

6.111 Final Project Report: Global Domination

Bo Shi, David Wang

December 11, 2004

Abstract

This final project aims to create a floating globe complete with an atmospheric layer. Two LED-array arcs mounted on a rotating pole spin at a high frequency (approximately 30 Hz) so that the rotating arcs will appear to display a 3-dimensional floating earth and cloud cover.

Contents

1	Overview	1
2	Globe Assembly	1
3	Modules	3
3.1	Timing Module (David Wang)	3
3.2	Map Module (Bo Shi)	5
3.3	LED Controller (David Wang)	6
3.4	RS232 Module (Bo Shi)	8
4	Conclusion	8
5	Appendix: Verilog Source	10
5.1	wdtop.v (Top Level Module)	10
5.2	mapper.v (Map Module)	12
5.3	enc_fsm.v (Map Component)	14
5.4	encoder.v (Map Component)	15
5.5	pulse.v	17
5.6	rs232.v (RS232 Module)	17
5.7	step2pulse.v	20
5.8	sync8.v (8-bit Synchronizer)	20
5.9	sync.v (Synchronizer)	20
5.10	theta_counter.v (Timing Component)	21
5.11	theta_divider.v (Timing Component)	21
5.12	timing_fsm.v (Timing Module)	22
5.13	decoder (LED Driver Controller)	23
5.14	led_clock.v (LED Driver Component)	25
5.15	led_enable.v (LED Driver Component)	25

List of Figures

1	Structure of the mechanical assembly.	2
2	Block Diagram of Verilog components.	2
3	Timing Module FSM	3
4	Timing module simulation waveform.	4
5	One cycle of map operation.	5
6	Map caching simulation.	6
7	LED Driver Chip Layout	7
8	LED Controller Waveform.	8
9	RS232 module FSM.	9
10	RS232 Address selection.	9

List of Tables

1	Custom serial protocol	5
---	----------------------------------	---

1 Overview

The globe assembly contains a significant mechanical and Verilog component. By rotating and strobing a vertical arc of LEDs at high frequencies, the human eyes will perceive an image. The Verilog component of the system contains four major modules:

- *LED Driver Decoder Module* This module will rotate along with the LED-arcs, receiving instructions from the Map Module via commutator brushes and managing the LED driver circuitry. A combination of 8-bit latches and Darlington arrays will be driven by a set of programmed PAL22V10 chips.
- *Map Module* This module maps the image stored in memory to the LED array based on the position of the LED-arc. This module also serializes the data and sends it to the LED Driver Module.
- *RS232 Module* This module is a minor FSM that accepts serial data via RS-232 and writes to the RAM. A simple computer program will transmit complete RAM images to this module which will then alter the RAM as necessary. Hyperterminal is used to send test data.
- *Timing Module* This module keeps track of the status of the mechanical hardware and controls the timing of the LEDs. The `sync_pulse` signal is received from the mechanical assembly once every rotation. The Timing module will also keep track of the rotation speed of the system and adjust LED timing parameters as necessary, preventing minor perturbations from the intended rotation speed from affecting operation.

Figure 1 shows the mechanical component of this project. A bus for power and communications connects the lab kit and the mechanical structure by using commutator brushes.

Figure 2 show the organization of the modules. The LED Driver module is not shown. The `LED Control` signal shown in both figures is routed to the LED Driver module. The Map module receives the vertical position of the LED arcs `theta` and retrieves the appropriate addresses in memory representing LED on/off configuration for that position.

2 Globe Assembly

One of the most time consuming aspects of this project involves building the rotating assembly. In order to ensure the blurring of the LEDs, an AC motor provides 1800 RPM rotation at 1/25 HP (equivalent to a 30 frames per second refresh rate). The nature of the AC motor gives a minimal reduction in RPM until the rated horse power is met, making achieving 30 fps easier with little initial consideration for loading trade offs. Nevertheless, the assembly incorporates several precautions to ensure a reduction in rotational friction. These precautions include two bearings to reduce vibrational losses and the selection of a serial instead of parallel data transfer method to reduce the friction from a large number of commutator brushes.

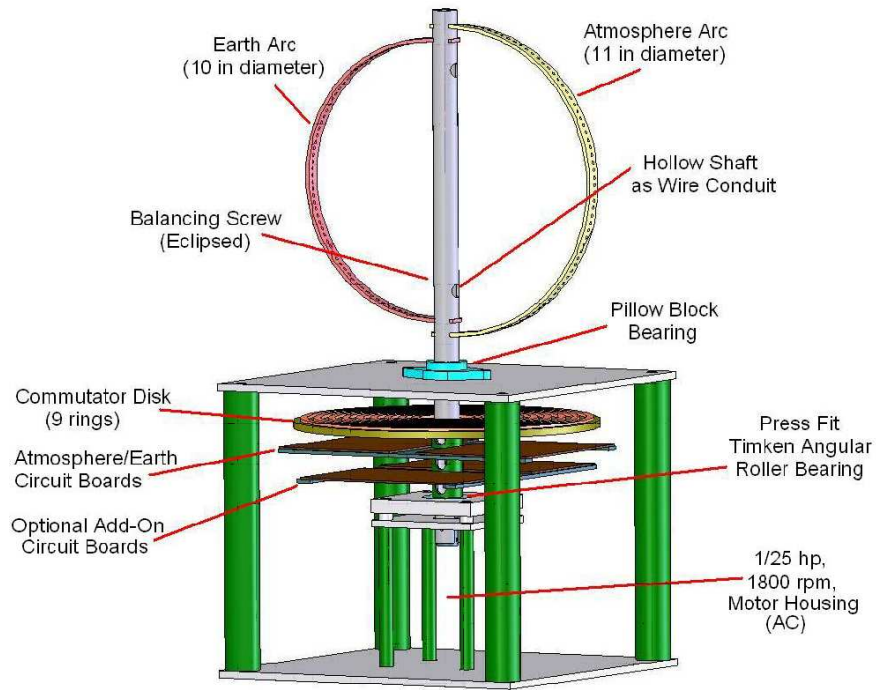


Figure 1: Structure of the mechanical assembly.

