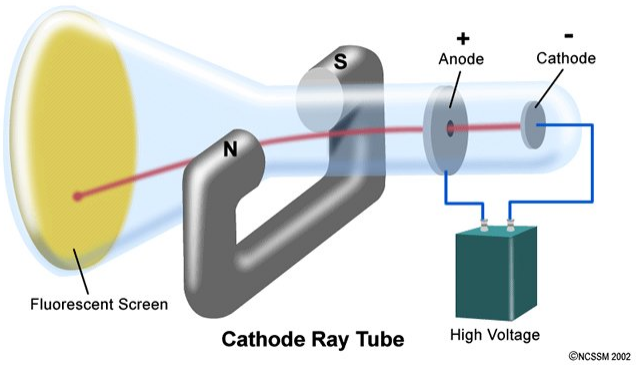
Display Technology

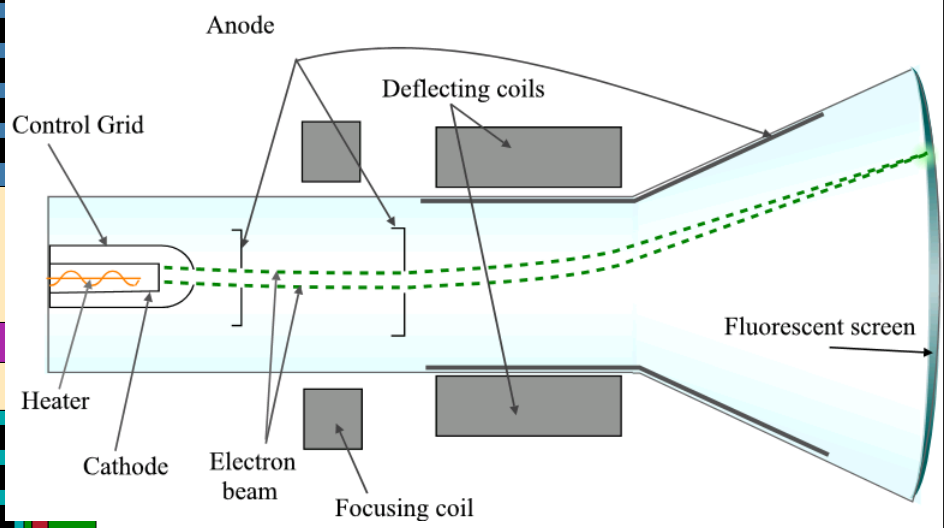
- Images stolen from various locations on the web...

