EECE 353: Digital Systems Design
Introduction to Lab 4

Lab 4

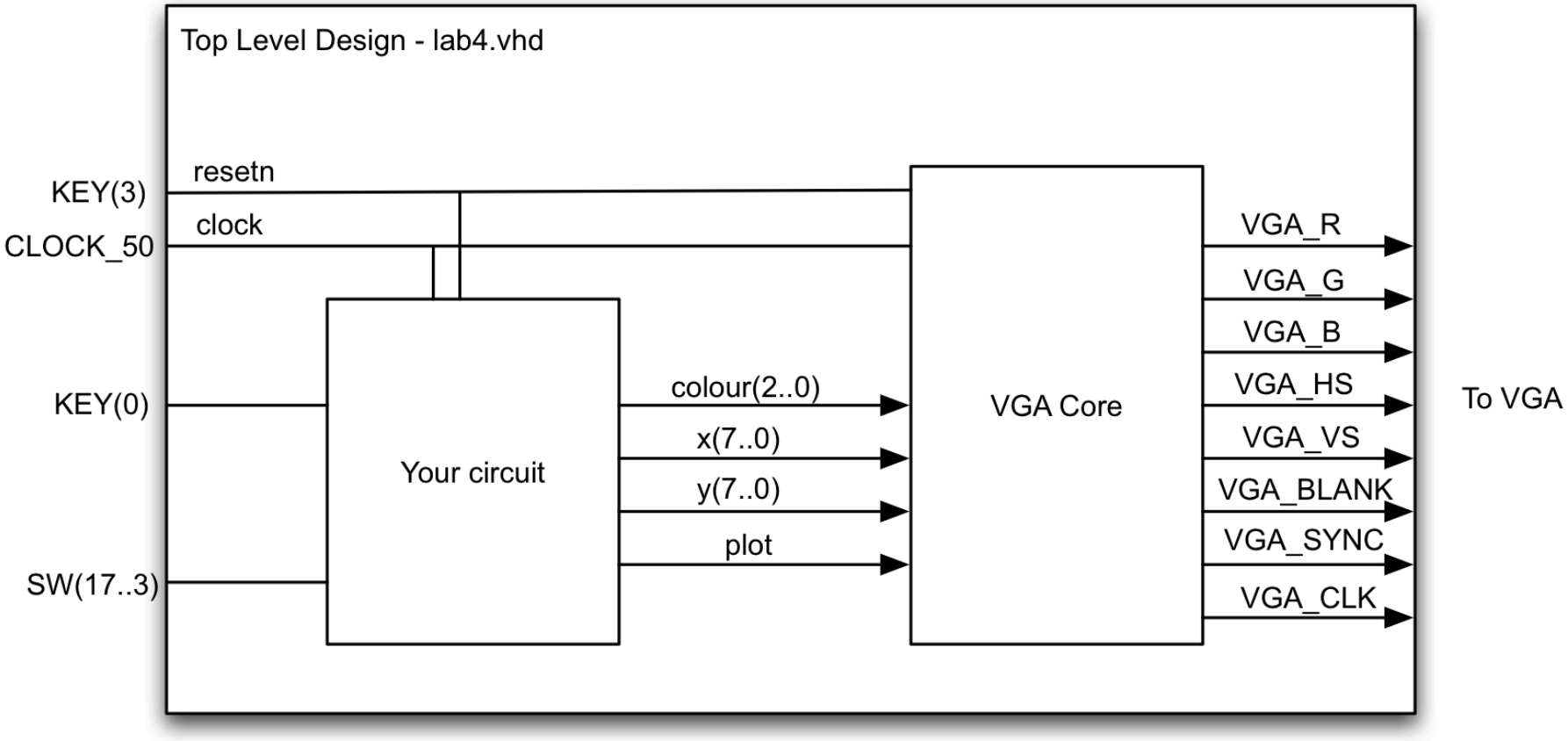
In this lab, you will create a slightly more complicated datapath.

- Unlike Lab 3, we won't give you the details of the datapath
- But this slide set will give you some hints…

We will also use a VGA core to allow you to draw pixels to a VGA screen.

