# Pre Laboratory 5:

# A SIMPLE PROCESSOR

## OBJECTIVES

- The purpose of this lab is to learn how to connect simple input (switches) and output devices (LEDs and 7-segment) to an FPGA chip and implement a circuit that uses these devices.
- Examine a simple processor.

## PREPARATION FOR LAB 5

- Students have to simulate all the exercises in Pre Lab 5 at home. All results (codes, waveform, RTL viewer, … ) have to be captured and submitted to instructors prior to the lab session. *If not, students will not participate in the lab and be considered absent this session.*

## REFERENCE

1. Intel FPGA training

# Pre Laboratory 5:

# A SIMPLE PROCESSOR

## INTRODUCTION: A simple processor

Figure 1 shows a digital system that contains a number of nine-bit registers, a multiplexer, an adder/subtractor unit, and a control unit (finite state machine). Data is input to this system via the nine-bit *DIN* input. This data can be loaded through the nine-bit wide multiplexer into the various registers, such as *R*0, *R1*,… , *R*7 and *A*. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output wires 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.

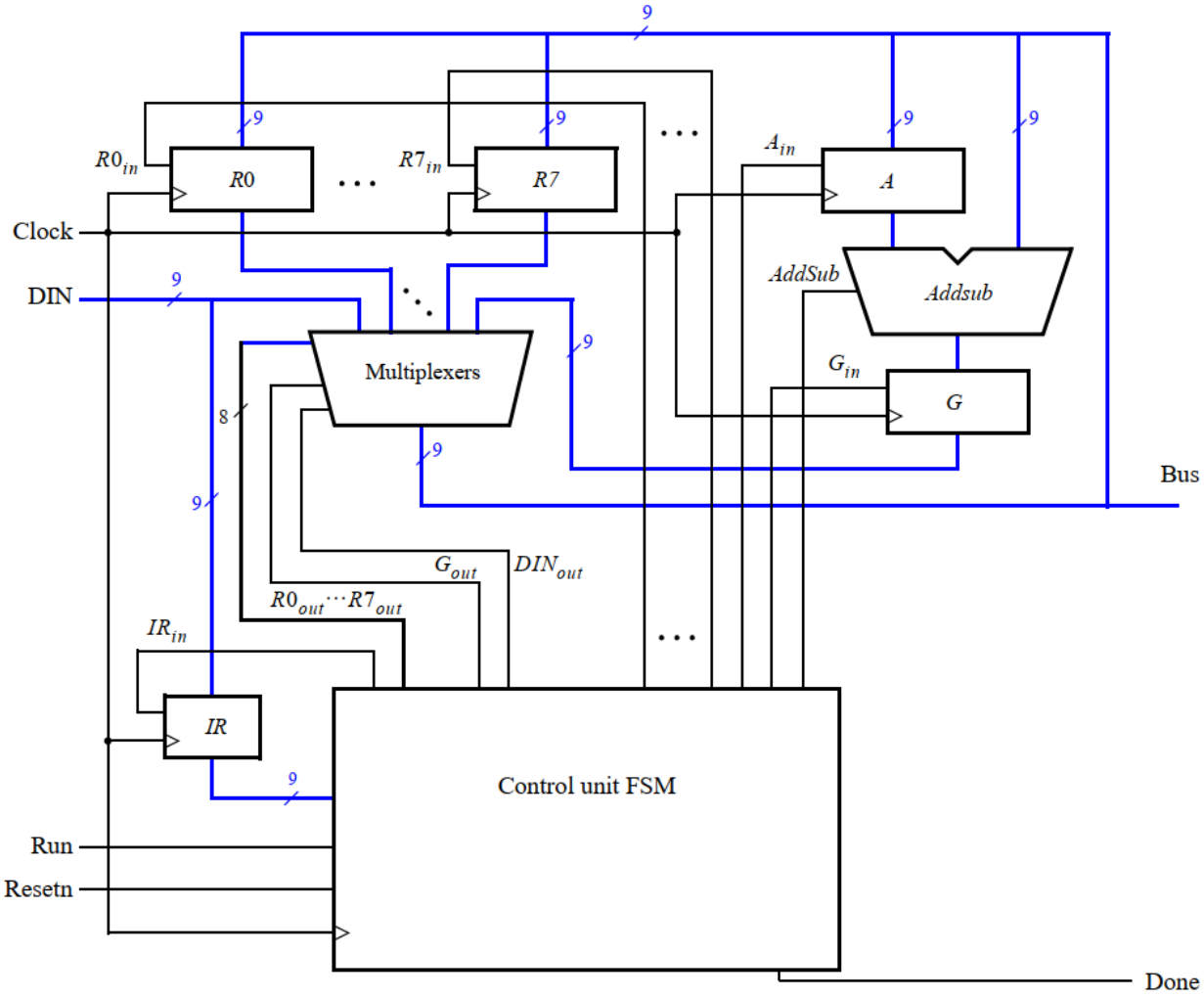



*Figure 1:* A digital system.

# Pre Laboratory 5:

# A SIMPLE PROCESSOR

Addition or subtraction of signed numbers is performed by using the multiplexer to first place one nine-bit number onto the bus wires and loading this number into register *A*. Once this is done, a second nine-bit number is placed onto the bus, the adder/subtractor unit performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred to one of the other registers as required.

The 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 *R*0*out* and *Ain*, then the multiplexer will place the contents of register *R*0 onto the bus and this data will be loaded on the next active clock edge into register *A*.

A system like the one in Figure 1 is often called a *processor*. It executes operations specified in the form of *instructions*. Table 1 lists the instructions that the processor has to support for this exercise. The left column shows the name of an instruction and its operands. The meaning of the syntax R*x* ← [R*y*] is that the contents of register R*y* are loaded into register R*x*. The **mv** (move) instruction allows data to be copied from one register to another. For the **mvi** (move immediate) instruction the expression R*x* ← D indicates that the nine-bit constant D is loaded into register R*x*.

