Lab Session 3
Design of Multiplexers, ALUs and the Conversion between RGB and Grayscale Using Verilog Coding

Advisor: Lih-Yih Chiou

Speaker: Juliana

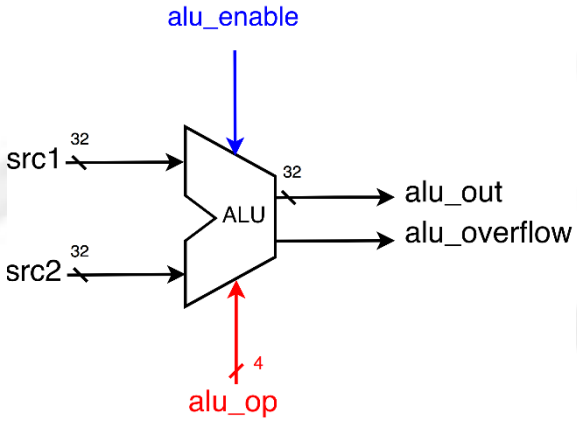
Date: 03/10/2021

Outline

- Introduction
- Combinational logic in Verilog
- Design of a 4-to-1 multiplexer(MUX)
- Testbench
- Design of an Arithmetic Logic Unit (ALU)
- Design of the conversion between RGB and grayscale
- Homework

Introduction

- Combinational circuit
  - Outputs only depends on inputs.
- ALU is one of the most critical components for any processor.



Combinational logic in Verilog

- Assign a wire (not in always block)

```
3   wire comb1;
4
5   assign comb1 = 1'b0;
```

- In level triggered always block

```
3   input temp;
4   reg comb2;
5
6   always @(*) begin
7       comb2 = temp;
8   end
```

```
3   input temp;
4   reg comb2;
5
6   always @(temp) begin
7       comb2 = temp;
8   end
```

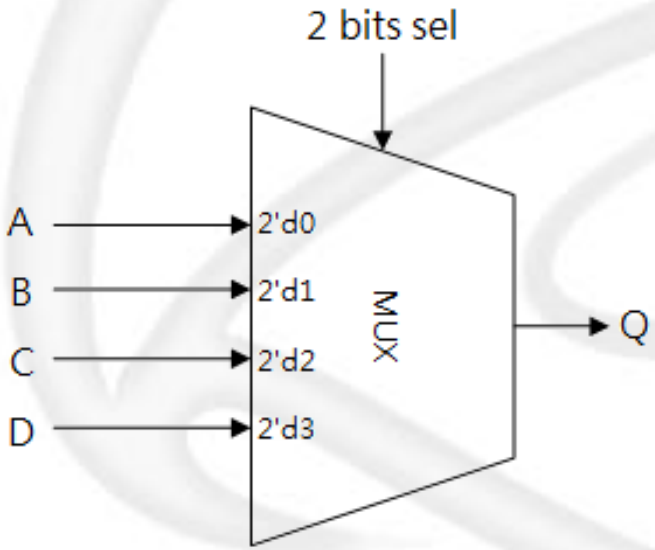
```
1   always @(*) begin
2       if(a) begin
3           temp = 0;
4       end
5       else if(b) begin
6           temp = 1;
7       end
8       else begin
9           temp = 0;
10      end
11  end
```

```
1   always @(*) begin
2       case(a)
3           0: begin
4               temp = 0;
5           end
6           1: begin
7               temp = 0;
8           end
9           default: begin
10              temp = 0;
11          end
12      endcase
13  end
```

Design of a 4-to-1 multiplexer(MUX)

Circuit Description

- Ports
  - 4 input data signals
  - 1 select signal
  - 1 output
- Output Q depends on select signal



Verilog Code (1)

```
module mux4to1 (A, B, C, D, sel, Q) ;
        input A, B, C, D ;
        input [1:0] sel;
        output Q ;
        reg Q ;
always @(*) begin
        case (sel)
        2'd0 : Q = A;
        2'd1 : Q = B;
        2'd2 : Q = C;
        2'd3 : Q = D;
        endcase
end
endmodule
```

Name & ports of the module

I/O Port declaration

Reg declaration

Internal logic

Remember "endmodule"!!

