

Implémentation d'un ordinateur dans le jeu de la vie

Le but de ce document est de décrire le fonctionnement d'un ordinateur implémenté dans le jeu de la vie. La description exhaustive de tous les mécanismes et tout les constituants de l'ordinateur serait beaucoup trop longue pour tenir ici. Seuls les principes fondamentaux et les composants principaux sont décrits. Je pense que cela est suffisant pour permettre à n'importe qui de reproduire un ordinateur semblable ou de réutiliser les idées et les composants de l'ordinateur dans un autre projet.

L'idée est d'illustrer la Turing-complétude du jeu de la vie avec un exemple plus impressionnant, plus proche de nos ordinateurs et plus facile à programmer qu'une simple machine de Turing. Ce projet représente environ 50h de travail.

Le projet est disponible ici :

<https://drive.google.com/file/d/0B-4kRwE7luROR1VHclhJUUF3Qnc/view>

1. Bases
2. Portes logiques
3. Assemblage des portes logiques, adaptateurs
4. Additionneur
5. Unité arithmétique et logique (ALU)
6. Mémoire
7. Programme
8. Architecture
9. Détail d'un cycle d'horloge
10. Programmation

1. Bases

L'information est transportée par des faisceaux de planeurs de période temporelle 60 et de période spatiale 30. Cela facilite le croisement des faisceaux. Les 4 constituants de base de l'ordinateur sont :

- un canon à planeur de période 60
<http://conwaylife.appspot.com/pattern/twogun>
- un réflecteur de planeur à 90°
<http://conwaylife.appspot.com/pattern/buckaroo>
- un duplicateur de planeurs
<http://conwaylife.appspot.com/pattern/gliderduplicator1>
- un absorbeur de planeur
<http://conwaylife.appspot.com/pattern/eater1>


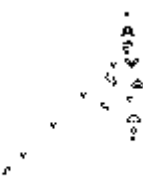





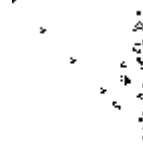


L'assemblage de ces composants a été réalisé avec Golly à la main et à l'aide de scripts Python pour les motifs trop grands pour être manipulés à la main ainsi que les motifs contenant des récurrences.

2. Portes logiques

Pour construire les portes logiques de base, on utilise le fait que deux faisceaux de planeurs orthogonaux peuvent s'annihiler ou former un absorbeur si ils se croisent avec le bon déphasage. Cela permet donc de contrôler un faisceau avec un autre faisceau. L'idée vient de cet article :

<http://www.rennard.org/alife/CollisionBasedRennard.pdf>

Table de vérité

a b	a OU b	a ET b	NON a
0 0			
0 1			
1 0			
1 1			

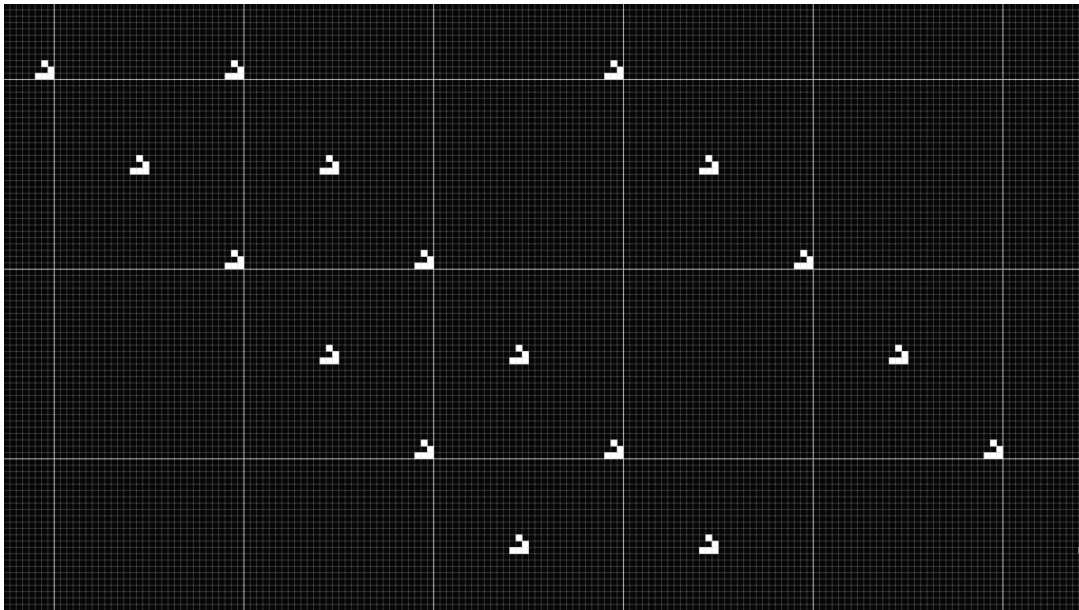
3. Assemblage des portes logiques, adaptateurs

