GYMNASE DE BEAULIEU
TRAVAIL DE MATURITÉ 2023

CRÉATION D'UN ORDINATEUR 8-BIT

DE LA CONCEPTION À L'ASSEMBLAGE PHYSIQUE

Répondant : Nicholas Wolff

Manaf Mhamdi Alaoui

Lausanne, le vendredi, 10 novembre 2023

Remerciements

Je tiens à adresser mes remerciements au répondant de ce travail de maturité, Nicholas Wolff, qui m'a accordé de son temps, suivi, conseillé et aiguillé durant toute cette année.

Je remercie également Elias Mhamdi Alaoui, mon frère, pour m'avoir conseillé et m'avoir appris à utiliser KiCad et à chercher les composants.

Enfin, j'aimerais remercier tout autre personne ayant contribué de loin ou de près à ce travail de maturité, que ce soit pour des conseils, de la relecture, ou simplement pour de la motivation.

Résumé

Auteur

MHAMDI ALAOUI Manaf 3M06

Répondant

Nicholas Wolff

Titre

Création d'un ordinateur 8-bit, de la conception à l'assemblage physique

Un ordinateur 8-bit est un ordinateur dont la mémoire, les nombres et l'architecture ont 8-bit de large. En informatique, un bit représente un 0 ou un 1 de mémoire.

Ce travail de maturité vise à concevoir schématiquement un ordinateur 8-bit sur un logiciel de simulation de logique appelé Logisim, puis de convertir le schéma à un logiciel de schématisation électrique appelé KiCad. Des outils de développement seront créés afin de programmer l'ordinateur. Un montage physique de l'ordinateur a été envisagé mais malheureusement, par manque d'avancement, a été écarté.

L'objectif global de ce TM est de visualiser plus en détail comment un ordinateur fonctionne. En effet, de nos jours, les ordinateurs sont comme une boîte noire. Nous avons du mal à nous visualiser tout ce qu'il se passe derrière l'écran.

Table des matières

Introduction

L'ordinateur devient de plus en plus commun mais de plus en plus dissimulé dans notre vie quotidienne. Il se cache, tels que dans les feux de signalisation, les puces de cartes de crédit, les cartes SIM, les tests de grossesse, etc. Ce que nous définissons d'ordinateur est communément l'ordinateur de bureau. Mais la définition est plus vague. Elle design toute conception capable de traiter de l'information, selon un algorithme. Il n'est même pas nécessaire d'utiliser de l'électricité. C'est en effet possible de concevoir un ordinateur fonctionnant à l'eau, bien que l'électricité reste le moyen le plus rapide, le plus stable et le plus efficace.

Dans ce travail de maturité, j'ai souhaité repartir de zéro pour comprendre ce qu'est un ordinateur et ce qui le compose. Je n'allais pas construire un ordinateur en utilisant des pièces détachées disponibles sur le marché mais schématiser une carte contenant toute la logique nécessaire. Compte tenu de la complexité d'un ordinateur moderne, je me suis limité à la vitesse et la puissance des ordinateurs d'environ 1970.

J'ai utilisé GitHub pour ce travail de maturité, tous les schémas et programmes se trouvent dans le dépôt. Voici le lien : https://github.com/manaf941/ordinateur-8-bit-tm

Motivations

J'ai toujours été intéressé par les ordinateurs dans le sens large. Depuis mes huit ans, j'ai toujours eu un intérêt particulier pour la création de jeu vidéo, site web, et le fonctionnement interne d'un ordinateur. J'ai profité des anciens ordinateurs non fonctionnels de ma famille pour en voir l'intérieur aux alentours de mes neuf ans. À mes onze ans, j'essayais pour la première fois les Arduino et les Raspberry Pi. Ce n'étaient pas de vrais projets mais j'en apprenais beaucoup. Ce n'est qu'à partir de mes douze ans que j'ai vraiment commencé à coder (en JavaScript) en autodidacte.

Cependant, le code en JavaScript est de très haut niveau. En effet, tous les principes complexes de l'ordinateur (la gestion de la mémoire, dépendances, communication réseau, etc.) sont abstraits par le langage. C'est aussi un langage compilé juste à temps (JIT), ce qui veut dire que la conversion entre le langage lisible par les humains et le langage lisible par la machine se fait juste avant l'exécution du programme.

Tous ces éléments font que le JavaScript est lent, comparé à d'autres langages de programmation. J'ai donc toujours voulu coder de plus en plus bas niveau. J'ai essayé le Go, le Rust, très rapidement le C++, mais le travail de maturité était une occasion de faire un projet matériel et logiciel de très bas niveau.

Planification

Dans un ordinateur numérique, l'information est encodée avec des bits. Un bit est une unité d'information, représentant soit un 0, soit un 1. Mon but était de concevoir un ordinateur 8-bit. C'est-à-dire que la mémoire, les nombres et l'architecture seraient composés de 8 bits de large. En pratique, cela voudrait dire que les nombres peuvent varier de 0 à 255 ($2^8 - 1$), et que tout serait structuré autour de cette taille. Par comparaison, les ordinateurs modernes utilisent une architecture 64 bits. Le nombre de bits n'influe pas directement la vitesse de calcul mais il facilite le traitement d'information.

Cependant, l'adressage, de la RAM et de la ROM n'utiliseraient pas 8-bit dans l'adressage. La RAM et la ROM sont des listes de données, accessible avec une adresse, c'est-à-dire une position des données. Utiliser une adresse en 8-bit limite le nombre de donnée à 256 octets (2^8). À l'inverse, utiliser une adresse 16 bits limite le nombre de donnée à 65536 octets (2^{16}), bien plus que nécessaire. Utiliser du 8-bit pour l'adressage du programme reviendrait à un programme de 256 instructions maximums. Pareil pour la mémoire RAM, 256 octets de mémoire pourrait potentiellement poser un problème sur les gros programmes. J'ai donc choisi d'utiliser 16 bits pour l'adressage de ma RAM et de ma ROM. La taille est donc de 65536 octets, soit 64 KiB.

Pour ce projet, je comptais réaliser un schéma capable d'être simulé sur Logisim, puis un schéma sur KiCad afin de commander un circuit imprimé de l'ordinateur. Je ne peux pas utiliser le schéma de Logisim pour réaliser le circuit imprimé, car les composants sur Logisim sont très théoriques et abstraits, là où KiCad utilise des vrais composants physiques disponibles sur le marché. Il faut donc refaire le schéma de Logisim, sur KiCad, manuellement.

