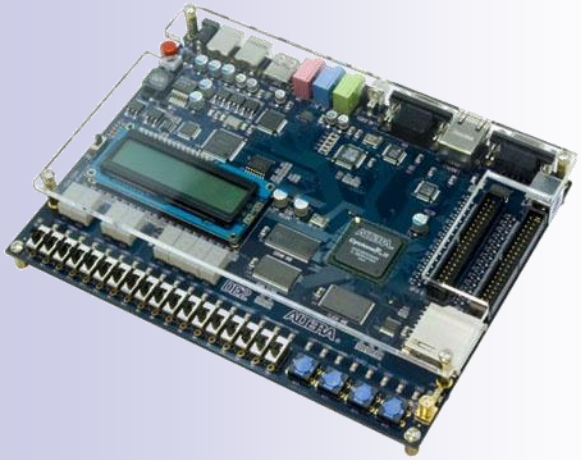
Logic Systems and Processors
cz:Logické systémy a procesory



Lecturer: Richard Šusta

richard@susta.cz, susta@fel.cvut.cz,
+420 2 2435 **7359**

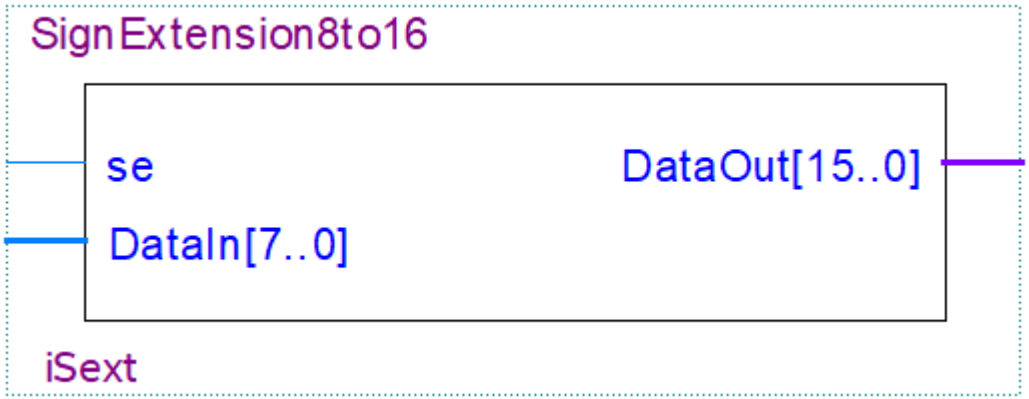
Version V1.1 - Oct 20, 2024
- corrections of typos

CTU-FEE in Prague, CR – subject BE5B35LSP

Two Examples

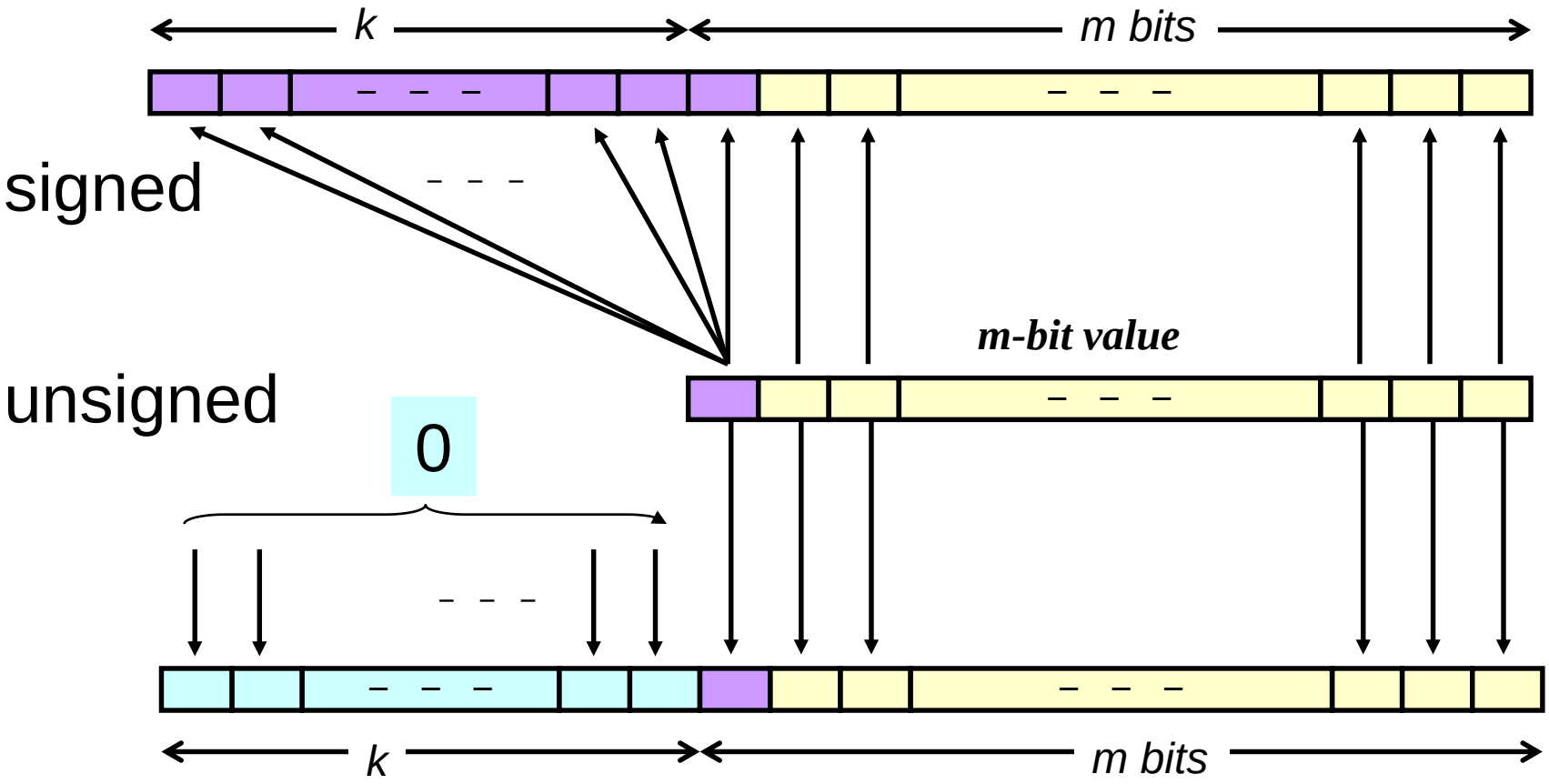
Both tasks focus on the coding styles appropriate for the description of circuits, which we consider to be more important than the perfect knowledge of the catalog of VHDL or Verilog statements.

Image: http://dr-wo.de/



Create Sign Extension Circuit

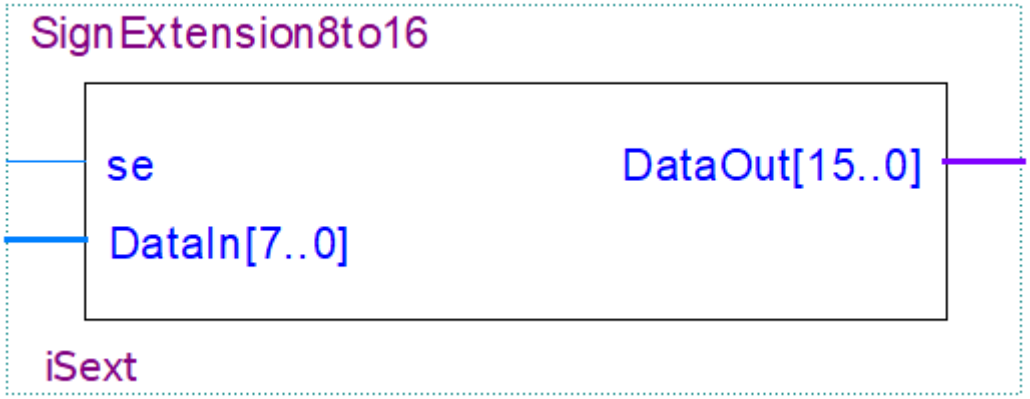
Unsigned / Signed Extensions



*The **sign extension** is commonly abbreviated as **sext***

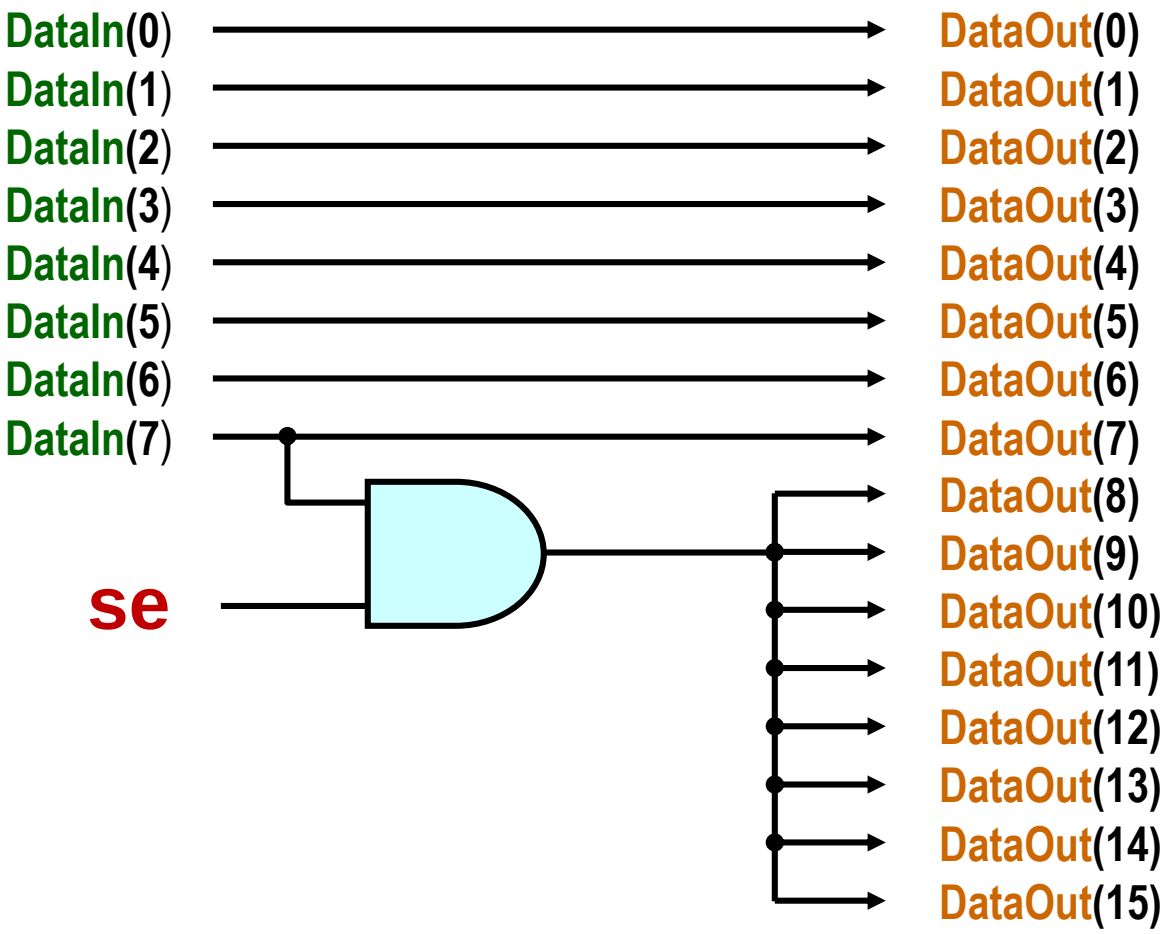
VHDL Entity - Step 1

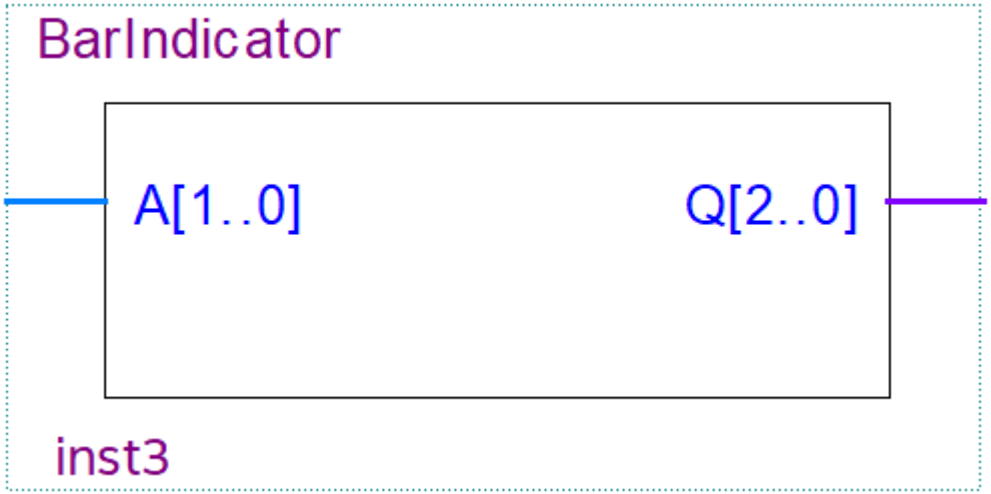
```
library ieee; use ieee.std_logic_1164.all;
entity Sext is
port( se : std_logic;
      DataIn : in std_logic_vector(7 downto 0);
      DataOut : out std_logic_vector(15 downto 0));
end entity;
```

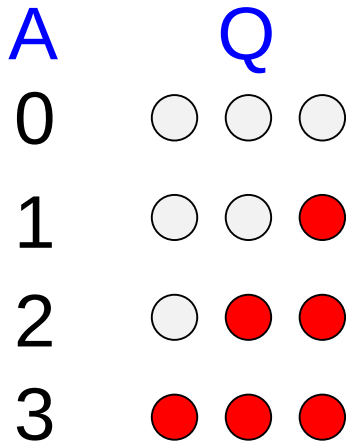


VHDL -> We describe Architecture - Step 2

Let's draw a schematic...

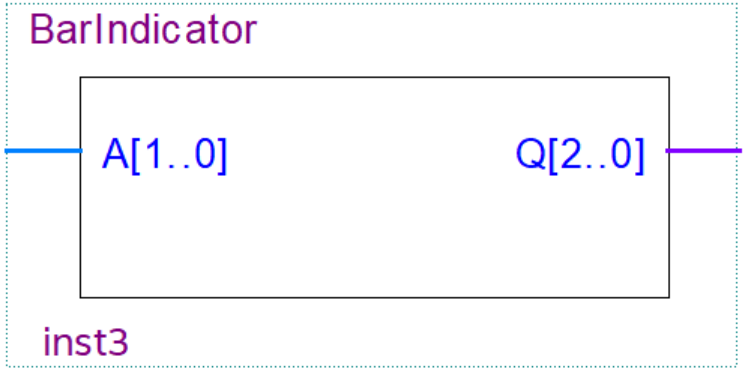






Create Bar Indicator

```
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity BarIndicator is
    port
    (  A : in std_logic_vector(1 downto 0);
       Q : out std_logic_vector(2 downto 0)  );
end entity;
```



Let's together discuss several possible VHDL architectures.

