CS 223

Project Report

Fall 2023-2024

Mehmet Akif Şahin
22203673 - Section 1

December 10, 2023

‘

Contents

1 UART Hardware Communication Protocol Design and Implementation

In this UART hardware communication protocol design, a modularized device is designed. Starting from the finished version of the device, all steps and modules of the design will be explained.

As seen in 1 final version of the design consists of button debouncers, the automatic mode controller and the UART device itself. Explanation will start with UART device and continue.

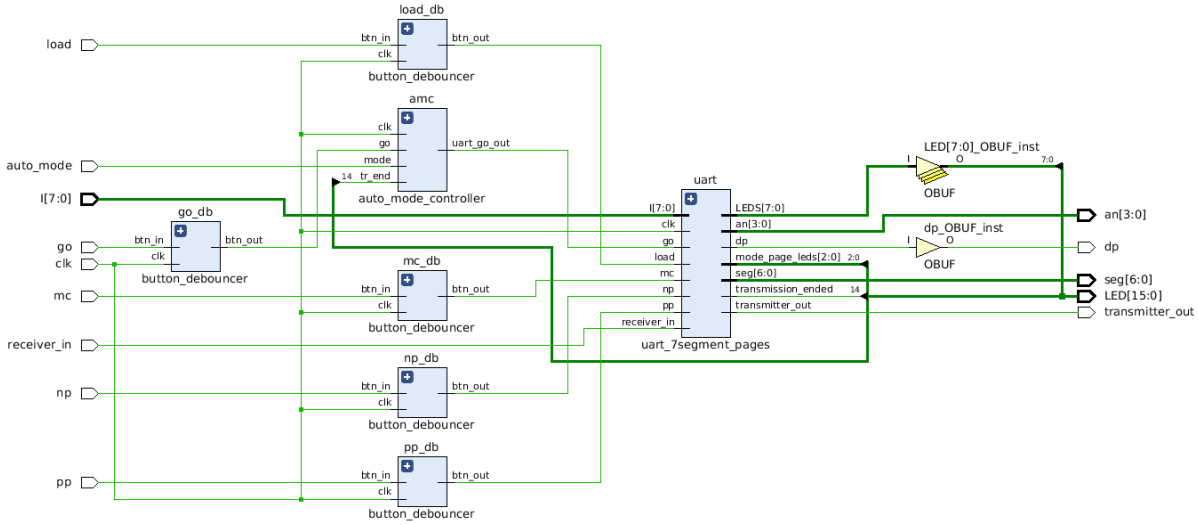

Figure 1: Uart Device Stage 4 Design

1.1 Uart Device

Uart device consists of 4 parts: a datapath for saving the values of 7 segment display's page and mode information, a reader module for reading 2 bytes from transmitters/receivers register file with 1 read port, a 7 segment display driver and the uart device.

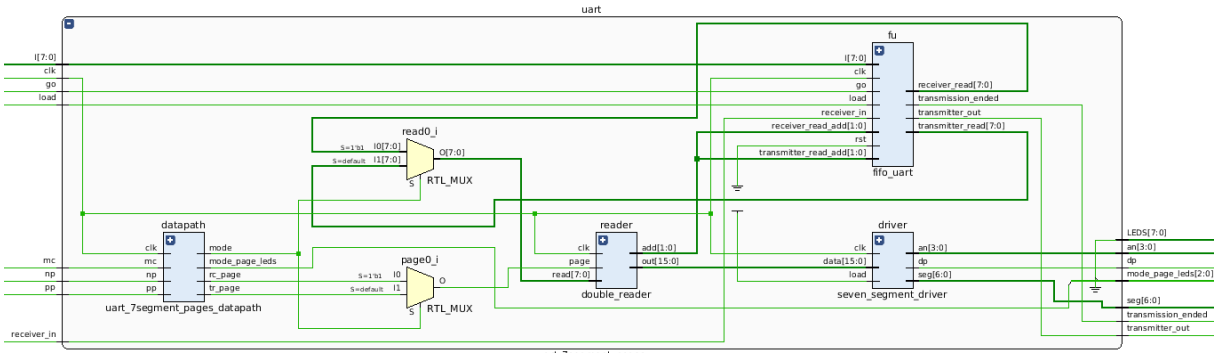

Figure 2: Uart Device with 7 Segment Display Design

1.1.1 Uart Device with FIFO Register

This module consists of transmitter with fifo register and receiver with fifo register.

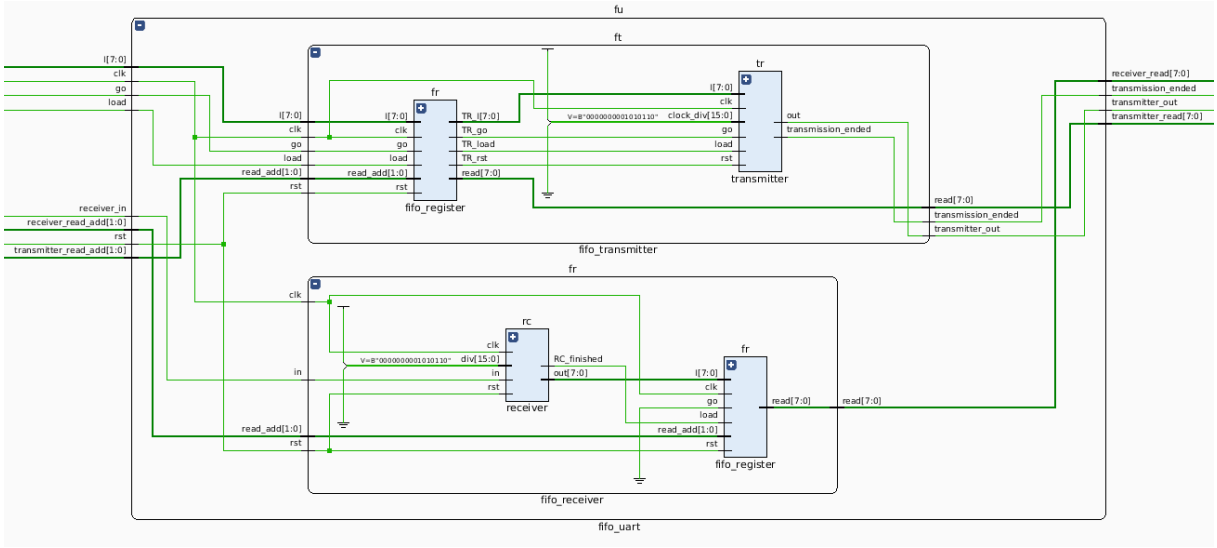

Figure 3: Uart Device with FIFO Register

1.1.1.1 Fifo Register Design

Inputs: 8 bit I, 1 bit rst, 1 bit load, 1 bit go.
Internal Registers: 8 bit reg0, 8 bit reg1, 8 bit reg2, 8 bit reg3.
Outputs: 8 bit I_out, 1 bit rst_out, 1 bit load_out, 1 bit go_out.

