
Projet PARM

Polytech ARM-based embedded processor

N. Bounouas, P.E. Novac, T. Niget, T. Prévost, E. Vouriot, B. Miramond
Polytech Nice Sophia

5 novembre 2021

https://bitbucket.org/edge-team-leat/parm_public



Table des matières

1	Présentation du projet	5
1.1	Le microprocesseur ARM Cortex-M0	5
1.2	Le projet	5
1.3	Logisim	5
1.3.1	Mode édition	7
1.3.2	Création d'un premier circuit	7
1.3.3	Mode simulation	7
1.3.4	Design hiérarchique	8
1.3.5	Bus et séparateurs	9
2	Décodeur 7 segment	10
2.1	Introduction	10
2.2	Table de correspondance	11
2.3	Mise en place sur logisim	11
3	Organisation du travail	14
4	ALU	14
4.1	Description	14
4.2	Interface	15
4.2.1	Entrées	15
4.2.2	Sorties	15
4.3	Opérations	15
4.4	Drapeaux	15
5	Banc de registres	16
5.1	Description	16
5.2	Interface	16
5.2.1	Entrées	16
5.2.2	Sorties	17
5.3	Interaction avec l'ALU	17
6	Contrôleur	17
6.1	Description	17
6.2	Interface	18
6.2.1	Entrées	18
6.2.2	Sorties	18
6.3	Sous composants	18
6.3.1	Décodeur d'instruction	18
6.3.1.1	Description	18
6.3.1.2	Interface	19
6.3.2	Shift, add, sub, mov	19
6.3.2.1	Description	19
6.3.2.2	Interface	20
6.3.3	Data Processing	20
6.3.3.1	Description	20
6.3.3.2	Interface	20
6.3.4	Flags APSR	21
6.3.4.1	Description	21
6.3.4.2	Interface	21
6.3.5	Load/Store	21
6.3.5.1	Description	21
6.3.5.2	Interface	22

6.3.6	SP Address	22
6.3.6.1	Description	22
6.3.6.2	Interface	23
6.3.7	Conditional	23
6.3.7.1	Description	23
6.3.7.2	Interface	23
6.3.8	Program Counter	23
6.3.8.1	Description	24
6.3.8.2	Interface	24
6.3.9	Stack Pointer	24
6.3.9.1	Description	24
6.3.9.2	Interface	25
7	Chemin de données	25
7.1	Description	25
7.2	CPU	25
7.2.1	Description	25
7.2.2	Interface	26
7.2.2.1	Entrées	26
7.2.2.2	Sorties	26
7.3	RAM	26
7.3.1	Description	26
7.3.2	Interface	27
7.3.2.1	Entrées	27
7.3.2.2	Sorties	27
7.4	ROM	27
7.4.1	Description	27
7.4.2	Interface	27
7.4.2.1	Entrées	27
7.4.2.2	Sorties	27
8	Assembleur	27
8.1	Introduction	28
8.2	Syntaxe	28
8.3	Syntaxe Logisim	29
9	Jeu d'instructions (Instruction Set Architecture)	29
9.1	Instructions à implémenter	29
9.1.1	Shift, add, sub, mov	29
9.1.1.1	LSL (immediate) : Logical Shift Left (p. 282)	29
9.1.1.2	LSR (immediate) : Logical Shift Right (p. 284)	30
9.1.1.3	ASR (immediate) : Arithmetic Shift Right (p. 203)	30
9.1.1.4	ADD (register) : Add register (p. 192)	30
9.1.1.5	SUB (register) : Subtract register (p. 404)	31
9.1.1.6	ADD (immediate) : Add 3-bit immediate (p. 190)	31
9.1.1.7	SUB (immediate) : Subtract 3-bit immediate (p. 402)	31
9.1.1.8	MOV (immediate) : Move (p. 291)	32
9.1.1.9	CMP (immediate) : Compare (p. 223)	32
9.1.1.10	ADD (immediate) : Add 8-bit immediate (p. 190)	32
9.1.1.11	SUB (immediate) : Subtract 8-bit immediate (p. 402)	33
9.1.2	Data processing	33
9.1.2.1	AND (register) : Bitwise AND (p. 201)	33
9.1.2.2	EOR (register) : Exclusive OR (p. 233)	34
9.1.2.3	LSL (register) : Logical Shift Left (p. 283)	34

9.1.2.4	LSR (register) : Logical Shift Right (p. 285)	34
9.1.2.5	ASR (register) : Arithmetic Shift Right (p. 204)	35
9.1.2.6	ADC (register) : Add with Carry (p. 188)	35
9.1.2.7	SBC (register) : Subtract with Carry (p. 347)	36
9.1.2.8	ROR (register) : Rotate Right (p. 339)	36
9.1.2.9	TST (register) : Set flags on bitwise AND (p. 420)	36
9.1.2.10	RSB (immediate) : Reverse Subtract from 0 (p. 341)	37
9.1.2.11	CMP (register) : Compare Registers (p. 224)	37
9.1.2.12	CMN (register) : Compare Negative (p. 222)	37
9.1.2.13	ORR (register) : Logical OR (p. 310)	38
9.1.2.14	MUL : Multiply Two Registers (p. 302)	38
9.1.2.15	BIC (register) : Bit Clear (p. 210)	38
9.1.2.16	MVN (register) : Bitwise NOT (p. 304)	39
9.1.3	Load/Store	39
9.1.3.1	STR (immediate) : Store Register (p. 386)	39
9.1.3.2	LDR (immediate) : Load Register (p. 246)	39
9.1.4	Miscellaneous 16-bit instructions	40
9.1.4.1	ADD (SP plus immediate) : Add Immediate to SP (p. 194)	40
9.1.4.2	SUB (SP minus immediate) : Subtract Immediate from SP (p. 402)	40
9.1.5	Branch	41
9.1.5.1	B : Conditional Branch (p. 205)	41
9.1.5.2	B : Unconditional Branch (p. 205)	41
9.2	Conditions (p. 178)	42
9.3	Drapeaux (p. 31)	42
10	Pour aller plus loin	42
10.1	Compilation de code C	42
10.2	Périphériques d'entrée/sortie mappés en mémoire	43
10.2.1	Exemple : DIP switch	44
10.2.2	Exemple : Terminal	44

1 Présentation du projet

1.1 Le microprocesseur ARM Cortex-M0

La famille des ARM Cortex-M regroupe des processeurs 32 bits. Ils peuvent être utilisés comme microprocesseur ou microcontrôleur. On les retrouve dans diverses applications : Arduino Due, machine à laver, distributeur de boissons... Les Cortex-M visent en majorité le marché de l'embarqué.

Le but de ce projet est de simuler le comportement d'un Cortex-M0 au moyen d'un logiciel de simulation électronique : Logisim. L'idée est ici d'obtenir un système ayant un comportement similaire à un Cortex-M0 et non une copie conforme du fait de la complexité d'un processeur réel.

1.2 Le projet

Durant ce projet nous allons implémenter notre microprocesseur en le divisant en plusieurs blocs :

- Partie matérielle :
 - ALU 32 bits
 - Banc de 8 registres
 - Contrôleur
- Partie logicielle :
 - Assembleur
 - Exportation sur FPGA

Spécificités de l'implémentation

- Pas de gestion des interruptions
- Pas de gestion des appels de fonctions
- Pas d'optimisation
- Pas de pipeline
- Pas de FPU¹
- Pas de MMU²
- Toutes les instructions s'exécutent en 1 cycle (excepté les instructions LDR/STR en 2 cycles)
- Instructions sur 16 bits
- Données sur 32 bits
- Adressage RAM/ROM sur 9 bits
- Adressage RAM uniquement sur la pile (en utilisant le *Stack Pointer*)

1.3 Logisim

Logisim est un programme permettant la modélisation et la simulation de circuit logique. La modélisation du circuit ne se fait que par dessin et glisser-déposer des différents éléments électroniques.

Installation : la version de Logisim utilisé dans le cadre du projet est la 3.5.0, accessible à l'adresse <https://github.com/logisim-evolution/logisim-evolution/releases/tag/v3.5.0>. Sur Windows, utilisez le fichier .msi pour installer. Sur Mac, utilisez le fichier .dmg. Sur Ubuntu/Debian, passez par le fichier .deb avec la commande :

¹Floating-Point Unit, unité de calcul en virgule flottante

²Memory Management Unit, unité de gestion mémoire, gère la traduction des adresses virtuelles en adresses physiques au sein du processeur

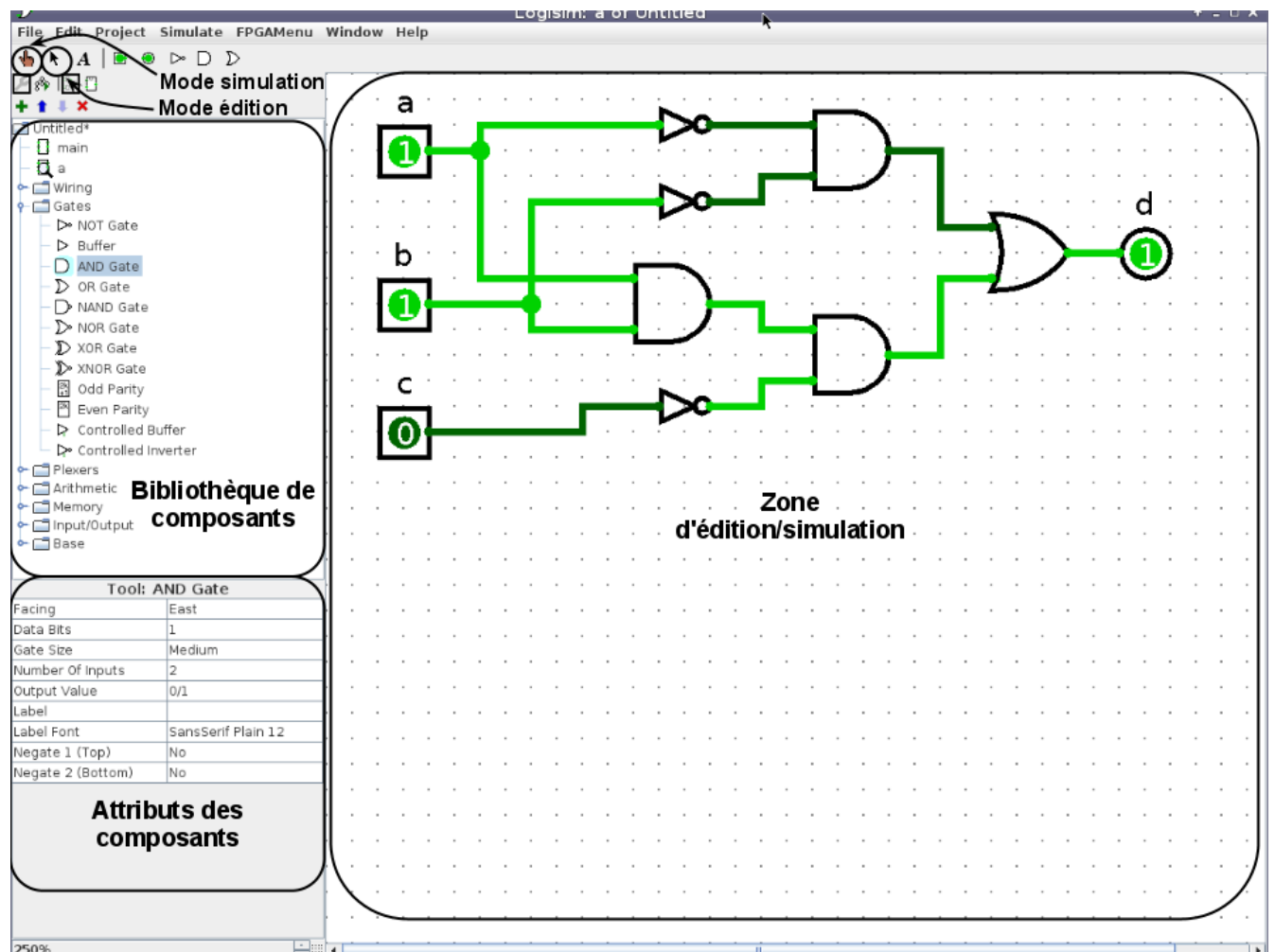


FIG. 1 : Interface de Logisim

```
sudo apt install gdebi
sudo gdebi logisim-evolution_3.5.0-1_amd64.deb
```

