Grenoble INP phelma

Démodulation de signaux par un système intégré

Année 2020-2021
Préorientation SEI

1 Préambule

Avant de donner de nombreuses explications techniques et des directives pour effectuer le projet, il est extrêmement important de comprendre le contexte de ce travail. Aujourd'hui, les systèmes intégrés se trouvent partout dans notre environnement qu'ils soient visibles ou non. Leurs applications se retrouvent dans des systèmes aussi variés que la téléphonie, l'informatique, l'automobile, l'avionique, le médical, les transports, ...Une des caractéristiques de ces systèmes intégrés est qu'ils sont en général constitués d'une puce matérielle mais aussi de logiciels.
Ce que nous entendons par systèmes intégrés est donc en fait constitué d'une puce composée d'éléments matériels (circuits d'amplification analogique, filtres, circuits numériques de traitement, microprocesseurs) et de briques logicielles (drivers, systèmes d'exploitation, logiciels applicatifs). Ainsi un système intégré est en général un dispositif complexe. Dans le cadre de ce projet de préorientation, nous réduirons nos prétentions en termes de circuits complexes mais nous essaierons de vous donner un aperçu d'un petit système (presque complètement intégré) comportant un « front-end » analogique, de la circuiterie numérique et un processeur très simple. L'objectif est de réaliser un petit système de démodulation de signaux ne faisant appel à quasiment aucun pré requis.

2 Présentation du système de démodulation et de son architecture

2.1 La modulation FSK

L'objectif du projet est de réaliser un dispositif de démodulation d'un signal numérique comprenant un filtre analogique et un processeur implémenté dans un composant programmable. La modulation utilisée est de type FSK (Frequency Shift Keying). Le principe de cette modulation est très simple. Les valeurs numériques 0 et 1 sont codées par des fréquences différentes : le 0 est codé par une fréquence f_0 et le 1 par une fréquence f_1. En considérant que $f_1 = 5 * f_0$ alors le signal représenté sur la figure 1 est porteur de la séquence d'information binaire 1010.

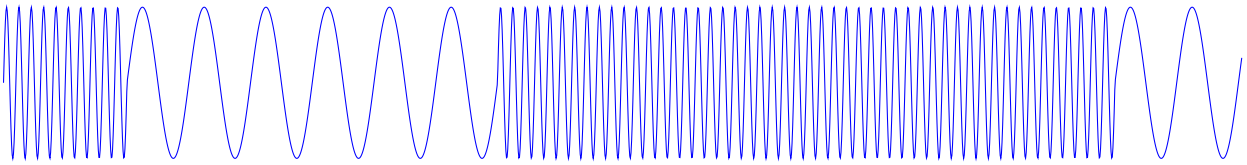

FIGURE 1 – Exemple de signal numérique modulé en FSK

2.2 L'architecture du système

Afin de réaliser ce système de démodulation nous allons mettre en oeuvre une chaîne de traitement numérique du signal dont la compréhension ne nécessite pas de pré-requis. Celle-ci est présentée sur la figure 2 et les différents signaux en jeu sur la figure 3.

La première étape consiste à utiliser un filtre passe-bas pour créer une modulation d'amplitude sur le signal. En effet, la fréquence f_0 (la plus basse) de notre modulation FSK ne sera pas atténuée par le filtre passe-bas alors que la fréquence f_1 (la plus haute) le sera. Ainsi en sortie du filtre, le signal est modulé en fréquence, mais aussi en amplitude. Le signal est ensuite numérisé par un CAN. Une fois dans le domaine numérique, il est traité par un processeur appelé nanoprocesseur dont le programme détecte si l'amplitude du signal correspond à une zone où la fréquence est f_0 (c'est-à-dire un 0) ou f_1 (c'est-à-dire un 1). Pour cela, si le signal numérisé par le CAN est supérieur à un seuil réglé par l'utilisateur, un compteur interne est maintenu à une valeur haute ; lorsque ce signal passe en dessous du seuil, il est décrémenté périodiquement. Le couple $(valeur\ haute; période\ décrément)$ est choisi de manière à ce que le compteur interne n'atteigne pas 0 dans la durée $\frac{1}{f_0}$ si bien que lorsque l'on reçoit le signal de fréquence f_0 le compteur a toujours une valeur supérieure à 0, par contre il atteindra la valeur 0 lorsque l'on reçoit un signal atténué de fréquence f_1. Lorsque le compteur a une valeur différente de 0 une valeur basse est écrite sur le CNA et lorsqu'il vaut 0 une valeur haute y est écrite.

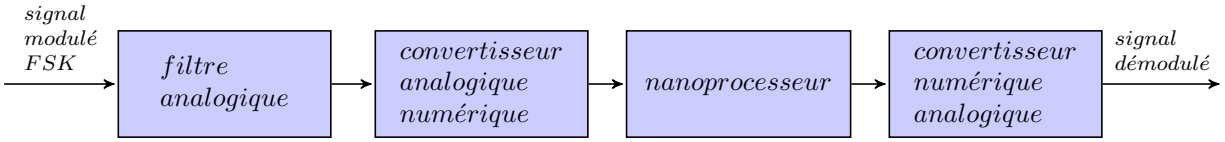

FIGURE 2 – Principe de la chaîne utilisée pour la démodulation FSK

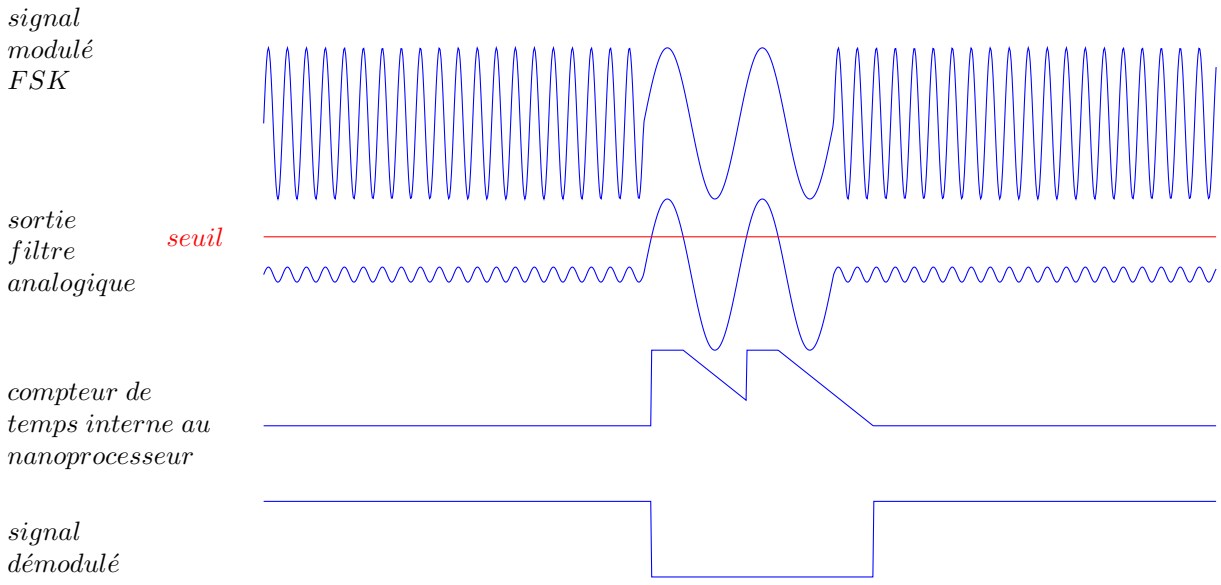

FIGURE 3 – Signaux présentant le principe de la démodulation

Le nanoprocesseur sera implanté sur le FPGA de la carte DE1 d'Altera déjà utilisée en TP de logique. Il sera cadencé à 50 MHz. Des boutons poussoirs seront à utiliser pour le réglage du seuil et des afficheurs 7 segments l'afficheront. Une carte d'extension, cf. figure 8, a été préparée afin de traiter le signal analogique d'entrée : elle comprend les éléments permettant d'implanter le filtre ainsi que les CAN et CNA.

3 Partie analogique

La partie analogique du système est composée d'un filtre passe-bas, actif et à temps continu. La conception d'un filtre est réalisée à partir d'une spécification qui est le plus souvent décrite par un gabarit représentant le gain du filtre dans le domaine fréquentiel, cf. figure 4. Ce gabarit définit notamment les zones, en rouge clair sur la figure, dans lesquelles le spectre du signal filtré ne devra pas se trouver.

