Samuel Beaussant

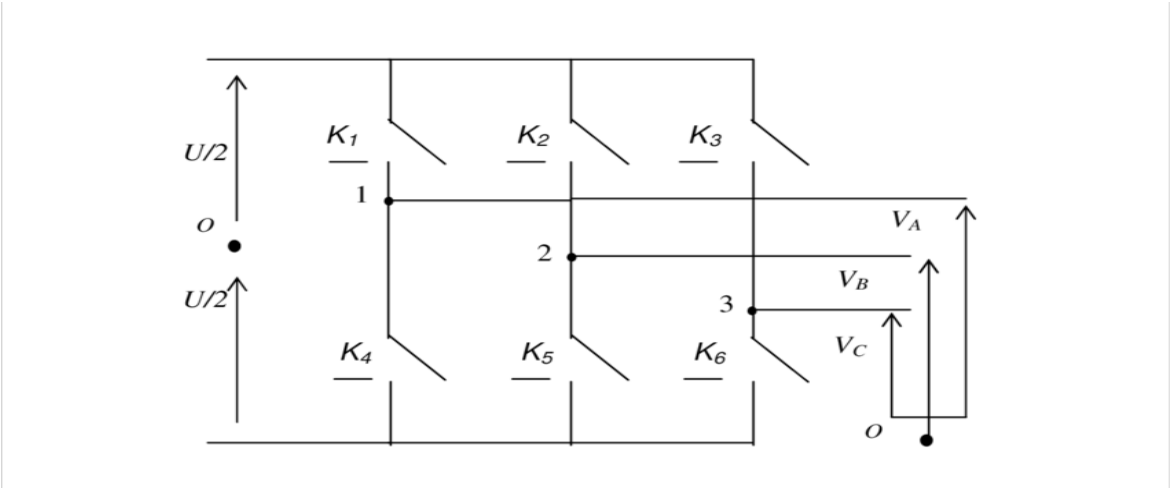
Rapport TP ESN10 :
Développement d'un onduleur sur SoC FPGA

Introduction

Le but de cette série de TP était de nous familiariser avec le système sur puce programmable DE10-standard contenant un FPGA Cyclone V de chez Altera. L'objectif était d'implémenter un onduleur triphasé capable de communiquer avec la partie HPS de la platine. Cette onduleur a pour but d'alimenter un moteur triphasé asynchrone via l'algorithme de la modulation vectorielle (ou SVM).

Spécification du système et comportemental:

L'onduleur doit permettre de commander un moteur asynchrone via une structure similaire à celle-ci :

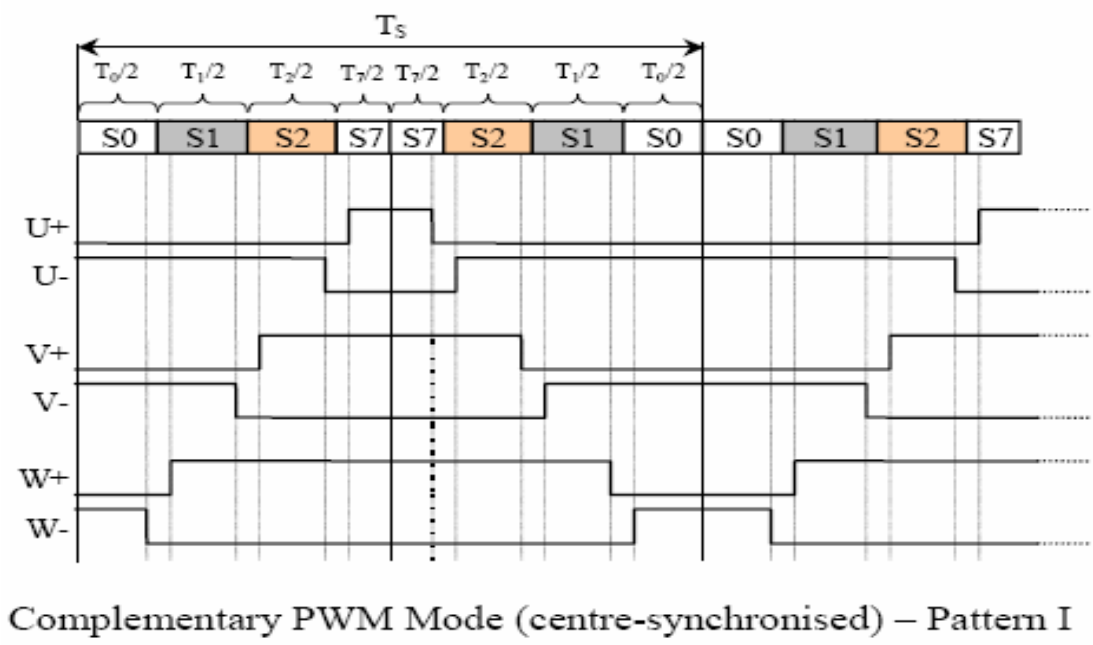


Pour éviter de court-circuiter le moteur lors de la commutation il est nécéssaire de considérer un délai pendant lequel les deux transistors commandant une phase seront bloqué. Ce délai est appelé le temps mort.

Le système à développer devaient avoir les spécifications suivante : fréquence des signaux générés comprise entre 5 kHz et 25 kHz (tolérance 1%), résolution minimale de 10 bits pour le réglage du rapport cyclique, gestion de bande morte réglable entre 0 et 4 µs, génération d'une interruption à chaque fin de période si nécessaire, contrôle du bloc à l'aide de 6 registres :

- contrôle et état
- réglage période
- réglage rapport cyclique phase U
- réglage rapport cyclique phase V
- réglage rapport cyclique phase W
- réglage bande morte

Le changement de période, de rapport cyclique et de largeur de bande morte doit être pris en compte uniquement lors du cycle suivant. Les formes d'onde devront être symétriques pour une meilleure CEM et entrée de mise en sécurité du bloc et finalement le système doit fonctionner à 100 kHz. Le système sera donc capable de générer 3 mli triphasées :



Complementary PWM Mode (centre-synchronised) – Pattern I

Les mli triphasées seront générées à partir de trois compteur-décompteurs de période de comptage égale à la demi-période et trois comparateurs. On aura donc un signal triangulaire de bande morte Tm :

