# Stepper Motor : Tutoriel

This document is a tutorial allowing the reader to understand the way of thinking that guided us throughout this project from its beginning to its end. This work was carried out within the framework of the Hardware/Software Platforms 2020-2021 course at the Faculté Polytechnique de Mons under the direction of Professor Carlos Alberto Valderrama Sakuyama and helped by his assistant Mohamed.

This document will be arranged around the arrangement described below, allowing you to better understand the construction of the project :

1. Objective
2. Constraints
3. Presentation of the different parts
4. VHDL
5. Test Bench
6. Platform designer, GHRD & Pin Planner
7. C code
8. Passage to the electronic board
9. Explanation of how it works
10. Conclusion

# 1. Objective

The objective of this project is to be able to control a stepper motor using an FPGA board and therefore to realize a hardware part and a software part to achieve this. Each of these two parts have a specific role and are more than necessary to obtain the expected result.

Firstly, the VHDL code is the code of the FPGA part of the board. This is the soul of the project. It is in this hardware part that we will find the definition of the entity as such. It will therefore give the margin to follow to the board in order to output what we expect according to the inputs we provide. Overall, it is in this part that the whole process will take place.

In the second part, namely the software part written in C, the thought pattern is quite different. This part, which will be managed by the processor, will have the objective of spying on the signals of the hardware part. This may seem complex, but it is not. The principle is as follows : we will use this code to carry out a sort of real simulation where we will replace the electronic control inputs (in our case) of the motor with a series of commands pre-written in C. This will then allow us to control the motor by computer. As for the spying part, this is more understandable from the point of view of a sensor and not a motor. We will actually come to listen to the registers that we know in the hardware part. In this way, we are able to use our knowledge of what is in these registers to print out the values that we want to know.

In short, the hardware side of the FPGA represents the structure carrying out its mission as we command it, and the software side, through our knowledge of the registers used and their addresses, will write to these registers in order to command them (in the case of the motor) or listen to what is happening in the registers so as to be able to externalise and print what is happening there (in the case of the sensor).

# 2. Constraints

In order to carry out this project, we were asked to use an Altera DE0 nano FPGA board which has a 50MHz clock and a special board to provide the necessary power to drive the stepper motor. Indeed, the FPGA board alone cannot provide enough power to do this on its own.

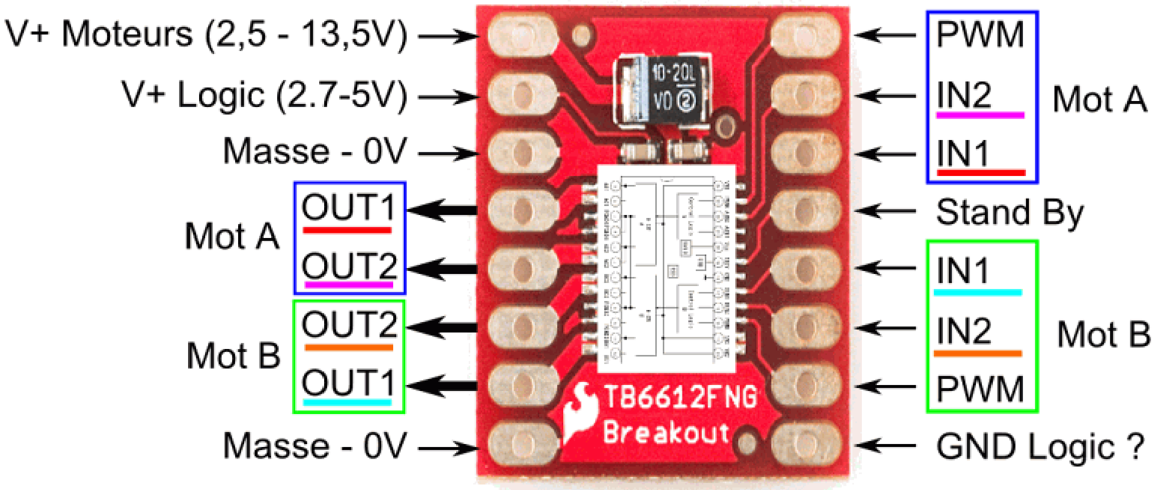



*Image of the board that provides the power to drive the motor.*
*(Dual 1A motor driver interface board based on the CI tb6612fng)*

It is a priori possible to control a bipolar stepper motor with this card. To do this:
STANDBY is connected to logic HIGH (+5V) : the IC is active.
PWM of each stage is connected to logic HIGH (+5V) : the state of the IN pins is reflected on the OUT pins in a transparent way.

