

PS/2 KEYBOARD CONTROLLER

DSD first year project

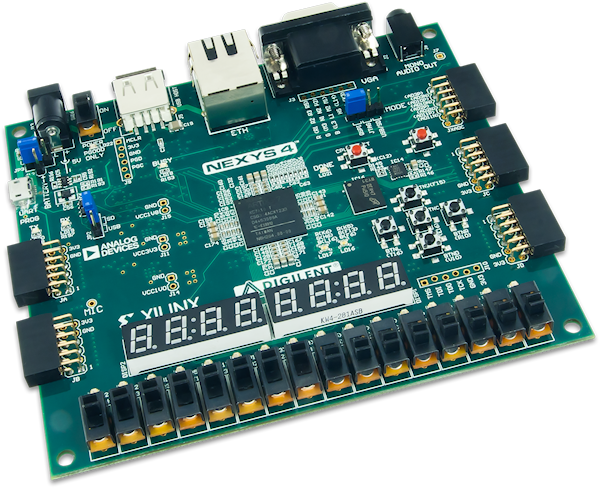




Table of Contents

[1](#_heading=h.gjdgxs) Specifications 2

[1.1](#_heading=h.30j0zll) Black Box 2

[1.2](#_heading=h.1fob9te) Control and Execution Unit 2

[1.2.1](#_heading=h.3znysh7) Mapping the inputs and outputs of the black box on the two components 3

[1.2.2](#_heading=h.2et92p0) Resources (breakdown of the Execution Unit) 3

[1.2.3](#_heading=h.tyjcwt) Block Diagram for first breakdown 18

[1.2.4](#_heading=h.3dy6vkm) State diagram of the Control Unit 18

[1.2.5](#_heading=h.1t3h5sf) Detailed diagram of the project 21

[2](#_heading=h.4d34og8) User manual 21

[3](#_heading=h.2s8eyo1) Technical justifications for the design 22

[3.1](#_heading=h.17dp8vu) The PS/2 keyboard protocol 22

[3.2](#_heading=h.3rdcrjn) How we interact with it. 22

[4](#_heading=h.26in1rg) Future developments 23

[5](#_heading=h.lnxbz9) References 23

PS/2 Keyboard Controller

# Specifications

Create a PS2 keyboard controller. It must read the pressed keys and show on the 8 7-segments slots the corresponding characters. The last 4 symbols must be shown and the control keys will have special functions (for example: “Enter” deletes the screen).

## Black Box

O imagine care conține diagramă, linie, captură de ecran

Descriere generată automat

*Figure 1 Black Box of the system*

## Control and Execution Unit

The system's black box must be further broken down in order to find implementable components. We will do a **top-down** breakdown of the problem until we get to known circuits, and then we will implement **bottom-up**.

The first breakdown of any system is one in which we will differentiate between the **control logic** in the system and the **system resources**. The control logic is represented by the Control Unit (CU) and the resources are represented by the Execution Unit (EU). Any algorithm can be broken down in this way (*the abstract representation of an algorithm is done through a flow-chart*).

### Mapping the inputs and outputs of the black box on the two components

*Figure 2 Mapping the inputs and outputs of the black box on the inputs and outputs of the units*

We can divide both inputs and outputs into 2 categories: **data** and **control**. This separation is essential at the beginning.

**Data inputs**: ps2\_ser, ps2\_clk

**Control inputs**: reset

**Data outputs**: anodes, cathodes

**Control outputs**: -

### Resources (breakdown of the Execution Unit)

In order to further establish the links between the CU and the EU, we must first identify **the resources on the basis of which we make decisions**. These resources can **generate signals to the control unit** and can be **controlled by the CU** via Enable or Reset signals.

Any decision-making information must come from a resource that generates that information and passes it on to UC.

Resources can be **simple circuits**, which can be implemented directly (counter, register, etc.) or **complex resources** (remainder algorithm, multiplication algorithm, etc.). These complex resources may appear in the first breakdown with black boxes to which we must establish inputs and outputs, but later they must be further broken down (usually also in CU and EU) until we reach known circuits.