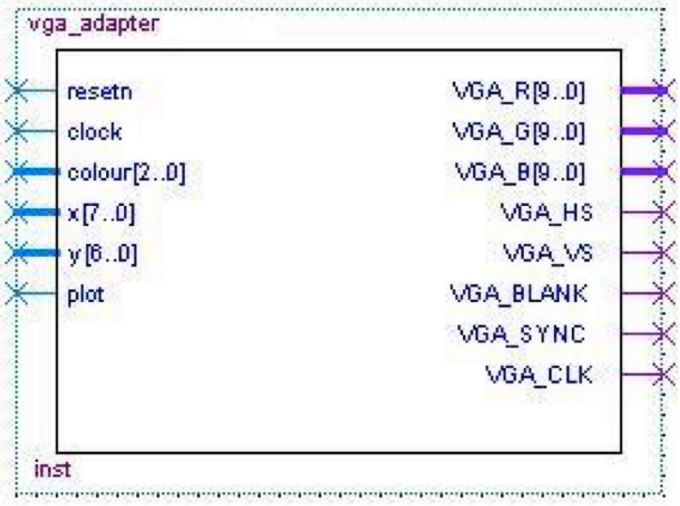
Final goal: a circuit that draws lines connecting two arbitrary points on the screen.

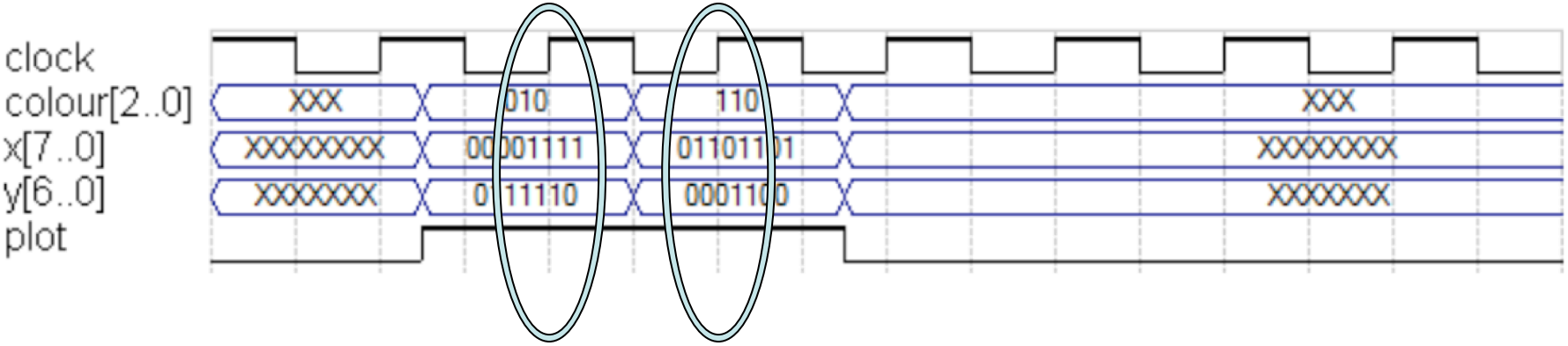
Overall Block Diagram



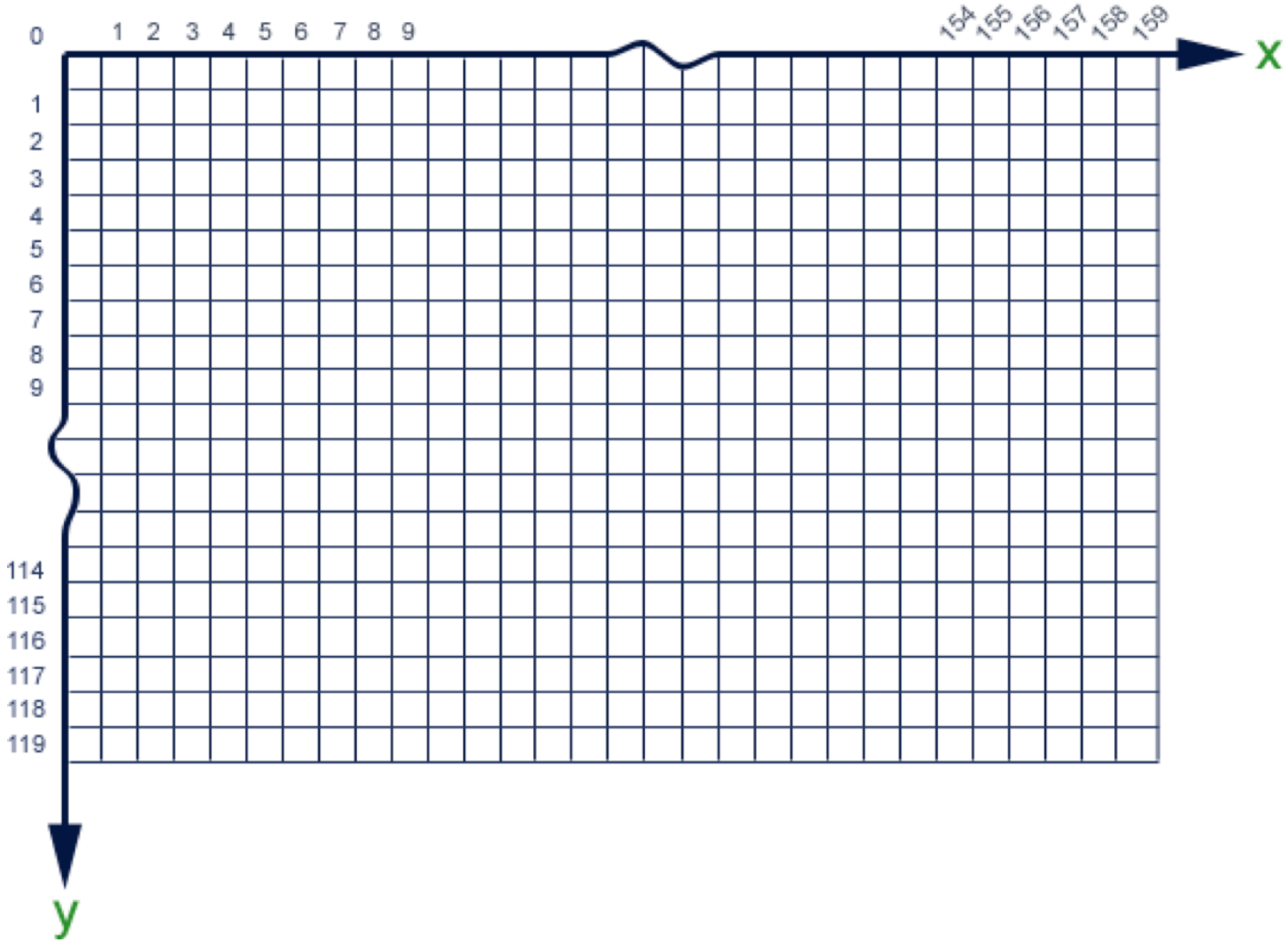