Cathode Ray Tube

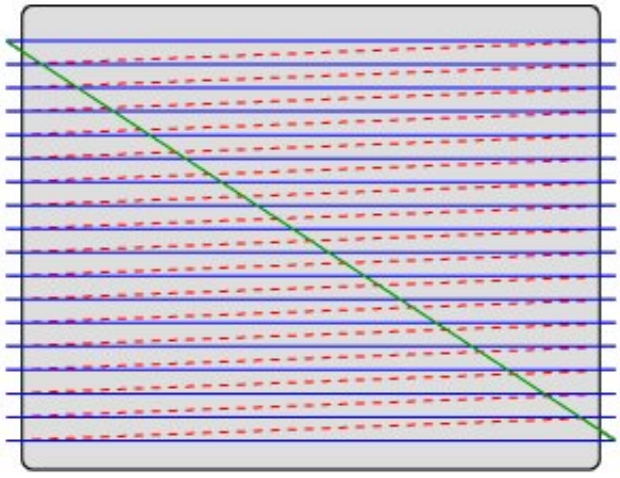


Cathode Ray Tube

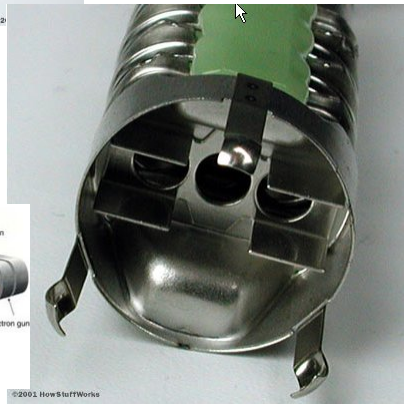
Cathode Ray Tube

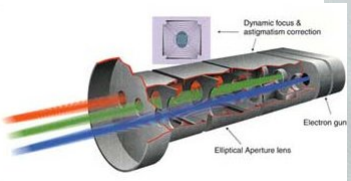


Raster Scanning



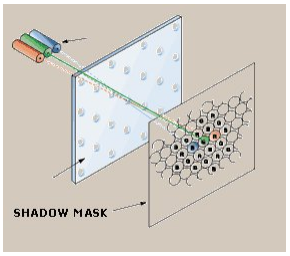
Electron Gun

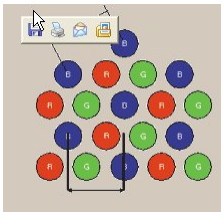


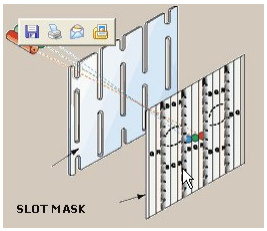


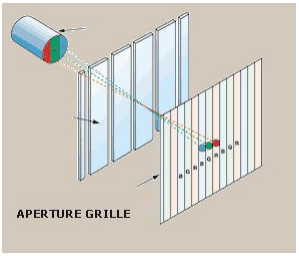
Beam Steering Coils

Color

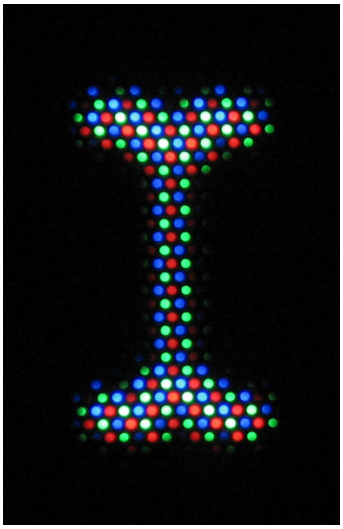


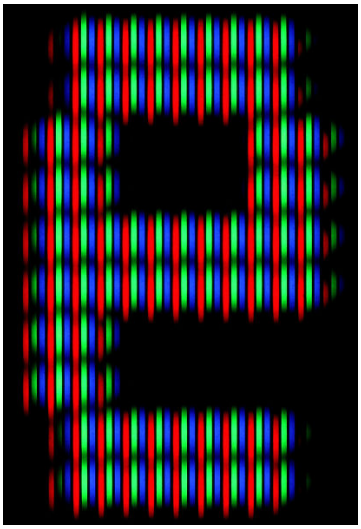




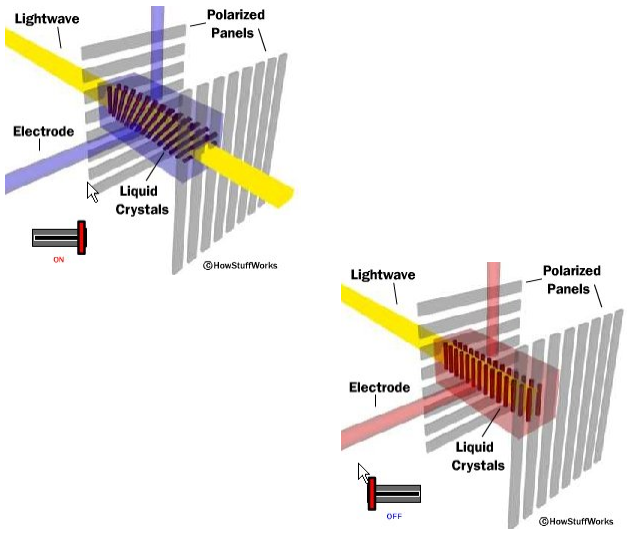


Shadow Mask and Aperture Grille

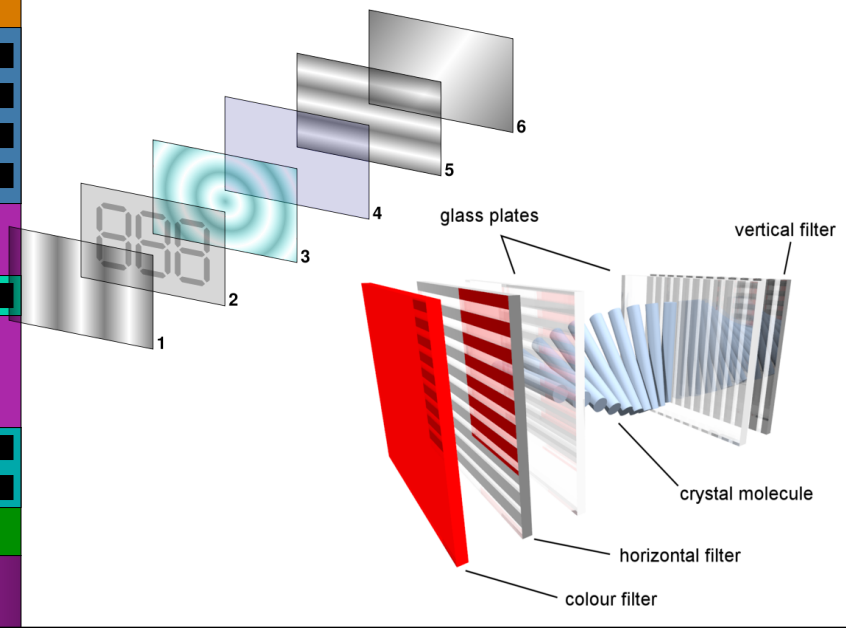




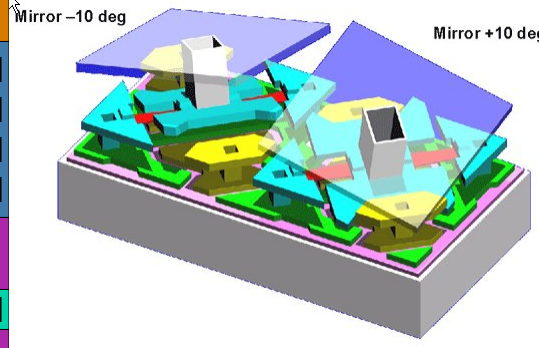
Liquid Crystal Displays

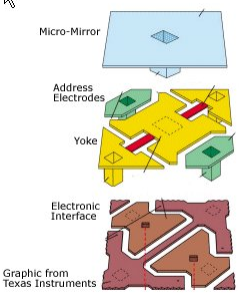


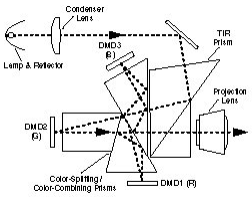
Liquid Crystal Displays



DLP Projector

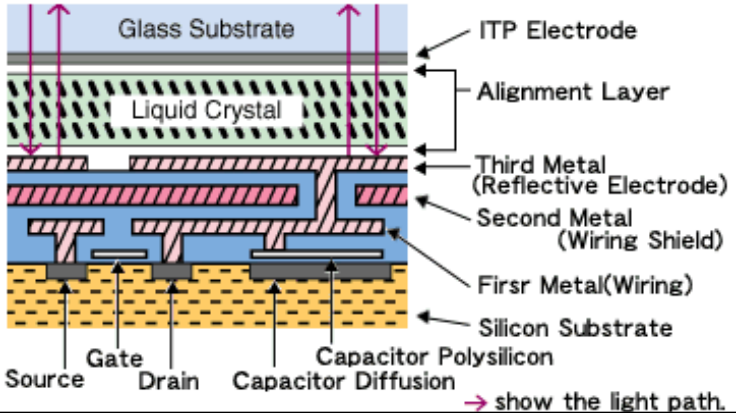




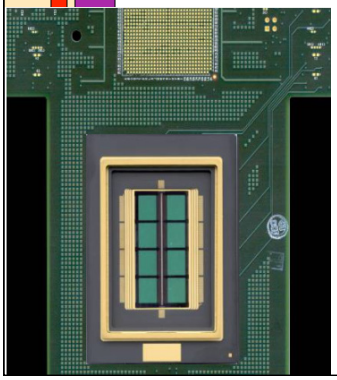


LCoS

- Liquid Crystal on Silicon
  - Put a liquid crystal between a reflective layer on a silicon chip

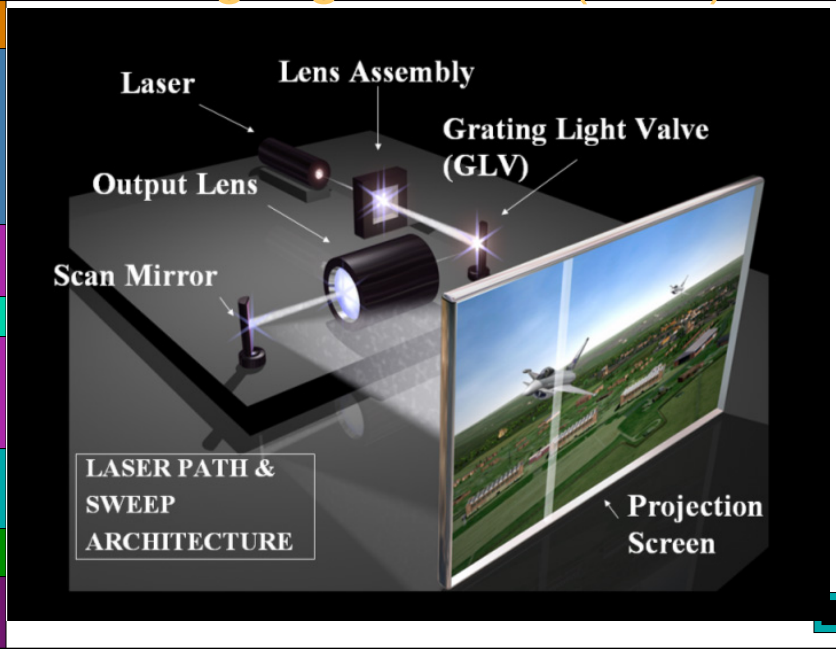


Grating Light Valve (GLS)

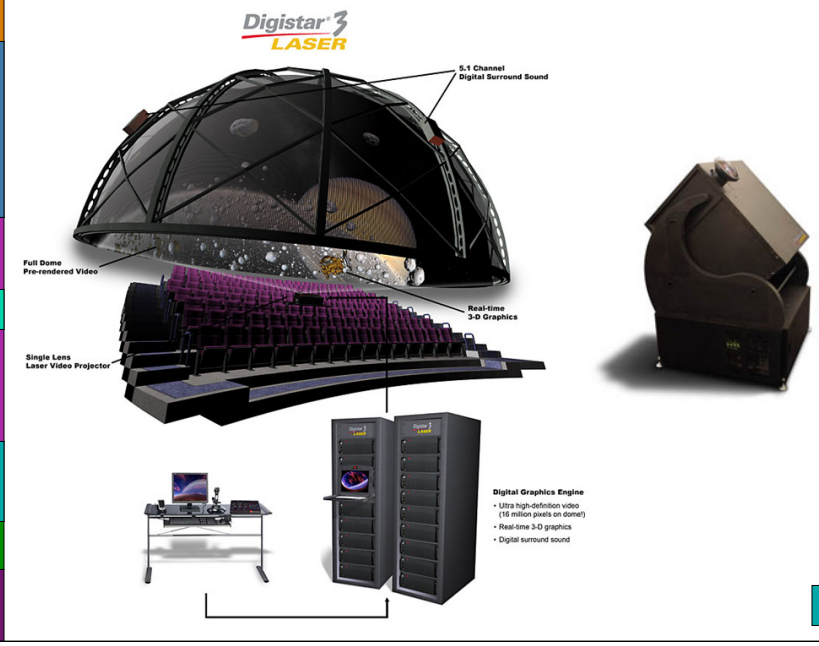


- lots (8000 currently) of micro ribbons that can bend slightly
  - Make them reflective
  - The bends make a diffraction grating that controls how much light goes where
  - Scan it with a laser for high light output
  - 4000 pixel wide frame at 60Hz

Grating Light Valve (GLS)



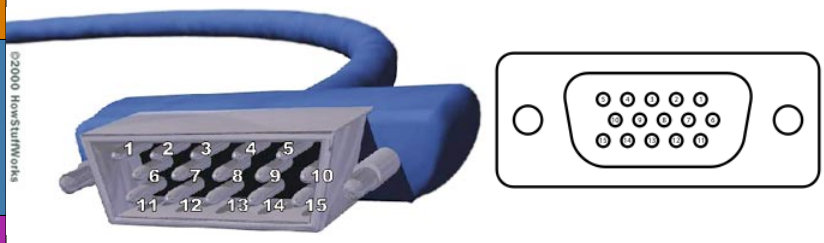
Digistar 3 Dome Projector



VGA

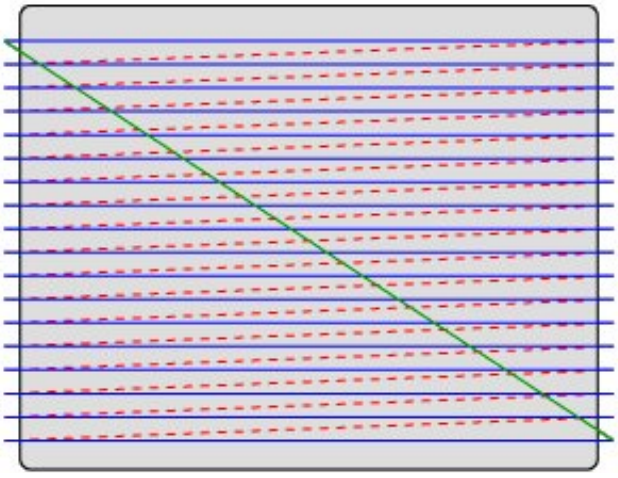
- Stands for Video Graphics Array
- A standard defined by IBM back in 1987
  - 640 x 480 pixels
  - Now superseded by much higher resolution standards...
- Also means a specific analog connector
  - 15-pin D-subminiature VGA connector

VGA Connector

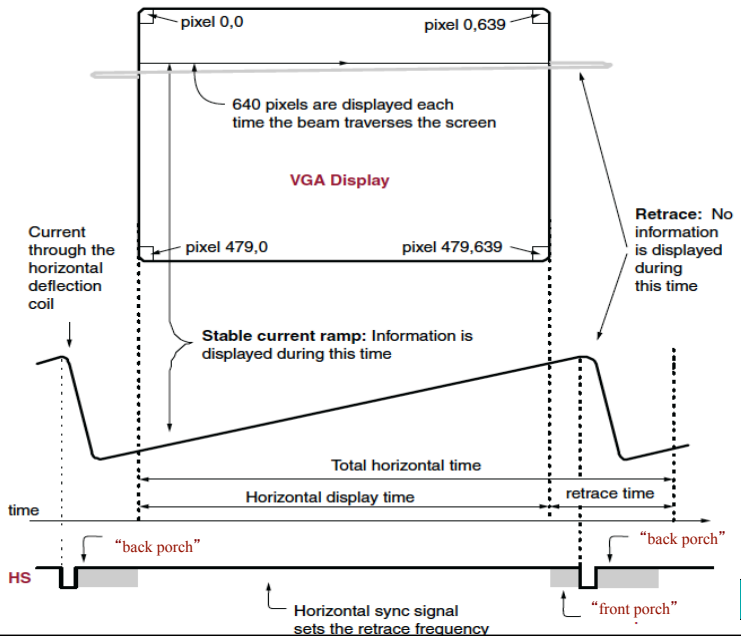


1: Red out	**6**: Red return (ground)	**11**: Monitor ID 0 in
2: Green out	**7**: Green return (ground)	**12**: Monitor ID 1 in or data from display
3: Blue out	**8**: Blue return (ground)	**13**: Horizontal Sync
4: Unused	**9**: Unused	**14**: Vertical Sync
5: Ground	**10**: Sync return (ground)	**15**: Monitor ID 3 in or data clock