**RESOURCES**

1. **For interacting with the keyboard**

Serial Reader

O imagine care conține text, captură de ecran, diagramă, Font

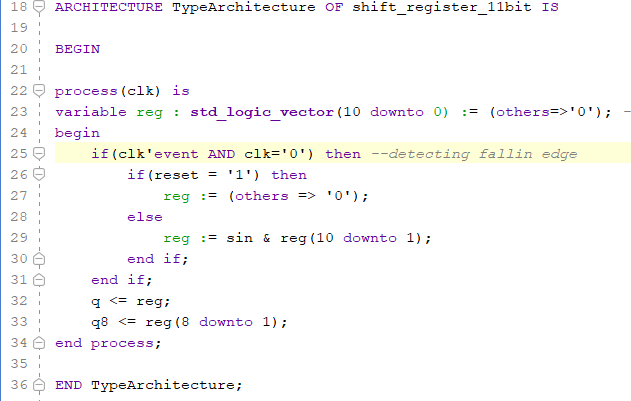
Descriere generată automat

*Figure 3. Black box for serial reader*

Serial reader’s purpose is to store the data bits sent by the keyboard in a serial manner via an 11-bit shift register and transmit them to the rest of the circuit on two buses for further processing. It is driven by the clock signal sent by the keyboard, after being filtered and synchronised with the clock signal of the board.

The *sin* port is connected to the data signal of the keyboard. The *reset* has the purpose of setting the register to zero, preparing the unit to read a new packet.

The 11-bit output bus is used to send data to the Validation Unit and to those that check for special situations, like *Enter* and beginning of a break or extended code, whereas the 8-bit one sends only the data bits for decoding the character to be displayed.



Modulo-11 counter

O imagine care conține text, diagramă, captură de ecran, Font

Descriere generată automat

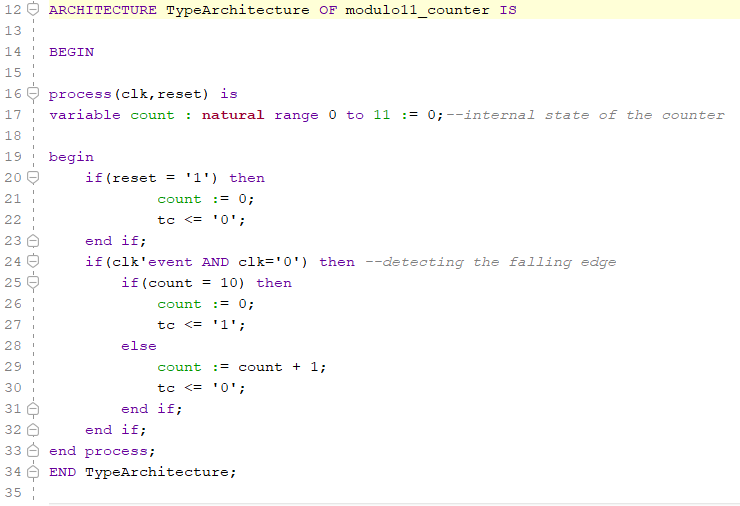
*Figure 4. Black box for modulo-11 counter*

The modulo-11 counter works alongside the Serial Reader, keeping track of the number of bits sent by the keyboard so far in the current transmission.

The connection and purpose of *clk* and *reset* is the same as for the Serial Reader.

*Tc* is a flag for the Control Unit to let it know when the transmission is complete.

The next 4 components check the outputs of the Serial Reader and send various flags to the Control Unit.



Validation Unit

O imagine care conține text, captură de ecran, Font, Grafică

Descriere generată automat

*Figure 5. Black box for validation unit*

Validation Unit is a CLC component which verifies that the code taken is correct or not, by checking the parity bit according to the other bits of the code, along with the stop bit and the start bit, which must always be 1, respectively 0.

E0 verification unit

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

*Figure 6. Black box for E0 verification unit*

This unit checks if the keyboard code contains E0 (hexadecimal), which represents the start of the extended code which we do not want to include in our application.

F0 verification unit

O imagine care conține text, captură de ecran, diagramă, Font

Descriere generată automat

*Figure 7. Black box for F0 verification unit*

F0 (hexadecimal) is the 8-bit number which signals the start of the **break code**. Break code is the set of binary values which we get when the user is going to release the key. We do not want to use the break code, this is just a validation that the user has released the key, therefore we need this unit to trigger a signal when f0 is read, so that we can ignore it and continue our reading.

Enter verification unit

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

*Figure 8. Black box for Enter verification unit*

This unit is useful for the user when they want to clear the slots of the 7-segments display. This unit checks if the code read is the one for “enter”: if yes, we are going to reset the display, if not, we want to print the current key.

Note: The architectures for the last three validation units were not included in this document due to their simplicity, consisting only of an if statement inside a process.