La dernière partie de ce projet était la création des outils de développement nécessaires à la programmation de cet ordinateur. Il me fallait un langage de programmation personnalisé conforme à ma table d'instruction ainsi qu'un émulateur tournant sur un ordinateur moderne pour accélérer le développement.

Simulation Logisim

Prise en main

Logisim est un logiciel de simulation, utilisé pour concevoir des circuits électroniques logiques. J'ai choisi Logisim car il n'y a pas beaucoup d'alternative pour macOS, et c'était le meilleur parmi ces dernières. Je n'avais jamais utilisé Logisim auparavant, j'ai dû d'abord m'habituer aux différentes fonctionnalités. J'ai rapidement créé quelques schémas d'essai et je me suis lancé sur mon travail de maturité.

Conception

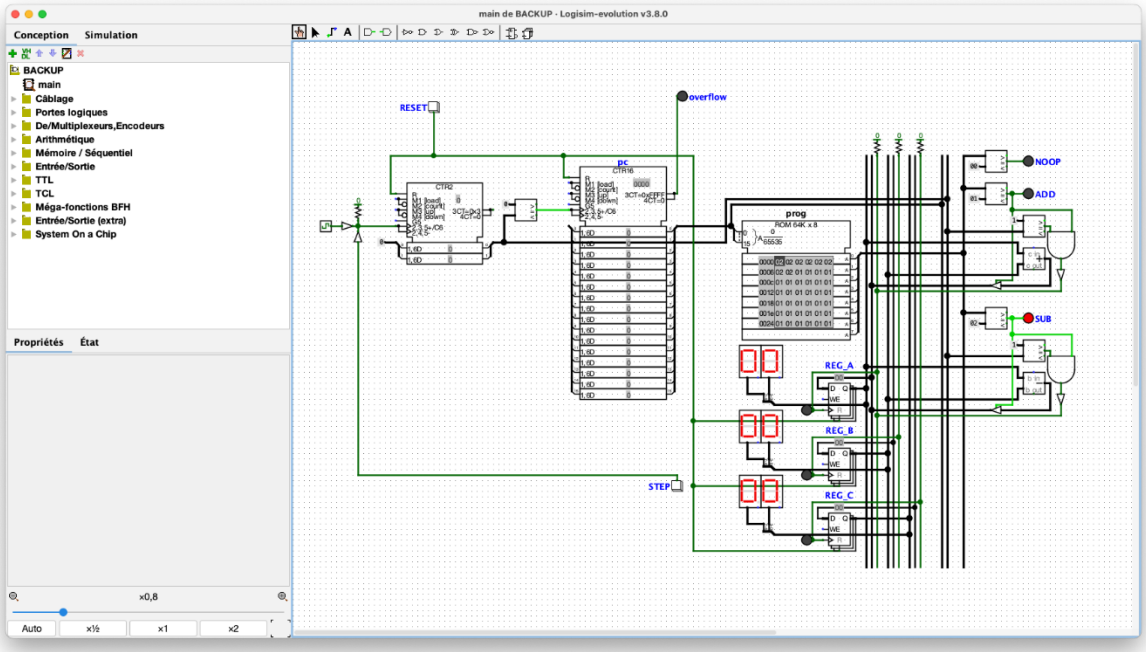


Figure 1 Schéma sur Logisim. Le schéma contient une horloge, un compteur d'étape, un compteur d'instruction, une ROM, trois registres, un BUS, et trois instructions (NO-OP, ADD, SUB)

J'ai commencé mon schéma en ajoutant une horloge, un composant qui pulse à une certaine fréquence afin de donner un rythme à l'ordinateur. Un compteur d'étape compte jusqu'à 4. Il permet de séparer les instructions en différentes étapes. Le compteur d'instruction situe l'ordinateur dans le code. Il s'incrémente à chaque fois que l'instruction précédente finit. La ROM est le composant contenant le programme qui s'exécute sur l'ordinateur. Elle a été programmée au préalable par l'assembleur. La ROM ne peut pas être modifiée pendant l'exécution du code.

J'ai choisi trois registres à savoir registre A, registre B et registre C. J'ai choisi d'utiliser les registres A et B pour les opérations mathématiques. Puisque l'ordinateur utilise 16 bits pour l'adressage, comme détaillé dans la section Planification, les registres B et C sont utilisés ensemble pour représenter des adresses 16 bits. En effet, l'utilisation de deux registres 8 bits est égale à 16 bits.