Raster Scanning

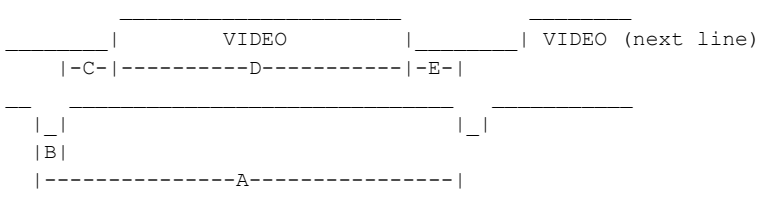


Raster Scanning



VGA Horizontal Timing

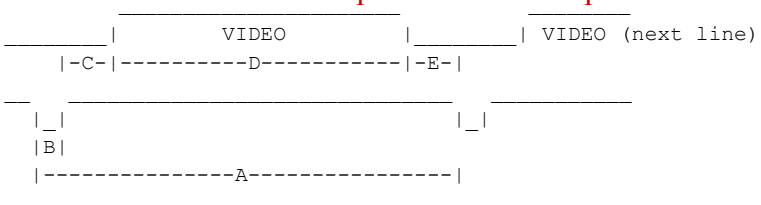
Horizonal Dots	640	
Vertical Scan Lines	480	60Hz vertical frequency
Horiz. Sync Polarity	NEG	
A (μs)	31.77	Scanline time
B (μs)	3.77	Sync pulse length
C (μs)	1.89	Back porch
D (μs)	25.17	Active video time
E (μs)	0.94	Front porch



VGA Horizontal Timing

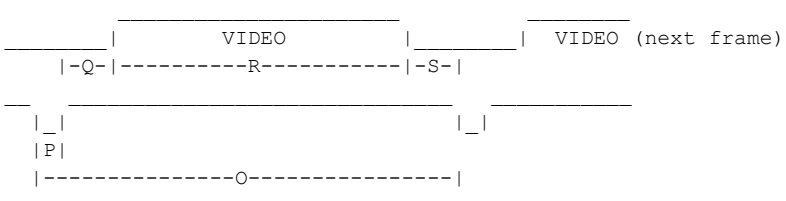
Horizonal Dots	640	
Vertical Scan Lines	480	60Hz vertical frequency
Horiz. Sync Polarity	NEG	
A (µs)	31.77	Scanline time
B (µs)	3.77	Sync pulse length
C (µs)	1.89	Back porch
D (µs)	25.17	Active video time
E (µs)	0.94	Front porch

25.17/640 = 39.33ns/pixel = 25.4MHz pixel clock



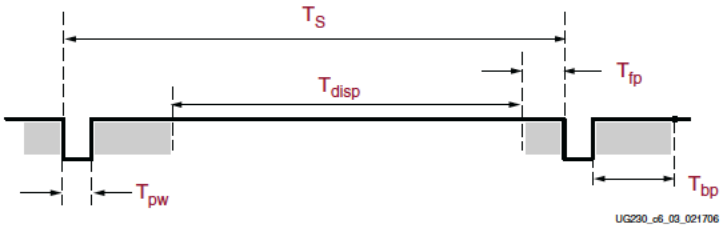
VGA Vertical Timing

Horizonal Dots	640	
Vertical Scan Lines	480	
Vert. Sync Polarity	NEG	
Vertical Frequency	60Hz	
O (ms)	16.68	Total frame time
P (ms)	0.06	Sync pulse length
Q (ms)	1.02	Back porch
R (ms)	15.25	Active video time
S (ms)	0.35	Front porch



VGA Timing Summary

Symbol	Parameter	Vertical Sync			Horizontal Sync	
		Time	Clocks	Lines	Time	Clocks
T_S	Sync pulse time	16.7 ms	416,800	521	32 µs	800
T_{DISP}	Display time	15.36 ms	384,000	480	25.6 µs	640
T_{PW}	Pulse width	64 µs	1,600	2	3.84 µs	96
T_{FP}	Front porch	320 µs	8,000	10	640 ns	16
T_{BP}	Back porch	928 µs	23,200	29	1.92 µs	48



60 Hz refresh and 25MHz pixel clock

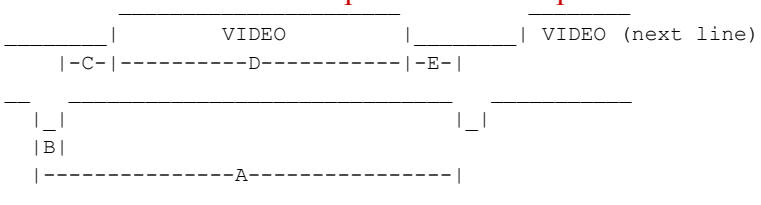
Relaxed VGA Timing

- This all sounds pretty strict and exact...
- It's not really... The only things a VGA monitor really cares about are:
  - Hsync
  - Vsync
  - Actually, all it cares about is the falling edge of those pulses!
  - The beam will retrace whenever you tell it to
  - It's up to you to make sure that the video signal is 0v when you are not painting (i.e. retracing)

Relaxed VGA Horizontal Timing

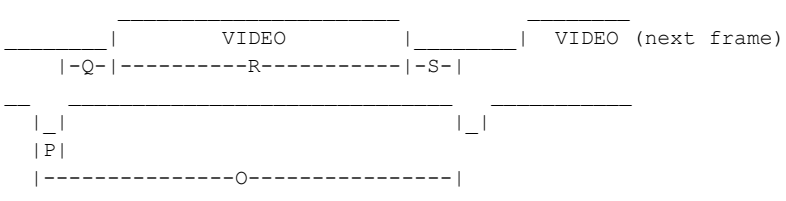
Horizonal Dots	128	
Vertical Scan Lines	?	60Hz vertical frequency
Horiz. Sync Polarity	NEG	
A (µs)	30.0	Scanline time
B (µs)	2.0	Sync pulse length
C (µs)	10.7	Back porch
D (µs)	12.8	Active video time
E (µs)	4.50	Front porch

12.8/128 = 100ns/pixel = 10 MHz pixel clock



VGA Relaxed Vertical Timing

Horizonal Dots	128	
Vertical Scan Lines	255	
Vert. Sync Polarity	NEG	
Vertical Frequency	60Hz	
O (ms)	16.68	Total frame time
P (ms)	0.09	Sync pulse length (3x30µs)
Q (ms)	4.86	Back porch
R (ms)	7.65	Active video time
S (ms)	4.08	Front porch



VGA on Spartan3e Starter

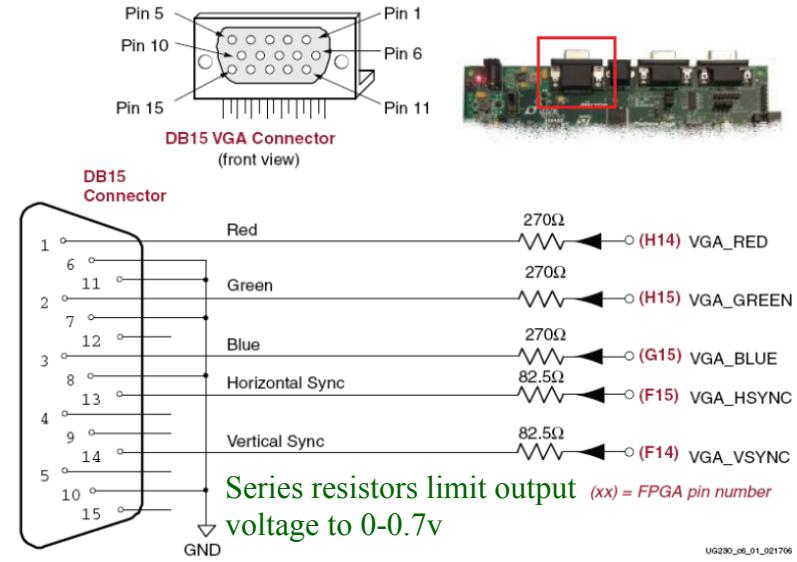


Figure 6-1: **VGA Connections from Spartan-3E Starter Kit Board**

VGA Voltage Levels

- Voltages on R, G, and B determine the color
  - Analog range from 0v (off) to +0.7v (on)
  - But, our pads produce 0-5v outputs!

VGA Voltage Levels

- Voltages on R, G, and B determine the color
  - Analog range from 0v (off) to +0.7v (on)
  - But, our pads produce 0-5v outputs!
  - For B&W output, just tie RGB together and let 0v=black and 5v=white
    - This overdrives the input amps, but won't really hurt anything
  - For color you can drive R, G, B separately
    - Of course, this is only 8 colors (including black and white)
    - Requires storing three bits at each pixel location

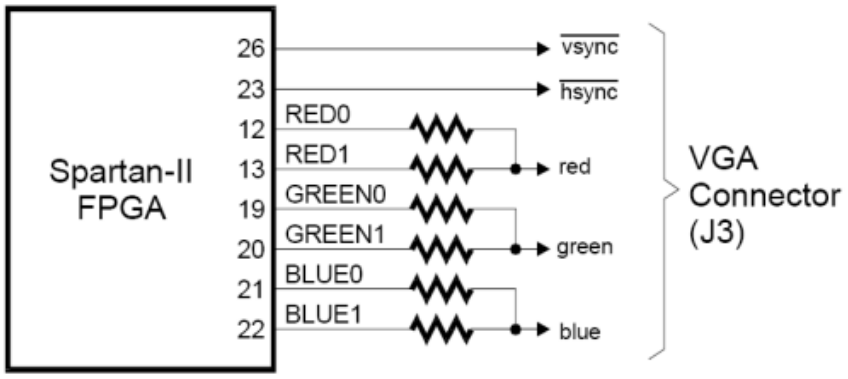
VGA on Spartan3e Starter

Table 6-1: **3-Bit Display Color Codes**

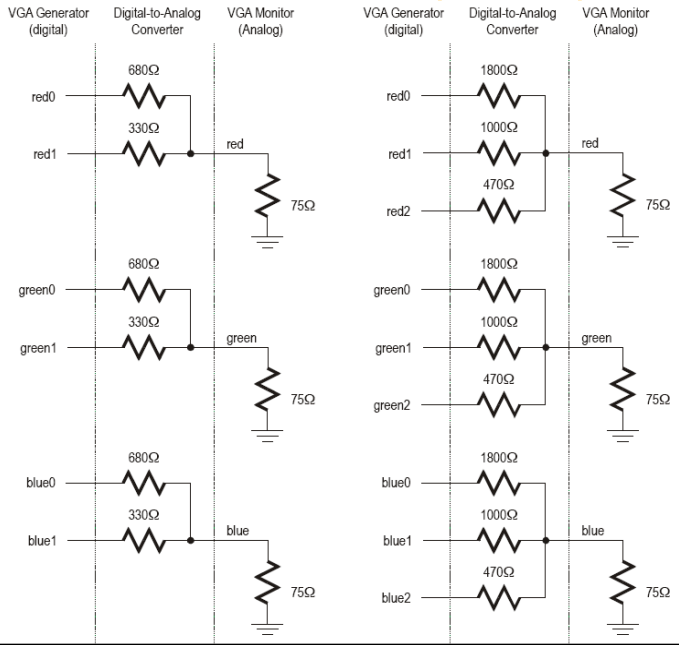
VGA_RED	VGA_GREEN	VGA_BLUE	Resulting Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

More colors

- More colors means more bits stored per pixel
- Also means D/A conversion to 0 to 0.7v range



More Colors (Xess)



What to Display?