Verilog Code (2)

```
module mux4to1 (A, B, C, D, sel, Q) ;
        input A, B, C, D ;
        input [1:0] sel;
        output Q ;

assign Q = (sel == 2'd0) ? A :
                (sel == 2'd1) ? B :
                (sel == 2'd2) ? C :
                D;

endmodule
```

Name & ports of the module

I/O Port declaration

Internal logic

Remember "endmodule"!!

Testbench

Testbench (1/5)

```
`timescale 1ns/10ps
`include "mux4to1.v"

module mux4to1_tb;
  reg A, B, C, D;
  reg [1:0] sel;
  wire Q;

  mux4to1 m0(.A(A), .B(B), .C(C), .D(D), .sel(sel), .Q(Q));

initial $monitor($time, " A=%d , B=%d , C=%d , D=%d , sel=%d , Q=%d ", A, B, C, D, sel, Q);

initial begin
        A=0; B=0; C=0; D=0; sel=2'd0;    // Declare Input patterns
  #10   A=1; sel=2'd0;
  #10   A=0; B=1; sel=2'd1;
  #10   B=0; C=1; sel=2'd2;
  #10   C=0; D=1; sel=2'd3;
  #10   sel=2'd0;
  #20   $finish ;                        // Stop simulate
end

initial begin                            // Generate the waveform file
`ifdef FSDB
  $fsdbDumpfile("mux4to1.fsdb") ;
  $fsdbDumpvars;
`endif
end

endmodule
```

Module name and pins

Module instantiation

I/O ports monitoring

Input patterns declaration

Waveform file generation

Testbench (2/5)

- `timescale Unit/Precision
  #time : create time delay
  - `timescale 10ns/1ns
    #2.55 //delay 25ns

```
`timescale 1ns/10ps
`include "mux4to1.v"

module mux4to1_tb;
  reg A, B, C, D;
  reg [1:0] sel;
  wire Q;
```

- module XXX_tb;
  - No ports are needed for testbench
  - Input ports will be declared as reg
  - Output ports will be declared as wire

Testbench (3/5)

- Module instantiation
  - Instance is a complete independent and concurrently active copy of the module
  - Positional mapping
    mux4to1 m0(A, B, C, D, sel, Q);
  - Named mapping ⟵ Recommended !!!
    mux4to1 m0(.A(A), .B(B), .C(C), .D(D), .sel(sel), .Q(Q));

```
mux4to1 m0(.A(A), .B(B), .C(C), .D(D), .sel(sel), .Q(Q));
```

instance name

- Procedural block
  - Always block: execute in a loop
  - Initial block: execute while the simulation start and only execute one time

Testbench (4/5)

- $monitor
  - System task
  - Displays the values of the argument list whenever any of the arguments change
- $time
  - System function
  - Returns the current simulation time
- $finish
  - System task that ends simulation

```
initial $monitor($time, " A=%d , B=%d , C=%d , D=%d , sel=%d , Q=%d ", A, B, C, D, sel, Q);
```

%d : decimal
%h : hex
%b : binary

Testbench (5/5)

- `ifdef XXX
  `endif
  - ➔ Contents will be executed when XXX is defined
  1. Add **+define+XXX** in command line
  2. Add **`define XXX** before `ifdef
- $fsdbDumpfile
  $fsdbDumpvars
  - ➔ Verdi's PLI (Program Language Interface) which is used to help Verilog simulator dump compressed waveform (fsdb format).

```
initial begin
`ifdef FSDB
  $fsdbDumpfile("mux4to1.fsdb");
  $fsdbDumpvars;
`endif
end
```

Waveform name

Design of a Arithmetic Logic Unit (ALU)

