"FPGA-Programming" Exercise Sheet X:

Digital Systems

In this exercise the design of digital systems, exemplified by the implementation of a simple processor, should be demonstrated.

As target hardware always choose FPGA chip Cyclone V SoC 5CSEMA5F31C6

Following figure shows a digital system that contains a number of 9-bit registers, a multiplexer, an adder/subtractor unit, and a control unit (finite state machine).

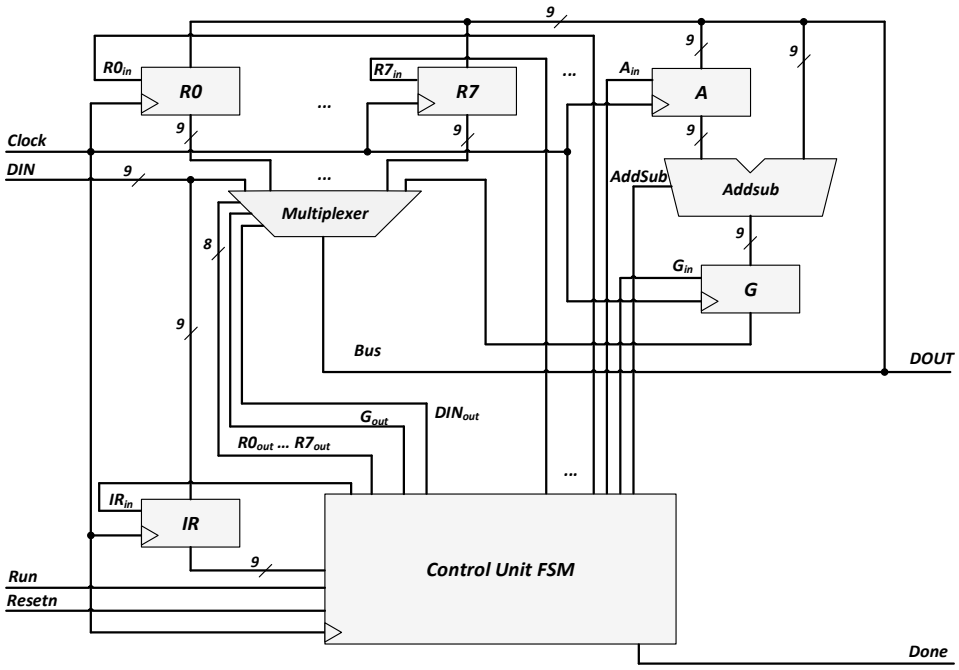

Figure 0.1: A digital system

Data is input to this system via the 9-bit **DIN** input. This data can be loaded through the 9-bit wide multiplexer into the various registers, such as $\mathbf{R0}, \ldots, \mathbf{R7}$ and $\mathbf{A}$. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output signals are called a **bus** in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another.

Addition or subtraction is performed by using the multiplexer to first place one 9-bit number onto the bus wires and loading this number into register $\mathbf{A}$. Once this is done, a second 9-bit number is placed onto the bus, the adder/subtractor unit (**ALU** - Arithmetic Logic Unit) performs the required operation, and the result is loaded into register $\mathbf{G}$. The data in $\mathbf{G}$ can then be transferred to one of the other registers as required.

The digital system can perform different operations in each clock cycle, as governed by the control unit. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data. For example, if the control unit

asserts the signals $\mathbf{R0_{out}}$ and $\mathbf{A_{in}}$, then the multiplexer will place the contents of register $\mathbf{R0}$ onto the bus and this data will be loaded by the next active clock edge into register $\mathbf{A}$.

A system like this is often called a **processor**. It executes operations specified in the form of instructions. Following table lists the instructions that the processor has to support for this exercise:

Operation	Function performed
mv Rx, Ry	$Rx \leftarrow [Ry]$
mvi Rx, $\#D$	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

The left column shows the name of an instruction and its operand. The meaning of the syntax $\mathbf{Rx} \leftarrow [\mathbf{Ry}]$ is that the contents of register $\mathbf{Ry}$ are loaded into register $\mathbf{Rx}$. The **mv** (*move*) instruction allows data to be copied from one register to another. For the **mvi** (*move immediate*) instruction the expression $\mathbf{Rx} \leftarrow \mathbf{D}$ indicates that the 9-bit constant $\mathbf{D}$ is loaded into register $\mathbf{Rx}$.

