KUZEY KIBRIS KAMPÜSÜ
ODTÜ METU
NORTHERN CYPRUS CAMPUS

Lab 7 Report

Berke Diler

2401503

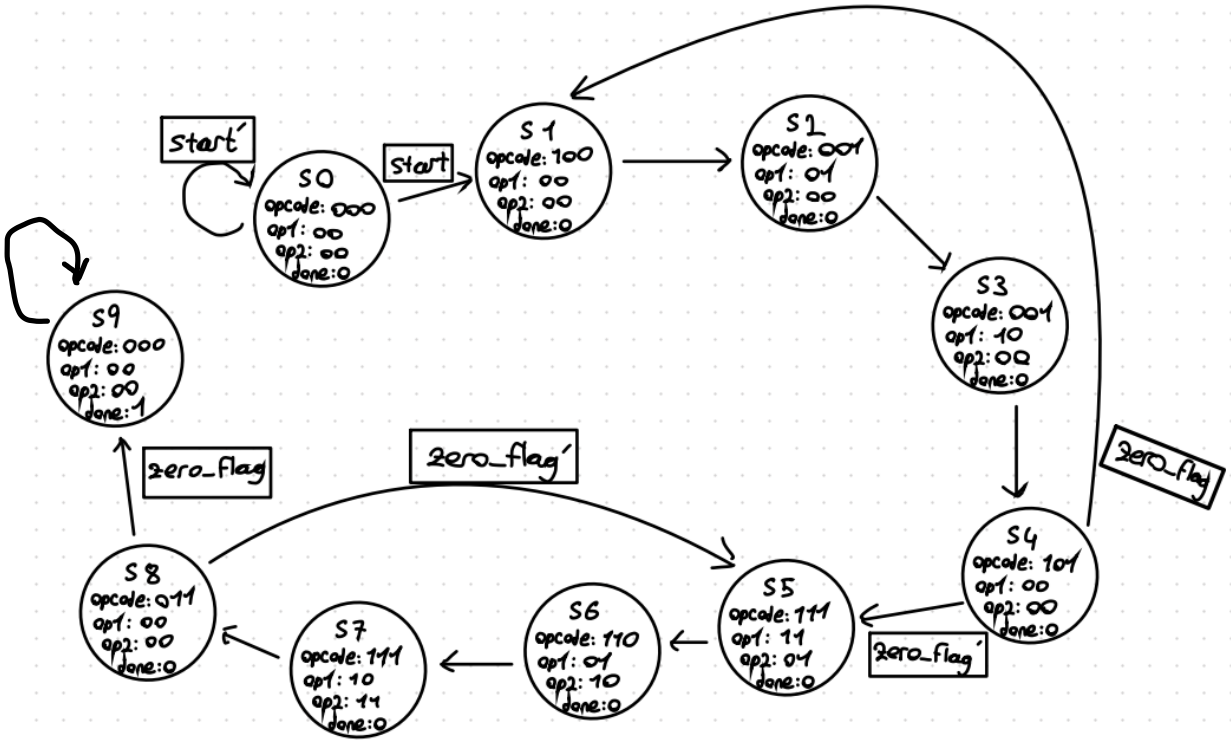
Introduction:

The goal of this project is to design a Fibonacci sequence calculator. Which will take a predefined value ‘count’ and then calculate the Fibonacci sequence until the sequence comes to the number of count. To give an example, if we define count=7. Then we will have the 7th value in the sequence which is the number 13.

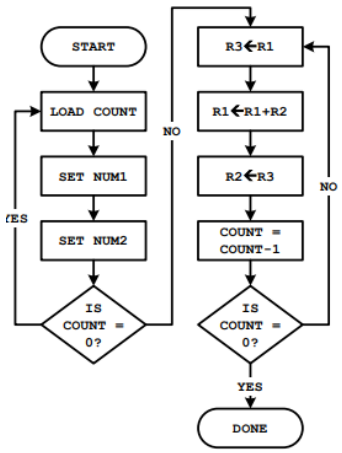
The Fibonacci sequence: 1,1,2,3,5,8,13,21,34,55,89,144,233,377,610…

Sub-components of the project are a datapath and a finite state machine. The datapath is divided into sub-components aswell which are; a 2-to-4 line decoder, four 4-bit AND gates, five 4-bit 2-to-1 multiplexers, five 4-bit registers, two 4-bit 4-to-1 Multiplexers, and a 4-bit ALU. The FSM module is divided into two modules which are; the finite state machine itself, and a decoder for the finite state machine which decodes the signals from the finite state machine and directs the signals to the datapath with proper. The project and its sub-components are designed using the Verilog programming language.

