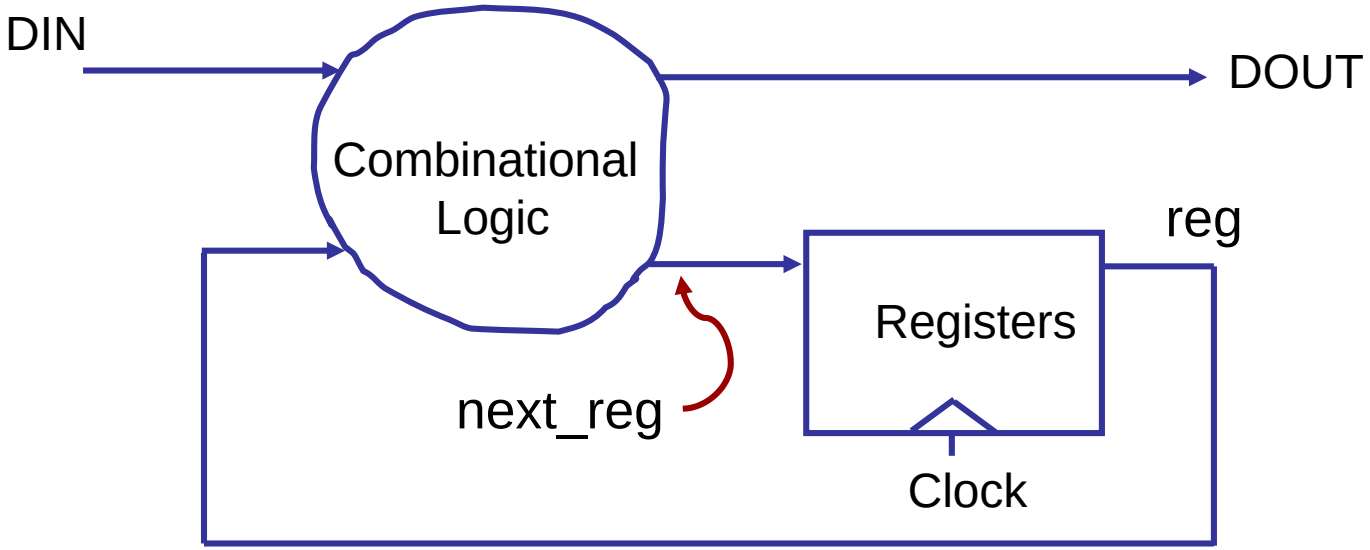
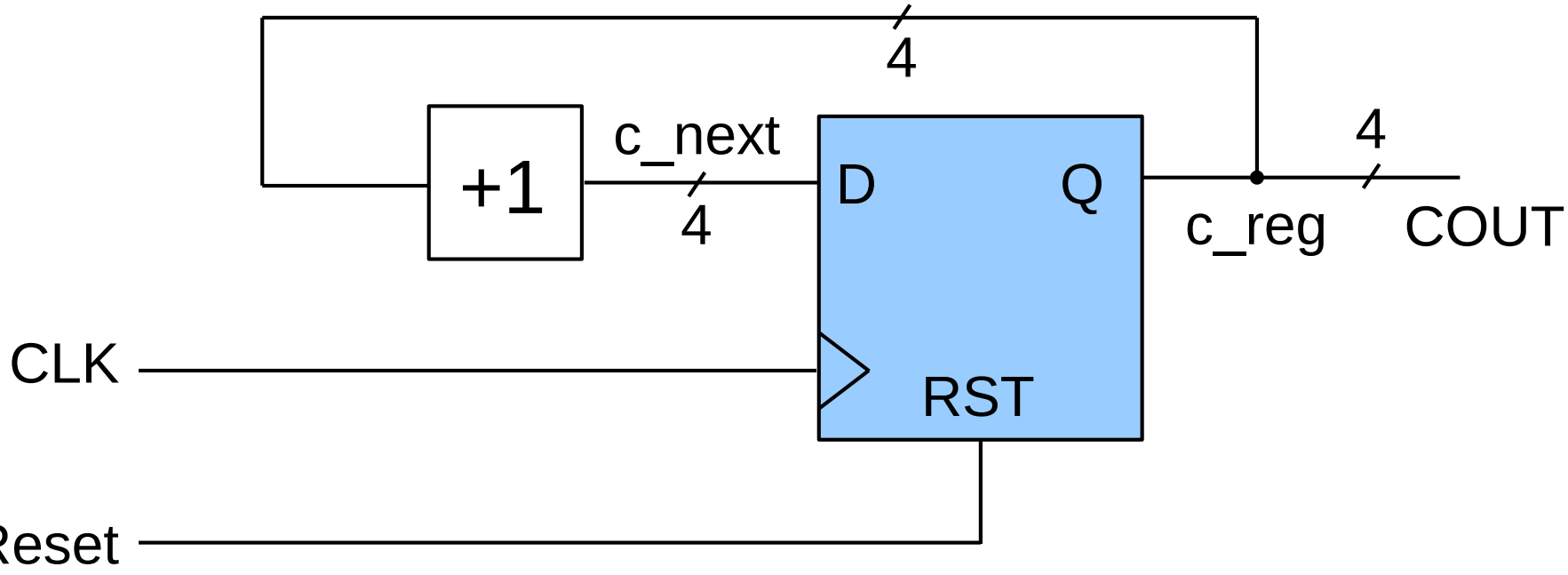
Modeling at the RT Level

- A register transfer level (RTL) design consists of a set of registers connected by combinational logic.



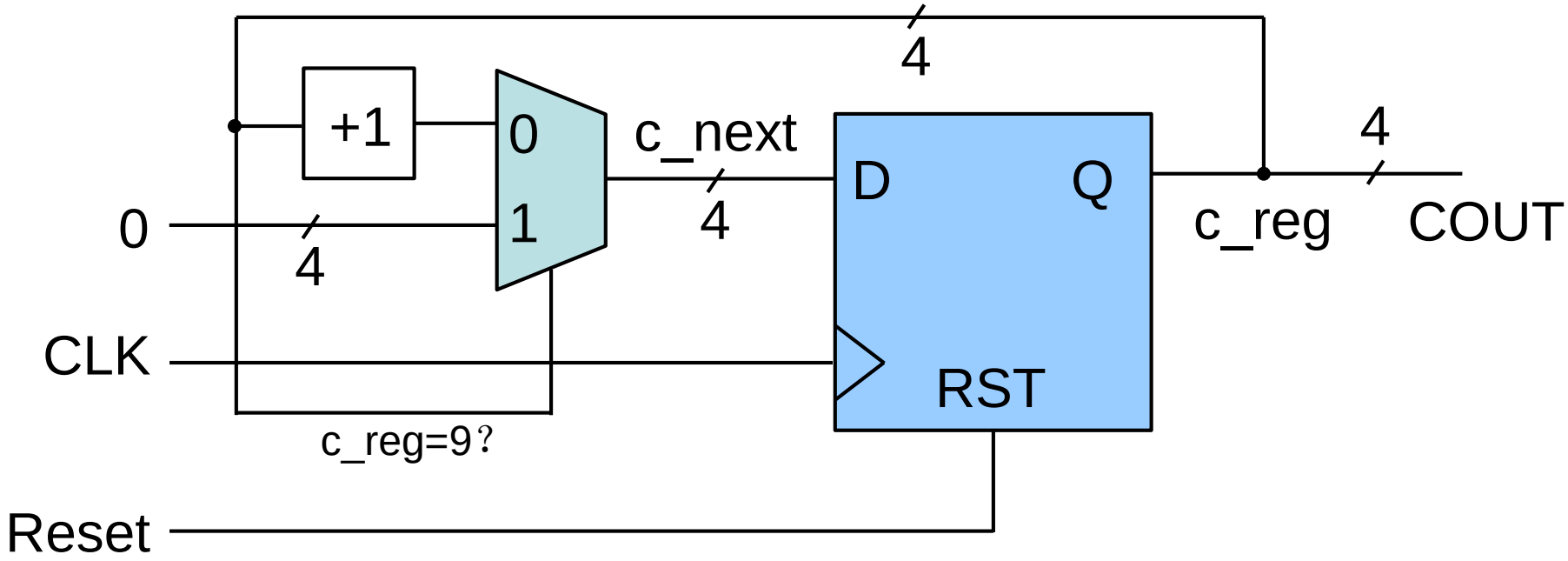
Free Running 4-bit counter




```
-- Synchronous segment
…
c_reg <= “0000”;  -- reset
…
c_reg <= c_next;  -- clock edge
```

```
-- Combinational segment
…
c_next <= c_reg+1;
```