VGA Core – We will give this to you

Can update **one** pixel each clock cycle





This would turn on two pixels

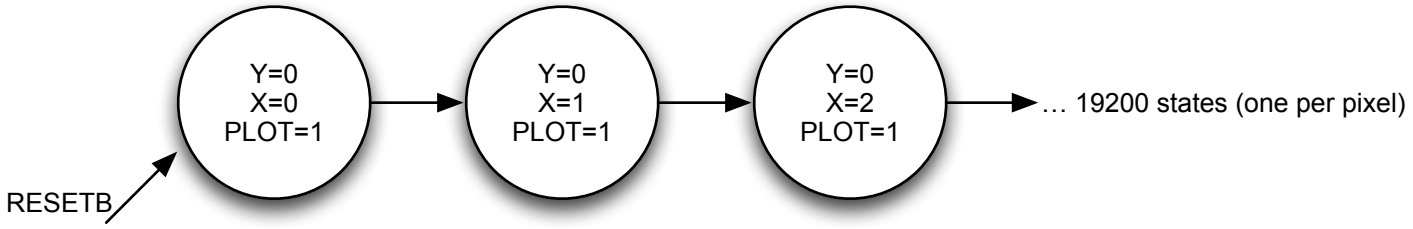

0
1 2 3 4 5 6 7 8 9
154 155 156 157 158 159
x
1
2
3
4
5
6
7
8
9
114
115
116
117
118
119
y

