CS223 Lab Project

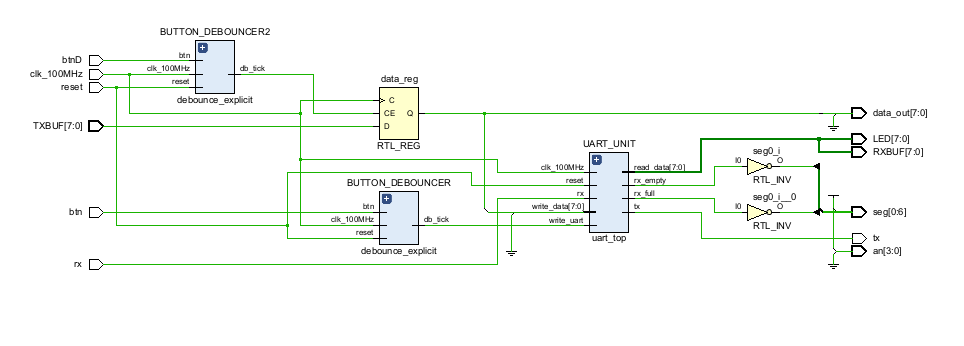
Altay İlker Yiğitel

CS223 Sec: 01

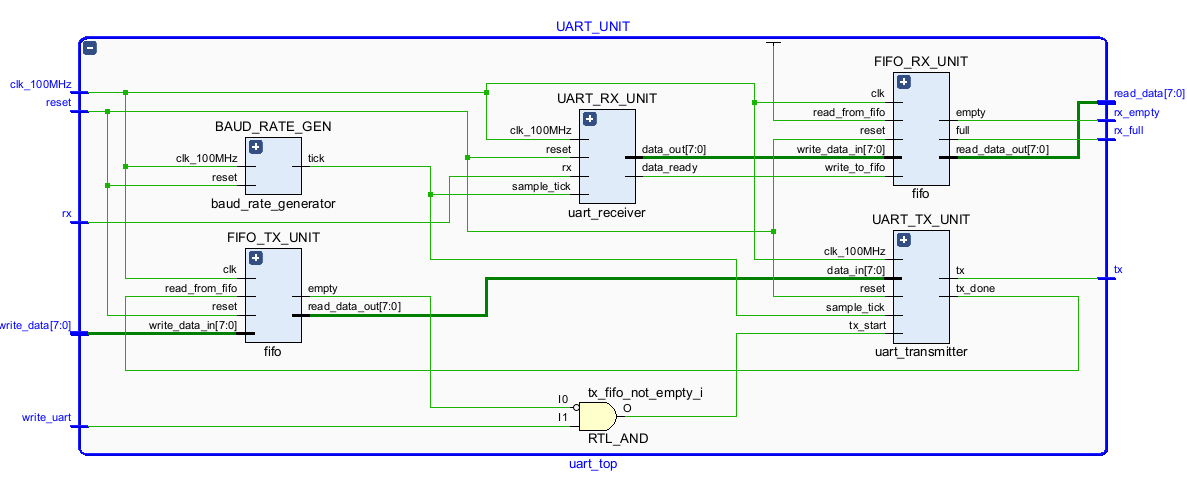
22203024

I have implemented the uart using main modules and connecting modules which are baud controller uart receiver transmitter debouncers seg7 displays and fifos connecting rx’s and tx’s. UART allows me to send and receive data from and to another device. It sends and receives data in bytes. There are 1 stop bit, 1 start bit and 1 parity bit.

