

## Procedure

This lab is comprised of three tasks. The first task involved creating a 32x4 RAM module, simulating inputs using peripherals.

### Task 1

I approached this task by first studying the block diagram given in the lab specification for the task. From this schematic, I extrapolated the input and output ports for the RAM module, then began working to define the internal logic. This involved creating the internal memory array, clocking inputs, and defining write logic.

After this module was developed and tested in ModelSim, a separate top-level module was created to instantiate an instance of the previous RAM module, and to define connections between board peripherals, and HEX displays. These modules were made separate as to make the RAM module more flexible – the control lines for the RAM module are not tied to any particular type of input source.

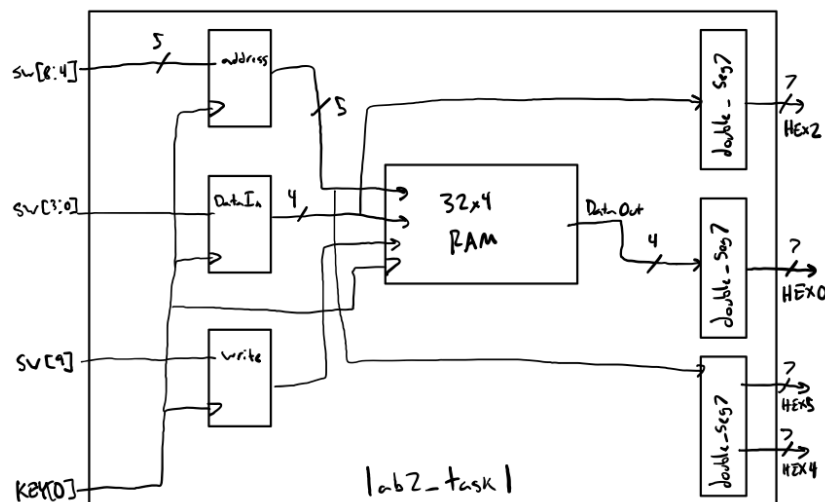


Figure 1. Top-Level Block Diagram for Lab 2 Task 1.

## Task 2

I approached this task by first following the directions to create an on-chip, dual-port, 32x4 RAM module in Quartus along with a memory initialization file to provide initial values for instances of this RAM module. I then created a top level module in Verilog to instantiate this RAM module and define connections between board peripherals and displays. Additionally, I included a five-bit counter in this top-level module that would continuously increment on every cycle of a divided clock. This was done to achieve the one second, periodic cycling through of reading each memory address in the RAM module.

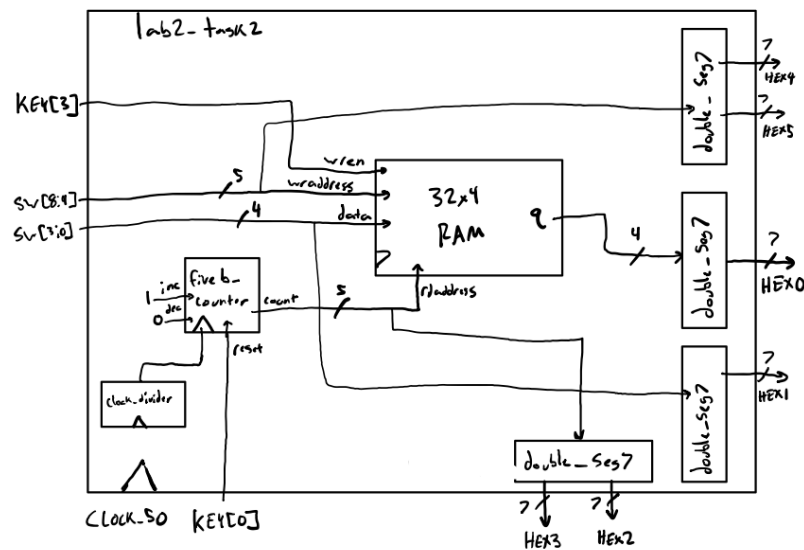


Figure 2. Top-Level Block Diagram for Lab 2 Task 2.

## Task 3

I approached this task by first configuring the dual port 16x8 RAM module to use for the buffer. After this, I designed the functionality by which the FIFO modules would keep track of the read and write addresses, which would act as the front and back pointers of the queue. To this end, I implemented both of these as five-bit counters, which would each increment on a read and write operation, respectively. I also decided to include a third five-bit counter, which would keep track of the number of elements in the buffer. This was done to distinguish between whether the buffer was full or empty.

Following this, I created the top level module that would instantiate the FIFO buffer modules and defined connections between peripherals and displays. I also decided to include an extra output signal "error" which would be raised when the user tries to read while empty, or to write while full. This was done to increase the level of feedback given by the system given inputs from the user.

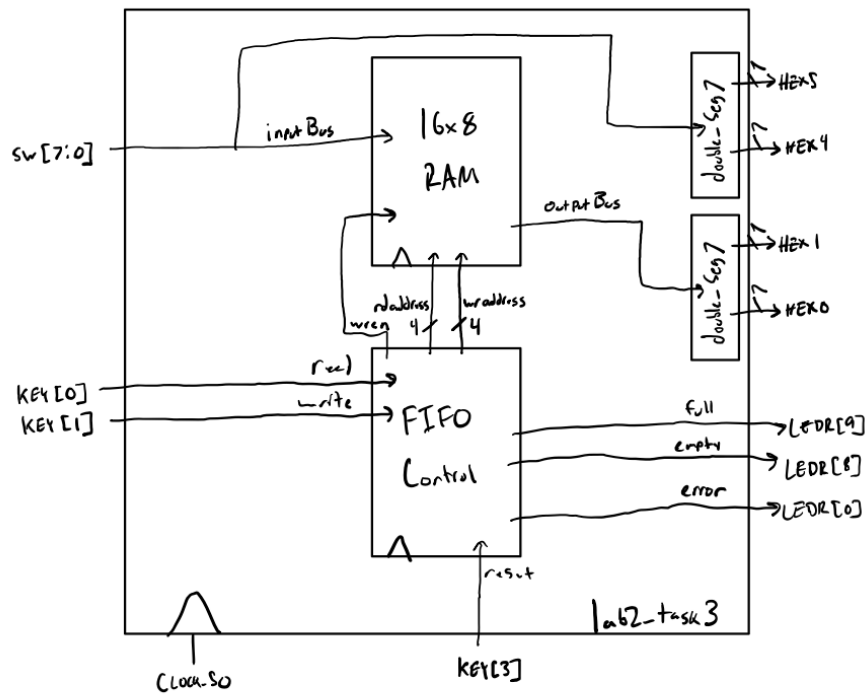


Figure 3. Top-Level Block Diagram for Lab 2 Task 3.

## Results

### Task 1

The following are screenshots from ModelSim simulations for each module used in the simulated RAM module.

