Practice 7. Design of a PPM modulator with VHDL

7.1. Objectives

To learn to design a pulse position modulation (PPM) system by using VHDL and its implementation in an FPGA. To learn to use different tools for the simulation and implementation of logic circuits using VHDL.

7.2. Expansion connectors of the Spartan-3A/3AN Starter Kit Board

The Spartan-3A/3AN Starter Kit Board offers a variety of expansion connectors for the communication with other boards or simply to the visualization or application of digital signals. The most versatile is the J17, which contains 100 pins, where an important percentage of them are accessible for I/O connection with the FPGA. Other useful connectors are the differential “transmit” and “receive” headers, which work at high data rates. In the next figures are shown the pinout and FPGA connections for connectors J17, J2 and J15.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>FPGA Pin</th>
<th>Shared</th>
<th>FX2 Connector</th>
<th>FPGA Pin</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply to FPGA I/O Banks 0, 1, 2</td>
<td>VCCO_012</td>
<td>J34</td>
<td>A (top)</td>
<td>SHIELD</td>
<td></td>
</tr>
<tr>
<td>TMS_B</td>
<td>VCCO_012</td>
<td></td>
<td>B (bottom)</td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>JTS_EL</td>
<td></td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>TDO_TX2</td>
<td></td>
<td></td>
<td></td>
<td>TDO1_XCVCC</td>
<td></td>
</tr>
<tr>
<td>FX2_J01</td>
<td>A13</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J02</td>
<td>B13</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J03</td>
<td>A14</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J04</td>
<td>B15</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J05</td>
<td>A15</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J06</td>
<td>A16</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J07</td>
<td>A17</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J08</td>
<td>B17</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J09</td>
<td>A18</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J10</td>
<td>C18</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J11</td>
<td>A19</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J12</td>
<td>B19</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J13</td>
<td>A20</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J14</td>
<td>B20</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J15</td>
<td>C19</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J16</td>
<td>D19</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J17</td>
<td>D18</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J18</td>
<td>E17</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J19</td>
<td>D20</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J20</td>
<td>D21</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J21</td>
<td>D22</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>FX2_J22</td>
<td>E22</td>
<td></td>
<td></td>
<td>GND</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1. Hirose 100-Pin FX2 Connector Pinout and FPGA Connections (J17)
Table 7.2. Hirose 100-Pin FX2 Connector Pinout and FPGA Connections (J17, continuation)

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>FPGA Pin</th>
<th>J17</th>
<th>A (top)</th>
<th>B (bottom)</th>
<th>FPGA Pin</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX2_IC23</td>
<td>F18</td>
<td>28</td>
<td>28</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC24</td>
<td>F19</td>
<td>29</td>
<td>29</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC25</td>
<td>E20</td>
<td>30</td>
<td>30</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC26</td>
<td>G20</td>
<td>31</td>
<td>31</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC27</td>
<td>G19</td>
<td>32</td>
<td>32</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC28</td>
<td>H19</td>
<td>33</td>
<td>33</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC29</td>
<td>H8</td>
<td>34</td>
<td>34</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC30</td>
<td>K18</td>
<td>35</td>
<td>35</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC31</td>
<td>K17</td>
<td>36</td>
<td>36</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC32</td>
<td>K19</td>
<td>37</td>
<td>37</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC33</td>
<td>L19</td>
<td>38</td>
<td>38</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC34</td>
<td>M20</td>
<td>39</td>
<td>39</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC35</td>
<td>M18</td>
<td>40</td>
<td>40</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC36</td>
<td>L18</td>
<td>41</td>
<td>41</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC37</td>
<td>L20</td>
<td>42</td>
<td>42</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC38</td>
<td>L20</td>
<td>43</td>
<td>43</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC39</td>
<td>M19</td>
<td>44</td>
<td>44</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_IC40</td>
<td>L17</td>
<td>45</td>
<td>45</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>FX2_CLKIN</td>
<td>L22</td>
<td>46</td>
<td>46</td>
<td></td>
<td>M22</td>
<td>FX2_CLKIN</td>
</tr>
<tr>
<td>FX2_CLKOUT</td>
<td>L22</td>
<td>47</td>
<td>47</td>
<td></td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>3.3V</td>
<td></td>
<td>48</td>
<td>48</td>
<td></td>
<td>L21</td>
<td>FX2_CLKIO</td>
</tr>
<tr>
<td>5.0V</td>
<td></td>
<td>50</td>
<td>50</td>
<td></td>
<td>SHIELD</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3. “Receive” header (J2)

