Middle East Technical University - Department of Computer Engineering

CENG 232

Logic Design

Spring '2018-2019

Lab 4

Part 1 Due Date: 5 May 2019, Sunday, 23:59
Part 2 Due Date: 12 May 2019, Sunday, 23:59
No late submissions

1 Part 1: RGB Memory (40 pts)

In this part, you are expected to implement basic memories as Verilog modules. These modules will be used to apply masking operations to a given pixel of an image and save the resulting value to the memory. Illustration of the modules is provided in Figure 1.

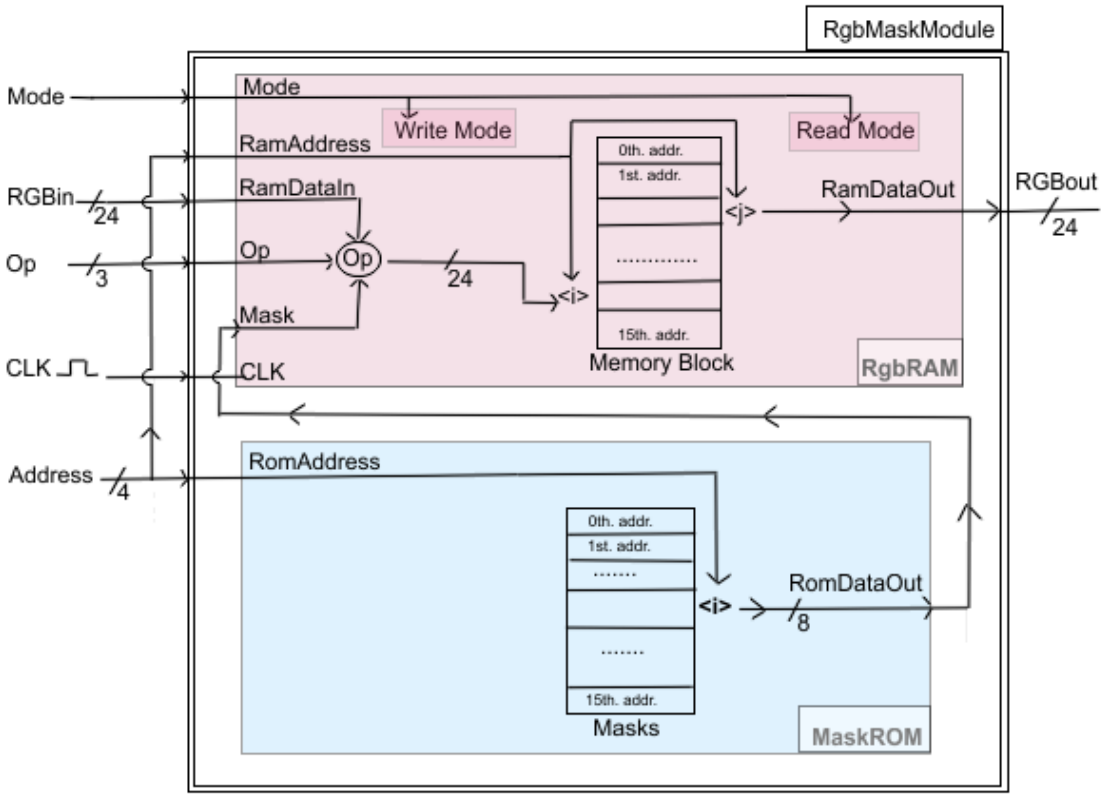

Figure 1: Illustration of the modules.

1.1 RgbMask Module

This is the upper module, in which inputs and outputs of other modules are defined. The inputs of this module; RGBin, Mode, Address and Op are distributed to RgbRAM and MaskROM modules.

- The pixel value, which will be given as input `RGBin`, consists of 3 color channels: Red, Green and Blue. Each of these channels contains 8 bits and has a range from 0 to 255. Illustration of `RGBin` is provided below:

`RGBin` (24 bits)		
Red	Green	Blue
8 bits	8 bits	8 bits

- Mode can be either Write(1) or Read(0).
- 4-bit Address input is used to point memory locations by both of the modules.
- The details of masking operations (Op) are provided in Table 2.

`RgbMaskModule` has already been implemented by us; hence, **you should not implement this module**. `MaskROM` and `RgbRAM` modules will be implemented by yourselves.

1.2 MaskROM Module

This module basically contains 16 registers. Each register has a size of 8 bits, and stores a unique Mask value. The module returns the value of the register pointed by the given RomAddress as output RomDataOut. It works as a combinational circuit. That means it is not triggered by a clock pulse; it is triggered by RomAddress change. The values of ROM should be set as given in Table 1.

