



FINAL PROJECT

Nate Lannan

Oklahoma State University

Electrical and Computer Engineering Department

Stillwater, OK 74078 USA

Objective

- This project has several key objectives:
 - The project is meant to give you a comprehensive view of the course and put all the lab skills you learned into one task.
 - It is mean to take all your skills and use them together to form a good structure for what you learned throughout the semester.
 - It will also take in key concepts for the 2nd part of the semester where we will discuss digital system design.
 - But it will also focus on verification and learning all you did through the whole semester.
- Final Project Repo: https://github.com/natelannan-osu/dldFinalProject_Spring24
- You may choose your own partners. If you are not in the same lab section, please meet during the least populated lab section time.
- Contact Alex or Cale by 4/8/24 to let them know who your partner is

What makes digital logic special?

- Digital logic is special in that it can control output for a given input sequence.
 - Hardware takes advantage of parallel computation!
- To do this well, a good separation of datapath and control is kept in mind.
- This is true for early computer games as they often had good amounts of control for a given display.
 - Early computer games like Asteroids, Defender, Pong used a lot of Finite State Machines (FSMs) to guarantee its output.
 - I believe Steve Jobs and Steve Wozniak worked on specific games for Atari.

Cellular Automata

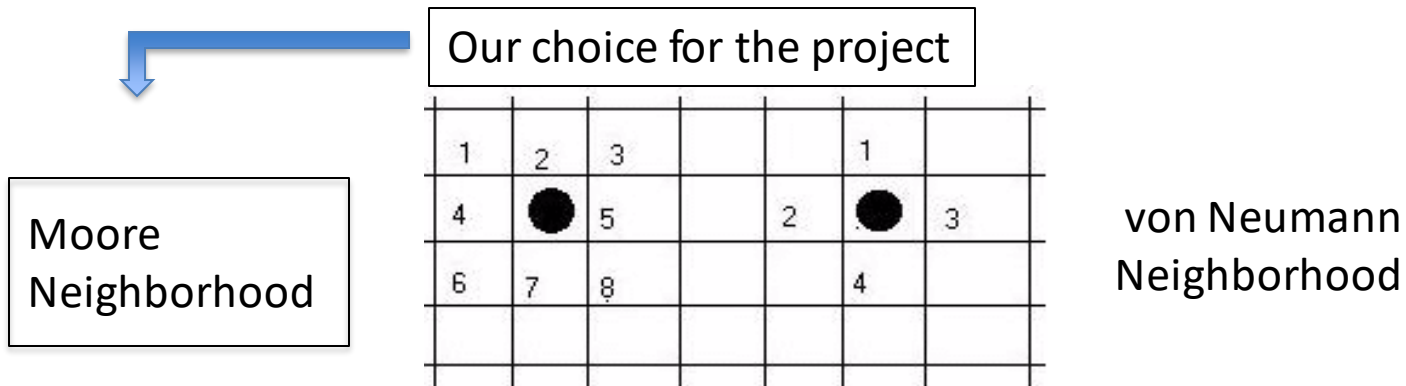
- A cellular automata is a family of simple, finite-state machines that exhibit interesting, **emergent** behaviors through their interactions in a population
- In 1947, John von Neumann was working in the field of science, focusing on biology. He was studying a self-replicating machine when he designed a two-dimensional CA model of the physics of our universe.
- He mathematically proved that the universe he made acted like a self-replicating machine and that it would make endless copies of itself.
- The von Neumann model of computer architecture is also still utilized today.



Jon von Neumann

Game of Life

- The best-known CA is John Horton Conway's "Game of Life".
 - Invented 1970 in Cambridge.
 - He was a researcher at Princeton University (1937-2020)
 - <https://www.nytimes.com/2020/12/28/science/math-conway-game-of-life.html>
- Objective: To make a 'game' as unpredictable as possible with the simplest possible rules.
- 2-dimensional grid of squares on a (possibly infinite) plane. Each square can be blank or occupied.



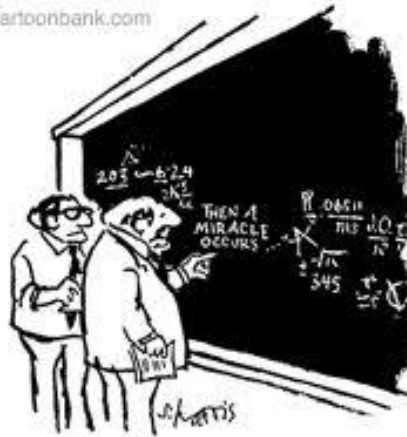
Game of Life

- The grid is populated with some initial dots.
- This is called a zero-player game because all you have to do is start things and off it goes.
- <http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/>
- Every time tick all squares are updated simultaneously, according to a few simple rules, depending on the local situation.
 - For any one cell, the cell changes based on the current values of itself and 8 immediate neighbors.
 - https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Game of Life Update Rules

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbor's lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

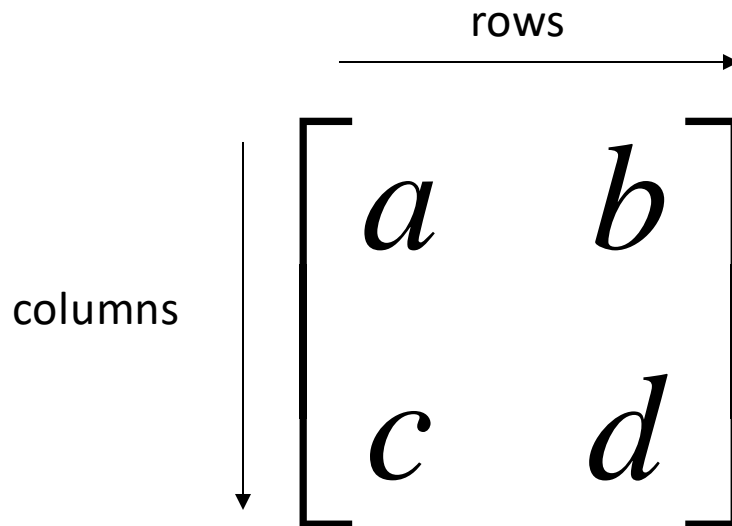
© Cartoonbank.com



"I think you should be more explicit here in step two."

What is a matrix?

- A matrix is a set of elements, organized into rows and columns.
- Conway's Game of Life involves computation of matrices or independent computations and then rules based on those computations.



Sample Computations

- Addition, Subtraction, Multiplication.
- Matrices are essential elements of all engineering degrees and I guarantee you will see more of them later in your degree program.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

Just add elements

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a-e & b-f \\ c-g & d-h \end{bmatrix}$$

Just subtract elements

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

**Multiply each row by
each column**

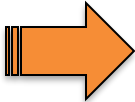
Matrix Multiplication

Matrix Multiplication

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 2 & 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 5 \\ 6 & 7 \\ 1 & 8 \end{bmatrix}$$

Example 4x4 : 1 iteration

1	0	1	1
0	0	1	1
1	1	1	1
0	0	0	0



0	1	1	1
1	0	0	0
0	1	0	1
0	1	1	0

- Different patterns can be made with different starting seeds!
- How many computations are done for one iteration?

GPUs

- This idea of processing matrices is very similar to what we do with Graphics Processor Units (GPUs).
- GPUs are truly parallel units that just process a ton of computations in parallel, like many matrices.

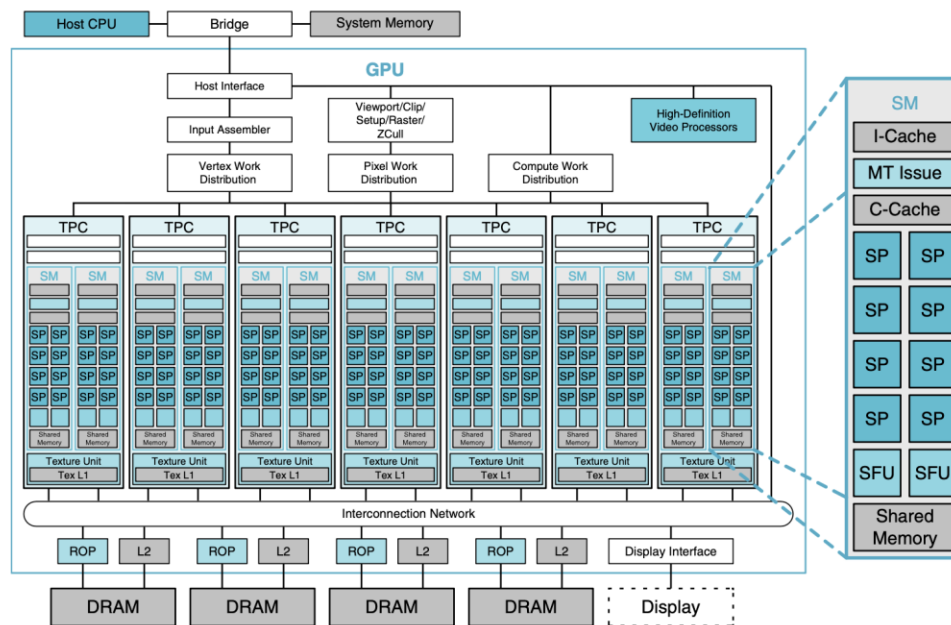


FIGURE B.2.5 Basic unified GPU architecture. Example GPU with 112 *streaming processor* (SP) cores organized in 14 *streaming multiprocessors* (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two *special function units* (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

Hardware Implementation

- Implementations vary, but our point is to do things in parallel similar to a GPU.
- Use register to hold ALL values in one contiguous element