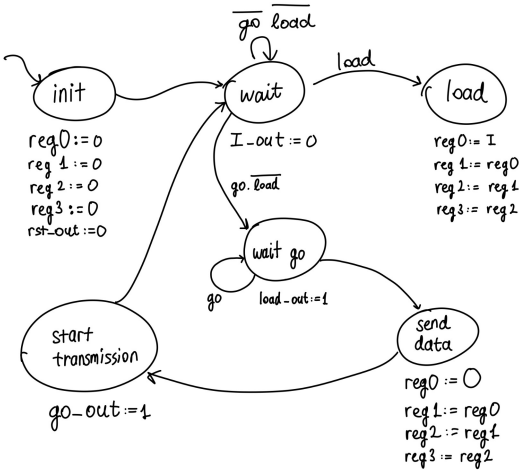

Figure 4: Fifo Register High Level Behavior

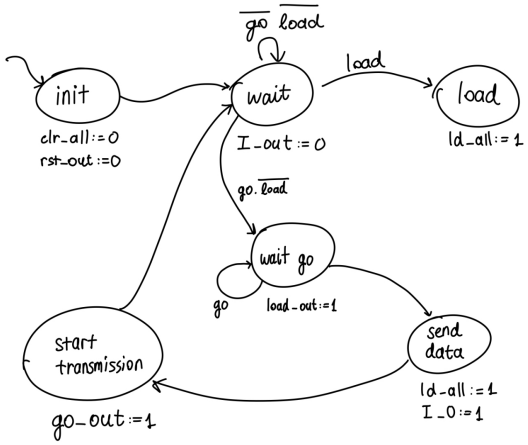

Figure 5: Fifo Register Controller

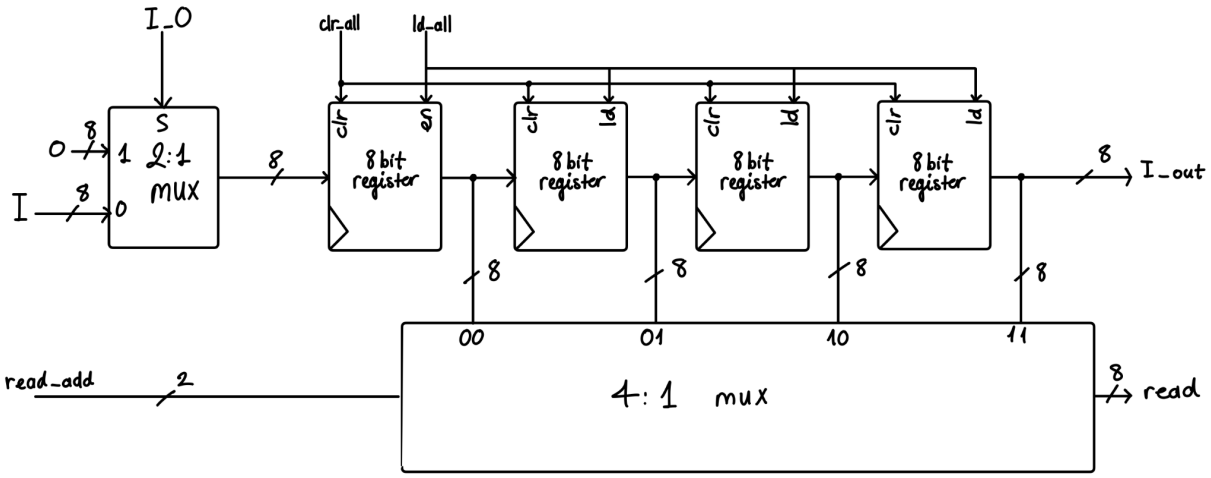

Figure 6: Fifo Register Datapath

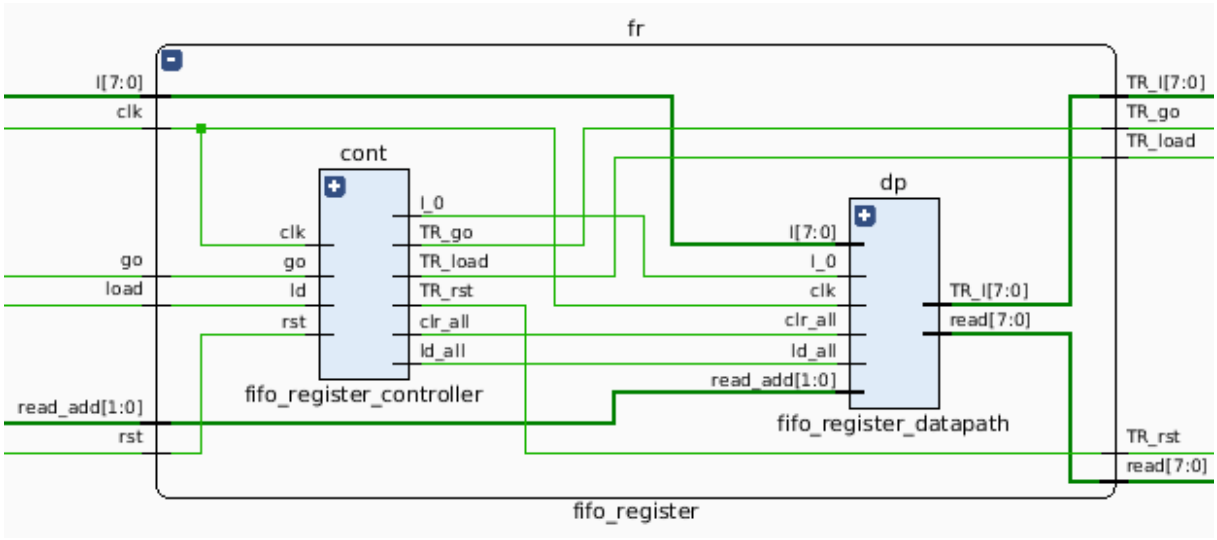

Figure 7: Fifo Register Block Diagram

1.1.1.2 Transmitter Design

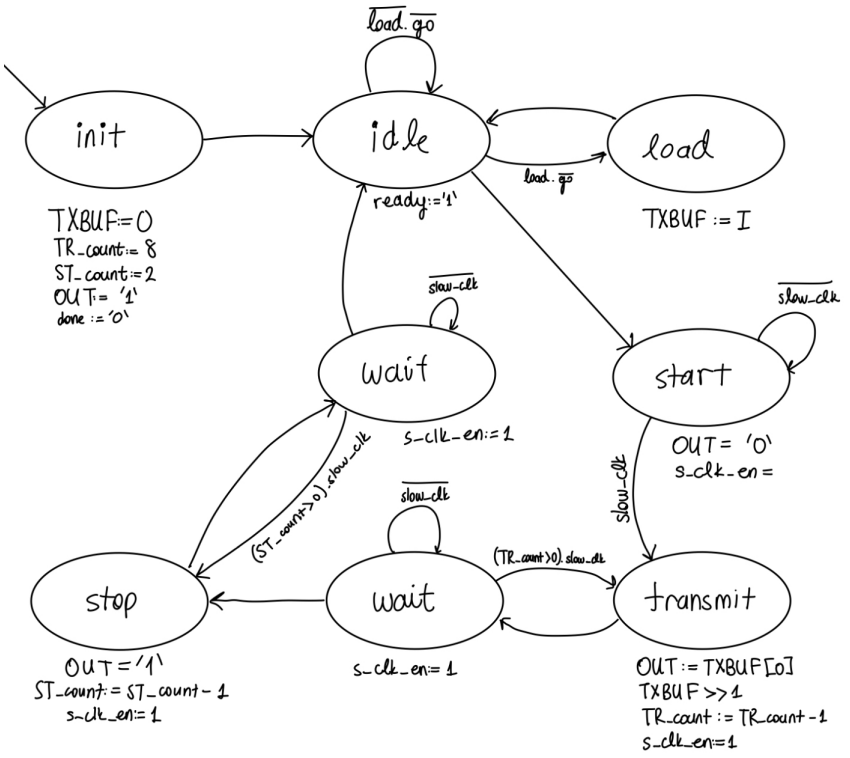

Figure 8: Transmitter High Level Behavior

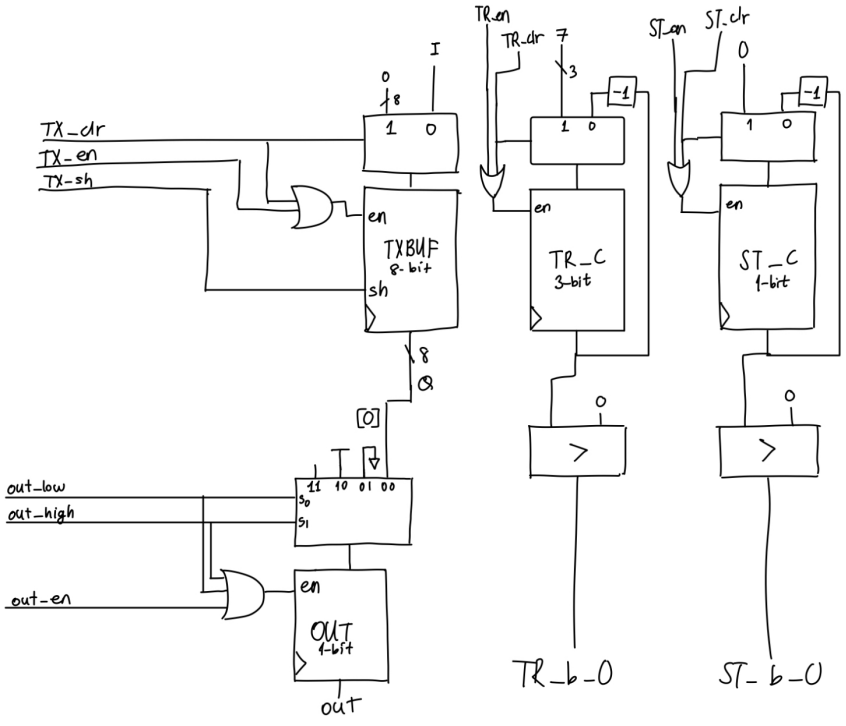

Figure 9: Transmitter Datapath

