21
        mov [bx], al
        db '$'
        db 0xff ; parameter for '$', doubles as test for zero
; switch
        xlatb
        jne pre_write
  ; next two lines can also be removed
  ; if the program ends with extra ']'
  ; and then we are at 100 bytes... :-)
the_end:
        mov dl, 0xC3
        int 0x21
        int 0x20
```

Masque of the Red Death

The Masque of the Red Death

THE "Red Death" had long devastated the country. No pestilence had ever been so fatal, or so hideous. Blood was its Avatar and its seal – the redness and the horror of blood. There were sharp pains, and sudden dizziness, and then profuse bleeding at the pores, with dissolution. The scarlet stains upon the body and especially upon the face of the victim, were the pest ban which shut him out from the aid and from the sympathy of his fellow-men. And the whole seizure, progress and termination of the disease, were the incidents of half an hour.

But the Prince Prospero was happy and dauntless and sagacious. When his dominions were half depopulated, he summoned to his presence a thousand hale and light-hearted friends from among the knights and dames of his court, and with these retired to the deep seclusion of one of his castellated abbeys. This was an extensive and magnificent structure, the creation of the prince's own eccentric yet august taste. A strong and lofty wall girdled it in. This wall had gates of iron. The courtiers, having entered, brought furnaces and massy hammers and welded the bolts. They resolved to leave means neither of ingress or egress to the sudden impulses of despair or of frenzy from within. The abbey was amply provisioned. With such precautions the courtiers might bid defiance to contagion. The external world could take care of itself. In the meantime it was folly to grieve, or to think. The prince had provided all the appliances of pleasure. There were buffoons, there were improvisatori, there were ballet-dancers, there were musicians, there was Beauty, there was wine. All these and security were within. Without was the "Red Death."

It was toward the close of the fifth or sixth month of his seclusion, and while the pestilence raged most furiously abroad, that the Prince Prospero entertained his thousand friends at a masked ball of the most unusual magnificence.

It was a voluptuous scene, that masquerade. But first let me tell of the rooms in which it was held. There were seven – an imperial suite. In many palaces, however, such suites form a long and straight vista, while the folding doors slide back nearly to the walls on either hand, so that the view of the whole extent is scarcely impeded. Here the case was very different; as might have been expected from the duke's love of the bizarre. The apartments were so irregularly disposed that the vision embraced but little more than one at a time. There was a sharp turn at every twenty or thirty yards, and at each turn a novel effect. To the right and left, in the middle of each wall, a tall and narrow Gothic window looked out upon a closed corridor which pursued the windings of the suite. These windows were of stained glass whose color varied in accordance with the prevailing hue of the decorations of the chamber into which it opened. That at the eastern extremity was hung, for example, in blue – and vividly blue were its windows. The second chamber was purple in its ornaments and tapestries, and here the panes were purple. The third was green throughout, and so were the casements. The fourth was furnished and lighted with orange – the fifth with white – the sixth with violet. The seventh apartment was closely shrouded in black velvet tapestries that hung all over the ceiling and down the walls, falling in heavy folds upon a

carpet of the same material and hue. But in this chamber only, the color of the windows failed to correspond with the decorations. The panes here were scarlet – a deep blood color. Now in no one of the seven apartments was there any lamp or candelabrum, amid the profusion of golden ornaments that lay scattered to and fro or depended from the roof. There was no light of any kind emanating from lamp or candle within the suite of chambers. But in the corridors that followed the suite, there stood, opposite to each window, a heavy tripod, bearing a brazier of fire that protected its rays through the tinted glass and so glaringly illumined the room. And thus were produced a multitude of gaudy and fantastic appearances. But in the western or black chamber the effect of the fire-light that streamed upon the dark hangings through the blood-tinted panes, was ghastly in the extreme, and produced so wild a look upon the countenances of those who entered, that there were few of the company bold enough to set foot within its precincts at all.

It was in this apartment, also, that there stood against the western wall, a gigantic clock of ebony. Its pendulum swung to and fro with a dull, heavy, monotonous clang; and when the minute-hand made the circuit of the face, and the hour was to be stricken, there came from the brazen lungs of the clock a sound which was clear and loud and deep and exceedingly musical, but of so peculiar a note and emphasis that, at each lapse of an hour, the musicians of the orchestra were constrained to pause, momentarily, in their performance, to hearken to the sound; and thus the waltzers perforce ceased their evolutions; and there was a brief disconcert of the whole gay company; and, while the chimes of the clock yet rang, it was observed that the giddiest grew pale, and the more aged and sedate passed their hands over their brows as if in confused reverie or meditation. But when the echoes had fully ceased, a light laughter at once pervaded the assembly; the musicians looked at each other and smiled as if at their own nervousness and folly, and made whispering vows, each to the other, that the next chiming of the clock should produce in them no similar emotion; and then, after the lapse of sixty minutes, (which embrace three thousand and six hundred seconds of the Time that flies,) there came yet another chiming of the clock, and then were the same disconcert and tremulousness and meditation as before.

But, in spite of these things, it was a gay and magnificent revel. The tastes of the duke were peculiar. He had a fine eye for colors and effects. He disregarded the decora of mere fashion. His plans were bold and fiery, and his conceptions glowed with barbaric lustre. There are some who would have thought him mad. His followers felt that he was not. It was necessary to hear and see and touch him to be sure that he was not.

He had directed, in great part, the moveable embellishments of the seven chambers, upon occasion of this great fete; and it was his own guiding taste which had given character to the masqueraders. Be sure they were grotesque. There were much glare and glitter and piquancy and phantasm – much of what has been since seen in "Hernani." There were arabesque figures with unsuited limbs and appointments. There were delirious fancies such as the madman fashions. There was much of the beautiful, much of the wanton, much of the bizarre, something of the terrible, and not a little of that which might have excited

disgust. To and fro in the seven chambers there stalked, in fact, a multitude of dreams. And these – the dreams – writhed in and about, taking hue from the rooms, and causing the wild music of the orchestra to seem as the echo of their steps. And, anon, there strikes the ebony clock which stands in the hall of the velvet. And then, for a moment, all is still, and all is silent save the voice of the clock. The dreams are stiff-frozen as they stand. But the echoes of the chime die away – they have endured but an instant – and a light, half-subdued laughter floats after them as they depart. And now again the music swells, and the dreams live, and writhe to and fro more merrily than ever, taking hue from the many-tinted windows through which stream the rays from the tripods. But to the chamber which lies most westwardly of the seven, there are now none of the maskers who venture; for the night is waning away; and there flows a ruddier light through the blood-colored panes; and the blackness of the sable drapery appals; and to him whose foot falls upon the sable carpet, there comes from the near clock of ebony a muffled peal more solemnly emphatic than any which reaches their ears who indulge in the more remote gaieties of the other apartments.