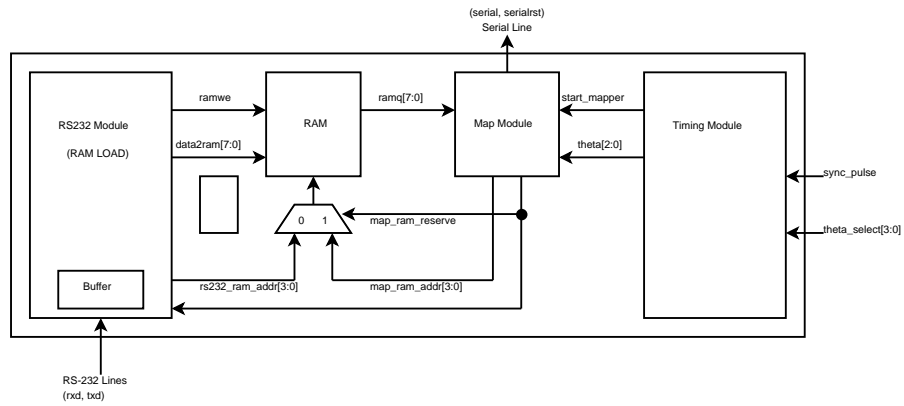


Figure 2: Block Diagram of Verilog components.

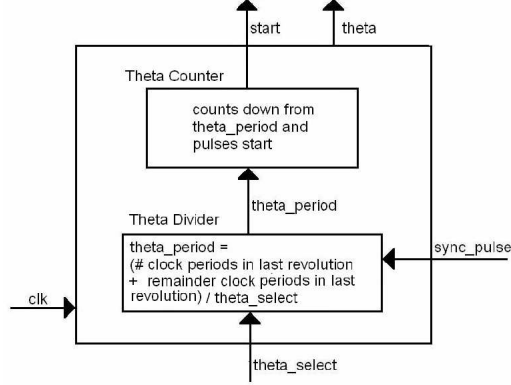


Figure 3: Timing Module FSM

The final design requires 7 commutator brushes: 1 brush for power, 1 for ground, and two pairs of serial data and reset signals coming from two lab kits (1 pair providing data for each arc). The final commutator brush sweeps past a 5V pin, designed to produce a 5V pulse for every revolution of the assembly. The result is 2 power signals, 4 signals into the assembly and one **sync_pulse** coming out of the assembly.

The 4 signals into the assembly are connected through Schmitt triggers in order to clean up the signals prior to being communicated through the commutator disk. 8 sets of 8 bus lines run from these boards through the central rotating shaft to each arc, allowing 64 LEDs per arc to be independently controlled via the LED Driver circuits. Specifically, these bus lines connect the Darlington arrays to the negative lead of the LEDs, requiring the positive leads to be wired to 5V. In order to reduce redundant bus lines, the positive leads of the LEDs connect to the globe assembly's arcs, which through a metallic bearing with conductive lubricant, are shorted to a 5V supply.

To ensure good data fidelity, Schmidt triggers are incorporated on the signal lines before being connected to the commutator brushes in a separate electronics box housing (not depicted in Figure 1). The sync pulse is also piped through a Schmitt trigger in the same box before being output to the lab kits.

The resulting assembly is depicted in Figure 1, minus the complexity of the wiring. The entire assembly measured about 26 inches tall with a 12 inch square base.

3 Modules

3.1 Timing Module (David Wang)

The timing module has the responsibility of telling the map module which theta (slice along the longitude) to draw and when it should begin drawing that theta. The map module then has the responsibility to work with the serial encoder and transmit data as fast as possible to send 8 packets of 8 bit data, to update phi, the LEDs in one arc. If the map module is unable to complete sending the 8 packets of data through the serial encoder, the timing

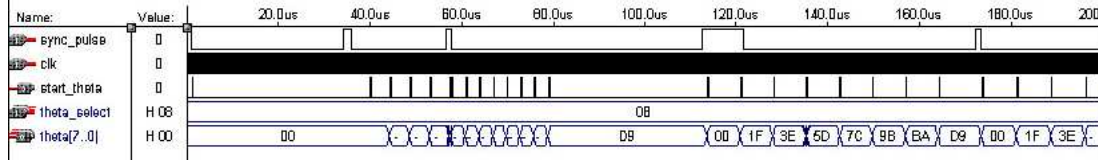


Figure 4: Timing module simulation waveform.

module simply prompts the map module to start the next theta.

The timing module comes up with an appropriate duration for each theta slice to be displayed by taking in three important inputs: **sync_pulse**, **theta_select**, and **clk**. Note that no reset is necessary. **sync_pulse** is a signal from the globe assembly that goes high for an undetermined number of clock periods once every revolution. **theta_select**, allows the user to select how many theta slices should get drawn per revolution, effectively allowing the user to select the horizontal resolution.

The module works in three functioning blocks: Theta Divider, Theta Counter, and then some input and output logic within the overall Timing Module block.

The two non synchronous inputs **sync_pulse** and **theta_select** are first sent through a series of registers to sync them with the lab kits 10 Mhz clock. The synced inputs are then sent to Theta Divider.

Theta Divider counts the number of clock periods between the two rising edges of the sync pulse adds it to the remainder from the last division and divides the total by **theta_select** (addition of the remainder from the last **sync_pulse** is necessary to prevent error build up causing the rotating image to get skewed). The continuous recalculation of **theta_period** by the timing module is necessary to counter act slow RPM variations that can come from the motor.

The resulting **theta_period** is sent to Theta Counter, where a simple counter outputs a start pulse every time it counts up to **theta_period**.