- You need data to display on the screen...
  - Brute force: put it all in a giant ram that has the same resolution as your screen and just walk through the RAM as you paint the screen
  - More clever: Fill a row buffer with data for a scan line
  - Multi-level: Fill a (smaller) row buffer with pointers to glyphs that are stored in another RAM/ROM
- Just keep track of where the beam is and where your data is...

VGA Breakdown

- vgaControl
  - Generate timing pulses at the right time
  - hSync, vSync, bright, hCount, vCount
- bitGen
  - Based on bright, hCount, vCount, turn on the bits

3 Types of bitGen

- Bitmapped
- Character/Glyph – based
- Hard-coded

3 Types of bitGen

- Bitmapped
  - Frame buffer holds a separate rgb color for every pixel
  - bitGen just grabs the pixel based on hCount and vCount and splats it to the screen
  - Chews up a LOT of memory
  - This memory would have to be off-chip…

3 Types of bitGen

- Character/Glyph-based
  - Break screen into nxm pixel chunks (e.g. 8x8)
  - For each chunk, point to one of k nxm glyphs
  - Those glyphs are stored in a separate memory
  - For 8x8 case (for example)
    - glyph number is hCount and vCount minus the low three bits
    - glyph bits are the low-order 3 bits in each of hCount and vCount
    - Figure out which screen chunk you're in, then reference the bits from the glyph memory

3 Types of bitGen

- Direct Graphics
  - Look at hCount and vCount to see where you are on the screen
  - Depending on where you are, force the output to a particular color
  - Tedious for complex things, nice for large, static things

```
parameter BLACK = 3'b 000, WHITE = 3'b111, RED = 3'b100;
// paint a white box on a red background
always@(*)
   if (~bright) rgb = BLACK; // force black if not bright
   // check to see if you're in the box
  else if (((hCount >= 100) && (hCount <= 300)) &&
            ((vCount >= 150) && (vCount <= 350))) rgb = WHITE;
  else rgb = RED; // background color
```

VGA Memory Requirements

- 640x480 VGA (bitmapped)
  - 307,200 pixels
  - 3 bits per pixel
  - Imagine using 24 bits per memory location (8 pixels)
  - 38.4 K-words with 24-bit words for 640x480
    - 115.2 K-bytes
  - FAR larger than you can put on your chip…
  - Not so bad with an off-chip RAM

VGA Memory Requirements

- 320x240 VGA (bitmapped)
  - 76,800 pixels
  - Each stored pixel is 2x2 screen pixels
  - 3 bits per pixel
  - 8 pixels per 24-bit word (for example)
  - 9.6k 24-bit words needed
    - 28.8 K-bytes
  - Much more realistic…but still significant memory if you want to put it on-chip

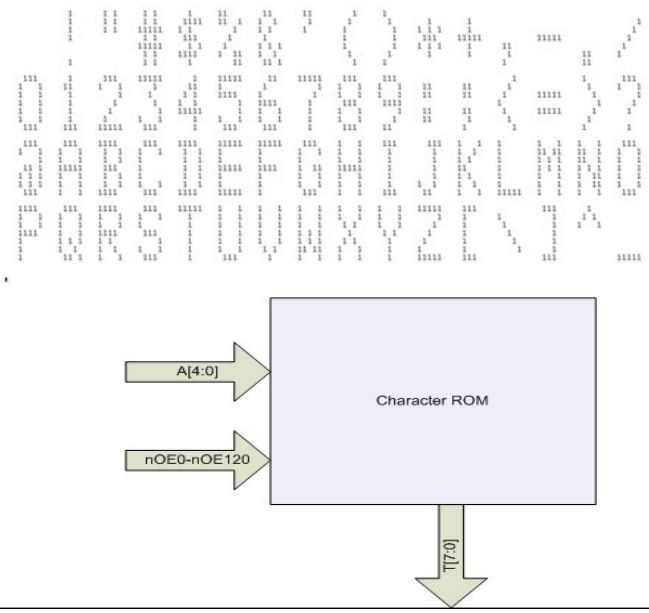
VGA Memory Requirements

- 80 char by 60 line display (8x8 glyphs)
  - 4800 locations
  - Each location has one of 256 char/glyphs
  - 8-bits per location
    - 2 locations per 16-bit word?
    - 2400 words for the frame buffer
  - Each char/glyph is (say) 8x8 pixels
    - results in 640x480 display…
  - 8x8x256 bits for char/glyph table
    - 16kbits (1k words) for char/glyph table
    - Will this fit on your chip?

VGA Memory Requirements

- 80 char by 60 line display (8x8 glyphs)
  - 4800 locations
  - Each location has one of 64 char/glyphs
  - 6-bits per location
    - 4 locations per 24-bit word?
    - 1200 words for frame buffer?
  - Each char/glyph is (say) 8x8 pixels
    - results in 640x480 display…
  - 8x8x64 bits for char/glyph table
    - 4kbits for char/glyph table (32 words, 128 b/word)
    - Will this fit on your chip?

CharROM



CharROM

The Character ROM contains the 64 member ASCII upper-case character set. The characters are addressed with a 5-bit binary address A[4:0] and a 16-bit unary decoded address, nOE0-nOE120. The Character ROM outputs a single row of the selected character at a time on the signals T[7:0].

A[4:3] decodes one of the four rows of 16 characters in the ROM.

A[4:3] == 0	- first row	" !"#$%&'()*+,-./"
A[4:3] == 1	- second row	"0123456789:;<=>?"
A[4:3] == 2	- third row	"@ABCDEFGHIJKLMNO"
A[4:3] == 3	- fourth row	"PQRSTUVWXYZ[\]^_"

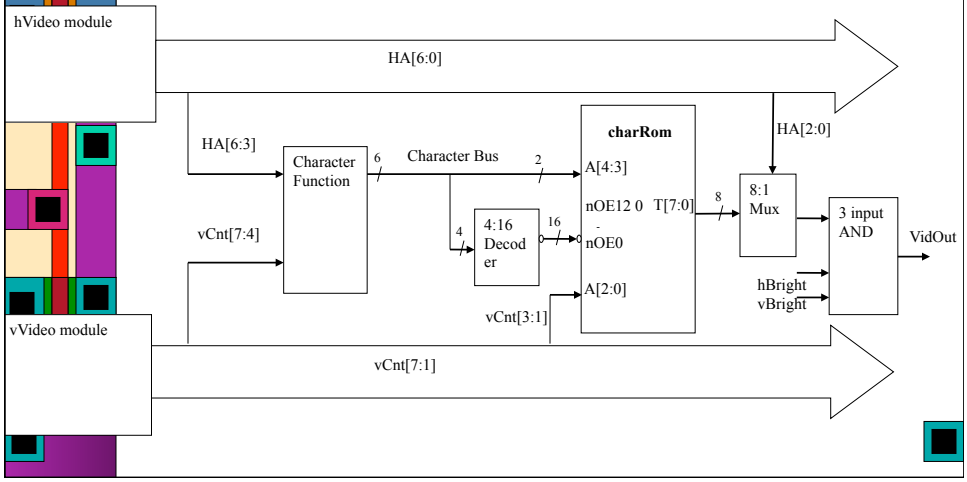
The sixteen signals nOE0, nOE8, nOE16, nOE24, nOE32, nOE40, nOE48, nOE56, nOE64, nOE72, nOE80, nOE88, nOE96, nOE104, nOE112, nOE120 select one of the sixteen columns of of four characters. These signals are active low and only one is asserted at any time. For instance, nOE0==0 selects the first column with the four characters " 0@P" in it and nOE7==0 selects "'7GW".

A[2:0] decodes one of the eight character rows. For instance, if the character "A" is selected with A[4:3]==2 and nOE8 then A[2:0] will produce the following binary output on T[7:0].

		Binary	Visible Output
A[2:0] == 0	- first row	00011100	***
A[2:0] == 1	- second row	00100010	* *
A[2:0] == 2	- third row	00100010	* *
A[2:0] == 3	- fourth row	00111110	*****
A[2:0] == 4	- fifth row	00100010	* *
A[2:0] == 5	- sixth row	00100010	* *
A[2:0] == 6	- seventh row	00100010	* *
A[2:0] == 7	- eight row	00000000	

CharROM

Fit the charROM into a VGA system
- hVideo walks along the row
- vVideo picks which row to walk along



Two Lines of Text

```
name rom
32 characters of six bits each
:16 32 6
00  ;
00  ;
28  ; H
25  ; E
2C  ; L
2C  ; L
2F  ; O
00  ;
37  ; W
2F  ; O
32  ; R
2C  ; L
24  ; D
00  ;
00  ;
00  ;

00  ;
00  ;
33  ; S
25  ; E
34  ; T
00  ;
2D  ; M
25  ; E
00  ;
26  ; F
32  ; R
25  ; E
25  ; E
00  ;
00  ;
00  ;
```

- Character Function…
  - … i.e. Frame Buffer
- 16 characters/line x 8 pixels/ char = 128pixels
- 6 bits to address a character
  - A[4:3] = row of CharRom
  - R[2:0] = column of CharRom
  - A[2:0] = row of character

RAM/ROM Generator

- Designed by Allen Tanner 8 years ago as his class project...
  - makemem
- Simple ROM arrays
  (Don't use the SRAM)

