IUT de TROYES
Dpt GEII

2013-2014
S. Moutou et G. Millon

TP MCENSL1

Introduction : *on se propose au travers d'un petit projet constitué de plusieurs exercices successifs d'implanter sur FPGA un calcul numérique de sinus et de cosinus par l'algorihme CORDIC. Les angles seront fournis par un processeur AVR et on affichera les signaux sur écran VGA. On enverra sur une broche de la carte la conversion analogique du signal sinusoïdal. On mettra en oeuvre la liaison série de l'AVR pour gérer la période du signal.*

Dans une première partie nous travaillerons avec la carte Basys 2 de Digilent dotée d'un FPGA Xilinx Spartan3 XC3S250ECP132. Ensuite, nous travaillerons avec la carte spartan3E starter kit de DIGILENT dotée d'un FPGA Xilinx Spartan3E XC3S500EFG320. Les documentations complètes de ces cartes peuvent se trouver, via leur *user guide* au format .pdf, sur le site internet suivant :

- http ://www.digilentinc.com/Products/Detail.cfm ?NavPath=2,400,790&Prod=BASYS2
- *http ://www.digilentinc.com/Products/Detail.cfm ?NavPath=2,400,792&Prod=S3EBOARD*

Cependant, la grande majorité des informations nécessaires aux exercices sera donnée dans le polycopié. Dans les figures, les component sont représentés en bleu. Les fonctions logiques décrites sous forme de process ou autre sont représentées en rose. Les fonctions logiques à décrire dans l'exercice sont représentées en rouge. Les vecteurs sont représentés par des trais épais et dimensionnés le plus souvent. Cependant, les figures ne représentent pas toujours la totalité des interconnexions, notamment lorsqu'elles sont nombreuses et pas indispensables à la compréhension de l'exercice.

Exercice 1 : implanter et programmer un processeur AVR - chenillard sur led

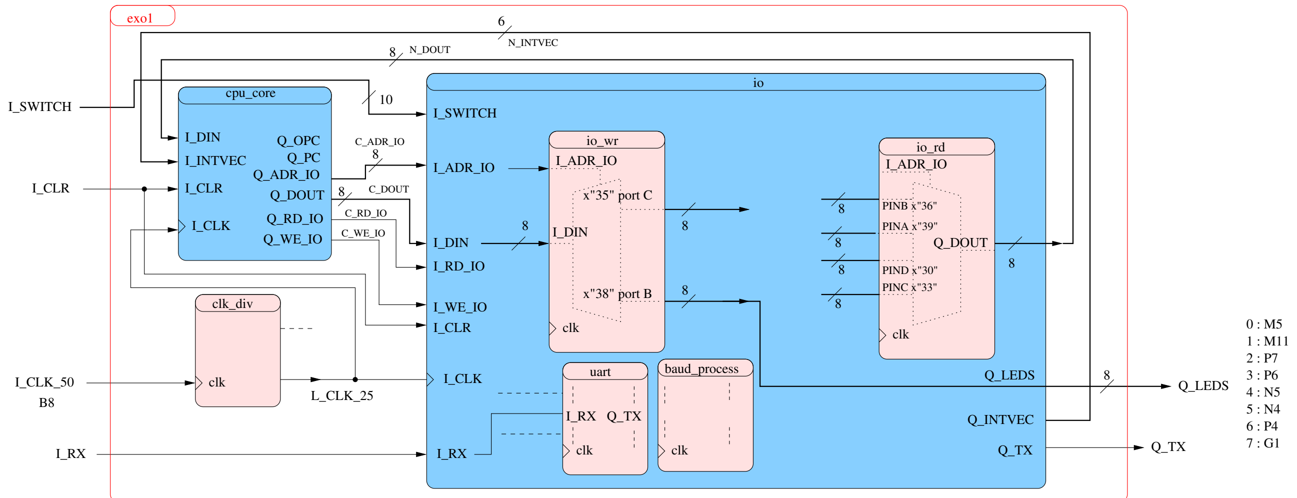
On vous propose ci-dessous, le schéma d'implantation d'une application de base pour une première mise en oeuvre du processeur AVR. Vous trouverez, sur le site internet de M. Moutou ou bien sur le serveur du département GEII, les fichiers VHDL nécessaires à l'implantation du processeur.