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	1	1	1	1	1	1	0	0	0	0

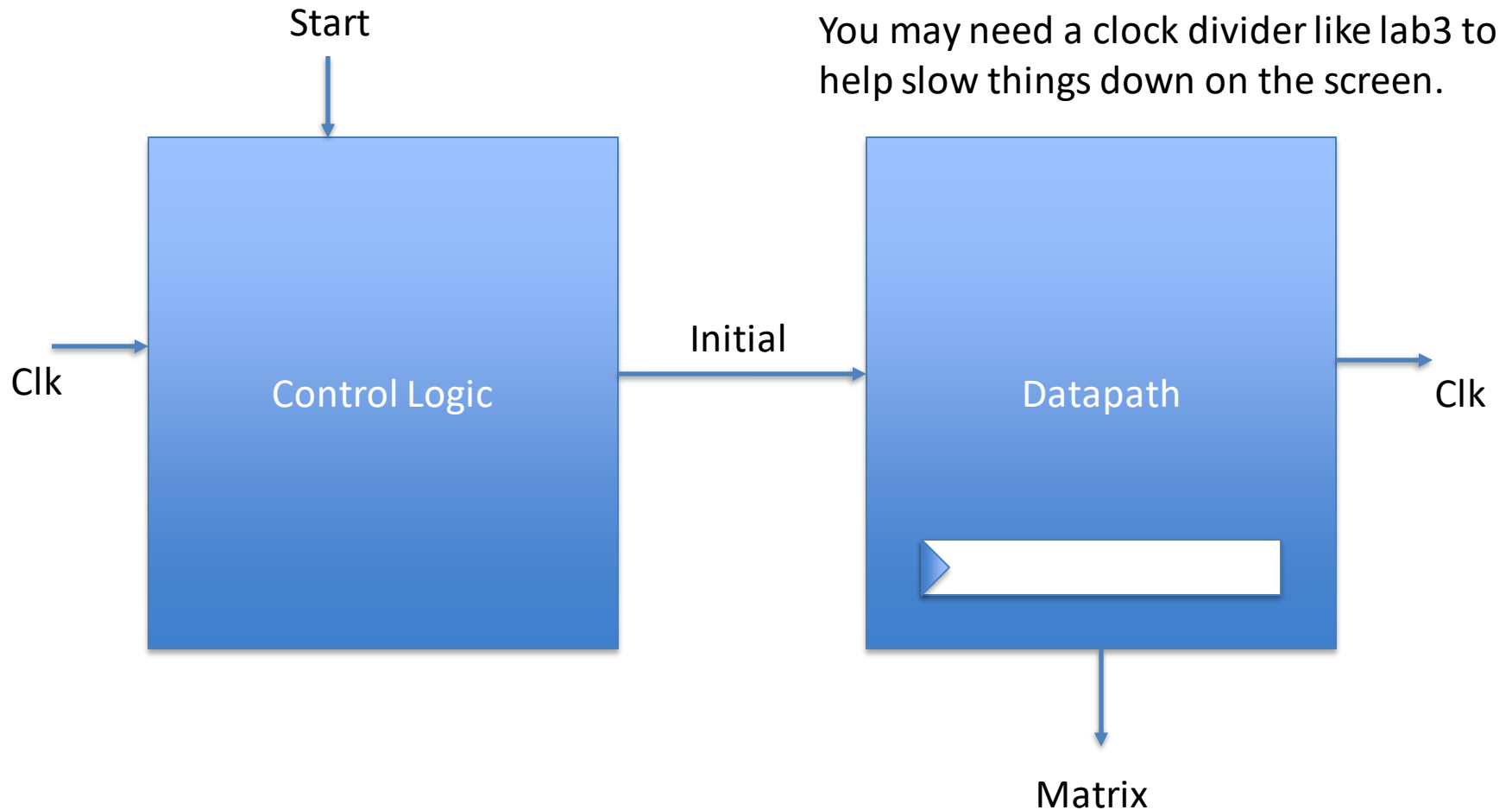
Example from Previous Slide

$$\text{Index} = (\text{row} - 1) * 8 + (\text{column} - 1)$$

```
assign grid = 64'h0412_6424_0034_3C28;
```

Figure 2 : Good Starting Seed

Digital System



The New Matrix is output for each new image at a clock cycle

Verilog (datapath)

```
module datapath ( grid, grid_evolve );
```

```
    output [63:0]    grid_evolve;
```

```
    input  [63:0]    grid;
```

```
    evolve3 e0_0 (grid_evolve[0], grid[1], grid[8], grid[9], grid[0]);
```

```
    evolve5 e0_1 (grid_evolve[1], grid[0], grid[2], grid[8], grid[9], grid[10], grid[1]);
```

```
    evolve5 e0_2 (grid_evolve[2], grid[1], grid[3], grid[9], grid[10], grid[11], grid[2]);
```

```
    evolve5 e0_3 (grid_evolve[3], grid[2], grid[4], grid[10], grid[11], grid[12], grid[3]);
```

```
    evolve5 e0_4 (grid_evolve[4], grid[3], grid[5], grid[11], grid[12], grid[13], grid[4]);
```

```
    evolve5 e0_5 (grid_evolve[5], grid[4], grid[6], grid[12], grid[13], grid[14], grid[5]);
```

```
    evolve5 e0_6 (grid_evolve[6], grid[5], grid[7], grid[13], grid[14], grid[15], grid[6]);
```

```
    evolve3 e0_7 (grid_evolve[7], grid[6], grid[14], grid[15], grid[7]);
```

...

Current datapath only processes one iteration!!!!!!

evolve

- Evolve has three different modules
- These modules compute $a+b+c$, $a+b+c+d+e$, and $a+b+c+d+e+f+g+h$
 - That is, it adds 3, 5, and 8 values
 - The rules are then examined at the end in a simple selection function based on Slide Page 7 (in this lecture)
- This can super easily be expanded to a matrix bigger than 8×8 by changing the datapath (given to you)
- HLS (High-Level Synthesis) is used to help you with the datapath.
 - Check it out!

Verilog (evolve3)

```
module evolve3 (next_state, vector1, vector2, vector3,
               current_state);

    input      vector1;
    input      vector2;
    input      vector3;
    input      current_state;
    output     next_state;

    wire [2:0] sum;

    assign sum = vector1 + vector2 + vector3;
    rules r1 (sum, current_state, next_state);

endmodule // evolve3
```

Verilog (rules)

```
module rules (pop_count, current_state, next_state);  
  
    input [2:0] pop_count;  
    input      current_state;  
    output     next_state;  
  
    assign next_state = (pop_count == 2 &  
        current_state) | pop_count == 3;  
  
endmodule // rules
```

LFSR initialization

- Linear Feedback Shift Register
 - Shift register where the input bit is a linear function of previous state
 - Very useful for quick generation of pseudo random sequences
 - Many options for what the linear function is for the shift in
- Maximal LFSR
 - An LFSR that cycles through all values before repeating (disregarding lock up states)
- We want to generate a 64 bit LFSR to cycle through all possible values changing at each clock cycle
 - You should have a button that shows the cycling of your initialization seed
 - You should have a button that loads a seed from your LFSR

LFSR initialization

- We will be using a simplified version of an LFSR based on this document: https://courses.cs.washington.edu/courses/cse369/16wi/labs/xapp052_LFSRs.pdf
- There is a lot of implementation in this document for avoiding the lock state of the LFSR, but we are going to go with a more simplified version (the XOR or XNOR implementation)
 - There are known "taps" that produce a maximal LFSR. These taps are bits that are used from the the bit field to generate the input to the shift register
 - A table for known taps for a given bit length is provided in the above document

LFSR initialization

n	XNOR from	n	XNOR from	n	XNOR from	n	XNOR from
3	3,2	45	45,44,42,41	87	87,74	129	129,124
4	4,3	46	46,45,26,25	88	88,87,17,16	130	130,127
5	5,3	47	47,42	89	89,51	131	131,130,84,83
6	6,5	48	48,47,21,20	90	90,89,72,71	132	132,103
7	7,6	49	49,40	91	91,90,8,7	133	133,132,82,81
8	8,6,5,4	50	50,49,24,23	92	92,91,80,79	134	134,77
9	9,5	51	51,50,36,35	93	93,91	135	135,124
10	10,7	52	52,49	94	94,73	136	136,135,11,10
11	11,9	53	53,52,38,37	95	95,84	137	137,116
12	12,6,4,1	54	54,53,18,17	96	96,94,49,47	138	138,137,131,130
13	13,4,3,1	55	55,31	97	97,91	139	139,136,134,131
14	14,5,3,1	56	56,55,35,34	98	98,87	140	140,111
15	15,14	57	57,50	99	99,97,54,52	141	141,140,110,109
16	16,15,13,4	58	58,39	100	100,63	142	142,121
17	17,14	59	59,58,38,37	101	101,100,95,94	143	143,142,123,122
18	18,11	60	60,59	102	102,101,36,35	144	144,143,75,74
19	19,6,2,1	61	61,60,46,45	103	103,94	145	145,93
20	20,17	62	62,61,6,5	104	104,103,94,93	146	146,145,87,86
21	21,19	63	63,62	105	105,89	147	147,146,110,109
22	22,21	64	64,63,61,60	106	106,91	148	148,121