For conciseness, HEX displays are encoded in hexadecimal. Use the following table to convert from hexadecimal code to the corresponding seven segment display:

ModelSim Hexadecimal Code	Seven Segment Display Character
40	0
79	1
24	2
30	3
19	4
12	5
02	6
78	7
00	8
10	9
08	A
03	b
27	c
21	d
06	E
0e	F

Table 1. Conversion between hexadecimal values shown in ModelSim and the character shown on a seven segment display.



Figure 4. ModelSim Waveform for the Double Seven Segment Display module testbench.

This waveform demonstrates the two digit, hexadecimal display given any number from 0x00 to 0xff. Refer to Table 1 to decode the hexadecimal values to their corresponding characters.

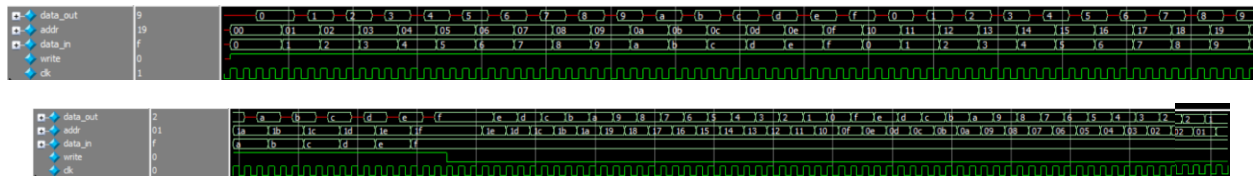


Figure 5. ModelSim Waveform for the 32x4 RAM module testbench.

These screenshots first show the process of writing data to every address in the module. Notice that it takes two clock cycles for the data on “data\_in” to appear on the “data\_out” line. Next, the waveform shows the process of reading from every address in the module. Notice that the value read from each address matches with the one previously written.

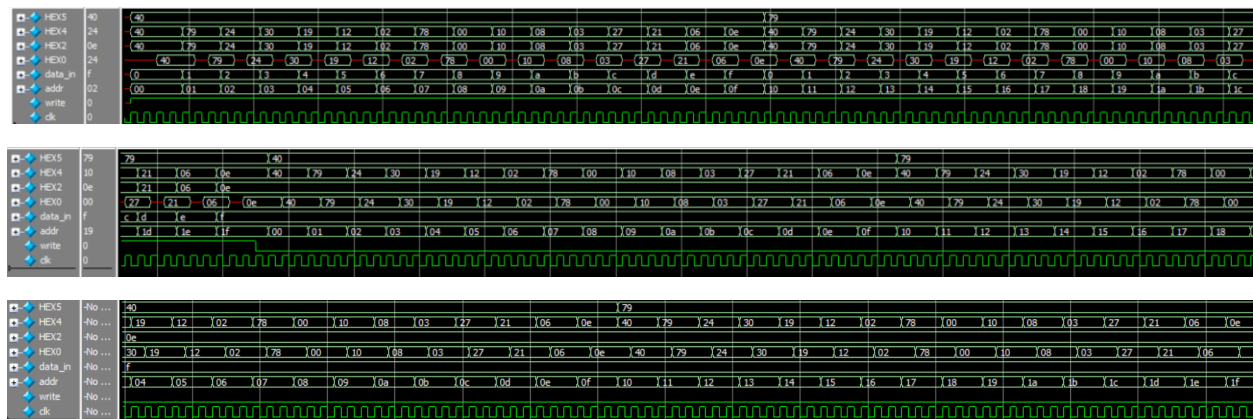


Figure 6. ModelSim Waveform for the Task 1 top-level module testbench.

Similar to Figure 5, these screenshots demonstrate the process of writing to every address in the RAM module, followed by reading from every address in the module. Additionally, the screenshots demonstrate that the correct values for the correct input and output values are shown on the HEX displays. (Refer to Table 1.)

## Task 2

Task 2 uses the same Double Seven Segment Display module from Task 1. Refer to Figure 4 for waveform screenshots.

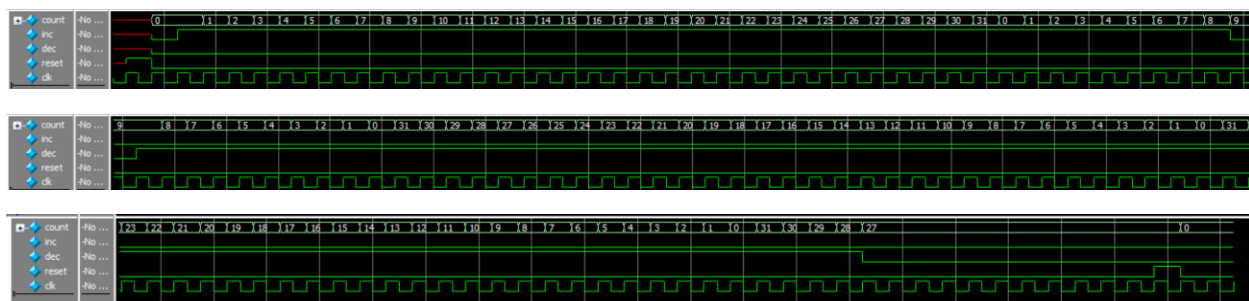


Figure 7. ModelSim Waveform for the five-bit counter module testbench.

These screenshots demonstrate incrementing the counter and overflowing, decrementing the counter and underflowing, holding the counter constant, and resetting the counter to zero.

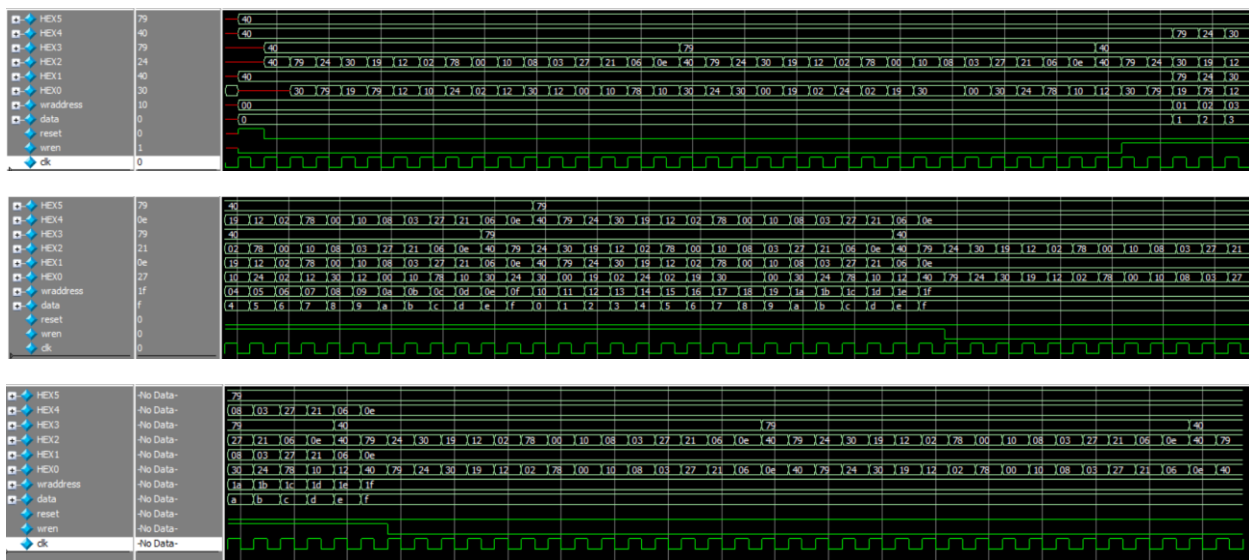


Figure 8. ModelSim Waveform for the Task 2 top-level module testbench.