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

*Table 1:* Instructions performed in the processor.

Each instruction can be encoded using the nine-bit format *IIIXXYYY* called **machine code**, where *III* specifies the instruction (opcode), *XXX* gives the R*x* register, and *YYY* gives the R*y* 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 the exercise. Assume that *III* = 000 for the **mv** instruction, 001 for **movi**, 010 for **add**, and 011 for **sub**. Instructions are loaded from the the external input *DIN*, and stored into the *IR* register, using the connection

indicated in Figure 1. For the **mvi** instruction the *YYY* field has no meaning, and the immediate data *#D* has to be supplied on the *DIN* input in the clock cycle after the **mvi** instruction word is stored into *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 *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. Table 2 indicates the control signals that can be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step 0 is *IRin*, 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* |

*Table 2:* Control signals asserted in each instruction/time step.

# Pre Laboratory 5:

# A SIMPLE PROCESSOR

**EXERCISE 1:**

***Objective:*** Known how to convert instructions to machine codes.

***Requirement:*** Use the information in the introduction, convert these instructions to machine codes.

```
mv   R1,R4
mv   R3,R2
mvi  R2,#5
mvi  R4,#-6
add  R3,R7
add  R2,R0
sub  R5,R6
```

***Check:*** Your report has to show two results:

- Describe the instructions.
- The machine code results.

# Pre Laboratory 5:

# A SIMPLE PROCESSOR

## EXERCISE 2

***Objective:*** Understand the control unit of the processor.

***Requirement:*** Use the information in the introduction, sketch the FSM of the control unit.

***Check:*** Your report has to show two results:

- FSM diagram.
- Control signals of each state.

