IUT de La Rochelle

Université La Rochelle

TP projet : conception sur CLP
Réalisation d'un modulateur 8QAM

par [redacted] *& Édouard Lumet*

IT La Rochelle

iut réseaux & télécoms

Sommaire

Introduction....3
1. Générateur pseudo-aléatoire....4
2. Codeurs I/Q....5
2.1. Codeur I....5
2.2. Codeur Q....7
3. Générateur de sinusoïdes....8
3.1. Générateur « sinus »....8
3.2. Générateur « cosinus »....9
4. Multiplexeur (MUX)....10
5. Horloge et divisions de fréquence, additionneur, multiplieur....11
Conclusion....12

Introduction

Après plusieurs TPs sur le langage VHDL, nous entamons notre TP projet qui se déroule sur 2 séances.

Ce TP a pour objectif, de réaliser un modulateur d'amplitude en quadrature comportant une constellation en 8 états (8QAM). Ce TP projet se déroulera sur 2 séances de 3H, par binôme.

Notre modulateur devra être intégrable dans le FPGA de la carte de développement DE2.

Notre projet comportera :

- 1 entrée d'initialisation permettant de configurer le CNA (INIT),
- 1 entrée de remise à zéro du système (RST),
- 1 entrée de sélection permettant d'aiguiller soit la signal modulé soit les signaux IQ vers le CNA,
- 1 sortie analogique disponible en sortie du CNA sur le connecteur LINE OUT de la carte DE2.

Pour réaliser, ce projet nous devons assembler 7 sous-projets :

- un générateur pseudo-aléatoire,
- un générateur de sinusoïdes,
- un codeur I,
- un codeur Q,
- un multiplieur,
- un additionneur,
- un multiplexeur (MUX).

1. Générateur pseudo-aléatoire

Ce générateur fournit un code pseudo-aléatoire sur 3 bits à une fréquence de 100Hz afin de pouvoir tester le modulateur. Il permet ainsi de simuler en entrée un signal non déterministe comme une série de données analogiques.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY gene_pa IS
        PORT (
                H : IN std_logic;
                NRZ : OUT std_logic_vector(2 downto 0) );
END gene_pa ;