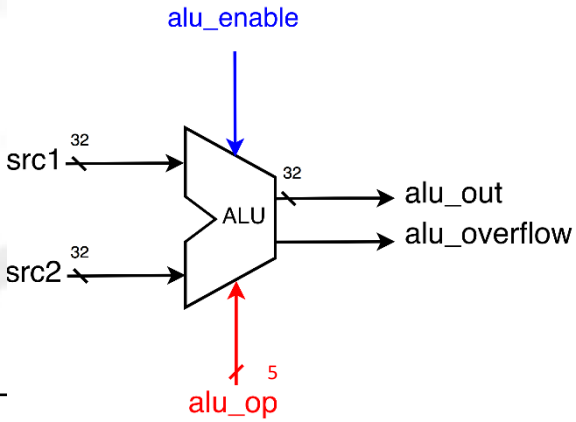
I/O of Arithmetic Logic Unit

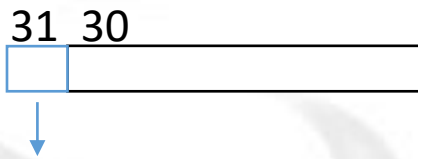
Signal	Type	Bits	Description
alu_enable	input	1	0→close 1→open
alu_op	input	5	Operation code select which operation to be executed
src1	input	32	ALU source 1
src2	input	32	ALU source 2
alu_out	output	32	ALU result
alu_overflow	output	1	0→no overflow 1→overflow

NOTE :

Definition of overflow : the result is greater than which a given register can store or represent.

ADD pos + pos = neg
ADD neg + neg = pos
SUB pos - neg = neg
SUB neg - pos = pos





sign bit (0 -> pos, 1 -> neg)

Operation

Category	alu_op	Operation	Description
Arithmetic	00000	ADD	alu_out = src1 + src2
	00001	SUB	alu_out = src1 - src2
Logical	00010	AND	alu_out = src1 & src2
	00011	OR	alu_out = src1 | src2
	00100	XOR	alu_out = src1 ^ src2
	00101	NOR	alu_out = ~(src1 | src2)
Barrel shifter	00110	SRL	alu_out = src1 >> src2
	00111	ROTR	alu_out = src1 rotate right by “src2 bits”

NOTE :

Difference between shift and rotate :

SRL 8’b1000_1101 >> 8’b0000_0011 = 8’b1111_0001

ROTR 8’b1000_1101 rotate right 8’b0000_0011 = 8’b1011_0001

SRLU 8’b1000_1101 >>(unsigned) 8’b0000_0011 = 8’b0001_0001

Reference Code (1/2)

```
`timescale 1ns/10ps

// ---------------------- define ---------------------- //

`define DataSize   32
`define ALUopSize   5
//define ALUop
`define ADDop      5'b00000
`define SUBop      5'b00001
`define ANDop      5'b00010
`define ORop       5'b00011
`define XORop      5'b00100
`define NORop      5'b00101
`define SRLop      5'b00110
`define ROTRop     5'b00111

module ALU (alu_enable, alu_op, src1, src2, alu_out, alu_overflow);

// ---------------------- input  ---------------------- //
input                    alu_enable;
input   [`ALUopSize-1:0] alu_op;
input   [`DataSize-1:0]  src1;
input   [`DataSize-1:0]  src2;

// --------------------- output ---------------------- //
output   [`DataSize-1:0] alu_out;
output                   alu_overflow;

// ----------------------  reg   ---------------------- //
reg     [`DataSize-1:0]  alu_out;
reg                      alu_overflow;
reg     [63:0]           temp;
```

Increase modifiability and readability

Define:

1. Can be used in any module after declare
2. declare:
`define ADD 5'b00000
3. use:
`ADD

Parameter:

1. Can be used in local module only
2. declare:
parameter ADD = 5'b00000;
3. use:
ADD

Difference between define and parameter

Reference Code (2/2)