The first motor phase is connected to stage A and the second motor phase to stage B. This will result in the following correspondence:

IN1-A = step 1 = input a
IN1-B = step 2 = input b
IN2-A = step 3 = input c
IN2-B = step 4 = input d

We must therefore play with these 4 wires which control the steps in order to drive our motor. A step in the clockwise direction performs the sequence from top to bottom = a-b-c-d. A step in the anti-clockwise direction will read the sequence in the opposite direction, i.e. d-c-b-a.

By simply reading this data sheet, we can see that the purpose of the FPGA is to provide these signals a,b,c,d to the red power board as well as a signal defining whether the motor should rotate or not and the direction of rotation.

# 3. Presentation of the different parts

In the previous point, it was determined that in order to be able to control the motor via the imposed power board, it is necessary that the FPGA board can provide the control signals for the step sequence a,b,c,d based on the control signals from the software part which are the direction and enable control signals to give the information respectively about the direction of rotation as well as whether there is rotation or not. All this is part of the hardware part of the project. We had to realize this hardware part according to the objectives and constraints.

The hardware part is written in VHDL using the INTEL Quartus platform. The hardware block is composed of the definition of the desired architecture (state machine -> a,b,c,d) as well as a driver in order to make the link between this state machine and the FPGA output. A bench test was also carried out in order to test this hardware part and to make sure that the output signals correspond to our expectations.

Finally, the aim being to carry out the control of the motor in practice from a computer, it is necessary to add a software part coded in C. This part allows to provide the motor control interface to the user in order to give instructions to the FPGA board.

It should be noted that we can directly drive our motor using the signals generated by the FPGA (and thanks to the power board). That's why we don't need to implement a communication protocol between our motor and the FPGA board, unlike if we had to exchange information between a sensor and the FPGA (and therefore we would have to implement a communication protocol like I2C).

# 4. VHDL

```vhdl
entity StepperMotorPorts is
    Port (
            StepDrive_A : out std_logic;
            StepDrive_B : out std_logic;
            StepDrive_C : out std_logic;
            StepDrive_D : out std_logic;
            clock : in std_logic;
            Direction : in std_logic;
            StepEnable : in std_logic;
            Reset : in std_logic --;
            --ProvideStaticHolding : in std_logic
      );
end StepperMotorPorts;

architecture StepDrive of StepperMotorPorts is

   signal StepDrive : std_logic_vector(3 downto 0);
   signal state : std_logic_vector(1 downto 0) := "00";
   signal StepCounter : std_logic_vector(31 downto 0) := (others => '0');
   constant StepLockOut : std_logic_vector(31 downto 0) := "00000000000000110000110101000000";

```

Here is the code allowing us to create the state machine. First we define the entity, that is to say we define the input and output ports of our block. The input ports are the clock (internal to the FPGA), the direction (clock = 1 or anti-clock = 0) and the StepEnable which allows us to say if we want to run the motor or not (1 or 0). The output signals are of course the signals a,b,c,d under the name of StepDrive_A, StepDrive_B, StepDrive_C, StepDrive_D. A reset signal is also added in order to reset the program if necessary.

Below the definition of the block ports is the definition of the signals. This is where the size of the registers associated with each signal is defined. The motor control signal is therefore a 4-bit register where each bit represents the signals a,b,c,d (StepDrive) respectively. The 2-bit state signal is used to define the state in which we are in the state machine ; 2 bits because it allows 4 combinations corresponding to the 4 states of our state machine (see below).StepCounter and StepLockOut, both coded on 32 bits, allow to implement a counter that delays the beginning of the program in order to give time to the control signal to make the motor run without it being done too quickly, because as a reminder, the operations are done on the basis of a clock at 50MHz. They define somehow the rotation speed of the motor.