The signal **theta** is calculated in the overall Timing Module block by dividing 256 (the number of slices stored in memory) by **theta_select** and storing the value into **theta_increment**. At the start of every **sync_pulse**, theta is zeroed and **theta_increment** is added to **theta** at every start pulse. Some additional mux logic is used to ensure that theta can never reach a value greater than 256.

Figure 4 depicts the timing module's simulation. The **sync_pulse** is spaced erratically on purpose to provide a more robust verification of the logic's performance. Notice that after the very first **sync_pulse** there is no change in **theta_select** or **start_theta**, this is because the timing module is designed to calculate theta parameters off of data collected from the previous revolution. A reset is therefore not necessary, because at 1800 RPM, it will take a maximum of 1 complete revolution for the LEDs to display appropriately, a comparably small fraction of time. After the second sync pulse, **start_theta** begins to pulse, assuming that the next revolution will take place in the same amount of time as the previous one. Unfortunately, the third sync pulse comes much faster, allowing only 4 **start_theta** pulses to get produced. After the third sync pulse, **start_theta** begins to pulse again, this time at

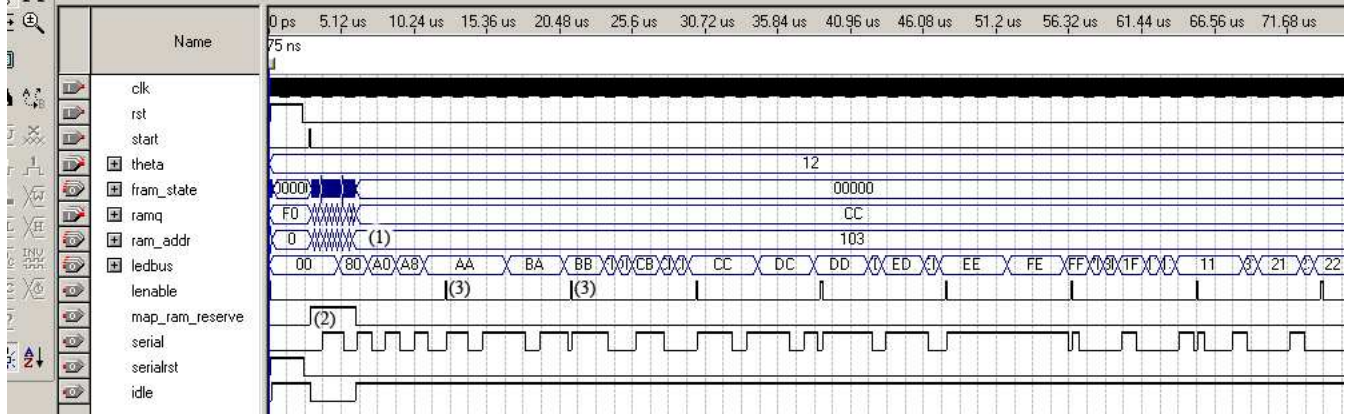


Figure 5: When the mapper receives a start pulse, it begins simultaneous operations of caching data from the RAM and serializing 64 bytes of data.

Table 1: Lab-Kit to Assembly Serial Protocol using a 10 Mhz clock.

ID	Start Bit	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Duration	5	10	10	10	10	10	10	10

a much higher rate, because the previous revolution time was so short. After `start_theta` pulses 8 times, as dictated by `theta_select`, it stops. However, at the fourth `sync_pulse`, the space of the `sync_pulses` become more regular resulting in a more equitable spacing the `start_theta` pulses.

3.2 Map Module (Bo Shi)

Figure 5 shows a simulation of the activity in the Map module after a start pulse is received. The high level of activity on `ramq` and `ram_addr` labeled as (1) is the caching operation. While the RAM is being accessed, the `map_ram_reserve` signal is asserted as shown near label (2). Label (3) on the `lenable` signal coincide with valid data on the `ledbus` signals above.

The encoder module serializes one byte of data (Figure 1). While it is operating, it outputs a high busy signal and does not accept any other commands. The data protocol used between the lab kit and the mechanical assembly is asynchronous and runs at approximately 115 Kbs. The format is as follows: 1 start bit followed by 8 bits beginning with the MSB. Using a 10 Mhz clock, The start bit is high and lasts a duration of five clock cycles. Each bit lasts 10 clock cycles. The encoder is implemented using a counter which counts 85 cycles and outputs the serial signal based on the time division discussed before.

The mapper module is responsible retrieving and serializing LED control signals. Two operations are initiated. The mapper first begins caching of 8 adjacent bytes of data from RAM. The beginning address is calculated from the `theta` parameter from the Timing FSM as follows: $address = theta * 8 + i$ where i is an integer from zero to seven. Since serializing

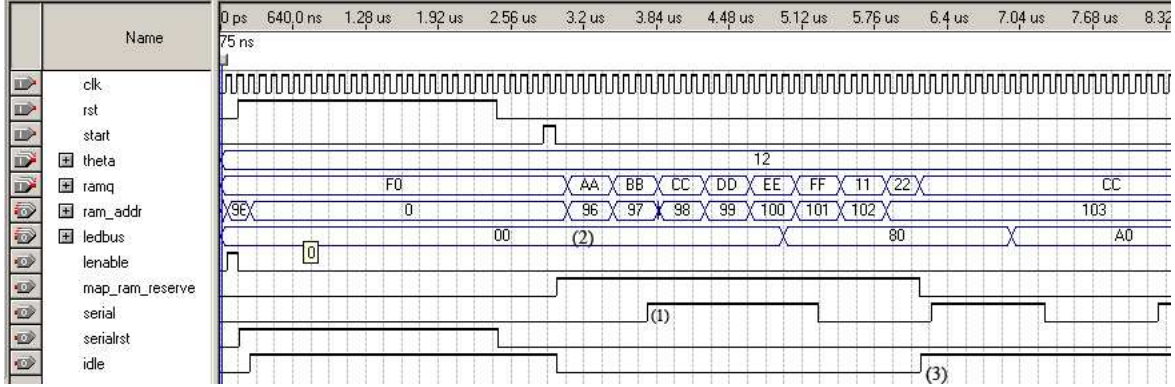


Figure 6: This figure is a close-up of Figure 5’s caching operation. Label (1) shows the serial signal. As can be seen from signals above label (2), data 0b10101010 is accessed from address 96 and the serial signal coincides with this.

one byte of data takes considerably longer than retrieving one byte of data from RAM, the serializing process will begin. The encoder module is controlled by the encoder FSM which initiates the encoder and waits for it to complete for 8 bytes. Figure 5 shows the overall process.