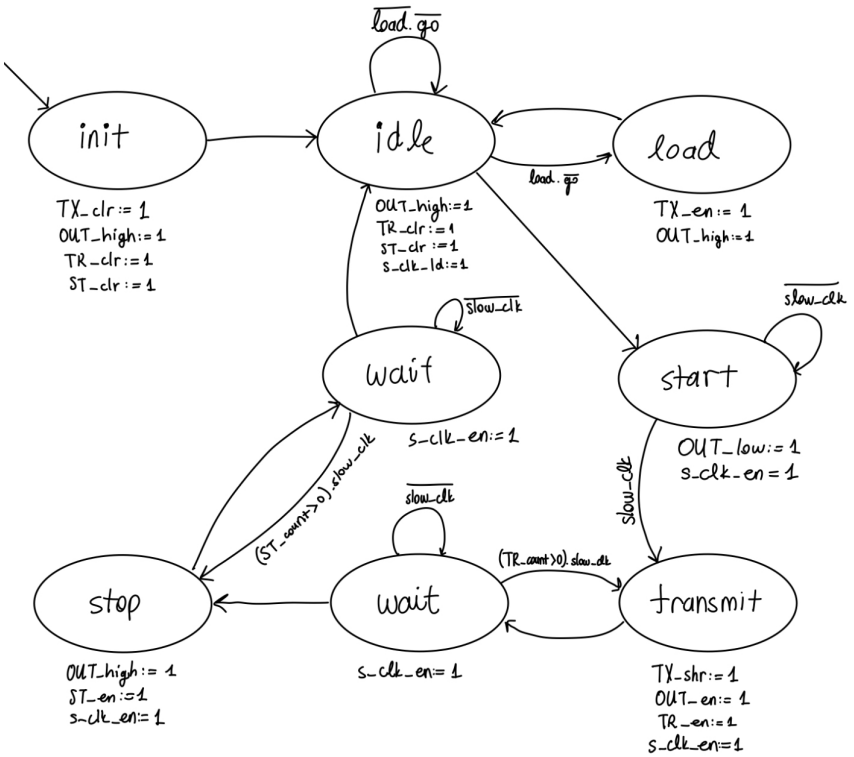

Figure 10: Transmitter Controller

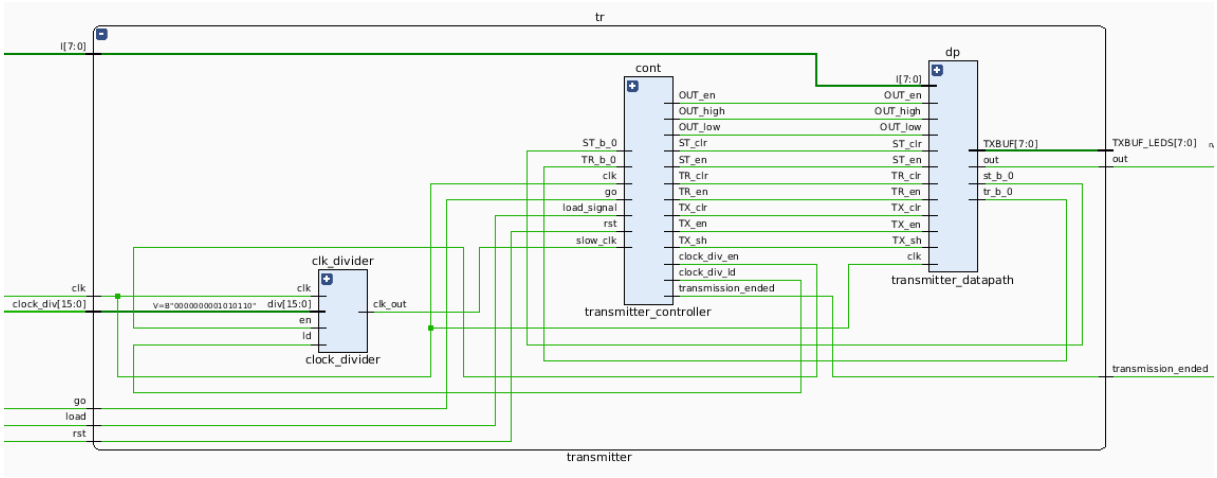

Figure 11: Transmitter Design Block Diagram

1.1.1.3 Receiver Design

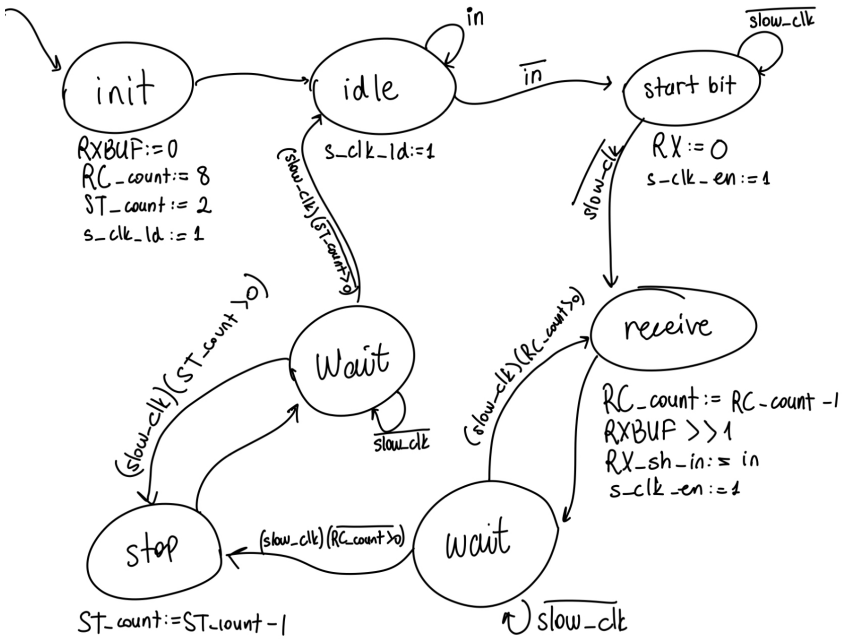

Figure 12: Receiver High Level Behavior

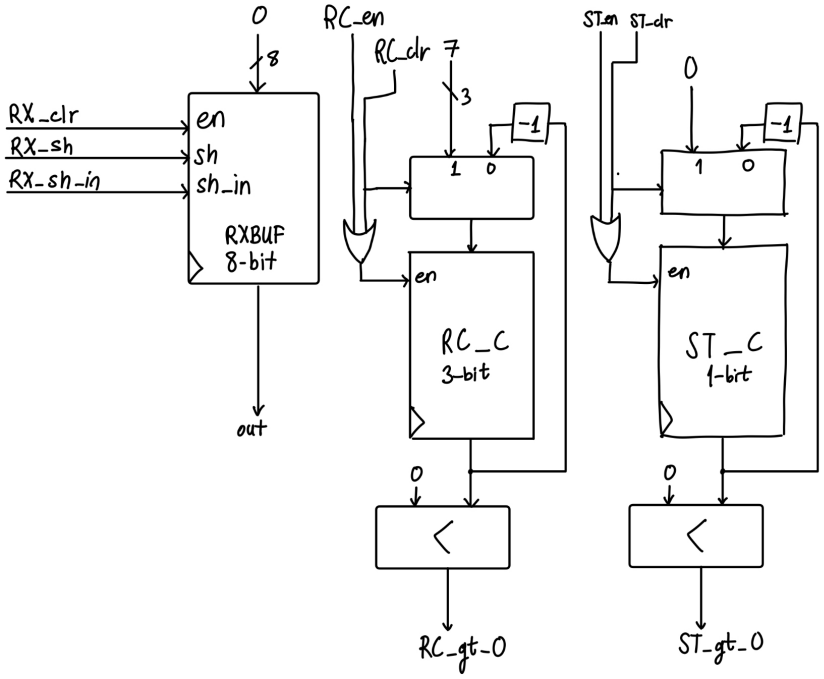

Figure 13: Receiver Datapath

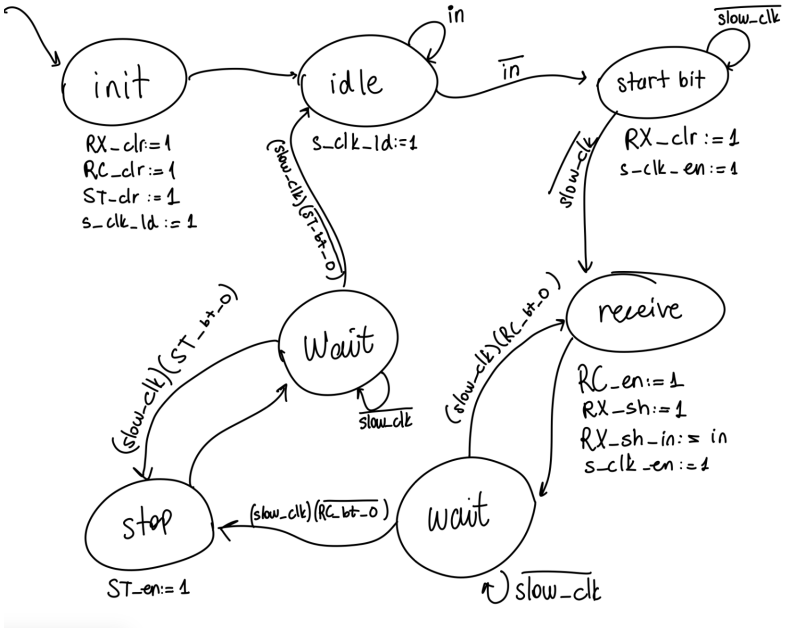

Figure 14: Receiver Controller

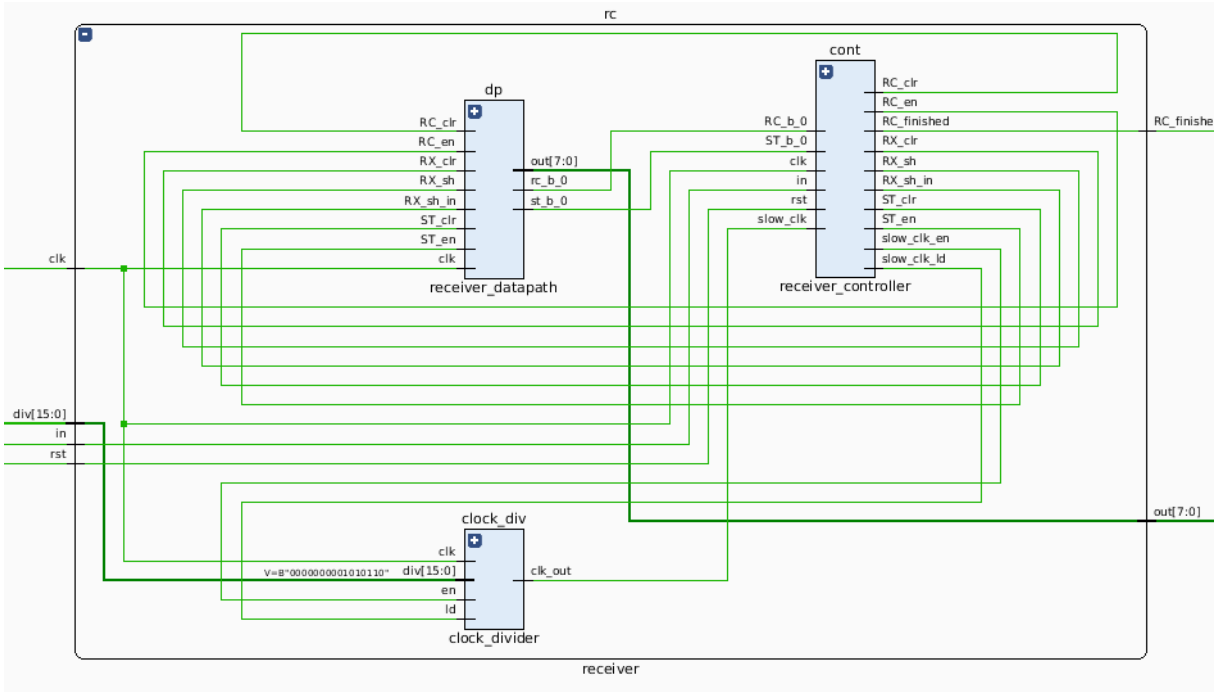

Figure 15: Receiver Design Block Diagram

1.1.1.4 Clock Divider Design