Overview of Descriptions

discussed during the lecture

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity BarIndicator is  port    (  A : in std_logic_vector(1 downto 0);    Q : out std_logic_vector(2 downto 0)    );
end entity;
/* C algorithm is great for compilation to assembler  but improper for circuits.
   uint bar(uint A) {  return (1<<A)-1;  }
*/
-- A list of values without hint ☹
architecture rtlMux of BarIndicator is
begin   with A select    Q <=    "000" when "00",    "001" when "01",   "011" when "10",    "111" when others;
end architecture;
-- We add hint
architecture rtlWhen of BarIndicator is
signal ua:unsigned(A'RANGE):="00";
begin   ua<=unsigned(A);
    Q(0) <=    '1' when ua>=1 else '0';
   Q(1) <=    '1' when ua>=2 else '0';
    Q(2) <=    '1' when ua>=3 else '0';
end architecture;
-- The description that can be quickly extended to wider output
architecture rtlGenerate of BarIndicator is
signal ua:unsigned(A'RANGE):="00";
begin   ua<=unsigned(A);    iloop:for i in 0 to 2 generate   Q(i) <='1' when i<ua else '0';    end generate;
end architecture;
```

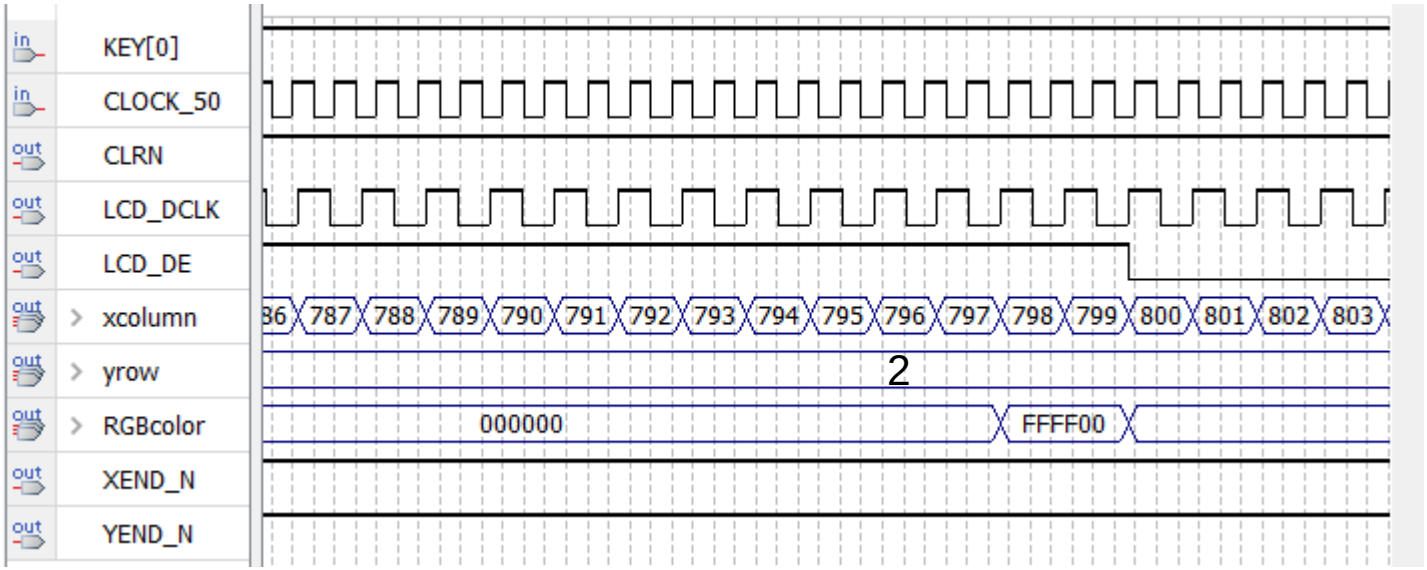
LCD Synchronization Generator

in VHDL

Image pixels are nowadays sent to LCDs over serial lines. The RBG colors are in screens stored in capacitors that require periodic refreshing.

```
// its code is a circuit implementation of the C code
unsigned short int xcolumn, yrow;
unsigned int RGBcolor;
bool LCD_DE, XEND_N, YEND_N;
for (yrow = 0; yrow < 525; yrow++)
{ for (xcolumn = 0; xcolumn <1024; xcolumn++)
    {  LCD_DE = xcolumn>=800 || yrow>=480 ? 0 : 1;
       XEND_N = xcolumn==1023 ? 0 : 1;
       YEND_N = yrow==524 ? 0 : 1;
       RGBcolor = LCDlogic( xcolumn, yrow ); // LCD Task
    }
}
```

Sample of the course



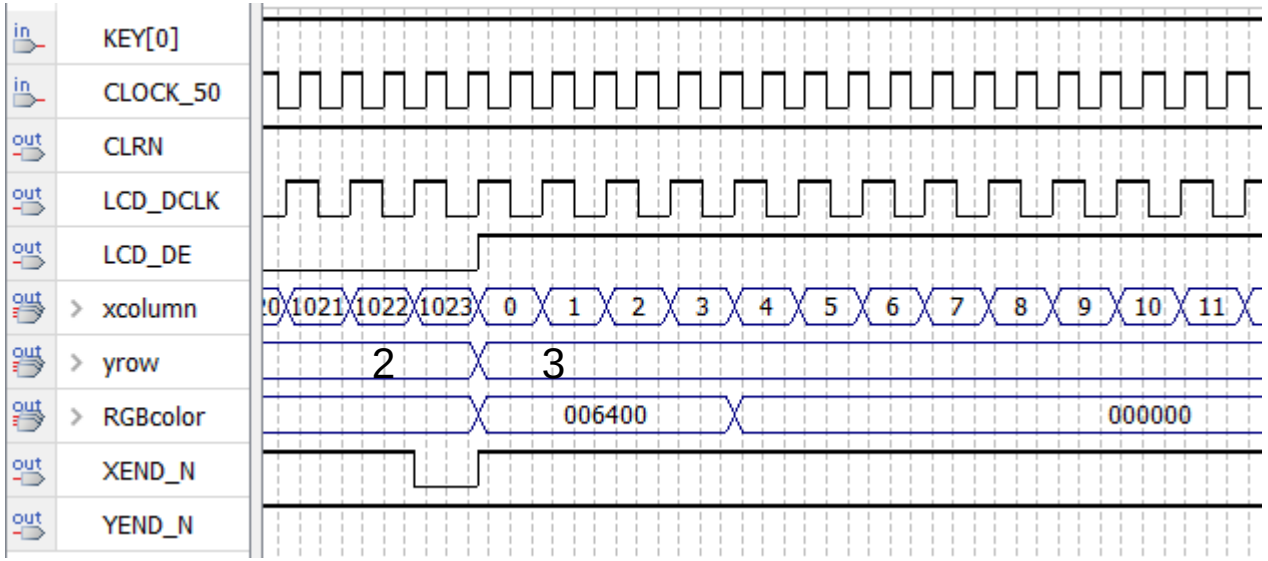
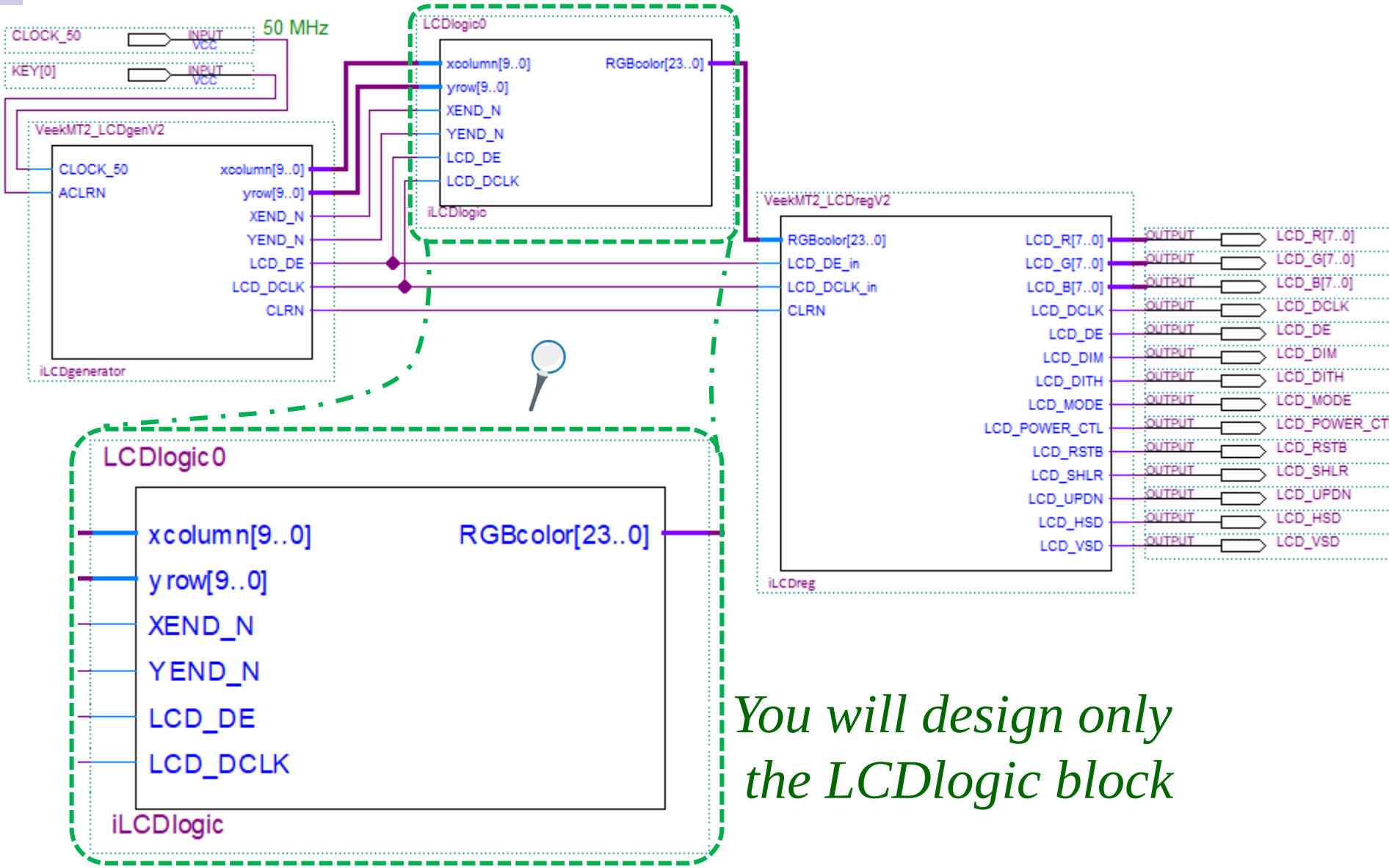


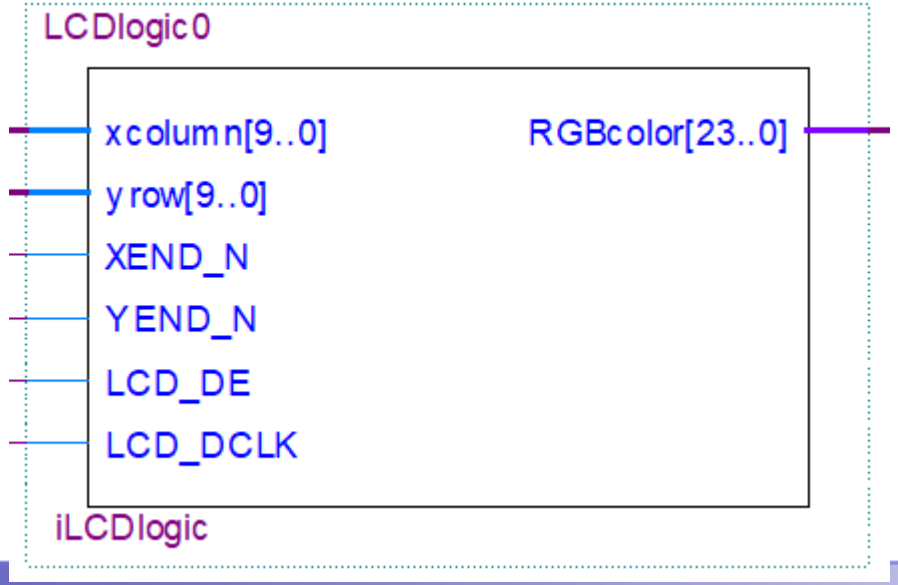
Diagram 2nd Tasks



You will design only the LCDlogic block

Your Entity LCDlogic0