Here we are in the heart of the program (below). At each rising edge of the 50 MHz clock, the counter will increment. Once the counter reaches the threshold value set previously, the counter is reset and the program can move on to the next step, i.e. the state machine, and thus produce an output signal. A reset block is also present to reset the state of all signals.

Before arriving at the state machine, the program will check if it is allowed to run and if so in which direction it can do so and thus know in which sense it will have to run the state machine.

```
begin

   process(clock)
   begin

      if ( (clock'Event) and (clock = '1') ) then


         if (Reset = '1') then

            state <= "00";
            StepCounter <= "00000000000000000000000000000000"; --(others => '0');
            StepDrive <= "0000";
            StepDrive_A <= '0';--StepDrive(3);
            StepDrive_B <= '0';--StepDrive(2);
            StepDrive_C <= '0';--StepDrive(1);
            StepDrive_D <= '0';--StepDrive(0);


         else

            StepCounter <= StepCounter + "00000000000000000000000000000001";

            if (StepCounter >= StepLockOut) then

               StepCounter <= "00000000000000000000000000000000";

               StepDrive <= "0000";

               if (StepEnable = '1') then

                  if (Direction = '1') then state <= state + "01"; end if;
                  if (Direction = '0') then state <= state - "01"; end if;
```

Finally, here is the state machine. The state machine defines the output signals depending on the state in which one is.

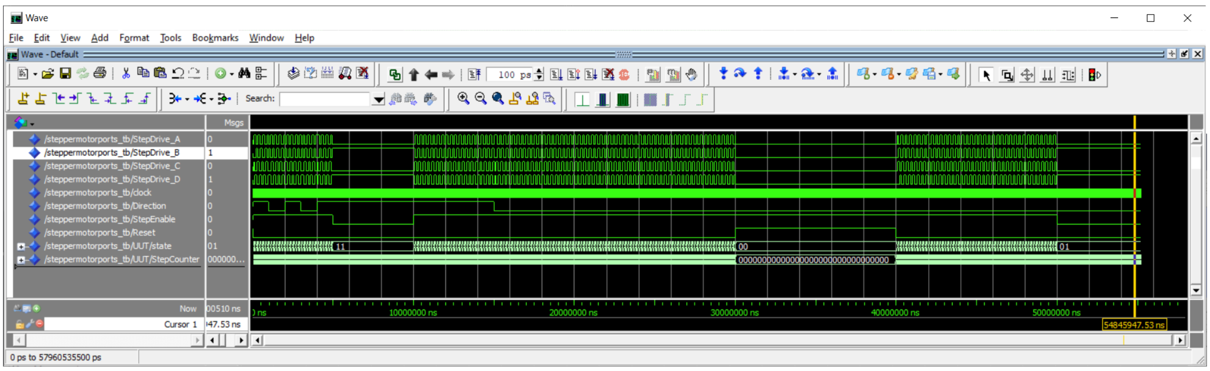
```
                  case state is

                     when "00" =>

                        StepDrive <= "1010";
                        StepDrive_A <= '1';--StepDrive(3);
                        StepDrive_B <= '0';--StepDrive(2);
                        StepDrive_C <= '1';--StepDrive(1);
                        StepDrive_D <= '0';--StepDrive(0);

                     when "01" =>

                        StepDrive <= "1001";
                        StepDrive_A <= '1';--StepDrive(3);
                        StepDrive_B <= '0';--StepDrive(2);
                        StepDrive_C <= '0';--StepDrive(1);
                        StepDrive_D <= '1';--StepDrive(0);

                     when "10" =>

                        StepDrive <= "0101";
                        StepDrive_A <= '0';--StepDrive(3);
                        StepDrive_B <= '1';--StepDrive(2);
                        StepDrive_C <= '0';--StepDrive(1);
                        StepDrive_D <= '1';--StepDrive(0);

                     when "11" =>

                        StepDrive <= "0110";
                        StepDrive_A <= '0';--StepDrive(3);
                        StepDrive_B <= '1';--StepDrive(2);
                        StepDrive_C <= '1';--StepDrive(1);
                        StepDrive_D <= '0';--StepDrive(0);

                     when others =>

                  end case; --state
```