Task 2: Fill the Screen

Turn on each pixel one at a time:

```
for y = 0 to 119 {
        for x = 0 to 159 {
                turn on pixel (x, y) with colour (y mod 8)
    }
}
```

Naïve way to do it:

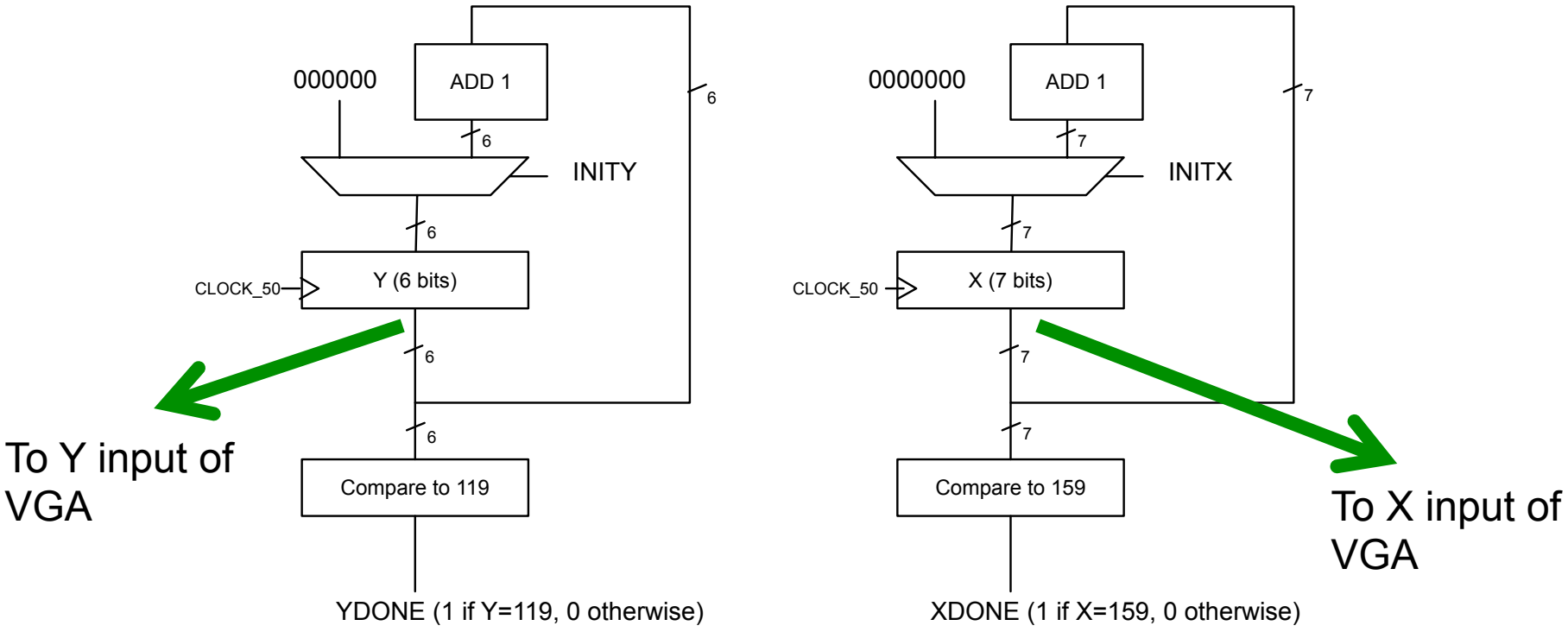


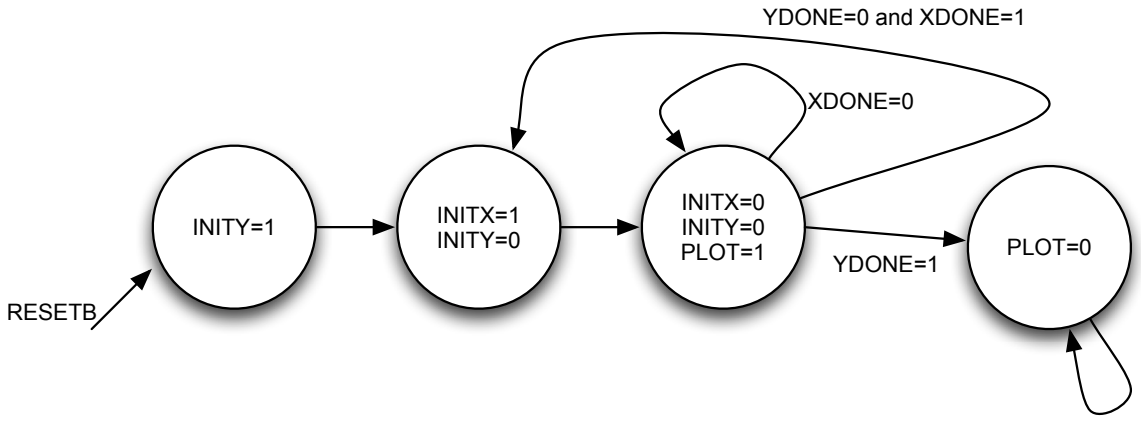
Using VHDL FOR Loops ?