La liste complète est la suivante : ***cpu_core.vhd, opc_fetch.vhd, prog_mem.vhd, opc_deco.vhd, data_path.vhd, alu.vhd, register_file.vhd, reg_16.vhd, status_reg.vhd, data_mem.vhd, io.vhd, uart.vhd, baudgen.vhd, uart_rx.vhd, uart_tx.vhd, memory.bmm*** et ***tpmcensl1_basys.ucf***. Le fichier ***tpmcensl1_basys.ucf*** est fourni car il contient des directives spécifiques à l'implantation de la mémoire programme de l'AVR dans le FPGA. Le fichier ***memory.bmm*** est nécessaire pour l'envoi, dans la mémoire programme de l'AVR, du programme C compilé. La mémoire programme de l'AVR utilise certains blocs BRAM du FPGA parmi tous ceux qu'il contient. Il faut donc indiquer dans quels blocs exactement envoyer le programme. Le fichier ***memory.bmm*** contient ces informations.

Travail demandé :

- Sous ISE, créer un projet (par exemple exo1) pour le FPGA SPARTAN3E de référence XC3S250ECP132. Ajouter à ce projet tous les fichiers VHDL listés ci-dessus. Créer un répertoire (par exemple soft) dans le répertoire du projet VHDL (exo1).
- Ouvrir le fichier VHDL toplevel (exo1.vhd) . Le compléter pour implanter le processeur (cpu_core) et l'interface d'entrées/sorties (io) selon la figure ci-dessous. L'entité sera générée en considérant les entrées et sorties du FPGA (cadre rouge).
- Prendre le temps d'étudier le listing (entité et architecture) du component ***io*** pour comprendre le principe de l'extension des ports d'entrée et de sortie du processeur et comprendre comment on peut envoyer des données du processeur sur des sorties physiques du FPGA. De manière duale, étudier le principe de l'entrée de données dans le FPGA pour en disposer au niveau du processeur.
- Compiler ce projet pour vérifier qu'il n'y a pas d'erreur de description matérielle dans le projet.
- Proposer un programme C générant un chenillard simple (aller et retour d'une led allumée sur 8 leds - leds allumées avec des '1' et éteintes avec des '0'). On pourra (par exemple) initialiser un registre à 0x01 (et l'envoyer sur un port de sortie connecté aux led de la carte FPGA) puis opérer des décalages successifs à gauche jusqu'à obtenir le bit de poids fort à '1' seul. Puis inverser le mouvement avec des décalages successifs à droite jusqu'à obtenir le bit de poids faible à '1' seul. Et reboucler en premanence. L'instruction ***while {}*** et les instructions de décalage >> et << vous serviront de base à votre programme. Vous aurez aussi besoin de temporisation pour ne pas balayer trop rapidement. Vous trouverez en fichier joint (chenille.c) un listing de programme C pour vous aider à démarrer. Vous pourrez taper le code sous gedit et appeler votre fichier, par exemple, ***pgm.c***. Dans un terminal, vous vous placerez dans le répertoire ***soft*** et taperez les 3 commandes de compilation suivantes :

  - avr-gcc -g -mmcu=atmega8 -Wall -Os -c pgm.c
  - avr-gcc -g -mmcu=atmega8 -o pgm.elf -W1,-Map,pgm.map pgm.o
  - /opt/Xilinx/14.5/ISE_DS/ISE/bin/lin/data2mem -bm ../memory.bmm -bd pgm.elf -bt ../exo1.bit -o b ../exo1_rp.bit
- Configurer le FPGA avec le fichier exo1_rp.bit. Il s'agit du fichier .bit initial modifié par le programme (à executer par le processeur) qui est à envoyer dans les blocs de mémoire (BRAM). On rappelle que le fichier .bit porte le nom de l'entité. Lancer l'outil de configuration ***Impact*** situé dans ***Configure Target Device*** de l'arborescence des process à lancer pour configurer le FPGA. Vérifier le bon fonctionnement de l'application et faire valider votre travail par l'enseignant.



Exercice 2 : synchronisation VGA et affichage d'un point

On se propose maintenant d'ajouter à l'exercice 1, la synchronisation d'un écran VGA avec affichage d'un petit carré à des coordonnées fixes. Ajouter le component VGA_synchro à votre application puis le dessin d'un carré de 3 pixels de côté sous la forme d'une description VHDL process . Pour cela, suivez le schéma synoptique de la figure ci-dessous.