LFSR initialization

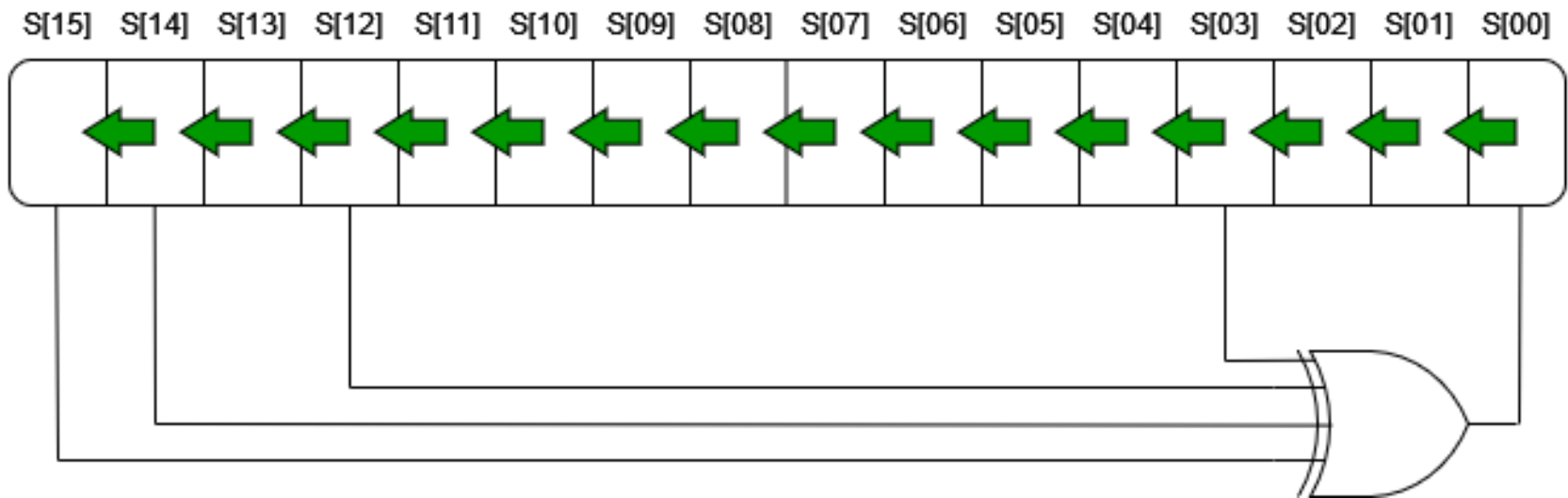
- This simplified process is also described in the document

An n-bit LFSR counter can have a maximum sequence length of $2^n - 1$. In that case, it goes through all possible code permutations except one, which would be a lock-up state. A maximum length n-bit LFSR counter consists of an n-bit shift register with an XNOR in the feedback path from the last output Q_n to the first input D_1 . The XNOR makes the lock-up state the all-ones state; an XOR would make it the all-zeros state. For normal Xilinx applications, all-ones is more easily avoided, since “by default” the flip-flops wake up in the all-zeros state. Table 3 describes the outputs that must be used as inputs of the XNOR. LFSR outputs are traditionally labeled 1 through n, with 1 being the first stage of the shift register, and n being the last stage. This is different from the conventional 0 to (n-1) notation for binary counters. A multi-input XNOR is also known as an even-parity circuit. Note that the connections described in this table are not necessarily unique; certain other connections may also result in maximum length sequences.

- The document suggests using an XNOR to avoid lock-up since most systems initialize in an all 0s state. We won't intentionally initialize with all 0s so XOR implementation works well for us

LFSR initialization

- 16 bit example with the maximal taps given by the Xilinx document

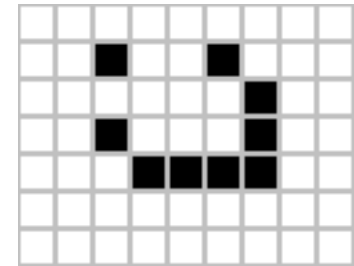


LFSR initialization

- You will need a LFSR module with a small bit size (8, 16) which can be used to prove that it is a maximal implementation which produces $2^n - 1$ values before repeating. We have provided a python script (first_repeat.py) to help with this.
- You will need to build a 64 bit LFSR from your smaller LFSR module to use to initialize your system
- You will need to be able to display your LFSR cycling through the psuedo random sequences, or load the output to your game and begin your game

Opportunities for XC (boost scores)

- Increase matrix (originally just 8x8)
- A more complicated LFSR which avoids Lock-up
- Add some cool patterns (e.g., gliders, spaceship)
- Mutations/Variations on Life (add extra rules)
- Make code easier to adapt size (matrix grid)
 - 2-dimensional – with color 😊.
- Add music
- Add poster to explain to masses!
- Create a youtube channel on this!



Regardless of extra credit, please get baseline project done first!!

What do you have to do?

- Run testbench on datapath to understand.
- Create a simple FSM to control the game, so that it plays itself indefinitely (or, better yet, stops when you hit a button).
 - You may need to swap grid_evolve to grid every iteration.
 - Run testbench on control! (document why this the case!)
- **Important Note: you cannot use “Oops Days” on your project. Its due May 3, 2024, at midnight (11:59:59 PM).**
 - **If you are late, you will get a 0 with no exceptions except a medical excuse.**
- Verify maximal LFSR with a reasonable bit length (8,16) with the use of a LFSR testbench
- Run testbench on combined datapath/control showing it working.
 - You will need to be able to load from the LFSR for your final implementation
 - You will have to create your own DO and testbench files, but you have lots of examples now.
- Output to a HDMI screen along with some sample Verilog to help.
- Write report on your project.
- Have fun!

Details

- You might finish early. Do something special and be creative.
- The game is simplified, so you can easily add extra features or push the boundaries of CA by adding other items.
- Get the baseline project done first and then work on enhancing the project for extra credit.
 - If you missed some labs or homeworks, adding extra credit is a great way of adding points to your grade.
 - Don't forget the report!

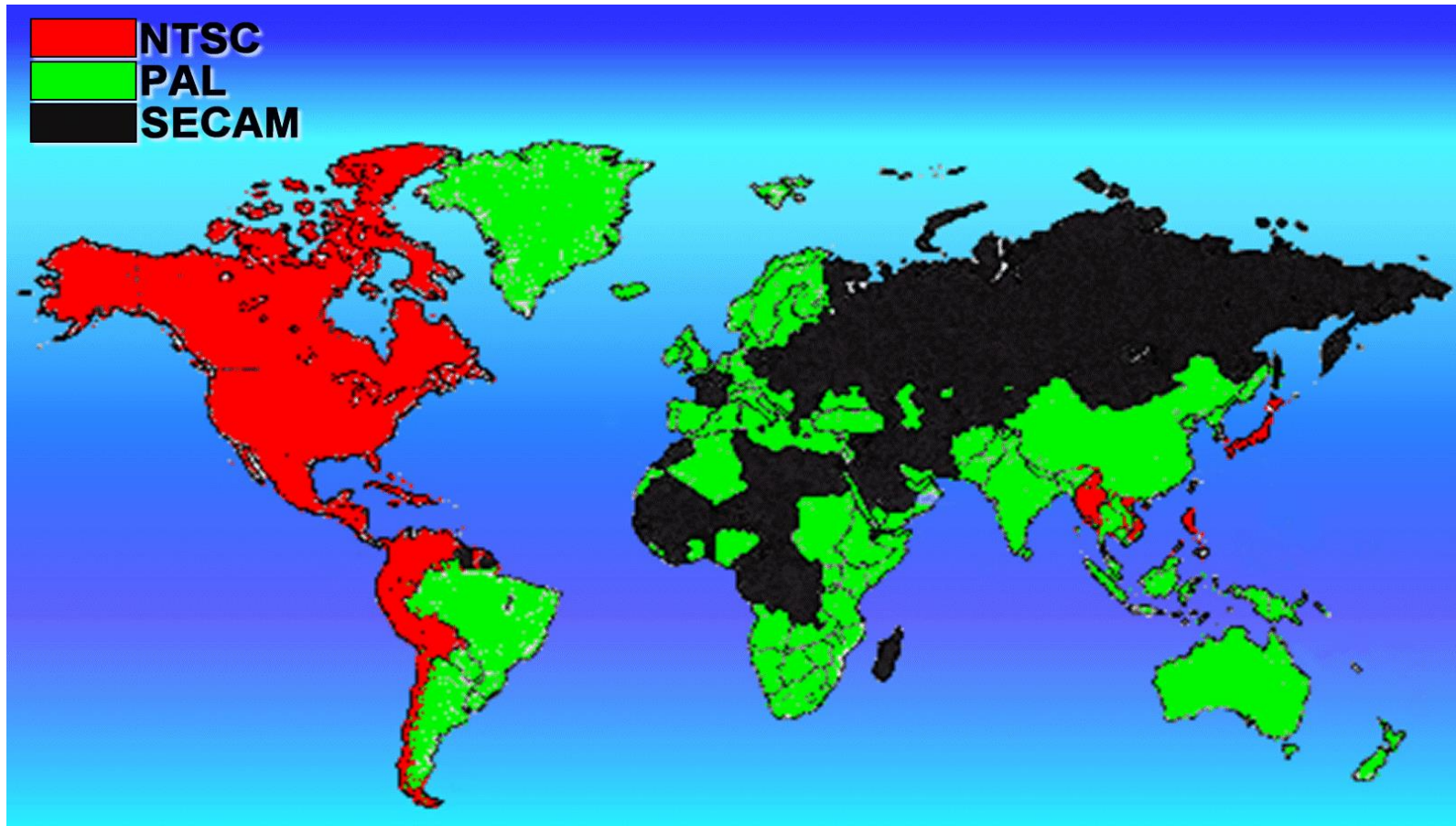
Project Basics

- The project is a significant amount of the grade within the labs.
- Its worth about the same as all your labs, so don't waste time starting.
- Please do not worry too much that the project is worth so many points; just go about your business as usual by showing up every week or doing something every week on the project and you will be fine.
- The report is also important, so don't forget to schedule 2-3 days to write it.
 - We have a writing center on campus that can help with edits and suggestions.
 - <https://osuwritingcenter.okstate.edu/>

Video what ya know!

- You probably already know a couple of things, because video is everywhere including your homes.
- There are several standards:
 - NTSC/PAL
 - SECAM
 - VGA/XGA
- These are the root elements.
 - We will get to HDTV later.