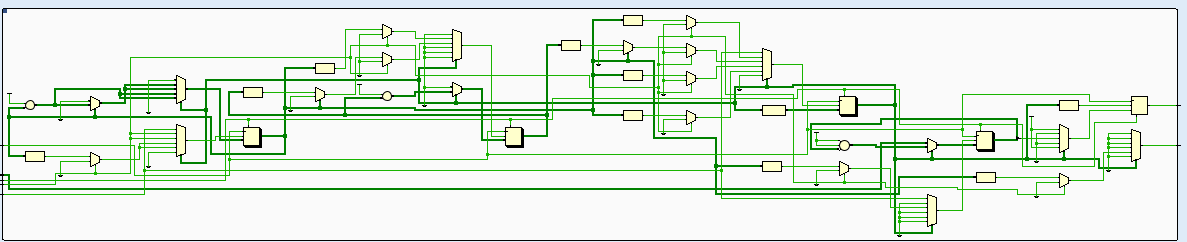
Uart top



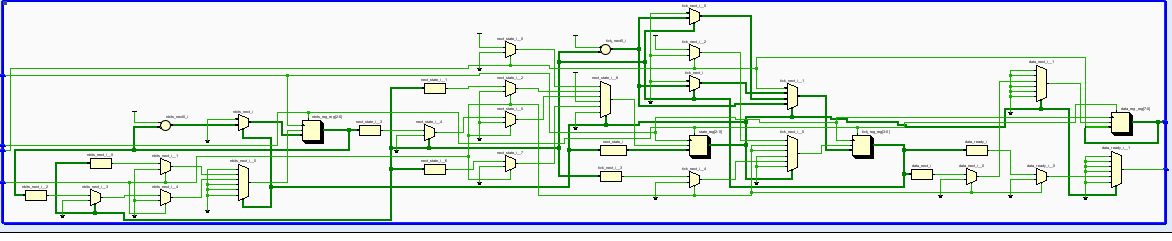
Uart

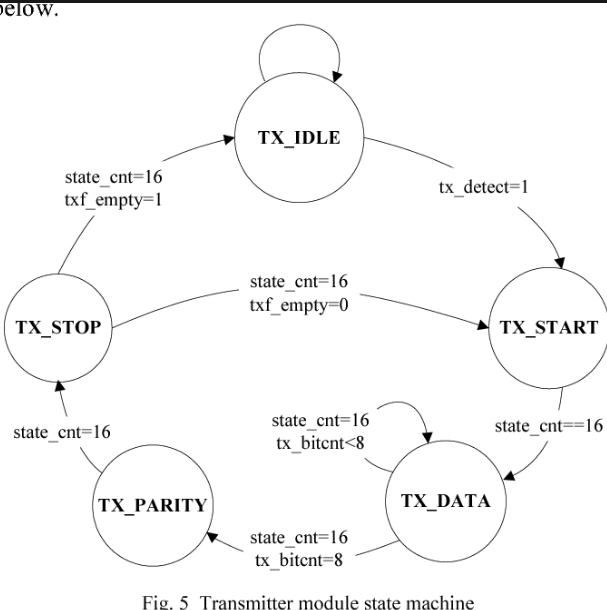


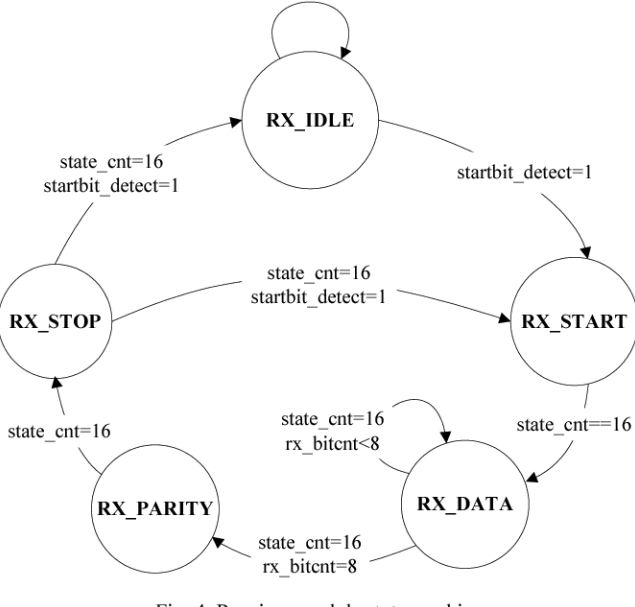
TX

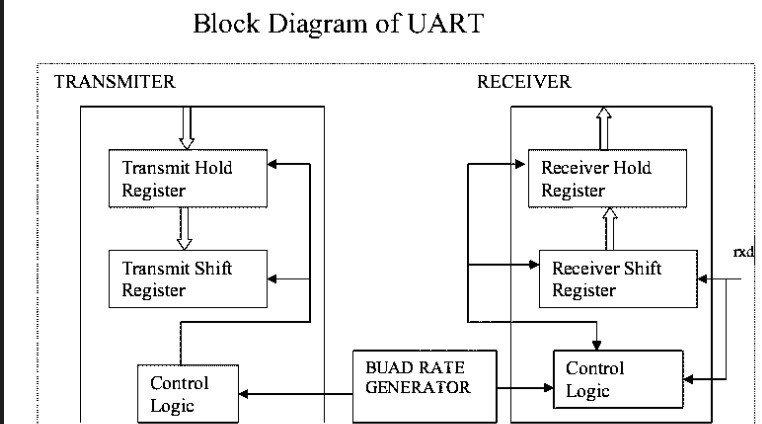


RX









module uart\_test(

input clk\_100MHz, // basys 3 FPGA clock signal

input reset, // btnR

input rx, // USB-RS232 Rx

input btn, // btnL (read and write FIFO operation)

input btnD,

input btnU,

input btnL,

input [7:0] TXBUF,

output tx, // USB-RS232 Tx

output [3:0] an, // 7 segment display digits

output [0:6] seg, // 7 segment display segments

output [7:0] LED, // data byte display

output [7:0] data\_out,

output reg [7:0] RXBUF

);

bit [7:0] hxxt[4];

bit [7:0] hxxr[4];

reg lastr = 0;

reg data;

reg [3:0] sec\_ones = 0;

reg [3:0] sec\_tens = 0;

reg [3:0] sec\_hundereds = 0;

reg [3:0] sec\_thousands = 0;

// Connection Signals

wire rx\_full, rx\_empty, btn\_tick,btn\_tick2,btn\_tick3,btn\_tick4;

wire [7:0] rec\_data, rec\_data1;

initial begin

for(int i = 0;i<$size(hxxt);i++)begin

hxxt[i] = 0;

hxxr[i] = 0;

end

end

always @(posedge clk\_100MHz)

if(btn\_tick2)

