Démodulation de signaux par un système intégré

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, etc. 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 prérequis.

1 Présentation du système de démodulation et de son architecture

1.1 La modulation FSK

L'objectif du TP 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. Par exemple, le '0' est codé par une fréquence F1 et le '1' par une fréquence F2.

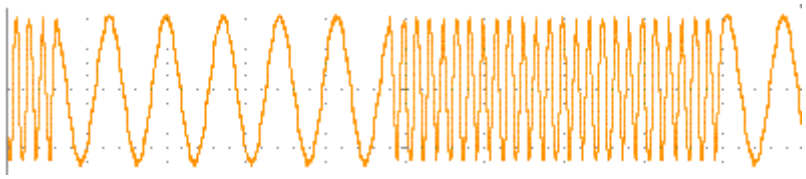

Fig 1: Signal numérique modulé en FSK

Si '1' correspond à la fréquence la plus élevée alors on peut lire la séquence suivante : 1010

1.2 L'architecture du système

Afin de réaliser ce système de démodulation numérique nous allons mettre en œuvre une chaîne de traitement de signal très simple dont la compréhension ne nécessite pas de pré-requis. La première étape de la démodulation consiste à utiliser un filtre passe-bas pour créer une modulation d'amplitude sur le signal. En effet, la fréquence F1 (la plus basse) de notre modulation FSK ne sera pas atténuée par notre filtre passe-bas alors que la fréquence F2 (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 traduit dans le domaine digital, le signal est traité par un petit processeur dont le programme détecte si l'amplitude du signal correspond à une zone où la fréquence est F1 (c'est-à-dire un '0') ou F2 (c'est-à-dire un '1').

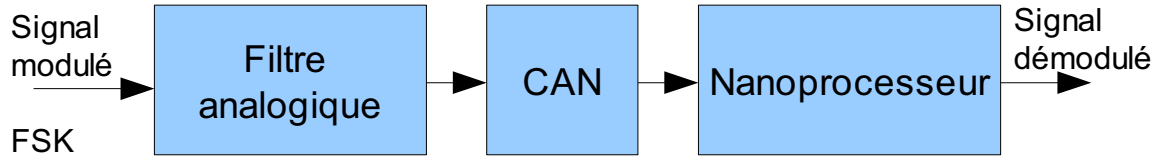

Fig 2: La chaîne de démodulation

2 La partie analogique

La partie analogique du système est composé 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 filtre dans le domaine fréquentiel (Cf. figure 3Erreur : source de la référence non trouvée). Ce gabarit définit notamment les zones (hachurées sur la figure) dans lesquelles le spectre du signal filtré ne devra pas se trouver.

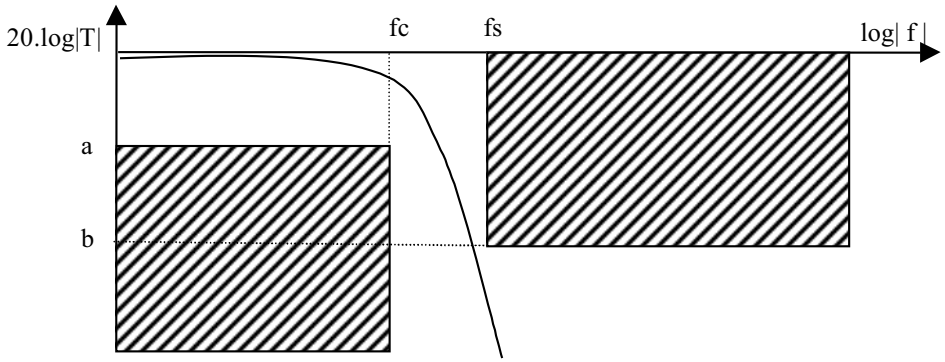

Fig 3: Gabarit du filtre

Dans l'application envisagée, nous utilisons un tel gabarit avec comme valeurs :

a = - 3dB
b = - 20 dB
fc = 1 kHz (fréquence de coupure)
fs = 3 kHz (fréquence d'arrêt)

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. La fonction de transfert du filtre est alors donnée par :

$$T(f)=\frac{1}{\sqrt{1+\varepsilon^2 C_n^2(f)}}$$

avec *Cn(f)* tel que $C_0(1) = 1$, $C_1(f) = f$ et $C_{n+1}(f) = 2f.C_n(f)-C_{n-1}(f)$.
La fonction de transfert de Chebychev correspondante est donnée surla figure 4.

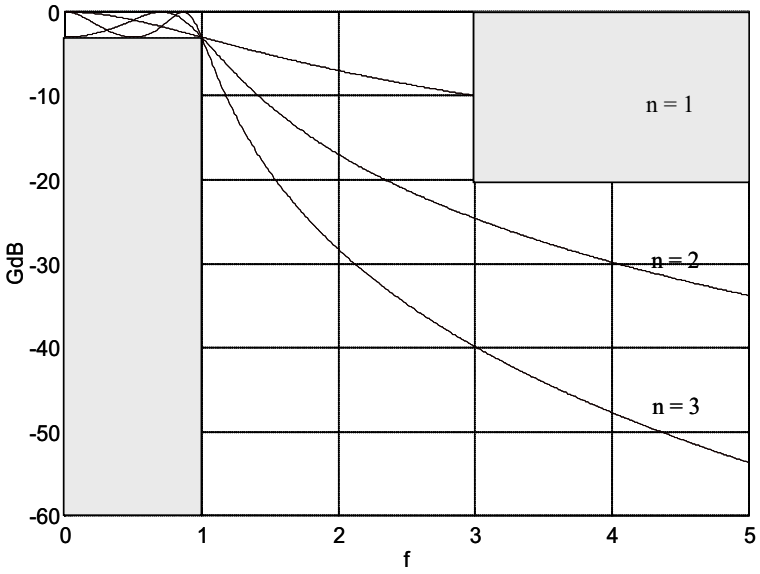

Fig 4: Abaque donnant les fonctions de Chebychev pour les ordres 1, 2 et 3

2.1 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. Calculer la valeur du coefficient epsilon
2. Grâce à l'abaque, déterminer l'ordre du filtre le plus approprié.
3. Écrire la fonction de transfert de notre filtre en variable de Laplace. Pour cela, il est important de ne conserver que les pôles à partie réelle négative (pour des raisons de stabilité) (réponses : A1 = -0.3218+0.7769i et A2 = -0.3218-0.7769i)

2.2 Conception matérielle du filtre

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. Le schéma utilisé pour réaliser notre filtre suit le modèle de Salen & Key. Le montage envisagée est présentée sur la figure 5et montre

une structure constitué de 4 admittances (Y1, Y2, Y3, Y4), de 2 résistances (R1, R2) et d'un amplificateur opérationnel.

