VGA Tile Graphics on an FPGA: A Tutorial

Stephen A. Edwards

Spring 2025

Tile-based graphics display images by repeating sub-images and thus using less memory than framebuffers. Countless video arcade games took this approach in the 1970s and ’80s, such as Namco’s Pac-Man (Fig. 1), as well as video game consoles, such as the Nintendo Entertainment System. Although such hardware is rarely mandatory in 21st century systems with the ready availability of gigabytes of video memory, its simplicity and power make it a good digital design exercise for FPGAs. This tutorial shows how to implement a tile-based VGA display on Terasic’s DE1-SoC board based around Altera’s Cyclone V SE 5CSEMA5F31C6 FPGA, which includes an HPS that includes a pair of ARM processor cores.

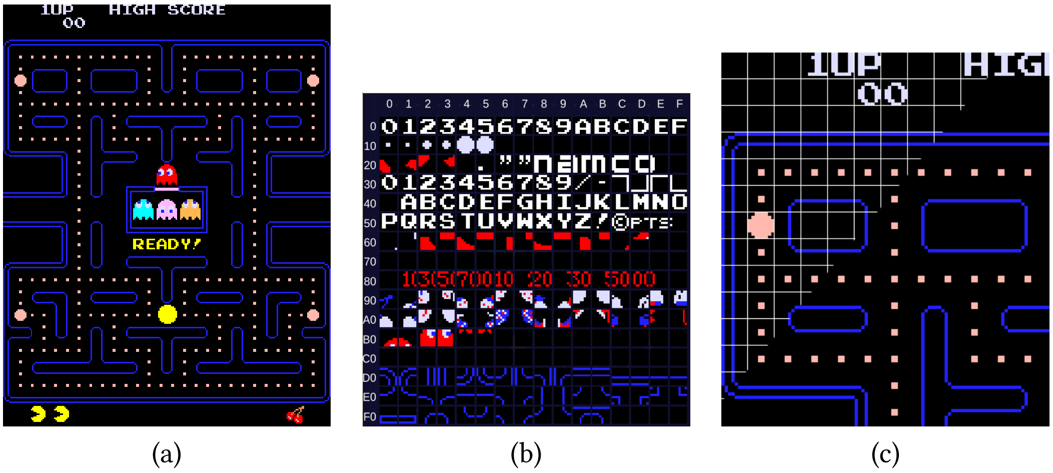

1UP
HIGH SCORE
00
READY!
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 10 20 30 40 50 60 70 80 90 A0 B0 C0 D0 E0 F0
(a)
(b)
(c)

Figure 1: Pac-Man, Namco 1980. (a) Starting game screen, 224 × 288; (b) Tile set: 256 8 × 8 tiles, 2 bpp; and (c) Game tile detail: 28 × 36 tiles, 2 bytes per tile (8b index, 5b palette)

1 Framebuffers and Tiles

Modern computers generally display raster images from frame buffers, which provides individual control over the color of each possible on-screen pixel. This provides flexibility at the cost of memory consumption. For example, modern displays often represent colors with 24-bit RGB values, which for a 640 × 480 VGA display, requires

$$640 \times 480 \times 3 \text{ bytes} = 921,600 \text{ bytes} = 900\text{K}.$$

This was a substantial amount of memory when VGA was introduced in 1987, so systems often reduced these demands by reducing the number of bits per pixel and employing a *palette*: a small, very fast memory that translated color codes to color values. For example, a framebuffer might use 8 bits per pixel (bpp) to represent a color index, allowing it to display 256 different colors at a time, but would use a 256 × 24 palette memory that allowed each of the 256 colors to be selected from a 24-bit color gamut. This would reduce framebuffer memory to

$$640 \times 480 \times 1 \text{ bytes} = 307,200 \text{ bytes} = 300 \text{ K}$$

plus 256 × 3 = 768 bytes for the palette. The aging GIF file format only supports indexed color, which can lead to artifacts.

At the extreme, a 1 bpp (monochrome) framebuffer still consumes

$$640 \times 480 = 307,000 \text{ bits} = 38,400 \text{ bytes} = 37.5 \text{ K}.$$

Tiles are effectively the palette approach applied also to sub-images. A text-mode display (i.e., that can only display letters and numbers) illustrates the idea: software is only able to control the identity of each character on the screen rather than each pixel. Consider an 80 × 30 character grid in which each character is taken from a font of 256. Each character is then 640 ÷ 80 = 8 pixels wide and 480 ÷ 30 = 16 pixels high (e.g., Fig. 2). The character grid requires

$$80 \times 30 = 2,400 \text{ bytes} = 2.34 \text{ K},$$

and the font, if it is only black-and-white, requires

$$256 \times 1 \text{ byte} \times 16 \text{ rows} = 4096 \text{ bytes} = 4 \text{ K},$$

which is a substantial reduction over the 37 K requirement for a monochrome framebuffer with the same resolution.

The original VGA text mode actually used 16 bits per character: 8 to select one of 256 different glyphs, four to select the background color, three for the foreground color, and one bit to make the character blink.

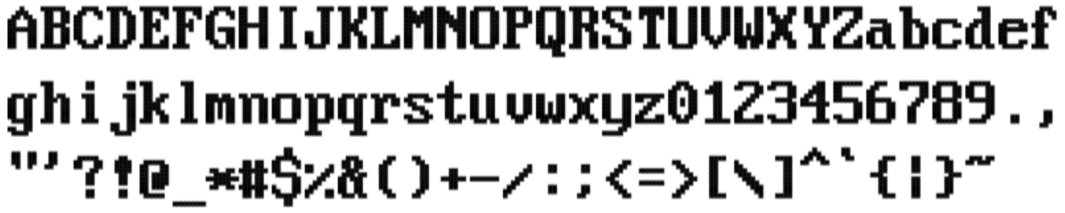

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef
ghijklmnopqrstuvwxyz0123456789.,
"'?!@_*#$%&()+-/:;<=>[\]^`{|}~

Figure 2: An 8 × 16 pixel font similar to that used on IBM PCs in the VGA era. Source: `https://fontstruct.com/fontstructions/show/1481905/dos-vga-9x16-1`