But these other apartments were densely crowded, and in them beat feverishly the heart of life. And the revel went whirlingly on, until at length there commenced the sounding of midnight upon the clock. And then the music ceased, as I have told; and the evolutions of the waltzers were quieted; and there was an uneasy cessation of all things as before. But now there were twelve strokes to be sounded by the bell of the clock; and thus it happened, perhaps, that more of thought crept, with more of time, into the meditations of the thoughtful among those who revelled. And thus, too, it happened, perhaps, that before the last echoes of the last chime had utterly sunk into silence, there were many individuals in the crowd who had found leisure to become aware of the presence of a masked figure which had arrested the attention of no single individual before. And the rumor of this new presence having spread itself whisperingly around, there arose at length from the whole company a buzz, or murmur, expressive of disapprobation and surprise – then, finally, of terror, of horror, and of disgust.

In an assembly of phantasms such as I have painted, it may well be supposed that no ordinary appearance could have excited such sensation. In truth the masquerade license of the night was nearly unlimited; but the figure in question had out-Heroded Herod, and gone beyond the bounds of even the prince's indefinite decorum. There are chords in the hearts of the most reckless which cannot be touched without emotion. Even with the utterly lost, to whom life and death are equally jests, there are matters of which no jest can be made. The whole company, indeed, seemed now deeply to feel that in the costume and bearing of the stranger neither wit nor propriety existed. The figure was tall and gaunt, and shrouded from head to foot in the habiliments of the grave. The mask which concealed the visage was made so nearly to resemble the countenance of a stiffened corpse that the closest scrutiny must have had difficulty in detecting the cheat. And yet all this might have been endured, if not approved, by the mad revellers around. But the mummer had gone so far as to assume

the type of the Red Death. His vesture was dabbled in blood – and his broad brow, with all the features of the face, was besprinkled with the scarlet horror.

When the eyes of Prince Prospero fell upon this spectral image (which with a slow and solemn movement, as if more fully to sustain its role, stalked to and fro among the waltzers) he was seen to be convulsed, in the first moment with a strong shudder either of terror or distaste; but, in the next, his brow reddened with rage.

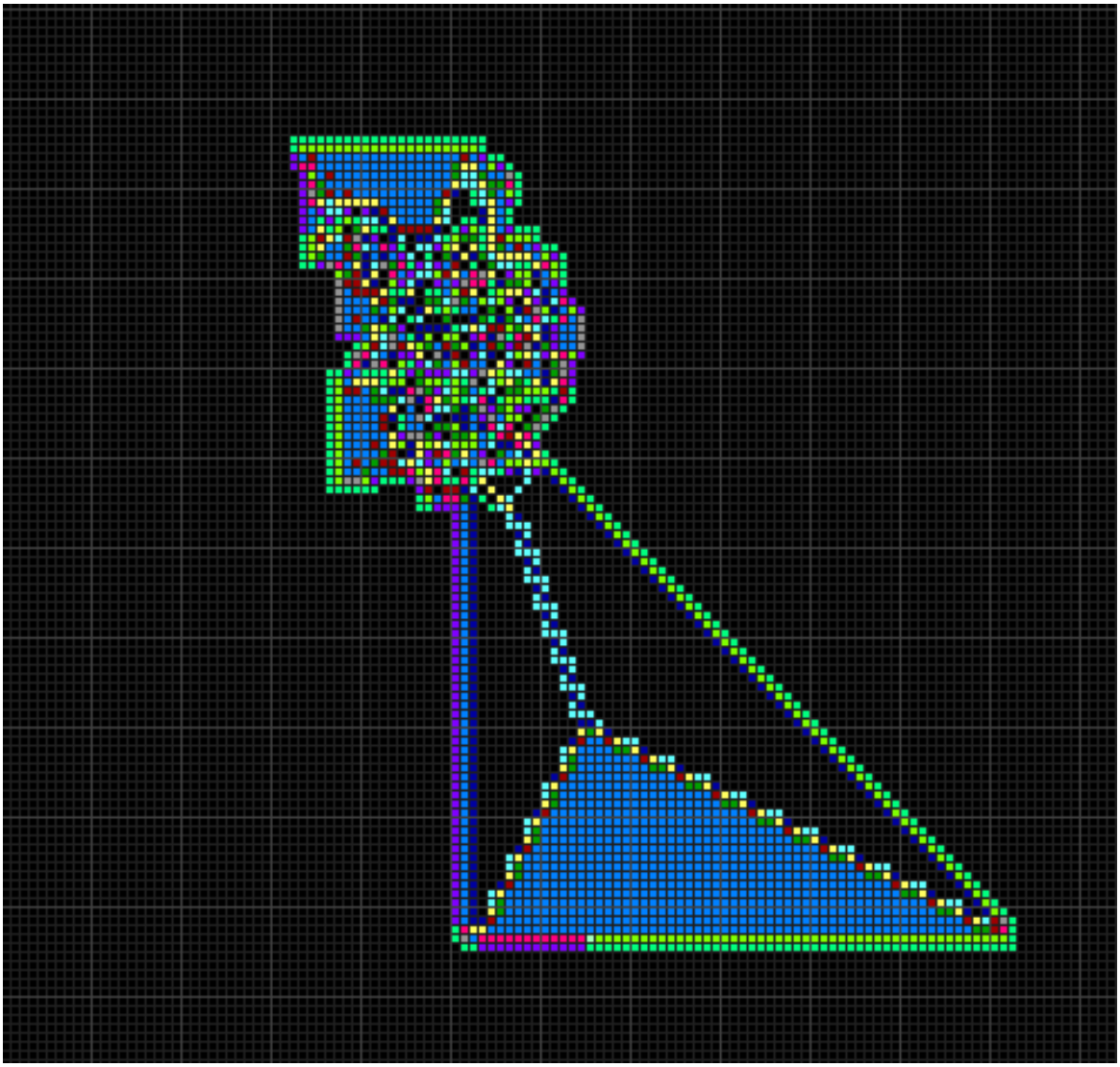
"Who dares?" he demanded hoarsely of the courtiers who stood near him – "who dares insult us with this blasphemous mockery? Seize him and unmask him – that we may know whom we have to hang at sunrise, from the battlements!"

It was in the eastern or blue chamber in which stood the Prince Prospero as he uttered these words. They rang throughout the seven rooms loudly and clearly – for the prince was a bold and robust man, and the music had become hushed at the waving of his hand.

It was in the blue room where stood the prince, with a group of pale courtiers by his side. At first, as he spoke, there was a slight rushing movement of this group in the direction of the intruder, who at the moment was also near at hand, and now, with deliberate and stately step, made closer approach to the speaker. But from a certain nameless awe with which the mad assumptions of the mummer had inspired the whole party, there were found none who put forth hand to seize him; so that, unimpeded, he passed within a yard of the prince's person; and, while the vast assembly, as if with one impulse, shrank from the centres of the rooms to the walls, he made his way uninterruptedly, but with the same solemn and measured step which had distinguished him from the first, through the blue chamber to the purple – through the purple to the green – through the green to the orange – through this again to the white – and even thence to the violet, ere a decided movement had been made to arrest him. It was then, however, that the Prince Prospero, maddening with rage and the shame of his own momentary cowardice, rushed hurriedly through the six chambers, while none followed him on account of a deadly terror that had seized upon all. He bore aloft a drawn dagger, and had approached, in rapid impetuosity, to within three or four feet of the retreating figure, when the latter, having attained the extremity of the velvet apartment, turned suddenly and confronted his pursuer. There was a sharp cry – and the dagger dropped gleaming upon the sable carpet, upon which, instantly afterwards, fell prostrate in death the Prince Prospero. Then, summoning the wild courage of despair, a throng of the revellers at once threw themselves into the black apartment, and, seizing the mummer, whose tall figure stood erect and motionless within the shadow of the ebony clock, gasped in unutterable horror at finding the grave-cerements and corpse-like mask which they handled with so violent a rudeness, untenanted by any tangible form.

And now was acknowledged the presence of the Red Death. He had come like a thief in the night. And one by one dropped the revellers in the blood-bedewed halls of their revel, and died each in the despairing posture of his fall. And the life of the ebony clock went out with that of the last of the gay. And the flames of the tripods expired. And Darkness and Decay and the Red Death held illimitable dominion over all.

Langton's ant



Wireworld

Wireworld Project

De Wiki LOGre
< Projet Wireworld

Language:	Français • English

Sommaire

- 1 Vocabulary
- 2 Simulators
  - 2.1 Ressources sharing
    - 2.1.1 File formats
      - 2.1.1.1 Input file
      - 2.1.1.2 Generic configuration file
      - 2.1.1.3 Configuration file
  - 2.2 SystemC simulator
- 3 Wireworld Computer Reverse Engineering
  - 3.1 Methodology
  - 3.2 Clock system
    - 3.2.1 Clock injector
    - 3.2.2 Clock divider
    - 3.2.3 Implemented feature
  - 3.3 Digit display
    - 3.3.1 7 segments display
    - 3.3.2 ROM
    - 3.3.3 ROM model
    - 3.3.4 Contrôleur de ROM
    - 3.3.5 Operating
    - 3.3.6 ROM controller model
  - 3.4 Data latch
    - 3.4.1 Inputs/Outputs
    - 3.4.2 Internal architecture
    - 3.4.3 Operating
  - 3.5 Binary/BCD converter
    - 3.5.1 Inputs/Outputs
    - 3.5.2 Internal architecture
      - 3.5.2.1 Binary adder
      - 3.5.2.2 Digit selecter
        - 3.5.2.2.1 Inputs/Outputs
        - 3.5.2.2.2 Internal architecture
        - 3.5.2.2.3 Operating
      - 3.5.2.3 Overflow detection loop and pulse generator
      - 3.5.2.4 Pulse controller
      - 3.5.2.5 Mega loops
    - 3.5.3 Operating
      - 3.5.3.1 Principle

        - 3.5.3.2 Arithmetic
        - 3.5.3.3 Simulation
    - 3.6 Registers access controller
        - 3.6.1 Inputs/Outputs
        - 3.6.2 Internal architecture
            - 3.6.2.1 Burst generator of n electrons in 6 microns
            - 3.6.2.2 Electron doubler in 6 microns
            - 3.6.2.3 Delay generator in 6 microns
        - 3.6.3 Operating principle
    - 3.7 Control Unit
        - 3.7.1 Operating principle
        - 3.7.2 Implementation
            - 3.7.2.1 Inputs/Outputs
            - 3.7.2.2 Internal architecture
            - 3.7.2.3 Operating
                - 3.7.2.3.1 PC/Data register selecter
                - 3.7.2.3.2 PC incrementer
    - 3.8 Registers
        - 3.8.1 Register write
            - 3.8.1.1 Operating principle
            - 3.8.1.2 Implementation
        - 3.8.2 Register read
            - 3.8.2.1 Operating principle
            - 3.8.2.2 Implementation
        - 3.8.3 Special registers
            - 3.8.3.1 Adder register
            - 3.8.3.2 Conditional register
                - 3.8.3.2.1 Internal architecture
                - 3.8.3.2.2 Operating
        - 3.8.4 Register configuration
- 4 Wireworld computer
    - 4.1 Functional model
        - 4.1.1 Inputs/Outputs
        - 4.1.2 Assembly format
    - 4.2 Use
- 5 Conclusions

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

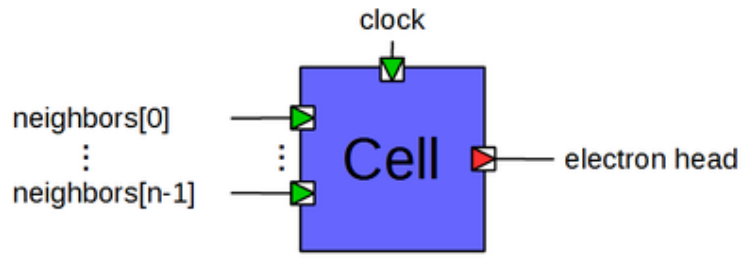
It complete the generic configuration file by defining if items are active or not.
File format is basic : **item_name:[0|1]**
Here is an example :