The FSM State Diagram is as follows:



The algorithm of the FSM:



Following this algorithm, the operations in the FSM and the state table is as follows,

START	ZERO_FLAG	Present State	Next State	Opcode	rd_addr1	rd_addr2	Done	Operation	
X	X	X	0000	000	00	00	0	Noop	✓
0	X	0000	0000	000	00	00	0	Noop	✓
1	X	0000	0001	000	00	00	0	Noop	✓
X	X	0001	0010	100	00	00	0	Load Count	✓
X	X	0010	0011	001	01	00	0	Set_Num1	✓
X	X	0011	0100	001	10	00	0	Set_Num2	✓
X	0	0100	0101	101	00	00	0	R0 ← Count	✓
X	1	0100	0001	101	00	00	0	Check Count	✓
X	X	0101	0110	111	11	01	0	R3←R1	✓
X	X	0110	0111	110	01	10	0	R1←R1+R2	✓
X	X	0111	1000	111	10	11	0	R2←R3	✓
X	0	1000	0101	011	00	00	0	R0←R0-1	→Cont. Calculation
X	1	1000	1001	000	00	00	0	Check Count	
X	X	1001	1001	000	00	00	1	Noop	

Verilog Codes and Explanations

Datapath Codes:

2-to-4 Line Decoder:

```
module decoder_2to4(A,out);
  input [1:0]A;
  output reg[3:0]out;

  always @(A)
  case(A)
    2'b00: out = 4'b0001;
    2'b01: out = 4'b0010;
    2'b10: out =4'b0100;
    2'b11: out = 4'b1000;

    endcase
endmodule
```

The 2-to-4 Line Decoder takes a 2-bit input which is the wrt_adder given in the lab manual schematic. After the decoder gets the input, it will activate one of its outputs at a time depending on the input. Each output of the decoder will then go through an AND gate and if the wrt_en input of the circuit is 1, a register will get the value from the output of a 2-to-1 mux which will be either the previous value of the ALU output or the count value. If the wrt_en is 0, the register will then hold its previous value.

4-bit register:

```
module Register(D,clk,Q);
parameter size = 4;
input [size-1:0] D;
input clk;
output reg[size-1:0] Q;
initial begin
  Q=0;
end
always @(posedge clk)
begin
Q <= D;//load data
end
endmodule
```

The 4-bit register's aim is to store the values of the algorithm in order to implement it. R0(00) holds the 'count' value, R1(01) stores the Fibonacci number. R2(10) is used to add numbers to R1 and R3(11) is used to copy the numbers of R2 and R1 to implement the algorithm.

4-bit 2-to-1 MUX:

```
module mux2to1(in1,in2,S,mux_out);

  parameter size =4;
  input [size-1:0]in1,in2;
  input S;
  output [size-1:0]mux_out;
  reg [size-1:0]mux_out;

  always@(S,in1,in2)
  if(S == 0)
    begin
      mux_out=in1;
    end
  else
    begin
      mux_out=in2;
    end


endmodule
```

There are four 4-bit 2-to-1 MUXes in the circuit which chooses between loading from the memory state to the registers or holding the previous value in the registers. There is an another MUX which controls the memory state which chooses either the value of ‘count’ or the ALU output and sends the output to the other MUXes.

4-bit 4-to-1 MUX:

```
module mux4to1(in1,in2,in3,in4,S,mux_out);

  parameter size =4;
  input [size-1:0] in1,in2,in3,in4;
  input [1:0] S;
  output [size-1:0]mux_out;
  reg [size-1:0] mux_out;

  always@(in1,in2,in3,in4,S)
  begin
    if(S[0]==0 && S[1]==0)
      begin
        mux_out=in1;
      end
    else if(S[0]==0 && S[1]==1)
      begin
        mux_out=in3;
      end
    else if(S[0]==1 && S[1]==0)
      begin
        mux_out=in2;
      end
    else
      begin
        mux_out=in4;
      end
    end
  endmodule
```

There are two 4-to-1 MUXes in the circuit which both has the same inputs. These MUXes will get their inputs from the registers and choose the value according to the rd_addr1 and rd_addr2 inputs which are the select inputs. Then after the registers are chosen, the two outputs will be sent to the ALU as inputs in order to implement the operations.