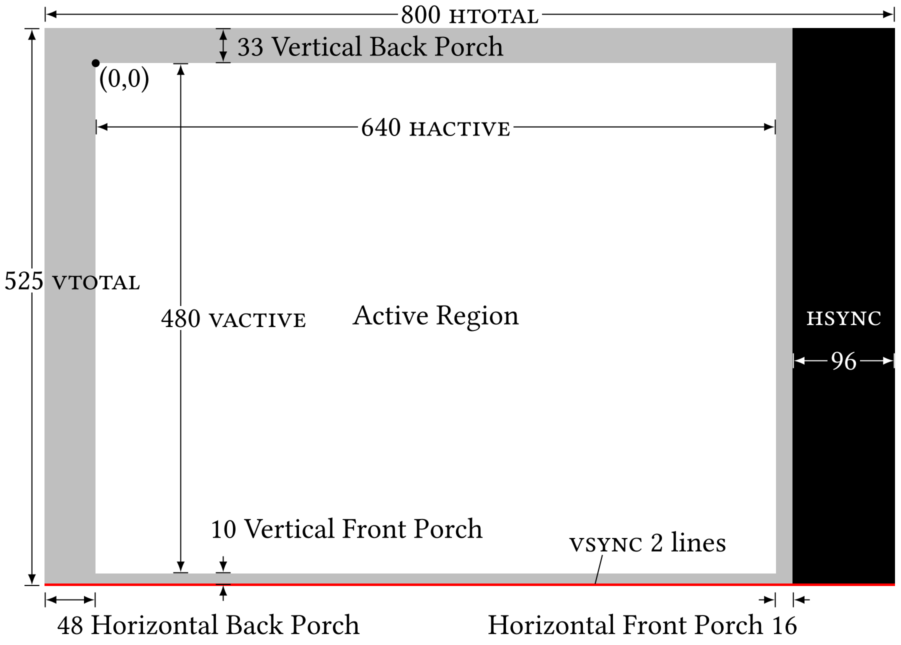


Figure 3: VGA signal timing. Horizontal dimensions are in pixels (25.175 MHz); vertical dimensions are in lines (31.46875 kHz).
Data from `http://www.tinyvga.com/vga-timing/640x480@60Hz`

2 The Video Graphics Array (VGA) Standard

Most video arcade games and consoles in the '70s and '80s produced raster-scanned, non-interlaced (progressive scan) NTSC-like video: 60 frames per second, 262 lines, but we will follow the slightly more modern Video Graphics Array (VGA) standard, which IBM introduced with their PS/2 line of personal computers in 1987 and remains available.

The basic VGA resolution is 640×480 at 60 fps with a 25.175 MHz dot clock: its timing is like a progressivly scanned version of interlaced NTSC color video. Modern LCD monitors display VGA signals by adapting to a wide variety of horizontal and vertical frequencies while digitizing the analog RGB VGA signal.

Fig. 3 illustrates VGA timing. Each line is 800 pixel periods wide, but only 640 of those are displayed; the remaining time is devoted to porches (blank periods) and synchronization. A new frame is displayed once every 525 line periods, but only 480 lines are displayed. Ten lines after the last active line is a two-line-long vertical synchronization pulse followed by 33 lines of additional "back porch" before the first line of the next frame.

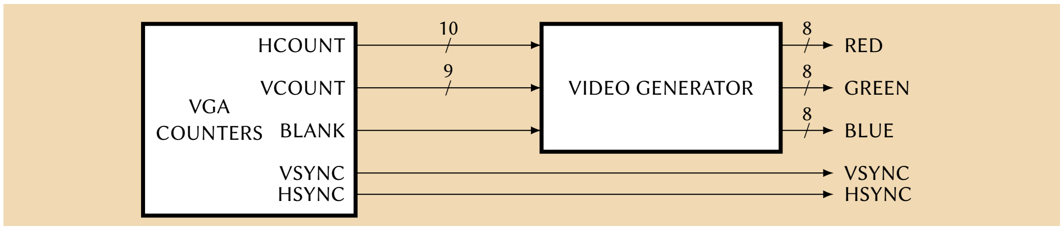


Figure 4: An abstract model of any VGA video generator, such as one for tiles. Counters generate horizontal and vertical coordinates as well as blanking and synchronization signals. The video generator translates these into three color signals.

3 VGA Counter Hardware

Fig. 4 illustrates the structure of a VGA video generator: counters generate horizontal and vertical coordinate values along with blanking signals, which are fed to a block that determines the color of the pixel at those coordinates.

Fig. 5 shows System Verilog that generates the horizontal and vertical synchronization signals along with blanking (true only during the active region) and horizontal and vertical coordinates.

The logic for the horizontal synchronization signal is subtle to save logic. Horizontal synchronization occurs from count 640 + 16 = 656 through count 640 + 16 + 96 − 1 = 751, which, in binary, are

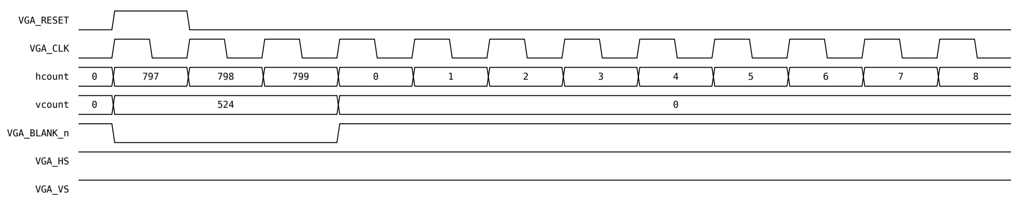
```
10 1001 0000
10 1110 1111
```

That is, *hcount*[9:7] is 101 and *hcount*[6:4] is not 000 or 111. The logic for the blanking signal employs similar trickery.

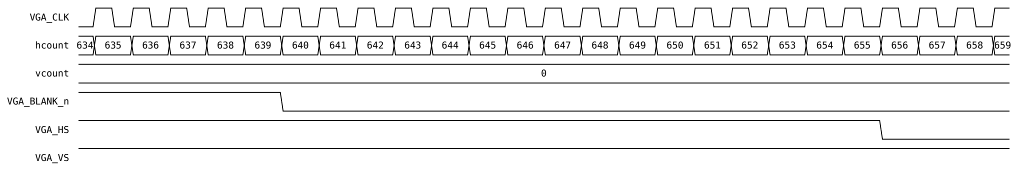
I wrote a simple Verilator testbench for this code, which applies a 25 MHz clock and reset and writes the simulation results as a *.vcd* file, which are displayed graphically in Fig. 6. These verify certain important behaviors.

```
module vga_counters(
   input  logic       VGA_CLK, VGA_RESET,
   output logic [9:0] hcount, // 0-639 active, 640-799 blank/sync
   output logic [9:0] vcount, // 0-479 active, 480-524 blank/sync
   output logic       VGA_HS, VGA_VS, VGA_BLANK_n);

   logic endOfLine;
   assign endOfLine = hcount == 10'd 799;

   always_ff @(posedge VGA_CLK or posedge VGA_RESET)
     if (VGA_RESET)      hcount <= 10'd 797;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 10'd 1;

   logic endOfFrame;
   assign endOfFrame = vcount == 10'd 524;

   always_ff @(posedge VGA_CLK or posedge VGA_RESET)
     if (VGA_RESET)      vcount <= 10'd 524;
     else if (endOfLine)
       if (endOfFrame)   vcount <= 10'd 0;
       else              vcount <= vcount + 10'd 1;

   // 656 <= hcount <= 751
   assign VGA_HS = !( hcount[9:7] == 3'b101 &
                      hcount[6:4] != 3'b000 & hcount[6:4] != 3'b111 );
   assign VGA_VS = !( vcount[9:1] == 9'd 245 ); // Lines 490 and 491

   // hcount < 640 && vcount < 480
   assign VGA_BLANK_n = !( hcount[9] & (hcount[8] | hcount[7]) ) &
                        !( vcount[9] | (vcount[8:5] == 4'b1111) );
endmodule
```