```
# 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

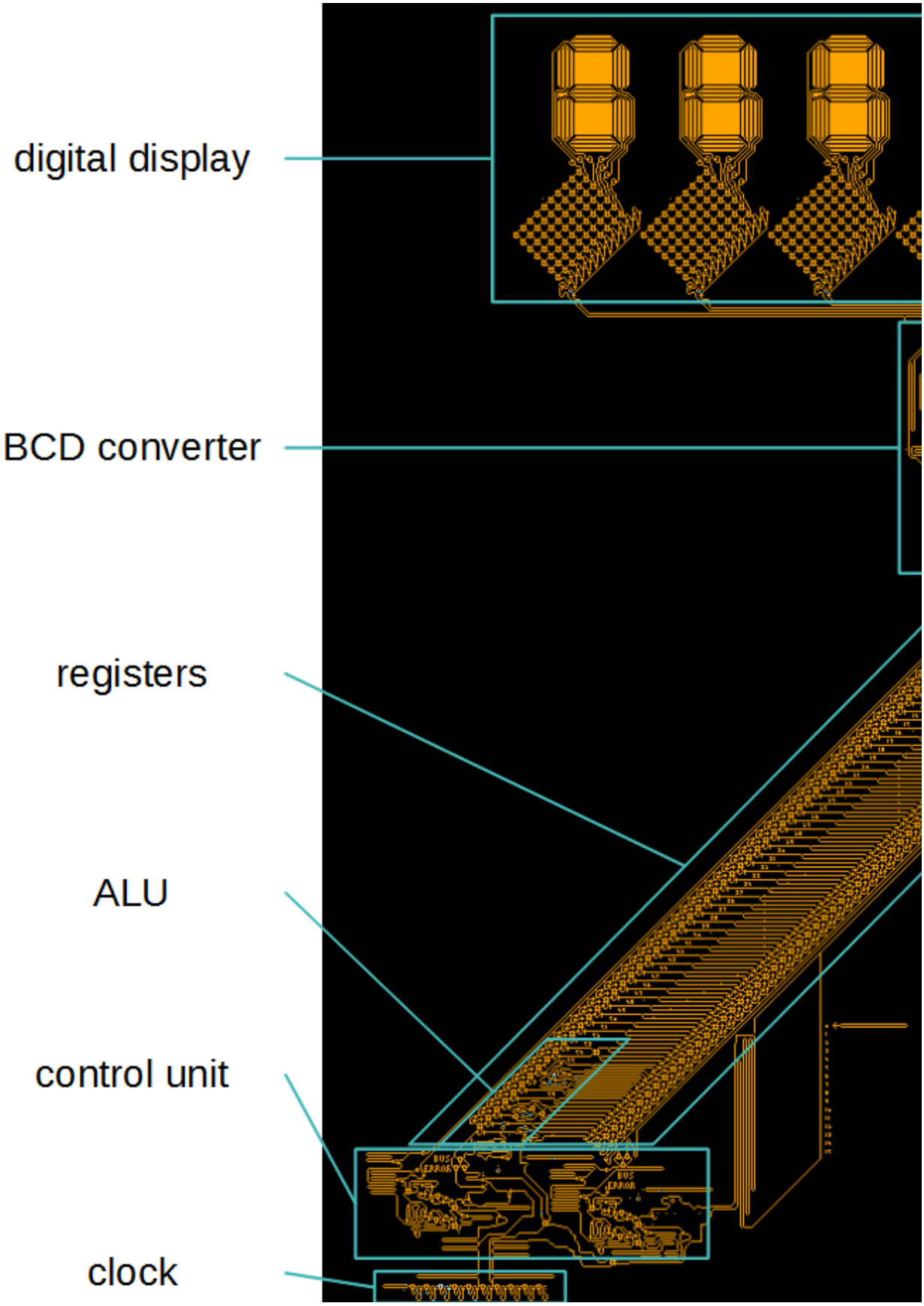
Output of neighbor cells are bound to cell's inputs.
In order to improve performances, SystemC module is templated on number of neighbor cells and the correct module type is instanciated by analyzer depending on neighbor number.