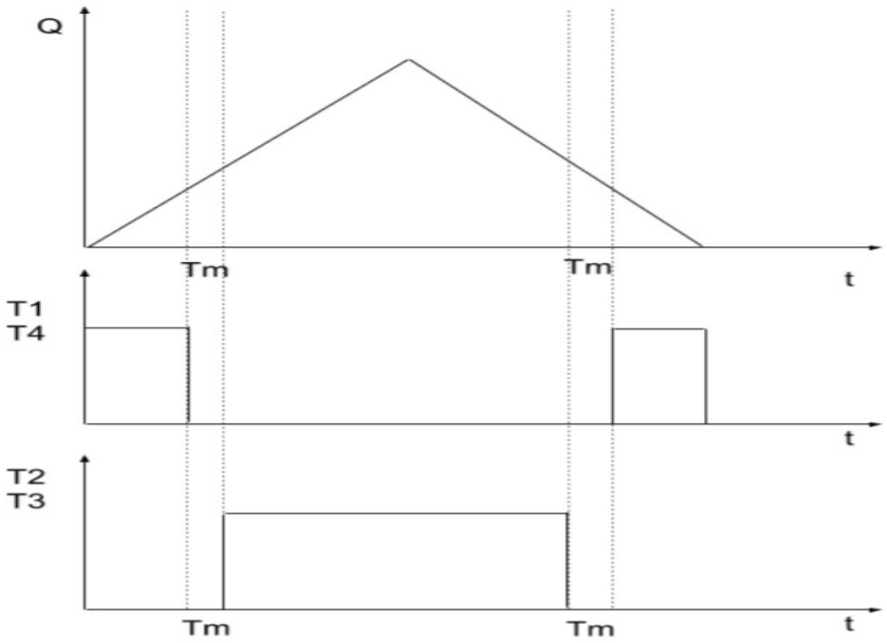


Figure 3 signal triangulaire et gestion des interrupteurs

Etant donné que l'on veut une fréquence de MLI comprise entre 5 kHz et 25 kHz avec une clock à 100 MHz, il est nécéssaire d'avoir un compteur-décompteur de période de comptage comprise entre 4000 (25kHz) et 20000 (5 kHz), on aura donc besoin d'un registre d'au moins 15 bits pour la valeur de la période . De même, le réglage de la bande morte à un maximum de 4us impose un registre de minimum 9 bit (soit 420 cycle de comptage).

Specification avalon :

Le bus permettant de faire la communication entre le FPGA et le HPS est un bus de taille fixe de 32 bits appelé le bus avalon. Ce bus est disponible pour différente utilisation mais dans le cas de notre application, il sera utilisé en memory-map particuliérement adapté à la communication avec des registres. Le bus avalon autorise les accès mémoire d'une taille multiple de 8 donc donc tout nos registres seront pris sur 16 bits. Prendre des tailles de registres de rapport cycliques sur 16 bits permet en plus de satisfaire la résolution minimum de 10 bits imposé.

Afin de pouvoir communiquer correctement avec le bus avalon il est nécessaire de développer une interface du côté de l'esclave disposant des signaux suivant :

- avs_adress : permet de sélectionner l'adress du registre dans lequel on veut lire.
- avs_writedata : bus de donnée en écriture.
- avs_readdata : bus de donnée en lecture.
- avs_clock : horloge de fonctionnement
- avs_reset : reset.
- avs_byteenable : signal permettant de choisir quel octet on doit écrire.
- avs_write : signal permettant d'indiquer une lecture.
- avs_read : signal permettant d'indiquer une écriture.

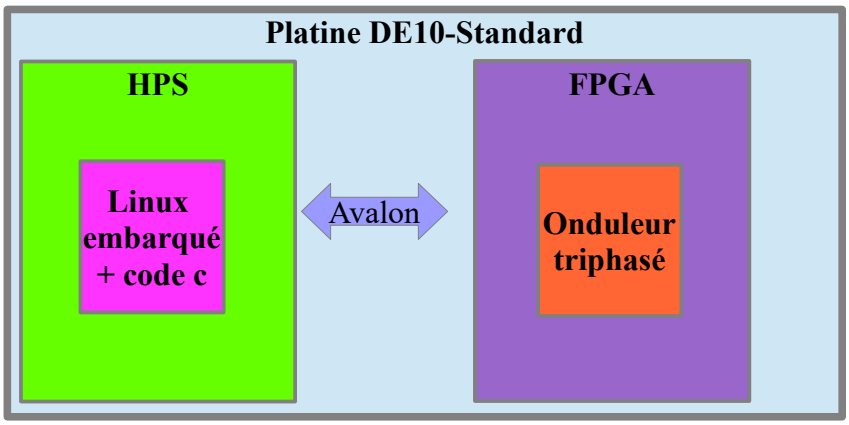
Le préfix « avs » permet à Qsys d'automatiquement détecter qu'il s'agit de signaux avalon. Afin de simplifier légérement le développement, la communication se fera avec des wait-state fixe.

Partitionnement matériel/logiciel

Le SoC dispose d'une partie matérielle dans le FPGA qui permet de développer des accélérateurs matérielles et d'une partie HPS constitué d'un processeur ARM cortex A9. Les deux communiquent via le bus avalon. L'architecture suivante a était développé pour répondre à la problématique :

- La partie matérielle disposera de l'onduleur et les registres écrit en VHDL et intégré à la plateforme via Qsys. Les différents blocs seront ensuite assemblé dans un schema bloc. Un bloc interfaceAvalon permet de faire office d'esclace avalon.

- La partie HPS servira de maître avalon et contiendra le code c permettant de faire tourner l'algorithme de la modulation vectorielle.

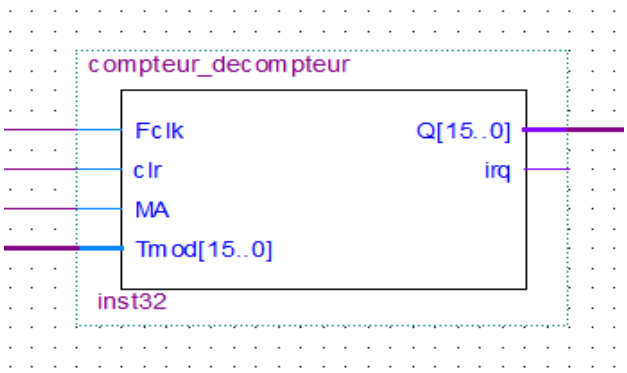


Etude des blocs matériels à développer

L'onduleur a était divisé en plusieurs fonctions élémentaires : un compteur-décompteur, un comparateur,un convertisseur duty cycle, un générateur de référence, un bloc verrou contenant les différents registres et l'interface avalon. Les fonctions des différents blocs seront expliciter en même temps que leur fonctionnement. Tous les blocs ont étaient testés individuellement via des testBench.

Compteur-décompteur :

Ce bloc permet d'obtenir le signal triangulaire ci-dessous. Il prend en entrée Tmod la demi-période qui correspond au temps de comptage pour arrivé au sommet du triangle. L'entré MA fait office de marche/arrêt pour le compteur-décompteur. La sortie Q contient la valeur de comptage et irq, le flag de fin de comptage.

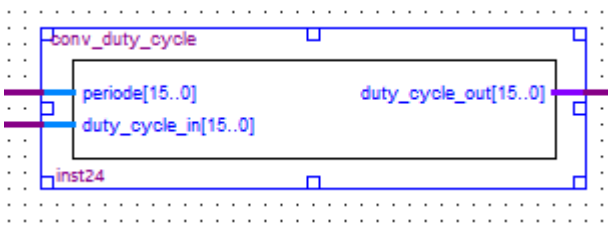



```
architecture Arch_cpt_dct of compteur_decompteur is
        signal Q_int : unsigned (Q'range);
        signal count_down : std_logic;
        begin
        -- Process de comptage/décomptage
        process (clr,Fclk) begin
            if(clr='1') then
                -- on redémare le compteur
                Q_int <= (others =>'0');
            elsif (Fclk' event and Fclk ='1' and count_down='0' and MA = '0') then
                -- on compte
                Q_int <= Q_int +1;
         elsif (Fclk' event and Fclk ='1' and count_down='1' and MA = '0') then
                -- on décompte
                Q_int <= Q_int-1;
            elsif (Fclk' event and Fclk ='1' and MA = '1') then
                -- on stop le comptage
                Q_int <= Q_int;
            end if;
        end process;
        Q <= Q_int;
        -- Process de génération de signal
        process(clr,Fclk) begin
            if(clr='1') then
                count_down <= '0';
            irq <= '1';
            elsif (Fclk'event and Fclk='1') then
                irq <='0';
                if(Q_int=Tmod) then
                    count_down <= '1';
                elsif(Q_int="0000000000000001" and count_down='1' ) then
                    -- on arrive en fin de cycle
                    -- on met le flag de fc à 1
                    count_down <= '0';
                    irq <='1';
                end if;
            end if;
        end process;
end Arch_cpt_dct;
```