data = TXBUF;

always@(posedge clk\_100MHz)

if(hxxt[0] == 0 && ~rx)begin

for(int i = $size(hxxt)-2;i>=0;i--)begin

hxxt[i] = hxxt[i+1];

end

hxxt[0] = TXBUF;

end

always@(posedge clk\_100MHz)

if(hxxr[0] == 0 && btn\_tick2)begin

for(int i = $size(hxxt)-2;i>=0;i--)begin

hxxr[i] = hxxr[i+1];

end

hxxr[0] = RXBUF;

if(lastr<3)

lastr++;

end

always@(posedge clk\_100MHz)

if(btn\_tick3) begin

if(sec\_thousands<1)

sec\_thousands++;

else

sec\_thousands = 0;

end

always@(posedge clk\_100MHz)

if(btn\_tick4) begin

if(sec\_hundereds<3)

sec\_hundereds++;

else

sec\_hundereds = 0;

end

always@(posedge clk\_100MHz)

if(sec\_thousands == 0)begin

sec\_tens = hxxt[sec\_hundereds] [3:0];

sec\_ones = hxxt[sec\_hundereds] [7:4];

end

else begin

sec\_tens = hxxr[sec\_hundereds] [3:0];

sec\_ones = hxxr[sec\_hundereds] [7:4];

end

// Complete UART Core

uart\_top UART\_UNIT

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

//.read\_uart(btn\_tick),

.write\_uart(btn\_tick),

.rx(rx),

.write\_data(data), //rec\_data1

.rx\_full(rx\_full),

.rx\_empty(rx\_empty),

.read\_data(rec\_data),

.tx(tx)

);