TV and Computer Display Specifications



NTSC

- NTSC was developed in late 1940s and in use for black and white TVs
- PAL (European) and SECAM (French) was soon developed right after.
- Each standard has several things it covers:
 - Number of lines on a screen
 - Refresh rate
 - Maximum color resolution
- Many standards now, but we will focus on these for right now.

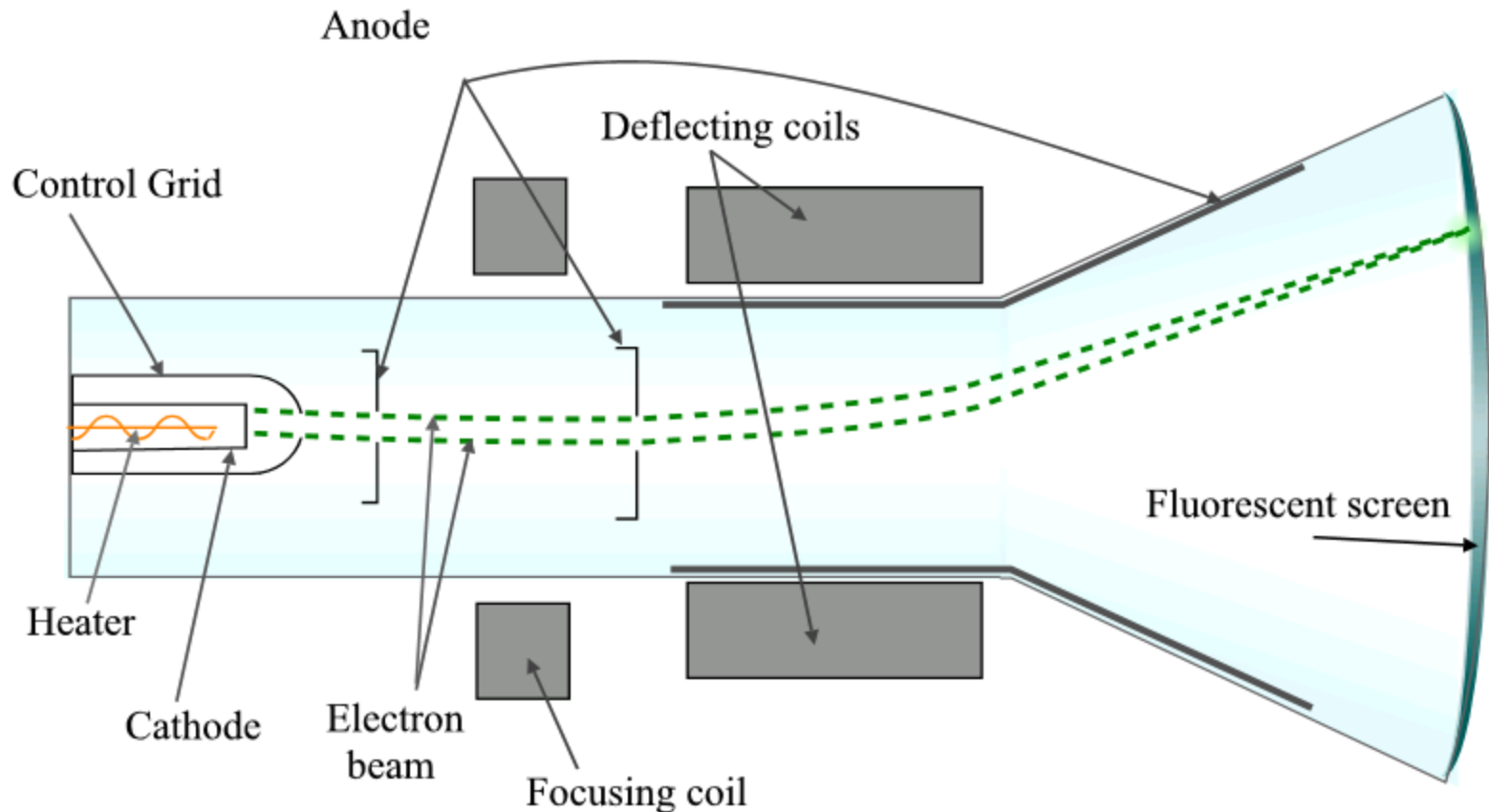
How Do Monitors Work?

- Origin is TV, so let's look at that
 - LCDs work on different principle, but all signaling still derived from TV of 1940s
- Relies on your brain to do two things
 - Integrate over space
 - Integrate over time

Basic Display

- Originally an electron beam was focused on a screen by way of a Cathode Ray Tube (CRT).
- An image or Raster Image is generated by sweeping an electron beam that excites phosphors on the front of the CRT.
- The Raster starts at the upper left, moving left to right, and top to bottom.
- Each Raster consists of several lines (rows) and several pixels or dots per row.
- The number of pixels per row and lines per image make up the picture's resolution.
 - NTSC has 525 lines
 - PAL has 625 lines
 - VGA has 480 lines

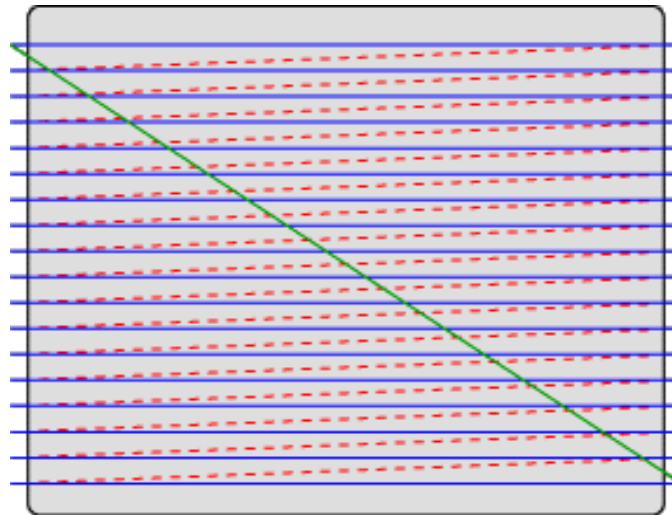
Cathode Ray Tube (CRT)



From wikipedia: http://en.wikipedia.org/wiki/Cathode_ray_tube

Simple Scanning TV

- Electron beam scans across
- Turned off when
 - Scanning back to the left (horizontal retrace ----)
 - Scanning to the top (vertical retrace _____)



©2000 How Stuff Works

Interlacing

- 60 frames/second is a lot of data for one screen.
- Even if the electron beam could move quickly, its still a lot of data.
- So, a clever scheme was invented to give engineers more time to draw things on the screen.
- So, two frames are used to create the image.
- This process is called interlacing.
 - NTSC then has 30 frames/second.

Scanning: Interlaced vs. Progressive

- TVs use *interlacing*
 - Every other scan line is swept per field
 - Two fields per frame (30Hz)
 - Way to make movement less disturbing
- Computers use *progressive scan*
 - Whole frame refreshed at once
 - 60Hz or more, 72Hz looks better
- Similar notation used for HD
 - *i* = interlaced (1080i)
 - *p* = progressive (1080p)
 - which better?

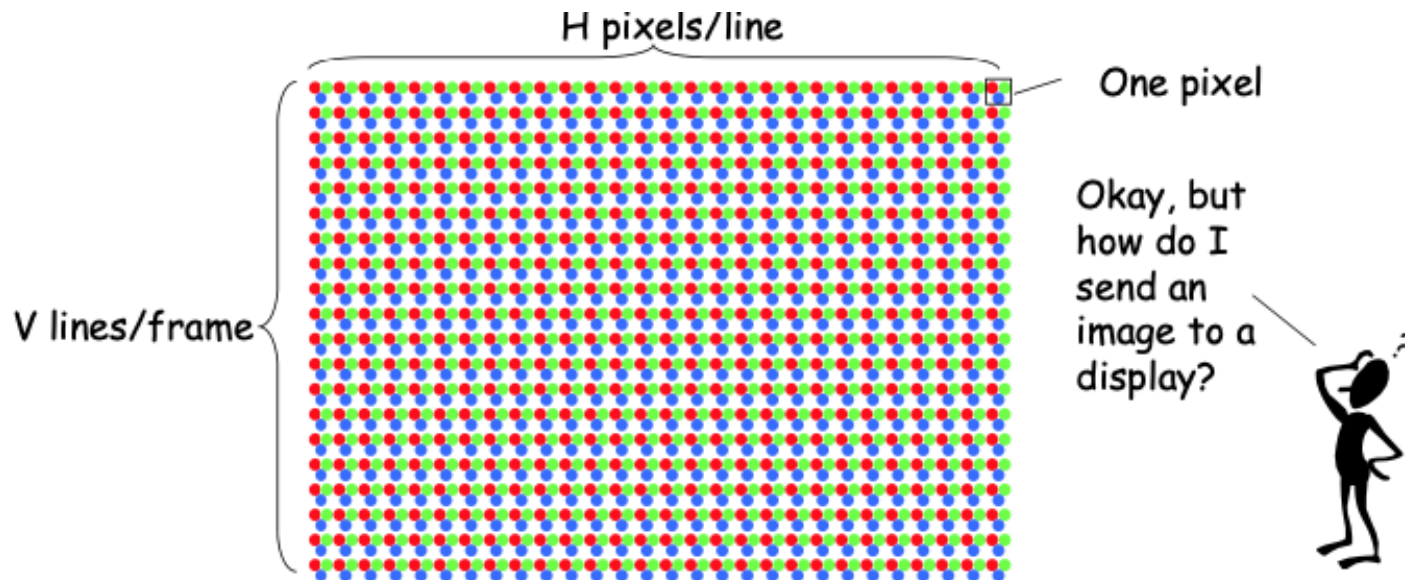
https://en.wikipedia.org/wiki/Interlaced_video

Basics

- Scanning process involves repeated number of steps to control movement of electron gun (still used to control insertion of dopants into semiconductors).
- Image produced is based on number of horizontal lines at end of scanning process.
- Electron gun must reset at bottom of screen and retrace back to upper left of screen and start again.
- Synchronization pulses (like clocks) to indicate these events:
 - One for horizontal axis (hsync)
 - One for vertical axis (vsync)
- Raster Image (NTSC)
 - 525 lines begins with a horizontal sync (hsync)
 - Actual pixel information (actually Amplitude Modulated (AM) signals)
 - Lines that compose image are drawn
 - Image is told to retrace by vertical sync (vsync)

The CRT: Generalized Video Display

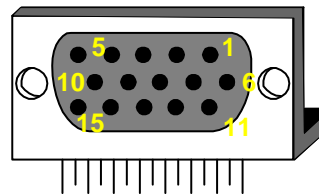
- You should think of a video display as a 2D grid of picture elements (pixels).
- Each pixel is made up of Red, Green and Blue (RGB) emitters.
- The relative intensities of RGB determine the apparent color of a particular pixel.