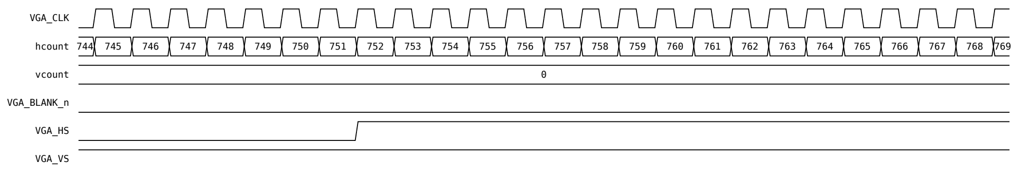
Figure 5: *vga_counters.sv*: Video counter module for VGA in System Verilog



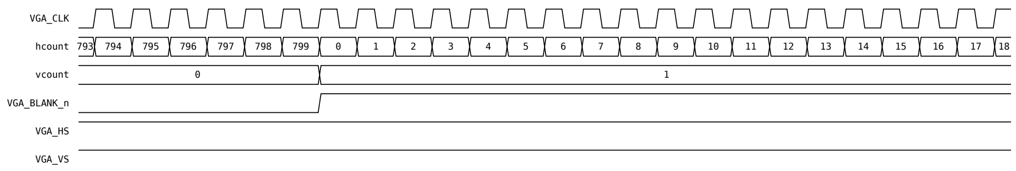
(a) Leaving reset; starting next frame after line 524; start of line 0



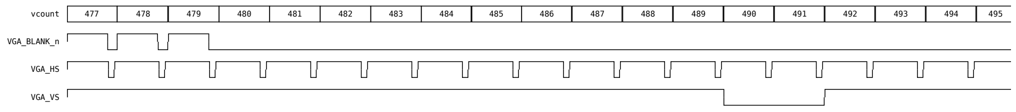
(b) End of Line 0 active: blanking then horizontal sync



(c) End of horizontal sync on line 0



(d) Start of line 1; blanking ends



(e) End of active region; vertical sync

Figure 6: Waveforms of the VGA counters.
After a script `https://github.com/phillbush/vcd2svg`

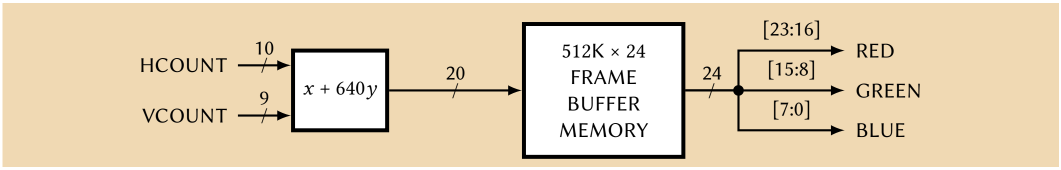


Figure 7: A 24 bpp VGA framebuffer built around a 1.5 MB memory

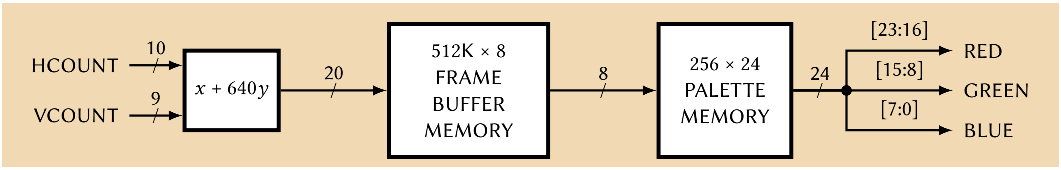


Figure 8: An 8 bpp VGA framebuffer that uses indexed colors from a palette memory. Chaining a second memory to the first reduces memory consumption by nearly 2/3 without reducing the resolution or number of available colors.

4 Framebuffer Design

Fig. 7 shows a design for a framebuffer: an address generator fetching data from frame buffer memory. Each word in the memory contains the color code for a single pixel, here, 24 bits. The address of each pixel is HCOUNT + VCOUNT × 640. Multiplying by 640 is fairly easy since $640 = 512 + 128$, so this could be implemented by adding HCOUNT, VCOUNT shifted left 7 bits, and VCOUNT shifted left 9 bits.

Indexed color using, say, 8 bpp, uses a smaller, 8-bit-wide framebuffer memory then feeds the 8-bit color code to a 256 × 24 palette memory, which ultimately produces a 24-bit color. Fig. 8 illustrates the structure of an indexed-color framebuffer, which has traded some control over pixel colors for a substantial reduction in memory. Note that the overall function being computed remains the same: the coordinate of each pixel in a 640×480 grid is being mapped to a 24-bit color.

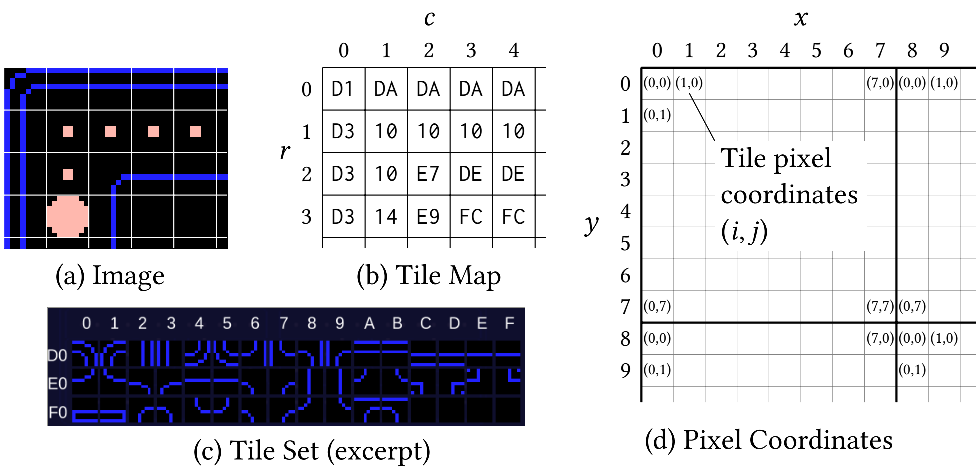


Figure 9: Tiles in Pac-Man: (a) Maze detail (b) Tile map: code for each tile (c) Partial tile set (d) Pixel and tile pixel coordinates

5 Tile Generator Design

A palette effectively compresses colors by expressing them using a small number of codes; tiles take this one step further: a *tile map* memory holds a two-dimensional array of tile codes, one for each tile on the screen; each code is used to index into memory that holds the pixel color codes in the *tile set*, a three-dimensional array. Finally, the color code is fed to a palette memory to look up the final color. Fig. 10 shows a block diagram of a tile video generator.