```
library ieee, work; use ieee.std_logic_1164.all;  use ieee.numeric_std.all;
entity LCDlogic0 is
 port(
   xcolumn, yrow : in unsigned(9 downto 0) := (others => '0'); -- x,y pixel
   XEND_N, YEND_N : in std_logic := '0';    -- max xcolumn/yrow
   LCD_DE : in std_logic := '0';          -- DataEnable control signal of LCD
   LCD_DCLK : in std_logic := '0';     -- LCD data clock, exactly 33 MHz
   RGBcolor : out std_logic_vector(23 downto 0):=(others=>'0') ); -- color data
end;
```

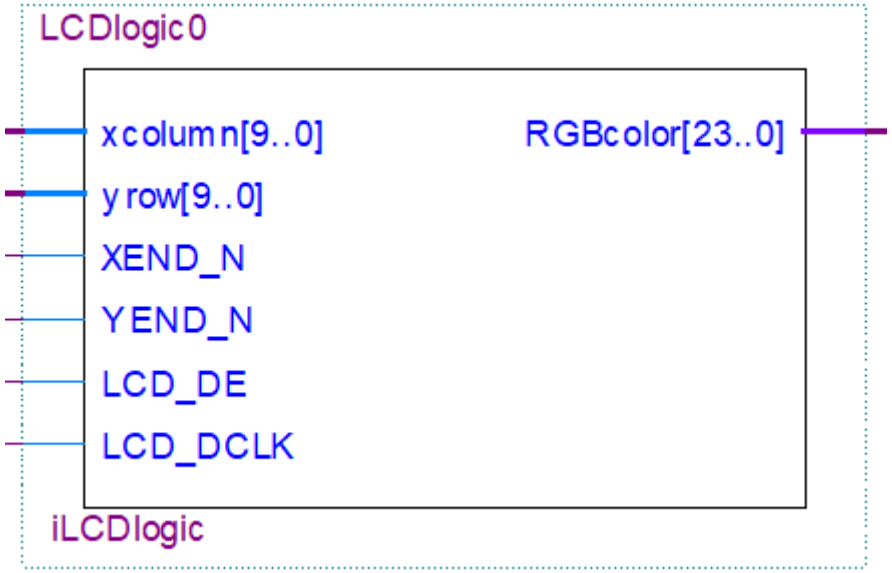


Let's wrap up handy definitions

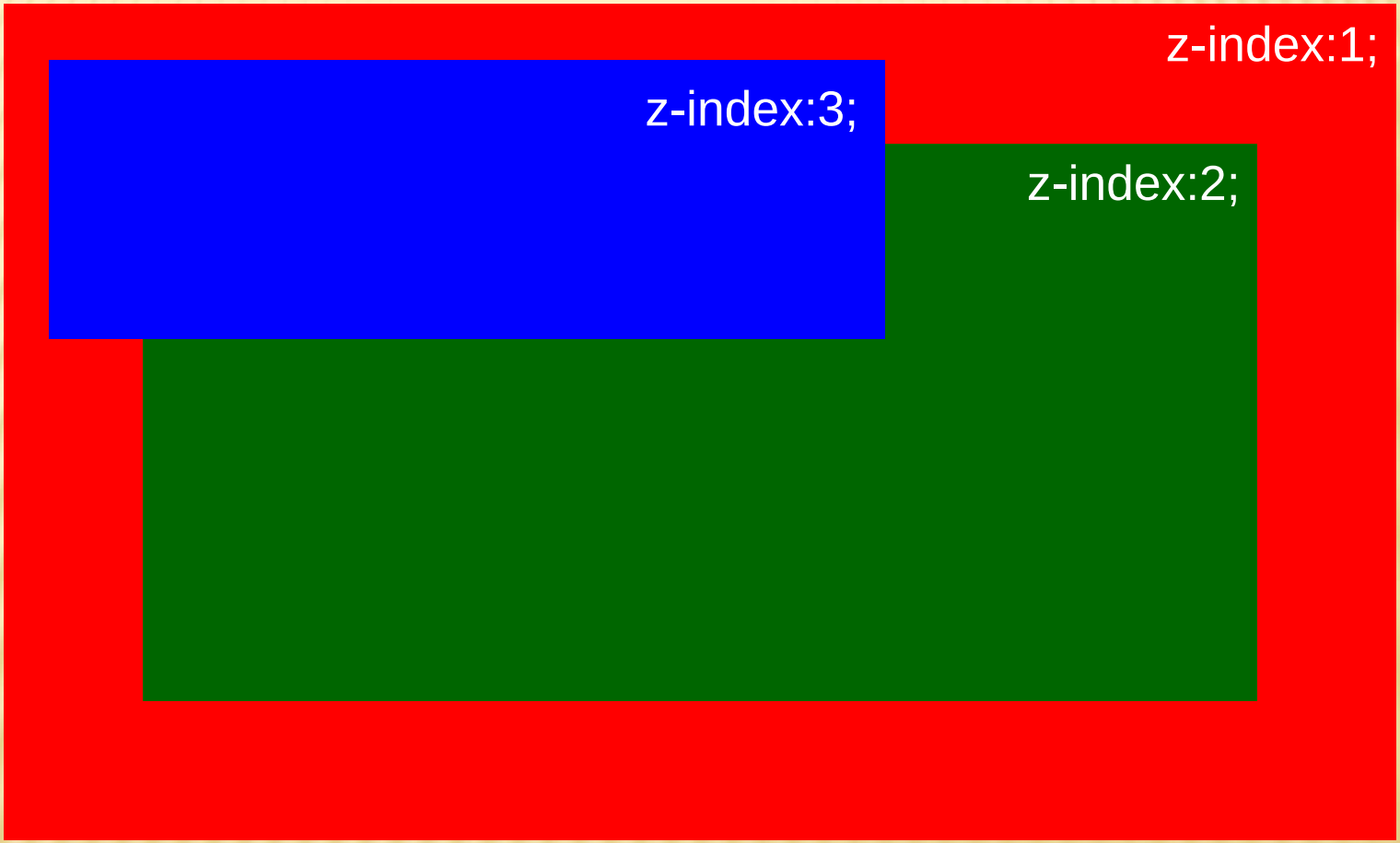
```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
package LCDpackV2 is
  constant LCD_WIDTH : integer := 800;  -- the visible xcolumn axis
  constant LCD_HEIGHT : integer := 480;  -- the visible yrow axis
  constant XCOLUMN_MAX : integer :=1023;   -- max. xcolumn lies in invisible part)
  constant YROW_MAX    : integer := 524;  -- max. yrow lies in invisible part)
  subtype  xy_t is unsigned(9 downto 0);  --xcolumn and yrow coordinates
  constant XY_ZERO : xy_t := (others=>'0');
  subtype RGB_t is std_logic_vector(23 downto 0); --color R:23..16, G:15..8, B:7..0
  -- 16 Windows colors palette
  constant BLACK   : RGB_t:=X"000000";   constant BLUE   : RGB_t:=X"0000FF";
  constant GREEN : RGB_t:=X"008000";    constant NAVY: RGB_t:=X"000080"; -- plus others
-- Some useful functions
  function ToRGB(r, g, b:natural) return RGB_t; -- plus others
end package LCDpackV2;
package body LCDpackV2 is  /* bodies of functions */
end package body LCDpackV2;
```

Your Entity LCDlogic0

```
library ieee, work; use ieee.std_logic_1164.all;  use ieee.numeric_std.all;
entity LCDlogic0 is
 port( xcolumn, yrow  : in  xy_t := XY_ZERO; -- x,y-coordinates of pixel
       XEND_N, YEND_N : in  std_logic := '0'; -- the last xcolumn/yrow
       LCD_DE     : in  std_logic := '0';       -- DataEnable LCD
       LCD_DCLK : in  std_logic := '0';      -- LCD data clock, 33 MHz
       RGBcolor    : out RGB_t := BLACK ); -- color assigned to x,y pixel
end;
```

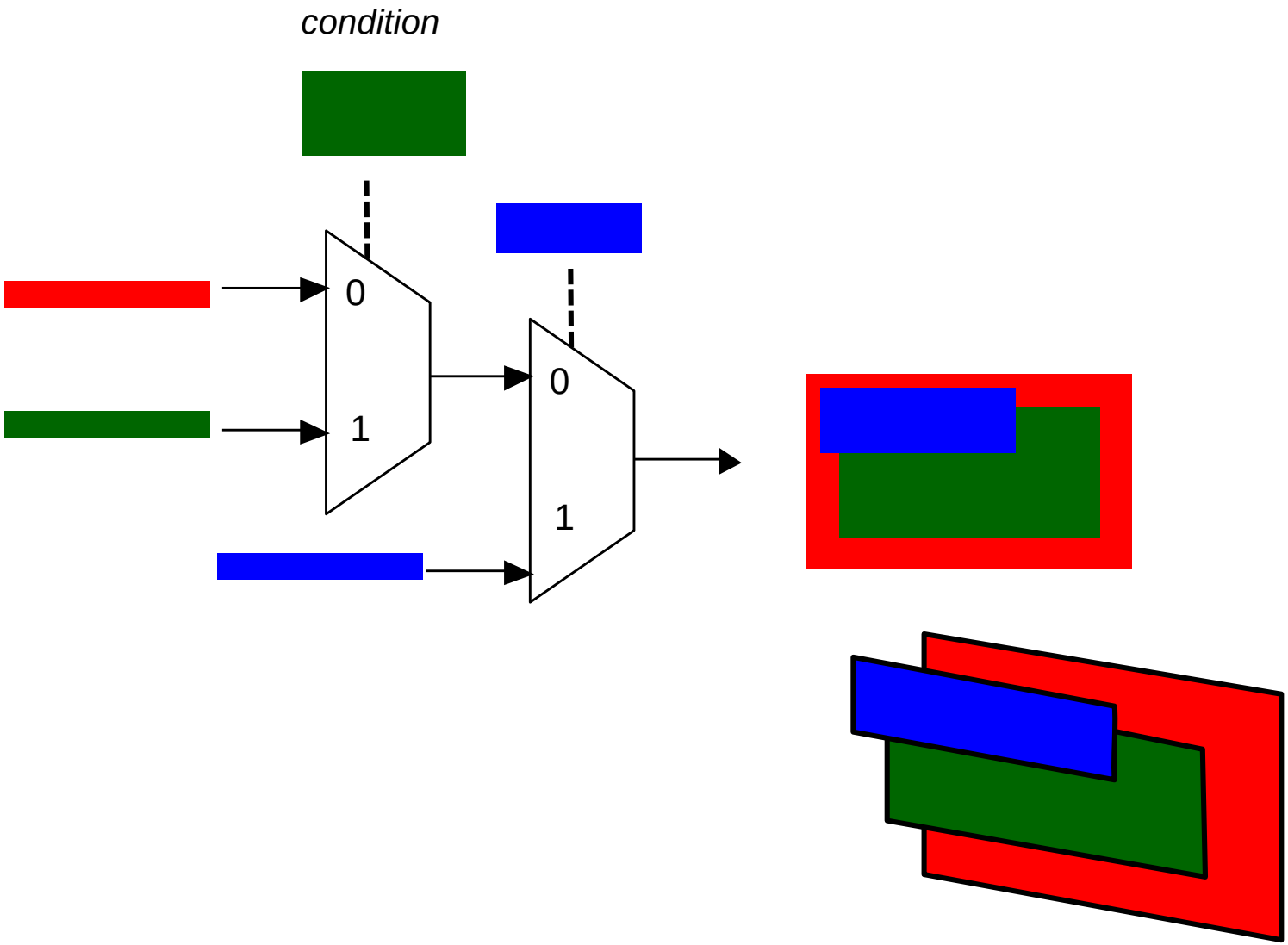


Priority task - visibility in a picture



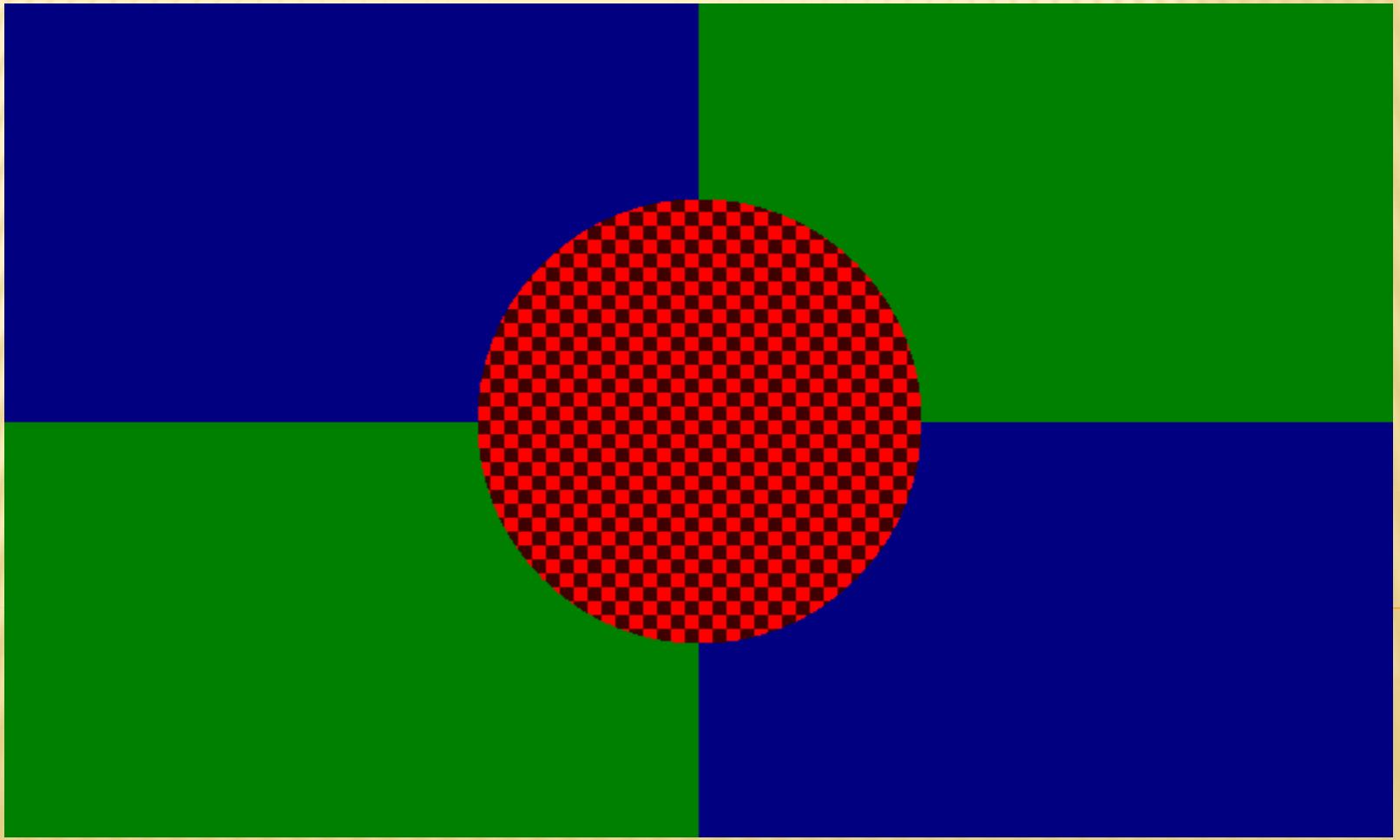
Z-index in CSS Styles

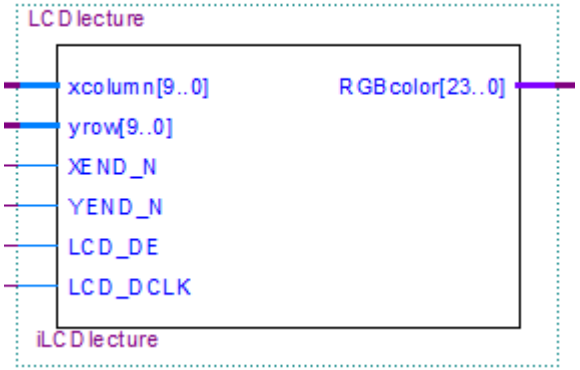
Image by Cascade of Multiplexers



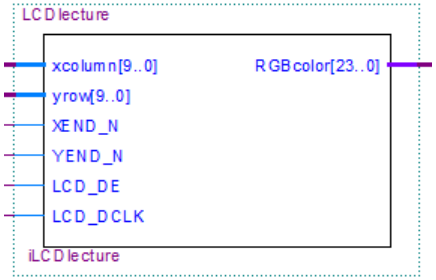
We create the image below in two ways

1. **In Concurrent Domain**
2. **In Sequential Domain**



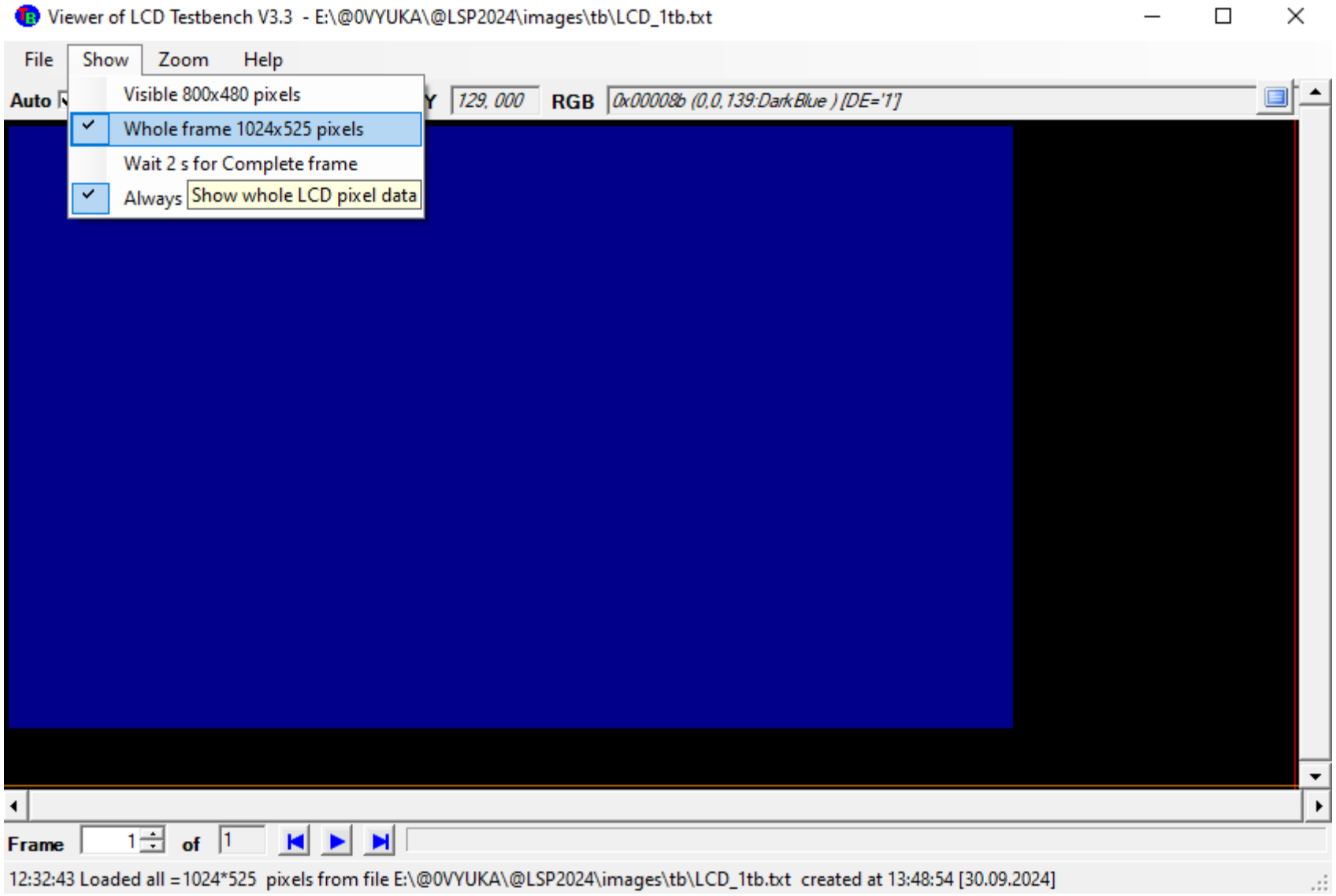