ConvDutyCycle :

Ce bloc permet de convertir un rapport cyclique en nombre de cycle d'horloge proportionnellement à la période.

```
architecture Arch_conv_duty_cycle of conv_duty_cycle is

        signal tmp : unsigned (31 downto 0);

        begin
            tmp <= (duty_cycle_in * periode)/65356;
            duty_cycle_out  <= resize(tmp,16);

end Arch_conv_duty_cycle;
```

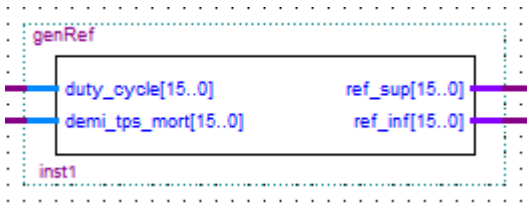


Il prend en entrée la periode et le rapport cyclique souhaité entre 0 (0%) et 65536 (100%) et sort le nombre de cyle correspondant entre 4000 et 20000.

GenRef :

Ce bloc permet de générer les références que les comparateurs vont utiliser pour créer les MLI complémentaires. Il se contente d'additionner ou soustraire les valeurs des demi-temps morts au rapport cyclique voulu. En sortie nous avons les références sup et inf pour les deux mli complémentaires.

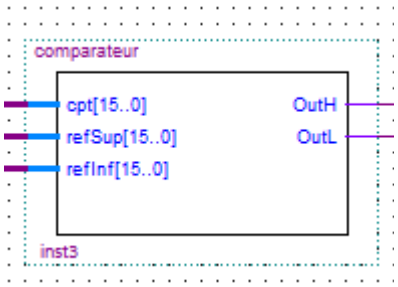
```
architecture Arch_genRef of genRef is


        signal ref1 : unsigned (15 downto 0);
         signal ref2 : unsigned (15 downto 0);

         begin
             ref1 <= duty_cycle + demi_tps_mort;
            ref2 <= duty_cycle - demi_tps_mort;
             ref_sup <= ref1;
             ref_inf <= ref2;

end Arch_genRef;
```



Comparateur :

Comparateur classique qui met sa sortie à '1' quand la référence est supérieur à son entré, '0' sinon.

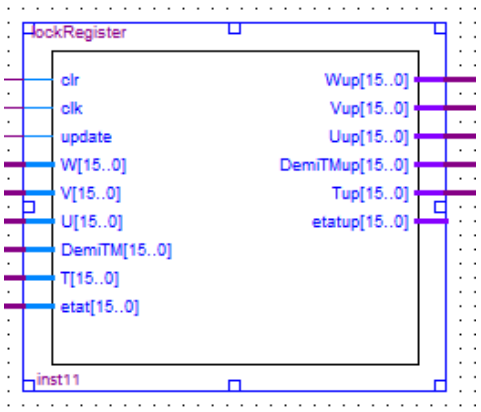
```
architecture Arch_comparateur of comparateur is

    begin
       process(cpt)
           begin
           if(cpt < refInf) then
               outH <= '1';
               outL <= '0';
           elsif(cpt > refSup) then
               outL <= '1';
               outH <= '0';
           elsif(cpt > refinf and cpt< refSup) then
               outL <= '0';
               outH <= '0';
           end if;
       end process;

end Arch_comparateur;
```



LockRegister :

C'est un verrou synchrone contenant tous les registres du systéme. Les signaux avec le suffix « up » sont les signaux de sorties mis à jour lors d'un front montant de « update », générer par le compteur-décompteur à chaque fin de comptage.

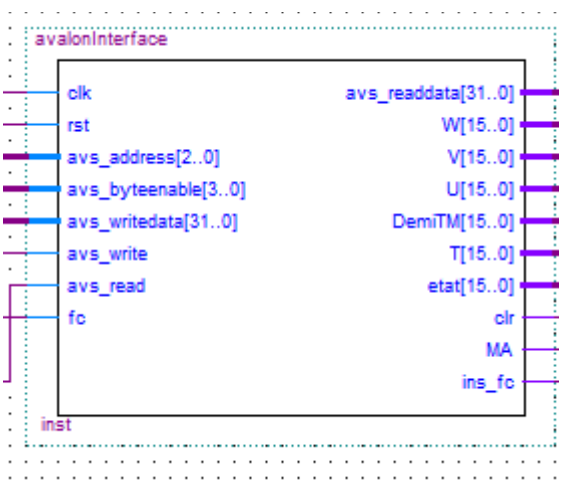
```
architecture archlockRegister of lockRegister is


    begin
        process (update, clr)
            begin
                if clr = '1' then                               -- Reset
                    Wup <= "0000000000000000";
                    Vup <= "0000000000000000";
                    Uup <= "0000000000000000";
                    Tup <= "0000000000000000";
                    DemiTMup <= "0000000000000000";
                    etatup <= "0000000000000000";
                elsif (clk'event and clk= '1') then     -- mis à jour des sorties
                    if(update='1') then
                        Wup <= W;
                        Vup <= V;
                        Uup <= U;
                        Tup <= T;
                        DemiTMup <= DemiTM;
                        etatup <= etat;
                    end if;
                end if;
        end process;

end archlockRegister;
```



AvalonInterface :