These screenshots demonstrate the process of reading from every address in the RAM to ensure the correct initial values were loaded, then the process of writing a new value to every address, then reading every value to ensure the correct values were written. Note: some signals are given aliases for better clarity.

### Task 3

Task 3 uses the same Double Seven Segment Display module and five-bit counter module from Task 2. Refer to Figure 4 and Figure 7 for waveform screenshots.

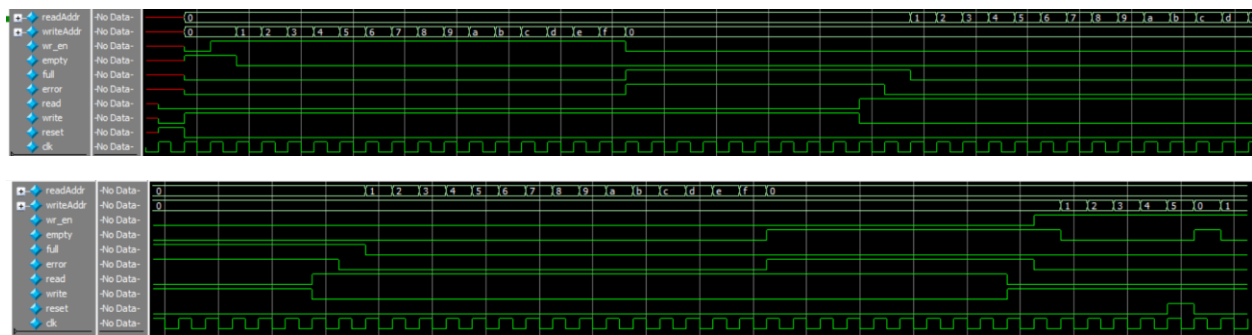


Figure 9. ModelSim Waveform for the FIFO Control module testbench.

These screenshots demonstrate the process of writing into the buffer until full, then attempting to write while the buffer is full. Then the process of reading values from the buffer until empty is shown, followed by attempting to read while the buffer is empty. These screenshots additionally demonstrate that the correct output signals are raised in the correct conditions (i.e. “full” when the buffer is full, “empty” when the buffer is empty, etc.).

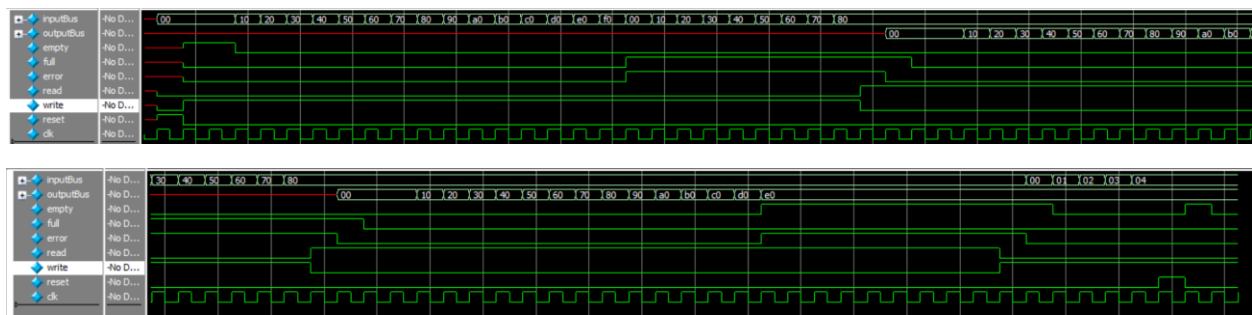


Figure 10. ModelSim Waveform for the FIFO module testbench.

These screenshots demonstrate the process of writing specific values into the buffer until full, then attempting to write while the buffer is full. Then the process of reading those same values from the buffer until empty is shown, followed by attempting to read while the buffer is empty. These screenshots additionally demonstrate that the correct output signals are raised in the correct conditions (i.e. “full” when the buffer is full, “empty” when the buffer is empty, etc.).

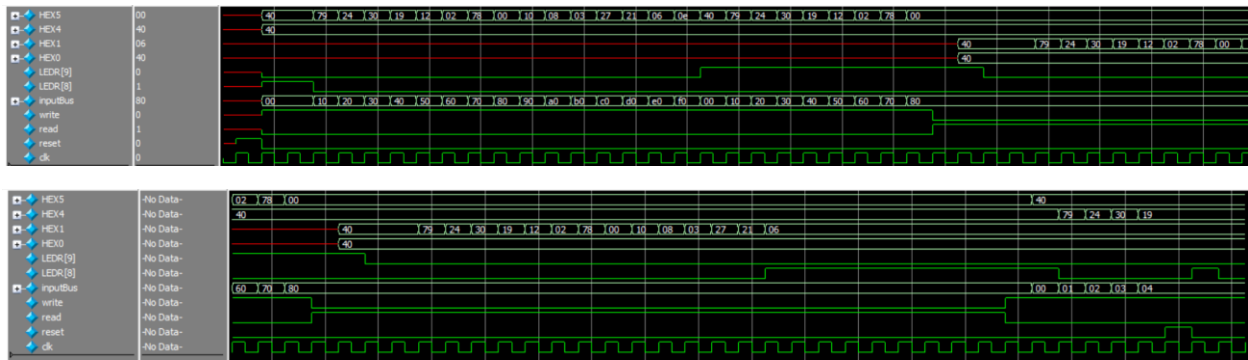


Figure 11. ModelSim Waveform for the Task 3 top-level module testbench.

These screenshots demonstrate the process of writing values into the buffer until full, then attempting to write while the buffer is full. Then the process of reading values from the buffer until empty is shown, followed by attempting to read while the buffer is empty. These screenshots additionally demonstrate that the correct output signals are raised in the correct conditions (i.e. “full” when the buffer is full, “empty” when the buffer is empty, etc.), as well as that the correct displays are shown in the correct conditions (i.e. HEX displays, LEDs, etc.). Note: some input signals are given aliases in these screenshots for better clarity of signal function.