1. **Logic for display**

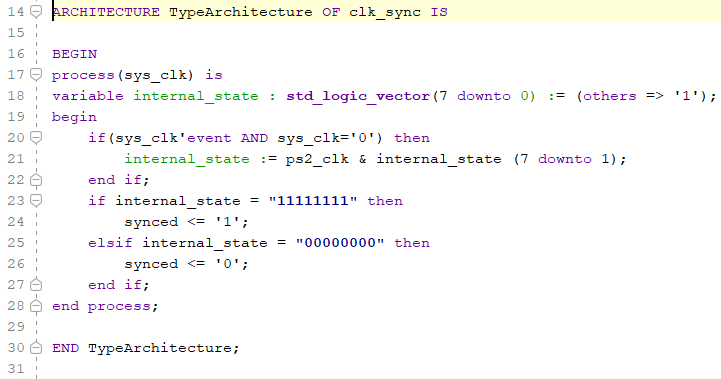
Clock synchroniser

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

*Figure 9.Black box for clock synchronizer*

Clock synchroniser generates the signal that drives the Serial Reader and the Modulo-11 counter. Its main purpose is to eliminate the noise on the clock signal generated by the keyboard, while having an output that is synchronous with the internal clock of the board. We implement this by changing the value of *synced* only when the *ps2\_clk* maintains its value for 8 consecutive clock cycles(80 ns).



Frequency divider

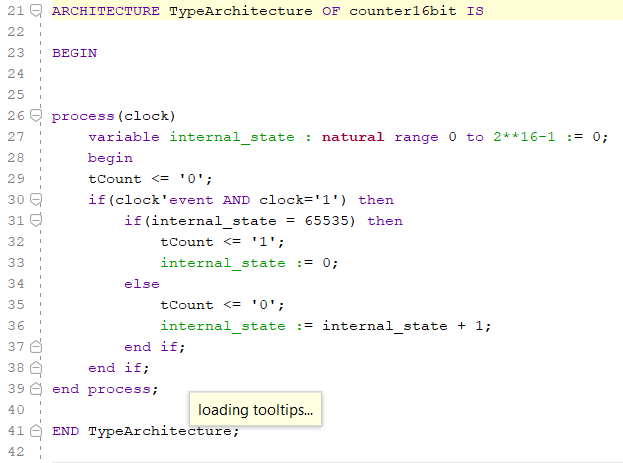
O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

*Figure 10. Black box for frequency divider*

The frequency divider transmits to the other components in this section a tCount signal which will be linked to the multiplexing logic units.

It sends a pulse on *tCount* every 65536( clock cycles(approx 1ms), without any duty cycle adjustment.



Memory for character codes

O imagine care conține text, captură de ecran, Font, siglă

Descriere generată automat

*Figure 11.Black box for memory for character codes*

This is the ROM memory which has the role of **converting** the code from the keyboard into the content which is going to be the input for the displaying buffer.

Architecture of component:

ARCHITECTURE TypeArchitecture OF decoder7sd IS

BEGIN

process(address)

begin

case address is

when x"16" => content <= b"01100000"; --1

when x"1E" => content <= b"11011010"; --2

when x"26" => content <= b"11110010"; --3

when x"25" => content <= b"01100110"; --4

when x"2E" => content <= b"10110110"; --5

when x"36" => content <= b"10111110"; --6

when x"3D" => content <= b"11100000"; --7

when x"3E" => content <= b"11111110"; --8

when x"46" => content <= b"11110110"; --9

when x"45" => content <= b"11111100"; --0

when x"15" => content <= b"11111101"; --q

when x"44" => content <= b"11111100"; --o

when x"1D" => content <= b"01111110"; --w

when x"24" => content <= b"10011110"; --e

when x"2D" => content <= b"00001010"; --small r

when x"2C" => content <= b"00011110"; --small t

when x"35" => content <= b"01110110"; --small y

when x"3C" => content <= b"01111100"; --U

when x"43" => content <= b"00001100"; --I

when x"4D" => content <= b"11001110"; --P

when x"1C" => content <= b"11101110"; --A

when x"1B" => content <= b"10110110"; --S

when x"23" => content <= b"11111100"; --D

when x"2B" => content <= b"10001110"; --F

when x"34" => content <= b"10111100"; --G

when x"33" => content <= b"01101110"; --H

when x"3B" => content <= b"01111000"; --J

when x"42" => content <= b"10101110"; --K

when x"4B" => content <= b"00011100"; --L

when x"1A" => content <= b"11011010"; --Z

when x"22" => content <= b"10010010"; --x

when x"21" => content <= b"10011100"; --C

when x"2A" => content <= b"01010100"; --V

when x"32" => content <= b"11111110"; --B

when x"31" => content <= b"00101010"; --small N

when x"3A" => content <= b"00101010"; --M

when x"41" => content <= b"00000001"; --,

when x"49" => content <= b"00000001"; --.