Table 7.4. “Transmit” header (J15)

7.3. Practical development

This practice is intended to the design, by means of VHDL, of a PPM modulator which is constituted by a random data generator and the PPM modulator itself, as it is shown in figure 7.1. As we can see in the figure, the random data generator has two input (clk and reset) and two outputs (data and sync), whereas the PPM modulator has three inputs (clk, reset and data) and four outputs (clk_data, clk_ppm, output and par_data_out). Moreover, the output clk_data is fed
back to the clock input of the data generator, whereas the output of the data generator is connected to the input data of the PPM modulator.

![Figure 7.1. Blocks diagram of the PPM system](image)

**7.3.1. Data generator**

In first place, we are going to design the random data generator and check its correct behaviour when it is evaluated alone, previously to design the PPM modulator and demonstrate the performance of the joint system.

1. We are going to create a package called *constants* which contains all the constants that will be used during the development of the practice. In reality, for the data generator we only require from the constant $N$ and the function *and_vector*. The remaining constants and functions will be used for the block “PPM modulator”. The code in VHDL would be the next one:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package constants is
    constant N : positive := 4;
    constant M : positive := 4;
    constant L : positive := 2^M;
    function and_vector (vector : in std_logic_vector(0 to N-1)) return std_logic;
    function find_K (M,L : positive) return positive;
    constant K : positive := find_K(M,L);
end constants;

package body constants is
    function and_vector (vector : in std_logic_vector(0 to N-1)) return std_logic is
        variable all_one : std_logic_vector(0 to N-1) := (others=>'1');
        variable result : std_logic;
        begin
            if vector = all_one then
                result := '1';
            else
                result := '0';
            end if;
        return result;
    end and_vector;

    function find_K (M,L : positive) return positive is
        variable K : positive;
```
PRÁCTICE 7:
DESIGN OF A PPM MODULATOR
WITH VHDL

begin
  if ((M mod 2) = 1) then
    K := M*L;
  elsif ((M/2 mod 2) = 1) and (M /= 2) then
    K := (M/2)*L;
  else
    K := 2*L;
  end if;
  return K;
end function find_K;
end constant

Observe as, apart from the constant $N$ which is referred to the length (number of registers) of the data generator and the function and_vector, which determines the function and-logic of a data vector, are also defined the constant $M$, the constant $L$ as $2^M$, the function find_K and a constant $K$ which is obtained by using the function find_K previously defined. The function find_K basically determines the least common multiple of $M$ and $L$ in order to obtain a factor $K$ which will be used as reference for the generation of the clock signals of the data generator and the PPM pulse generator. Later we will insist on this again.

2. The block “data generator” requires from registers for its implementation, therefore we will have to design a component which performs in such a way. A possible code in VHDL could be the following:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity register is
  Port ( clk,preset,D : in  STD_LOGIC;
        Q : out  STD_LOGIC);
end register;

architecture Behavioral of register is
begin
  process(clk,preset)
  begin
    if preset='1' then
      Q <= '1';
    elsif clk'event and clk='1' then
      Q <= D;
    end if;
  end process;
end Behavioral;
```

Observe how the register responds to the changes in the input signals $clk$ (clock signal) and $preset$ (set signal which puts the register in logic state ‘1’ when this is activated to high level). Whenever a rising edge of the clock signal $clk$ occurs, the data signal at the input $D$ of the register is transferred to its output $Q$. 
3. In the figure 7.2 is shown the structure of the random data generator which is constituted by four registers, which yields a pseudo-random sequence with length $2^4 - 1 = 15$.