3.3 LED Controller (David Wang)

The LED Driver module is best represented in the circuit diagram depicted in Figure 7. This module has the responsibility of decoding the serial data and latching it onto the appropriate LEDs. While this might sound simple to do, the implementation is made difficult because of a 10 register limitation imposed by the PALs.

The final division of the code requires 3 PALs. One PAL used to serve as the Decoder FSM (Figure 7), another as the clock/counter to store the state of the Decoder FSM (Figure 7), and the final as the Enable, to enable one latch at a time to latch the data on the bus lines out of the decoder. It is important to note that the circuitry shown in Figure 7 exists only on the circuit boards of the Assembly, and is duplicated: one to drive the Earth Arc and another to drive the Atmosphere Arc. Furthermore, the circuitry is driven by a 10Mhz clock independent of the off board clock controlling the lab kit’s FPGA.

The architecture shown in Figure 7 is designed to update 8 sets of 8 LEDs at a time (one arc). At the start pulse of the `serial_in`, the Decoder PAL sets the `reset_Clk` signal to low, allowing the Clock PAL to begin counting. Using the counter vales from the Clock PAL, the Decoder PAL latches the serial data on the LED 0 bus line after 10 clock periods, on LED 2 bus line after an additional 10 clock periods and so on. At the end of 85 clock periods, the data on the bus lines LED[7:0] should all be latched to either 1 or 0. At this point the Clock PAL would increment `select[2:0]` to the next set of 8 LEDs to get updated, and the Decoder would send a `L_Enable` pulse to Enable PAL. Using the `select` and `l_enable` signals, the Enable PAL, then selects the appropriate latch to enable, and gives the latch

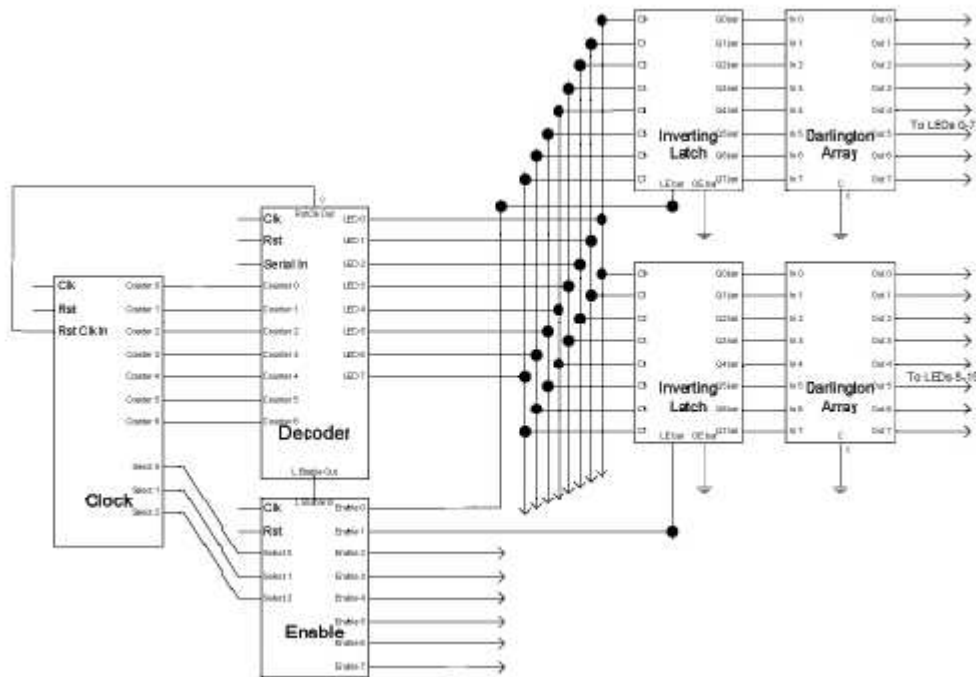


Figure 7: Two pairs of Latches and Darlington Arrays are shown of a total of 8 pairs. The entirety of this circuit would need to be used twice, once to control the 64 LEDs on the Earth Arc and another to control the 64 LEDs on the Atmosphere Arc.

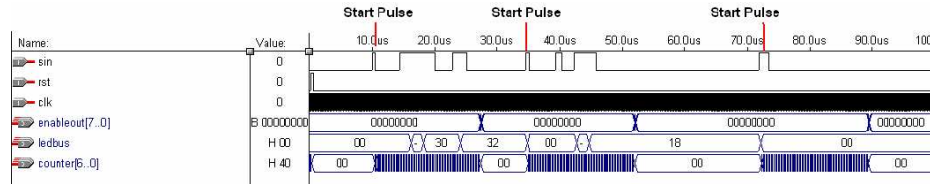


Figure 8: The waveform shown in this figure depicts three start pulses and the ideal behavior of the digital logic on the three PALs. After the initial reset pulse, the first start pulse starts the **counter** in the Clock PAL. This causes values to be latched onto the **LEDbus** in the Decoder PAL every 10 clock cycles. Once the clock counts up to 85 clock cycles, the value of **enableout** changes for one clock period to make the appropriate latch store the value on the **LEDbus**.

one pulse so that it may latch the value of the LED bus line. From here, the latched data is simply passed to the Darlington array, where the Darlington circuitry serves as a voltage to current switch, driving current to the appropriate LEDs whenever the input voltage is high. At the next clock period, the Decoder PAL uses a register to store that it is in an idle state, and continuously sends a high `reset_clk` signal to the Clock PAL, forcing the Clock PAL to continuously be reset until the next start pulse of the `serial_in`. This process repeats continuously for each arc, requiring 8 sets/packets of 8 bit data to be decoded per theta slice.

