ELECTRICAL AND ELECTRONICS ENGINEERING
&
COMPUTER ENGINEERING

E E E 2 4 8 | C N G 2 3 2
L O G I C D E S I G N

2 1 | S p r i n g | 2 2

LABORATORIES

D r . G ü r t a ç Y e m i ş ç i o ğ l u
M b o n e a G o d w i n M j e m a
A h m e d M a m d o u h

6 TABLE OF CONTENTS

REGULATIONS

- Students are not permitted to perform an experiment without doing **the preliminary work** before coming to the laboratory. It is **not allowed** to do the preliminary work at the laboratory during the experiment. **Students, who do not turn in the complete preliminary work printout at the beginning of the laboratory session, cannot attend the lab. No "make-up" is given in that case.**

- No food or drink in the lab.

- There may be a quiz before each laboratory session, which will start promptly at the beginning of the lab. **Students who miss the quiz cannot attend the lab. No "make-up" is given in that case.** There won't be any extensions in the quiz time for latecomers. Therefore, students have to be at the lab on time for the quiz.

- Only the following excuses are valid for taking lab make-up:
  1. **Health Make-up:** Having a health report from METU Medical Center.
  2. **Exam Make-up:** Having an exam coinciding with the time of the laboratory session. The student needs to notify the instructor in advance if this is the case.

- Experiments will be done **individually**. The lab instructor will inform you of any part that you can do in groups.

- **Cheating or plagiarism is not tolerated. Plagiarism is a form of cheating as is using someone else's written word with minor changes and no acknowledgement. If you are caught cheating or plagiarising, you will at the very least receive a zero for the whole experiment. Disciplinary action may be taken.**

- Students who miss the lab **two times** without a valid excuse get zero as the laboratory portion of the course grade.

- *Those who fail to get a satisfactory score from the laboratory portion may fail the class. This score is expected to be 70% but may be adjusted up or down with the initiative of the course instructor.*

EXPERIMENT #6 & #7

4-BIT FIBONACCI SERIES CALCULATOR DATAPATH AND CONTROL FSM DESIGN SIMULATION AND IMPLEMENTATION

DUE WEEK 14

6.1 OBJECTIVE

In the final project, you are required to design and implement a Datapath and a Controller FSM for a 4-bit Fibonacci Series Calculator. Altera toolset and Modelsim® will be used to code, simulate, and implement the design.

In LAB 6 (week 14) the Datapath design will be demonstrated with your partner to the LAB instructor as usual. The report must include all the modules simulations and top-level design and simulations in Modelsim®.

In LAB 7 (week 15) the final Controller FSM will be designed and demonstrated individually (not as a group) to the LAB instructor during a reserved ~15-minute time slot. Also, each student will write an individual project report that contains the details of the design and simulation results.

6.2 FINITE STATE MACHINES (TUTORIAL)

Finite State Machines (FSM) are sequential circuit used in many digital systems to control the behavior of systems and dataflow paths. Examples of FSM include control units and sequencers. This lab introduces the concept of two types of FSMs, Mealy and Moore, and the modeling styles to develop such machines.

6.2.1 Objectives

After completing this lab, you will be able to:

- Model Mealy FSMs
- Model Moore FSMs

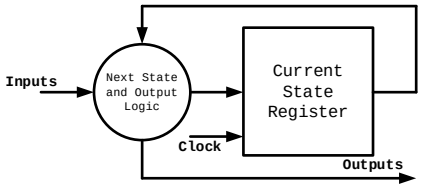
6.2.2 Mealy FSM

A finite-state machine (FSM) or simply a state machine is used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of user-defined states. The machine is in only one state at a time; the state it is in at any given time is called the *current state*. It can change from one state to another when initiated by a triggering event or condition; this is called a *transition*.

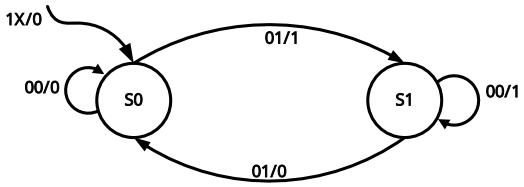
A particular FSM is defined by a list of its states, and the triggering condition for each transition. The behavior of state machines can be observed in many devices in modern society performing a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines which dispense products when the proper combination of coins are deposited, elevators which drop riders off at upper floors before going down, traffic lights which change sequence when cars are waiting, and combination locks which require the input of combination numbers in the proper order.