Travail demandé :

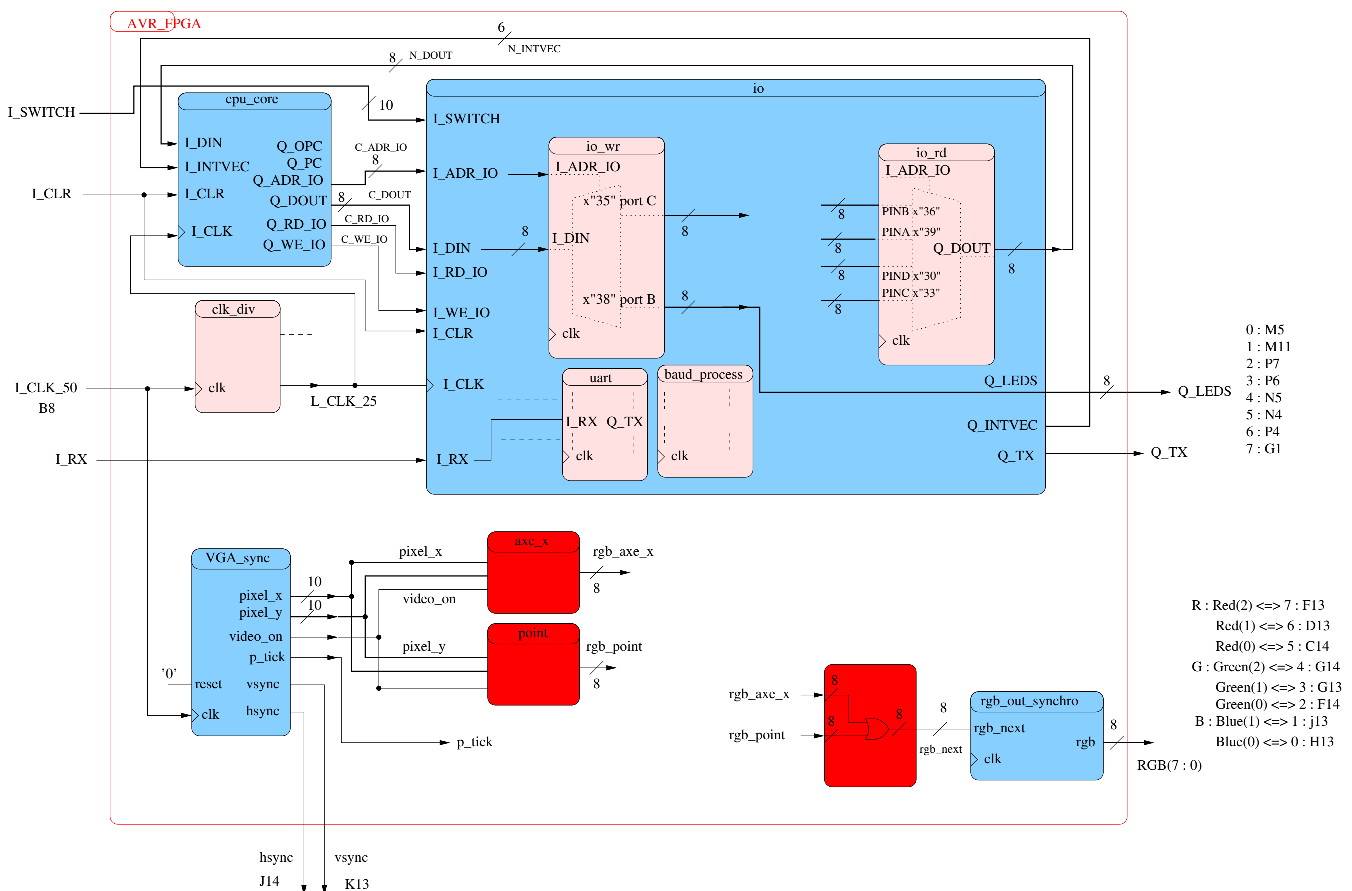
- Créer un nouveau projet exo2 et y placer tous les fichiers .vhd, .ucf, .bmm du projet exo1 ainsi que le répertoire soft et son contenu. Ajouter le fichier VGA_synchro.vhd fourni dans les ressources.
- Ajouter les component VGA_sync et rgb_out_synchro dans le fichier toplevel avec toutes les modifications que cela entraine (entité, déclaration de component, port map, fichier .ucf...). VGA_sync fournit les signaux de synchronisation directement envoyés à l'écran. Il fournit par ailleurs en temps réel les coordonnées du pixel courant à l'écran au cours du balayage de l'écran. Pixel qui est à gérer en temps réel pour décider de l'allumer ou non et de quelle couleur. De fait, on décide d'éclairer le pixel en noir (éteint) ou bien d'une autre couleur (allumé).
- Ajouter un process combinatoire (axe_x) de description VHDL du tracé d'une droite horizontale à mi-hauteur de l'écran (coordonnée y=240 dans l'écran 640 x 480). On testera si video_on est à '1' ce qui autorise de dessiner des pixels car cela signifie qu'à cet instant, on balaie des pixels de l'écran (sinon on est dans les top de synchronisation et il faut absolument des données RGB à 0). Ainsi donc quand on est avec video_on à '1', on comparera en temps réel l'ordonnée du pixel courant (pixel_y) avec l'ordonnée de la ligne à dessiner (y=240) que l'on souhaite afficher. Si oui , on affichera le pixel allumé en blanc, sinon on le laisse éteint en noir. On notera que $y = 240$ est une valeur fixe que l'on décrira "en dur" dans la decription VHDL. Dans la mesure où il s'agit d'un test, ***if*** ou ***elsif*** en VHDL, on pourra exprimer cette valeur en décimal. L'algorithme est donc le suivant :

```
si video_on = 0 alors
    rgb = 000
sinon si pixel_y = 240 alors
    rgb = 111
sinon
    rgb = 000
```