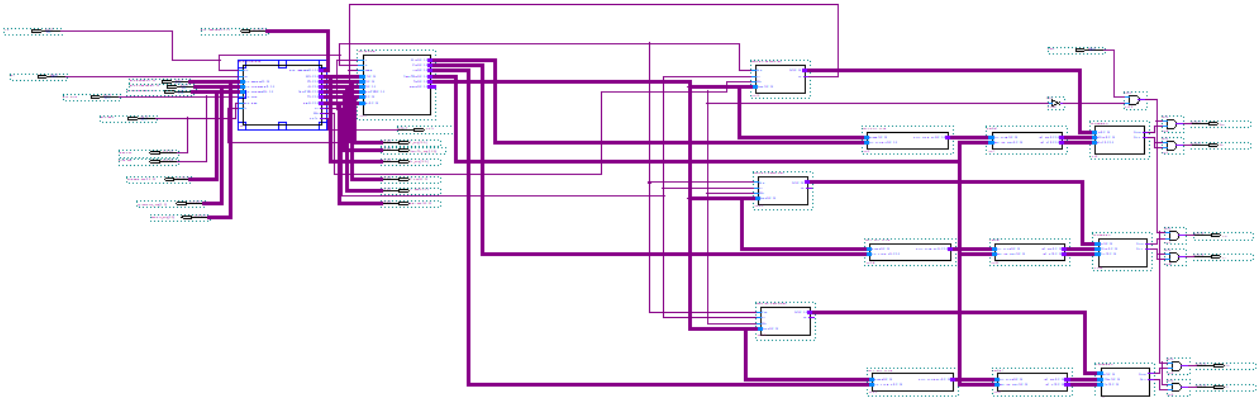
C'est ce bloc qui assure la communication avec le bus avalon. Il joue donc le rôle d'esclave avalon. Comme expliqué précédemment il dispose en entrée des signaux permettant de communiquer via le bus avalon :

```
begin
    process (clk
        variable
        begin
            if(c
                signal Treg : unsigned(15 downto 0) :="0000000000000000";          -- @001
                signal DemiTMreg : unsigned(15 downto 0) := "0000000000000000";    -- @010
                signal Wreg : unsigned(15 downto 0) := "0000000000000000";         -- @011
                signal Vreg : unsigned(15 downto 0) := "0000000000000000";         -- @100
                signal Ureg : unsigned(15 downto 0) := "0000000000000000";         -- @101
                signal etatreg : unsigned(15 downto 0) :="0000000000000111";       -- @110    interrupt_enable|Arret|clear actif

                if(rst = '1') then
                -- on repart de l'état initial
                    Treg <= "0000000000000000";
                    DemiTMreg <= "0000000000000000";
                    Wreg <= "0000000000000000";
                    Vreg <= "0000000000000000";
                    Ureg <= "0000000000000000";
                    when "001110100" =>
                        avs_readdata <= "0000000000000000" & std_logic_vector(Vreg);
                    when "001110101" =>
                        avs_readdata <= "0000000000000000" & std_logic_vector(Ureg);
                    when "001100110" =>
                        avs_readdata <= "0000000000000000" & std_logic_vector(etatreg);
                    when others =>
                        -- on ne fait rien
                        --avs_readdata <= "00000000000000000000000000000000";

                end case;
            end if;
    end process;

    -- MAJ des registres et bits
    W <= Wreg;
    V <= Vreg;
    U <= Ureg;
    DemiTM <= DemiTMreg;
    T <= Treg;
    ins_fc <= fc and etatreg(2);        -- si ien est 0 (interrupt disable) alors ins_fc est à 0
    etat <= etatreg;
    clr <= etatreg(0) or rst; -- on clr cpt si on reset depuis avalon ou depuis registre d'état
    MA <= etatreg(1);

end archavalonInterface;
```

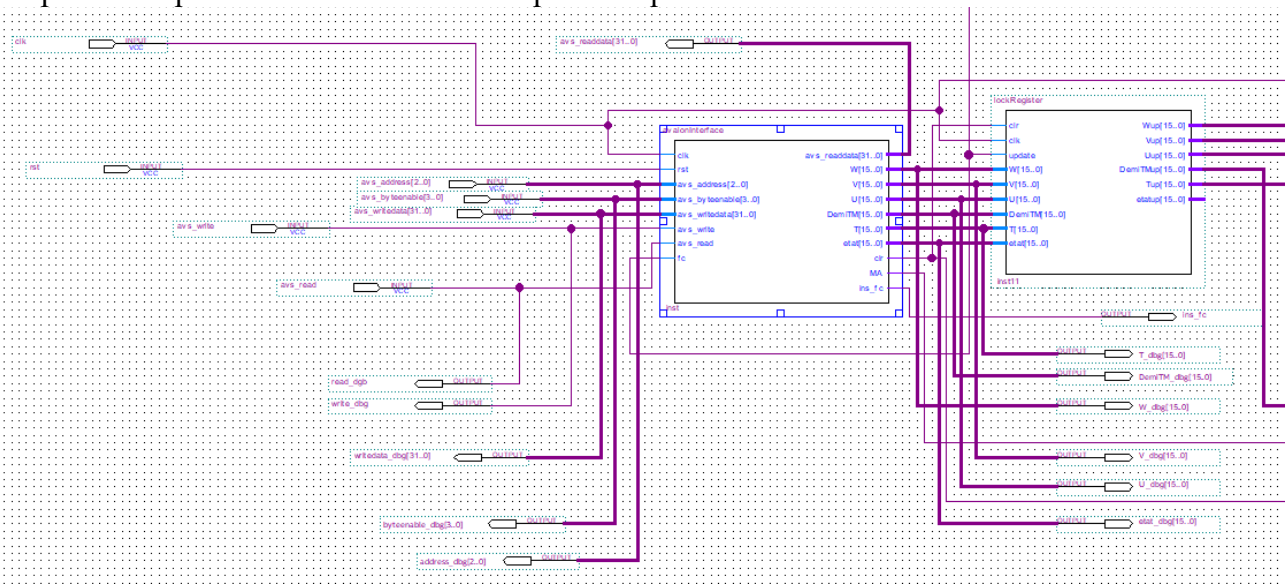


Etant donné que l'on dispose de 6 registres de 3 bits d'adresses. Le bus de donnée est sur 32 bits donc on a besoin de 4 bits pour le byteenable. Les signaux des registres suffixés « reg » sont représentés par des std_logic_vector de 16 bits. La variable frame représente une trame avalon. En fonction de sa valeur on effectue différente action. Ce n'est pas un signal car les signaux sont mis à jour avec un cycle de retard ce qui pose probléme. Les registres de sorties sont mis à jour à la fin du process. La sortie ins_fc servira d'interrupt sender pour indiquer à la partie HPS la fin de comptage et aussi mettre à jour les sorties du verrou.

Système complet :

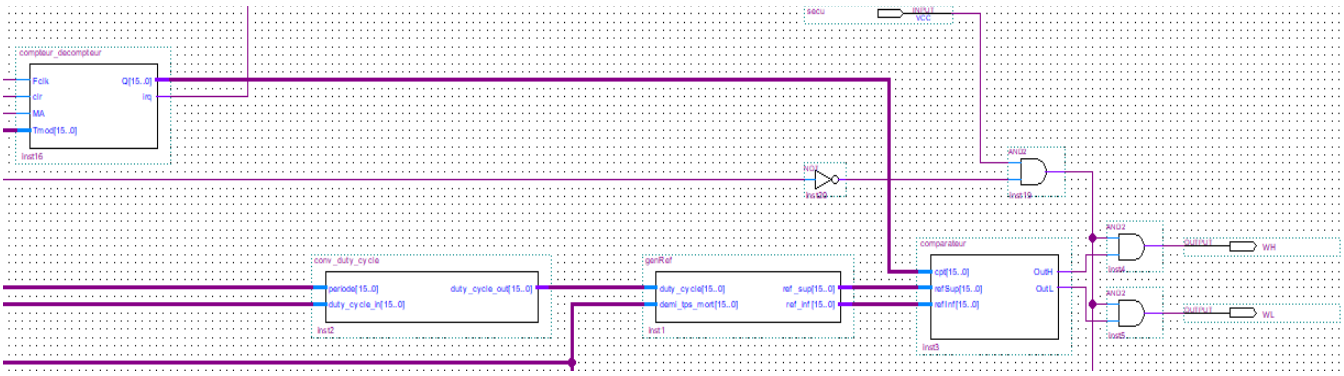


La partie reception de donnée est assurée par cette partie :

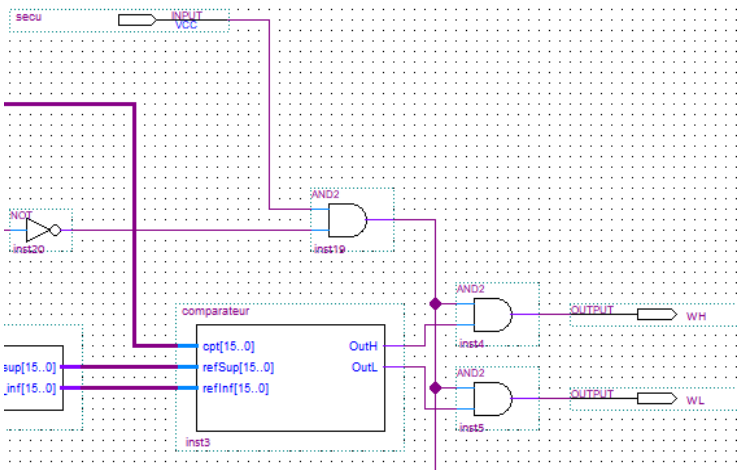


Des signaux de debug sont reliés au entrée et ajoutés en sortie pour pouvoir les visualiser sur signal tap et vérifier le bon fonctionnement de la communication avalon.

