"FPGA-Programming" Exercise Sheet XI:

Advanced Digital Systems

In Exercise Sheet X) a simple processor as an example of a digital system has been described. Goal of this exercise is the description of further components of processor designs.

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

1) *Extension of the simple processor with test-bench*

In this part the design of the simple processors should be extended in a way so that the external counter is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. Therefore three new types of instructions should be added to the processor, which are displayed in following table.

Operation	Function performed
ld Rx, $[Ry]$	$Rx \leftarrow [[Ry]]$
st Rx, $[Ry]$	$[Ry] \leftarrow [Rx]$
mvnz Rx, Ry	if $G \neq 0$, $Rx \leftarrow [Ry]$

The **ld** (*load*) instruction loads data into register $\mathbf{RX}$ from the external memory address specified in register $\mathbf{RY}$. The **st** (*store*) instruction stores the data contained in register $\mathbf{RX}$ into the memory address found in $\mathbf{RY}$. Finally, the instruction **mvnz** (*move if not zero*) allows a **mv** (*move*) operation to be executed only under the condition that the current contents of register $\mathbf{G}$ are not equal to 0.

Following figure shows the schematic of the enhanced simple processor.

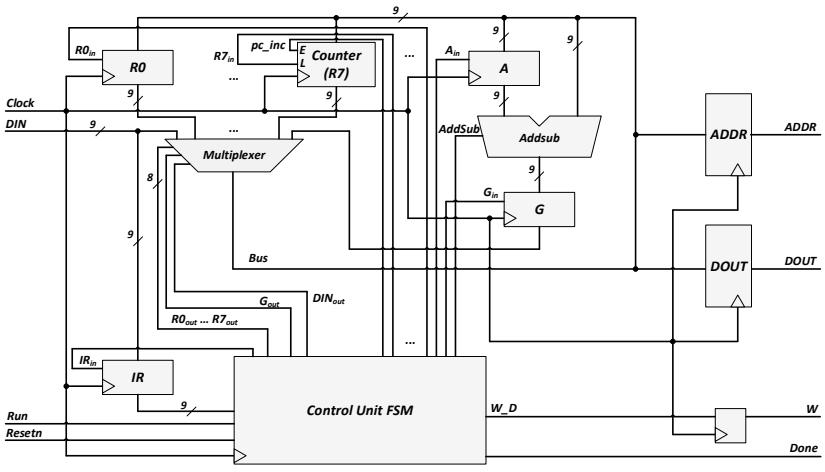

Figure 1.1: An extended digital system

The registers $\mathbf{R0}$ bis $\mathbf{R6}$ are the same as the one of Exercise Sheet X), but register $\mathbf{R7}$ has been changed to a counter.

This counter is used to provide the addresses in the memory from which the processor's instructions are read. Register **R7** is called the processor's **Programm Counter** (*PC*), a terminology which is common for real processors available in the industry. When the processor is reset, **PC** is set to address 0. At the start of each instruction (in time step 0) the contents of **PC** are used as an address to read an instruction from the memory. The instruction is stored in **IR** and the **PC** is automatically incremented to point to the next instruction (in the case of **mvi** the **PC** provides the address of the immediate data and is then incremented again).

The processor's control unit increments **PC** by using the **pc_inc** signal, which is just an enable on this counter. It is also possible to directly load an address into **PC** by having the processor execute a **mv** or **mvi** instruction in which the destination register is specified as **R7**. In this case the control unit uses the signal **R7_in** to perform a parallel load of the counter. In this way, the processor can execute instructions at any address in memory, as opposed to only being able to execute instructions that are stored in successive addresses. Similarly, the current contents of **PC** can be copied into another register by using a **mv** instruction.