If there is any change on the **RHS** of "=", the always block will be executed

readability

```
always@(*)begin
  alu_overflow = 1'b0;
  if(alu_enable)begin
    case(alu_op)
      `ADDop  :  begin
                 alu_out = src1 + src2;
                 if((src1[31]==1'b0 && src2[31]==1'b0 && alu_out[31]==1'b1)||
                    (src1[31]==1'b1 && src2[31]==1'b1 && alu_out[31]==1'b0))
                   alu_overflow = 1'b1;
                 else
                   alu_overflow = 1'b0;
                 end
      `SUBop  :  begin
                 alu_out = src1 - src2;
                 if((src1[31]==1'b0 && src2[31]==1'b1 && alu_out[31]==1'b1)||
                    (src1[31]==1'b1 && src2[31]==1'b0 && alu_out[31]==1'b0))
                   alu_overflow = 1'b1;
                 else
                   alu_overflow = 1'b0;
                 end
      `ANDop  :  alu_out = src1 & src2;
      `ORop   :  alu_out = src1 | src2;
      `XORop  :  alu_out = src1 ^ src2;
      `NORop  :  alu_out = ~(src1 | src2);
      `SRLop  :  alu_out = $signed(src1) >>> $signed(src2);
      `ROTRop :  begin
                 temp = {src1,src1};
                 temp = temp >> (src2%32);
                 alu_out = temp[31:0];
                 end
      default ;  begin
                 alu_out = 32'b0;
                 end
    endcase
  end
  else
    alu_out = 32'b0;
  end
endmodule
```

case

full case

更正为:

The Conversion between RGB and Grayscale

Image Format

- **RGB (Red, Green, Blue)**
  - Each pixel can be represented in the computer memory or interface as binary values for the red, green, and blue color components.
- **Current typical display adapters use 24 bits of information for each pixel.**
  - Each color has 8 bits (0-255)
  - Represent as (255, 0, 0)
  - In hexadecimal #FF0000
- **Total color**
  - **256 * 256 * 256 = 16,777,216**

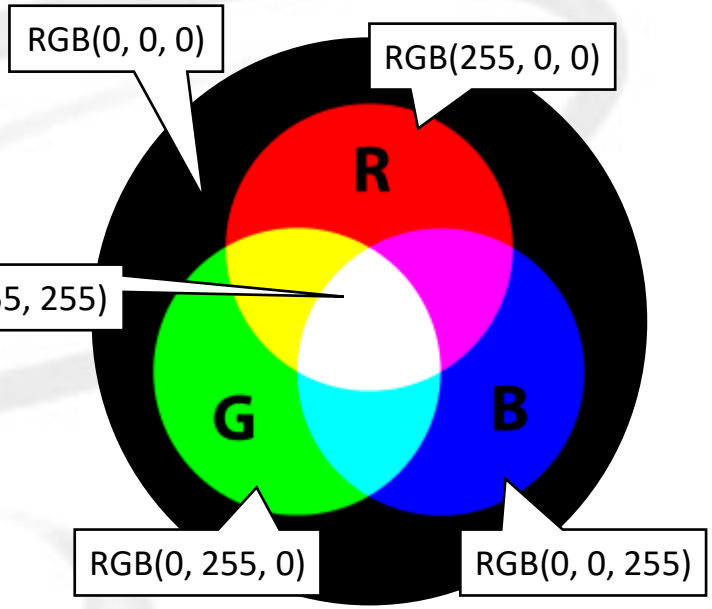


Image Format

- Image decomposed into red, green and blue component

Red component

Green component

Blue component

Grayscale

- How to turn RGB image into grayscale?
  - Suppose the RGB value of a pixel is (r, g, b)
  - The grayscale y=0.299r+0.587g+0.114b (0-255)
  - The pixel is now (y, y, y)