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

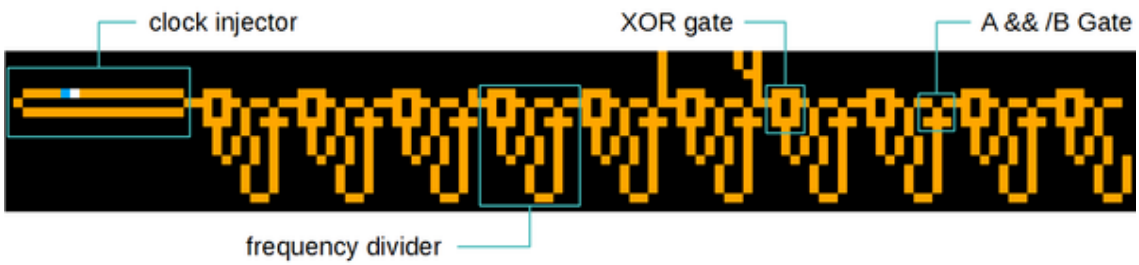
Methodology

- 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

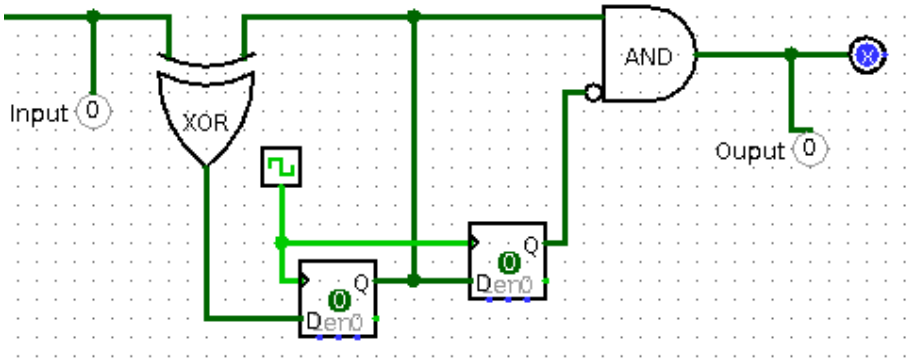
Clock divider

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**

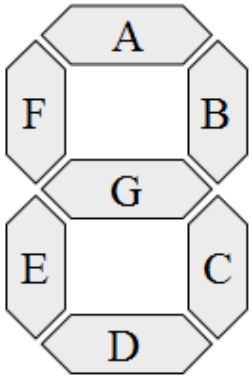
Digit display

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

- 1 7 segments display

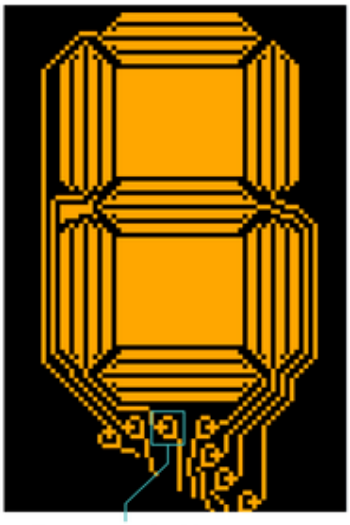
- 1 ROM with 10 inputs and 7 outputd
- 1 ROM controller

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:

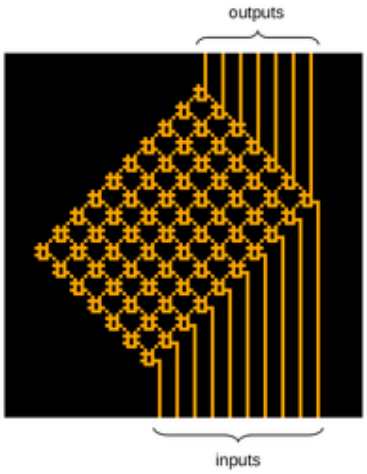
- Each incoming electron (period 6) is duplicated and delay of 3 generations before being reintroduced in an OR gate which generates 2 electronds with period 3



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.

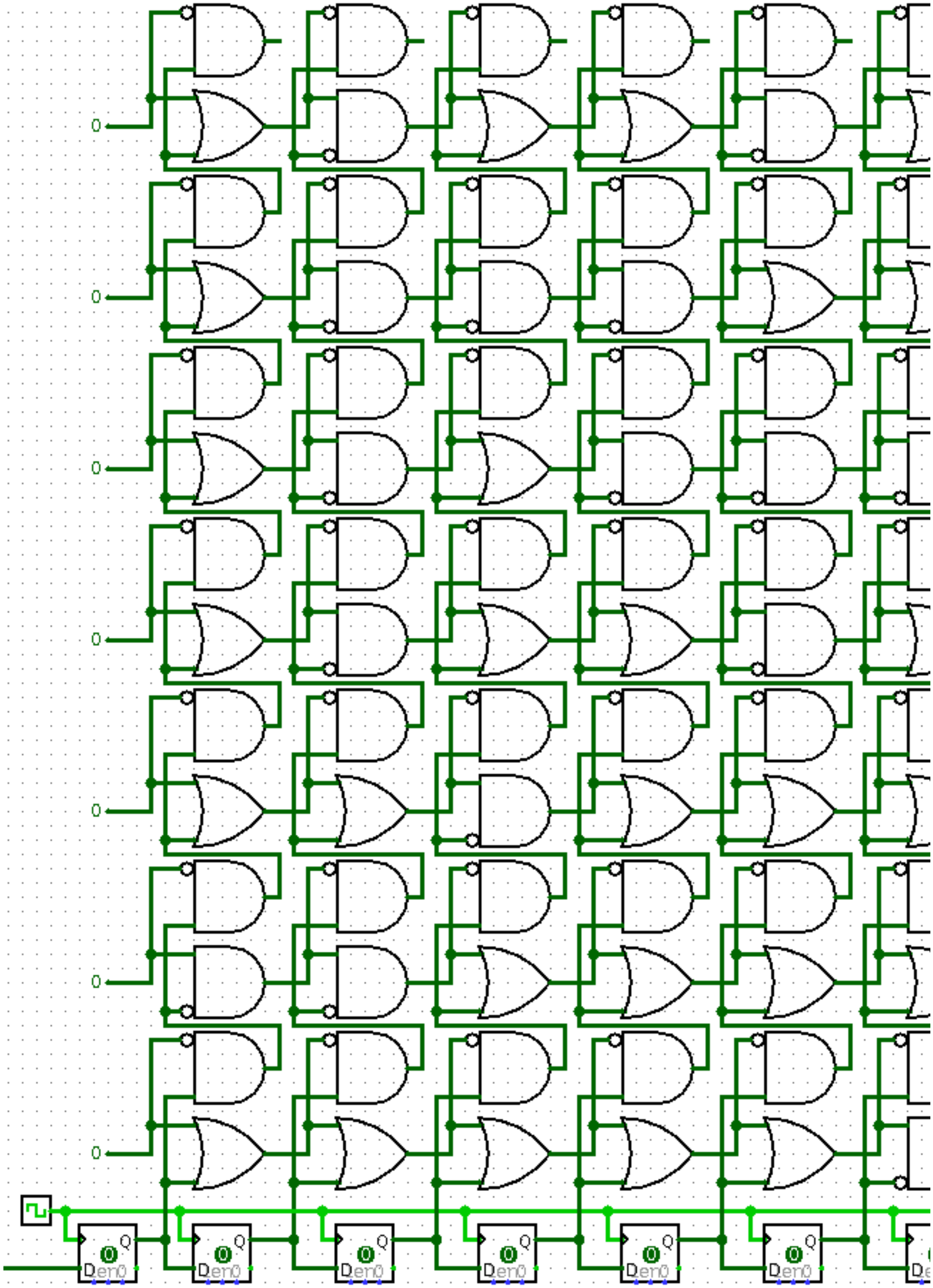
For one electron coming on a ROM input, one electron will be generated on each ouput of the ROM coding 1.
To maintain segments lighted the ROM input correspoding to the digit must be continuously supplied.
This feature is implemented by ROM controller



ROM model

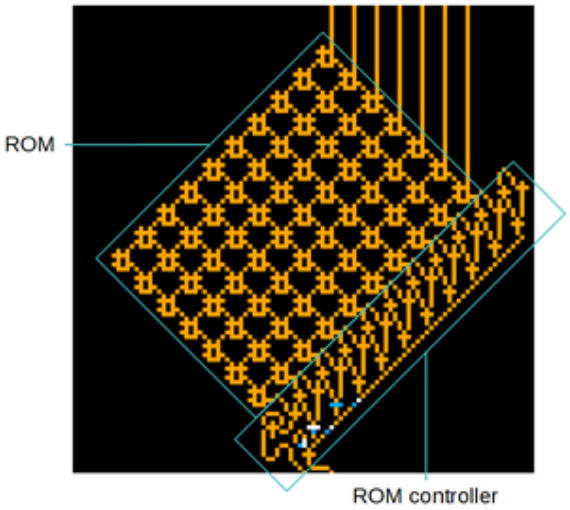
For this component electron propagation time is not meaningfull functionnally so they are not modelled.
The ROM is partially modelled: only digits 0 to 6 are managed

- 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.