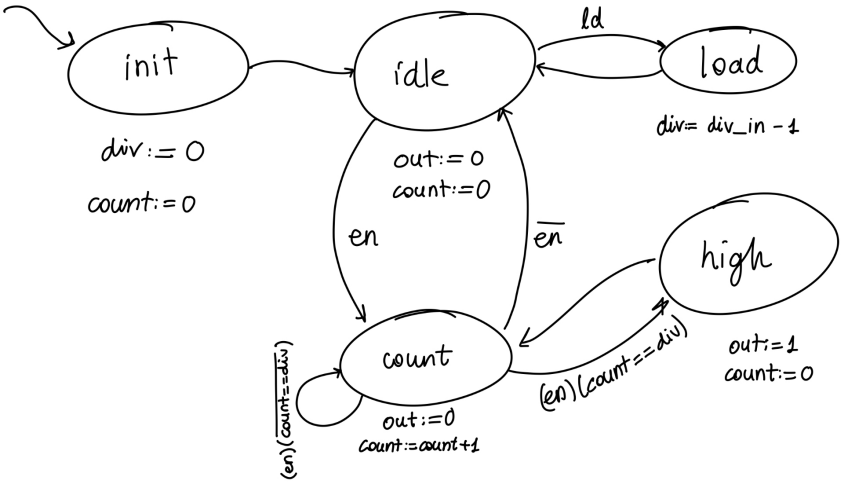

Figure 16: Clock Divider High Level Behavior

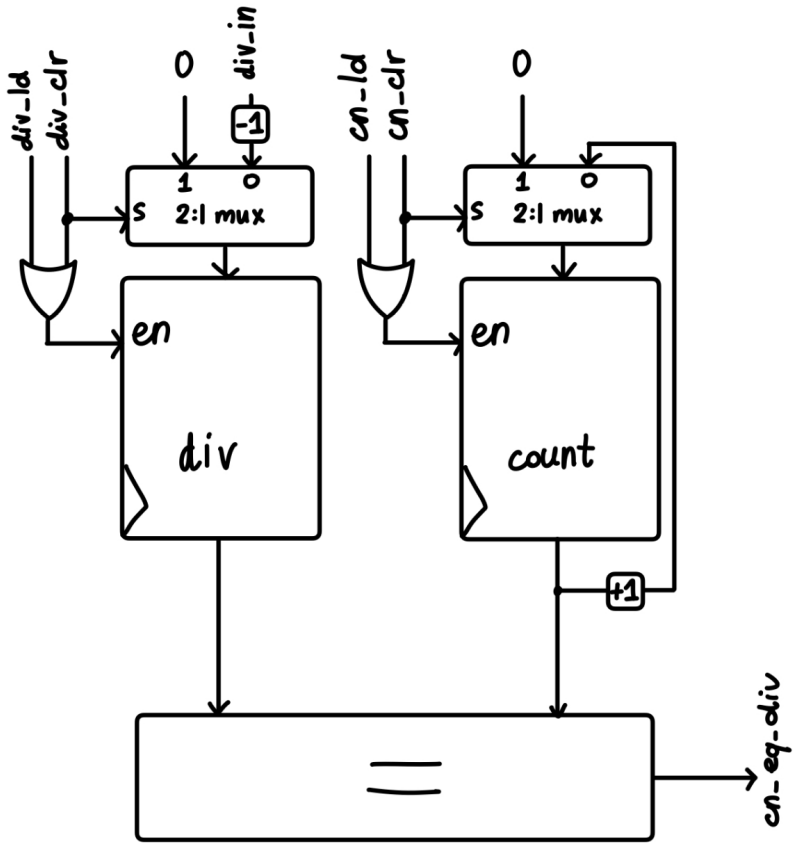

Figure 17: Clock Divider Datapath

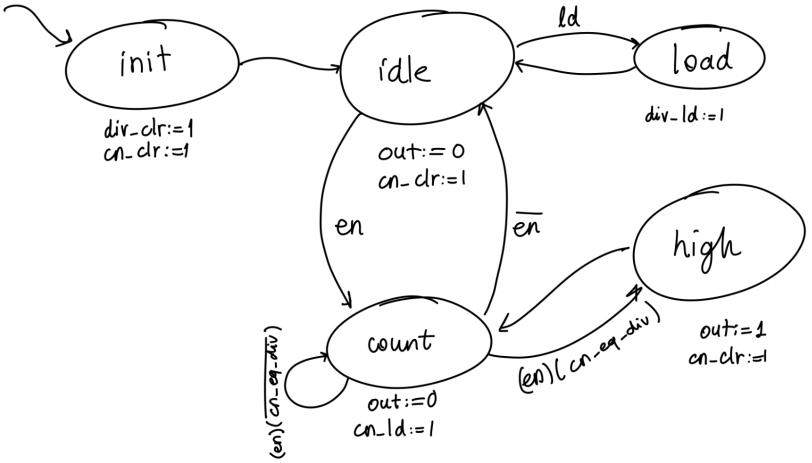

Figure 18: Clock Divider Controller

1.1.2 Datapath for Page and Mode Information

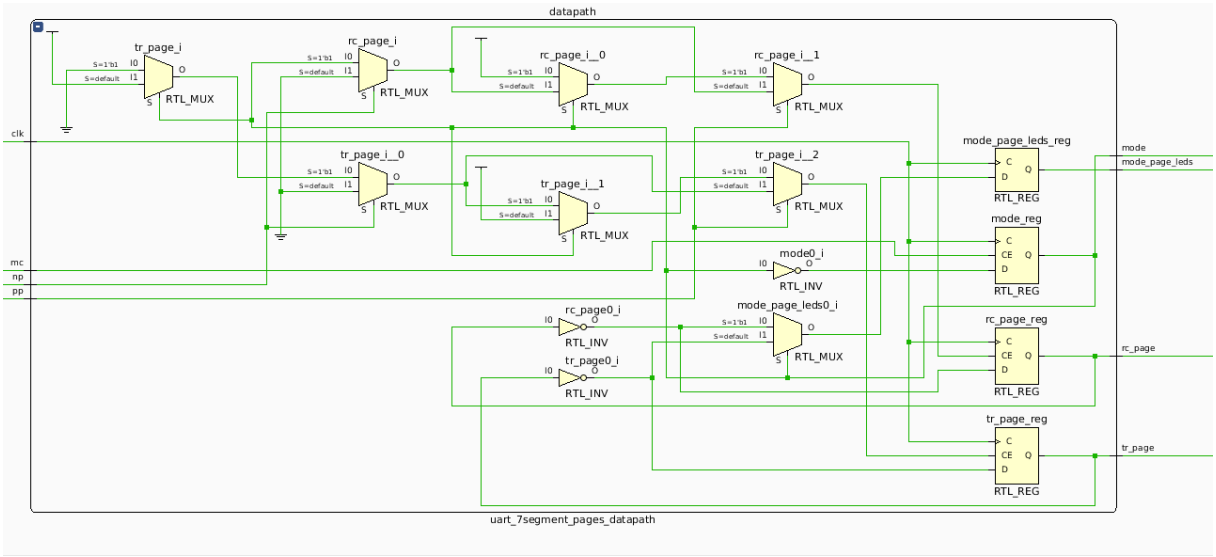

Figure 19: Datapath for Page and Mode Information

1.1.3 Double Reader

This module reads 2 bytes one by one from the appropriate register files (transmitters or receivers) according to the page and mode inputs.