ARCHITECTURE comport OF gene_pa IS
        SIGNAL Q : STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
        PROCESS (H)
        BEGIN
        if (RISING_EDGE(H)) then
                Q(0) <= Q(3) XNOR Q(2);
                Q(1) <= Q(0);
                Q(2) <= Q(1);
                Q(3) <= Q(2);
        end if;
        END PROCESS;
        NRZ <= Q(3) & Q(2) & Q(1);
END comport;
```

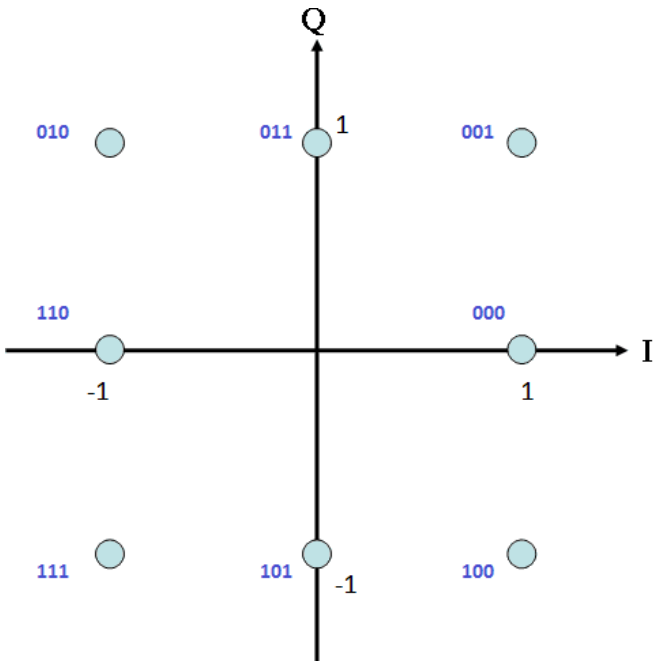
Description VHDL de notre générateur pseudo-aléatoire

Il génère de façon pseudo-aléatoire l'un des codes suivants : *000*, *001*, *010*, *011*, *100*, *101* et *111*. En réalité, si à échelle temporelle réduite le code paraît aléatoire, à une échelle plus importante on peut identifier une série de 2^n bits qui se répète (où n est le nombre de bascules composant le générateur).

Le message ainsi fournit sera codé par un codeur I et un codeur Q.

2. Codeurs I/Q

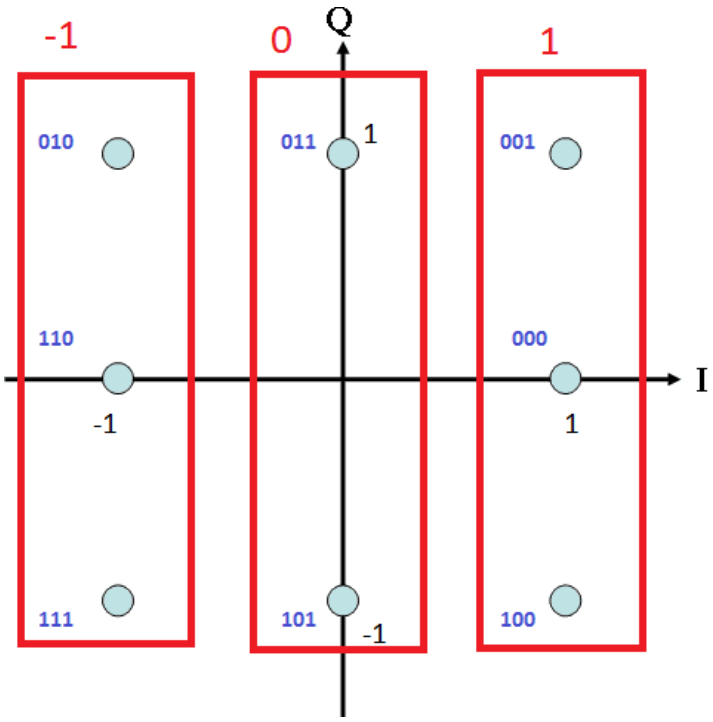
La constellation décrivant le codeur I/Q est la suivante :



2.1. Codeur I

Nous avons programmé le codeur I en VHDL comme suit :

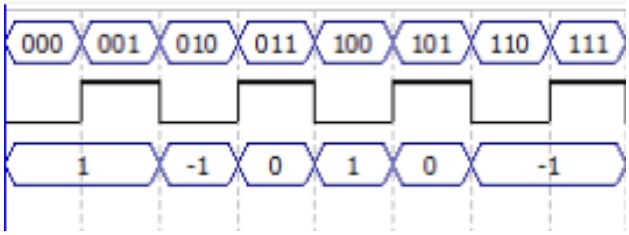
```
library ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY codeur_i IS
            PORT
            (
                    e: IN STD_LOGIC_VECTOR (2 downto 0);
                    s: OUT STD_LOGIC_VECTOR (2 downto 0)
                    );
END codeur_i;