Pour les autres systèmes, vous pouvez lancer directement le fichier .jar sans installer. La commande à utiliser est :

```
java -Dsun.java2d.opengl=true -jar logisim-evolution.jar
```

Cette section est un extrait du tutoriel officiel « Introduction à l'utilisation de Logisim ». Pour plus d'informations, se référer à la documentation officielle.

Vous pouvez voir l'interface de Logisim sur la Figure 1.

Une des particularités de Logisim est de pouvoir éditer et simuler un circuit en même temps. Nous expliquerons plus tard dans ce document comment simuler un circuit, puis comment l'implémenter sur la carte du laboratoire.

1.3.1 Mode édition

1. Pour utiliser le mode édition, il faut simplement sélectionner la flèche comme indiqué en haut de la figure 1.
2. On peut alors choisir un composant dans la bibliothèque sur la gauche. Pour l'ajouter dans son schéma, il suffit de cliquer sur le composant désiré, puis de cliquer sur le schéma.
3. Chaque composant que vous utiliserez aura des attributs modifiables dans la zone inférieur gauche de Logisim. Par exemple si l'on pose une porte AND, on peut modifier le nombre de signaux qu'elle prend en entrée, ou encore mettre un inverseur sur une de ses entrées.
4. Il est aussi possible de faire des copier/coller d'un ou plusieurs composants. Dans ce cas, les composants conserveront aussi tous les attributs préalablement définis.
5. Une fois que l'on a posé tous les composants, il faut alors les connecter. Pour cela il suffit de placer le curseur avec la souris sur un des ports à connecter et, en gardant pressé le bouton gauche de la souris, le déplacer jusqu'au port de destination.

1.3.2 Création d'un premier circuit

Tous les circuits réalisés dans Logisim peuvent être réutilisés dans d'autres circuits. Afin de créer un nouveau circuit, il faut aller dans **Projet → Ajouter Circuit...** et nommer le circuit. Le circuit créé devient un composant disponible dans la bibliothèque.

1.3.3 Mode simulation

Logisim est capable de simuler le circuit en affichant les valeurs des signaux directement sur le schéma. L'utilisateur peut alors définir les valeurs des bits en entrée et observer la réaction du design.

1. Pour utiliser le mode simulation, il faut sélectionner la main en haut à gauche de Logisim (cf Figure 1)
2. Il est alors possible de contrôler l'état des différentes entrées en cliquant directement dessus.
3. En cliquant sur une entrée, la valeur doit alterner entre 0 ou 1.
4. Voici un descriptif des couleurs utilisées pour les signaux en mode simulation.

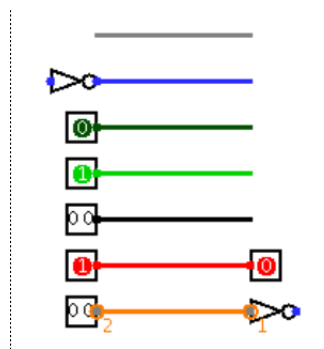


FIG. 2 : Couleurs des fils en simulation

- **Gris** : La taille du fil est inconnue. Le fil n'est relié à aucune entrée ou sortie.
- **Bleu** : Le fil comporte une valeur, cependant elle est inconnue.
- **Vert foncé** : Le fil comporte la valeur 0.
- **Vert clair** : Le fil comporte la valeur 1.
- **Noir** : Le fil comporte plusieurs bits (bus).
- **Rouge** : Le fil comporte une erreur.

- **Orange** : Les composants reliés au fil n'ont pas la bonne taille.

1.3.4 Design hiérarchique

La méthodologie de design que l'on vient d'utiliser est valable pour la conception de systèmes numériques plutôt simples, c'est-à-dire avec un nombre de portes logiques plutôt bas. Lorsque l'on vise des systèmes plus compliqués on risque de voir le nombre de portes et de connexions exploser. Dans ce cas, le risque d'introduire des erreurs devient très important.

La clé pour gérer correctement une complexité plus grande est d'utiliser le design hiérarchique. Grâce au design hiérarchique on peut travailler à différents niveaux d'abstraction. D'abord on décrit des blocs de base à l'aide des portes logiques, pour ensuite utiliser ces blocs de base comme parties d'un système plus large.

Pour créer un design hiérarchique il faudra suivre les pas suivants :

1. Créez un nouveau circuit comme déjà expliqué dans la section 1.3.2 et nommez le. Pour passer de l'édition d'un circuit à l'autre, il suffit de double-cliquer sur le nom de celui désiré dans le menu de gauche.
2. Il est alors possible d'ajouter un sous circuit de la même manière que l'utilisation d'un composant quelconque. On clique sur le sous circuit en question dans le menu indiqué sur la Figure 3, puis on le place en cliquant sur la zone d'édition.
3. Si le sous circuit avait été créé correctement, alors il devrait être représenté par un petit bloc, avec sur sa gauche des points bleus correspondant aux entrées, et sur sa droite des points verts correspondant aux sorties.
4. Si les sorties apparaissent en bleu et non en vert sur le schéma, vérifiez que vous avez bien affecté l'attribut `Sortie?` = Oui dans les Pins de sortie.

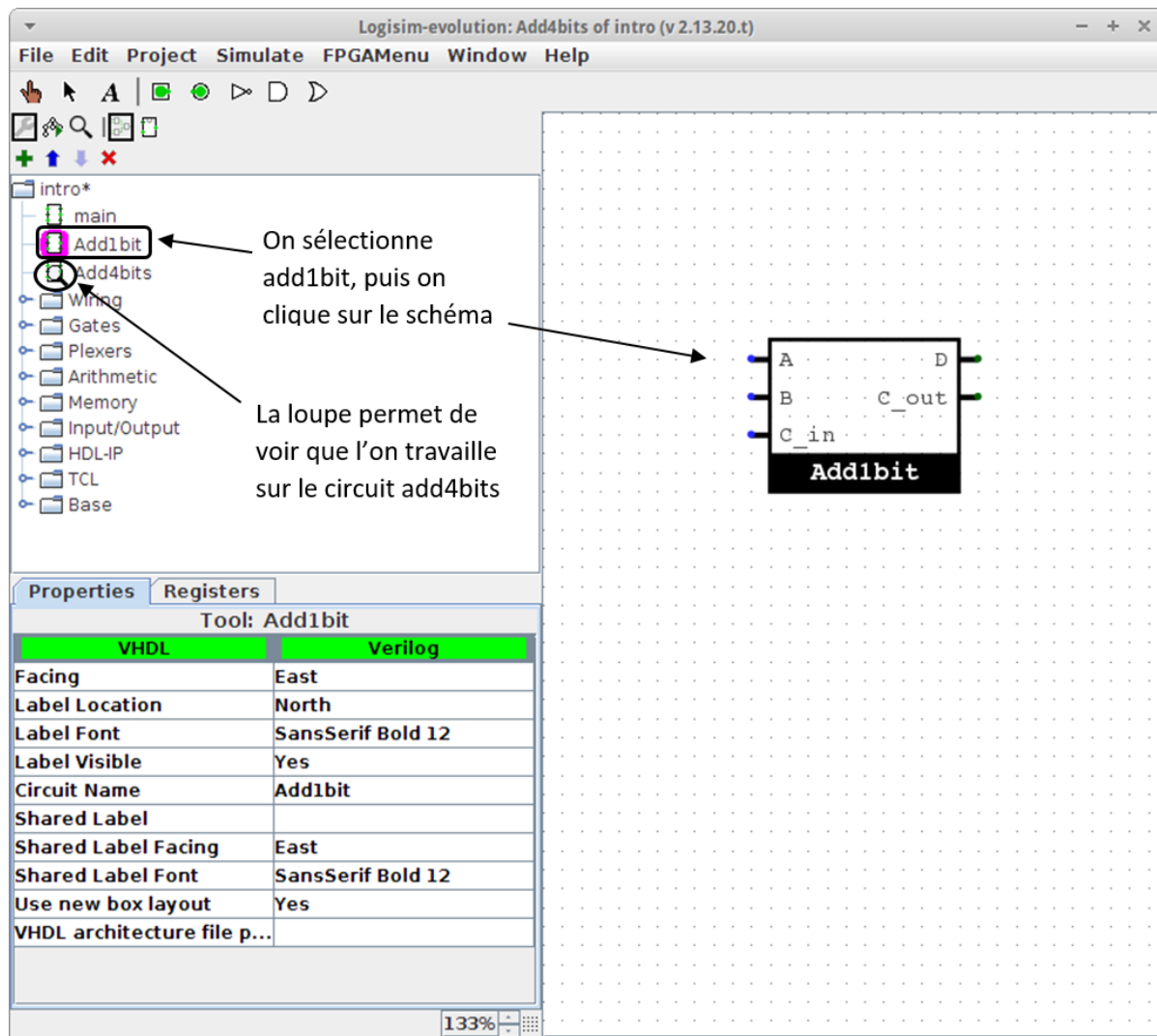


FIG. 3 : Sous circuit

1.3.5 Bus et séparateurs

Pour l'implémentation de composant travaillant avec des données sur plusieurs bits (un additionneur par exemple), il devient nécessaire de regrouper plusieurs signaux en créant un bus de données. Par exemple, pour définir l'entrée A comme un bus de 4 bits, il faut ajouter un élément Pin et définir sa taille via l'attribut `Data bits = 4`.

