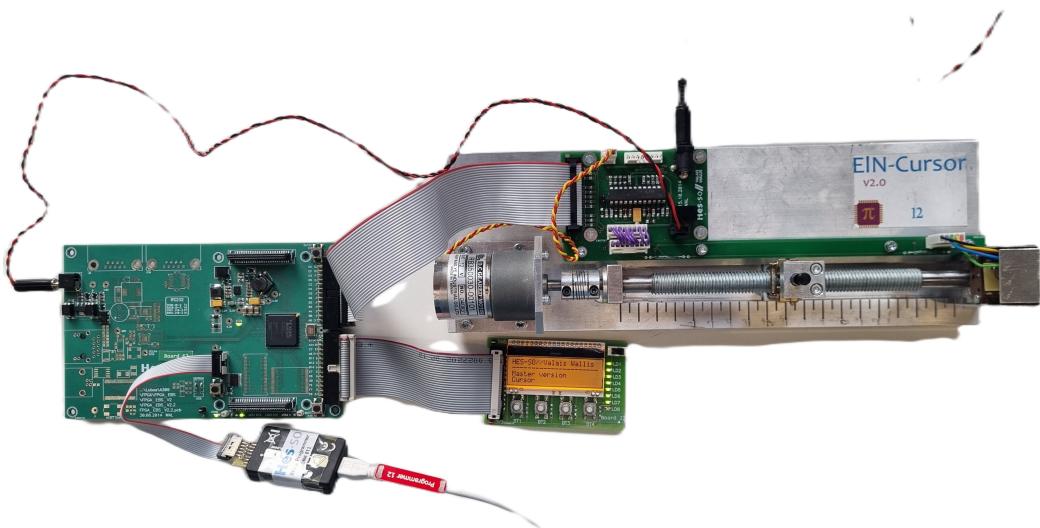


# DiD Labs projects

Project cursor

*Création de la logique de base*



**Hes·so** // VALAIS  
WALLIS

 School of Engineering

Institut: Haute École d'Ingénierie Valais

Auteurs: Astori Mattia, Crestin Alex

Date: 18 Janvier 2025

Cours: Conception numérique, filière SYND

Version: 2.0

## Résumé

Ce projet détaille la conception en VHDL de la logique de contrôle pour le système projet Cursor.

L'architecture repose sur l'implémentation de compteurs et d'additionneurs permettant de gérer précisément le positionnement et la génération des signaux PWM.

Le système coordonne également une interface LCD et des machines à états (FSM) pour assurer le pilotage des moteurs et le retour visuel.

Ce rapport documente la hiérarchie numérique, la synthèse des composants, ainsi que les erreurs courantes rencontrées lors du développement, offrant ainsi une analyse complète des défis techniques liés au fonctionnement global du dispositif.

## Contacts

### Mattia Astori

E-Mail: [mattia.astori@hes-so.ch](mailto:mattia.astori@hes-so.ch)

Tel.: [+41 79 790 81 64](tel:+41797908164)

### Alex Crestin

E-Mail: [alex.crestin@hes-so.ch](mailto:alex.crestin@hes-so.ch)

Tel.: [+41 79 466 09 60](tel:+41794660960)

### HEI Sion

Réponsable du cours: Silvan Zahno

E-Mail: [silvan.zahno@hes-so.ch](mailto:silvan.zahno@hes-so.ch)

# Sommaire

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Objectif . . . . .	3
1.2	Outils . . . . .	3
<b>2</b>	<b>Spécifications</b>	<b>4</b>
2.1	Fonctions et fonctionnement . . . . .	4
2.2	Signal et bus . . . . .	4
2.2.1	Types de signaux . . . . .	4
2.2.2	Types de bus . . . . .	4
2.3	Codification de la position . . . . .	5
2.4	Dynamique du déplacement . . . . .	5
2.4.1	Delta PWM et dimension du bit-shift . . . . .	7
2.4.2	Vitesse minimale . . . . .	7
<b>3</b>	<b>Composants</b>	<b>8</b>
3.1	Moteur et control du moteur (pont en H) . . . . .	8
3.1.1	Explication fréquence 100kHz . . . . .	8
3.2	Encoder . . . . .	9
3.2.1	Fréquence de clock . . . . .	9
<b>4</b>	<b>Aperçu du projet</b>	<b>10</b>
4.1	Concept de fonctionnement . . . . .	10
4.2	Compteurs itératives . . . . .	11
4.3	Additionneur à propagation de report . . . . .	12
<b>5</b>	<b>Description des blocs</b>	<b>13</b>
5.1	Button manager . . . . .	13
5.2	Main controller . . . . .	13
5.2.1	Rotation Sous . . . . .	15
5.2.2	Bloc ButtonControlFSM . . . . .	17
5.3	Motor control . . . . .	19
5.3.1	Change detector . . . . .	21
5.3.2	Relatif counter . . . . .	21
5.3.3	Compteur de position . . . . .	22
5.3.4	Position de ralentissement (AddSous) . . . . .	23
5.3.5	Motor FSM . . . . .	24
5.3.6	PWM Generator . . . . .	25
5.4	Deux registres . . . . .	28
<b>6</b>	<b>Fonctions supplémentaires</b>	<b>29</b>
6.1	LCD . . . . .	29
6.1.1	Entrées . . . . .	29
6.1.2	Sorties . . . . .	29
6.2	LEDs . . . . .	31
6.3	Bouton 4 . . . . .	33
6.3.1	Countdown counter . . . . .	35
<b>7</b>	<b>Simulation et erreurs</b>	<b>36</b>
7.1	Erreurs . . . . .	36
7.1.1	Glitches . . . . .	36
7.1.2	PWM . . . . .	36
7.1.3	Latches . . . . .	36
<b>8</b>	<b>Conclusion</b>	<b>37</b>
8.1	Proposition d'améliorations . . . . .	37
8.1.1	Rotation en fonction de l'encodeur . . . . .	37
8.1.2	Vitesse minimale en fonction de l'encodeur . . . . .	37

8.2 Commentaire personal . . . . .	37
<b>Références</b>	<b>39</b>
<b>Glossaire</b>	<b>39</b>
<b>Acronymes</b>	<b>39</b>
<b>Liste des figures</b>	<b>40</b>
<b>Liste des tableaux</b>	<b>40</b>
<b>Liste des annexes</b>	<b>41</b>

# 1 Introduction

Le projet "curseur" est le projet semestriel pour les étudiants de Systèmes Industrielles dans le cours de conception numérique.

Le projet s'est développé en plusieurs phases, la réalisation de la logique du bloc principal<sup>1</sup>, la création des sous blocs, la simulation indépendante de chaque sous blocs, la simulation du projet dans son ensemble, l'amélioration et l'implémentation d'autres fonctions et enfin l'écriture du rapport qui à été fait en parallel au reste.

## 1.1 Objectif

Le but de ce projet est de créer la logique utilisé pour programmer un **FPGA**, un circuit logique intégré programmable sur le terrain[2]. Le **FPGA** doit contrôler le déplacement d'un "chariot" sur une vise sans fin en respectant des positions, des vitesses et des capteurs analogiques.

Dans la creation de la logique un système avec plusieurs niveaux doit être utilisé, c.a.d des créer des blocs qui eux mêmes contiennent des sous blocs, etc...

## 1.2 Outils

Le software utilisé pour la programmation est **VHDL** Designer®, en couple avec ModelSim® pour la simulation.

One-note a aussi été utilisé pour créer et partager les dessins des blocs.

---

<sup>1</sup>"Top level logic"

## 2 Spécifications

La partie de hardware est déjà prête à utiliser, le composants sont décrit dans le chapitre 3.

### 2.1 Fonctions et fonctionnement

Les fonctions selon le cahier des charges [4] sont les suivantes.

#### 1. Objectif Principal:

L'objectif est de concevoir un système de contrôle pour un moteur à courant continu afin de déplacer précisément un chariot le long d'une vis vers des positions prédéfinies.

#### 2. Fonctions de Base Obligatoires:

- Restart: Le curseur doit se déplacer vers la position initiale (position de départ), identifiée par un relais Reed (sensor1).
- Position 1: Le chariot doit atteindre la position située à 8 cm.
- Position 2: Le chariot doit atteindre la position située à 12 cm.

#### 3. Profil de Vitesse (Rampes):

Le déplacement ne se fait pas à vitesse constante. Pour les positions 1 et 2, le cycle doit être le suivant :

- Accélération régulière jusqu'à la pleine vitesse.
- Avance à pleine vitesse.
- Décélération régulière pour s'arrêter précisément sur la position.

Note technique importante : Les rampes d'accélération et de décélération doivent dépendre de la position et non du temps 2.4. La distance de freinage/accélération doit être d'environ 1 cm.

#### 4. Matériel et Signaux:

- Contrôle Moteur: Utilisation d'un pont en H (L6207). La vitesse est modulée par PWM avec une fréquence maximale de 100 kHz 3.1.1.
- Capteurs de fin de course: Deux relais Reed détectent les limites physiques (gauche/droite) du rail.
- Cerveau du système: Une carte FPGA (Spartan-3) cadencée à 66 MHz (pour la carte EBS2).

### 2.2 Signal et bus

Un signal en conception numérique est une information logique qui représente l'état d'un circuit, généralement 0 ou 1. Tandis qu'un bus est un ensemble de signaux utilisé pour transporter des données sur plusieurs bits en parallèle [5].

#### 2.2.1 Types de signaux

En VHDL, le "std\_ulogic" est un type de signal logique à plusieurs valeurs, permettant de décrire plus fidèlement le comportement réel des circuits.

Tous les signaux dans ce projet sont des "std\_ulogic" car le signal std\_ulogic est non résolu : un seul bloc peut le piloter. Tout conflit est détecté, ce qui évite les erreurs de conception.

#### 2.2.2 Types de bus

En VHDL, un bus "std\_ulogic\_vector" est un vecteur de signaux logiques à plusieurs valeurs, utilisé pour représenter fidèlement les états matériels (0, 1, indéterminé, etc.). Il est non résolu, donc chaque bit ne peut être piloté que par une seule source, ce qui permet de détecter les conflits. Ce type de bus est utilisé pour des transports de données interne.

Un bus "unsigned" représente un nombre entier non signé. Il est destiné aux opérations arithmétiques (addition, comparaison...) et s'utilise lorsque le bus transporte une valeur numérique plutôt que de simples niveaux logiques.

### 2.3 Codification de la position

La codification de la position du curseur est sur 18 bits, le nombre de bits a été choisi avec la formule suivante:

$$N_{bits} = \log_2\left(\frac{190}{\frac{1.75}{500*4}}\right) \approx 17.72 \rightarrow 18 \quad (1)$$

La position est représentée par un compteur itératif qui compte quand le chariot avance et décompte quand le chariot recule.

Vu que les signaux A et B de l'encodeur changent 500 fois chaque tour et les deux sont décalé d'un demi-signal, l'état de l'ensemble de A et B change 2000 ( $500 * 4$ ) fois par tour. La vis a un pas de 1.75mm le calcul  $\frac{1.75}{2000}$  donne donc la précision maximal ( $0.000875mm$ ) qu'il est possible d'atteindre avec l'encodeur et la vis utilisées.

La longueur maximale du déplacement est d'env. 19cm, en divisant ça par la précision il est possible de trouver le nombre de steps que le compteur doit avoir. Le logarithme en base 2 retourne simplement le nombre de bit (à arrondir vers le haut) du compteur.

Le sens de la rotation du moteur est obtenu en soustrayant la position-but à la position réelle du chariot. Si le nombre est négatif le chariot doit reculer, sinon avancer.

### 2.4 Dynamique du déplacement

Le curseur doit se déplacer comme dans l'image 1, avec une accélération au début et une décélération à la fin.

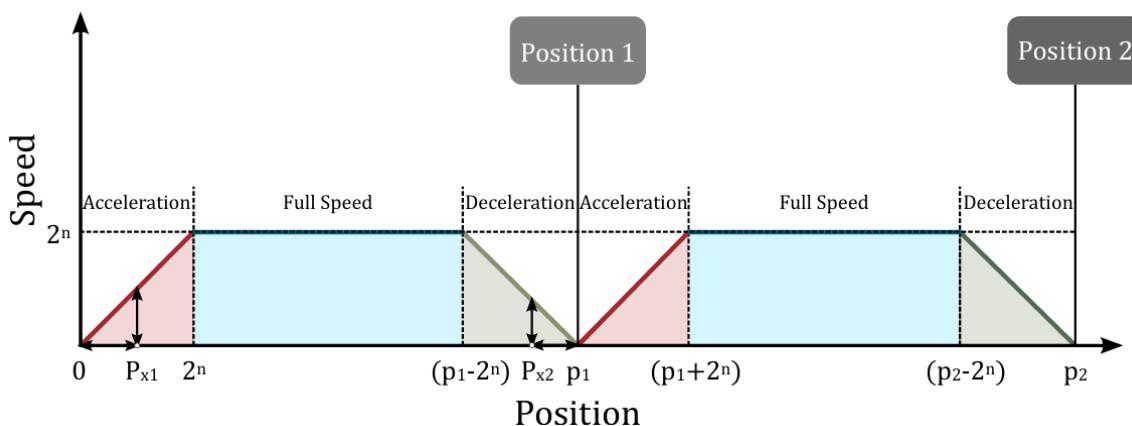


Figure 1: Déplacement du curseur [4]

Pour régler la vitesse du moteur un signal de **PWM** est appliqué au signal side1 ou side2 du moteur (voir [3.1](#) pour infos sur les signaux).

La figure [2](#) montre le fonctionnement de **PWM**.

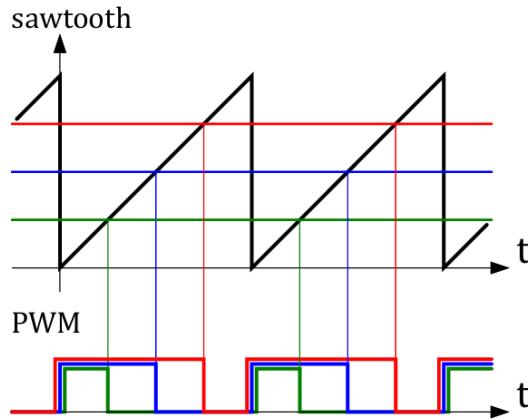


Figure 2: Fonctionnement de PWM [4]

Un signal quelconque (en couleur dans l'image qui sera appelé "duty cycle") est comparé avec une dent de scie de fréquence de 100 kHz ??, si le signal duty cycle est plus petit que la valeur de la dent de scie le signal de sortie PWM est '1', sinon '0'.

De cette manière il est possible de moduler la durée du '1' par rapport à une période de la dent de scie, si le signal est à '1' pendant la moitié du temps alors le moteur va bouger à 50% de la vitesse maximale.

La figure 3 est une schématisation simplifiée du fonctionnement de la PWM.

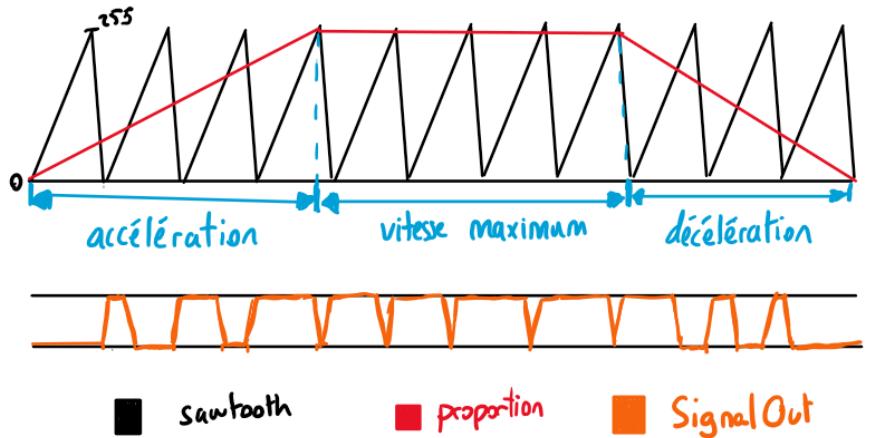


Figure 3: Schématisation de la PWM

La dent de scie de 100 kHz est codée sur 8 bits et est générée dans le bloc tr66MHzto100kHz 5.3.6.

Le signal duty cycle est obtenu à travers un compteur itératif (compteur relatif) qui utilise le même "enable" du compteur de position, le compteur relatif est initialisé à la vitesse minimale 2.4.2, pour accélérer le moteur, le compteur augmente, et pour ralentir, le compteur décompte à la vitesse maximale. Un mux fait en sorte que l'enable n'arrivera pas au compteur pour le bloquer.

Duty cycle est codé sur 14 bits mais se fait transformé en 8 bits par un right bit-shift <sup>2</sup> de 6 bits. pour la comparaison avec la dent de scie.

<sup>2</sup>Les premiers 8 bits du signal deviennent les 8 bits finales, c'est l'équivalent de faire une division par 64 et arrondir vers le bas

### 2.4.1 Delta PWM et dimension du bit-shift

La **PWM** peut se baser sur le temps ou sur le déplacement, cela dépend du enable utilisé pour le compteur du duty cycle.

Soit l'enable se base su la fréquence du clock (basé sur le temps), soit, comme dans le cas de ce projet, le enable dépend du encodeur, du coup la distance de l'accélération dépend de la distance parcourue par le chariot.

Sans le bit-shift, le compteur va arriver à la vitesse maximale (255 ou "11111111") après 256 coup de clock. Ceci se traduit, en  $256 * 0.000875mm = 0.224mm$ , une distance beaucoup trop courte pour une accélération.

En multipliant:  $64 * (256 * 0.000875) = 14.336mm$  le résultat est beaucoup plus acceptable comme distance.

L'équivalent en steps du compteur est 16'384, en réalité, vu que le moteur ne démarre pas à une vitesse nulle il y'a une partie de l'accélération qui est ignorée; du coup le curseur ralentissait un peut trop tôt. La distance d'accélération a été réduite à 13'000 steps, soit env. 11.375mm.

Le compteur relatif est donc à 14 bits et il y'a un bit-shift pour ralentir la variation du duty cycle et agrandir la distance d'accélération.

### 2.4.2 Vitesse minimale

Si le rapport entre le duty cycle et la dent de scie est trop bas, le signal de **PWM** qui arrive au moteur génère une tension moyenne trop basse pour que le moteur aille assez de puissance pour surpasser la friction interne et l'inertie mécanique.

Il y'a donc un seuil duty cycle a surpasser pour faire bouger le moteur, cette dynamique a amené à quelque problème dans le fonctionnement du projet. Les détails de la résolution sont dans le chapitre 7.

### 3 Composants

Cette partie de rapport sert seulement à donner des informations sur les composantes qui sont fondamentales pour la compréhension de la continuation du rapport, pour toute info précise sur les composantes voir le document de labo [4]

Le circuit se compose de 3 platines différentes comme montré dans la figure 4.

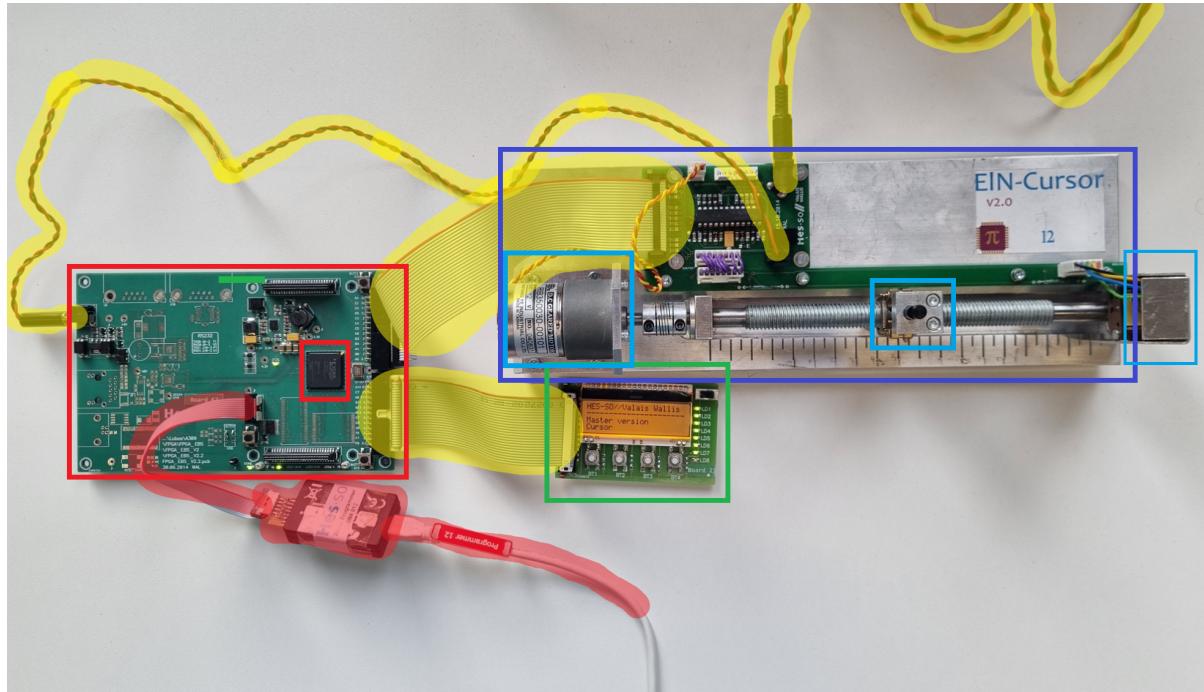


Figure 4: Les composants du curseur

Les parties principales sont:

1. Assemblage avec chariot et **PCB** (Bleu)
2. Carte de développement **FPGA** (Rouge)
3. Carte de control avec 8 LEDS e 4 boutons (Vert)

Encadrées en azuré sont le moteur, le chariot et l'encodeur et en jaune sont toutes les connections de transmission des signaux et de l'alimentation des platines.

#### 3.1 Moteur et control du moteur (pont en H)

Le chariot est propulsé par un moteur à courant continu 12 V, piloté par un driver en pont en H (L6207) qui permet de gérer le sens de rotation et la vitesse. Cette vitesse est modulée via un signal **PWM** dont la fréquence ne doit pas excéder 100 kHz pour respecter les limites du composant(2.4). Le moteur entraîne une vis M12 x 1,75, transformant chaque rotation complète en un déplacement linéaire précis de 1,75 mm.

##### 3.1.1 Explication fréquence 100kHz

Selon un document de Portscap [1], les paramètres à regarder pour choisir la fréquence de la dent de scie sont les suivantes:

Une fréquence majeure de 20 kHz est conseillée pour éviter des bruits audibles.

Plus la fréquence est haute, plus le moteur réagit bien au signal **PWM**, la période de la dent de scie est tellement courte que le signal d'alimentation est presque continu.

La fréquence de 100 kHz est la limite maximum pour obtenir un bon fonctionnement, à plus de 100 kHz le "switching" des **mosfets** du pont H n'est pas garanti et peut donc causer des problèmes.

100kHz est donc le bon compromis entre un fonctionnement fluide du moteur et la sécurité du circuit. Dans le cas de ce projet, le pont H utilisé est un L6207 de [STMicroelectronics®](#), selon le [datasheet](#) du pont, la fréquence maximale de travail est d'exactement 100kHz, voir extrait dans figure 5.

## Features

- Operating supply voltage from 8 to 52 V
- 5.6 A output peak current (2.8 A DC)
- $R_{DS(ON)}$  0.3  $\Omega$  typ. value at  $T_j = 25^\circ\text{C}$
- Operating frequency up to 100 KHz
- Non-dissipative overcurrent protection
- Dual independent constant  $t_{OFF}$  PWM current controllers
- Slow decay synchronous rectification
- Cross conduction protection

Figure 5: Extrait du datasheet du pont h

## 3.2 Encoder

L'angle de rotation de la vis est mesuré par un codeur incrémental (AEDB-9140-A12) [4]. Il génère 500 impulsions par tour (CPR) sur deux canaux, Channel A et Channel B. En analysant le déphasage entre ces deux signaux, le système peut déterminer avec précision le sens de rotation et la position exacte du chariot. Un troisième canal, Channel I (Index), fournit une impulsion de référence par tour complet. Cependant le Channel I n'est pas utilisé.