Address	Value (8 bits)
0	00000000
1	00001111
2	00011110
3	00110000
4	01010000
5	01100110
6	01101010
7	01111110
8	10000001
9	10100000
10	10100110
11	10111101
12	11000000
13	11010000
14	11010011
15	11100110

Table 1: ROM Structure

Use the following Verilog definition for the module:

```
module MaskROM (
input [3:0] RomAddress,
output reg [7:0] RomDataOut
);
```

1.3 RgbRAM Module

RgbRAM module is used to store/retrieve 24-bit RGB pixel values. In write mode(Mode=1), a specific mask operation(Op) is applied and the result is saved to the RamAddress location of the memory. In read mode(Mode=0) the 24-bit pixel value is retrieved from the RamAddress location of the memory.

Use the following Verilog definition for the module:

```
module RgbRAM (
  input Mode,
  input [3:0] RamAddress,
  input [23:0] RamDataIn,
  input [7:0] Mask,
  input [2:0] Op,
  input CLK,
  output reg [23:0] RamDataOut
  );
```

1.3.1 Read Mode:

In Read Mode, when RamAddress value is given as i, the value stored in the ith index of the RgbRAM will be returned as output RamDataOut. No masking or write operation is conducted on the memory during this mode. This operation will be combinational. That means it is **not** triggered by a clock pulse; but triggered by the following events:

- When Mode is changed from Write to Read, or
- When the value of RamAddress is changed during Read mode.

Note: Initially, the values of all RAM registers will be 0 .

1.3.2 Write Mode:

In Write Mode, you will perform the requested masking operation (Op) on each of R, G and B channels of `RamDataIn` in the same clock cycle. After applying the operation, you will **save** the 24-bit result to RamAddress location of the memory. This procedure will be sequential, and it is triggered by the **positive edge** of the clock pulse.
Op values and their corresponding masking operations are provided in Table 2.

Op	**Operation**
000	Bitwise AND
001	Bitwise OR
010	Bitwise XOR
011	Add
100	Subtract
101	Increment
110	Decrement
111	Rotate Left

Table 2: Mask Operations

Detailed explanations and examples for the operations are provided below. Note that `Mask` is the value that comes from MaskROM. Also note that you should **save** the result of the operation to the RamAddress location of the memory.

- **Bitwise AND**: You should perform AND operation between the Mask and each channel of RamDataIn.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	10100110	10000000	10000010	00000010

- **Bitwise OR**: You should perform OR operation between the Mask and each channel of RamDataIn.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	10100110	10100111	11100111	11100110

- **Bitwise XOR**: You should perform XOR operation between the Mask and each channel of RamDataIn.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	10100110	00100111	01100101	11100100

- **Add**: You should add the Mask value to each channel of RamDataIn. The channel value cannot be greater than 255. Set result to 255 if the output of the addition is greater than 255.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	10100110	11111111	11111111	11101000

- **Subtract**: The Mask value should be subtracted from each channel of RamDataIn. The channel value cannot be less than 0. Set the result to 0 if the output of the subtraction is less than 0.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	10100110	00000000	00011101	00000000

- **Increment**: Increment each channel of RamDataIn by 1. The channel value cannot be greater than 255. As in Add operation, you should set the result to 255 if the output of the addition is greater than 255.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	not applied	10000010	11000100	01000011

- **Decrement**: Decrement each channel of RamDataIn by 1. The channel value cannot be less than 0. As in Subtract operation, the result should be 0 if the output of the subtraction is less than 0.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	not applied	10000000	11000010	01000001

- **Rotate Left**: Shift all the bits of each channel to the left by one. In addition, set the rightmost bit to the previous value of the leftmost bit.

RamDataIn (24 bit)			Mask (8 bit)	Result (24 bit)		
10000001	11000011	01000010	not applied	00000011	10000111	10000100

1.4 Deliverables

- Implement both modules in a single Verilog file: **lab4_1.v**. Do NOT submit your testbenches. You can share your testbenches on the ODTUClass discussion.
- Submit the file through the ODTUClass system before the given deadline. **5 May 2019, Sunday, 23:59**
- This is an individual work, any kind of cheating is not allowed.