ALU:

```
module ALU(reg1,reg2,alu_opcode,ALU_out,zero_flag);
parameter size = 4;
input [size-1:0] reg1,reg2;
input [2:0] alu_opcode;
output [size-1:0] ALU_out;
reg [size-1:0] ALU_out;
output zero_flag;
reg zero_flag;
reg [size-1:0] temp;
always @(*)
begin
if (alu_opcode==3'b001)
temp=4'b0001;//set
else if (alu_opcode==3'b010)
temp=reg1+1;//increment
else if (alu_opcode==3'b011)
temp=reg1-1;//decrement
else if (alu_opcode==3'b100)
temp=temp;//load
else if (alu_opcode==3'b101)
temp=reg1;//store
else if (alu_opcode==3'b110)
temp=reg1+reg2;//add
else if (alu_opcode==3'b111)
temp=reg1;//copy
else
temp=temp;//noop
end
always @(*)
begin
ALU_out=temp;
zero_flag = ~(|temp);//check result is 0 or not
end

endmodule
```

ALU is the brain and the essential part of the circuit, does all the computations based on an operation code sent to it and then produces the wanted output.

Datapath:

```
module datapath(wrt_addr,rd_addr1,rd_addr2,wrt_en,clk,load_data,alu_opcode,zero_flag,data,count);
  parameter size=4;
  input [2:0]alu_opcode;
  input [1:0]wrt_addr,rd_addr1,rd_addr2;
  input [size-1:0]count;
  input wrt_en,clk,load_data;
  output [size-1:0]data;
  output zero_flag;
  wire [3:0]out;
  wire W0,W1,W2,W3,S0,S1,S2,S3;
  wire [size-1:0]Q0,Q1,Q2,Q3,Q4,D0,D1,D2,D3,D4,reg1,reg2;
  decoder_2to4 decoder(wrt_addr,out); //decoder
  assign W0 = out[0];
  assign W1 = out[1];
  assign W2 = out[2];
  assign W3 = out[3];
  //and gates
  assign S0=W0&wrt_en;
  assign S1=W1&wrt_en;
  assign S2=W2&wrt_en;
  assign S3=W3&wrt_en;
  //2tol muxes
  mux2tol Mux2tol_0(Q0,D4,S0,D0);
  mux2tol Mux2tol_1(Q1,D4,S1,D1);
  mux2tol Mux2tol_2(Q2,D4,S2,D2);
  mux2tol Mux2tol_3(Q3,D4,S3,D3);
  mux2tol Mux2tol_4(Q4,count,load_data,D4);
  //registers
  Register R0(D0,clk,Q0);
  Register R1(D1,clk,Q1);
  Register R2(D2,clk,Q2);
  Register R3(D3,clk,Q3);
  //4tol muxes
  mux4tol Mux4tol_1(Q0,Q1,Q2,Q3,rd_addr1,reg1);
  mux4tol Mux4tol_2(Q0,Q1,Q2,Q3,rd_addr2,reg2);
  //alu
  ALU alu(reg1,reg2,alu_opcode,Q4,zero_flag);
  assign data=Q1;
endmodule
```

All the other components of the circuit are gathered together in the datapath to get a reasonable output. The datapath receives signals from the FSM to do the operations. And it sends signals to the FSM in order to get a signal to continue computing or to stop.