```
library ieee, work; use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- num
use work.LCDpackV2.all; -- our definitions
entity LCDlecture is
 port( xcolumn, yrow  : in  xy_t := XY_ZERO; -- x,y-coordinates of pixel
       XEND_N, YEND_N : in  std_logic := '0'; -- the last xcolumn/yrow
       LCD_DE         : in  std_logic := '0'; -- DataEnable LCD
       LCD_DCLK       : in  std_logic := '0'; -- LCD data clock, 33 MHz
       RGBcolor       : out RGB_t := BLACK);       --  color data
end entity;
```




```
architecture concurrentImage of LCDlecture is

begin -- architecture

        RGBcolor <= BLACK when LCD_DE='0' else  NAVY;

end architecture;
```

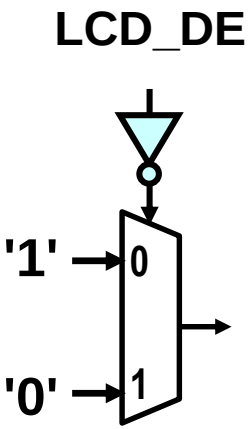
Appearance of Whole LCD Frame



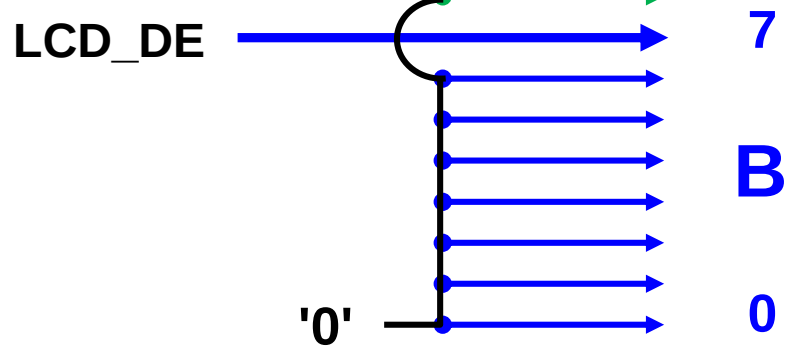
The red lines in the Viewer only highlight the last row, column. Their pixels have black colors.

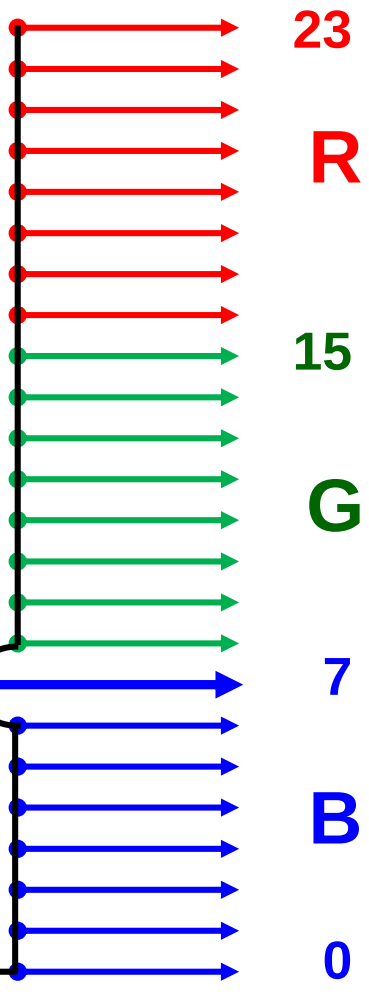
Implementation in Circuit

```
RGBcolor <= X "000000" when LCD_DE='0'
                --BLACK
            else X "000080"; -- NAVY
```

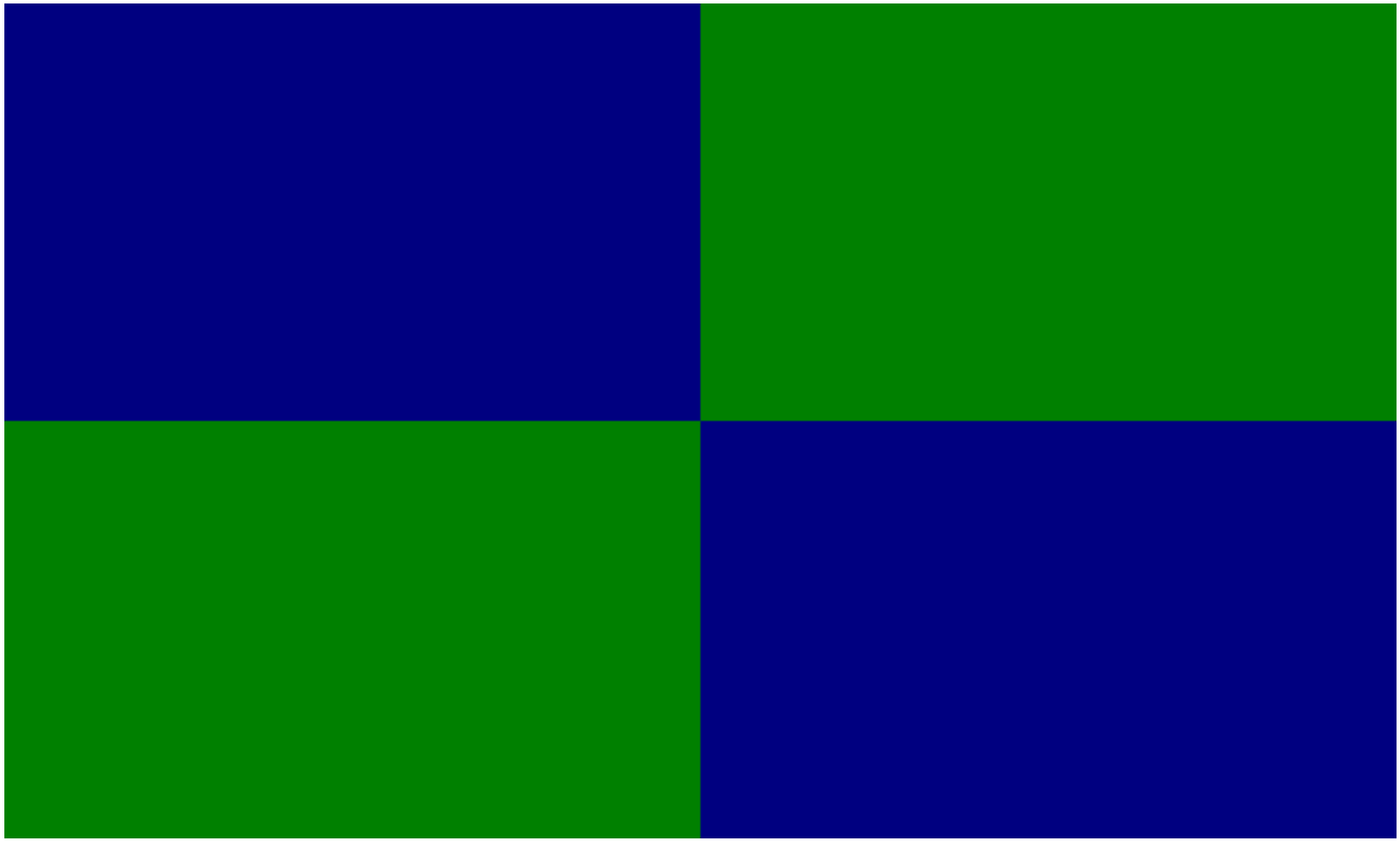


LCDcolor(7)





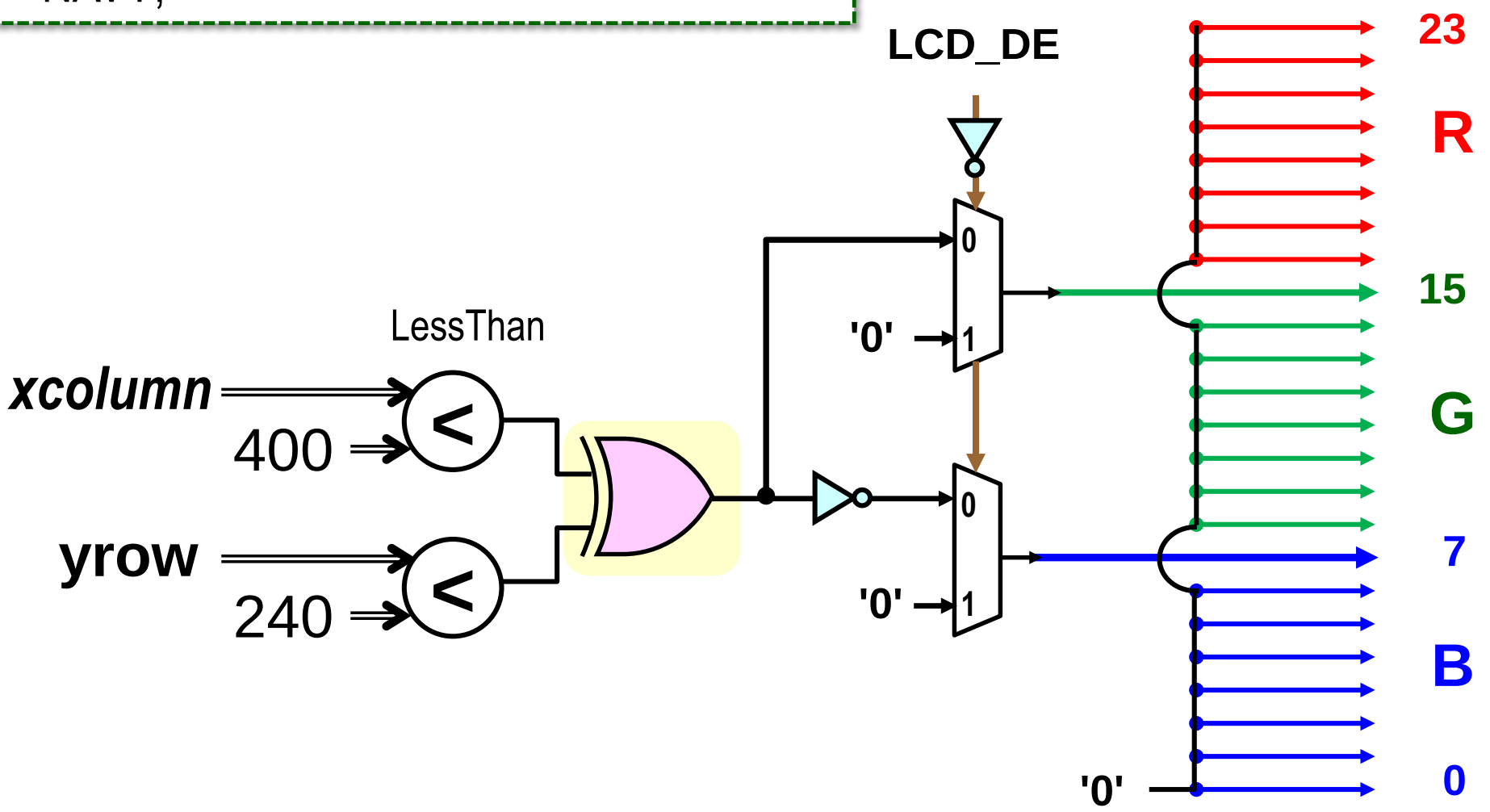
Modifying to Two XOR Squares



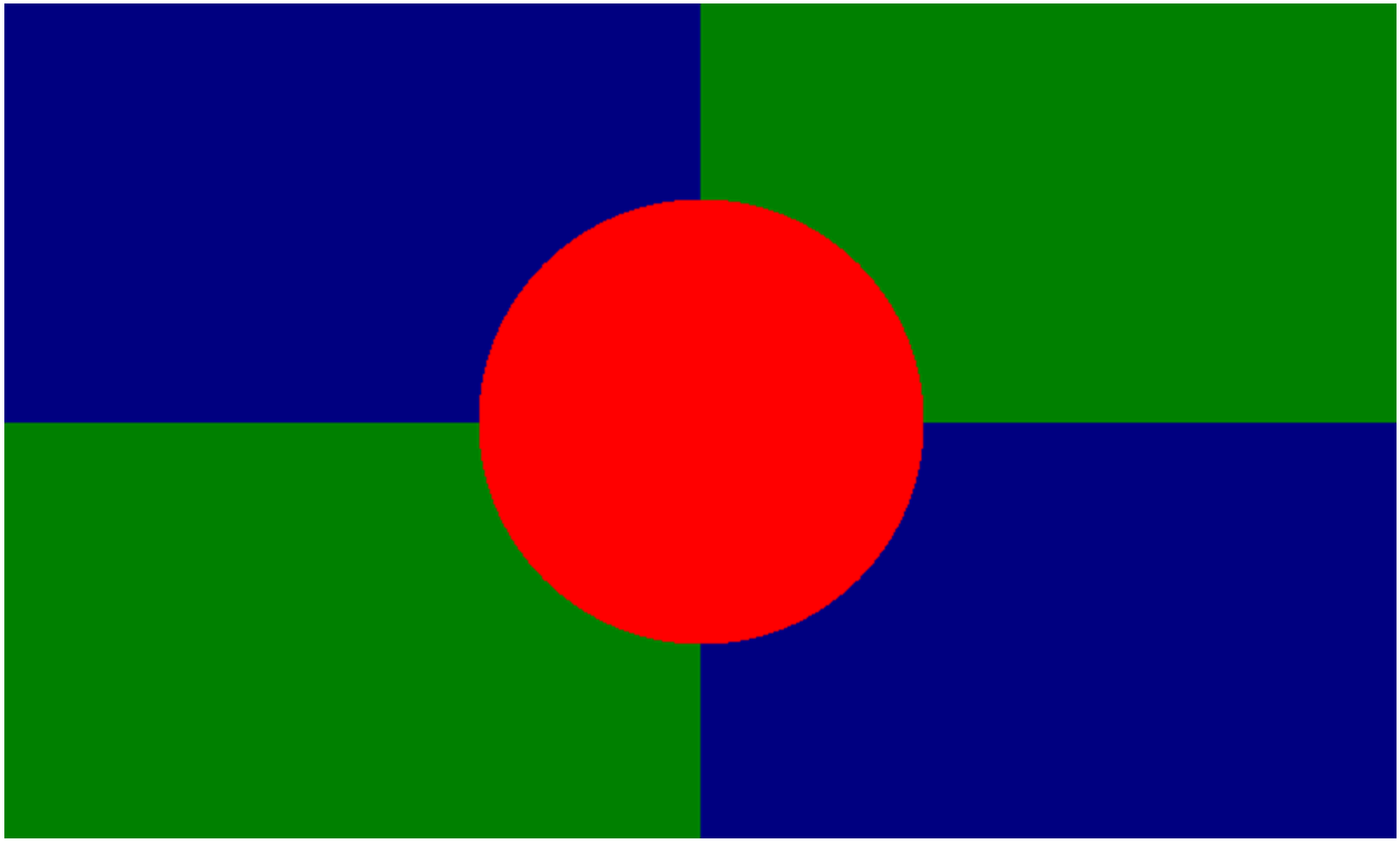
Xor Squares