2 Part 2: Gas Station (60 pts)

2.1 Problem Definition

In this part of the homework, you are asked to develop an entry system for a gas station. The system is summarized as follows:

1. The following 2 types of fuel can be sold in the station:
   (a) Gasoline (0)
   (b) Diesel (1)
2. There can be at most **6 fuel pumps** in the station (there may be less than 6 pumps). Each fuel pump is dedicated to one type of fuel.
3. The station may sell only one type of fuel or two types of fuel.
   (a) There can be no gasoline pumps in the station. In this case, the number of diesel pumps can be 1 to 6.
   (b) There can be no diesel pumps in the station. In this case, the number of gasoline pumps can be 1 to 6.
   (c) There can be both two types of fuel pumps in the station. In this case, the sum of the numbers of gasoline and diesel pumps should be at most 6.
4. The system records status of 6 pumps with a single variable (`pump_status`). Each bit of `pump_status` corresponds to a fuel pump. 1 means the pump is available for a car to use and 0 means the pump is currently servicing a car or it is not added to the station. The order of the pump bits in the `pump_status` is as follows:
5. The status lights show the status of each pump individually starting from the left-most light with gasoline pumps (if any). After the gasoline pumps are finished, the diesel pumps (if any) follow without any empty space.
   (a) If there are gasoline pumps, they are represented by the left-most bits.
   E.g. If the system has 1 gasoline pump and no diesel pumps then initial `pump_status` is 100000 where the left-most digit represents a gasoline pump.

pump_status index	0	1	2	3	4	5
Fuel Type	G_1					
Led	LD7	LD6	LD5	LD4	LD3	LD2

   (b) If there are gasoline and diesel pumps, then corresponding bits of diesel pumps are located right after of the gasoline bits, E.g. If the system has 2 gasoline pumps and 3 diesel pumps then initial `pump_status` is 111110.

pump_status index	0	1	2	3	4	5
Fuel Type	G_1	G_2	D_1	D_2	D_3	
Led	LD7	LD6	LD5	LD4	LD3	LD2

   (c) If there are only diesel pumps, they are represented by the left-most bits. E.g. The system has no gasoline pumps and 3 diesel pumps then initial `pump_status` is 111000.

pump_status index	0	1	2	3	4	5
Fuel Type	D_1	D_2	D_3			
Led	LD7	LD6	LD5	LD4	LD3	LD2

6. The availability of each queue is recorded (`is_gasoline_queue_not_full`, `is_diesel_queue_not_full`). 1 means the queue is not full and there are available slots for new cars and 0 means the queue is full.

7. A customer can demand at most **8 gallons** of fuel (`fuel_amount`). The system should give a warning (`invalid_gasoline_car`, `invalid_diesel_car`) if a new car demands more than 8 gallons or demands no fuel (0 gallons).
8. A fuel pump can fill 1 gallon of fuel in one unit of time.
9. For each type of fuel (gasoline or diesel), there is a separate car waiting queue. If all of the pumps for the fuel type is busy, then the new car can wait in the queue of that fuel type.
10. There can be at most **2 waiting queues**; 1 for gasoline cars and 1 for diesel cars. If the station sells just one type of fuel, then there should be only one waiting queue.
11. The waiting queues are first in first out (FIFO) queues. E.g. the first car entered the queue will use the empty pump first.
12. Each waiting queue has a capacity of **8 cars**. If a queue for a fuel type is full, then no new car can enter the station for this fuel type until there is a free space in the queue.
13. The number of cars waiting in each queue is recorded (`n_cars_in_gasoline_queue`, `n_cars_in_diesel_queue`). When a new car enters the waiting queue or when a car leaves the queue to use a pump, the number of cars in that queue should be updated.
14. The amount of fuel that is needed to fill the cars in the station is recorded (`total_gasoline_needed`, `total_diesel_needed`). This amount includes both the fuel demands of the cars waiting in the queue and remaining fuels to be filled by the pumps.

2.2 Simulation

There will be three modes in the system which are all synchronous:

1. Setup Mode ($1X$):
   (a) This mode is used to set the number of gasoline pumps and diesel pumps.
   (b) This mode can also be used to reset the system and setup a new station if a simulation is already running.
   (c) In this mode, the waiting queues and all other recorded information should be set to their initial states. E.g. the queues should be empty after this mode.
   (d) If the total number of pumps (diesel + gasoline) is more than 6 or equal to 0, then the system should give a warning (`invalid_setup_params`). In this case:
      - Take no action.
      - Do not clear any data or reset the simulation.
      - The simulation may continue from where it's left when the mode is changed back to simulation (00 or 01).
   (e) You must turn off 7-segment displays if there are no pumps for a fuel type.
   Use the following values, the display implementation is provided in the Board232.v
      - Set `n_cars_in_(gas|dsl)_queue` to `4'b1111`
      - Set `total_(gas|dsl)_needed` to `8'b11111111`
2. Simulation Mode (00):
   (a) The setup must be done before running this mode. If no setup was done before this mode, then the system should take no action.
   (b) If setup was done, this mode is used to simulate the time. 1 unit of time passes in each clock.
   (c) The simulation works in the following order:
      i. If there are available pumps in the station, cars waiting in the queues move to empty pumps starting from the left-most pump of the corresponding fuel type.
         - More than one car may move from each queue to the pumps. If there are cars waiting in the queues, all available pumps must be used by them.
      ii. A new car may try to enter the station. It should be handled according to the specifications explained in the item 3.