An example of code that uses the PC register to implement a loop is shown below, where the text after the % symbol on each line is just a comment.

```
mvi   R2,#1
mvi   R4,#10000000 % binary delay value
mv    R5,R7 % save address of next instruction
sub   R4,R2 % decrement delay count
mvnz  R7,R5 % continue subtracting until delay count gets to 0
```

The instruction **mv R5, R7** places into **R5** the address in memory of the instruction **sub R4, R2**. Then, the instruction **mvnz R7, R5** causes the **sub** instruction to be executed repeatedly until **R4** becomes 0. This type of loop could be used in a larger program as a way of creating a *delay*.

Above figure shows two registers in the processor that are used for data transfers. The **ADDR** register is used to send addresses to an external device, such as a memory module, and the **DOUT** register is used by the processor to provide data that can be stored outside the processor. One use of the **ADDR** register is for reading, or fetching, instructions from memory; when the processor wants to fetch an instruction, the contents of **PC** are transferred across the bus and loaded into **ADDR**. This address is provided to memory. In addition to fetching instructions, the processor can read data at any address by using the **ADDR** register. Both data and instructions are read into the processor on the **DIN** input port. The processor can write data for storage at an external address by placing this address into the **ADDR** register, placing the data to be stored into its **DOUT** register, and asserting the output of the **W** (*write*) flip-flop to 1.

Due to the more complex input / output interface, the actions per time step have to be changed according to following tables:

	T_0	T_1	T_2
(**mv**): I_0	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}
(**mvi**): I_1	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}
(**add**): I_2	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}
(**sub**): I_3	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}
(**ld**): I_4	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}
(**st**): I_5	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}
(**mvnz**): I_6	$R7_{out}$, $ADDR_{in}$, (if $run = 1 : pc\_inc$)	$R7_{out}$, $ADDR_{in}$	IR_{in}

	T_3	T_4	T_5
(**mv**): I_0	RY_{out}, RX_{in}, $Done$		
(**mvi**): I_1	DIN_{out}, RX_{in}, pc_inc, $Done$		
(**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}, $Done$
(**ld**): I_4	RY_{out}, $ADDR_{in}$	$-$	DIN_{out}, RX_{in}. $Done$
(**st**): I_5	RY_{out}, $ADDR_{in}$	RX_{out}, $DOUT_{in}$, W_D	$Done$
(**mvnz**): I_6	(only if $G \neq 0 : RY_{out}$, RX_{in}), $Done$		

Perform following steps:

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

ii) Create the necessary VHDL sub-level entities (**e_dec3to8**, **e_regn**, **e_flipflop**, **e_pc_count**).

iii) Describe the VHDL top-level entity **e_my_ext_proc** of the processor shown in figure 1.1. For the description of the functionality keep in kind that the inputs of the external memory module will be registered.

iv) Create a test-bench (**e_my_ext_proc.vht**). Create the framework of the test-bench in Quartus by pressing **Processing** $\rightarrow$ **Start** $\rightarrow$ **Start Testbench Template Writer**.

v) 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.

vi) Create a process named **p_stimulus** in which the necessary procedures of the test-bench will be declared and instantiated. Write the procedures **P_stable**, **P_sync**, **P_mv_op**, **P_mvi_op**, **P_add_op** and **P_sub_op** with the same functionality as in Exercise Sheet X).

vii) Code additional procedures **P_ld_op**, **P_st_op** and **P_mvnz_op** to be able to test the new instructions *load*, *store* and *move if not zero*.

viii) Test the functionality of the processor and the test-bench. Therefore use appropriate instances of the created procedures. Ascertain yourself about the possibility to implement a delay loop.

2) *Implementation of the extended processor*

Following figure shows, how the extended simple processor can be interfaced with other components.