## Final Product

The overarching goal of this lab was to gain practice with interacting with RAM modules in various ways. Task 1 necessitated building the RAM module mostly from scratch while still utilizing on chip memory. This task was completed fully in line with the specification. In Task 2, I used a RAM module configured from the IP catalog, then built a lightweight system around it to cycle through its contents and be able to write to a specified address. This was also completed in line with the specification. And in Task 3, I again used a configured RAM module from the IP catalog, but this time to build a more involved FIFO buffer. The results of this task were more open ended, so there were more individual decisions made about timing constraints, edge case behavior, and overall data flow.



## Appendix: SystemVerilog Code

(See next page)

```

1  /*****
2
3  double_seg7.sv
4
5  *****/
6  /*
7   * Connor Aksama
8   * 01/18/2023
9   * CSE 371
10  * Lab 2
11  */
12
13 /**
14  * Defines data for 2-digit hexadecimal HEX display given a 8-bit unsigned integer
15
16  * Inputs:
17  *     num [8 bit] - An unsigned integer value [0x0-0x7F] to display
18
19  * Outputs:
20  *     HEX0 [7 bit] - A HEX display for the least significant digit of the input num,
21  *     formatted in standard seven segment display format
22  *     HEX1 [7 bit] - A HEX display for the most significant digit of the input num,
23  *     formatted in standard seven segment display format
24  */
25 module double_seg7(
26     output logic [6:0] HEX0, HEX1
27     ,input logic [7:0] num
28 );
29
30     // Drive HEX output signals given num
31     always_comb begin
32         // Light HEX0 using LSD of num
33         case (num[3:0])
34             //     Light: 6543210
35             0: HEX0 = ~7'b0111111; // 0
36             1: HEX0 = ~7'b0000110; // 1
37             2: HEX0 = ~7'b1011011; // 2
38             3: HEX0 = ~7'b1001111; // 3
39             4: HEX0 = ~7'b1100110; // 4
40             5: HEX0 = ~7'b1101101; // 5
41             6: HEX0 = ~7'b1111101; // 6
42             7: HEX0 = ~7'b0000111; // 7
43             8: HEX0 = ~7'b1111111; // 8
44             9: HEX0 = ~7'b1101111; // 9
45             10: HEX0 = ~7'b1110111; // A
46             11: HEX0 = ~7'b1111100; // b
47             12: HEX0 = ~7'b1011000; // c
48             13: HEX0 = ~7'b1011110; // d
49             14: HEX0 = ~7'b1111001; // E
50             15: HEX0 = ~7'b1110001; // F
51             default: HEX0 = 7'bX;
52         endcase
53
54         // Light HEX1 using MSD of num
55         case (num >> 4)
56             //     Light: 6543210
57             0: HEX1 = ~7'b0111111; // 0
58             1: HEX1 = ~7'b0000110; // 1
59             2: HEX1 = ~7'b1011011; // 2
60             3: HEX1 = ~7'b1001111; // 3
61             4: HEX1 = ~7'b1100110; // 4
62             5: HEX1 = ~7'b1101101; // 5
63             6: HEX1 = ~7'b1111101; // 6
64             7: HEX1 = ~7'b0000111; // 7
65             8: HEX1 = ~7'b1111111; // 8
66             9: HEX1 = ~7'b1101111; // 9
67             10: HEX1 = ~7'b1110111; // A
68             11: HEX1 = ~7'b1111100; // b
69             12: HEX1 = ~7'b1011000; // c

```

```

68         13: HEX1 = ~7'b1011110; // d
69         14: HEX1 = ~7'b1111001; // E
70         15: HEX1 = ~7'b1110001; // F
71         default: HEX1 = 7'bx;
72     endcase
73
74     end
75
76 endmodule // double_seg7
77
78 /*
79  * Tests the functionality of the double_seg7 module.
80  */
81 module double_seg7_testbench();
82
83     logic [6:0] HEX0, HEX1;
84     logic [7:0] num;
85
86     double_seg7 dut (.HEX0, .HEX1, .num);
87
88     initial begin
89
90         integer i;
91
92         // Check HEX displays for integers 0x0-0xff
93         for (i = 0; i <= 255; i++) begin
94             #10 num = i;
95         end
96         #50;
97     end
98
99 endmodule // double_seg7_testbench
100
101 /*****
102
103 ram.sv
104
105 *****/
106
107 /*
108  * Connor Aksama
109  * 01/18/2023
110  * CSE 371
111  * Lab 2
112  */
113
114 /**
115  * 32x4 RAM Module
116
117  * Inputs:
118  *  addr    [5 bit] - Address to manipulate in the given module.
119  *  data_in [4 bit] - Data to store at the specified address in the module.
120  *  write   [1 bit] - Signal to enable writing the given data_in to the specified addr.
121  *  If raised, the value at addr is updated to data_in; o.w. the value is unchanged.
122  *  clk     [1 bit] - The clock to use for this module.
123
124  * Outputs:
125  *  data_out [4 bit] - The data stored at the location specified by addr.
126  */
127 module ram(
128     output logic [3:0] data_out
129     ,input logic [4:0] addr
130     ,input logic [3:0] data_in
131     ,input logic write, clk
132 );
133
134     // Flip-flop outputs
135     logic [4:0] addr_reg;
136     logic [3:0] data_in_reg;

```

```

136     logic write_reg;
137
138     // Memory cells
139     logic [3:0] memory_array [31:0];
140
141     // Use the registered address, output data at location
142     assign data_out = memory_array[addr_reg];
143
144     always_ff @(posedge clk) begin
145
146         // Synchronize inputs
147         addr_reg <= addr;
148         data_in_reg <= data_in;
149         write_reg <= write;
150
151         // Write to addr if write enabled
152         if (write_reg) memory_array[addr_reg] <= data_in_reg;
153
154     end
155
156 endmodule // ram
157
158 // Module to test the functionality of the ram module
159 module ram_testbench();
160
161     logic [3:0] data_out;
162     logic [4:0] addr;
163     logic [3:0] data_in;
164     logic write, clk;
165
166     ram dut (.data_out, .addr, .data_in, .write, .clk);
167
168     parameter CLOCK_PERIOD = 100;
169     initial begin
170         clk <= 0;
171         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
172     end
173
174     initial begin
175         integer i;
176
177         // Write to every address
178         @(posedge clk) write <= 1; addr <= 0; data_in <= 0;
179         for (i = 0; i < 32; i++) begin
180             @(posedge clk) addr <= i; data_in <= i;
181             @(posedge clk);
182             @(posedge clk);
183         end
184
185         // Read from every address
186         @(posedge clk) write <= 0;
187         for (i = 31; i >= 0; i--) begin
188             @(posedge clk) addr <= i;
189             @(posedge clk);
190         end
191         $stop;
192     end
193
194 endmodule // ram_testbench
195
196 /*****
197
198 lab2_task1.sv
199
200 *****/
201
202 /*
203  * Connor Aksama
204  * 01/18/2023