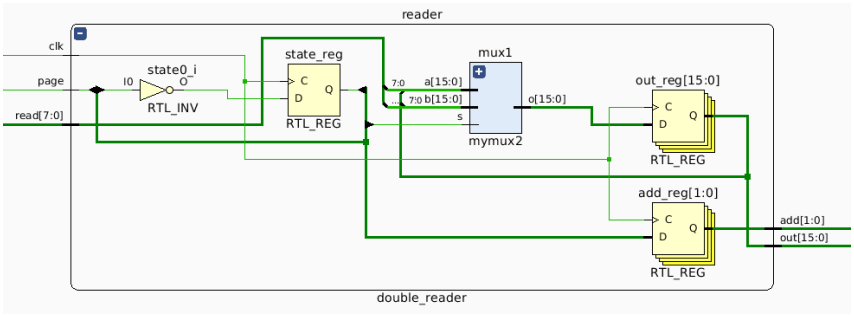

Figure 20: Schematic for Double Reader

1.2 Automatic Mode Controller

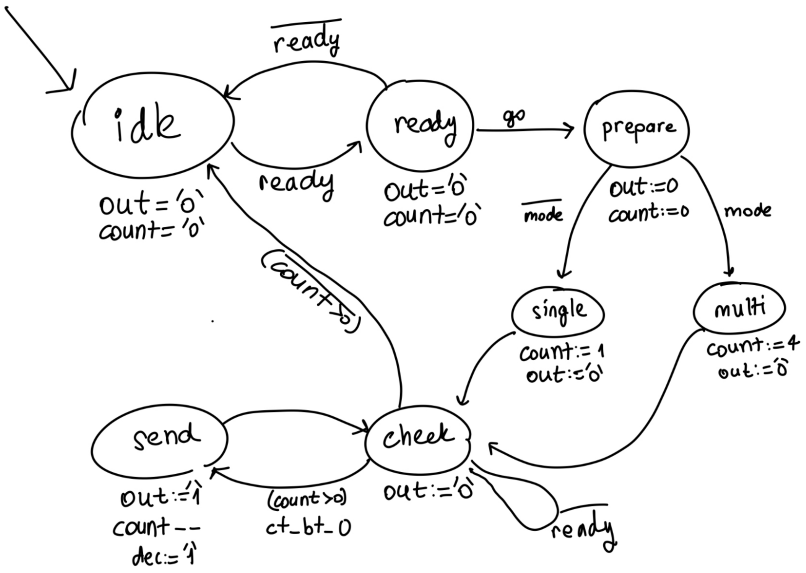

Figure 21: Auto Mode Controller High Level Behavior

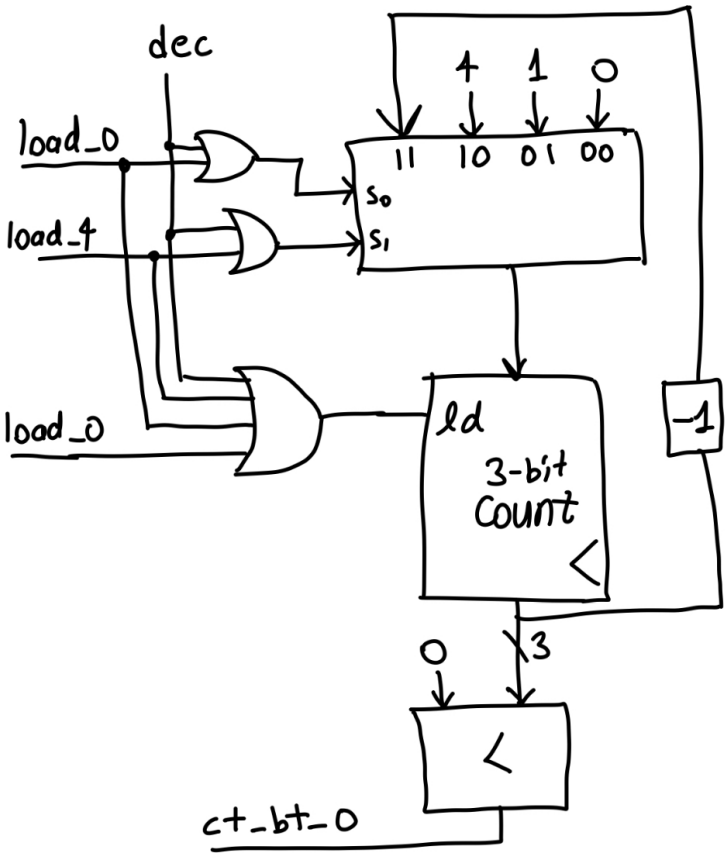

Figure 22: Auto Mode Controller Datapath

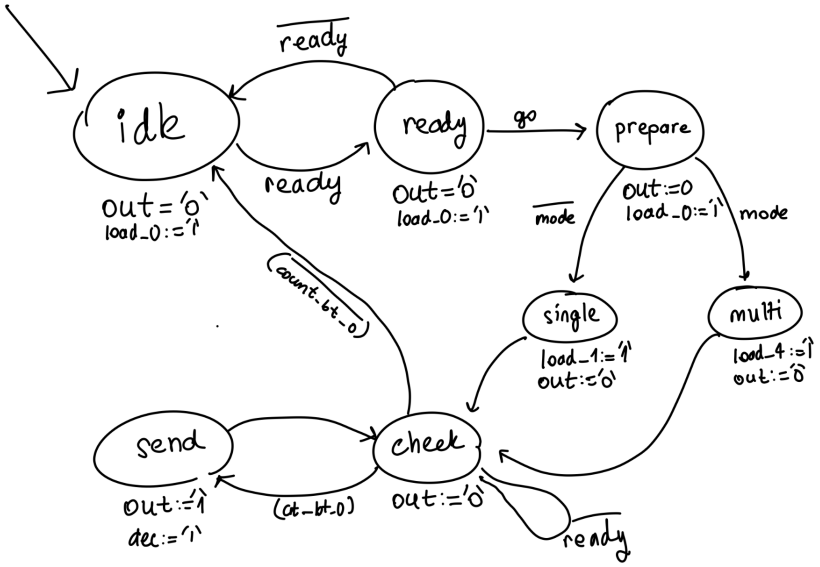

Figure 23: Controller of Automatic Mode Controller

1.3 Button Debouncers

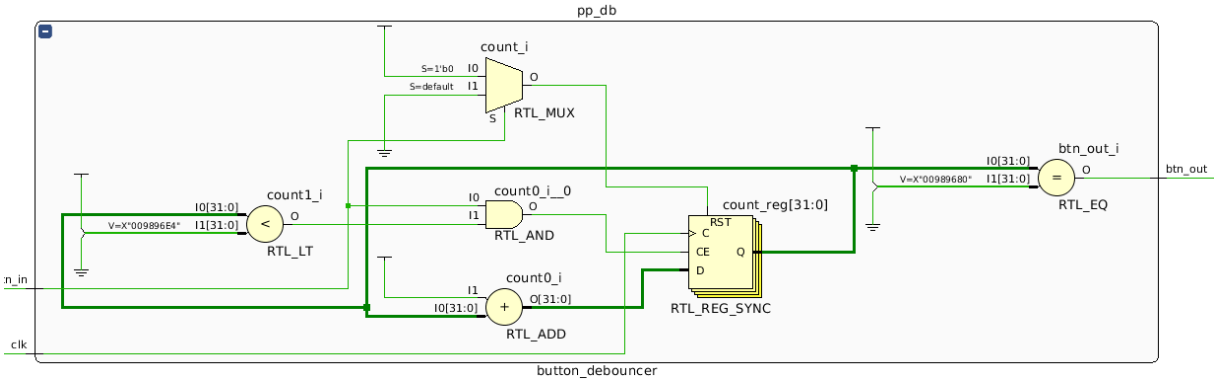

Figure 24: Button Debouncer Schematic

2 Codes

```
module uart_stage4
#(DATA_WIDTH = 8, STOP_WIDTH = 2, BAUD_RATE = 115200, FREQ = 100_000_00)
(
input clk, load, go, np, pp, mc, auto_mode,
input [7:0] I,
output [15:0] LED,
output [6:0] seg,
output [3:0] an,
output dp,
input receiver_in,
output transmitter_out
);



wire np_, pp_, mc_, go_, load_;

button_debouncer np_db (clk, np, np_);
button_debouncer pp_db (clk, pp, pp_);
button_debouncer mc_db (clk, mc, mc_);
button_debouncer go_db (clk, go, go_);
button_debouncer load_db (clk, load, load_);


wire transmission_ended;
wire uart_go;
assign LED[14] = transmission_ended;

auto_mode_controller amc (
clk, auto_mode, go_, transmission_ended, uart_go
);

uart_7segment_pages
#(DATA_WIDTH, STOP_WIDTH, BAUD_RATE, FREQ)
uart (
clk, load_, uart_go, np_, pp_, mc_,
I,
LED[7:0],
```

```
        seg,
        an,
        dp,
        receiver_in,
        transmitter_out,
        {LED[15], LED[9:8]},
        transmission_ended
        );


        endmodule`
```

2.1 Uart Device

```
module uart_7segment_pages
#(DATA_WIDTH = 8, STOP_WIDTH = 2, BAUD_RATE = 115200, FREQ = 100_000_00)
(
input clk, load, go, np, pp, mc,
input [DATA_WIDTH-1:0] I,
output [DATA_WIDTH-1:0] LEDS,
output [6:0] seg,
output [3:0] an,
output dp,
input receiver_in,
output transmitter_out,
output logic [2:0] mode_page_leds,
output transmission_ended
);
// mode=0 => TRANSMITTER_ARRAY, mode=1 => RECEIVER_ARRAY

wire mode, tr_page, rc_page;

uart_7segment_pages_datapath datapath ( clk, np, pp, mc, mode, tr_page, rc_page, mode_p

wire [DATA_WIDTH*2-1:0]reader_out;
wire [1:0] rc_add, tr_add;
wire [DATA_WIDTH-1:0] rc_read, tr_read;

seven_segment_driver driver (
clk,
reader_out,
1'b1,
seg,
an,
dp
);

wire [1:0] reader_add_out;

//     mymux2 #2 mux2 (mode, tr_add, rc_add, reader_add_out);

double_reader reader (
clk,
mode?rc_page:tr_page,
mode?rc_read:tr_read,
reader_add_out,
reader_out
```

```
);

fifo_uart #(DATA_WIDTH, STOP_WIDTH, BAUD_RATE, FREQ) fu (
clk, 1'b0, load, go,
I,
receiver_in,
transmitter_out,
reader_add_out, reader_add_out,
rc_read, tr_read,
transmission_ended
);

endmodule
```