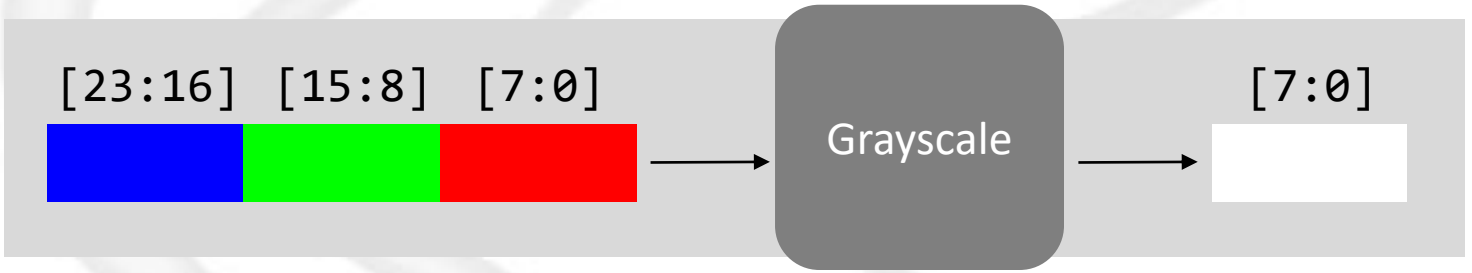
Pure Red (255,0,0)	Equivalent Gray (76,76,76)
Pure Green (0,255,0)	Equivalent Gray (150,150,150)
Pure Blue (0,0,255)	Equivalent Gray (29,29,29)
Cyan (0,255,255)	Equivalent Gray (179,179,179)
Magenta (255,0,255)	Equivalent Gray (105,105,105)
Yellow (255,255,0)	Equivalent Gray (226,226,226)
Brown (158,85,54)	Equivalent Gray (103,103,103)
Olive (155,160,52)	Equivalent Gray (146,146,146)
Purple (100,0,150)	Equivalent Gray (47,47,47)

- Can you convert a grayscale value back to an RGB color code?
  - NO!

(0,170,0)	(100,100,100)
(230,53,0)	(100,100,100)
(80,83,240)	(100,100,100)
(230,14,200)	(100,100,100)

Introduction

- There is an operation that changes RGB color image into a grayscale image. In this Lab we are going to design a unit that does this conversion.
- This conversion component takes 24-bit RGB color values as input, convert them and produce 8-bit grayscale values.
- There are 3 segments in a 24-bit color value, bits 23~16 for blue, bits 15~8 for green and bits 7~0 for red.



The Conversion Unit (1)

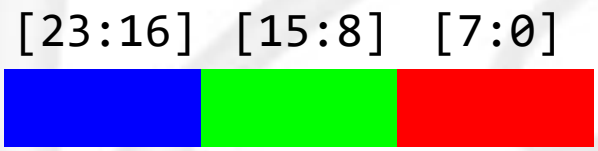
RGB to grayscale

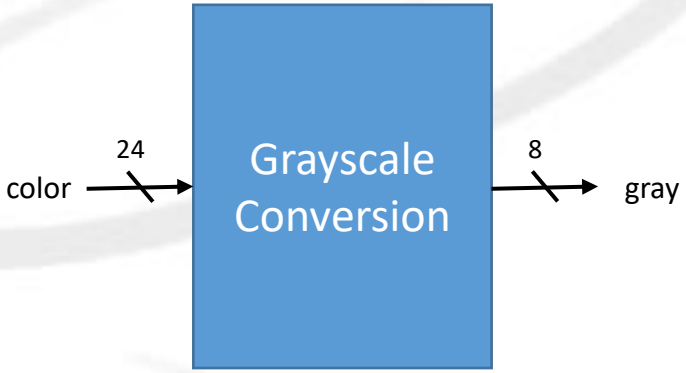
- The grayscale operation y=0.299r+0.587g+0.114b
- In this Lab, for simplicity, please use the operation as follows:

$$y = 0.3125r + 0.5625g + 0.125b$$

where r, g, b are the values of segments of the 24-bit input respectively, and y is the 8-bit output.