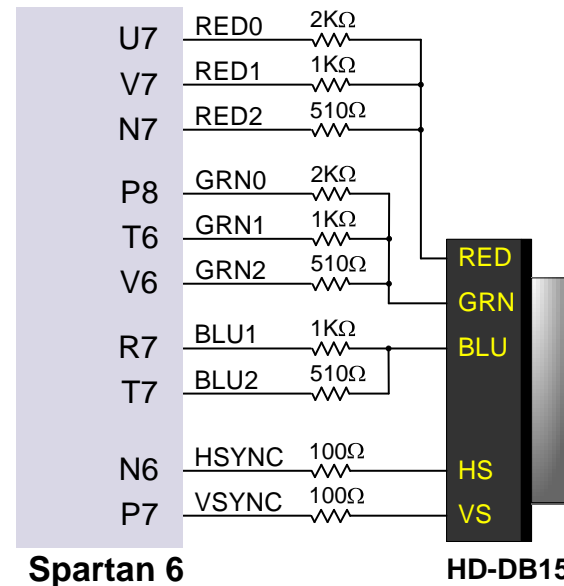
Traditionally $H/V = 4/3$ or with the advent of high-def $16/9$. Lots of choices for H,V and display technologies (CRT, LCD, ...)

Old-School VGA Signaling

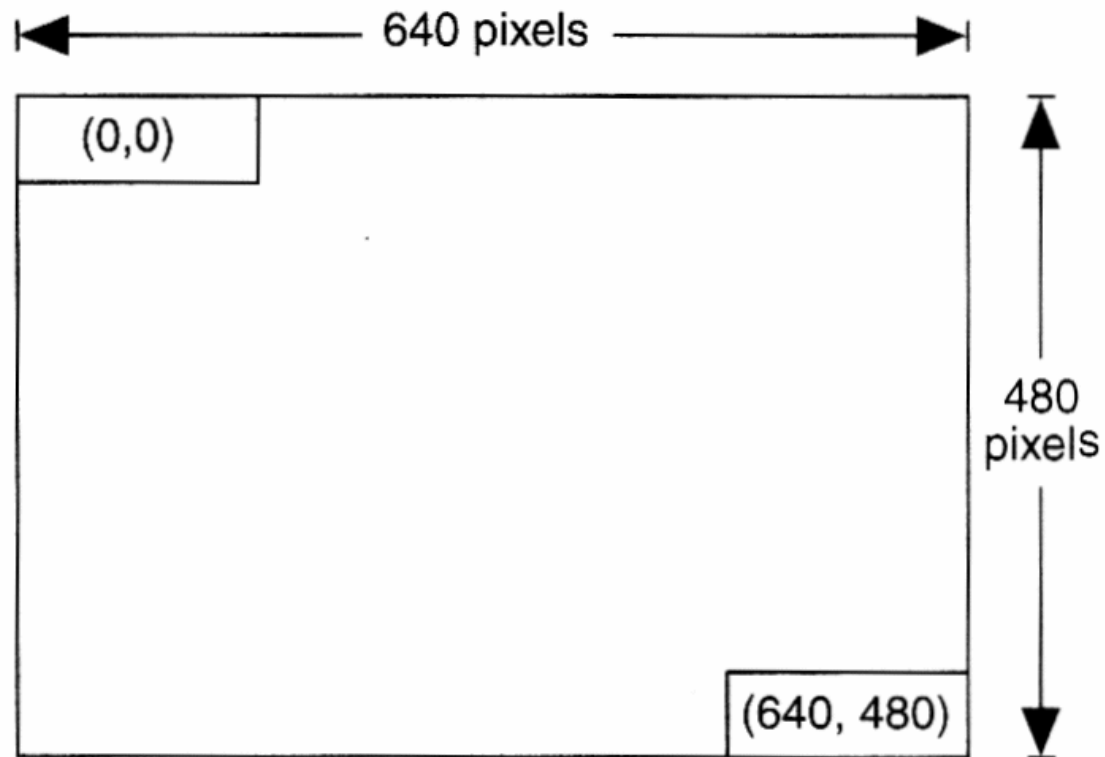
- Timing signals
 - horizontal sync
 - vertical sync
- Color values
 - R, G, B
 - 8-bit value



Pin 1: Red	Pin 5: GND
Pin 2: Grn	Pin 6: Red GND
Pin 3: Blue	Pin 7: Grn GND
Pin 13: HS	Pin 8: Blu GND
Pin 14: VS	Pin 10: Sync GND



VGA Screen (LCD)



VGA basics

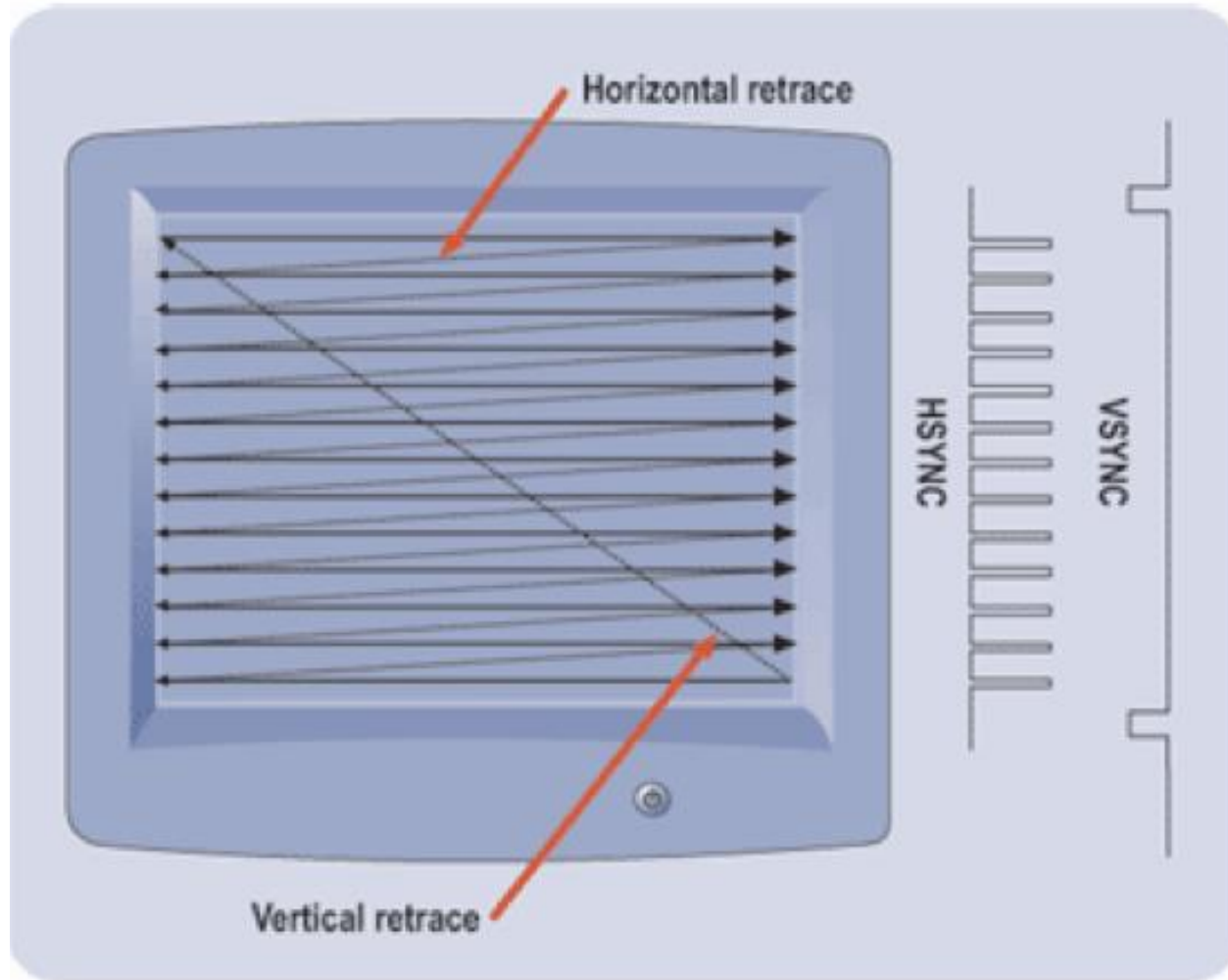
- Same as before but separate RGB signals so don't have to worry about them.
- 640x480 pixels
- Refresh rate of 60 Hz.
- Master timing = 25.175 MHz (39.722 ns)
- Spec is 0.7 V (but 1.0 V seems to work too).
- RGB (0.7 V, 0.7 V, 0.7 V) = color white
- Use a D-subminiature (dsub) 15 connection.