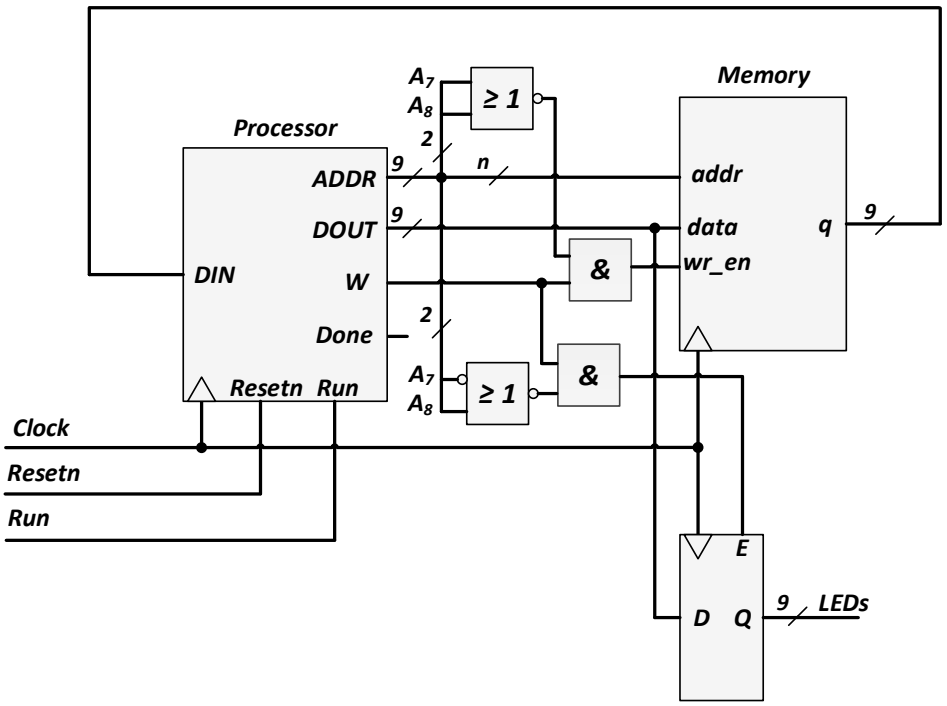

Figure 1.2: Connecting the enhanced processor to a memory and output register

The memory unit in the figure supports both read and write operations and therefore has both address and data inputs, as well as a write enable input. The memory also has a clock input, because the address, data, and write enable inputs must be loaded into the memory on an active clock edge. This type of memory unit is usually called a **static random access memory** (synchronous *SRAM*).

The design also includes a 9-bit register that can be used to store data from the processor. This is for evaluation purposes as part of this exercise sheet. To allow the processor to select either the memory unit or register when performing a write operation, the circuit includes some logic gates that perform address decoding: if the upper address lines are $\mathbf{A_8A_7} = \mathbf{00}$, then the memory module will be written at the address given on the lower address lines $\mathbf{A_6A_5A_4A_3A_2A_1A_0}$. For this exercise a memory with 128 words is probably sufficient, which implies that $n = 7$. For addresses in which $\mathbf{A_8A_7} = \mathbf{01}$, the data written by the processor is loaded into the register whose outputs drive the red LEDs.

The extended processor of part 1) in this part should be embed into a VHDL top-level entity according to figure 1.2. The functionality should be evaluated in a hardware implementation.

Perform following steps:

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

ii) Include the VHDL entities **e_dec3to8**, **e_regn**, **e_flipflop**, **e_pc_count** and the extended processor as **e_ext_proc**.