3.4 RS232 Module (Bo Shi)

The RS232 module implements the expansion capabilities of the project. It has the capability to modify the contents of the RAM when the Map module is not busy. In Figure 2, the 2-bit mux selects which address to use among the Map module and the RS232 module.

The FSM of the RS232 module is shown in Figure 9. Since the Map module has priority access to the RAM, RS232 data is buffered using a FIFO MegaFunction. The FIFO module has a write request signal which is enabled when the RS232 module signals that data is ready on the buffer. When data is ready, 2 clock cycles (states UPDATE1 and UPDATE2) are devoted to fetching data from the FIFO and writing the data to the RAM.

The RS232 module writes data sequentially in RAM. When the module is reset, the internal address counter is set to zero. Each time a byte is written to RAM, the counter increments. Consequently, in order to make any changes, a computer program must write all 2048 bytes. Unfortunately, this is somewhat inefficient. A better protocol would perhaps be able to specify a 10-bit address in addition to 8-bits of data to write to said address.

4 Conclusion

The prototype assembly resulting from the project was only able to drive 8 LEDs on the atmosphere arc unreliably. While this project did have many important aspects of digital design including, error accommodation, fitting coded logic into hardware constraints, serial

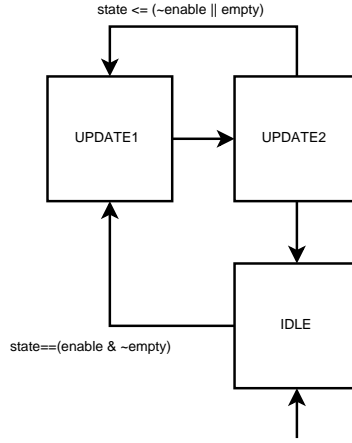


Figure 9: The `enable` signal comes from the Map module. This signal is the inverse of the `map_enable_ram` signal and indicates when the RS232 Module has permission to access the RAM. The `empty` signal comes from the FIFO module. RAM is only updated (states `UPDATE1` and `UPDATE2`) when the FIFO is not empty and the Map module is not busy. When the `enable` signal goes low, the FSM goes into the `IDLE` state as quickly as possible, meaning that it will finish any RAM update and stop.

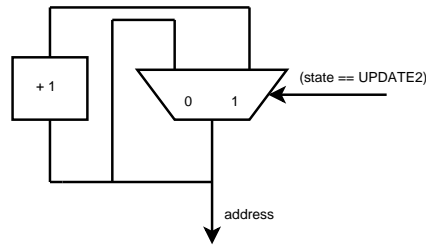


Figure 10: The address is incremented each time the FSM enters the `UPDATE2` state.

data transfer, and timing of memory usage, it had a very large mechanical component that ended up providing many large time intensive mechanical problems. In simulations on the lab kit, the LEDs performed with great reliability, but loss of data fidelity over the commutator brushes despite sequences of Schmidt triggers made the final assembly perform poorly.

A large mistake was the assumption that the integration of the hardware and digital logic would be simple once each portion was completed. The original design strategy was therefore an attempt to get all 128 LEDs working in the first try. After many attempts, a prototype featuring only 8 LEDs was assembled, allowing us to verify in the time remaining that it was commutator unreliability that resulted in the noisy displays.

While the commutators proved to be the final obstacle to overcome, several intermediate obstacles present themselves.

One of these were glitchy signals that came from the PALs in the LED Driver Module as a result of the high 10 Mhz clock frequency. Ultimately, the glitches occurred at time intervals that did not violate any setup and hold times, allowing the performance of the LED Driver Module to be unhindered. Another problem was that the plastic support for the commutator rings was collecting static electricity that affected signals on the commutator brushes. A grounding wire seemed to correct this problem.

As mentioned, one of the primary challenges was signal integrity between the lab kit and the mechanical assembly. Unusable sync pulse signals were fixed by using a low resistance long lever switch which did not require a commutator brush. A possible solution to the other signal problems is to use wireless communication. By mounting an IR receiver at the top of the shaft of the assembly, one could eliminate both friction and noise from commutator brushes.

5 Appendix: Verilog Source

5.1 wdtop.v (Top Level Module)

```
module wdtop(clk, rst, theta_select, rxd, txd, sync_pulse, serial, serialrst
// debug
,rready
);
output rready;
// end debug
input clk, rst;
input [7:0] theta_select;
input rxd, sync_pulse;
output txd, serial, serialrst;

wire [7:0] theta;
wire [7:0] data2ram, ramq;
wire start_mapper;
```

10

```

wire ramwe;
wire map_ram_reserve;
wire [10:0] rs232_ram_addr, map_ram_addr;
wire [10:0] address;
wire reset;

```

20

```

sync syncrst(clk, rst, reset);

```

```

timing_fsm major_mod(
    .clk(clk),
    .sync_pulse(sync_pulse),
    .theta_select(theta_select),
    .theta(theta),
    .start_theta(start_mapper)
);

```

30

```

// mux selector for address
assign address = map_ram_reserve ? map_ram_addr : rs232_ram_addr;

```

```

rs232 modrs232(
    .clk(clk),
    .reset(reset),
    .rxd(rxd),      // toplevel input (1)
    .txd(txd),
    .enable(~map_ram_reserve), // controls off/on operation of this module(1)
    .ramwe(ramwe),  // ram control signal (1)
    .data2ram(data2ram), // ram write data (8)
    .ram_addr(rs232_ram_addr) // (11)
    // debug
    ,.rready(rready)
);

```

40

```

ram modram(
    .address(address),
    .we(ramwe),
    .data(data2ram),
    .q(ramq)
);