### 3.2.1 Fréquence de clock

La fréquence d'horloge du système (clock) s'élève à 66 MHz lors de l'utilisation de la carte FPGA-EBS 2 [4]. Cette fréquence est imposée par l'oscillateur local qui génère le signal de référence nécessaire au cadencement de la puce Xilinx Spartan.

## 4 Aperçu du projet

Au début le projet se présente comme dans la figure 6.

Tous les inputs et outputs nécessaires au fonctionnement du circuit sont connectés au bloc encagé en rouge. Le bloc est vide et doit être rempli avec des sous blocs qui créent la logique du système.

La partie en jaune est la gestion des input des quatre boutons, avec un bloc "debounce" pour annuler le rebond mécanique du bouton.

Le rectangle bleu continent tous les inputs de la platine du curseur, avec les reed-relais avec un debounce et les trois inputs de l'encodeur.

Entouré en vert sont tous les outputs pour les 8 LEDs et l'écran LCD, et en violet les trois sorties qui viennent utilisées pour bouger le moteur.

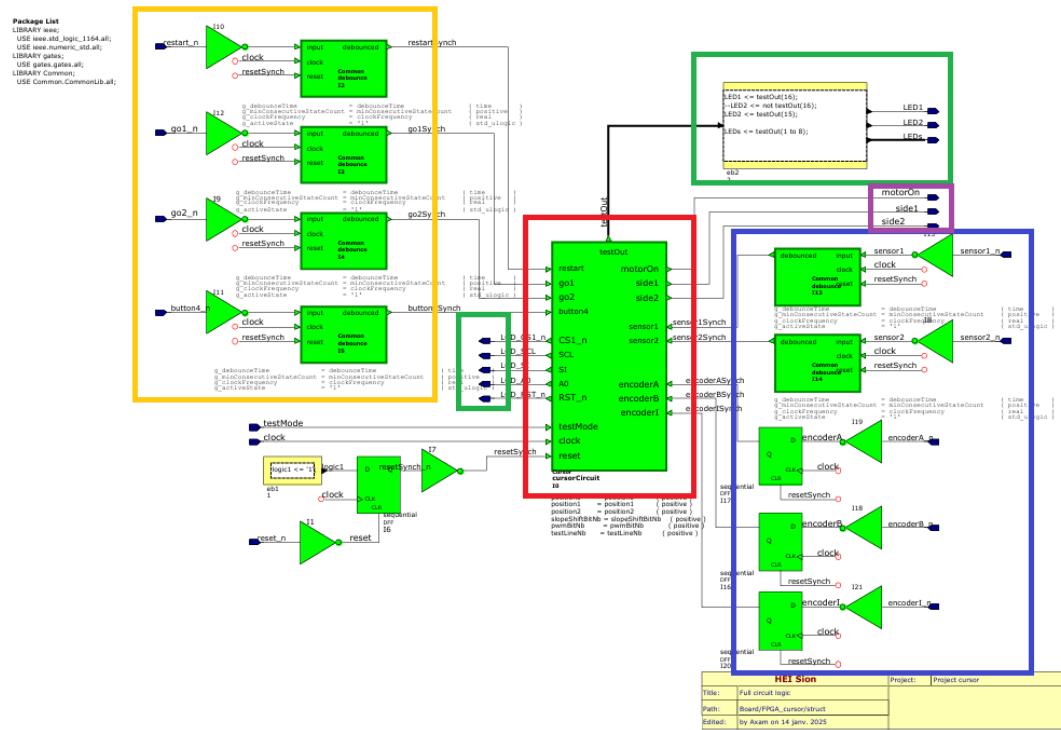


Figure 6: Extrait du circuit complet (voir pdf complet dans l'annexe nr.1)

La partie de logique en bas à gauche sert à générer le signal **clock** et le **reset** initial.

### 4.1 Concept de fonctionnement

L'idée est de créer un bloc de control principal qui envoie des infos concernant le déplacement a un bloc qui contrôle le moteur.

Un bloc de control des boutons fournit l'état des boutons au bloc de control principal.

Un bloc qui lit la valeur des senseurs et voit de quelle direction le moteur bouge est utilisé pour arrêter le moteur dans le cas ou le curseur touche un des deux cotées.

Toute cette logique<sup>3</sup> est programmée dans le bloc principal de la figure 6.

Chaque bloc décrit peut être composé de nouveaux sous-blocs, jusqu'à arriver à la logique de base qui se compose soit de portes logiques, soit d'une **FSM** c.a.d une machine d'état.

<sup>3</sup>Depuis maintenant "logique de haut niveau" ou "top level logic"

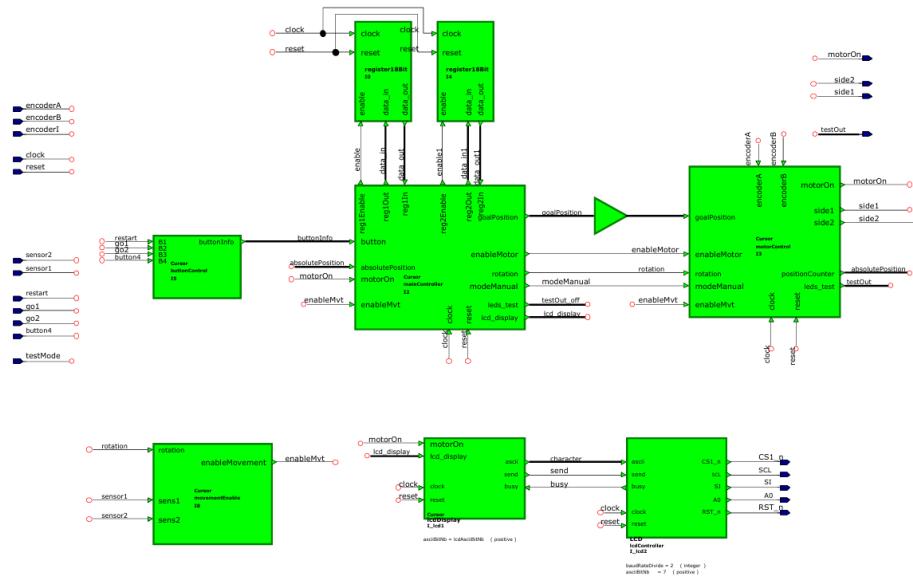


Figure 7: Logique de haut niveau

L'image 7 montre la logique de haut niveau avec tous ces blocs principaux:

- **buttonControl** → compréhension des boutons
- **mainController** → action en fonction des boutons et position
- **motorControl** → Bouge le moteur en fonction de l'action décidé par **mainController**
- **register18Bit** (2x) → sauver de manière dynamique deux positions-but
- **movementEnable** → gestion stop aux limites du curseur
- **lcdDisplay** et **lcdController** → Logique pour écrire dans l'écran **LCD** en bas e droite

En général le système fonctionne comme suivant:

1. Le bloc **buttonControl** lit les valeurs des boutons et met en sortie un bus à 4 bit.
2. Le bloc **main controller** reçoit le bus à 4 bit et connaît la position du curseur. Il peut donc fournir des informations pour bouger le chariot, comme la position-but, le sens de rotation et un signal **enableMotor** qui est utilisé pour dire au bloc du moteur qu'il faut commencer à bouger.
3. Le bloc **motorControl** reçoit les trois informations et bouge en conséquence. En faisant ça il s'occupe d'accélérer et décélérer le moteur. Pendant tout le processus il fait la mise à jour de la position actuelle du chariot.
4. Une fois que le moteur ne bouge plus le **mainController** est prêt pour envoyer des nouvelles instructions.

## 4.2 Compteurs itératives

Le gros du système se base sur des compteurs itératifs pour gérer les positions et la vitesse du moteur. Les compteurs se forment de plusieurs compteurs à 1 bit mise en série par un cycle **FOR** dans **VHDL**.

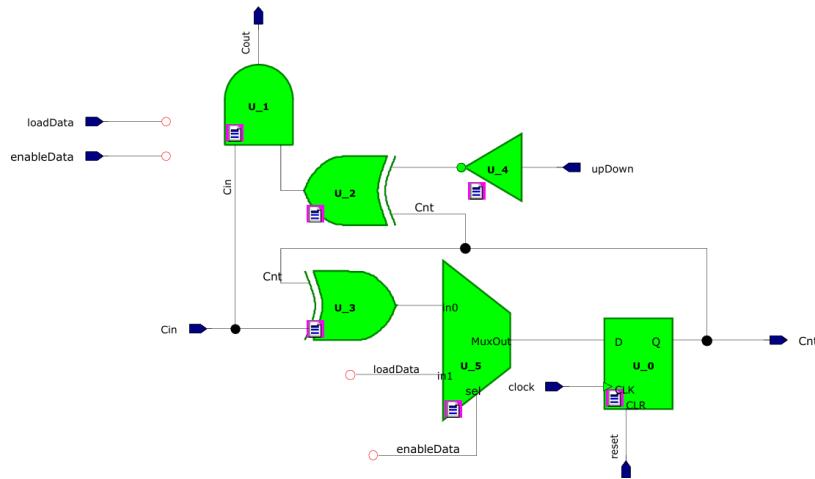


Figure 8: Logique de un compteur 1-bit

La figure 8 montre la logique d'un compteur à 1 bit Il contient une partie qui permet un data-load et une partie pour choisir si compter et décompter.

### 4.3 Additionneur à propagation de report

Les explications suivantes sont inspirée du document de labo "Additionneurs Binaires" [3].

L'additionneur à propagation de report est une structure arithmétique modulaire. Il repose sur la mise en cascade de plusieurs blocs itératifs, où chaque bloc traite les bits de même poids des deux opérandes.

Pour chaque rang  $i$ , le bloc effectue l'addition de trois entrées: les deux bits de données ( $a_i$  et  $b_i$ ) ainsi que la retenue d'entrée ( $c_i$ ) provenant du rang précédent.

Chaque bloc produit ensuite deux sorties distinctes:

- Le bit de somme ( $s_i$ ): représente le résultat de l'addition.
- La retenue de sortie ( $(c_{i+1})$ ): elle est immédiatement transmise au bloc de rang supérieur ( $i+1$ ).

La figure 9 montre le fonctionnement d'un additionneur à propagation de report.

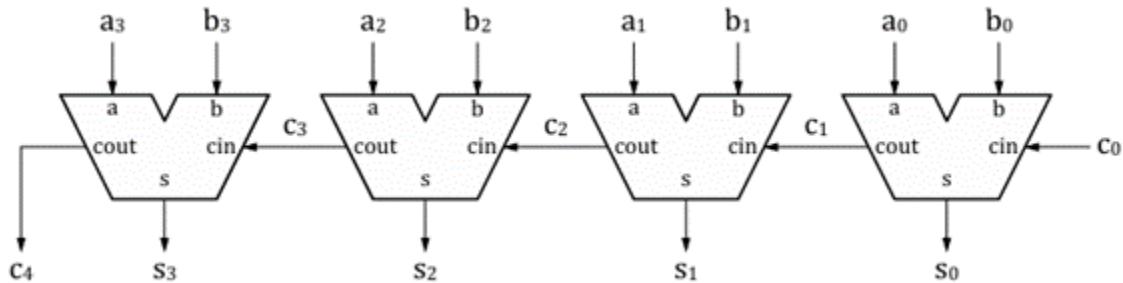


Figure 9: Fonctionnement d'un additionneur à propagation de report

## 5 Description des blocs

Ce chapitre se focalise sur la description des différentes blocs, pour voir leur contenu de manière détaillée, consulter les annexes [8.2](#).

### 5.1 Button manager

Le bloc `buttonManager` reçoit en entrée les signaux des 4 boutons, `restart`, `go1`, `go2` et `button4` renommée `B1`, `B2`, `B3`, `B4` respectivement et les envois directement sur le bus de sortie `ButtonInfo` à 4 bits.

Le rôle de ce bloc est de regrouper les 4 signaux des boutons en un bus qui diminue le nombre d'entrée dans d'autres blocs (figure [10](#)). Ce bloc servait aussi à bloquer tout usage des boutons lors que le moteur tournait. Cette sécurité a dû être retiré à cause du [6.3](#).

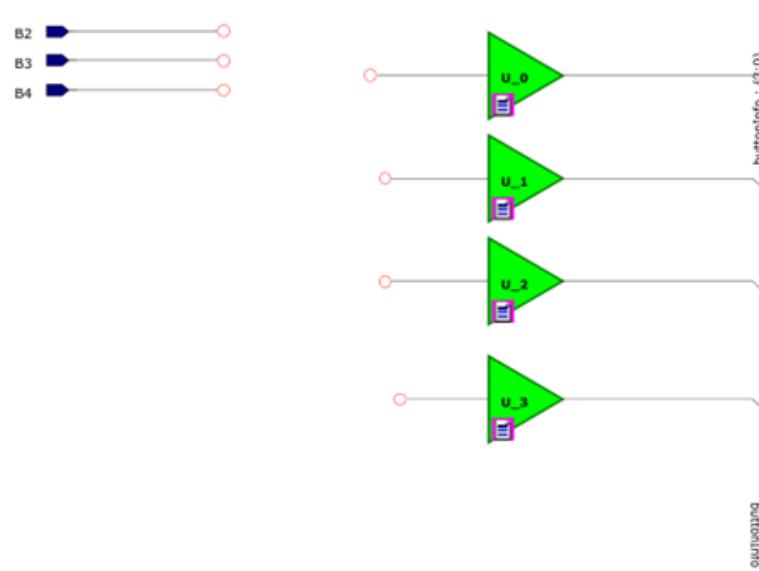


Figure 10: Logique du bloc "buttonManager"

### 5.2 Main controller

La figure [11](#) montre le bloc `mainController` avec tous ses inputs et outputs.

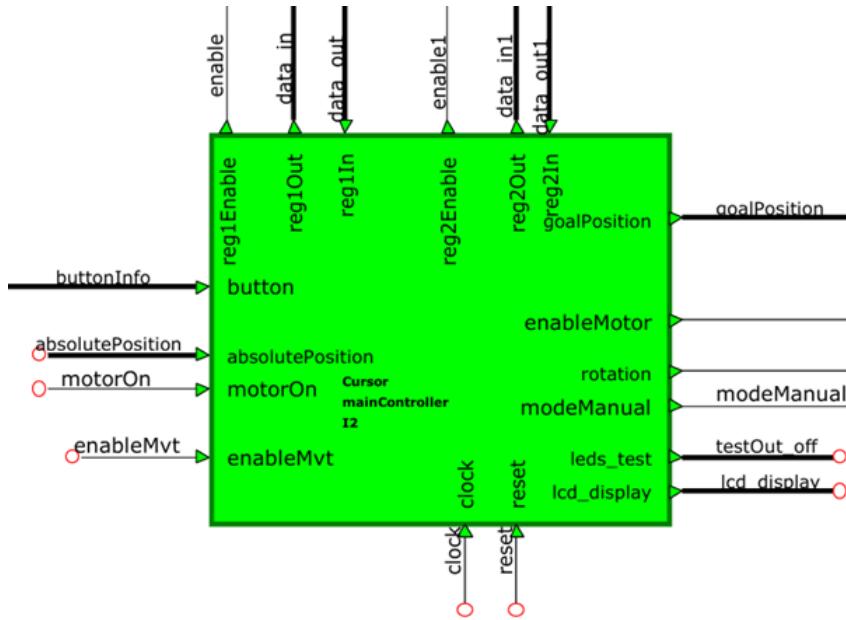


Figure 11: Inputs et outputs de "mainController"

Les inputs sont :

- **motorOn**: signal de l'état du moteur.
- **enableMvt**: signal d'arrêt provenant de l'état des capteurs.
- **button**: un bus contenant l'information de l'état des signaux des boutons (4 bits).
- **absolutePosition**: position actuelle du chariot (18 bits).
- **reg1In**: bus de lecture du registre 1 (18 bits).
- **reg2In**: bus de lecture du registre 2 (18 bits).
- **clock** et **reset**: signal du clock et reset initial.

Les outputs sont :

- **reg1Enable**: signal enable qui permet d'écrire dans le registre 1.
- **reg2Enable**: signal enable qui permet d'écrire dans le registre 2.
- **reg1Out**: bus transportant la valeur à enregistrer dans le registre 1.
- **reg2Out**: bus transportant la valeur à enregistrer dans le registre 2.
- **goalPosition**: position-but que le chariot doit atteindre (bus 18 bits).
- **enableMotor**: signal qui dit au bloc **motorControl** d'arrêter le moteur (position final atteint).
- **rotation**: sens de rotation (0=avant, 1=arrière).
- **modeManual**: un signal qui passe le programme en mode manuel.
- **leds\_test**: un bus disponible pour tester le fonctionnement du programme lors de la mise en service (non utilisé à la fin du projet).
- **lcd\_display**: un bus utilisé pour contrôler l'affichage du **LCD**.

Le bloc regroupe les suivantes sous-blocs:

- **Rotation\_Sous**.
- **ButtonControlFSM**.
- **Counter 28 bits pour le countdown du bouton 4**.

A regarder en annexe pour voir la mise en place des sous blocs énuméré ci-dessus.

Particularité:

- Un transformateur est utilisé pour convertir le bus `button4counter` de type `unsigned` en `std_ulogic_vector` (figure 12).
- Une bascule-E est utilisée pour éliminer les `glitches` du signal `side` et pour seulement lire la valeur de la première opération du bloc `Add/Sous`.

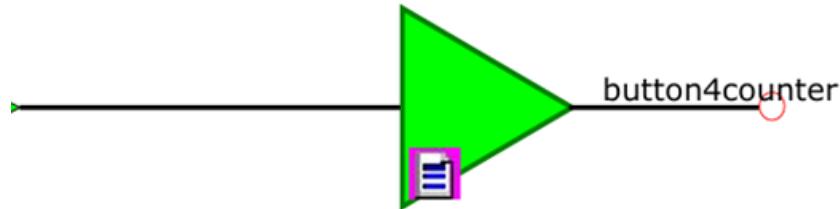


Figure 12: Bloc mainController, transformateur unsigned → std\_ulogic\_vector

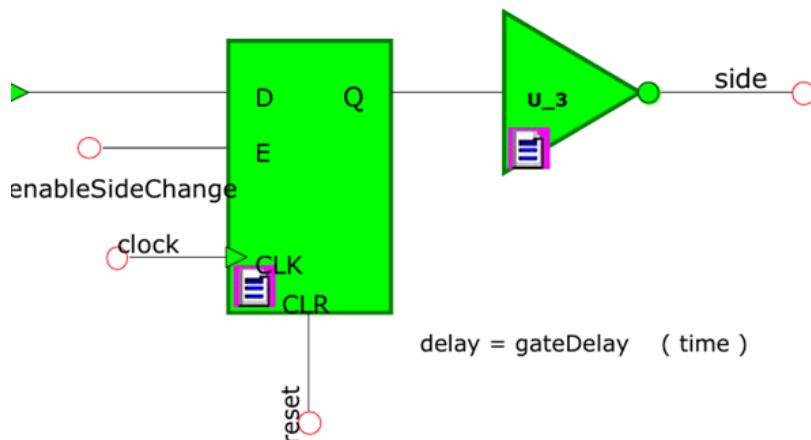


Figure 13: Bascule "E" pour "glitches"

### 5.2.1 Rotation Sous

Le bloc `rotation_Sous` a en entrée les bus `absolutePosition` et `goalPosition` à 18 bits et le signal de sortie `CntOut1`.

Le bloc prend la position actuel(`absolutePosition`) donnée par le `compteur absolu` et soustrait à la position d'arrivée (`GoalPosition`). Pour cela le concept d'un `additionneur à propagation de retour` et le complément à deux est utilisé (figure 14).

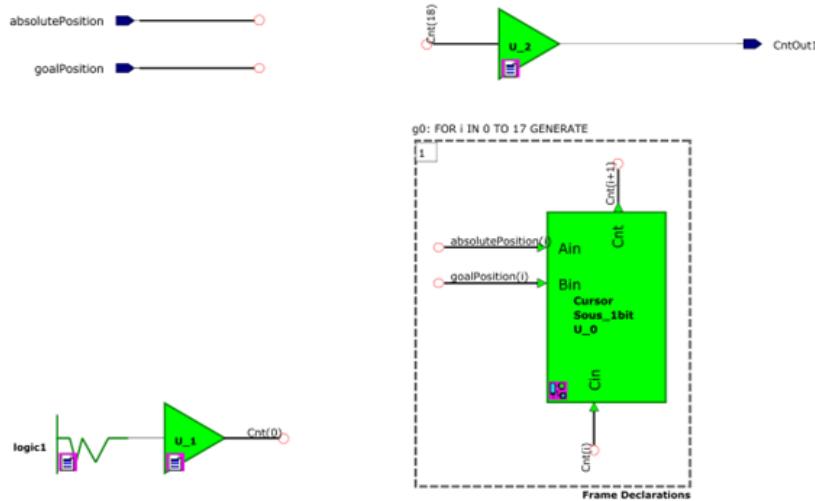


Figure 14: Logique du bloc "Rotation\_Sous"

La valeur de CurrentPosition est inversée puis additionnée à Goalposition. Il faut aussi additionner 1, donc le premier bit du bus cnt est initialisé à 1 (figure 15).

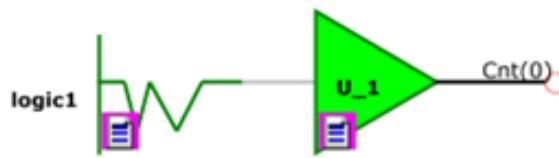


Figure 15: Initialisation à 1 dans le bloc Rotation\_Sous

La valeur de la soustraction n'est pas utilisée dans son entiereté, seulement le 19ème bit de la soustraction [cnt(18)] est utilisé (figure 16). Il permet de savoir si la position actuelle est plus grande que la position d'arrivée en regardant si le résultat de la soustraction est positif ou négatif.



Figure 16: Rotation\_Sous, transfère du bit 19

La logique du bloc **Sous\_1bit** se base sur un soustracteur de base. Cependant la logique a été réduit pour ne seulement calculer la retenue de sortie (figure 17).

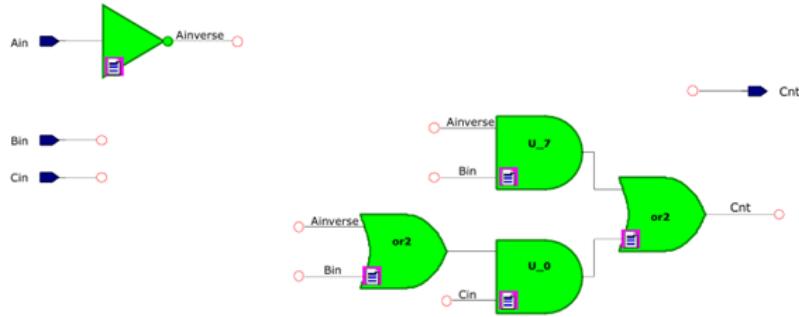


Figure 17: Bloc de soustraction simplifié

### 5.2.2 Bloc ButtonControlFSM

La figure 18 montre le bloc ButtonControlFSM.