2.1.1 Uart Device with FIFO Register

```
module fifo_uart
# (parameter DATA_WIDTH = 8, STOP_WIDTH = 2, BAUD_RATE = 115200, FREQ = 100_000_000)
(
input logic clk, rst, load, go,
input logic [DATA_WIDTH-1:0] I,
input logic receiver_in,
output logic transmitter_out,
input logic [1:0] receiver_read_add, transmitter_read_add,
output logic [DATA_WIDTH-1:0] receiver_read, transmitter_read,
output transmission_ended
);

fifo_receiver #(DATA_WIDTH, STOP_WIDTH, BAUD_RATE, FREQ) fr (clk, rst, receiver_in, rece
fifo_transmitter #(DATA_WIDTH, STOP_WIDTH, BAUD_RATE, FREQ) ft (clk, rst, load, go, I, t

endmodule
```

2.1.1.1 Fifo Register Design

```
module fifo_register
#(parameter DATA_LENGTH = 8)
(
input logic clk,
input logic [DATA_LENGTH-1:0] I,
input logic rst, load, go,
output logic [DATA_LENGTH-1:0] TR_I,
output logic TR_rst, TR_load, TR_go,
input logic [1:0] read_add,
output logic [7:0] read
);

wire ld_all, clr_all, I_0;

fifo_register_controller cont (clk, rst, load, go, ld_all, clr_all, I_0, TR_rst, TR_load
fifo_register_datapath #(DATA_LENGTH) dp (clk, I, ld_all, clr_all, I_0, TR_I, read_add,

endmodule
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
```

```
// Create Date: 12/05/2023 01:41:51 PM
// Design Name:
// Module Name: fifo_register_controller
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fifo_register_controller(
input logic clk, rst, ld, go,
output logic ld_all, clr_all, I_0, TR_rst, TR_load, TR_go
);

logic [2:0] state, next_state;

localparam init = 3'b000;
localparam idle = 3'b001;
localparam load = 3'b010;
localparam wait_load = 3'b011;
localparam wait_go = 3'b100;
localparam send_data = 3'b101;
localparam start_transmission = 3'b111;

always_ff @ (posedge clk) begin
if (rst) state <= init;
else state <= next_state;
end

always_comb begin
case (state)
default begin
ld_all = 0;
clr_all = 0;
I_0 = 0;
TR_rst = 0;
TR_load = 0;
TR_go = 0;

next_state = init;
end
init: begin
ld_all = 0;
clr_all = 1;
I_0 = 0;
TR_rst = 1;
TR_load = 0;
TR_go = 0;

next_state = idle;
```

```
end
idle: begin
ld_all = 0;
clr_all = 0;
I_0 = 0;
TR_rst = 0;
TR_load = 0;
TR_go = 0;

if (ld) next_state = load;
else if (go) next_state = wait_go;
else next_state = idle;
end
load: begin
ld_all = 1;
clr_all = 0;
I_0 = 0;
TR_rst = 0;
TR_load = 0;
TR_go = 0;

if (ld) next_state = wait_load;
else next_state = idle;
end
wait_load: begin
ld_all = 0;
clr_all = 0;
I_0 = 0;
TR_rst = 0;
TR_load = 0;
TR_go = 0;

if (ld) next_state = wait_load;
else next_state = idle;
end
wait_go: begin
ld_all = 0;
clr_all = 0;
I_0 = 0;
TR_rst = 0;
TR_load = 1;
TR_go = 0;

if (go) next_state = wait_go;
else next_state = send_data;
end
send_data: begin
ld_all = 1;
clr_all = 0;
I_0 = 1;
TR_rst = 0;
TR_load = 0;
TR_go = 0;

next_state = start_transmission;
end
start_transmission: begin
ld_all = 0;
```

```
clr_all = 0;
I_0 = 0;
TR_rst = 0;
TR_load = 0;
TR_go = 1;

next_state = idle;
end
endcase
end
endmodule
module fifo_register_datapath
#(parameter DATA_LENGTH = 8)
(
input logic clk,
input logic [DATA_LENGTH-1:0] I,
input logic ld_all, clr_all, I_0,
output logic [DATA_LENGTH-1:0] TR_I,
input logic [1:0] read_add,
output logic [7:0] read
);

wire [DATA_LENGTH-1:0] reg0_in;
wire [DATA_LENGTH-1:0] reg0_out, reg1_out, reg2_out;

mymux2 #(DATA_LENGTH) mux1 (I_0, 0, I, reg0_in);

myreg_clear #(DATA_LENGTH) reg0 (clk, ld_all, clr_all, reg0_in, reg0_out);
myreg_clear #(DATA_LENGTH) reg1 (clk, ld_all, clr_all, reg0_out, reg1_out);
myreg_clear #(DATA_LENGTH) reg2 (clk, ld_all, clr_all, reg1_out, reg2_out);
myreg_clear #(DATA_LENGTH) reg3 (clk, ld_all, clr_all, reg2_out, TR_I);

mymux4 #(DATA_LENGTH) mux2 (read_add, TR_I, reg2_out, reg1_out, reg0_out, read);

endmodule
```

2.1.1.2 Transmitter Design

```
module fifo_transmitter
#(parameter DATA_WIDTH = 8, STOP_WIDTH = 1, BAUD_RATE = 115200, FREQ = 100_000_000)
(
input logic clk, rst, load, go,
input logic [DATA_WIDTH-1:0] I,
output logic transmitter_out,
input logic [1:0] read_add,
output logic [DATA_WIDTH-1:0] read,
output transmission_ended
);

localparam CLOCK_DIV = FREQ / BAUD_RATE;

wire [7:0] TR_I;
wire TR_go, TR_ld, TR_rst;

fifo_register #(DATA_WIDTH) fr (
clk, I, rst, load, go, TR_I, TR_rst, TR_ld, TR_go, read_add, read
);
wire garbage;
```

```
transmitter #(DATA_WIDTH, STOP_WIDTH) tr (
clk,
TR_rst,
CLOCK_DIV,
TR_ld,
TR_go,
TR_I,
transmitter_out,
garbage,
transmission_ended
);
endmodule
module transmitter
# (parameter DATA_WIDTH = 8, STOP_WIDTH = 1)
(
input logic clk,
input logic rst,
input logic [15:0] clock_div,
input logic load,
input logic go,
input logic [DATA_WIDTH-1:0] I,
output logic out,
output logic [DATA_WIDTH-1:0] TXBUF_LEDS,
output transmission_ended
);
wire tr_b_0, st_b_0;
wire TX_clr, TX_en, TX_sh, OUT_low, OUT_high, OUT_en, TR_en, TR_clr, ST_en, ST_clr;

wire slow_clk, div_ld, div_en;

transmitter_controller  cont (clk, slow_clk, div_ld, div_en, rst, load, go, tr_b_0, st_b
transmitter_datapath #(DATA_WIDTH, STOP_WIDTH) dp  (clk, I, TX_clr, TX_en, TX_sh, OUT_lo

clock_divider clk_divider (clk, clock_div, div_ld, div_en, slow_clk);
endmodule
module transmitter_controller
(
input logic clk,
input logic slow_clk,
output logic clock_div_ld,
output logic clock_div_en,
input logic rst,
input logic load_signal,
input logic go,
input logic TR_b_0,
input logic ST_b_0,
output logic TX_clr,
output logic TX_en,
output logic TX_sh,
output logic OUT_low,
output logic OUT_high,
output logic OUT_en,
output logic TR_en,
output logic TR_clr,
output logic ST_en,
output logic ST_clr,
output logic transmission_ended
);
```

```
logic [3:0] state;
logic [3:0] next_state;
//     myregister #(3) state_reg (clk, next_state, 1, state);

parameter init = 4'b0000;
parameter idle = 4'b0001;
parameter load = 4'b0010;
parameter go_wait = 4'b0011;
parameter start = 4'b0100;
parameter transmit = 4'b0101;
parameter transmit_wait = 4'b0110;
parameter stop = 4'b0111;
parameter stop_wait = 4'b1000;
parameter finito = 4'b1001;

always_ff @ (posedge clk) begin
if (rst) state <= init;
else state <= next_state;
end

always_comb begin
case (state)
init: begin
TX_clr = 1;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 1;
OUT_en = 0;
TR_en = 0;
TR_clr = 1;
ST_en = 0;
ST_clr = 1;

transmission_ended = 0;


clock_div_ld = 1;
clock_div_en = 0;

next_state = idle;
end
idle: begin
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 1;
OUT_en = 0;
TR_en = 0;
TR_clr = 1;
ST_en = 0;
ST_clr = 1;

clock_div_ld = 1;
clock_div_en = 0;

transmission_ended = 0;
```

```
if (load_signal) next_state = load;
else if (go) next_state = go_wait;
else next_state = idle;
end
load: begin
TX_clr = 0;
TX_en = 1;
TX_sh = 0;
OUT_low = 0;
OUT_high = 1;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;

clock_div_ld = 0;
clock_div_en = 0;

transmission_ended = 0;

next_state = idle;
end
go_wait: begin
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 1;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;

clock_div_ld = 0;
clock_div_en = 0;

transmission_ended = 0;

if (go) next_state = go_wait;
else next_state = start;
end
start: begin
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 1;
OUT_high = 0;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;

transmission_ended = 0;
```

```
clock_div_ld = 0;
clock_div_en = 1;

if (slow_clk) next_state = transmit;
else next_state = start;
end
transmit: begin
TX_clr = 0;
TX_en = 0;
TX_sh = 1;
OUT_low = 0;
OUT_high = 0;
OUT_en = 1;
TR_en = 1;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;

transmission_ended = 0;

clock_div_ld = 0;
clock_div_en = 1;

next_state = transmit_wait;
end
transmit_wait: begin
clock_div_ld = 0;
clock_div_en = 1;
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 0;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;

transmission_ended = 0;

if (slow_clk) begin
if (TR_b_0) next_state = transmit;
else next_state = stop;
end
else next_state = transmit_wait;
end
stop: begin
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 1;
OUT_en = 0;
TR_en = 1;
TR_clr = 0;
ST_en = 1;
ST_clr = 0;
```

```
clock_div_ld = 0;
clock_div_en = 1;
transmission_ended = 0;

next_state = stop_wait;
end
stop_wait: begin
clock_div_ld = 0;
clock_div_en = 1;
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 0;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;
transmission_ended = 0;

if (slow_clk) begin
if (ST_b_0) next_state = stop;
else next_state = finito;
end
else next_state = stop_wait;
end
finito: begin
clock_div_ld = 0;
clock_div_en = 0;
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 0;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;
transmission_ended = 1;
next_state = idle;
end
default begin
TX_clr = 0;
TX_en = 0;
TX_sh = 0;
OUT_low = 0;
OUT_high = 0;
OUT_en = 0;
TR_en = 0;
TR_clr = 0;
ST_en = 0;
ST_clr = 0;
transmission_ended = 0;

clock_div_ld = 0;
```

```
clock_div_en = 1;

next_state = init;
end
endcase
end


endmodule
module transmitter_datapath
# (parameter DATA_WIDTH = 8, STOP_WIDTH = 1)
(
input clk,
input logic [DATA_WIDTH-1:0]I,
input logic TX_clr, TX_en, TX_sh, OUT_low, OUT_high, OUT_en, TR_en, TR_clr, ST_en, ST_cl
output logic tr_b_0, st_b_0, out,
output logic [DATA_WIDTH-1:0] TXBUF
);
wire [DATA_WIDTH-1:0] TXBUF_in;
wire TXBUF_en;
myregister_shift #(DATA_WIDTH) TXBUF_reg (clk, TXBUF_in, TXBUF_en, TX_sh, 0, 0, TXBUF);
mymux2 #(DATA_WIDTH) mux1 (TX_clr, 0, I, TXBUF_in);
or (TXBUF_en, TX_en, TX_clr);

wire out_en, out_in;
myregister #(1) OUT_reg (clk, out_in, out_en, out);
or3 or1 (out_en, OUT_en, OUT_high, OUT_low);
mymux4 mux2 ({OUT_high, OUT_low}, 1, 1, 0, TXBUF[0], out_in);

wire tr_en;
wire [3:0] tr_in, tr_out;
myregister #(4) TR_reg (clk, tr_in, tr_en, tr_out);
or (tr_en, TR_en, TR_clr);
mymux2 #(4) mux3 (TR_clr, DATA_WIDTH, tr_out - 1, tr_in);
assign tr_b_0 = tr_out > 0 ? 1'b1 : 1'b0;

wire st_en;
wire [1:0] st_in, st_out;
myregister #(2) ST_reg (clk, st_in, st_en, st_out);
or (st_en, ST_en, ST_clr);
mymux2 #(2) mux4 (ST_clr, STOP_WIDTH, st_out - 1, st_in);
assign st_b_0 = st_out > 0 ? 1'b1 : 1'b0;

endmodule
```