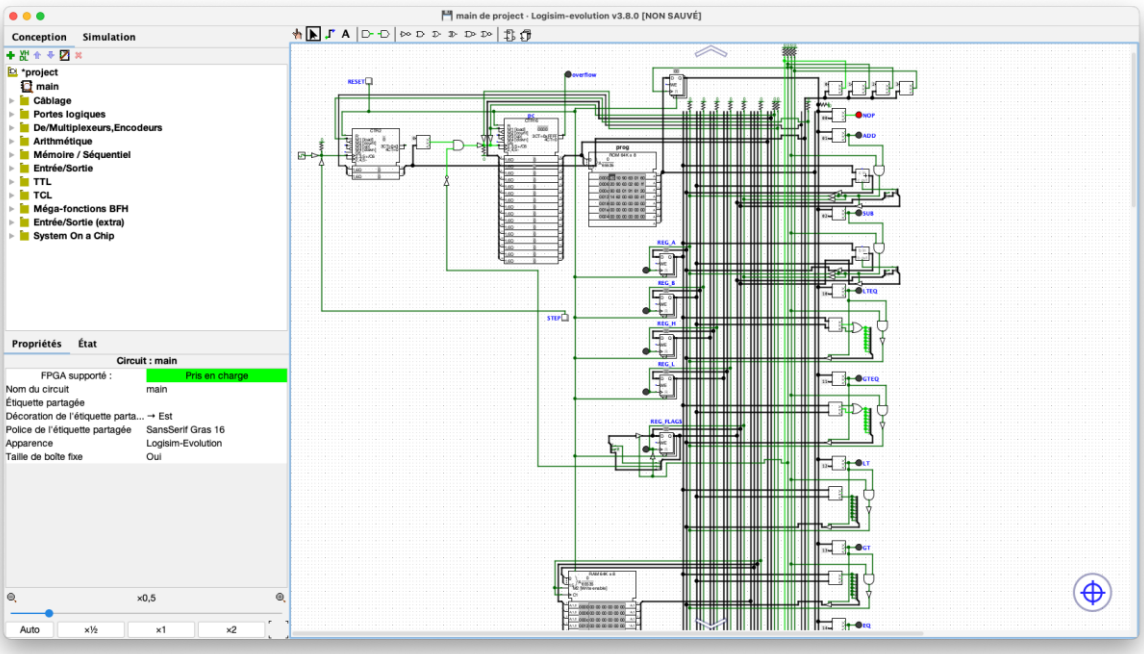
Première Table d'instruction

Une table d'instruction est un tableau détaillant les instructions supportées par l'ordinateur, et comment elles devraient être implémentés. En rédiger une était nécessaire pour garder une qualité constante à travers les différentes implémentations (Logisim, KiCad, émulateur) de l'ordinateur. Ce tableau me servait de documentation interne pour implémenter les instructions. Il me servait aussi de liste de tâche, pour garder facilement en mémoire ce qu'il me restait à faire.

IMPLEMENTATION	CODE	OP	Inputs (Registers)	Outputs (Registers)	Description
✅	`0x00`	NOP	`[]`	`[]`	No Operation
✅	`0x01`	ADD	`[A, B]`	`[A:((A + B) % 256)]`	Performs an addition on `A` and `B`, returns `A + B`
✅	`0x02`	SUB	`[A, B]`	`[A:((A - B) % 256)]`	Performs a substraction of `A` by `B`, returns `A - B`
❌	`0x03`	MUL	`[A, B]`	`[A:((A * B) % 256)]`	Performs a multiplication on `A` and `B`, returns `A * B`
❌	`0x04`	DIV	`[A, B]`	`[A:(A / B)]`	Performs a division on `A` by `B`, returns `A / B`
✅	`0x10`	LTEQ	`[A, B]`	`[A:(A <= B)]`	Returns `0xFF` if `A` is less than or equal to `B`, `0` otherwise `0x00`
✅	`0x11`	GTEQ	`[A, B]`	`[A:(A >= B)]`	Returns `0xFF` if `A` is greater than or equal to `B`, `0` otherwise `0x00`
✅	`0x12`	LT	`[A, B]`	`[A:(A < B)]`	Returns `0xFF` if `A` is greater than `B`, `0` otherwise `0x00`
✅	`0x13`	GT	`[A, B]`	`[A:(A > B)]`	Returns `0xFF` if `A` is less than `B`, `0` otherwise `0x00`
✅	`0x14`	EQ	`[A, B]`	`[A:(A == B)]`	Returns `0xFF` if `A` is equal to `B`, `0` otherwise `0x00`
✅	`0x20`	AND	`[A, B]`	`[A:(A & B)]`	bitwise AND operation
✅	`0x21`	OR	`[A, B]`	`[A:(A | B)]`	bitwise OR operation
✅	`0x22`	XOR	`[A, B]`	`[A:(A ^ B)]`	bitwise XOR operation,
✅	`0x23`	NOT	`[A]`	`[A:(~A)]`	bitwise NOT operation
✅	`0x30`	PC	`[]`	`[C:pc[0:8], D:pc[8:16]]`	Returns the positional counter
✅	`0x31`	CARRY	`[]`	`[A:carry]`	Returns `0xFF` if the last arithmetic operation had a carry, otherwise `0x00`
✅	`0x40`	JUMP	`[C:dest[0:8], D:dest[8:16]]`	`[]`	Jumps to `dest`
✅	`0x41`	JUMPI	`[A:condition, C:dest[0:8], D:dest[8:16]]`	`[]`	Jumps to `dest` if `condition` is equal to `0xFF`
✅	`0x42`	JUMPZ	`[A:condition, C:dest[0:8], D:dest[8:16]]`	`[]`	Jumps to `dest` if `condition` is equal to `0x00`
❌	`0x50`	RI	`[B:port]`	`[A:data]`	Read from the I/O port `port`
❌	`0x51`	WI	`[A:data, B:port]`	`[]`	Write byte `data` to the I/O port `port`
✅	`0x60`	LDA	`[]`	`[A:data]`	Set register `A` to the next byte of the rom;
✅	`0x61`	LDB	`[]`	`[B:data]`	Set register `B` to the next byte of the rom;
✅	`0x62`	LDC	`[]`	`[C:data]`	Set register `C` to the next byte of the rom;
✅	`0x63`	LDD	`[]`	`[C:data]`	Set register `D` to the next byte of the rom;
❌	`0x70`	SWAPAB	`[A:data1, B:data2]`	`[A:data2, B:data1]`	Swap registers `A` and `B`
❌	`0x71`	SWAPAC	`[A:data1, C:data2]`	`[A:data2, C:data1]`	Swap registers `A` and `C`
❌	`0x72`	SWAPBC	`[B:data1, C:data2]`	`[B:data2, C:data1]`	Swap registers `B` and `C`
❌	`0x80`	COPYAB	`[A:data]`	`[A:data, B:data]`	Copy register `A` to register `B`
❌	`0x81`	COPYAC	`[A:data]`	`[A:data, C:data]`	Copy register `A` to register `C`
❌	`0x82`	COPYBC	`[B:data]`	`[B:data, C:data]`	Copy register `B` to register `C`
✅	`0x90`	MSET	`[A:data, C:dest[0:8], D:dest[8:16]]`	`[]`	Write to the memory `data` at position `dest`.
✅	`0x91`	MLOAD	`[C:dest[0:8], D:dest[8:16]]`	`[A:data]`	Read the memory at position `dest`

Figure 2 Table d'instruction au 03.05.23

D'après la table d'instruction à la figure 2, on remarque la présence de 23 instructions avec une coche verte, signifiant la finalisation de l'intégration de l'instruction au schéma Logisim, comme visible dans la figure 3. En parallèle, les instructions avec une croix rouge n'ont pas été finalisées, pour diverses raisons détaillés quelques paragraphes ci-dessous.