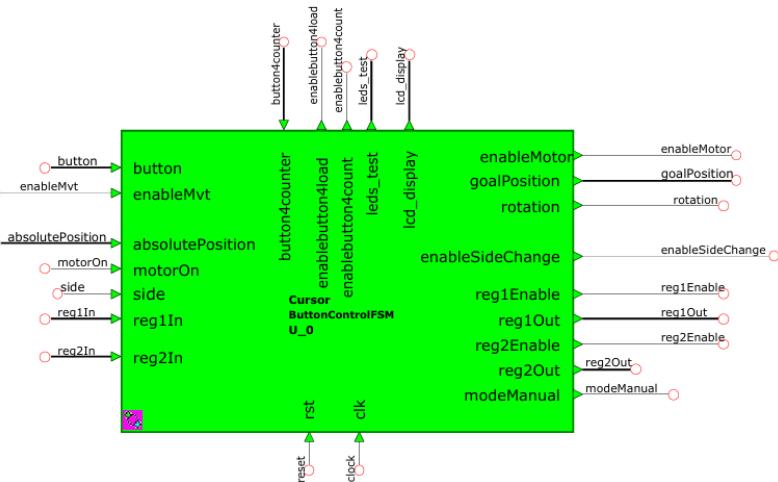


Figure 18: Bloc ButtonControlFSM avec ses inputs et outputs

Les inputs de base sont les mêmes que ceux énumérés dans le chapitre du [mainController](#) (5.2).

Les inputs supplémentaires du bloc sont :

- Side: signal provenant du bloc **Rotation\_Sous**.
  - Button4counter: un bus (28 bits) qui contient une valeur utilisée pour savoir combien de temps le bouton 4 est maintenu.

Les outputs supplémentaires sont :

- `Enablebutton4load`: un enable qui réinitialise le compteur à 28 bits à 0.
  - `Enablebutton4count`: un enable qui actionne le comptage du compteur à 28 bits.
  - `enableSideChange`: un enable qui permet de lire seulement la première opération du bloc `Rotation_Sous`.

Le bloc `buttonControlFSM` à 2 parties :

- La partie demandée dans le cahier des charges.
  - Une deuxième partie qui gère toutes les options supplémentaires. Celle-ci complète la première partie, voir chapitre 6.

En annexe se trouve la FSM du bloc `buttonControlFSM` dans son intégrité. La figure 19 montre la partie de la `FSM` dédié à la gestion des fonctions obligatoires selon le cahier des charges (2.1).

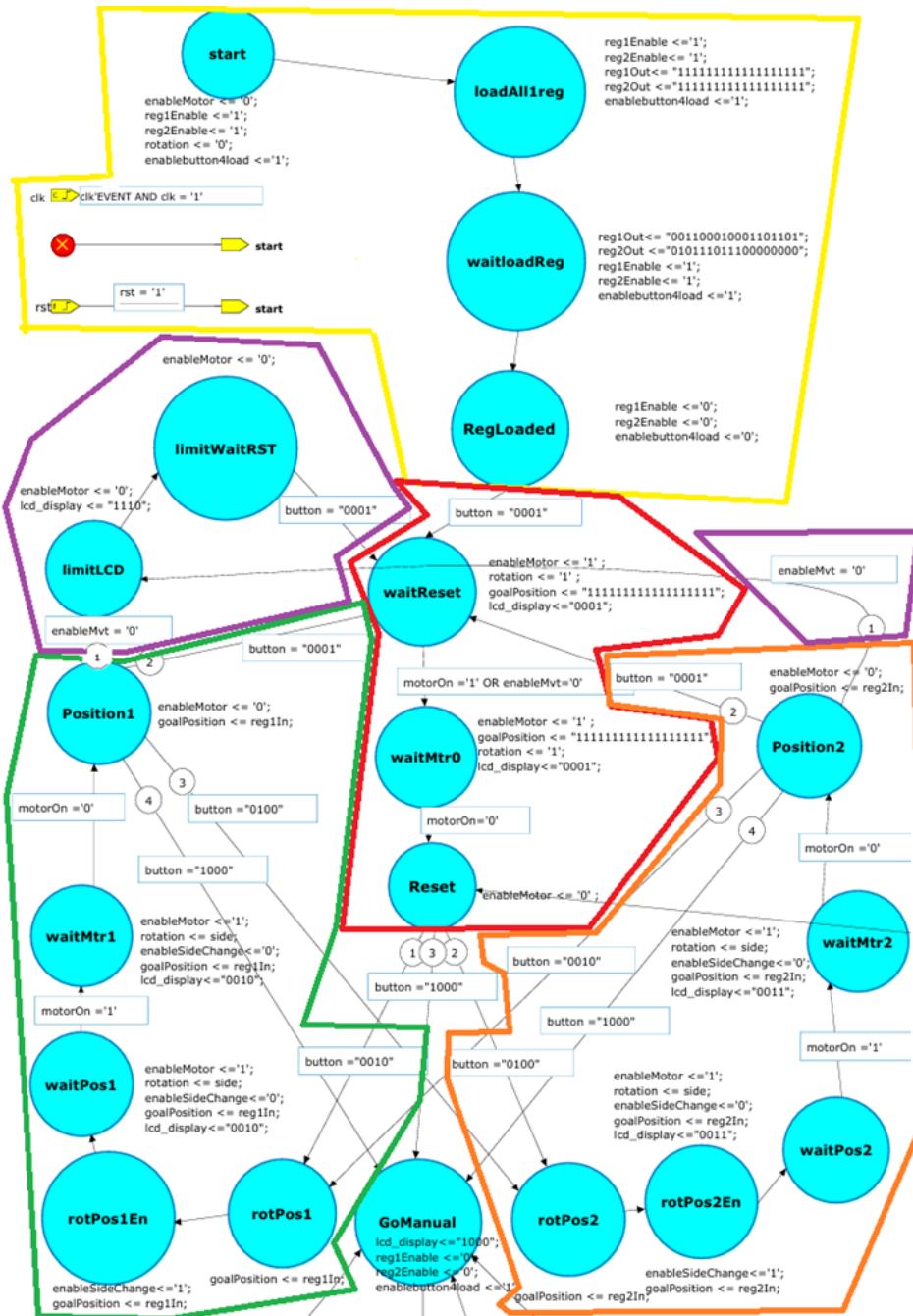


Figure 19: Partie obligatoire de la FSM, color coded

LA FSM est construite en 5 parties :

- La partie jaune: l'initialisation.
- La partie rouge: restart.
- La partie verte: position 1.
- La partie orange: position2.
- La partie violette: limite.

Pour simplifier la compréhension de la FSM un diagramme regroupant l'idée général du fonctionnement de la **FSM** est montré dans l'image 20.

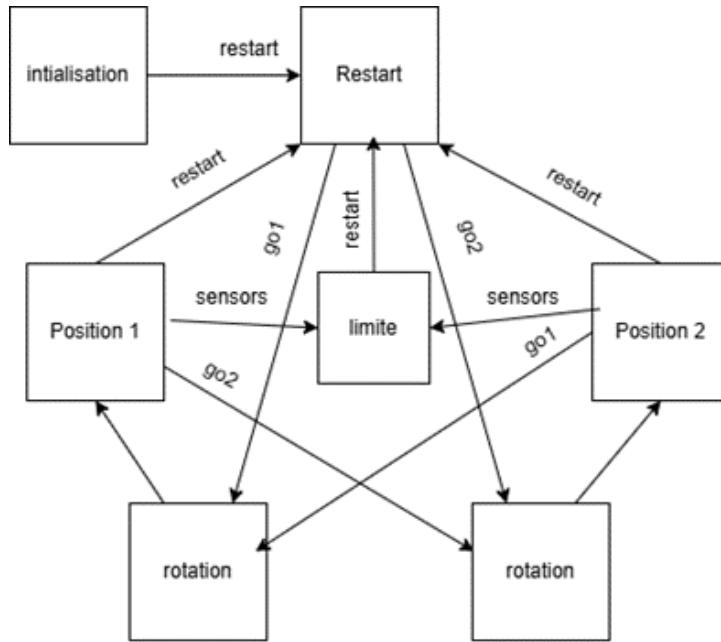


Figure 20: Button control FSM, diagramme de fonctionnement

Dès l'alimentation du système, la FSM effectue la phase d'initialisation. Les valeurs des positions 1 et 2 sont écrites dans les registres 1 et 2 respectivement, et le compteur pour l'option supplémentaire est mis à 0. Rien ne se passe tant que le bouton *restart* n'est pas appuyé.

Quand le bouton *restart* est actionné, la FSM donne une rotation par défaut de 1, une position de fin (1111111111111111) et l'autorisation de bouger le moteur (*enableMotor*). Dès que soit le capteur s'active ou *motorOn* passe à 0, la FSM sait que le chariot se trouve à la position initiale (*reset*).

À ce stade, il est possible de soit appuyer sur le bouton *go1* ou *go2*. La rotation se calcule de la même manière dans les deux cas. La seule différence est que si *go1* est actionné, la position d'arrivée est mémorisée dans le registre 1. Pour *go2*, c'est l'inverse (mémorisation dans le registre 2). Dès que le signal *motorOn* passe à 0, la FSM sait que le chariot est arrivé à sa destination. Il est donc possible d'actionner à nouveau l'un des 2 boutons (autre que celui utilisé précédemment) pour recommencer un déplacement.

Une sécurité supplémentaire a été ajoutée. Si l'un des deux capteurs s'actionne pendant un déplacement vers la position 1 ou 2, la FSM passe à l'état *limite*. Le moteur s'éteint et seul le bouton *restart* peut être actionné.

### 5.3 Motor control

La figure 21 montre le bloc `motorControl`.

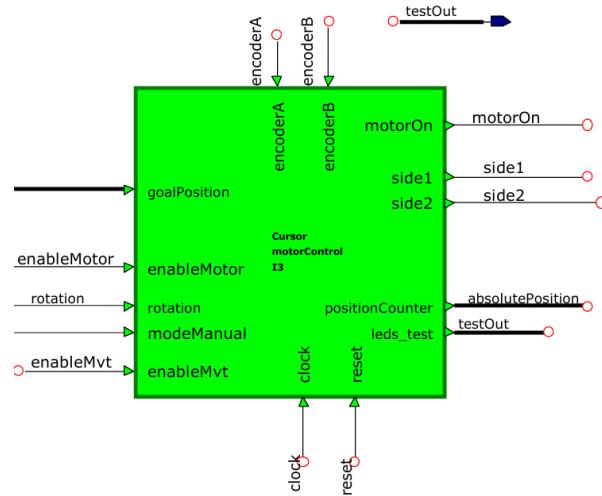


Figure 21: Bloc motorControl - inputs et outputs

Les inputs sont:

- **goalPosition**: position-but que le chariot doit atteindre (bus 18 bits).
- **enableMotor**: signal qui dit au bloc de commencer à bouger le moteur.
- **rotation**: sens de rotation du moteur (0=avant, 1=arrière).
- **enableMvt**: signal qui dit au bloc d'arrêter le moteur (limites atteintes).
- **modeManual**: mode manuel ou automatique (explication dans le chapitre 6).
- **encoderA** et **encoderB**: signaux de l'encodeur pour la mise à jour de la position.
- **clock** et **reset**: signal de clock et reset initial.

Les outputs sont:

- **Side1** et **Side2**: signaux pour bouger le moteur (voir 3.1 pour plus d'infos).
- **positionCounter**: position actuelle du chariot (bus 18 bits).
- **motorOn**: signal d'activation du moteur.
- **testOut**: bus pour allumer les LEDs (utilisé pour débogage au début, bus 2 bits).

Le but de ce bloc est de bouger le curseur en fonction des informations qui arrivent par le bloc **mainController**. Il renvoie au bloc **mainController** la position actuelle du chariot et l'état du moteur (allumé ou éteint) pour que le **mainController** sache quand envoyer des nouvelles instructions et puisse calculer le sens de rotation en fonction de la position-but.

Les blocs internes (visable dans l'annexe) sont:

- **ChangeDetector**: Logique pour détecter le changement d'état de l'encodeur.
- **relCounterPWM**: compteur relatif pour la gestion de la vitesse.
- **itératiCounter**: nom pas trop descriptif, compteur de position absolue.
- **AddSous**: Logique pour calculer la position où le chariot commence à décélérer.
- **motorFSM**: bloc qui gère l'état du moteur (arrêté, accélération, vitesse constante, décélération).
- **PWMGenerator**: Générateur du signal **PWM** pour bouger le moteur.
- **ledPowerFSM**: Machine d'état pour allumer les LEDs en fonction de la puissance appliquée au moteur.
- **alleLEDBusCreator**: Logique pour mettre le bus dels leds à 8 bits ensemble avec le bus de leds de débogage de la **FSM** à 2 bits.

### 5.3.1 Change detector

Le but de ce bloc est de générer un signal à 1 bit qui sera utilisé comme "enable" pour les compteurs de position absolue et relative.

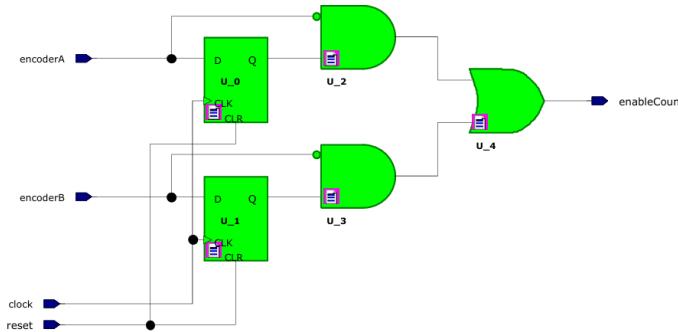


Figure 22: Logique du bloc change detector

La logique montrée dans la figure 22 fait en sorte que la de sortie est '1' chaque fois que l'état représenté par l'ensemble des signaux `encoderA` et `encoderB` change.

Le circuit se compose de deux détecteurs changement de bit mis ensemble dans un xor. Chaque fois que l'un des deux bits change, la sortie devient '1'.

Le tableau 1 montre les états actuels et futures de l'encodeur (ensemble de A et B).

$Q_A$	$Q_B$	$Q_A^+$	$Q_B^+$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	1

Tableau 1: État actuel et suivant de l'encodeur

### 5.3.2 Relatif counter

Le bloc `relativCounterPWM` est un compteur itératif avec `count/decount`, `loadData` et `enableLoad`, le but de ce compteur est de générer le signal duty-cycle utilisé dans la modulation de la vitesse du moteur. Le signal de sortie est en suite envoyé au bloc `motorFSM`.

La figure 23 montre les deux blocs contenues dans `relCounterPWM`: un compteur itératif à 14 bits et un bloc `rightShifter` qui s'occupe du bit-shift du bus en sortie du compteur.

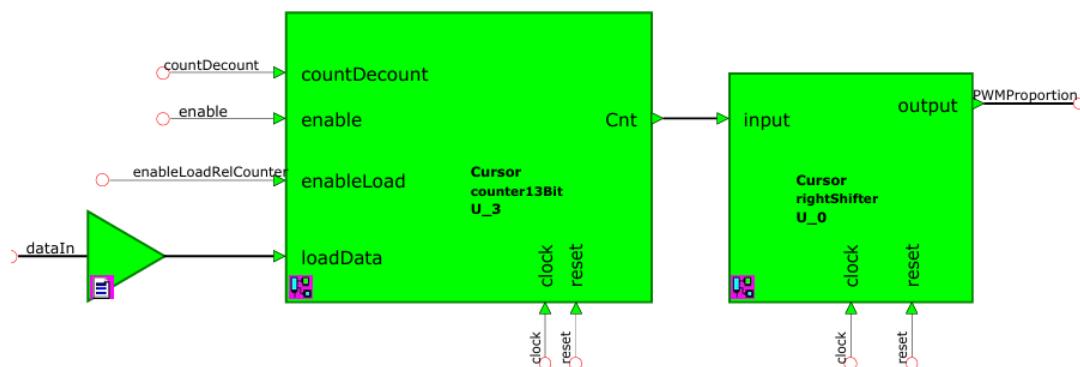


Figure 23: Logique du bloc relCounterPWM

Pour le choix du nombre de bit-shift voir 2.4.1, pour la logique du bloc voir l'annexe 23

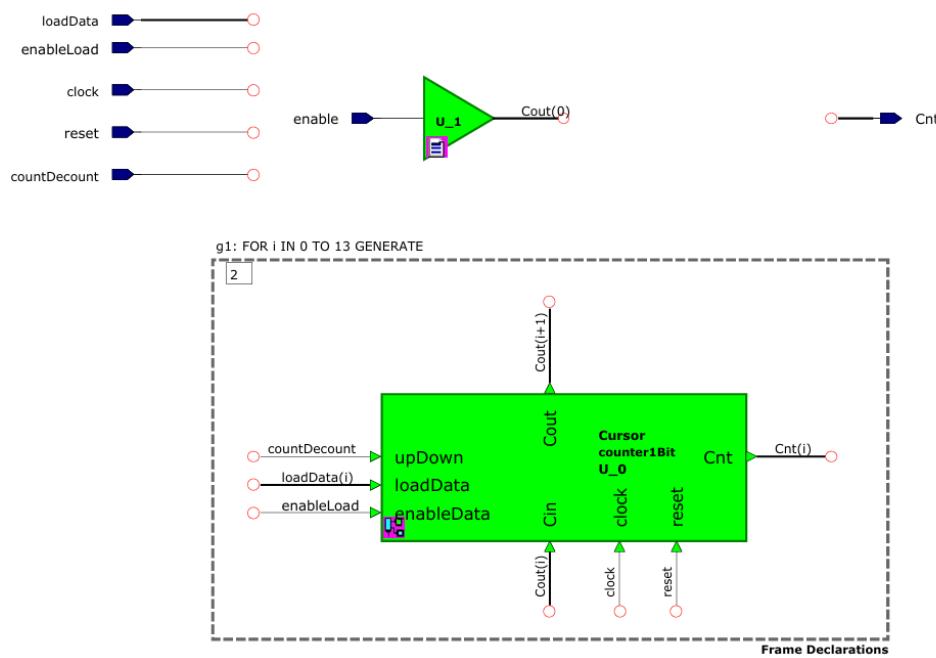


Figure 24: Boucle FOR pour le compteur relatif

La figure 24 montre la boucle FOR utilisée pour créer le compteur itératif à 14 bits.

### 5.3.3 Compteur de position

Le bloc `iterativCounter` est un compteur itératif, le but de ce compteur est de garder la position actuelle du chariot.

Ce bloc est le coeur du système vu qu'il s'occupe de la position actuelle du chariot. Cette position est utilisée dans le bloc de `motorControl` mais aussi dans le bloc principal `mainController` (après une transformation de "unsigned" à "std\_ulogic\_vector").

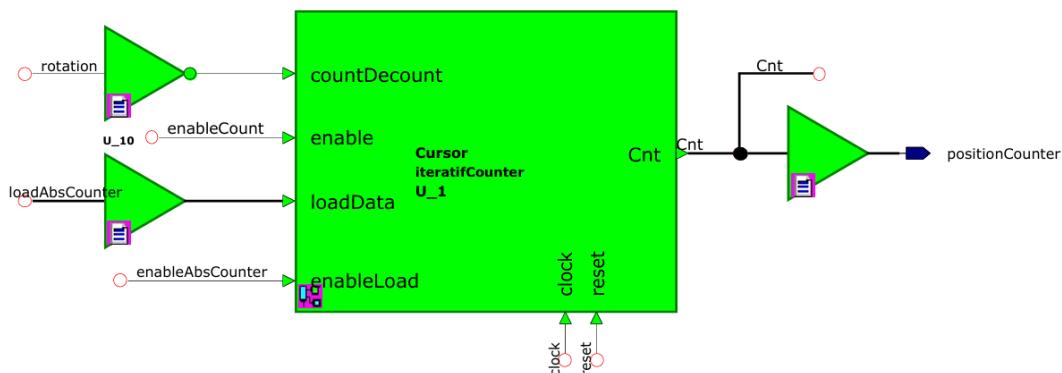


Figure 25: Compteur de position

La figure 25 montre l'extérieur du compteur. De plus, il est visible que le signal `count/decount`, c'est-à-dire le signal qui décide si le compteur doit compter ou décompter, est géré par le signal de `rotation` qui vient du bloc principal `mainController`. De cette manière, si le chariot doit avancer, le compteur compte; sinon il décompte.

Un inverseur est utilisé pas ce que au début le compteur fonctionnait à l'envers.

### 5.3.4 Position de ralentissement (AddSous)

Le bloc **AddSous** a en entrée le bus **goalPosition** à 18 bits, le bus **delta** à 18 bits et le signal **rotation** et en sortie le bus **start** à 18 bits.

Ce bloc a comme objectif de soit additionner ou soustraire à la position d'arrivée la valeur de **deltaPWM**. Le résultat de l'opération est la position absolue du début de la décélération. Il est transmis par le bus **start** à la **FSM** (figure 26).

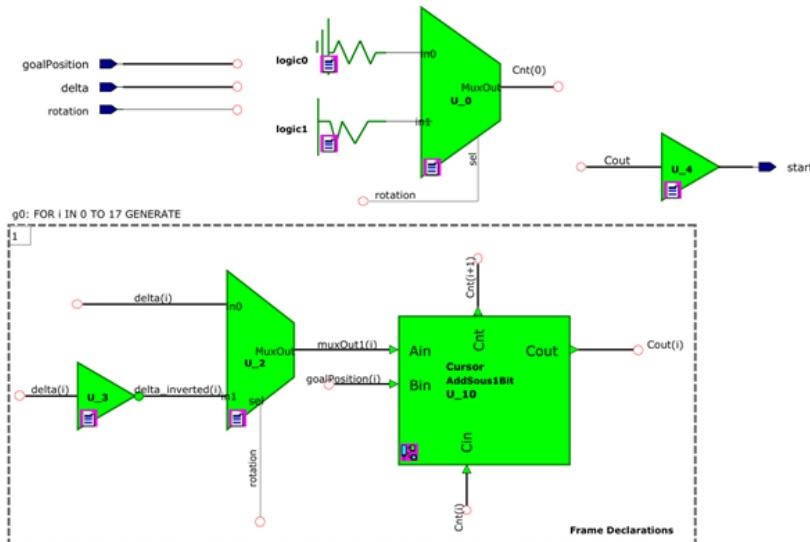


Figure 26: Logique du bloc "addSous"

Pour que ce bloc fonctionne comme voulu, un **additionneur à propagation de report** est utilisé. Le complément à deux est aussi utilisé pour effectuer la soustraction. Le signal **rotation** permet de sélectionner quelle opération va être faite. Par exemple si **rotation = '1'** le bloc va soustraire **delta** à **goalPosition** et inversement va additionner **delta** à **goal Position**. Ceci est nécessaire car ce projet travaille avec des positions absolues.

La figure 27 est un dessin explicatif fictionnel de la réflexion derrière la logique du bloc **AddSous**.

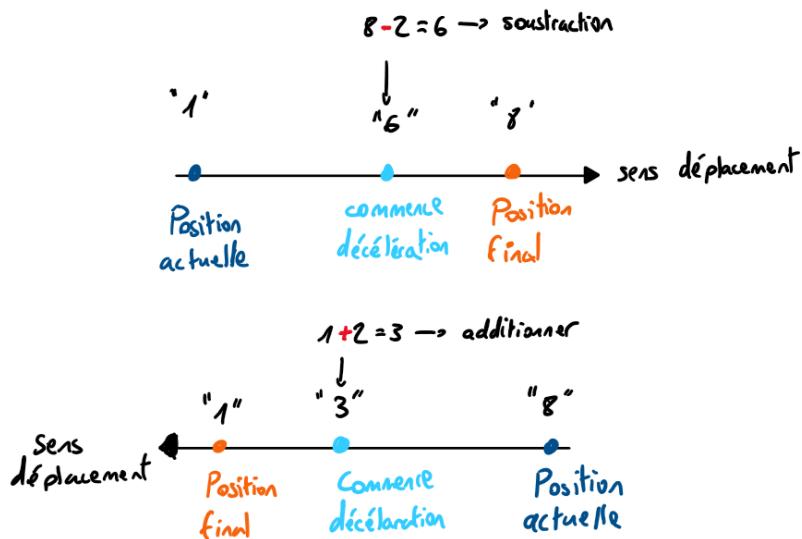


Figure 27: Dessin explicatif du fonctionnement de **addSous**

### 5.3.5 Motor FSM

Le bloc `motor` FSM contient un seul sous-bloc, `motorControlFSM` qui contient la machine d'état dédiée à la gestion du moteur.

