Laboratory 1 report

Cédric Hölzl Antoine Brunner

March 2020

1 PWM - servomoteurs

Dans la première partie du labo, on a du créer un générateur PWM. Nous avons suivi les conseils et d'abord dessiné un schéma block pour avoir une vue d'ensemble de l'implémentation, comme montré dans la figure 1.

Ce n'est qu'un schéma, donc certains détails d'implémentation ne sont pas présents. Notamment, nous n'avons pas dessiné la logique qui permet d'ignorer les entrées invalides (quand le duty cycle est plus grand que la période par exemple). Nous n'avons pas non plus dessiné les interfaces avec l'extérieur du composant.

Le registre principal est le compteur et permet de savoir à quel moment du signal on se trouve. Pour chacun des deux paramètres principaux du PWM (duty cycle et période) nous avons deux registres. Un registre stocke la valeur actuelle du paramètre, et l'autre stocke la prochaine valeur, celle entrée par l'utilisateur du composant.

La valeur actuelle des paramètres change soit quand l'utilisateur active le PWM, soit quand le compteur atteint la période. La sortie du PWM n'est pas montrée dans le diagramme car elle peut être calculée de façon combinatoire à partir de tous les signaux déjà présents.

Finalement, nous avons écrit les parties de code C pour écrire dans les registres nécessaire pour configurer/démarer/arreter les moteurs (le signal PWM).