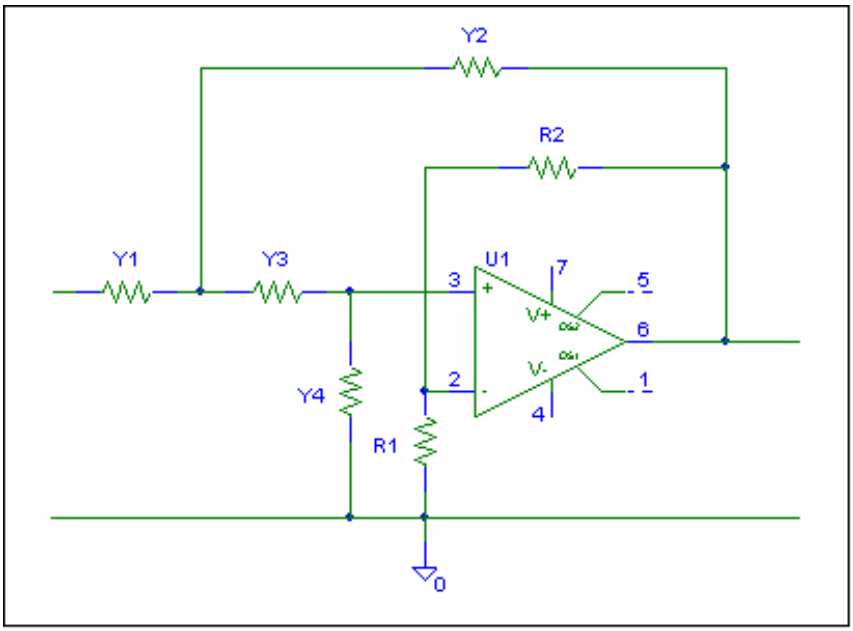

Fig 5: Schéma de principe d'un filtre de Salen & Key

1. Montrer que la fonction de transfert de ce filtre vaut :

$$T = \frac{KY1Y3}{(Y1+Y2)(Y3+Y4)+Y3(Y4-KY2)} \quad \text{avec} \quad K = \frac{R1+R2}{R1}$$

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 faire cette opération, il faut choisir des valeurs en tenant compte des valeurs de résistances et de condensateurs disponibles. Le choix de certaines valeurs peut-être fait de façon arbitraire mais devra tout de même suivre une certaine logique ! On vous demande donc de justifier votre démarche.

2.3 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 ce faire, vous devez utilisez la platine analogique pré-câblée (platine verte). Le schéma donné sur la figure 6 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 œil sur la photo (Figure 7).

Travail à effectuer :

1. Câbler votre montage et injecter un signal de fréquence 100 Hz et d'amplitude 2 V (-1V / +1V) sur le connecteur analogique d'entrée de la carte. Ajuster votre signal de sortie avec le potentiomètre associé à l'amplificateur de sorte que son excursion varie entre -1.25 et +1.25 V.
2. Observer la sortie du filtre avec des signaux sinusoïdaux et rectangulaires à diverses fréquences. Obtient-on bien un filtre passe-bas ?
3. Tracer les diagrammes de Bode en amplitude et en phase. Vérifier que l'on respecte bien le gabarit donné pour la spécification.

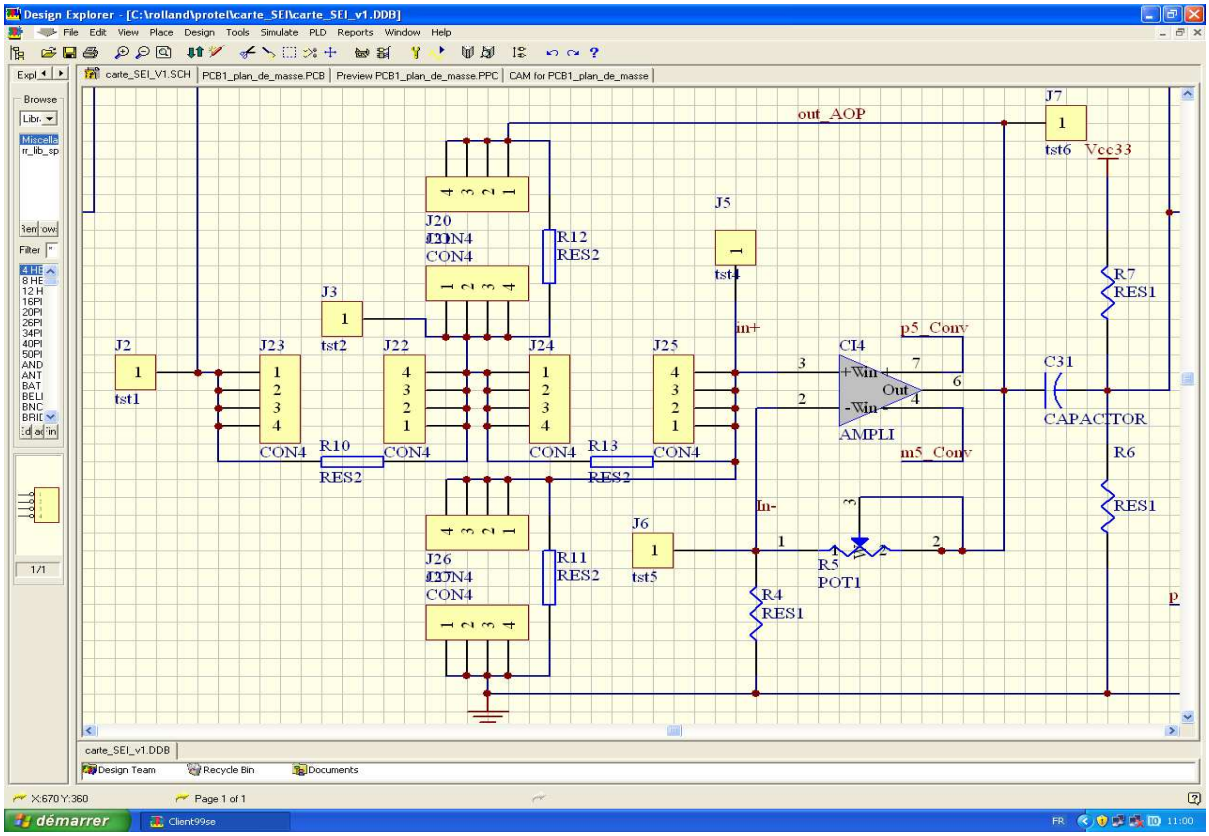

Fig 6: Schéma du filtre de Salen & Key

2.4 Analyse du dispositif de conversion analogique-numérique et de conversion numérique-analogique

Dans cette partie, il est demandé d'observer les signaux de contrôle utilisés pour le CAN (convertisseur analogique-numérique) et le CNA (convertisseur numérique-analogique).
Pour ce faire, il faut commencer par charger un fichier de programmation du FPGA sur la carte DE1. Ce fichier de programmation contient une petite application matérielle numérique qui pilote les CAN et CNA. Les échantillons prélevées sur le CAN sont alors recopiés sur le CNA. On observe ainsi des signaux numérisés, puis convertis à nouveau dans le domaine analogique.

Travail à effectuer :

1. Charger le code de programmation du FPGA avec le logiciel Quartus (onglet tools, puis programmer). Le fichier de programmation s'appelle test_can_cna.sof.
2. Observer les signaux conv_stb, cs_rdb, eocb, wr_csb et les signaux en entrée du CAN et en sortie du CNA.
3. Tracer un chronogramme de ces signaux et valider leur comportement avec les datasheets du CAN et du CNA.