```

```

205     * CSE 371
206     * Lab 2
207 */
208
209 /**
210  * Top-level module for Lab 2 Task 1. Instantiates a 32x4 RAM module and connects I/O to
    board peripherals
211
212  * Inputs:
213  *   SW [10 bit] - The 10 onboard switches, respectively.
214  *   KEY [4 bit] - The 4 onboard keys, respectively.
215
216  * Outputs:
217  *   HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
    display format.
218  *   HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
    display format.
219  *   HEX2 [7 bit] - Data to show on the HEX2 display, formatted in standard 7 segment
    display format.
220  *   HEX3 [7 bit] - Data to show on the HEX3 display, formatted in standard 7 segment
    display format.
221  *   HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
    display format.
222  *   HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
    display format.
223 */
224 module lab2_task1(
225     output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
226     ,input logic [9:0] SW
227     ,input logic [3:0] KEY
228 );
229
230     // Turn off HEX3 and HEX1
231     assign HEX3 = ~7'b0;
232     assign HEX1 = ~7'b0;
233
234     // Use KEY0 as system clock
235     logic SYS_CLOCK;
236     assign SYS_CLOCK = KEY[0];
237
238     logic [3:0] data_in, data_out;
239     logic [4:0] addr;
240     logic write;
241
242     assign data_in = SW[3:0]; // SW3-0 specifies input data
243     assign addr = SW[8:4];    // SW8-4 specifies address
244     assign write = SW[9];    // SW9 is write signal
245
246     // Connect RAM input ports to SW inputs
247     // Store RAM output in local logic
248     ram r (.data_out, .addr, .data_in, .write(write), .clk(SYS_CLOCK));
249
250     // Display address on HEX4-5
251     double_seg7 addr_display (.HEX0(HEX4), .HEX1(HEX5), .num({3'b0, addr}));
252
253     // Display data_in on HEX2
254     double_seg7 data_in_display (.HEX0(HEX2), .HEX1(), .num({4'b0, data_in}));
255
256     // Display data_out on HEX0
257     double_seg7 data_out_display(.HEX0(HEX0), .HEX1(), .num({4'b0, data_out}));
258
259 endmodule
260
261 // Module to test the functionality of the lab2_task1 module
262 module lab2_task1_testbench();
263
264     logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
265     logic [9:0] SW;
266     logic [3:0] KEY;

```

```

267
268     lab2_task1 dut (.HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0, .SW, .KEY);
269
270     logic write, clk;
271     logic [3:0] data_in;
272     logic [4:0] addr;
273     assign KEY[0] = clk;
274     assign SW[9] = write;
275     assign SW[3:0] = data_in;
276     assign SW[8:4] = addr;
277
278     parameter CLOCK_PERIOD = 100;
279     initial begin
280         clk <= 0;
281         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
282     end
283
284     initial begin
285         integer i;
286
287         // Write to every address in the RAM
288         @(posedge clk) write <= 1; addr <= 0; data_in <= 0;
289         for (i = 0; i < 32; i++) begin
290             @(posedge clk) addr <= i; data_in <= i;
291             @(posedge clk);
292             @(posedge clk);
293         end
294
295         // Read from every address in the RAM
296         @(posedge clk) write <= 0;
297         for (i = 0; i < 32; i++) begin
298             @(posedge clk) addr <= i;
299             @(posedge clk);
300             @(posedge clk);
301         end
302
303         $stop;
304     end
305
306 endmodule
307
308 /*****
309
310 fiveb_counter.sv
311
312 *****/
313
314 /*
315  * Connor Aksama
316  * 01/19/2023
317  * CSE 371
318  * Lab 2
319  */
320
321 /**
322  * A five-bit increment/decrement counter. Overflows to 5'b00000; underflows to 5'b11111.
323  * reset takes precedence; if inc and dec are simultaneously raised, count is unchanged.
324  * Inputs:
325  *   inc [1 bit] - Increments the count by 1 when raised. Does nothing if signal is low.
326  *   dec [1 bit] - Decrements the count by 1 when raised. Does nothing if signal is low.
327  *   clk [1 bit] - The clock to use for this module.
328  *   reset [1 bit] - Resets the counter to 0 when raised. Does nothing if signal is low.
329
330  * Outputs:
331  *   count [5 bit] - The count of this counter.
332  */
333 module fiveb_counter(
334     output logic [4:0] count
335     ,input logic inc, dec, clk, reset

```

```

336 );
337
338 // Handles increment/decrement/reset operations
339 always_ff @(posedge clk) begin
340
341     if (reset)
342         count <= 5'b0;
343     else if (inc & ~dec)
344         count <= count + 5'b1;
345     else if (dec & ~inc)
346         count <= count - 5'b1;
347
348 end
349
350 endmodule // fiveb_counter
351
352 /*
353  * Tests the functionality of the fiveb_counter module.
354  */
355 module fiveb_counter_testbench();
356
357     logic [4:0] count;
358     logic inc, dec, clk, reset;
359
360     fiveb_counter dut (.count, .inc, .dec, .clk, .reset);
361
362     parameter CLOCK_PERIOD = 100;
363     initial begin
364         clk <= 0;
365         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
366     end
367
368     initial begin
369         integer i;
370
371         @(posedge clk) reset <= 1;
372         @(posedge clk) reset <= 0; inc <= 0; dec <= 0;
373
374         // Increment counter from 0 to max, then past max
375         @(posedge clk) inc <= 1;
376         for (i = 0; i < 40; i++) begin
377             @(posedge clk);
378         end
379
380         // Hold counter constant
381         @(posedge clk) inc <= 0;
382         for (i = 0; i < 10; i++) begin
383             @(posedge clk);
384         end
385
386         // Decrement counter below 0, then max to 0, etc.
387         @(posedge clk) dec <= 1;
388         for (i = 0; i < 45; i++) begin
389             @(posedge clk);
390         end
391
392         // Hold counter constant
393         @(posedge clk) dec <= 0;
394         for (i = 0; i < 10; i++) begin
395             @(posedge clk);
396         end
397
398         // reset to 0
399         @(posedge clk) reset <= 1;
400         @(posedge clk) reset <= 0;
401
402         @(posedge clk)
403         @(posedge clk)
404