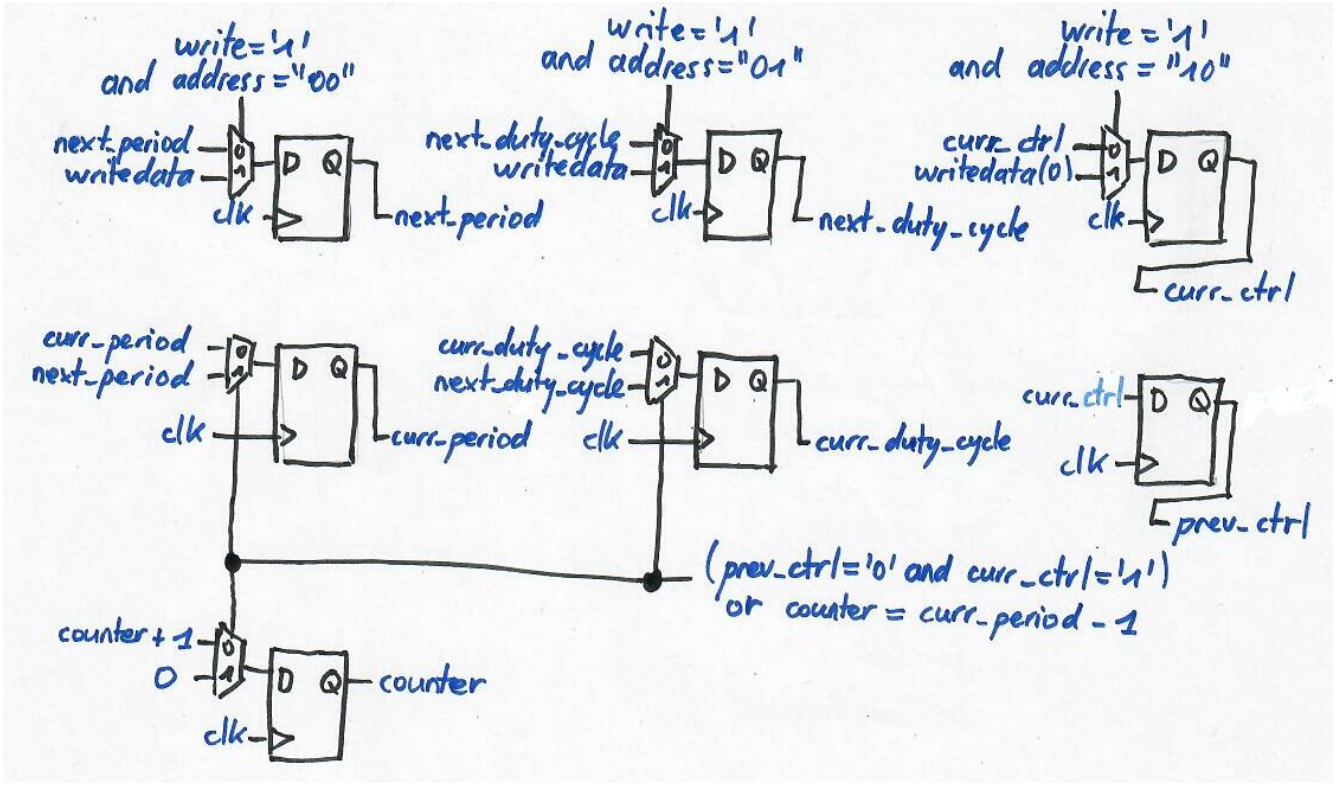

Figure 1: Block diagram of the PWM circuit

2 SPI - joystick

Dans cette deuxième partie, nous avons designé un composant permettant d'interfacer un convertisseur ADC avec le protocole SPI. Nous avons denouveau commencé par dessiner un schéma de FSM, voir figure 2.

Notez que le schéma de la FSM n'est pas complet, mais n'est qu'une esquisse qui nous a permit l'implémentation du VHDL. Il manque notamment la logique qui permet de charger *channel* depuis le bus lorsqu'un transfer est initié. Il manque également la logique qui permet de lire les données qui sont reçues sur le bus, lorsqu'on est dans l'état RCV_D.

Nous avons ensuite implémenté cette FSM en VHDL ainsi que sa logique de read/write . Nous avons décidé également d'utiliser un unique etat pour recevoir (RCV_D) et envoyer (SND_D) les données. On utilise un compteur indépendant pour suivre le progrès de l'envoi/reception.

Finalement, nous avons écrit le code C nécéssaire pour connecter la position des joystick au signal PWM.

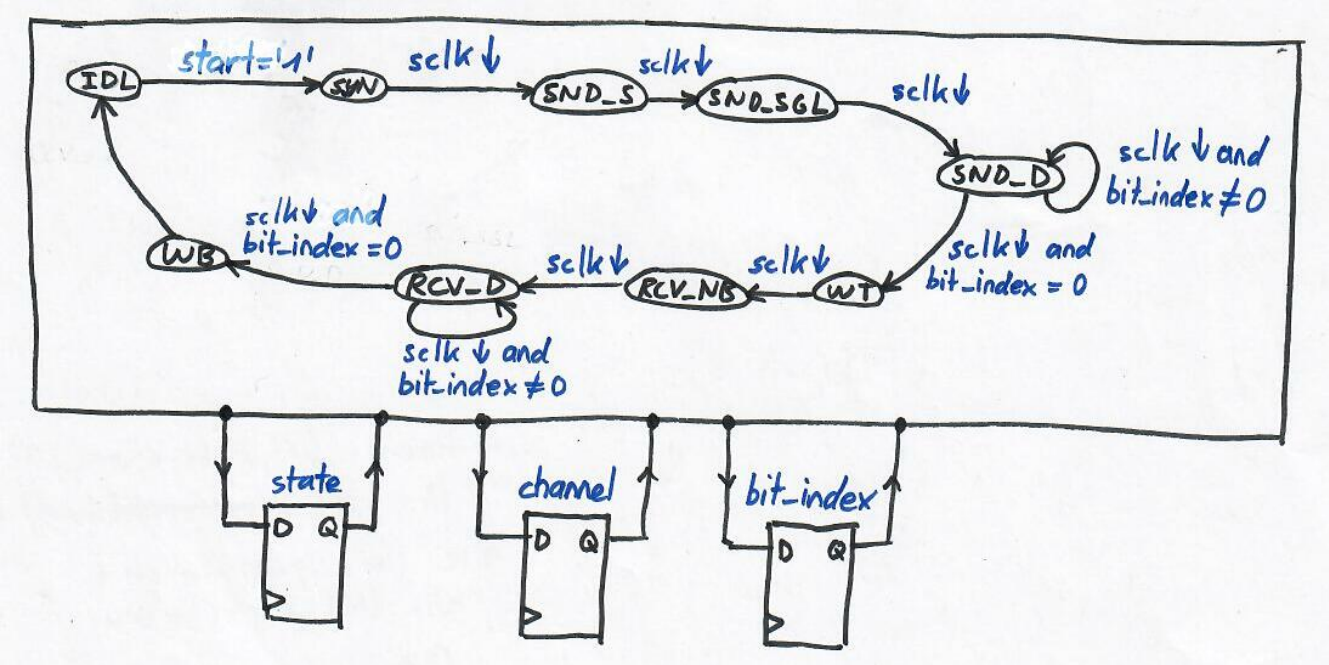

Figure 2: FSM for the SPI protocol

3 Résultats avec l'analyseur logique

Pour vérifier que notre implémentation fonctionne (bien que le fait que la caméra tourne est déjà bon signe), nous avons branché l'analyseur logique sur la sortie du PWM et capturé les signaux pendant que l'on tenait le joystick en bas, en haut, à gauche et à droite.