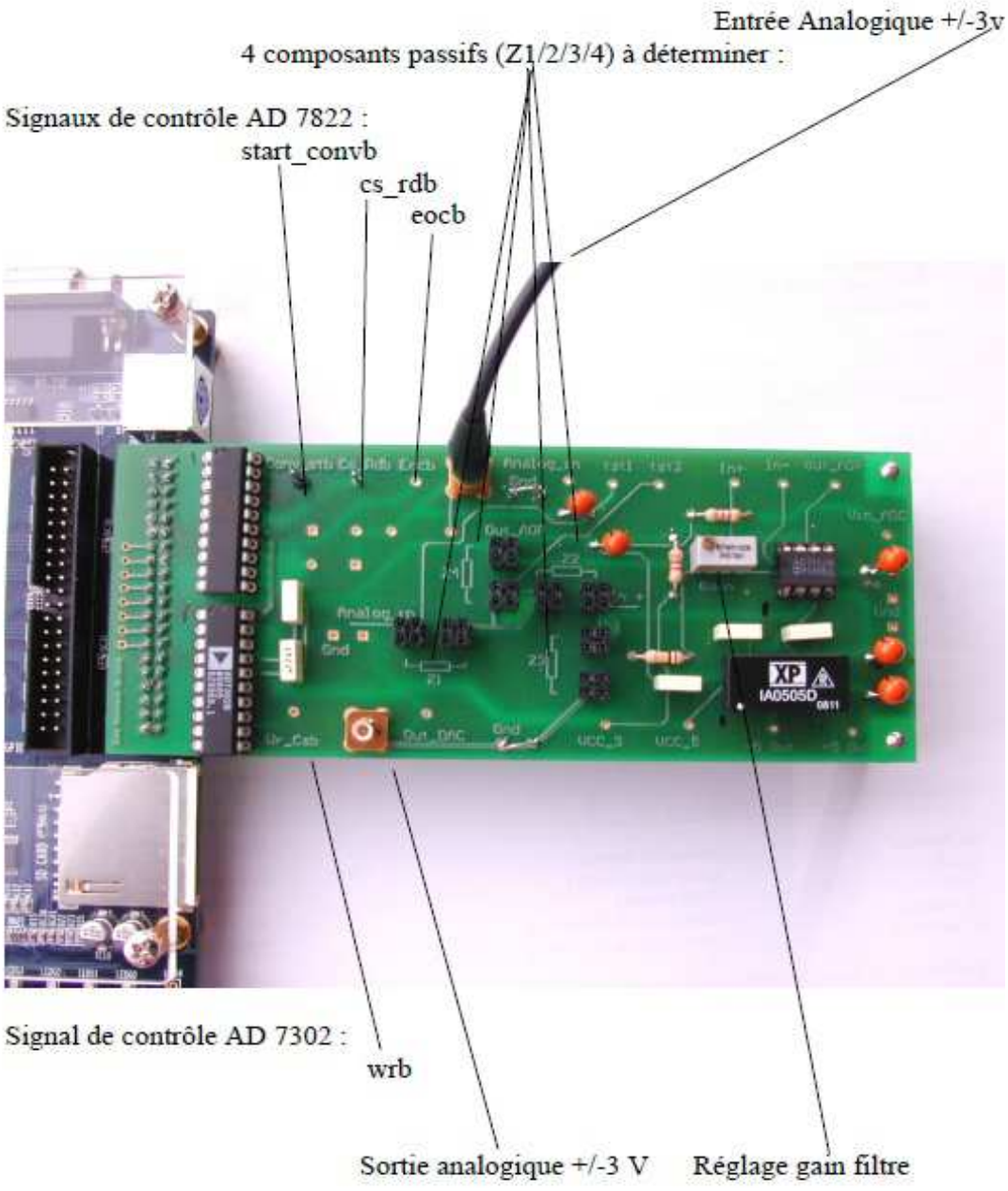

Fig 7: Photo de la platine analogique

3 La partie processeur numérique

3.1 Introduction

Un processeur est une machine programmable, capable d'effectuer des millions d'opérations par seconde. Il constitue le cœur des ordinateurs qui exécute les instructions et traite des données numériques.

Notre objectif est d'introduire la notion de processeur à travers un exemple de processeur élémentaire (baptisé *nano_processeur*). Nous allons réaliser un système complet qui permet l'exécution de programmes ; l'ensemble étant implanté sur une carte FPGA DE1 d'Altera.

On souhaite disposer d'un processeur organisé autour de 8 registres de données (dont un est mobilisé pour le Program-Counter), une unité arithmétique et logique munie d'un registre accumulateur et un registre de résultat, d'un bus externe capable d'adresser 256 mots, le tout étant cadencé par un séquenceur qui permet l'exécution des instructions du programme. Un signal d'horloge de 50 MHz est disponible sur la carte DE1.

Le coeur du projet est donc la réalisation d'un processeur qui communique avec un espace mémoire selon la figure 8:

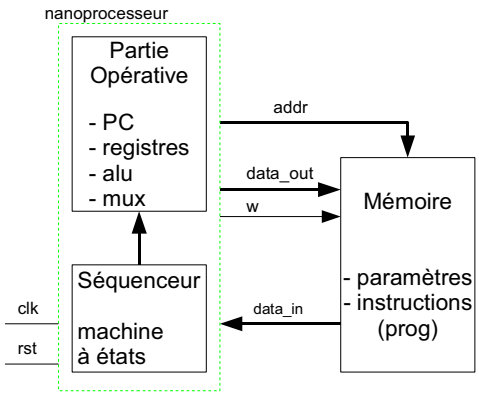

Fig 8. Le nano_processeur et son espace mémoire

Ce processeur est intégré dans un circuit logique programmable (FPGA) composé de cellules logiques élémentaires programmables par l'utilisateur.

Nous utilisons un kit de développement DE1 d'Altera, qui dispose d'afficheurs 7 segments à LED, de boutons poussoirs, d'une horloge à 50 MHz, de connecteurs d'entrées et sorties logiques. Une carte d'extension a été préparée pour ce BE afin de traiter les signaux analogiques (signal modulé en entrée et signal démodulé en sortie) comme suit :

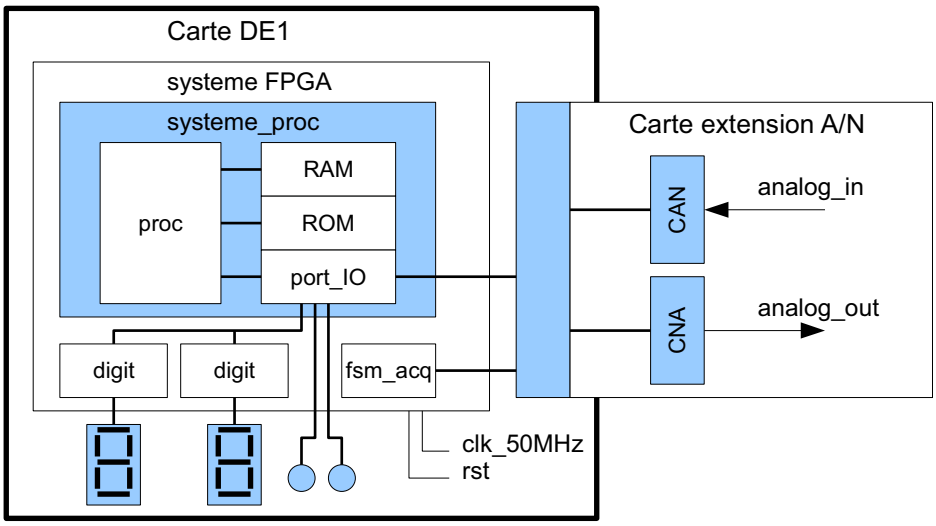

Fig 9.Système complet intégré au kit de développement DE1

Nous présentons dans la suite les éléments de spécification du processeur à concevoir issus du cahier des charges indiqué succinctement dans l'introduction. La description des principaux étages constitue l'étape de conception fonctionnelle. Nous en déduisons une architecture matérielle pour le nano_processeur.

Ce processeur s'inscrit dans un système processeur constitué du processeur et des périphériques.

L'ensemble est lui-même inclus dans le système FPGA incluant les interfaces aux éléments de la carte (afficheurs, boutons, entrées/sorties logiques) qui permet de générer la configuration du circuit logique programmable (cyclone II 2C20F484 d'Altera qui comprend 18752 Logic Elements, 240 Kbit de RAM, 315 I/O).

3.2 Spécification du processeur

3.2.1 L'environnement du processeur

L'environnement du processeur à concevoir est constitué :

- de mémoires permettant de stocker les instructions de programme ainsi que les données traitées
- de différentes entrées/sorties numériques afin de connecter boutons poussoirs, afficheurs, convertisseur AN.

Une horloge (clk) et un signal de remise à zéro (rst) sont indispensables pour assurer un fonctionnement synchrone.