L'assemblage de portes logiques entre elles n'est possible que si les faisceaux de planeurs sont bien en phase et que l'écartement entre les faisceaux est le bon.

Pendant la fabrication de chaque composant contenant plusieurs portes logiques j'ai donc pris les conventions suivantes :

- utilisation d'un quadrillage 30*30,
- chaque faisceau de planeur doit passer par les coins du quadrillage et les planeurs doivent épouser les coins
- les faisceaux de planeurs principaux proviennent du coin haut gauche

Ainsi, les faisceaux suivant sont bien en phase :



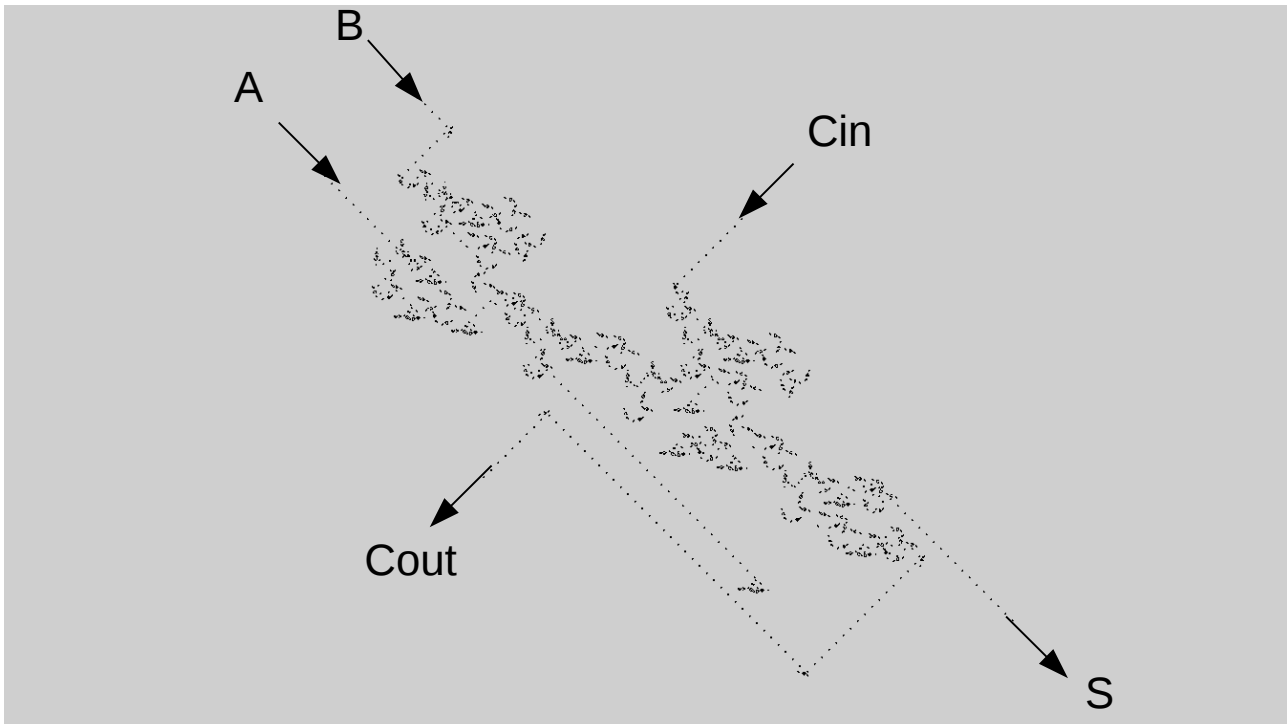
Pour changer la phase ou la position d'un faisceau, on utilise une série de 3 réflecteurs qu'on appellera adaptateur. Le dossier tools contient deux script python permettant la fabrication de tels adaptateurs. Le script decamaker.py permet de translater horizontalement d'un multiple de 15 pixels un faisceau déjà bien en phase. Le script delaymaker.py fabrique un adaptateur quelconque à partir de 3 longueurs d1,d2,d3 :



Des adaptateurs sont déjà inclus à la sortie et aux entrées des portes logiques du dossier « logic gates », ainsi toutes leurs entrées et sorties sont bien en phase et il suffit d'utiliser le script decamaker.py pour les assembler. J'ai construit les composants principaux de l'ordinateur à partir de telles portes logiques.