ARCHITECTURE codeurarch of codeur_i IS
BEGIN
with e select
s <=                "001" when "001",
                    "001" when "000",
                    "001" when "100",
                    "111" when "010",
                    "111" when "110",
                    "000" when "011",
                    "000" when "101",
                    "111" when "111";
END codeurarch;
```



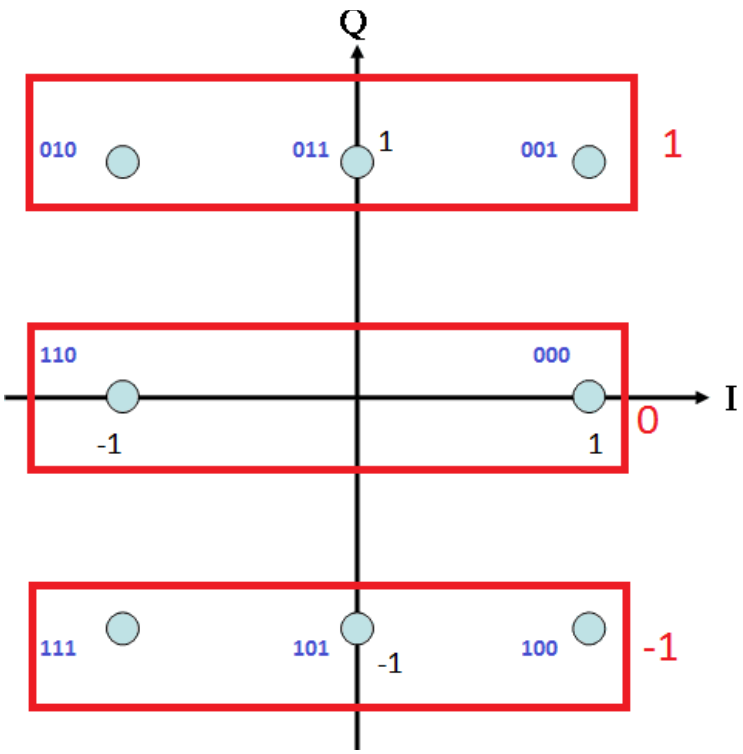
Son comportement est en effet le suivant :

- lorsque l'entrée est *011* ou *101*, il code la valeur *0* en sortie,
- lorsque l'entrée est *001* ou *000* ou *100*, il code *1* en sortie,
- lorsque l'entrée est *010* ou *110* ou *111*, il code *-1* en sortie.

Une simulation permet de s'assurer du bon fonctionnement du codeur. En haut on a les valeurs données en entrées, et les valeurs en sortie sont en bas du graphe.



2.2. Codeur Q



Le comportement du codeur Q est quant à lui le suivant :

- lorsque l'entrée est *000* ou *110*, il code *0* en sortie,
- lorsque l'entrée est *001* ou *010* ou *011*, il code *1* en sortie,
- lorsque l'entrée est *100* ou *101* ou *111*, il code *-1* en sortie.

La programmation en VHDL de ce codeur est la suivante :

```
library ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY codeur_q IS
            PORT
            (
                  e: IN STD_LOGIC_VECTOR (2 downto 0);
                  s: OUT STD_LOGIC_VECTOR (2 downto 0)
                  );
END codeur_q;

ARCHITECTURE codeurarch of codeur_q IS
BEGIN
with e select
s <=              "001" when "001",
                  "001" when "010",
                  "001" when "011",
                  "111" when "100",
                  "111" when "101",
                  "000" when "110",
                  "000" when "000",
                  "111" when "111";
END codeurarch;
```

3. Générateur de sinusoïdes

Nous avons décidé séparer les fonctions sinus et cosinus, créant ainsi deux générateurs de sinusoïdes.

3.1. Générateur « sinus »

Pour programmer le générateur « sinus », nous avons repris le code de « romsinus » du TP3 qui permettait de générer une sinusoïde sur 16 valeurs. Ici, nous avons remplacé les 16 valeurs par les 256 valeurs fournies sur Moodle et nous avons ajusté la taille des bus d'adresses et de données. La description VHDL est la suivante :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY gene_sinus IS
PORT(
        H       : IN std_logic;
        Read: IN std_logic;
        A       : IN std_logic_vector(7 downto 0);
        D       : OUT std_logic_vector(15 downto 0));
END gene_sinus ;
ARCHITECTURE comport OF gene_sinus IS
        TYPE ROM_Array IS ARRAY (0 to 255) of std_logic_vector(15 downto 0);

        constant Contenu: ROM_Array := (
        0 => x"fcdd" ,
        1 => x"fe6e" ,
        ...
        252 => x"f69d" ,
        253 => x"f82b" ,
        254 => x"f9bb" ,
        255 => x"fb4b" );

BEGIN
 PROCESS(H)
 BEGIN
        IF rising_edge(H) THEN
                IF Read = '1' THEN
                                D       <= Contenu(conv_integer(A));
                ELSE
                                D       <= "ZZZZZZZZZZZZZZZZ";
                END IF;
        END IF;
 END PROCESS;
END comport;
```

3.2. Générateur « cosinus »

Pour le générateur « cosinus », la description VHDL est la même que précédemment. Il a simplement fallu intégrer un déphasage de π/2 en déplaçant les 64 dernières valeurs au début. Nous avons du ensuite renuméroter les index.