Each instruction can be encoded and stored in the $\mathbf{IR}$ register using a 9-bit format encoded as $\mathbf{IIIXXXYYY}$, where $\mathbf{III}$ represents the instruction, $\mathbf{XXX}$ gives the $\mathbf{Rx}$ register, and $\mathbf{YYY}$ gives the $\mathbf{Ry}$ register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of this exercise. Instructions are loaded from an external source, hence $\mathbf{IR}$ has to be connected to the nine bits of the $\mathbf{DIN}$ input. For the **mvi** instruction the $\mathbf{YYY}$ field has no meaning, and the immediate data $\#\mathbf{D}$ has to be supplied on the 9-bit $\mathbf{DIN}$ input after the **mvi** instruction word is stored into $\mathbf{IR}$.

Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The finite state machine in the control unit *steps through* such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. The processor starts executing the instruction on the $\mathbf{DIN}$ input when the $\mathbf{Run}$ signal is asserted and the processor asserts the $\mathbf{Done}$ output when the instruction is finished. Following table indicates the control signals that can be asserted in each time step to implement the instructions of above table. Note that the only control signal asserted in time step 0 is $\mathbf{IR_{in}}$, so this time step is not shown in the table.

	T_1	T_2	T_3
(**mv**): I_0	RY_{out}, RX_{in}, $Done$		
(**mvi**): I_1	DIN_{out}, RX_{in}, $Done$		
(**add**): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, $Done$
(**sub**): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, $AddSub$	G_{out}, RX_{in}, $Done$

1) *Simple processor with test-bench*

In this part the simple processor of figure 0.1 should be designed and implemented.

For the fast realization a few VHDL code snippets are given. Following snippet shows the port specification of the entity **e_my_simple_proc**.

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

entity e_my_simple_proc is
   port( slv_DIN:       in       std_logic_vector(8 downto 0);
         sl_Resetn:     in       std_logic;
         sl_Clock:      in       std_logic;
         sl_Run:        in       std_logic;
         sl_Done:       out      std_logic;
         slv_DOUT:      out      std_logic_vector(8 downto 0)
       );
end entity e_my_simple_proc;
```

In addition following incomplete architecture description **a_my_simple_proc** of the top-level entity is given:

```
architecture a_my_simple_proc of e_my_simple_proc is

   -- TBD: declare components

   -- TBD: declare signals

   -- TBD: declare constants for the operations

   type t_fsm_states is (S_T0, S_T1, S_T2, S_T3);
   signal fsm_state, fsm_nextstate: t_fsm_states;

begin

   slv_I_int <= slv_IR_int(1 to 3);

   -- TBD: Further signal assignments

   I_decX: e_dec3to8 port map (slv_IR_int(4 to 6), '1', slv_Xreg_int);
   I_decY: e_dec3to8 port map (slv_IR_int(7 to 9), '1', slv_Yreg_int);

   p_FSM_Nextstate: process (sl_Clock, sl_Resetn)
   begin
      -- TBD: define sequential process with async. low-active reset
   end process p_FSM_Nextstate;
```

```
   p_FSM_Transitions: process (fsm_state, sl_Run, sl_Done_int)
   begin
      case fsm_state is
         -- data is loaded into IR register in this time step
         when S_T0 => if(sl_Run = '0') then
                         fsm_nextstate <= S_T0;
                      else
                         fsm_nextstate <= S_T1;
                      end if;
         -- TBD: other states
      end case;
   end process p_FSM_Transitions;

   p_FSM_Signals: process (fsm_state, slv_I_int, slv_Xreg_int,
                           slv_Yreg_int)
   begin
      -- TBD: specify initial/default values
      case fsm_state is
         -- store slv_DIN in IR as long as fsm_nextstate = 0
         when S_T0 => sl_IRin_int <= '1';
         -- define signals in time step T1
         when S_T1 =>
            case slv_I_int is
               -- TBD: ...
            end case;
         -- define signals in time step T2
         when S_T2 =>
            case slv_I_int is
              --TBD: ...
            end case;
         -- define signals in time step T3
         when S_T3 =>
            case slv_I_int is
              --TBD: ...
            end case;
      end case;
   end process p_FSM_Signals;

   I_reg_0: e_regn generic map (n => 9)
                   port map (slv_Bus_int, slv_Rin_int(0), sl_Clock,
                             sl_Resetn, slv_R0_int);