- Ajouter un process combinatoire (point) de description VHDL du dessin à l'écran d'un carré blanc sur fond noir de 3x3 pixels. Ainsi, on comparera en temps réel les coordonnées du pixel courant avec les coordonnées du point (x,y) que l'on souhaite afficher à + ou -1 pixels . Ici, on pourra à nouveau exprimer les valeurs de x et y en décimal avec des valeurs fixes. On calculera les limites, pour avoir un carré 3x3, en prenant bien en compte le fait que les comparaisons sont strictes (>et <). L'algorithme est donc le suivant :

```
si video_on = 0 alors
    rgb = 000
sinon si pixel_x > x-2 et pixel_x < x+2 et pixel_y >y-2 et pixel_y < y+2 alors
    rgb = 111
sinon
    rgb = 000
```

- Ajouter une fonction logique combinatoire de synthèse du signal RGB. Il s'agit d'un simple ***ou-logique*** comme la figure le propose. Et connecter le signal RGB résultant au component de synchronisation rgb_out_synchro.
- Compiler la description matérielle ainsi obtenue pour vérifier qu'elle est correcte. On peut à ce moment configurer le FPGA pour vérifier de suite que l'écran VGA est bien géré comme prévu (fichier .bit et pas fichier_rp.bit !) .
- Compiler le programme C sur le modèle des 3 lignes de commandes de l'exercice 1. Configurer le FPGA pour vérifier le fonctionnement simultané du chenillard et de l'affichage sur l'écran VGA (fichier_rp.bit).



Exercice 3 : algorithme CORDIC et affichage VGA

On ajoute maintenant un coeur de calcul de l'algorithme CORDIC. L'algorithme CORDIC permet de calculer de manière itérative le sinus et le cosinus d'un angle. Le coeur CORDIC qui vous est fourni traite les angles (en radian) situés entre $-\pi$ et $+\pi$. Il est décrit en VHDL et est piloté par une machine d'état qui reçoit un ordre de démarrage du calcul par le processeur et qui renvoit un signal de fin de calcul au processeur. Cette machine d'état gère également le déroulement du calcul décrit en VHDL (signaux de reset, enable, valeur de compteur,

...) de la partie matérielle décrite en VHDL qui réalise le calcul CORDIC . On se propose d'exploiter ce calcul CORDIC sans l'étudier dans le détail. On affichera alors la valeur du sinus comme un point sur l'écran VGA . Le processeur proposant divers angles depuis $-\pi$ jusqu'à $+\pi$ au cours du temps, on verra osciller verticalement le point sur l'écran VGA.

Travail demandé :

- Créer un nouveau projet (exo3) à partir du projet de l'exercice 2 et ajouter le fichier cordic_fsm_style.vhd au projet. A chaque étape suivante, faites vérifier le code VHDL généré par l'enseignant.
- Implanter le component ***cordic_fsm*** selon le schéma de la seconde figure ci-dessous en modifiant donc le component ***io***. On ajoutera les signaux et les entrées/sorties nécessaires.
- Modifier l'architecture du component ***cordic_fsm*** en y ajoutant la description VHDL de la machine d'état de la figure ci-dessous.
- Ajouter la description VHDL des process dessinés en rouge :
  - calcul de la valeur absolue du sinus (***val_abs***) : on met en oeuvre le complément à 2.