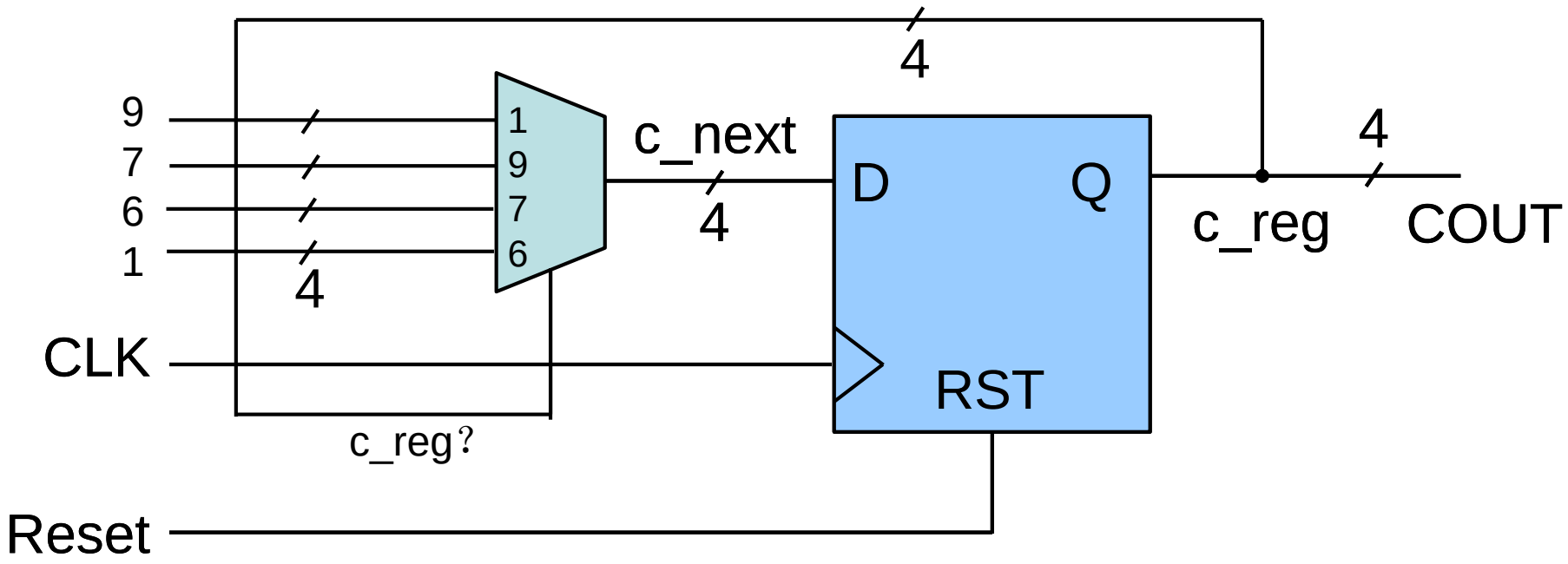
4-bit modulo 10 counter




```
-- Synchronous segment
…
c_reg <= “0000”;  -- reset
…
c_reg <= c_next;  -- clock edge
```

```
-- Combinational segment
…
c_next <= “0000” when c_reg=9 else
          c_reg+1;
```

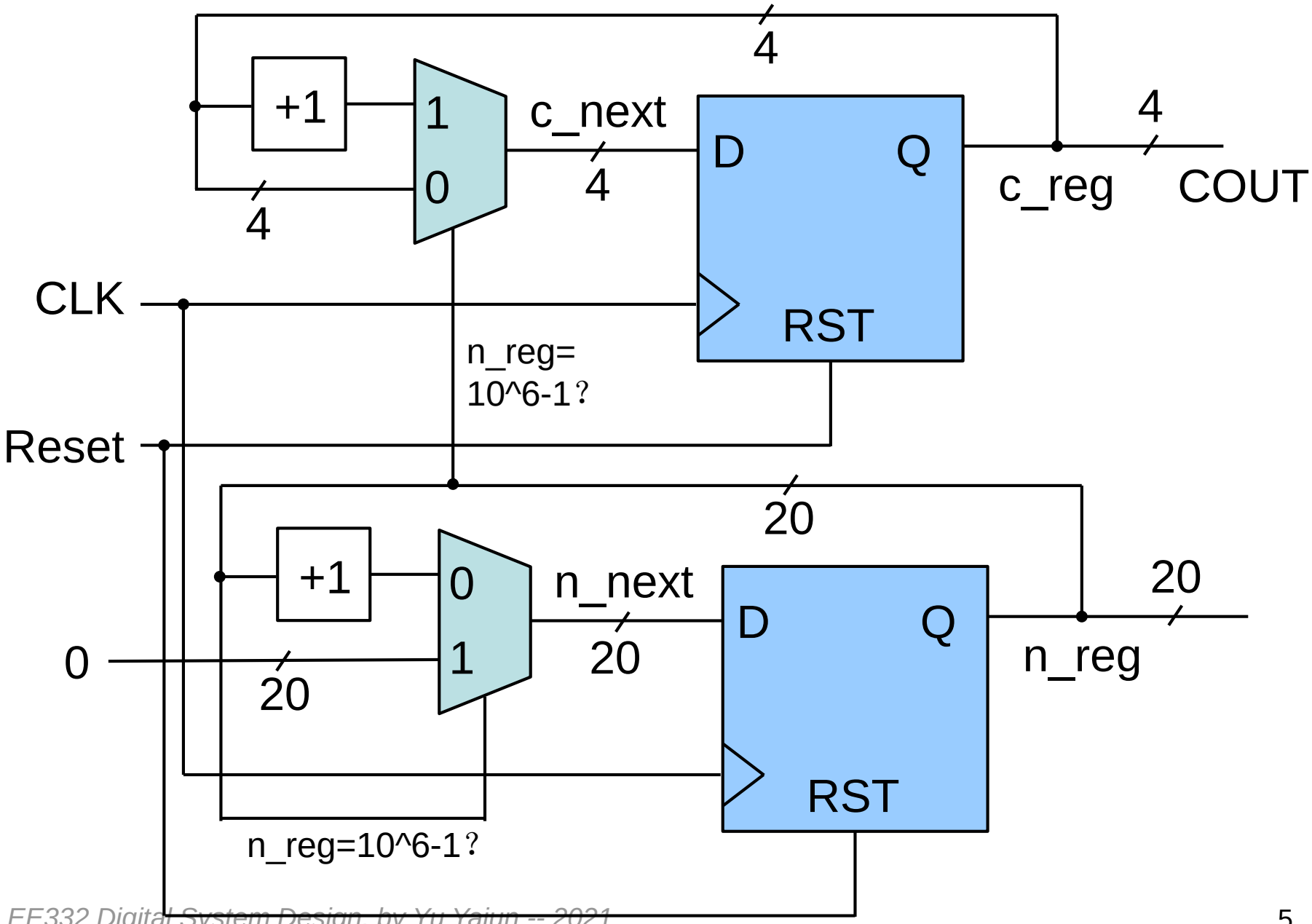
Count the sequence of 1, 9, 7, 6 and repeat




```
-- Synchronous segment
…
c_reg <= “0001”;  -- reset
…
c_reg <= c_next;  -- clock edge
```

```
-- Combinational segment
…
c_next <= “1001” when c_reg=1 else
          “0111” when c_reg=9 else
          “0110” when c_reg=7 else
          “0001”;
```

4-bit counter, increase 1 for every 10^6 clock cycles



4-bit counter, increase 1 for every 10^6 clock cycles

```
-- Synchronous segment
…
c_reg <= “0000”;
n_reg <= “0000”; -- reset
…
c_reg <= c_next;
n_reg <= n_next; -- clock edge

-- Combinational segment
…
c_next <= c_reg + 1 when n_reg = 99999 else
          c_reg;
n_next <= “00….00” when n_reg = 99999 else
          n_reg + 1;
```

Setting

The duty cycle of a square wave is defined as the percentage of the on interval (i.e., logic 1) in a period. A PWM (pulse width modulation) circuit can generate an output with variable duty cycle. For a PWM with 4-bit resolution, the period of the square wave is 16 clock cycles. A 4-bit control signal, w, specifies the duty cycle. The w signal is interpreted as an unsigned integer and the duty cycle is w/16.

Design a completely synchronous programmable square-wave generator circuit by doing the following two steps.

1) Sketch the block diagram of the programmable square-wave generator circuit using components of registers, adders, multiplexers and other necessary functional blocks.

2) Based on the block diagram, develop the VHDL code of the circuit by using the two-segment coding style.

Open Question is only supported on Version 2.0 or newer.

Answer

Sequential Add-and-Shift Multiplier

1. Multiply the digits of the multiplier (b4, b3, b2, b1 and b0) by the multiplicand A = (a4, a3, a2, a1, a0) one at a time to obtain b4*A, b3*A, b2*A, b1*A and b0*A.

$$bi * A = (a4 \cdot bi, a3 \cdot bi, a2 \cdot bi, a1 \cdot bi, a0 \cdot bi)$$

2. Shift bi * A to left by i position.
3. Add the shifted bi * A terms to obtain the final product.

						a_4	a_3	a_2	a_1	a_0
x						b_4	b_3	b_2	b_1	b_0
						a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
					a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
				a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+			a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
		a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

```
n = 0;
p = 0;
while (n!=8) {
    if (b_in(n) = 1)
    then {
        p = p + (a_in << n);
    }
    n = n + 1;
}
r_out = p
```

```
a = a_in;
b = b_in;
n = 8;
p = 0;
while (n!=0) {
    if (b(0) = 1) then {
        p = p + a;
    }
    a = a << 1;
    b = b >> 1;
    n = n - 1;
}
r_out = p
```

```
a = a_in;
b = b_in;
n = 8;
p = 0;
op: if b(0) = 1 then {
    p = p + a;
}
a = a << 1;
b = b >> 1;
n = n - 1;
if (n != 0) then{
    goto op;
}
r_out = p
```

					a_4 b_4	a_3 b_3	a_2 b_2	a_1 b_1	a_0 b_0
x										
					a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
				a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
			a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+		a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

```
        a = a_in;
        b = b_in;
        n = 8;
        p = 0;
op:     if b(0) = 1 then {
            p = p + a;
        }
        a = a << 1;
        b = b >> 1;
        n = n - 1;
        if (n != 0) then{
            goto op;
        }
        r_out = p
```

- Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm is as follows:

Example 1, 2018

- If x is the count of bits of the multiplicand, and y is the count of bits of the multiplier:

  Step 1: Draw a table with three rows and $x + y + 1$ columns. The width of each column is 1 bit. Label the rows as A (add), S (subtract), and P (product), respectively.