4

Figure 3 Schéma au 03.05.23, implémentation de 23 instructions. Le schéma ne rentre pas en entier dans la capture d'écran. (voir annexe 3)

J'ai aussi ajouté un registre pour les flags. Il est utilisé uniquement pour le flag CARRY, pour savoir si l'opération précédente a overflow (c'est-à-dire quand le résultat de l'opération dépasse la valeur limite maximale autorisée, ce qui le fait revenir à 0) ou underflow (c'est-à-dire quand le résultat de l'opération dépasse la valeur limite minimale autorisée, ce qui le fait revenir à 255). En effet, si le résultat d'une opération est négatif ou plus grand que 255, l'ordinateur ne peut pas représenter le nombre sur 8 bits. Au lieu de perdre l'information de la retenue lors d'une opération mathématique, l'ordinateur ajoute la retenue dans un registre, récupérable plus tard.

J'ai aussi ajouté un quatrième registre. Les registres se nomment maintenant A, B, H et L. Les registres H et L sont utilisés exclusivement pour l'adressage de la RAM et de la ROM (lors d'un Jump). Ainsi, il n'y a plus de confusion entre les opérations mathématiques et l'adressage. Les deux registres se nomment H et L, signifiant « High » et « Low », décrivant l'ordre de leurs bits dans l'adressage 16-bits.

Même si l'ordinateur a été amélioré par l'ajout de registres supplémentaires, je remarquais que quelques problèmes se posaient avec mon schéma et ma table d'instruction actuelle.

Problèmes rencontrés

Complexité du BUS

Un BUS est un réseau de connexions servant à transférer des données d'un point de l'ordinateur à un autre. Avec ces cinq registres, une RAM, une ROM, un compteur d'étape, un compteur d'instruction, l'instruction elle-même, le BUS du schéma devenait trop compliqué. Je n'aurais jamais pu l'implémenter dans KiCad.

Limitations des registres

Avec uniquement quatre registres utilisable, les possibilités sont limitées.

Répétition des composants

Les instructions ne sont pas assez optimisées car les composants se répètent. Par exemple, la gestion des registres dans les opérations mathématiques et de comparaison se répète partout. Puisque chaque

instruction est isolée, elle ne bénéficie pas de la réutilisation des composants des autres instructions suivant à peu près le même fonctionnement.

Taille du schéma

Par ce problème d'optimisation, le schéma prend énormément de place car chaque instruction a besoin d'un accès au BUS, qui est placé sur une ligne droite verticale.

Multiplication / Division

Initialement, je voulais intégrer les fonctions de multiplication et de division dans mon ordinateur, mais sur conseil de mon répondant, j'ai décidé de ne pas les implémenter par souci de complexité.

SWAP/COPY

Les instructions SWAP/COPY sont trop compliquées, car il faut en implémenter une par pair de registres, ce qui complexifie largement le schéma et la table d'instruction.

Deuxième table d'instruction

La réécriture de la table d'instruction prévoit une solution à chacun des problèmes énumérés au-dessus. Cette table d'instruction prévoit 8 registres, la ré usabilité des composants sur le schéma/PCB en classant les instructions similaires entre elles, un schéma plus petit (moins d'instructions inutiles). Cependant, La multiplication et la division ne sont pas supportées.

IMPLEMENTATION	N bytes	CODE	OP	Inputs (Registers)	Outputs (Registers)	Description
✅	1	0b0000_0000	NOP	[]	[]	ø
✅	2	0b1000_0xxx	ADD	[lsrc, rsrc]	[dst]	dst = lsrc + rsrc
✅	2	0b1000_1xxx	SUB	[lsrc, rsrc]	[dst]	dst = lsrc - rsrc
✅	2	0b1001_0xxx	LTEQ	[lsrc, rsrc]	[dst]	dst = lsrc <= rsrc
✅	2	0b1001_1xxx	GTEQ	[lsrc, rsrc]	[dst]	dst = lsrc >= rsrc
✅	2	0b1010_0xxx	LT	[lsrc, rsrc]	[dst]	dst = lsrc < rsrc
✅	2	0b1010_1xxx	GT	[lsrc, rsrc]	[dst]	dst = lsrc > rsrc
✅	2	0b1011_0xxx	EQ	[lsrc, rsrc]	[dst]	dst = lsrc == rsrc
✅	2	0b1011_1xxx	AND	[lsrc, rsrc]	[dst]	dst = lsrc & rsrc
✅	2	0b1100_0xxx	OR	[lsrc, rsrc]	[dst]	dst = lsrc | rsrc
✅	2	0b1100_1xxx	XOR	[lsrc, rsrc]	[dst]	dst = lsrc ^ rsrc
✅	2	0b1101_0xxx	NOT	[lsrc]	[dst]	dst = ~lsrc
✅	2	0b1101_1xxx	LDX	[]	[dst]	dst = next byte
❌	2	0b1110_0xxx	RI	[lsrc]	[dst]	dst = IO[lsrc]
✅	2	0b1110_1xxx	CARRY	[]	[dst]	dst = carry
❌	2	0b1111_0xxx	WI	[lsrc, rsrc]	[]	IO[lsrc] = rsrc
✅	2	0b1111_1xxx	COPY	[lsrc]	[dst]	dst = lsrc
✅	1	0b0010_0xxx	JUMP	[]	[]	pc = HL
✅	1	0b0010_1xxx	JUMPI	[dst]	[]	pc = HL if dst == 0xFF
✅	1	0b0011_0xxx	JUMPZ	[dst]	[]	pc = HL if dst == 0x00
✅	1	0b0011_1xxx	JUMPC	[]	[]	pc = HL if carry == 1
✅	1	0b0100_0xxx	MWRITE	[dst]	[]	ram[HL] = lsrc
✅	1	0b0100_1xxx	MREAD	[]	[dst]	dst = ram[HL]

Figure 4 Deuxième table d'instruction

Chaque code d'instruction est composé de 5 bits. Les trois derniers bits, si requis par l'instruction, sont l'identificateur du registre dans lequel le résultat devrait être écrit. Cette manière de faire permet d'utiliser toute l'information à notre disposition afin d'avoir un système de registre flexible. Les opérations ne sont plus limitées à A et B et les codes d'instructions sont compactes.

Le premier bit du code d'instruction est 1 sur les instructions qui partagent le circuit de gestion de registre. Ces instructions prennent aussi deux entrées au maximum, dont les identificateurs des registres sont encodés dans le prochain octet.