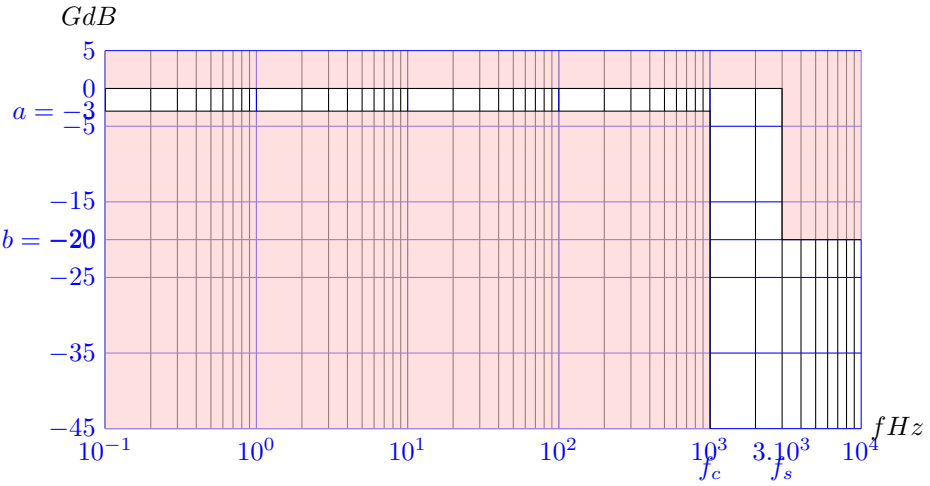

FIGURE 4 – Gabarit du filtre

Comme représenté sur la figure 4, le filtre à réaliser est de type passe bas avec les caractéristiques suivantes :

- $a = -3dB$
- $b = -20dB$
- $f_c = 1kHz$ (fréquence de coupure)
- $f_s = 3kHz$ (fréquence de stop)

Une fois la spécification établie, il est important de déterminer une fonction susceptible de rentrer dans le gabarit. Il en existe bien évidemment plusieurs, mais pour des raisons de conception (notamment l'ordre du filtre), nous choisissons d'utiliser une construction à base de polynômes de Chebychev.
Le gain du filtre [1] est donc :

$$G_n(\omega) = \frac{1}{\sqrt{1+\epsilon^2 C_n^2(\omega)}}$$

avec

$$\begin{cases} C_0(\omega) = 1 \\ C_1(\omega) = \omega \\ C_{n+1}(\omega) = 2\omega C_n(\omega) - C_{n-1}(\omega) \end{cases}$$

La figure 5 présente un tracé cette fonction pour $n = 1, 2, 3$.

3.1 Travail de spécification

Dans cette première partie, il vous est demandé de travailler sur la spécification du filtre afin de disposer d'une fonction de transfert avec de vraies valeurs numériques.

1. $w = \frac{2\pi f}{2\pi f_c} = \frac{2\pi f}{w_c}$ est la pulsation normalisée. Le passage en variable de Laplace se fait avec la relation $p = jw$. Dans la suite, une étape de dénormalisation permettra de se ramener à la spécification initiale.

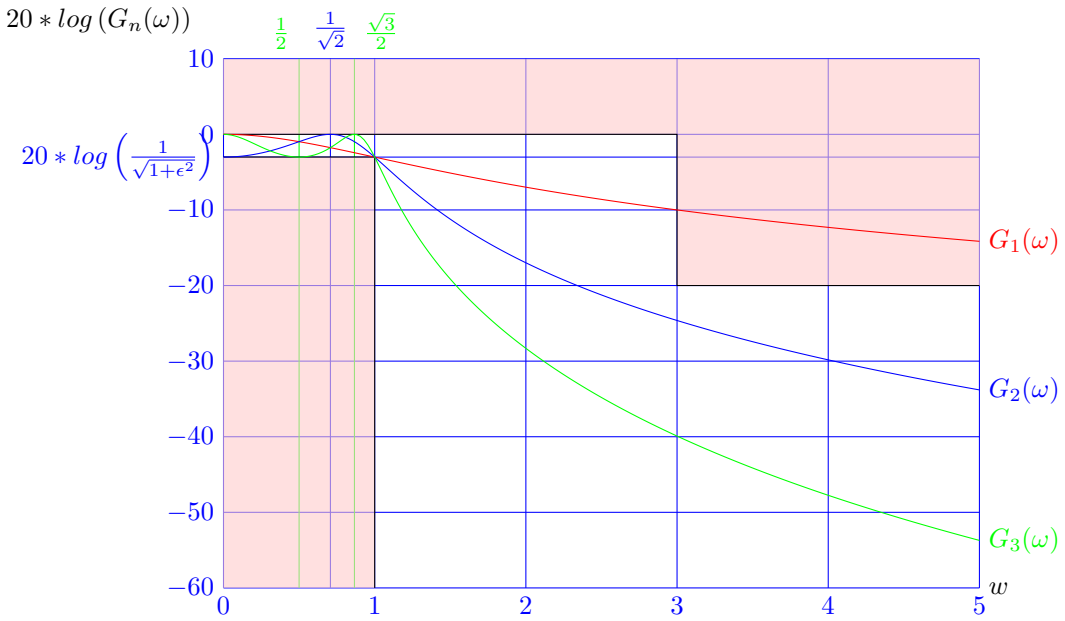

FIGURE 5 – Abaque donnant les fonctions de Chebychev pour les ordres 1, 2 et 3

1. Une étude des variations de la fonction sur l'intervalle $[0;1]$ montre que $G_n(\omega) \in \left[\frac{1}{\sqrt{1+\epsilon^2}};1\right]$ sur celui-ci. En déduire la valeur du coefficient ϵ.
2. Grâce à l'abaque représenté sur la figure 5, déterminer l'ordre du filtre le plus approprié (les $G_n(\omega)$ ont été tracés avec la valeur de ϵ que vous avez trouvée à la question précédente).
3. Écrire la fonction de transfert complexe $H\,(p)$ du filtre en variable de Laplace $p = jw$. Pour cela :
   — vous exploiterez le fait que $H(p)H^*(p) = |H\,(p)|^2 = G\left(\frac{p}{j}\right)^2$,
   — vous calculerez les pôles de la fonction et ne conserverez, pour des raisons de stabilité, que les pôles à partie réelle négative. Pour indication, vous devriez trouver les pôles :

$$\begin{cases} p_1 = -\left(\frac{1}{2}\right)^{\frac{1}{4}} e^{j\frac{3\pi}{8}} \\ \\ p_1^* = -\left(\frac{1}{2}\right)^{\frac{1}{4}} e^{-j\frac{3\pi}{8}} \end{cases}$$

4. Exprimer la fonction de transfert sous la forme :

$$H\,(p) = \frac{K}{\alpha p^2 + \beta p + 1}$$

La conception matérielle du filtre est ensuite réalisée en utilisant des schémas électriques permettant de réaliser des filtres d'ordre 1 et 2. Pour les ordres supérieurs à 2, il suffit d'associer des filtres d'ordre 1 et 2 pour obtenir l'ordre souhaité. Cette approche est toujours possible car on peut décomposer tout polynôme avec des polynômes premiers (ordre 1 ou 2). La maquette que nous utilisons permet d'implémenter un ordre 1 ou 2 selon la structure de Salen Key. Le montage envisagé est présenté sur la 6 et montre une structure constituée de 4 admittances (Y_1, Y_2, Y_3, Y_4), de 2 résistances (R_a, R_b) et d'un amplificateur opérationnel.

1. Montrer que la fonction de transfert de ce filtre est $\frac{v_{out}}{v_{in}} = \frac{KY_1Y_3}{(Y_1+Y_2)(Y_3+Y_4)+Y_3(Y_4-KY_2)}$ avec $K = \frac{R_a+R_b}{R_a}$