iii) Create a VHDL top-level entity (**e_my_ext_processor**) and instantiate the circuit of the extended processor, the memory (**e_inst_mem**) and the register within the top-level entity. For the description of the memory use Quartus **IP Catalog** in order to generate a **RAM: 1-PORT** LPM module. Create a RAM with 9-bit wide read-write interface and a depth of 128 words. Ensure that the memory does not exhibit output registers. Use following MIF file (**inst_mem.mif**):

```
depth = 128;
width = 9;
address_radix = HEX;
data_radix = BIN;
content
begin
% This code displays a count (in Register R2) on the red LEDs
00: 001001000; %       mvi  R1,#1          // initialize R                    %
01: 000000001;
02: 001010000; %       mvi  R2,#0          // counter to display on LEDs      %
03: 000000000;
04: 001011000; % Loop  mvi  R3,#010000000 // R3 = address of LED register   %
05: 010000000;
06: 101010011; %       st   R2,R3          // write to LEDs                   %
07: 010010001; %       add  R2,R1          // increment counter for LEDs      %
08: 001011000; %       mvi  R3,#111111111 // delay value                     %
09: 111111111;
0A: 000101111; %       mv   R5,R7          // save address of next inst.      %
0B: 001100000; % Outer mvi  R4,#111111111 // nested delay loop               %
0C: 111111111;
0D: 000000111; %       mv   R0,R7          // save address of next inst.      %
0E: 011100001; % Inner sub  R4,R1          // decrement loop delay variable   %
0F: 110111000; %       mvnz R7,R0          // continue Inner loop if R4 != 0  %
10: 011011001; %       sub  R3,R1          // decrement outer loop delay      %
11: 110111101; %       mvnz R7,R5          // continue Outer loop if R3 != 0  %
12: 001111000; %       mvi  R7,#Loop       // execute again                   %
13: 000000100;
```

iv) Use switch $SW[9]$ as **sl_Run** input, push-button $KEY[0]$ as low-active reset signal **sl_Resetn** and the 50 MHz clock signal for **sl_Clock**. Since the circuit must be able to run on 50 MHz, ensure to set an appropriate clock constraint in Quartus **Time Quest Timing Analyzer**. After compilation check the report of the Time Quest Timing Analyzer. If the timing requirements cannot be hold, modify the circuit until the timing fits. Furthermore notice that the **sl_Run** signal is asynchronous. Ensure to synchronize it by using two instances of the component **e_flipflop**. Connect the outputs of the register to $LEDR[8-0]$ to be able to observe the output of the processor.

v) Import the necessary pin assignments, compile the circuit and download it into the FPGA device. Verify the correct behaviour of the extended processor in hardware.

3) *Extended processor with output module*

In this part an additional output module should be added to the circuit of part 2). Therefore a module named **e_seg7_scroll** should be added. This module should contain one register for each 7-segment display of the Eval-Board. Each register should directly drive the segment lights for one 7-segment display, so that the processor can write characters onto these displays. Create the necessary address decoding to allow the processor to write to the registers in the **e_seg7_scroll** module.

Perform following steps:

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

ii) Describe the circuit of the entity **e_seg7_scroll**, embed it into the extended simple processor design of part 2) and change the logic control accordingly.

iii) Add **timing constraints** and import the necessary pin assignments. Create a **MIF** file, which instructs the processor to write symbols to the 7-segment displays. A simple program would write specific symbols and terminate. A sophisticated program could scroll a message accross the displays, or scroll a word across the displays in the left, right, or both directions.

iv) Compile the circuit and download it into the FPGA device. Test the functionality of the design with the described memory contents.

4) *Extended processor with input module*

In this part the circuit of part 3) should be extended by an addition module named **e_port_n**. This should enable the processor to read the position of some switches of the Eval-Board. The value set by the switches should be stored into a register and the processor should be able to perform a **ld** (*load*) instruction to read this register. A logic circuit must be developed which uses address decoding and multiplexers to allow the processor to read from either the memory or **e_port_n**.

Perform following steps:

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

ii) Describe the circuit of the entity **e_port_n**, embed it into the extended simple processor design of part 3) and change the logic control accordingly.

iii) Add **timing constraints** and import the necessary pin assignments. Create a **MIF** file which demonstrates the usage of the module **e_port_n**. A sophisticated program instructs the processor to scroll a message (e.g.: **dE1**) across the 7-segment displays and use the values read from the **e_port_n** module to change the speed at which the message is scrolled.

iv) Compile the circuit and download it into the FPGA device. Test the functionality of the design with the described memory contents.