Pre Laboratory 4:
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 4

- Students have to simulate all the exercises in Pre Lab 4 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 4:

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.

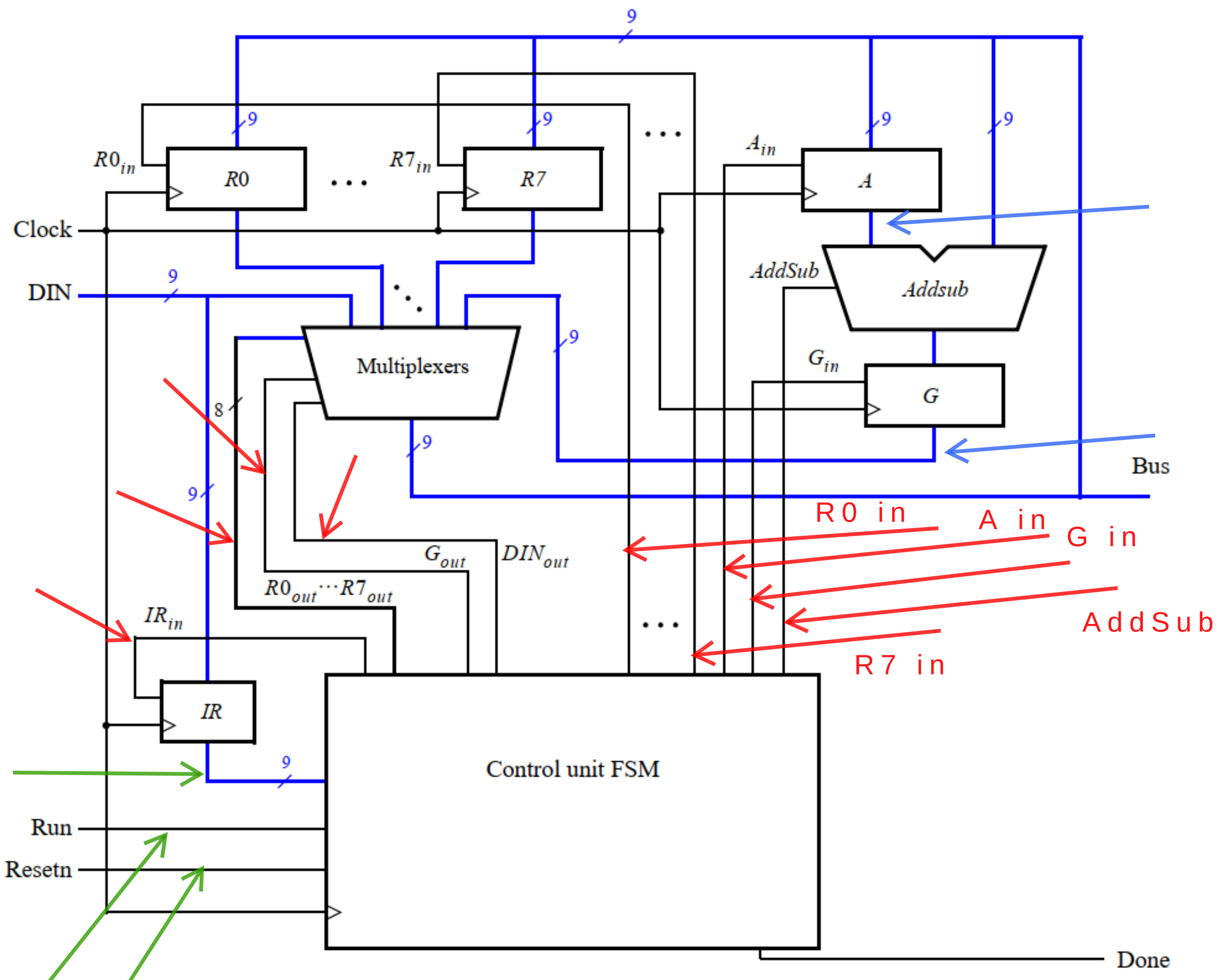


Figure 1: A digital system.

