Peripheral Interfacing (SRAM, ROM, Keypad, 7-Segment)

Lewis Collum

March 19, 2019

Contents

1 Specification

Our goal is to design a SRAM based "programmable 8-bit counter of arbitrary sequence" that runs on a one Hz clock.

Note: we will refer to the six 7-segment displays on the DE2 board by the names: HEX5 to HEX0.

1.1 System initialization

The system will use a power-on reset to initialize the content of the SRAM by loading a default data sequence from a 1-Port ROM. The 256 x 16 bit ROM will be built using Quartus II's Megawizard plug-in Manager and a memory initialization file "sine.mif" (posted on Moodle). The counter should also use a reset button (KEY0) to reload the default sequence into the SRAM.

When the design is downloaded onto the Cyclone II FPGA, the 7-seg displays (HEX3 to HEX0) will show the contents of the SRAM. The address of the contents from the SRAM will be displayed on HEX5 & HEX4.

1.2 System Operation: "Run" and "Edit" Modes

The contents of the SRAM can be programmed manually using a keypad attached to the DE2 board via the 40 pin ribbon cable and a bread board. The "shift-key" on the keypad is used to toggle between (a) the "edit" mode and (b) the "run" mode. In "edit" mode, the LED named LEDG0, on the DE2 board, will be off. Conversely, in "run" mode, the LEDG0 will be on. When the system is in the "edit" mode, the "H" key is used to toggle between two modes: (1) SRAM "edit address" mode and (2) SRAM "edit data" mode.

1.2.1 Edit Mode: Keypad "Address Edit" Mode

In the Keypad "address edit" mode, when any of the numbers between 0-9 and A-F is pressed on the keypad, HEX4 will display its HEX value. If the second such key is pressed, the digit occupying HEX4 location will move to HEX5 location and the new value will be shown at HEX4. The 8-bit number represents an SRAM address.

1.2.2 Edit Mode: Keypad "Data Edit" Mode

In the Keypad "data edit" mode, when any of the numbers between 0-9 and A-F is pressed on the keypad the right most 7-segment display HEX0 will display its HEX value. If the second such key is pressed, the digit occupying HEX0 location will move to HEX1 location and the new value will be shown at HEX0. If the third and the forth key press with shift the display to the left until HEX3 through HEX0 will show a 16-bit binary number. This number will be used as data that will be loaded into the SRAM.

1.2.3 Sending the Data from the Keypad to the SRAM

When ready to enter the data into the SRAM (while in the system "edit" mode) the "L" key should be pressed (once). The four-digit HEX data displayed on HEX3 down to HEX0 will now be loaded in to the SRAM memory address location displayed on HEX5 to HEX4.

1.2.4 Run Mode

Subsequently, while still in the system "edit" mode, a shift-key press will toggle to the system "run" mode. The 7-segment displays (HEX3 to HEX0) will show the 16-bit content of the SRAM memory at address location 0X00 of which will be displayed on HEX5 to HEX4.

The Address Counter will begin counting only when the "H" key is pressed. When the "H" key is pressed, HEX5 to HEX4 will cycle through the address locations from 0X00 to 0XFF and the contents of the SRAM will be shown on HEX3 to HEX0. Pressing the "H" key in the system "run" mode will enable and disable the counter.

Furthermore, in the system "run" mode, the "L" key can be used to toggle the direction of the counter, between the forward and the backward directions.