```
architecture concurrentImage of LCDlecture is
signal x, y : integer range 0 to 2**xy_t'LENGTH-1:=0;
constant LXC:integer := LCD_WIDTH/2; -- visible xcolumn center
constant LYC:integer := LCD_HEIGHT/2; -- visible yrow center

begin

  x<=to_integer(xcolumn); y<=to_integer(yrow);

RGBcolor <=  BLACK when LCD_DE='0' else

                GREEN when x<LXC xor y<LYC else

                NAVY;

end architecture;
```

```
RGBcolor <=  BLACK when LCD_DE='0' else
     GREEN when x<LXC xor y<LYC else
     NAVY;
```



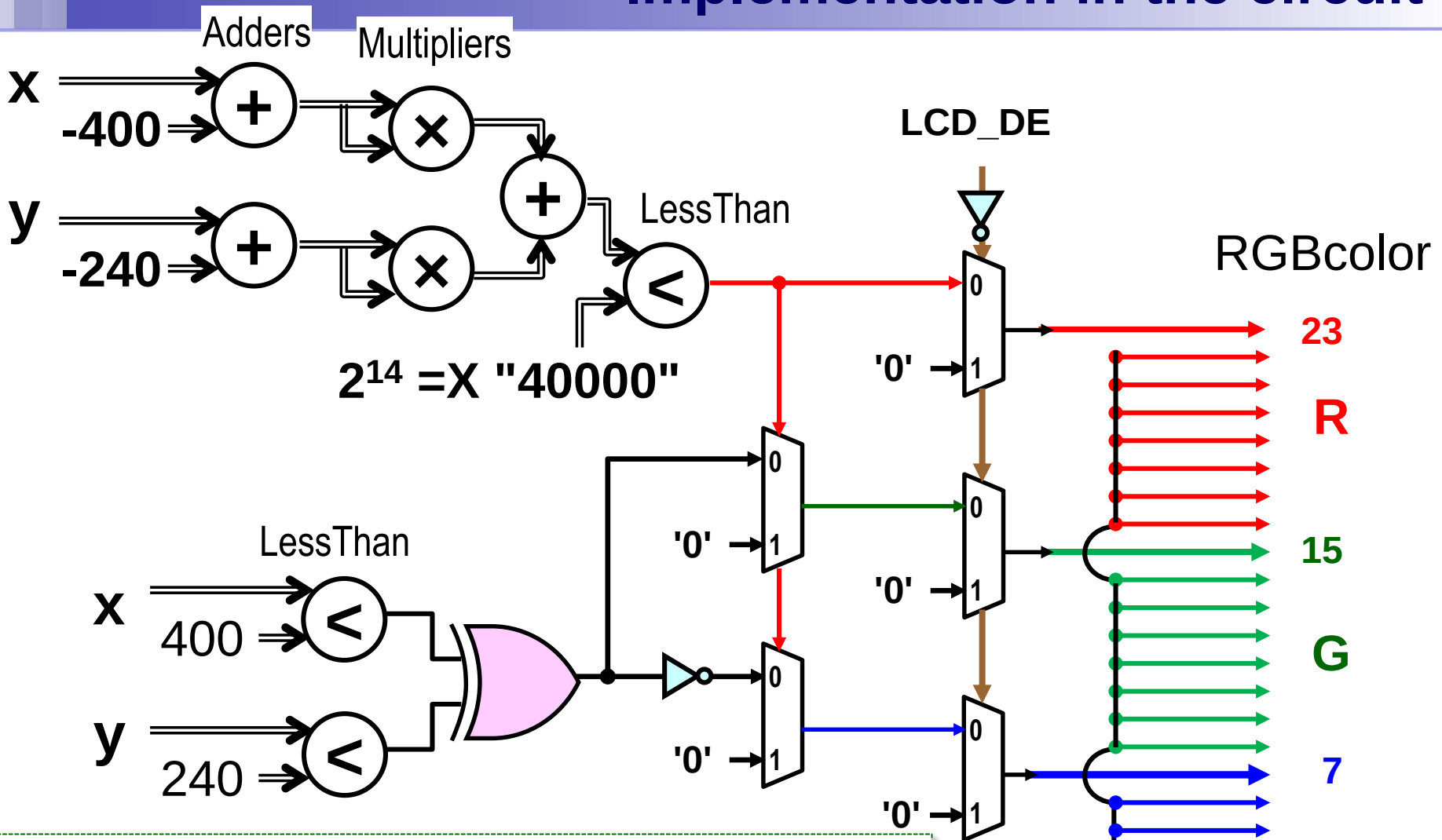
Adding Circle



Two Squares plus Circle

```
architecture concurrentImage of LCDlecture is
signal x,y : integer range 0 to 2**xy_t'LENGTH-1:=0;
constant LXC:integer := LCD_WIDTH/2; -- visible xcolumn center
constant LYC:integer := LCD_HEIGHT/2; -- visible yrow center

begin

  x<=to_integer(xcolumn); y<=to_integer(yrow);

RGBcolor <=  BLACK when LCD_DE='0' else

                RED when (x-LXC)**2+(y-LYC)**2<128**2 else

               GREEN when x<LXC xor y<LYC else

                NAVY;