```

```

405         $stop;
406     end
407
408 endmodule // fiveb_counter_testbench
409
410 /*****
411
412 lab2_task2.sv
413
414 *****/
415
416 /*
417  * Connor Aksama
418  * 01/19/2023
419  * CSE 371
420  * Lab 2
421  */
422
423 /**
424  * Top-level module for Lab 2 Task 2. Instantiates a 32x4 RAM module and connects I/O to
425  board peripherals
426
427  * Inputs:
428  * SW [10 bit] - The 10 onboard switches, respectively.
429  * KEY [4 bit] - The 4 onboard keys, respectively.
430  * CLOCK_50 [1 bit] - The system clock to use for this module.
431
432  * Outputs:
433  * HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
434  display format.
435  * HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
436  display format.
437  * HEX2 [7 bit] - Data to show on the HEX2 display, formatted in standard 7 segment
438  display format.
439  * HEX3 [7 bit] - Data to show on the HEX3 display, formatted in standard 7 segment
440  display format.
441  * HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
442  display format.
443  * HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
444  display format.
445  */
446 module lab2_task2 #(
447     parameter which_clock = 25
448 )
449 (
450     output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
451     ,input logic [9:0] SW
452     ,input logic [3:0] KEY
453     ,input logic CLOCK_50
454 );
455
456 // Clock divider to cycle through reading RAM addresses
457 // Divides the 50 MHz clock into an array of divided clocks (clk)
458 logic [31:0] clk;
459 clock_divider clk_div (.clock(CLOCK_50), .divided_clocks(clk));
460
461 // Aliases for peripherals and RAM output
462 logic [3:0] data; // Data_In
463 logic [4:0] rdaddress; // Read Address
464 logic [4:0] wraddress; // Write Address
465 logic wren; // Write Enable
466 logic [3:0] q; // RAM Data_Out
467 logic reset;
468
469 // Assign peripherals to aliases
470 assign reset = ~KEY[0];
471 assign wren = ~KEY[3];
472 assign wraddress = SW[8:4];
473 assign data = SW[3:0];

```



```

467
468 // 32x4 RAM Module
469 // Inputs: 50 MHz sys. clock, Data_In, Read Address, Write Address, Write Enable
470 // Outputs: q -> RAM Data_Out
471 ram32x4 ram (.clock(CLOCK_50), .data, .rdaddress, .wraddress, .wren, .q);
472
473 // HEX Display for RAM Data_Out
474 double_seg7 q_display (.HEX0(HEX0), .HEX1(), .num(q));
475
476 // HEX Display for Read Address
477 double_seg7 rdaddress_display (.HEX0(HEX2), .HEX1(HEX3), .num(rdaddress));
478
479 // HEX Display for Write Address
480 double_seg7 wraddress_display (.HEX0(HEX4), .HEX1(HEX5), .num(wraddress));
481
482 // HEX Display for Data_In
483 double_seg7 data_display (.HEX0(HEX1), .HEX1(), .num(data));
484
485 // Counter to cycle through RAM addresses, runs on divided clock
486 // Output count to Read Address
487 fiveb_counter counter (.count(rdaddress), .inc(1), .dec(0), .clk(clk[which_clock]), .
reset);
488
489 endmodule // lab2_task2
490
491 // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] =
0.75Hz, ...
492 module clock_divider (
493     input logic clock
494     ,output logic [31:0] divided_clocks
495 );
496
497 initial begin
498     divided_clocks = '0;
499 end
500
501 always_ff @(posedge clock) begin
502     divided_clocks <= divided_clocks + 'd1;
503 end
504
505 endmodule // clock_divider
506
507 /**
508  * Identical to lab2_task2 module, without clock division. Used for testing purposes.
509
510  * Inputs:
511  * SW [10 bit] - The 10 onboard switches, respectively.
512  * KEY [4 bit] - The 4 onboard keys, respectively.
513  * CLOCK_50 [1 bit] - The system clock to use for this module.
514
515  * Outputs:
516  * HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
display format.
517  * HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
display format.
518  * HEX2 [7 bit] - Data to show on the HEX2 display, formatted in standard 7 segment
display format.
519  * HEX3 [7 bit] - Data to show on the HEX3 display, formatted in standard 7 segment
display format.
520  * HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
display format.
521  * HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
display format.
522 */
523 `timescale 1 ps / 1 ps
524 module lab2_task2_testmodule
525 (
526     output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
527     ,input logic [9:0] SW

```

```

528 ,input logic [3:0] KEY
529 ,input logic CLOCK_50
530 );
531
532 // Aliases for peripherals and RAM output
533 logic [3:0] data; // Data_In
534 logic [4:0] rdaddress; // Read Address
535 logic [4:0] wraddress; // Write Address
536 logic wren; // Write Enable
537 logic [3:0] q; // RAM Data_Out
538 logic reset;
539
540 // Assign peripherals to aliases
541 assign reset = ~KEY[0];
542 assign wren = ~KEY[3];
543 assign wraddress = SW[8:4];
544 assign data = SW[3:0];
545
546 // 32x4 RAM Module
547 // Inputs: 50 MHz sys. clock, Data_In, Read Address, Write Address, Write Enable
548 // Outputs: q -> RAM Data_Out
549 ram32x4 ram (.clock(CLOCK_50), .data, .rdaddress, .wraddress, .wren, .q);
550
551 // HEX Display for RAM Data_Out
552 double_seg7 q_display (.HEX0(HEX0), .HEX1(), .num(q));
553
554 // HEX Display for Read Address
555 double_seg7 rdaddress_display (.HEX0(HEX2), .HEX1(HEX3), .num(rdaddress));
556
557 // HEX Display for Write Address
558 double_seg7 wraddress_display (.HEX0(HEX4), .HEX1(HEX5), .num(wraddress));
559
560 // HEX Display for Data_In
561 double_seg7 data_display (.HEX0(HEX1), .HEX1(), .num(data));
562
563 // Counter to cycle through RAM addresses, runs on sys clock
564 // Output count to Read Address
565 fiveb_counter counter (.count(rdaddress), .inc(1), .dec(0), .clk(CLOCK_50), .reset);
566
567 endmodule // lab2_task2_testmodule
568
569 // Module to test the functionality of the lab2_task2 module
570 module lab2_task2_testbench();
571
572 logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
573 logic [9:0] SW;
574 logic [3:0] KEY;
575 logic clk;
576
577 logic [4:0] wraddress;
578 logic [3:0] data;
579 logic reset, wren;
580
581 assign SW[8:4] = wraddress;
582 assign SW[3:0] = data;
583 assign KEY[0] = ~reset;
584 assign KEY[3] = ~wren;
585
586 lab2_task2_testmodule dut (.HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0, .SW, .KEY, .
CLOCK_50(clk));
587
588 parameter CLOCK_PERIOD = 100;
589 initial begin
590     clk <= 0;
591     forever #(CLOCK_PERIOD / 2) clk <= ~clk;
592 end
593
594 initial begin
595     integer i;