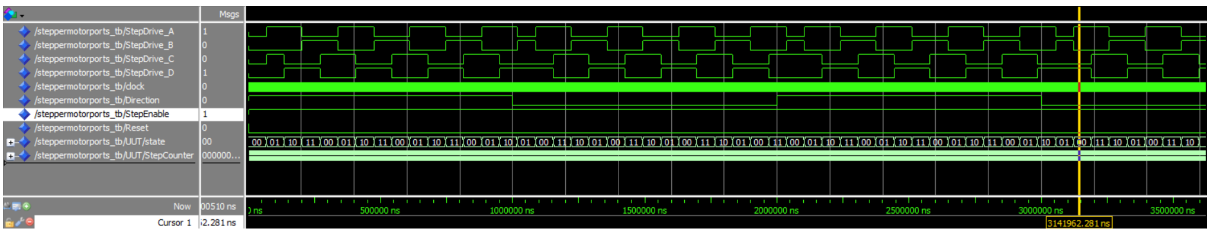
# 5. Test Bench

This part is fundamental before moving on to the next steps. Indeed, it will allow us to verify through a simulation that our VHDL code is working properly, and thus to ensure a coherent response of the sensor to the expectations we have in its favour. In our configuration, this test bench code is also written in VHDL and will run through an Altera ModelSim RTL simulation. But what is this simulation?

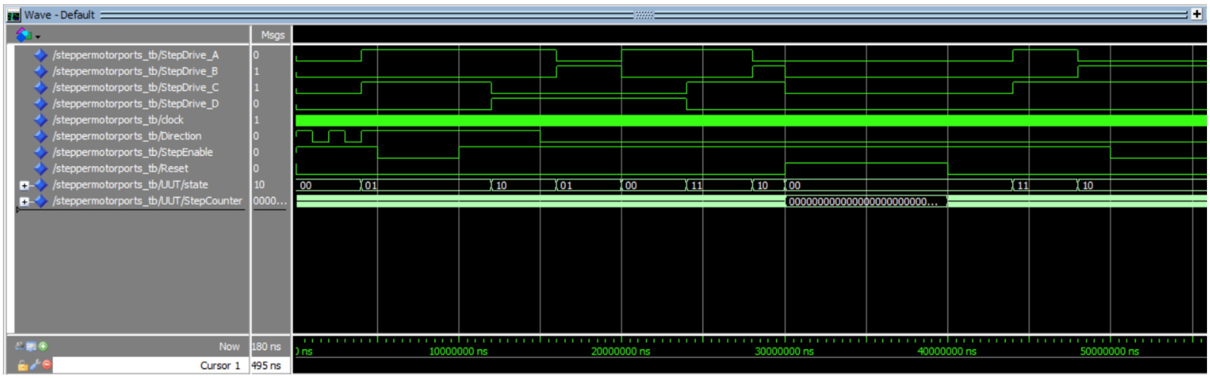
To set the scene, the code we have previously written has (as we have defined it) inputs and outputs, and what we want to do here is to check the behaviour of the outputs against the inputs, to simply check that what we have written is a correct translation of our expectations. To do this, we will define signals that we will vary over time and place them at the input of our entity. Once this is done, we can then observe the output signals.

N.B.: As we are simulating the behaviour here, we are not really connected to a clock on the card. We must therefore write a process for this clock and define its period.