Lorsque l'on tire un fil de l'une de ces entrées, ce n'est plus un simple signal mais un bus de 4 bits. Pour pouvoir connecter les éléments de ce bus aux entrées de plusieurs composants travaillant sur chacun un bit, on va devoir séparer les différents fils du bus afin de pouvoir les traiter un par un. L'élément Séparateur de Câblage permet d'effectuer ces conversions dans les deux sens : d'un bus de 4 bits vers 4 fils, et de 4 fils vers un bus de 4 bits – voir Figure 4.

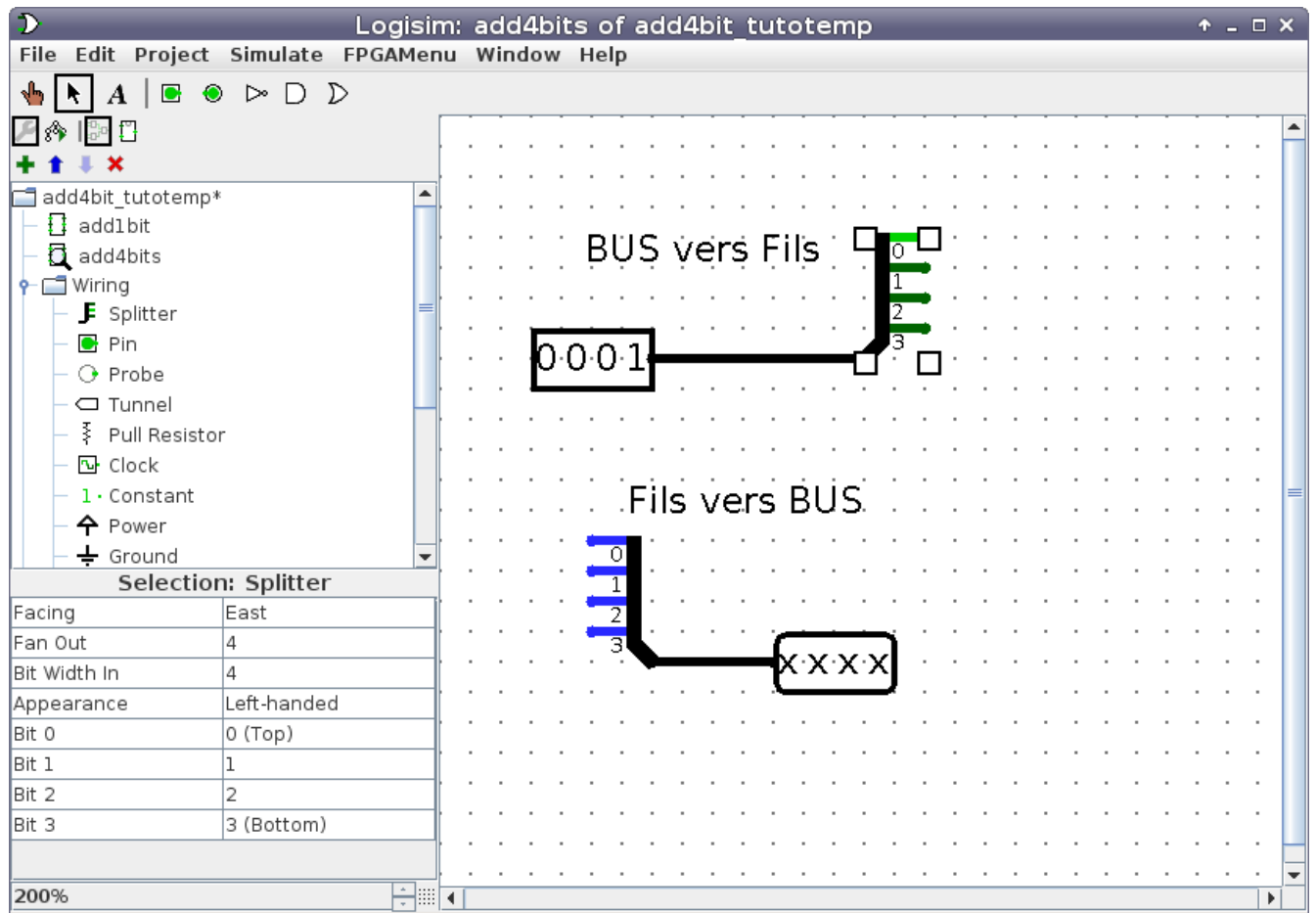


FIG. 4 : Exemples splitters

Il faut définir les tailles d'entrées et de sorties du Séparateur via les attributs Ventilation en sortie et Largeur de bits en entrée.

Note : le bit de poids faible est indexé à 0 en sortie du splitter.

Remarque : n'utiliser que des composants dont l'indicateur VHDL dans la zone Properties est vert. Les autres (Buffer contrôlé par exemple) ne pourront pas être déployés sur la carte FPGA.

2 Décodeur 7 segment

2.1 Introduction

Nous allons commencer par une prise en main de Logisim en réalisant un décodeur 7 segments. Ce type d'afficheur est un grand classique en ce qui concerne l'affichage de caractères hexadécimaux.

Le principe de cet afficheur est très simple, En allumant plusieurs segments en même temps nous allons pouvoir représenter les caractères suivants : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

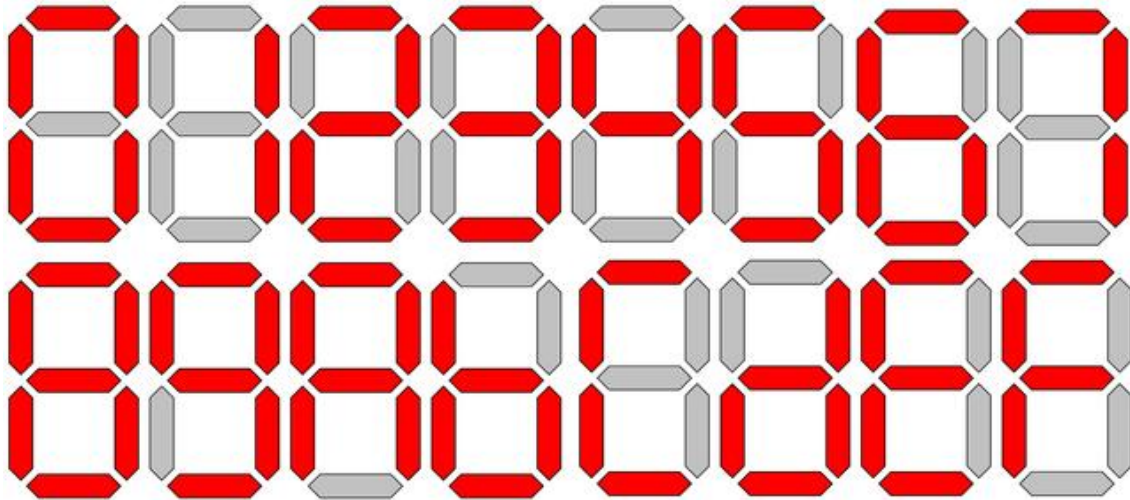


FIG. 5 : Un affichage hexadécimal par 7 segments. Merci à @Skywodd - <https://www.carnetdumaker.net/>

L'objectif est ici de recevoir une information sur 4 bits (comprise entre 0b0000 et 0b1111) et de la traduire sur 7 bits correspondant aux segments à allumer et à ceux à éteindre³.

2.2 Table de correspondance

Nous allons ici associer à chaque valeur à sa décomposition en segment d'après le schéma suivant :

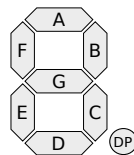





FIG. 6 : Détail et Annotation d'un 7 segment par H2g2bob -   

2.3 Mise en place sur logisim

La table de vérité présentée dans la section précédente peut être utilisée pour obtenir une fonction simplifiée de chaque segment en utilisant les tables de Karnaugh⁴. Logisim embarque une fonctionnalité permettant d'effectuer cette analyse de manière simplifiée. Pour cela il est nécessaire de lancer Logisim avec l'option -analyze, soit dans notre cas :

```
java -Dsun.java2d.opengl=true -jar logisim-evolution.jar -analyze
```

³<http://www.electronics-tutorials.ws/blog/7-segment-display-tutorial.html>

⁴https://fr.wikipedia.org/wiki/Table_de_Karnaugh

En utilisant ce paramètre une option apparaît dans le menu "Projet" -> "Analyser le circuit". Il est possible dans l'onglet "table" de remplir la table de vérité du circuit. Une fois le tableau complété, cliquer sur "Construire le circuit" générera le circuit correspondant. Ce dernier devrait ressembler à ceci :

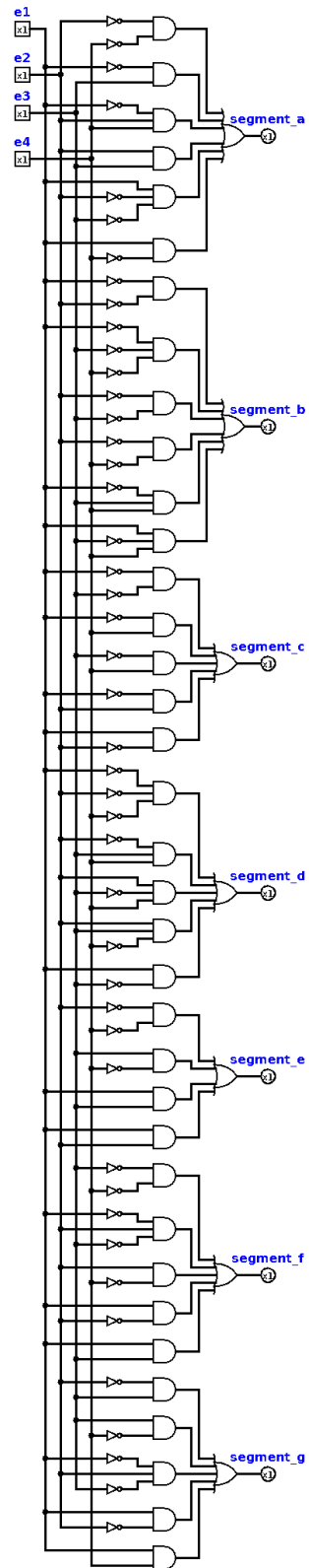



FIG. 7 : Contenu du composant "Seven segment decoder"

3 Organisation du travail

Le projet est constitué des parties / tâches suivantes qui seront à répartir au sein du groupe. Cette répartition n'est pas "gravée dans la roche", assurez-vous simplement qu'à la fin du projet il n'y ait pas de trop grandes disparités de quantité de travail effectuée par les différents membres du groupe.

Chaque partie dans la liste est accompagnée de sa difficulté approximative.