4. Additionneur

Une fois les portes logiques implémentées, on peut construire n'importe quel circuit logique asynchrone. Ici, un additionneur complet :



A,B : entrées

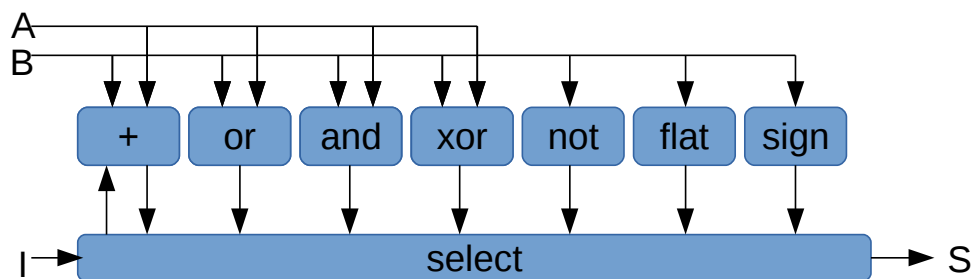
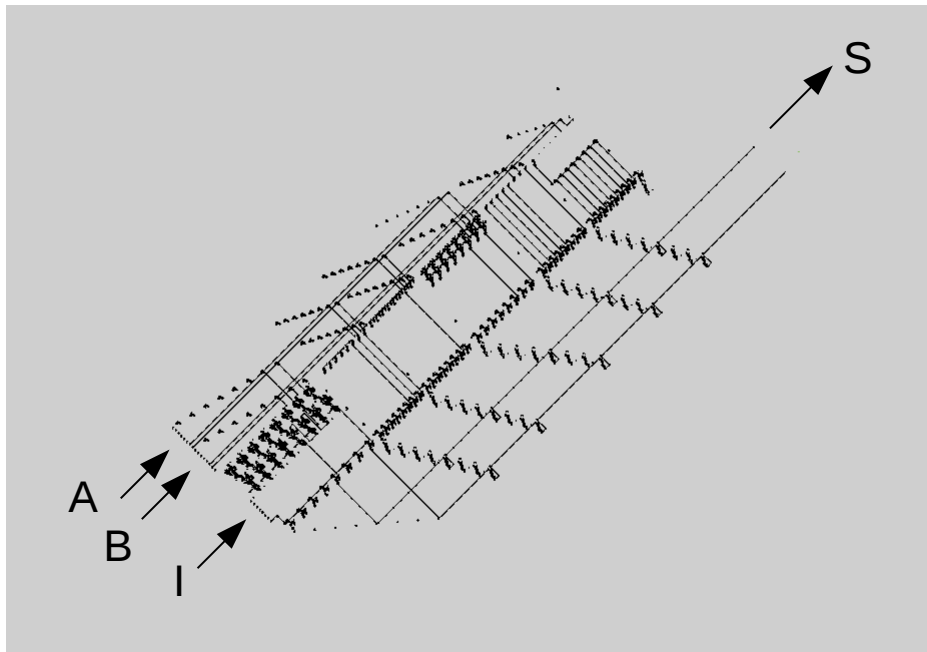
Cin : entrée de la retenue

Cout : sortie de la retenue

S : sortie

Il suffit ensuite d'en juxtaposer n pour obtenir un additionneur n bit.