2.1.1.3 Receiver Design

```
module fifo_receiver

#(parameter DATA_WIDTH = 8, STOP_WIDTH = 1, BAUD_RATE = 115200, FREQ = 100_000_000)

(
input logic clk, rst, in,
input logic [1:0] read_add,
output logic [DATA_WIDTH-1:0] read
);

localparam CLOCK_DIV = FREQ / BAUD_RATE;
```

```
wire finished;
wire [DATA_WIDTH-1:0] out;
wire TR_I, TR_rst, TR_ld, TR_go;

receiver #(DATA_WIDTH, STOP_WIDTH) rc (clk, CLOCK_DIV, rst, in, out, finished);

fifo_register #(DATA_WIDTH) fr (
clk, out, rst, finished, 0, TR_I, TR_rst, TR_ld, TR_go, read_add, read
);

endmodule
module receiver
# (parameter DATA_WIDTH = 8, STOP_WIDTH = 1)
(
input logic clk,
input logic [15:0] div,
input logic rst, in,
output [DATA_WIDTH - 1:0] out,
output logic RC_finished
);

wire RX_clr, RX_sh, RX_sh_in, RC_en, RC_clr, ST_en, ST_clr;
wire RC_b_0, ST_b_0;

wire clock_div_ld, clock_div_en, slow_clk;

receiver_controller cont (clk, slow_clk, clock_div_ld, clock_div_en, rst, in, RC_b_0, ST
receiver_datapath #(DATA_WIDTH, STOP_WIDTH) dp (clk, RX_clr, RX_sh, RX_sh_in, RC_en, RC

clock_divider clock_div (clk, div, clock_div_ld, clock_div_en, slow_clk);

endmodule
module receiver_controller(
input logic clk, slow_clk,
output logic slow_clk_ld, slow_clk_en,
input logic rst, in, RC_b_0, ST_b_0,
output logic RX_clr, RX_sh, RX_sh_in, RC_en, RC_clr, ST_en, ST_clr,
output logic RC_finished
);
logic [2:0] state;
logic [2:0] next_state;
//myregister #(3) state_reg (clk, next_state, 1, state);

parameter init = 3'b000;
parameter idle = 3'b001;
parameter start_bit = 3'b010;
parameter receive = 3'b011;
parameter receive_wait = 3'b100;
parameter stop = 3'b101;
parameter stop_wait = 3'b110;
parameter finito = 3'b111;

always_ff @ (posedge clk) begin
if (rst) state <= init;
else state <= next_state;
end
```

```
always_comb begin
case (state)
init: begin
RX_clr = 1;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 1;
ST_en = 0;
ST_clr = 1;
RC_finished = 0;

slow_clk_ld = 1;
slow_clk_en = 0;

next_state = idle;
end
idle: begin
RX_clr = 0;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 1;
ST_en = 0;
ST_clr = 1;
RC_finished = 0;

slow_clk_ld = 1;
slow_clk_en = 0;

if (in) next_state = idle;
else next_state = start_bit;
end
start_bit: begin
RX_clr = 1;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 0;
ST_en = 0;
ST_clr = 0;
RC_finished = 0;

slow_clk_ld = 0;
slow_clk_en = 1;

if (slow_clk) next_state = receive;
else next_state = start_bit;
end
receive: begin
RX_clr = 0;
RX_sh = 1;
RX_sh_in = in;
RC_en = 1;
RC_clr = 0;
ST_en = 0;
ST_clr = 0;
RC_finished = 0;
```

```
slow_clk_ld = 0;
slow_clk_en = 1;

next_state = receive_wait;
end
receive_wait: begin
RX_clr = 0;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 0;
ST_en = 0;
ST_clr = 0;
RC_finished = 0;

slow_clk_ld = 0;
slow_clk_en = 1;

if (slow_clk) begin
if (RC_b_0) next_state = receive;
else next_state = stop;
end
else next_state = receive_wait;
end

stop: begin
RX_clr = 0;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 0;
ST_en = 1;
ST_clr = 0;
RC_finished = 0;

slow_clk_ld = 0;
slow_clk_en = 1;

next_state = stop_wait;
end
stop_wait: begin
RX_clr = 0;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 0;
ST_en = 0;
ST_clr = 0;
RC_finished = 0;

slow_clk_ld = 0;
slow_clk_en = 1;

if (slow_clk) begin
if (ST_b_0) next_state = stop;
else next_state = finito;
end
```

```
else next_state = stop_wait;
end
finito: begin
RX_clr = 0;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 0;
ST_en = 0;
ST_clr = 0;
RC_finished = 1;

slow_clk_ld = 0;
slow_clk_en = 0;

next_state = idle;
end
default begin
RX_clr = 0;
RX_sh = 0;
RX_sh_in = 0;
RC_en = 0;
RC_clr = 0;
ST_en = 0;
ST_clr = 0;

slow_clk_ld = 0;
slow_clk_en = 0;

next_state = init;
end
endcase
end

endmodule
module receiver_datapath
# (parameter DATA_WIDTH = 8, STOP_WIDTH = 1)
(
input logic clk, RX_clr, RX_sh, RX_sh_in, RC_en, RC_clr, ST_en, ST_clr,
output logic rc_b_0, st_b_0,
output logic [DATA_WIDTH - 1:0] out
);
myregister_shift #(DATA_WIDTH) RXBUF (clk, 0, RX_clr, RX_sh, 0, RX_sh_in, out);

wire rc_en;
wire [3:0] rc_in, rc_out;
myregister #(4) RC_reg (clk, rc_in, rc_en, rc_out);
or (rc_en, RC_en, RC_clr);
mymux2 #(4) mux3 (RC_clr, DATA_WIDTH, rc_out - 1, rc_in);
assign rc_b_0 = rc_out > 0 ? 1'b1 : 1'b0;

wire st_en;
wire [1:0] st_in, st_out;
myregister #(2) ST_reg (clk, st_in, st_en, st_out);
or (st_en, ST_en, ST_clr);
mymux2 #(2) mux4 (ST_clr, STOP_WIDTH, st_out - 1, st_in);
assign st_b_0 = st_out > 0 ? 1'b1 : 1'b0;
```

```
endmodule
```