- ★★ ALU (Matériel)
 - Réaliser les blocs d'opérateurs arithmétiques et logiques
 - Calculer les flags
- ★ Banc de registres (Matériel)
- ★ Contrôleur - Décodeur d'instruction (Matériel) 
- ★★★ Contrôleur - Shift, add, sub, mov (Matériel)
- ★★★ Contrôleur - Data Processing (Matériel)
- ★★ Contrôleur - Flags APSR (Matériel)
- ★★★ Contrôleur - Load/Store (Matériel)
- ★★ Contrôleur - SP Address (Matériel)
- ★★ Contrôleur - Conditional (Matériel)
- ★★ Chemin de données (Matériel)
 - Relier le contrôleur, le banc de registres et l'ALU
- ★★★ Assembleur (Logiciel)
 - Parser un fichier assembleur
 - Générer le fichier binaire à charger dans la mémoire d'instruction de Logisim
- ★ Compilation de code C (Logiciel)

4 ALU

4.1 Description

L'unité arithmétique et logique est l'élément qui se charge des calculs au sein du processeur. Les ALU les plus basiques ne font que des opérations sur des entiers cependant on trouve des ALU spécialisées. Les calculs sur ces dernières vont des opérations à virgule flottante jusqu'à des calculs plus complexes tels que des racines carrées, des logarithmes, des sinus ou cosinus... Notre ALU n'effectuera que des calculs simples (addition, soustraction, multiplication, décalage) sur des entiers de 32 bits.

Une ALU comporte deux entrées amenant les données à traiter. Une troisième entrée permet de désigner le calcul à effectuer. En sortie on retrouvera le résultat de l'opération ainsi que des drapeaux. Ces drapeaux représentent une série d'état à la suite d'un calcul : un résultat négatif, un résultat nul, un débordement ou encore une retenue.

L'entrée *Shift* indique le nombre de décalage pour les opérations de décalage.

Remarque : les instructions TST, CMP, CMN n'enregistrent pas le résultat de l'opération. Seuls les drapeaux sont mis à jour. Pour ces opérations en particulier, il faudra recopier l'entrée B sur la sortie S. Le contrôleur définira le même registre pour le registre R_n d'opérande B que pour le registre R_d de destination de la sortie S.

Remarque 2 : pour l'instruction SBC, la retenue entrante doit être inversée (voir 9.1.2.7 SBC (register) : *Substract with Carry* (p. 347)).

4.2 Interface

4.2.1 Entrées

Port	Taille	Description
A	32	Première opérande
B	32	Seconde opérande
Shift	5	Nombre de décalage
CarryIn	1	Retenu entrente
Codop	4	Code opération ALU

4.2.2 Sorties

Port	Taille	Description
S	32	Registre résultat
Flags	4	Registre drapeaux, ordre : NZCV

4.3 Opérations

Ces opérations de l'ALU correspondent exactement aux instructions de la catégorie *Data Processing*. En cas de doute, se référer à 9 *Jeu d'instructions (Instruction Set Architecture)*.

Codop	Opération	Instructions	Remarque
0000	A and B	AND	
0001	A xor B	EOR	
0010	B << Shift	LSL	Retenue sortante, voir jeu d'instructions
0011	B >> Shift	LSR	Retenue sortante, voir jeu d'instructions
0100	B >> Shift (arith)	ASR	Retenue sortante, voir jeu d'instructions
0101	A + B + CarryIn	ADC	
0110	B - A + CarryIn - 1	SBC	Retenue entrante inversée
0111	B >> Shift (rot)	ROR	Retenue sortante, voir jeu d'instructions
1000	A and B	TST	Résultat perdu, seuls les drapeaux sont mis à jour
1001	-A	RSB	Registre Rm utilisé plutôt que Rn
1010	B - A	CMP	Résultat perdu, seuls les drapeaux sont mis à jour
1011	A + B	CMN	Résultat perdu, seuls les drapeaux sont mis à jour
1100	A or B	ORR	
1101	A * B	MUL	
1110	B and not A	BIC	
1111	not B	MVN	Complément binaire

4.4 Drapeaux

Ces drapeaux de l'ALU correspondent exactement aux drapeaux de l'architecture ARM. En cas de doute, se référer à 9 *Jeu d'instructions (Instruction Set Architecture)* et 9.3 *Drapeaux (p. 31)*.

Symbole	Nom	Description
N	Negative	Résultat négatif
Z	Zero	Résultat nul
C	CarryOut	Retenue sortante (dépassement de capacité non-signé).
V	Overflow	Dépassement de capacité signé

5 Banc de registres

5.1 Description

Le processeur ne stocke pas le résultat d'un calcul directement en RAM, pour cela il travaille avec une mémoire rapide mais petite constituée de quelques registres (20 registres de 32 bits un dans Cortex-M0).

Parmi les 20 registres présents dans un Cortex-M0, seuls 13 d'entre eux sont dédiés à un usage général. Les 7 autres ont un usage bien particulier (voir tableau suivant)

Nom	Accès	Valeur de remise à zéro	Description	A implémenter
R0 - R12	LE	Inconnue	Registres à usage général	OUI (uniquement 8)
MSP	LE	?	Pointeur de pile	Oui (contrôleur)
PSP	LE	Inconnue	Pointeur de pile	Oui (contrôleur)
LR	LE	Inconnue	Registre de lien	Non
PC	LE	?	Compteur d'instruction	Oui (contrôleur)
PSR	LE	Inconnue	Statut du programme	Non
ASPSR	LE	Inconnue	Statut d'application du programme	Non
IPSR	L	0x00000000	Statut d'interruption du programme	Non
EPSR	L	Inconnue	Statut d'exécution du programme	Non
PRIMASK	LE	0x00000000	Masque de priorité	Non
CONTROL	LE	0x00000000	Registre de contrôle	Non

Dans notre cas, le banc de registre ne contient que 8 registres R0-R7 par soucis de simplicité. Les registres PC et SP sont implémentés dans le contrôleur.

5.2 Interface

5.2.1 Entrées

Port	Taille	Description
DataIn	32	Données entrantes, à enregistrer dans le registre sélectionné
RegDest	3	Sélection du registre de destination des données entrantes
Clk	1	Horloge
Reset	1	Remise à zéro
RegA	3	Sélection du registre A pour les données sortantes
RegB	3	Sélection du registre B pour les données sortantes

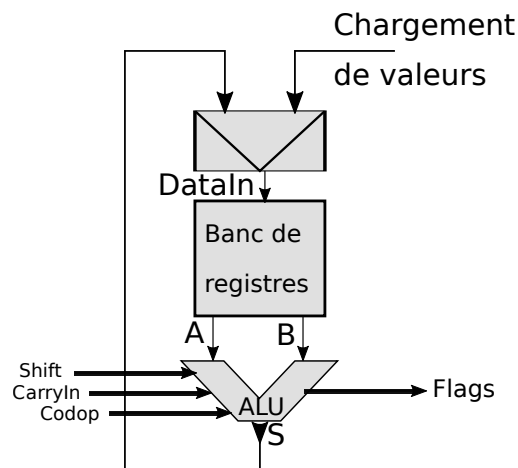
5.2.2 Sorties

Port	Taille	Description
AOut	32	Données sortantes du registre sélectionné A
BOut	32	Données sortantes du registre sélectionné B
R0	32	Registre interne 0
R1	32	Registre interne 1
R2	32	Registre interne 2
R3	32	Registre interne 3
R4	32	Registre interne 4
R5	32	Registre interne 5
R6	32	Registre interne 6
R7	32	Registre interne 7

Les sorties R0–R7 ne seront pas utilisées pour implémenter une quelconque fonctionnalité. Elles sont présentes pour aider à visualiser le comportement du processeur.

5.3 Interaction avec l'ALU

Après avoir réalisé l'ALU et le banc de registres, l'interaction entre ces composants peut être mise en oeuvre de la manière suivante :



Il est ainsi possible de valider leur comportement en enregistrant des données dans le banc de registre (à l'aide des ports DataIn et RegDest) puis en effectuant diverses opération par l'ALU en spécifiant le Codop.

6 Contrôleur

6.1 Description

Le contrôleur ou unité de contrôle (Control unit en anglais) joue le rôle de chef d'orchestre au sein du processeur. Il est en charge du décodage des instructions. En fonction des informations récupérées au sein de l'instruction et des différents drapeaux, le contrôleur va agir sur le chemin de données.

Il est responsable du choix de la source fournissant les données. La sortie Load selon si elle est à 0 ou à 1

choisira respectivement un chargement depuis l'ALU ou directement depuis la RAM. Le même mécanisme est utilisé pour choisir la source du nombre de décalage, des immédiats 8 (voir tableau des sorties).

Le contrôleur abrite de plus le compteur ordinal et l'incrémente lorsqu'il traite une instruction.

6.2 Interface

6.2.1 Entrées

Port	Taille	Description
Inst	16	Instruction à traiter
Flags	4	Drapeaux en entrée, ordre NZCV
Clk	1	Horloge
Reset	1	Remise à Zero

6.2.2 Sorties

Port	Taille	Description
Carry	1	Retenue sortante (provenant des drapeaux) à destination de l'ALU
DP_Shift	1	Provenance du nombre de décalage : 0 pour registre A, 1 pour Imm5
Imm32_Enable	1	Provenance de la donnée A : 0 pour registre, 1 pour Imm32
Imm5	5	Nombre de décalage pour les instructions de décalage de la catégorie <i>Shift, add, sub, mov</i>
Imm32	8	Valeur pour les instructions MOV, CMP, ADD et SUB
ALU_Opcode	4	Code opération à destination de l'ALU
Rm	3	Sélection du registre pour lecture de l'opérande A
Rn	3	Sélection du registre pour lecture de l'opérande B
Rd	3	Sélection du registre pour enregistrement du résultat
RAM_Addr	32	Adresse mémoire des instructions <i>Load/Store</i>
Load	1	Provenance des données en entrée du banc de registre
Store	1	1 pour stocker la valeur du registre Rm à l'adresse RAM_Address
PC	16	Compteur ordinal : adresse de la prochaine instruction en ROM
Flags	4	Valeur des drapeaux issue de Flags APSR
SP	32	Valeur de SP

6.3 Sous composants

6.3.1 Décodeur d'instruction

6.3.1.1 Description

Le bloc *Opcode Decoder* active l'une de ses sorties en fonction de la catégorie d'instruction reconnue, afin d'activer les blocs correspondant du contrôleur.

L'entrée *Opcode* correspond au code opération pré-extrait de l'instruction, c'est à dire les 6 bits de poids fort de l'instruction.

Les sorties doivent être activées en fonction du code binaire correspondant à chacune d'elle. Voir 9 *Jeu d'instructions (Instruction Set Architecture)*.

6.3.1.2 Interface

Entrées

Port	Taille	Description
Opcode	6	Code opération à décoder

Sorties

Port	Taille	Description
Shift	1	Active le bloc <i>Shift, add, sub, mov</i>
Data_Processing	1	Active le bloc <i>Data Processing</i>
Load_Store	1	Active le bloc <i>Load/Store</i>
SP_Address	1	Active le bloc <i>SP Address</i>
Branch	1	Active le bloc <i>Conditional</i>

6.3.2 Shift, add, sub, mov

6.3.2.1 Description

Le bloc *Shift, add, sub, mov* traite les instructions de calcul de la catégorie *Shift, add, sub, mov* (voir 9.1.1 *Shift, add, sub, mov*).