5. Unité arithmétique et logique (ALU)



A, B : entrées

I : instruction

S : sortie

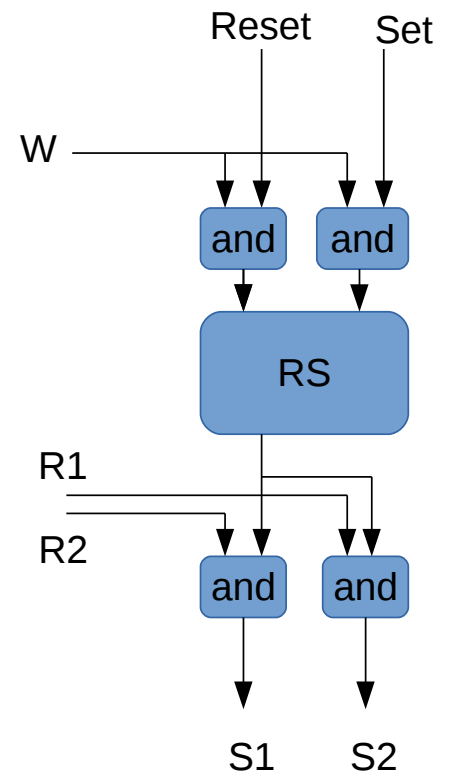
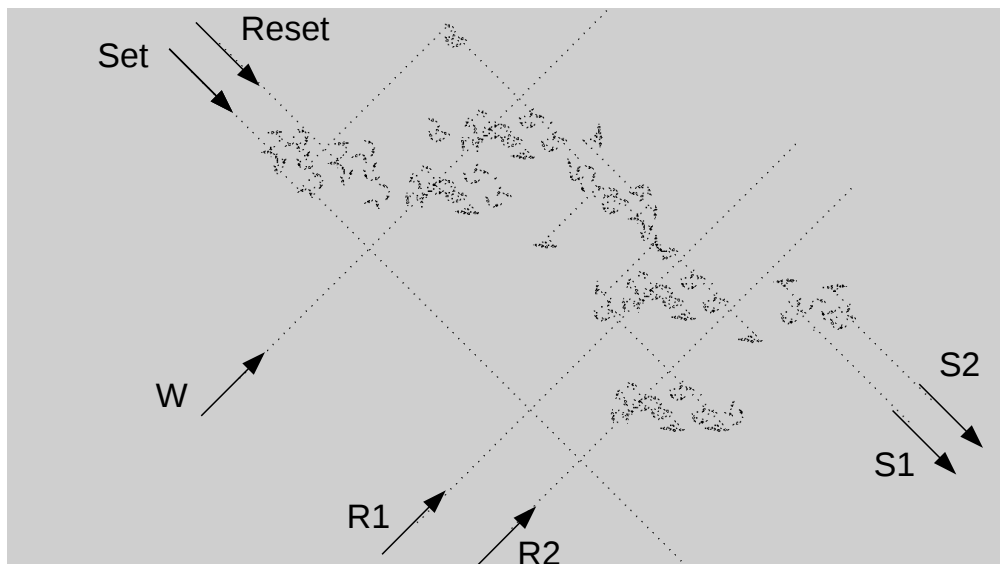
L'ALU est constituée d'un assemblage de fonctions logiques et d'une unité de sélection du résultat constituée de portes ET. Les entrées sont sur 8 bit chacune, l'instruction est constituée de 8 bits indépendants. L'opération flat consiste à renvoyer 1111111 si $B \neq 00000000$, l'opération sign sélectionne le bit de poids le plus fort de B. Le poids des bits en entrée et en sortie croît vers le haut gauche. La flèche partant du module select vers l'additionneur représente l'instruction increment.

Codes des instructions :

+	00000001
or	00000010
and	00000100
xor	00001000
not	00010000
flat	00100000
sign	01000000
increment	10000000

6. Mémoire

Module mémoire :