  Step 2: Fill the first x bits of each row with the followings using two's complement notation:

  - A: the multiplicand
  - S: the negative of the multiplicand (in 2's complement format)
  - P: zeroes

Step 3: Fill the next *y* bits of each row as follows:

- *A*: zeroes
- *S*: zeroes
- *P*: the multiplier

Step 4: Fill the last bit of each row with a zero.

Step 5: Repeat the following two steps (Step 5.1 and Step 5.2) *y* times:

Step 5.1: If the last two bits in the product are

00 or 11: do nothing.

01: $P = P + A$. Ignore any overflow.

10: $P = P + S$. Ignore any overflow.

Step 5.2: Arithmetically shift the product right by one position.

Step 6: Drop the last bit (the least significant bit) from the product for the final result.

An example is to find $3 \times (-4)$, with $x = 4$ and $y = 4$. We have:

A = 0011 0000 0

S = 1101 0000 0

P = 0000 1100 0

Perform Step 5.1 and Step 5.2 four times:

P = 0000 1100 0. The last two bits are 00.

P = 0000 0110 0. A right shift.

P = 0000 0110 0. The last two bits are 00.

P = 0000 0011 0. A right shift.

P = 0000 0011 0. The last two bits are 10.

P = 1101 0011 0. P = P + S.

P = 1110 1001 1. A right shift.

P = 1110 1001 1. The last two bits are 11.

P = 1111 0100 1. A right shift.

Drop the last bit. The product is 1111 0100, which is –12.

Design using FSM with Datapath

Example 2, 2019

- Let `y_in` and `d_in` are two non-negative numbers. Repetitive-subtraction division is an algorithm to implement division operation (`y_in/d_in`), where `y_in` and `d_in` are the dividend and divisor, respectively. The algorithm obtains the quotient (`q_out`) and the remainder (`r_out`) by subtracting `d_in` from `y_in` repeatedly until the remainder of `y_in` is smaller than `d_in`. The reminder of `y_in` is the reminder of the division and the times of the subtraction is the quotient of the division.
- Assume that all the input and output signals are *M*-bit wide `std_logic_vector` and interpreted as unsigned integers.
- When `d_in=0`, both `q_out` and `r_out` return *M*-bit '1'.

Setting

```
if (d = 0) then {
        y0 = “11…11”; n = “11…11”; go to stop;
} -- set both q and r M-bit ‘1’ when the divisor d is 0.
else{
        y0 = y_in;
        n = 0;
comp: if (y0 < d_in) then go to stop;
        else{
sub:        y0 = y0 – d_in;
            n = n + 1;
            go to comp;
         }
}
stop: q = n;   r = y0;
```

Open Question is only supported on Version 2.0 or newer.

Answer

- A re-triggerable one-shot pulse generator is a circuit that generates a single fixed-width pulse upon activation of a trigger signal. We assume that the width of the pulse is five clock cycles. The detailed specifications are listed below.

Example 3, 2020

a) There are two input signals, `go` and `stop`, and one output signal `pulse_out`.

b) The `go` signal is the trigger signal that is usually asserted for only one clock cycle. During normal operation, assertion of the `go` signal activates the pulse signal for five clock cycles.

c) For each time that the `go` signal is asserted again during this interval, a new timing cycle is started, i.e., the count of the 5 clock cycles restarts.

d) If the `stop` signal is asserted during this interval, the `pulse_out` signal will be cut short and return to '0'.

Design the re-triggerable one-shot pulse generator by using finite state machine with datapath. The design can be completed by using two data registers: one register, named as `pulse`, is to hold the status whether the output is during the interval of the generated pulse, and the other register, named as `count`, is to hold the counts of the pulse width.

Let gcd(x,y) represent a function of fining the GCD (greatest common divisor) of two positive numbers x and y. The function gcd(x,y) has following properties, which are easy to verify:

Example 4, 2021

$$\gcd(x,y) = \begin{cases} x & if\ x = y \\ 2\gcd\left(\frac{x}{2},\frac{y}{2}\right) & if\ x \neq y\ and\ x,y\ even \\ \gcd\left(x,\frac{y}{2}\right) & if\ x \neq y\ and\ x\ odd\ y\ even \\ \gcd\left(\frac{x}{2},y\right) & if\ x \neq y\ and\ x\ even\ y\ odd \\ \gcd\left(\frac{x-y}{2},y\right) & if\ x > y\ and\ x,y\ odd \\ \gcd\left(x,\frac{y-x}{2}\right) & if\ y > x\ and\ x,y\ odd \end{cases}$$

Based on these properties, the pseudo code of a binary GCD algorithm, which finds the GCD *out* of two positive integer numbers *xi* and *yi* in the range of [1 1023], is given as follows. (Note that "<< k" in the code stands for "shift to left by *k* bits", ">> k" stands for "shift to right by *k* bits", and "rem(a,b)" stands for "the remainder of *a* divided by *b*".)

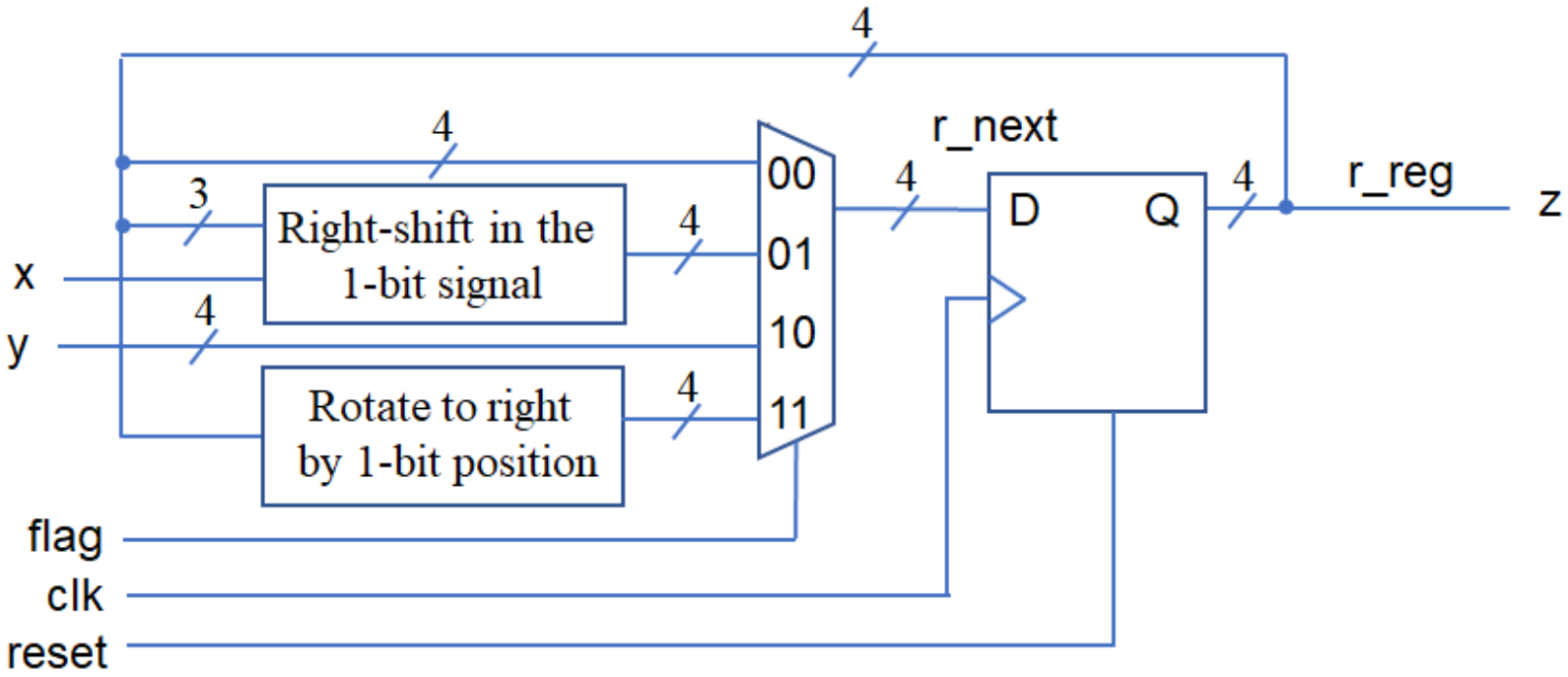
```
start:   x = xi;  y = yi;  r = 0;
equal:   if x = y { out = x << r; go to start; }
even:    if rem(x, 2) = 0 and rem(y, 2) = 0 {
             x = x >> 1; y = y >>1; r = r + 1; go to even; }
         if rem(x, 2) = 0 { x = x >> 1; go to even; }
         if rem(y, 2) = 0 { y = y >> 1; go to even; }
         -- both x, y are odd if the following step is reached
comp:    if (x > y) { x = (x – y) >> 1; go to equal;}
          if (x < y) { y = (y – x) >> 1; go to equal;}
         go to equal;
```

(1)Develop the algorithm state machine (ASM) chart for the algorithm.
(2)State how many registers are required.
(3)List the RT operation of register *x* for each state.
(4)Plot the block diagram of register *x*.