La sortie `ALU_Opcode` devra être définie pour chacune des instructions afin d'exécuter la bonne opération dans l'ALU (voir 4.3 *Opérations*).

Certaines instructions ne mettent pas à jour tous les drapeaux. La sortie `Flags_Update_Mask` doit être définie en conséquence avec le masque dont les bits à 1 correspondent aux drapeaux à mettre à jour.

La sortie `Carry` force la valeur de la retenue entrante pour l'ALU. Les instructions `ADD` et `SUB` n'utilisent pas de retenue entrante. Elle doit être définie à 0 pour `ADD` et à 1 pour `SUB` étant donnée qu'elle est inversée pour la soustraction dans l'ALU.

La sortie `Imm32` est utilisée pour communiquer les valeurs des immédiats de `MOV`, `CMP`, `ADD` et `SUB` à l'ALU. Dans ce cas, la sortie `Imm32_Enable` doit être définie à 1. Les immédiats devront être étendus de 3 ou 8 bits vers 32 bits en complétant par des zéros (ce sont des entiers non-signés).

La sortie `Imm5` est utilisée en tant que valeur du nombre de décalage pour les instructions de décalage `LSL`, `LSR` et `ASR`.

Si l'entrée `Enable` est à 0, les sorties sont forcées à 0.

Remarque : l'immédiat pour l'instruction `MOV` passe par l'ALU pour être enregistré dans le registre de destination. Pour que la valeur ne soit pas modifiée, on peut l'inverser ici et utiliser l'opération `RSB` dans l'ALU.

Remarque 2 : pour les opérations de décalage, on préférera utiliser le registre `Rn` à la place de `Rm` pour rester cohérent avec les instructions de la catégorie *Data Processing*. L'ALU travaillera uniquement sur l'opérande B.

Remarque 3 : quand une instruction ne fait pas usage de certaines sorties, elles doivent être définies à 0.

6.3.2.2 Interface Entrées

Port	Taille	Description
Instruction	16	Instruction <i>Shift, add, sub, mov</i> à décoder
Enable	1	Active le composant. Si 0, les sortie sont forcées à 0

Sorties

Port	Taille	Description
ALU_Opcode	4	Code opération à destination de l'ALU
Rm	3	Sélection du registre opérande A
Rn	3	Sélection du registre opérande B
Rd	3	Sélection du registre résultat
Flags_Update_Mask	4	Masque de mise à jour des drapeaux à destination du bloc <i>Flags APSR</i>
Carry	1	Valeur à utiliser en tant que retenue entrante pour l'ALU
Imm32_Enable	1	Indique au processeur d'utiliser la sortie Imm32 à la place du registre Rm
Imm5	5	Nombre de décalage à destination de l'ALU
Imm32	32	Valeur à utiliser en tant qu'opérande A de l'ALU

6.3.3 Data Processing

6.3.3.1 Description

Le bloc *Data Processing* traite les instructions de calcul de la catégorie *Data Processing* (voir 9.1.2 *Data processing*).

La sortie *ALU_Opcode* correspond aux bits 6 à 9, les code opérations de l'ALU ayant été implémenté en conséquence.

Certaines instructions ne mettent pas à jour tous les drapeaux. La sortie *Flags_Update_Mask* doit être définie en conséquence avec le masque dont les bits à 1 correspondent aux drapeaux à mettre à jour.

Si l'entrée *Enable* est à 0, les sorties sont forcées à 0.

Remarque : les instructions *TST*, *CMP* et *CMN* ne mettent à jour que les drapeaux et n'enregistrent pas le résultat de l'opération. Pour celles-ci, le registre *Rd* sera défini à *Rn* afin d'être cohérent avec les autres instructions, et que l'ALU puisse copier le contenu de l'opérande B dans le résultat.

Remarque 2 : pour les instructions *RSB* et *MUL*, les bits 3 à 5 définissent le registre *Rn*, contrairement aux autres instructions pour lesquelles ces bits définissent le registre *Rm*. Pour simplifier l'implémentation du contrôleur, on considèrera toujours que les bits 3 à 5 correspondent à *Rm* et que les bits 0 à 2 correspondent à *Rdn*. Pour *RSB*, l'ALU devra alors faire usage de l'opérande A (en provenance du registre *Rm*).

6.3.3.2 Interface Entrées

Port	Taille	Description
Instruction	16	Instruction <i>Data Processing</i> à décoder
Enable	1	Active le composant. Si 0, les sortie sont forcées à 0

Sorties

Port	Taille	Description
ALU_Opcode	4	Code opération à destination de l'ALU
Rm	3	Sélection du registre opérande A
Rn	3	Sélection du registre opérande B
Rd	3	Sélection du registre résultat
Flags_Update_Mask	4	Masque de mise à jour des drapeaux à destination du bloc <i>Flags APSR</i>

6.3.4 Flags APSR

6.3.4.1 Description

Le bloc *Flags APSR* correspond au 4 bits de poids fort du registre Application Program Status Register de l'architecture ARM. Il conserve les drapeaux générés par la dernière instruction, afin qu'ils soient disponibles pour la prochaine instruction.

L'entrée *Update_Mask* permet de réinjecter l'ancien état d'un drapeau si le bit correspondant est à 0. Si le bit est à 1, le drapeau est mis à jour.

6.3.4.2 Interface

Entrées

Port	Taille	Description
Flags_In	4	Nouveaux drapeaux générés par l'instruction courante, ordre NZCV
Update_Mask	4	Masque d'enregistrement des drapeaux, ordre NZCV
Clk	1	Horloge
Reset	1	Remise à zéro

Sorties

Port	Taille	Description
Flags_Out	4	Drapeaux générés par l'instruction précédente, ordre NZCV

6.3.5 Load/Store

6.3.5.1 Description

Le bloc *Load/Store* traite les instructions de lecture/écriture en mémoire (voir 9.1.3 *Load/Store*).

La sortie *RAM_Addr* correspond à l'adresse mémoire à laquelle effectuer l'opération. Elle est calculée en fonction de la valeur actuelle du *Stack_Pointer* et de l'offset provenant de l'instruction.

La sortie *Store* indique à la mémoire de stocker la donnée du registre *Rm* à l'adresse *RAM_Addr*. La donnée sera écrite au cycle suivant.

La sortie *Load* indique au processeur de présenter la donnée à l'adresse *RAM_Addr* en entrée du banc de registre (*DataIn*). La donnée sera disponible au cycle suivant.

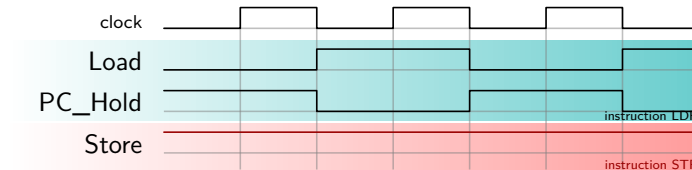
La sortie *PC_Hold* retarde l'incrémentation du *Program Counter* d'un coup d'horloge. La RAM étant syn-

chone, elle a besoin d'un cycle pour traiter l'opération de lecture. La donnée lue n'est donc pas disponible immédiatement, et le processeur ne peut pas commencer à exécuter l'instruction suivante. La sortie Load devra donc elle aussi être retardée d'un cycle.

Si l'entrée Enable est à 0, les sorties sont forcées à 0.

Astuce : utiliser une bascule D pour activer PC_Hold et retarder Load.

Pour vous aider, voici un chronogramme du comportement attendu :



6.3.5.2 Interface

Entrées

Port	Taille	Description
Instruction	16	Instruction de lecture/écriture en mémoire à décoder
Enable	1	Active le composant. Si 0, les sortie sont forcées à 0
Stack_Pointer	32	Valeur courante du pointeur de pile
Clk	1	Horloge
Reset	1	Remise à zéro

Sorties

Port	Taille	Description
Store	1	Informe d'une opération d'écriture
Load	1	Informe d'une opération de lecture
PC_Hold	1	Retarde l'incrémentatation du compteur ordinal d'un cycle
Rmd	3	Registre cible / source pour l'opération de lecture / écriture
RAM_Addr	32	Adresse mémoire à laquelle effectuer les opérations

6.3.6 SP Address

6.3.6.1 Description

Le bloc SP Address traite les instructions de mise à jour du pointeur de pile (voir 9.1.4 Miscellaneous 16-bit instructions).

La sortie New_Stack_Pointer correspond à la nouvelle valeur du pointeur de pile qui sera enregistré dans le Stack Pointer. Elle est calculée en fonction de la valeur actuelle du Stack_Pointer et de l'opération en provenance de l'instruction (addition ou soustraction d'un immédiat).

Si l'entrée Enable est à 0, la sortie Write_Enable est forcée à 0.

6.3.6.2 Interface

Entrées

Port	Taille	Description
Instruction	16	Instruction de mise à jour du pointeur de pile à décoder
Enable	1	Active le composant. Si 0, les sortie sont forcées à 0
Stack_Pointer	32	Valeur courante du pointeur de pile

Sorties

Port	Taille	Description
Write_Enable	1	Le registre du pointeur de pile sera mis à jour avec la valeur New_Stack_Pointer
New_Stack_Pointer	32	Nouvelle valeur du pointeur de pile

6.3.7 Conditional

6.3.7.1 Description

Le bloc conditionnel traite les instructions de branchement (voir 9.1.5 *Branch*).

Il vérifie éventuellement la condition à l'aide des drapeaux du registre *Flags APSR*, selon le tableau 9.2 *Conditions* (p. 178). Si la condition est vérifiée, la sortie *Verified* passe à 1.

La sortie *Offset* correspond à l'offset qui sera ajouté au *Program Counter*, en provenance de l'instruction.

Si l'entrée *Enable* est à 0, la sortie *Verified* est forcée à 0.

6.3.7.2 Interface

Entrées

Port	Taille	Description
Instruction	16	Instruction du branchement conditionnel à décoder
Enable	1	Active le composant. Si 0, les sortie sont forcées à 0
N	1	Drapeau négatif
Z	1	Drapeau nul
C	1	Drapeau retenue
V	1	Drapeau dépassement de capacité

Sorties

Port	Taille	Description
Verified	1	La condition est vérifiée
Offset	8	Offset à appliquer au Program Counter si la condition est vérifiée

6.3.8 Program Counter

6.3.8.1 Description

Le compteur ordinal (*Program Counter*), aussi appelé pointeur d'instruction (*Instruction Pointer*), est un compteur 16 bits dont la valeur donne l'adresse de la prochaine instruction à exécuter. Il est déjà implémenté directement dans le contrôleur.

Il s'incrémente automatiquement à chaque front descendant d'horloge lorsque le signal `Count` est à l'état haut. Le signal `PC_Hold` sortant du sous-composant *Load/Store* retarde cette incrémentation d'un cycle afin de laisser le temps à la RAM de présenter la donnée sélectionnée sur sa sortie. En effet, étant synchrone, elle n'agit qu'au coup d'horloge suivant.