Ce déphasage correspond à 1/4 du cercle trigonométrique, soit un décalage des valeurs de 1/4. Or 256/4=64, ce qui explique le fait d'avoir déplacer les 64 dernières valeurs.

4. Multiplexeur (MUX)

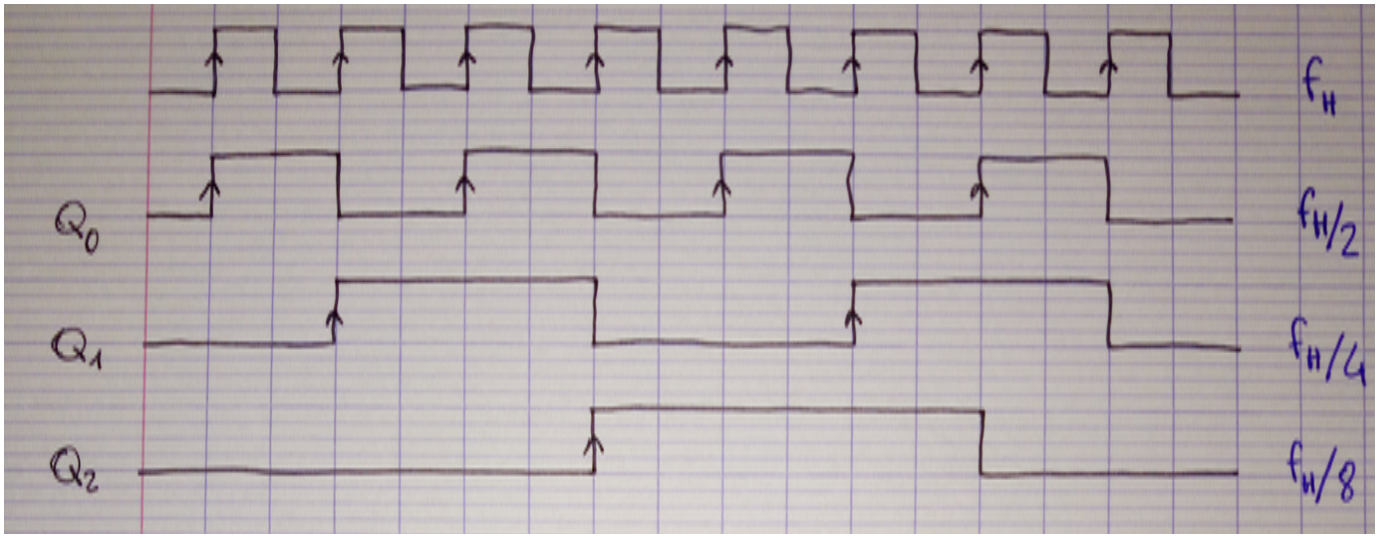
Le principe de ce multiplexeur est de diriger vers le CNA soit la sortie du modulateur soit les sorties des codeurs I et Q. Cependant, le CNA est un CNA 24 bits donc il faut adapter les données du modulateur depuis un format 19 bits et adapter les données du codeur I/Q depuis un format 6 bits. Dans le premier cas, on ajoute 5 zéros à la fin des données. Dans le second cas on teste si, sur 3 bits en complément à 2, I ou Q vaut 1, 0 ou -1 pour ensuite coder en sortie respectivement les valeurs maximale, nulle ou minimale sur 24 bits en complément à 2. La description VHDL de MUX est alors la suivante :

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY mux0 IS
        PORT (
                I,Q : IN std_logic_vector(2 downto 0);
                SIG : IN std_logic_vector (18 downto 0);
                S : IN std_logic;
                CNAL,CNAR : OUT std_logic_vector (23 downto 0)
                );
END mux0;