end architecture;
```

Implementation in the circuit




```
RGBcolor <=   BLACK when LCD_DE='0' else
               RED when (x-LXC)**2+(y-LYC)**2<128**2 else
             GREEN when x<LXC xor y<LYC else
             NAVY;
```

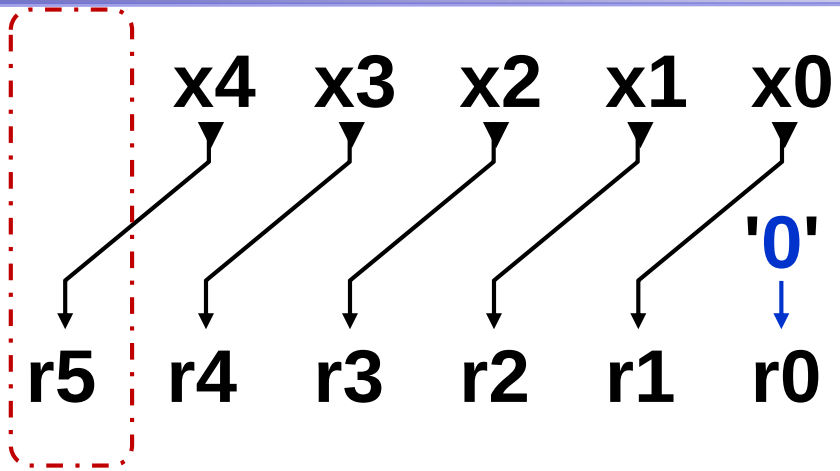
Arithmetic in Circuits

How is it done?

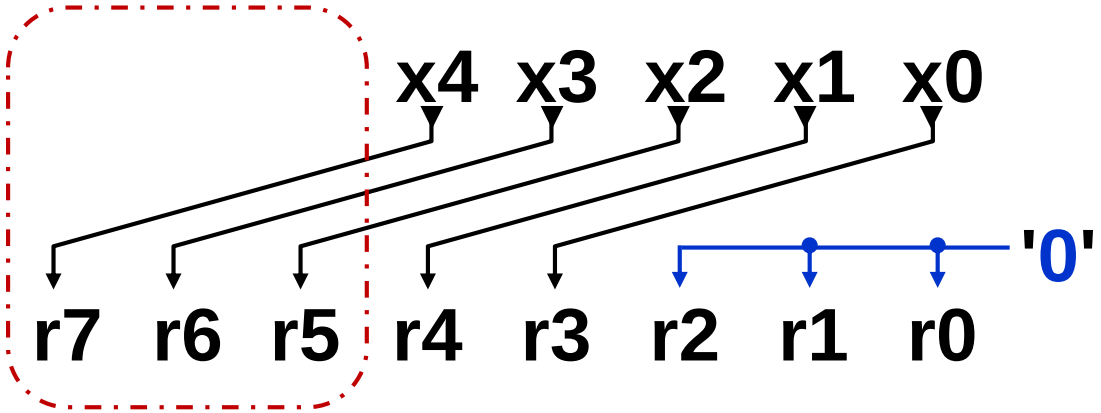
Logic Circuits textbook, chapter 6.

Multiplication by a Power of 2

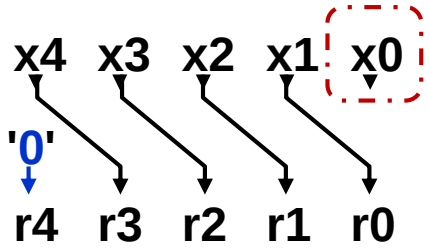
$x * 2^1$



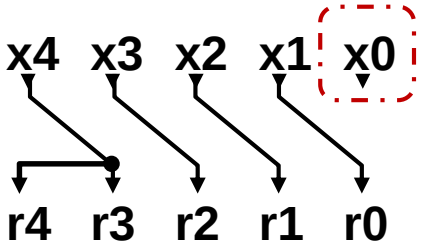
$x * 2^3$



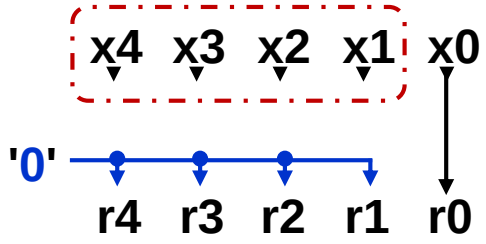
Division by Powers of 2 - Wires



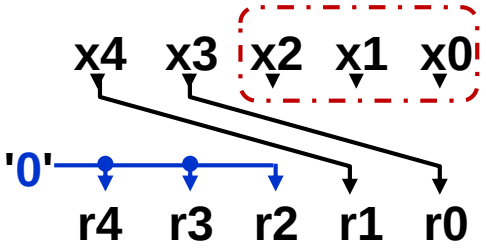
unsigned **(x) / 2^1**



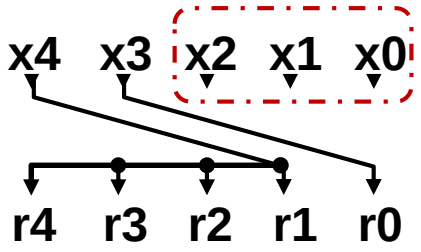
signed **(x) / 2^1**



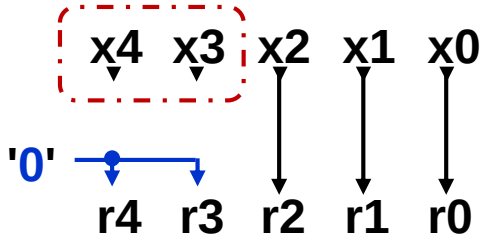
x mod 2^1



unsigned **(x) / 2^3**

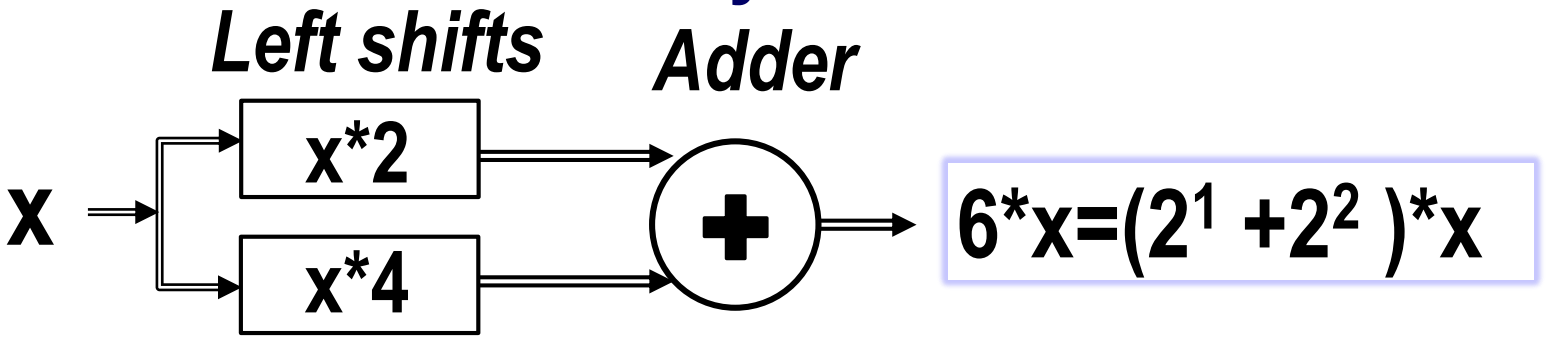


signed **(x) / 2^3**

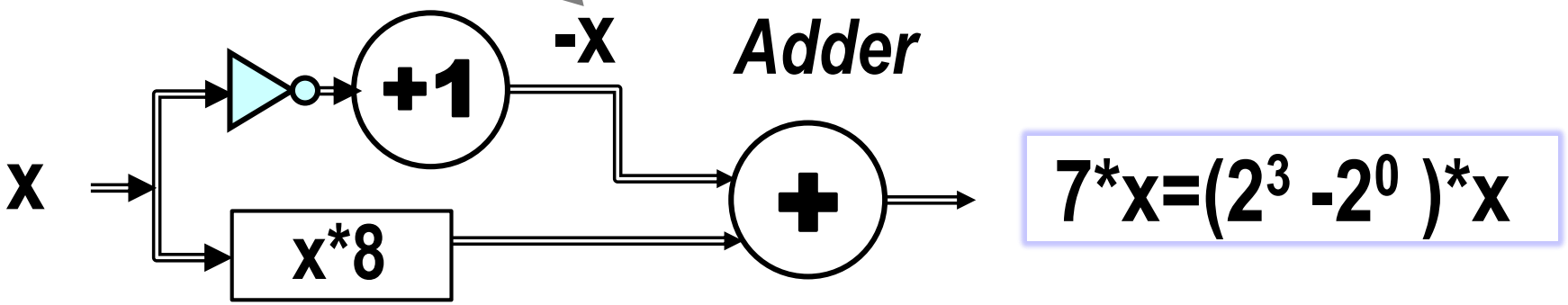


x mod 2^3

Multiplication by the Sum of Powers of 2

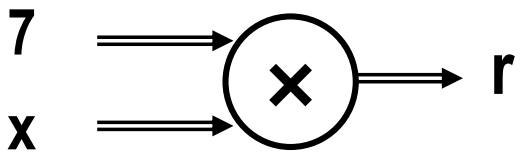


a binary complement of x



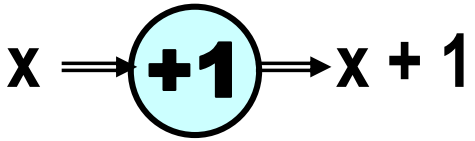
Quartus multiplies by 2^N -1 sometimes

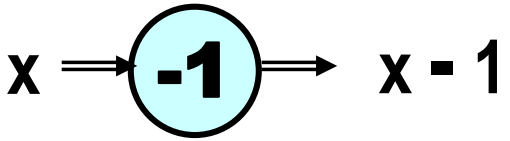
- *by using an adder...*
- *by a multiplier*



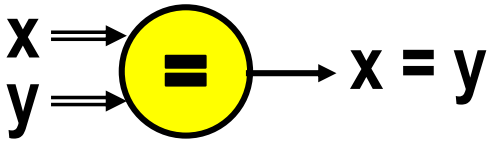
By minimizing the SoP or PoS, it is appropriate to design:

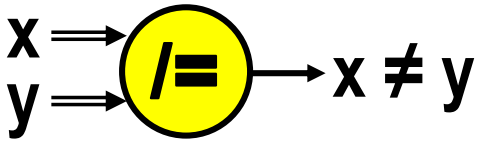
Adders +/-1



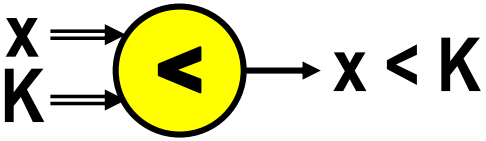


Comparators
equal and not equal





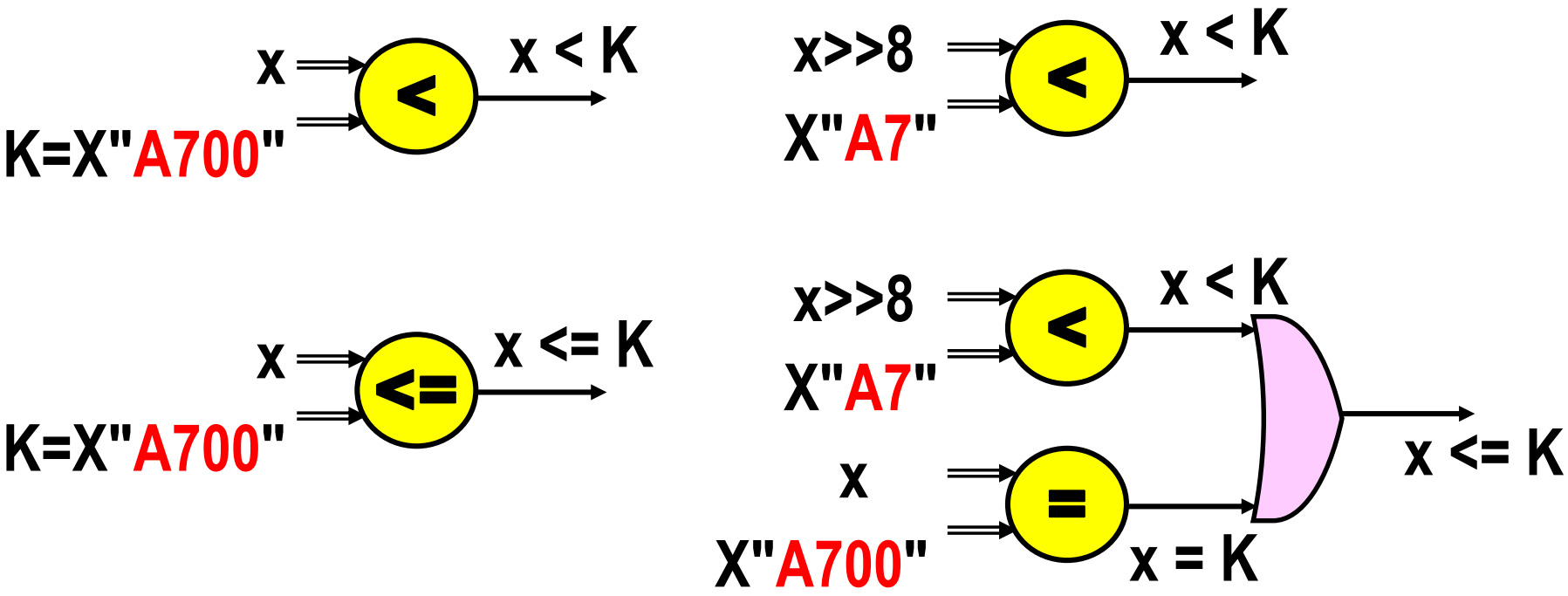
Comparators
with K constant



and other types <= = /= >=

≠ is **!=** in C, **~=** in Matlab, <> in Basic or Pascal, but **/=** in VHDL

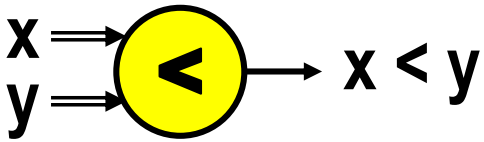
If a constant ***K | 2^N*** *, i.e., K is divisible by a N power of 2, where integer N>0, then K has* ***N lower bits = '0'****.*
In that case, ***x<K*** *and* ***x>=K*** *comparators are smaller because they can test only the upper non-zero bits.*



Operation with two general numbers x,y

SoP and PoS minimization give too complex results for the following circuits. Their decompositions are necessary:

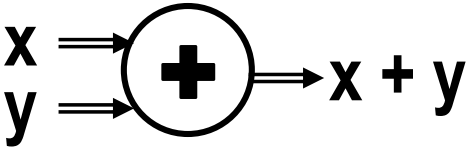
- Comparators

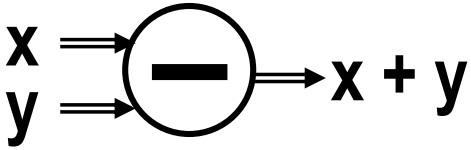


and comparisons

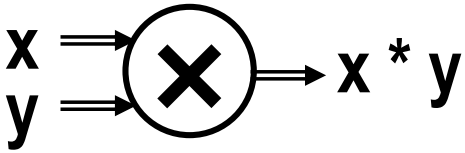
>= > <=

- Adders and subtractors

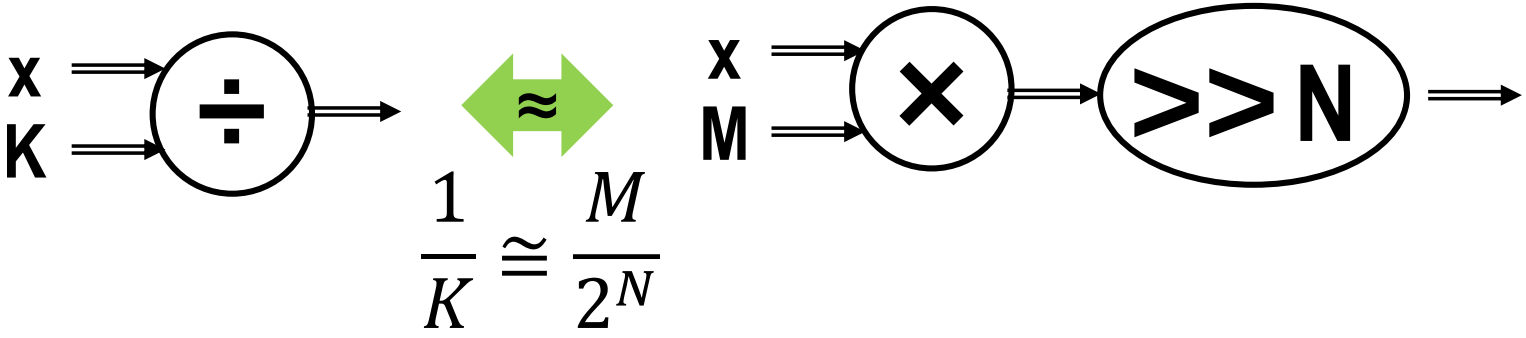




- Multipliers

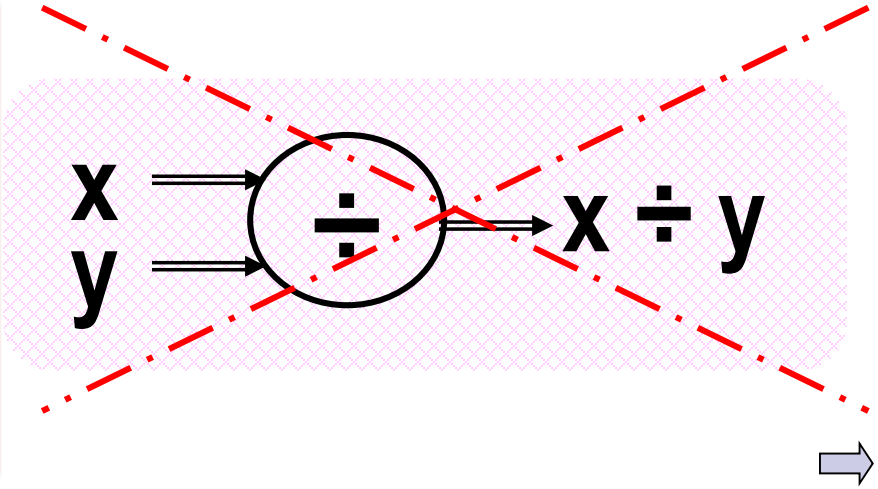


Divisions by the constants K≠2 N can be approximated

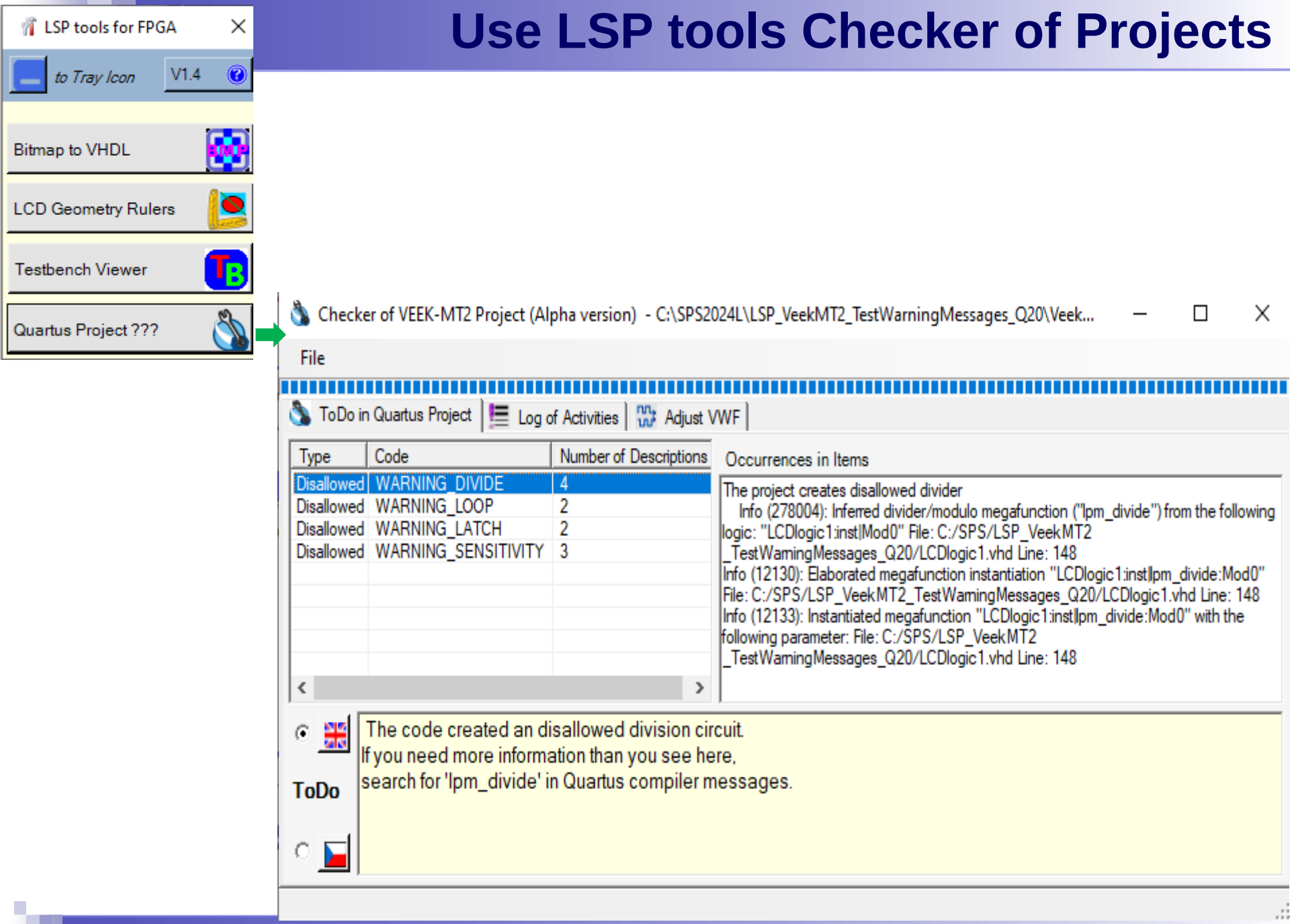


Note: In LCD, xcolumn and yrow coordinates can be divided by any number with the aid of counters, see Task_LCDbackground.pdf page 16.

Do not use
the direct divisions by
a general number or by constant ≠ 2^N
because they have extremely
complex circuit implementations
!!!



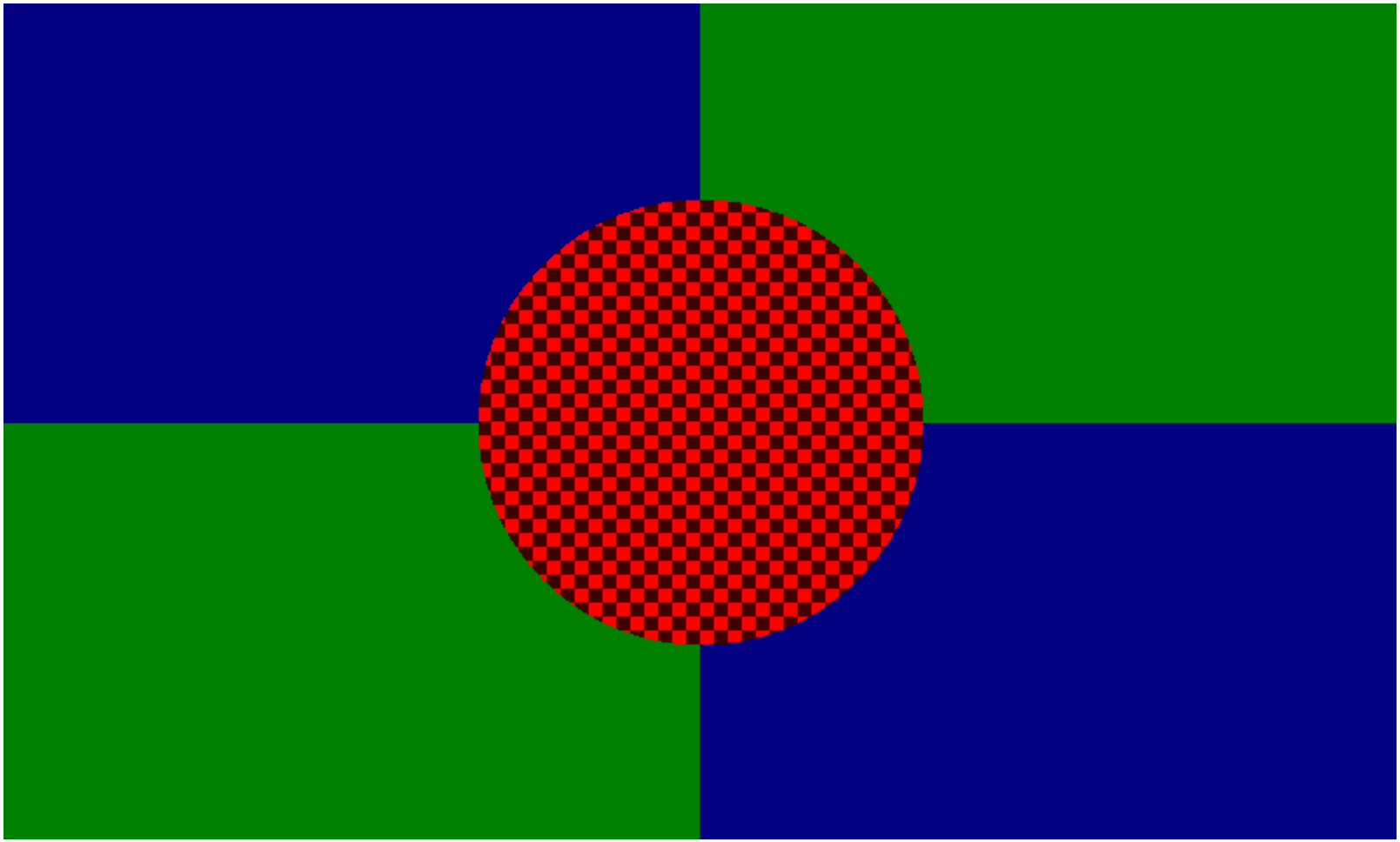
Use LSP tools Checker of Projects



Repeated patterns

by divisions and modulo

Circle with Chessboard



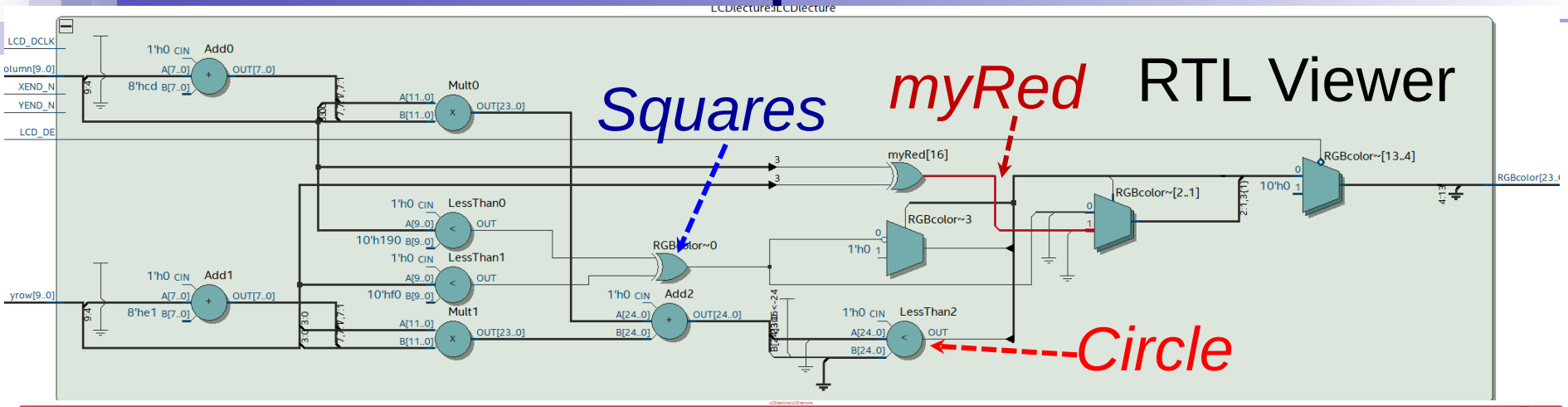
Full Image