Fig. 9 illustrates the relationships among the tile set, tile map, tile numbers, and pixels

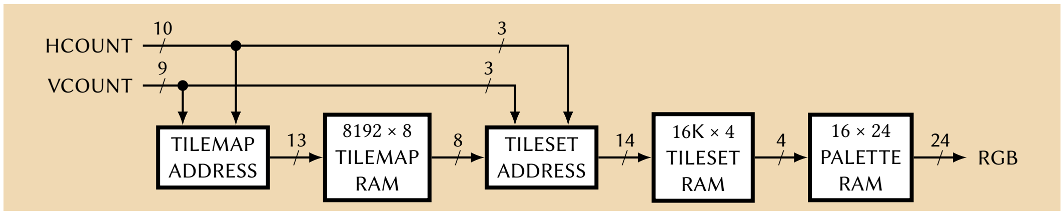


Figure 10: An 8 × 8 tile VGA generator. HCOUNT and VCOUNT deliver current pixel coordinates (x, y). TILEMAP ADDRESS transforms these to tile coordinates (c, r) to compute to address in TILEMAP RAM for the tile number t. TILESET ADDRESS combines t with local tile coordinates (i, j) to form the address for the pixel in TILESET RAM, which returns a 4-bit color code c' that the PALETTE RAM translates to a 24-bit RGB value.

(w, h)			Tile width and height (pixels)
(x, y)			Screen pixel coordinates
(c, r)	$=$	$(\lfloor x \div w \rfloor, \lfloor y \div h \rfloor)$	Tile column and row (tiles)
t	$=$	$\text{tilemap}[c, r]$	Tile number (from tile map)
(i, j)	$=$	$(x \bmod w, y \bmod h)$	Tile local coordinates (pixels)
c'	$=$	$\text{tileset}[i, j, t]$	Pixel color code (from tile set)
(r, g, b)	$=$	$\text{palette}[c']$	Pixel color (from palette)

Figure 11: Calculating the color (r, g, b) of the pixel at coordinates (x, y) in an array of $w \times h$-pixel tiles. $\text{tilemap}[c, r]$ is the tile number at column c and row r; $\text{tileset}[t, i, j]$ is the color code for pixel (i, j) in tile t; and $\text{palette}[c']$ is the RGB color for color code c'.

in an 8×8 tile system like that in Pac-Man. Fig. 9(a) shows a fragment of the maze in the top left corner of the screen; each square is an 8×8 pixel tile. Fig. 9(b) is the top left fragment of the tile map: the 2D array holding the code selecting each on-screen tile. Codes and the tiles they represent are shown in Fig. 9(c). Fig. 9(d) shows the relationship between pixel coordinates and tile coordinates. The tile in row 0, column 0 starts at pixel $(0, 0)$ and extends to pixel $(7, 7)$. The tile to the right of this, at row 0, column 1, starts at pixel $(8, 0)$, which is tile pixel $(0, 0)$.

Fig. 11 lists the rules for determining the pixel at screen coordinates (x, y) in a regular array of $w \times h$-pixel tiles. The column and row (c, r) of the tile containing the pixel is simply the quotients from dividing each coordinate by the size of the tile. These are used to look up the tile number in the tilemap array. The coordinates of the pixel within that tile (i.e., relative to the tile's top left corner) (i, j) is the remainder of these divisions. These coordinates and the tile number are used to look up the color code of the pixel in the tile set. Finally, the color code is used to look up the actual RGB color of the pixel.

Implementing a tile generator in hardware amounts to implementing the rules in Fig. 11. First, we will choose fixed-size 8×8 tiles: $(w, h) = (8, 8)$. Choosing these numbers to be fixed powers of two greatly simplifies the implemention of the division operations as well as the address calculations for the tile map and tile set memories. Square tiles are also natural for designing graphics, although they are a little awkward for Roman letters.

Next, we need to determine the exact number of bits for data and memory in Fig. 11. For VGA, x ranges from 0 to 639, which we will represent as a 10-bit binary number whose bits we will write $x_9\, x_8 \cdots x_0$ (little-endian subscripts). Similarly, y ranges from 0 to 479, which takes 9 bits $y_8\, y_7 \cdots y_0$. Note that these are active screen coordinates; VCOUNT actually uses 10 bits because it needs to count to 524.

Because $640 \div 8 = 80$ and $480 \div 8 = 60$, c we need 7 bits for c: $c_6\, c_5 \cdots c_0$, and 6 bits for r: $r_5\, r_4 \cdots r_0$. Furthermore, the tile local coordinates (i, j) will each be 3 bits since each range over 0–7.

Thanks to our choice of 8-pixel-square tiles, dividing by the tile size amounts to shifting right by 3 bits. Calculating the tile column and row (c, r) along with the tile-local pixel coordinates (i, j) amounts to splitting each pixel coordinate into a 3-bit local tile coordinate and a 7- or 6-bit tile column and row:

$$(\underbrace{x_9\, x_8 \cdots x_3}_{c_6\, c_5 \cdots c_0}\ \underbrace{x_2\, x_1\, x_0}_{i_2\, i_1\, i_0},\ \underbrace{y_8\, y_7 \cdots y_3}_{r_5\, r_4 \cdots r_0}\ \underbrace{y_2\, y_1\, y_0}_{j_2\, j_1\, j_0}). \tag{1}$$

The tile map needs to hold an 80×60 array, so its entries could be addressed as $c + 80r$, but this requires a constant multiplication and addition. While this could be done with three additions because $80 = 64 + 16$, it is easier to round 80 up to the next power of two: 128. This wastes some memory ($128 \times 64 = 8192$ versus $80 \times 60 = 4800$), but this memory is a small fraction of the total available on the FPGA, which contains minimum-sized memory chunks, anyway. Also, 4800 is greater than 4096, which was the previous natural power-of-two size for the memory.