*Zoom on the change of direction*



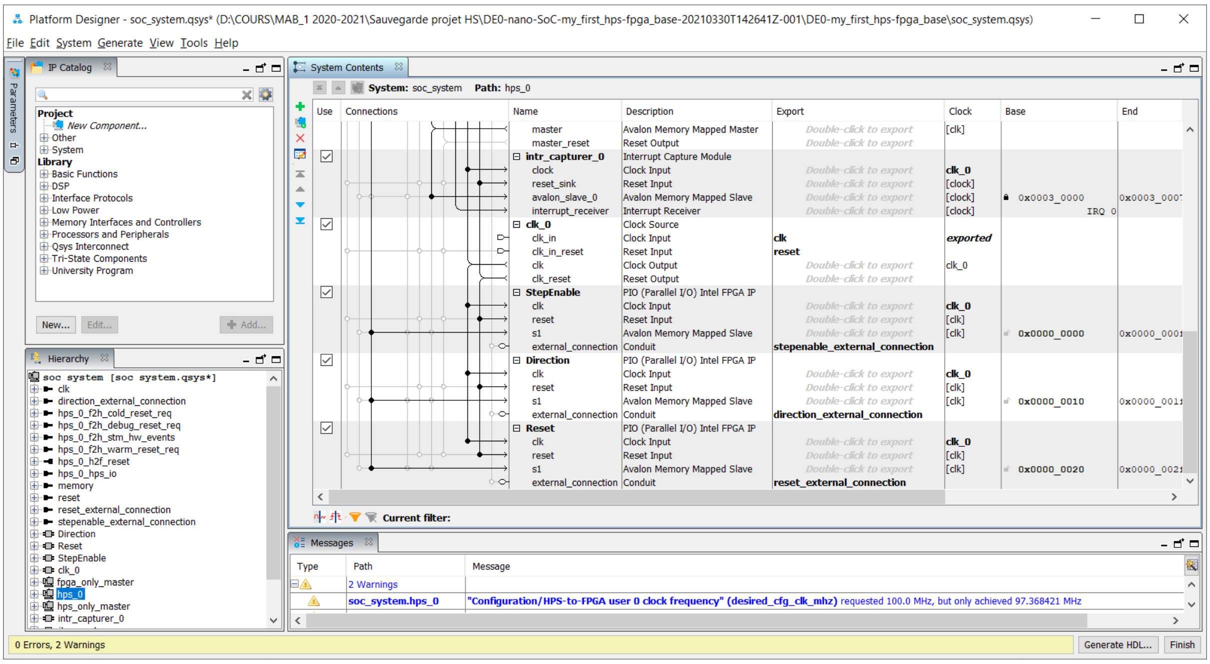

*Situation where the speed of rotation is lower than the frequency of the change of direction*

The images above are the views of the signals proposed by ModelSim. As expected, you can see the four outputs A, B, C and D varying according to the states, which themselves vary in a direction defined by the direction. We can also see that the StepEnable variable blocks at the current state and that the reset variable sets all our outputs to zero.

# 6. Platform designer, GHRD & Pin Planner

This short although simple-looking stage is rich in interest. The objective of this step is in fact to bring the link between the different parts. We will indeed allow the link between our different variables, their link to the outside and define their register and address.

First of all, it should be noted that it is necessary to have well defined our variables and outputs of the entity. Once this is done, we move on to Platform designer to assign their registers. To do this, as you can see below, it is necessary to create registers with their names, connect these registers to the wires that we see on the left in order to grant them an access and finally to define these registers as outputs ("external connection").




Then comes the very important step of the ghrd. This file is the big planner of what is in our FPGA. The information that we will add to it is essential. However, for the sake of simplicity, it is to a large part, synthesised by Platform Designer.

The information to be added is as follows :