```

50

```

mapper gmap(
    .clk(clk),

```

```

.reset(reset),
.start(start_mapper), // from dave's module
.ramq(ramq),
.theta(theta), // from dave's module
.ram_addr(map_ram_addr),
.map_ram_reserve(map_ram_reserve),
.serialout(serial), // toplevel output
.serialrst(serialrst) // toplevel output
);

```

60

endmodule

5.2 mapper.v (Map Module)

```

// contains mapper and encoder
// mapper has a cache

```

```

// address algorithm: theta * 8 + phi
module mapper(clk, reset, start, ramq, theta,
    ram_addr, map_ram_reserve, serialout, serialrst
    // debug
    //,idle, fram_state
);
    //output idle;
    //output [4:0] fram_state;
    // end debug

```

10

```

// the signal which goes to the mux that selects an address
// from rs232 module or this module
output map_ram_reserve;

```

```

input clk, reset;
input start; // a pulse to command the mapper to cache 8 bytes
input [7:0] theta, ramq;
output serialout, serialrst;
output [10:0] ram_addr;

```

20

```

wire serialout, serialrst;
// internal signals
wire send; // the control signal to make the encoder send a byte
wire enc_busy; // the encoder is busy, making the encorder fsm wait
wire [7:0] led_data; // the data the encoder uses (taken from cache by encoder fsm)

```

30

```

reg map_ram_reserve;
reg start_enc_fsm;
reg [10:0] ram_addr;
reg [63:0] cache;
reg idle; // this idle represents the status of caching
// ram stuff
// first two bits are a 4 cycle delay
// three high order bit are 0-7 counter
reg [4:0] fram_state; // "fetch"ram state

```

40

```

// encoder module, serializes byte data
encoder genc_mod(.clk(clk), .rst(reset), .send(send),
    .data(led_data), .busy(enc_busy), .sout(serialout));
// encoder module controller, sends 8 bytes when told to start
enc_fsm genc_ctl(.clk(clk), .rst(reset), .start(start_enc_fsm),
    .enc_busy(enc_busy), .cache(cache), .out(led_data), .enc_en(send));

```

```

assign serialrst = reset;

```

50

```

always @ (posedge clk) begin
    if(reset) begin
        idle <= 1;
        fram_state <= 0;
        ram_addr <= 0;
        start_enc_fsm <= 0;
    end
    if(start) begin // fetching ram
        idle <= 0;
        fram_state <= 0;
        map_ram_reserve <= 1;
    end

```

60

```

    if(~idle) begin
        if(fram_state != 5'b11111) fram_state <= fram_state + 1;
        if(fram_state[1:0] == 2'b10) case (fram_state[4:2])
            7: cache[63:56] <= ramq;
            6: cache[55:48] <= ramq;
            5: cache[47:40] <= ramq;
            4: cache[39:32] <= ramq;
            3: cache[31:24] <= ramq;

```

70


```

    2: cache[23:16] <= ramq;
    1: cache[15:8] <= ramq;
    0: cache[7:0] <= ramq;
endcase
// change the address
// the ram address is calculated as follows:
// address = theta * 8 * latitude
// where latitude is between 0 & 7
if(fram_state[1:0] == 2'b00) ram_addr <= theta * 8 + fram_state[4:2];    80
// we are done caching, reset
if(fram_state == 5'b11111) begin
    map_ram_reserve <= 0;
    idle <= 1;
    fram_state <= 0;
end
// since serializing data is very slow, and accessing memory
// is fast, once we have the first byte of memory cached, we
// can begin encoding (the 3 high order bits equal 2)    90
start_enc_fsm <= (fram_state == 5'b00100);
end
end
endmodule

```

5.3 enc_fsm.v (Map Component)

```

module enc_fsm(clk, rst, start, enc_busy, cache, out, enc_en);
    input clk, rst;
    input start; // start pulse to make this guy start up
    input [63:0] cache; // 8-byte register file
    input enc_busy;
    output [7:0] out; // the byte to serialize (sent to encoder)
    output enc_en; // start pulse for encoder

    // state[4] done bit/idle    10
    // state[3:1] counter
    // state[0] wait/nowait
    reg [4:0] state;

    // handles control signal for encoder
    wire enc_en;
    step2pulse gs2p(clk, (~rst && ~state[4] && ~state[0]), enc_en); // step to pulse conversion

```

```

// handles data for encoder
assign out = (state[3:1] == 7) ? cache[63:56] :
    (state[3:1] == 6) ? cache[55:48] :
    (state[3:1] == 5) ? cache[47:40] :
    (state[3:1] == 4) ? cache[39:32] :
    (state[3:1] == 3) ? cache[31:24] :
    (state[3:1] == 2) ? cache[23:16] :
    (state[3:1] == 1) ? cache[15:8] : cache[7:0]; // last one is for state[3:1] == 0

always @ (posedge clk) begin
    if(rst) state <= 5'b10000;
    else begin
        if(start) state <= 0;
        // if not idle
        // state transitions
        if(~state[4] && ((enc_busy && ~state[0]) || (~enc_busy && state[0]))) state <= state[3:1];
    end
end
endmodule

```

5.4 encoder.v (Map Component)

```

// Clock rate: 10 Mhz
// Protocol: [start bit] [data(7:0)]
// Optional parity bit may be used as an extension
module encoder(clk, rst, send, data, busy, sout);
    input clk, rst, send;
    input [7:0] data;
    output busy, sout;

    // When the send pulse is sent, data is latched and
    // serialized.

    // each byte lasts 85 clock cycles:
    parameter DURATION = 85;
    // parameter BITT = 10;
    // parameter STARTT = 5;

    reg sout;
    reg idle;