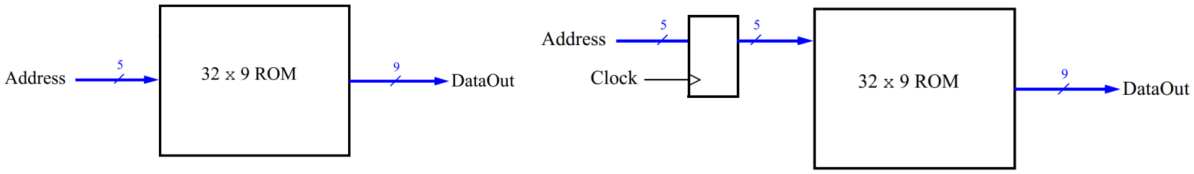
# Pre Laboratory 5:

# A SIMPLE PROCESSOR

## EXERCISE 3

***Objective:*** Understand the memory block (ROM) organization.

***Requirement:*** A diagram of the random access memory (ROM) module that we will implement is shown in Figure 2a. It contains 32 four-bit words (rows), which are accessed using a nine-bit *address* port, a four-bit *data* port.




(a) ROM organization (b) ROM implementation

*Figure 2:* A 32 x 9 ROM module.

***Instruction:*** The VHDL code for ROM (nine-bit *address* port, a four-bit *data* port) is shown:

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

entity myROM is
   generic(
      addr_width : integer := 32; -- store 32 elements
      addr_bits  : integer := 5; -- required bits to store 32 elements
      data_width : integer := 9 -- each element has 9-bits
      );
port(
   addr : in std_logic_vector(addr_bits-1 downto 0);
   data : out std_logic_vector(data_width-1 downto 0)
);
end myROM;

architecture arch of myROM is

   type rom_type is array (0 to addr_width-1) of std_logic_vector(data_width-1 downto 0);

   signal user_ROM : rom_type;

    -- note that 'ram_init_file' is not the user-defined-name (it is attribute name)
   attribute ram_init_file : string;
   -- "rom_data.mif" is the relative address with respect to project directory
   -- suppose ".mif" file is saved in folder "ROM", then use "ROM/rom_data.mif"
   attribute ram_init_file of user_ROM : signal is "rom_data.mif";

begin
```

```
    data <= user_ROM (to_integer(unsigned(addr)));
end arch;
```

After constructing ROM, data inside is empty. We can define data in ROM by using "my_ROM.mif" file. This file is only used with Quartus tool (see comments for details). An example of '.mif' file is shown below. In '.mif' file, the comments are written between two '% %' signs (both single line and multiline). Further, we need to define certain parameters i.e. data and address types (see comments for details). Lastly, in file, we set the values at all the addresses as '0' and then values are assigned at each address. This can be useful, when we want to store data at fewer locations. (Remember that in this file mif example, ROM is 16x7).

```
% rom_data.mif %
% ROM data for seven segment display %

% data width and total data %
width=7;  % number of bits in each data %
depth=16; % total number of data (i.e. total address) %

%
   format of data and address stored in this file
   uns : unsigned,  dec : decimal,  hex : hexadecimal
   bin : binary,  oct : octal
%
address_radix=uns;  % address is unsigned-type %
data_radix=bin;     % data is binary-type %

% ROM data %
content begin
   [0..15] : 0000000;      % optional : assign 0 to all address %
   0 : 1000000;            % format => signed : binary %
   1 : 1111001;
   2 : 0100100;
   3 : 0110000;
   4 : 0011001;
   5 : 0010010;
   6 : 0000010;
   7 : 1111000;
   8 : 0000000;
   9 : 0010000;
   10 : 0001000;
   11 : 0000011;
   12 : 1000110;
   13 : 0100001;
   14 : 0000110;
   15 : 0001110;

end;
```

# Pre Laboratory 5:

# A SIMPLE PROCESSOR

VHDL top module for ROM is:

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

entity top_module_ROM is
port(
   SW : in std_logic_vector(4 downto 0);
   LEDR : out std_logic_vector(8 downto 0)
);
end top_module_ROM;

architecture arch of top_module_ROM is
   -- signal to store received data
   signal temp_data : std_logic_vector (8 downto 0);
begin
   user_ROM: entity work.myROM port map (addr=>SW, data=>temp_data);

   LEDR <= temp_data;  -- display on LEDs
end arch;
```

***Check:*** Modify the code above to construct the circuit in Figure 2b. Data in ROM is defined by the machine code in exercise 1.