// Button Debouncer

debounce\_explicit BUTTON\_DEBOUNCER

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.btn(btn),

.db\_level(),

.db\_tick(btn\_tick)

);

debounce\_explicit BUTTON\_DEBOUNCER2

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.btn(btnD),

.db\_level(),

.db\_tick(btn\_tick2)

);

debounce\_explicit BUTTON\_DEBOUNCER3

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.btn(btnU),

.db\_level(),

.db\_tick(btn\_tick3)

);

debounce\_explicit BUTTON\_DEBOUNCER4

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.btn(btnL),

.db\_level(),

.db\_tick(btn\_tick4)

);

seg7\_control seg7(.clk\_100MHz(clk\_100MHz), .reset(res), .ones(sec\_ones), .tens(sec\_tens), .hundereds(sec\_hundereds),

.thousands(sec\_thousands), .seg(seg), .an(an));

// Signal Logic

assign rec\_data1 = rec\_data + 1; // add 1 to ascii value of received data (to transmit)

assign RXBUF = rec\_data;

// Output Logic

assign LED = hxxr[lastr]; // data byte received displayed on LEDs

assign data\_out = hxxt[0];

Endmodule

module uart\_top

#(

parameter DBITS = 8, // number of data bits in a word

SB\_TICK = 16, // number of stop bit / oversampling ticks

BR\_LIMIT = 52, // baud rate generator counter limit

BR\_BITS = 6, // number of baud rate generator counter bits

FIFO\_EXP = 2 // exponent for number of FIFO addresses (2^2 = 4)

)

(

input clk\_100MHz, // FPGA clock

input reset, // reset

//input read\_uart, // button

input write\_uart, // button

input rx, // serial data in

input [DBITS-1:0] write\_data, // data from Tx FIFO

output rx\_full, // do not write data to FIFO

output rx\_empty, // no data to read from FIFO

output tx, // serial data out

output [DBITS-1:0] read\_data // data to Rx FIFO

);

// Connection Signals

wire tick; // sample tick from baud rate generator

wire rx\_done\_tick; // data word received

wire tx\_done\_tick; // data transmission complete

wire tx\_empty; // Tx FIFO has no data to transmit

wire tx\_fifo\_not\_empty; // Tx FIFO contains data to transmit

wire [DBITS-1:0] tx\_fifo\_out; // from Tx FIFO to UART transmitter

wire [DBITS-1:0] rx\_data\_out; // from UART receiver to Rx FIFO

// Instantiate Modules for UART Core

baud\_rate\_generator

#(

.M(BR\_LIMIT),

.N(BR\_BITS)

)

BAUD\_RATE\_GEN

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.tick(tick)

);

uart\_receiver

#(

.DBITS(DBITS),

.SB\_TICK(SB\_TICK)

)

UART\_RX\_UNIT

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.rx(rx),

.sample\_tick(tick),

.data\_ready(rx\_done\_tick),

.data\_out(rx\_data\_out)

);

uart\_transmitter

#(

.DBITS(DBITS),

.SB\_TICK(SB\_TICK)

)

UART\_TX\_UNIT

(

.clk\_100MHz(clk\_100MHz),

.reset(reset),

.tx\_start(tx\_fifo\_not\_empty),

.sample\_tick(tick),

.data\_in(tx\_fifo\_out),

.tx\_done(tx\_done\_tick),

.tx(tx)

);

fifo

#(

.DATA\_SIZE(DBITS),

.ADDR\_SPACE\_EXP(FIFO\_EXP)

)

FIFO\_RX\_UNIT

(

.clk(clk\_100MHz),

.reset(reset),

.write\_to\_fifo(rx\_done\_tick),

.read\_from\_fifo(1'b1), //read\_uart

.write\_data\_in(rx\_data\_out),

.read\_data\_out(read\_data),

.empty(rx\_empty),

.full(rx\_full)

);

fifo

#(

.DATA\_SIZE(DBITS),

.ADDR\_SPACE\_EXP(FIFO\_EXP)

)