when others =>content<= b"00000000";

end case;

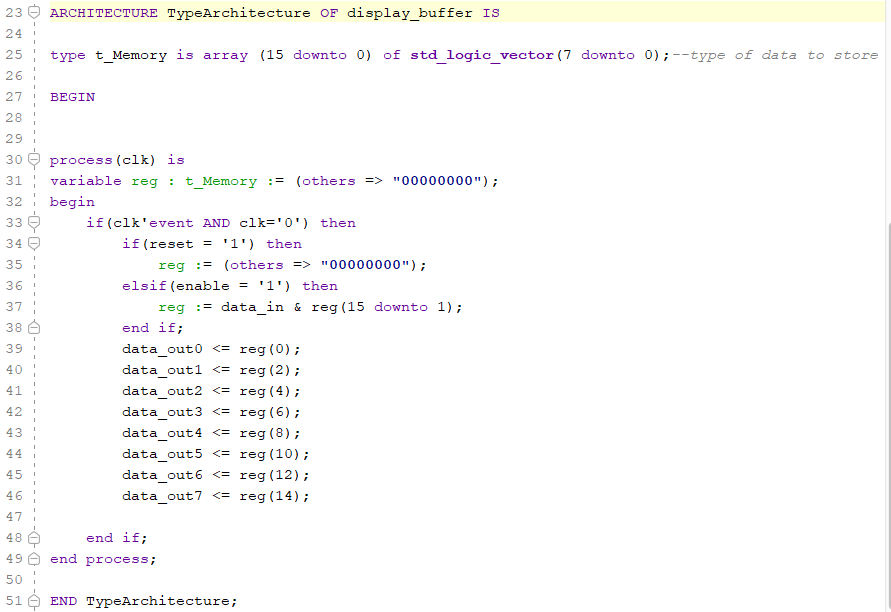
Display buffer

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

*Figure 12. Black box for display buffer*

This buffer is the component that uses shift operations in order to select the favourable position on the 7-segment slots. The outputs are the cathodes which will be chosen with the aid of the frequency divider, a counter on 3 bits and a multiplexer.



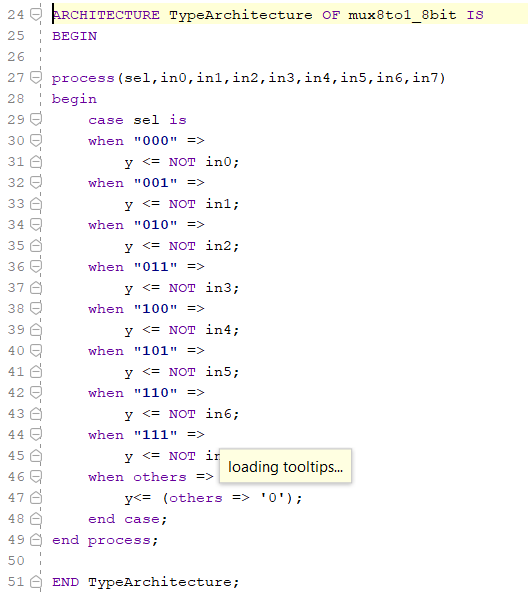
8 to 1 Multiplexer for driving the cathodes

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

*Figure 13.Black box for 8 to 1 Multiplexer 8-bit*

This is the multiplexer which will give as output the common cathode for the display. By using the frequency divider along with the counter on 3 bits, we generate the states which will be the selection for this mux.