La partie génération de mli par ces blocs :



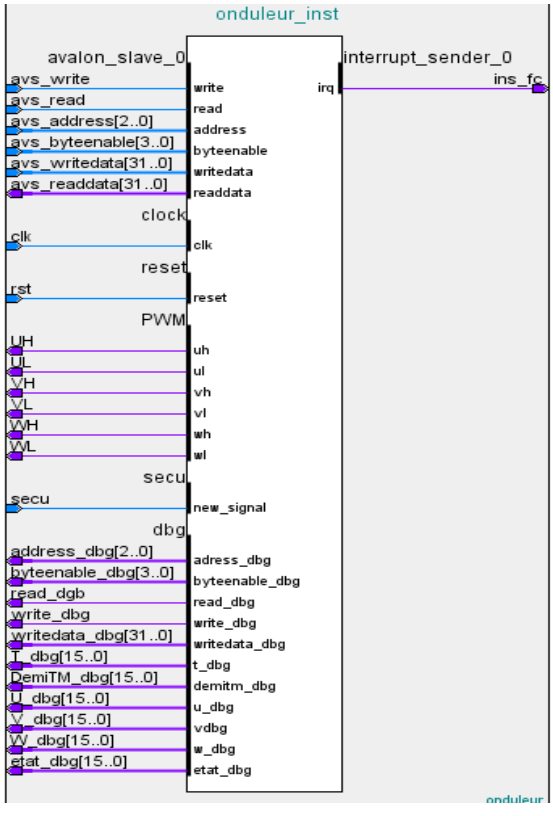
et finalement la partie mise en sécurité est assuré via cette partie :

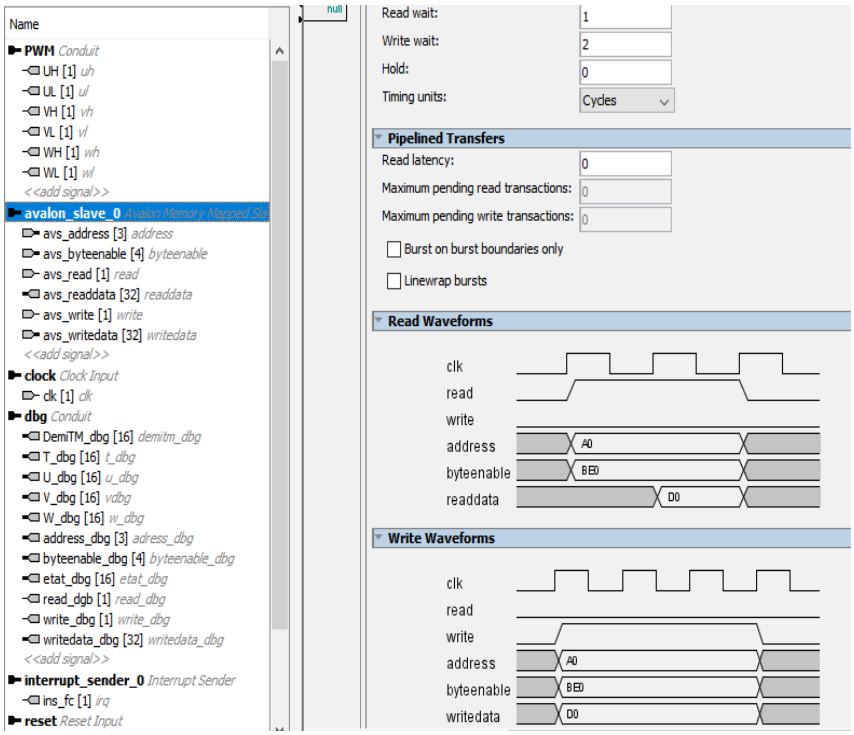


L'entrée sécu sera ensuite relié à un bouton sur la platine.

Conception du systéme et intégration avec Qsys :

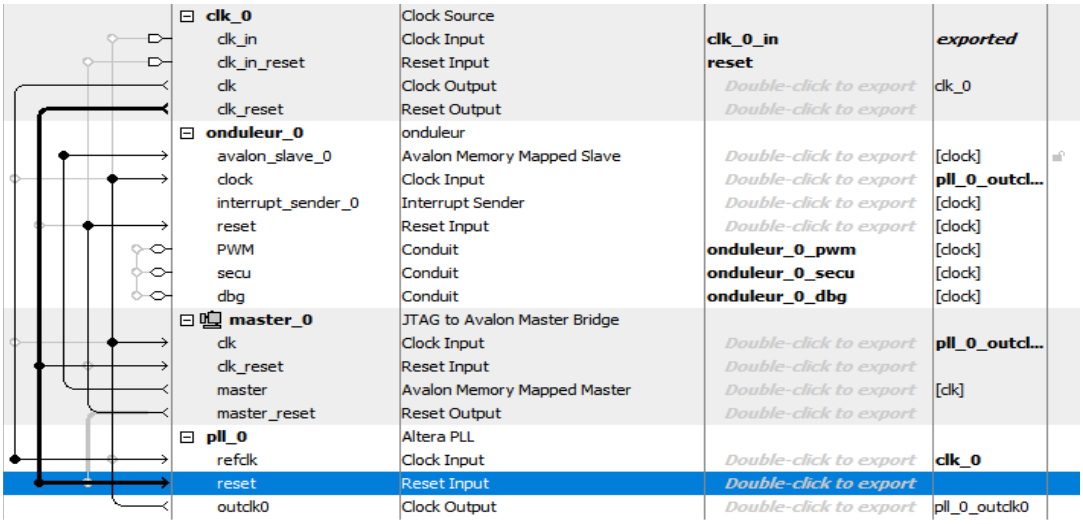
Afin d'intégrer notre IP onduleur, il est nécessaire de passer par l'outil Qsys. C'est un outil permettant d'intégrer un IP dans un SoC FPGA. La première étape est de créer l'IP onduleur pour pouvoir ensuite l'utiliser dans Qsys :