La figure 28 montre les inputs et outputs du bloc `motorFSM`.

En bleu sont encadrées les outputs utilisées pour la gestion du compteur relatif et en rouge pour la gestion du compteur de position.

Le signal `PWMProportion` correspond au duty cycle du compteur relatif et `PWMProportion_OUT` c'est le signal de duty cycle qui est envoyée au bloc `PWMGenerator`.

`goalPosition`, `enableMotor` et `modeManual` sont des informations qui viennent de `mainController`; `startPWMdown` est la position où le chariot doit commencer à décélérer.

`absCounter` correspond à la position actuelle du chariot qui vient du compteur de position.

La sortie `deltaPWM_18_bit` est une constante qui correspond à la distance d'accélération et décélération du chariot (voir 2.4.1).

`motorOn` c'est le signal envoyé au moteur pour l'allumer ou l'éteindre, ce signal est aussi envoyé au bloc principal `mainController` pour savoir si le moteur est allumé ou éteint.

`leds_test` c'est un bus de 2 bits utilisé pour allumer les LEDs de débogage de la `FSM`.

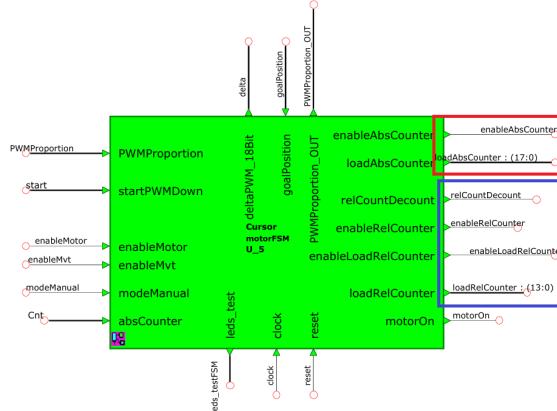


Figure 28: Bloc `motorFSM`, ses inputs et outputs

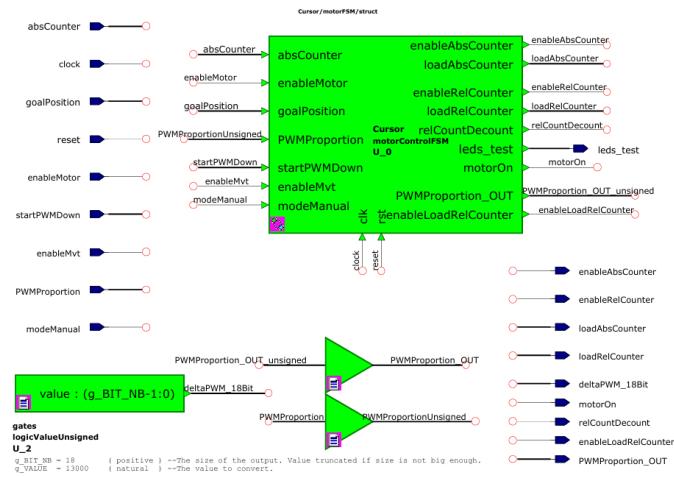


Figure 29: Contenu du bloc `motorFSM`

La subdivision en sous-blocs a été faite pour séparer les transformations de type de données (`std::ulogic_vector` à `unsigned` et viceversa) de la logique extérieure. Comme on le voit dans la figure 29 les seules autres

bloc sont des transformateurs et un bloc de constante (`deltaPWM_18_bit`) qui est mis directement en sortie.

Ce bloc de constante au début entrait aussi dans la FSM mais dans la version finale ceci n'est plus nécessaire mais le bloc est quand même resté là.

La constante contenue dans le bloc est 13'000. Elle représente la distance à accélérer et décélérer (détails dans le chapitre [2.4.1](#)).

La machine d'état du `motorControlFSM` est visible en entier dans l'annexe [16](#), le diagramme dans l'image [30](#) montre le fonctionnement général simplifié de la [FSM](#).

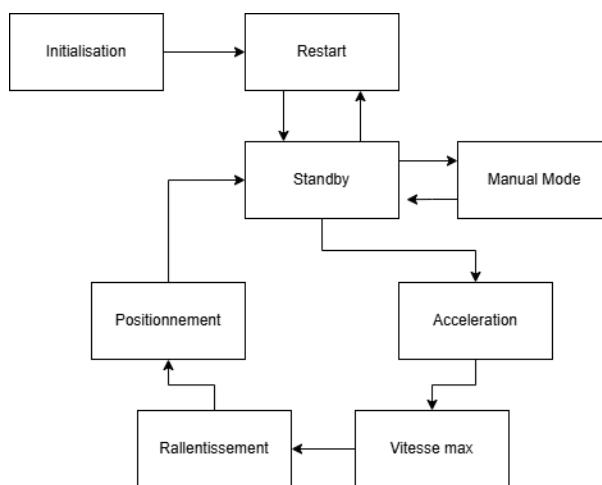


Figure 30: Diagramme du fonctionnement de `motorControlFSM`

Au début la machine démarre dans le bloc `initialisation`, la machine attend que le `mainController` envoie le signal `restart` pour pouvoir passer dans la partie `restart` où le chariot se déplace en arrière jusqu'au senseur reed de gauche (input `enableMvt`).

Vu que `mainController` peut communiquer avec `motorControl` seulement avec `rotation`, `enableMotor` et `goalPosition`, pour envoyer l'info "restart" le `mainController` envoie la position-but 111111111111111111 (le maximum possible), le signal `enableMotor` '1' et `rotation = '1'` (arrière).

Vu que la position 1111111111111111 n'existe pas réellement (le chariot ne peut pas aller au-delà du senseur reed de droite), la [FSM](#) du moteur sait que elle doit faire le `restart`.

Une fois que le chariot est en position de `restart` la machine entre en `standby`, attendant les nouvelles instructions.

Une instruction arrive quand `enableMotor` devient '1', la machine d'état commence l'accélération du moteur en chargeant la vitesse minimale sur le compteur relatif, une fois que la proportion arrive à 255 la machine passe en vitesse maximale et bloque le "enable" sur le compteur relatif.

Grace à le bloc `AddSous` (voir [5.3.4](#)) la machine sait quand commencer la décélération, la machine donne le signal `count/decoount` au compteur relatif pour le faire décompter, une fois que la proportion arrive à la vitesse minimale la machine entre en "positionnement", elle force la vitesse minimale en sortie sur `PWMProportion_OUT` et attend que la position actuelle soit égale à la position-but.

Une fois la position atteinte la machine arrête le moteur et retourne en `standby`.

### 5.3.6 PWM Generator

Le bloc `PWMGenerator` reçoit en entrée seulement le signal duty cycle<sup>4</sup> (`PWMProportion_OUT`) venant de la [FSM](#) et met en sortie le signal de `PWM` (`pwmOut`) qui va au moteur.

Les sous-blocs (figure [31](#)) sont un transformateur de fréquence (de 66MHz à 100kHz), un compteur à 8 bits qui génère la dent de scie et un comparateur entre le signal duty cycle et la dent de scie.

<sup>4</sup>plus clock et reset

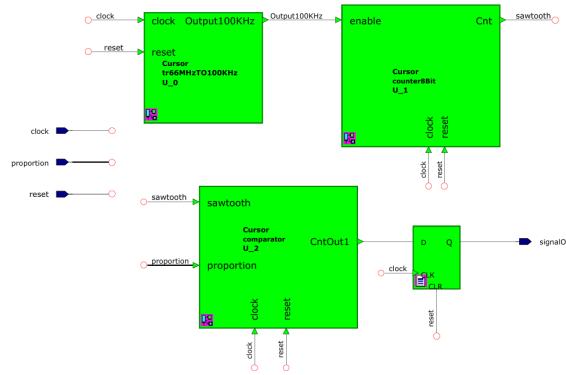


Figure 31: Logique interne du bloc PWMGenerator

Pour la transformation de fréquence, il est nécessaire de diviser la fréquence du clock de 66 MHz et la utiliser comme signal enable du compteur à 8 bits qui génère la dent de scie de 100kHz.

Vu que le compteur a 256 steps, le calcul pour la division de clock est le suivant:

$$N_{div} = \frac{66MHz}{100kHz * 256} = \frac{66000000}{100000 * 256} \approx 2.58 \rightarrow 3 \quad (2)$$

Normalement il est impossible de diviser le clock par un nombre qui est pas une puissance de 2, le système utilisé est donc un compteur de 0 à 2 (3 steps) qui génère un signal enable chaque fois qu'il se trouve à 2.

De cette manière l'enable arrive chaque 3 cycles de clock, donc la fréquence d'enable est de 22MHz et la fréquence de la dent de scie est:

$$f_{dentdescie} = \frac{22MHz}{256} \approx 85.9kHz \quad (3)$$

La fréquence de 85.9kHz est acceptable vu que le but principal était d'éviter des bruits audibles (au-dessus de 20kHz) et de ne pas dépasser les 100kHz pour le switching des mosfets du pont H([3.1.1](#)).

La table [2](#) montre le tableau de vérité utilisé pour créer le compteur.

$Q_1$	$Q_0$	$Q_1^+$	$Q_0^+$
0	0	0	1
0	1	1	0
1	0	0	0
1	1	-	-

Tableau 2: Tableau de vérité pour compteur 0-2

Les équations du circuit obtenues par le tableau sont les suivantes:

$$Q_0^+ = D_0 = \overline{Q_0 \oplus Q_1} \quad (4)$$

$$Q_1^+ = D_1 = Q_0 \quad (5)$$

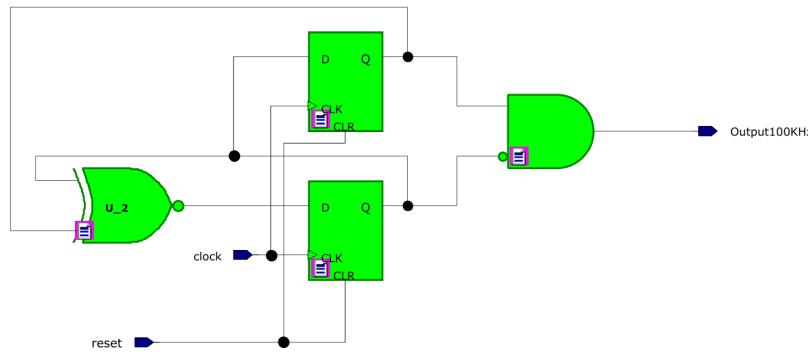


Figure 32: Compteur de 0 à 2

La figure 32 montre la logique du compteur de 0 à 2.

Le signal enable rentre dans un compteur à 8 bits qui se compose de 8 compteur à 1 bit simples<sup>5</sup> mis ensemble dans une boucle "FOR" comme montré dans l'image 33.

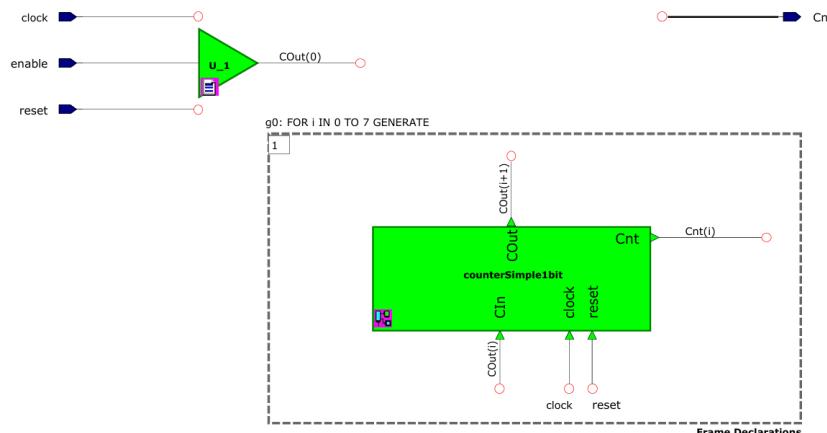


Figure 33: Compteur à 8 bits

La sortie du compteur est directement connecté au bloc **comparator** qui reçoit en entré aussi la proportion de la **PWM**.

Son fonctionnement est le même que le bloc **AddSous**. La seule différence est qu'il travaille avec 8 bits et pas 18 bits(34).

---

<sup>5</sup>Sans la partie de load data et count/decount

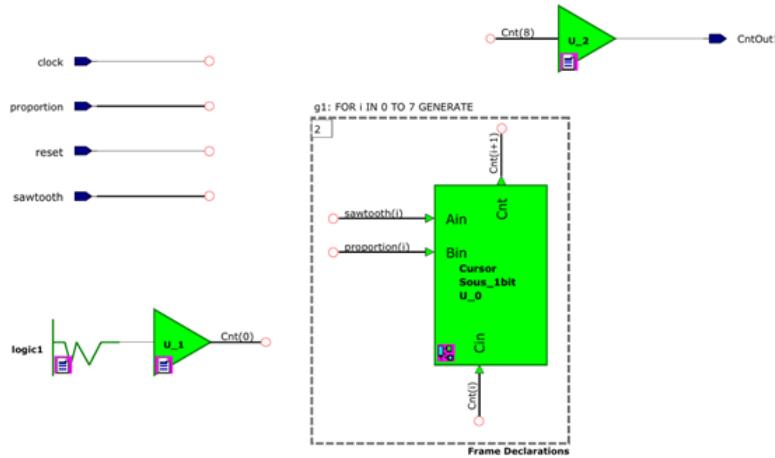


Figure 34: Logique du bloc "comparator"

À l'extérieur de ce bloc, sur sa sortie, une bascule D a été ajoutée, sa fonction est d'enlever les "glitches" qui se passent pendant la soustraction. Pour avoir plus d'informations sur l'utilisation d'une bascule, voir le chapitre [glitches](#).

#### 5.4 Deux registres

Les deux blocs `reg1` et `reg2` sont des registres. Ils ont en entrée le bus `data_in` à 18 bits et le signal `enable` puis en sortie le bus `data_out` à 18 bits.

Un registre a le but de mémoriser des informations. Par la répétition de 18 bascules E, un pour chaque bit des bus, il est possible de mémoriser une valeur quand `enable` est à '0' et de charger une nouvelle valeur quand `enable` est à '1'.

La figure 35 montre la boucle for avec les bascules E.

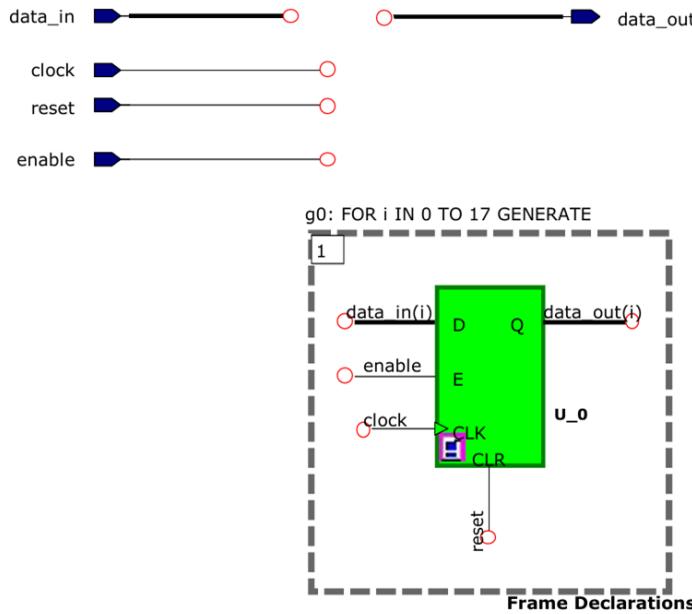


Figure 35: Circuit logique des registres

## 6 Fonctions supplémentaires

Les fonction supplémentaires ajoutés au circuit sont les suivantes:

- Déplacement manuel du chariot
- Sauvegarde de nouvelles positions
- LEDs en fonction de la puissance du moteur
- Écriture de l'état de la machine sur l'écran LCD

### 6.1 LCD

La figure 36 montre l'ensemble des blocs traitant l'écran lcd.

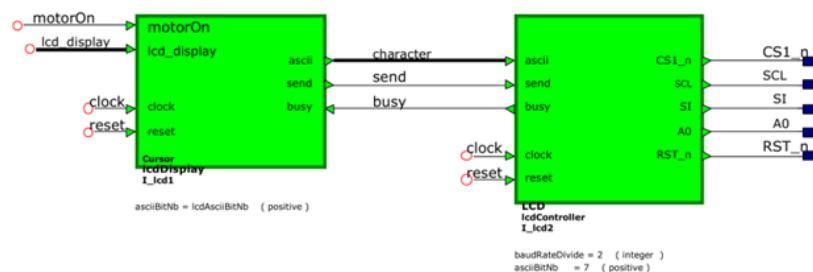


Figure 36: Blocs dédié à l'écran

Le bloc `lcdController` (bloc de droite) était déjà fourni. Son rôle principal est de gérer la couche physique de la communication. Il reçoit le code `ASCII` (bus `character`) et le découpe bit par bit pour le traitement.

Le bloc `lcdDisplay` était déjà créé mais a dû être rempli.

#### 6.1.1 Entrées

- `motorOn`: signal à 1 quand le moteur tourne
- `lcd_display`: un bus (4 bits) provenant de `buttonControlFSM`. Il contient les informations pour sélectionner quel message doit être affiché à l'écran LCD
- `busy`: un signal qui est à 0 quand le bloc `lcdController` peut recevoir le prochain `character` à traiter
- `Clock` et `restart`: horloge et réinitialisation de base

#### 6.1.2 Sorties

- `Character`: bus (8 bits) qui transmet la valeur numérique du caractère à afficher
- `send`: le bloc `lcdController` lira l'entrée `ASCII` que lorsqu'il verra une impulsion sur le signal `send`

Le bloc `lcdDisplay` est une grande machine d'état. Son but est de transmettre les messages à afficher sur l'écran LCD. La figure 37 illustre les actions à effectuer dans une bulle d'état.

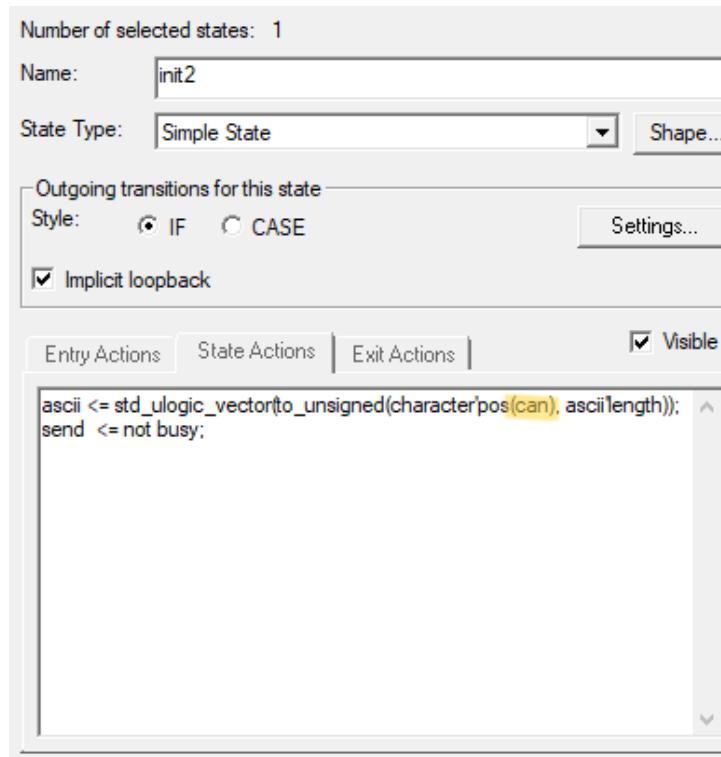


Figure 37: Boule d'état, efface l'affichage

Toutes les bulles d'états ont la même "Action". Seule la valeur entre les parenthèses de `pos`, en jaune en haut, change. 5 valeurs ont été utilisées dans ce projet:

- `stx`: va à la ligne 0 et le caractère 0
- `can`: efface tout l'affichage
- `lf`: retour à la ligne
- `cr`: va au début de la ligne
- "`xx` représente le symbole ASCII à afficher. Il faut la mettre entre parenthèses.

La figure 38 est un exemple d'une ligne d'affichage.

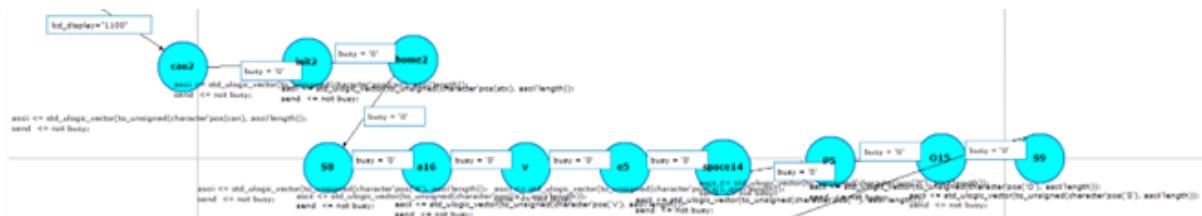


Figure 38: Ligne de texte dans la FSM de l'écran

Dès que la FSM reçoit la valeur `lcd_display = "1100"`, l'affichage est effacé (`can2`). Quand `busy = "0"`, la prochaine bulle peut être effectuée. La position d'écriture (`home2`) est déplacée à la ligne 0 et le caractère 0. Chaque caractère d'une phrase a besoin de sa propre bulle. Donc pour écrire "save POS", il faut 8 bulles.

La documentation explique l'intégralité du domaine avec les autres valeurs disponibles.

La figure 39 regroupe tous les messages à transmettre.

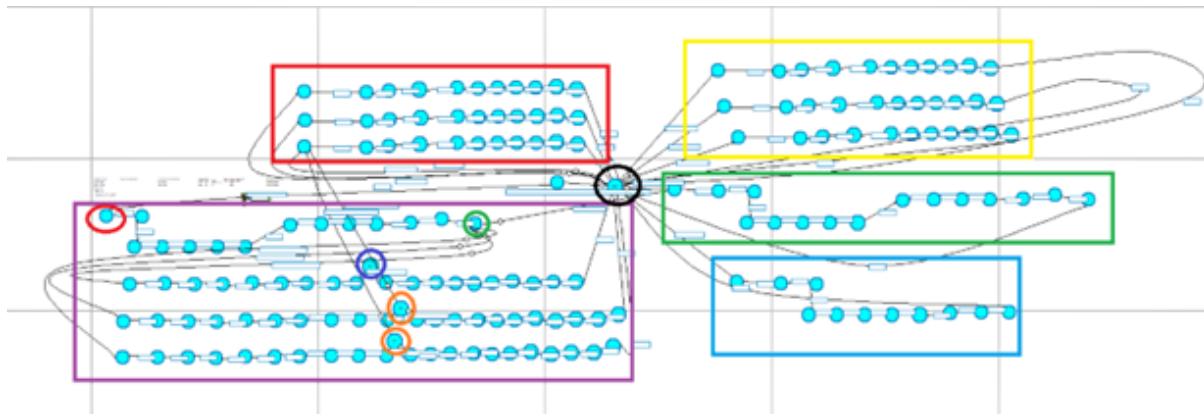


Figure 39: Bloc LcdDisplay, color coded

Il y a 6 grandes parties:

- **Cercle noir:** le pivot de tous les messages. C'est un état où la FSM attend de recevoir un code valide du bus `lcd_display`.
- **Partie violette:** s'occupe du mode automatique. Dès l'alimentation du système, la FSM commence dans le cercle rouge et affiche le message mode AUTO. Il fait aussi deux retours à la ligne et se place au début de la ligne. Au cercle vert, il repasse au cercle noir et attend qu'un des boutons soit sélectionné. Quand un des boutons est sélectionné, la FSM refait la partie expliquée ci-dessus (cercle rouge → cercle vert), puis écrit soit "go to RST", "go to POS1" ou "go to POS2". Il s'arrête et attend dans les états encerclés en bleu et orange. Dès qu'il reçoit le signal que le chariot est bien arrivé à sa position finale, la FSM passe dans le prochain état, efface la deuxième ligne et réécrit soit "RESTART", "POSITION1" ou "POSITION2".
- **Partie verte:** s'occupe du mode manuel. Il affiche "mode MANU" sur la première ligne quand le bouton 4 est poussé.
- **Partie bleue:** s'occupe de la sauvegarde d'une nouvelle position. Il affiche "save POS" quand le bouton 4 a été maintenu pendant plus de 3 secondes.
- **Partie jaune:** s'occupe d'afficher le décompte quand le bouton 4 est maintenu. Il affiche chaque seconde maintenu le message "hold for" avec soit "3", "2" ou "1" à la fin du message.
- **Partie rouge:** s'occupe d'afficher la deuxième ligne du mode manuel. Il affiche "Move left" quand le bouton 2 est pressé, "Move right" quand le bouton 3 est pressé et "At Limit !" si le chariot atteint la limite du parcours. "At Limit !" peut aussi être affiché à partir des pivots (cercles orange de la partie violette) si le chariot se trouve devant les capteurs pendant qu'il se déplace vers la position 1 ou 2.