FIFO\_TX\_UNIT

(

.clk(clk\_100MHz),

.reset(reset),

.write\_to\_fifo(write\_uart2),

.read\_from\_fifo(tx\_done\_tick),

.write\_data\_in(write\_data),

.read\_data\_out(tx\_fifo\_out),

.empty(tx\_empty),

.full() // intentionally disconnected

);

// debounce\_explicit debounce1(

// .clk\_100MHz(clk\_100MHz),

// .reset(reset),

// .btn(write\_uart),

// .db\_level(),

// .db\_tick(write\_uart2)

// );

// Signal Logic

assign tx\_fifo\_not\_empty = ~tx\_empty && write\_uart;

Endmodule

module baud\_rate\_generator

#( // 115200 baud

parameter N = 6, // number of counter bits

M = 52 // counter limit value

)

(

input clk\_100MHz, // basys 3 clock

input reset, // reset

output tick // sample tick

);

// Counter Register

reg [N-1:0] counter; // counter value

wire [N-1:0] next; // next counter value

// Register Logic

always @(posedge clk\_100MHz, posedge reset)

if(reset)

counter <= 0;

else

counter <= next;

// Next Counter Value Logic

assign next = (counter == (M-1)) ? 0 : counter + 1;

// Output Logic

assign tick = (counter == (M-1)) ? 1'b1 : 1'b0;

Endmodule

module uart\_receiver

#(

parameter DBITS = 8, // number of data bits in a data word

SB\_TICK = 16 // number of stop bit / oversampling ticks (1 stop bit)

)

(

input clk\_100MHz, // basys 3 FPGA

input reset, // reset

input rx, // receiver data line

input sample\_tick, // sample tick from baud rate generator

output reg data\_ready, // signal when new data word is complete (received)

output [DBITS-1:0] data\_out // data to FIFO

);

// State Machine States

localparam [2:0] idle = 3'b000,

start = 3'b001,

data = 3'b010,

stop = 3'b011,

parity = 3'b100;

// Registers

reg [2:0] state, next\_state; // state registers

reg [3:0] tick\_reg, tick\_next; // number of ticks received from baud rate generator

reg [2:0] nbits\_reg, nbits\_next; // number of bits received in data state

reg [7:0] data\_reg, data\_next; // reassembled data word

// Register Logic

always @(posedge clk\_100MHz, posedge reset)

if(reset) begin

state <= idle;

tick\_reg <= 0;

nbits\_reg <= 0;

data\_reg <= 0;

end

else begin

state <= next\_state;

tick\_reg <= tick\_next;

nbits\_reg <= nbits\_next;

data\_reg <= data\_next;

end

// State Machine Logic

always @\* begin

next\_state = state;

data\_ready = 1'b0;

tick\_next = tick\_reg;

nbits\_next = nbits\_reg;

data\_next = data\_reg;

case(state)

idle:

if(~rx) begin // when data line goes LOW (start condition)

next\_state = start;

tick\_next = 0;

end

start:

if(sample\_tick)

if(tick\_reg == 7) begin

next\_state = data;

tick\_next = 0;

nbits\_next = 0;

end

else

tick\_next = tick\_reg + 1;

data:

if(sample\_tick)

if(tick\_reg == 15) begin

tick\_next = 0;

data\_next = {rx, data\_reg[7:1]};

if(nbits\_reg == (DBITS-1))

next\_state = stop;

else

nbits\_next = nbits\_reg + 1;

end

else

tick\_next = tick\_reg + 1;

parity: begin

;

next\_state = stop;

end

stop:

if(sample\_tick)

if(tick\_reg == (SB\_TICK-1)) begin

next\_state = idle;

data\_ready = 1'b1;

end

else

tick\_next = tick\_reg + 1;

endcase

end

// Output Logic

assign data\_out = data\_reg;

Endmodule

module uart\_transmitter

#(

parameter DBITS = 8, // number of data bits

SB\_TICK = 16 // number of stop bit / oversampling ticks (1 stop bit)

)