iii. All the pumps in the station service the cars for one unit time. Abdullah: ?

(d) Each car using a pump is filled with 1 gallon of fuel. Therefore, the amount of fuel that is needed (`total_gasoline_needed`, `total_diesel_needed`) should be updated.

(e) If a car is filled up, it leaves the station. In this case, the status of the pump should be updated to reflect the change.

(f) If one or more gas pumps become empty, one unit of time is needed for waiting cars to leave the waiting queue and use the empty gas pumps.

(g) A green status light (using `pump_status`) means the pumps is available for a new car. If the light is off, it means the pump is working or it is not added to the station during the setup.

3. Car Entrance Mode (01):

   (a) The simulation continues in this mode.

   (b) A new car with a fuel amount and a fuel type tries to enter the station.

   (c) All actions must take the fuel type into consideration.

   (d) A new car may move to an empty pump if there are no cars in the queue.

   (e) If there are cars waiting in a queue, the new car must be added to the end of the queue.

   (f) If the car expects an out of range ([1..8]) fuel amount or the queue is full, `invalid_xxx_car` warning light is set and the car is discarded.

   (g) All actions in the simulation mode must be done in this mode in the order described in the item 2.

4. In case of an erroneous input (`invalid_gasoline_car`, `invalid_diesel_car`, `invalid_setup_params`) the system should set the warning bits. The warning message should be cleared in the next clock cycle unless another erroneous input is given again.

2.3 Input / Output Specifications

The inputs and outputs for Part 2 are presented in Table 3. The initial values are also given in Table 4.

Table 3: Input / Output Specifications

Name	Type	Size
mode	Input	2 bits
n_gasoline_pumps	Input	3 bits
n_diesel_pumps	Input	3 bits
fuel_amount	Input	4 bits
fuel_type	Input	1 bit
CLK	Input	1 bit
pump_status	Output	6 bits
is_gasoline_queue_not_full	Output	1 bit
is_diesel_queue_not_full	Output	1 bit
n_cars_in_gasoline_queue	Output	4 bits
n_cars_in_diesel_queue	Output	4 bits
total_gasoline_needed	Output	8 bits
total_diesel_needed	Output	8 bits
invalid_gasoline_car	Output	1 bit
invalid_diesel_car	Output	1 bit
invalid_setup_params	Output	1 bit

Table 4: Initial Values

pump_status	6'b000000
is_gasoline_queue_not_full	0
is_diesel_queue_not_full	0
n_cars_in_gasoline_queue	4'b1111
n_cars_in_diesel_queue	4'b1111
total_gasoline_needed	8'b11111111
total_diesel_needed	8'b11111111
invalid_gasoline_car	0
invalid_diesel_car	0
invalid_setup_params	0

2.4 FPGA Implementation

The input and output connections are provided in the supplemented *Board232.v* file. The mapping is provided in Table 5. In addition to the inputs and outputs given in Section 2.3, **BTN1** works as a switch for the display modes between the numbers of cars and the fuel amounts. While the button is held pressed, the 7-segment displays will display the fuel amounts, otherwise they will display the number of cars in each queue. This part is implemented for you. Moreover, **BTN3** works as another clock. You can press and hold it and the simulation will continue with successive clock ticks. The mapping between the items and the board is also provided in Figure 2.