This won't work because you are trying to do everything in one clock cycle. You can only update one pixel per clock cycle

```
process (CLK)
begin
    for Y in 0 to 119 loop
      for X in 0 to 159 loop
        <tell VGA Adaptor to update pixel>
      end loop
    end loop
end process
```

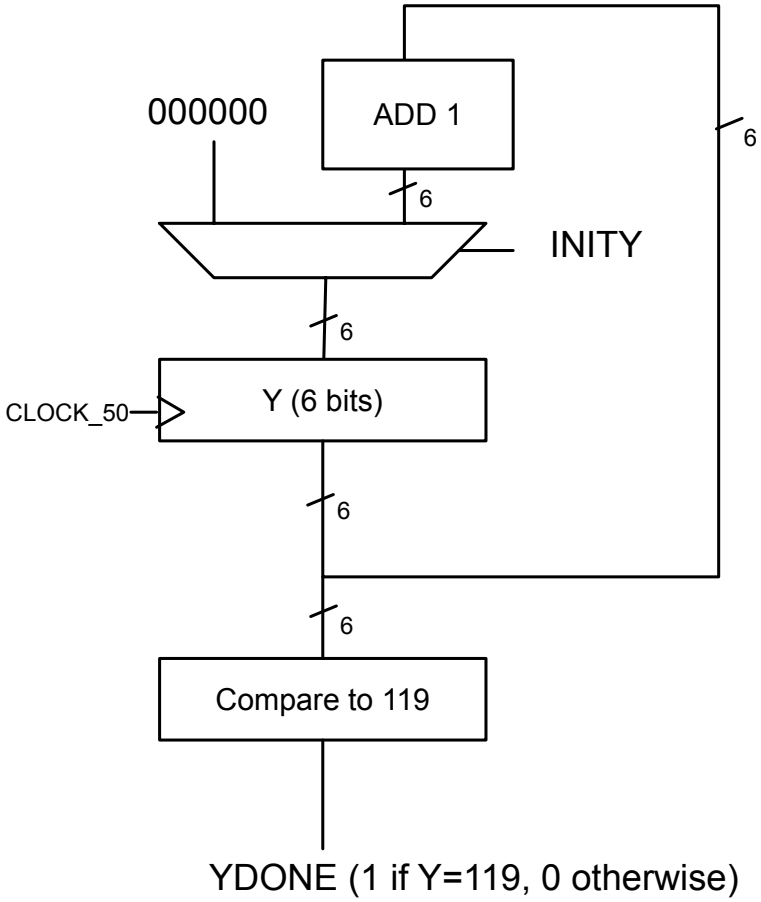
To fill the screen, you need 120 x 160 = 19200 clock cycles