- 24-bit input for pixel RGB value
- 8-bit output for pixel grayscale value





The Conversion Unit (2)

- **The operation**

$$y = 0.3125r + 0.5625g + 0.125b$$

- **Using straight decimals may be not synthesizable,**
- **e.g. `0.3125 * … .`**
- **Here use fixed point numbers instead.**

- **How to implement multiplication and decimals?**

A way to implement multiplication (1)

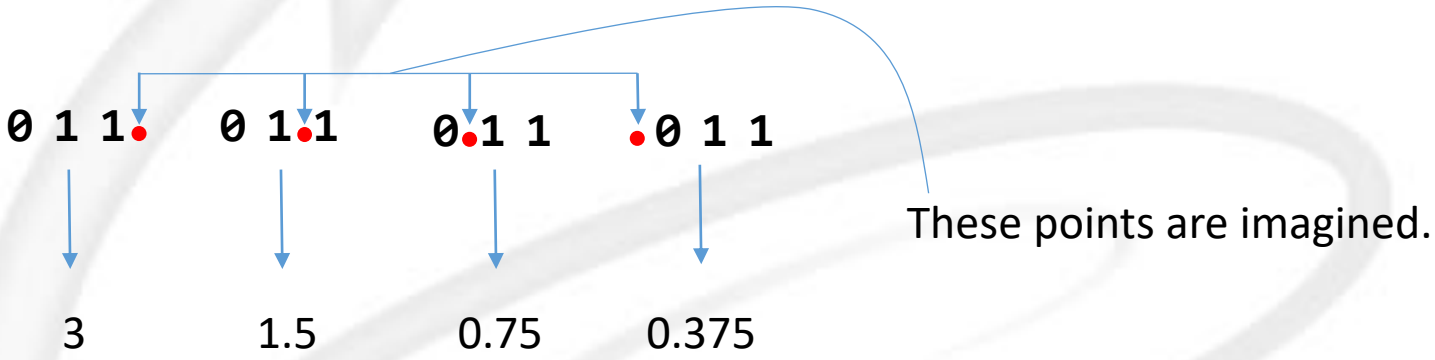
- **Shift and addition**
  - 1. x*4 → x<<2
  - 2. x*5 → (x<<2) + x
  - 3. x*6 → (x<<2) + (x<<1)

1 0 ⟶ 1 0 0 0

- **Be careful of the bit-lengths of operands and the result.**
  - e.g.
```
wire [3:0] y;
assign y = 4'h3 << 3;
```
  - 4’b0011 << 3
  - 4’b0110 << 2
  - 4’b1100 << 1
  - y is 4’b1000

A way to implement multiplication (2)

- How to deal with decimals?
  - Designers determine the position of the decimal point.
  - For example, a unsigned number 3'b011 can represent 3, 1.5, 0.75 or 0.375. It depends on the design.



  - You should note that for 0010_b * 0011_b = 0110_b, if you take 0010_b to be 0.5 and 0011_b to be 1.5, the result 0110_b should represent 0.75 (0.110_b) rather than 6.
  - Notice how the position of the decimal point changes.
  - Be careful when handling signed numbers.

Examples

- Consider the operation:
- y = x * 0.5 , where x and y are signed numbers.
- How can we deal with it?
- Suppose x and y are 4-bit integers.

- Note: there are methods to turn a decimal into an integer, e.g. rounding, rounding up, rounding down.

Examples (1)

- y = x * 0.5
- Using shift

```
1  module lab4_example1(
2    input [3:0] x,
3    output [3:0] y
4    );
5
6    wire signed [4:0] x_temp;
7    reg signed [4:0] y_temp;
8
9    assign x_temp[4:1] = x;
10   assign y = y_temp[4:1]; //Round down the number to an integer.
11
12   always@(*) begin
13     y_temp = x_temp >>> 1;
14   end
15 endmodule
```

Here we emphasize that y_temp[4:1] is the integer part and y_temp[0] is the decimal part, though y_temp[0] is not needed for the result, because of rounding down. See example 3 for a simpler style.