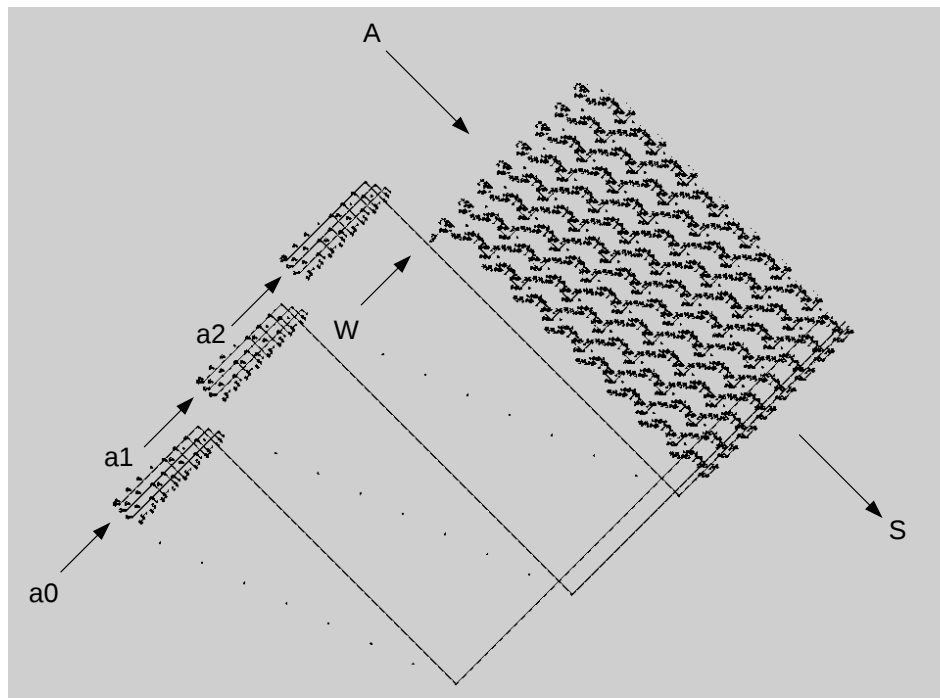
W : écrire 1 si set=1 0 si reset=1

R1 : lire l'état de la mémoire sur la sortie S1

R1 : lire l'état de la mémoire sur la sortie S2

Un module mémoire est formé d'un verrou logique RS associé à des portes ET qui gèrent l'écriture et la lecture. La mémoire est un assemblage de 64 modules de ce type en 8 lignes et 8 colonnes pour former une mémoire de 8x8 octets.

Mémoire :



A : données à écrire, 1 octet
a0 : adresse pour l'écriture, 3 bits
a1 : adresse pour la lecture, 3 bits
a2 : adresse pour la lecture, 3 bits
S : sortie, 2 octets
W : écrit A à l'adresse a0 si W=1

Les 3 modules sur la droite de la mémoire sont des décodeurs 3 bit vers 8 qui permettent de décoder les adresses. Cette mémoire permet de lire les données de deux adresses a1 et a2 en même temps. La sortie comporte donc 16 bits alternés entre S1 et S2.

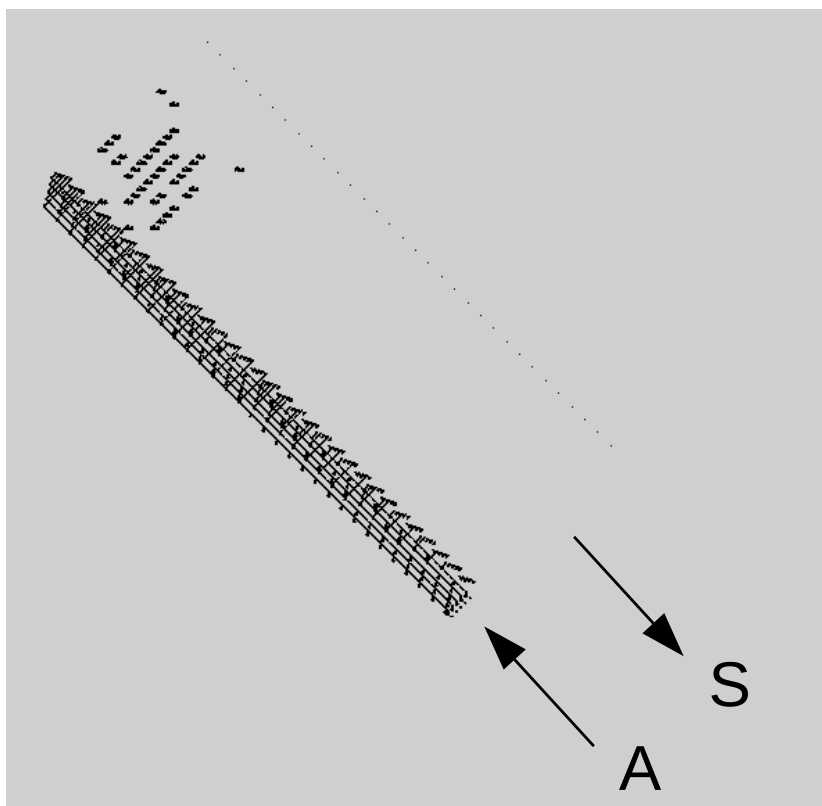
7. Programme

Le programme est constitué d'une mémoire statique. Le grand module rectangulaire est un décodeur 5 vers 32 bits qui permet de sélectionner une ligne du programme en fonction de l'adresse entrée.