## 6.2 LEDs

Pour représenter la puissance du moteur sur les 8 leds à côté de l'écran LCD, une **FSM** est utilisé.

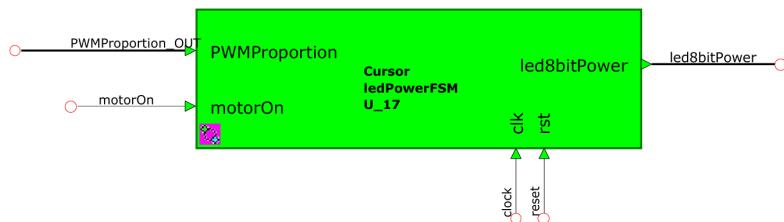


Figure 40: Bloc ledPowerFSM

Les inputs du bloc ledPowerFSM sont: `clock`, `reset`, `motorOn` et `PWMProportion_OUT` (figure 40).

La figure 41 montre la succession des boules dans la FSM, chaque boule allume un led, la machine démarre dans `idle`.

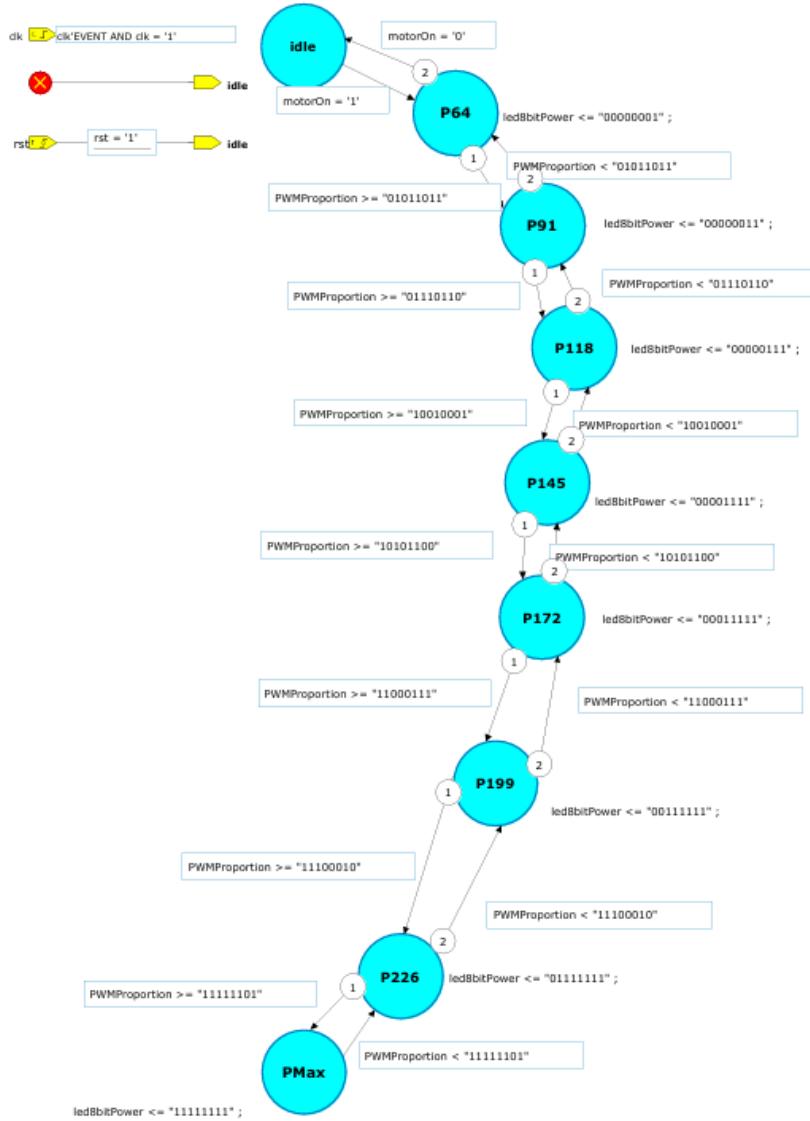


Figure 41: Machine d'état gestion des LEDs

Une fois que le moteur est allumé la machine entre dans la boucle qui allume le premier led. Pour passer à l'état suivant la puissance (`PWMProportion_OUT`) doit dépasser la valeur de 91 (sur 255). Le reste des boucles suivent la même logique, une fois que la puissance passe un certain niveau un nouveau led s'allume. Si la puissance descend la logique fait en sorte que la machine rentre dans un état précédent et un led s'éteint.

Le calcul pour trouver les différentes puissances est le suivant:

$$step = \frac{P_{max} - P_{min}}{8} = \frac{255 - 64}{8} \approx 23.88 \rightarrow 23 \quad (6)$$

L'approximation est vers le bas pour que la dernière puissance ne soit pas supérieure à la puissance maximale.

Pour allumer les leds, un bus à 8 bits est utilisé, le premier bit représente le premier led.

### 6.3 Bouton 4

Le bouton 4 fait deux fonctions:

- Choix entre mode manuel et automatique
- Sauvegarde de la position

Pour distinguer les deux fonctions un countdown est implémenté, si le bouton est allumé pendant plus que 3.5 s (détails dans [6.3.1](#)), le système rentre en mode sauvegarde, sinon le bouton change entre automatique et manuel<sup>6</sup>.

Une fois que le système est en mode sauvegarde, la nouvelle position peut être sauvegardé soit dans le registre 1 en poussant le bouton go1 soit dans le registre 2 en poussant le bouton go2.

---

<sup>6</sup>Pour éviter des boucles infinies, pour aller depuis manuel à automatique le bouton reset doit aussi être activé

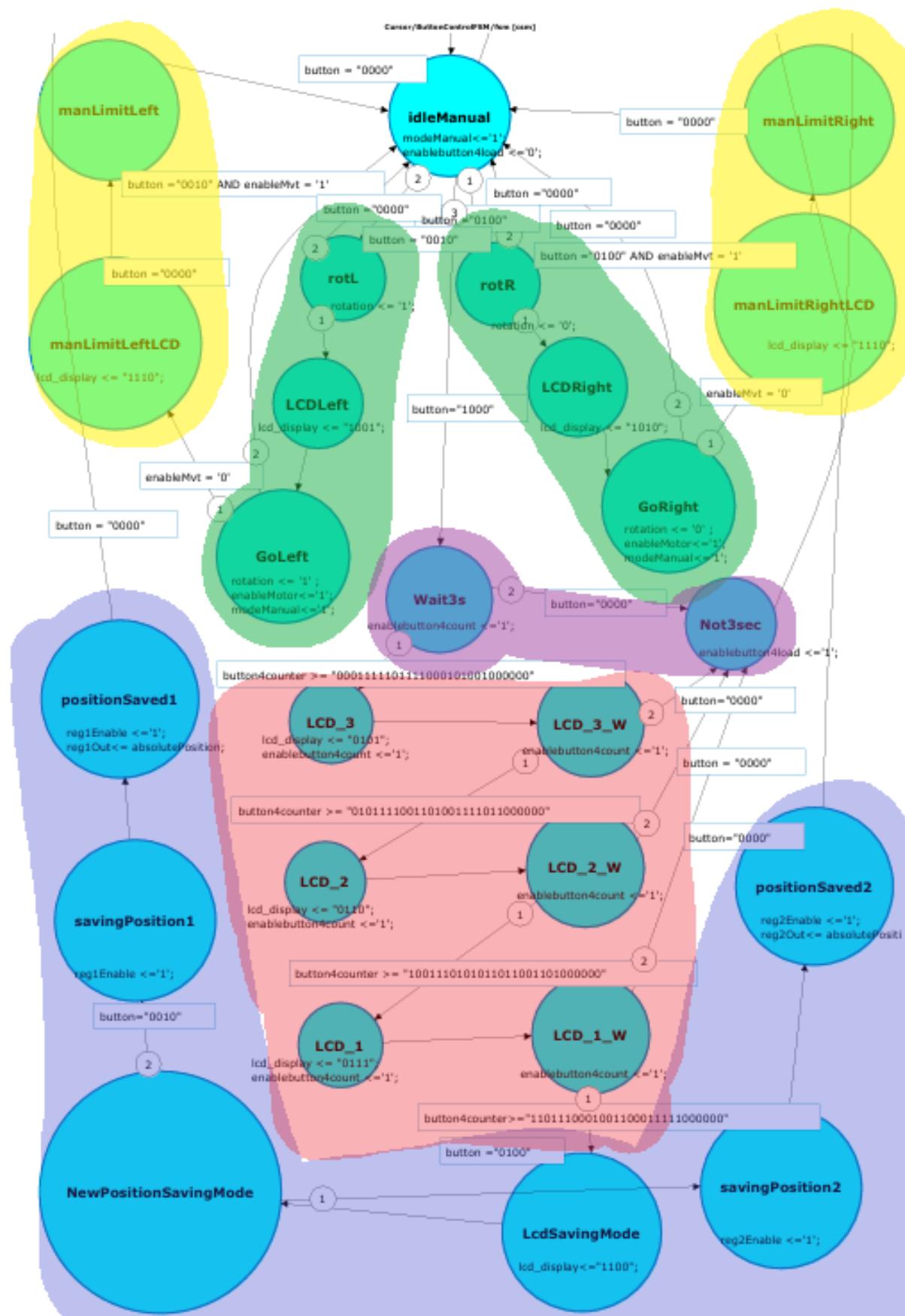


Figure 42: Extrait de la partie de FSM dédié aux fonctions auxiliaires

L'image 42 montre la partie dédiée aux extras de la FSM du `mainController`, les couleurs signifiant:

- Vert: déplacement manuel
- Rouge: partie du countdown, avec l'affichage du countdown sur l'écran
- Violet: retour en mode auto
- Bleu: sauvegarde dans position 1 ou position 2
- Jaune: sécurité, au cas où le chariot arrive aux limites

Pour plus d'info sur la structure de la machine d'état voir l'annexe 6.

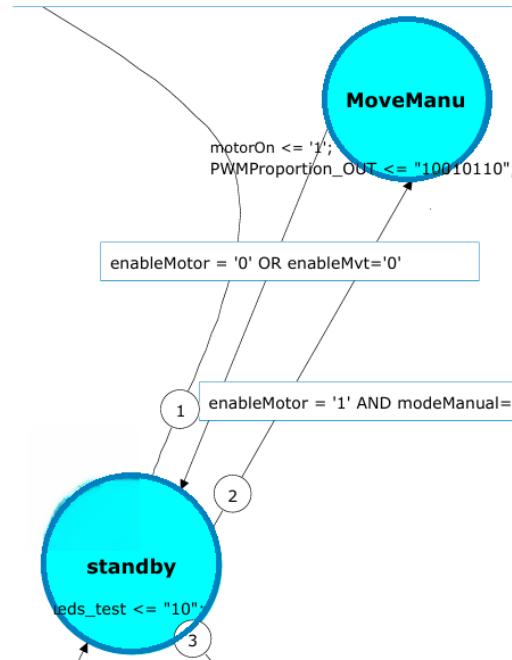


Figure 43: Partie de la FSM du `motorControl` dédiée aux extras

La figure 43 montre la partie dédié au mode manuel du `motorControl`. Le but de cette boucle est de bouger le moteur soit à droite soit à gauche avec une puissance moyenne pour une meilleure précision.

### 6.3.1 Countdown counter

## 7 Simulation et erreurs

### 7.1 Erreurs

Si la simulation VHDL a permis de valider le bon fonctionnement théorique du projet Cursor, l'installation réelle sur la carte a fait apparaître des contraintes techniques liées directement au matériel. Ces différences montrent bien qu'entre le code informatique et le circuit physique, il existe des réalités de terrain que seule la pratique permet de découvrir. Ces divergences s'expliquent par plusieurs phénomènes physiques et logiques:

1. Glitches
2. PWM
3. Latches

#### 7.1.1 Glitches

Lors de la simulation du projet des glitches ont apparu sur des signaux. Le glitch résulte d'un défaut de synchronisation entre les signaux d'entrée (fig. 44).



Figure 44: Glitches: signal accept avec un glitch encerclé en rouge [6]

L'élimination des aléas s'effectue par une resynchronisation de l'étage de sortie via une bascule D. En intégrant le signal accept dans une bascule D cadencé par le clock, le système n'échantillonne la valeur que lors des fronts actifs.

Cette méthode permet de filtrer les pics parasites transitoires, car le glitch, se produisant entre deux fronts d'horloge, n'est jamais capturé par le registre.

#### 7.1.2 PWM

Au niveau de la **PWM** le gros problème est lié à la vitesse minimale. Au début du projet la **PWM** ne fonctionnait pas parce que la vitesse minimale était de 1 sur 255, cela donnait une puissance trop basse pour bouger le moteur (voir chapitre 2.4.2 pour les détails sur ce sujet).

#### 7.1.3 Latches

Une attention particulière a dû être portée à l'inférence de latches. Ces verrous matériels apparaissent lorsque la logique **VHDL** n'est pas exhaustive. Contrairement à la simulation qui peut masquer ce comportement, les latches créent des chemins asynchrones instables sur FPGA, rendant le comportement des FSM erratique. La solution a été de mettre par défaut des valeurs dans la FSM.

Un autre problème au niveau des latches était que le logiciel de programmation de l'**FPGA** voyait pas de variation dans les bascules des registres, donc au lieu de les créer comme éléments de mémoire, le logiciel les imposait comme des 1 et 0 logiques.

## 8 Conclusion

Ce projet avait pour objectif la conception de la logique de contrôle d'un système de déplacement linéaire basé sur un FPGA. L'ensemble du travail a permis de développer une architecture numérique complète capable de gérer le positionnement précis d'un chariot, le contrôle du moteur à courant continu et l'interface utilisateur.

La logique du système repose principalement sur l'utilisation de compteurs itératifs, d'additionneurs à propagation de report et de machines à états finis (FSM). Ces éléments ont permis de gérer la position absolue du chariot, le sens de rotation du moteur ainsi que les différentes phases de déplacement, telles que l'accélération, la vitesse constante et la décélération. L'utilisation d'un signal PWM a assuré un contrôle efficace de la vitesse du moteur tout en respectant les contraintes matérielles imposées par le pont en H.

L'intégration des capteurs de fin de course et de l'encodeur incrémental a permis d'obtenir un retour fiable sur la position du chariot et de garantir un fonctionnement sécurisé du système. De plus, la hiérarchisation du projet en blocs et sous-blocs a facilité la compréhension, la simulation et le débogage de la logique globale.

Les simulations ont joué un rôle essentiel dans la validation du fonctionnement du système et dans l'identification de problèmes tels que les glitches ou les comportements indésirables liés au matériel réel. Les solutions mises en place, notamment l'utilisation de bascules pour la synchronisation des signaux, ont permis d'améliorer la stabilité et la robustesse du circuit.

En conclusion, ce projet démontre qu'une approche structurée et modulaire permet de concevoir un système numérique complexe et fonctionnel. La logique développée répond aux exigences du cahier des charges et constitue une base solide pour le contrôle précis d'un système électromécanique à l'aide d'un FPGA.

### 8.1 Proposition d'améliorations

Le projet fonctionne comme prévu mais il y'a quelque proposition de amélioration qui vaut la pêne de tenir en compte:

- Rotation en fonction de l'encodeur
- Vitesse minimale en fonction de l'encodeur
- 

#### 8.1.1 Rotation en fonction de l'encodeur

Utiliser l'encodeur pour donner le signal de count/decount au compteur de position peut être une bonne manière d'éviter des erreurs.

Vu que les signaux A et B sont décalées dans l'encodeur un bloc qui analyse si le flanc montant de A arrive avant celui de B peut très simplement donner le bon signal de rotation.

Ce système utiliserait un concept d'auto-feedback où le moteur bouge en fonction de l'encodeur, et l'encodeur donne les bonnes informations au bloc qui bouge le moteur.

#### 8.1.2 Vitesse minimale en fonction de l'encodeur

Actuellement la vitesse minimale a été trouvé en "essayant" des valeurs jusqu'à que le moteur bougeait. En réalité il est possible d'augmenter le signal PWMProportion dans le compteur relatif sans le signal enable qui vient du bloc `changeDetector` et utiliser le premier mouvement détecté par l'encodeur pour connaître la vitesse minimale.

## 8.2 Commentaire personal

Globalement en faisant ce projet on a appris beaucoup et on à surtout renforcé nos connaissances en conception numérique.

Sûrement on aurait pu développer un système plus simple qui remplissait toutes les critères de fonctionnement indiquées dans le cahier des charges, mais le défi de créer quelque chose de plus avancée nous à bien intéressé. L'idée du mode manuel et sauvegarde des position était déjà là depuis le début du

projet et ça nous a permis de commencer la création en faisant un système déjà flexible et prêt pour l'implémentation des nouvelles fonctions.

Voir le projet marcher était satisfaisant.

Au niveau du rapport, avoir eu fait un système plus complexe nous a rendu le travail plus long et difficile, surtout vu que l'écriture du rapport nous a pris au total au moins 25 heures chacun.

## Références

- [1] Portescap. *CONTROLLING BRUSHED DC MOTORS USING PWM – OPTIMAL FREQUENCY, CURRENT RIPPLE, AND LIFE CONSIDERATIONS*. [https://www.portescap.com/-/media/projet/automation-specialty/portescap/portescap/pdf/whitepapers/wp\\_controlling\\_brushed\\_dc\\_motors\\_using\\_pwm.pdf](https://www.portescap.com/-/media/projet/automation-specialty/portescap/portescap/pdf/whitepapers/wp_controlling_brushed_dc_motors_using_pwm.pdf). 2019.
- [2] Hannes Sakulin. *Introduction to Field Programmable Gate Arrays*. Tech. rep. International School of Trigger and Data Acquisition 2023, 14 Juin 2023.
- [3] S. Zahno, C. Bianchi, and F. Corhay. *Additionneurs Binaires*. Tech. rep. HES-SO Valais, 2025.
- [4] S. Zahno et al. *Cursor*. Tech. rep. HES-SO Valais, 13 Janvier 2025.
- [5] S. Zahno et al. *Etats logiques*. Tech. rep. HES-SO Valais, 25 août 2022.
- [6] S. Zahno et al. *Introduction aux outils EDA*. Tech. rep. HES-SO Valais, 2025.

## Glossaire

**mosfet** Transistor à effet de champ métal-oxyde-semiconducteur, un type de transistor utilisé pour amplifier ou commuter des signaux électroniques.. [8](#)

## Acronymes

**ASCII** American Standard Code for Information Interchange. [29](#), [30](#)

**FPGA** Field Programmable Gate Array. [3](#), [8](#), [36](#)

**FSM** Finite State Machine. [10](#), [17](#), [18](#), [20](#), [23–25](#), [30–32](#), [35](#), [40](#)

**LCD** Liquid Crystal Display. [11](#), [14](#), [29](#), [31](#)

**PCB** Printed Circuit Board. [8](#)

**PWM** Pulse Width Modulation. [4–8](#), [20](#), [25](#), [27](#), [36](#), [39](#)

**VHDL** VHSIC Hardware Description Language. [3](#), [11](#), [36](#)

## Liste des figures

1	Déplacement du curseur [4]	5
2	Fonctionnement de PWM [4]	6
3	Schématisation de la PWM	6
4	Les composants du curseur	8
5	Extrait du datasheet du pont h	9
6	Extrait du circuit complet (voir pdf complet dans l'annexe nr.1)	10
7	Logique de haut niveau	11
8	Logique de un compteur 1-bit	12
9	Fonctionnement d'un additionneur à propagation de report	12
10	Logique du bloc "buttonManager"	13
11	Inputs et outputs de "mainController"	14
12	Bloc mainController, transformateur unsigned →std_ulogic_vector	15
13	Bascule "E" pour "glitches"	15
14	Logique du bloc "Rotation_Sous"	16
15	Initialisation à 1 dans le bloc Rotation_Sous	16
16	Rotation_Sous, transfère du bit 19	16
17	Bloc de soustraction simplifié	17
18	Bloc ButtonControlFSM avec ses inputs et outputs	17
19	Partie obligatoire de la FSM, color coded	18
20	Button control FSM, diagramme de fonctionnement	19
21	Bloc motorControl - inputs et outputs	20

22	Logique du bloc change detector . . . . .	21
23	Logique du bloc relCounterPWM . . . . .	21
24	Boucle FOR pour le compteur relatif . . . . .	22
25	Compteur de position . . . . .	22
26	Logique du bloc "addSous" . . . . .	23
27	Dessin explicatif du fonctionnement de addSous . . . . .	23
28	Bloc motorFSM, ses inputs et outputs . . . . .	24
29	Contenu du bloc motorFSM . . . . .	24
30	Diagramme du fonctionnement de motorControlFSM . . . . .	25
31	Logique interne du bloc PWMGenerator . . . . .	26
32	Compteur de 0 à 2 . . . . .	27
33	Compteur à 8 bits . . . . .	27
34	Logique du bloc "comparator" . . . . .	28
35	Circuit logique des registres . . . . .	28
36	Blocs dédié à l'écran . . . . .	29
37	Boule d'état, efface l'affichage . . . . .	30
38	Ligne de texte dans la FSM de l'écran . . . . .	30
39	Bloc LcdDisplay, color coded . . . . .	31
40	Bloc ledPowerFSM . . . . .	31
41	Machine d'état gestion des LEDs . . . . .	32
42	Extrait de la partie de FSM dédié aux fonctions auxiliaires . . . . .	34
43	Partie de la FSM du motorControl dédiée aux extras . . . . .	35
44	Glitches: signal accept avec un glitch encerclé en rouge [6] . . . . .	36

## Liste des tableaux

1	État actuel et suivant de l'encodeur . . . . .	21
2	Tableau de vérité pour compteur 0-2 . . . . .	26