   -- TBD: instantiate other registers

   -- TBD: create a process p_ALU for slv_Sum_int

   -- TBD: define selection signal slv_Sel_int

   -- TBD: define the internal bus, create a process p_busmux

end architecture a_my_simple_proc;
```

The circuit of the sub-level entity **e_dec3to8** can be described as follows:

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

entity e_dec3to8 is
   port( slv_W:    in   std_logic_vector(2 downto 0);
         sl_En:    in   std_logic;
         slv_Y:    out  std_logic_vector(0 to 7)
       );
end entity e_dec3to8;

architecture a_dec3to8 of e_dec3to8 is

begin

   p_decode: process (slv_W, sl_En)
   begin
      if(sl_En = '1') then
         case slv_W is
            when "000"  => slv_Y <= "10000000";
            when "001"  => slv_Y <= "01000000";
            when "010"  => slv_Y <= "00100000";
            when "011"  => slv_Y <= "00010000";
            when "100"  => slv_Y <= "00001000";
            when "101"  => slv_Y <= "00000100";
            when "110"  => slv_Y <= "00000010";
            when "111"  => slv_Y <= "00000001";
            when others => slv_Y <= "00000000";
         end case;
      else
         slv_Y <= "00000000";
      end if;
   end process p_decode;

end architecture a_dec3to8;
```

The circuit of the sub-level entity **e_regn** can be described as follows:

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

entity e_regn is
   generic( n: integer := 9);
   port( slv_R:       in   std_logic_vector(n-1 downto 0);
         sl_Rin:      in   std_logic;
         sl_Clock:    in   std_logic;
         sl_Resetn:   in   std_logic;
         slv_Q:       out  std_logic_vector(n-1 downto 0)
       );
end entity e_regn;

architecture a_regn of e_regn is

begin

   p_reg: process (sl_Clock, sl_Resetn)
   begin
      if(sl_Resetn = '0') then
         slv_Q <= (others => '0');
      elsif(rising_edge(Clock)) then
         if(sl_Rin = '1') then
           slv_Q <= slv_R;
         end if;
      end if;
   end process p_reg;