![Figure 7.2. Internal structure of the pseudo-random data generator](image)

Observe how the reset signal in the system is used for inducing a set (not a reset) in the registers, establishing the output of all of them to ‘1’. This is necessary because if we established all the registers to zero, the data generator would keep that state indefinitely and its output would always be zero. Therefore, we require from activating the registers setting them to the logic state ‘1’, thus inducing the beginning of a new pseudo-random sequence. We can check as the input of the first register is fed by means of the xor-operation of its own output and that of the last register. By using this simple operation is possible to obtain at the output of the last register a pseudo-random sequence which is repeated each 15 bits. On the other hand, an and gate has been used as control device of this repetition, since its output is ‘1’ whenever all the output of the registers are at high logic level, which only occurs once along the period of the pseudo-random sequence. Hence, this signal can be used as synchronism signal which indicates the beginning of a new repetition of the sequence, so the name that was assigned to it.

A possible code in VHDL which allows us to implement the pseudo-random data generator of figure 7.2 is the next one:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constants.ALL;

entity data_gen is
    generic (Nreg : positive := N);
    port (clk,reset : in  STD_LOGIC;
          data,sync : out  STD_LOGIC);
end data_gen;

architecture Behavioral of data_gen is

component register is
    port (clk,reset,D : in std_logic;
         Q : out std_logic);
end component register;
```
signal sig_xor : std_logic;
signal Q_int : std_logic_vector(0 to Nreg-1);
begin
  Data_generator: for I in 0 to Nreg-1 generate
    Reg00: if (I=0) generate
      Reg0: Register port map (clk,reset,sig_xor,Q_int(0));
      end generate;
    Regs: if I>0 generate
      Reg: Register port map (clk,reset,Q_int(I-1),Q_int(I));
      end generate;
    end generate;
data <= Q_int(Nreg-1);
sig_xor <= Q_int(0) xor Q_int(Nreg-1);
sync <= and_vector(Q_int);
end Behavioral;

We can see as we have defined a generic Nreg to be able to design data generators of any desired length, which we have set to the value N, initialized to four in the package constants. We can check that we have used the statement generate in order to instantiate the different registers which constitute the generator, whereas a if statement determines the connections of each register as a function of its location in the total structure. In the same way, the output of the last register is sent to the output data of the generator, whereas the xor-logic operation is carried out over the outputs of the first and last registers, and the synchronism output (sync) is obtained by means of an and-logic operation over the outputs of all the registers.

It is important to notice that only certain interconnections by using a xor-logic function are valid to obtain a pseudo-random sequence with a length of \(2^N - 1\) bits, where \(N\) is the total number of registers of the data generator. For this reason, in despite of having used a generic to define any length Nreg for the data generator, connecting the outputs of the first and last register to the feedback xor gate would only work, a priori, for data generators of 4 bits, as it is the situation in this example.

4. By using the ISE Simulator, check the correct performance of the data generator. Which length has the pseudo-random sequence? When, during the period of the data signal, is the signal sync at high logic level?

7.3.2. PPM modulator

Now, we are going to carry out the design of the PPM modulator itself. However, previously to it, we are going to consider the synchronism requirements of the system in order to obtain a good performance of this one. First of all, we have a data generator as input to the PPM modulator. On the other hand, we have a modulator which has to generate \(L\) chips during the same period in which the data generator supplies \(M\) bits to it, being \(L = 2^M\). Moreover, we have a clock signal of 50 MHz (E12) which is supplied by the development board. We are going to suppose a simple example as it could be 16-PPM, where \(L = 16\) chips (one of them at high level) are generated during the same time in which the data generator supplies \(M = 4\) bits. In this case, we