The state machines are modeled using two basic types of sequential networks- Mealy and Moore. In a Mealy machine, the output depends on both the present (current) state and the present (current) inputs. In Moore machine, the output depends only on the present state.

A general model of a Mealy sequential machine consists of a combinatorial network, which generates the outputs and the next state, and a state register which holds the present state as shown below. The state register is normally modeled as D flip-flops. The state register must be sensitive to a clock edge. The other block(s) can be modeled either using the `always` procedural block or a mixture of the `always` procedural block and dataflow modeling statements; the `always` procedural block will have to be sensitive to all inputs being read into the block and must have all output defined for every branch in order to model it as a combinatorial block. The two blocks Mealy machine can be viewed as

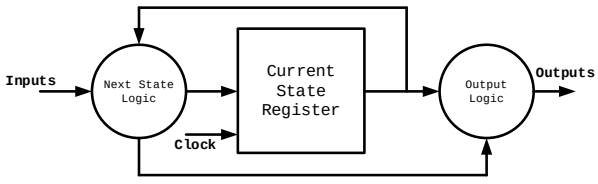


Here are the state diagram of a parity checker Mealy machine and the associated model.




```
module mealy_2processes(input clk, input reset, input x, output reg parity);
  reg state, nextstate;
  parameter S0=0, S1=1;
always @(posedge clk or posedge reset) // always block to update state
if (reset)
  state <= S0;
else
  state <= nextstate;
always @(state or x) // always block to compute both output & nextstate
begin
  parity = 1'b0;
  case(state)
    S0: if(x)
        begin
          parity = 1; nextstate = S1;
        end
        else
          nextstate = S0;
    S1: if(x)
          nextstate = S0;
       else
       begin
          parity = 1; nextstate = S1;
       end
    default:
       nextstate = S0;
    endcase
end
endmodule
```

The three blocks Mealy machine and the associated model are shown below.

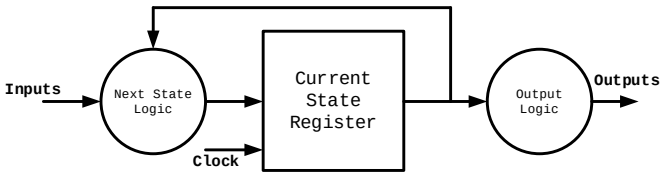


```
module mealy_3processes(input clk, input reset, input x, output reg parity);
    reg state, nextstate;
    parameter S0=0, S1=1;
  always @(posedge clk or posedge reset) // always block to update state
  if (reset)
      state <= S0;
  else
      state <= nextstate;
  always @(state or x) // always block to compute output
  begin
      parity = 1'b0;
      case(state)
        S0: if(x)
          parity = 1;
        S1: if(!x)
          parity = 1;
      endcase
  end
  always @(state or x) // always block to compute nextstate
  begin
     nextstate = S0;
     case(state)
       S0: if(x)
         nextstate = S1;
       S1: if(!x)
         nextstate = S1;
     endcase
  end
endmodule
```

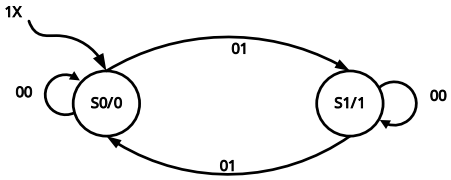
The state assignments can be of one-hot, binary, gray-code, and other types. Usually, the synthesis tool will determine the type of the state assignment, but user can also force a particular type by changing the synthesis property as shown below. The state assignment type will have an impact on the number of bits used in the state register; one-hot encoding using maximum number of bits but decodes very fast to compact (binary) encoding using smallest number of bits but taking longer to decode.

6.2.3 Moore FSM

A general model of a Moore sequential machine is shown below. Its output is generated from the state register block. The next state is determined using the present (current) input and the present (current) state. Here the state register is also modelled using D flip-flops. Normally Moore machines are described using three blocks, one of which must be a sequential and the other two can be modelled using always blocks or a combination of always and dataflow modelling constructs.