    ```
    si le msb du sinus = 1 alors
       val_abs = not(sinus) +1
    sinon
       val_abs = sinus
    finsi
    ```

  - calcul de l'ordonnée du point (***point***) : quand le sinus vaut 0, le point est situé en y0. Lorsque le sinus est positif, le point est plus haut dans l'écran VGA donc avec une ordonnée plus petite (origine du plan en haut et à gauche de l'écran). Lorsque le sinus est négatif, le point est plus bas dans l'écran donc avec une ordonnée plus grande. D'où l'algorithme suivant :

    ```
    si le msb du sinus = 1 alors (sinus negatif)
       ymin = y0 + val_abs -2
       ymax = y0 + val_abs +2
    sinon (sinus positif)
       ymin = y0 - val_abs -2
       ymin = y0 - val_abs +2
    ```

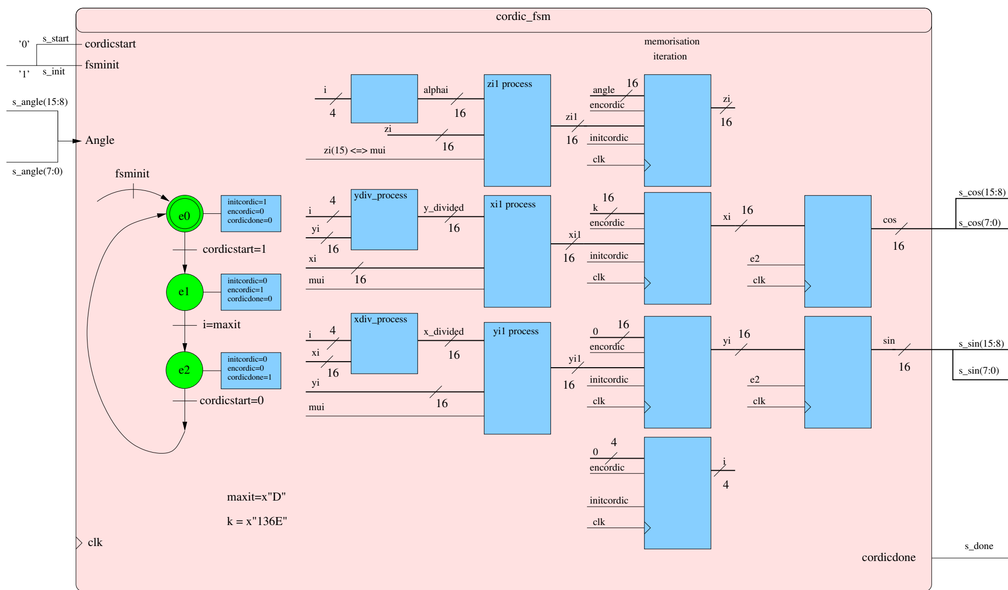
    Le sinus fourni par le calcul CORDIC et sa valeur absolue utilisée dans l'algorithme sont des nombres sur 16 bits. Les 3 bits MSB correspondent à la partie entière. Les 13 bits LSB correspondent à la partie "décimale" (en réalité c'est bien du binaire donc ce sont des 1/2, 1/4, 1/8, etc et pas des 1/10, 1/100, 1/1000, etc). Cependant, on doit d'une part disposer de nombres ($ymin$ et $ymax$) sur 10 bits pour correspondre à la taille du nombre binaire représentant les coordonnées des pixels en x comme en y . D'autre part le 0 du sinus devant corresponndre à l'ordonnée 256 et l'ordonnée maximale de l'écran étant 479, on doit disposer d'un sinus s'étendant sur 128 pixels au maximum de part et d'autre de la valeur 256 pour sa valeur à $+/-1$). On prendra donc les bits 13 à 6 de val_abs, qui représente 128 au plus. On placera 2 bits à 0 en poids fort MSB concaténés à val_abs(13 : 6) pour obtenir ainsi 10 bits. Enfin, $y0$ sera aussi un nombre sur 10 bits (tout comme$ymin$ et $ymax$) que l'on forcera à 256.
  - calcul du point en rgb (***rgb_point***) et synthèse additive du signal rgb : sur le même principe que le dessin du point de l'exercice précédent, il s'agit d'adapter le calcul du signal rgb_sin à partir des limites obtenues depuis le process point. Pour la position du point selon l'axe x on prendra une position fixe sur 3 pixels comme dans l'exercice précédent. On a donc une valeur fixe portée par $x0$ sur 10bits, et que l'on forcera à 32 en binaire par exemple. On réalisera ensuite la synthèse additive des 2 signaux rgb pour fournir LE signal rgb à transmettre à l'écran comme dans l'exercice précédent.
- Ecrire le programme C qui permet de balayer les angles de $-\pi$ à $+\pi$ en les envoyant au coeur de calcul CORDIC. On rappelle que les angles sont sur 16 bits au format Q3.13. On calculera au préalable la représentation en Q3.13 de $+\pi$. On prendra son complément à deux pour obtenir $-\pi$ au format Q3.13. L'algorithme est le suivant :