2 Top-Level Design

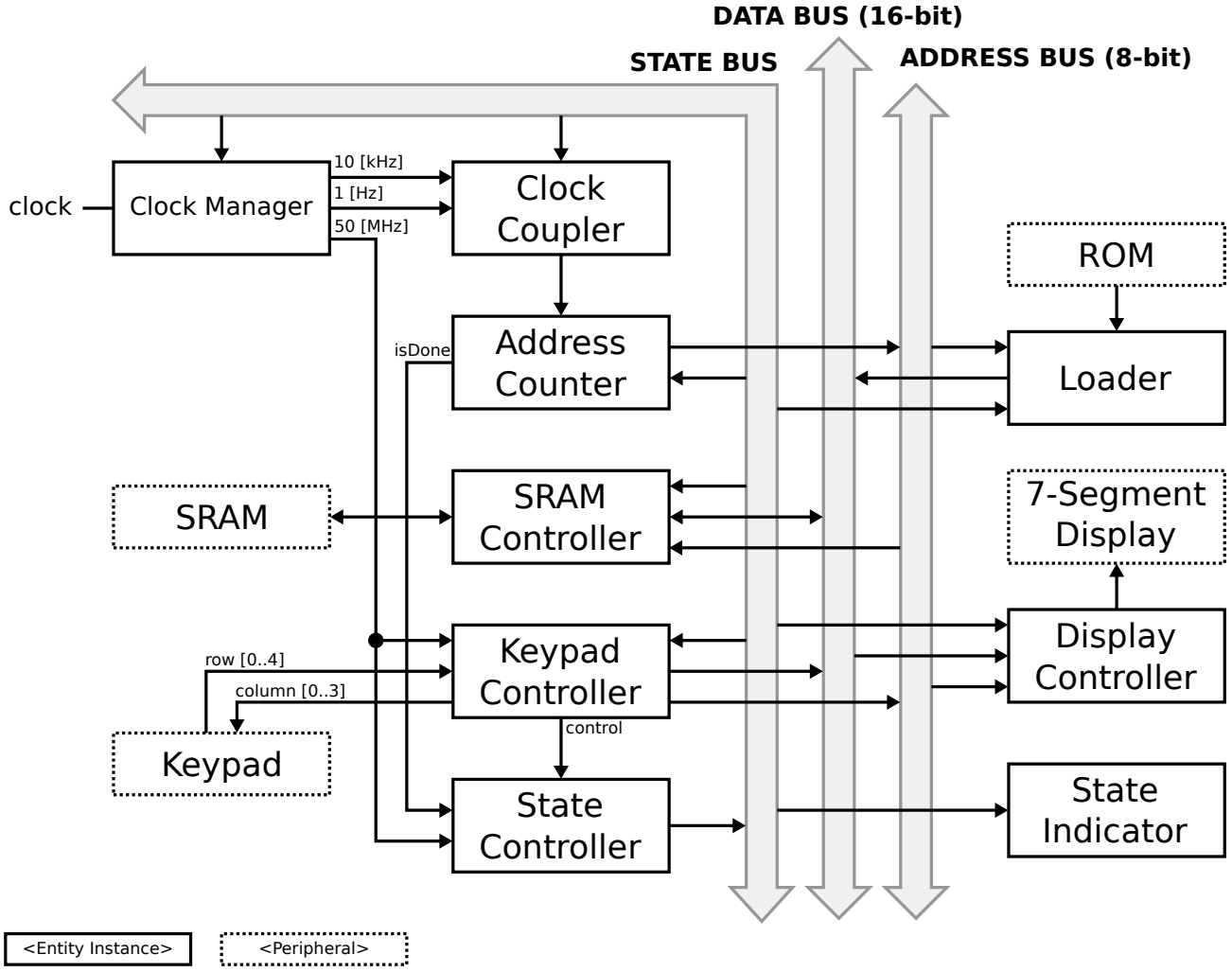

Figure 1: A state-controlled top-level. Each block represents an entity in VHDL. Most blocks consist of sub-entities (See Keypad Controller). The state-control design approach allows for a loosely coupled system.

3 A State Controlled System

3.1 States

Table 1: State Diagram Translation

In Code	In Text
`systemInitialize`	System Initialize
`loadRomToRam`	Load
`run`	Run
`edit`	Edit

1. **System Initialize**
   The system begins in the System Initialize state. In this state, all temporary memory is cleared. This includes RAM, Keypad Controller memory, and all other flip-flops. If the "reset" button is pressed the system also returns to this state.

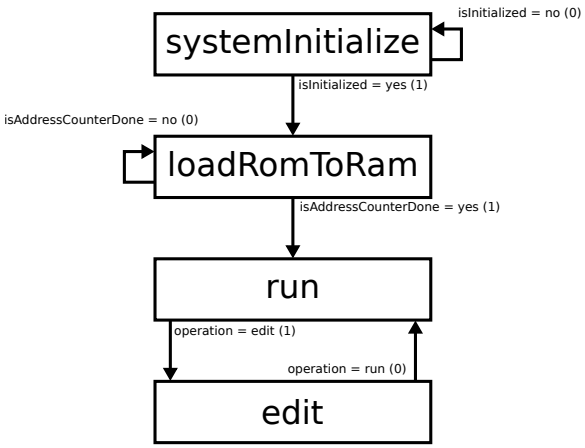

Figure 2: System States

2. **Load**
   The Load state allows the transfer of data from the ROM to the RAM (refer to figure 1). This is done by enabling the Address Counter to send addresses to the Address Bus. The Loader then uses the addresses from the Address Bus to address the ROM. For each address, the corresponding data from the ROM is sent to the Data Bus. At this point, the SRAM Controller can pull the data from the Data Bus and store it into its respective address in the SRAM.