Table 5: Input / Output Connections for FPGA board

Name	**FPGA Board**	**Description**	
mode	SW[1,0]	(A)	2 right-most switches
n_gasoline_pumps	SW[7..5]	(B)	3 left-most switches
n_diesel_pumps	SW[4..2]	(C)	3 switches to the right of A
fuel_amount	SW[7..4]	(D)	4 left-most switches
fuel_type	SW[3]	(E)	The switch to the right of D
CLK	BTN[0]	(F)	The right-most button
pump_status	LD[7..2]	(G)	6 left-most status lights
is_gasoline_queue_not_full	LD[1]	(H)	The status light to right of G
is_diesel_queue_not_full	LD[0]	(I)	The status light to right of H
n_cars_in_gasoline_queue	AN[2]	(J)	The second left-most 7-seg disp
n_cars_in_diesel_queue	AN[0]	(K)	The right-most 7-seg disp
total_gasoline_needed	AN[3,2]	(L)	2 left-most 7-seg disp
total_diesel_needed	AN[1,0]	(M)	2 right-most 7-seg disp
invalid_gasoline_car	AN[2] dot	(N)	The dot in 7-seg disp of J
invalid_diesel_car	AN[0] dot	(O)	The dot in 7-seg disp of K
invalid_setup_params	AN[3..0] dots	(P)	4 dots in all 7-seg disp
Display mode switch	BTN[1]	(Q)	The button to the left to F
Press & hold clock	BTN[3]	(R)	The left-most button

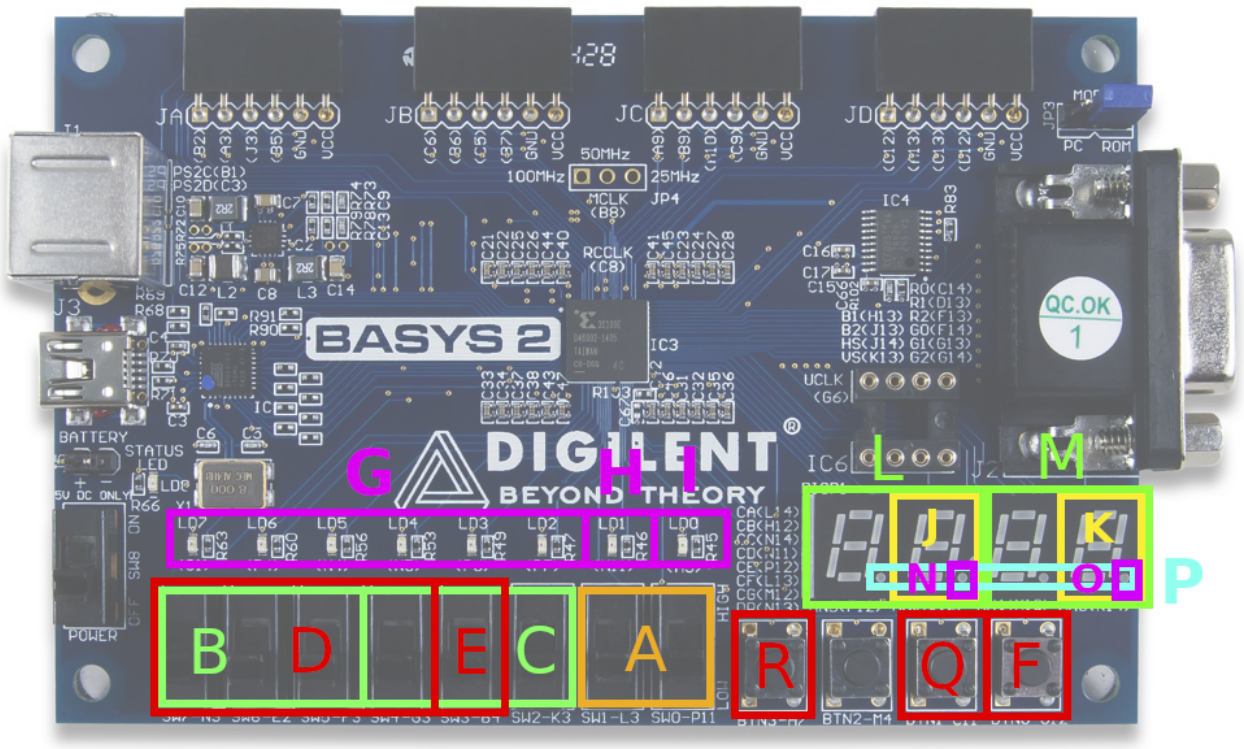

Figure 2: Board figure with labels

Figure 3: 7-segment displays in detail

2.5 Deliverables

- Implement both modules in a single Verilog file: **lab4_2.v**. Do NOT submit your testbenches. You can share your testbenches on the ODTUClass discussion.
- You will get **0** if your file (lab4_2.v) fails to generate the bit file.
- Submit the file through the ODTUClass system before the given deadline. **12 May 2019, Sunday, 23:59**
- This is an individual work, any kind of cheating is not allowed.