3.2.1 Les entrées/sorties

La description de l'environnement du processeur permet de préciser ses entrées/sorties. Le processeur communique avec l'extérieur à l'aide d'un bus (ensemble de signaux d'adresses, de

données et de contrôle). Ce bus est organisé en 8 bits d'adresses et 16 bits de données ; les données sont séparées pour les entrées et les sorties du processeur.

Les signaux d'entrées/sorties du processeur sont donnés dans le tableau 1 :

nature	nom	type	détail
entrée	clk	boolean	horloge générale
entrée	rst	boolean	remise à 0
entrée	data_in	array[0..15] of boolean	données de la mémoire
sortie	addr	array[0..7] of boolean	adresse pour accès à la mémoire
sortie	data_out	array[0..15] of boolean	donnée à écrire en mémoire
sortie	w	boolean	validation écriture en mémoire

Tableau 1. Signaux d'entrées/sorties du processeur

3.2.2 La liste des instructions

Un jeu d'instruction est disponible pour rédiger le programme à exécuter sur le nano_processeur (tableau 2).

Le mode d'adressage immédiat permet de charger une valeur dans un registre (instruction load immédiat – ldi). Dans ce mode, l'instruction est codée sur 2 mots, le 2ème mot désigne la donnée immédiate stockée sur 16 bits.

Le mode d'adressage registre désigne les transferts entre registres (instruction mv) ; ils peuvent être conditionnels avec les instructions mvnz et mvgt, en fonction du résultat de la dernière opération de l'ALU stockée dans le registre Rg.

Le mode d'adressage indirect permet l'accès à une donnée située en mémoire dont l'adresse est contenue dans un registre, en lecture (instruction load -ld) ou en écriture (instruction store – st).

Plusieurs instructions de branchement sont présentes pour les ruptures de séquence sans condition (branch always – bra) ou selon le résultat de la dernière opération de l'ALU stockée dans Rg. Le déplacement est codé sur 6 bits en complément à 2, il est donc compris entre -32 et +31 à partir de PC+1, si PC pointe le code op du branchement.

Instruction	opération	Mot 1 (6 + 10 bits) code	champ X	champ Y	Mot 2 (16 bits) opérande	nombre cycles	Drapeaux
mv Rx,Ry	Rx <- Ry	0000	xxx	yyy		3	
ldi Rx,#YYYY	Rx <- #YYYY	0001	xxx	-	YYYY(hex)	4	
add Rx,Ry	Rx <- Rx + Ry	0010	xxx	yyy		5	Z G
sub Rx,Ry	Rx <- Rx - Ry	0011	xxx	yyy		5	Z G
ld Rx,[Ry]	Rx <- [Ry]	0100	xxx	yyy		4	
st Rx,[Ry]	[Ry] <- Rx	0101	xxx	yyy		4	
mvnz Rx,Ry	Rx <- Ry si Rg ≠ 0	0110	xxx	yyy		3	
mvgt Rx,Ry	Rx <- Ry si Rg ≥ 0	0111	xxx	yyy		3	
and Rx,Ry	Rx <- Rx and Ry	1000	xxx	yyy		5	Z G
bra offset	PC <- PC + 1 + offset	1001	offset			3	
brnz offset	Si Rg ≠ 0 PC <- PC + 1 + offset sinon PC <- PC + 1	1010	offset			3	
brgt offset	Si Rg ≥ 0 PC <- PC + 1 + offset sinon PC <- PC + 1	1011	offset			3	
brz offset	Si Rg = 0 PC <- PC + 1 + offset sinon PC <- PC + 1	1100	offset			3	
brmi offset	Si Rg < 0 PC <- PC + 1 + offset sinon PC <- PC + 1	1101	offset			3	

Tableau 2. Liste des instructions du nano_processeur

Une instruction de programme est codée sur 10 bits, dont les 4 bits de poids forts définissent l'opération à exécuter, 3 bits suivants le numéro du registre Rx et les 3 derniers bits désignent le numéro du registre Ry.

Le code d'une instruction est de la forme :

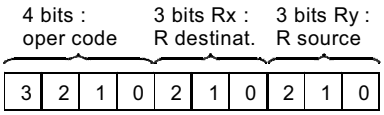

Fig 10. codage d'une instruction sur 10 bits

3.2.3 Exemple de programme

Afin d'illustrer la façon de coder les instructions du *nano_processeur*, un programme qui effectue l'addition de deux registres est donné ci-dessous (avec des mots de 16 bits codés en hexa) :

adresse	code op	étiquette	instruction	commentaire
0x00	0040 AAAA	debut	ldi r0,#AAAA	charge AAAA dans r0
0x02	0058 000F		ldi r3,#000F	charge 000F dans r3
0x04	0083		add r0,r3	r0 ← r0 + r3
0x05	027F	boucle	bra boucle	boucle infinie

Tableau 3. Liste assemblée du programme d'addition de deux registres

Dans la mémoire instruction, ce programme apparaît à partir de l'adresse 0x00 :

```
0x00 : 0040 AAAA 0058 000F 0083 027F
```

3.3 Conception fonctionnelle

L'étape de conception fonctionnelle consiste à élaborer une description des aspects fonctionnels du dispostif en restant indépendant de la technologie.

Nous détaillons ci-après les éléments principaux du nano_processeur.

3.3.1 L'unité Arithmétique et Logique (UAL – ALU)

A partir de 2 opérandes entières codées sur 16 bits, l'ALU génère le résultat d'une opération arithmétique (ou logique), cette opération étant précisée par le code opératoire : *alu_code*.

Le fonctionnement de l'ALU est précisé à l'aide de l'algorithme suivant :

```
action alu with
        ( input a,b             : integer range
0..65535;
          input alu_code        : integer range
0..2;
          output r              : integer range
0..65535 );

begin
        case alu_code is
        ALU_ADD :
                r:= a + b ;
        ALU_SUB :
                r:= b - a ;
        ALU_AND :
                r:= a and b ;
        end case;
end;
```

Fig 11. Algorithme de description de l'ALU

Les opérandes peuvent être codées en entier non signé ou en complément à deux pour représenter des nombres relatifs.

3.3.2 Les drapeaux

Certaines instructions sont sensibles au résultat de la dernière opération de l'ALU. Deux drapeaux sont positionnés en fonction du contenu du registre Rg (résultat de la dernière opération add, sub ou and) :

On a :

•Z_bit = 1 si Rg = 0

•G_bit = 1 si Rg ≥ 0

L 'exécution des instructions mvnz, mvgt, brnz, brgt, brz et brmi dépend de la valeur de ces deux bits Z_bit et G_bit.

3.3.3 gestion du Program Counter (PC)

Au reset, le registre PC doit être initialisé à 0. Le programme exécutable commence à l'adresse 0 de la mémoire.

Le registre PC est l'un des 8 registres du processeur ; il peut être utilisé comme les autres registres en lecture ou en écriture. Plusieurs signaux issus du séquenceur affectent le registre PC :

–R_ld(7) : entrée de chargement du registre R(7) = PC : l'activation de cette entrée entraîne le chargement du PC avec la valeur du bus interne nanobus (branchement en mode absolu avec l'instruction mv).

–inc_PC : entrée d'incrémentation du registre PC

–br_PC : entrée indiquant une instruction de branchement (branchement relatif avec offset codé dans l'instruction sur 6 bits en complément à 2).