```

```

reg [7:0] buffer;
reg [6:0] ct;

assign busy = ~idle;

always @ (posedge clk) begin
    if(rst) begin
        ct <= DURATION;
        idle <= 1;
        sout <= 0;
    end
    else begin
        // begin Tx
        if(idle && send) begin
            idle <= 0;
            buffer <= data;
        end

        if(~idle) begin
            ct <= ct - 1;
            if(ct == 0) begin
                idle <= 1;
                sout <= 0;
                ct <= DURATION;
            end
            if(ct <= 127 && ct > 85) sout <= 0;
            if(ct <= 85 && ct > 80) sout <= 1;
            if(ct <= 80 && ct > 70) sout <= buffer[7];
            if(ct <= 70 && ct > 60) sout <= buffer[6];
            if(ct <= 60 && ct > 50) sout <= buffer[5];
            if(ct <= 50 && ct > 40) sout <= buffer[4];
            if(ct <= 40 && ct > 30) sout <= buffer[3];
            if(ct <= 30 && ct > 20) sout <= buffer[2];
            if(ct <= 20 && ct > 10) sout <= buffer[1];
            if(ct <= 10 && ct > 0) sout <= buffer[0];
        end
    end
end
endmodule

```

5.5 pulse.v

```
module pulse(clk, sync`pulse, sync`pulse`p, sync`pulse`reset);

input clk, sync`pulse;
output sync`pulse`p;
output sync`pulse`reset;

reg sync`pulse`reset;
reg sync`pulse`p;

always @ (posedge clk)
  begin
    sync`pulse`p <= (sync`pulse & !sync`pulse`reset) ? 1 : 0;
    sync`pulse`reset <= sync`pulse ? 1 : 0;
  end

endmodule
```

10

5.6 rs232.v (RS232 Module)

```
// receives a RS232C data from rxd and outputs the data to io[7:0]
// and also transmits data+1 to txd.
// clk : 10Mhz
// Baud rate is set to 115200
// Parity : none
// Data bits : 8
// Stop bit : 1
// Flow control : None
module rs232(
  clk,
  reset,    // 1
  rxd,     // 1
  txd,
  enable,   // controls off/on operation of this module(1)
  ramwe,    // ram control signal (1)
  data2ram, // ram write data (8)
  ram`addr // (11)
  // debug
  ,rready
);
output rready;
// end debug
```

10

20

```
////////////////////////////////////
```

```
input clk;  
input rxd;  
input reset;
```

```
output txd;
```

```
input enable; // enable from MAP MODULE
```

30

```
output ramwe; // ram write enable  
output [7:0] data2ram;  
output [10:0] ram`addr; // ram address
```

```
parameter IDLE = 0;  
parameter UPDATE1 = 1;  
parameter UPDATE2 = 2;
```

```
wire rready; // receiver ready, data comes out on the next clock cycle  
wire tbusy; // rs232 transmit signals  
wire [7:0] rdata; // receiver data
```

40

```
wire [7:0] data2ram;  
// for the main RAM unit  
wire ramwe;  
// FIFO status signals  
wire full, empty;  
// FIFO control signals  
wire rdreq;  
reg wrreq;  
reg tstart;
```

50

```
reg [1:0] state;  
reg [10:0] ram`addr;  
reg [7:0] buff;
```

```
assign ramwe = (state == UPDATE1 || state == UPDATE2);  
assign rdreq = (state == UPDATE1);
```

60

```
wire [7:0] ramq;  
ram test (  
    ram`addr,  
    ramwe,
```

```

    data2ram,
    ramq);

// rs232 reciever module
rs232c`receiver rs232r(.clk(clk),.rxdin( rxd), .data(rdata), .ready(rready));
// transmit module: dumps transmitted data right back
rs232c`transmitter rs232t(.clk(clk), .txdout(txd), .data(rdata), .start(tstart), .busy(tbusy))

// FIFO
fifo qfifo(
    .data(buff),
    .wrreq(wrreq),
    .rdreq(rdreq),
    .clock(clk),
    //sclr(reset), // available on quartus, not on max
    .q(data2ram),
    .full(full),
    .empty(empty));

always @(posedge clk) begin
    if (reset) begin
        state <= IDLE;
        buff <= 0;
        ram`addr <= 0;
    end
    else begin
        buff <= rready ? rdata : buff;
        // data (buff) is ready the clock cycle after the 'rready' pulse
        // we use a register for wrreq to make a pulse for write request
        // to FIFO
        wrreq <= rready;
        tstart <= rready; // for readability
    end

// transition diagram
case(state)
    UPDATE1: state <= UPDATE2;
    UPDATE2: state <= (~enable || empty) ? IDLE : UPDATE1;
    IDLE: state <= (enable && ~empty) ? UPDATE1 : IDLE;
endcase

// outputs

```

```

    if(state == UPDATE2) ram'addr <= ram'addr + 1;
end

endmodule

```

110

5.7 step2pulse.v

```

module step2pulse(clk, in, out);
    input in, clk;
    output out;
    reg r;
    assign out = (in && ~r);
    always @ (posedge clk) r <= in;
endmodule

```

5.8 sync8.v (8-bit Synchronizer

```

module sync8(clk,in[7:0],out[7:0]);
    input clk;
    input [7:0] in;
    output [7:0] out;

    reg [7:0] out;
    reg [7:0] temp;

    always @ (posedge clk)
begin
    temp[7:0] <= in[7:0];
    out[7:0] <= temp[7:0];
end

endmodule

```

10

5.9 sync.v (Synchronizer

```

module sync(clk, in, out);
    input clk, in;
    output out;
    reg r1, out;
    always @(posedge clk) begin
        r1 <= in;
        out <= r1;
    end
end

```

endmodule

10

5.10 theta_counter.v (Timing Component)

module theta_counter(clk,start_counter,theta_period,change_theta);

input clk,start_counter;

input [23:0] theta_period;

output change_theta;

reg [23:0] counter_theta_period;

reg change_theta;

reg change_theta_reset;