FSM Codes:

FSM:

```
module FSM(start,zero_flag,clk,done,opcode,op1,op2);
  input zero_flag;
  input clk,start;
  reg [3:0]state,nextstate;
  output reg [2:0]opcode;
  output reg [1:0]op1,op2;
  output reg done;
  parameter S0=4'b0000,S1=4'b0001,S2=4'b0010,S3=4'b0011,S4=4'b0100,S5=4'b0101,S6=4'b0110,S7=4'b0111,S8=4'b1000,S9=4'b1001;
  always @(posedge clk or posedge start)
  if(start)
    state<=nextstate; //fsm will keep continuing as long as start is 1
  else
    state<=S0; //if start input is 0, the fsm will begin from state 0

    always @(state or zero_flag or start)
    begin
      op1=2'b00;op2=2'b00;opcode=3'b000;done=0;nextstate=S0;//inital state
      case(state)
        S0: //reset state
        if(start)
          begin
            opcode=3'b000;
            op1=2'b00;
            op2=2'b00;
            done=0;
            nextstate=S1;
          end
        else
          nextstate=S0;

          S1:  //load count value
          begin
            opcode=3'b100;
            op1=2'b00;
            op2=2'b00;
            done=0;
            nextstate=S2;
          end
          S2:  //set num1
          begin
            opcode=3'b001;
            op1=2'b01;
            op2=2'b00;
            done=0;
            nextstate=S3;
          end
          S3: //set num2
          begin
            opcode=3'b001;
            op1=2'b10;
            op2=2'b00;
            done=0;
            nextstate=S4;
          end
          S4: //check count
          begin
            opcode=3'b101;
            op1=2'b00;
            op2=2'b00;
            done=1'b0;
            if(zero_flag)
              nextstate=S1; //if count is 0 load new count
            else
              nextstate=S5; //if count is not 0, continue calculating
            end
           S5: // R3<-R1
           begin
              opcode=3'b111; //copy
              op1=2'b11; //R3
              op2=2'b01; //R1
              done=0;
              nextstate=S6;
            end
            S6: //R1<-R1+R2
            begin
              opcode=3'b110; //add
              op1=2'b01;
              op2=2'b10;
              done=1'b0;
              nextstate=S7;
            end
            S7: //R2<-R3
            begin
              opcode=3'b111;
              op1=2'b10;
              op2=2'b11;
              done=1'b0;
              nextstate=S8;
            end
```

```
            S8: //count<-count-1
            begin
              opcode=3'b011;
              op1=2'b00;
              op2=2'b00;
              done=1'b0;
              if(zero_flag)
                nextstate=4'b1001;
              else
                nextstate=4'b0101;
              end
              S9: //last state, does no operation, sets done=1
              begin
                opcode=3'b000;
                op1=2'b00;
                op2=2'b00;
                done=1'b1;
                nextstate=4'b1001;
              end
            endcase
          end
        endmodule
```

The FSM is actually an abstract machine which is responsible of deciding on operations, starting and stopping the calculations in the circuit. Each state in the FSM is actually a step in the algorithm. By starting this algorithm, we get the desired result.

FSM_DECO:

```
module FSM_DECO(opcode,op1,op2,alu_opcode,rd_addr1,rd_addr2,wrt_addr,wrt_en,load_data);
  input[2:0]opcode;
  input[1:0]op1,op2;
  output reg [1:0]wrt_addr,rd_addr1,rd_addr2;
  output reg load_data,wrt_en;
  output reg[2:0]alu_opcode;
  always @(opcode,op1,op2)
  begin
    if(opcode==3'b000)//no operation
    begin
      alu_opcode=3'b000;
      rd_addr1=2'b00;
      rd_addr2=2'b00;
      wrt_addr=2'b00;
      wrt_en=1'b0;
      load_data=1'b0;
    end
  else if(opcode==3'b001)//set
    begin
      alu_opcode=3'b001;
      rd_addr1=2'b00;
      rd_addr2=2'b00;
      wrt_addr=op1;
      wrt_en=1'b1;
      load_data=1'b0;
    end
  else if(opcode==3'b010)//increment
      begin
      alu_opcode=3'b010;
      rd_addr1=op1;
      rd_addr2=2'b00;
      wrt_addr=op1;
      wrt_en=1'b1;
      load_data=1'b0;
    end
  else if(opcode==3'b011) //decrement
    begin
      alu_opcode=3'b011;
      rd_addr1=op1;
      rd_addr2=2'b00;
      wrt_addr=op1;
      wrt_en=1'b1;
      load_data=1'b0;
    end
```

```
  else if(opcode==3'b100) //load
    begin
      alu_opcode=3'b100;
      rd_addr1=2'b00;
      rd_addr2=2'b00;
      wrt_addr=op1;
      wrt_en=1'b1;
      load_data=1'b1;
    end
  else if(opcode==3'b101) //store
    begin
      alu_opcode=3'b101;
      rd_addr1=op1;
      rd_addr2=2'b00;
      wrt_addr=2'b00;
      wrt_en=1'b0;
      load_data=1'b0;
    end
  else if(opcode==3'b110) //add
    begin
      alu_opcode=3'b110;
      rd_addr1=op1;
      rd_addr2=op2;
      wrt_addr=op1;
      wrt_en=1'b1;
      load_data=1'b0;
    end
  else  //copy
    begin
      alu_opcode=3'b111;
      rd_addr1=op2;
      rd_addr2=2'b00;
      wrt_addr=op1;
      wrt_en=1'b1;
      load_data=1'b0;
    end
  end
endmodule
```