ARCHITECTURE comport OF mux0 IS
        BEGIN
        PROCESS(S)
        BEGIN
        if (S='0') then
                CNAL <= SIG & "00000" ;
                CNAR <= SIG & "00000" ;
        else
                if (I="111") then
                        CNAR <= "100000000000000000000000";
                elsif (I="000") then
                        CNAR <= "000000000000000000000000";
                elsif (I="001") then
                        CNAR <= "011111111111111111111111";
                elsif (Q="111") then
                        CNAL <= "100000000000000000000000";
                elsif (Q="000") then
                        CNAL <= "000000000000000000000000";
                elsif (Q="001") then
                        CNAL <= "011111111111111111111111";
                end if;
        end if;
        END PROCESS;
END comport;
```

5. Horloge et divisions de fréquence, additionneur, multiplieur

Pour que le modulateur soit synchrone, on utilise la même horloge 50MHz présente sur la carte. Pour disposer de plusieurs fréquences à partir de cette même horloge, on utilise un diviseur de fréquence. Nous créons celui-ci en détournant un compteur de type 'lpm_counter'.

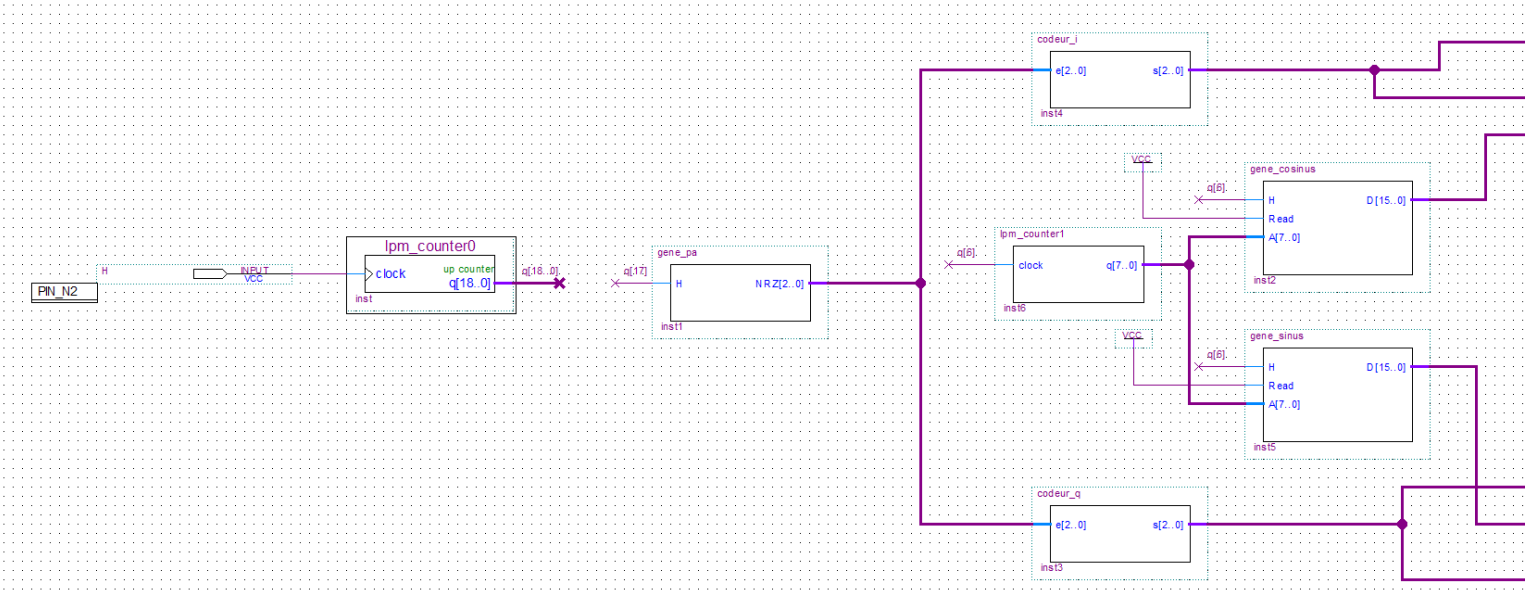