000000
ADD 1
6
6
INITY
6
CLOCK_50
Y (6 bits)
6
6
Compare to 119
YDONE (1 if Y=119, 0 otherwise)
To Y input of VGA
0000000
ADD 1
7
7
INITX
7
CLOCK_50
X (7 bits)
7
7
Compare to 159
XDONE (1 if X=159, 0 otherwise)
To X input of VGA


YDONE=0 and XDONE=1
XDONE=0
INITY=1
INITX=1
INITY=0
INITX=0
INITY=0
PLOT=1
YDONE=1
PLOT=0
RESETB

You know how to write that datapath and state machine in VHDL.

Short-cut: don't need to do everything structurally:



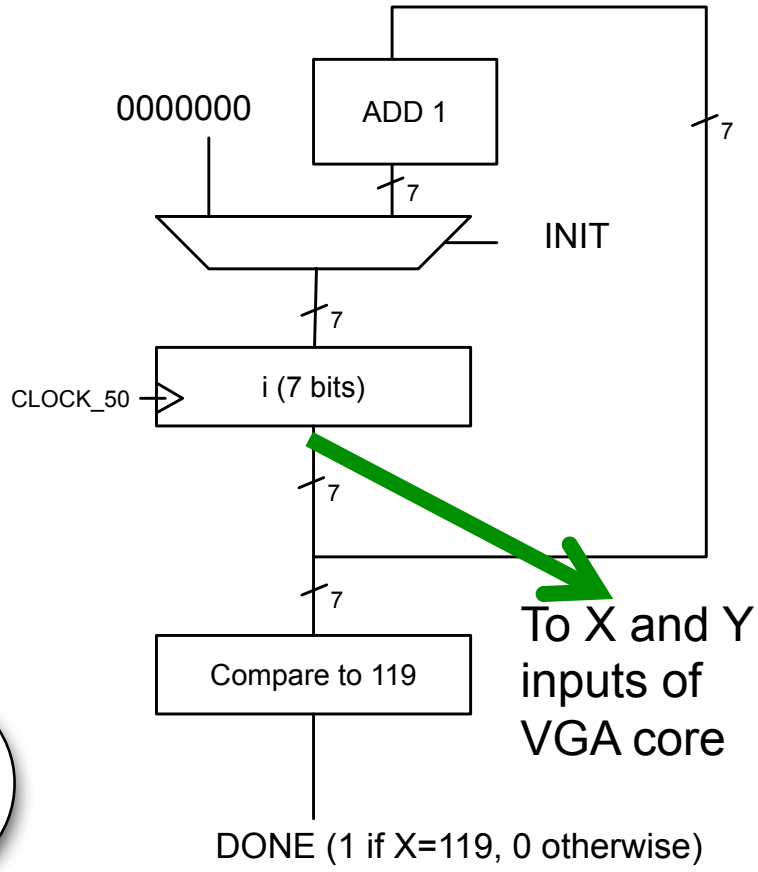

```
process(CLOCK_50)
variable Y : unsigned(5 downto 0);
begin
    if rising_edge(CLOCK_50) then
        if (INITY = '1') then
            Y := "000000";
        else
            Y := Y + 1;
        end if;
        YDONE <= '0';
        if (Y = 119) then
           YDONE <= '1';
        end if;
    end if;
end process;
```