10

always @ (posedge clk)

begin

if ((& (~counter_theta_period)) || start_counter)

begin

counter_theta_period <= theta_period;

change_theta_reset <= 1;

change_theta <= change_theta_reset ? 0 : 1;

end

else

begin

counter_theta_period <= counter_theta_period - 1;

change_theta <= 0;

change_theta_reset <= 0;

end

end

20

endmodule

5.11 theta_divider.v (Timing Component)

module theta_divider(clk,sync_pulse,theta_select,
theta_period,start_counter);

input clk, sync_pulse;

input [7:0] theta_select;

output [23:0] theta_period;

output start_counter;


```
wire [7:0] theta_remainder;
```

10

```
// at 20 fps with a 10Mhz clk, there will be about 500,000 clk periods per  
// rotation at 1 fps with a 10Mhz clk, there will be about 10,000,000 clk  
// periods per rotation
```

```
reg [23:0] counter; // this is equivalent to  $2^{14} = 16384$  max.
```

```
reg [23:0] rotation_period;
```

```
reg start_counter;
```

```
always @ (posedge clk)
```

```
begin
```

```
    if (sync_pulse)
```

20

```
        begin
```

```
            rotation_period <= counter + theta_remainder;
```

```
            counter <= 0;
```

```
            start_counter <= 1;
```

```
        end
```

```
    else
```

```
        begin
```

```
            counter <= counter + 1;
```

```
            start_counter <= 0;
```

```
        end
```

30

```
end
```

```
divider_p24 divider_c(rotation_period, theta_select,  
    theta_period, theta_remainder);
```

```
endmodule
```

5.12 timing_fsm.v (Timing Module)

```
module timing_fsm(clk, sync_pulse, theta_select, theta, start_theta);
```

```
    input clk, sync_pulse;
```

```
    input [7:0] theta_select;
```

```
    output [7:0] theta;
```

```
    output start_theta;
```

```
    wire [23:0] theta_period;
```

```
    wire [7:0] theta_select's;
```

10

```
    sync_sync_pulse(clk, sync_pulse, sync_pulse's);
```

```
    sync8_theta_select(clk, theta_select, theta_select's);
```

```

pulse_pulse_sync_pulse
    (clk, sync_pulse_s, sync_pulse_sp);

theta_divider theta_divider_c
    (clk, sync_pulse_sp, theta_select_s, theta_period, start_counter);

theta_counter theta_counter_c
    (clk, start_counter, theta_period, change_theta);

divider_256 divider_c
    (8'd255, theta_select_s, theta_increment, theta_remainder);

reg [7:0] theta;
reg start_theta;
reg delay_theta;
reg [7:0] theta_reset_counter;

always @ (posedge clk)
begin
    delay_theta <= (!(theta_reset_counter >= theta_select_s))
        ? change_theta : 0;
    start_theta <= delay_theta;
    theta_reset_counter <= start_counter
        ? 0 : change_theta
        ? theta_reset_counter + 1 : theta_reset_counter;
    if(theta_reset_counter==0)
        theta <= 0;
    else
        if (!(theta_reset_counter >= theta_select_s) && change_theta)
            begin
                theta <= theta + theta_increment;
            end
end

endmodule

```

5.13 decoder (LED Driver Controller)

```

// Clock rate: 10 Mhz
// Protocol: [start bit] [data(7:0)]
// Optional parity bit may be used as an extension
// [start bit] duration is 5 clock cycles

```

```

module decoder(clk, rst, sin, ct, reset`clk, ledbus, lenable);

    input clk, rst, sin;
        input [6:0] ct;
        output reset`clk;
    output [7:0] ledbus;
    output lenable; // Latch counter enable

    // each byte transmission lasts 85 ticks
    // we sample 5, 15, 25, 35, 45, 55, 65, 75
    // from encoder
    // parameter BIT`T = 10;
    // parameter START`T = 5;

    reg [7:0] ledbus;
    reg lenable; // serves as lenable
    reg idle; // serves as clock reset
    assign reset`clk = idle;

    always @ (posedge clk) begin
        // lencoder signal is a pulse
        lenable <= rst ? 0 : (ct == 7'd84) ? 1 : 0;

        if (rst)
            idle <= 1;
        else begin
            if(idle && sin) begin
                idle <= 0; // start bit recieved
                ledbus <= 0;
            end
            if(~idle) begin
                case(ct)
                    9: ledbus[7] <= sin;
                    19: ledbus[6] <= sin;
                    29: ledbus[5] <= sin;
                    39: ledbus[4] <= sin;
                    49: ledbus[3] <= sin;
                    59: ledbus[2] <= sin;
                    69: ledbus[1] <= sin;
                    79: ledbus[0] <= sin;
                    default: ; // do nothing
                endcase
            end
        end
    end

```

```

        endcase
        idle <= (ct == 7'd84) ? 1 : 0;
    end
end

    end
endmodule

```

50

5.14 led_clock.v (LED Driver Component)

```

module led_clock(clk, rst, reset_clk, select, counter);

    input clk, rst, reset_clk;
    output [6:0] counter;
    output [2:0] select;

    reg [6:0] counter;
    reg [2:0] select;

    always @ (posedge clk)
    begin
        if (rst || reset_clk)
            counter <= 0;
        else
            counter <= counter + 1;
            select <= rst ? 3'b111 : (counter == 7'd84) ? select + 1 : select;
    end

endmodule

```

10

5.15 led_enable.v (LED Driver Component)

```

module led_enable(clk,rst,lenable, select, enableout);

    input clk, rst, lenable;
    input [2:0] select;
    output [7:0] enableout;

    reg [7:0] enableout;

    always @ (posedge clk)
    begin

```

10

```

if (rst)
    enableout <= 0;
else
    case (select)
        0: enableout[0] <= lenable;
        1: enableout[1] <= lenable;
        2: enableout[2] <= lenable;
        3: enableout[3] <= lenable;
        4: enableout[4] <= lenable;
        5: enableout[5] <= lenable;
        6: enableout[6] <= lenable;
        7: enableout[7] <= lenable;
    endcase
end

endmodule

```

20