## Liste des annexes

1	Full circuit . . . . .	43
2	Top level logic . . . . .	44
3	Button manager . . . . .	45
4	Main controller . . . . .	46
5	Button control FSM . . . . .	47
6	Button control FSM extras . . . . .	48
6	Button control FSM extras . . . . .	49
7	Counter 28 bits . . . . .	50
8	Rotation sous . . . . .	51
9	Sous 1 bit . . . . .	52
10	Motor control . . . . .	53
10	Motor control . . . . .	54
11	Absolute counter . . . . .	55
12	Counter 1 bit . . . . .	56
13	Delta PWM . . . . .	57
14	Change detector . . . . .	58
15	Motor FSM output . . . . .	59
16	Motor FSM logic . . . . .	60
17	PWM generator . . . . .	61
18	Comparator . . . . .	62
19	Counter 8 bit . . . . .	63
20	66 MHz to 100 kHz transformator . . . . .	64
21	Relative counter PWM . . . . .	65
22	Counter 13 bit . . . . .	66
23	Right shifter . . . . .	67
24	All LEDs bus creator . . . . .	68

25	LED power FSM	69
26	Register logic	70

## Validation et Approbation

Ce document intitulé:

**DiD Labs projects**

a été écrit par:

**Mattia Astori**

---

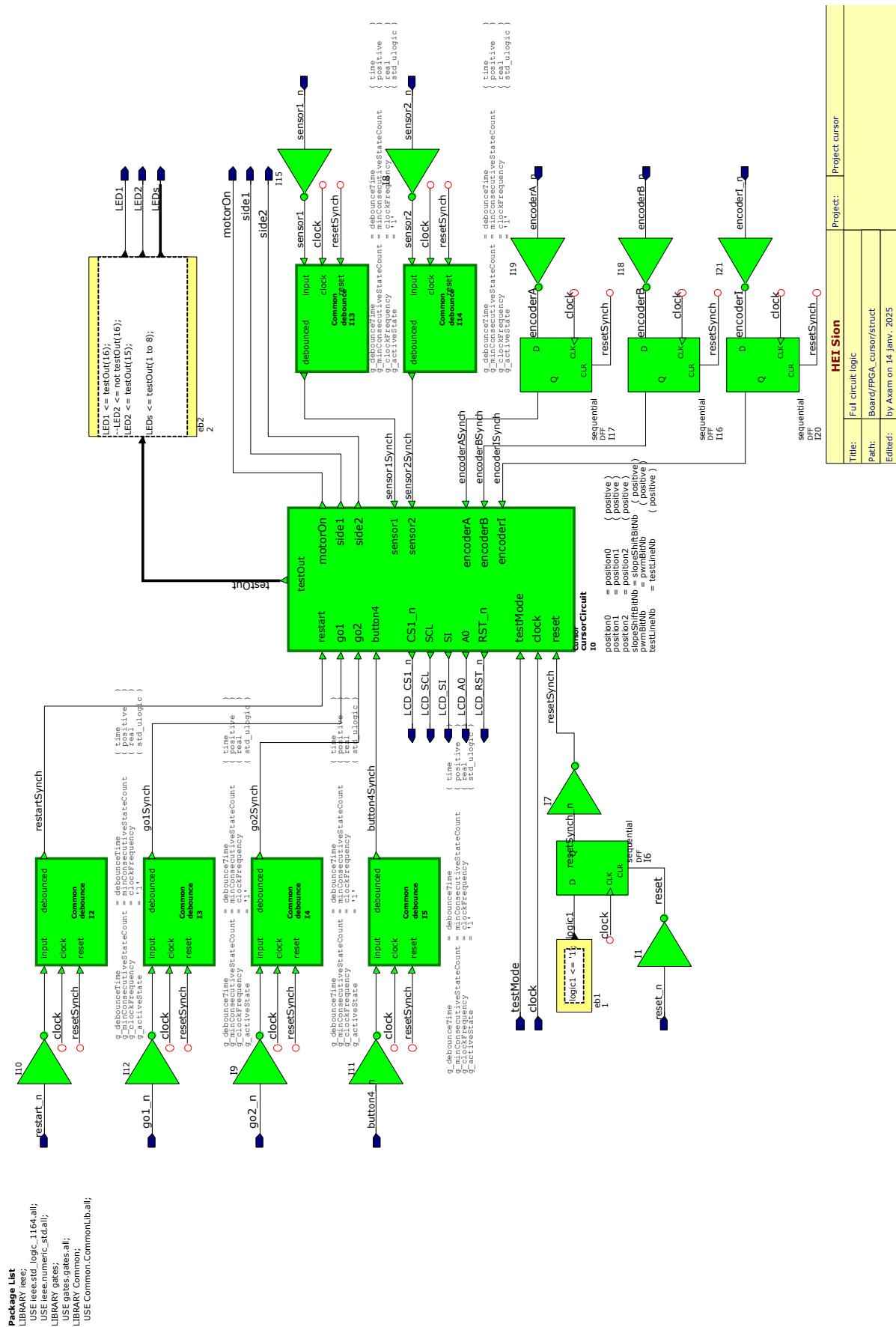
**Alex Crestin**

---

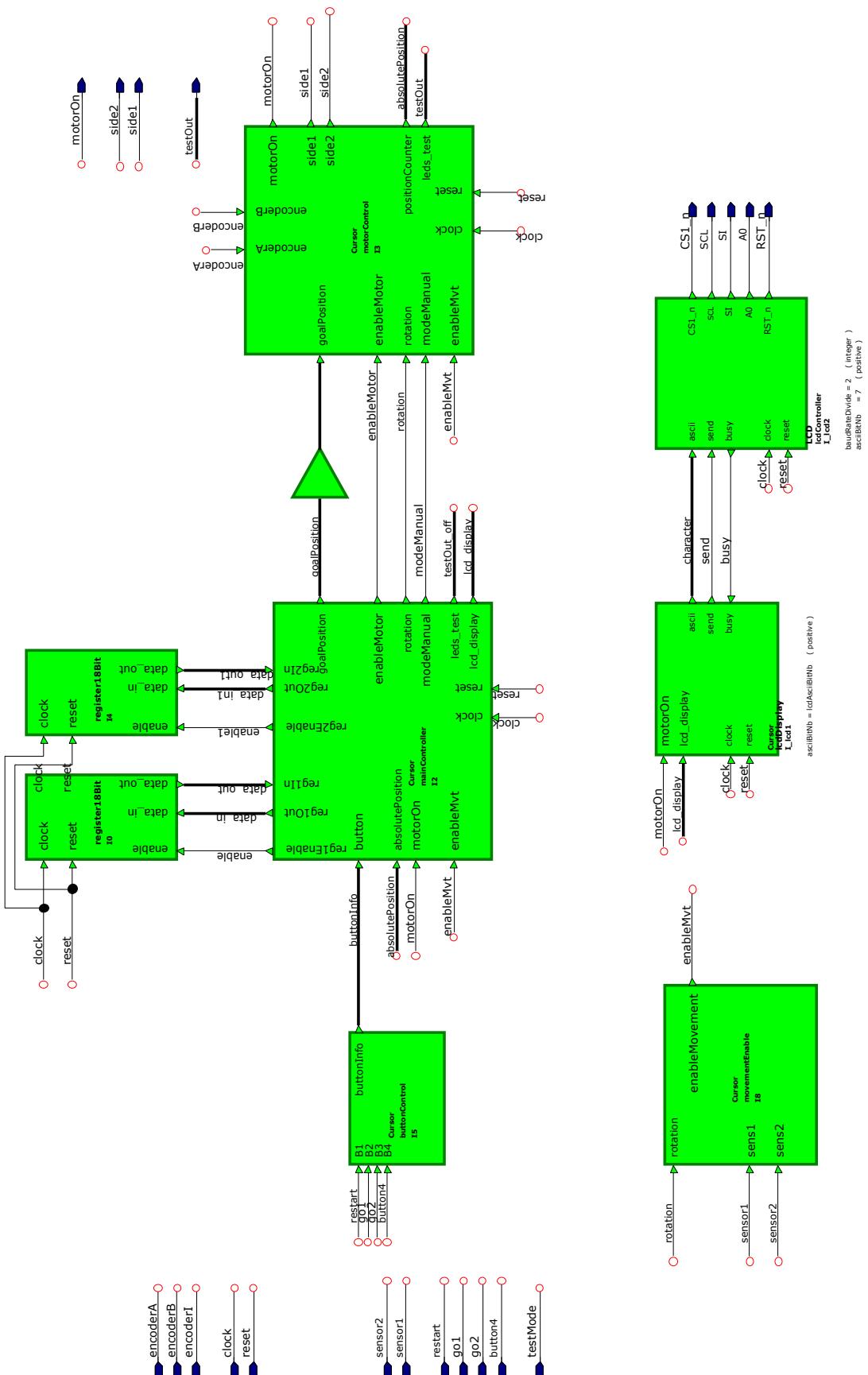


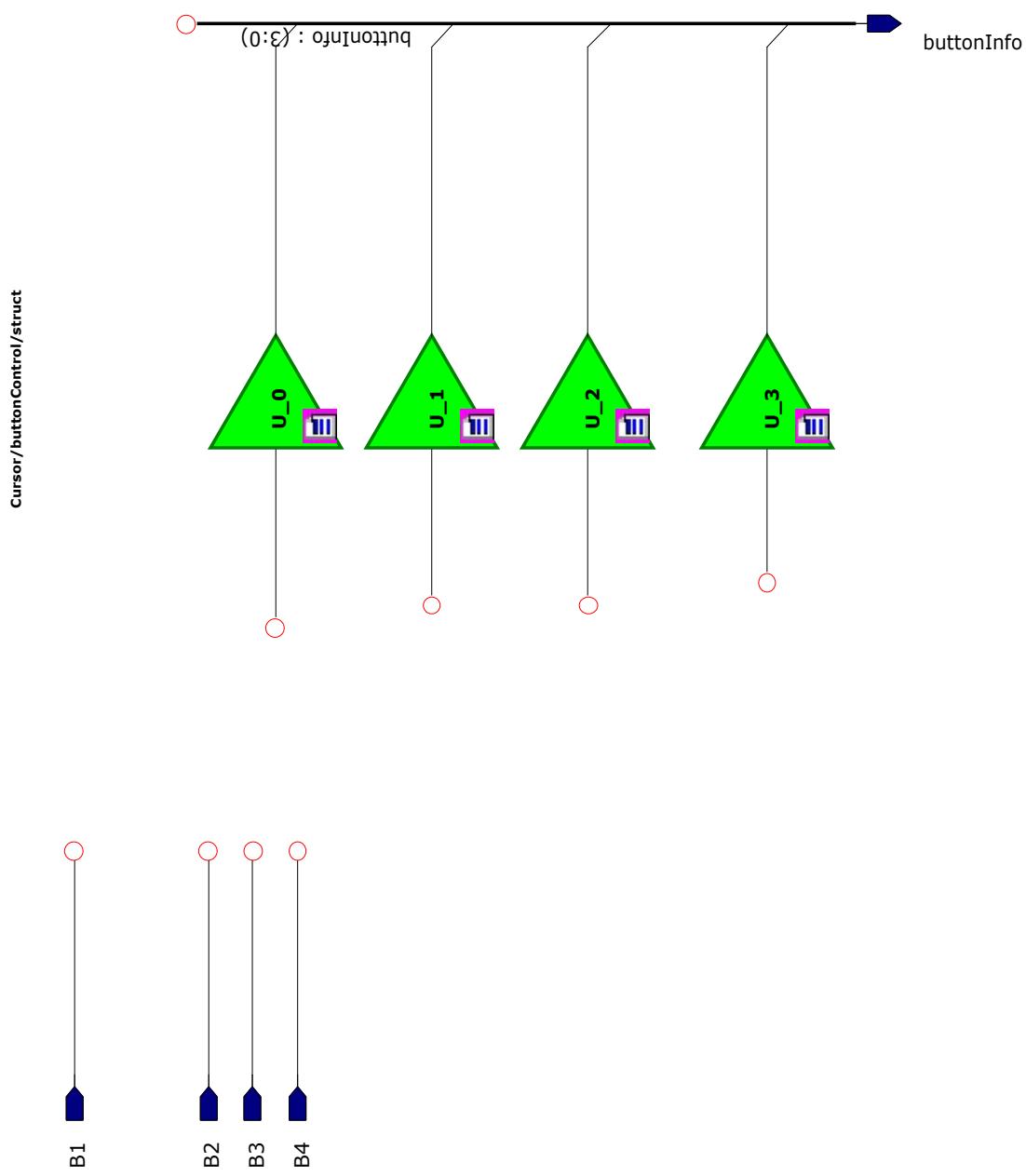
Date: \_\_\_\_\_

## Annexe nr.1 - Full circuit

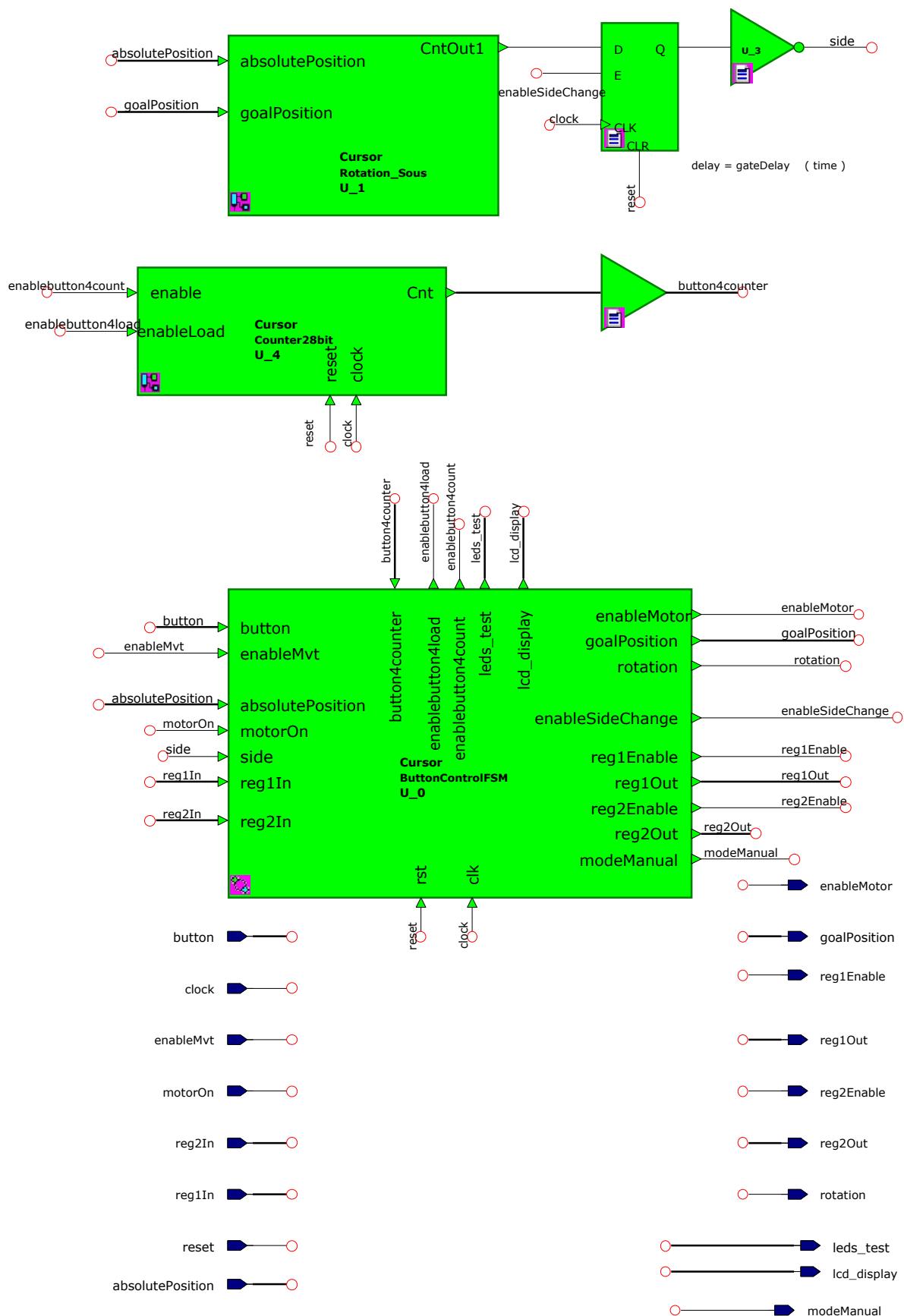


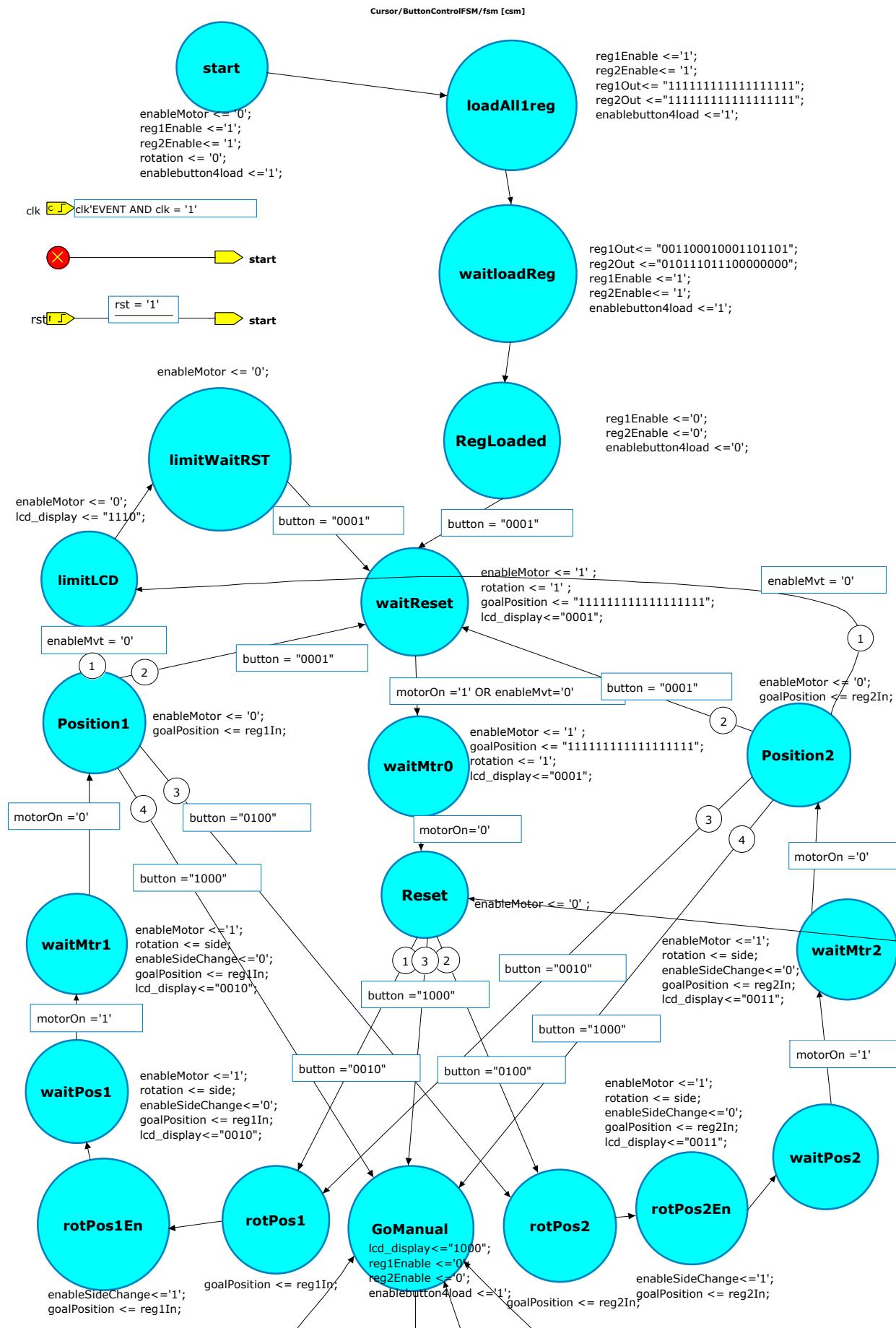
Annexe nr.2 - Top level logic

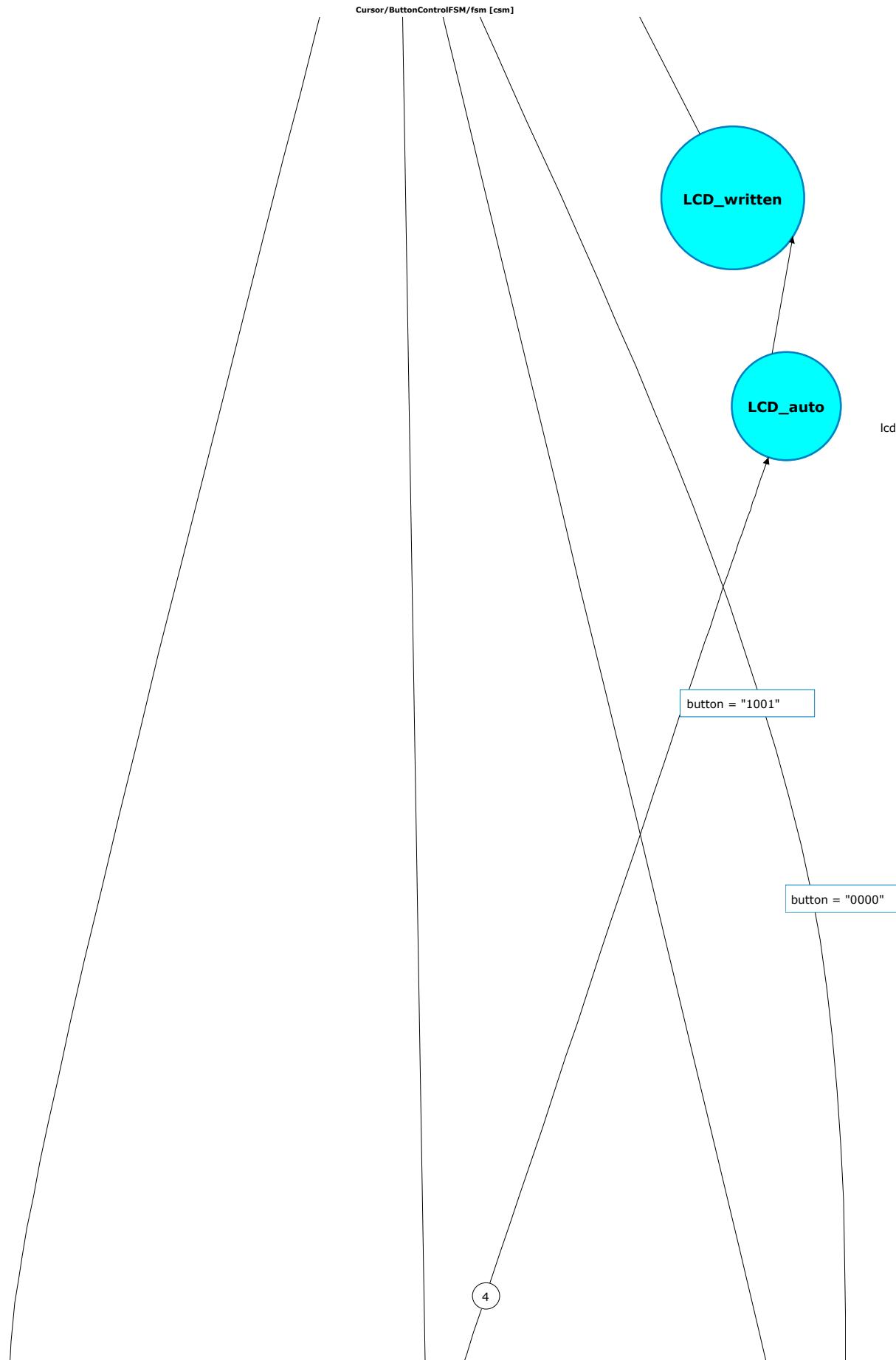




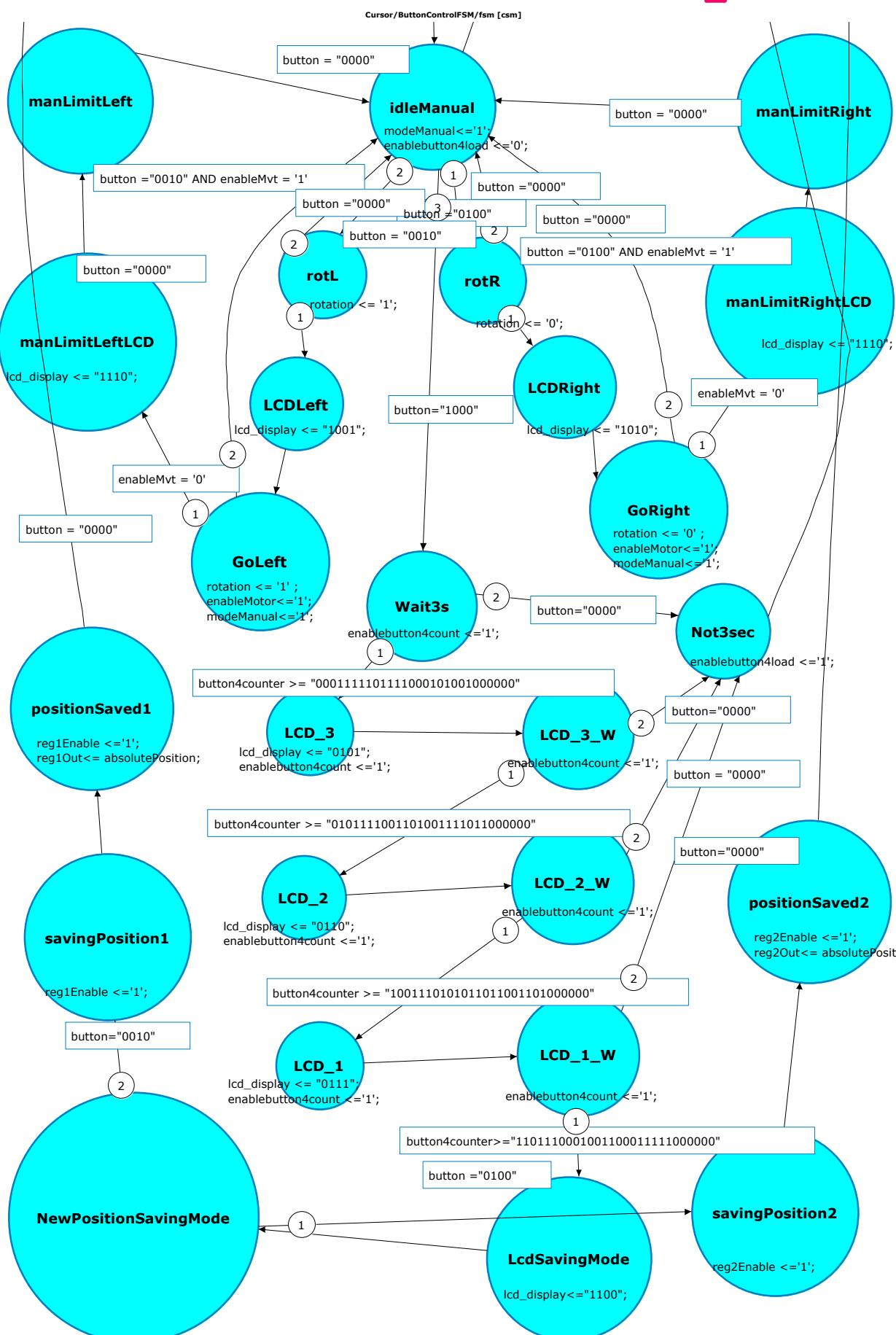
Cursor/mainController/struct