Le fonctionnement du registre PC est précisé figure 12 :

```
action gestion_PC sensible event h with
        ( input nanobus, offset                 : integer
range 0..65535;
          input R_ld(7), inc_PC, br_PC,rst      :
boolean;
          output PC                             : integer range
0..65535 );

begin
        cycle h :
                if ( rst )          then PC := 0;
                end if;
                if ( inc_PC )       then PC := PC+1;
                end if;
                if ( br_PC )        then PC := PC + offset;
                end if;
                if ( R_ld(7) )      then PC := nanobus;
                end if;
        end cycle;
end;
```

Fig 12. Algorithme de description de l'ALU

3.4 Architecture

La conception architecturale consiste à adopter une solution d'implantation de la conception fonctionnelle exprimée précédemment.

Le nano_processeur est constitué d'une partie opérative et d'un séquenceur, le coeur de la partie opérative étant l'ALU qui traite des données de 16 bits.

3.4.1 La partie opérative

La partie opérative rassemble principalement les 8 registres de données (16 bits), l'ALU et deux registres A (accumulateur) et G(résultat), un multiplexeur (MUX) qui positionne le bus interne « nanobus ».

Le bus interne (nanobus) permet l'acheminement des données à l'ensemble des registres du processeur. Ce bus est positionné par la sortie du multiplexeur dont les entrées sont les différents registres internes.

Deux autres registres Rad (8 bits, adresses externes) et Rd (16 bits, données en sortie), un registre Ri de 10 bits pour l'instruction et deux bits d'état (Z_bit) et (G_bit) complètent la partie opérative.

3.4.2 Le séquenceur

Le séquenceur est la partie de commande du processeur. A partir du code de l'instruction à exécuter, et des bits d'état (Z_bit et G_bit), le séquenceur positionne l'ensemble des signaux de commande de la partie opérative.

3.4.3 Synoptique du nano_processeur

Nous adoptons l'architecture suivante, bâtie autour d'un bus central « nanobus ».

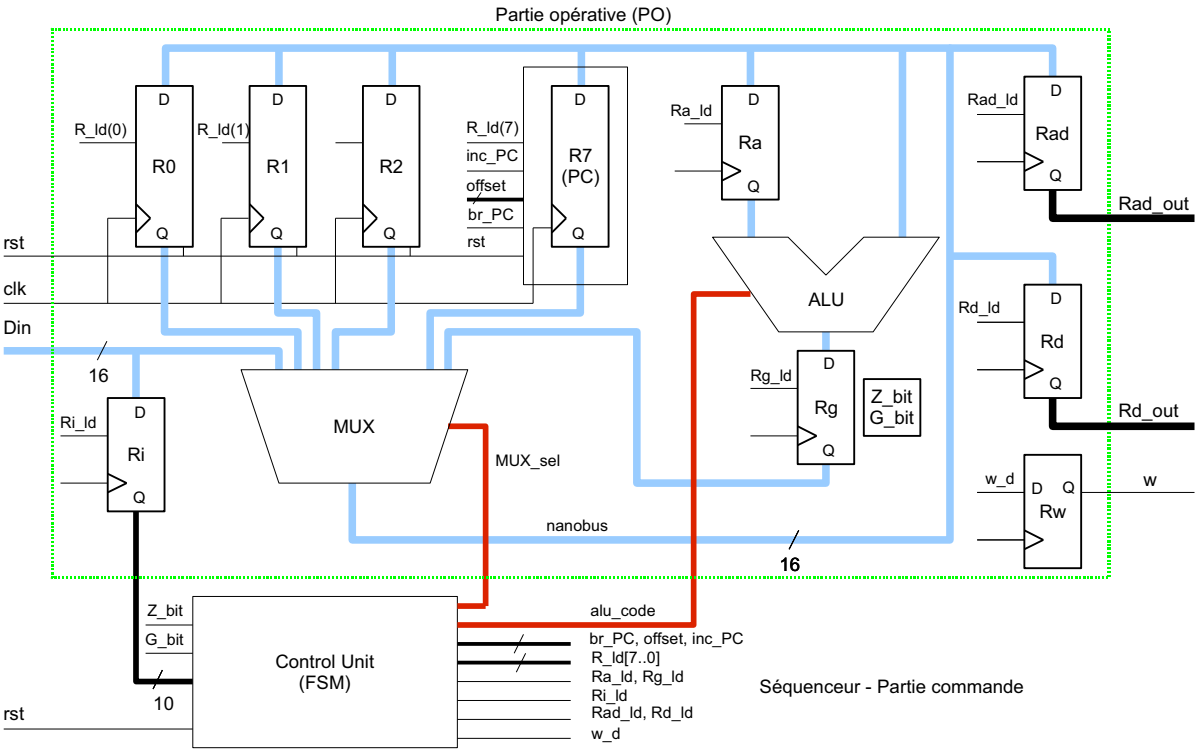

Fig 13. Architecture interne du nano_processeur

3.4.4 Exécution des instructions

L'exécution d'une instruction commence par l'accès au code instruction situé en mémoire. Le cycle fetch1 permet de sortir la valeur du registre PC sur le bus d'adresses externe pour accéder au code de l'instruction. Le cycle fetch2 est celui de la lecture de la donnée issue de la mémoire qui représente le code de l'instruction.

•Instruction mv Rx,Ry :

L'exécution de l'instruction mv consiste à lire la valeur de Ry et positionner le bus interne nanobus (cycle exe1). Au cours de ce cycle, le signal Rx_ld est actif pour permettre l'écriture de la valeur du nanobus dans le registre Rx lors du prochain front actif d'horloge. Le tableau 4 indique le contenu de nanobus et le niveau logique de R_ld(x) au cours des différents états du séquenceur.

	fetch1	**fetch2**	**exe1**	**fetch1**
nanobus	PC	-	Ry	PC + 1
Rx_ld	0	0	1	0
cycle	accès instr.	lecture instr.	charge Ry	Ecrit Rx + suite

Tableau 4. Séquence d'exécution d'une instruction mv Rx,Ry en 3 cycles

•Instruction ld Rx,[Ry] :

L'exécution de l'instruction ld Rx,[Ry] consiste à lire la valeur de Ry et positionner le bus interne nanobus (cycle exe1). Au cours de ce cycle, le signal Rad_ld est actif pour sortir cette valeur de Ry sur le bus d'adresse du processeur. Le cycle exe2 concerne la lecture de la donnée stockée à l'adresse mémoire contenue dans le registre Ry. Cette donnée est ensuite stockée dans Rx (Rx_ld est actif, voir ci-dessus). Le tableau 5 indique le contenu de nanobus et le niveau logique de R_ld(x) au cours des différents états du séquenceur.

	fetch1	**fetch2**	**exe1**	**exe2**	**fetch1**
nanobus	PC	-	Ry	din = [Ry]	PC + 1
Rx_ld	0	0	0	1	0
cycle	accès instr.	lecture instr.	accès [Ry]	charge din	Ecrit Rx + suite

Tableau 5. Séquence d'exécution d'une instruction ld Rx,[Ry] en 4 cycles

3.4.5 Machine à états du séquenceur

Voici le détail de la machine à états de la partie commande du nano_processeur, la transition entre les états s'effectue à chaque cycle d'horloge (clk), selon le code de l'instruction I :