Chaque ligne est composée de 21 bits répartis comme suit en allant vers le coin haut droit :

- 4 bits d'instruction
- 3 bits pour une adresse d'écriture
- 3 bits pour une adresse de lecture
- 3 bits pour une adresse de lecture
- 8 bits de données

Le script `assembly.py` contient plus d'information sur les différentes instructions et la programmation de l'ordinateur.



A : adresse de la ligne à lire, 5 bits

S : sortie : 21 bits

8. Architecture

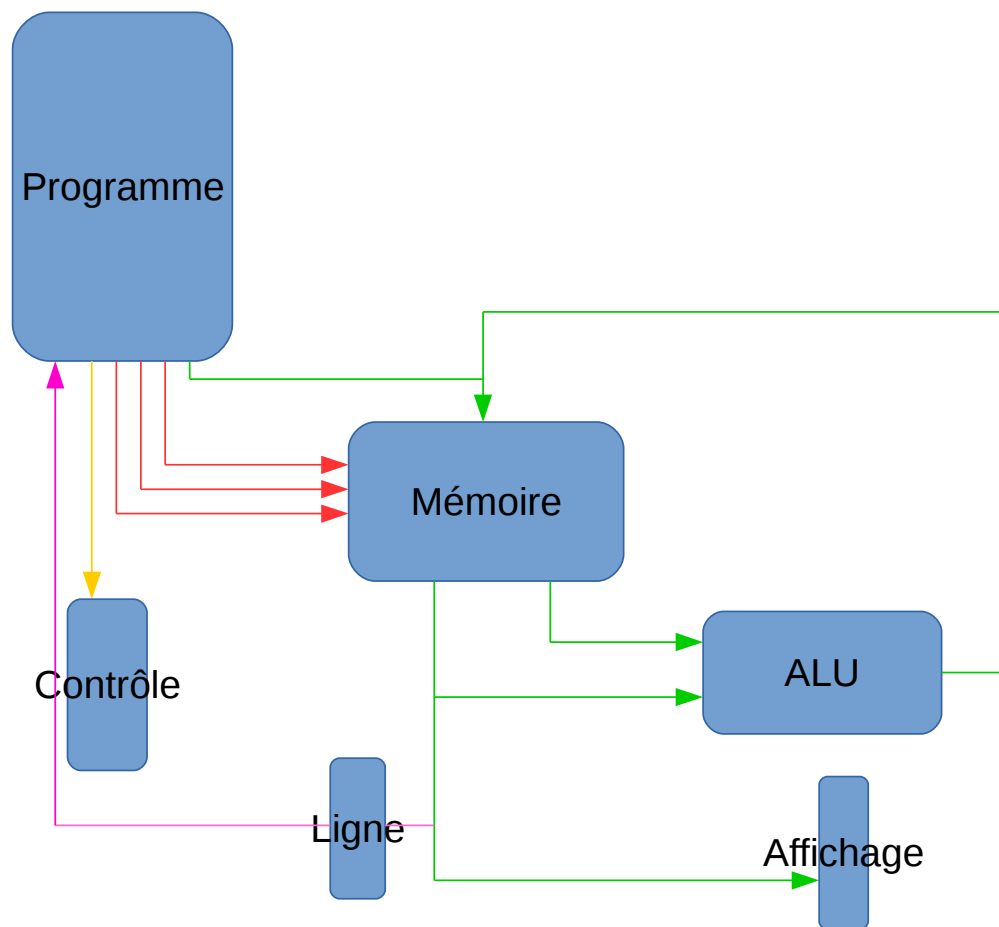


Schéma de l'architecture de l'ordinateur. Seul les principaux canaux d'information sont représentés. Le module «Ligne» est une mémoire de 5 bits qui permet de stocker l'adresse de la ligne du programme en cours d'exécution (PC). Le module « contrôle » est essentiellement un décodeur qui prend en entrée le numéro d'une instruction sur 4 bits, la décode sur 16 bits indépendants et qui communique avec tout les autres constituants de l'ordinateur pour faire exécuter l'instruction.