Le principe étant le suivant :

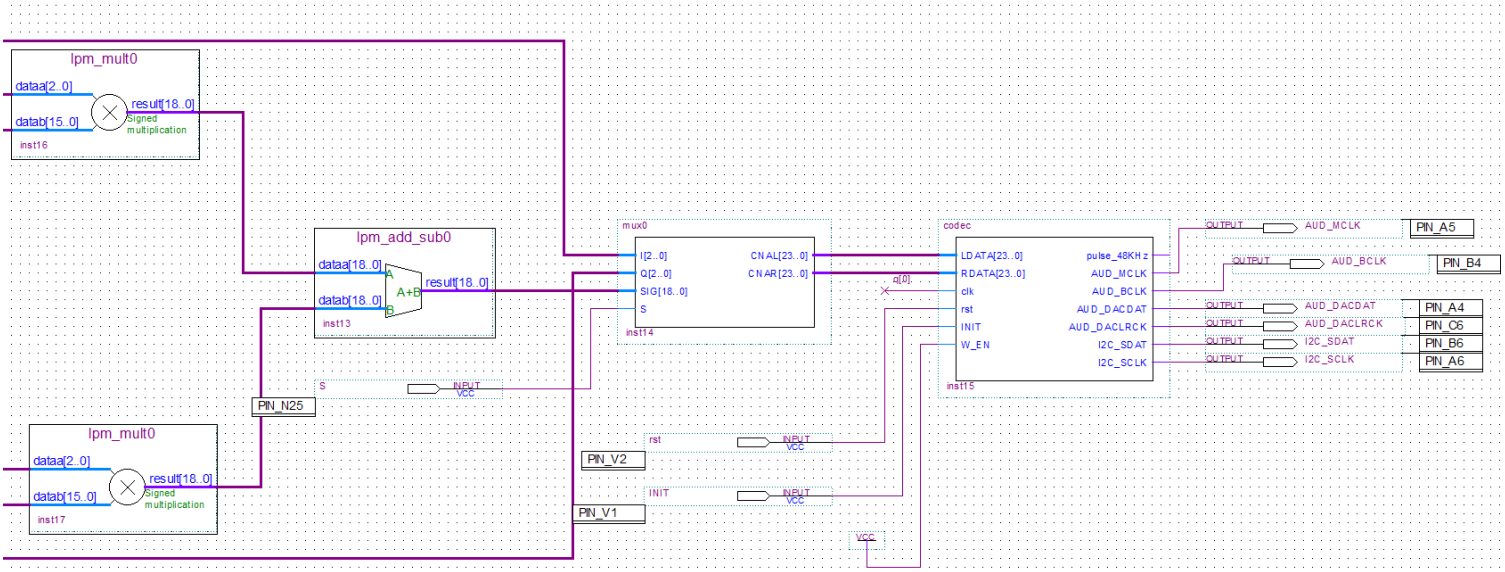


Ensuite, concernant les additionneurs et le multiplieur, on utilise également des fonctions fournies par Quartus. Elles sont respectivement lpm_add_sub et lpm_mult.

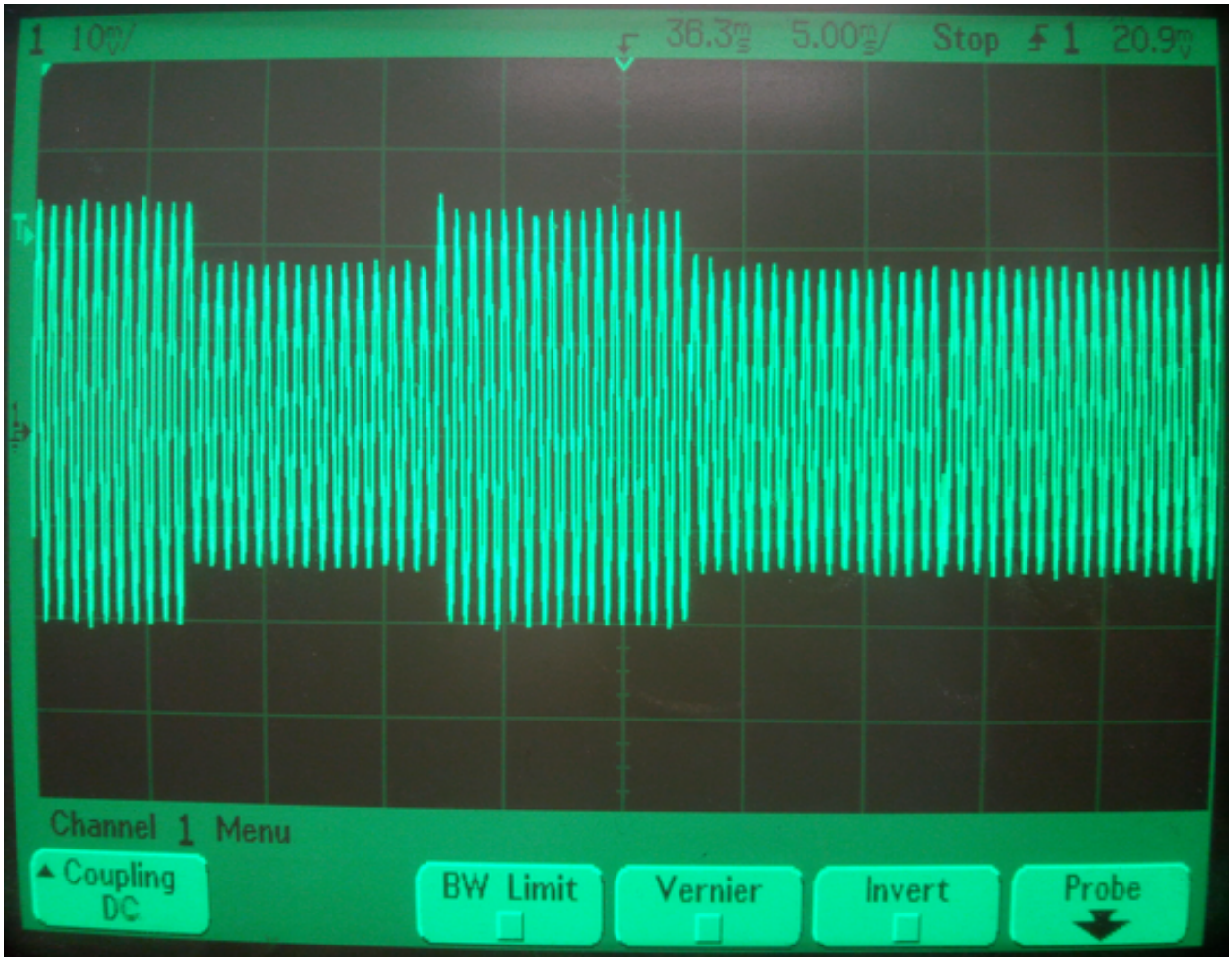
Conclusion

Notre projet global à l'allure suivante :



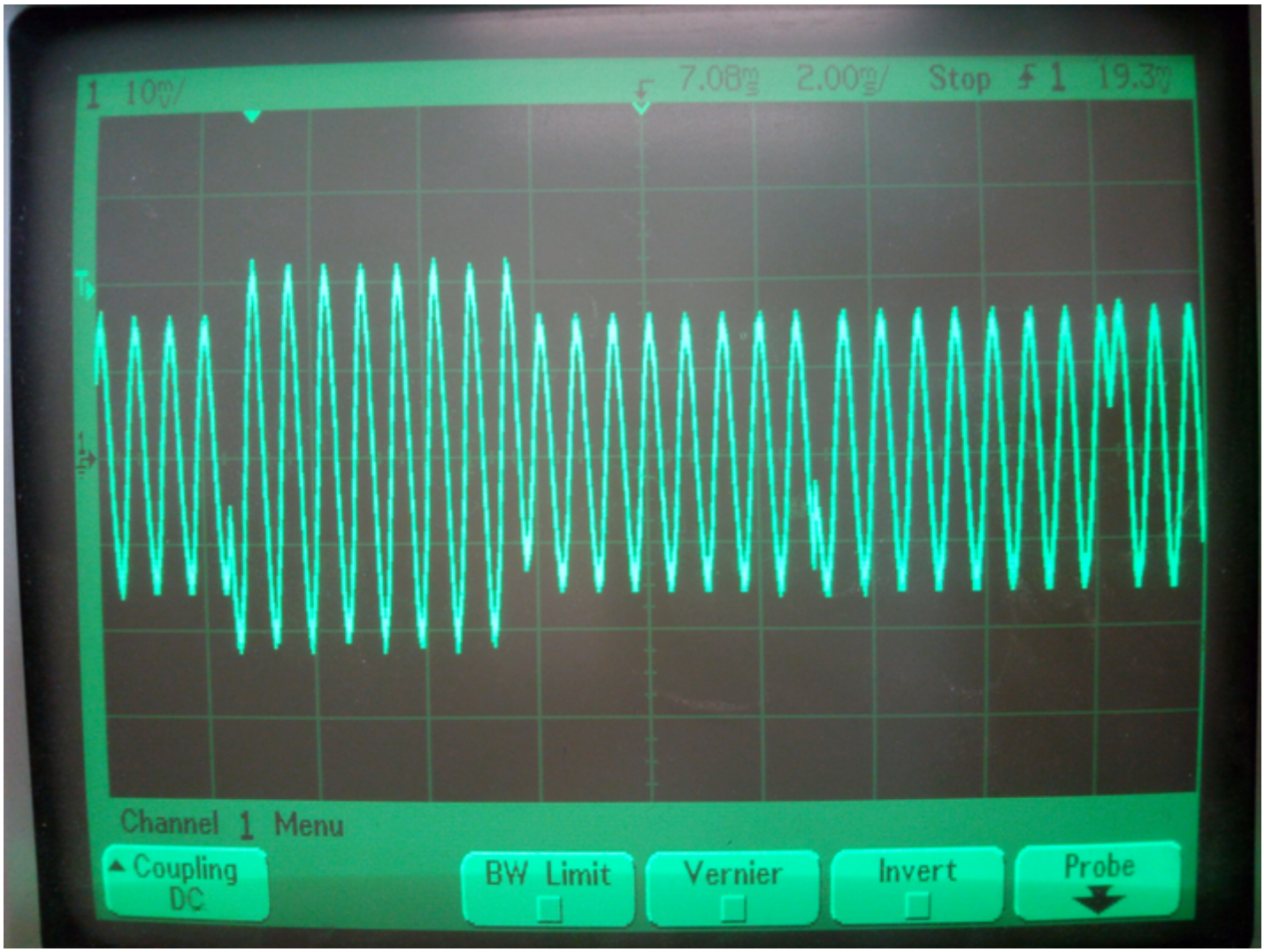


Nous avons ensuite relié la carte à l'oscilloscope afin de visualiser le résultat final :



On peut voir ici que l'on obtient bien deux niveaux d'amplitude seulement. De plus, un motif du message a une largeur de deux divisions soit 100ms, ce qui correspond bien à une fréquence du message de 100Hz. En revanche, le nombre de périodes dans un message devrait être de 10 environ (f(message) = 100Hz, f(modulation) = 1kHz), or il est ici du double environ. On peut supposer que la fréquence des générateurs de sinusoïdes est deux fois trop élevée. En effet, après réflexion, avec q[6] on a $50000000/2^{6+1}*256 \approx 1500$. q[7] nous donnerait un meilleur résultat.

Enfin, un second problème apparaît comme on peut le voir sur cet oscillogramme :



On observe un changement d'amplitude sans saut de phase, ce qui n'est pas possible d'après la constellation à 8 états.