FSM_DECO is like the translator of the circuit. It decodes the signals received from the FSM and sends the appropriate signals to the datapath to do the desired operations.

FIBO_FSM:

```
module FIBO_FSM(start,zero_flag,clk,alu_opcode,rd_addr1,rd_addr2,wrt_addr,wrt_en,load_data,done);
  input clk,start,zero_flag;
  output wrt_en,done,load_data;
  output[1:0]wrt_addr,rd_addr1,rd_addr2;
  output[2:0]alu_opcode;
  wire [2:0]opcode;
  wire [1:0]op1,op2;
  FSM fsm(start,zero_flag,clk,done,opcode,op1,op2);
  FSM_DECO deco(opcode,op1,op2,alu_opcode,rd_addr1,rd_addr2,wrt_addr,wrt_en,load_data);
endmodule
```

FIBO_FSM connects the FSM to FSM_DECO in order to send signals to the datapath.

Top_level:

```
module top_level(start,clk,done,count,data);
  input start,clk;
  input [3:0]count;
  output done;
  output [3:0]data;
  wire zero_flag;
  wire [2:0]alu_opcode;
  wire [1:0]rd_addr1,rd_addr2,wrt_addr;
  wire wrt_en,load_data;
  wire [3:0]temp;
  assign temp=count-2;
  FIBO_FSM FSM(start,zero_flag,clk,alu_opcode,rd_addr1,rd_addr2,wrt_addr,wrt_en,load_data,done);
  datapath DATAPATH(wrt_addr,rd_addr1,rd_addr2,wrt_en,clk,load_data,alu_opcode,zero_flag,data,temp);
endmodule
```

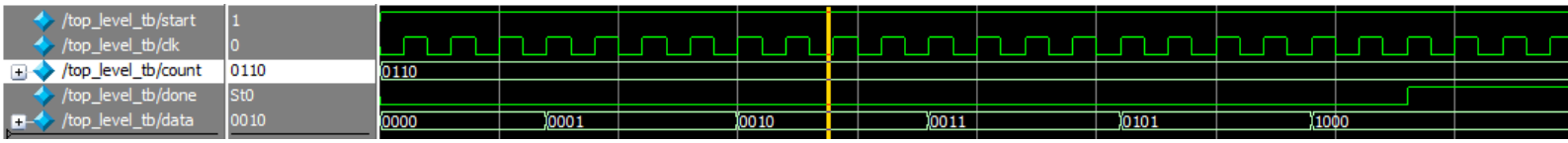
The last step in the hierarchical design process. Connects the FSM to the Datapath to calculate the Fibonacci sequence.

Testbench

```
module top_level_tb;
  reg start,clk;
  reg [3:0]count;
  wire done;
  wire [3:0]data;
  top_level fibo(start,clk,done,count,data);
  always
  #10 clk=~clk;
  initial
  begin
    clk=0;start=1;count=4'b0110;

  end
endmodule
```

Simulation



Conclusion:

After implementing all the components together in the top level design, we can observe that the output of the simulation is correct. The count value is 6 and the output is supposed to be the 6th Fibonacci number in the sequence which is 8. We can observe that the output is indeed 8 and the ‘done’ output is 1 after the operation is done. By finishing this experiment I can say that I am confident with my set of skills in combinational and sequential logic circuit designing concepts.