Les résultats de l'analyseur logique sont présenté dans la figure 3. Les données attendues étaient une période de 25 ms. Pour le mouvement vertical, le duty cycle devait varier entre 0.95 ms et 2.15 ms. Pour le mouvement horizontal, entre 1 ms et 2 ms.

Comme on peut le voir dans la figure 3, les valeur réelles sont très proches de celles attendues, ce qui montre que notre implémentation se comporte comme elle le devrait dans cette situation.

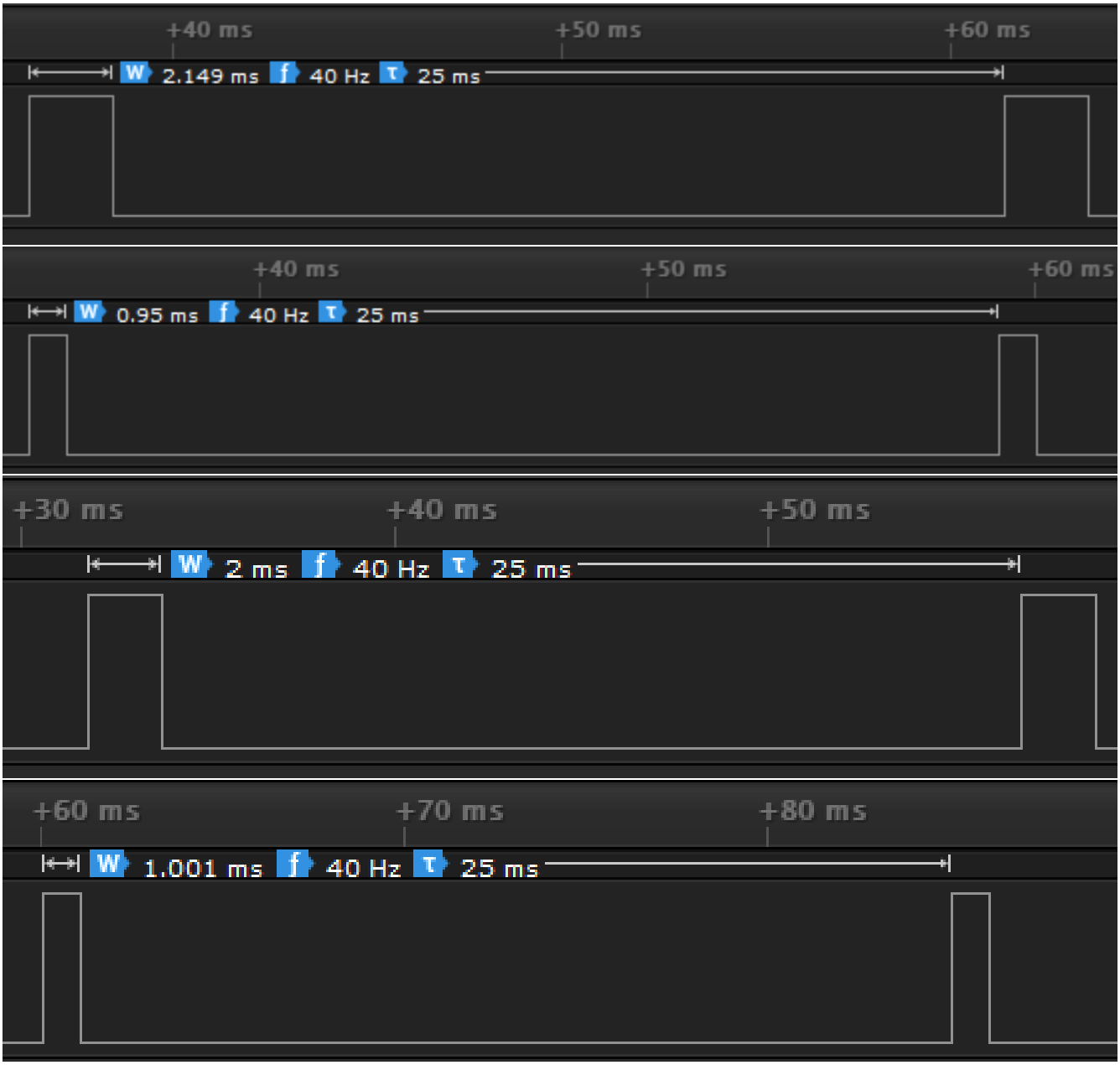

Figure 3: From top to bottom: PWM output when the joystick is at the bottom, top, left and right