To support 256 tiles with a 128×64 tile map, the tile number t will be 8 bits, the address to tile map memory will be 13 bits ($128 \times 64 = 8192 = 2^{13}$), and the address for the tile map memory will consist of the 6 r bits for the MSBs followed by the 7 c bits:

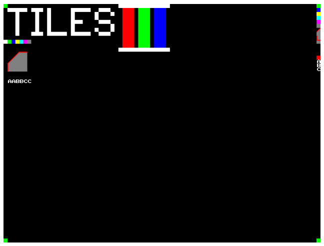
$$t_7\, t_6 \cdots t_0 = \text{tilemap}[r_5\, r_4 \cdots r_0\, c_6\, c_5 \cdots c_0]. \tag{2}$$

Putting the column number in the least significant bits gives a row-major layout: tiles in a row appear in successive memory locations; tiles in the row below appear in memory after all the tiles for the row above.

Pac-Man used 4-bit color codes; to store the color codes for 256 8×8 tiles takes a 4-bit-wide memory with $256 \times 8 \times 8 = 16384 = 2^{14}$ entries. Again, addressing this memory is easy because the tiles are a power of two: the least significant bits of the 15-bit address starts with 3 bits of i (the horizontal tile-local coordinate) followed by 3 bits of j (the vertical tile-local coordinate) followed by the 8 bit tile number.

$$c'_3\, c'_2\, c'_1\, c'_0 = \text{tileset}[t_7\, t_6 \cdots t_0\, j_2\, j_1\, j_0\, i_2\, i_1\, i_0]. \tag{3}$$

This is also a row-major layout: the pixels for a tile appear as eight rows of pixels; data for each tile starts at a multiple of 64 pixels.



(a) Image

```
02 00 00 00 00 00 00 00 00 00 ... 00
00 01 01 01 01 01 00 01 01 01 ... 00
00 00 00 01 00 00 00 00 01 00 ... 00
00 00 00 01 00 00 00 00 01 00 ... 00
00 00 00 01 00 00 00 00 01 00 ... 00
00 00 00 01 00 00 00 00 01 00 ... 00
00 00 00 01 00 00 00 00 01 00 ... 00
00 00 00 01 00 00 00 01 01 01 ... 00
00 00 00 00 00 00 00 00 00 00 ... 00
...
```

(b) Tilemap

```
...
00 00 00 0f 0f 0f 00 00
00 00 0f 0f 00 0f 0f 00
00 0f 0f 00 00 00 0f 0f
00 0f 0f 00 00 00 0f 0f
00 0f 0f 0f 0f 0f 0f 0f
00 0f 0f 00 00 00 0f 0f
00 0f 0f 00 00 00 0f 0f
00 00 00 00 00 00 00 00

00 0f 0f 0f 0f 0f 0f 00
00 0f 0f 00 00 00 0f 0f
00 0f 0f 00 00 00 0f 0f
00 0f 0f 0f 0f 0f 0f 00
00 0f 0f 00 00 00 0f 0f
00 0f 0f 00 00 00 0f 0f
00 0f 0f 0f 0f 0f 0f 00
00 00 00 00 00 00 00 00
...
```

(c) Tileset

```
00 00 00 00
FF 00 00 00
00 FF 00 00
00 00 FF 00
FF FF 00 00
00 FF FF 00
FF 00 FF 00
80 80 80 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
FF FF FF 00
```

(d) Palette

Figure 12: (a) Image generated by *tiles2ppm*. (b) The Tilemap file: 02 is a green square; the 01s (white squares) spell out "TI." (c) "A" and "B" tiles in the Tileset. (d) The 16-color palette (black, red, green, blue, yellow, etc.). Each 24-bit value is padded to be 32-bit aligned.

6 A Software Prototype for the Tile Generator

Complex hardware is always designed by creating an increasingly detailed series of models. The first is very abstract (e.g., a statement like "let's display graphics with tiles"); the last is the implementation itself. At each step, earlier models guide the implemention of the next, details are added, and each model is checked for conformance with the last. Fig. 10 is one such model of our tile generator; it will guide our eventual System Verilog model that we will use to implement the circuit on an FPGA.

Designers often use an executable software model as part of the development process. It can be very abstract and only model the algorithm, it can be a very detailed "cycle-accurate" model that models clock-by-clock hardware behavior, or something in between.

Fig. 13 shows an abstract software model coded in C designed to verify the "algorithm": effectively Fig. 10 and equations (1), (2), and (3). It is also useful for testing binary palette, tileset, and tilemap data, the filenames for which are supplied on the command line. It runs the algorithm to generate a 640 × 480 image and writes the result as a PPM file,[1] an uncompressed image format suitable for previewing. The body of the nested *for* loops in *main*() implement the rules in Fig. 11; all the rest is testbench scaffolding.

Fig. 12 shows the output of this program for a test tilemap, tileset, and palette. With a text editor, I created text files for each then converted them to binary using the *xxd* Linux command-line utility. The tilemap file is 64 lines of 128 bytes each; the tileset file is 256 tiles, each 8 lines of 8 bytes each (4 bits only); the palette file is 16 lines of four bytes each.

[1]https://netpbm.sourceforge.net/doc/ppm.html

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>

#define HACTIVE 640
#define VACTIVE 480

typedef struct { uint8_t red, green, blue; uint8_t _padto32; } rgb_t; // 24-bit color

void *mapfile(const char *filename, size_t length)
{
  int fd = open(filename, O_RDONLY);
  if (fd == -1)
    fprintf(stderr, "Error opening \"%s\": ", filename), perror(NULL), exit(1);
  void *p = mmap(NULL, length, PROT_READ, MAP_SHARED, fd, 0);
  if (p == MAP_FAILED)
    fprintf(stderr, "Error mapping \"%s\": ", filename), perror(NULL), exit(1);
  close(fd);
  return p;
}

int main(int argc, const char *argv[])
{
  if (argc != 4)
    fprintf(stderr, "Usage: tiles2ppm <tilemap> <tileset> <palette>\n"), exit(1);

  uint8_t *tilemap = (uint8_t *) mapfile(argv[1], 8192);
  uint8_t *tileset = (uint8_t *) mapfile(argv[2], 16384);
  rgb_t   *palette = (rgb_t *)   mapfile(argv[3], 16 * sizeof(rgb_t));