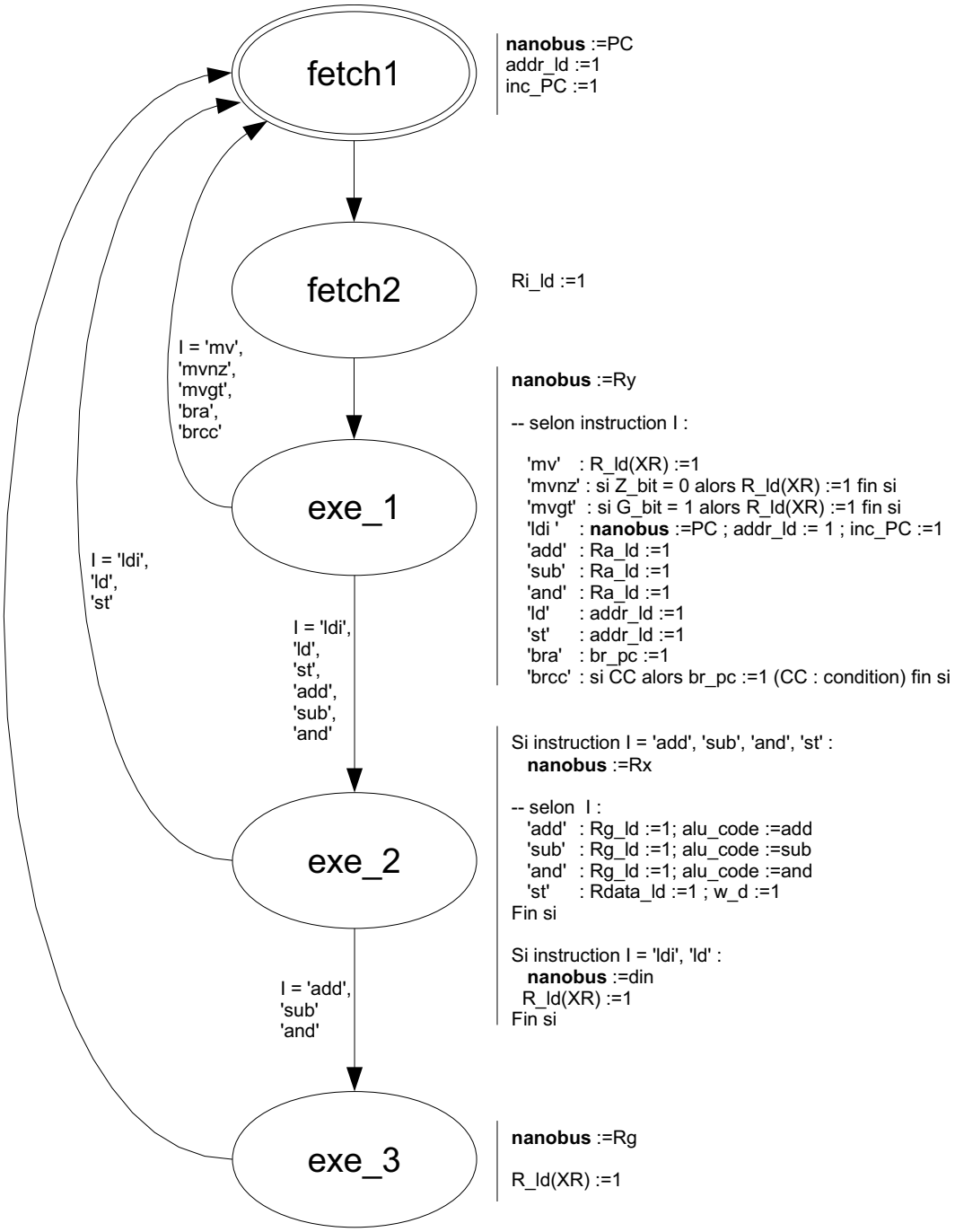

Fig 14. Machine à états du séquenceur

3.5 Le système nanoprocesseur

Un processeur ne peut fonctionner sans ses périphériques. Le nano_processeur est doté de mémoire ROM pour stocker le programme à exécuter, de mémoire RAM pour stocker des paramètres variables et de ports d'entrées/sorties afin de dialoguer avec l'extérieur. La figure15 illustre le système nano_processeur :

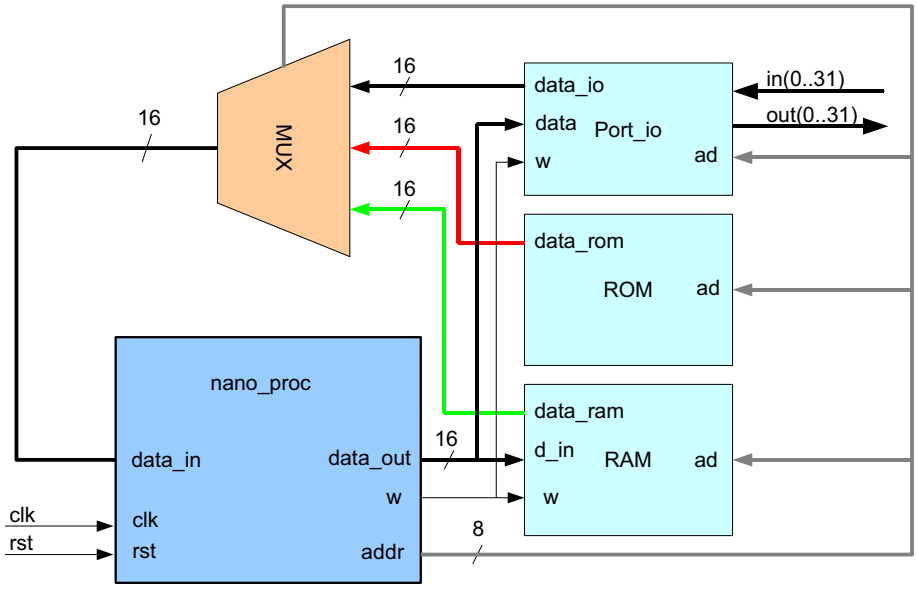

Fig 15. Le système autour du nano_processeur

Le bus d'adresses (8 bits) dessert les mémoires et périphériques. Un multiplexeur avec deux bits de sélection assure le décodage d'adresse nécessaire pour sélectionner la donnée à entrer dans le nano_processeur.

Ce décodage d'adresse est effectué pour aboutir au plan mémoire suivant :

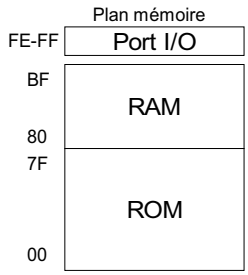

Fig 16. Plan de la mémoire du système

3.6 Le système complet

Il est nécessaire d'ajouter les interfaces avec l'extérieur pour terminer la conception du circuit logique programmable. Nous disposons de 32 bits (deux mots de 16 bits) en entrée et en sortie.

La figure 17 détaille l'affectation des bits d'entrée-sortie du système processeur.

En entrée, il faut disposer de 2 bits relatifs aux boutons poussoirs et de 8 bits correspondant à l'échantillon du signal analogique d'entrée (CAN - AD7822).

En sortie, il est prévu d'afficher un octet en hexadécimal sur 2 afficheurs 7 segments et de fournir un échantillon au convertisseur numérique analogique (CNA - AD7302).

Une machine à état (fsm_acq figure 14) permet de piloter l'acquisition des signaux analogiques en boucle, au rythme de 2 M échantillons par seconde.

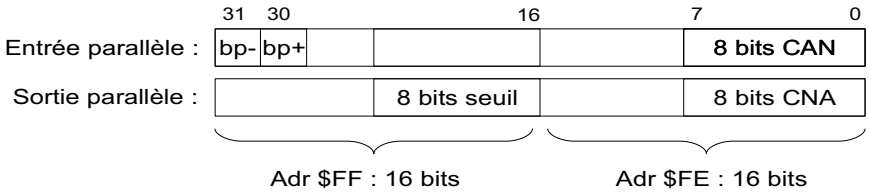

Fig 17. définition des entrées/sorties du système nano_proc

Attention à la polarité des boutons poussoirs : 1 logique = inactif, 0 logique = appui

3.7 Réalisation

Dans le cas général de la conception d'un système numérique, la réalisation consiste à décrire le système à concevoir à l'aide d'un langage de description de matériel. Après validation par simulation, cette description subira des étapes d'élaboration jusqu'à pouvoir être implantée sur la cible technologique.