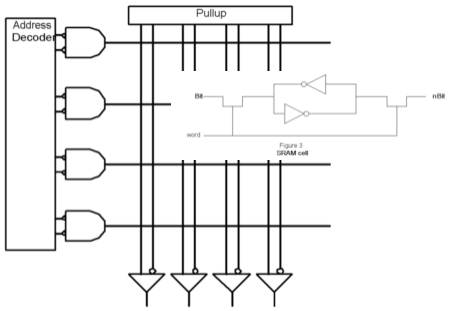
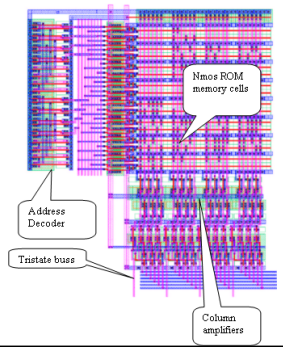


Figure 4
SRAM Design



makemem

```
102 vladimir:~> java -cp /uusoc/facility/cad_common/local/Cadence/lib/mem/j makemem -h
makemem v2.2  Nov 8, 2004
 Allen Tanner University of Utah CS6710

Enter the following:
java makemem choice options
  Where: choice selects the creation of either ROM or SRAM.
   for ROM  enter:-r rname       : rname.rom  is the file name.
                    :
   for SRAM enter:-s  r c        : Version 1 SRAM single port.
   for SRAM enter:-s1 r c        : Version 2 SRAM single port.
   for SRAM enter:-s2 r c        : Version 2 SRAM dual port.
   for SRAM enter:-s3 r c        : Version 2 SRAM triple port.
                    : r is the number of rows (decimal).
                    : c is the number of columns (decimal).
                    :
            :-h -H        : help (no processing occurs when help is requested).
            :-f fname     : output file name. Used with .cif, .v & .il files.
            :-n sname rname : sname for array top cell name.
            :             : rname for ROM (only) dockable ROM array top cell name
            :-t n         : use tristate buffers on the outputs of ROM.
            :-q           : output hello.txt file to find the working file directory.
103 vladimir:~>
```

makemem Limits

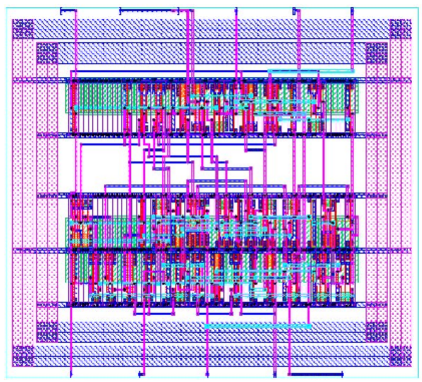
- Number of rows is limited to 64 by address decoder design
  - Columns are not restricted
- For ROM you can add a tristate bus at the output which is another level of decoding
  - width must be an even number
- SRAM has single, dual, and triple port options
  - But, fabricated versions are very uneven…

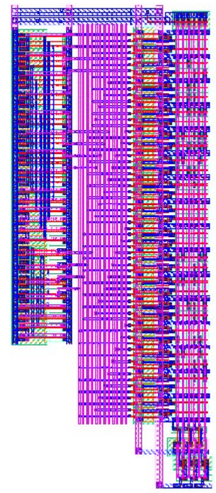
ROM vs. Verilog

```
name rom
32 characters of six bits each
:16 32 6
00  ;
00  ;
28  ; H
25  ; E
2C  ; L
2C  ; L
2F  ; O
00  ;
37  ; W
2F  ; O
32  ; R
2C  ; L
24  ; D
00  ;
00  ;
00  ;

00  ;
00  ;
33  ; S
25  ; E
34  ; T
00  ;
2D  ; M
25  ; E
00  ;
26  ; F
32  ; R
25  ; E
25  ; E
00  ;
00  ;
00  ;
```

```
module mywords(addr, char);
   input [4:0] addr;
   output reg [5:0] char;

always @(addr)
 begin
  case(addr)
     'h00 : char = 'h00  ;  //
     'h01 : char = 'h00  ;  //
     'h02 : char = 'h28  ;  // H
     'h03 : char = 'h25  ;  // E
     'h04 : char = 'h2C  ;  // L
     'h05 : char = 'h2C  ;  // L
     'h06 : char = 'h2F  ;  // O
     'h07 : char = 'h00  ;  //
     'h08 : char = 'h37  ;  // W
     'h09 : char = 'h2F  ;  // O
     'h0A : char = 'h32  ;  // R
     'h0B : char = 'h2C  ;  // L
     'h0C : char = 'h24  ;  // D
     'h0D : char = 'h00  ;  //
     'h0E : char = 'h00  ;  //
     'h0F : char = 'h00  ;  //

     'h10 : char = 'h00  ;  //
     'h11 : char = 'h00  ;  //
     'h12 : char = 'h33  ;  // S
     'h13 : char = 'h25  ;  // E
     'h14 : char = 'h34  ;  // T
     'h15 : char = 'h00  ;  //
     'h16 : char = 'h2D  ;  // M
     'h17 : char = 'h25  ;  // E
     'h18 : char = 'h00  ;  //
     'h19 : char = 'h26  ;  // F
     'h1A : char = 'h32  ;  // R
     'h1B : char = 'h25  ;  // E
     'h1C : char = 'h25  ;  // E
     'h1D : char = 'h00  ;  //
     'h1E : char = 'h00  ;  //
     'h1F : char = 'h00  ;  //
  endcase
 end
endmodule // mywords
```

ROM vs. Verilog

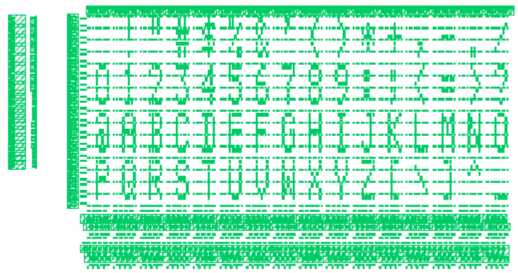




ROM vs. Verilog

Character generator rom.
64 characters (upper case, numbers and a few special characters, no lower case)
This file is taken from the Free Software Foundation
Modified for ugly 8, Q, G, A
groups of 8 bits with tristate output
Allen Tanner University of Utah CS6710
:2 32 128 8
0000000000001000000101000001010000001000001100000001100000011000000001000001000000000000000000000000000000000000000000000000000000
0000000000001000000101000001010000011110001100100010010000001000000010000000100000001000000010000000000000000000000000000000000001
0000000000001000000101000011111000101000000001000010100000010000000100000000100001010100000100000000000000000000000000000000000010
0000000000001000000000000001010000011100000010000001000000000000000100000000100000111000011111000000000011111000000000000000100
0000000000000000000000000001111100000101000010000000101010000000000000100000000010000101010000010000001100000000000000000000001000
0000000000000000000000000001010000111100001001100010010000000000000000100000001000000010000000100000001000000000000000110000010000
0000000000001000000000000001010000001000000000110000110100000000000000100000100000000000000000000000001000000000000000110000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000111000000100000011100001111100000010000111110000011000011111000011100000111000000000000000000000000100000000000000100000001110
0001000100001100000100010000000100000011000010000000010000000000100010001000100010000110000001100000001000000000000000010000010001
0001000100000100000000010000010000000101000011110000100000000000100010001000100010000110000001100000010000001111100000010000000001
0001000100000100000000010000000100001001000000000100011110000000100000011100000111100000000000000000001000000000000000000100000010
0001000100000100000000100000000010001111100000001000100010000010000010001000000001000011000000110000001000000111110000001000000100
0001000100000100000001000000010001000000100001000100010001000001000001000100000001000001100000001000000010000000000000000100000000000
0000111000000111000011111000011100000001000001110000011100000010000001110000011000000000000000010000000001000000000000000100000000100
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000111000000111000011110000011100001111000011111000111110000111000010001000011100000011100010001000100000000100010001000100001110
0001000100010001000010010001000100001001000100000000100000001000100010001000001000000000100001001000010000000110110001100100010001
0000000100010001000010010001000000000100100010000000100000001000000010001000001000000000100001010000010000000101010001100100010001
0000110100011111000011100001000000000100100011111000111110000100110001111100000100000000100001100000010000000101010001010100010001
0001010100010001000010010001000000000100100010000000100000001000100010001000001000000000100001010000010000000100010001001100010001
0001010100010001000010010001000100001001000100000000100000001000100010001000001000000100100001001000010000000100010001001100010001
0000111000010001000111100000111000011110000111110001000000001110000100010000111000000110000010001000111110001000100010001000001110
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0001111000000111000011110000011100001111100010001000100010001000100010001000100010001000111110000111000000000000011100000010000000000
0001000100010001000100010001000100000100000100010001000100010001000100010001000100000001000010000001000000000010000010100000000000
0001000100010001000100010001000000000100000100010001000100010001000010100001000100000001000000010000000010000000000100001000100000000
0001111000010001000111100000111000000010000010001000100010001010100000100000010100000010000000100000000010000000010000000000000000000
0001000000010101000101000000000100000100000100010001000100010101000010100000010000000100000000100000000010000000100000000000000000
0001000000001001000010010000100010000010000010001000010100001101100010001000001000000100000000010000000000100000001000000000000000000
0001000000001101000100010000111000000100000011100000010000010001000100010000010000011111000011100000000000000111000000000000011111
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