```
architecture concurrentImage of LCDlecture is
signal x,y : integer range 0 to 2**xy_t'LENGTH-1:=0;
constant LXC:integer := LCD_WIDTH/2; -- visible xcolumn center
constant LYC:integer := LCD_HEIGHT/2; -- visible yrow center
constant BLOODRED:RGB_t:=x"660000"; -- web safe color
signal myRed:RGB_t:=BLACK;
begin
  x<=to_integer(xcolumn); y<=to_integer(yrow);
  myRed<= RED when (xcolumn(3) xor yrow(3)) else BLOODRED;
RGBcolor <=    BLACK when LCD_DE='0' else
                  myRED when (x-LXC)**2+(y-LYC)**2<128**2 else
                  GREEN when x<LXC xor y<LYC else
                  NAVY;
end architecture;
```

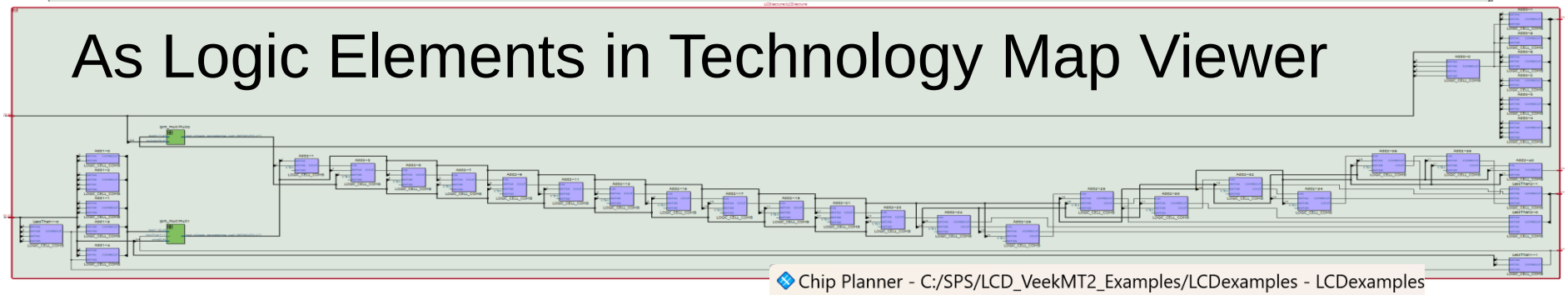
Alternative:

```
myRed<= RED when ((x/8) mod 2)=0 xor ((y/8) mod 2)=0 else BLOODRED;
```

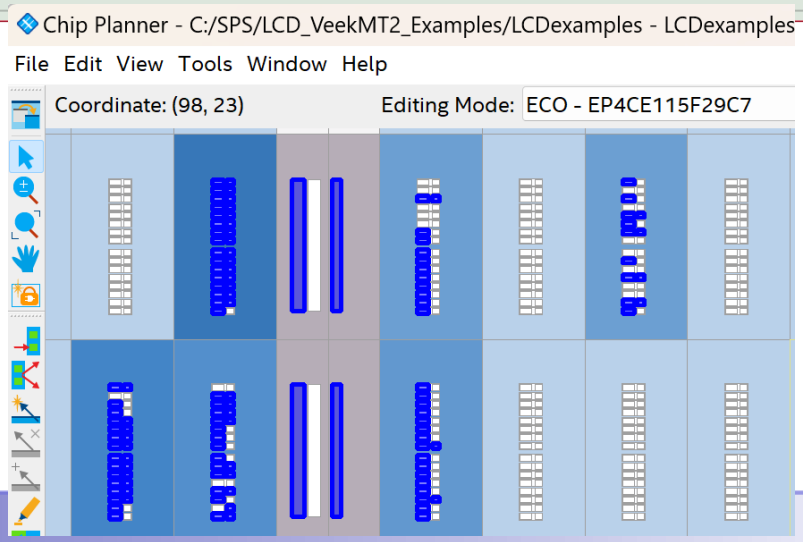
Real Implementation in the Circuit



As Logic Elements in Technology Map Viewer



In Chip Planner



Dataflow Limits

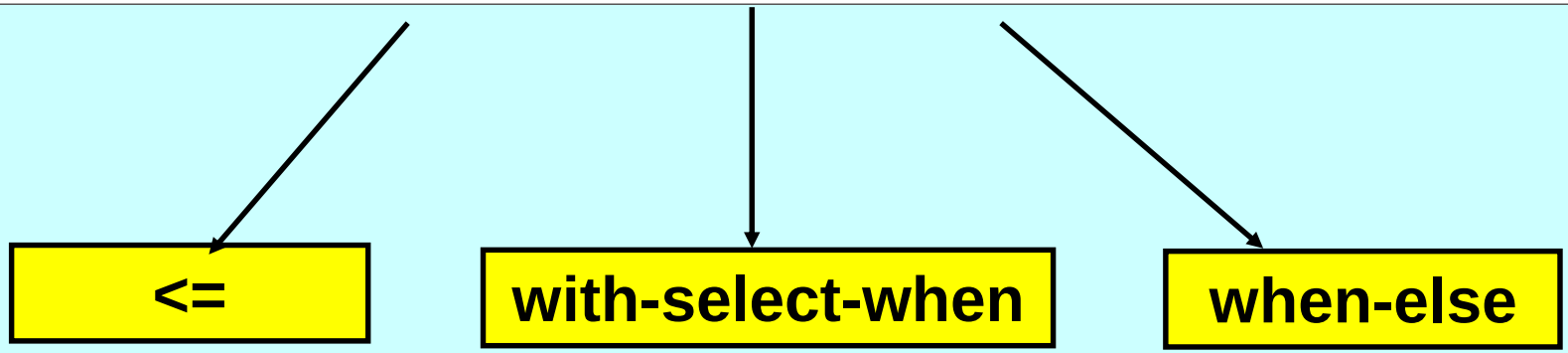
- Their conditions are growing in the case of nested rules.
  — The concurrent select with or when else statements cannot contain inserted statements, only expressions. So, we must prepare values in auxiliary signals.
- The sequential domain allows nested statements and also overwriting values of variables. The compiler transforms its code into concurrent statements by introducing auxiliary signals.

Sequential VHDL domain

is surrounded by keywords:

- **function ... end function;**
- **procedure ... end procedure;**
- **process ... end process;**
- It also includes the passive process that starts with the optional begin keyword inside an entity declaration to its end.

Three types of "concurrent" commands
we are familiar with them from the concurrent domain.



<=

with-select-when

when-else

Visit For

Concurrent assignments

Selective assignments

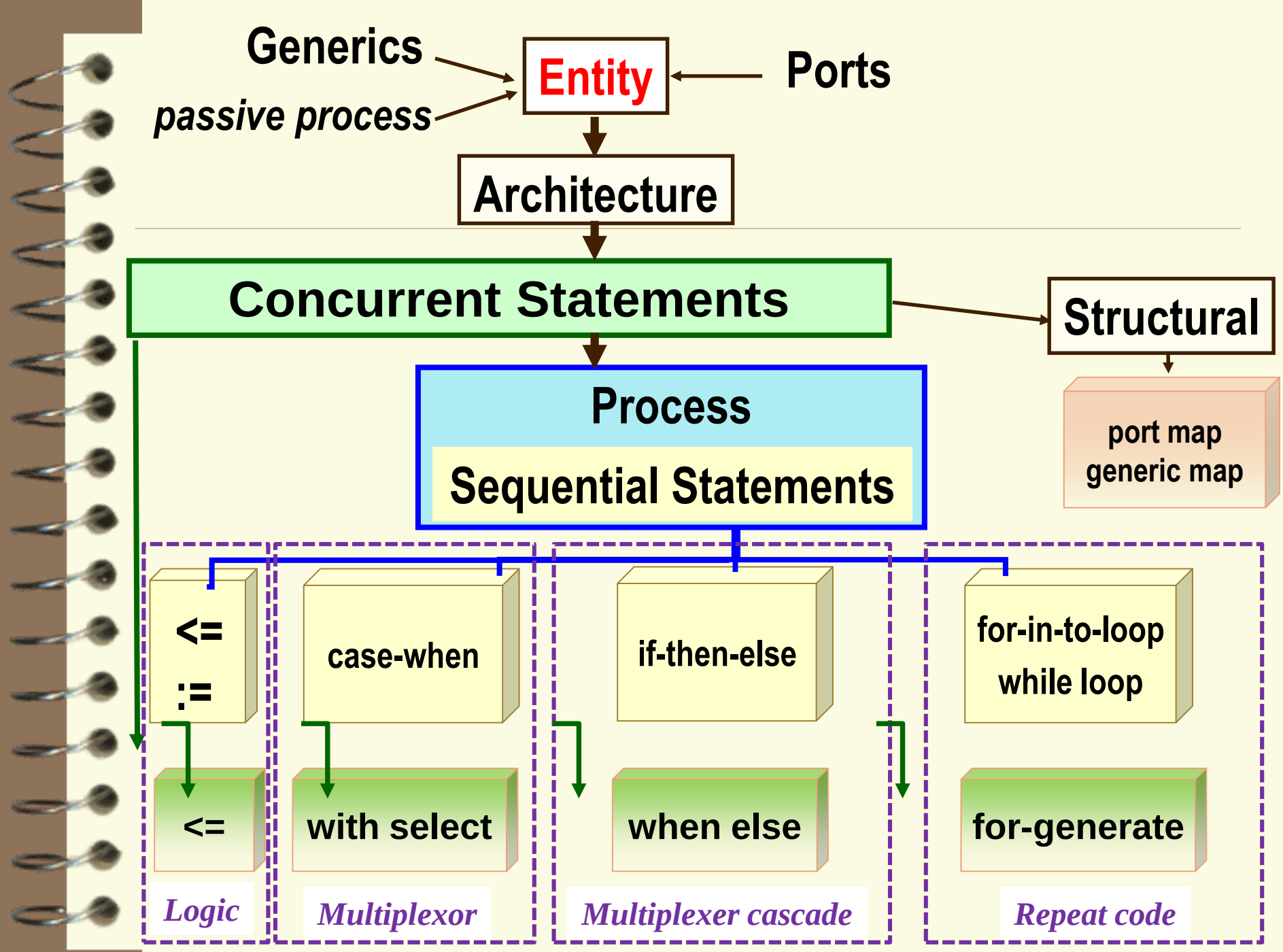
Conditional assignments

implemented using

logic circuit

multiplexers

multiplexer cascades 2:1


Generics
passive process
Entity
Ports
Architecture
Concurrent Statements
Structural
port map
generic map
Process
Sequential Statements
<=
:=
case-when
if-then-else
for-in-to-loop
while loop
<=
with select
when else
for-generate
Logic
Multiplexor
Multiplexer cascade
Repeat code

VHDL Process

We'll start with its application

Sequential domain: Variables

- In the sequential domain, we cannot define signals, only variables, but by their declarations are similar to signals:

  variable variable-name, variable-name , ... : **data-type**;

- New values are assigned to variables by **blocking** assignments:

  variable-name **:=** value-expression;

- Signals represent connections by wires and need **non-blocking** concurrent assignments:

  signal-name **<=** signal-expression;

  The reasons for this will be the topic of the following lecture.

- **In VHDL 2008**, we can also use "concurrent statements" when-else and with-select in the sequential domain.

  But the Quartus Lite versions do not allow them here.

Seq. domain needs variables

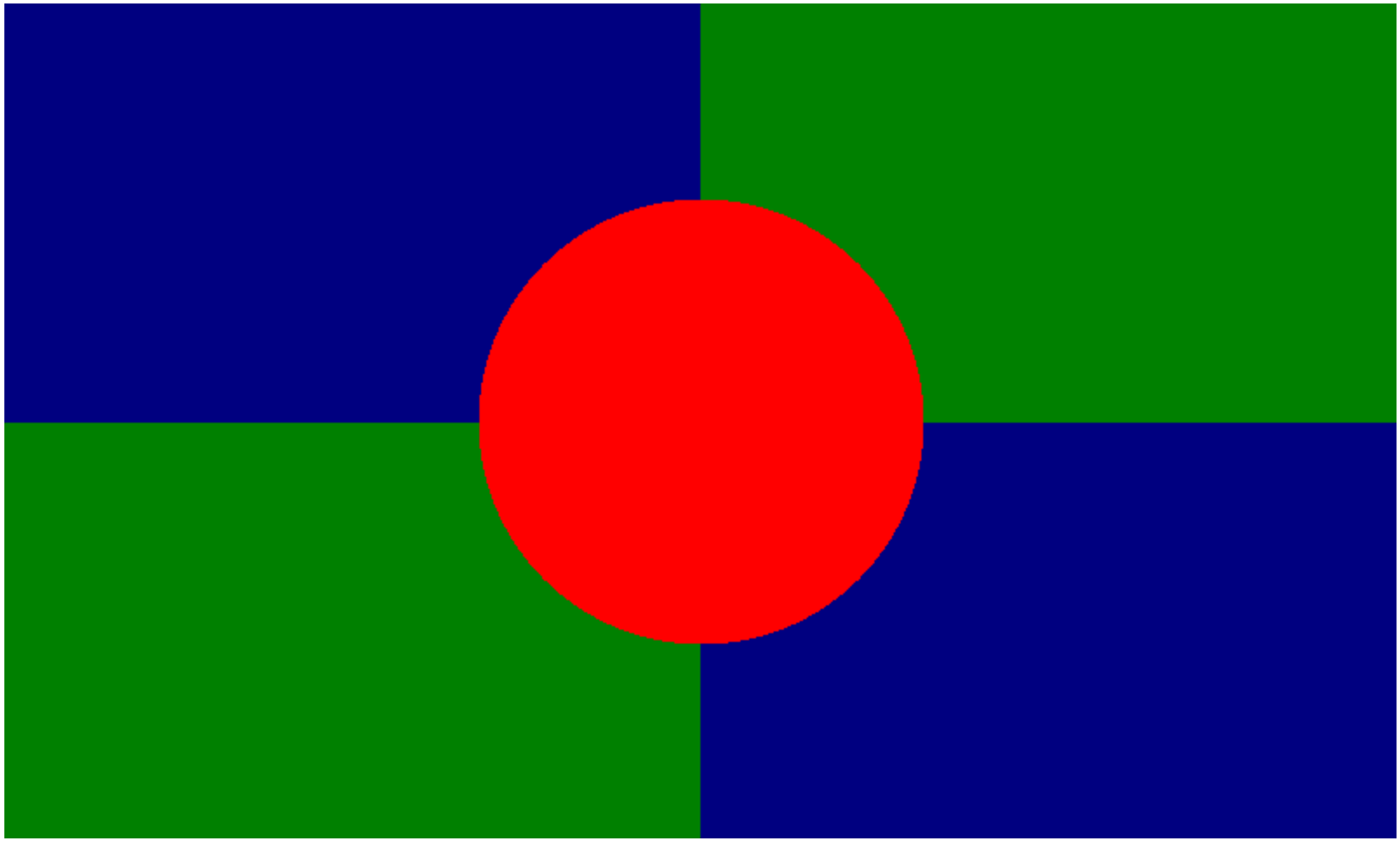
```
architecture sequantialImage of LCDlecture is
constant LXC:integer := LCD_WIDTH/2; -- visible xcolumn center
constant LYC:integer := LCD_HEIGHT/2; -- visible yrow center
constant BLOODRED:RGB_t:=x"660000"; -- web safe color
begin
 process(LCD_DE, xcolumn, yrow)
 variable rgb:  RGB_t:=BLACK;
 variable x,y : integer range 0 to 2**xy_t'LENGTH-1:=0;
 begin
-- Next slides
 end process;
end architecture;
```

Seq. domain allows overwriting