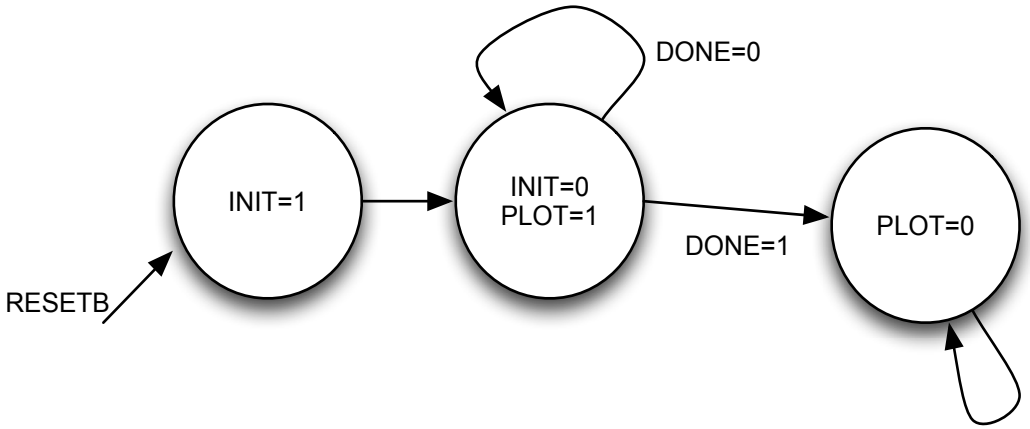
```
process(CLOCK_50)
variable Y : unsigned(5 downto 0);
variable X : unsigned(6 downto 0);
begin
    if rising_edge(CLOCK_50) then
        if (INITY = '1') then
            Y := "000000";
        else
            Y := Y + 1;
        end if;
        if (INITX = '1') then
            X := "0000000";
        else
            X := X + 1;
        end if;
        YDONE <= '0';
        XDONE <= '0';
        if (Y = 119) then
           YDONE <= '1';
        end if;
        if (X = 159) then
           XDONE <= '1';
        end if;
    end if;
```

Diagonal Lines

Not directly part of this lab, but say we want to draw a diagonal line from (0,0) to (119, 119):

```
for i = 0 to 119 {
    turn on pixel (i, i)
}
```





Task 3: Lines connecting arbitrary points

You can imagine it would be simple enough to work out the slope of the line (*m*) and, each iteration, increment y by *m* and x by 1.

Problem: this involves fractional arithmetic

- We will talk about fractional arithmetic later, however, for now, understand it would be more complex than it needs to be

Solution: Bresenham Line Algorithm:

- Uses only integer arithmetic

We are connecting (x0, y0) to (x1, y1):

```
dx := abs(x1-x0)
dy := abs(y1-y0)

if x0 < x1 then sx := 1 else sx := -1
if y0 < y1 then sy := 1 else sy := -1

err := dx-dy

x := x0
y := y0

loop
  setPixel(x, y)

  if x = x1 and y = y1 exit loop

  e2 := 2*err

  if e2 > -dy then
    err := err – dy
    x := x + sx
  end if

  if e2 < dx then
    err := err + dx
    y := y + sy
  end if
end loop
```

We are connecting (x0, y0) to (x1, y1):

```
dx := abs(x1-x0)
dy := abs(y1-y0)

if x0 < x1 then sx := 1 else sx := -1
if y0 < y1 then sy := 1 else sy := -1

err := dx-dy

x := x0
y := y0

loop
  setPixel(x, y)

  if x = x1 and y = y1 exit loop

  e2 := 2*err

  if e2 > -dy then
    err := err – dy
    x := x + sx
  end if

  if e2 < dx then
    err := err + dx
    y := y + sy
  end if
end loop
```

Identify Registers to store computed values

Looks like we need 7:

- dx
- dy
- sx
- sy
- err
- x
- y
- e2

so draw those

We are connecting (x0, y0) to (x1, y1):

```
dx := abs(x1-x0)
dy := abs(y1-y0)

if x0 < x1 then sx := 1 else sx := -1
if y0 < y1 then sy := 1 else sy := -1

err := dx-dy

x := x0
y := y0

loop
  setPixel(x, y)

  if x = x1 and y = y1 exit loop

  e2 := 2*err

  if e2 > -dy then
    err := err – dy
    x := x + sx
  end if

  if e2 < dx then
    err := err + dx
    y := y + sy
  end if
end loop
```

For each of these registers (pieces of data), identify the algorithm's statements that WRITE to the register.

The value being written (right-hand-side) is usually some expression. These expressions will be implemented by combinational logic.