En ce qui nous concerne, l'ensemble du système est décrit en langage VHDL.

La solution fournie en annexe est validée par simulation à ce premier niveau de description dit "fonctionnel". A cet fin, plusieurs bancs de test (testbench) sont définis.

Une fois validée, cette description est synthétisée et placée/routée puis validée avec les même bancs de test. Contrairement à la première simulation cette simulation pour validation après placement routage prend en compte les temps de propagation dans le circuit.

3.7.1 Décomposition hiérarchique du système

Le projet que nous vous fournissons est constitué de toute la hiérarchie définie au cours des étapes de

conception précédentes. La structure est la suivante :

–Le module top est le module de plus haut niveau hiérarchique : system_fpga.vhd

–system_proc.vhd, rom_decod.vhd

–proc.vhd, rom.vhd, ram.vhd, port_io.vhd

–alu.vhd, dec3to8.vhd

On a regroupé des constantes et la déclaration des registres de donnée dans le package : nano_pkg.vhd

L'ensemble des fichiers vhdl est fourni en annexe.

3.8 Travail demandé

Nous allons concevoir le système complet (nano_processeur et ses périphériques et système FPGA) pour mettre en oeuvre l'ensemble dans le circuit logique programmable de la carte DE1 d'Altera.

Au cours des séances de travail, nous allons étudier le fonctionnement de l'unité arithmétique et logique, simuler la séquence d'exécution de plusieurs programmes, intégrer et réaliser un prototype complet sur la carte de développement.

3.8.1 - séance - L'unité arithmétique et logique du processeur

Le code VHDL décrivant l'ALU est donné en annexe 1 . A partir de l'exemple de fichier bench.vhd, il est demandé :

•d'écrire un fichier bench.vhd permettant de simuler sous « modelsim » le fonctionnement de l'ALU

•de modifier le code VHDL de l'ALU pour ajouter l'instruction 'not'

•de simuler ce nouvel ALU

•d'indiquer les ressources utilisées lors de la synthèse de cet ALU avec l'outil « precision »

3.8.2 séance - Le séquenceur - la machine à états

L'objet de la séance est de détailler l'exécution d'un premier programme sur le nano_processeur. Il est demandé de calculer la somme des j premiers entiers :

$$S=\sum_{i=0}^{j} i$$

à l'aide des instructions du tableau 2, une boucle sera utilisée sans test de fin.

•écrire ce programme à l'aide des instructions disponibles

•assembler le programme et modifier le code rom.vhd

•simuler l'exécution du programme à l'aide du fichier bench.vhd

3.8.3 séance - Programmation du nano_processeur

On souhaite réaliser un programme de démodulation du signal FSK. Le principe utilisé est très simple puisque le filtre analogique situé en amont a pour objet de couper la fréquence haute du signal modulé FSK. Seule la composante BF du signal est présente et convertie par le CAN.

La démodulation proposée consiste à détecter la présence de la composante BF, à partir d'un seuil programmable. Cette détection positionne la sortie à l'état 0. En l'absence de signal (le signal analogique reste inférieur au seuil) la sortie du démodulateur est à 1.

Dans un premier temps, nous allons réaliser une boucle permettant de modifier le seuil de détection. A partir des boutons poussoirs, il est possible de faire varier la valeur du seuil de détection. Cette valeur du seuil est affichée en hexadécimal sur 2 afficheurs 7 segments. Ecrire le programme réalisant cette fonction.

Ensuite, il est nécessaire d'effectuer la comparaison du signal entrant avec le seuil. Ce signal étant de période longue comparativement au temps d'exécution d'une boucle de programme, il sera nécessaire d'introduire un timer afin de mémoriser l'état 0 pendant au moins une période du signal périodique de la composante BF.

Il est demandé de simuler le programme correspondant à l'algorithme donné en annexe 7.

3.8.4 Séance - Intégration et prototypage du nano_processeur

Le but est de réaliser le prototype complet sur la carte DE1 d'Altera.

Nous allons réaliser la synthèse et le placement routage du système FPGA, pour générer un fichier de programmation du circuit logique programmable.

Il est demandé dans un premier temps de générer le fichier de programmation du FPGA pour le système d'ajustement du seuil avec les boutons poussoirs (sans mettre en oeuvre les convertisseurs A-N).

Il sera mis à disposition un fichier vhdl permettant de reboucler l'entrée numérique de la carte (donnée numérique d'entrée convertie par le CAN) sur la sortie (donnée numérique de sortie délivrée au CNA). Effectuer la programmation du FPGA pour réaliser ce test de signal rebouclé à l'aide du matériel mis à votre disposition (générateur BF et FSK, oscilloscope).

A présent, il est possible de tester l'ensemble du système intégrant le programme de démodulation FSK.

Générer le fichier de programmation du FPGA et tester l'ensemble du système.

3.9 Références

- Documents relatifs à la carte DE1 d'Altera :

http://www.terasic.com/downloads/cd-rom/de1/

- Data Sheet des circuits de la famille Cyclone II d'Altera

http://www.altera.com/literature/lit-cyc2.jsp

- Cours de logique 1A Phelma : http://huet.phelma.grenoble-inp.fr

3.10 Annexes

Annexe 0 : Organisation des répertoires de travail et flot de conception

Annexe 1 : Le module top est le module de plus haut niveau hiérarchique : system.vhd

Annexe 2 : system_proc.vhd

Annexe 3 : rom_decod.vhd

Annexe 4 : proc.vhd

Annexe 5 : rom.vhd, ram.vhd, port_io.vhd

Annexe 6 : alu.vhd, dec3to8.vhd

Annexe 7 : algorithme du programme de démodulation FSK

Annexe 8 : schéma de la carte d'extension filtrage + conversion A/N

Organisation des répertoires de travail *Annexe 0*

Durant ce projet vous travaillerez dans le répertoire `nano-proc` qui se trouve à la racine de votre compte. Ce répertoire se structure comme suit:

```
+-nano-proc/                répertoire racine du projet
  |
  +-analog/                 répertoire qui contient
  |                         (1) un fichier de programmation du FPGA (system.sof)
  |                         (2) un script de programmation de la carte (script_pgm)
  |                         pour le test du filtre (sortie filtre>CAN>CNA)
  |
  +-alu/                    répertoire du projet alu de la partie numérique
  |
  +-alu_pb/                 répertoire du projet alu_bp de la partie numérique
  |
  +-doc/                    répertoire contenant les documents du projet
  |
  +-lib_altera/             répertoire contenant les bibliothèques de simulation après placement
  |                         routage pour le cycloneii
  |
  +-proc/                   répertoire du projet proc de la partie numérique
```