  printf("P3\n%d %d\n255\n", HACTIVE, VACTIVE); // Plain PPM header, 24 bpp

  uint16_t x, y;
  for (y = 0 ; y < VACTIVE ; y++)
    for (x = 0 ; x < HACTIVE ; x++) {             // The tile algorithm:
      uint8_t r     = y >> 3;                     // Row          0-59
      uint8_t c     = x >> 3;                     // Column       0-79
      uint8_t t     = tilemap[r << 7 | c];        // Tile number  0-255
      uint8_t i     = x & 0x7;                    // Tile local x 0-7
      uint8_t j     = y & 0x7;                    // Tile local y 0-7
      uint8_t color = tileset[t << 6 | j << 3 | i]; // Color        0-15
      rgb_t rgb     = palette[color];             // RGB color    24 bits
      printf("%d %d %d\n", rgb.red, rgb.green, rgb.blue);
    }

  return 0;
}
```

Figure 13: *tiles2ppm.c*: An untimed C model of the tile video generator that loads tilemap, tileset, and palette binary files using *mmap*() then calculates the color of each pixel and writes it as a PPM format file.

7 Tile Hardware Pipeline Design

From Fig. 10, the tile generator needs three small memories: tilemap, tileset, and palette. Each is a different size (both word size and number of words), but similar in that they both need to retrieve information for the video generator and we want to be able to read and write to them from software. Arbitrating memory access between CPU and graphics controller has long been a challenge, but our FPGA provides a convenient solution: the on-chip block RAMS are dual-ported, meaning they can perform two independent read or write operations every cycle, one in each clock domain. We will dedicate one port to the video generator (which will only read) and the other to the CPU (technically, an the Avalon agent bus interface coming from the HPS); each will run on their own clock.

We will use the parametric two-port BRAM module in Fig. 14 for these three memories. To instantiate it, you supply two compile-time two parameters: DATA_BITS and ADDRESS_BITS. The body is coded so that Quartus will marshal the appropriate collection of M10K blocks to implement it.

Fig. 15 shows a block diagram for the core of the VGA tile generator. This is Fig. 10 augmented with the VGA counters block, second memory ports for software access to the three synchronous block RAMS, and pipeline registers on the HCOUNT, BLANK, and HS signals.

The block diagram of Fig. 10 only models function, not timing; Fig. 15 models timing and adds pipeline registers to remain consistent. The block RAMS on the FPGA are synchronous: a read operation always takes a cycle, so the output of the tilemap memory comes a cycle after its address. Because the HCOUNT signal changes every pixel cycle, feeding HCOUNT directly to the tileset address generator would be inconsistent (it would be one higher than it should be, causing a graphical glitch), so I inserted a pipeline register on HCOUNT between the output of the VGA counter and the tileset address generator. Similar concerns apply to the BLANK and hs signals, which need to be generated in alignment with the pixel color data.

It would be natural to add pipeline registers on VCOUNT and VS, but because VCOUNT only changes at the end of each scanline, we can just use its unpipelined value without any change in behavior. Omitting pipeline registers on the VS signal does change its timing slightly, but the monitor ignores a 3 cycle (119 ns) delay on this 60 Hz signal.

Fig. 16 is the first half of code implementing the pipelined tile generator in Fig. 15. It consists of the module interface (clock and reset, VGA signals, and three memory ports), local wires and registers, and an instance of the VGA counter module from Fig. 5.

Fig. 17 is the second half of the tile generator hardware. It instantiates the three memories and implements the three groups of pipeline registers. The first port of each memory is used for video generation and operated in read-only mode.

The Tilemap address generator is just the expression { *vcount*[8:3], *hcount*[9:3] } fed to the tilemap's *addr1*, which is too small to warrant a separate module. Similarly, the Tileset address generator is the expression { *tilenumber*, *vcount*[2:0], *hcount1* }.

```
module twoportbram
  #(parameter int DATA_BITS = 8,  ADDRESS_BITS = 10)
   (input logic                     clk1,  clk2,
    input logic [ADDRESS_BITS-1:0] addr1, addr2,
    input logic [DATA_BITS-1:0]    din1,  din2,
    input logic                     we1,   we2,
    output logic [DATA_BITS-1:0]   dout1, dout2);

   localparam WORDS = 1 << ADDRESS_BITS;

   /* verilator lint_off MULTIDRIVEN */
   logic [DATA_BITS-1:0]           mem [WORDS-1:0];
   /* verilator lint_on MULTIDRIVEN */

   always_ff @(posedge clk1)
     if (we1) begin
        mem[addr1] <= din1;
        dout1 <= din1;
     end else dout1 <= mem[addr1];

   always_ff @(posedge clk2)
     if (we2) begin
        mem[addr2] <= din2;
        dout2 <= din2;
     end else dout2 <= mem[addr2];

endmodule
```

Figure 14: *twoportbram.sv*: Parametric two-port synchronous BRAM module for implementing the tilemap, tileset, and palette memories. The second port will enable software access to the memory. Note that each port has its own clock.

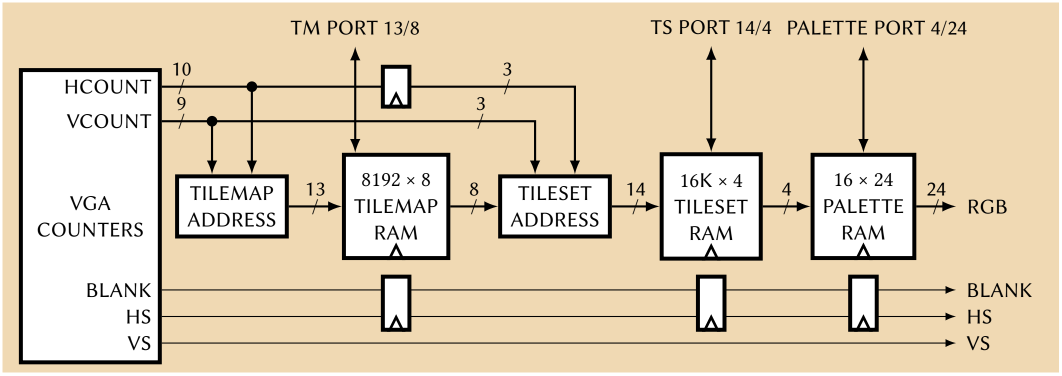


Figure 15: A pipelined 8 × 8 tile VGA generator with ports for accessing the tilemap, tileset, and palette memories and a BLANK signal needed by the VGA DAC.