How did I get?

$$f_T = \frac{1}{T}$$

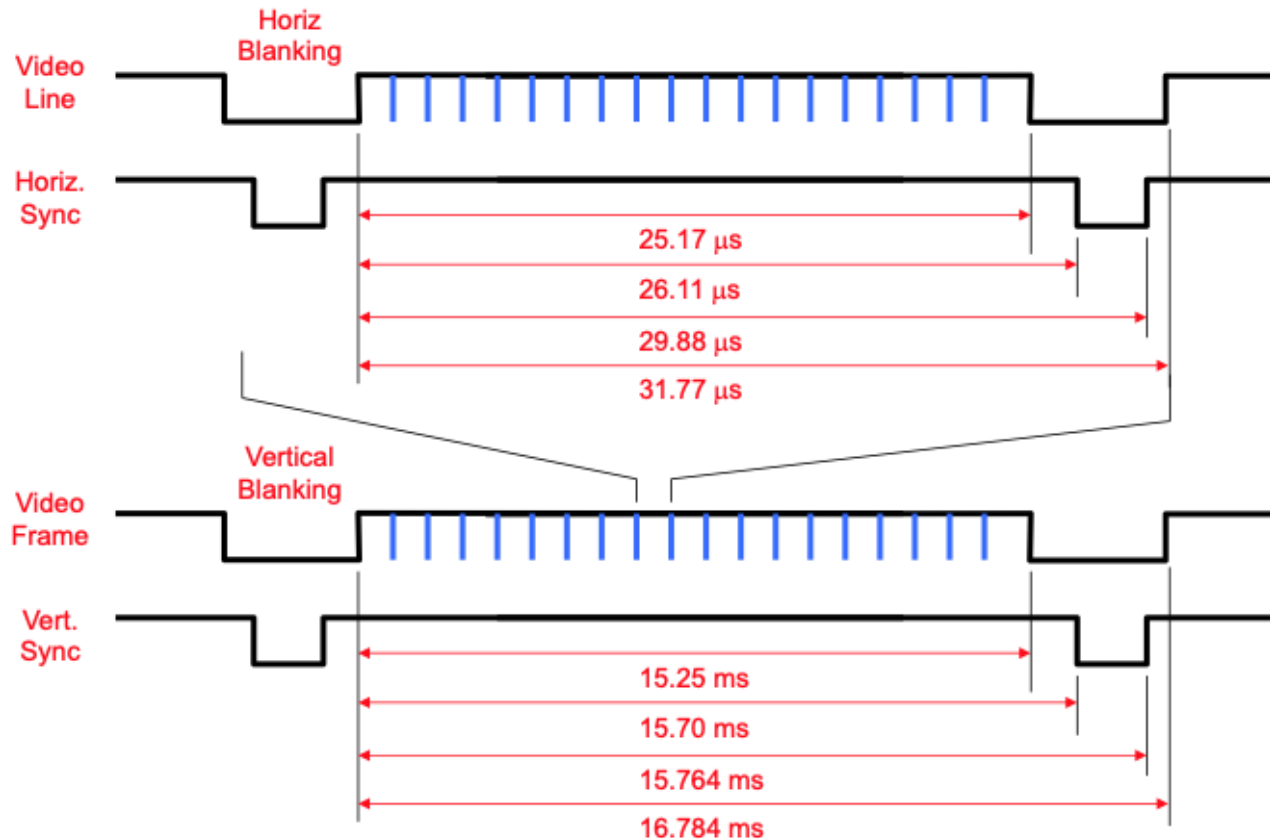
Sync Signals (HS and VS)



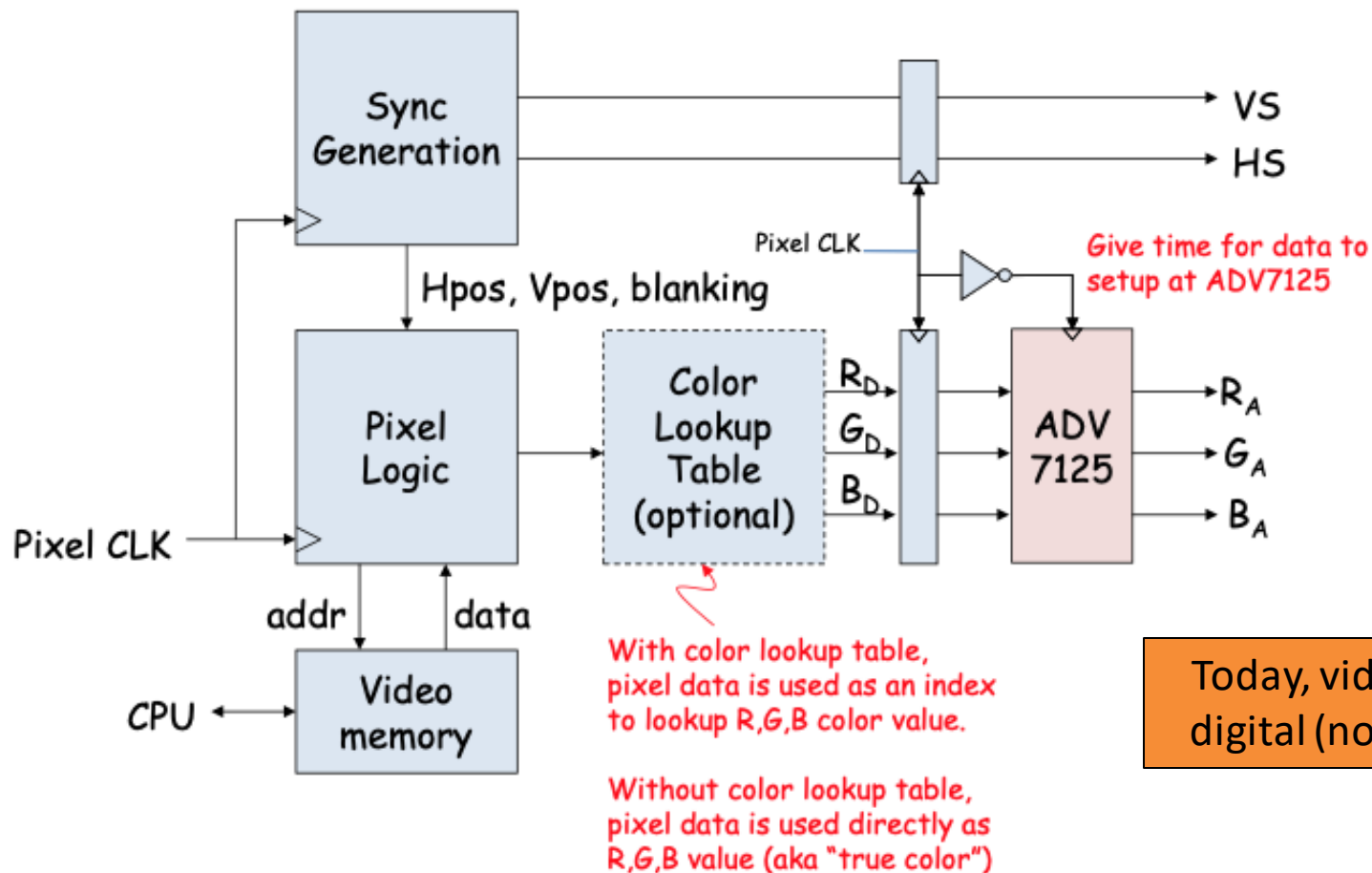
VSYNC/HSYNC are just like clocks reminding the unit when to start a new screen.

VGA (640x480) Video

The most common ways to send an image to a video display (even displays that don't use deflection coils, eg, LCDs) require you to generate two sync signals: one for the horizontal dimension (HS) and one for the vertical dimension (VS).



Process Generating VGA-style Video

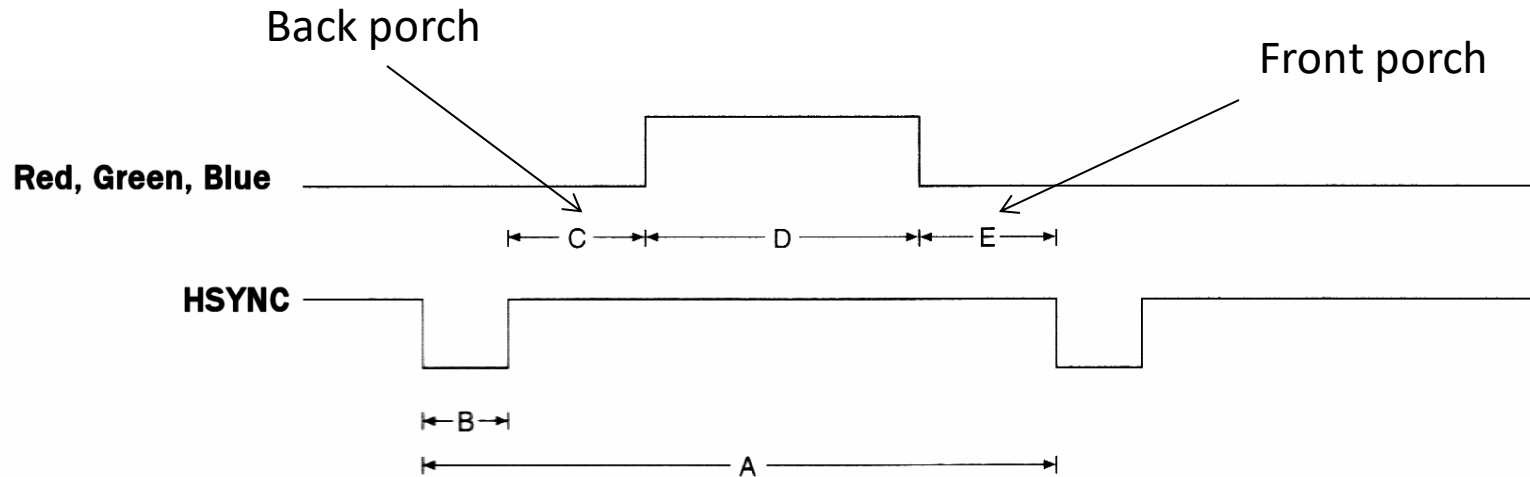


Today, video is just digital (no analog!)