Here is the state graph of the same parity checker to be modelled as a Moore machine. The associate model is shown below.




```
module moore_3processes(input clk, input reset, input x, output reg parity);
    reg state, nextstate;
    parameter S0=0, S1=1;
  always @(posedge clk or posedge reset) // always block to update state
  if (reset)
    state <= S0;
  else
    state <= nextstate;
  always @(state) // always block to compute output
  begin
    case(state)
      S0: parity = 0;
      S1: parity = 1;
    endcase
  end
  always @(state or x) // always block to compute nextstate
  begin
    nextstate = S0;
    case(state)
      S0: if(x)
        nextstate = S1;
      S1: if(!x)
        nextstate = S1;
      endcase
  end
endmodule
```

The output block when it is simple, as in this example, can be modelled using dataflow modelling constructs. The following code can be used instead of the always block. You also need to change the output type from reg to wire.

```
assign parity = (state==S0) ? 1'b0: 1'b1;
```

6.3 DATAPATH DESIGN (WEEK 14)

To achieve a 4-bit Fibonacci Series **F(1)=1, F(2)=1 and F(N)=F(N-2)+F(N-1)**, the Datapath requires certain logic operations including, 2-to-4 Line decoder, 4-bit AND gate, five 4-bit 2-to-1 Multiplexers, five 4-bit registers, two 4-bit 4-to-1 Multiplexers, and 4-bit ALU as depicted in Figure 1.