3. **Run**
   The Run state allows 16-bit data to be cycled from the RAM to the Display Controller (7-Segment Display) via the Data Bus. The Address Counter provides an address to the SRAM Controller. The SRAM controller then fetches data from the RAM and sends it to the Data Bus. Data from the Data Bus is encoded with combinational logic within the Display Controller and displayed on the seven-segment.

4. **Edit**
   The Edit state allows 16-bit data to be sent from the Keypad Controller to the RAM via the Data Bus. The Keypad Controller utilizes four sets of one 4-bit serial-in-parallel-out shift registers. Ultimately, a key-press represents four bits going in to the registers and the 16-bit output of the registers corresponds to data sent to the Data Bus. The Keypad Controller is elaborated in section 4.

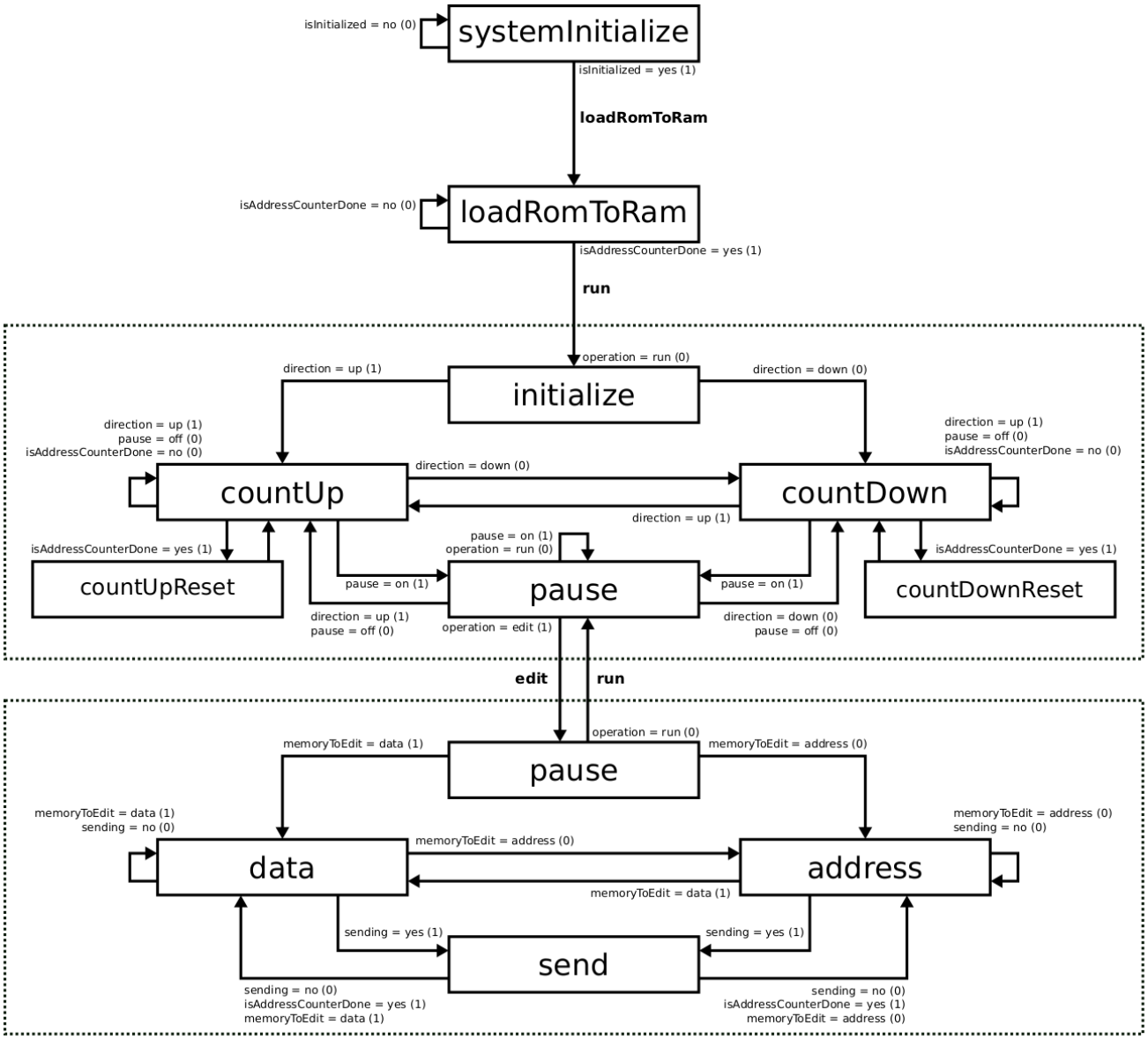

Figure 3: Full State Diagram. Depicts elaborated run and edit states. The run state allows data to be cycled up or down. The edit state allows the data value at a specified address to be sent (to the SRAM from the Keypad Controller).

3.2 State Controller - Encapsulation of State Changes

The State Controller allows state changes to happen all in one place. The State Controller receives flags from entities and outputs states based on the flags. The entities then receive states and react to them accordingly. Importantly, flags are not sent from one entity to another, but specifically all entities to the State Controller. This decouples entities from each other.