end architecture a_regn;
```

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_simple_proc**.

ii) Create the necessary VHDL files. Use above code snippets and compile the completed circuit.

iii) Create a test-bench (**e_my_simple_proc.vht**) and verify the correct behaviour of the circuit.

iv) Ensure that the clock signal will be assigned with a default value of 0 and add a free running $50MHz$ clock to the test-bench.

v) Create a process named **p_stimulus** in which the necessary procedures of the test-bench will be declared and instantiated. First create a procedure named **P_stable** in which a stable post-reset condition will be induced. Therefore assign default values to the signals **slv_DIN**, **sl_Run** and **sl_Resetn**. Remove the reset after one clock cycle.

vi) Write a procedure named **P_sync** with one parameter value, which let pass a selectable amount of clock cycles. Use the non-synthesizable VHDL construct **while - loop**.

vii) Code a procedure named **P_mv_op** with two parameter values **c_Rx** and **c_Ry**. After half of a clock cycle the input **slv_DIN** should be assigned with the instruction **mv** Rx, [Ry] and the **sl_Run** signal should be set. After one clock cycle the **sl_Run** signal should be de-asserted and the data input should be assigned with a zero vector.

viii) Write a procedure named **P_mvi_op** with two parameter values **c_DIN** and **c_Rx**. After half of a clock cycle the input **slv_DIN** should be assigned with the instruction **mvi** Rx, #D and the **sl_Run** signal should be set. After one clock cycle the **sl_Run** signal should be de-asserted and the data input should be assigned with the data **c_DIN**. After one additional clock cycle the data input should be assigned with a zero vector.

ix) Code a procedure named **P_add_op** with two parameter values **c_Rx** and **c_Ry**. After half of a clock cycle the input **slv_DIN** should be assigned with the instruction **add** Rx, Ry and the **sl_Run** signal should be set. After one clock cycle the **sl_Run** signal should be de-asserted and the data input should be assigned with a zero vector. Hereupon three additional clock cycles should elapse.

x) Write a procedure named **P_sub_op** with two parameter values **c_Rx** and **c_Ry**. After half of a clock cycle the input **slv_DIN** should be assigned with the instruction **sub** Rx, Ry and the **sl_Run** signal should be set. After one clock cycle the **sl_Run** signal should be de-asserted and the data input should be assigned with a zero vector. Hereupon three additional clock cycles should elapse.

xi) Test the functionality of the processor and the test-bench. Therefore use appropriate instances of the created procedures.

2) *Implementation of the simple processor*

The simple processor created in part 1) in this part should be embed into a VHDL top-level entity. The functionality should be evaluated in a hardware implementation.

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_processor**.

ii) Include the VHDL entities **e_dec3to8**, **e_regn** and the processor as **e_simple_proc**.

iii) Create the top-level entity (**e_my_processor**) and instantiate the circuit of the simple processor within the top-level entity. Use switches $SW[8-0]$ for the assignment of input **slv_DIN** and switch $SW[9]$ as **sl_Run** signal. Use push-button $KEY[0]$ for the **sl_Reset** input and $KEY[1]$ as a manual **sl_Clock** input. Connect the **slv_Bus** of the processor with $LEDR[8-0]$ and the **sl_Done** signal with $LEDR[9]$.

iv) Import the necessary pin assignments, compile the circuit and download it into the FPGA chip.

v) Test the functionality of the processor. Since the clock must be applied manually, the execution of the individual instructions can be observed step by step.

3) *Simple processor with memory interface*

In this part a circuit as given in following figure should be implemented:

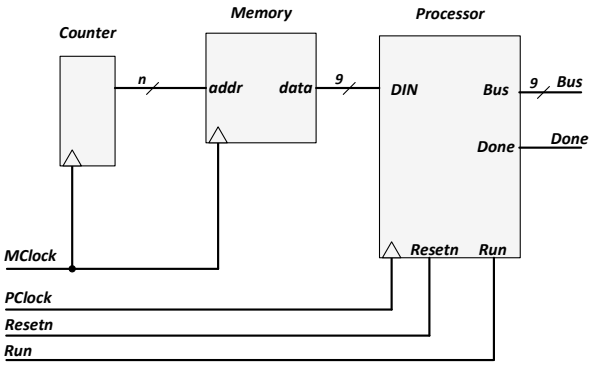

Figure 2.1: Connecting the processor to a memory

Here a memory module and a counter will be connected to the processor designed in part 1). The counter is used to read the contents of successive addresses in the memory, and this data is provided to the processor as a stream of instructions. To simplify the design and testing of this circuit separate clock signals are used, one for the processor (**PClock**) and one for the memory and counter (**MClock**).

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_ifx_proc**.

ii) Create the top-level entity (**e_my_ifx_proc**) which instantiates the processor (**e_simple_proc**), the memory (**e_rom**) and the counter (**e_cnt**). Again use Quartus **IP Catalog** for the creation of an LPM module for the memory. The correct module a **Read Only Memory** (ROM), is found under **Basic Functions** $\rightarrow$ **On Chip Memory** with name **ROM: 1-PORT**. Follow the instructions provided by the wizard to create a memory that has one 9-bit wide read data port and is 32 words deep. Since this memory has only a read port, and no write port, it is called a **synchronous read-only memory** (synchronous ROM). Note that the memory includes a register for synchronously loading addresses. This register is required due to the design of the M10K memory resources on the Cyclone series of FPGAs.

iii) To place processor instructions into the memory, initial values that should be stored in the memory once the circuit has been programmed into the FPGA chip must be specified. This can be done by telling the wizard to initialize the memory using the contents of a **Memory Initialization File** (MIF). The name of the MIF file should be **inst_mem.mif**. Set the contents of your MIF file such that it provides enough processor instructions to test the circuit.

iv) Ensure that the design exhibits the needed port descriptions for hardware test. Use switch $SW[9]$ for driving the **sl_Run** signal, push-button $KEY[0]$ as a low-active **sl_Reset**, push-button $KEY[1]$ as manual clock input of the ROM (**sl_MClock**)

and push-button $KEY[2]$ as manual clock input of the processor (**sl_PClock**). Connect the **slv_Bus** of the processor with $LEDR[8-0]$ and the **sl_Done** signal with $LEDR[9]$.

v) Import the necessary pin assignments, compile the circuit and download it into the FPGA device.

vi) Verify the correct functionality of the processor. Since the clock must be applied manually, the execution of the individual instructions can be observed step by step.