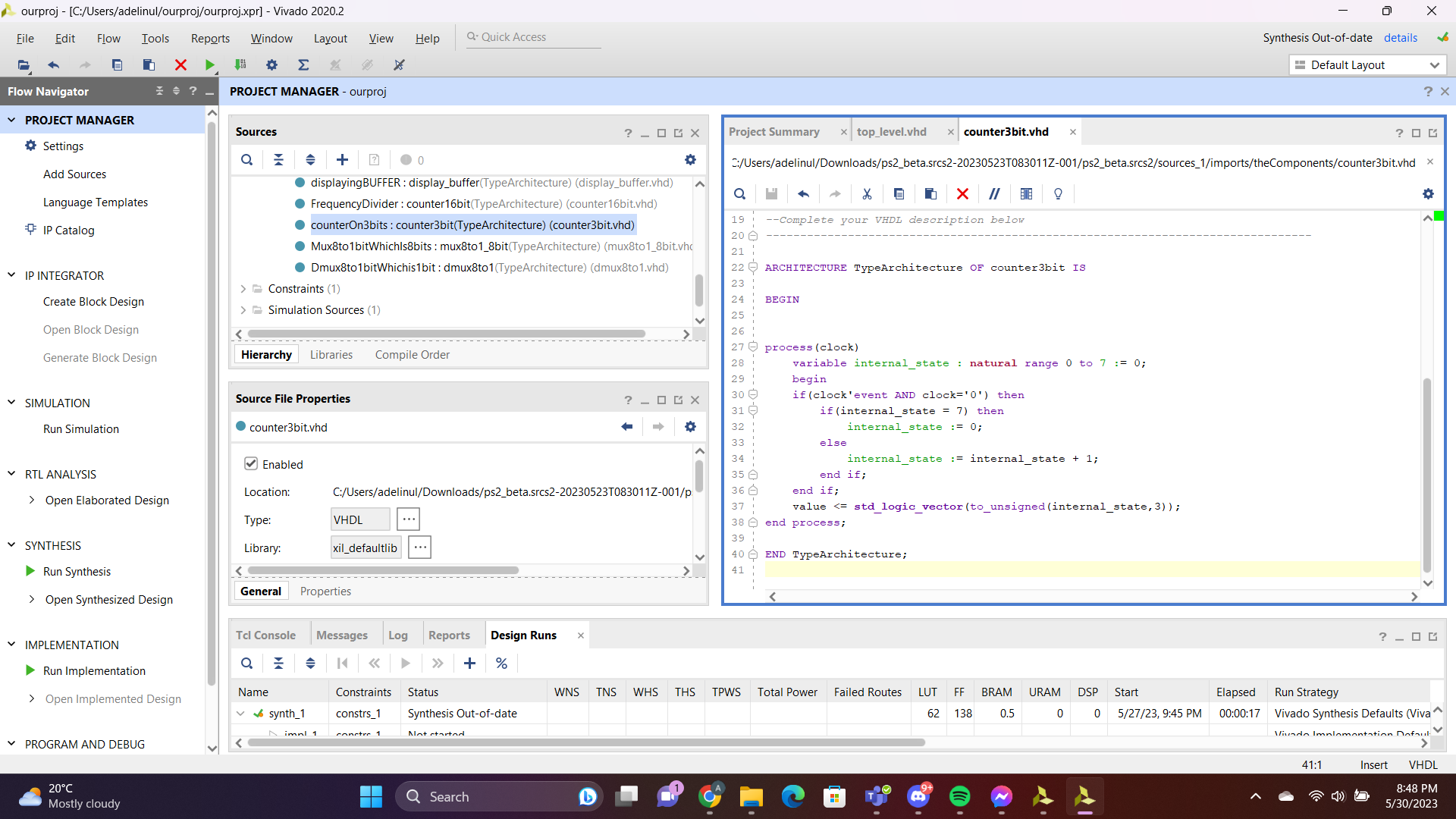
Counter on 3 bits

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

*Figure 14.Black box for 3-bit counter*

This counter has as input the terminal count from the frequency divider. The output values are the selection bits for both the multiplexer (used for the common cathode) and the demultiplexer (used for anodes).



1 to 8 Demultiplexer for driving the anodes

O imagine care conține text, captură de ecran, diagramă, număr

Descriere generată automat

*Figure 15.Black box for 1 to 8 Demultiplexer*

The input value for this component will always be 0 and the implicit values for the outputs are 1, since the anodes are active low on the FPGA board. According to our selection bits, the output at this binary position will be 0.

ARCHITECTURE TypeArchitecture OF dmux8to1 IS

BEGIN

process(val,sel)

begin

y0<='1';

y1<='1';

y2<='1';

y3<='1';

y4<='1';

y5<='1';

y6<='1';

y7<='1';

case sel is

when "000" =>

y0 <= val;

when "001" =>

y1 <= val;

when "010" =>

y2 <= val;

when "011" =>

y3 <= val;

when "100" =>

y4 <= val;

when "101" =>

y5 <= val;

when "110" =>

y6 <= val;

when "111" =>

y7 <= val;

when others =>

y0<='1';

y1<='1';

y2<='1';

y3<='1';

y4<='1';

y5<='1';

y6<='1';

y7<='1';