(

input clk\_100MHz, // basys 3 FPGA

input reset, // reset

input tx\_start, // begin data transmission (FIFO NOT empty)

input sample\_tick, // from baud rate generator

input [DBITS-1:0] data\_in, // data word from FIFO

output reg tx\_done, // end of transmission

output tx // transmitter data line

);

// State Machine States

localparam [2:0] idle = 3'b000,

start = 3'b001,

data = 3'b010,

stop = 3'b011,

parity = 3'b100;

// Registers

reg [2:0] state, next\_state; // state registers

reg [3:0] tick\_reg, tick\_next; // number of ticks received from baud rate generator

reg [2:0] nbits\_reg, nbits\_next; // number of bits transmitted in data state

reg [DBITS-1:0] data\_reg, data\_next; // assembled data word to transmit serially

reg tx\_reg, tx\_next; // data filter for potential glitches

// Register Logic

always @(posedge clk\_100MHz, posedge reset)

if(reset) begin

state <= idle;

tick\_reg <= 0;

nbits\_reg <= 0;

data\_reg <= 0;

tx\_reg <= 1'b1;

end

else begin

state <= next\_state;

tick\_reg <= tick\_next;

nbits\_reg <= nbits\_next;

data\_reg <= data\_next;

tx\_reg <= tx\_next;

end

// State Machine Logic

always @\* begin

next\_state = state;

tx\_done = 1'b0;

tick\_next = tick\_reg;

nbits\_next = nbits\_reg;

data\_next = data\_reg;

tx\_next = tx\_reg;

case(state)

idle: begin // no data in FIFO

tx\_next = 1'b1; // transmit idle

if(tx\_start) begin // when FIFO is NOT empty

next\_state = start;

tick\_next = 0;

data\_next = data\_in;

end

end

start: begin

tx\_next = 1'b0; // start bit

if(sample\_tick)

if(tick\_reg == 15) begin

next\_state = data;

tick\_next = 0;

nbits\_next = 0;

end

else

tick\_next = tick\_reg + 1;

end

data: begin

tx\_next = data\_reg[0];

if(sample\_tick)

if(tick\_reg == 15) begin

tick\_next = 0;

data\_next = data\_reg >> 1;

if(nbits\_reg == (DBITS-1))

next\_state = parity;

else

nbits\_next = nbits\_reg + 1;

end

else

tick\_next = tick\_reg + 1;

end

parity: begin

tx\_next = 1'b0;

if(sample\_tick)

if(tick\_reg == (SB\_TICK-1)) begin

next\_state = stop;

end

else

tick\_next = tick\_reg + 1;

end

stop: begin

tx\_next = 1'b1; // back to idle

if(sample\_tick)

if(tick\_reg == (SB\_TICK-1)) begin

next\_state = idle;

tx\_done = 1'b1;

end

else

tick\_next = tick\_reg + 1;

end

endcase

end

// Output Logic

assign tx = tx\_reg;

Endmodule

module fifo

#(

parameter DATA\_SIZE = 8, // number of bits in a data word

ADDR\_SPACE\_EXP = 4 // number of address bits (2^4 = 16 addresses)

)

(

input clk, // FPGA clock

input reset, // reset button

input write\_to\_fifo, // signal start writing to FIFO

input read\_from\_fifo, // signal start reading from FIFO

input [DATA\_SIZE-1:0] write\_data\_in, // data word into FIFO

output [DATA\_SIZE-1:0] read\_data\_out, // data word out of FIFO

output empty, // FIFO is empty (no read)

output full // FIFO is full (no write)

);

// signal declaration

reg [DATA\_SIZE-1:0] memory [2\*\*ADDR\_SPACE\_EXP-1:0]; // memory array register

reg [ADDR\_SPACE\_EXP-1:0] current\_write\_addr, current\_write\_addr\_buff, next\_write\_addr;

reg [ADDR\_SPACE\_EXP-1:0] current\_read\_addr, current\_read\_addr\_buff, next\_read\_addr;

reg fifo\_full, fifo\_empty, full\_buff, empty\_buff;

wire write\_enabled;