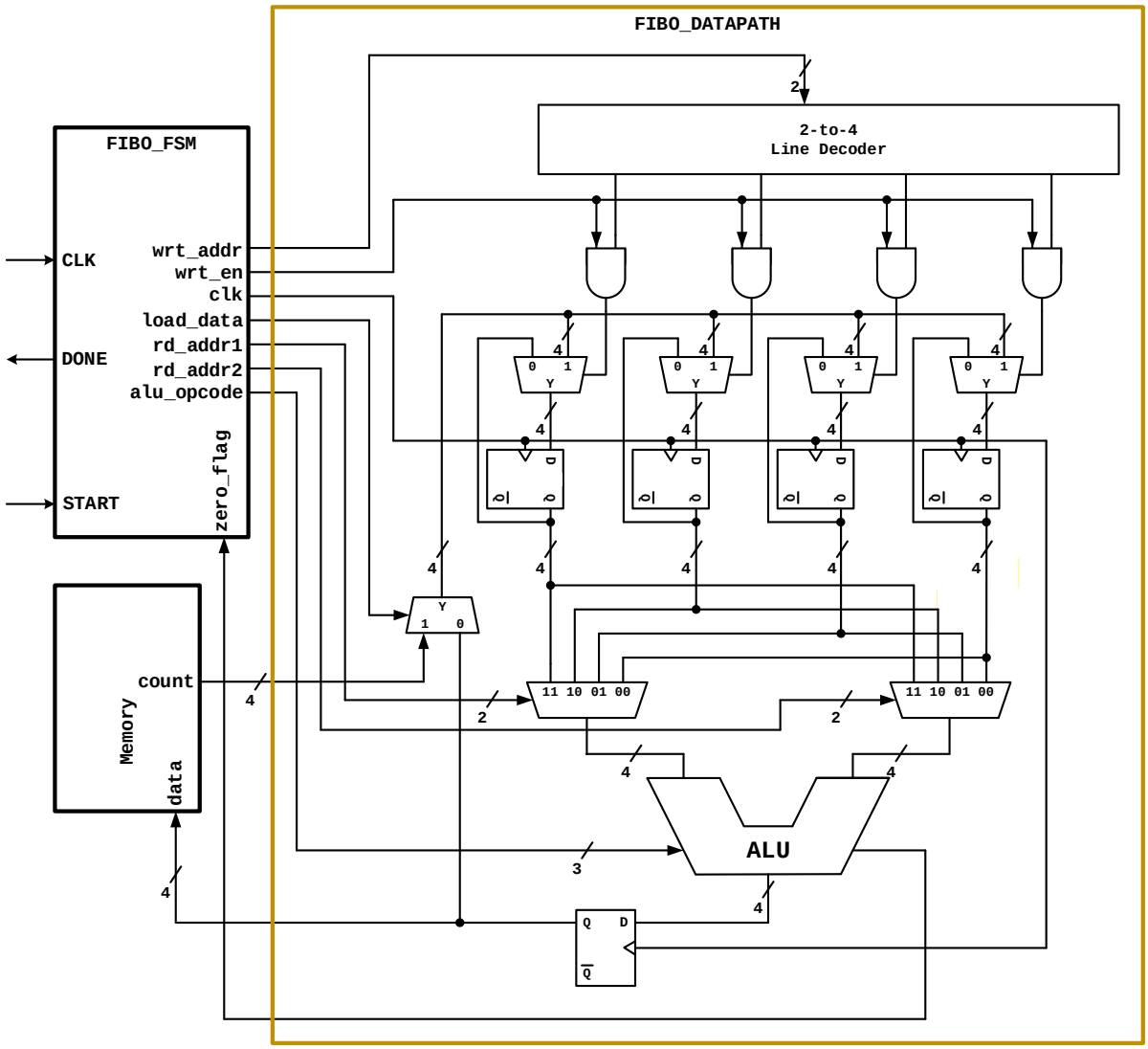


Figure 1: Fibonacci Series Calculator FSM & Datapath

Each logic block of Datapath must be design and simulated separately and imported into the top-level design. The interconnection between each logic block must be done on the top-level design **FIBO_DATAPATH** using hierarchical design approach.

This design should follow a parametric model so that the bit size of the calculator can be adjusted. Write a parameterized Verilog code to define your Datapath. The CLK, and the other control signals should come from the FSM.

The FSM control signals and Datapath operations:

- **wrt_addr** signal provides an address to activate the required register as stated in Table 1.

Table 1: 2-to-4 Decoder Operation.

wrt_addr		Registers			
A	B	REG_3	REG_2	REG_1	REG_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

- **wrt_en** works together with the **wrt_addr** to select either a new value or hold the previous value.
- **load_data** signal selects either the count stored in memory or the ALU output result.
- **rd_addr1** and **rd_addr2** signals provide the register addresses that are required to perform the **ALU** operation between two registers using the **alu_opcode** provided. The ALU must be designed with the following operations stated in Table 2. You can use the behavioral approach to design the **ALU**.

Table 2: ALU Opcodes and Instructions.

alu_opcode	Opr 1	Opr 2	Op	Instruction	
000	--	--	noop	--	--
001	xx	--	set	R[xx] =	1
010	xx	--	increment	R[xx] =	R[xx] + 1
011	xx	--	decrement	R[xx] =	R[xx] - 1
100	xx	--	load	R[xx] =	Data_in
101	xx	--	store	Data_out =	R[xx]
110	xx	yy	add	R[xx] =	R[xx] + R[yy]
111	xx	yy	copy	R[xx] =	R[yy]

6.4 FSM DESIGN (WEEK 15)

The control unit of this Datapath will be implemented by the **FIBO_FSM**. **FIBO_FSM** is divided into two main subcomponents as **FSM** and **FSM_DECO.**

The **FSM** will have four external inputs **START**, **ZERO_FLAG**, **CLK** and **RST** and one output **DONE** to indicate that the calculation is completed. This component performs all the required control operations providing three outputs as: **opcode**, **operand1** and **operand2**.

The **FSM_DECO** will use these three outputs as an input and decode the signals as stated in Table 3. The output of the **FSM_DECO** will be used to control the Fibonacci Series calculation on the Datapath as described in Figure 2.

Table 3: FSM_DECO Operations.

Op	op_code	Opr1	Opr2	ALU	Rd_addr1	Rd_addr2	wrt_addr	wrt_en	load_data
noop	000	--	--	000	--	--	--	0	-
set	001	xx	--	001	--	--	xx	1	0
inc	010	xx	--	010	xx	--	xx	1	0
dec	011	xx	--	011	xx	--	xx	1	0
load	100	xx	--	100	--	--	xx	1	1
store	101	xx	--	101	xx	--	--	0	-
add	110	xx	yy	110	xx	yy	xx	1	0
copy	111	xx	yy	111	yy	--	Xx	1	0

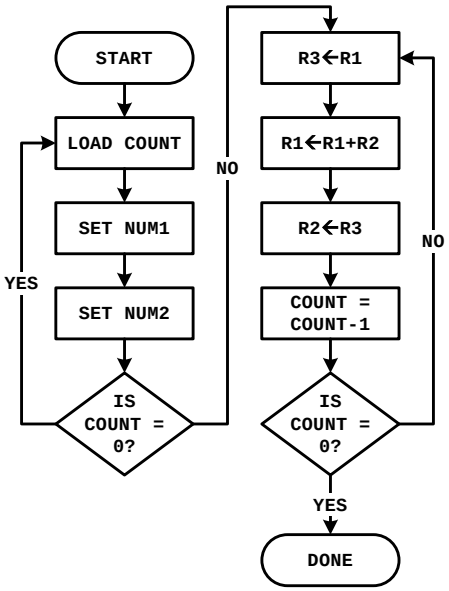


Figure 2: Fibonacci Series Hardware Algorithm.