```

```

596
597 // Read every address
598 @(posedge clk) reset <= 1; wraddress <= 0; data <= 0; wren <= 0;
599 @(posedge clk) reset <= 0;
600 for (i = 0; i < 32; i++) begin
601     @(posedge clk);
602 end
603
604 // Write to every address
605 @(posedge clk) wren <= 1; wraddress <= 0; data <= 0;
606 for (i = 0; i < 32; i++) begin
607     @(posedge clk) wraddress <= i; data <= i;
608 end
609
610 // Read every address
611 @(posedge clk) wren <= 0;
612 for (i = 0; i < 32; i++) begin
613     @(posedge clk);
614 end
615 $stop;
616 end
617
618 endmodule // lab2_task2_testbench
619
620 /*****
621
622 FIFO.sv
623
624 *****/
625
626 /*
627  * Connor Aksama
628  * 01/19/2023
629  * CSE 371
630  * Lab 2
631  */
632
633 /**
634  * FIFO Module to handle logic between the FIFO Controller and the physical RAM.
635
636  * Inputs:
637  *   inputBus [8 (width) bit] - Data to enqueue in the buffer.
638  *   read      [1 bit] - Signal to enable reading an element from the buffer. If raised
639  *   and buffer is not empty, the least recent word is output on the outputBus, o.w. the
640  *   outputBus is unchanged.
641  *   write     [1 bit] - Signal to enable writing an element to the buffer. If raised and
642  *   buffer is not full, the word on the inputBus is enqueued, o.w. the buffer is unchanged.
643  *   reset     [1 bit] - Resets the FIFO buffer - removes all elements from the buffer.
644  *   clk       [1 bit] - The clock to use for this module.
645
646  * Outputs:
647  *   outputBus [8 (width) bit] - The data most recently dequeued from the buffer.
648  *   empty     [1 bit] - This signal is raised when there are no elements in the buffer,
649  *   and lowered otherwise.
650  *   full      [1 bit] - This signal is raised when the buffer is at capacity, and
651  *   lowered otherwise.
652  *   error     [1 bit] - This signal is raised when reading while the buffer is empty, or
653  *   when writing while the buffer is full
654  */
655 module FIFO #(
656     parameter depth = 4,
657     parameter width = 8
658 ) (
659     output logic empty, full, error
660     ,output logic [width-1:0] outputBus
661     ,input logic clk, reset
662     ,input logic read, write
663     ,input logic [width-1:0] inputBus
664 );

```

```

659
660 logic [3:0] rdaddress; // Read Address from Control module
661 logic [3:0] wraddress; // Write Address from Control module
662 logic wren; // Write enable from Control Module
663 logic [width-1:0] outputBus_temp; // Temporary read output from RAM
664
665 // 16x8 RAM module instantiation
666 // Inputs: sys clock, Data_In from user, Read Address, Write Address, Write Enable
667 // Outputs: Data at Read Address -> outputBus_temp
668 ram16x8 ram (.clock(clk), .data(inputBus), .rdaddress, .wraddress, .wren, .q(
outputBus_temp));

669
670
671 // FIFO Control Module
672 // Inputs: sys clock, read signal from user, write from user
673 // Outputs: Write Enable, Empty/Full/Error, Read Address, Write Address
674 FIFO_Control #(depth) FC (.clk, .reset,
675                             .read,
676                             .write,
677                             .wr_en(wren),
678                             .empty,
679                             .full,
680                             .error,
681                             .readAddr(rdaddress),
682                             .writeAddr(wraddress)
683                             );
684
685 always_ff @(posedge clk) begin
686     // Change outputBus only if valid read is raised
687     if (read & ~empty) outputBus <= outputBus_temp;
688     else outputBus <= outputBus;
689 end
690
691 endmodule // FIFO
692
693 // Module to test the functionality of the FIFO module
694 `timescale 1 ps / 1 ps
695 module FIFO_testbench();
696
697     parameter depth = 4, width = 8;
698
699     logic clk, reset;
700     logic read, write;
701     logic [width-1:0] inputBus;
702     logic empty, full, error;
703     logic [width-1:0] outputBus;
704
705     FIFO #(depth, width) dut (.*);
706
707     parameter CLK_Period = 100;
708     initial begin
709         clk <= 1'b0;
710         forever #(CLK_Period / 2) clk <= ~clk;
711     end
712
713     initial begin
714         integer i;
715
716         @(posedge clk) reset <= 1; write <= 0; read <= 0; inputBus <= 0;
717
718         // Write to queue until full, try to write while full
719         @(posedge clk) write <= 1; reset <= 0;
720         for (i = 0; i < 25; i++) begin
721             @(posedge clk) inputBus <= (i << 4);
722         end
723
724         // Read from queue until empty, try to read while empty
725         @(posedge clk) write <= 0; read <= 1;
726         for (i = 24; i >= 0; i--) begin

```

```

727         @(posedge clk);
728     end
729
730     // Write some elements
731     @(posedge clk) write <= 1; read <= 0;
732     for (i = 0; i < 5; i++) begin
733         @(posedge clk) inputBus <= i;
734     end
735
736     // Reset
737     @(posedge clk) reset <= 1;
738     @(posedge clk) reset <= 0;
739     @(posedge clk);
740     @(posedge clk);
741
742     $stop;
743 end
744
745 endmodule // FIFO_testbench
746
747 /*****
748
749 FIFO_Control.sv
750
751 *****/
752
753 /*
754  * Connor Aksama
755  * 01/19/2023
756  * CSE 371
757  * Lab 2
758  */
759
760 /**
761  * FIFO Control Module to handle logic of what and when to read/write from the RAM given
762  * user inputs.
763
764  * Inputs:
765  *   read  [1 bit] - Signal to enable reading an element from the buffer. If raised and
766  *                   buffer is not empty, the next address to read is output on readAddr
767  *   write [1 bit] - Signal to enable writing an element to the buffer. If raised and
768  *                   buffer is not full, the next address to write is output on writeAddr
769  *   reset [1 bit] - Resets the FIFO buffer - removes all elements from the buffer.
770  *   clk   [1 bit] - The clock to use for this module.
771
772  * Outputs:
773  *   readAddr [4 (depth) bit] - The next address to read from in the RAM.
774  *   writeAddr [4 (depth) bit] - The next address to write to in the RAM.
775  *   empty     [1 bit] - This signal is raised when there are no elements in the buffer,
776  *                       and lowered otherwise.
777  *   full      [1 bit] - This signal is raised when the buffer is at capacity, and
778  *                       lowered otherwise.
779  *   error     [1 bit] - This signal is raised when reading while the buffer is empty, or
780  *                       when writing while the buffer is full
781  *   wr_en     [1 bit] - This signal is raised when data should be written to the
782  *                       location specified by writeAddr, and lowered otherwise.
783  */
784 module FIFO_Control #(
785     parameter depth = 4
786 ) (
787     output logic wr_en
788     ,output logic empty, full, error
789     ,output logic [depth-1:0] readAddr, writeAddr
790     ,input  logic clk, reset
791     ,input  logic read, write
792
793 );
794
795 logic [4:0] fifo_size; // Number of elements in the buffer