Ne sont pas représentés sur ce schéma :

- les canaux de contrôle reliant le module de contrôle aux autres modules
- l'horloge
- les canaux de contrôle reliant l'horloge aux autres modules

Code couleur :

Vert : bus de données 8bit

Rouge : adresse 3bits

Jaune : instruction 4bits

Rose : adresse 5bits

9. Détail d'un cycle d'horloge

L'horloge est constituée par 4 boucles formées par des réflecteurs de planeurs dans lesquelles tournent des faisceaux de planeurs. Sur chaque boucle, un duplicateur de planeur est placé sur le passage du faisceau de planeur. Cela permet d'extraire le signal de l'horloge. Les 4 boucles permettent chacune de rythmer: l'exécution d'une instruction, l'écriture sur la mémoire, l'incréméntation de PC, l'écriture en mémoire de PC.

PC est stocké dans la mémoire à l'adresse 000, cependant, on a besoin de le lire tout le long de l'exécution d'une instruction. Il est donc copié dans le module Ligne à chaque cycle. Le tableau suivant résume un cycle d'horloge.

Génération	Boucle d'horloge	Front du signal	Action
0	0	Montant	Lecture d'une instruction, exécution de celle ci par l'ALU sans écrire sur la mémoire
590 000	1	Montant	Écriture sur la mémoire de la sortie de l'ALU
650 000	1	Descendant	Fin d'écriture sur la mémoire
860 000	0	Descendant	Fin de lecture de l'instruction
1 100 000	2	Montant	Lecture de PC de l'adresse 000 vers l'ALU pour incréméntation Copie de PC de 000 vers le module Ligne
1 440 000	3	Montant	Écriture de la sortie de l'ALU (PC+1) à l'adresse 000
1 480 000	3	Descendant	Fin de l'écriture de PC+1 à l'adresse 000
1 600 000	2	Descendant	Fin de la lecture de l'adresse 000 et fin de la copie de 000 vers le module ligne. A cet instant, PC+1 a été écrit en 000 et sur le module Ligne

La colonne « boucle d'horloge » correspond à la position de la boucle qui génère le signal. L'indice croît vers le haut droit. Le numéro de la génération correspond à l'instant où le signal sort de la boucle d'horloge et non à l'instant où il atteint le module cible (arrondi à 10 000 près). Chaque cycle se décompose en 2 parties : exécution de l'instruction (boucles 0 et 1) et incréméntation de PC (boucles 2 et 3). Chaque cycle a une période d'exactement 2 003 880 générations.

10. Programmation

La programmation de l'ordinateur se fait à l'aide du script Golly assembly.py. Les 8 variables disponibles sont nommées a,b,c,d,e,f,g,h. h correspond à l'adresse 000 et est donc utilisée pour stocker PC.

Instruction	Effet
write a n	Écrit le nombre n dans la variable a
goto n	Va à la ligne n du programme
move a b	b=a
jumpif a	Saute la ligne suivante si a!=0
print a	Affiche a
add a b c	c=a+b
or a b c	c=(a ou b)
and a b c	c=(a et b)
xor a b c	c=(a ou exclusif b)
not a b	b=non(a)
flat a b	B=0 si a=0 ; b=11111111 sinon
sign a b	Écrit le bit le plus fort de a dans b (signe de a si a est écrit en complément à 2)
increment a	a=a+1

Exemple : algorithme d'Euclide

Affiche a modulo b		
ligne	instruction	pseudo-code
0	write a 8	a = 8
1	write b 3	b = 3
2	write e 1	d = -b
3	not b d	
4	add d e d	
5	add a d a	
6	sign a c	si a>=0
7	jumpif c	
8	goto 5	aller a la ligne 5
9	add a b a	a = a+b
10	print a	afficher a
11	goto 11	fin

Affiche PGCD(a,b)		
ligne	instruction	pseudo-code
0	write a 8	a = 8
1	write b 6	b = 6
2	write e 1	c = a modulo b
3	not b d	
4	add d e d	
5	add a d a	
6	sign a f	
7	jumpif f	
8	goto 5	
9	add a b c	
10	jumpif c	si c = 0
11	goto 15	aller a la ligne 15
12	move b a	a = b
13	move c b	b = c
14	goto 3	aller a la ligne 3
15	print b	afficher b
16	goto 16	fin