Example 5, 2022

- A rotation circuit, producing a 4-bit output `z`, has four operation modes controlled by a 2-bit `flag` signal:
  - When `flag` = "00", the circuit keeps the output unchanged.
  - When `flag` = "01", the circuit right-shifts in a 1-bit signal `x` into the register at each clock cycle.
  - When `flag` = "10", the circuit loads a 4-bit signal `y` into the register.
  - When `flag` = "11", the circuit rotates the signal in the register to the right by 1-bit position every clock cycle.

- The conceptual diagram of the circuit is shown in the following figure. Develop the VHDL program to describe the circuit using a two-segment coding style.



Example 6, 2022

- A rotation circuit takes a 4-bit signal `x` as an input to rotate, and the rotation direction and the number of bits to rotate is determined by another 3-bit input signal `y`. When `y`(2) is 1, `x` is rotated to left by `y`(1 downto 0)-bit positions, while `y`(2) is 0, `x` is rotated to right by `y`(1 downto 0)-bit positions. Design a finite statement machine with data path (FSMD) to realize the rotation. A `start` signal is used to initiate the rotation, and a `ready` signal is asserted when the required rotation is completed. Assume that only 2 shifters, of which one can shift the signal to right and the other can shift the signal to the left by 1-bit position are available.

i) Develop the Algorithm State Machine (ASM) Chart realizing the rotation circuit.

ii) List the registers used.

iii) List the RT operations of the register holding the signal to be rotated under different states.

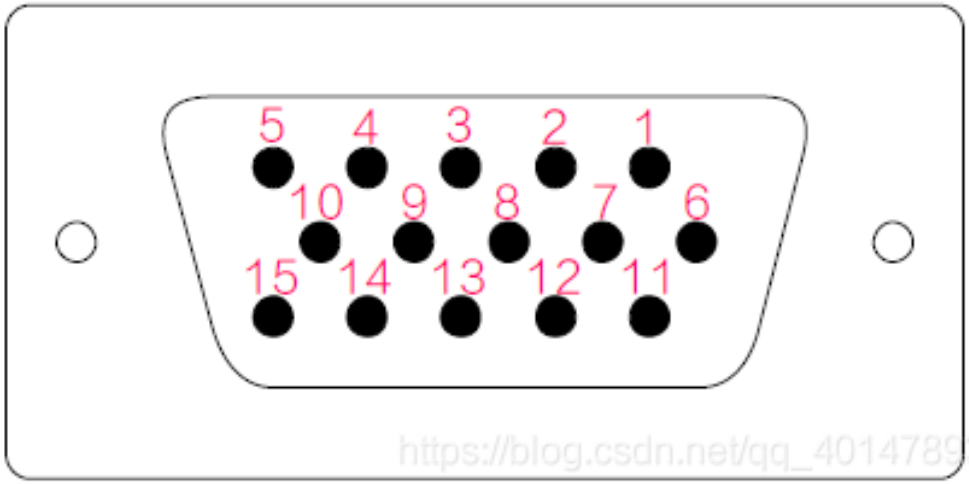
iv) Sketch the conceptual diagram of the circuit associated with the register holding the signal to be rotated.

实战：FPGA驱动VGA显示

- 目标
  - 实现一个独立的FPGA驱动VGA显示的模块
  - 显示彩条
  - 显示一幅静态的图像
  - 结合你自己的项目，显示相应的信息，比如
    - 用摄像头抓取信号，实时显示
    - 对图像处理，显示处理后图像的效果

理解VGA显示原理

- VGA接口 (Video Graphic Array)



Pin 1: Red
Pin 2: Grn
Pin 3: Blue
Pin 13: HS
Pin 14: VS

Pin 5: GND
Pin 6: Red GND
Pin 7: Grn GND
Pin 8: Blu GND
Pin 10: Sync GND

理解VGA显示原理

- VGA接口管脚表

管脚	定义
1	红基色(Red) 
2	绿基色(Green) 
3	蓝基色(Blue) 
13	行同步
14	场同步

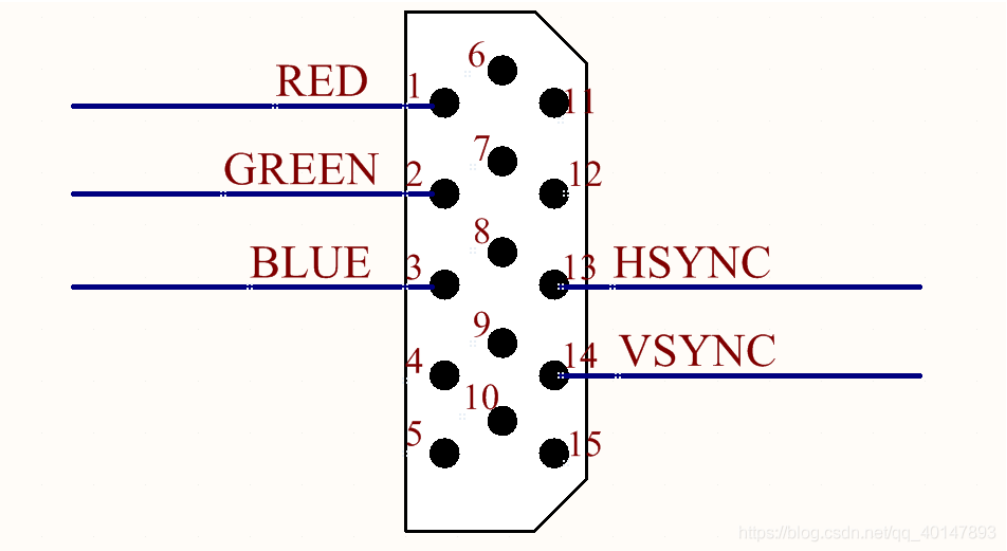
理解VGA显示原理

- **像素点构成：**
  - VGA显示器上每一个像素点可以有多种颜色，由三基色信号R，G，B组合构成。

	R(bit)	G(bit)	B(bit)	可显示颜色数（色彩分辨率）
	1	1	1	2×2×2=8
	3	3	2	8×8×4=256
Artix-7	4	4	4	16x16x16=4096

理解VGA显示原理

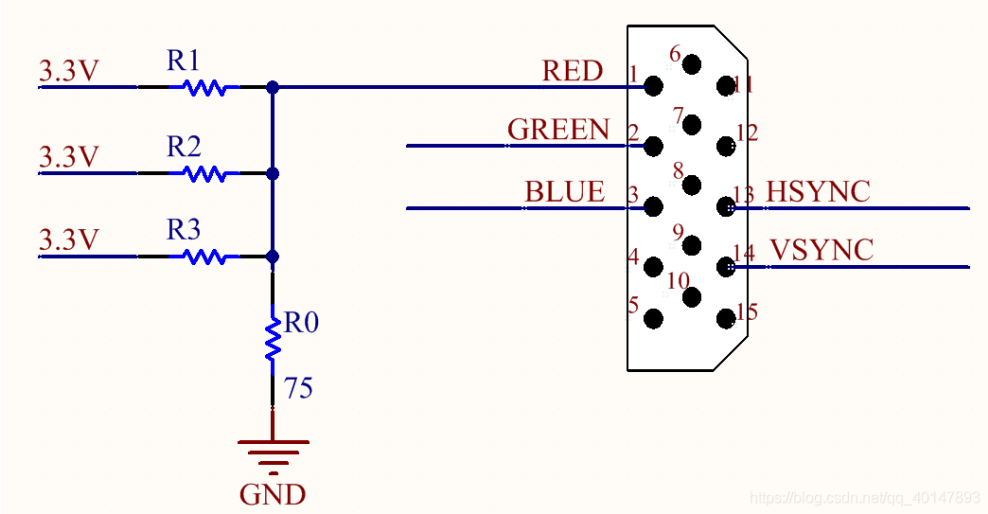
- VGA是如何实现显示的
  - VGA数据引脚1，2，3(RED，GREEN，BLUE)输入的不是0，1数字信号，而是模拟电压0V-0.714V。0V显示黑色，0.714V显示最强的颜色。
  - 1，2，3引脚具有不同的电压时，VGA显示器显示不同的颜色。