Lorsque le signal `Load` est à l'état haut, le compteur charge la valeur suivante à partir de son entrée au prochain coup d'horloge. Les instructions de branchement (voir 9.1.5 *Branch*) du sous-composant *Conditional* exploitent cette possibilité afin d'altérer le flot d'exécution du programme en sautant à une certaine adresse.

Remarque : afin d'être cohérent avec le comportement d'un vrai Cortex-M0 et donc rester compatible avec les programmes compilés par GCC ou LLVM, l'offset du branchement est incrémenté de 3.

6.3.8.2 Interface

Entrées

Port	Taille	Description
Data	16	Valeur du compteur à définir si Load est à l'état haut
Count	1	Active l'incrément automatique du compteur
UpDown	1	0 : décrémente, 1 : incrémente. Forcé à 1.
Load	1	Charge la valeur à partir de l'entrée Data
Clk	1	Horloge
Clear	1	Remise à Zero

Sorties

Port	Taille	Description
Output	16	Valeur courante du compteur ordinal
Carry	1	1 si le compteur atteint sa valeur maximale. Non utilisé ici.

6.3.9 Stack Pointer

6.3.9.1 Description

Le pointeur de pile (*Stack Pointer*) est un simple registre 32 bits indiquant l'adresse en mémoire du début de la pile. Il est déjà implémenté directement dans le contrôleur.

Son contenu est modifié par les instructions `ADD` et `SUB` (voir 9.1.4 *Miscellaneous 16-bit instructions*) du sous-composant *SP Address*.

Son contenu est utilisé par les instructions `LDR` et `STR` (voir 9.1.3 *Load/Store*) du sous-composant *Load/Store*.

Remarque : en général dans un programme, on commence par décrémente (avec `SUB`) le pointeur de pile de la quantité d'espace mémoire dont on aura besoin. Par la suite, pour les accès mémoire on utilisera `LDR` et `STR` avec un offset pour sélectionner le bon emplacement dans la pile. À la fin, on incrémente (avec `ADD`) le pointeur de pile pour revenir à l'état initial.

On peut dire que la pile grandit vers le bas.

6.3.9.2 Interface Entrées

Port	Taille	Description
Data	32	Nouvelle valeur du pointeur de pile
Enable	1	Active l'enregistrement de la valeur au prochain coup d'horloge
Clk	1	Horloge
Reset	1	Remise à Zero

Sorties

Port	Taille	Description
Output	32	Valeur courante du pointeur de pile

7 Chemin de données

7.1 Description

Il s'agit d'interfacer les différents composants CPU, RAM et ROM afin d'obtenir une machine fonctionnelle qui puisse exécuter un programme automatiquement.

Un bouton `Reset` contrôle l'entrée `Reset` du processeur afin de remettre à zéro son état et recommencer l'exécution du programme depuis le début.

Un bouton `Clock` contrôle l'entrée `Clk` du processeur et de la RAM afin de contrôler manuellement l'exécution de chacun des instructions. Pour une exécution automatique et pour le déploiement sur FPGA, on utilisera le composant `Horloge` de Logisim.

7.2 CPU

7.2.1 Description

Le processeur regroupe le contrôleur, le banc de registres et l'ALU.

À chaque coup d'horloge, l'instruction est décodée par le contrôleur, le calcul est effectué par l'ALU tandis que le banc de registres lui fournit les données et enregistre le résultat.

Lors d'une instruction `LDR`, le signal `Load` du contrôleur est activé, et la donnée en entrée du banc de registres doit être lue à partir de l'entrée `RAM_In` plutôt qu'à partir du résultat de l'ALU.

Lors d'une instruction `STR`, le signal `Store` du contrôleur est activé et est passé à la RAM. Les données en sortie `RAM_Out` sont fournies par la sortie `A0ut` du banc de registres.

Lors d'une instruction de la catégorie *Shift, add, sub, mov*, le signal `DP_Shift` du contrôleur est activé et l'immédiat `Imm5` en sortie du contrôleur est utilisé pour l'entrée `Shift` de l'ALU.

Lors d'une instruction de la catégorie *Data Processing*, le signal `DP_Shift` du contrôleur est désactivé et les 5 bits de poids faible de la sortie `A0ut` sont utilisés pour l'entrée `Shift` de l'ALU.

Lors d'une instruction MOV ou ADD / SUB / CMP avec immédiat, le signal `Imm32_Enable` du contrôleur est activé et la donnée à l'entrée A de l'ALU doit être lue à partir de la sortie `Imm32` du contrôleur plutôt qu'à partir de la sortie `AOut` du banc de registres.

Les sorties `Rm`, `Rn` et `Rd` du contrôleur se connectent respectivement aux entrées `RegA`, `RegB` et `RegDest` du banc de registres pour sélectionner les registres correspondant.

Les sorties `PC` et `RAM_Addr` se connectent respectivement aux sorties `ROM_Addr` et `RAM_Addr` de ce composant CPU pour sélectionner les adresses de la ROM et de la RAM.

Les sorties `Flags` et `SP` se connectent aux broches du même nom pour communiquer ces valeurs aux afficheurs de la façade.

7.2.2 Interface

7.2.2.1 Entrées

Port	Taille	Description
<code>RAM_In</code>	32	Données chargées à partir de la RAM
<code>ROM_In</code>	16	Instruction chargée à partir de la ROM
<code>Clk</code>	1	Horloge
<code>Reset</code>	1	Remise à Zero

7.2.2.2 Sorties

Port	Taille	Description
<code>ROM_Addr</code>	16	Adresse de la prochaine instruction
<code>RAM_Addr</code>	32	Adresse de la donnée à lire/écrire en mémoire
<code>RAM_Out</code>	32	Donnée à écrire en mémoire
<code>Store</code>	1	Indique à la RAM de sauvegarder la donnée <code>RAM_Out</code> à l'adresse <code>RAM_Addr</code>
<code>R0-R7</code>	8×32	Sortie des registres utilisées à des fins de débogage et visualisation du comportement
<code>Flags</code>	4	Valeur des drapeaux, issue du contrôleur
<code>SP</code>	32	Valeur de SP, issue du contrôleur

7.3 RAM

7.3.1 Description

La RAM est la mémoire de données dans laquelle le programme vient stocker le contenu d'un registre ou lire une donnée pour remplir un registre. Synchronisée, elle ne lit ou n'enregistre les données qu'au coup d'horloge suivant.

Elle est adressée sur 8 bits et contient des données de 32 bits.

Le signal `Load` est défini à 1 de manière à toujours obtenir la donnée à l'adresse `Address` sur la sortie `Data`.

Lorsque le signal `Store` est activé, les données présentes sur l'entrée `Input` sont enregistrées dans la RAM à l'adresse `Address`.

Sous Logisim, le paramètre `Databus implementation` doit être défini à `Separate databus for read and write` et le paramètre `Trigger` à `Rising Edge`.

Remarque : le jeu d'instructions ARMv7-M adresse la RAM octet par octet (1 adresse = 8 bits de données) tandis que sous Logisim, le composant RAM est adressé mot par mot (1 adresse = 32 bits de données). Cela permet de simplifier la gestion de la RAM puisqu'il est possible de charger des données directement dans les registres 32 bits du CPU.

7.3.2 Interface

7.3.2.1 Entrées

Port	Taille	Description
Address	8	Adresse à laquelle lire/écrire les données
Load	1	Lire les données au prochain coup d'horloge
Store	1	Enregistrer les données au prochain coup d'horloge
Clock	1	Horloge
Input	32	Données à écrire

7.3.2.2 Sorties

Port	Taille	Description
Data	32	Données lues

7.4 ROM

7.4.1 Description

La ROM est la mémoire d'instruction à partir de laquelle les instructions du programme sont lues. Elle est accessible en lecture uniquement et est asynchrone.

Elle est adressée sur 16 bits et contient des données de 16 bits (largeur d'une instruction Thumb).

Le programme, assemblé par l'assembleur, devra être chargé dans cette ROM

7.4.2 Interface

7.4.2.1 Entrées

Port	Taille	Description
Address	16	Adresse de l'instruction à charger

7.4.2.2 Sorties

Port	Taille	Description
Data	16	Instruction lue

8 Assembleur

8.1 Introduction

Le code binaire à exécuter est obtenu par l'assemblage d'instructions issues du jeu d'instructions ARMv7 (contre un jeu ARMv6 dans le Cortex-M0 réel).

Le rôle de l'assembleur est de traduire un programme écrit en langage assembleur dans une représentation que le processeur saura interpréter.

Ici, le langage assembleur sera l'UAL (*Unified Assembler Language*) de ARM, restreint aux instructions ARMv7 implémentées. La représentation des instructions en sortie correspond au codage Thumb des instructions, c'est à dire uniquement sur 16-bits (voir 9 *Jeu d'instructions (Instruction Set Architecture)*).

Le format de sortie aura la particularité d'être un fichier texte lisible par Logisim. Les instructions Thumb devront donc être codées en hexadécimal dans un format décrit ci-après.

8.2 Syntaxe

Syntaxe UAL :

S : māj des drapeaux

<c> : condition

R_m : registre opérande 1

R_n : registre opérande B

R_d : registre destination

<immN> : immédiat sur N bits

SP : registre de pointeur de pile en mémoire

opcode : code de l'instruction, peut occuper jusqu'à la taille indiquée

[] : argument optionnel

Exemple :

Nous allons partir d'un exemple très simple : 3 variables sont déclarées sur la pile, *a* et *b* ont une valeur qui leur est propre et nous stockons dans *c* le résultat de l'addition *a + b*.

Code C :

```
int main() {
    int a, b, c;
    a = 0;
    b = 1;
    c = a + b;
}
```

Code assembleur pour le Cortex-M0 avec un pseudo-code C équivalent, et l'encodage correspondant :

```
b083  sub    sp, #12
2000  movs   r0, #0
9002  str    r0, [sp, #8]
2101  movs   r1, #1
9101  str    r1, [sp, #4]
9902  ldr    r1, [sp, #8]
9a01  ldr    r2, [sp, #4]
1889  adds   r1, r1, r2
9100  str    r1, [sp]
b003  add    sp, #12
```

```
sp      = sp - 3;
r0      = 0;
sp[2]   = r0;
r1      = 1;
sp[1]   = r1;
r1      = sp[2];
r2      = sp[1];
r1      = r1 + r2;
sp[0]   = r1;
sp      = sp + 3;
```

8.3 Syntaxe Logisim

Voici un exemple de fichier lisible par Logisim pour remplir le contenu de la ROM obtenu par assemblage du code assembleur ci-dessus :

```
v2.0 raw
b083 2000 9002 2101 9101 9902 9a01 1889 9100 b003
```

On observe donc un entête `v2.0 raw`, toujours présent sur la première ligne.