Pre Laboratory 4:
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 *IIIXXXYYY* 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 4:

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

Check: Your report has to show two results:

- Describe the instructions.
- The machine code results.

mv R1, R4
opcode: 000
register X: 001
register Y: 100
machine code: 000 001 100

mv R3, R2
opcode: 000
register X: 011
register Y: 010
machine code: 000 011 010

mvi R2, #5
opcode: 001
register X: 010
registerY: 000 (don't care)
machine code: 001 010 000
immediate D: 000 000 101

mvi R4, #-6
opcode: 001
register X: 100
register Y: 000 (don't care)
machine code: 001 100 000
immediate D: 000 001 010

add R3, R7
opcode: 010
register X: 011
register Y: 111
machine code: 010 011 111

add R2, R0
opcode: 010
register X: 010
registerY: 000
machine code: 010 010 000

sub R5, R6
opcode: 011
register X: 101
register Y: 110
machine code: 011 101 110

Pre Laboratory 4:

A SIMPLE PROCESSOR

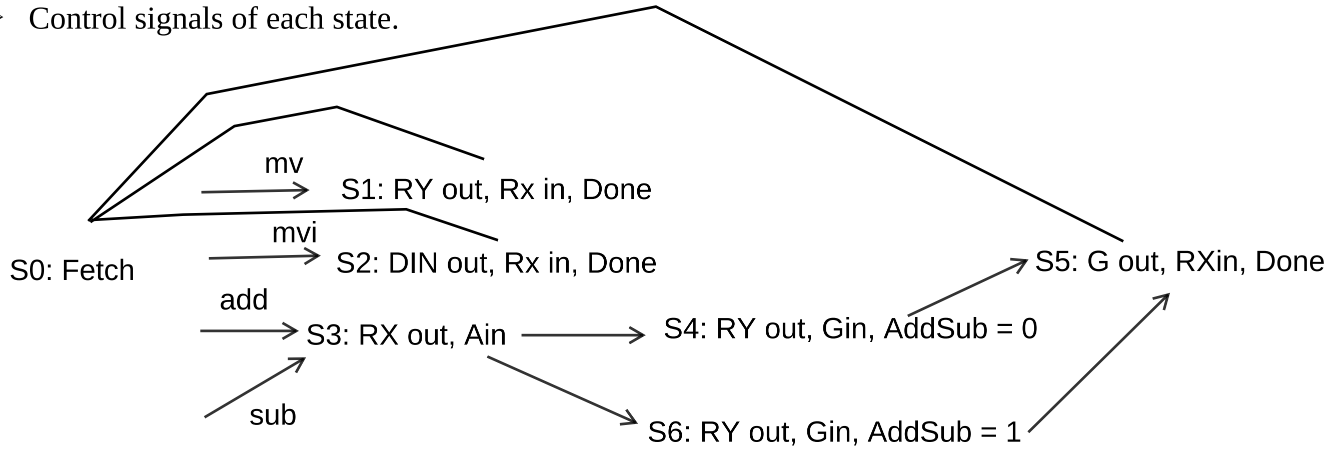
EXERCISE 2

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.



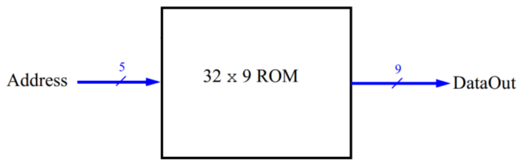
Pre Laboratory 4:

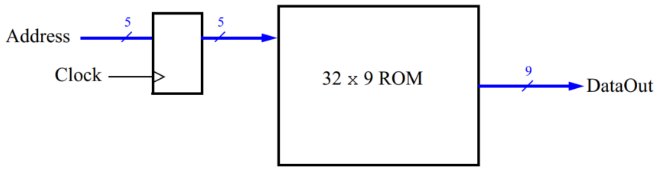
A SIMPLE PROCESSOR

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 System Verilog code for ROM (nine-bit *address* port, a four-bit *data* port) is shown:

```
module MyROM
#(parameter int unsigned width = 9,
parameter int unsigned depth = 32,
parameter intFile = "my_ROM.mif",
parameter int unsigned addrBits = 5)
(
input logic CLK,
input logic [addrBits-1:0] ADDRESS,
output logic [width-1:0] DATAOUT
);

logic [width-1:0] rom [0:depth-1];

// initialise ROM contents
initial begin
$readmemh(intFile,rom);
end

always_ff @ (posedge CLK)
begin
DATAOUT <= rom[ADDRESS];
end
endmodule : MyROM
```

Pre Laboratory 4:
A SIMPLE PROCESSOR

The $readmemh task reads a hex file to the rom array which is effectively the memory. The name of the hex file is passed to the MyROM module as a parameter.

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).

```
% my_ROM.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;
```

System Verilog top module for ROM is:

```
module top_mem (
          input logic CLK,
          input logic [4:0] ADDRESS,
          output logic [8:0] DATA
      );

      MyROM U0 (
      .CLK (CLK ),
      .ADDRESS (ADDRESS ),
      .DATAOUT (DATA )
      );

      endmodule : top_mem
```

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