Par exemple, la soustraction du registre A par B, où le résultat est enregistré dans C s'encodera avec $10001\_010\ 00\_000\_001$, 10001 étant la soustraction, 010 étant le registre C, 000 étant le registre A et 001 étant le registre B. Cela correspond à $C = A - B$.

Implémentation de la table d'instruction sur Logisim

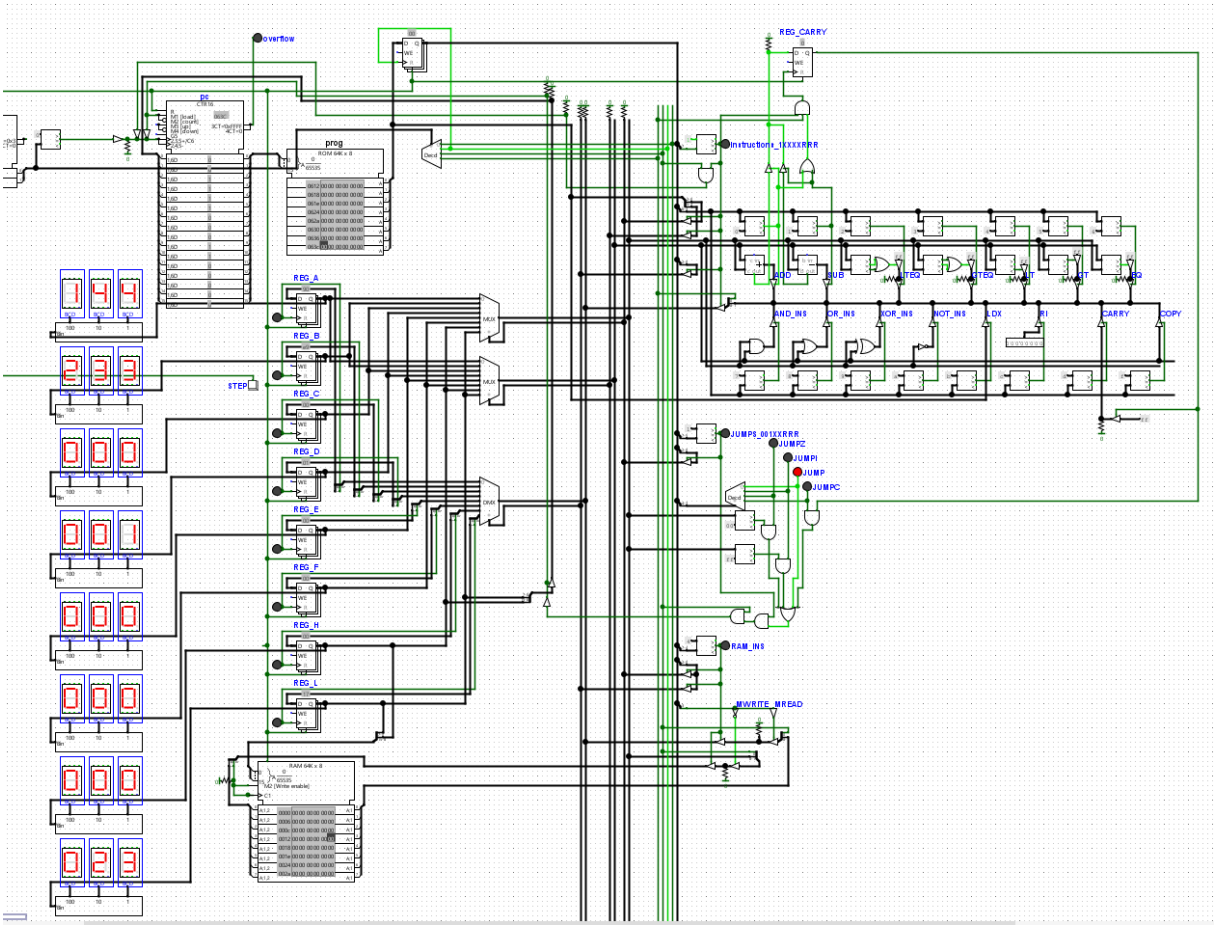


Figure 5 Implémentation sur Logisim de la deuxième table d'instruction.

Comme prévu par la table d'instruction, l'implémentation regroupe les instructions similaires dans cette zone condensée en haut à droite de l'image. C'est la partie ALU (Arithmetic Logic Unit) de l'ordinateur. Seulement les instructions concernant la RAM et les sauts dans le programme sont en dehors de cette zone. Dans la partie à gauche, les 8 registres sont présents et sont connectés avec des multiplexeurs au reste du schéma. Le BUS au milieu permet la communication entre les différents composants de l'ordinateur. La RAM se trouve sous les registres, et est directement reliée à ses instructions.

La première version de l'implémentation utilisait 8 étapes dans l'exécution des instructions, mais après un peu de réflexion, j'ai réussi à descendre le nombre d'étape à 4.

Interface Graphique

Un de mes objectifs était d'accomplir une interface graphique pour l'ordinateur. En d'autres termes, ça voudrait dire ajouter la possibilité de le brancher à un écran afin d'y afficher les résultats, et potentiellement des jeux, des graphiques, des tableaux, etc. Cependant, par manque de temps, je n'ai pas pu accomplir cette partie du projet. J'ai dû me limiter à des afficheurs 7 segments pour voir le contenu des registres.

Assembleur

Première table d'instruction

Écrire efficacement des programmes pour l'ordinateur demandait d'abord d'écrire un assembleur. En effet, c'est impossible sur le long terme, d'encoder soi-même les instructions et de les écrire dans un fichier, et c'est en particulier le cas avec la deuxième table d'instruction qui contient des données à l'intérieur du code d'instruction. L'utilisation d'un assembleur permet de mieux gérer les appels d'instructions et surtout d'alléger la partie manuelle de retranscription des codes d'instructions depuis la table.

J'ai commencé avec un simple script d'assemblage (avec la première table d'instruction) codé en JavaScript (interprété par Node.js) afin de tester plus rapidement les dernières fonctionnalités que j'ajoutais au schéma sur Logisim. Voici un exemple de programme :

```
label start
LDA 20
jump add

label add
LDB 01
ADD
JUMPI start
JUMP add
```

Figure 6 Exemple de programme fait pour l'ordinateur

Ce petit programme charge 32 (20 en hexadécimal) dans le registre A, et additionne 1 à A en boucle jusqu'à ce que A soit égal à 255 (FF en hexadécimal), ce à quoi il va recommencer depuis le début en boucle. Le programme n'est pas vraiment utile en soit, mais il a beaucoup servi à tester le comportement des jumps ainsi que celui de l'addition. Le code équivalent en python serait :

```python
while True:
    a = 0x20

    while True:
        b = 1
        a = a + b
        if a == 0xff:
            break
```

Figure 7 Équivalent du programme à la figure 6, en Python

La logique de cet assembleur se décomposait en plusieurs étapes :

1. Lecture du fichier contenant le programme
2. Lecture de chaque ligne individuelle et de l'instruction contenue dans cette ligne.