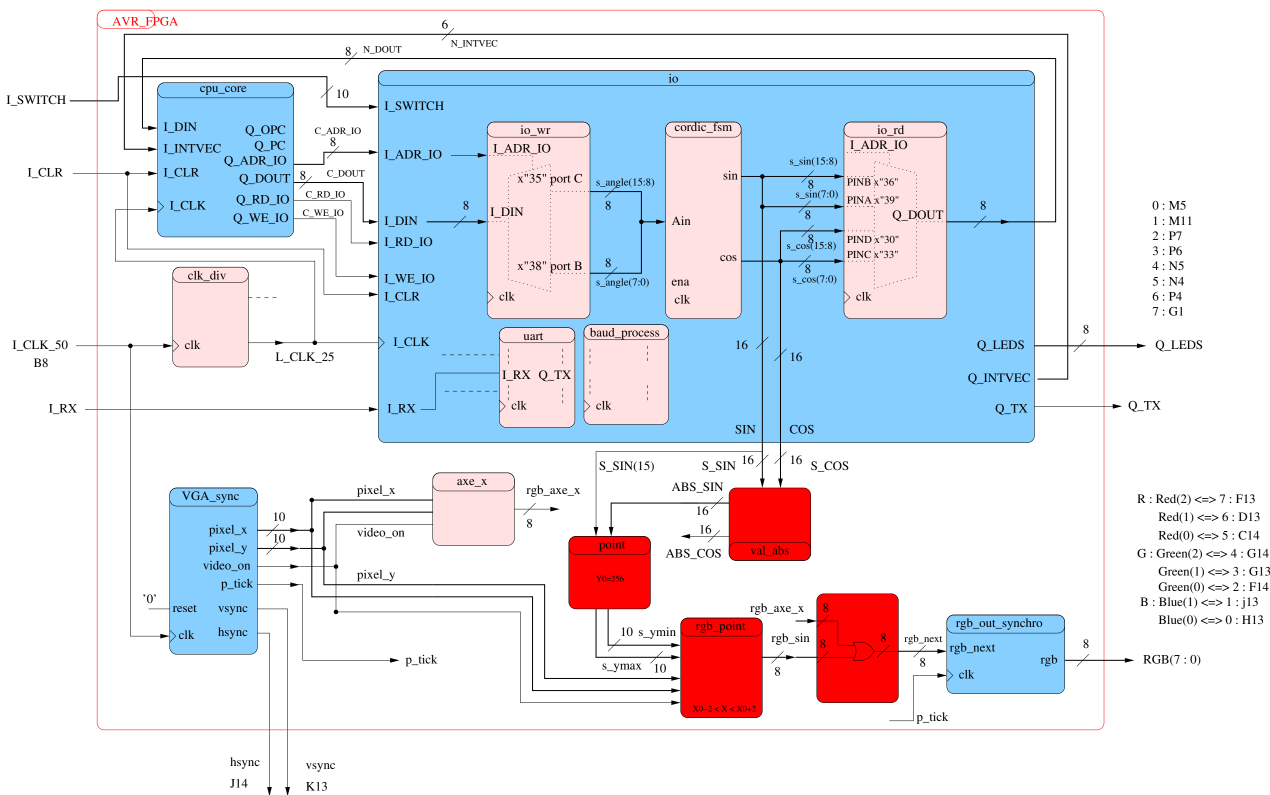
  ```
  calcul cordic terminé = faux
  initialisation de la machine d'état
  délai (éventuellement)
  arrêt de l'initialisation
  tant que 1 (faire en permanence)

     pour angle = −π à +π (incrementation de l'angle en q3.13 à bien choisir)
  ```

```
          envoi angle(8 LSB)
          envoi angle(8 MSB)
          lancement du calcul du coeur cordic
          tant que ( calcul cordic pas terminé )
            lire (calcul cordic terminé)
          fin tant que
          arrêt du calcul du coeur cordic
          delai ( à bien régler)
      fin pour
  fin tant que
```

Compiler le programme C pour vérifier au moins sa syntaxe.

– Compiler l'ensemble de l'application et constater le fonctionnement de l'application. On doit observer l'oscillation verticale du point (carré de 3 pixels de côté) entre les ordonnées 128 et 384. Faites valider par l'enseignant.





Exercice 4 : Tracé de sinusoide

On se propose désormais de modifier encore un peu l'exercice 3 pour obtenir un début de tracé de sinusoïde sur écran VGA. En réalité, ce sera un point qui glissera sur le tracé de la sinusoïde mais sans obtenir un affichage rémanent (net et complet) de celle-ci. Cela est dû à la vitesse de calcul du sinus combinée au nombre de points dans une période et à la fréquence de rafraîchissement de l'écran.