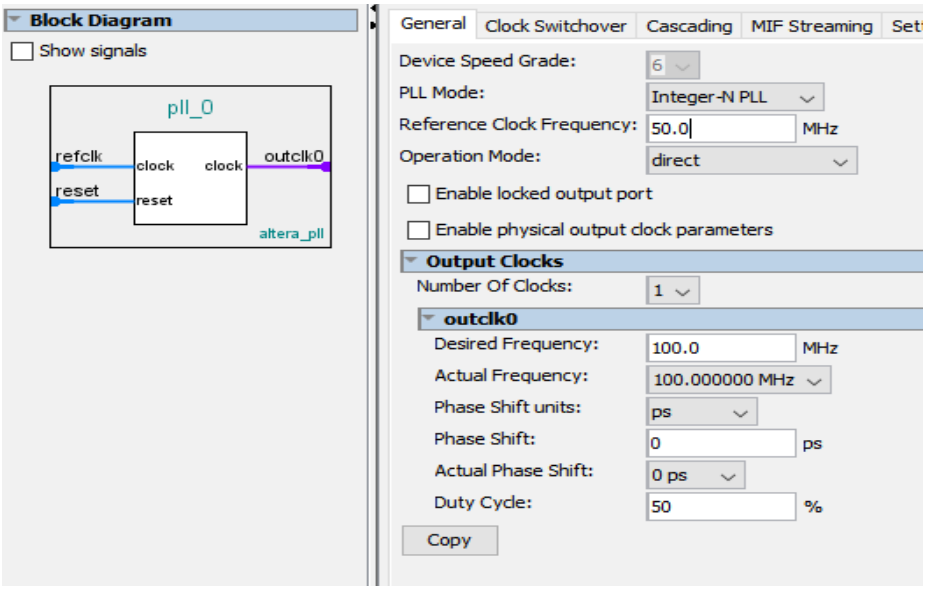


Les signaux qui ne sont pas de type avalon slave sont déclaré en conduit pour pouvoir ensuite les exporter et les assigner à des pin sur la plate-forme physique. L'interface dbg sera utile pour vérifier le bon fonctionnement de la communication. Il faut aussi dans cette partie penser à bien configurer les valeurs pour les read wait et write wait. Pour cette application j'ai choisi 1 pour le read wait et 2 pour le write wait afin de coller à l'exemple de la documentation. Le signal ins_fc est déclarer en interrupt_sender irq sur front montant.

Une fois l'IP créer, il faut l'intégrer dans le système :



On ajoute une pll en sortie de la clock à 50MHz afin d'obtenir les 100 MHz recquis :



Les 100 MHz sur la sortie outclock_0 sont utilisés comme clock pour le reste du système. Afin de tester la communication avalon avant de relier l'IP à la partie HPS, on utilise un IP jtag to avalon master bridge pour faire office de maitre avalon. Il sera ensuite possible d'envoyer des commandes via system_console sur la liaison jtag. Toutes les autres sorties sont exportées. Dans ce cas la, le signal interrupt sender est inutile. Il suffit ensuite de cliquer sur generate hdl pour obtenir le vhdl de tout le système. On ajoute ensuite le fichier .qip ainsi généré au projet avant de le mettre en top level entity et lancer l'analyse synthèse pour que quartus détecte correctement les I/O. Il est maintenant possible d'affecter des pins aux entrées sorties dans pins planner :

onduleur_..._read_dbg	Output	PIN_AB12	3A	B3A_N0	PIN_AB12	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_...write_dbg	Output	PIN_AA12	3A	B3A_N0	PIN_AA12	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_0_pwm_uh	Output	PIN_AJ2	3A	B3A_N0	PIN_AJ2	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_0_pwm_ul	Output	PIN_AJ1	3A	B3A_N0	PIN_AJ1	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_...der_0_irq	Unknown	PIN_AH2	3A	B3A_N0		3.3-V LVTTL		16mA (default)	
clk_0_in_clk	Input	PIN_AF14	3B	B3B_N0	PIN_AF14	3.3-V LVTTL		16mA (default)	
onduleur_0_pwm_vh	Output	PIN_AK3	3B	B3B_N0	PIN_AK3	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_0_pwm_vl	Output	PIN_Y16	3B	B3B_N0	PIN_Y16	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_0_pwm_wh	Output	PIN_AK2	3B	B3B_N0	PIN_AK2	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_0_pwm_wl	Output	PIN_W15	3B	B3B_N0	PIN_W15	3.3-V LVTTL		16mA (default)	1 (default)
onduleur_...ew_signal	Input	PIN_AK4	3B	B3B_N0	PIN_AK4	3.3-V LVTTL		16mA (default)	
reset_reset_n	Input	PIN_AJ4	3B	B3B_N0	PIN_AJ4	2.5 V		12mA (default)	

Test de la communication avalon :

Afin de valider la communication avalon, il est possible d'utiliser 2 outils : signal tap, un analyser logique embarqué et system_console. Signal tap est un outil qui est intégré dans la partie matérielle du FPGA lors de la compilation. Il utilise donc des blocs logiques et de la RAM pour permettre la visualisation des signaux. Il faut donc prendre en compte ce paramètre lors de son utilisation. Dans notre, cas nous avons largement assez de ressource pour nous en servir sans restriction. System console va nous permettre d'envoyer nos commandes avalon via la liaison jtag à notre esclave avalon. Voici les commandes utilisées :

```
% get_service_paths master
/devices/5CSEBA6(.|ES)|5CSEMA6|..@2#USB-1#DE-SoC/(link)/JTAG/alt_sld_fab_sldfabric.node_0/phy_0/master_0.master
% set m_path [ lindex [get_service_paths master] 0]
/devices/5CSEBA6(.|ES)|5CSEMA6|..@2#USB-1#DE-SoC/(link)/JTAG/alt_sld_fab_sldfabric.node_0/phy_0/master_0.master
% open_service master $m_path

% master_write_16 $m_path 0x00300004 50

% master_write_16 $m_path 0x00300008 5

% master_write_16 $m_path 0x0030000c 45000

% master_write_16 $m_path 0x00300010 30000

% master_write_16 $m_path 0x00300014 15000

% master_write_16 $m_path 0x00300018 4
```

```
% master_read_16 $m_path 0x00300004 1
0x0032
```

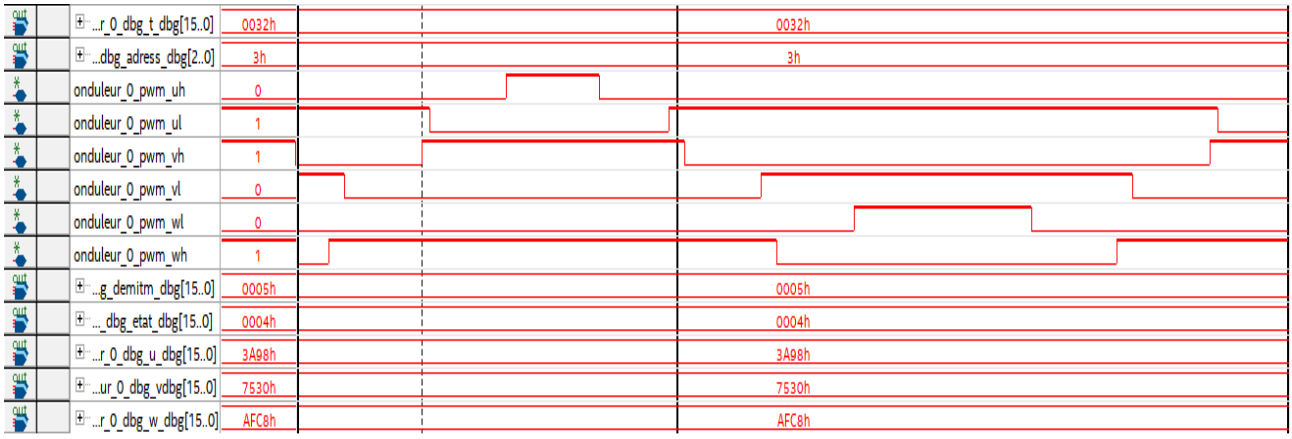
On écrit dans les registre 16 bits via master_write_16 en spécifiant l'adresse et la valeur. Il faut prendre en compte l'alignement de l'adresse indiquer dans la documentation sur le bus avalon. C'est pour cette raison qu'on passe d'un registre à l'autre en augmentat l'adresse de 4.