3. Liaison de toutes les instructions utilisées dans une seule liste, en laissant de la place pour les destinations des jumps.
4. Une fois que l'on connait la taille finale du programme, on remplace les destinations des jumps provisoires par leur vrai destination.
5. Écriture du résultat dans un fichier.

Cet assembleur était très basique mais était suffisant pour la première table d'instruction. Il n'était écrit qu'en un seul fichier et ne faisait que 240 lignes. Cependant, il a fallu tout réécrire pour la deuxième table d'instruction.

Deuxième table d'instruction

Puisque la table d'instruction ainsi que le langage d'assembleur que je souhaitais viser initialement étaient plus grand, j'ai décidé de soigner le code dans le but de le rendre le plus lisible possible. Pour le faire, j'ai créé un dossier dans le dépôt GitHub et j'ai configuré un projet TypeScript (variant du JavaScript mais typé). J'en ai profité pour y écrire un émulateur directement dans le même projet, afin de partager le plus de code possible (notamment la table d'instruction). Voici un exemple de programme :

```
a = 5
b = 3

# Copie des données originales pour ne pas modifier b
d = b

# while d != 0
tag loop
jumpz d end

# c += a
c = c + a

# d--
e = 1
d = d-e
jump loop

tag end
# l'algorithme effectue donc c = a + a + a
# c == 15
# Il faut noter que la complexité de cet algorithme est de O(n)
# et qu'un algorithme plus performant pourrait être utilisé à la place
```

Figure 8 Programme multipliant a et b entre eux

Ce programme d'exemple effectue une multiplication de a par b. Il additionne a, b fois, tel que :

$$a \cdot b = \sum^{b} a$$

La nouvelle table d'instruction autorise jusqu'à 8 registres, et le nouvel assembleur permet une écriture plus logique du code.

L'assembleur lit le fichier ligne par ligne, retire les commentaires, et lit les instructions lignes par lignes. Des expressions régulières sont utilisées pour identifier les différentes opérations de l'ALU (Arithmetic Logic Unit).

Émulateur

L'émulateur est un programme permettant d'exécuter du code destiné à un autre appareil sur un ordinateur normal. Les émulateurs sont utilisés la plupart du temps pour jouer à des jeux vidéo incompatible avec l'ordinateur, tel que les jeux fait pour console ou les vieux jeux rétro. C'est aussi utilisé dans le développement d'application mobile afin de tester les applications sur plusieurs appareils différent sans les posséder physiquement.

Dans mon cas, avec un émulateur, je peux exécuter, tester et débugger du code prévu pour l'ordinateur 8-bit que je produis, depuis mon ordinateur personnel. L'émulateur est pratique dans le cas de test unitaires, de débogage plus avancé (possibilité de mettre sur pause l'exécution du programme et d'inspecter plus en détail ce qu'il se passe) et simplement d'exécuter un programme lorsque l'on n'a pas l'ordinateur déjà construit.

Mon objectif pour l'émulateur était d'être précisément égal à la version sur Logisim ou physique, le but étant qu'aucune différence logique ne soit présente entre les différents environnements d'exécution.
Pour pouvoir interagir avec l'émulateur, une interface de programmation d'application (API) et une interface de ligne de commande (CLI) sont requises.

L'interface de programmation d'application est destinée aux programmes externes souhaitant lancer automatiquement du code, tel que dans les tests unitaires. L'interface de ligne de commande (cli) est prévue pour l'utilisation standard.

Le nouvel assembleur et l'émulateur sont plus complexes que le premier assembleur. Ils totalisent 34 fichiers et 1088 lignes de code. Le code est plus lisible, mieux segmenté et mieux structuré que mon premier script d'assembleur. Ils sont disponibles dans le dépôt GitHub sous le dossier « silver-lamp »

KiCad

La troisième et dernière partie de ce TM consiste à convertir le schéma Logisim sur KiCad. En effet, les schéma Logisim ne permet pas de créer de circuits imprimés. Convertir le schéma sur KiCad demande aussi la recherche de composants compatible chez des fournisseurs tels que Digikey et Distrelec. Le PCB final peut être commandé chez PCBWay.

L'élément le plus difficile de cette partie est de trouver les bons composants. Ils doivent correspondre à mes besoins, avoir la bonne tension, fréquence, température, résistance et capacité.

Ensuite, les composants sur Logisim sont très théoriques et peuvent dépasser ce qui est vraiment disponible sur les marchés. Par exemple, je n'ai pas pu trouver de multiplexeur, composant qui permet la sélection de donnée parmi plusieurs entrées, comportant 8-bit de données. Tous les multiplexeurs que j'ai pu voir n'avaient que 1 bit de donnée. Il faudrait que j'en dispose 8, chacun gérant 1 bit, afin d'obtenir ce dont j'ai besoin. Avec 3 sélecteurs de registres, j'arrive à 24 multiplexeurs, ce qui complexifie grandement le schéma.

L'horloge, composant qui alterne entre 0 et 1 à une fréquence donnée, est aussi très théorique sur Logisim.