VGA Signals

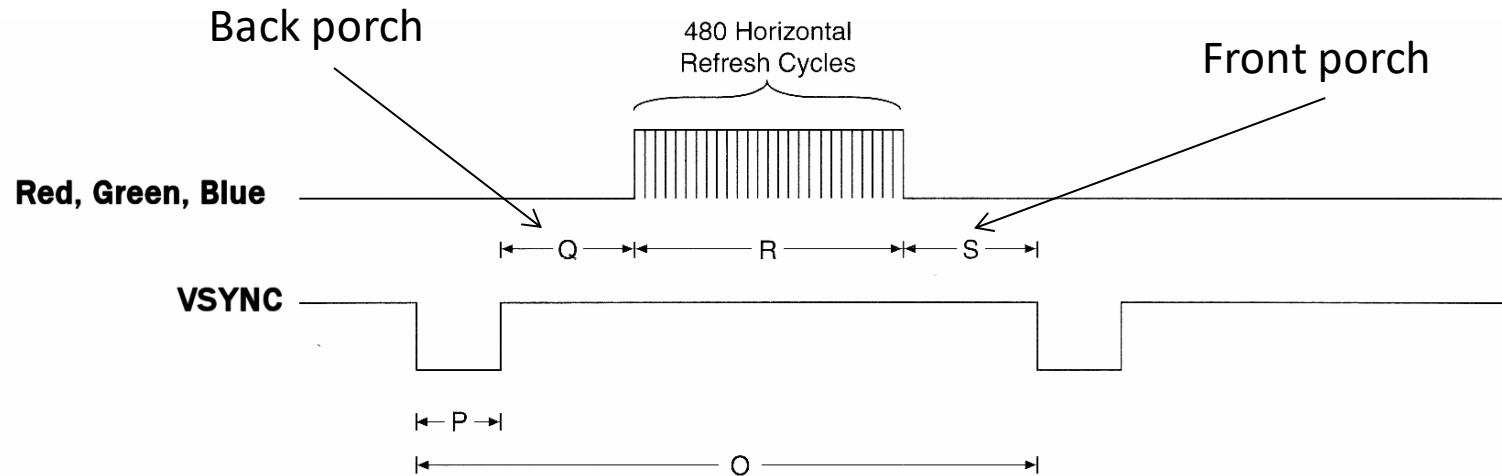
- Each RGB channel must be between 0 and V_{DD} Volts.
- 24-bit color would be 8 bits/channel.
 - Each channel would be 0-255 values.
- Video Digital to Analog Converters or DACs convert signals from digital to analog.
 - Remember VGA is an analog signal.
 - There are special chips with these vDACs or video DACs to handle conversion to db15.

Horizontal Timing



Parameters	A	B	C	D	E
Time	31.77 μ s	3.77 μ s	1.89 μ s	25.17 μ s	0.94 μ s

Vertical Timing



Parameters	O	P	Q	R	S
Time	16.6 ms	64 μ s	1.02 ms	15.25 ms	0.35 ms

Horizontal Timing

Signal	Timing [us]	Clocks
H-sync	3.77	95
Blank	1.57	41
Video	25.49	640
Blank	0.94	24

800 cycles or 31.77 us is non-negotiable!!

C-code

```
// Draw Line Function
// assume pixels are stored in 640 byte array in RGB format
Unsigned char pixel_data[640] = {...};
// render line
for (pixel = 0; pixel <= 799; pixel++) {
    // test for sync
    if (pixel >= 0 && pixel <= 94)
        hsync(on);
    else
        hsync(off);
    // test for blanking periods
    if ((pixel >= 95 && pixel <= 135) || (pixel >= 776 && pixel <= 799)) {
        VGA_video(0); // send black
    }
    if (pixel >= 136 && pixel <= 775) {
        VGA_video(pixel_data[pixel-136]);
    }
}
```

Complete Vertical Frame

Signal	Time [sec]	Lines
V-sync	64 μ	2
Top Blank Lines	1.02 m	32
Active 480	15.24 m	480
Bottom Blank Lines	0.35 m	11

C-code

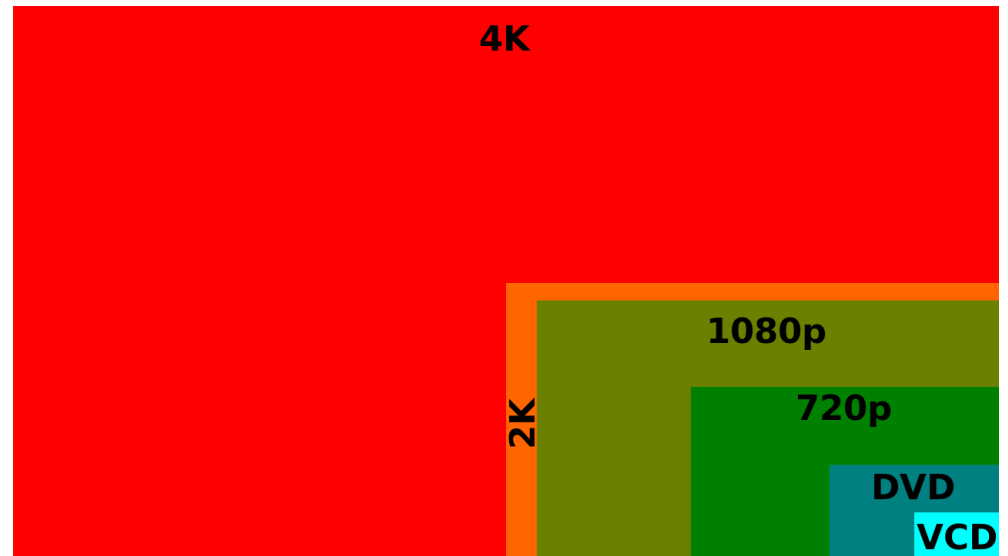
```
// render frame
for (line=0; line <= 524; line++) {
    // test for sync
    if (line >= 0 && line <= 1) {
        Vsync(ON);
    }
    else
        Vsync(OFF);
    // test for blanking periods
    if ((line >= 2 && line <= 33) || (line >= 514 && line <= 524))
        VGA_video(0); // send black
    // test for active video
    if (line >= 34 && line <= 514)
        draw_line();
} // end line
```

HDTV

- HDTV still works the same way.
- Number of lines in the vertical display is called the resolution.
- There are still progressive and interlaced scanning.
- Number of frames/second is still based on power as a historical element.
 - 24p = 24 progressive scan frames/sec
 - 50i = 25 interlaced frames/second.
 - But things are fast enough that it doesn't have to be exact!

4k and 8k Resolution

- Horizontal resolution is 4,000 pixels (sometimes called UHDTV).
 - Really 3840 x 2160
- This is not the end...only the beginning. Currently at 8k (UHD).
- Need faster processors to push out pixels at the correct frequencies for hsync and vsync.



[Wiki]

Constants

```
//##### Module constants from (http://tinyvga.com/vga-timing/640x480@60Hz)
// horizontal display area
parameter HDisplayArea = 640;
// maximum horizontal amount (limit)
parameter HLimit = 800;
// h. front porch
parameter HFrontPorch = 16;
// h. back porch
parameter HBackPorch = 48;
// h. pulse width
parameter HSyncWidth = 96;
// vertical display area
parameter VDisplayArea = 480;
// maximum vertical amount (limit)
parameter VLimit = 525;
// v. front porch
parameter VFrontPorch = 10;
// v. back porch
parameter VBackPorch = 33;
// v. pulse width
parameter VSyncWidth = 2;
```

HDL

```
// simulate the vertical and horizontal positions
always @(posedge CLK_25MHz)
begin
    if (CurHPos < HLimit-1)
        begin
            CurHPos <= CurHPos + 1;
        end
    else
        begin
            CurHPos <= 0;

            if (CurVPos < VLimit-1)
                CurVPos <= CurVPos + 1;
            else
                CurVPos <= 0;
        end
    end
end
```

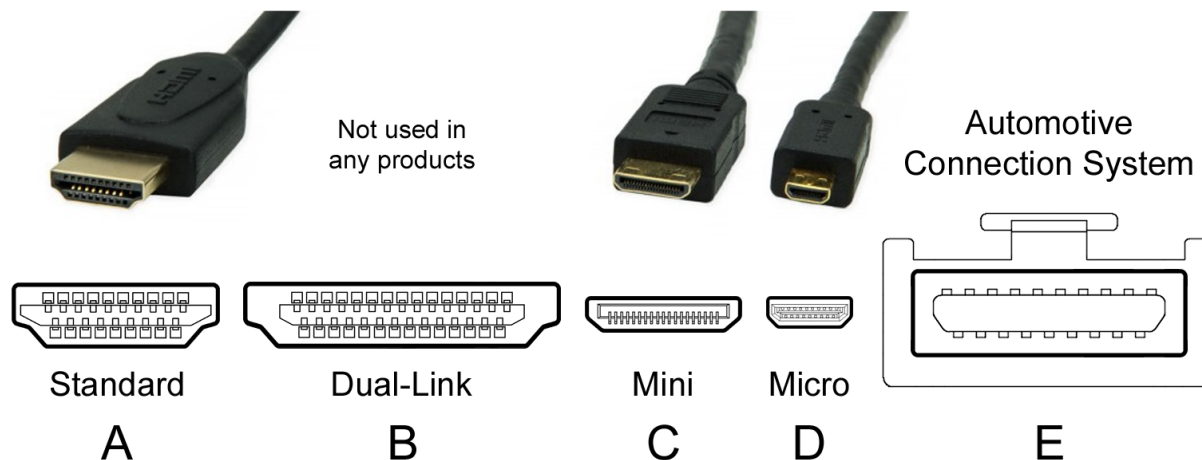

VSYNC/HSYNC HDL

```
// HSync logic
always @(posedge CLK_25MHz)
    if (CurHPos < HSyncWidth)
        HS <= 1;
    else
        HS <= 0;
```

```
// VSync logic
always @(posedge CLK_25MHz)
    if (CurVPos < VSyncWidth)
        VS <= 1;
    else
        VS <= 0;
```

HDMI

- HDMI or High-Definition Multimedia Interface (HDMI) is a proprietary video data for uncompressed video data.
 - It can also handle uncompressed and/or compressed audio data, as well.
- Most systems have HDMI interfaces although newer computers may use mini/micro HDMI.