Les deux premières séances du projet sont dédiées à la partie analogique. Vous n'utiliserez alors que les répertoires `doc` et `analog`. Le répertoire `analog` contient un fichier de configuration du FPGA qui implante un système qui convertit la sortie du filtre analogique en numérique via le convertisseur analogique numérique et recopie le résultat de cette conversion sur le convertisseur numérique analogique. Pour programmer le FPGA avec ce fichier de programmation, il suffit de se rendre dans le répertoire `analog` et d'exécuter le script en tapant la commande `./script_pgm`
Les 4 séances suivantes sont consacrées à la partie numérique du projet. Vous serez alors amenés à travailler sur les 3 projets `alu, alu_bp, proc`, chacun se trouvant dans un répertoire propre. Lors de la première séance de tp numérique, vous prendrez le flot de développement en main avec le projet `alu`. Le circuit conçu dans ce projet est purement combinatoire. S'il vous reste du temps lors de cette première séance, vous pourrez également travailler sur le projet `alu_pb` qui introduit la dimension séquentielle du langage vhdl à l'aide d'un exemple de machine à états. Les séances suivantes, vous travaillerez sur le projet `proc`.
Chacun de ces trois projets est structuré selon l'arborescence suivante:

```
+-nom_projet/                répertoire racine du projet (nom_projet=alu ou alu_bp ou proc)
  |
  +-bench/                   répertoire de test du circuit
  | |
  | +-script                 script bash qui
  | |                        (1) crée la bibliothèque lib_nanoproc_bench
  | |                        (2) compile bench_system.vhd, cfg_bench_system.vhd
  | |
  | +-cfg_bench_system_pr.vhd configuration du banc de test pour simulation après placement routage
  | |
  | +-script_pr              script bash qui compile cfg_bench_system_pr.vhd
  | |
  | +-bench_system.vhd       banc de test du système
  | |
  | +-clean                  script bash qui nettoie le répertoire
  | |
  | +-modelsim_system.do     script modelsim qui
  | |                        (1) lance la simulation de la configuration cfg_bench_system
  | |                        (2) configure la fenêtre wave
  | |
  | +-cfg_bench_system.vhd   configuration du banc de test pour simulation fonctionnelle
  | |
  | +-modelsim_system_pr.do  script modelsim qui
  |                          (1) lance la simulation de la configuration cfg_bench_system_pr
  |                          (2) configure la fenêtre wave
  |
  +-config/                  répertoire de configuration de l'environnement
  | |
  | +-modelsim.ini           fichier modelsim.ini du projet
  | |
  | +-config_modelsim        script bash de de configuration de l'environnement pour
  |                          modelsim
  |
  +-pr/                      répertoire pour le placement routage
  | |
  | +-script                 script bash qui
  | |                        (1) copie le fichier edif system.edf obtenu après synthèse dans le
  | |                        répertoire courant
  | |                        (2) appelle quartus pour faire le placement routage et la
  | |                        génération des fichiers de simulation après placement routage
  | |                        (3) crée la bibliothèque lib_nano_proc_pr
  | |                        (4) compile le fichier system.vho pour la simulation après
  | |                        placement routage
  | |
  | +-script_pgm             script bash qui programme la carte avec le fichier system.sof
  | |
  | +-system.tcl             script tcl appelé par quartus, par script ci-dessus, pour
  | |                        la création du projet utilisé pour le placement routage
  | |
  | +-clean                  script bash qui nettoie le répertoire
  |
  +-synth/                   répertoire de la synthèse
  | |
  | +-synth_nanoproc.tcl     script tcl appelé par precision pour faire la synthèse, cf. le
  | |                        script de ce répertoire
  | |
  | +-clean                  script qui nettoie le répertoire
  | |
  | +-script                 script bash qui
  |                          (1) appelle le script clean
  |                          (2) appelle precision pour faire la synthèse
  |
  +-lib/                     répertoire des bibliothèques modelsim lib_nanoproc, lib_nanoproc_bench
  | |                        lib_nanoproc_pr
  | |
  | +-clean                  nettoyage du répertoire
  |
  +-vhd/                     répertoire des sources vhdl de description du système au niveau
  | |                        fonctionnel
  | |
  | +-script                 script bash qui
  | |                        (1) crée la bibliothèque lib_nanoproc
  | |                        (2) compile tous les fichiers vhd du répertoire
  | |
  | +-clean                  script bash qui nettoie le répertoire
  |
  +-clean                    script bash qui nettoie tous les répertoires du projet
  |
  +-script                   script bash qui déroule tout le flot
                             (1) configure l'environnement    (2) lance le script ./vhd/script
                             (3) lance le script ./bench/script (4) lance le script ./synth/script
                             (5) lance le script ./pr/script  (6) lance le script ./bench/script_pr
```

Prise en main de l'environnement de développement, exemple de l'alu

Le flot de développement repose sur 3 outils:
- l'outil ModelSim de Mentor Graphics pour la simulation des circuits aussi bien au niveau fonctionnel qu'après placement routage,
- l'outil Precision Synthesis également de Mentor Graphics qui est un outil de synthèse logique,
- l'outil Quartus d'Altera qui sera uniquement utilisé pour le placement routage, la génération des fichiers de programmation du FPGA et la programmation du FPGA. Quartus sera appelé en ligne de commande, au travers de scripts.

Remarques sur la prise en main du flot:
- toutes les commandes à entrer sont en `courrier new`
- les commandes à entrer dans le terminal sont précédées de l'invite `$`
- les commandes à entrer dans ModelSim sont précédées de l'invite `ModelSim>`
- simplifiez vous la vie, utilisez la complétion automatique (touche tab)
- ne pas hésiter à regarder le contenu des scripts et à demander leur explication

1 Sélection du projet

- aller dans le répertoire du projet alu

```
$ cd nano-proc/alu
```

2 Configuration de l'environnement de travail

- aller dans le répertoire config

```
$ cd config
```

- "sourcer" le fichier de configuration de modelsim (initialise des variables d'environnement utilisées par ModelSim et dans le fichier modesim.ini)

```
$ source config_modelsim
```

3 Travail au niveau fonctionnel

- aller dans le répertoire vhd

```
$ cd ../vhd
```

- éditer les fichiers vhd et essayer de les comprendre, de deviner ce que fait le système.

```
$ gedit alu.vhd
$ gedit system.vhd
...
```

- compiler les fichiers vhd

```
$./script
```

4 Validation au niveau fonctionnel

- aller dans le répertoire bench

```
$ cd ../bench
```

- éditer le banc de test, essayer de comprendre.

```
$ gedit bench_system.vhd
```

- éditer la configuration utilisée pour la simulation fonctionnelle, essayer de comprendre.

```
$ gedit cfg_bench_system.vhd
```

- compilation de ces deux fichiers vhd

```
$ ./script
```

- lancer ModelSim

```
$ vsim &
```

- exécuter le script qui lance la simulation et configure la fenêtre wave

```
ModelSim> do modelsim_system.do
```

- simuler

```
ModelSim> run 100 ns
```

- quitter modelsim

5 Synthèse

- aller dans le répertoire synth
```
$ cd ../synth
```
- lancer le script de synthèse
```
$ ./script
```
- attendre la fin de la synthèse et observer les résultats dans l'onglet transcript
- cliquer sur le menu Schematics situé sur la partie gauche de la fenêtre, puis cliquer sur "View RTL Schematic", observer le schéma.
- cliquer sur le "View Tech Schematic", observer le schéma, quelle est la différence avec le schéma précédent?
- quitter l'outil de synthèse, l'outil de placement routage utilisera le fichier `./project_nanoproc_impl/system.edf`

6 Placement routage

- aller dans le répertoire pr
```
$ cd ../pr
```
- lancer le script de placement routage
```
$ ./script
```
- le script de placement routage génère les fichiers de programmation du FPGA (sof et pof) et les fichiers pour la simulation du système après placement routage (`./simulation/modelsim/system.vho` et `./simulation/modelsim/system_vhd.sdo`)

7 Validation après placement routage

- aller dans le répertoire bench
```
$ cd ../bench
```
- éditer la configuration utilisée pour la simulation après placement routage et essayer de la comprendre. Quelle est la différence avec le fichier de configuration cfg_bench_system.vhd?
```
$ gedit cfg_bench_system_pr.vhd
```
- compiler la configuration
```
$ ./script_pr
```
- lancer ModelSim
```
$ vsim &
```
- exécuter le script qui lance la simulation et configure la fenêtre wave pour le système obtenu après placement routage
```
ModelSim> do modelsim_system_pr.do
```
- simuler
```
ModelSim> run 100 ns
```
- Quelles différences avec la simulation au niveau fonctionnel?

8 Programmation de la carte

- aller dans le répertoire pr
```
$ cd ../pr
```
- programmer la carte
```
$ ./script_pgm
```
- valider le fonctionnement du circuit