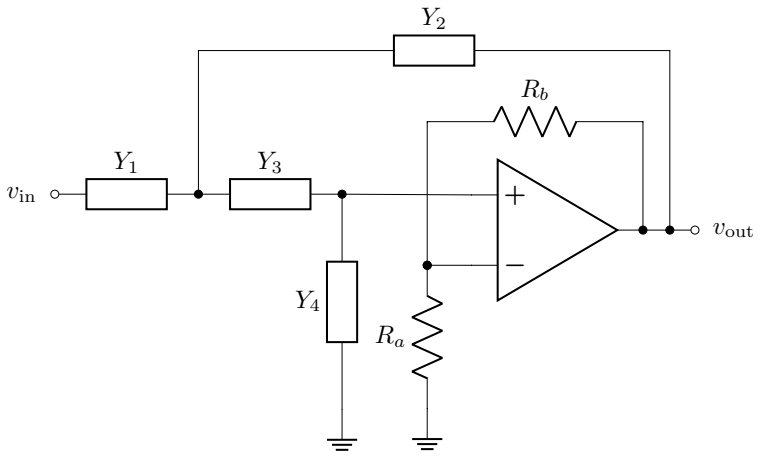

FIGURE 6 – Filtre de Salen Key

2. Sachant que les inductances ne sont pas facilement intégrables (et donc pas utilisées), déterminer la nature des admittances (résistance ou capacité) pour obtenir un comportement de type passe-bas.
3. Calculer les valeurs des admittances permettant de réaliser le filtre souhaité.
   Pour cela, il faut déterminer les valeurs des composants Y_1, Y_2, Y_3, Y_4 par identification de la fonction de transfert $H(p)$ avec la fonction de transfert obtenue dans la question précédente.
   Vous remarquerez que $K = \frac{R_a+R_b}{R_a} \geq 1$, il n'est donc pas possible d'obtenir $K = \frac{1}{\sqrt{2}}$. Aussi, vous considèrerez dans la suite que $K = 1$, soit $R_b = 0$. Cela a pour effet de translater le gain de $+3dB$. Cela n'est pas gênant : l'objectif est de conserver une atténuation de $17dB$ entre la bande passante et la bande atténuée.
   Attention il faudra penser à dénormaliser $H(p)$ en posant $p = \frac{p}{w_c}$ avec $w_c = 2\pi f_c$.
4. Effectuer une simulation du filtre obtenu en traçant la fonction de transfert avec spice :
   (a) aller dans le répertoire `analogique`
   (b) éditer le fichier `sallen-key.cir` et entrer les valeurs de composants calculées
   (c) lancer la simulation, attention à l'échelle log en abscisse : le pas de la grille dépend du zoom de la fenêtre :
   — au CIME sur Linux :
   — ouvrir un terminal
   — aller dans le répertoire `analogique`
   — lancer la simulation depuis le terminal avec la commande `ngspice ./sallen-key.cir`
   — la commande `quit` permet de quitter ngspice
   — à PHELMA sur Windows :
   — aller dans le répertoire `analogique`
   — double cliquer sur le fichier `sim.bat`
   (d) après avoir simulé, vous trouverez des fichiers dat (fichiers des points des courbes), plt (scripts gnuplot permettant de générer les fichiers eps à partir des fichiers dat) et eps (courbes qu'il est possible de visualiser avec evince sous Linux) préfixés par la valeur des composants utilisés lors des simulations.
5. Dans la réalité, il faut choisir des composants en tenant compte des valeurs normalisées de résistances et de condensateurs disponibles. Faire une nouvelle simulation avec spice en utilisant cette fois-ci des composants choisis parmis les valeurs normalisées suivantes et vérifier la pente observée sur le gain dans la bande de fréquences f_c,f_s. Les valeurs des composants disponibles sont :
   — Résistances : 820Ω, $2.7K\Omega$, $5.6K\Omega$, $12K\Omega$
   — Capacités : $10nF$, $20nF$, $50nF$, $100nF$
   Remarque : il est possible de modifier les scripts gnuplot pour tracer sur le même graphe la courbe théorique obtenue dans le point 4 précédent et la courbe obtenue avec les valeurs normalisées.

3.2 Implémentation du filtre et mesures

Dans cette partie du projet, nous vous demandons de réaliser l'implémentation matérielle du filtre en utilisant les résistances et capacités que vous venez de déterminer. Pour cela, vous devez utiliser la platine analogique pré-câblée (platine verte). Le schéma donné sur la figure 7 montre la structure du filtre. Vous devez donc placer vos composants passifs sur les connecteurs prévus à cet effet (représentés par les rectangles jaunes sur le schéma). La carte filtre est directement alimentée par la carte FPGA, vous n'avez pas besoin de vous en préoccuper. Vous pouvez également jeter un oeil sur la photo de la carte analogique qui se trouve sur la figure 8 pour savoir où placer vos composants.

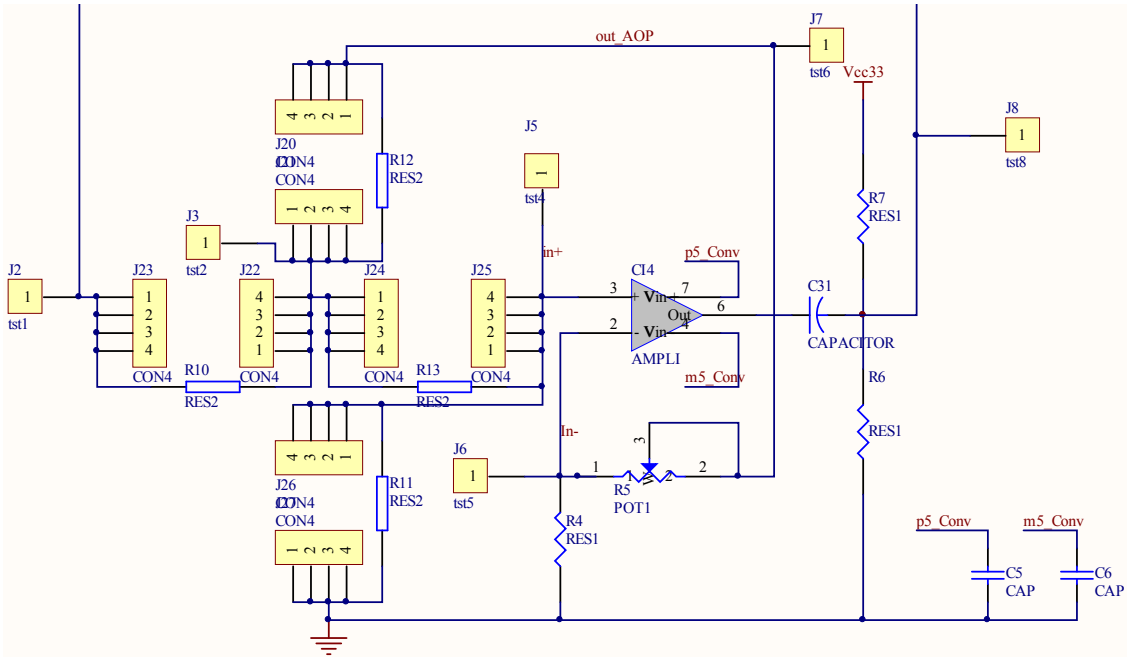

FIGURE 7 – Schéma électrique du filtre

Travail à effectuer : l'objectif est de régler le filtre pour profiter de toute la dynamique de l'étage de conversion analogique numérique qui suit :

1. Placer un signal d'amplitude $2V$ en entrée ($-1V$ / $+1V$) du filtre, dans la bande passante. En observant out_AOP, régler le potentiomètre pour avoir un gain minimal (ce qui équivaut à $R_b = 0$). Puis se placer à la fréquence à laquelle le gain du filtre est maximal et vérifier que cette fréquence correspond à celle obtenue par simulation.
2. Observer le signal Vin_ADC et régler la dynamique du signal d'entrée permettant de profiter au maximum de la dynamique d'entrée du CAN.
3. Tracer les diagrammes de Bode en amplitude et en phase du filtre réalisé. Vérifier que vous respectez bien le gabarit de la spécification et obtenu avec la simulation spice en terme de pente du gain dans la bande de fréquence f_c,f_s. Le fichier `doc/3decades.pdf` est une feuille de papier semi-log sur laquelle vous pouvez tracer votre diagramme, il est également possible d'utiliser un tableur ou gnuplot.

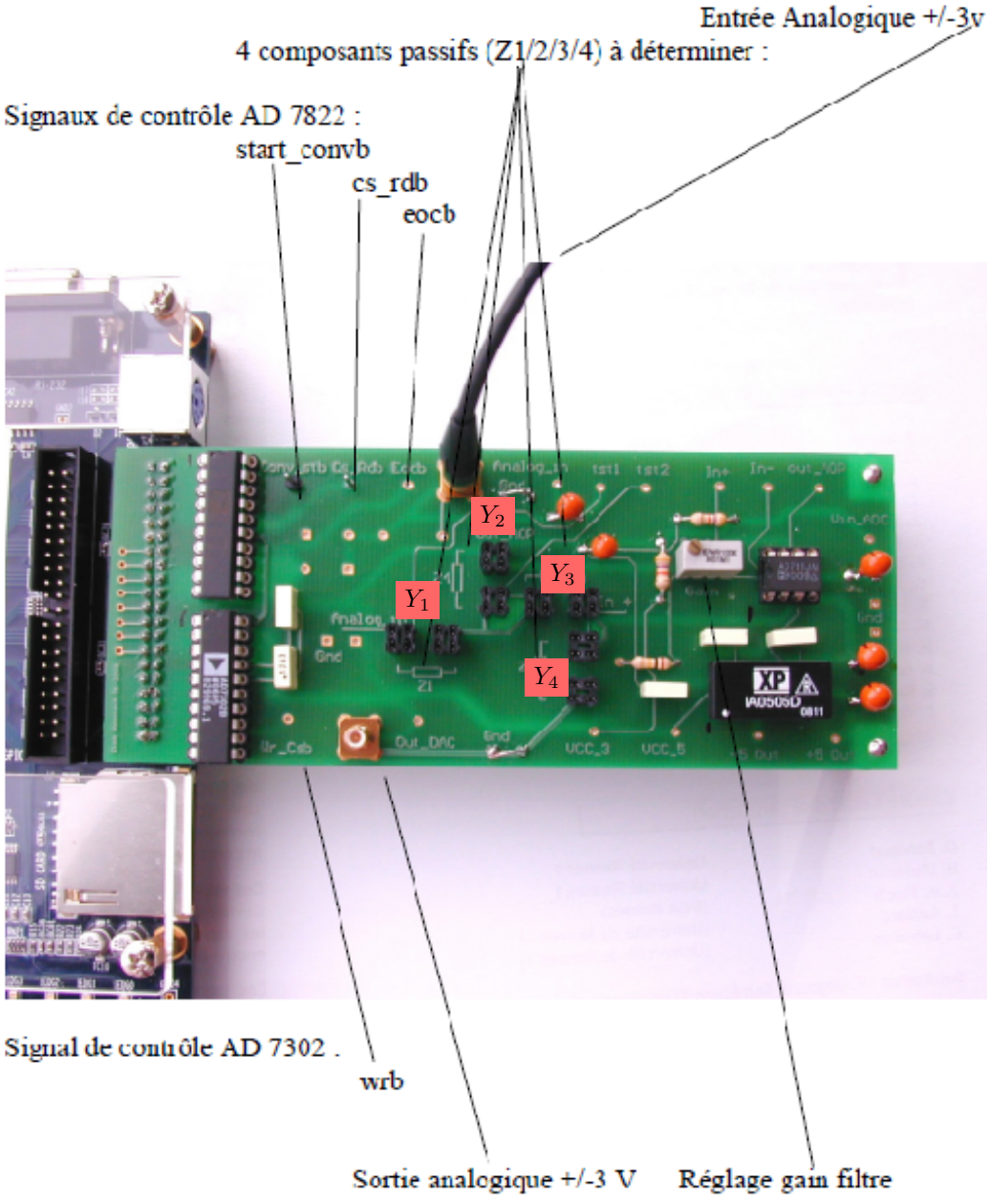

FIGURE 8 – Photo de la carte analogique

4 Dispositifs de conversion analogique-numérique et numérique-analogique

4.1 Analyse des datasheets

Travail à effectuer (les deux premières questions nécessitent l'étude des datasheets des composants de conversion analogique numérique et numérique analogique qui se trouvent dans le répertoire `doc`) :