Sign extension shift right

Examples (2)

- y = x * 0.5
-

```
1  module lab4_example3(
2    input [3:0] x,
3    output [3:0] y
4    );
5
6    assign y = $signed(x) >>> 1;
7
8  endmodule
```

Examples (3)

- y = x * 0.5
-

```
1  module lab4_example4(
2    input [3:0] x,
3    output [3:0] y
4    );
5
6    assign y = {x[3], x[3:1]};
7
8  endmodule
```

Homework

Homework

- 1. Due day : 2021-03-17, Wed, 15:00
- 2. Contents :
  - Prob. A
    - Design a 8-to-1 multiplexer.
    - Write a testbench to verify if the multiplexer is functionally correct.
  - Prob. B
    - Design a 32-bit ALU.
  - Prob. C
    - Design a unit that can convert RGB to grayscale.
    - Do not use straight "*" for multiplication in your code. Use another method to implement multiplication.
    - You can use behavior modeling in this problem.
  - Finish the report.
  - More detailed information is described in the report.
- 3. Name the report Lab3_StudentID.doc (Do not paste your code in it!)

- 備註1：請將所有檔案壓成Lab3_ StudentID.tar上傳
- e.g. Lab3_E20000000.tar
- 備註2：作業視完成度計分

Simulation Commands

- These commands will be used to simulate your designs.

Problem		Command
ProbA	Compile	% ncverilog mux8to1.v
	Simulate	% ncverilog mux8to1_tb.v +define+FSDB +access+r
ProbB	Compile	% ncverilog ALU.v
	Simulate	% ncverilog ALU_tb.v +define+FSDB +access+r
ProbC	Compile	% ncverilog grayscale.v
	Simulate	% ncverilog grayscale_tb.v +define+FSDB +access+r

- Please include all needed Verilog files in your testbench.

The Testbench for Prob. C (1)

Input patterns

```
initial begin
  color = 'h0;
  #`INTERVAL color = 24'hffffff;
  #`INTERVAL color = 24'h101010;
  #`INTERVAL color = 24'h010101;
  #`INTERVAL color = 24'h010205;
  #`INTERVAL color = 24'h7bde31;
  #`INTERVAL color = 24'hcb9e96;
  #`INTERVAL color = 24'h5366a5;
  #`INTERVAL color = 24'hbc524c;
  #`INTERVAL color = 24'h48162f;
  #`INTERVAL color = 24'hfeadef;
end
```

Correct results

```
initial begin
  golden[0] = 'h0;
  golden[1] = 'hff;
  golden[2] = 'h10;
  golden[3] = 'h1;
  golden[4] = 'h3;
  golden[5] = 'h9c;
  golden[6] = 'ha1;
  golden[7] = 'h77;
  golden[8] = 'h5d;
  golden[9] = 'h24;
end
```

The Testbench for Prob. C (2)

Output comparison

```
initial begin
  err = 0;
  #(`INTERVAL-1)
  for(i=0;i<10;i=i+1)begin
    if(gray===golden[i])
      $display("Result No.%0d is correct.", i+1);
    else begin
      $display("Result No.%0d is not correct. The result is %d, but %d is expected.", i+1, gray, golden[i]);
      err = err + 1;
    end
    #`INTERVAL;
  end
  if(err===0)begin
    $display("                                          ");
    $display("    ****************************************");
    $display("    *                                      *");
    $display("    *  Congrats! All results are correct.  *");
    $display("    *                                      *");
    $display("    ****************************************");
    $display("                                          ");
  end
  $finish;
end
```

Remind

1. DO NOT paste code in report file!

2. ONLY *.tar* formats are accepted!

※ Violating these rules will lead to credit *deduction*

Thank you
For your attention!!