Figure 9 Horloge sur Logisim

En revanche, le schéma d'une horloge devient beaucoup plus complexe sur KiCad :

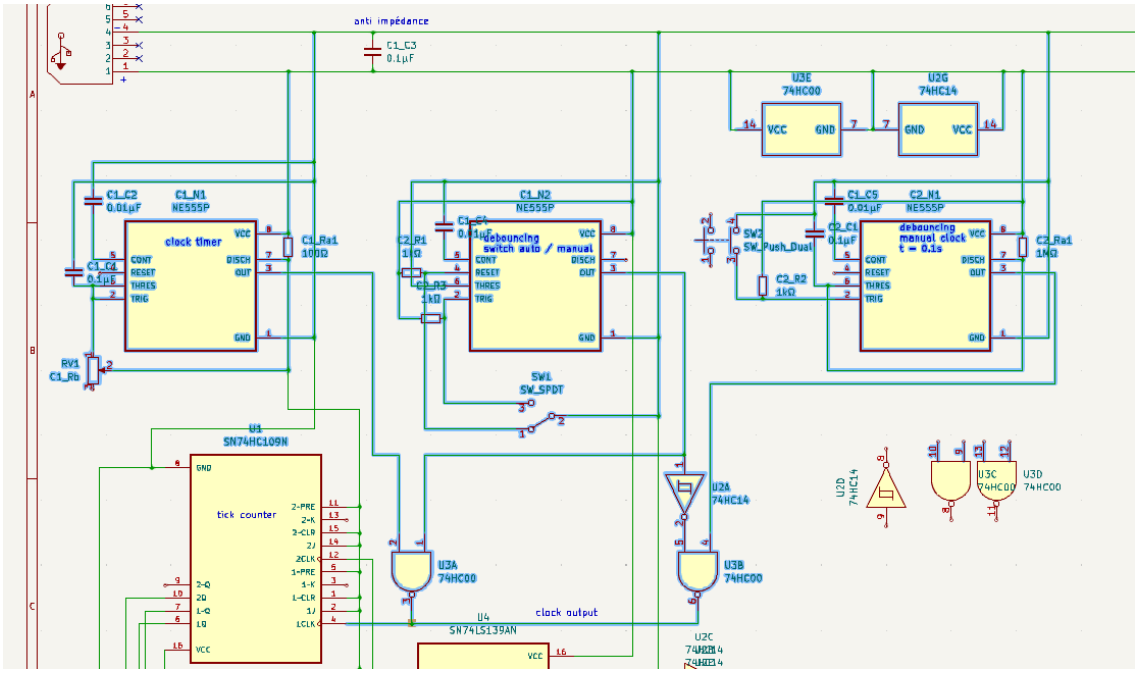


Figure 10 Horloge avec KiCad. Les composants utilisés pour l'horloge sont sélectionnés en bleu

L'horloge utilises trois temporisateurs NE555P. Il n'y a pas de composant déjà prêt pour les horloges qui permet autant que ce montage. Cette horloge permet l'oscillation constante avec une fréquence variable, c'est-à-dire que la vitesse de l'ordinateur peut être ajustée selon les besoins avec un potentiomètre (par exemple pour débugger un programme). L'horloge peut aussi être contrôlée manuellement et n'avancer que lorsqu'un bouton est appuyé.

N'ayant jamais fait de conception schématique auparavant, la conception de l'horloge m'était difficile. Je me suis documenté sur les composants utilisés lors de situations similaire. La chaine YouTube Ben Eater était une excellente ressource.

Le compteur d'étape et le compteur d'instruction sont implémentés avec des JK flip flop (voir annexe 1). Le compteur d'étape en utilise deux et le compteur d'instruction en utilise seize. Les JK flip flop sont des composants « divisant » le signal qui leur parvient. Les mettre en série crée un compteur de « n » bits.

Ensuite, Un décodeur prend le résultat du compteur d'étape et le sépare en quatre signaux distincts, comme sur Logisim.

Les données du compteur d'instruction sont envoyées dans l'adressage de la ROM afin de charger l'instruction. Le résultat est stocké temporairement dans un registre.

Les registres sont déjà sur le schéma mais je n'ai pas encore eu l'occasion d'y placer les multiplexeurs.

Conclusion

La première partie de conception sur Logisim s'est terminée sans trop de problème, mis à part l'abandon de l'interface graphique. J'ai su adapter le schéma et surmonter les difficultés du premier design. L'ordinateur est plus flexible que premièrement envisagé.

La programmation de l'assembleur et de l'émulateur a été effectué sans souci, je suis satisfait de cette partie. Le langage d'assembleur créé est lisible, et facile à utiliser. C'était la partie la plus simple car elle ne reposait que sur de la programmation en JavaScript/TypeScript, langages que je maîtrise déjà.

Je n'ai pas encore fini le schéma sur KiCad. Je vais utiliser le temps entre le rendu du dossier écrit et la défense des TM. Les problèmes de complexité entre Logisim et KiCad, ainsi que la recherche de composant m'ont fait aller plus lentement que prévu. Je ne pense pas qu'une réalisation physique de l'ordinateur soit possible à ce stade à cause du retard pris. Finir le schéma KiCad ne me permettrait sûrement pas d'effectuer une commande, et même si j'arrive, c'est tout ou rien. Si je me plante sur le schéma, je n'aurais pas le temps de faire une réitération. Dans tous les cas, la priorité actuelle est de finir le schéma KiCad.