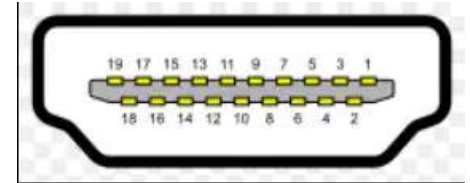
16bits for each color/pixel!

	HDMI version				
	1.0–1.2a	1.3–1.3a	1.4–1.4b	2.0–2.0b	2.1
Release date	Dec 2002 (1.0) ^[129] May 2004 (1.1) Aug 2005 (1.2) ^[130] Dec 2005 (1.2a) ^[131]	Jun 2006 (1.3) ^[132] Nov 2006 (1.3a) ^[5]	Jun 2009 (1.4) ^[133] Mar 2010 (1.4a) ^[109] Oct 2011 (1.4b)	Sep 2013 (2.0) ^[113] Apr 2015 (2.0a) ^[134] Mar 2016 (2.0b)	Nov 2017 ^[135]
Signal specifications					
Max. transmission bit rate (Gbit/s) ^[a]	4.95	10.2	10.2	18.0	48.0
Max. data rate (Gbit/s) ^[b]	3.96	8.16	8.16	14.4	42.0
Max. TMDS character rate (MHz) ^[c]	165 ^{[96]:§3}	340 ^[132]	340	600 ^{[114]:§6.1.1}	—
Data channels	3	3	3	3	4
Encoding scheme ^[d]	TMDS ^{[96]:§5.1}	TMDS	TMDS	TMDS	16b/18b ^[128]
Encoding efficiency	80%	80%	80%	80%	88.8%
Compression	—	—	—	—	DSC 1.2 (optional) ^[136]
Color format support					
RGB	Yes ^{[96]:§6.2.3}	Yes	Yes	Yes	Yes
Y′C _B C _R 4:4:4	Yes ^{[96]:§6.2.3}	Yes	Yes	Yes	Yes
Y′C _B C _R 4:2:2	Yes ^{[96]:§6.2.3}	Yes	Yes	Yes	Yes
Y′C _B C _R 4:2:0	No	No	No ^[e]	Yes ^{[114]:§7.1}	Yes
Color depth support					
8 bpc (24 bit/px)	Yes ^{[96]:§3}	Yes	Yes	Yes	Yes
10 bpc (30 bit/px)	Yes ^[f]	Yes	Yes	Yes	Yes
12 bpc (36 bit/px)	Yes ^[f]	Yes	Yes	Yes	Yes
16 bpc (48 bit/px)	No	Yes ^{[5]:§6.5}	Yes	Yes	Yes

[Wikipedia]

HDMI Pinout

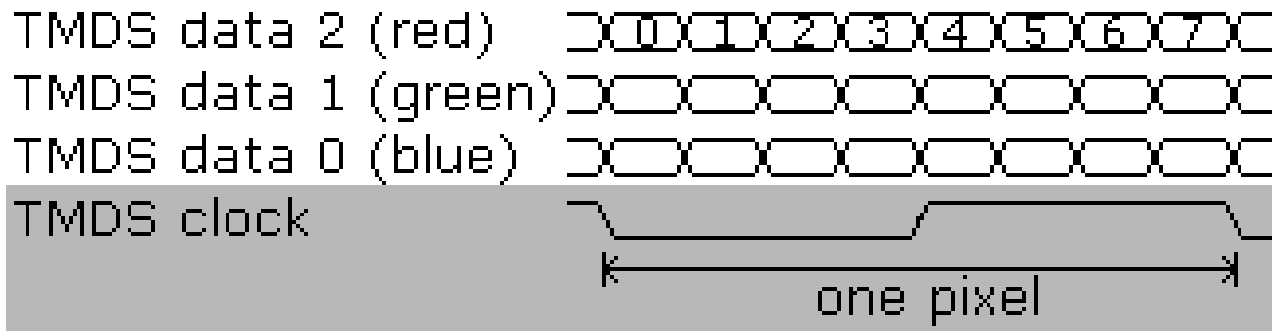
- The standard type A connector is different than VGA.
 - Pins 1-3 carry Data channel 2 (+/ground or data shield/- respectively)
 - Pins 4-6 carry Data channel 1 (+/ground or data shield/- respectively)
 - Pins 7-9 carry Data channel 0 (+/ground or data shield/- respectively)
 - Pins 10-12 carry the clock channel that synchronizes the signals (+/ground or data shield/- respectively)
 - Pin 13 is a CEC channel allowing devices to control each other
 - Pin 14 has no current use
 - Pins 15-16 carry the Display Data Channel (DDC) which communicates extended display ID information
 - Pin 17 is the data shield for pins 13, 15, and 16
 - Pin 18 is a low-voltage power supply (5v)
 - Pin 19 is the Hot Plug Detect, and it monitors power and plug/unplug events



- HDMI is a digital technology so doesn't suffer from analog issues.
- There is also something called a Display Port which doesn't have the CEC controls.

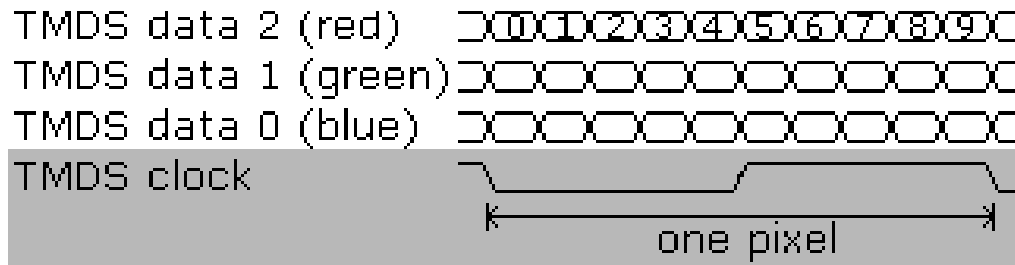
TMDS Signals

- Let's create a 640x480 RGB 24bpp @ 60Hz video signal.
 - That's 307,200 pixels per frame, and since each pixel has 24 bits (8 bits for red, green and blue), at 60Hz, the HDMI link transports 0.44Gbps of "useful" data.
- The FPGA has 4 TMDS differential pairs to drive.
- First, the TMDS clock is simply the pixel clock, so it runs at 25MHz.
- The other 3 pairs transmit the red, green and blue 8bit signals, so we get something like this.



Differential?

- Why differential?
 - Differential lines are used (also in USB) to handle noise and allow better signal transmission.
- Things are in fact just a bit more complicated. HDMI requires that we scramble the data and add 2 bits per color lane, so we have 10 bits instead of 8 and the link ends up transporting 30 bits per pixel.
- The scrambling and extra bits are needed by the HDMI receiver to properly synchronize to and acquire each lane (make sure to check the DVI and HDMI specifications for more details).



Video Generation 800x525

```
logic [9:0] CounterX; // counts from 0 to 799
always @(posedge pixclk)
    CounterX <= (CounterX==799) ? 0 : CounterX+1;
logic [9:0] CounterY; // counts from 0 to 524
always @(posedge pixclk)
    If (CounterX==799)
        CounterY <= (CounterY==524) ? 0 : CounterY+1;
```

Create H-sync/V-sync

```
logic hSync = (CounterX>=656) && (CounterX<752);  
logic vSync = (CounterY>=490) && (CounterY<492);  
logic DrawArea = (CounterX<640) && (CounterY<480);
```


Generate 10 bits + TMDS

```
logic [9:0] TMDS_red, TMDS_green, TMDS_blue;
TMDS_encoder encode_R(.clk(pixclk), .VD(red), .TMDS(TMDS_red),
    .CD(2'b00), .VDE(DrawArea));
TMDS_encoder encode_G(.clk(pixclk), .VD(green),
    .TMDS(TMDS_green), .CD(2'b00), .VDE(DrawArea));
TMDS_encoder encode_B(.clk(pixclk), .VD(blue), .TMDS(TMDS_blue)
    , .CD({vSync,hSync}), .VDE(DrawArea));
```

Now, we have three 10 bits values to be sent for every pixel clock period. We multiply the 25MHz clock by 10 to generate a 250MHz clock...

```
logic clk_TMDS, DCM_TMDS_CLKFX;
DCM_SP #(.CLKFX_MULTIPLY(10)) DCM_TMDS_inst(.CLKIN(pixclk),
    .CLKFX(DCM_TMDS_CLKFX), .RST(1'b0));
BUFG BUFG_TMDSp(.I(DCM_TMDS_CLKFX), .O(clk_TMDS)); // 250 MHz
```

Provided HDMI Code

- You will be provided sample HDMI code.
- Your job is to output a 1 or 0 for each 8x8 Conway's Game of Life matrix.
- Again, you need to slow down the generation of the datapath.
- The HDMI should just be output as a position for each 1.
 - I would make the 1 a set size (e.g., 40 pixels).