4 Appendix: code

Voici le code décrit dans les sections précédentes, si vous préférez le lire depuis le pdf. Afin que le rapport ne soit pas trop long, nous avons mis uniquement le code que nous avons du compléter.

```
#define MICROSEC_TO_CLK(time, freq) ((time)*((freq)/1000000))

void pwm_configure(pwm_dev *dev, uint32_t duty_cycle, uint32_t period, uint32_t
↪  module_frequency) {
        IOWR_32DIRECT(dev->base, PWM_PERIOD_OFST, MICROSEC_TO_CLK(period,
        ↪  module_frequency));
        IOWR_32DIRECT(dev->base, PWM_DUTY_CYCLE_OFST, MICROSEC_TO_CLK(duty_cycle,
        ↪  module_frequency));
}

void pwm_start(pwm_dev *dev) {
        IOWR_32DIRECT(dev->base, PWM_CTRL_OFST, PWM_CTRL_START_MASK);
}

void pwm_stop(pwm_dev *dev) {
        IOWR_32DIRECT(dev->base, PWM_CTRL_OFST, PWM_CTRL_STOP_MASK);
}
```

Listing 1: Le code écrit dans pwm.c

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.pwm_constants.all;

entity pwm is
    port(
        -- Avalon Clock interface
        clk : in std_logic;

        -- Avalon Reset interface
        reset : in std_logic;

        -- Avalon-MM Slave interface
        address   : in  std_logic_vector(1 downto 0);
        read      : in  std_logic;
        write     : in  std_logic;
        readdata  : out std_logic_vector(31 downto 0);
        writedata : in  std_logic_vector(31 downto 0);

        -- Avalon Conduit interface
```

```
        pwm_out : out std_logic
    );
end pwm;

architecture rtl of pwm is

    -- The period of the current and next PWM cycle
    signal reg_next_period : unsigned(writedata'range) :=
    ↪  to_unsigned(DEFAULT_PERIOD, writedata'length);
    signal reg_current_period : unsigned(writedata'range) :=
    ↪  to_unsigned(DEFAULT_PERIOD, writedata'length);

    -- The duty cycle of the current and next PWM cycle
    signal reg_next_dutycycle : unsigned(writedata'range) :=
    ↪  to_unsigned(DEFAULT_DUTY_CYCLE, writedata'length);
    signal reg_current_dutycycle : unsigned(writedata'range) :=
    ↪  to_unsigned(DEFAULT_DUTY_CYCLE, writedata'length);

    -- The status of the current and next PWM cycle
    signal reg_prev_ctrl : std_logic := '0';
    signal reg_current_ctrl : std_logic := '0';

    -- The internal counter of the PWN
    signal reg_counter : unsigned(writedata'range) := to_unsigned(0,
    ↪  writedata'length);

begin

        --Avalon-MM slave write
        process(clk, reset)
        begin
            if reset = '1' then
                reg_next_period <= to_unsigned(DEFAULT_PERIOD, writedata'length);
                reg_next_dutycycle <= to_unsigned(DEFAULT_DUTY_CYCLE,
                ↪  writedata'length);
                reg_current_ctrl <= '0';
            elsif rising_edge(clk) then
                if write = '1' then
                    case address is
                        when REG_PERIOD_OFST =>
                            if unsigned(writedata) >= to_unsigned(2,
                            ↪  writedata'length) then
                                reg_next_period <= unsigned(writedata);
                            end if;
                        when REG_DUTY_CYCLE_OFST =>
                            if (unsigned(writedata) >= to_unsigned(1,
                            ↪  writedata'length)) and
```

```
                    (unsigned(writedata) <= reg_next_period) then
                     reg_next_dutycycle <= unsigned(writedata);
                end if;
            when REG_CTRL_OFST =>
                reg_current_ctrl <= writedata(0);
            when others => null;
        end case;
    end if;
end if;
end process;