```
process(LCD_DE, xcolumn, yrow)
 variable rgb:RGB_t:=BLACK;
 variable x,y : integer range 0 to 2**xy_t'LENGTH-1:=0;
 begin
     x:=to_integer(xcolumn); y:=to_integer(yrow);
    rgb:=NAVY;
    if x<LXC xor y<LYC then
          rgb:=GREEN;   end if;
    if (x-LXC)**2+(y-LYC)**2<128**2 then
          rgb:=RED;       end if;
   if not LCD_DE then
          rgb:=BLACK;   end if;
    RGBcolor<=rgb;
 end process;
end architecture;
```

The VHDL compiler rewrites the code to its concurrent version and removes all forbidden overwriting of values.

The image remains the same



Process Format

[label:] process (*sensitivity list*)

declarations

begin

sequential statements

end process [label];

-- *Label is an optional parameter.*
However, we can see it in the simulation,
in which it serves as a landmark.

Process and its Sensitivity List

- A ***process is a VHDL construct that contains a set of actions to be executed sequentially.*** These actions are known as ***sequential statements. The process itself is a concurrent statement.*** It can be interpreted as a circuit part enclosed inside a black box whose behavior is described by the sequential statements. We may or may not be able to construct physical hardware that exhibits the desired behavior.
- The **sensitivity-list is a list of signals to which the process responds (i.e., is "sensitive** to"). The **declarations part consists of various declarations that are local to the process.**
- The list of sensitivities in the process are not formal parameters, but hints to the compiler as to what inputs will cause the outputs to change.
- Whereas the appearance of a VHDL process is like a function or procedure of a traditional programming language, the behavior of the process is very different. A VHDL process is not invoked (or called) by another routine. It acts like a circuit part, which is either active (known as ***activated) or inactive (known as suspended).***
- *In simulations****, a VHDL process is activated when*** a signal in the sensitivity list changes its value, like a circuit responding to an input signal. Once a process is activated, its statements will be executed sequentially until the end of the process. The process is then suspended until the next change of signal.

[VHDL 1993 Reference, IEEE]

Process Declarations

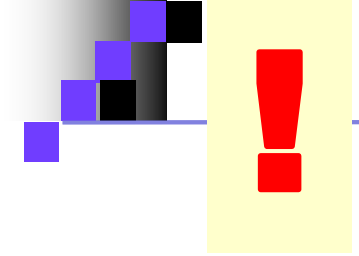
- **type**, **subtype**, **constant** and **variable**
- **function**, **procedure** - sequential codes of circuit parts for local use. They are not called in the circuit, but inserted by inline expansion technique.
- **use** keyword allows adding additional libraries locally available only here, if they are needed.

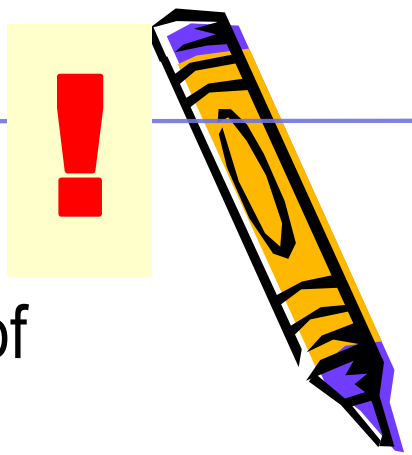
❖ No **signal** or **component** declarations are allowed here.

Variables

- A variable describes the abstract behavior of the system.
- We can declare variables in a process, function or procedure. We are not allowed to declare signals there, they belong only to the architecture.
- Variables are always local, i.e. they cannot be referenced from elsewhere than from the part that declared them.
- *Note: Shared variables were introduced in VHDL 93 for simulation purposes. Although they can be declared outside the process, they are difficult to implement in hardware. As a rule, they are circumvented by complex automatic conversions, which sometimes have unpredictable results.*
- Shared variables are strictly prohibited in the circuit synthesis portions of LSP projects. The rule is also applied by professional development companies.

Beware of signals in sequential domains



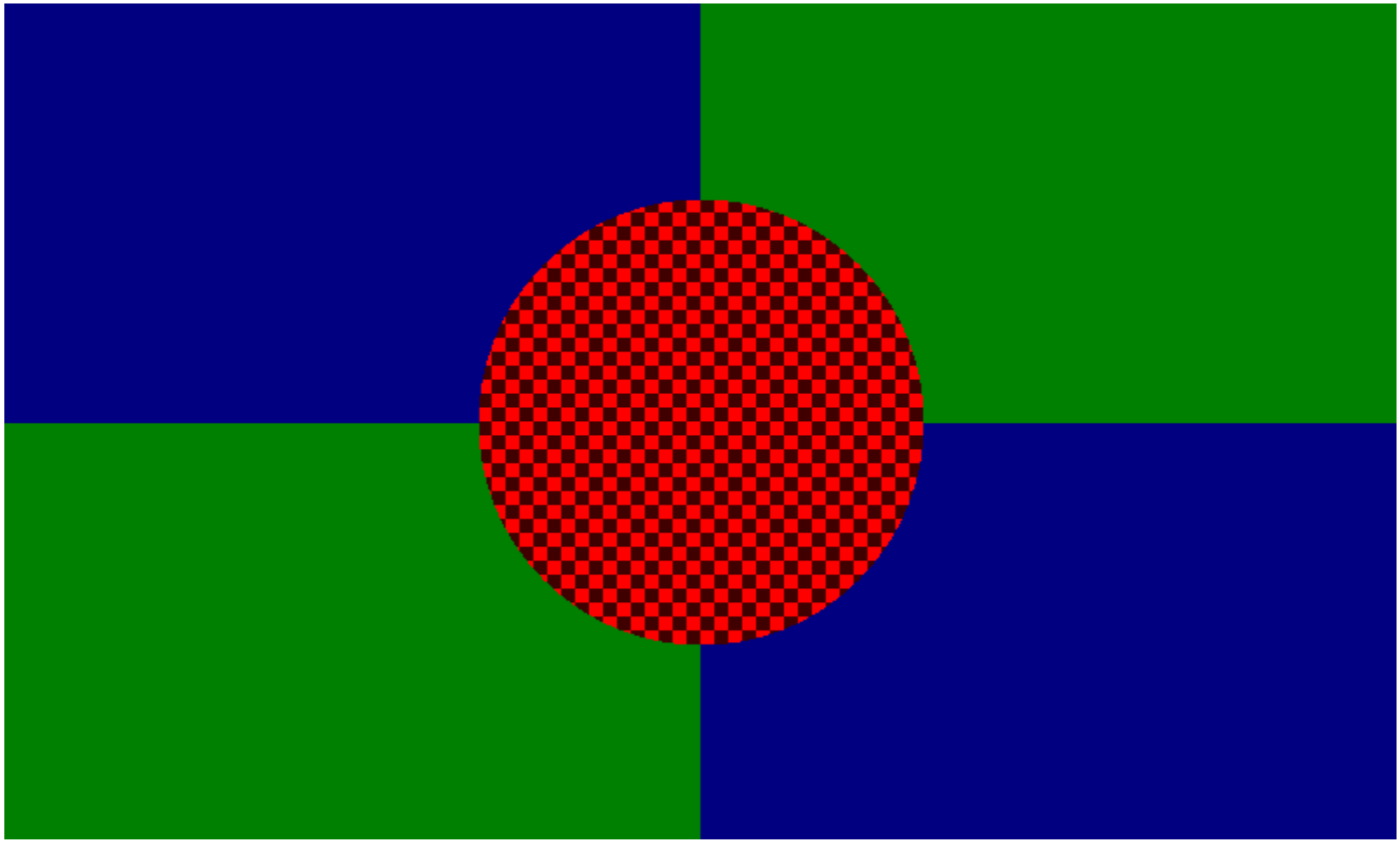


- The signal types always emulate the inertial delay of actual wires by postponing updates of their values.
- If we improperly use a signal, we have unwanted side effects. We will discuss this topic in our lectures.
- In processes, we should always prefer **variables** that behave here similarly to classic programming, e.g., C language.
- We freely read values of signals but **assign each signal at most only once**, usually at the end of a process.

Task_LCDbackground - pg. 10

If-then-else can be nested

```
architecture sequantialImage of LCDlecture is
constant LXC:integer := LCD_WIDTH/2; -- visible xcolumn center
constant LYC:integer := LCD_HEIGHT/2; -- visible yrow center
constant BLOODRED:RGB_t:=x"660000"; -- web safe color
begin
 process(LCD_DE, xcolumn, yrow)
 variable rgb:RGB_t:=BLACK;
 variable x,y : integer range 0 to 2**xy_t'LENGTH-1:=0;
 begin
   x:=to_integer(xcolumn); y:=to_integer(yrow);
    rgb:=NAVY;
    if x<LXC xor y<LYC then rgb:=GREEN; end if;
    if (x-LXC)**2+(y-LYC)**2<128**2 then
       if (xcolumn(3) xor yrow(3)) then rgb:=RED; else rgb:=BLOODRED; end if;
     end if;
    if not LCD_DE then rgb:=BLACK; end if;
    RGBcolor<=rgb;
 end process;
end architecture;
```



IF-ELSIF Style

```
process(LCD_DE, xcolumn, yrow)
 variable rgb:RGB_t:=BLACK;
 variable x,y : integer range 0 to 2**xy_t'LENGTH-1:=0;
 begin
  x:=to_integer(xcolumn); y:=to_integer(yrow);
  if not LCD_DE then rgb:=BLACK;
  elsif (x-LXC)**2+(y-LYC)**2<128**2 then
      if (xcolumn(3) xor yrow(3)) then rgb:=RED; else rgb:=BLOODRED; end if;
  elsif x<LXC xor y<LYC then rgb:=GREEN;
  else rgb:=NAVY;
  end if;
   RGBcolor<=rgb;
 end process;
```

IF-ELSIF versus more IFs

```
process(LCD_DE, xcolumn, yrow)
 variable rgb:RGB_t:=BLACK;
 variable x,y : integer
          range 0 to 2**xy_t'LENGTH-1:=0;
 begin
  x:=to_integer(xcolumn); y:=to_integer(yrow);
  if not LCD_DE then rgb:=BLACK;
  elsif (x-LXC)**2+(y-LYC)**2<128**2 then
     if (xcolumn(3) xor yrow(3)) then
       rgb:=RED; else rgb:=BLOODRED;
     end if;
  elsif x<LXC xor y<LYC then rgb:=GREEN;
  else rgb:=NAVY;
  end if;
   RGBcolor<=rgb;
end process;
```

```
process(LCD_DE, xcolumn, yrow)
 variable rgb:RGB_t:=BLACK;
 variable x,y : integer
          range 0 to 2**xy_t'LENGTH-1:=0;
 begin
  x:=to_integer(xcolumn); y:=to_integer(yrow);
  rgb:=NAVY;
  if x<LXC xor y<LYC then rgb:=GREEN; end if;
  if (x-LXC)**2+(y-LYC)**2<128**2 then
    if (xcolumn(3) xor yrow(3)) then
      rgb:=RED; else rgb:=BLOODRED;
    end if;
  end if;
  if not LCD_DE then rgb:=BLACK; end if;
  RGBcolor<=rgb;
end process;
```

Note: if then elsif vs more if then

- Circuits with more if statements do not have more complex implementations.
- The if then elsif else chains specify that the RTL meta-schemas of the first compile-pass contain cascade of two-input multiplexers starting with the **highest priority** condition.
- More if then statements begin with the **lowest priority** condition, and the compiler also transforms them to the chain of two-input multiplexors, usually in the first pass, otherwise in the next one.
- The both methods lead to the optimal circuit. If someone is more familiar with multiple IFs, let them create that way.