Master Byte Address (1)	Access	32-Bit Master Data		
		When Accessing an 8-Bit Slave Interface	When Accessing a 16-Bit Slave Interface	When Accessing a 64-Bit Slave Interface
0x00	1	OFFSET[0]$_{7..0}$	OFFSET[0]$_{15..0}$ (2)	OFFSET[0]$_{31..0}$
	2	OFFSET[1]$_{7..0}$	OFFSET[1]$_{15..0}$	—
	3	OFFSET[2]$_{7..0}$	—	—
	4	OFFSET[3]$_{7..0}$	—	—
0x04	1	OFFSET[4]$_{7..0}$	OFFSET[2]$_{15..0}$	OFFSET[0]$_{63..32}$
	2	OFFSET[5]$_{7..0}$	OFFSET[3]$_{15..0}$	—
	3	OFFSET[6]$_{7..0}$	—	—
	4	OFFSET[7]$_{7..0}$	—	—
0x08	1	OFFSET[8]$_{7..0}$	OFFSET[4]$_{15..0}$	OFFSET[1]$_{31..0}$
	2	OFFSET[9]$_{7..0}$	OFFSET[5]$_{15..0}$	—
	3	OFFSET[10]$_{7..0}$	—	—
	4	OFFSET[11]$_{7..0}$	—	—
0x0C	1	OFFSET[12]$_{7..0}$	OFFSET[6]$_{15..0}$	OFFSET[1]$_{63..32}$

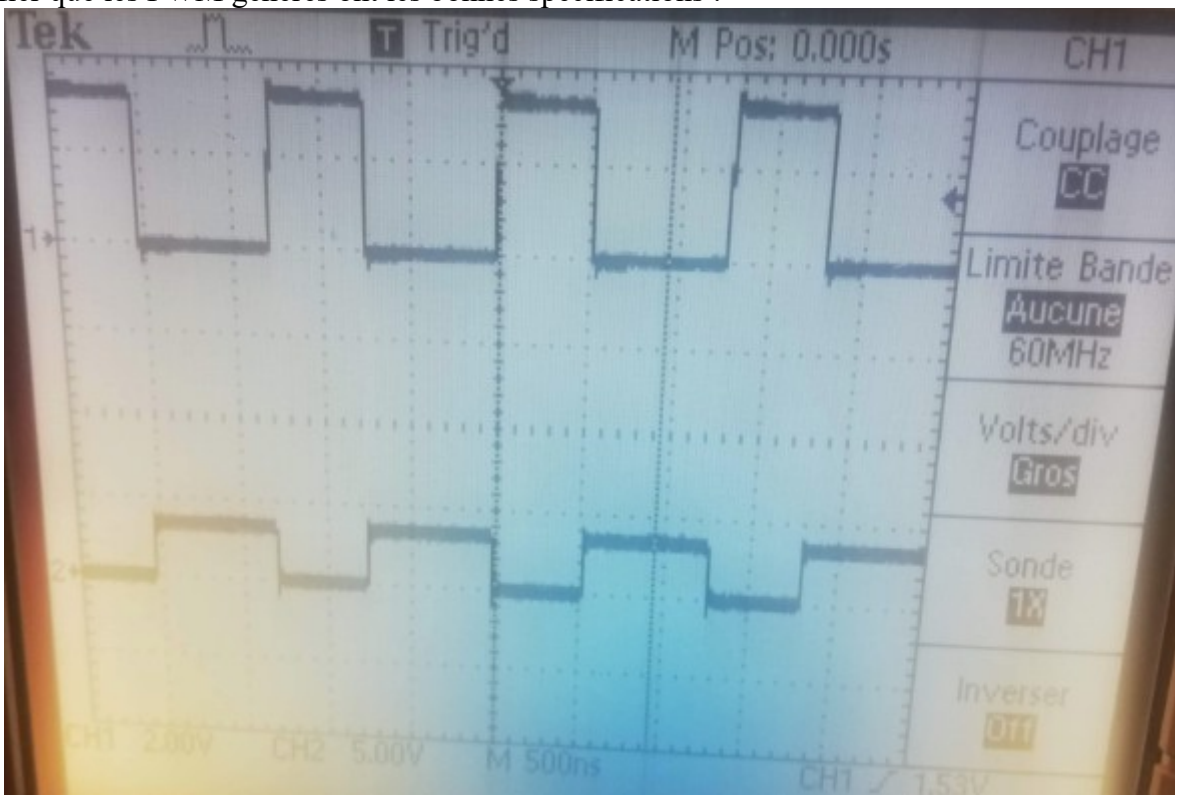
continued...

Initialement l'onduleur est arrêté. On le lance en clearant le bit MA du registre d'état ce qui correspond à la dernière commande. On vérifie ensuite la lecture avec la commande

master_read_16. Ensuite sur signal tap, on peut visualiser les signaux obtenues et vérifier la bonne écriture dans les registres via les signaux de débug :



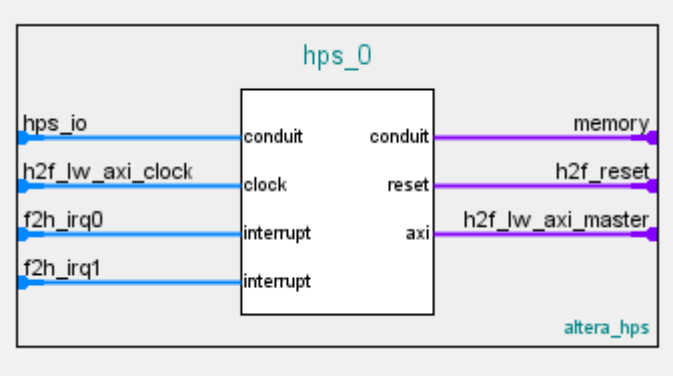
Les PWM sont correctement générés et l'écriture dans les registres se déroule correctement. Donc on peut en déduire que la communication avalon est validée. Suite à cela, il est nécéssaire de vérifier que les PWM générés ont les bonnes spécifications :