would need a clock for the data generator which has a rate 4 times slower than that of the PPM modulator. To that end, we could use the clock signal of 50 MHz for the PPM modulator and design a counter which generated a pulse every four edges of this basis signal, being the output of the counter used as clock signal for the data generator. Another possibility is to use two counters, one of them for the modulator and the other one for the generator, the latter counting from 0 to 1, whereas the second counts from 0 to 7, establishing both of them to logic level ‘1’ whenever their counts reach the value 0. In the next figure is schematized the previously said.

Observe how, in this way, every four edges of the clock signal of the PPM modulator, only one edge of the data clock takes place. In this simple case we have not had problems, but we are going to think of the case for \( M = 3 \) bits, then being \( L = 2^3 = 8 \) chips. In this second case, it is not so simple to generate all the clock signals from that of higher frequency, since eight is not divisible by three. To resolve this case, we would need searching a least common multiple. Given that 2, 3, 5, 7, 9, 11 and 13 are prime numbers and, in general, we will not have to design a PPM modulator of more than 8 data bits, we can apply as simple rule to determine the least common multiple the following: \( K = M \cdot L \). For \( M = 6 \) and \( M = 10 \), the least common multiple is easily calculated by means of \( K = (M/2) \cdot L \), whereas, for the remaining cases, \( L \) is divisible by \( M \), therefore we can take as criterion \( K = 2L \). The function \texttt{find\_K} that is integrated inside the package \texttt{constants} exactly applies the previous criteria in order to determine the factor \( K \).

1. Next is shown a possible code for the PPM modulator, where the initial part is centred in the generation of the clock signals:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constant_s.ALL;

entity ppm is
  generic (nchips : positive := L;
            nbits : positive := M;
            factor : positive := K);
  Port ( clk,reset,data : in  STD_LOGIC;
         clk_data,clk_ppm,par_data_out : out  STD_LOGIC;
         clk_data,clk_ppm,par_data_out : out std_logic_vector(nbits-1 downto 0));
end ppm;
```
architecture Behavioral of ppm is
signal par_data,serial_data : std_logic_vector(nbits-1 downto 0);
signal clk_ppm_int,clk_data_int,clk_par : std_logic;
component Register is
  Port ( clk,reset,D : in STD_LOGIC;
        Q : out STD_LOGIC);
end component register;
signal count_ppm : natural range 0 to nchips := 0;
begin
  Clocks_gen: process(clk,reset)
  variable count : natural range 0 to factor := 0;
  begin
    if reset = '1' then
      count := 0;
      clk_data_int <= '1';
      clk_ppm_int <= '1';
      clk_par <= '1';
    elsif clk'event and clk='1' then
      count := count + 1;
      if count = factor then
        count := 0;
        clk_ppm_int <= '1';
        clk_data_int <= '1';
        clk_par <= '1';
      else
        clk_par <= '0';
        if (count mod (factor/nchips)) = 0 then
          clk_ppm_int <= '1';
        else
          clk_ppm_int <= '0';
        end if;
      end if;
    else
      clk_par <= '0';
      if (count mod (factor/nbits)) = 0 then
        clk_data_int <= '1';
      else
        clk_data_int <= '0';
      end if;
    end if;
  end if;
end process Clocks_gen;
clk_data <= clk_data_int;
clk_ppm <= clk_ppm_int;
Parallel_register: for I in 0 to nbits-1 generate
  Reg0: if I=0 generate
    Reg_0: Register port map
      (clk_data_int,reset,dat,serial_data(0));
  end generate;
  RegM: if I>0 generate
    Reg_in: Register port map
      (clk_data_int,reset,serial_data(I-1),serial_data(I));
  end generate;
  Reg_par: Register port map
    (clk_par,reset,serial_data(I),par_data(I));
end generate;
ppm_gen: process(clk_ppm_int,reset)
begin
  if reset='1' then
    count_ppm <= 0;
  elsif clk_ppm_int'event and clk_ppm_int='1' then
    if count_ppm = nchips-1 then
      count_ppm <= 0;
    else
      count_ppm <= count_ppm + 1;
    end if;
  end if;
PRÁCTICE 7:
DESIGN OF A PPM MODULATOR
WITH VHDL