ROM vs. Verilog



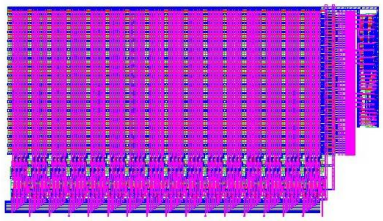
ROM vs. Verilog

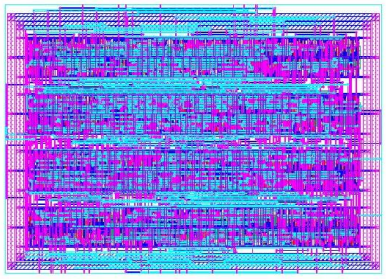
```
// Verilog HDL for "Ax", "char10" "behavioral"
//2Character generator rom.
//64 characters (upper case, numbers and a few special characters, no lower case)
//This file is taken from the Free Software Foundation
//Modified for ugly 8, Q, G, A
//groups of 8 bits with tristate output
//Allen Tanner University of Utah CS6710

module char10(ROM_row, nOE0, nOE8, nOE16, nOE24, nOE32, nOE40, nOE48, nOE56, nOE64, nOE72, nOE80, nOE88, nOE96, nOE104, nOE112, nOE120, T);
   input [4:0]ROM_row;
   input      nOE0, nOE8, nOE16, nOE24, nOE32, nOE40, nOE48, nOE56, nOE64, nOE72, nOE80, nOE88, nOE96, nOE104, nOE112, nOE120;
   output [7:0] T;

   wire [7:0]   twist;
   reg [127:0] ROW;

   always @(ROM_row)
     case(ROM_row)

       5'h00 : ROW = 128'b0000000000000010000001010000010100000001000001100000000110000001100000000100000100000000000000000000000000000000000000000000
       5'h01 : ROW = 128'b0000000000000010000001010000010100000111100011001000100100000001000000010000000100000001000000010000000000000000000000000000001
       5'h02 : ROW = 128'b0000000000000010000001010000111110001010000000001000010100000001000000010000000001000010101000001000000000000000000000000000000010
       5'h03 : ROW = 128'b0000000000000010000000000000001010000011100000010000001000000000000000010000000001000001110000111110000000000011111000000000000000100
       5'h04 : ROW = 128'b0000000000000000000000000000011111000001010000100000010101000000000000010000000001000010101000001000000110000000000000000000000001000
       5'h05 : ROW = 128'b0000000000000000000000000000010100001111000010011000100100000000000000010000000100000001000000010000000100000000000000011000001000000
       5'h06 : ROW = 128'b0000000000000010000000000000001010000000100000000110000110100000000000000010000010000000000000000000000000100000000000000110000000000
       5'h07 : ROW = 128'b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
       5'h08 : ROW = 128'b0000111000000010000000111000011111000000100001111100000110000111110000111000000111000000000000000000000000100000000000000010000000111 0
       5'h09 : ROW = 128'b0001000100001100000010001000000010000001100001000000000100000000001000100010001000100001100000011000000010000000000000000100000100001
       5'h0A : ROW = 128'b0001000100000010000000000100000100000001010000111100001000000000010000100010001000100001100000011000000100000011111000000100000000001
       5'h0B : ROW = 128'b0001000100000010000000000100000001000010010000000001000111100000001000000111000000111100000000000000000001000000000000000000100000010
       5'h0C : ROW = 128'b0001000100000010000000010000000001000111110000000010001000100000100000100010000000010000110000000110000000100000011111000000010000000100
       5'h0D : ROW = 128'b0001000100000010000000100000001000100000001000010001000100010000010000010001000000010000011000000001000000010000000000000000000100000000
       5'h0E : ROW = 128'b0000111000000111000001111100001110000000001000001110000001110000000100000011100000110000000000000001000000000100000000000000010000000100
       5'h0F : ROW = 128'b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
       5'h10 : ROW = 128'b0000111000000111000001111000000111000001111000001111100011111000011100001000100001110000000111000100010001000000001000100010001000011 10
       5'h11 : ROW = 128'b0001000100010001000010010001000100001001000100000000100000001000100010001000000010000000001000010010000100000000110110001100100010001
       5'h12 : ROW = 128'b0000000100010001000010010001000000000100100010000000010000000010000000010001000000100000000010000101000001000000010101000110010001000 1
       5'h13 : ROW = 128'b0000110100011111000011100001000000000100100011111000111100001001100011111000000100000000100001100000010000000101010001010100010001
       5'h14 : ROW = 128'b0001010100010001000010010001000000000100100010000000010000000010001000100010000001000000000100001010000010000000100010001001100010001
       5'h15 : ROW = 128'b0001010100010001000010010001000100001001000100000000100000001000100010001000000100000100100001001000010000000100010001001100010001
       5'h16 : ROW = 128'b0000111000010001000111100000111000011110000111110001000000001110000100010000111000001100000100010001111100010001000100010000111 0
       5'h17 : ROW = 128'b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
       5'h18 : ROW = 128'b0001111100000111000011110000011100001111100010001000100010001000100010001000100010001111100001110000000000000111000000100000000000
       5'h19 : ROW = 128'b0001000100010001000100010001000100010000010000010001000100010001000100010001000100010000000010000100000010000000000100000101000000000
       5'h1A : ROW = 128'b0001000100010001000100010001000000000100000100010001000100010000101000010001000000010000010000000100000000010000100010000000000
       5'h1B : ROW = 128'b0001111000010001000111100000111000000010000010001000100010001010100000100000010100000001000000100000000010000000001000000000000000000
       5'h1C : ROW = 128'b0001000000001010100010100000000001000001000001000100010001000101010000101000000100000001000000001000000000100000001000000000000000000
       5'h1D : ROW = 128'b0001000000000100100001001000010001000001000001000100001010000110110001000100000100000100000000010000000000100000010000000000000000000
       5'h1E : ROW = 128'b0001000000000110100010001000011100000001000000111000000100000100010001000100000100000111110000111000000000000001110000000000000011111
       5'h1F : ROW = 128'b0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     endcase
```

ROM vs. Verilog

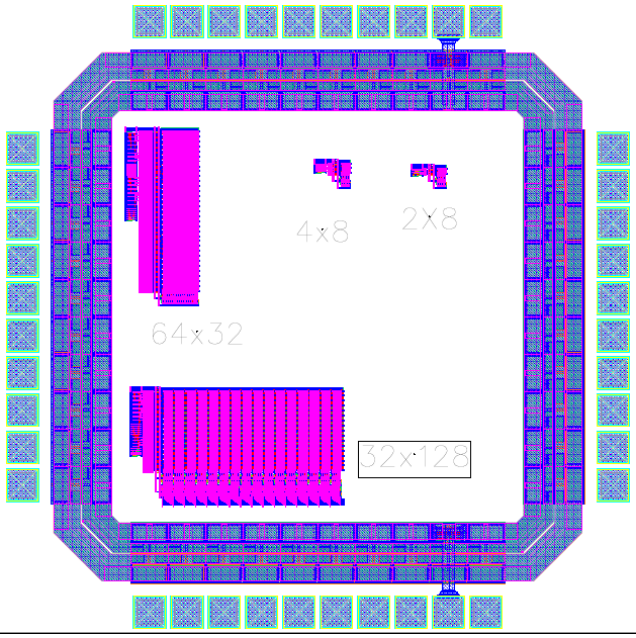
```
assign      twist = {8{  nOE0}} & ROW[  7:  0]
                  | {8{  nOE8}} & ROW[ 15:  8]
                  | {8{ nOE16}} & ROW[ 23: 16]
                  | {8{ nOE24}} & ROW[ 31: 24]
                  | {8{ nOE32}} & ROW[ 39: 32]
                  | {8{ nOE40}} & ROW[ 47: 40]
                  | {8{ nOE48}} & ROW[ 55: 48]
                  | {8{ nOE56}} & ROW[ 63: 56]
                  | {8{ nOE64}} & ROW[ 71: 64]
                  | {8{ nOE72}} & ROW[ 79: 72]
                  | {8{ nOE80}} & ROW[ 87: 80]
                  | {8{ nOE88}} & ROW[ 95: 88]
                  | {8{ nOE96}} & ROW[103: 96]
                  | {8{nOE104}} & ROW[111:104]
                  | {8{nOE112}} & ROW[119:112]
                  | {8{nOE120}} & ROW[127:120];

assign      T = {twist[0],twist[1],twist[2],twist[3],twist[4],twist[5],twist[6],twist[7]};

endmodule // char10
```

ROM vs. Verilog





ROM size comparison



SRAM

- Makemem also generates SRAM
  - Three different variants: single, dual, triple port
  - Each port is independent R/W
  - But, no automatic arbitration, so make sure you're not using the same address on multiple ports

BUT! It's not working well
Use memCellsF09 instead!!!

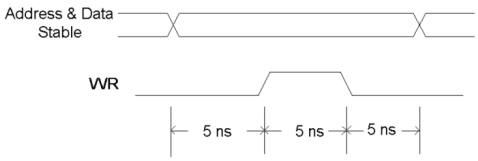


Figure 5
Write Timing

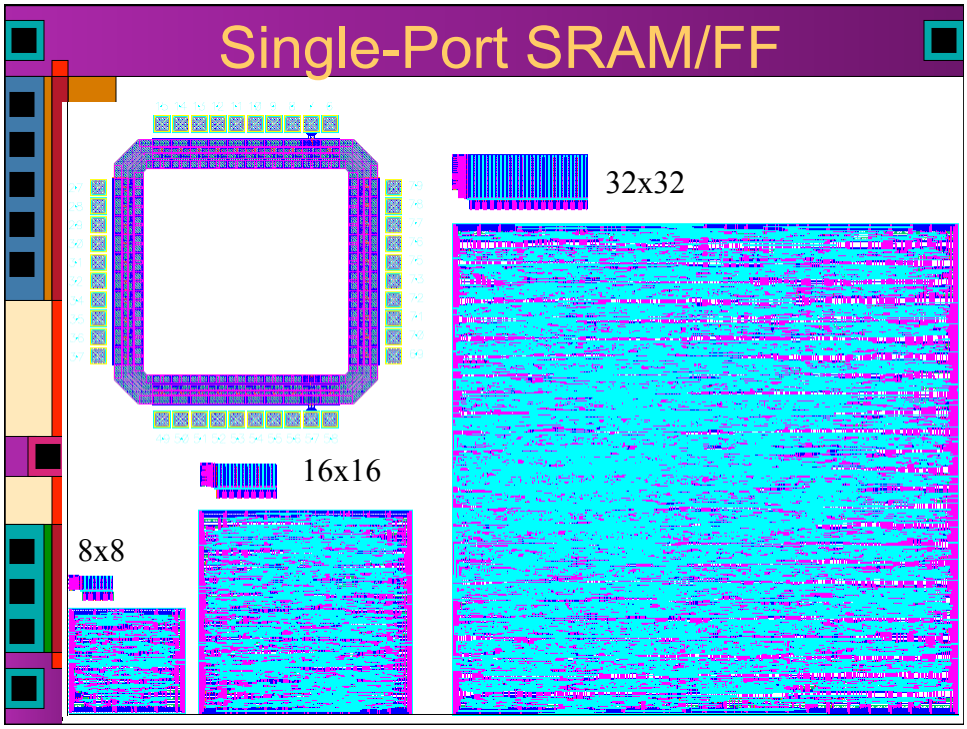
SRAM vs FF-registers

```
module regfile #(parameter WIDTH = 8, REGBITS = 3)
                (input               clk, regwrite,
                 input  [REGBITS-1:0] ra1, ra2, wa,
                 input  [WIDTH-1:0]   wd,
                 output [WIDTH-1:0]   rd1, rd2);
  reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];
  // read two ports (combinational)
  // write third port on rising edge of clock
  always @(posedge clk)
    if (regwrite) RAM[wa] <= wd;

  assign rd1 =  RAM[ra1];
  assign rd2 =  RAM[ra2];
endmodule
```