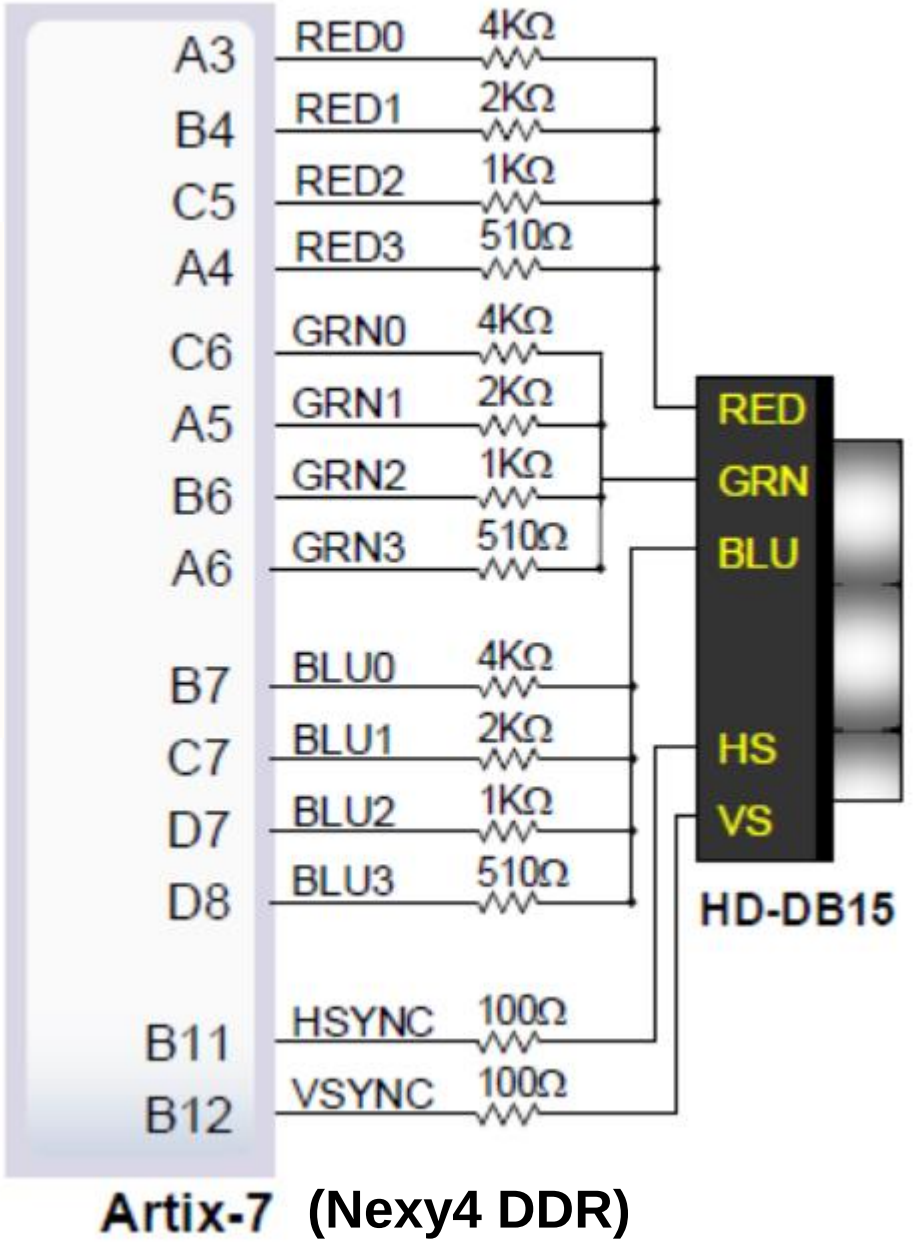
  - 但FPGA只能产生数字信号

理解VGA显示原理

- 利用电阻网络作DA转换

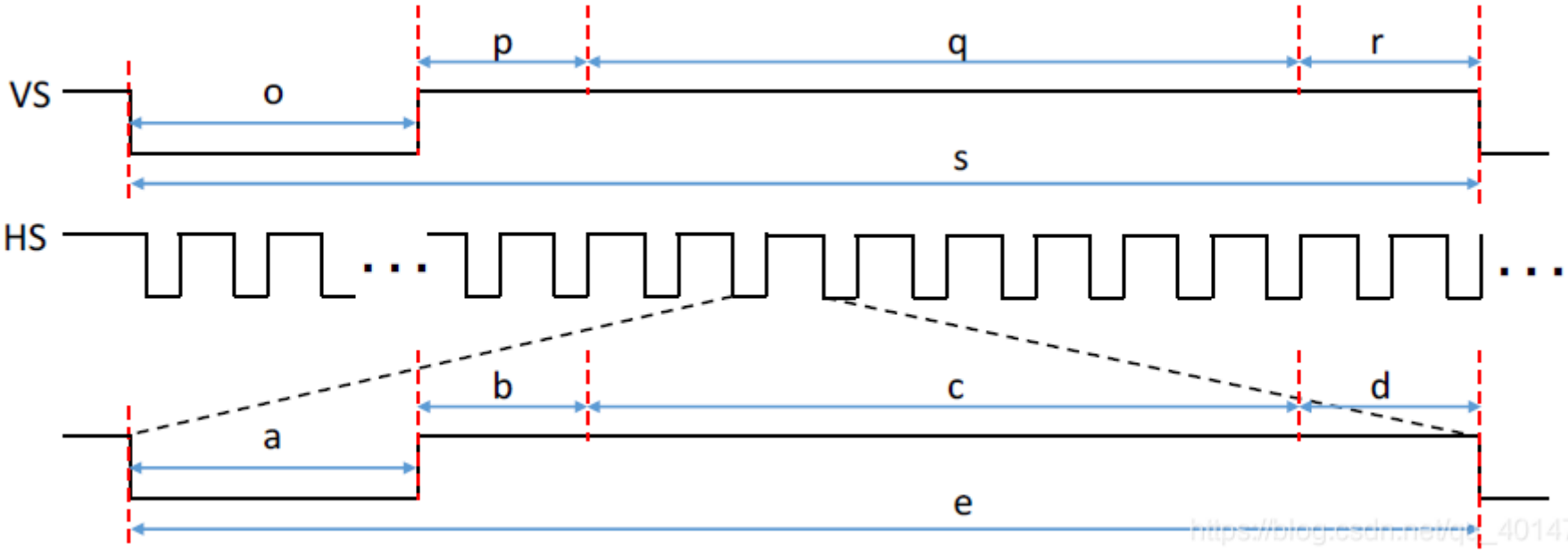


R使用3bit数字信号示意图



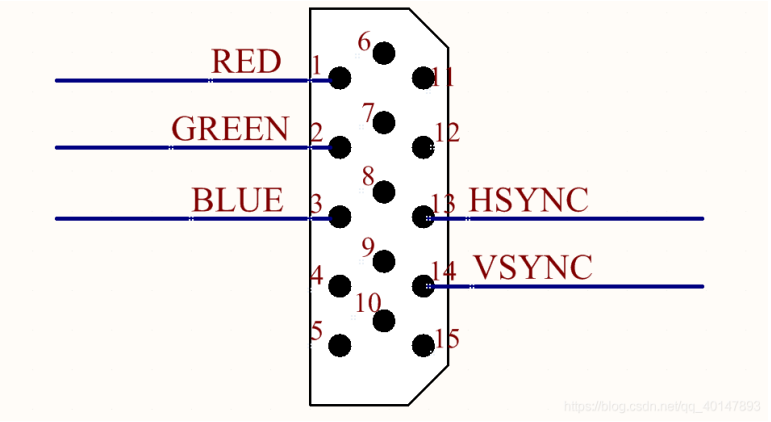
VGA通信协议

- VGA通信时序

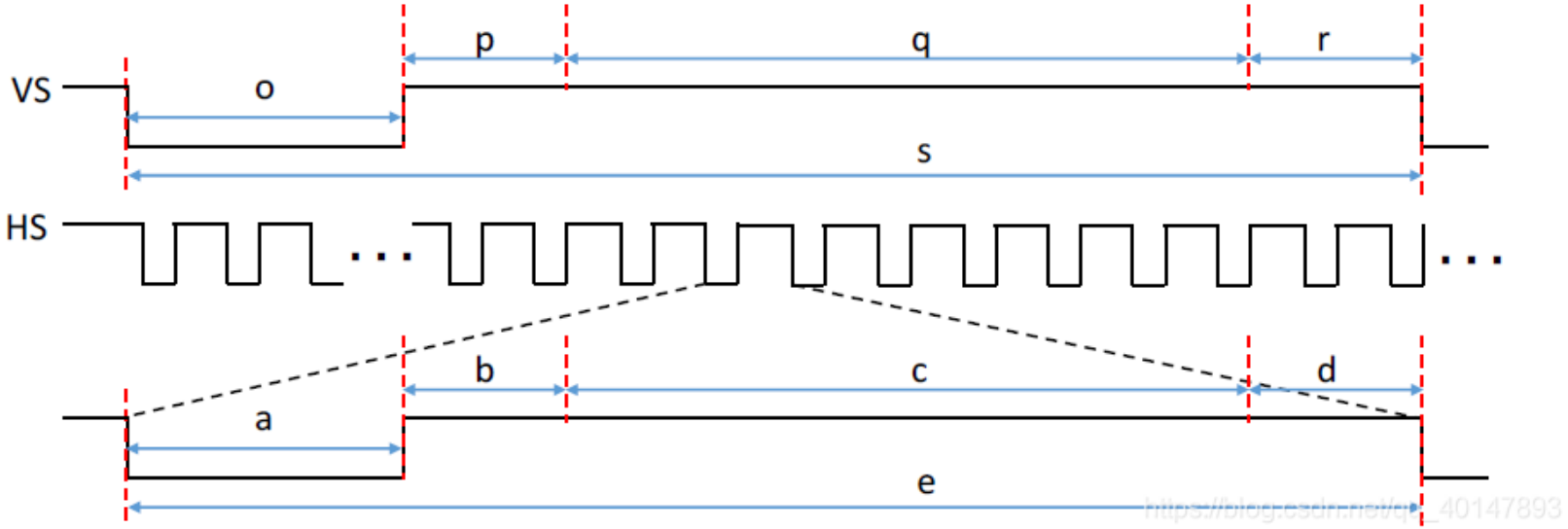


- 帧时序和行时序都有四部分，分别是同步脉冲(Sync)、显示后沿(Back porch)、显示时序段(Display interval)和显示前沿(Front porch)
  - o，p，q，r 和 a，b，c，d