--Avalon-MM slave read
process(clk, reset)
begin
    if rising_edge(clk) then
        if read = '1' then
            case address is
                when REG_PERIOD_OFST =>
                    readdata <= std_logic_vector(reg_current_period);
                when REG_DUTY_CYCLE_OFST =>
                    readdata <= std_logic_vector(reg_current_dutycycle);
                when others =>
                    readdata <= (others => '0');
            end case;
        end if;
    end if;
end process;

-- Internal synchronous logic
process(clk, reset)
begin
    if reset = '1' then
        reg_counter <= to_unsigned(0, writedata'length);
        reg_prev_ctrl <= '0';
    elsif rising_edge(clk) then
        if ((reg_prev_ctrl = '0') and (reg_current_ctrl = '1')) or
           (reg_counter = reg_current_period - 1) then
            reg_current_period <= reg_next_period;
            reg_current_dutycycle <= reg_next_dutycycle;
            reg_counter <= to_unsigned(0, writedata'length);
        elsif (reg_current_ctrl = '1') then
            reg_counter <= reg_counter + 1;
        end if;
        reg_prev_ctrl <= reg_current_ctrl;
    end if;
end process;
```

```
        -- Avalon Conduit interface
        process(clk, reset)
        begin
            if rising_edge(clk) then

                if (reg_counter < reg_current_dutycycle) and (reg_current_ctrl =
                ↪  '1') then
                    pwm_out <= '1';
                else
                    pwm_out <= '0';
                end if;
            end if;
        end process;