1. Comprendre le rôle des signaux d'interface des CAN et CNA. Leurs interfaces sont données sur les figures 9 et 10.
2. Comprendre les chronogrammes fonctionnement des CAN et CNA donnés figures 11 et 12.
3. Donner l'interface du circuit décrit dans le fichier `can-cna/adc_ctrl.vhd` et extraire de ce fichier la machine à états qui y est décrite. Ce composant est utilisé dans la partie numérique afin que le nanoprocesseur récupère les échantillons du signal analogique à démoduler.
4. Est-ce que la machine à états faisant l'objet de la question précédente répond bien au chronogramme de la figure 11 ? Est-ce qu'il y a une condition à respecter sur la fréquence d'horloge ?

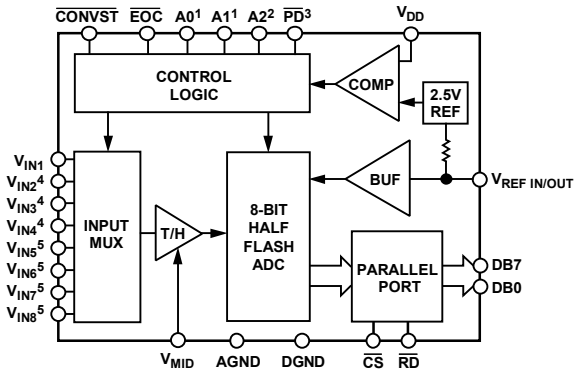

FIGURE 9 – Interface du CAN AD7822

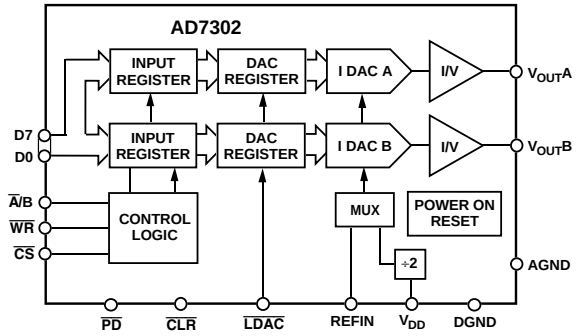

FIGURE 10 – Interface du CNA AD7302

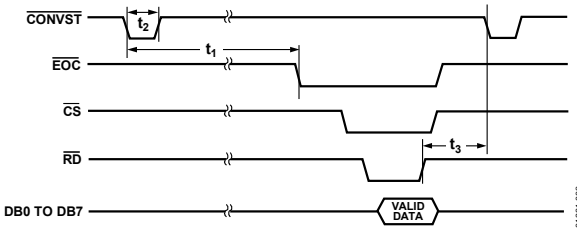

FIGURE 11 – Chronogramme du CAN AD7822

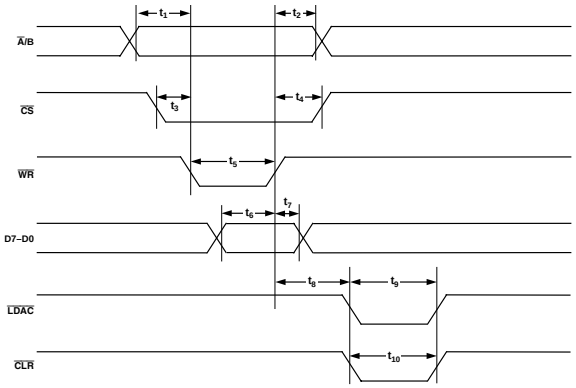

FIGURE 12 – Chronogramme du CNA AD7302

4.2 Observation des signaux de la carte

Dans cette partie, il est demandé d'observer les signaux de contrôle utilisés par les CAN (convertisseur analogique-numérique) et CNA (convertisseur numérique-analogique). Pour cela, il faut commencer par configurer le FPGA de la carte DE1 via le projet Quartus qui se trouve dans le répertoire `can-cna`. Ce projet contient une petite application matérielle numérique qui pilote les CAN et CNA : les échantillons prélevés sur le CAN sont simplement recopiés sur le CNA. Travail à effectuer :

1. Programmer le FPGA avec le logiciel Quartus (ouvrir le projet qui se trouve dans le répertoire `can-cna`, lancer la compilation du projet avec le raccourci `ctrl+L` puis aller dans le menu `tools` pour lancer l'outil `programmer`).
2. Faire en sorte qu'il y ait un signal analogique en entrée de CAN.
3. Observer les signaux $conv\_stb$, cs_rdb, $eocb$, wr_csb, le signal analogique numérisé par le CAN et le signal analogique en sortie CNA.
4. Tracer un chronogramme de ces signaux et vérifier qu'ils ont bien les caractéristiques identifiées lors de l'analyse des datasheets faite précédemment.

5 Partie processeur numérique

Un processeur est une machine programmable capable d'effectuer des millions d'opérations par seconde. Il constitue le coeur des ordinateurs qui exécute les instructions et traite des données numériques. Nous vous sensibilisons aux architectures de processeur au travers d'un processeur élémentaire, que nous avons appelé nanoprocesseur. Celui-ci permettra l'exécution du programme de démodulation, l'ensemble étant implanté sur une carte FPGA DE1 d'Altera. Son architecture est décrite avec Quartus en schématique et langage VHDL. Le projet Quartus correspondant se trouve dans le répertoire `numerique/nanoproc`. Son jeu d'instruction est détaillé sur la figure 14. Il comprend des instructions permettant de réaliser :

— des transferts non conditionnels :`mv,ldi,ld,st`
— des transferts conditionnels : `mvnz,mvgt`
— des calculs arithmétiques ou logiques : `add,sub,and`
— des branchements (sauts) relatifs non conditionnels : `bra`
— des branchements (sauts) relatifs conditionnels : `brnz,brgt,brz,brmi`

La description du nanoprocesseur fait intervenir des constantes, présentées sur la figure 13, afin de gagner en généricité. Il n'est pas nécessaire de lire ce tableau dans le détail, vous y reviendrez plus tard, si besoin.

`system.bdf`				
`inst_nano_proc`	`DW`	Data Width	16	Largeur des bus de données et d'adresse
`inst_memory_and_io`	`DW`	Data Width	16	Largeur des bus de données et d'adresse
`nanoproc.bdf`				
	`ACW`	Alu Code Width	2	Nombre de bits utilisés pour coder l'opération de l'unité arithmétique et logique
	`AW`	Address Width	8	Nombre de bits du bus d'adresse des mémoire ROM et RAM
	`OCW`	OpCode Width	4	Nombre de bits utilisés pour représenter le code opération de l'instruction
	`DW`	Data Width	16	Nombre de bits des bus de données
	`RNW`	Register Number Width	2	Nombre de bits utilisés pour représenter les numéros des registres
	`IBSW`	Internal Bus Selection Width	3	Nombre de bits utilisés représenter le signal de sélection du multiplexeur pilotant le bus de données interne
`pc_register.bdf`				
	`DW`	Data Width	16	Nombre de bits des bus de données
	`RNW`	Register Number Width	2	Nombre de bits utilisés pour représenter les numéros des registres
`alu.bdf`				
	`DW`	Data Width	16	Nombre de bits des bus de données
	`ACW`	Alu Code Width	2	Nombre de bits utilisés pour coder l'opération de l'unité arithmétique et logique
`nanoproc_pc.vhd`				
	`ACW`	Alu Code Width	2	Nombre de bits utilisés pour coder l'opération de l'unité arithmétique et logique
	`OCW`	OpCode Width	4	Nombre de bits utilisés pour représenter le code opération de l'instruction
	`RNW`	Register Number Width	2	Nombre de bits utilisés pour représenter les numéros des registres
	`IBSW`	Internal Bus Selection Width	3	Nombre de bits utilisés représenter le signal de sélection du multiplexeur pilotant le bus de données interne
`memory_and_io.vhd`				
	`DW`	Data Width	16	Nombre de bits des bus de données
	`AW`	Address Width	8	Nombre de bits du bus d'adresse des mémoire ROM et RAM
	`DAW`	Decoder Address Width	3	Nombre de bits utilisés pour le décodage d'adresse
`inst_ram`	`DW`	Data Width	16	Largeur du bus de données
`inst_ram`	`AW`	Address Width	8	Largeur du bus d'adresse
`inst_rom`	`DW`	Data Width	16	Largeur du bus de données
`inst_rom`	`AW`	Address Width	8	Largeur du bus d'adresse