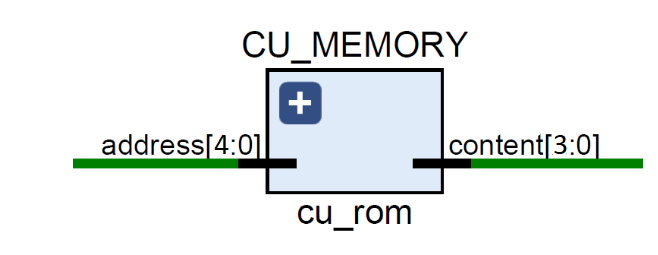
end case;

end process;

END TypeArchitecture;

1. **Control Unit**

Control Unit Memory

**

*Figure 16.Black box for control unit memory*

Probably the most important component of the Control Unit, the memory is the unit which converts the current state into the next state, passing this value to the 4-bit register.

Note: The code for this component is presented in the State Diagram sub-chapter.

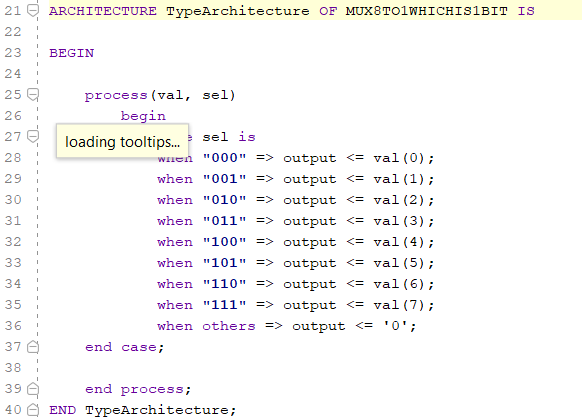
Control Unit Multiplexer

O imagine care conține text, captură de ecran, Font, diagramă

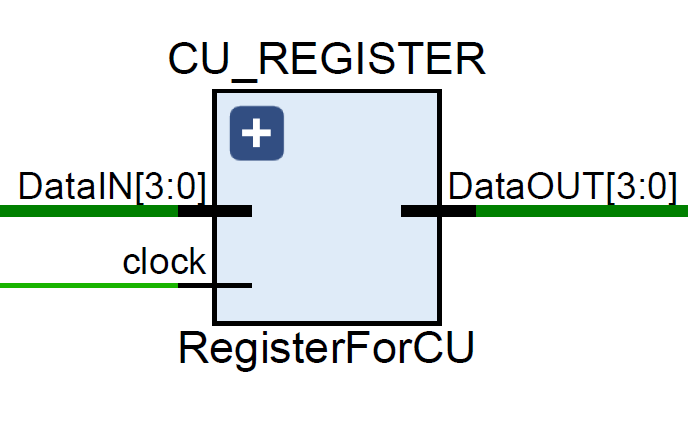
Descriere generată automat

*Figure 17.Black box for 8 to 1 Multiplexer 1-bit*

This is the multiplexer used for the control unit. We only need 3 bits from the current state to determine the output which is going to be the decision for the next state. If we do not have to make a decision in our diagram we do not care about the output, so we can have any value we want. This is linked to A4 ( the most significant bit of the address).

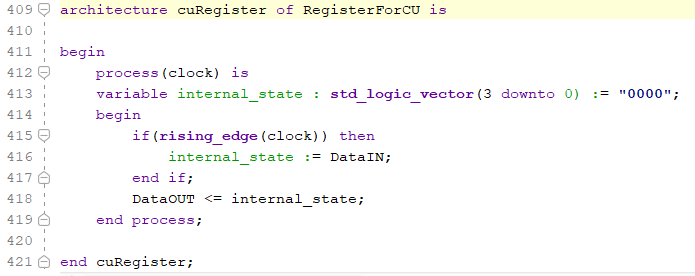


Control Unit Register



*Figure 18. Black box for Control Unit Register*

This storing register has the sole purpose of synchronising the outputs of the ROM memory with the future clock signal.



Outputs of Control Unit

O imagine care conține text, captură de ecran, diagramă, Font

Descriere generată automat

*Figure 19. Black box for Control Unit Outputs*

This unit generates the inputs for the multiplexer of our Control Unit Memory. When we are in a certain state, we need a signal such that we can know what operations we should perform. Clear will be active when we reach the is\_Enter state and en\_shift when we encounter the Printing state.