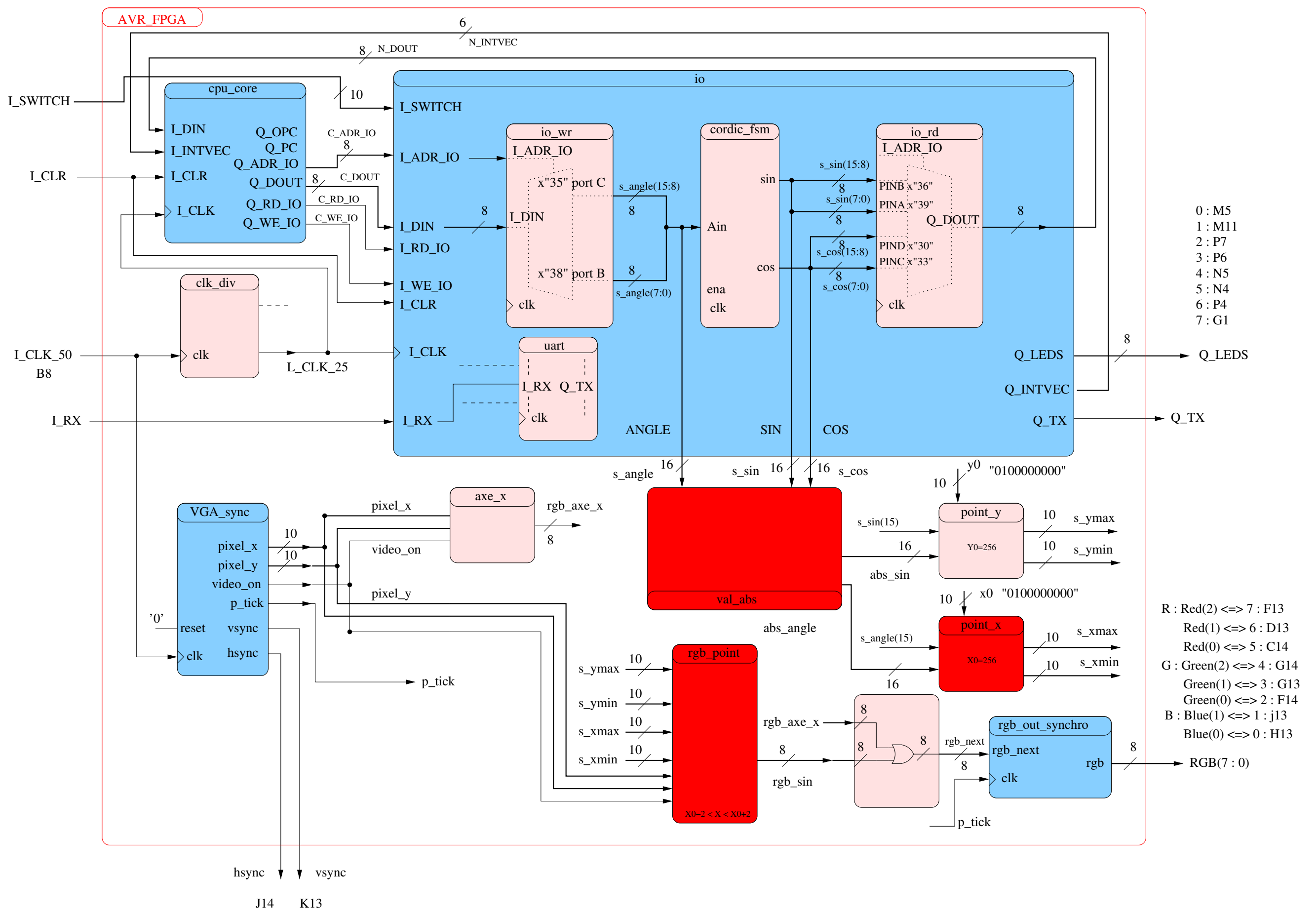
Il s'agit donc de faire évoluer la position en x (qui était précédemment figée) du point. On se servira naturellement de la valeur de l'angle (qui évolue de $-\pi$ à $+\pi$) présentée par le processeur au coeur de calcul CORDIC. Il s'agit d'un angle exprimé dans le format Q3.13. On rappelle encore que le MSB donne le signe de l'angle. Les 2 bits MSB suivants donnent la partie entière de l'angle (donc jusqu'à 3 pour π) . Les 13 bits suivants donnent la partie "décimale" de l'angle avec la même remarque que dans l'exercice précédent, à savoir , ce sont des 1/2, 1/4, 1/8, ... en d'autres termes des $1/2^{-i}$.