FIGURE 13 – Constantes utilisées dans la description

	nombre de bits du champ	$DW-OCW-2*RNW$	OCW	RNW	RNW	$Affecte\ les\ drapeaux$	
instruction	opération	DW-1 . . .OCW+2*RNW	OCW+2*RNW-1 . . .2*RNW	2*RNW-1 . . .RNW	RNW-1 . . .0	Z	G
$mv\ r_{dst}, r_{src}$	$r_{dst} \leftarrow r_{src}$	inutilisé	0000	r_{dst}	r_{src}	non	non
$ldi\ r_{dst}, \#val$	$r_{dst} \leftarrow val$	inutilisé	0001	r_{dst}	inutilisé	non	non
val est dans le mot mémoire qui suit ldi		val					
$add\ r_{dst}, r_{src}$	$r_{dst} \leftarrow r_{dst} + r_{src}$	inutilisé	0010	r_{dst}	r_{src}	oui	oui
$sub\ r_{dst}, r_{src}$	$r_{dst} \leftarrow r_{dst} - r_{src}$	inutilisé	0011	r_{dst}	r_{src}	oui	oui
$ld\ r_{dst}, [r_{src}]$	$r_{dst} \leftarrow [r_{src}]$	inutilisé	0100	r_{dst}	r_{src}	non	non
$st\ [r_{dst}], r_{src}$	$[r_{dst}] \leftarrow r_{src}$	inutilisé	0101	r_{dst}	r_{src}	non	non
$mvnz\ r_{dst}, r_{src}$	$r_{dst} \leftarrow r_{src}\ si\ \overline{Z}$	inutilisé	0110	r_{dst}	r_{src}	non	non
$mvgt\ r_{dst}, r_{src}$	$r_{dst} \leftarrow r_{src}\ si\ G$	inutilisé	0111	r_{dst}	r_{src}	non	non
$and\ r_{dst}, r_{src}$	$r_{dst} \leftarrow r_{dst}.r_{src}$	inutilisé	1000	r_{dst}	r_{src}	oui	oui
$bra\ offset$	$pc \leftarrow pc + 1 + offset$	inutilisé	1001	$offset$		non	non
$brnz\ offset$	$pc \leftarrow pc + 1 + offset\ si\ \overline{Z}$	inutilisé	1010	$offset$		non	non
$brgt\ offset$	$pc \leftarrow pc + 1 + offset\ si\ G$	inutilisé	1011	$offset$		non	non
$brz\ offset$	$pc \leftarrow pc + 1 + offset\ si\ Z$	inutilisé	1100	$offset$		non	non
$brmi\ offset$	$pc \leftarrow pc + 1 + offset\ si\ \overline{Z}.\overline{G}$	inutilisé	1101	$offset$		non	non

FIGURE 14 – Liste des instructions du nanoprocesseur

	nombre de bits du champ	$DW-OCW-2*RNW$								OCW				RNW		RNW		valeur hexadécimale
adresse mémoire	instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	$ldi\ r_1, \#0xaaaa$	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	$0x0014$
1	valeur immédiate $0xaaaa$	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	$0xaaaa$
2	$ldi\ r_2, \#0x000f$	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	$0x0018$
3	valeur immédiate $0x000f$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	$0x000f$
4	$ldi\ r_3, \#0x0200$	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	$0x001c$
5	valeur immédiate $0x0200$	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	$0x0200$
6	$and\ r_1, r_2$	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	$0x0086$
7	$st\ [r_3], r_1$	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	$0x005d$
8	$bra\ -1$	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	$0x009f$

FIGURE 15 – Tableau d'assemblage du code assembleur d'exemple

5.1 Présentation du processeur

La figure 17 présente le composant `system.bdf` de plus haut niveau hiérarchique. L'interface du circuit est composée d'entrées/sorties lui permettant de s'interfacer avec le monde extérieur. La figure 16 les présente.

nom	direction	largeur en bits	commentaire
clk	in	1	relié à une horloge à 50 MHz
key	in	3	relié au boutons poussoirs
sw	in	10	relié aux interrupteurs
adc_eoc_b	in	1	relié à un signal du CAN
adc_data_bus	in	8	relié à des signaux du CAN
adc_convst_b	out	1	relié à un signal du CAN
adc_csrd_b	out	1	relié à un signal du CAN
dac_cswr_b	out	1	relié à un signal du CNA
dac_data_bus	out	8	relié à des signaux du CNA
$hex0$	out	7	relié à un afficheur 7 segments
$hex1$	out	7	relié à un afficheur 7 segments
$hex2$	out	7	relié à un afficheur 7 segments
$hex3$	out	7	relié à un afficheur 7 segments
$ledr$	out	10	relié aux leds rouges

FIGURE 16 – Interface du système

Le système s'articule autour :
- d'un bloc `nanoproc` qui est le coeur du système : il comprend une partie opérative et une partie commande,
- d'un bloc `memory_and_io` qui contient les mémoires ROM et RAM du processeur ainsi que les blocs matériels permettant au processeur de communiquer avec l'extérieur.

Ces deux blocs communiquent au travers d'un bus constitué :
- d'un bus d'adresse de DW bits piloté par le bloc `nanoproc` pour indiquer l'adresse mémoire à laquelle il souhaite accéder,
- d'un signal wr piloté par le bloc `nanoproc` qu'il utilise pour indiquer la nature de l'opération qu'il souhaite faire à l'adresse qu'il a positionné (0 pour la lecture et 1 pour l'écriture),
- d'un bus de données de DW bits de sortie du bloc `nanoproc` qu'il utilise pour positionner des données en entrée du bloc `memory_and_io`,
- d'un bus de données d'entrée de DW bits du bloc `nanoproc` qu'il utilise pour récupérer des données en provenance du bloc `memory_and_io`.

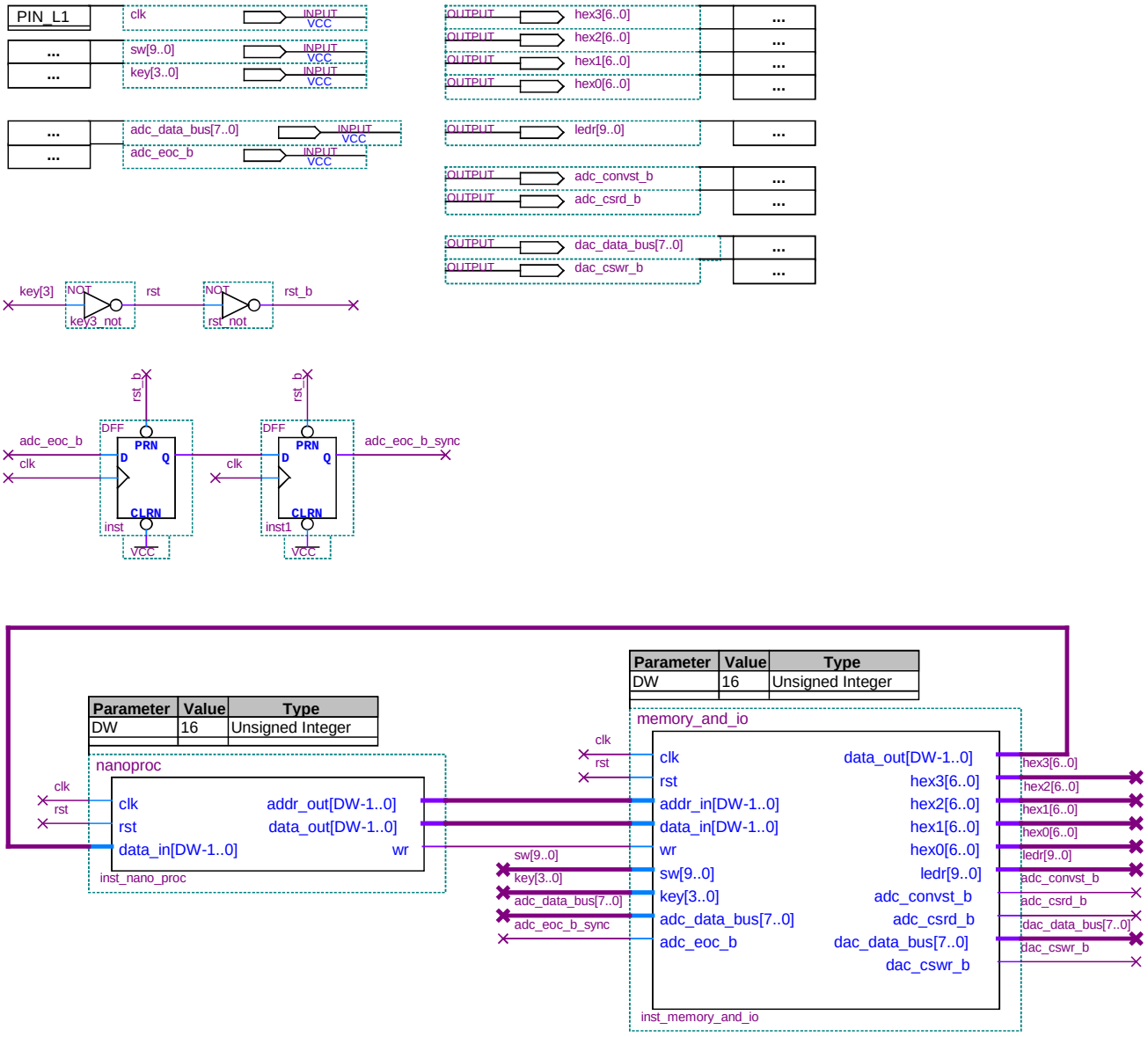

FIGURE 17 – Schéma du composant `system` de plus haut niveau hiérarchique

Le bloc nanoproc est détaillé sur la figure 18. Il se structure autour d'une partie commande, bloc nanoproc_pc et d'une partie opérative. Cette partie opérative comprend les éléments qui suivent :

— INSTRUCTION REGISTER est composé d'un registre qui permet de mémoriser l'instruction en cours d'exécution et d'un jeu de câblage qui décompose chacun des éléments de l'instruction : op_code, offset, r_src, r_dst. Enfin, instr_reg_en permet de piloter le chargement de ce registre.