For example

```
x := x + sx
```

Means that at some point, the input to the x register will be connected to an adder whose operands are the contents of the `x` and `sx` registers.

We are connecting (x0, y0) to (x1, y1):

```
dx := abs(x1-x0)
dy := abs(y1-y0)

if x0 < x1 then sx := 1 else sx := -1
if y0 < y1 then sy := 1 else sy := -1

err := dx-dy

x := x0
y := y0

loop
  setPixel(x, y)

  if x = x1 and y = y1 exit loop

  e2 := 2*err

  if e2 > -dy then
    err := err – dy
    x := x + sx
  end if

  if e2 < dx then
    err := err + dx
    y := y + sy
  end if
end loop
```

Some registers will be written to with different expressions at different points of the algorithm.

In hardware, we use a MUX to connect these different possible values to the register input.

The MUX is controlled by signals from the FSM which will be keeping track of the algorithm's execution.

For example, the x register is connected to x0 during the initialization state, and connected to x + sx under some condition during computation.

We are connecting (x0, y0) to (x1, y1):

```
dx := abs(x1-x0)                              -- State0
dy := abs(y1-y0)                              -- State1

if x0 < x1 then sx := 1 else sx := -1         -- State2
if y0 < y1 then sy := 1 else sy := -1         -- State3

err := dx-dy                                  -- State4

x := x0                                       -- State5
y := y0                                       -- State6

loop
  setPixel(x, y)

  if x = x1 and y = y1 exit loop

  e2 := 2*err

  if e2 > -dy then
    err := err – dy
    x := x + sx
  end if

  if e2 < dx then
    err := err + dx
    y := y + sy
  end if
end loop
```

Next step:

Figure out when each operation happens (draw state machine)

We are connecting (x0, y0) to (x1, y1):

```
dx := abs(x1-x0)                              -- State0
dy := abs(y1-y0)                              -- State0

if x0 < x1 then sx := 1 else sx := -1         -- State0
if y0 < y1 then sy := 1 else sy := -1         -- State0

err := dx-dy                                  -- State1

x := x0                                       -- State0
y := y0                                       -- State0

loop
  setPixel(x, y)

  if x = x1 and y = y1 exit loop

  e2 := 2*err

  if e2 > -dy then
    err := err – dy
    x := x + sx
  end if

  if e2 < dx then
    err := err + dx
    y := y + sy
  end if
end loop
```

Try to identify parts that can be done in parallel
(in the same state)

A few Hints and Warnings:

1. Plan your datapath and FSM before you even start writing any VHDL code.

2. The inputs and outputs of each datapath and controller will depend on your design. Everyone's might be different.

3. The x0 and y0 registers of the datapath should be connected directly to the x and y inputs of the VGA controller. The state machine will generate the PLOT signal.

4. Do not specify x and y coordinates outside of the screen dimensions. The VGA core does not clip as you might expect.

5. Remember you do not need to describe each datapath component as a separate file (as you did in Lab 3). I can imagine your entire datapath as one entity.

6. In all the arithmetic operations, you are using *signed* arithmetic now (just use type “signed” instead of “unsigned”).

7. Be mindful of your variable/signal vector sizes. Are they sized large enough? For example, in the Bresenham Algorithim, “e2 := 2*error”. “e2” must be 1-bit larger than “error” to not overflow.

8. Take care if you are casting between **UNSIGNED** and **SIGNED**.
   These types are still just vectors of bits.
   Casting will NOT perform 2s complement conversion

```
variable u : UNSIGNED(2 downto 0); -- can represent 0 to 7
variable s : SIGNED(2 downto 0);   -- can represent -4 to 3

...

u := "111";       -- This is interpreted as decimal 7

s := SIGNED(u);   -- This is now interpreted as -1
                  -- "111" in 2s complement is -1
```

Challenge Task:

Use a Linear Feedback Shift Register to generate the X and Y coordinates and draw lines quickly and randomly.

Warning: with a 50Mhz clock, the screen will fill up so fast you won't see anything

- Need to slow it down. But don't slow down the clear-screen!