end architecture rtl;
```

Listing 2: Le code écrit dans pwm.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mcp3204_spi is
    port(
        -- 50 MHz
        clk        : in  std_logic;
        reset      : in  std_logic;
        busy       : out std_logic;
        start      : in  std_logic;
        channel    : in  std_logic_vector(1 downto 0);
        data_valid : out std_logic;
        data       : out std_logic_vector(11 downto 0);

        -- 1 MHz
        SCLK : out std_logic;
        CS_N : out std_logic;
        MOSI : out std_logic;
        MISO : in  std_logic
    );
end mcp3204_spi;

architecture rtl of mcp3204_spi is
    -- The signals that drive the clock divider
    signal reg_clk_divider_counter : unsigned(4 downto 0) := (others => '0');  --
    ↪  need to be able to count until 24
```

```
    signal reg_spi_en              : std_logic            := '0';  -- pulses
    ↪  every 0.5 MHz
    signal reg_rising_edge_sclk    : std_logic            := '0';
    signal reg_falling_edge_sclk   : std_logic            := '0';
    signal reg_sclk : std_logic := '0';

    -- The state related to the FSM
    type state_type is (IDL, SYN, SND_S, SND_SGL, SND_D, WT, RCV_NB, RCV_D, WB);
    signal reg_state, next_state : state_type := IDL;
    signal reg_bit_idx : unsigned(3 downto 0) := (others => '0');
    signal reg_channel : unsigned(1 downto 0);

    -- The register that holds the transmitted data
    signal reg_data : unsigned(11 downto 0) := (others => '0');

begin
    clk_divider_generation : process(clk, reset)
    begin
        if reset = '1' then
            reg_clk_divider_counter <= (others => '0');
        elsif rising_edge(clk) then
            reg_clk_divider_counter <= reg_clk_divider_counter + 1;
            reg_spi_en              <= '0';
            reg_rising_edge_sclk    <= '0';
            reg_falling_edge_sclk   <= '0';

            if reg_clk_divider_counter = 24 then
                reg_clk_divider_counter <= (others => '0');
                reg_spi_en              <= '1';

                if reg_sclk = '0' then
                    reg_rising_edge_sclk <= '1';
                elsif reg_sclk = '1' then
                    reg_falling_edge_sclk <= '1';
                end if;
            end if;
        end if;
    end process;

    SCLK_generation : process(clk, reset)
    begin
        if reset = '1' then
            reg_sclk <= '0';
        elsif rising_edge(clk) then
            if reg_spi_en = '1' then
                reg_sclk <= not reg_sclk;
            end if;
```

```
    end if;
end process;

STATE_LOGIC : process(clk, reset)
begin
    if reset = '1' then
        reg_state <= IDL;
        reg_bit_idx <= (others => '0');
    elsif rising_edge(clk) then
        reg_state <= next_state;

        case reg_state is
            when IDL =>
                if next_state = SYN then
                    reg_channel <= unsigned(channel);
                end if;
            when SND_SGL =>
                if next_state = SND_D then
                    reg_bit_idx <= to_unsigned(2, reg_bit_idx'length);
                end if;
            when RCV_NB =>
                if next_state = RCV_D then
                    reg_bit_idx <= to_unsigned(11, reg_bit_idx'length);
                end if;
            when SND_D | RCV_D =>
                if reg_falling_edge_sclk = '1' then
                    reg_bit_idx <= reg_bit_idx - 1;
                end if;
            when others =>
                null;
        end case;
    end if;
end process;

-- This is the combinatory logic to compute the next state
next_state <=
    SYN     when reg_state = IDL and start = '1' else
    SND_S   when reg_state = SYN and reg_falling_edge_sclk = '1' else
    SND_SGL when reg_state = SND_S and reg_falling_edge_sclk = '1' else
    SND_D   when reg_state = SND_SGL and reg_falling_edge_sclk = '1' else
    WT      when reg_state = SND_D and reg_falling_edge_sclk = '1' and
    ↪  reg_bit_idx = 0 else
    RCV_NB  when reg_state = WT and reg_falling_edge_sclk = '1' else
    RCV_D   when reg_state = RCV_NB and reg_falling_edge_sclk = '1' else
    WB      when reg_state = RCV_D and reg_falling_edge_sclk = '1' and
    ↪  reg_bit_idx = 0 else
    IDL     when reg_state = WB else
```

```
        reg_state;

    -- This process reads the bits sent from the ADC
    ADC_READ : process(clk, reset)
    begin
        if reset = '1' then
            reg_data <= (others => '0');
        elsif rising_edge(clk) then
            if reg_state = RCV_D and reg_rising_edge_sclk = '1' then
                reg_data(to_integer(reg_bit_idx)) <= MISO;
            end if;
        end if;
    end process;

    -- This is the combinatory logic to the ADC converter
    SCLK <= reg_sclk;
    CS_N <= '1' when reg_state = IDL or reg_state = SYN or reg_state = WB else
    ↪  '0';
    MOSI <=
        '1' when reg_state = SND_S or reg_state = SND_SGL else
        '0' when reg_state = SND_D and reg_bit_idx = 2 else
        reg_channel(to_integer(reg_bit_idx)) when reg_state = SND_D else
        '0';

    -- This is the combinatory logic to the SPI manager
    busy <= '0' when reg_state = IDL else
            '1';
    data_valid <= '1' when reg_state = WB else
                  '0';
    data <= std_logic_vector(reg_data) when reg_state = WB else
            (others => '0');

end architecture rtl;
```

Listing 3: Le code écrit dans mcp3204$_spi.vhd$

```
uint32_t mcp3204_read(mcp3204_dev *dev, uint32_t channel) {
    return channel < 4 ? IORD_32DIRECT(dev->base, channel * 4) : 0;
}
```

Listing 4: Le code écrit dans mcp3204.c

```
uint32_t joysticks_read_left_vertical(joysticks_dev *dev) {
        return JOYSTICKS_MAX_VALUE - mcp3204_read(&dev->mcp3204,LV_CHANNEL);
}
```

```c
uint32_t joysticks_read_left_horizontal(joysticks_dev *dev) {
        return mcp3204_read(&dev->mcp3204,LH_CHANNEL);
}

uint32_t joysticks_read_right_vertical(joysticks_dev *dev) {
        return JOYSTICKS_MAX_VALUE - mcp3204_read(&dev->mcp3204,RV_CHANNEL);
}

uint32_t joysticks_read_right_horizontal(joysticks_dev *dev) {
        return mcp3204_read(&dev->mcp3204,RH_CHANNEL);
}
```

Listing 5: Le code écrit dans joysticks.c

```c
uint32_t interpolate(uint32_t input,
                     uint32_t input_lower_bound,
                     uint32_t input_upper_bound,
                     uint32_t output_lower_bound,
                     uint32_t output_upper_bound) {
return (input - input_lower_bound) * (output_upper_bound - output_lower_bound) /
↪  (input_upper_bound - input_lower_bound) + output_lower_bound;
}
```

Listing 6: Le code écrit dans app.c