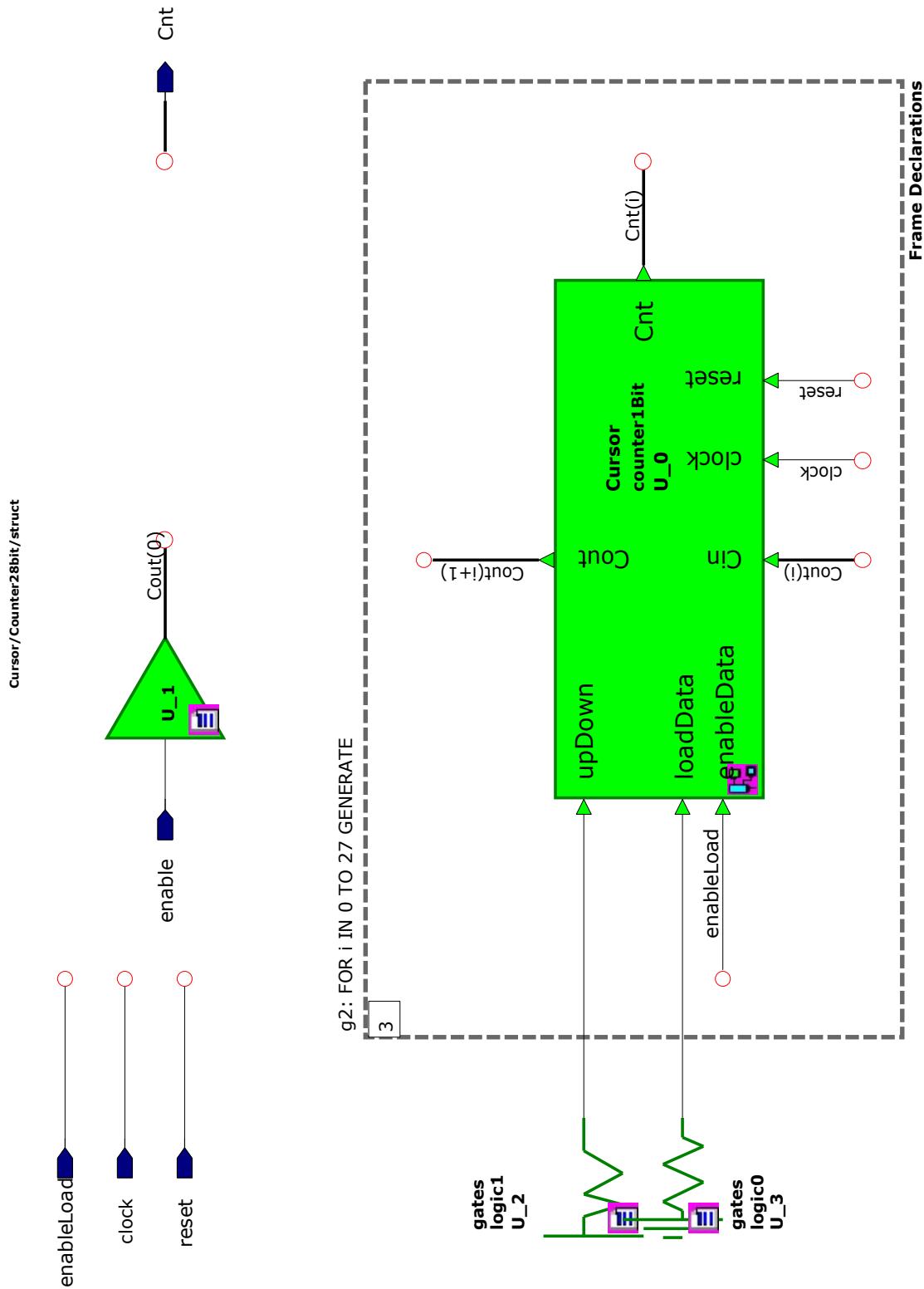


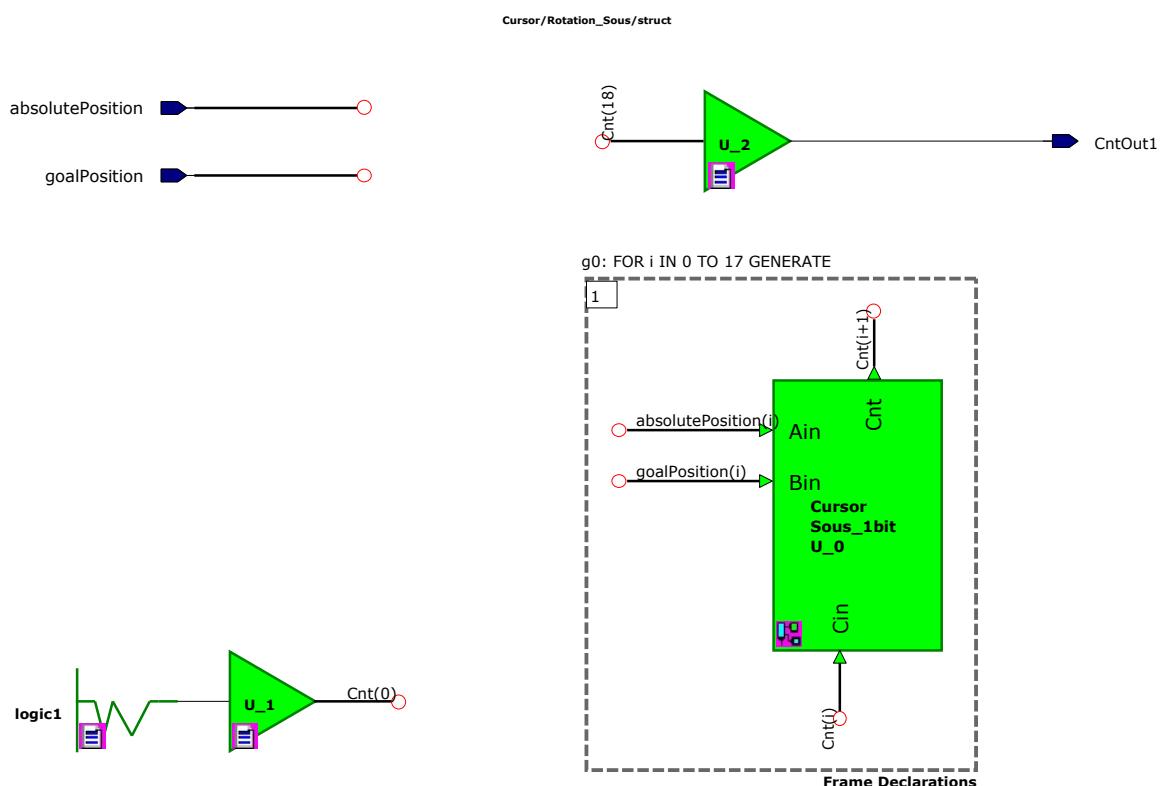




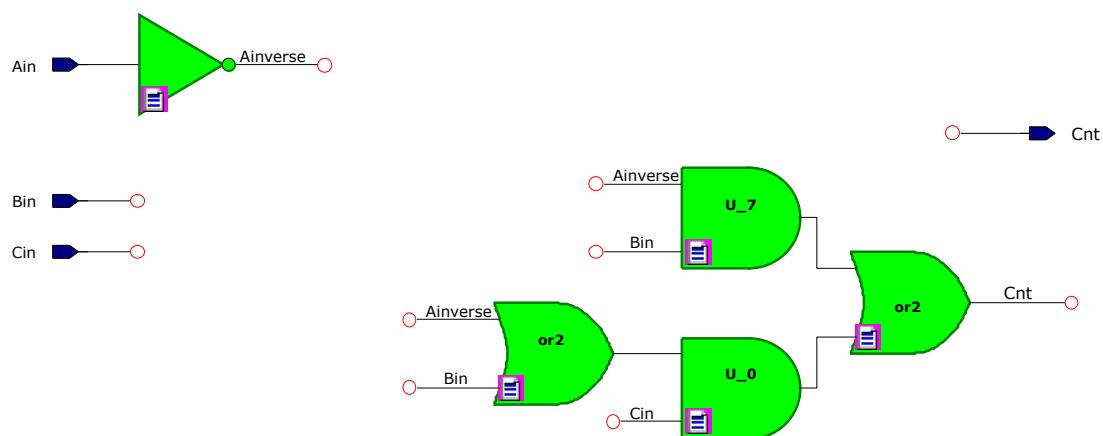
Annexe nr.6 - Button control FSM extras

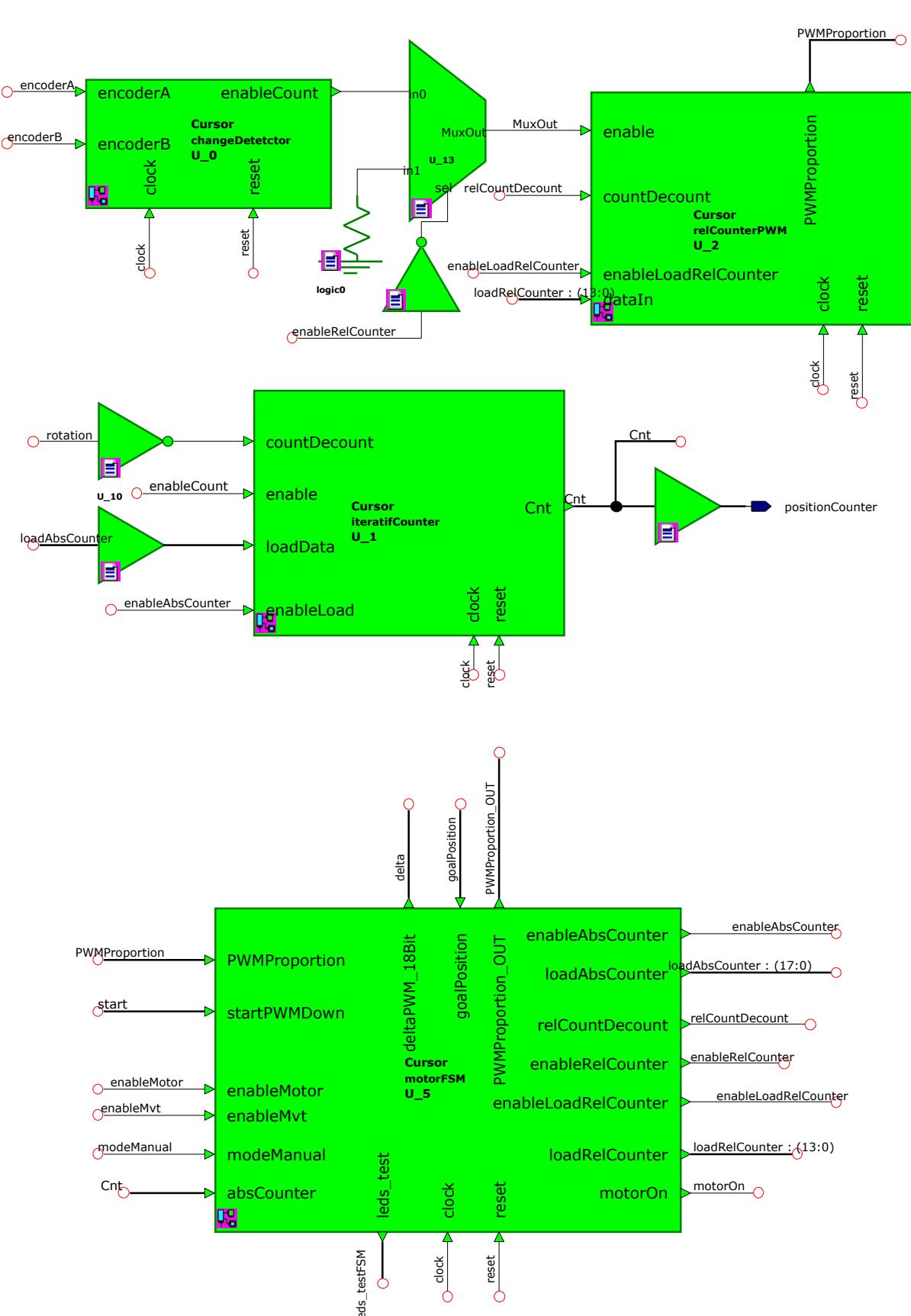


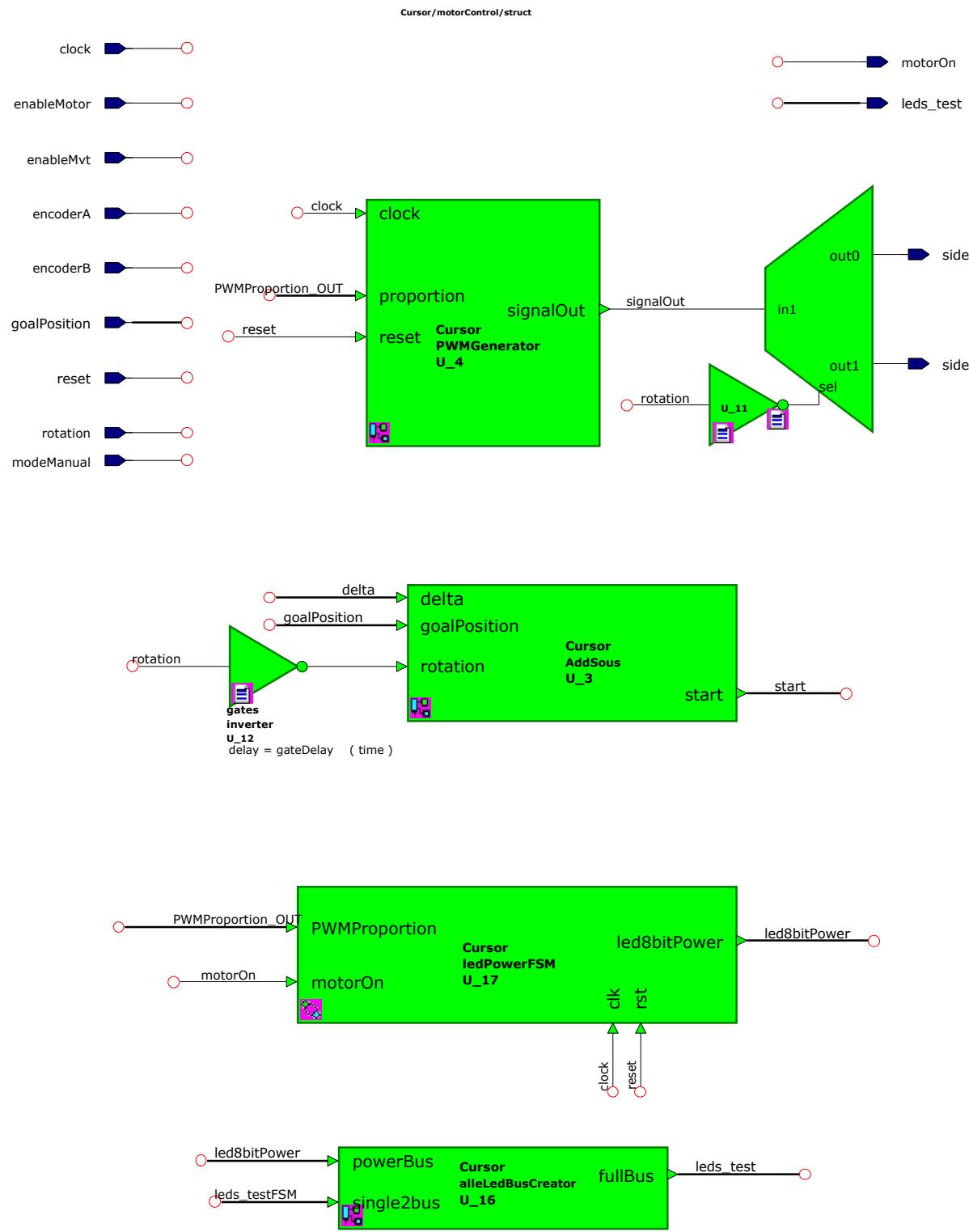


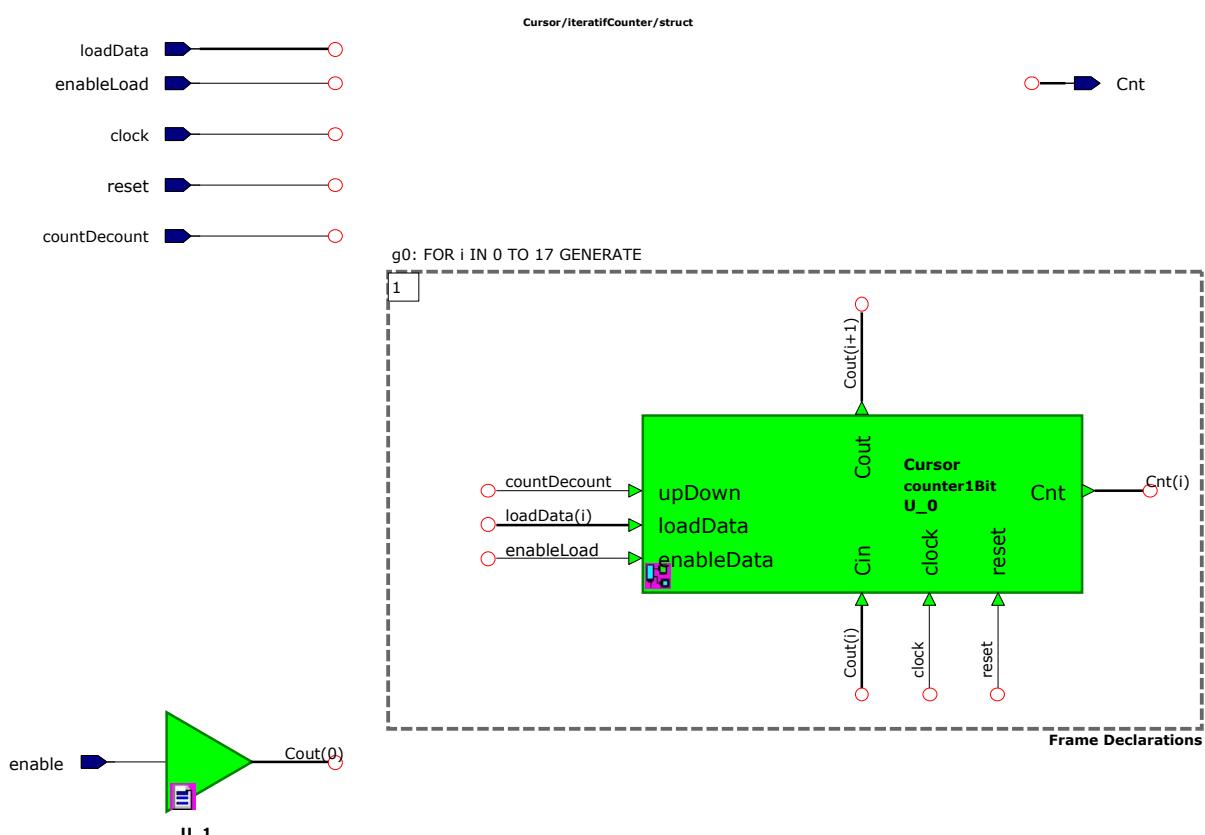


Cursor/Sous\_1bit/struct

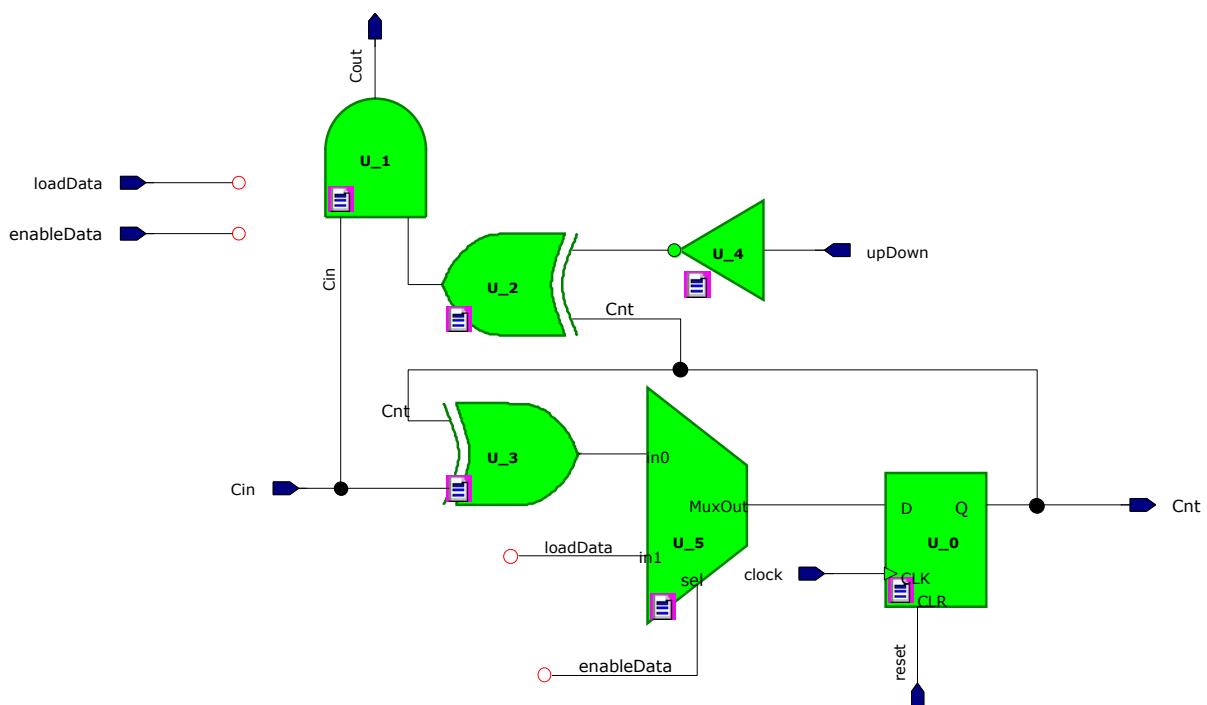


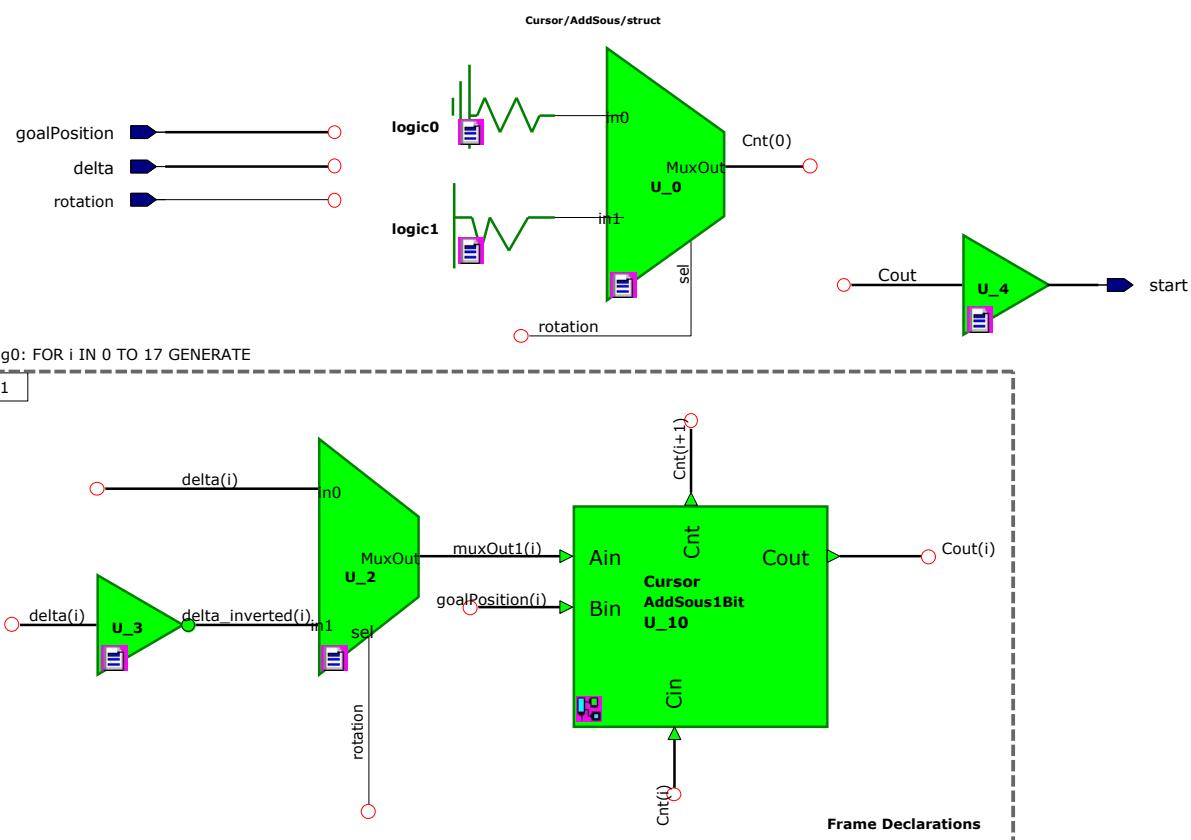


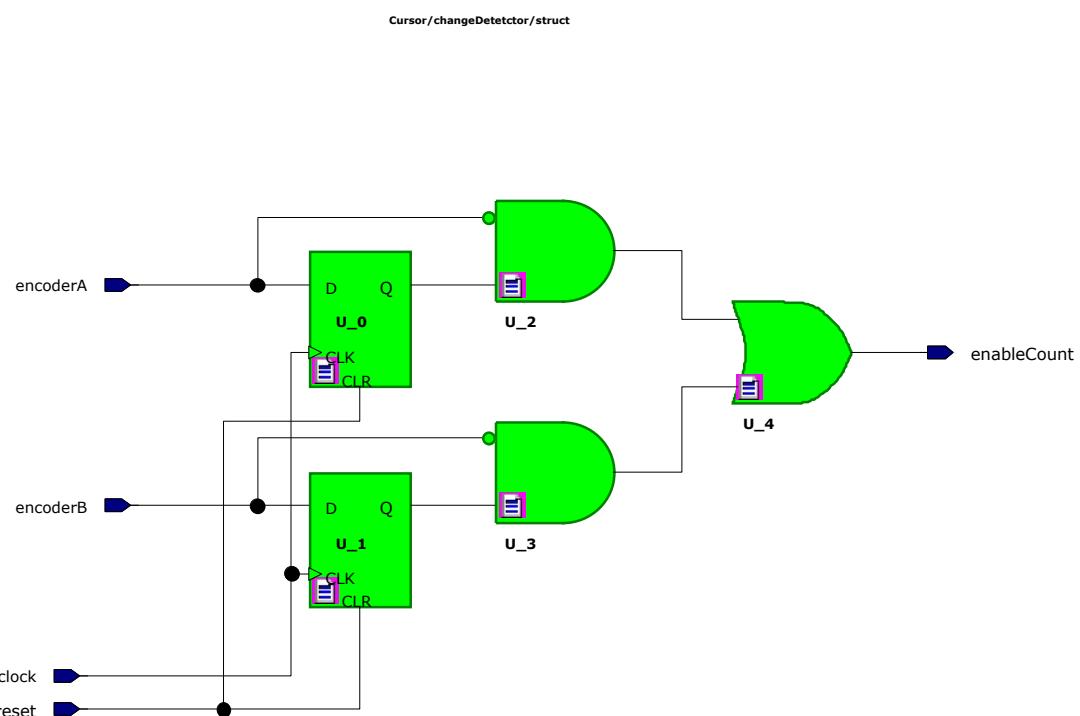


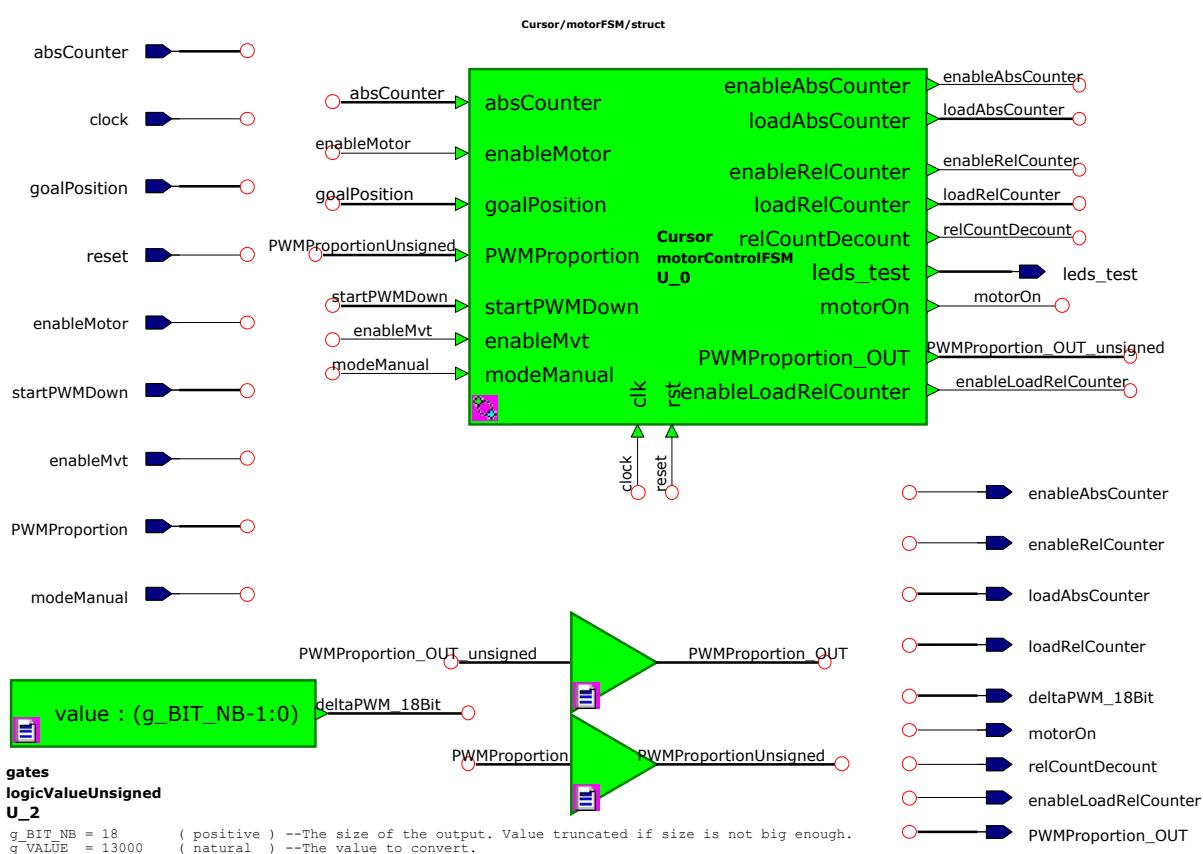


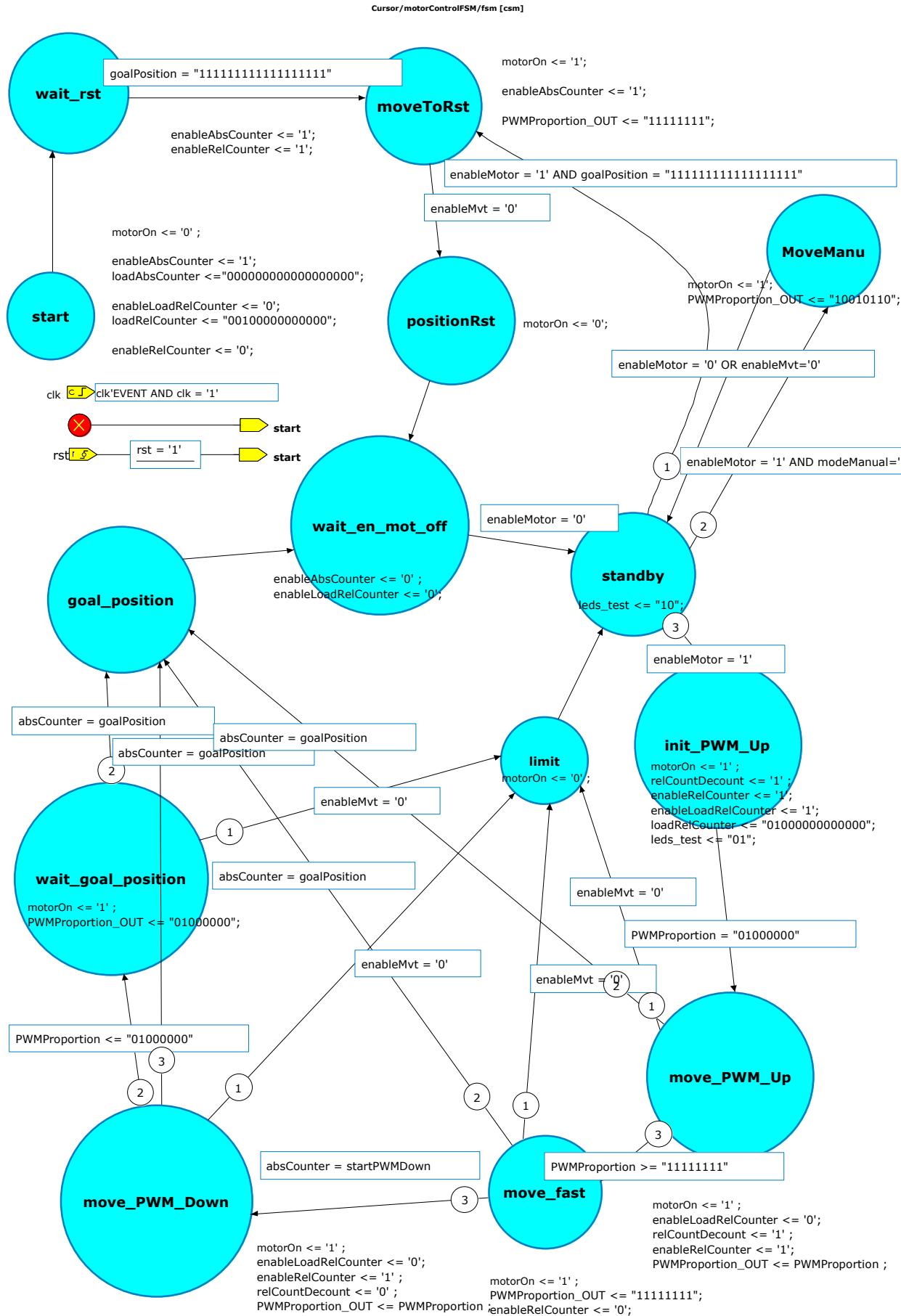
Cursor/counter1Bit/struct

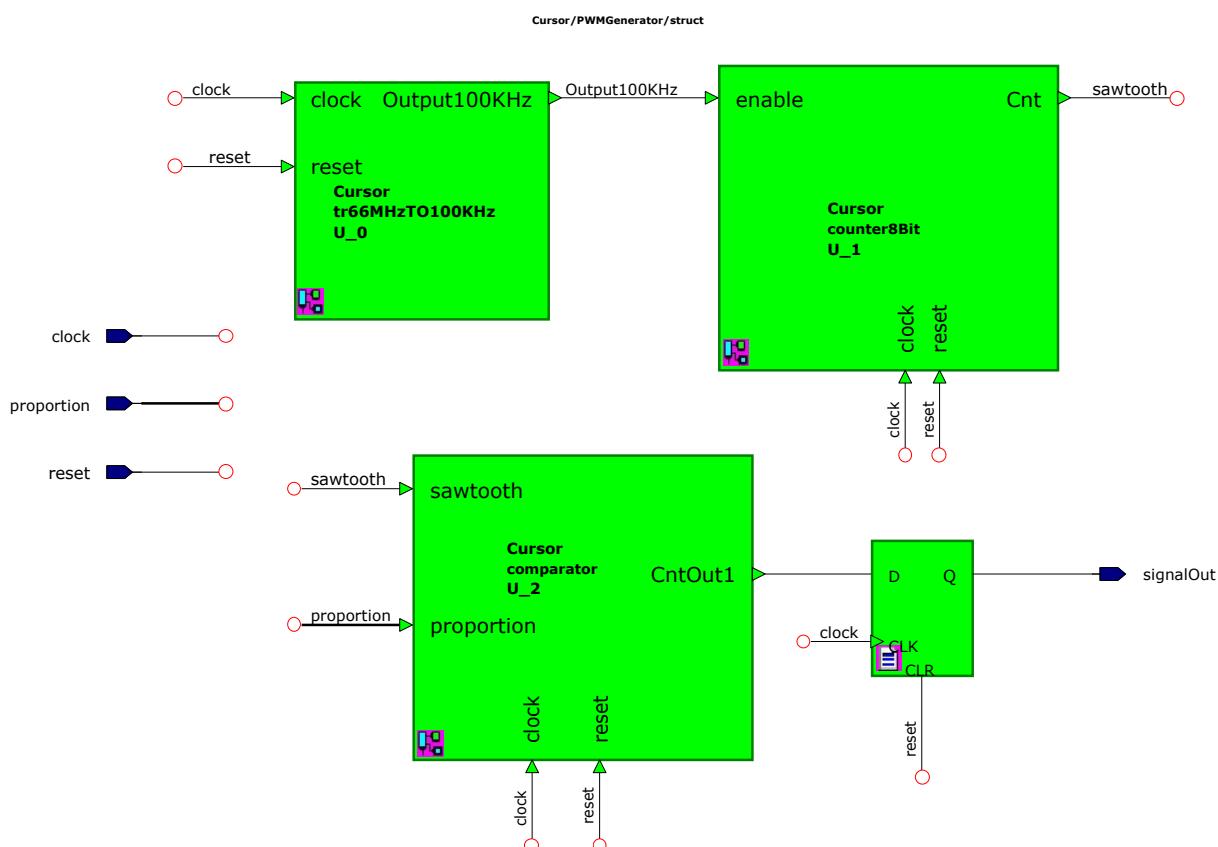


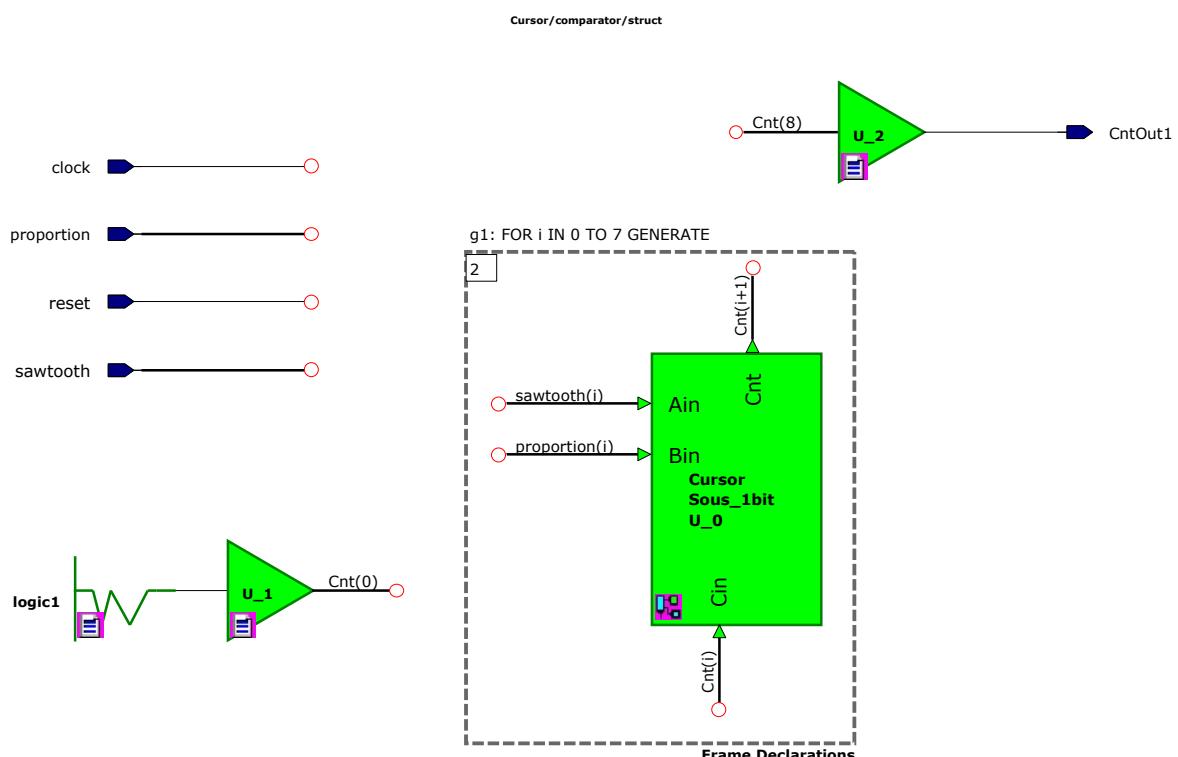


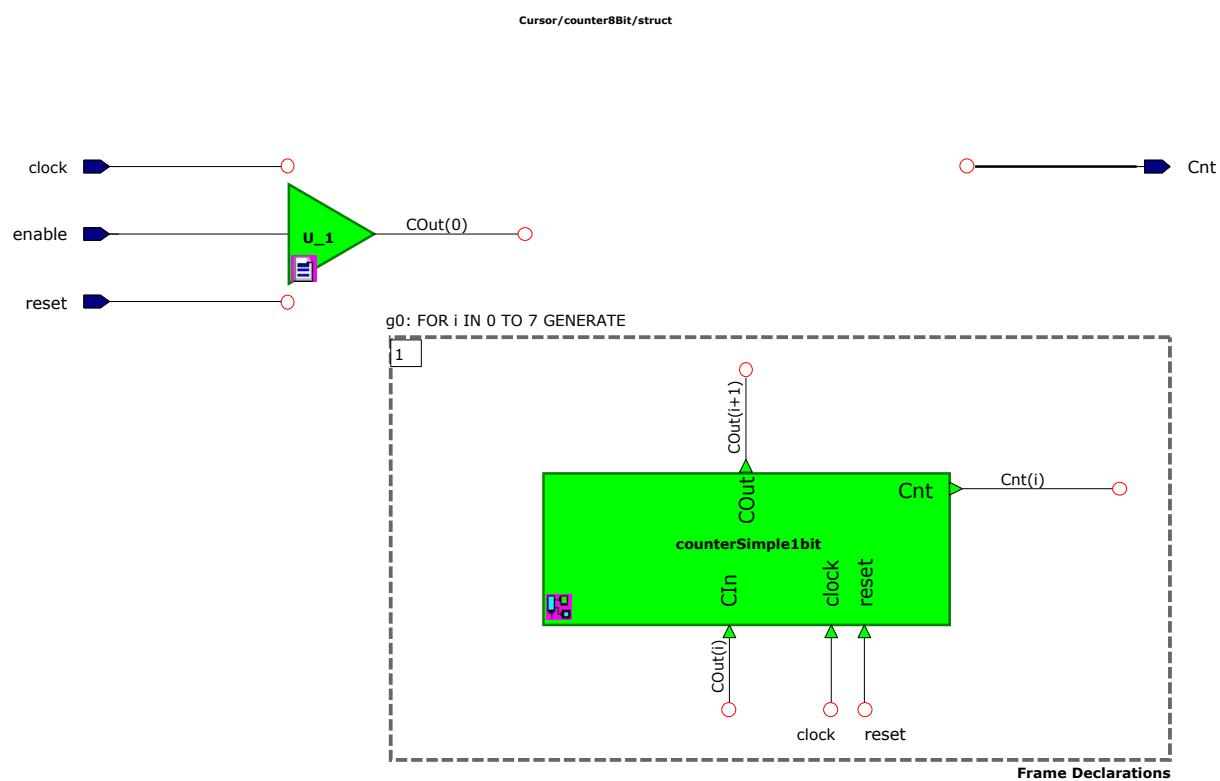












Cursor/tr66MHzTO100KHz/struct