2.1.1.4 Clock Divider Design

```
module clock_divider(
input logic clk,
input logic [15:0] div,
input logic ld,
input logic en,
output logic clk_out
);
wire div_ld, div_clr, cn_ld, cn_clr;
wire cn_eq_div;

clock_divider_controller cont (clk, ld, en, cn_eq_div, clk_out, div_ld, div_clr, cn_ld,
clock_divider_datapath dp (clk, div, div_ld, div_clr, cn_ld, cn_clr, cn_eq_div);
endmodule
module clock_divider_controller(
input logic clk, ld, en, cn_eq_div,
output logic clk_out, div_ld, div_clr, cn_ld, cn_clr
);

logic [2:0] state, next_state;
myregister #(3) state_reg (clk, next_state, 1, state);

parameter init = 3'b000;
parameter idle = 3'b001;
parameter load = 3'b010;
parameter count = 3'b011;
parameter high = 3'b100;

always @ (*) begin
case (state)
default begin
next_state = init;
end
init: begin
clk_out = 0;
div_ld = 0;
div_clr = 1;
cn_ld = 0;
cn_clr = 1;

next_state = idle;
end
idle: begin
clk_out = 0;
div_ld = 0;
div_clr = 0;
cn_ld = 0;
cn_clr = 1;

if (ld) next_state = load;
else if (en) next_state = count;
else next_state = idle;
end
load: begin
clk_out = 0;
```

```
div_ld = 1;
div_clr = 0;
cn_ld = 0;
cn_clr = 0;

next_state = idle;
end
count: begin
clk_out = 0;
div_ld = 0;
div_clr = 0;
cn_ld = 1;
cn_clr = 0;

if (~en) next_state = idle;
else if (~cn_eq_div) next_state = count;
else next_state = high;
end
high: begin
clk_out = 1;
div_ld = 0;
div_clr = 0;
cn_ld = 0;
cn_clr = 1;

next_state = count;
end
endcase
end


endmodule
module clock_divider_datapath(
input logic clk,
input logic [15:0] div,
input logic  div_ld, div_clr, cn_ld, cn_clr,
output logic cn_eq_div
);

wire div_en;
wire [15:0] div_in, div_out;
or or1 (div_en, div_ld, div_clr);
mymux2 #(16) mux1 (div_clr, 0, div-1, div_in);
myregister #(16) div_reg (clk, div_in, div_en, div_out);

wire cn_en;
wire [15:0] cn_in, cn_out;
or (cn_en, cn_ld, cn_clr);
mymux2 #(16) mux2  (cn_clr, 1, cn_out + 1, cn_in);
myregister #(16) count_reg (clk, cn_in, cn_en, cn_out);

assign cn_eq_div = div_out == cn_out ? 1'b1 : 1'b0;

endmodule
```

2.1.2 Datapath for Page and Mode Information

```
module uart_7segment_pages_datapath(
```

```
input logic clk, np, pp, mc,
output reg mode = 1'b0, tr_page = 1'b0, rc_page = 1'b0,
output reg mode_page_leds
);

always_ff @ (posedge clk) begin
if (np)
if (mode)
rc_page <= ~rc_page;
else
tr_page <= ~tr_page;
if (pp)
if (mode)
rc_page <= ~rc_page;
else
tr_page <= ~tr_page;
if (mc)
mode <= ~mode;
mode_page_leds <= {mode,mode ? rc_page : tr_page, mode ? ~rc_page : ~tr_page};
end


endmodule
```

2.1.3 Double Reader

```
module double_reader(
input clk,
input page,
input [7:0] read,
output logic [1:0] add,
output logic [15:0] out
);

reg state = 0;

wire [15:0] out_in;

always_ff @ (posedge clk) begin
state <= ~state;
add <= {page, state};
out <= out_in;
end

mymux2 #16 mux1 (state, {out[15:8], read}, {read, out[7:0]}, out_in);


endmodule
```

2.1.4 Seven Segment Driver

```
module seven_segment_driver(
input clk,
input [15:0] data,
input load,
output [6:0] seg,
output [3:0] an,
```

```
output dp
);

localparam div = 150_000;

reg [32:0] counter = 0;

always @(posedge clk) begin
if(counter>=div)
counter <= 0;
else
counter <= counter + 1;
end

logic slow_clock;

assign slow_clock = (counter == div)?1'b1:1'b0;

assign dp = 1;

wire [3:0] hex;

hex4_led timer (
clk, slow_clock, data, load, hex, an
);

wire [6:0] leds;
assign seg = {<<{leds}};

hex_to_led transformer (hex, leds);



endmodule
module hex4_led(
input clk, slow_clk,
input [15:0] data,
input load,
output [3:0] hex,
output reg [3:0] pos
);

reg [1:0] counter_reg = 2'b00;
reg [15:0] data_reg;

always_ff @ (posedge clk) begin
data_reg <= data;
end

always_ff @ (posedge slow_clk) begin
counter_reg <= counter_reg + 1;
end

mymux4 #4 mux1 (counter_reg, data_reg[15:12], data_reg[11:8], data_reg[7:4], data_reg[3:

always_comb begin
if (counter_reg == 2'b00) pos = 4'b1110;
else if (counter_reg == 2'b01) pos = 4'b1101;
```

```
else if (counter_reg == 2'b10) pos = 4'b1011;
else pos = 4'b0111;
end

endmodule
module hex_to_led(
input logic [3:0] hex,
output logic [6:0] led
);

always_comb begin
if (hex == 4'h0) begin
led = 7'b0000001;
end
if (hex == 4'h1) begin
led = 7'b1001111;
end
if (hex == 4'h2) begin
led = 7'b0010010;
end
if (hex == 4'h3) begin
led = 7'b0000110;
end
if (hex == 4'h4) begin
led = 7'b1001100;
end
if (hex == 4'h5) begin
led = 7'b0100100;
end
if (hex == 4'h6) begin
led = 7'b0100000;
end
if (hex == 4'h7) begin
led = 7'b0001111;
end
if (hex == 4'h8) begin
led = 7'b0000000;
end
if (hex == 4'h9) begin
led = 7'b0000100;
end
if (hex == 4'hA) begin
led = 7'b0001000;
end
if (hex == 4'hB) begin
led = 7'b1100000;
end
if (hex == 4'hC) begin
led = 7'b0110001;
end
if (hex == 4'hD) begin
led = 7'b1000010;
end
if (hex == 4'hE) begin
led = 7'b0110000;
end
if (hex == 4'hF) begin
led = 7'b0111000;
```

```
end
end

endmodule
```

2.2 Automatic Mode Controller

```
module auto_mode_controller(
input clk,
input mode,
input go,
input tr_end,
output logic uart_go_out
);

logic uart_go;

reg [2:0] state, next_state;
reg [3:0] count, count_in;
reg [10:0] rest_count, rest_count_in;;

localparam single = 3'b000;
localparam ssend = 3'b001;
localparam swait = 3'b010;
localparam multi = 3'b011;
localparam mwait = 3'b100;
localparam msend = 3'b101;
localparam mletitrest = 3'b110;

always_ff @ (posedge clk) begin
state <= next_state;
count <= count_in;
uart_go_out <= uart_go;
rest_count <= rest_count_in;
end

always_comb begin
case (state)
default begin
next_state = single;
end
single: begin
uart_go = 0;
if (mode) next_state = multi;
else if (go) next_state = ssend;
else next_state = single;
end
ssend: begin
uart_go = 1;
next_state = swait;
end
swait: begin
uart_go = 0;
if (tr_end) next_state = single;
else next_state = swait;
end
multi: begin
uart_go = 0;
```

```
count_in = 4'b1000;
if (~mode) next_state = single;
else if (go) next_state = msend;
else next_state = multi;
end
mwait: begin
uart_go = 0;
rest_count_in = 0;
if (tr_end & ~(count == 0)) next_state = mletitrest;
else if (tr_end & (count == 0)) next_state = multi;
else next_state = mwait;
end
msend: begin
uart_go = 1;
count_in = count - 1;
next_state = mwait;
end
mletitrest: begin
if (rest_count == 10'h0ff) next_state = msend;
else rest_count_in = rest_count + 1;
end
endcase
end

endmodule
```

2.3 Button Debouncers

```
module button_debouncer(
input clk, btn_in,
output logic btn_out
);

reg [31:0] count;

always_ff @ (posedge clk) begin
if (btn_in & count < 10_000_100) count = count+1;
if (~btn_in) count = 0;
end

assign btn_out = (count == 10_000_000);
endmodule
```

2.4 Additional Modules

```
module mymux4
#(parameter w = 1)
(
input logic [1:0] s,
input logic [w-1:0] a, b, c, d,
output logic [w-1:0] o
);
assign o = s[1] ? ( s[0] ? a : b ) : ( s[0] ? c : d );
endmodule

module myregister_shift
#(parameter register_width = 8)
```

```
(
input clk,
input [register_width-1:0] q_prime,
input en,
input sh,
input dir_left,
input sh_in,
output reg [register_width-1:0] q
);

always @(posedge clk) begin
if (en) q <= q_prime;
if (sh) begin
if (dir_left) begin
q <= {q[register_width-2:0], sh_in};
end
else begin
q <= {sh_in, q[register_width-1:1]};
end
end
end
endmodule
module myregister
#(parameter register_width = 8)
(
input clk,
input [register_width-1:0] q_prime,
input en,
output logic [register_width-1:0] q
);

always @(posedge clk) begin
if (en) q <= q_prime;
end
endmodule
module myreg_clear
#(parameter w = 8)
(
input logic clk, en, clr,
input logic [w-1:0] in,
output reg [w-1:0] out
);

always_ff @(posedge clk) begin
if (clr) out <= 0;
else if (en) out <= in;
end

endmodule
module mymux4
#(parameter w = 1)
(
input logic [1:0] s,
input logic [w-1:0] a, b, c, d,
output logic [w-1:0] o
);
assign o = s[1] ? ( s[0] ? a : b ) : ( s[0] ? c : d );
endmodule
```