// register file (memory) write operation

always @(posedge clk)

if(write\_enabled)

memory[current\_write\_addr] <= write\_data\_in;

// register file (memory) read operation

assign read\_data\_out = memory[current\_read\_addr];

// only allow write operation when FIFO is NOT full

assign write\_enabled = write\_to\_fifo & ~fifo\_full;

// FIFO control logic

// register logic

always @(posedge clk or posedge reset)

if(reset) begin

current\_write\_addr <= 0;

current\_read\_addr <= 0;

fifo\_full <= 1'b0;

fifo\_empty <= 1'b1; // FIFO is empty after reset

end

else begin

current\_write\_addr <= current\_write\_addr\_buff;

current\_read\_addr <= current\_read\_addr\_buff;

fifo\_full <= full\_buff;

fifo\_empty <= empty\_buff;

end

// next state logic for read and write address pointers

always @\* begin

// successive pointer values

next\_write\_addr = current\_write\_addr + 1;

next\_read\_addr = current\_read\_addr + 1;

// default: keep old values

current\_write\_addr\_buff = current\_write\_addr;

current\_read\_addr\_buff = current\_read\_addr;

full\_buff = fifo\_full;

empty\_buff = fifo\_empty;

// Button press logic

case({write\_to\_fifo, 1'b1}) // check both buttons

// 2'b00: neither buttons pressed, do nothing

2'b01: // read button pressed?

if(~fifo\_empty) begin // FIFO not empty

current\_read\_addr\_buff = next\_read\_addr;

full\_buff = 1'b0; // after read, FIFO not full anymore

if(next\_read\_addr == current\_write\_addr)

empty\_buff = 1'b1;

end

2'b10: // write button pressed?

if(~fifo\_full) begin // FIFO not full

current\_write\_addr\_buff = next\_write\_addr;

empty\_buff = 1'b0; // after write, FIFO not empty anymore

if(next\_write\_addr == current\_read\_addr)

full\_buff = 1'b1;

end

2'b11: begin // write and read

current\_write\_addr\_buff = next\_write\_addr;

current\_read\_addr\_buff = next\_read\_addr;

end

endcase

end

// output

assign full = fifo\_full;

assign empty = fifo\_empty;

Endmodule

module debounce\_explicit(

input clk\_100MHz,

input reset,

input btn, // button input

output reg db\_level, // for switches

output reg db\_tick // for buttons

);

// state declarations

parameter [1:0] zero = 2'b00,

wait0 = 2'b01,

one = 2'b10,

wait1 = 2'b11;

// Artix-7 has a 100MHz clk with a period of 10ns

// number of counter bits (2^N \* 10ns = ~40ms)

parameter N = 22;

// signal declaration

reg [1:0] state\_reg, next\_state;

reg [N-1:0] q\_reg;

wire [N-1:0] q\_next;

wire q\_zero;

reg q\_load, q\_dec;

// body

// FSMD state and data registers

always @(posedge clk\_100MHz or posedge reset)

if(reset) begin

state\_reg <= zero;

q\_reg <= 0;

end

else begin

state\_reg <= next\_state;

q\_reg <= q\_next;

end

// FSMD data path (counter) next state logic

assign q\_next = (q\_load) ? {N{1'b1}} : // load all 1s

(q\_dec) ? q\_reg - 1 : // decrement

q\_reg; // no change in q

// status signal

assign q\_zero = (q\_next == 0);

// FSMD control path next state logic

always @\* begin

next\_state = state\_reg;

q\_load = 1'b0;

q\_dec = 1'b0;

db\_tick = 1'b0;

case(state\_reg)

zero : begin

db\_level = 1'b0;

if(btn) begin

next\_state = wait1;

q\_load = 1'b1;

end

end

wait1 : begin

db\_level = 1'b0;

if(btn) begin

q\_dec = 1'b1;

if(q\_zero) begin

next\_state = one;

db\_tick = 1'b1;

end

end

else

next\_state = zero;

end

one : begin

db\_level = 1'b1;

if(~btn) begin

q\_dec = 1'b1;

if(q\_zero)

next\_state = zero;

end

else

next\_state = one;

end

default : next\_state = zero;

endcase

end

Endmodule

module seg7\_control(

input clk\_100MHz,

input reset,

input [3:0] ones,

input [3:0] tens,

input [3:0] hundereds,

input [3:0] thousands,

output reg [0:6] seg,

output reg [3:0] an

);