— PC REGISTER est composé d'un bloc qui implante le registre r0 qui est utilisé comme compteur programme : il contient l'adresse de la prochaine instruction à exécuter. Il peut être chargé par une information venant du bus de données interne internal_bus (pc_sel_internal_bus) ou bien il est possible de lui ajouter une valeur d'offset contenue dans l'instruction en cours d'exécution (pc_sel_internal_bus) ou il est simplement possible de l'incrémenter (pc_sel_inc). Dans tous les cas le signal reg_en[0] permet de l'activer.

— GENERAL PURSPOSE REGISTERS contient un groupe de 3 registres d'usage général,r1, r2, r3, qui ont leurs entrées câblées sur le bus de données interne internal_bus . Chacun est activé par un signal reg_en[i] .

— ACCU REGISTER, ALU, ALU REGISTER, FLAGS : l'unité arithmétique et logique inst_alu permet de réaliser des opérations sur 2 opérandes. Un premier est mémorisé dans le registre accu_reg alimenté par le bus de données interne internal_bus et activé par le signal accu_reg_en et le second provient directement du bus de données interne internal_bus . La sortie de l'unité arithmétique et logique peut être mémorisée par le registre alu_reg en activant le signal alu_reg_en . La sortie de ce registre entre dans un bloc de logique combinatoire flags_comparator qui sort les deux drapeaux Z et G , qui indiquent respectivement si son contenu est nul ou supérieur strictement à 0 (en considérant qu'il s'agit d'un entier signé représenté en complément à 2).

— Le multiplexeur internal_bus_mux redirige sur le bus de données interne internal_bus , selon internal_bus_mux_sel , soit :
  — le compteur programme r0 ,
  — un registre d'usage général r1, r2, R3 ,
  — alu_reg_out la sortie du registre de l'unité arithmétique et logique,
  — data_in le bus de données externe d'entrée.

— BUS REGISTERS est composé :
  — d'un registre addr_reg activé par le signal addr_reg_en qui mémorise l'adresse déposée sur le bus externe,
  — d'un registre data_reg activé par data_reg_en qui mémorise la donnée déposée sur le bus externe
  — d'un registre wr_reg qui permet de générer un signal d'écriture sur le bus externe en phase avec le dépôt d'une nouvelle donnée ou adresse.

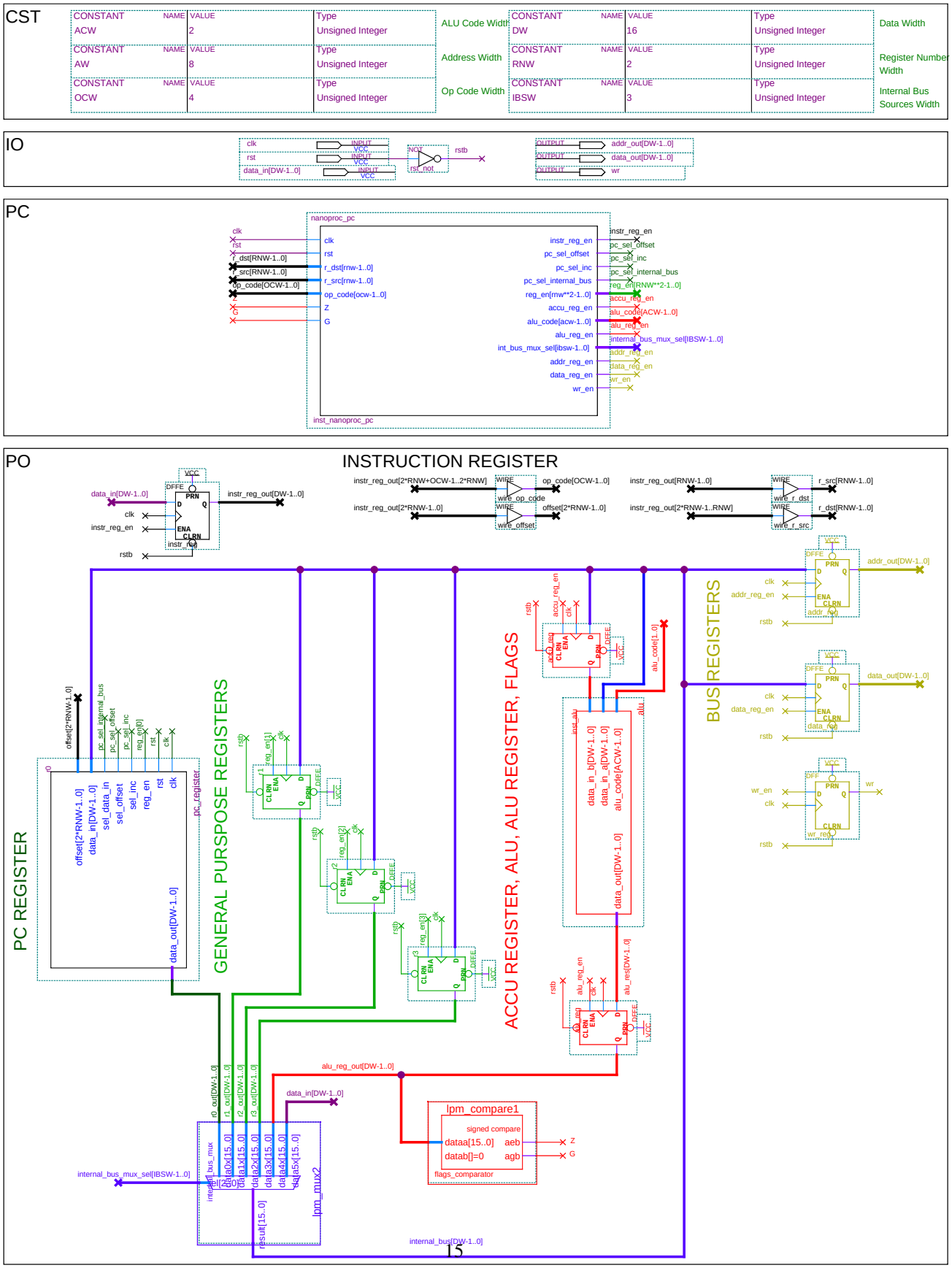

FIGURE 18 – Schema du nanoprocesseur

La partie commande `nanoproc_pc` est décrite en VHDL dans le fichier `nanoproc_pc.vhd`. La machine à états correspondant se trouve figure 19. Elle fait apparaître les différentes étapes permettant l'exécution des instructions : 2 états `Fetch1,Fetch2` sont utilisés pour chercher le code opération et jusqu'à 3 états, `Exec1,Exec2,Exec3`, peuvent être nécessaires à l'exécution de l'instruction.

Exemple du déroulement de l'exécution de l'instruction $add\ r_1, r_2$:

- `Fetch1`
  - `pc_sel_inc, reg_en(0)` : incrémentation du compteur programme au front montant d'horloge suivant pour préparation de l'adresse de la prochaine instruction
  - `addr_reg_en` : mémorisation au front montant d'horloge suivant de la valeur courante du compteur programme dans le registre d'adresse
- `Fetch2`
  - `instr_reg_en` : mémorisation au front montant d'horloge suivant de l'instruction
  - `int_bus_mux_sel` $\Leftarrow r_{src}$, , `addr_reg_en` l'adresse de la prochaine instruction (ou de la valeur immédiate si `ldi`) est mise au front montant d'horloge suivant sur le bus d'adresse
- `Exec1`
  - `int_bus_mux_sel` $\Leftarrow r_{src}$: le contenu du registre source est mis sur le bus de données interne
  - `acc_reg_en` : le registre accumulateur est activé ce qui permet d'y mémoriser le registre source au front montant d'horloge suivant
- `Exec2`
  - `alu_code` $\Leftarrow$ `code_add` : le code de l'opération à réaliser, ici celui correspondant à une addition, est indiqué à l'unité arithmétique et logique
  - `int_bus_mux_sel` $\Leftarrow r_{dst}$: le contenu du registre destination est mis sur le bus de données interne, le résultat de l'addition sera donc disponible dans ce cycle d'horloge
  - `alu_reg_en` : le résultat étant disponible dans ce cycle d'horloge, le registre qui mémorise le résultat de calcul est activé ce qui permet de mémoriser le résultat au front montant d'horloge suivant
- `Exec3`
  - `int_bus_mux_sel` $\Leftarrow alu\_reg\_out$: le contenu du registre contenant le résultat est mis sur le bus de données interne
  - `reg_en(`r_{dst}`)` le registre destination est activé, ce qui permet d'y mémoriser le résultat du calcul d'addition

Exemple du déroulement de l'exécution de l'instruction $brz\ 0x2$ en supposant que $Z = 1$ et que cette instruction est à l'adresse 10.