Predefined COUNT will be stored in a memory. Once a START signal is asserted, the Fibonacci Series calculation process will start and continue until the down counter = 0.

Draw the **state diagram** for the control FSM showing all the inputs, outputs, and the state transitions clearly. Indicate the opcode required for each state.

6.5 TOP-LEVEL DESIGN

When the two core parts of Fibonacci Series calculator design is done, connect the blocks in a top-level design as depicted in Figure 1. This top-level implementation will be simulated in Modelsim® and programmed into your FPGA Board using Quartus. You will use three displays for the outputs where one of the displays on the left will show which state the FSM is in.

6.6 PROJECT REPORT (WEEK 15)

Include your name, course name, course code, date, an objective statement, and a conclusion in your report. Add short comments to briefly explain each Verilog code, schematic, and simulation that you submit. Once everything works well in simulations, check your timing analysis report to find out how fast your design is expected to work. Specify the $\mathbf{f_{max}}$ in your report and append a printed copy of the timing analysis summary to your report.

6.6.1 STATE DIAGRAM OF THE FIBO_FSM

Show your state diagram and state table of the FSM designed in section **6.3**. Show all the inputs, outputs and the state transactions clearly.

6.6.2 PARAMETERIZED VERILOG CODE

Include your Verilog codes and block schematics of the FSM, Datapath and all other components you have used in your design.

6.6.3 TESTBENCH

Write a test bench code for your Datapath to test the Fibonacci Series calculation for **7** sequences. Add your test bench code and simulation results. To debug your code, add outputs to your registers and check if there are non-desired outputs observed.

6.6.4 TIMING CONSTRAINTS

Altera place and route tools may not "try hard" to optimize the timing of your signals if you do not specify desired to do so. The compiler by default trades off design timing optimization and compilation time. This will result in long propagation delays for some signals and may even cause functional failures at low clock rates due to hold time or clock skew issues. If you see problems when testing on hardware even though your simulation works fine, try forcing the compiler to work harder on timing optimization by going to **Fitter (Place & Route) → Edit Settings** and marking "**Optimize hold timing**'' as "**All paths**". In addition, mark "**Fitter effort**" as "**Standard fit**" and change "**Desired worst-case slack**" to **5 ns**. You should not nominally have any path with a higher delay than 20ns. You can check the **Fitter Timing analysis report** after the compilation to make sure you do not have any timing violation (appears in red). If you have setup time problems, you can usually fix them by slowing down your system clock. For hold problems, or clock skew problems, the previous settings should help.

6.7 EXPERIMENTAL WORK

6.7.1 EXPERIMENTAL SETUP

- Verify to make sure your workbench has all of the following items:
- A Personal Computer (PC) with Altera Quartus II ISE 13.0 Service pack 1 Project Navigator
- DEO Demo Board with Cyclone III EP3C16F484C6 FPGA installed on a card with 10 input toggle switches, three pushbuttons, and four 7-Segment LED displays, among other components.
- A cable connected between the demo board and the PC using USB interface in order to transfer design information from the PC to the Demo Board.

6.7.2 EVALUATION

During your final demo, you will be evaluated on your understanding of:

- Combinational and sequential concepts,
- Fibonacci FSM design details,
- Verilog and schematic based design entry, and
- Simulation flows.

You will program your final design onto an FPGA and demonstrate it. If something goes wrong with the hardware demo or does not seem to work consistently using a manual clock (from push-button) or free-running on-board clock, you should **be ready** to do a quick post-route simulation of your design to demonstrate the corresponding functionality in simulation. (You can only do this if you understand your simulations very well.) Prepare various test bench waveforms in advance to be able to do this. Remember, you will have very limited time with the Teaching Assistant to demonstrate your working design, AND you fully understand your design. He/she will perform the debugging for you. Start your lab early so you can resolve any problems ahead of time during office hours, and come to the final demo fully prepared!

6.8 REFERENCES

- DE0 Board User Manual, Terasic Technologies Inc.
- Digital Design with An Introduction to the Verilog HDL, 5th Edition, M. Morris Mano & Michael D. Ciletti, Pearson
- Quartus II Introduction Using Schematic Design, ALTERA.