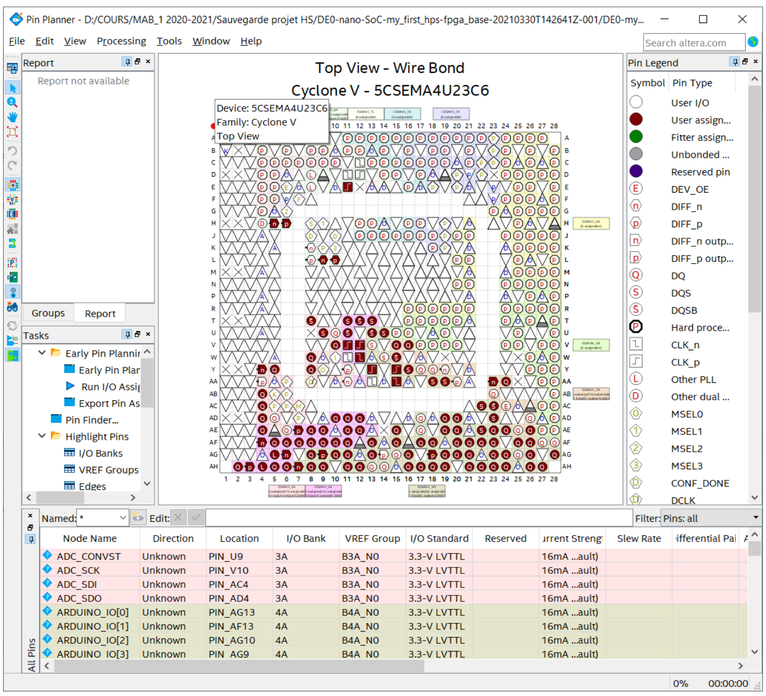
1. The wires allowing to link the input variables of our entity and their connection. We thus find below the definition "wire ..." of our three wires, followed in "soc_system u0" by their connection. In fact, as we can see, we place in the parentheses of our variables (set in "external connection export" mode) their wires.

```verilog
//=======================================================
//  REG/WIRE declarations
//=======================================================
// internal wires and registers declaration
  wire [1:0]  fpga_debounced_buttons;
  wire [7:0]  fpga_led_internal;
  wire        hps_fpga_reset_n;
  wire [2:0]  hps_reset_req;
  wire        hps_cold_reset;
  wire        hps_warm_reset;
  wire        hps_debug_reset;
  wire [27:0] stm_hw_events;

// connection of internal logics
  assign stm_hw_events    = {{13{1'b0}},SW, fpga_led_internal, fpga_debounced_buttons};


//=======================================================
//  Structural coding
//=======================================================
   wire stepenable_tofpga;
   wire direction_tofpga;
   wire reset_tofpga;

 soc_system u0 (
      .stepenable_external_connection_export(stepenable_tofpga),
      .reset_external_connection_export(reset_tofpga),
      .direction_external_connection_export(direction_tofpga),

      //Clock&Reset
      .clk_clk                              (FPGA_CLK1_50 ),            //  clk.clk
      .reset_reset_n                        (1'b1         ),            //  reset.reset_n
```

2. Our unit and all its variables. We will then again place the wires created just before in connection with the input variables. But we will also connect our clock to the card's clock and finally also assign the output pins (the GPIOs) to our output variables.

```verilog
StepperMotorPorts stp(
  .StepDrive_A(GPIO_0[0]),
  .StepDrive_B(GPIO_0[1]),
  .StepDrive_C(GPIO_0[2]),
  .StepDrive_D(GPIO_0[3]),
  .clock(FPGA_CLK1_50),
  .Direction(direction_tofpga),
  .StepEnable(stepenable_tofpga),
  .Reset(reset_tofpga)
);endmodule
```

When this second operation is done, a third may be necessary. Indeed, in the case of bidirectional pins, we need to indicate that these particular pins are bidirectional. We then use the "Pin Planner" tool (see below). All we have to do is change the nature of the pins we want to make bidirectional.



# 7. C code

The C code is used to control the motor through the FPGA. It is from this code that we give the instructions for the inputs (on the FPGA side) : enable (the motor must turn or not), direction (clockwise or anti-clockwise) and reset.