end if;
end process ppm_gen;

output <= '1' when
(std_logic_vector(conv_unsigned(count_ppm,nbits)) = par_data)
else '0';
par_data_out <= par_data;
end Behavioral;

The IEEE library called STD_LOGIC_UNSIGNED has been included to allow us to operate with the function conv_unsigned which converts an integer or natural number to a data type unsigned with the indicated number of bits. The process that is in charge of the generation of the clock signals uses a counter and a variety of if statements which control whether the variable of type natural count goes through certain values in order to activate the data clock signal (clk_data_int) and the clock signal of the PPM modulator (clk_ppm_int), according to the previously mentioned considerations. Moreover, every cycle of the variable count (which goes from 0 to K – 1) a clock signal for the parallel register is also activated, which induces the transference of the data introduced serially through the input data towards the output registers. In the figure is shown the structure of a serial to parallel register of 4 bits.

2. By using the ISE Simulator, check the correct performance of the clock generator for M = 2 and M = 3 bits.
3. Next is shown a possible which integrates all the previously designed blocks:

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constants.ALL;

entity system is
    Port (clk,reset : in STD_LOGIC;
          data, sync, ppm_out : out STD_LOGIC;
          );
end system;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constants.ALL;

entity system is
    Port (clk,reset : in STD_LOGIC;
          data, sync, ppm_out : out STD_LOGIC;
          );
end system;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constants.ALL;

entity system is
    Port (clk,reset : in STD_LOGIC;
          data, sync, ppm_out : out STD_LOGIC;
          );
end system;

end Behavioral;
par_data : out std_logic_vector(M-1 downto 0));
end system;

architecture Behavioral of system is
component data_gen is
  generic (Nreg : positive);
  port (clk,reset : in STD_LOGIC;
       data,sync : out STD_LOGIC);
end component data_gen;
component ppm is
  generic (nchips : positive;
           nbits : positive;
           factor : positive);
  port (clk,reset,dat : in STD_LOGIC;
        clk_data,clk_ppm,output : out STD_LOGIC;
        par_data_out : out std_logic_vector(M-1 downto 0));
end component ppm;
signal data_int,clk_data,clk_ppm : std_logic;
begins
Data_gen0 : data_gen generic map (N) port map
            (clk_data,reset,data_int,sync);
ppm_gen : ppm generic map (L,M,K) port map
           (clk,reset,data_int,clk_data,clk_ppm,ppm_out,par_data);
data <= data_int;
end Behavioral;

Observe (see the code for entity ppm) as the generation of the PPM signal is carried out by monitoring a counting signal (count_ppm) and the parallel data (par_data). Just in the instant when both quantities coincide, the PPM output signal is activated at high level, therefore the pulse is sent exactly in the position corresponding to the value saved in par_data. Check the correct performance of the total system by means for the ISE Simulator.

4. Try to implement in the FPGA a design where \(M = 3\) bits. What occurs in that case? How could this be solved?

5. Implement in the FPGA a design where \(M = 4\) bits. Observe the output signals by means of a logic analyzer. Next is shown a possible configuration for the I/O connections with the FPGA:

```
NET "reset" LOC = T15;
NET "clk" LOC = E12;
NET "data" LOC = A13;
NET "sync" LOC = B13;
NET "ppm_out" LOC = A14;
NET "par_data<0>" LOC = B15;
NET "par_data<1>" LOC = A15;
NET "par_data<2>" LOC = A16;
NET "par_data<3>" LOC = A17;
```