VGA通信协议

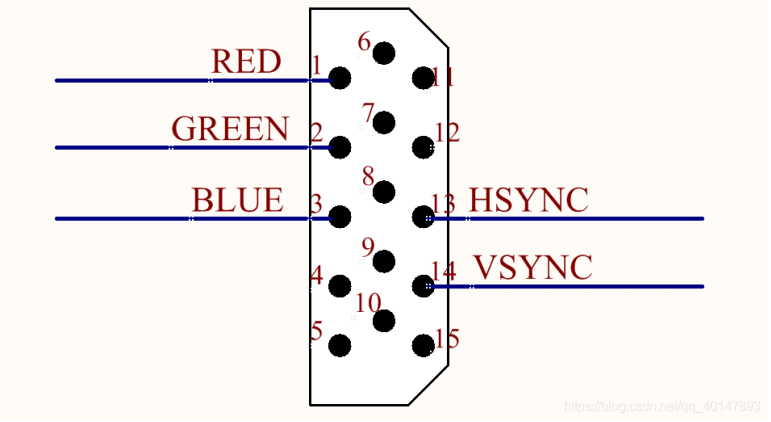


- 帧时序

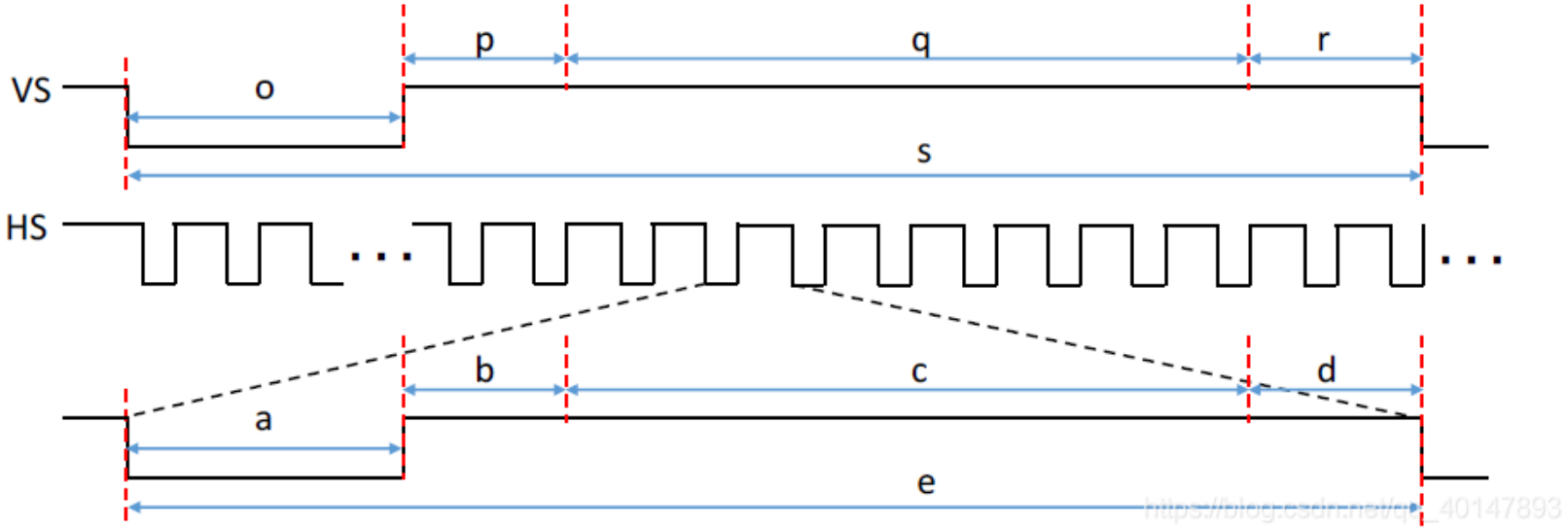


o	p	q	r
同步脉冲(Sync)	显示后沿(Back porch)	显示时序段(Display interval)	显示前沿(Front porch)
RGB信号无效	RGB信号无效	有效数据区	RGB信号无效

VGA通信协议



- 行时序

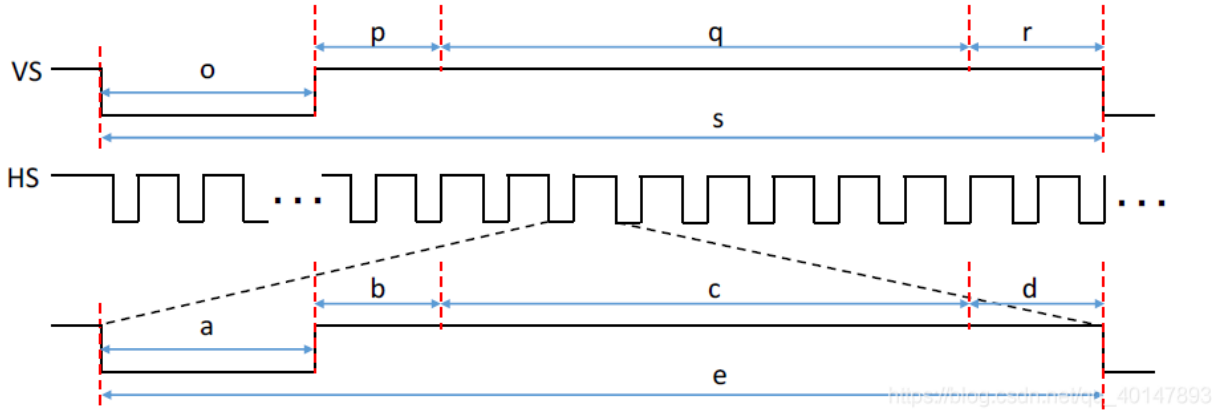


a	b	c	d
同步脉冲(Sync)	显示后沿(Back porch)	显示时序段(Display interval)	显示前沿(Front porch)
RGB信号无效	RGB信号无效	有效数据区	RGB信号无效

VGA显示的空间分辨率和刷新频率

- 空间分辨率：
  - 在VGA显示器上可以显示的像素的个数，例如800x600
- 刷新频率
  - 每秒钟显示的帧数，比如60Hz
- 不同的分辨率，它的时序是不一样的。
  - **例如**800*600@60Hz**的**VGA**时序**：

- 800*600@60Hz



显示模式	时钟（MHz）	行时序（像素数）					帧时序（行数）				
		a	b	c	d	e	o	p	q	r	s
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628

行时序(HSYNC数据线):

a	b	c	d	e
拉低的128个列像素	拉高的88个列像素	拉高的800个列像素	拉高的40个列像素	总共1056个列像素

帧时序(VSYNC数据线):

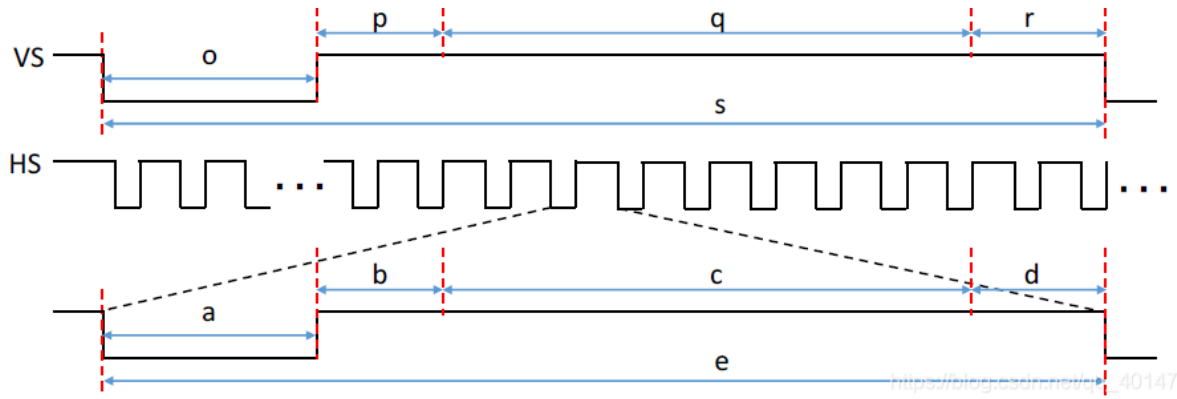
o	p	q	r	s
拉低的4个行像素	拉高的23个行像素	拉高的600个行像素	拉高的1个行像素	总共628个行像素