To do this, the addresses of the registers for the above-mentioned signals must first be defined in the C code, as well as the pointers that allow access to them. This is done in part thanks to the .h file that was previously created.

```c
int main() {

    //pointer to the different address spaces

    void *virtual_base;
    void *axi_virtual_base;
    int fd;


    void *h2p_lw_StepEnable;
    void *h2p_lw_Direction;
    void *h2p_lw_Reset;
    //void *h2p_lw_myBus_addr;
```

*Creation of the different pointers*

```c
//the address of the two input (reg1 and reg2) registers and the output register (reg3)
h2p_lw_StepEnable = virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST + StepEnable_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
h2p_lw_Direction = virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST + Direction_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
h2p_lw_Reset = virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST + Reset_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
```

*Assigning pointer addresses using parameter names from the .h file*

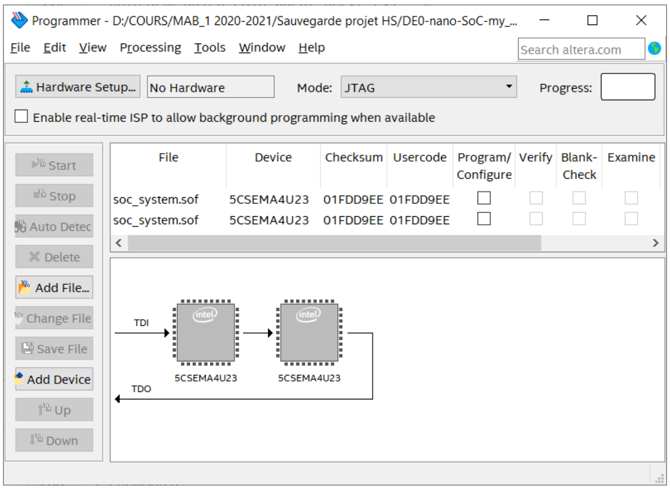
Here is the test code that we have done. A series of variable modification operations are performed. These commands are sent to the FPGA and according to this, the FPGA generates the signals a,b,c,d which will ultimately allow the motor to run.
Each time, we write directly to the register of the variable we wish to modify. Here, we will make the motor turn by +90° then stop it before making it turn again by +180°, +270° and finally -180° with a delay between each command in order to observe the changes.

```c
printf("\nRotation de +90 degres\n");

*(uint32_t *)h2p_lw_StepEnable = 1;
*(uint32_t *)h2p_lw_Reset = 0;

delay(Duree1tour / 4);
*(uint32_t *)h2p_lw_StepEnable = 0;

delay(3000);

printf("\nRotation de +180 degres\n");

*(uint32_t *)h2p_lw_StepEnable = 1;

delay(Duree1tour / 2);
*(uint32_t *)h2p_lw_StepEnable = 0;

delay(3000);

printf("\nRotation de +270 degres\n");

*(uint32_t *)h2p_lw_StepEnable = 1;

delay(Duree1tour * 0.75);
*(uint32_t *)h2p_lw_StepEnable = 0;

delay(3000);

printf("\nRotation de -180 degres\n");

*(uint32_t *)h2p_lw_StepEnable = 1;
*(uint32_t *)h2p_lw_Direction = 0;

delay(Duree1tour / 2);
*(uint32_t *)h2p_lw_StepEnable = 0;

delay(5000);
```