```
module tiles
  (input logic         VGA_CLK, VGA_RESET,
   output logic [7:0]  VGA_R, VGA_G, VGA_B,
   output logic        VGA_HS, VGA_VS, VGA_BLANK_n,

   input logic         mem_clk,         // Clock for memory ports

   input logic [12:0]  tm_address,      // Tilemap memory port
   input logic         tm_we,
   input logic [7:0]   tm_din,
   output logic [7:0]  tm_dout,

   input logic [13:0]  ts_address,      // Tileset memory port
   input logic         ts_we,
   input logic [3:0]   ts_din,
   output logic [3:0]  ts_dout,

   input logic [3:0]   palette_address, // Palette memory port
   input logic         palette_we,
   input logic [23:0]  palette_din,
   output logic [23:0] palette_dout);

   logic [9:0]         hcount;          // From counters
   logic [8:0]         vcount;

   logic [2:0]         hcount1;         // Pipeline registers
   logic               VGA_HS0, VGA_HS1, VGA_HS2;
   logic               VGA_BLANK_n0, VGA_BLANK_n1, VGA_BLANK_n2;

   logic [7:0]         tilenumber;      // Memory outputs
   logic [3:0]         colorindex;

   /* verilator lint_off UNUSED */
   logic               unconnected; // Extra vcount bit from counters
   /* verilator lint_on UNUSED */
```

Figure 16: *tiles.sv*: The tile video generator (part 1/2). This interface takes in the VGA pixel clock and generates the VGA color and synchronization signals. It also exposes read/write memory ports for the tilemap, tileset, and palette.

```
  vga_counters cntrs(.vcount( {unconnected, vcount} ), // VGA Counters
                     .VGA_BLANK_n( VGA_BLANK_n0 ),
                     .VGA_HS( VGA_HS0 ),
                     .*);

  twoportbram #(.DATA_BITS(8), .ADDRESS_BITS(13))  // Tile Map
  tilemap(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
          .addr1 ( { vcount[8:3], hcount[9:3] } ),
          .we1   ( 1'b0 ), .din1( 8'h X ), .dout1( tilenumber ),
          .addr2 ( tm_address ),
          .we2   ( tm_we ), .din2( tm_din ), .dout2( tm_dout ));

  always_ff @(posedge VGA_CLK)                     // Pipeline registers
    { hcount1, VGA_BLANK_n1, VGA_HS1 } <=
      { hcount[2:0], VGA_BLANK_n0, VGA_HS0 };

  twoportbram #(.DATA_BITS(4), .ADDRESS_BITS(14))  // Tile Set
  tileset(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
          .addr1 ( { tilenumber, vcount[2:0], hcount1 } ),
          .we1   ( 1'b0 ), .din1( 4'h X), .dout1( colorindex ),
          .addr2 ( ts_address ),
          .we2   ( ts_we ), .din2( ts_din ), .dout2( ts_dout ));

  always_ff @(posedge VGA_CLK)                     // Pipeline registers
    { VGA_BLANK_n2, VGA_HS2 } <= { VGA_BLANK_n1, VGA_HS1 };

  twoportbram #(.DATA_BITS(24), .ADDRESS_BITS(4))  // Palette
  palette(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
          .addr1 ( colorindex ),
          .we1   ( 1'b0 ), .din1( 24'h X ), .dout1( { VGA_B, VGA_G, VGA_R } ),
          .addr2 ( palette_address ),
          .we2   ( palette_we ), .din2( palette_din ), .dout2( palette_dout ));

  always_ff @(posedge VGA_CLK)                     // Pipeline registers
    { VGA_BLANK_n, VGA_HS } <= { VGA_BLANK_n2, VGA_HS2 };