VGA显示的空间分辨率和刷新频率

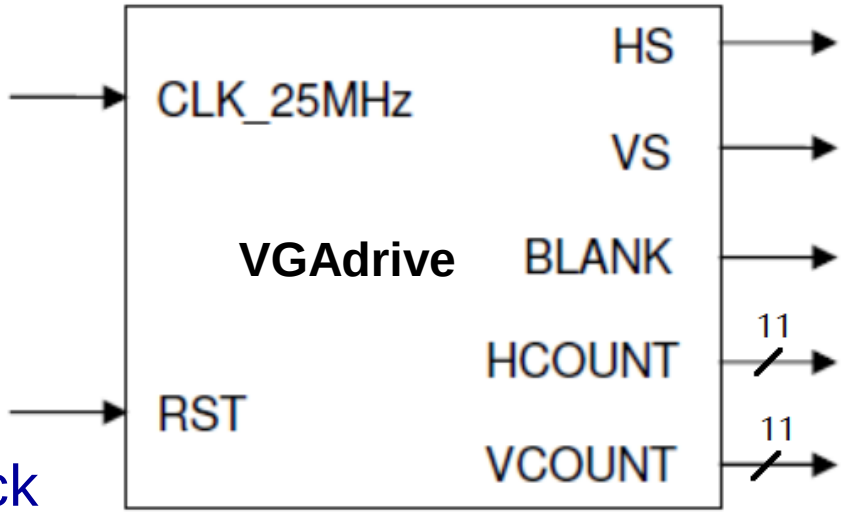
显示模式	时钟（MHz）	行时序（像素数）						帧时序（行数）			
		a	b	c	d	e	o	p	q	r	s
640x480@60	25.175	96	48	640	16	800	2	33	480	10	525
640x480@75	31.5	64	120	640	16	840	3	16	480	1	500
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628
800x600@60	49.5	80	160	800	16	1056	3	21	600	1	625
1024x768@60	65	136	160	1024	24	1344	6	29	768	3	806
1024x768@75	78.8	176	176	1024	16	1312	3	28	768	1	800
1280x1024@60	108.0	112	248	1280	48	1688	3	38	1024	1	1066
1280x800@60	83.46	136	200	1280	64	1680	3	24	800	1	828
1440x900@60	106.47	152	232	1440	80	1904	3	28	900	1	932

Pixel clock

每一时钟周期显示一个像素



VGA驱动模块



- 640x480@60Hz

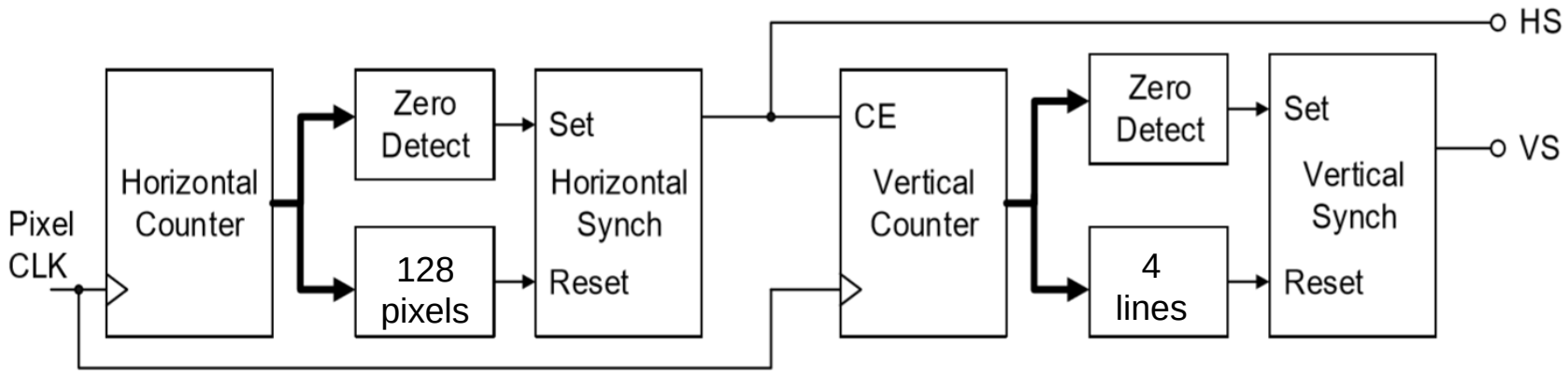
- RST: global reset signal
- CLK_25MHz: input, 25MHz clock

- HS: output, to monitor, horizontal sync signal
- VS: output, to monitor, vertical sync signal
- BLANK: output, to client, blank signal, active when pixel is not in visible area
- HCOUNT: output, 11 bits, to client, horizontal pixel counter
- VCOUNT: output, 11 bits, to client, vertical lines counter

VGA驱动模块

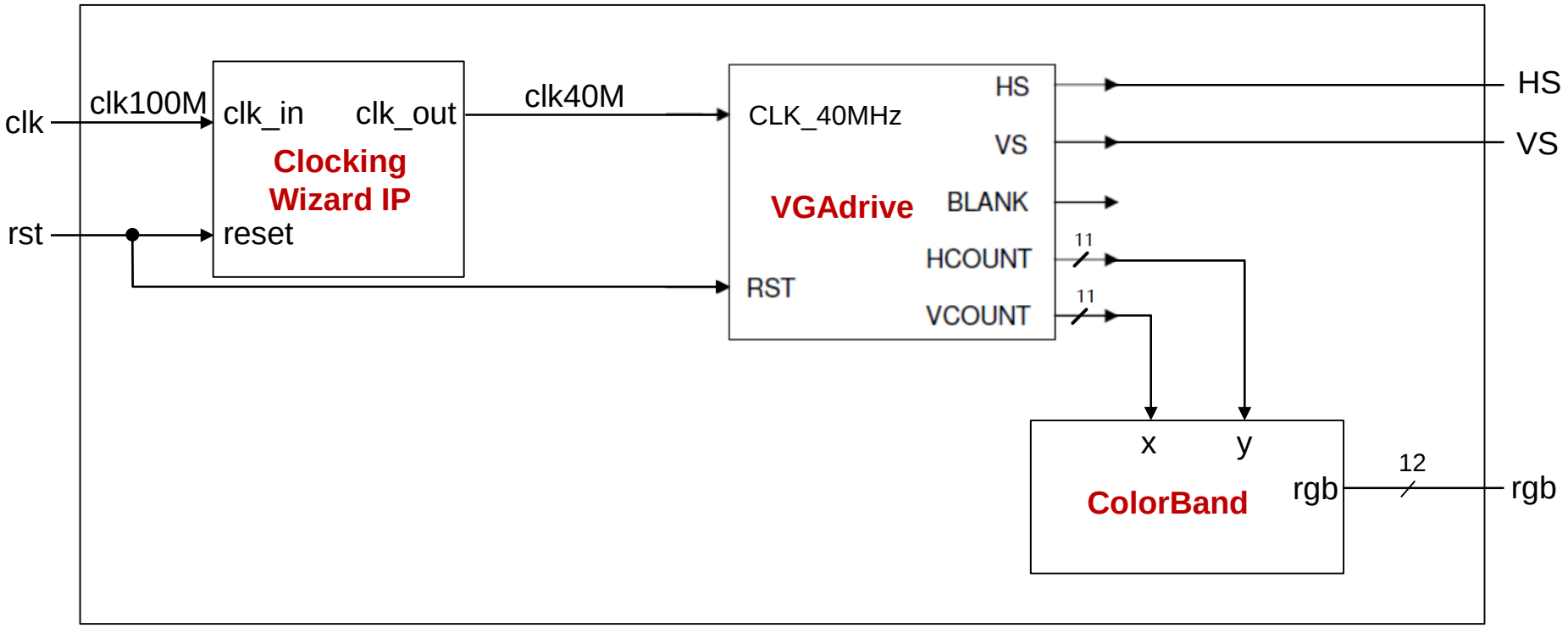
- HS和VS的产生

显示模式	时钟（MHz）	行时序（像素数）					帧时序（行数）				
		a	b	c	d	e	o	p	q	r	s
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628