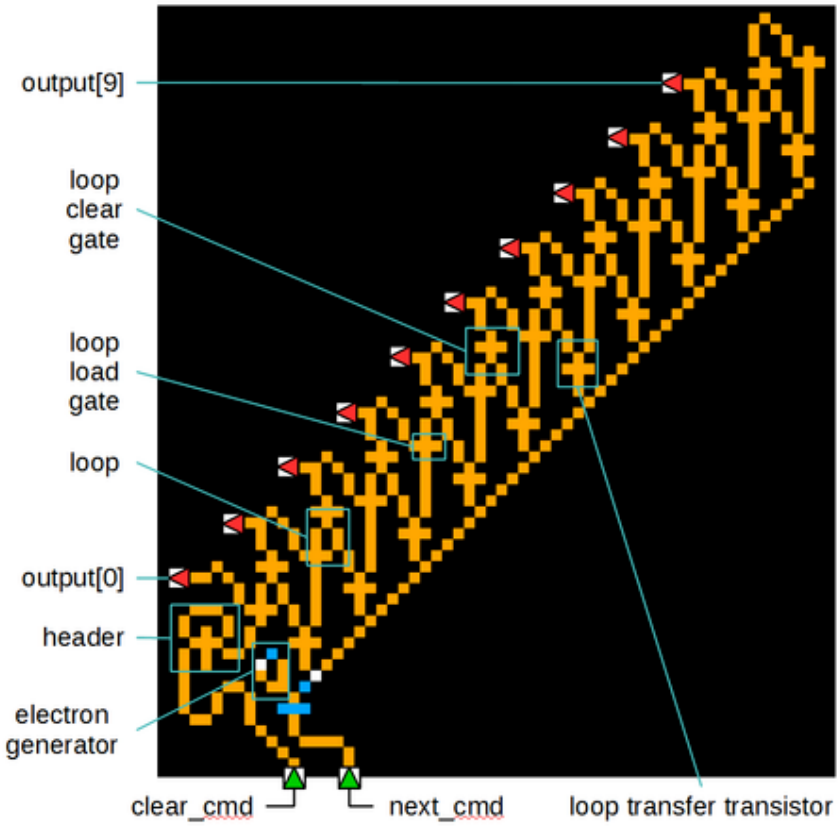
Remark:

- 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

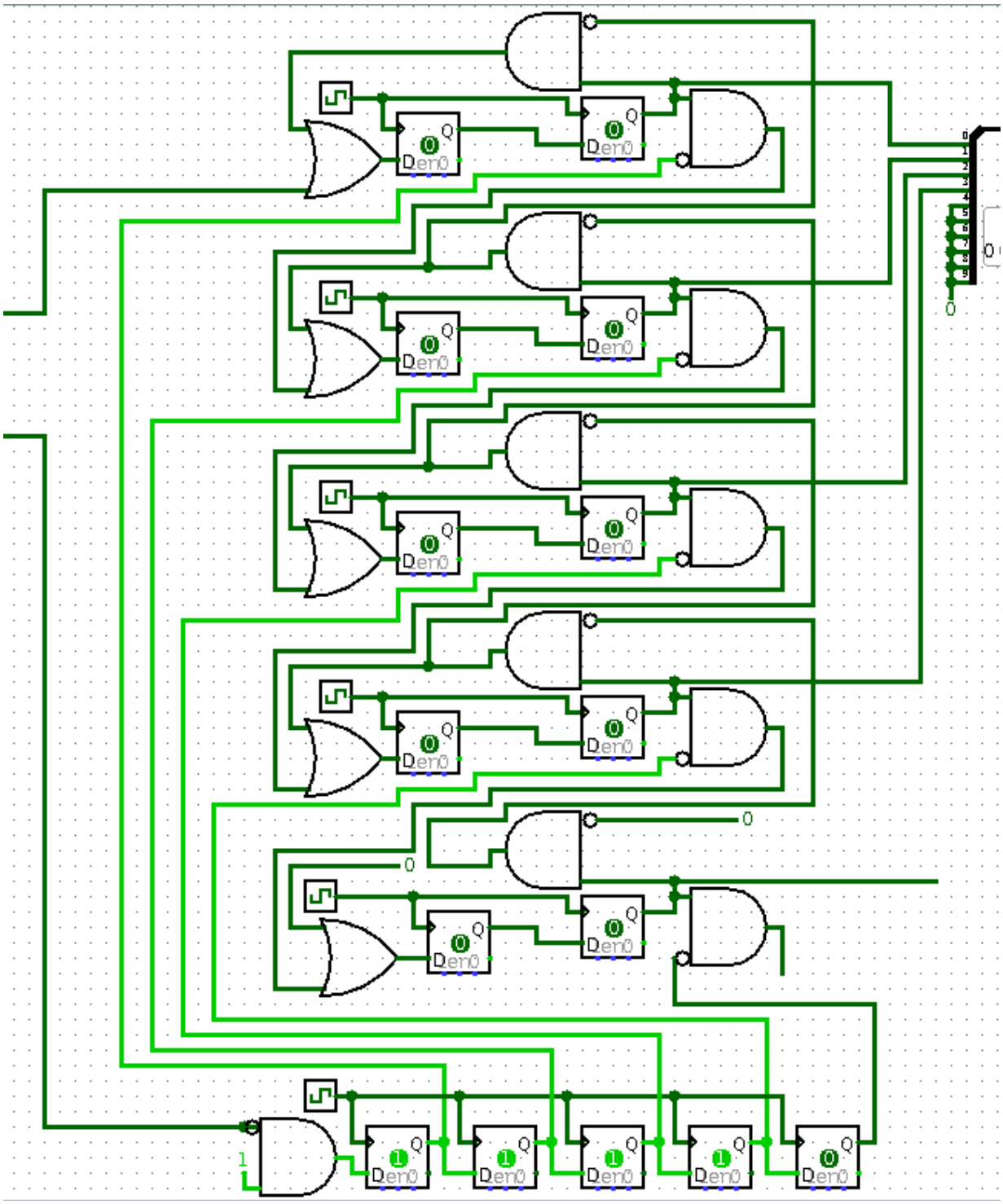
display.
- If we want to display 5 and the displayed digit is zerp then it is needed to send 5 electrons on loop transfer control wire to activate loop 5



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

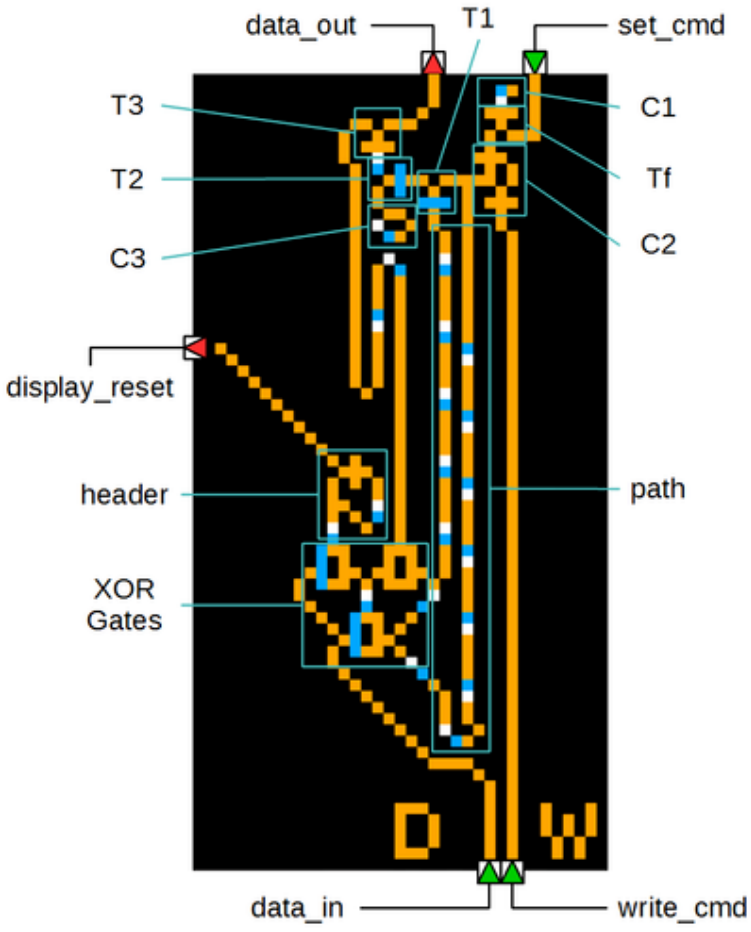
It has 2 outputs:

- 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

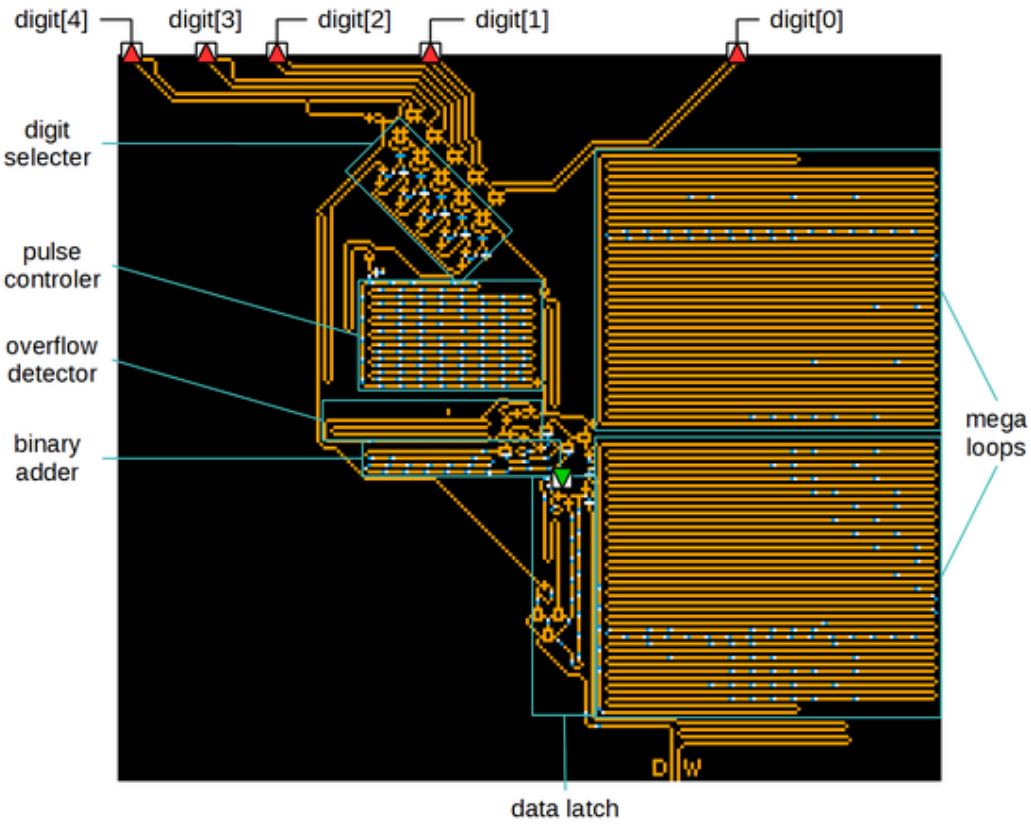
Binary/BCD converter is composed of several parts:

- 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

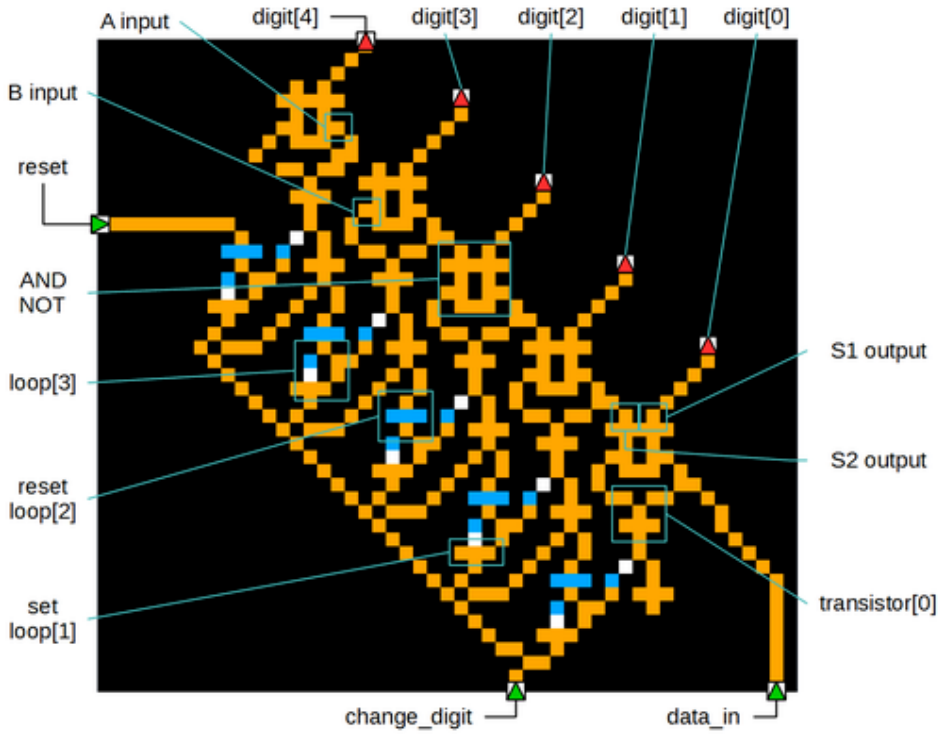
and 5 outputs:

- 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

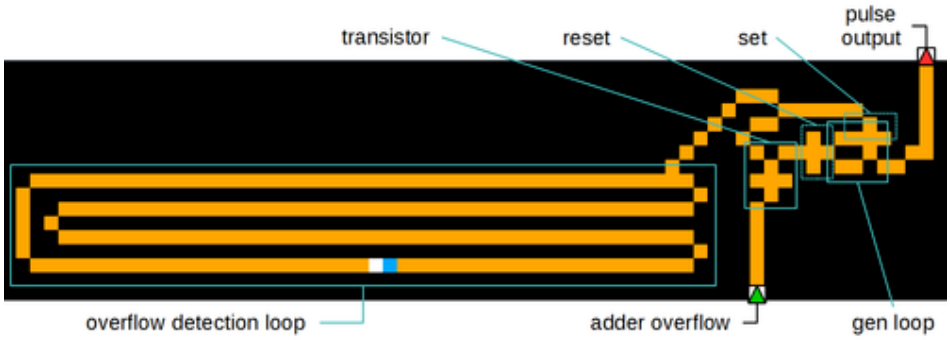
gate[0] has its **B input** bound to **pulse generator output** while **gates[n]** have their **B input** boudn to **S2 output** of **gate[n-1]**
By this way when **loop[n]** is active it locks **transistor[n]** so **gate[n]** has its **A input** at 0 so **S2 = B** making electrons coming from pulse 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

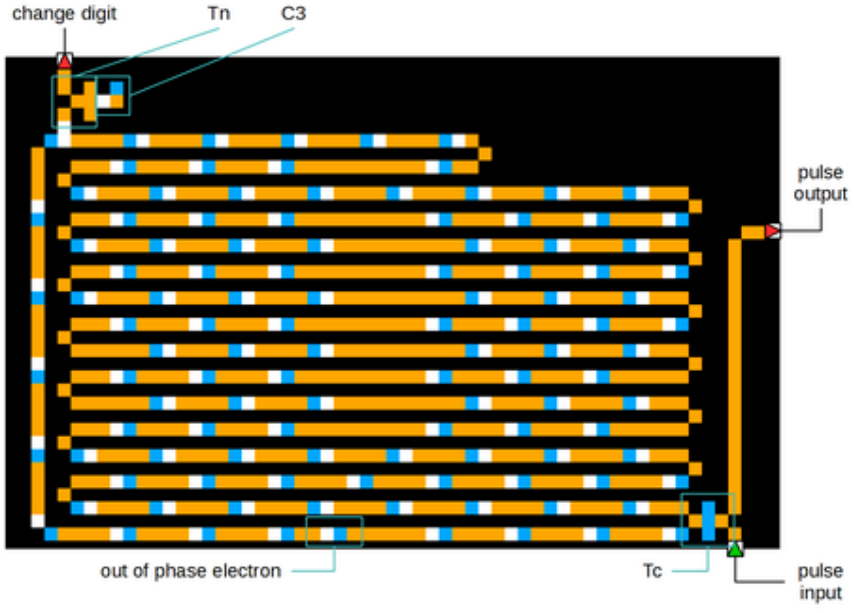
Pulse controller

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 période 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:

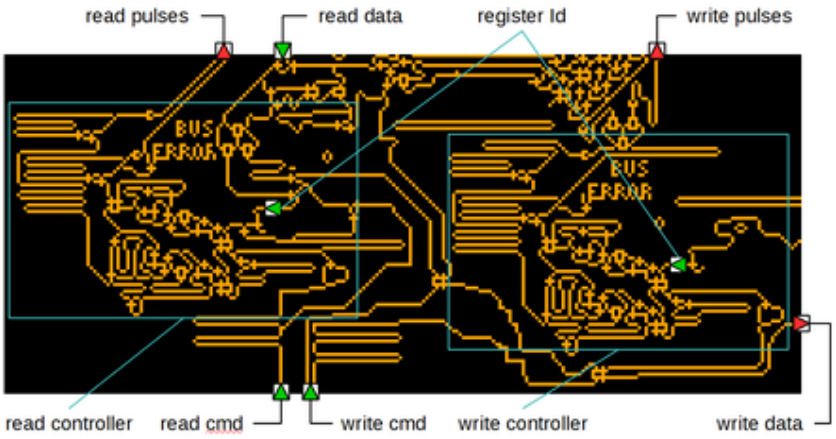
```
$ ./binary2bcd.exe 0x100
Input number is 256
Step[ 0]: carry : 0x00000000 & ~(Top_loop[ 0] : 0xffffff80) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 1]: carry : 0x00000000 & ~(Top_loop[ 1] : 0x00000000) => Adjusted carry 0x00000000 ^ Bot_loop[
Step[ 2]: carry : 0x00000000 & ~(Top_loop[ 2] : 0x00000000)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

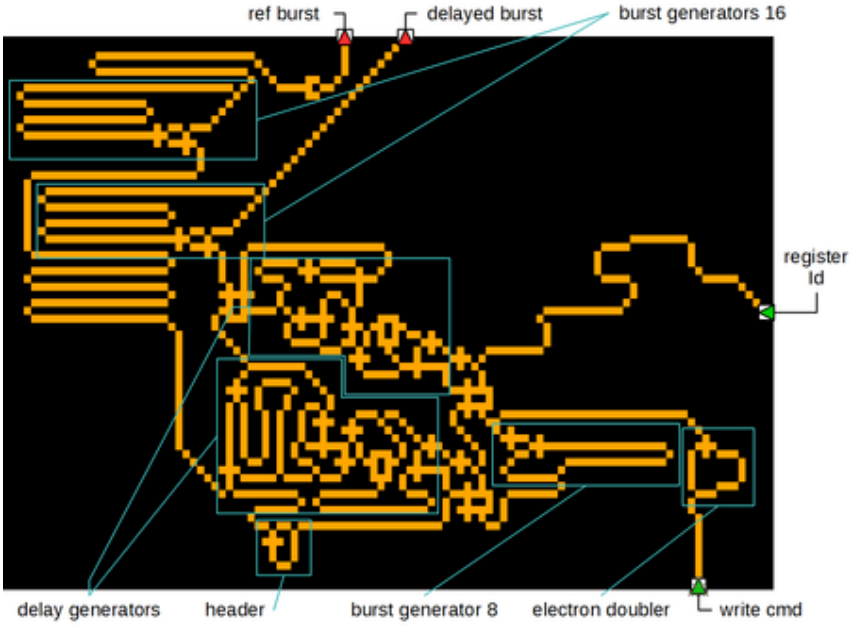


Wireworld Computer contains 64 registers 16 bits width, registers access controllers allow to select register to write in or to read from
There is a write access controller and a read access controller
Register selection is done by sending two burst of 16 electrons in 6 microns, one burst going in direction 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

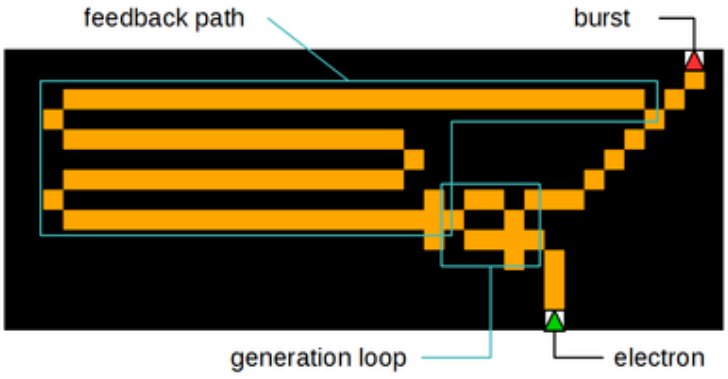
Each access controller is made up of the following elements:

- 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

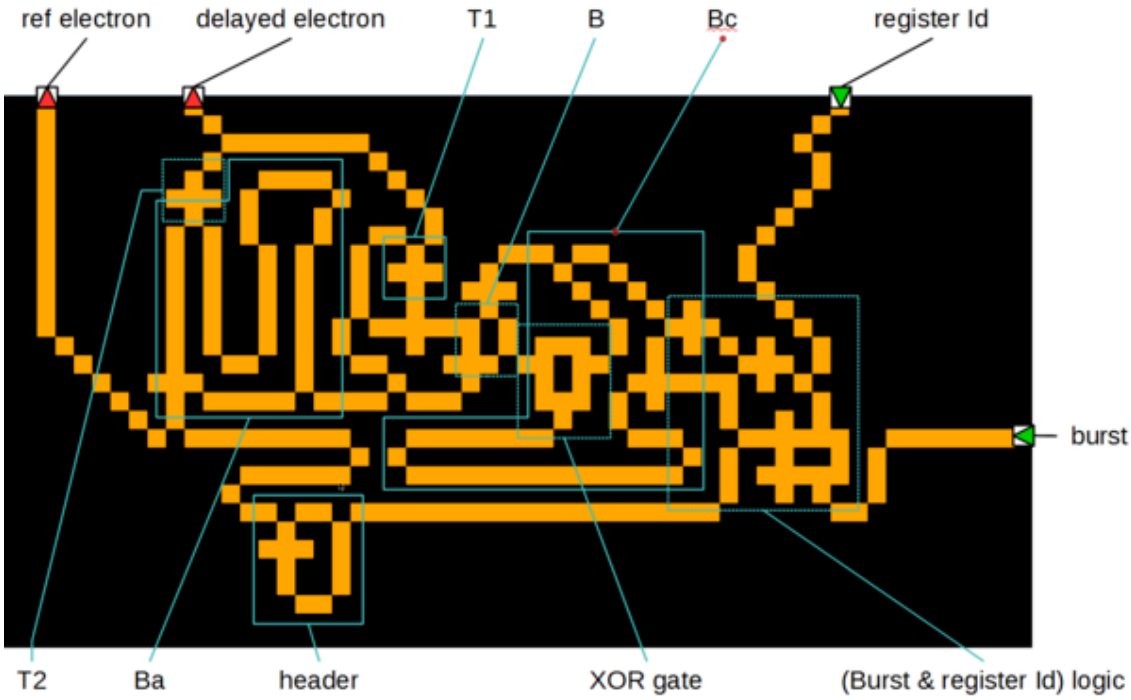
Delay generator in 6 microns

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 **période** 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

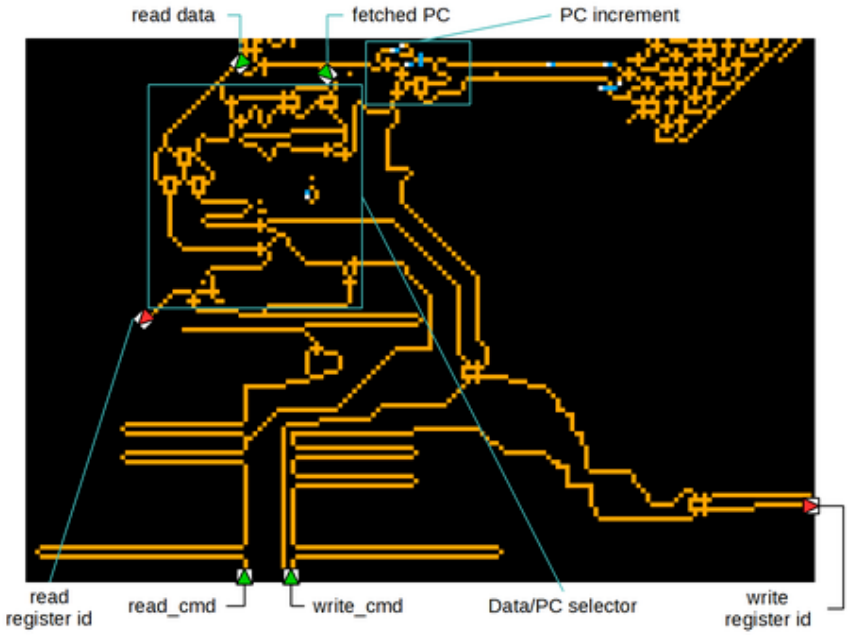
Operation 1	Operation 2	Operation 3	Read Register Id	Write Register Id	Data read	Data to write
Fetch(1/2)	Execute(1/2)	Decode(2/2)	PC value	Result from previous Fetch(2/2) = Rt	V = Content Register Rs	
Fetch(2/2)	Execute(2/2)	Decode(1/2)	Rs		Instruction pointed by PC	V

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