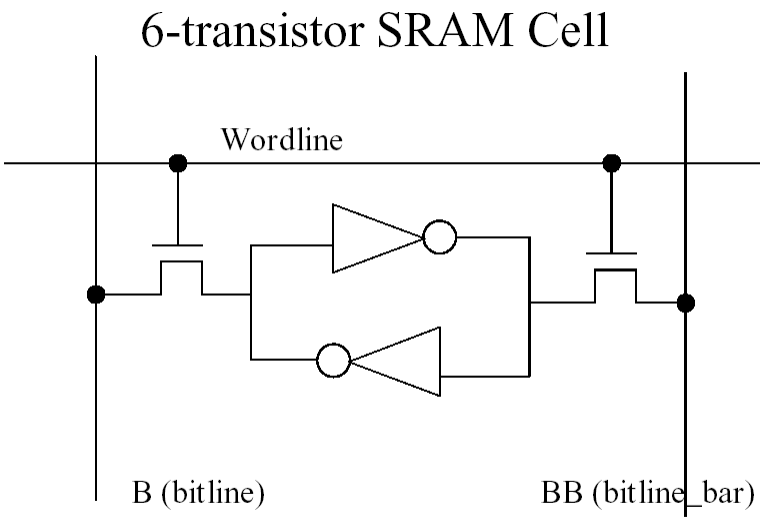
SRAM vs FF-registers

```
module SRAM #(parameter WIDTH = 8, REGBITS = 3)
                (input               clk, WE,
                 input  [REGBITS-1:0] addr,
                 input  [WIDTH-1:0]   wd,
                 output [WIDTH-1:0]   data);
  reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

// on clk, write if WE is high
  always @(posedge clk)
    if (WE) RAM[addr] <= wd;

  // Read asynchronously from addr
  assign data =  RAM[addr];

endmodule
```

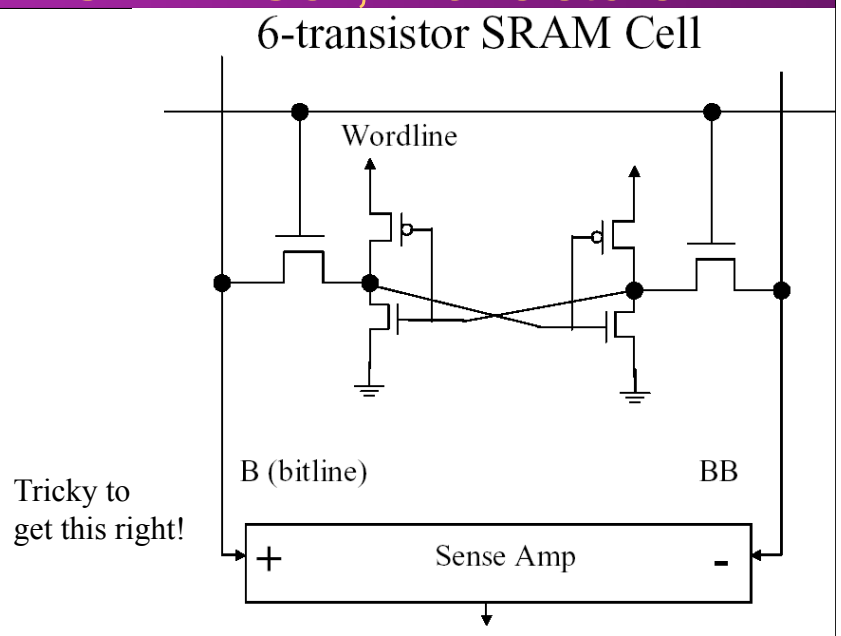
Single-Port SRAM/FF



SRAM Circuits

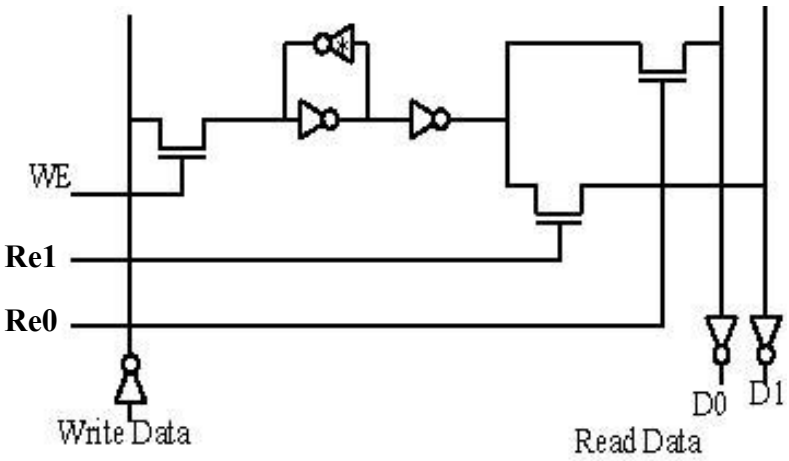


SRAM Cell, Transistors



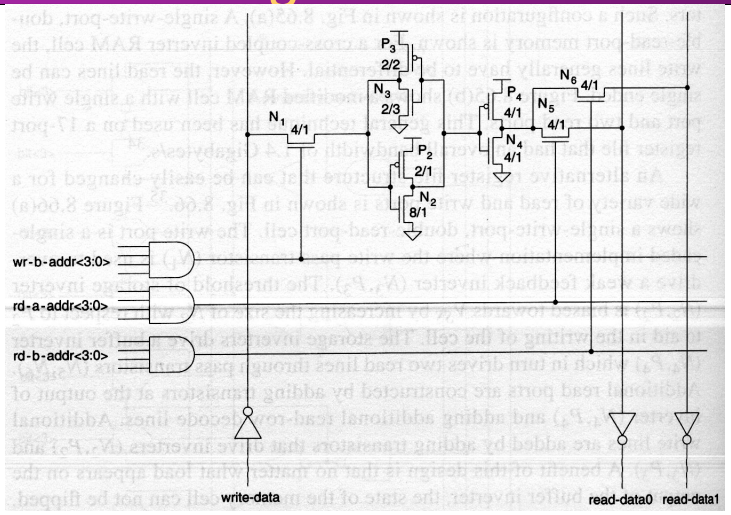
Tricky to get this right!

Multi-Port Register



- Register file cell with single-ended read – makes a great register file

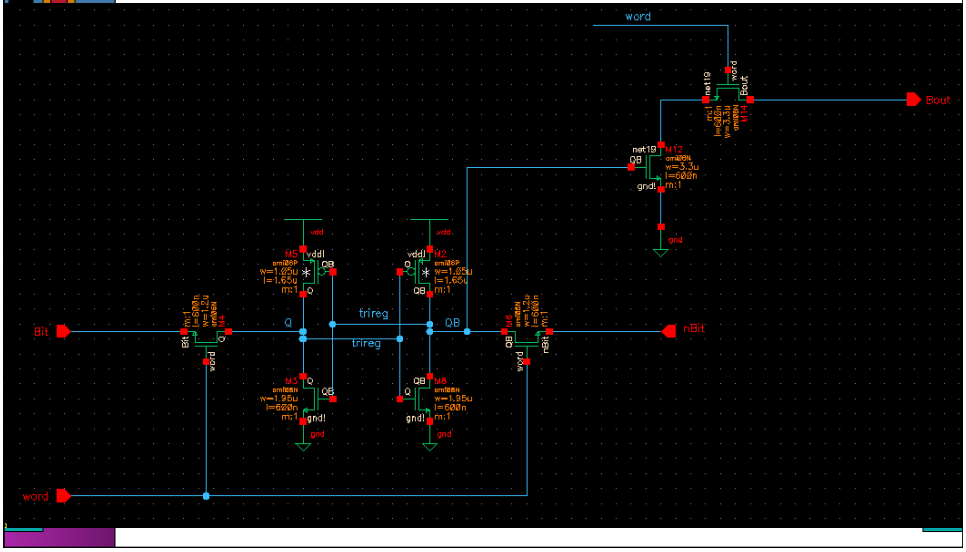
Register File



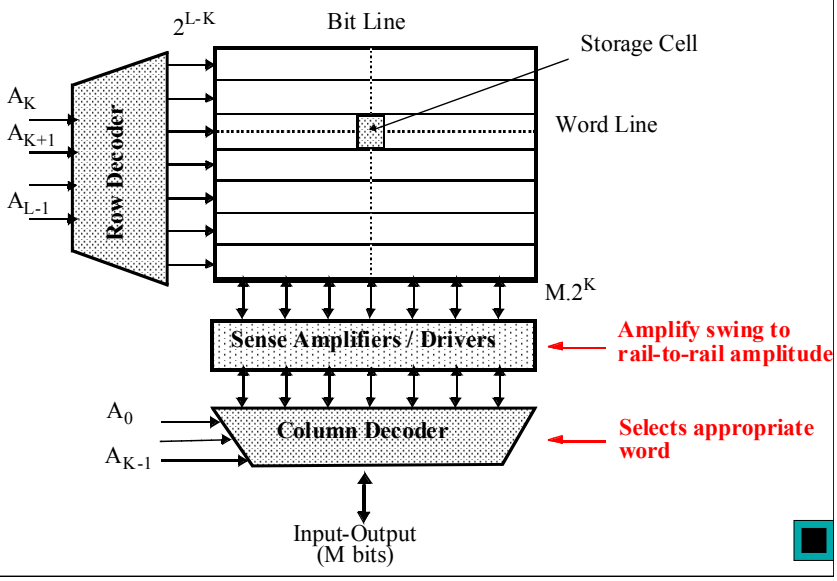
- Slightly larger cell, but with single-ended read – makes a great register file