- `Fetch1`
  - `pc_sel_inc, reg_en(0)` : incrémentation du compteur programme au front montant d'horloge suivant pour préparation de l'adresse de la prochaine instruction
  - `addr_reg_en` : mémorisation au front montant d'horloge suivant de la valeur courante du compteur programme dans le registre d'adresse
- `Fetch2`
  - `instr_reg_en` : mémorisation au front montant d'horloge suivant de l'instruction
  - `int_bus_mux_sel` $\Leftarrow r_{src}$, , `addr_reg_en` l'adresse de la prochaine instruction (ou de la valeur immédiate si `ldi`) est mise au front montant d'horloge suivant sur le bus d'adresse
- `Exec1`
  - `pc_sel_offset, reg_en(0)` : on ajoute la valeur $offset$ donc 2 au registre r_0 qui a pour valeur 11 et il est activé, il vaudra donc 13 au cycle d'horloge suivant. La prochaine instruction à s'exécuter sera donc celle qui se trouve à l'adresse 13

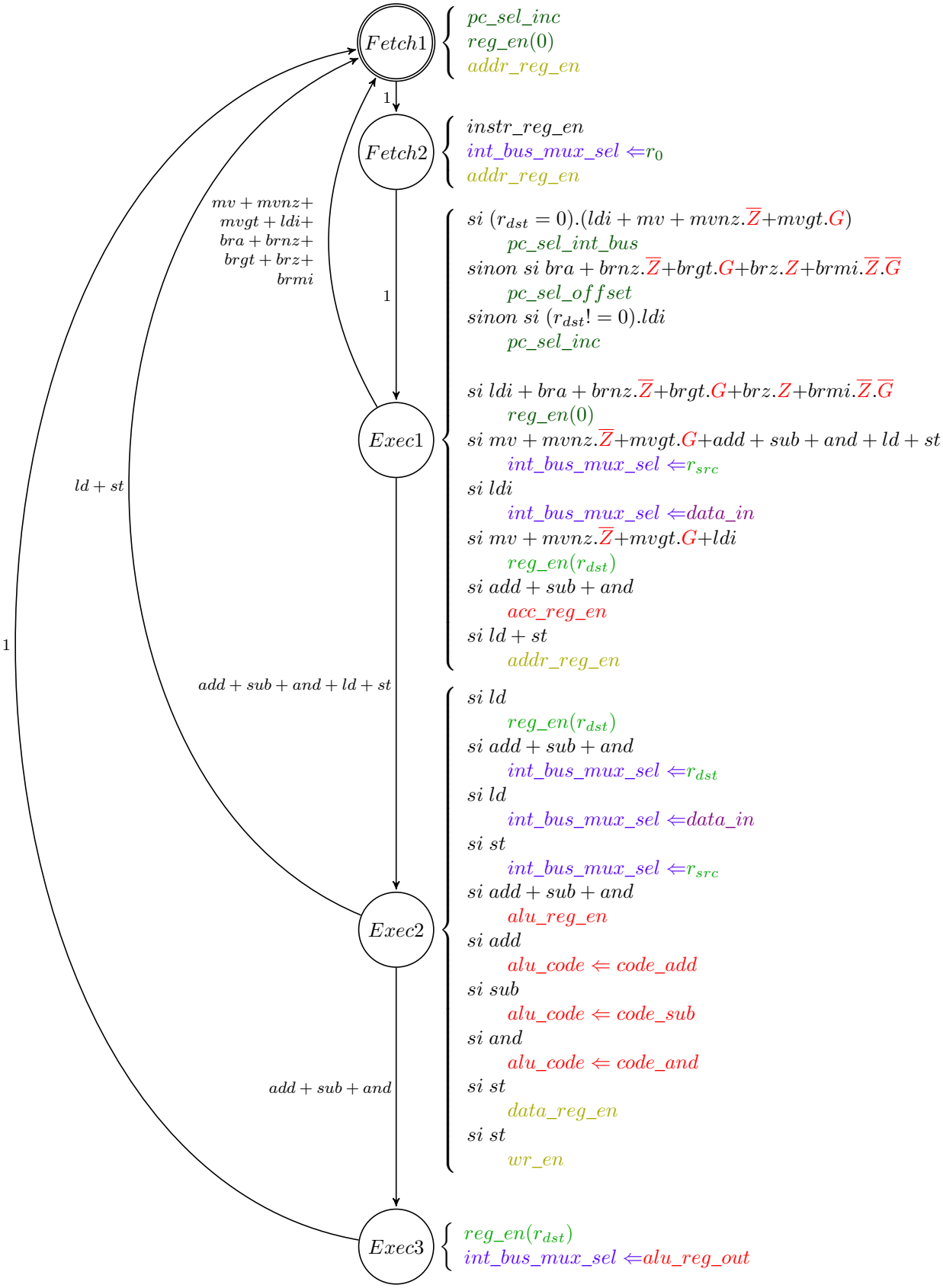

FIGURE 19 – Partie commande du nanoprocesseur

Nous avons vu précédemment que le bloc `nanoproc` échange avec le bloc `memory_and_io` au travers d'un bus. Ce dernier bloc est détaillé sur la figure 20. Il contient la mémoire `ROM` de 256 mots de 16 bits où le programme est stocké, la mémoire `RAM` de taille identique à la mémoire `ROM` où il est possible de stocker de l'information en cours d'exécution du programme et des blocs permettant de s'interfacer avec les entrées/sorties de la carte : `HEX`, `LEDR`, `ADC`, `DAC`.
Afin que les échanges soient possibles avec chacun de ces éléments, une plage d'adresse mémoire leur est affectée. Une lecture du schéma, blocs `Address decoding` et `Output data bus`, montre que les trois bits `addr_in[10..8]` sont utilisés pour identifier l'élément auquel `nanoproc` s'adresse. Ci-après quelques exemples de fonctionnement :
Lorsque `addr_in[10..8]="001"` la donnée à l'adresse indiquée sur `addr_in[7..0]` dans la mémoire `RAM` sera disponible sur le bus de données de sortie `data_out[DW-1..0]`. Si en plus `wr=1` alors une écriture sera faite dans la mémoire `RAM` à l'adresse indiquée sur `addr_in[7..0]`.
Lorsque `addr_in[10..8]="010"` la donnée affichée sur les 4 afficheurs 7 segments est disponible sur le bus de données de sortie `data_out[DW-1..0]`. Si en plus `wr=1` alors une écriture de la valeur présente sur le bus de donnée d'entrée `data_in[DW-1..0]` sera faite dans le registre qui mémorise la valeur présentée aux afficheurs.