Sur les lignes suivantes, les instructions sont disposées par groupes de 4 caractères hexadécimaux séparés par des espaces, ce qui représente 16 bits. On a donc une instruction par groupe. Les retours à la ligne sont optionnels.

9 Jeu d'instructions (Instruction Set Architecture)

Toutes les informations présentes dans cette section proviennent directement du manuel de référence de l'architecture ARMv7-M (*ARM v7-M Architecture Reference Manual*). Elles ont été traduites et réorganisées pour en faciliter la lecture. En cas de doute, ou pour en savoir plus, les pages du manuel sont indiquées entre parenthèses.

9.1 Instructions à implémenter

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode															

9.1.1 Shift, add, sub, mov

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	opcode													

9.1.1.1 LSL (immediate) : Logical Shift Left (p. 282)

Description :

Décale le contenu du registre R_m vers la gauche d'un nombre de bits donné par l'immédiate $imm5$, écrit le résultat dans le registre R_d .

Des zéros sont insérés à droite.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = R_m < 0 - shift >$ avec $shift$ le nombre de décalage. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

LSLS <Rd>, <Rm>, # <imm5>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Rm			Rd		

9.1.1.2 LSR (immediate) : Logical Shift Right (p. 284)**Description :**

Décale le contenu du registre R_m vers la droite d'un nombre de bits donné par l'immédiate $imm5$, écrit le résultat dans le registre R_d .

Des zéros sont insérés à gauche.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = R_m<shift - 1>$ avec shift le nombre de décalage. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

LSRS <Rd>, <Rm>, # <imm5>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	imm5					Rm			Rd		

9.1.1.3 ASR (immediate) : Arithmetic Shift Right (p. 203)**Description :**

Décale le contenu du registre R_m vers la droite d'un nombre de bits donné par l'immédiate $imm5$, écrit le résultat dans le registre R_d .

Le bit de signe de R_m est ré-inséré à gauche.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = R_m<shift - 1>$ avec shift le nombre de décalage. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

ASRS <Rd>, <Rm>, # <imm5>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm5					Rm			Rd		

9.1.1.4 ADD (register) : Add register (p. 192)**Description :**

Ajoute le contenu du registre R_n au contenu du registre R_m , écrit le résultat dans le registre R_d .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

ADDS <Rd>, <Rn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		

9.1.1.5 SUB (register) : Subtract register (p. 404)

Description : Soustrait le contenu du registre R_m au contenu du registre R_n , écrit le résultat dans le registre R_d .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

SUBS <Rd>, <Rn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	Rm			Rn			Rd		

9.1.1.6 ADD (immediate) : Add 3-bit immediate (p. 190)

Description :

Ajoute l'immédiat $Imm3$ au contenu du registre R_n , écrit le résultat dans le registre R_d .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

ADDS <Rd>, <Rn>, <#imm3>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	Imm3			Rn			Rd		

9.1.1.7 SUB (immediate) : Subtract 3-bit immediate (p. 402)

Description :

Soustrait l'immédiate `Imm3` au contenu du registre `Rn`, écrit le résultat dans le registre `Rd`.

Les drapeaux suivants sont mis à jour :

`N = 1` si résultat < 0 , `N = 0` sinon.

`Z = 1` si résultat $= 0$, `Z = 0` sinon.

`C = 1` en cas de dépassement de capacité lors d'une opération non signée.

`V = 1` en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

`SUBS <Rd>, <Rn>, # <imm3>`

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	Imm3			Rn			Rd		

9.1.1.8 MOV (immediate) : Move (p. 291)

Description :

Écrit l'immédiate `imm8` dans le registre `Rd`.

Les drapeaux suivants sont mis à jour :

`N = 1` si résultat < 0 , `N = 0` sinon.

`Z = 1` si résultat $= 0$, `Z = 0` sinon.

Assembleur : T1

`MOVS <Rd>, # <imm8>`

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

9.1.1.9 CMP (immediate) : Compare (p. 223)

Description :

Soustrait l'immédiate `imm8` au contenu du registre `Rn`, le résultat n'est pas écrit.

Les drapeaux suivants sont mis à jour :

`N = 1` si résultat < 0 , `N = 0` sinon.

`Z = 1` si résultat $= 0$, `Z = 0` sinon.

`C = 1` en cas de dépassement de capacité lors d'une opération non signée.

`V = 1` en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

`CMP <Rd>, # <imm8>`

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rd			imm8							

9.1.1.10 ADD (immediate) : Add 8-bit immediate (p. 190)

Description :

Ajoute l'immédiat `imm8` au contenu du registre `Rdn`, écrit le résultat dans celui-ci.

Les drapeaux suivants sont mis à jour :

`N = 1` si `résultat < 0`, `N = 0` sinon.

`Z = 1` si `résultat = 0`, `Z = 0` sinon.

`C = 1` en cas de dépassement de capacité lors d'une opération non signée.

`V = 1` en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T2

`ADDS <Rdn>, [<Rdn> ,] #<imm8>`

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rd			imm8							

9.1.1.11 SUB (immediate) : Subtract 8-bit immediate (p. 402)

Description :

Soustrait l'immédiat `imm8` au contenu du registre `Rdn`, écrit le résultat dans celui-ci.

Les drapeaux suivants sont mis à jour :

`N = 1` si `résultat < 0`, `N = 0` sinon.

`Z = 1` si `résultat = 0`, `Z = 0` sinon.

`C = 1` en cas de dépassement de capacité lors d'une opération non signée.

`V = 1` en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T2

`SUBS <Rdn>, [<Rdn> ,] #<imm8>`

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rd			imm8							

9.1.2 Data processing

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	opcode									

9.1.2.1 AND (register) : Bitwise AND (p. 201)

Description :

Effectue un ET binaire entre le contenu du registre `Rdn` et le contenu du registre `Rm`, écrit le résultat dans le registre `Rdn`.

Les drapeaux suivants sont mis à jour :

`N = 1` si `résultat < 0`, `N = 0` sinon.

$Z = 1$ si résultat = 0, $Z = 0$ sinon.
 $C = 0$

Assembleur : T1

ANDS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

9.1.2.2 EOR (register) : Exclusive OR (p. 233)

Description :

Effectue un OU exclusif binaire entre le contenu du registre R_{dn} et le contenu du registre R_m , écrit le résultat dans le registre R_{dn} .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0, $N = 0$ sinon.

$Z = 1$ si résultat = 0, $Z = 0$ sinon.

$C = 0$

Assembleur : T1

EORS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		

9.1.2.3 LSL (register) : Logical Shift Left (p. 283)

Description :

Décale le contenu du registre R_{dn} vers la gauche d'un nombre de bits donné par l'octet inférieur du registre R_m , écrit le résultat dans le registre R_{dn} .

Des zéros sont insérés à droite.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0, $N = 0$ sinon.

$Z = 1$ si résultat = 0, $Z = 0$ sinon.

$C = R_n < 0 - \text{shift} >$ avec shift le nombre de décalage. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

LSLS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rm			Rdn		

9.1.2.4 LSR (register) : Logical Shift Right (p. 285)

Description :

Décale le contenu du registre R_{dn} vers la droite d'un nombre de bits donné par l'octet inférieur du registre

R_m , écrit le résultat dans le registre R_{dn} .

Des zéros sont insérés à gauche.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = R_n<shift - 1>$ avec shift le nombre de décalage. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

LSRS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rm			Rdn		

9.1.2.5 ASR (register) : Arithmetic Shift Right (p. 204)

Description :

Décale le contenu du registre R_{dn} vers la droite d'un nombre de bits donné par l'octet inférieur du registre R_m , écrit le résultat dans le registre R_{dn} .

Le bit de signe de R_{dn} est ré-inséré à gauche.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = R_n<shift - 1>$ avec shift le nombre de décalage. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

ASRS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rm			Rdn		

9.1.2.6 ADC (register) : Add with Carry (p. 188)

Description :

Ajoute le contenu du registre R_m et le drapeau de retenu au contenu du registre R_{dn} , écrit le résultat dans le registre R_{dn} .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

ADCS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

9.1.2.7 SBC (register) : Subtract with Carry (p. 347)**Description :**

Soustrait le contenu du registre R_m et le complément du drapeau de retenu au contenu du registre R_{dn} , écrit le résultat dans le registre R_{dn} .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

SBCS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		

9.1.2.8 ROR (register) : Rotate Right (p. 339)**Description :**

Pivote le contenu du registre R_{dn} vers la droite d'un nombre de bits donné par l'octet inférieur du registre R_m , écrit le résultat dans le registre R_{dn} .

Les bits de R_{dn} sortant à droite sont ré-insérés à gauche.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = \text{résultat} \langle 31 \rangle$. Autrement dit, C est égal au dernier bit sortant.

Assembleur : T1

RORS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rm			Rdn		

9.1.2.9 TST (register) : Set flags on bitwise AND (p. 420)**Description :**

Effectue un ET logique entre le contenu du registre R_n et le contenu du registre R_m , le résultat n'est pas écrit.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 0$

Assembleur : T1

TST <Rn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	0	Rm			Rn		

9.1.2.10 RSB (immediate) : Reverse Subtract from 0 (p. 341)**Description :**

Soustrait le contenu du registre R_n à l'immédiat 0, écrit le résultat dans le registre R_d .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 0$.

$V = 0$.

Assembleur : T1

RSBS <Rd>, <Rn>, #0

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	1	Rn			Rd		

9.1.2.11 CMP (register) : Compare Registers (p. 224)**Description :**

Soustrait le contenu du registre R_m au contenu du registre R_n , le résultat n'est pas écrit.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

CMP <Rn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

9.1.2.12 CMN (register) : Compare Negative (p. 222)**Description :**

Ajoute le contenu du registre R_m au contenu du registre R_n , le résultat n'est pas écrit.

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 1$ en cas de dépassement de capacité lors d'une opération non signée.

$V = 1$ en cas de dépassement de capacité lors d'une opération signée.

Assembleur : T1

CMN <Rn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Rm			Rn		

9.1.2.13 ORR (register) : Logical OR (p. 310)**Description :**

Effectue un OU binaire entre le contenu du registre R_{dn} et le contenu du registre R_m , écrit le résultat dans le registre R_{dn} .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 0$.

Assembleur : T1

ORRS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

9.1.2.14 MUL : Multiply Two Registers (p. 302)**Description :**

Multiplie le contenu du registre R_n avec le contenu du registre R_{dm} , écrit les 32 bits de poids faible du résultat dans le registre R_{dm} .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

Assembleur : T1

MULS <Rdm>, <Rn>, <Rdm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	1	Rn			Rdm		

9.1.2.15 BIC (register) : Bit Clear (p. 210)**Description :**

Effectue un ET binaire entre le contenu du registre R_{dn} et le complément du contenu du registre R_m , écrit le résultat dans le registre R_{dn} .

Les drapeaux suivants sont mis à jour :

$N = 1$ si résultat < 0 , $N = 0$ sinon.