4 Keypad Controller

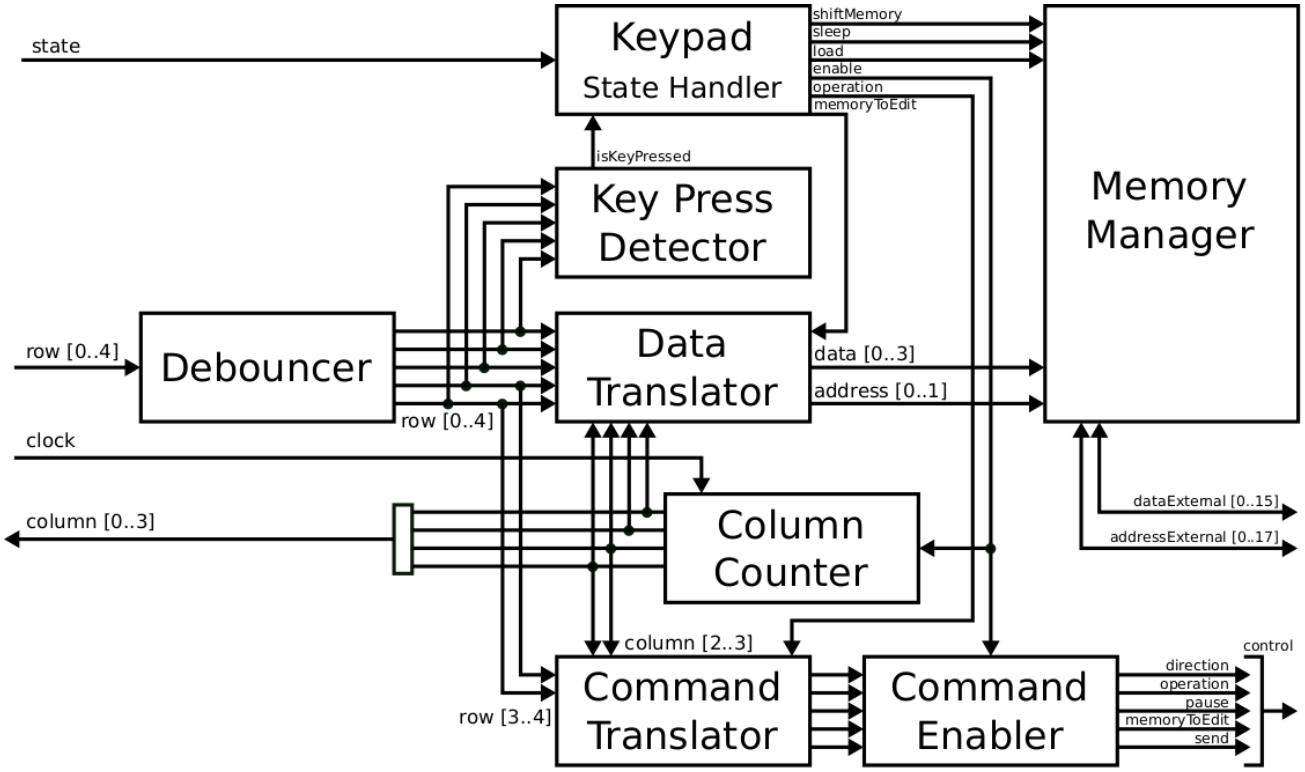

Figure 4: Keypad Controller

4.1 Column Counter

The Column Counter cycles a logic "low" signal to the peripheral keypad's columns. If a key is pressed, and the logic "low" signal corresponds to the column of the pressed key, then a logic "high" signal is generated from the row that corresponds to the pressed key. This can be understood from figure 5. The row data can then be debounced and translated.

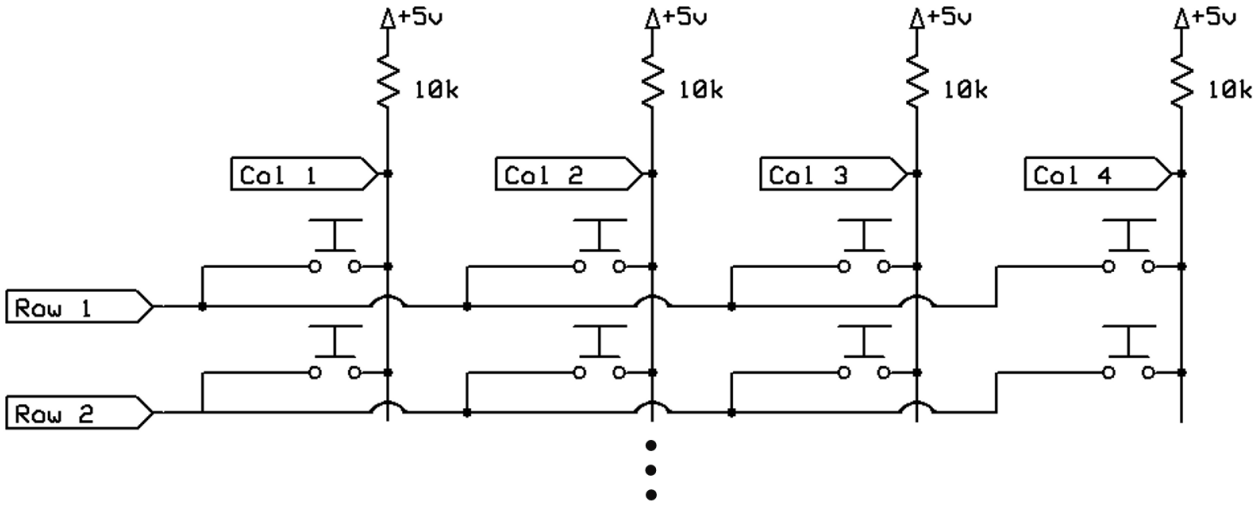

Figure 5: Keypad matrix circuitry.

4.2 Data Translator, Command Translator, Key Press Detector

After the row data is debounced, it can be interpretted. If a data key is pressed (i.e. 0, 1, ... , E, F), then the Data Translator will convert this to its respective hex value. For example, if 'E' is pressed on the keypad, the Data Translator will convert the debounced row data to "1110".

If a command key is pressed (i.e. 'L', 'H', "SHIFT"), then the Command Translator will output a set of commands that are used in the top-level. See section 1.2 for more details about the command keys.

In general, the Key Press Detector will detect if a key is pressed, which will be used as an internal flag.

4.3 State Handler, Command Enabler

The behavior of the Keypad Controller is based on the system state (i.e. System Initialize, Load, Run, Edit). The Keypad Controller will put the Memory Manager in sleep mode, and disable the Command Enabler, when the system state is System Initialize or Load. When the system state is Run, the Command Enabler will be enabled and allow commands to be sent to the top-level. In the Edit state, both the Command Enabler will be enabled and the Memory Manager will be out of sleep mode. In turn, this allows keypad data to be sent to the Data Bus, and commands to be sent to the top-level State Controller.

4.4 Memory Manager

The Memory Manager is a SIPO shift-register (4 bits in, 16 bits out) controlled with flags from the State Handler.

5 Appendix B: VHDL Code

Do not know where the code went, GitHub project was deleted, and files no longer exist on the computer we did it. This is an old top-level meant to test some Keypad Controller entities.