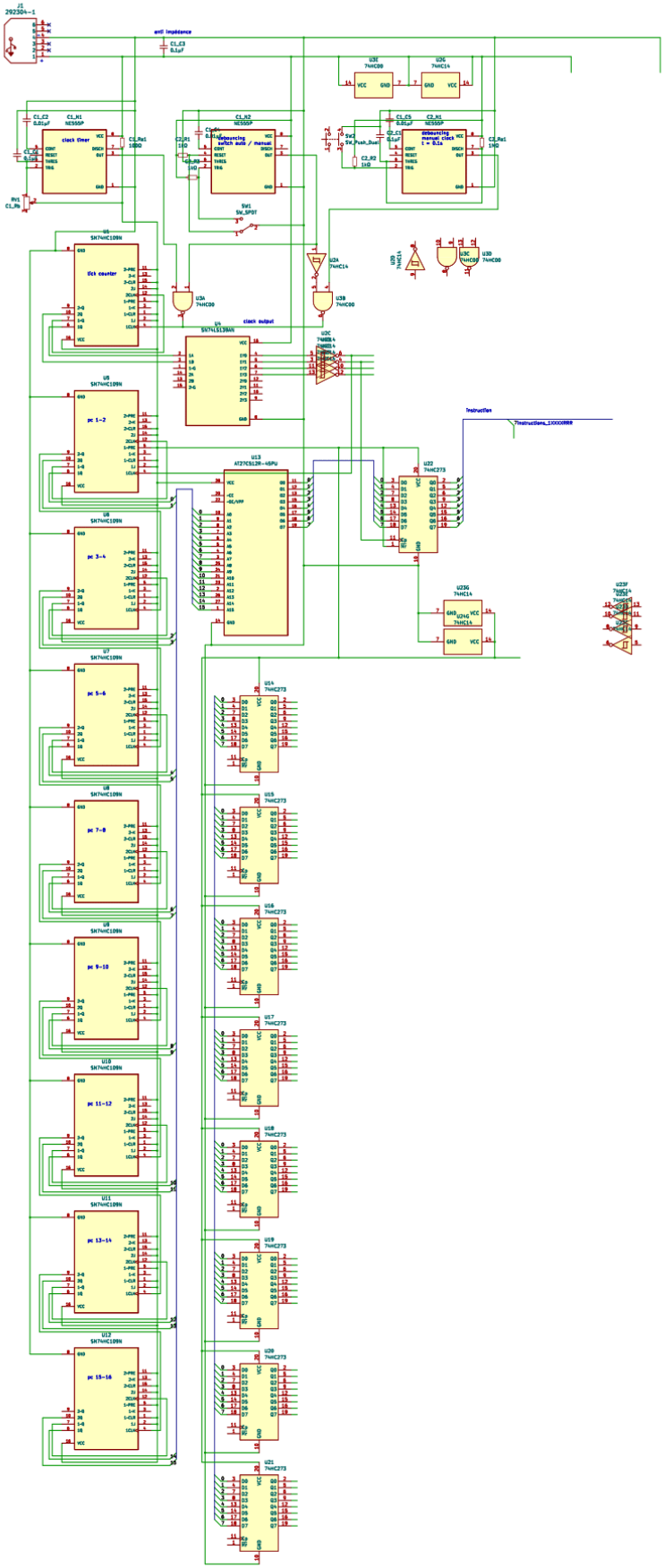
Durant ce travail de maturité, j'ai beaucoup appris sur le fonctionnement interne des ordinateurs, et je me suis beaucoup amusé. J'ai travaillé avec des logiciels que je n'avais jamais utilisés auparavant et avec des connaissances que je n'avais pas.

Bibliographie

Je n'ai pas utilisé d'image ou média externe ne venant pas de moi dans ce travail de maturité. La seule ressource utilisée dans ce travail est :

EATER Ben, Astable 555 timer - 8-bit computer clock - part 1, YouTube, 18 mars 2016, https://www.youtube.com/watch?v=kRlSFm519Bo, consulté le 5 octobre 2023.

Annexe 1 : État actuel du schéma KiCad



Cette annexe est disponible en haute résolution en PDF sur le dépôt GitHub dans « Dossier TM » ou en scannant ce code QR.

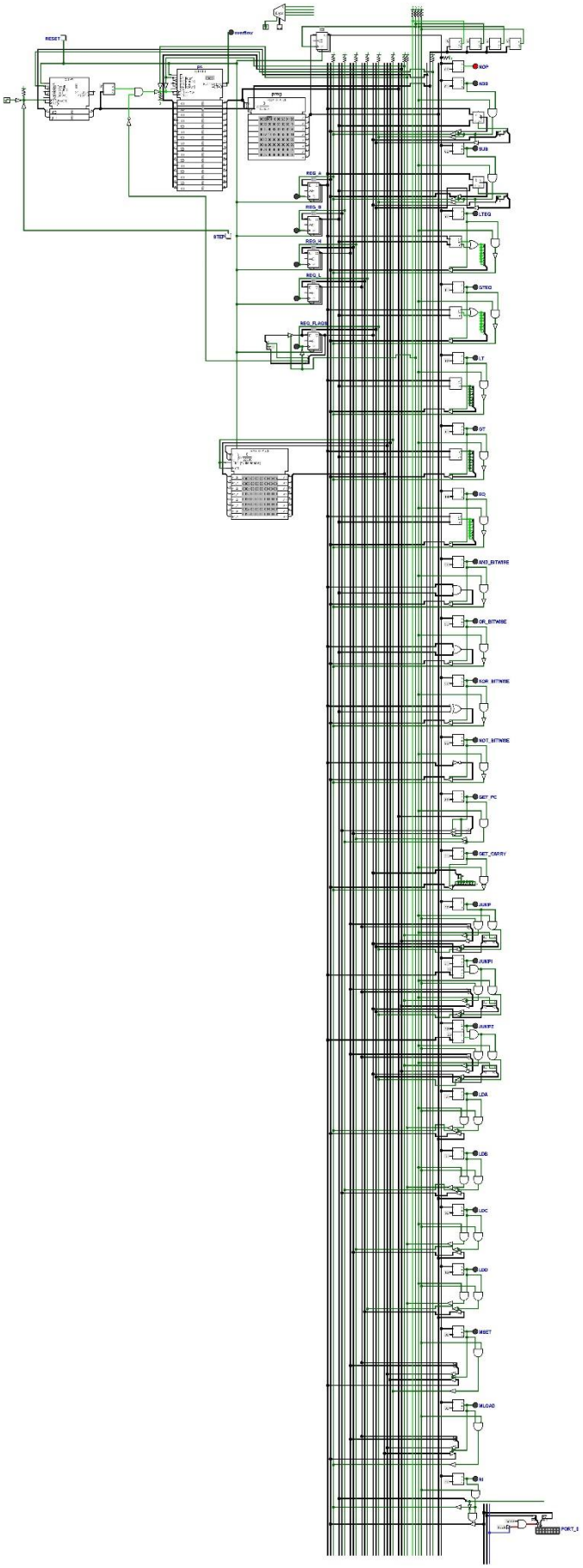
Annexe 2 : Suite de Fibonacci

```
# 12 is the max before overflow
f = 12 # terms

a = 0 #n1
b = 1 #n2

tag loop
# while(f > 0)
# while(f !== 0){
jumpz f end

# nth = n1+n2
d = a + b
# n1 = n2
a = b
# n2 = nth
b = d

# f--
d = 1
f = f-d
jump loop

tag end
```

Figure 11 La suite de Fibonacci, implémenté avec le langage d'assembleur. L'exemple est aussi disponible dans le dépôt GitHub, sous silver-lamp/examples/fibonacci.sl

Le résultat de ce programme devrait être $a = 144$ et $b = 233$.

Annexe 3 : Schéma Logisim avant la réécriture de la table d'instruction



Cette annexe est disponible en haute résolution en JPG sur le dépôt GitHub dans « Dossier TM » ou en scannant ce code QR.