$Z = 1$ si résultat $= 0$, $Z = 0$ sinon.

$C = 0$.

Assembleur : T1

BICS <Rdn>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		

9.1.2.16 MVN (register) : Bitwise NOT (p. 304)**Description :**

Effectue un NON binaire sur le contenu du registre Rm, écrit le résultat dans le registre Rd.

Les drapeaux suivants sont mis à jour :

N = 1 si résultat < 0, N = 0 sinon.

Z = 1 si résultat = 0, Z = 0 sinon.

C = 0.

Assembleur : T1

MVNS <Rd>, <Rm>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

9.1.3 Load/Store**Binaire :**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	opcode											

9.1.3.1 STR (immediate) : Store Register (p. 386)**Description :**

Écrit un mot de 32 bits contenu dans le registre Rt à l'adresse mémoire spécifiée.

L'adresse mémoire est calculée à partir du contenu du registre SP plus l'immédiat imm8.

L'immédiat imm8 correspond à l'offset divisé par 4 par l'assembleur.

Assembleur : T2

STR <Rt>, [SP, #<offset>]

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

9.1.3.2 LDR (immediate) : Load Register (p. 246)**Description :**

Charge un mot de 32 bits contenu à l'adresse mémoire spécifiée, écrit le résultat dans le registre Rt. L'adresse mémoire est calculée à partir du contenu du registre SP plus l'immédiat `imm8`. L'immédiat `imm8` correspond à l'offset divisé par 4 par l'assembleur.

Assembleur : T2

LDR <Rt> ,[SP{, #<offset>}]

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rt			imm8							

9.1.4 Miscellaneous 16-bit instructions

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	opcode											

9.1.4.1 ADD (SP plus immediate) : Add Immediate to SP (p. 194)

Description :

Ajoute l'immédiat `imm7` à la valeur du registre SP, écrit le résultat dans le registre SP. L'immédiat `imm7` correspond à l'offset divisé par 4 par l'assembleur. Les drapeaux ne sont pas mis à jour.

Assembleur : T2

ADD [SP,] SP, #<offset>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

9.1.4.2 SUB (SP minus immediate) : Subtract Immediate from SP (p. 402)

Description :

Soustrait l'immédiat `imm7` à la valeur du registre SP, écrit le résultat dans le registre SP. L'immédiat `imm7` correspond à l'offset divisé par 4 par l'assembleur. Les drapeaux ne sont pas mis à jour.

Assembleur : T1

SUB [SP,] SP, #<offset>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

9.1.5 Branch

Dans le code assembleur, les instructions de branchement ont pour opérande des étiquettes, pointant vers d'autres endroits dans le code. Lors de l'encodage par le compilateur, la valeur de l'immédiat (imm8 ou imm11) est égale à $N_{cible} - N_{source} - 3$, où N représente le numéro de l'instruction dans le programme (indexée à 0). Cette valeur peut être négative (si on veut sauter à une instruction présente plus tôt dans le programme) et sera donc encodée comme valeur signée en complément à 2.

9.1.5.1 B : Conditional Branch (p. 205)

Description :

Continue l'exécution à partir de l'étiquette `label` si la condition `<c>` est vérifiée.

Assembleur : T1

B<c> <label>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

9.1.5.2 B : Unconditional Branch (p. 205)

Description :

Continue l'exécution à partir de l'étiquette `label`.

Assembleur : T2

B <label>

Binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

9.2 Conditions (p. 178)

code	symbole	signification	drapeaux
0000	EQ	égalité	Z == 1
0001	NE	différence	Z == 0
0010	CS ou HS	retenue	C == 1
0011	CC ou LO	pas de retenue	C == 0
0100	MI	négatif	N == 1
0101	PL	positif ou nul	N == 0
0110	VS	dépassement de capacité	V == 1
0111	VC	pas de dépassement de capacité	V == 0
1000	HI	supérieur (non signé)	C == 1 et Z == 0
1001	LS	inférieur ou égal (non signé)	C == 0 ou Z == 1
1010	GE	supérieur ou égal (signé)	N == V
1011	LT	inférieur (signé)	N != V
1100	GT	supérieur (signé)	Z == 0 et N == V
1101	LE	inférieur ou égal (signé)	Z == 1 ou N != V
1110	AL	toujours vrai	

Remarque : Par complétude, vous pouvez considérer que la condition 1111 correspond à "toujours faux". En réalité, cette condition n'est jamais utilisée, mais lors de la réalisation de circuits il est déconseillé de laisser des entrées non reliées.

9.3 Drapeaux (p. 31)

- N : résultat négatif, égal au bit de poids fort du résultat
- Z : résultat nul, égal à 1 si le résultat est 0
- C : retenue
- V : dépassement de capacité

10 Pour aller plus loin

10.1 Compilation de code C

Avec le jeu d'instructions de ce processeur, il est possible d'exécuter du code C compilé par Clang. Il doit cependant rester relativement simple avec une structure bien précise.

En particulier, on évitera :

- Les appels de fonctions (LR, PUSH, POP non implémentés)
- Les variables globales et `static` (adressage uniquement sur la pile)
- Les adressages indirects (donc l'utilisation de tableaux ou de chaînes)

On s'assurera donc :

- D'écrire tout le code dans la fonction `void run()`
- De déclarer toutes les variables dans le corps de cette dernière

Voici le code de démarrage pour un programme C dans le cadre du projet :

```
#include <parm.h>

void run()
{
    BEGIN();
    // code ici
    END();
}
```

On utilise ici une fonction s'appelant `run` et non pas `main` car les compilateurs C émettent un warning lorsque la fonction `main()` ne se termine pas (par exemple, si elle contient une boucle infinie). Or, c'est ici un cas souhaité, et appeler la fonction autrement évite ce warning.

Les appels `BEGIN` et `END` sont des macros servant à allouer l'espace mémoire nécessaire pour gérer les périphériques d'entrée/sortie. Il faut systématiquement les écrire dans vos programmes en début et en fin de fonction, sans quoi vous ne pourrez pas utiliser la RAM et ne pourrez donc pas utiliser de variables.

La commande à utiliser est la suivante (écrivez la, si vous la copiez depuis le PDF les caractères ne seront pas bons) :

```
clang -S -target arm-none-eabi -mcpu=cortex-m0 -O0 -mthumb -nostdlib\
-I./include main.c
```

avec `main.c` le fichier source C. Veillez à être dans le dossier `code_c` en lançant cette commande.

Cela créera un fichier de sortie `main.s` contenant les instructions assembleur, entourées de directives spéciales qui ne nous intéressent pas ici.

Remarque : depuis la version 9 de Clang, l'adressage mémoire est réalisé différemment en utilisant des instructions `add <Rd>,SP,#<imm8>` et `ldr <Rt>,<Rn>,#<imm5>` qui ne sont pas prises en charge par l'implémentation. Il faut donc s'assurer d'utiliser une version 4 à 8 de Clang (paquet `clang-8` sous Ubuntu).

Le code assembleur devra être passé à l'assembleur écrit dans le cadre du projet pour générer le fichier lisible par Logisim et pouvoir l'importer dans la ROM.

10.2 Périphériques d'entrée/sortie mappés en mémoire

Actuellement, notre processeur ne peut interagir avec l'utilisateur que de façon très primitive :

- En écrivant une valeur dans un registre
- En lisant ou écrivant une valeur dans la RAM

Dans les deux cas, cela limite à des valeurs numériques, et cela implique de prévoir à l'avance, par convention, dans quel registre ou case mémoire lire ou écrire.

Dans la réalité, vous lisez ce document sur un ordinateur avec lequel vous interagissez via des périphériques d'entrée/sortie : un clavier, une souris, un écran, etc.

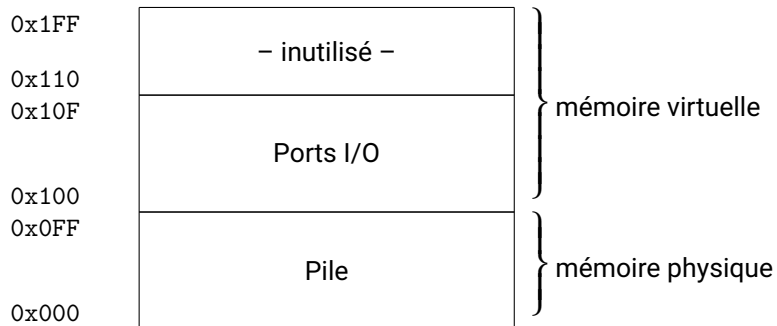
L'espace mémoire de l'ordinateur contient des sections correspondant non pas à une vraie mémoire, mais à des registres partagés avec des périphériques. Les instructions de gestion de RAM sont utilisés pour lire ou écrire dans ces registres.

Le processeur PARM possède une RAM de 256 mots de 32 bits et un bus d'adresse sur 9 bits. Son espace



FIG. 8 : DIP switch

mémoire est disposé de la manière suivante :



Les 256 premiers mots (adresses [0; 256[) correspondent à la mémoire RAM physique, tandis que les 256 suivants sont "virtuels", ils sont gérés par le contrôleur MMIO présent en bas du circuit, qui se charge de déléguer aux différents périphériques. Ici, 16 emplacements sont alloués, donc les adresses [256; 271[correspondent à des périphériques et les adresses [271; 512[ne sont "branchées" à rien.

La documentation des différents périphériques est disponible dans le fichier `include/parm.h`.

10.2.1 Exemple : DIP switch

Un composant d'entrée très courant est le DIP switch. Il s'agit d'un petit bloc comportant une série de n interrupteurs, encodant un nombre sur n bits. Voici à quoi ça ressemble :

Notre processeur est relié à trois switches de ce genre, sur les entrées 3, 4 et 5 (adresses 259, 260 et 261). Du point de vue d'un programme, lire depuis les DIP switches revient à lire depuis une variable qui serait stockée à ces emplacements, c'est entièrement transparent.

10.2.2 Exemple : Terminal

La manière la plus simple d'afficher du texte est via un terminal. Nommé d'après les terminaux physiques qui équipaient les mini-ordinateurs des années 1970, son fonctionnement est simple : quand il reçoit un caractère, il l'affiche. Certains caractères ont un effet particulier :

- `\n` (newline) effectue un retour à la ligne
- `\b` (backspace) efface le dernier caractère saisi
- `\f` (form-feed) efface l'écran

Ainsi, ça évite d'avoir des entrées spécifiques pour toutes ces actions.

En pratique, le terminal de Logisim possède plusieurs entrées : l'horloge, le caractère, la mise à jour et la réinitialisation. Cette dernière n'est utilisée que quand on appuie sur le bouton "Reset" du processeur.

À chaque front montant d'horloge, si la broche "mise à jour" est haute, alors le caractère est affiché.

Une seule sortie est fournie aux programmes : caractère. Sa valeur est reliée à l'entrée "caractère" de l'écran, et dès qu'une écriture est faite dans cette sortie, l'entrée "mise à jour" est passée à 1.