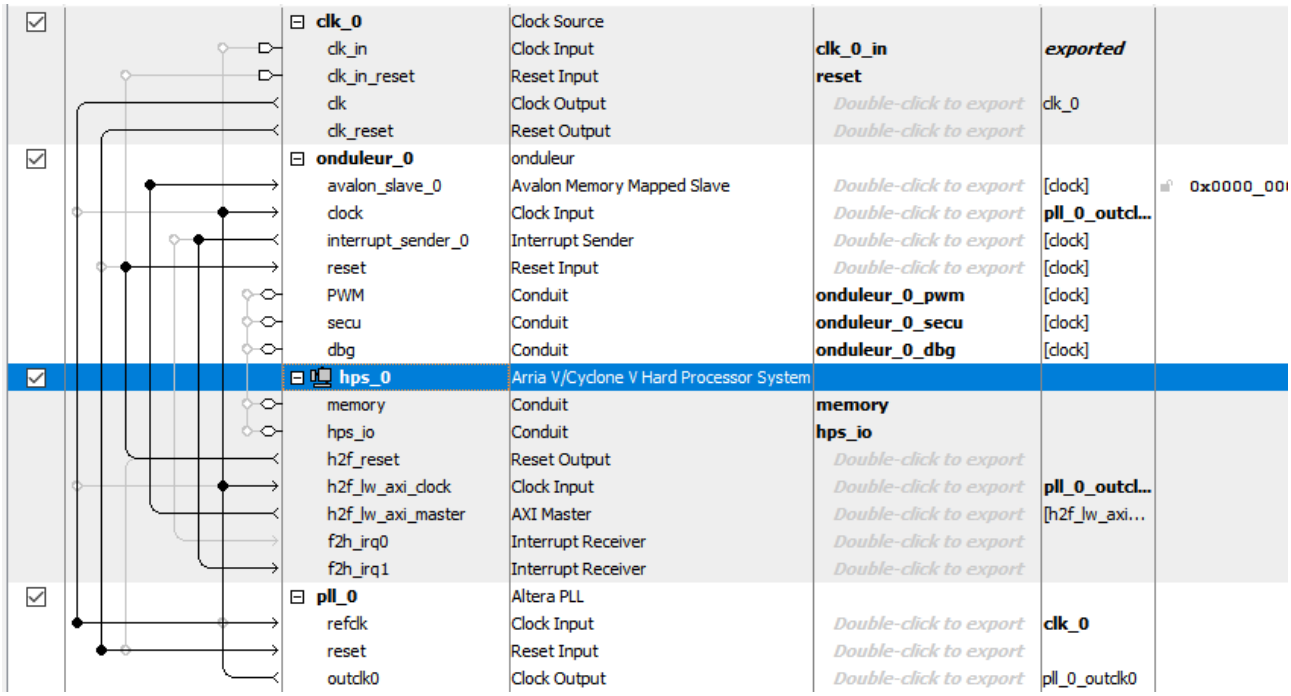
La période sur la figure est d'environ 1250 ns, sachant la demi-période spécifier était de 64 cycle d'horloge on obtient 1250/128=9,7 soit 10 ns, ce qui correpond à 100 MHz. Le système est donc fonctionnel.

Etude des fonctions logiciels à implémenter

Dans cette partie on s'intéresse à l'intégration de la partie HPS dans le design déjà fonctionnel. Il est donc nécéssaire de remplacer la jtag-to-avalon-master-bridge par le HPS qui fera office de maître avalon. Il faut donc retourner dans Qsys pour re-synthétiser le .qip correspondant. Premièrement, on configure le HPS :



On coche interruptions FPGA2HPS pour disposer des intéruptions avec l'interrupt sender de l'onduleur. Le bus choisi est le light-weight 32 bit qui est, selon la documentation très bien adapté à l'écriture dans des registres. Les autres valeurs ont étaient prise dans l'exemple du GHRD du cd_rom téléchargeable sur le site de Terasic. On intégre ensuite cette IP dans le système existant.



L' esclave avalon est directement relié au maître axi de l'HPS via une plateforme d'interconnexion non visible sur Qsys. Cette plate-forme permet de na pas avoir à nous en occuper nous même. Après avoir regénérer le VHDL, il est nécéssaire d'éxecuter deux scripts TCL qui permettront de contraindre les pins de l'HPS. Afin de préparer l'écriture du driver on doit générer un header contenant différente Macro utile pour le programme c du driver. Cette header est générer via un script trouver dans le GHRD appelé « generate_hps_qsys_header » :

```
#!/bin/sh
sopc-create-header-files "onduleur_debug_bis.sopcinfo" --single hps_0.h --module hps_0
```

Cette commande nécéssite d'avoir télécharger EDS et ajouter le dossier contenant les fichiers binaires dans le path de nos variables d'environement. Le fichier générer est le suivant

```
#ifndef _ALTERA_HPS_0_H_
#define _ALTERA_HPS_0_H_

/*
 * This file was automatically generated by the swinfo2header utility.
 *
 * Created from SOPC Builder system 'onduleur_debug_bis' in
 * file 'onduleur_debug_bis.sopcinfo'.
 */

/*
 * This file contains macros for module 'hps_0' and devices
 * connected to the following master:
 *   h2f_lw_axi_master
 *
 * Do not include this header file and another header file created for a
 * different module or master group at the same time.
 * Doing so may result in duplicate macro names.
 * Instead, use the system header file which has macros with unique names.
 */

/*
 * Macros for device 'onduleur_0', class 'onduleur'
 * The macros are prefixed with 'ONDULEUR_0_'.
 * The prefix is the slave descriptor.
 */
#define ONDULEUR_0_COMPONENT_TYPE onduleur
#define ONDULEUR_0_COMPONENT_NAME onduleur_0
#define ONDULEUR_0_BASE 0x0
#define ONDULEUR_0_SPAN 32
#define ONDULEUR_0_END 0x1f


#endif /* _ALTERA_HPS_0_H_ */
```

Un prototype de driver a était écrit mais non tester par manque de temps . Il ne contient que l'initialisation avec la lecture de la mémoire et le mmapping.

```
#include <stdio.h>
#include <stdliblib.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <signal.h>

#define REG_BASE 0xff200000
#define REG_SPAN 0x00200000


void *virtual_base;
uint32_t *onduleur;
int fd;

void handler (int signo) {

    *onduleur = 0;
    munmap(virtual_base,REG_SPAN);
    close(fd);
    exit(0);
}
```

```
int main() {


    int i =0;
    // map the address space for the LED registers into user space so we can interact with them.
    // we'll actually map in the entire CSR span of the HPS since we want to access various registers within that span

    if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
        printf( "ERROR: could not open \"/dev/mem\"...\n" );
        return( 1 );
    }

    virtual_base = mmap( NULL, REG_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, REG_BASE );

    if( virtual_base == MAP_FAILED ) {
        printf( "ERROR: mmap() failed...\n" );
        close( fd );
        return( 1 );
    }

    onduleur = (uint32_t)(virtual_base + ONDULEUR_BASE);
    signal(SIGINT,handler);

}
```

La suite aurait était de générer le bit stream .rbf et l'ajouter à la clé bootable contenant l'image Linux. Cross-compiler le code svm.c et l'envoyer via ssh sur le SoC puis l'éxecuter.

Conclusion :

Ce TP a était très fomatteur en ce qui concerne le flot de conception sur FPGA. La grande place laissé à l'autonomie dans ce TP a était dans un premier temps déroutant mais finalement très bénéfique. J'ai non seulemet appris à concevoir un système complet sur SoC FPGA mais aussi chercher et trouver les informations sur google. Les vidéos de formation intel sur system_console et signal tap ont était particuliérement utile. Les documentations aussi étant très bien détaillées trouver les bonnes informations n'étaient pas très compliqué. La difficulté était de les comprendre et les assimiler. L'autre point dur était de comprendre les erreurs de compilations qui ne sont parfois pas très explicite. Le système mis au point est fonctionnel du point de vue la communication avalon, cependant, la partie HPS n'as pas pu être terminée par manque de temps. J'ai cependant une bonne idée de ce qu'il serait nécéssaire de faire pour aller plus loin et je vais malgrés la fin des TP continuer à travailler sur ce projet pour le terminer.