```

```

789     logic read_reg, write_reg; // Registered read/write inputs
790     logic dead; // Temporary variable for five bit counter output
791
792     // Counter to store current Read Address in RAM
793     // Inputs: Increment signal -> read detected and not empty
794     // Outputs: Count -> Read Address
795     five_counter read_ptr (.count({dead, readAddr}), .inc(read_reg & ~empty), .dec(0),
        .clk, .reset);
796
797     // Counter to store current Write Address in RAM
798     // Inputs: Increment signal -> write detected and not full
799     // Outputs: Count -> Write Address
800     five_counter write_ptr (.count({dead, writeAddr}), .inc(write_reg & ~full), .dec(0),
        .clk, .reset);
801
802     // Counter to store number of elements in buffer
803     // Inputs: Increment signal -> write detected and not full; Decrement signal -> read
        detected and not empty
804     // Outputs: Count -> FIFO Size
805     five_counter num_elts (.count(fifo_size), .inc(write_reg & ~full), .dec(read_reg & ~
        empty), .clk, .reset);
806
807     // Compare against FIFO size
808     assign empty = fifo_size == '0;
809     assign full = fifo_size == (depth ** 2);
810     // Check for invalid read/writes
811     assign error = (read_reg & empty) | (write_reg & full);
812
813     // Enable write if write detected and not full
814     assign wr_en = write_reg & ~full;
815
816     always_ff @(posedge clk) begin
817         // Register read/write inputs
818         read_reg <= read;
819         write_reg <= write;
820     end
821
822 endmodule // FIFO_Control
823
824 // Module to test the functionality of the FIFO_Control module
825 `timescale 1 ps / 1 ps
826 module FIFO_Control_testbench();
827
828     parameter depth = 4;
829
830     logic wr_en;
831     logic empty, full, error;
832     logic [depth-1:0] readAddr, writeAddr;
833     logic clk, reset;
834     logic read, write;
835
836     FIFO_Control #(depth) dut (.*);
837
838     parameter CLOCK_PERIOD = 100;
839     initial begin
840         clk <= 0;
841         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
842     end
843
844     initial begin
845         integer i;
846
847         @(posedge clk) reset <= 1; write <= 0; read <= 0;
848
849         // Write until full, try writing while full
850         @(posedge clk) write <= 1; reset <= 0;
851         for (i = 0; i < 25; i++) begin
852             @(posedge clk);
853             end

```

```

854
855     // Read until empty, try reading while empty
856     @(posedge clk) read <= 1; write <= 0;
857     for (i = 0; i < 25; i++) begin
858         @(posedge clk);
859     end
860
861     // Write some elements
862     @(posedge clk) write <= 1; read <= 0;
863     for (i = 0; i < 5; i++) begin
864         @(posedge clk);
865     end
866
867     // Reset
868     @(posedge clk) reset <= 1;
869     @(posedge clk) reset <= 0;
870     @(posedge clk);
871     @(posedge clk);
872
873     $stop;
874 end
875
876 endmodule // FIFO_Control_testbench
877
878 /*****
879
880 lab2_task3.sv
881
882 *****/
883
884 /*
885  * Connor Aksama
886  * 01/19/2023
887  * CSE 371
888  * Lab 2
889  */
890
891 /**
892  * Top-level module for Lab 2 Task 3. Instantiates a 16x8 RAM module, FIFO Controller,
893  * and connects I/O to board peripherals
894
895  * Inputs:
896  *   SW [10 bit] - The 10 onboard switches, respectively.
897  *   KEY [4 bit] - The 4 onboard keys, respectively.
898  *   CLOCK_50 [1 bit] - The system clock to use for this module.
899
900  * Outputs:
901  *   HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
902  *   display format.
903  *   HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
904  *   display format.
905  *   HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
906  *   display format.
907  *   HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
908  *   display format.
909  */
910 module lab2_task3 #(
911     parameter depth = 4
912     ,parameter width = 8
913 )
914 (
915     output logic [6:0] HEX5, HEX4, HEX1, HEX0
916     ,output logic [9:0] LEDR
917     ,input logic [9:0] SW
918     ,input logic [3:0] KEY
919     ,input logic CLOCK_50
920 );
921
922     logic [width-1:0] output_data;

```

```

918
919 // HEX display for FIFO output
920 double_seg7 output_data_display (.HEX0(HEX0), .HEX1(HEX1), .num(output_data));
921
922 // HEX display for input data
923 double_seg7 input_data_display (.HEX0(HEX4), .HEX1(HEX5), .num(SW[7:0]));
924
925 // FIFO Module
926 // Inputs: read/write from KEYs, input data from SW[7:0]
927 // Outputs: output element line from outputBus; empty/full/error signals -> LEDRs
928 FIFO #(depth, width) fifo (
929
930     .empty(LED[8])
931     , .full(LED[9])
932     , .error(LED[0])
933     , .outputBus(output_data)
934     , .clk(CLOCK_50)
935     , .reset(~KEY[3])
936     , .read(~KEY[0])
937     , .write(~KEY[1])
938     , .inputBus(SW[7:0])
939 );
940
941 endmodule // lab3_task3
942
943 // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] =
944 // 0.75Hz, ...
945 module clock_divider (
946     input logic clock
947     , output logic [31:0] divided_clocks
948 );
949
950 initial begin
951     divided_clocks = '0;
952 end
953
954 always_ff @(posedge clock) begin
955     divided_clocks <= divided_clocks + 'd1;
956 end
957
958 endmodule // clock_divider
959
960 // Module to test the functionality of the lab2_task3 module
961 `timescale 1 ps / 1 ps
962 module lab2_task3_testbench();
963     parameter depth = 4;
964     parameter width = 8;
965
966     logic [6:0] HEX5, HEX4, HEX1, HEX0;
967     logic [9:0] LEDR;
968     logic [9:0] SW;
969     logic [3:0] KEY;
970     logic clk;
971
972     logic write, read, reset;
973     logic [width-1:0] inputBus;
974
975     assign KEY[1] = ~write;
976     assign KEY[0] = ~read;
977     assign KEY[3] = ~reset;
978     assign SW[7:0] = inputBus;
979
980     lab2_task3 #(depth, width) dut (.HEX5, .HEX4, .HEX1, .HEX0, .LEDR, .SW, .KEY, .
981     CLOCK_50(clk));
982
983     parameter CLOCK_PERIOD = 100;
984     initial begin
985         clk <= 0;
986         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
987     end
988 end

```



```

985
986     initial begin
987         integer i;
988
989         @(posedge clk) reset <= 1;
990
991         // Write to queue until full, try to write while full
992         @(posedge clk) write <= 1; read <= 0; reset <= 0; inputBus <= 0;
993         for (i = 0; i < 25; i++) begin
994             @(posedge clk) inputBus <= (i << 4);
995         end
996
997         // Read from queue until empty, try to read while empty
998         @(posedge clk) write <= 0; read <= 1;
999         for (i = 24; i >= 0; i--) begin
1000             @(posedge clk);
1001         end
1002
1003         // Write some elements
1004         @(posedge clk) write <= 1; read <= 0;
1005         for (i = 0; i < 5; i++) begin
1006             @(posedge clk) inputBus <= i;
1007         end
1008
1009         // Reset
1010         @(posedge clk) reset <= 1;
1011         @(posedge clk) reset <= 0;
1012         @(posedge clk);
1013         @(posedge clk);
1014
1015         $stop;
1016     end
1017
1018 endmodule // lab2_task3_testbench

```