SRAM Cell

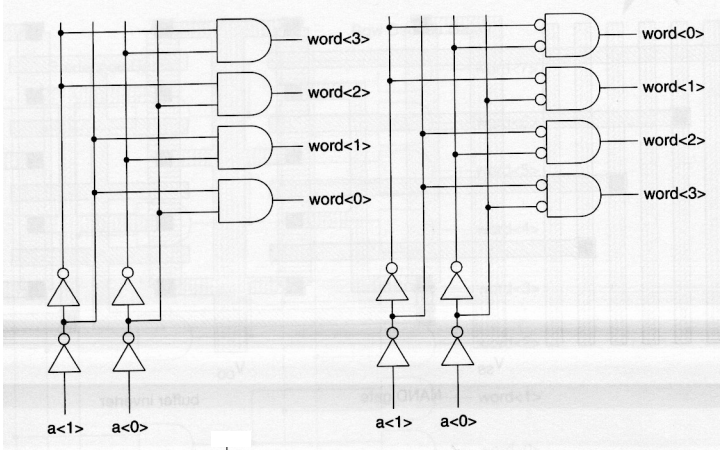
Yet another cell – differential write, single-ended read



Array-Structured Memory



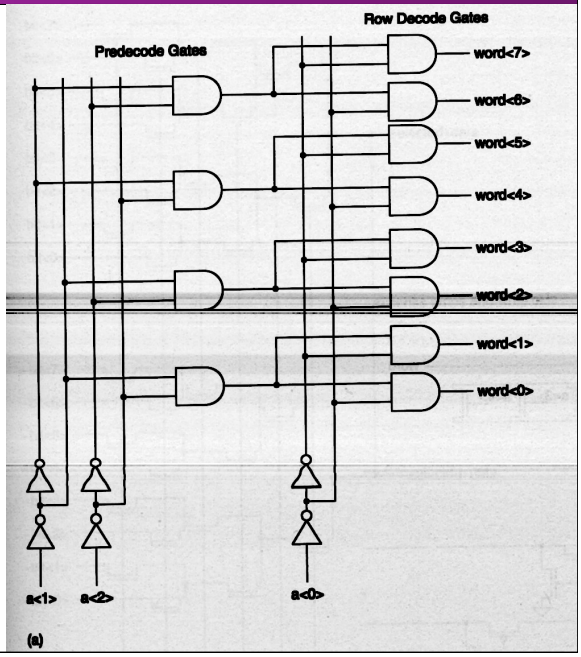
Row Decoders



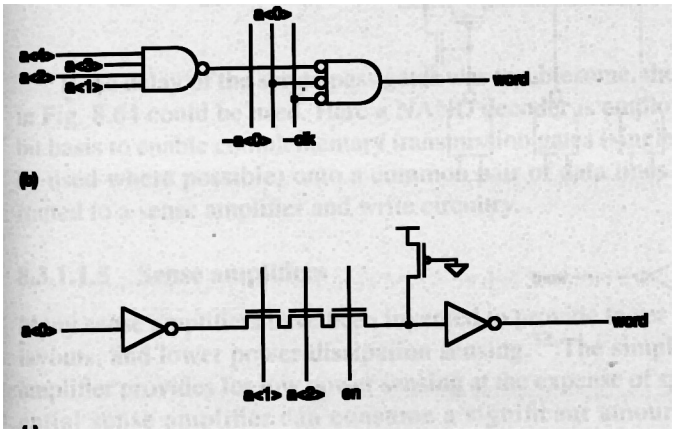
- Select exactly one of the memory rows
  - Simple versions are just gates

Pre-decode Row Decoder

- Multiple levels of decoding can be more efficient layout

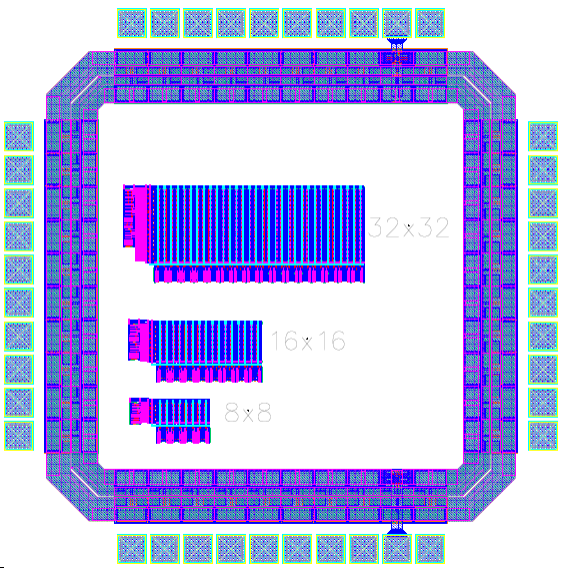


Pre-decode Row Decoder



- Other circuit tricks for building row decoders…

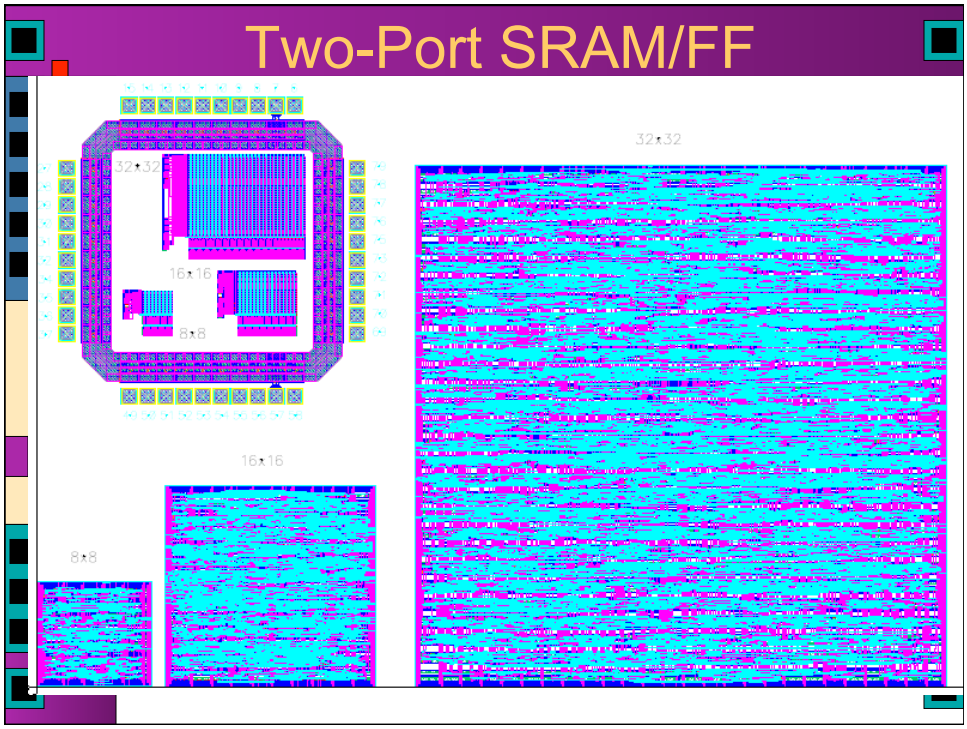
Single-Port SRAM



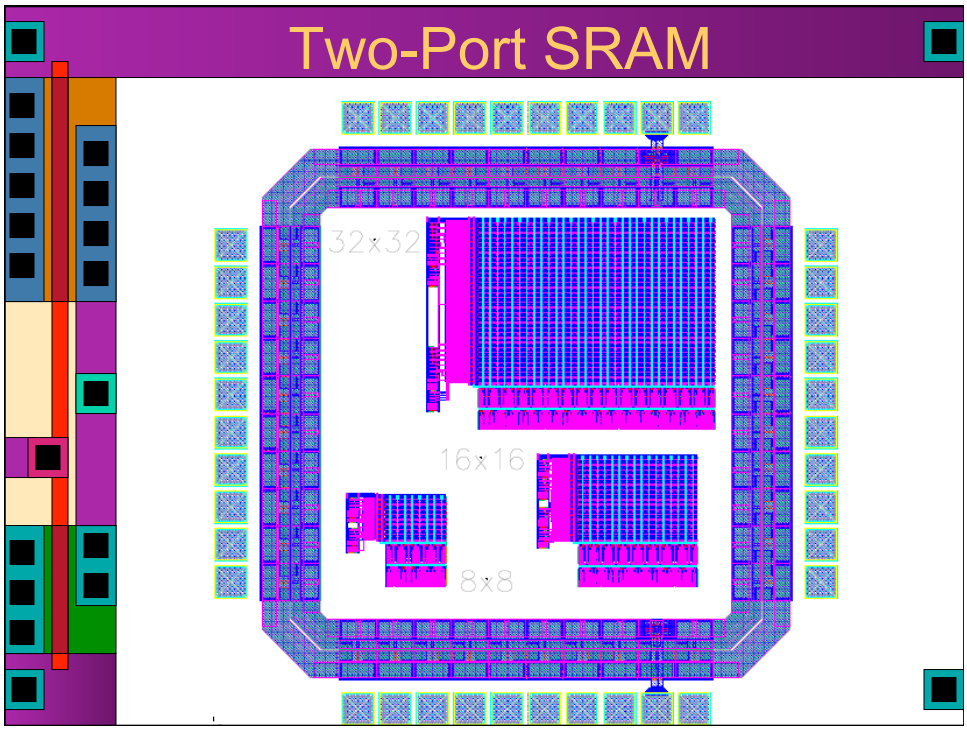
Two-Port SRAM/FF

```
module SRAM2 #(parameter WIDTH = 8, REGBITS = 3)
              (input                clk, WE,
               input  [REGBITS-1:0] addr, raddr,
               input  [WIDTH-1:0]   wd,
               output [WIDTH-1:0]   data, rdata);
  reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

// on clk, write if WE is high
  always @(posedge clk)
    if (WE) RAM[addr] <= wd;

  // Read asynchronously from addr & raddr
  assign data =  RAM[addr];
  assign rdata = RAM[raddr];
endmodule
```

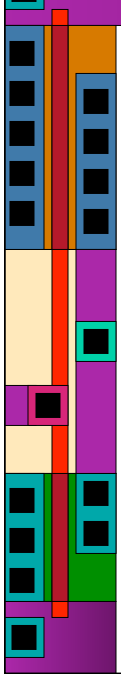
Two-Port SRAM/FF



Two-Port SRAM



Conclusions



- Try out memCellsF09 for SRAM
  - Details on the class web page
  - But, as you can see, you can't fit much on a chip
- ROMs are very useful for tables of data
  - I'd use Verilog case-statements…
- If you're using VGA
  - Check out the mini-project from 2005
  - Again, on the class website