# 8. Passage to the electronic board

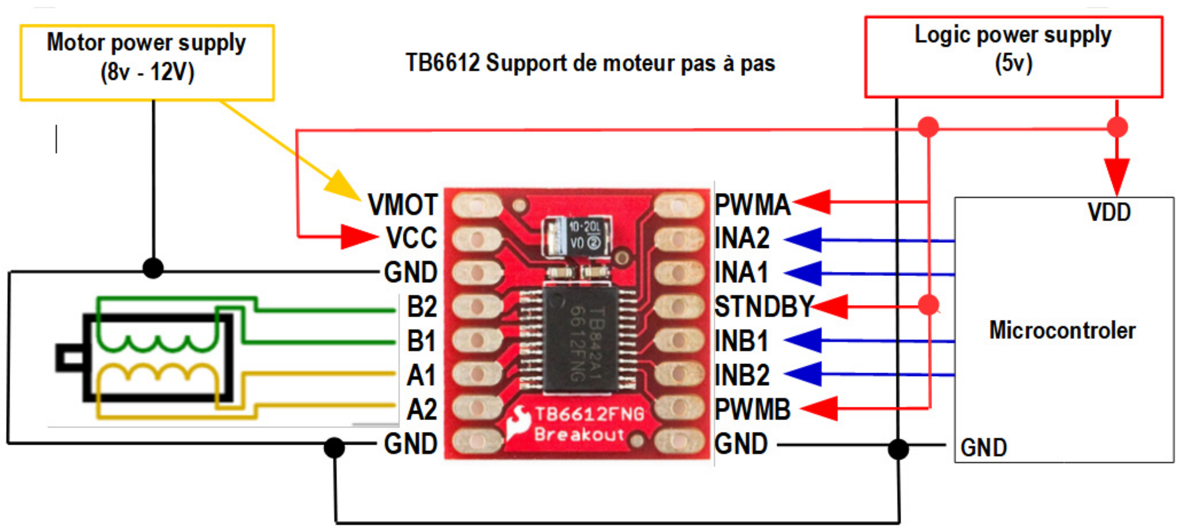
Arrived at this stage, bravo ! We will now be able to look at the deposit of your various codes on the card. This one, in the same way as the writing of the different hardware and software parts, will be done in several steps:

1. **Hardware** : For this part, nothing particularly complicated. You will just have to go through the "programmer" tool of Quartus once the project is fully compiled. Once in this page, select your files (and add the missing ones if necessary) so as to have successively your FPGA and your .sof generated at the end of your compilation. Once this is done, you can upload. The coded hardware part is now on your board.

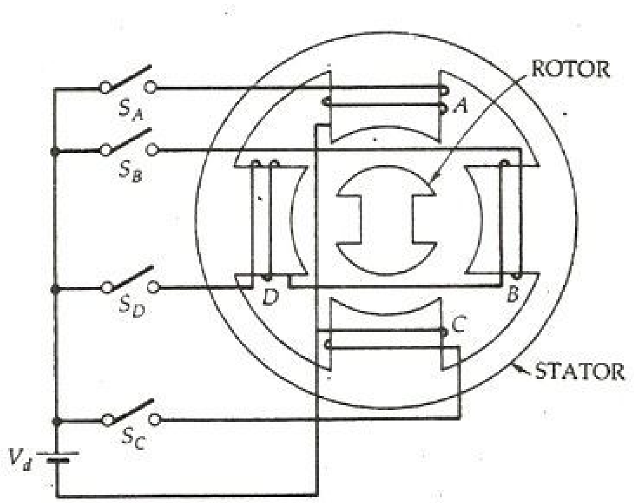


2. **Software** : Unfortunately this phase is a bit longer and more complex and will require more than one step.
    a. Compile your C code with your "Altera Soc EDS command shell" (go to your file and type make) and give it the name of your application. It is now fully created. The goal is now to put it on the board.
    b. It is then necessary to go through a step of using PuTTy. To do this, connect to the card through the serial port, set the baudrate in PuTTy and take control.
    c. Take the role of root (by entering the password) and use this to retrieve the IP address of the card (having been connected by ethernet) on the network.
    d. You will then be able to copy this compiled code to the card using the IP address you have retrieved by placing yourself in the project file from the PC.
    e. Once this is done you can launch the program thanks to "./program name" on the card.
    f. Observe your engine and/or terminal (if there is a print).

# 9. Explanation of how it works




This is a diagram of the overall operation of our system. The microcontroller is the FPGA board. The red board is only there to provide power to our control signal, so the signal at its input is of the same form as the signal at its output. The signals a,b,c,d are used to power the 4 coils that make up our stepper motor.




This last diagram shows the 4 coils fed by the 4 signals a,b,c,d.

We also checked that the signals sent from the FPGA board to the motor are correct. To do this, we used an oscilloscope to observe these signals.

# 10. Conclusion

At the end of this project, we manage to realize a Hardware/Software platform whose goal is to control a stepper motor and make it turn for a given angle. Because a video is better than long explanations, please take a look at the video link in the Read_me file of the github to see the results. We hope that this tutorial will help anyone who wants to learn more about hardware/software programming.