// Parameters for segment values

parameter NULL = 7'b111\_1111; // Turn off all segments

parameter ZERO = 7'b000\_0001; // 0

parameter ONE = 7'b100\_1111; // 1

parameter TWO = 7'b001\_0010; // 2

parameter THREE = 7'b000\_0110; // 3

parameter FOUR = 7'b100\_1100; // 4

parameter FIVE = 7'b010\_0100; // 5

parameter SIX = 7'b010\_0000; // 6

parameter SEVEN = 7'b000\_1111; // 7

parameter EIGHT = 7'b000\_0000; // 8

parameter NINE = 7'b000\_0100; // 9

parameter F = 7'b011\_1000; //F;

parameter E = 7'b011\_0000; //E;

parameter d = 7'b100\_0010; //d;

parameter c = 7'b111\_0010; //c;

parameter b = 7'b110\_0000; //b;

parameter A = 7'b000\_1000; //A;

parameter t = 7'b111\_1000; //t;

parameter r = 7'b111\_1010; //r;

// To select each anode in turn

reg [1:0] anode\_select;

reg [16:0] anode\_timer;

always @(posedge clk\_100MHz or posedge reset) begin

if(reset) begin

anode\_select <= 0;

anode\_timer <= 0;

end

else

if(anode\_timer == 99\_999) begin

anode\_timer <= 0;

anode\_select <= anode\_select + 1;

end

else

anode\_timer <= anode\_timer + 1;

end

always @(anode\_select) begin

case(anode\_select)

2'b00 : an = 4'b0111;

2'b01 : an = 4'b1011;

2'b10 : an = 4'b1101;

2'b11 : an = 4'b1110;

endcase

end

// To drive the segments

always @\*

case(anode\_select)

2'b00 : begin

case(thousands)

4'b0000 : seg = t;

4'b0001 : seg = r;

endcase

end

2'b01 : begin

case(hundereds)

4'b0000 : seg = ZERO;

4'b0001 : seg = ONE;

4'b0010 : seg = TWO;

4'b0011 : seg = THREE;

endcase

end

2'b10 : begin

case(tens)

4'b0000 : seg = ZERO;

4'b0001 : seg = ONE;

4'b0010 : seg = TWO;

4'b0011 : seg = THREE;

4'b0100 : seg = FOUR;

4'b0101 : seg = FIVE;

4'b0110 : seg = SIX;

4'b0111 : seg = SEVEN;

4'b1000 : seg = EIGHT;

4'b1001 : seg = NINE;

4'b1010 : seg = F;

4'b1011 : seg = E;

4'b1100 : seg = d;

4'b1101 : seg = c;

4'b1110 : seg = b;

4'b1111 : seg = A;

endcase

end

2'b11 : begin

case(ones)

4'b0000 : seg = ZERO;

4'b0001 : seg = ONE;

4'b0010 : seg = TWO;

4'b0011 : seg = THREE;

4'b0100 : seg = FOUR;

4'b0101 : seg = FIVE;

4'b0110 : seg = SIX;

4'b0111 : seg = SEVEN;

4'b1000 : seg = EIGHT;

4'b1001 : seg = NINE;

4'b1010 : seg = F;

4'b1011 : seg = E;

4'b1100 : seg = d;

4'b1101 : seg = c;

4'b1110 : seg = b;

4'b1111 : seg = A;

endcase

end

endcase

Endmodule

References:

Github FGPADude, FGPA Projects. https://github.com/FPGADude/Digital-Design/tree/main/FPGA%20Projects/UART