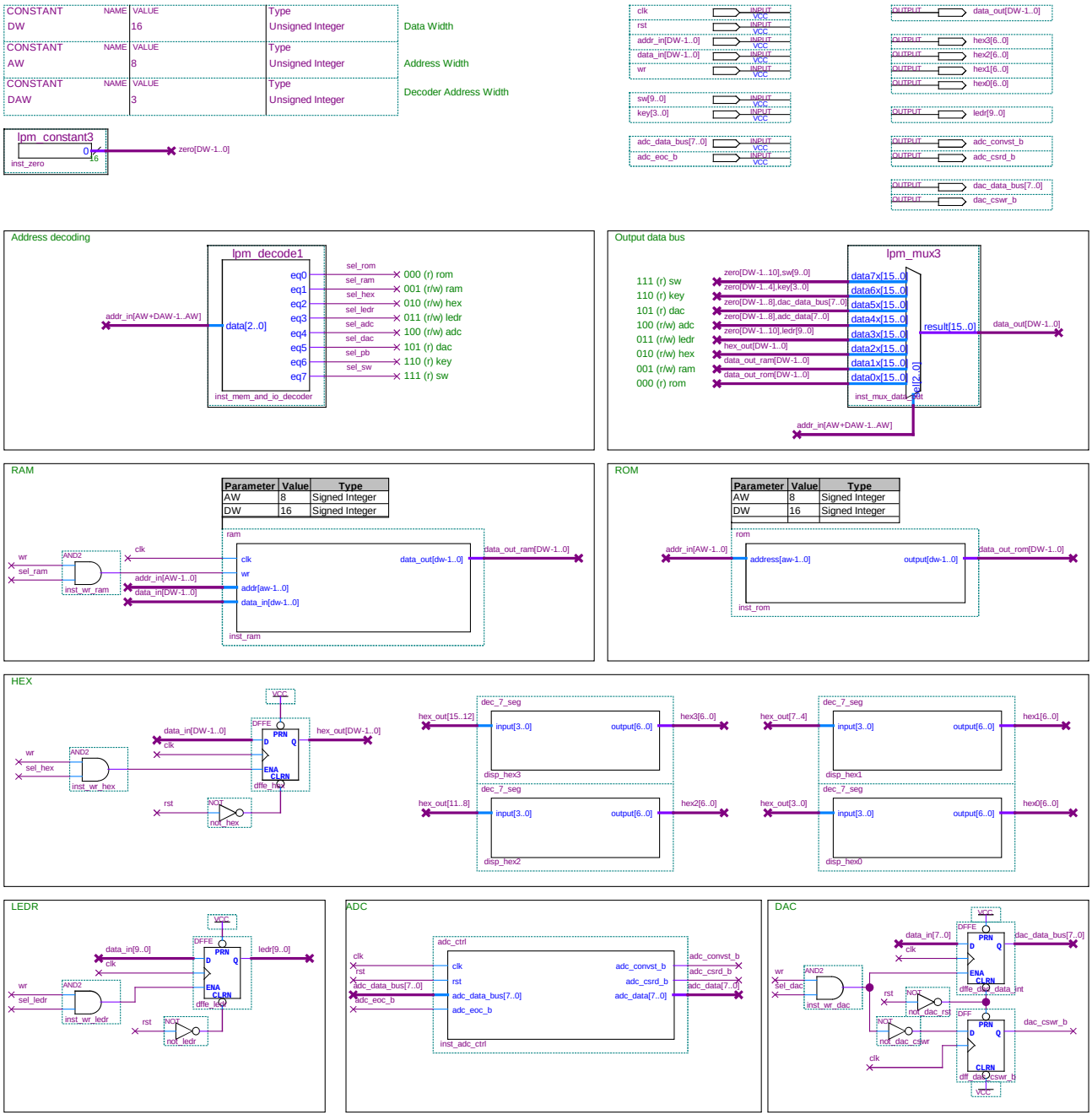

FIGURE 20 – Schema de la partie mémoire et entrées/sorties `memory_and_io`

5.2 Programmation du nanoprocesseur

La programmation du nanoprocesseur commence par l'écriture d'un programme assembleur exprimé avec le jeu d'instruction présenté sur la figure 14. Ensuite ce programme est traduit en langage machine de manière à obtenir la suite des mots le représentant. Après-cela, ces mots sont introduits dans la description du contenu de la mémoire `ROM` (fichier `rom.vhd`) du nanoprocesseur.
Cette démarche est illustrée au travers de l'implantation du programme assembleur présenté sur le listing 1. Celui-ci réalise un et logique entre deux entiers puis affiche le résultat sur les afficheurs 7 segments. Pour cela, tout d'abord, le programme est traduit en langage machine à l'aide du tableau d'assemblage présenté sur la figure 15. Ensuite, ces informations sont introduites dans la description VHDL de la mémoire ROM, cf. listing 2.
Après cela, le programme peut être vérifié par simulation. Il convient alors de lui présenter des stimuli réalistes ce qui peut nécessiter des modifications du banc de test. Il est ensuite possible de tester son fonctionnement sur carte.

```
  ldi r1,#0xAAAA
  ldi r2,#0x080F
  ldi r3,#0x0200 ; adresse afficheur
  and r1,r2
  st  [r3],r1
loop:
  bra loop
```

Listing 1 – Exemple de code assembleur : et logique et affichage du résultat

```
library ieee;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all;

entity rom is
generic (
  AW  : integer := 8;   -- Address Width
  DW  : integer := 16   -- Data Width
);
port (
  address : in std_logic_vector(AW-1 downto 0) ;
  output  : out std_logic_vector(DW-1 downto 0) ) ;
end rom;

architecture arch of rom is

type tab_rom is array (0 to 2**AW-1) of bit_vector(DW-1 downto 0) ;

constant my_rom : tab_rom := (
  000 => x"0014", -- ldi r1,#0xaaaa
  001 => x"aaaa", --
  002 => x"0018", -- ldi r2,#0x080f
  003 => x"080f", --
  004 => x"001c", -- ldi r3,#0x0200      adresse afficheur
  005 => x"0200", --
  006 => x"0086", -- and r1,r2
  007 => x"005d", -- st [r3],r1
  -- loop
  008 => x"009f", -- bra loop
  others=>x"0000");
begin

  output <= to_stdlogicvector(my_rom(to_integer(unsigned(address)))) ;

end architecture;
```

Listing 2 – Code VHDL de la mémoire ROM contenant le code assembleur d'exemple

5.3 Travail à réaliser

5.3.1 Prise en main du nanoprocesseur

1. Parcourir les différentes feuilles de schéma du projet Quartus du nanoprocesseur.
2. Editer et parcourir le banc de test `system_vhd_tst.vhd`
3. Editer le fichier `rom.vhd` et vérifier qu'il s'agit bien du code présenté sur dans listing 2.
4. Lancer une simulation :
   — Pour cela il faudra lancer Modelsim. Au CIME sur Linux : taper la comande `vsim` dans un terminal, à PHELMA sous Windows : lancer Modelsim depuis le menu `Démarrer`.
   — Aller dans le répertoire `simulation/modelsim` par exemple via le menu `File>Change Directory`.
   — Dans le transcript taper les commandes `do rtl_sim_prepare.do` pour la compilation et `do rtl_sim_launch.do` pour lancer la simulation. Ce sera à vous de lancer la simulation pendant une durée pertinente.
5. Vérifier que l'exécution des instructions se passe comme attendu (normalement, il devrait y avoir un problème).
6. Modifier l'unité arithmétique et logique pour qu'elle puisse réaliser l'opération manquante et valider le bon fonctionnement par simulation.
7. Tester le programme sur carte (ouvrir le projet Quartus, lancer sa compilation avec le racourci `ctrl+L` puis aller dans le menu `tools` pour lancer l'outil `programmer`).
8. Quelle est l'amplitude maximale des sauts relatifs `bra,brnz,brgt,brz,brmi` ?
9. Que faudrait-il modifier pour augmenter leur amplitude ?

5.3.2 Calcul de la somme des N premiers entiers

L'objectif est ici de réaliser un programme qui calcule la somme des N premiers entiers :

$$sum = \sum_{i=0}^{N} i$$

1. Faire un programme qui calcule sum avec la valeur de N fixée dans le programme et sum dans un registre. A valider par simulation.
2. Compléter le programme précédent pour qu'il affiche le résultat sur les afficheurs 7 segments. Valider dans un premier temps par simulation puis sur carte.
3. Compléter le programme pour qu'il soit possible :
   — de rentrer la valeur de N avec les interrupteurs,
   — de déclencher le calcul avec un appui sur un bouton poussoir.
   Valider par simulation puis sur carte.

5.3.3 Développement du programme de démodulation

Arrivés à ce stade, nous vous fournissons un assembleur que nous avons écrit ainsi que ses sources. Il se trouve dans le dossier `numerique/asm`. Il faudra le recompiler :

— au CIME sur Linux : ouvrir un terminal et aller dans le dossier `numerique/asm` puis taper `make`
— à PHELMA sur Windows : double cliquer sur le fichier `bash.bat` puis taper `make` dans le terminal

Pour votre culture il semble intéressant de s'intéresser à ce programme. Il utilise l'analyseur lexical lex (fichier `asm/asm.lex`), l'analyseur grammatical yacc (fichier `asm/asm.y`) et une bibliothèque permettant de construire des listes chaînées (répertoire `util`). Pour plus d'informations sur lex et yacc, vous pouvez consulter cette page : http ://www.linux-france.org/article/devl/lexyacc/.
Si vous avez modifié le codage des instructions, par exemple pour réserver plus de place pour coder l'offset des instructions de branchement, il faudra le prendre en compte dans l'assembleur.

Travail à faire :

1. Le listing 3 présente un pseudo code C de démodulation qui travaille selon le principe présenté sur la figure 3.
2. Réaliser le programme final graduellement en validant chaque étape (réglage du seuil, démodulation) par simulation, ce qui peut nécessiter de modifier le banc de test, puis sur carte.
3. Pour assembler un programme il faut utiliser la ligne de commande suivante si vous êtes dans le répertoire du binaire (Attention, il faut que vos programmes assembleur se terminent par le mot clef `END`) :
   — au CIME sur Linux `./assembler -i fichier_asm -o fichier_vhd -t VHDL`
   — à PHELMA sur Windows `./assembler.exe -i fichier_asm -o fichier_vhd -t VHDL`

```
// @rom   0x00--
// @ram   0x01--
// @hex   0x02--
// @ledr  0x03--
// @adc   0x04--
// @dac   0x05--
// @key   0x06--
// @sw    0x07--
// @thres 0x0100
// @timer 0x0101

thres = INIT_THRES;
timer = INIT_TIMER;
hex = thres;

while( 1 ) {

  if ( key_0 == 0 ) {
    thres++;
  } else if ( key_1 == 0 ) {
    thres--;
  }
  hex = thres;
  while( key_0 == 0 || key_1 == 0 );

  if ( adc > thres ) {
    dac = 0;
    timer = INIT_TIMER;

  } else if ( timer != 0 ) {
    timer--;
    if ( timer == 0 ) {
      dac = 255;
    }
  }
}
```

Listing 3 – Pseudo code C de démodulation

5.3.4 Pour aller plus loin

Pistes d'amélioration :

1. Modifier le matériel pour pouvoir détecter les appuis sur les touches `key` non pas sur niveau, mais sur fronts.
2. Si cela n'a pas été fait, réserver plus de place coder l'offset dans les codes des instructions de branchement. Il faudra modifier l'assembleur en conséquence.
3. Faire en sorte que le registre r_0 puisse être destination des instructions add, sub, and, ld.
4. Compléter le jeu d'instructions.
5. Voir s'il est possible d'optimiser la machine à états de la partie commande.