Ainsi, on pourra associer l'étendue de l'angle (2π) à une étendue sur l'écran VGA selon l'axe x. Par exemple, on pourra associer 512 pixels pour une période. Comme l'angle est positif ou négatif ($-\pi$ à $+\pi$), sa valeur absolue correspond à la demi période soit 256 pixels. De ce fait, on prendra abs_angle(14:7) (**penser à concaténer des 0 en MSB pour avoir 10 bits de largeur**) pour calculer la position en x qui représentera environ 128 pixels. Sur le modèle du calcul de la position en y dans l'exercice 3, l'algorithme sera le suivant :

```
si le msb de l'angle = 1 alors (angle negatif)

    xmin = x0 - abs_angle -2
    xmax = x0 - abs_angle +2

sinon (angle positif)

    xmin = x0 + abs_angle -2
    xmin = x0 + abs_angle +2
```

Travail demandé :

- Créer un nouveau projet exo4 et y placer tous les fichiers .vhd, .ucf, .bmm du projet exo3 ainsi que le répertoire ***soft*** et son contenu.
- Modifier le fichir ***io.vhd*** pour permettre de fournir l'angle à la partie VHDL traitant de l'aspect VGA comme vous l'avez fait pour le sinus et le cosinus dans l'exercice précédent.
- Modifier le process de calcul de valeur absolue (***val_ abs***) pour y ajouter la valeur absolue de l'angle. L'algorithme est bien sûr le même que pour la valeur absolue du sinus.
- Ajouter le process de calcul de la position en x du point selon l'algorithme proposé ci-dessus (***point_ x***).
- Modifier le process (***rgb_ point***) de calcul du signal ***rgb_ sin*** en prenant en compte ***s_ xmin*** et ***s_ xmax*** pour les comparer à ***pixel_ x*** .
- Reprendre le programme C en modifiant éventuellement les délais et le pas d'incrémentation de l'angle pour obtenir un affichage acceptable. Le compiler et générer le fichier .bit
- Télécharger l'application et constater le tracé en pointillés plus ou moins réguliers du sinus. Faites valider par l'enseignant.


AVR_FPGA
I_SWITCH
I_CLR
I_CLK_50
B8
I_RX
cpu_core
I_DIN
I_INTVEC
I_CLR
I_CLK
Q_OPC
Q_PC
Q_ADR_IO
Q_DOUT
Q_RD_IO
Q_WE_IO
N_DOUT
N_INTVEC
C_ADR_IO
C_DOUT
C_RD_IO
C_WE_IO
clk_div
clk
L_CLK_25
io
I_SWITCH
I_ADR_IO
I_DIN
I_RD_IO
I_WE_IO
I_CLR
I_CLK
I_RX
io_wr
I_ADR_IO
I_DIN
x"35" port C
x"38" port B
clk
s_angle(15:8)
s_angle(7:0)
cordic_fsm
sin
cos
Ain
ena
clk
io_rd
I_ADR_IO
PINB x"36"
PINA x"39"
PIND x"30"
PINC x"33"
Q_DOUT
clk
s_sin(15:8)
s_sin(7:0)
s_cos(15:8)
s_cos(7:0)
uart
I_RX
Q_TX
clk
ANGLE
SIN
COS
Q_LEDS
Q_INTVEC
Q_TX
0 : M5
1 : M11
2 : P7
3 : P6
4 : N5
5 : N4
6 : P4
7 : G1
s_angle
s_sin
s_cos
val_abs
abs_angle
abs_sin
s_sin(15)
s_angle(15)
y0
"0100000000"
x0
"0100000000"
point_y
Y0=256
point_x
X0=256
s_ymax
s_ymin
s_xmax
s_xmin
VGA_sync
pixel_x
pixel_y
video_on
p_tick
'0'
reset
vsync
clk
hsync
axe_x
pixel_x
video_on
pixel_y
rgb_axe_x
p_tick
rgb_point
X0-2 < X < X0+2
rgb_axe_x
rgb_sin
rgb_next
rgb_out_synchro
rgb_next
rgb
clk
p_tick
RGB(7 : 0)
R : Red(2) <=> 7 : F13
Red(1) <=> 6 : D13
Red(0) <=> 5 : C14
G : Green(2) <=> 4 : G14
Green(1) <=> 3 : G13
Green(0) <=> 2 : F14
B : Blue(1) <=> 1 : j13
Blue(0) <=> 0 : H13
hsync
vsync
J14
K13