ARCHITECTURE TypeArchitecture OF CU\_Outputs IS

BEGIN

process(state) is

begin

if state = "110" then

en\_shift <= '1';

else

en\_shift <= '0';

end if;

if state = "111" then

clear <= '1';

else

clear <= '0';

end if;

end process;

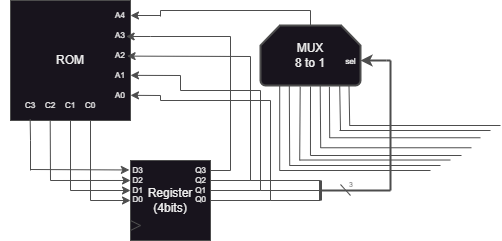
END TypeArchitecture;

### Block Diagram for first breakdown

*Figure 20. Block diagram for first breakdown*

### State diagram of the Control Unit

The control unit works using a ROM memory along with a storing register and a multiplexer. The idea behind creating the system which will generate the next states is simple: first of all, by creating the state diagram and encoding the states we came to the conclusion that 4 bits are necessary in order to make use of the states. However, 4 bits are not enough because we also need to make decisions (represented by diamonds in the state diagram). This is why the fifth bit for our address is vital. Its purpose will be to make decisions according to the current state. We need to know which value is going to be the input at this position of the address, therefore we are using a Multiplexer. Normally, we would have a 16 to 1 multiplexer, but by observing that we are making decisions only for 8 states, we can encode the states in such a way that we will use only the last 3 bits of the current address. That is why an 8 to 1 multiplexer is enough.



*Figure 21. Block diagram for the Control Unit*

VHDL code for creating next state:

ARCHITECTURE TypeArchitecture OF CU\_ROM IS

BEGIN

-- 0000 is the idle state

-- 0001 is the reading state

-- 0010 is the valid and not e0 state

-- 0011 is the wait2 state

-- 0100 is the f0 state

-- 0101 is the wait1 state

-- 0110 is the isEnter state

-- 0111 is the readAndIgnore state

-- 1000 is the printing state

-- 1001 is the clear 7SD state

-- 1010 is the wait2 begin

-- chosen this way because it does not need an input to make a decision

-- the MSB is given by the MUX

process (address)

begin

case address is

when "00000" => content <= "0001";

when "10000" => content <= "0000";

when "00001" => content <= "0001";

when "10001" => content <= "0010";

when "00010" => content <= "1010";

when "10010" => content <= "0100";

when "00011" => content <= "0011"; --wait2 state

when "10011" => content <= "0000";

when "00100" => content <= "0110";

when "10100" => content <= "0101";

when "00101" => content <= "0101";

when "10101" => content <= "0111";

when "00110" => content <= "1000";

when "10110" => content <= "1001";

when "00111" => content <= "0111";

when "10111" => content <= "1010";

when "01000" => content <= "1010";

when "11000" => content <= "1010";

when "01001" => content <= "1010";

when "11001" => content <= "1010";

when "01010" => content <= "0011"; --begin wait2

when "11010" => content <= "0011";

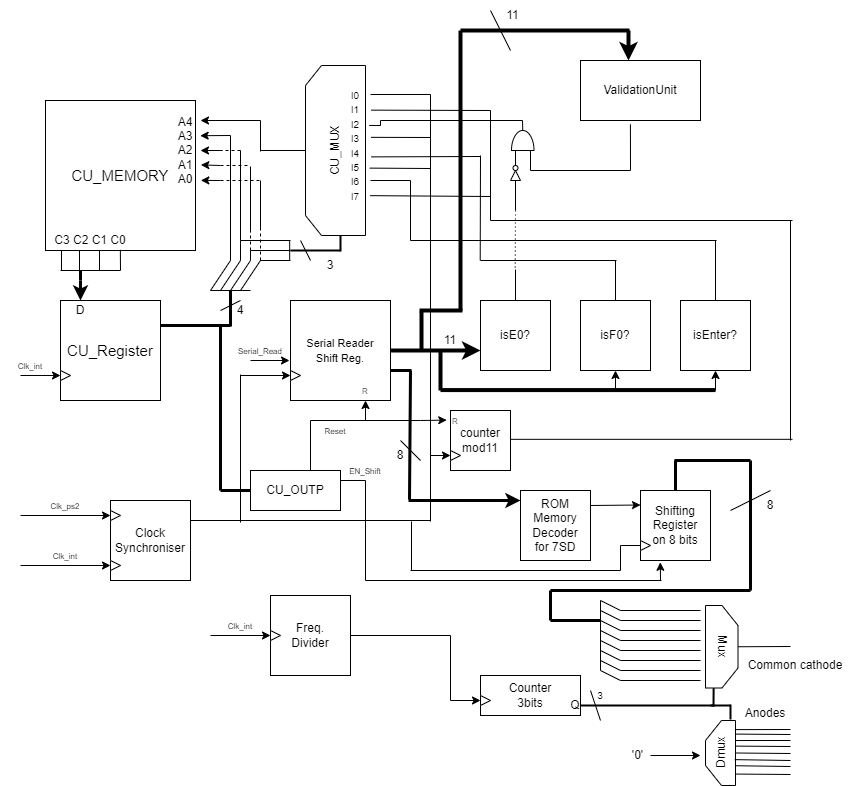
when others => content <= "0000";