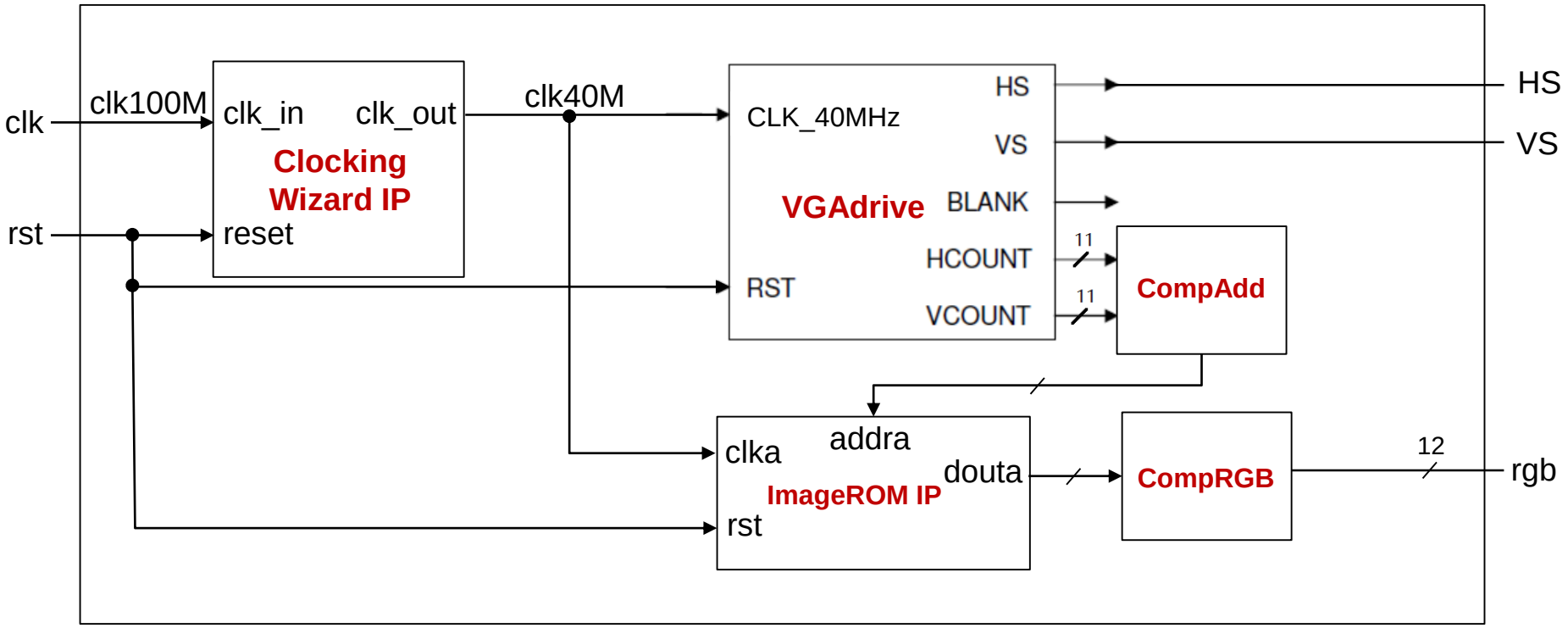


- 另外还需产生HCOUNT和VCOUNT

显示彩条


clk
clk100M
rst
clk_in
clk_out
reset
Clocking
Wizard IP
clk40M
CLK_40MHz
RST
HS
VS
BLANK
HCOUNT
VCOUNT
VGAdrive
11
11
x
y
ColorBand
rgb
12
HS
VS
rgb

显示一幅静态图像


clk
clk100M
clk_in
clk_out
Clocking
Wizard IP
reset
rst
clk40M
CLK_40MHz
HS
VS
VGAdrive
BLANK
HCOUNT
RST
VCOUNT
11
11
CompAdd
HS
VS
clka
addra
douta
ImageROM IP
rst
CompRGB
12
rgb

Use Generic

```
library ieee;  use ieee.std_logic_1164.all;
entity reduced_xor is
        generic (WID: natural); -- generic declaration
        port(
            a: in std_logic_vector(WID–1 downto 0);
            y: out std_logic
        );
end entity reduced_xor;

architecture loop_linear_arch of reduced_xor is
        signal tmp: std_logic_vector(WID–1 downto 0);
begin
        process (a, tmp) is
        begin
            tmp(0) <= a(0); -- boundary bit
            for i in 1 to (WID–1) loop
                    tmp(i) <= a(i) xor tmp(i–1);
            end loop;
        end process;
        y <= tmp(WID-1);
end architecture loop_linear_arch;
```

Use Unconstrained Array

```
library ieee;  use ieee.std_logic_1164.all;
entity unconstrain_reduced_xor is
        port(
            a: in std_logic_vector;
            y: out std_logic
        );
end entity unconstrain_reduced_xor;


architecture better_arch of unconstrain_reduced_xor is
        constant WID: natural := a'length;
        signal tmp: std_logic_vector(WID–1 downto 0);
        signal aa: std_logic_vector(WID-1 downto 0);
begin
        aa <= a;
        process (aa, tmp) is
        begin
            tmp(0) <= aa(0);
            for i in 1 to (WID–1) loop
                    tmp(i) <= aa(i) xor tmp(i–1);
            end loop;
        end process;
        y <= tmp(WID-1);
end architecture better_arch;
```

Use Generate Statement

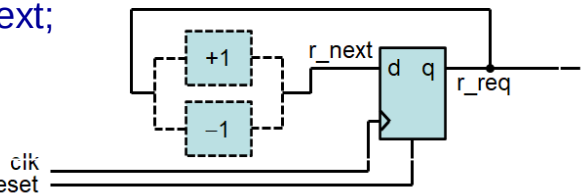
```
architecture gen_linear_arch of reduced_xor is
        signal tmp: std_logic_vector(WID–1 downto 0);
begin
        tmp(0) <= a(0);
        xor_gen:
             for i in 1 to (WID–1) generate
                        tmp(i) <= a(i) xor tmp(i–1);
             end generate;
         y <= tmp(WID-1);
end architecture gen_linear_arch;
```

Use Generate and if Generate Statement

```
architecture gen_if_arch of reduced_xor is
        signal tmp: std_logic_vector(WID–2 downto 1);
begin
        xor_gen:
        for i in 1 to (WID–1) generate
             -- leftmost stage
             left_gen: if i = 1 generate
                        tmp(i) <= a(i) xor a(0);
             end generate;
             -- middle stage
             middle_gen: if (i>1) and (i<(WID –1)) generate
                        tmp(i) <= a(i) xor tmp(i–1);
             end generate;
             -- rightmost stage
             right_gen: if i = (WID –1) generate
                        y <= a(i) xor tmp(i–1);
             end generate;
        end generate;
end architecture gen_if_arch;
```

Up or Down Counter

```
entity up_or_down_counter is
        generic(WID: natural;  UP: natural);
        port(clk, reset: in std_logic;
             code: out std_logic_vector(WID–1 downto 0)
        );
end up_or_down_counter;
architecture arch of up_or_down_counter is
        signal r_reg, r_next: unsigned(WID–1 downto 0);
begin
        -- register
        process (clk, reset)
        begin
             if (reset = '1') then
                      r_reg <= (others => '0')
             elsif (clk'event and clk='1') then
                      r_reg <= r_next;
             end if;
        end process;
        -- next-state logic
        inc_gen: -- incrementor
        if UP = 1 generate
                r_next <= r_reg + 1;
        end generate;
        dec_gen: -- decrementor
        if UP /= 1 generate
                r_next <= r_reg – 1;
        end generate;
        q <= std_logic_vector(r_reg);  -- output logic
end architecture arch;
```



Up and Down Counter

```
entity up_and_down_counter is
        generic map (WID: natural)
        port map(  clk, reset: in std_logic;
                   mode: in std_logic;
                   code: out std_logic_vector
                              (2**WID–1 downto 0)
        );
end up_and_down_counterr;
architecture arch of up_and_down_counter is
        signal r_reg, r_next: unsigned(WID–1 downto 0);
begin
        -- register
        process (clk, reset)
        begin
             if (reset = '1') then
                      r_reg <= (others => '0')
             elsif (clk'event and clk='1') then
                      r_reg <= r_next;
             end if;
        end process;
        -- next-state logic
        r_next <= r_reg + 1 when mode ='0' else
                        r_reg -1;
        -- output logic
        q <= std_logic_vector(r_reg);
end architecture arch;
```

Binary Decoder
using Generate Statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bin_decoder is
        generic(WIDTH: natural);
        port(    a: in std_logic_vector(WIDTH–1 downto 0);
                 code: out std_logic_vector(2**WIDTH–1 downto 0)
        );
end bin_decoder;

architecture gen_arch of bin_decoder is
begin
    comp_gen:
    for i in 0 to (2**WIDTH–1) generate
        code(i) <= '1' when i = to_integer(unsigned(a)) else
                          '0';
    end generate;
end architecture gen_arch;
```

using Loop Statement

```
architecture loop_arch of bin_decoder is
begin
    process (a)
    begin
        for i in 0 to (2**WIDTH–1) loop
            if i = to_integer(unsigned(a)) then
                code(i) <= '1';
            else code(i) <= '0';
            end if;
        end loop;
    end process;
end architecture gen_arch;
```

Reduced XOR circuit using Loop Statement

```
library ieee;  use ieee.std_logic_1164.all;
entity reduced_xor is
        generic (WIDTH: natural); -- generic declaration
        port(
             a: in std_logic_vector(WIDTH–1 downto 0);
             y: out std_logic
        );
end entity reduced_xor;

architecture loop_linear_arch of reduced_xor is
        signal tmp: std_logic_vector(WIDTH–1 downto 0);
begin
        process (a, tmp) is
        begin
             tmp(0) <= a(0); -- boundary bit
             for i in 1 to (WIDTH–1) loop
                       tmp(i) <= a(i) xor tmp(i–1);
             end loop;
        end process;
        y <= tmp(WIDTH-1);
end architecture loop_linear_arch;
```

using Generate Statement

```
architecture gen_linear_arch of reduced_xor is
        signal tmp: std_logic_vector(WIDTH–1 downto 0);
begin
        tmp(0) <= a(0);
        xor_gen:
             for i in 1 to (WIDTH–1) generate
                       tmp(i) <= a(i) xor tmp(i–1);
             end generate;
         y <= tmp(WIDTH-1);
end architecture gen_linear_arch;
```