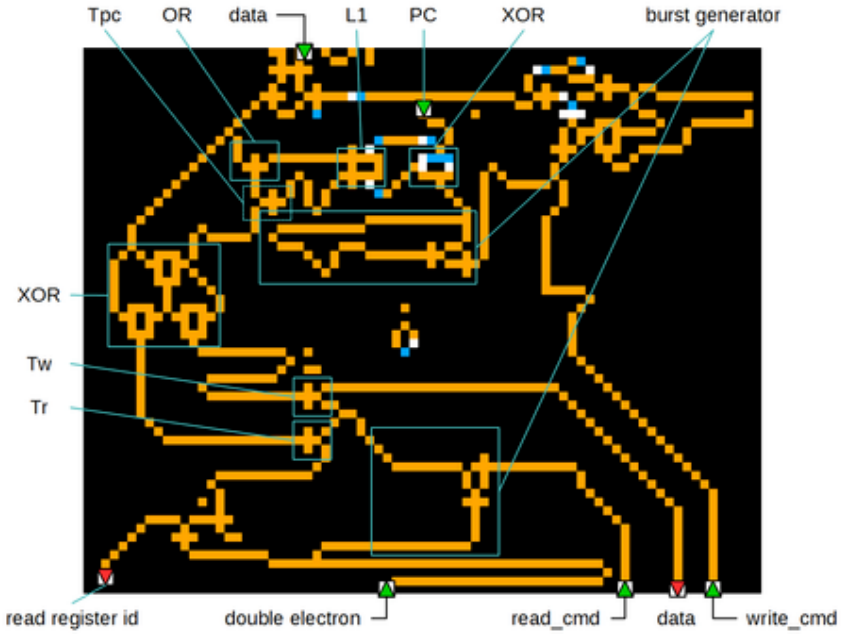
The path of Data bus Data is very long so that Data arrives at register input at the it is activated by rising and falling burst of register write access as defined by *Decode(2/2)*'.
During *Decode(1/2)* value is on Data bus is not an instruction so timings are computed so that it arrives on Register 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.

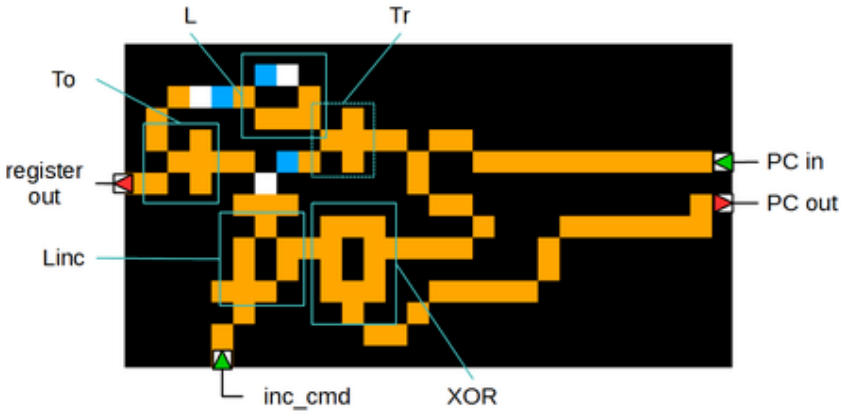


In case of **PC** fetch **OR** gate receive in input value coming from registers and **NOT(PC)**.
Tpc transistor is driven by **PC** do transistor output is **((NOT(DATA)) OR (NOT(PC))) & NOT(PC)**.

- 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

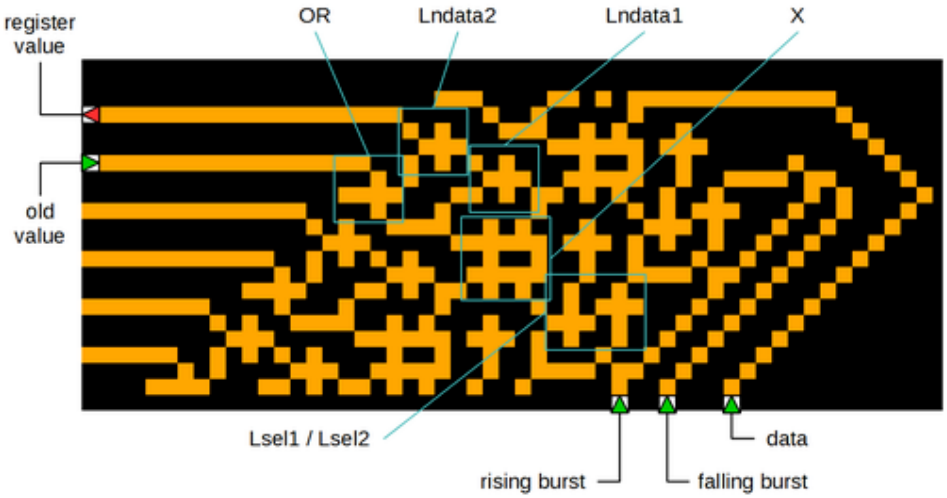
So cells located in input of each register should implemnt the following features:

- 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

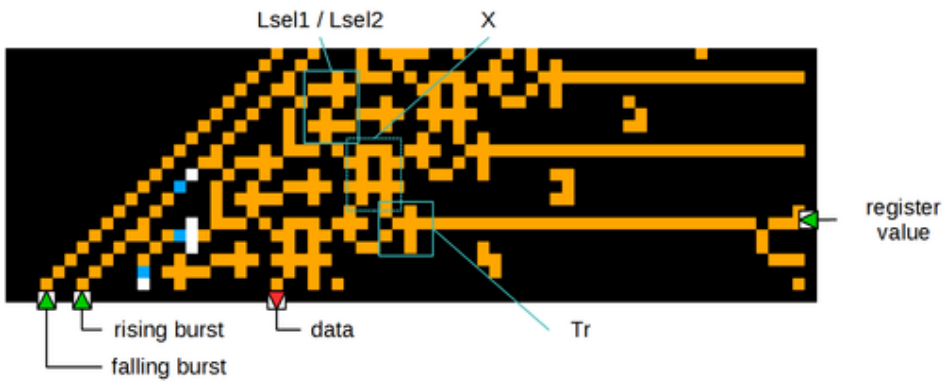
Cells located in output of each register should implement the following features:

- 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

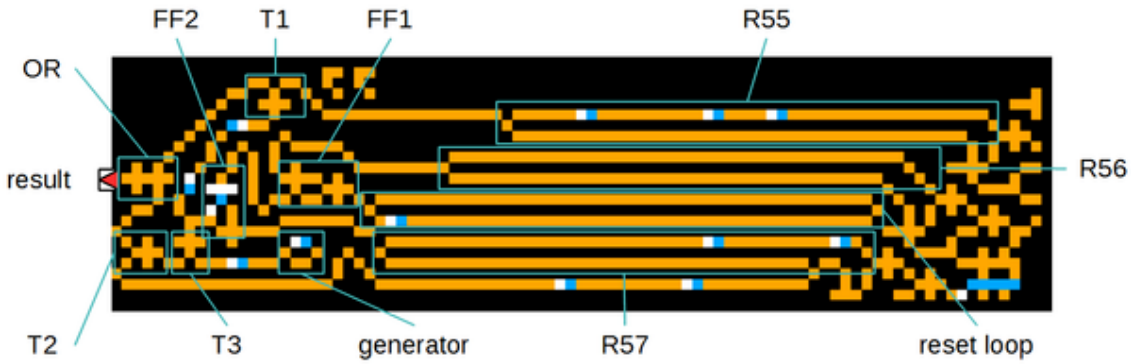
Read of Register R56 return R55 if R56 is not null else R57

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 operano
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

Catégories : Software | C++ | Projets

Further reading

Bernhardt, Chris. 2016. Turing's Vision: The Birth of Computer Science.

Matt J. Keeling & Pejman Rohani. 2007. Modeling Infectious Diseases in Humans and Animals.

Parikka, Jussi. 2007. Digital Contagions.