end case;

end process;

END TypeArchitecture;

### Detailed diagram of the project



*Figure 22. Detailed schematic of the project*

# User manual

The user can interact with the controller by the means of the reset button or by typing on the actual keyboard.

Pressing the reset button will reset the internal state of the components implicated in the actual components that are involved in the reading of the serial signal sent by the keyboard.

Regarding the interaction via keyboard, there are several functionalities: printing a new character and clearing the display. When an **alphanumeric** key is pressed its symbol will appear on the rightmost seven segment display, other pre-existing symbols being shifted to the left. When ***Enter*** is pressed it will act as a new-line, clearing the display.

# Technical justifications for the design

## The PS/2 keyboard protocol

The PS/2 keyboard protocol is a way for the peripheral device to transmit serial information to the computer and vice versa. Due to its age, it is simpler than modern protocols, like USB, making the job of projecting devices that can work with it relatively easier.

For the purpose of the current project, we will ignore the physical implementation of the port and we will consider only the clock and data signals that are specified in the protocol, to which we will refer as PS2\_Clk and PS2\_Ser.

There are several states of communication which dictate if the information is sent by the keyboard or the host(computer), thus enabling a bi-directional protocol. We will consider only the state where the keyboards can send data, the so-called *Idle state*, where PS2\_Clk and PS2\_Ser are HIGH.

When the keyboard begins to send data the PS2\_Clk becomes 0. The data is to be read by the host from PS2\_Ser on the falling edge of the PS2\_Clk. The structure of a packet sent is:

* Start Bit, always 0.
* Data Bits,8, the least significant bit is sent first.
* Parity Bit, odd parity.
* Stop Bit, always 1.

In total there are 11 bits in a packet sent by the keyboard.

The PS2\_Clk can be between 10 and 16.66 kHz.

O imagine care conține Font, text, alb, captură de ecran

Descriere generată automat

*Figure 22.Device-to-host communication. The Data line changes state when Clock is high and that data is valid when Clock is low.*

## How we interact with it

We will read the bits sent by the keyboard in an 11-bit shift register and transmit the value to the other components in the circuit that need it. Here we encounter our first problem. We cannot use the clock of the PS2 signal directly to drive the clock of the shift register due to 2 big reasons: noise and the fact that it is not in sync with the rest of the circuit. One solution is a filter for the signal, similar in principle with a *debouncer*. Also, we will use the same synchronised clock signal to drive a counter for keeping track of the number of bits read so far.

Another important aspect to mention here is that any reading state of the system should last at least until the PS2\_Clk becomes HIGH i.e., the transmission is finished.

## Codes sent by the keyboard

There are 2 kinds of codes:

* Make Codes – sent when a key is being pressed.
* Break Codes – sent when a key is released.

They can be normal or extended codes, depending on the key. Extended codes have an additional packet sent containing “E0”.

# Future developments

4.1. Arrow key functionality

The user will have control over the position of the cursor, thus allowing the insertion of characters on arbitrary positions.

4.2. *Backspace* key functionality

The user will be able to delete the last character introduced. This can be implemented by right shifting the characters on the display.

4.3. Key bindings

The user can activate special operating modes by pressing more keys simultaneously. (For example: *Ctrl* and *C)*. One approach for this could be the addition of a flag register that keeps track of the special keys pressed and additional states that read those flags and check if any bidding is used.

# References

* http://www.burtonsys.com/ps2\_chapweske.htm
* https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes
* https://www.youtube.com/watch?v=BBtD4PCXqlE&ab\_channel=FPGATherapy
* https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual
* https://archive.org/details/ebooksclub.org\_\_FPGA\_Prototyping\_by\_VHDL\_Examples\_\_Xilinx\_Spartan\_3\_Version?q=vhdl+ps2+keyboard+to+7+segment+display