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

entity main is
  port (
        CLOCK_27: in STD_LOGIC; -- On Board 27 MHz
        CLOCK_50: in STD_LOGIC; -- On Board 50 MHz
        KEY: in STD_LOGIC_VECTOR(3 downto 0); -- Pushbutton[3:0]
        HEX0: out STD_LOGIC_VECTOR(6 downto 0); -- Seven Segment Digit 0
        HEX1: out STD_LOGIC_VECTOR(6 downto 0);        -- Seven Segment Digit 1
        HEX2: out STD_LOGIC_VECTOR(6 downto 0);        -- Seven Segment Digit 2
        HEX3: out STD_LOGIC_VECTOR(6 downto 0);        -- Seven Segment Digit 3
        HEX4: out STD_LOGIC_VECTOR(6 downto 0);        -- Seven Segment Digit 4
        HEX5: out STD_LOGIC_VECTOR(6 downto 0);        -- Seven Segment Digit 5
        LEDG: out STD_LOGIC_VECTOR(8 downto 0);        -- LED Green[8:0]
        LEDR: out STD_LOGIC_VECTOR(17 downto 0); -- LED Red[17:0]
        SRAM_DQ: INOUT STD_LOGIC_VECTOR(15 downto 0);        -- SRAM Data bus 16 Bits
        SRAM_ADDR: out STD_LOGIC_VECTOR(17 downto 0); -- SRAM Address bus 18 Bits
        SRAM_UB_N: out STD_LOGIC; -- SRAM High-byte Data Mask
        SRAM_LB_N: out STD_LOGIC; -- SRAM Low-byte Data Mask
        SRAM_WE_N: out STD_LOGIC; -- SRAM Write Enable
        SRAM_CE_N: out STD_LOGIC; -- SRAM Chip Enable
        SRAM_OE_N: out STD_LOGIC; -- SRAM Output Enable
        GPIO_0: INOUT STD_LOGIC_VECTOR(35 downto 0)); -- GPIO Connection 0
end main;
```

```
architecture structural of main is

  component KeyPressDetector is
        port(
          row: in unsigned(4 downto 0);
          isReadyForKeyPress: in std_logic;
          isKeyPressed: out std_logic);
  end component;

  component ColumnCounter is
        port (
          enable: in std_logic;
          clock: in std_logic;
          column: out unsigned(3 downto 0));
  end component;

  component DataTranslator is
        port(
          row: in unsigned(4 downto 0);
          column: in unsigned(3 downto 0);
          state: in std_logic;
          data: out unsigned(3 downto 0);
          address: out unsigned(1 downto 0)
          );
  end component;

  signal isKeyPressed: std_logic;
  signal column: unsigned(3 downto 0);
  signal row: unsigned(4 downto 0);
  signal data: unsigned(3 downto 0);
  signal slowClock: std_logic := '0';
  signal isReadyForKeyPress: std_logic := '1';

BEGIN

  process(CLOCK_50)
        subtype counterRange is integer range 0 to 1000000;
        variable counter: counterRange := 0;
  begin
        if rising_edge (CLOCK_50) then
          if counter = counterRange'high then
                counter := 0;
                slowClock <= not slowClock;
                LEDR(17) <= slowClock;
          else
                counter := counter+1;
          end if;
        end if;
  end process;

  stateHandler: process(isKeyPressed)
```

```
  begin
        isReadyForKeyPress <= not isReadyForKeyPress;
  end process stateHandler;

  buttonPress: KeyPressDetector
        port map(
          row => row,
          isReadyForKeyPress => isReadyForKeyPress,
          isKeyPressed => isKeyPressed
          );

  gpioColumnCounter: ColumnCounter
        port map(
          enable => not isKeyPressed,
          clock => slowClock,
          column => column);

  kepadDataTranslator: DataTranslator
        port map(
          row => row,
          column => column,
          state => isKeyPressed,
          data => data);

  row <= unsigned(GPIO_0(8 downto 4));
  GPIO_0(3 downto 0) <= std_logic_vector(column);
  LEDR(11 downto 8) <= std_logic_vector(data);

  LEDR(16 downto 13) <= std_logic_vector(column);
  LEDR(4 downto 0) <= GPIO_0(8 downto 4);
  LEDG(0) <= isKeyPressed;

end structural;
```

List of Figures