endmodule
```

Figure 17: *tiles.sv*: The tile video generator (part 2/2). This instatiates the VGA counters and connects the three memories and pipeline register groups following Fig. 15. Note that the memory ports have their own clock.

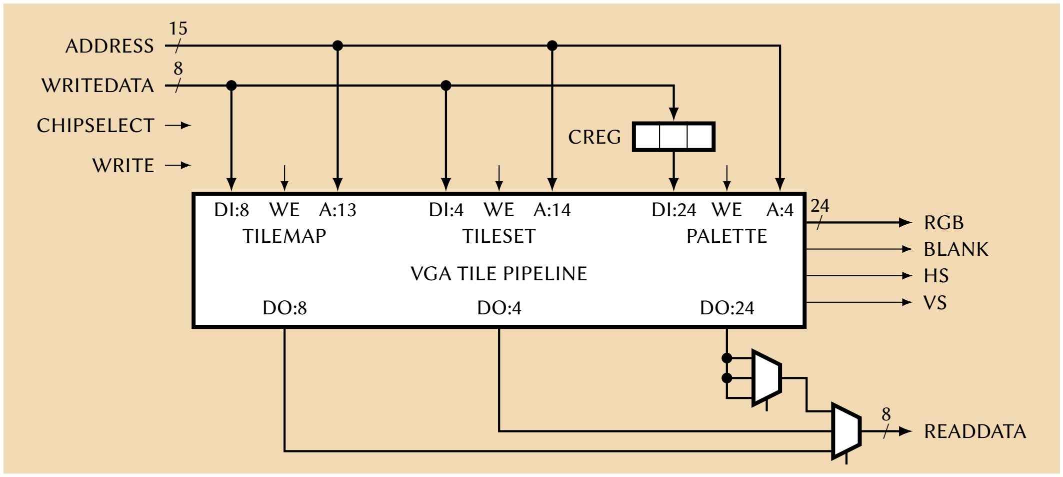


Figure 18: The VGA tile generator component with an Avalon MM Agent interface

a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
0	0	m_{12}	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0	Tilemap
0	1	0	0	0	0	0	0	0	p_3	p_2	p_1	p_0	b_1	b_0	Palette
1	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0	Tileset

(a) Address Encoding: the a are bus (byte) address bits; m are Tilemap address bits; p are Palette (color number) bits, b are "byte" bits, and the s are Tileset address bits.

Byte Offset	**Region**	**Contents**
`0000 - 1FFF`	Tilemap	8K, 8 bit tile numbers
`2000 - 203F`	Palette	64 bytes, 4 bytes per 24-bit color
`4000 - 7FFF`	Tileset	16K, 4 bit color numbers (LSBs)

(b) Memory Map

	0	1	2	3
0	red 0	green 0	blue 0	write 0
4	red 1	green 1	blue 1	write 1
		⋮		
60	red 15	green 15	blue 15	write 15

Reading from red n/green n/blue n returns the component from color n in memory.

Writing to red n/green n/blue n stores the component in *creg*; n ignored.

Reading from write n returns 0.

Writing to write n writes *creg* to color n; the written value is ignored.

(c) Bytes in the Palette region

Figure 19: Software model of the tile component

8 Implementing VGA Tiles: an Avalon MM Agent

We want to control the tile generator from software, so we will create a Platform Designer component for the tile generator that includes an Avalon Memory-Mapped Agent port. The basic challenge is adapting this protocol to the three memory port interfaces in Fig. 17.

Fig. 18 shows the datapath for the interface. We will choose an 8-bit wide bus interface because the tilemap memory is 8 bits wide and the tileset memory is 4 bits wide. The 24-bit wide palette memory is a little problematic, but this is easier to deal with than adapting the software to work with a 32-bit interface.

The tilemap memory port is the easiest to connect: the address and data-in bits can come straight from the bus. When the software reads from this memory, the mux in the lower right will route its data-out bits to *readdata* on the interface.

The tileset memory is almost as easy: the address bits can come straight from the bus. We only need 4 data bits, so we can take those from the lowest 4 bus bits. This may seem like a waste, but the extra bits are never stored in the tileset memory.

The 24-bit-wide palette memory is more challenging. While the BRAMs have the ability to have different width ports, we will solve it by allowing the software to write a byte at a time to a single 24-bit register (*creg* in Fig. 18) that can ultimately be written to the palette memory. Reading palette memory gives 3 bytes; another mux selects one of them.

Fig. 19 shows the software model, which will dictate the datapath control for Fig. 18.

Fig. 19(a) shows the encoding of bus addresses. We have three memory regions: the 8K Tilemap; the 16K Tileset, and the 16-entry Palette. We start with the largest region (Tileset), which requires 14 address bits, and add a leading 1 to distinguish it from the other regions. The Tilemap region only requires 13 address bits, so we set its top two address bits to 00. This leaves the code 01, which will select the palette. Because each color is 24 bits, we need at least three byte addresses per color, but it is easier to use a power of two, so we will use 4 bytes per color. The two least significant bits will indicate the color component (byte) and the next four will select the color number (0–15).

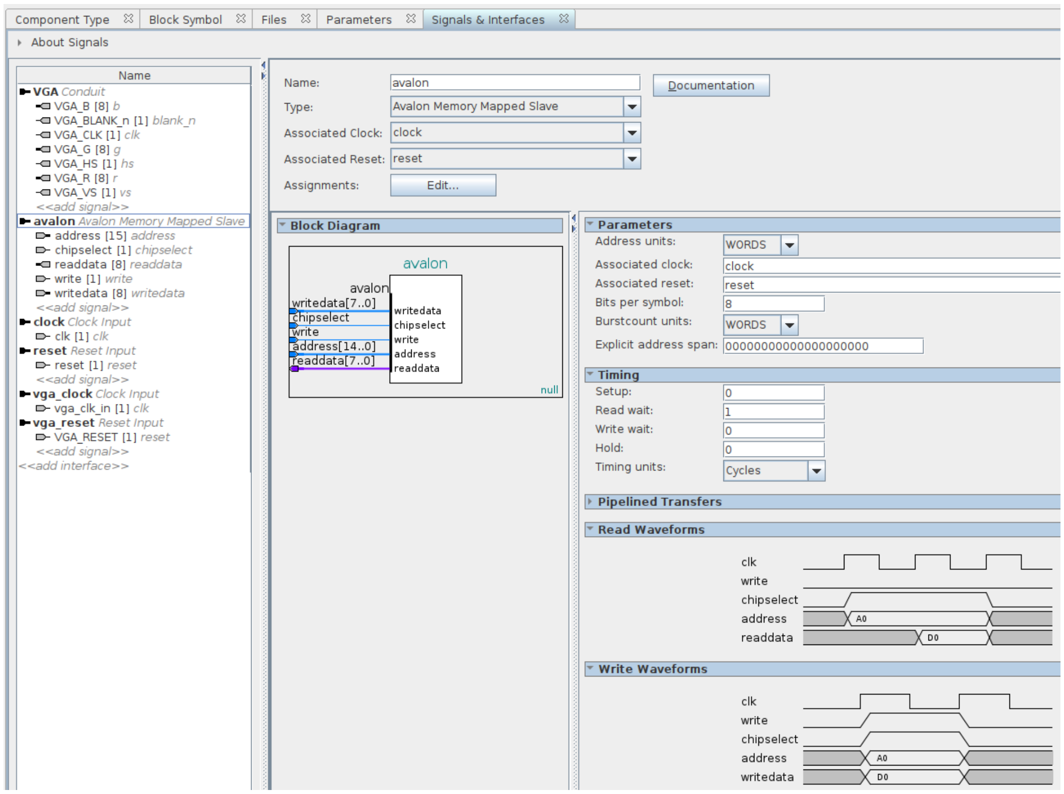
Fig. 19(b) is a memory map, essentially the address encoding in hexadecimal.

To make it possible to write palette colors into memory sequentially one byte at a time, writing to the first three bytes of each four-byte color only writes into one of the three bytes of the *creg* register, not to palette memory. When the fourth byte is written (addresses 3, 7, 11, …), the 24-bit *creg* is written to palette memory at the byte address shifted right 2 bits. Fig. 19(c) depicts the byte-by-byte layout of the palette region.

Fig. 20 shows the System Verilog code for the tile component in Fig. 15. The combinational block performs address decoding by examining *chipselect* and address bits. Avalon's single-cycle writes demands this be combinational: the BRAM write enable signals must appear in the same cycle as *write*. The multiplexers that steer data to the *readdata* port are implicit in the various assignments to *readdata*. The sequential block implements creg; the bits of *creg_write* select which byte is loaded from *writedata* on a write to palette addresses.

```
module vga_tiles
  (input logic        clk, reset,                    // Avalon MM Agent port
   input logic        chipselect, write,             // read == chipselect & !write
   input logic [14:0] address,                       // 32K window
   input logic [7:0]  writedata,                     // 8-bit interface
   output logic [7:0] readdata,

   input logic        vga_clk_in, VGA_RESET,         // VGA signals
   output logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n);

   logic [2:0]        creg_write;                    // Latch enable per byte
   logic              tm_we, ts_we, palette_we;      // Memory write enables
   logic [7:0]        tm_dout;                       // Data from tilemap
   logic [3:0]        ts_dout;                       // Data from tileset
   logic [23:0]       creg, palette_dout;            // Data to/from palette

   tiles tiles(.mem_clk        ( clk            ),
               .tm_address     ( address[12:0] ), .tm_din     ( writedata      ),
               .ts_address     ( address[13:0] ), .ts_din     ( writedata[3:0] ),
               .palette_address( address[5:2]  ), .palette_din( creg           ), .*);
   assign VGA_CLK = vga_clk_in;

   always_comb begin                                  // Address Decoder
      {tm_we, ts_we, palette_we, creg_write, readdata } = { 6'b 0, 8'h xx };
      if (chipselect)
        if (address[14] == 1'b 1) begin               // Tileset 1--------------
           ts_we    = write;                          //  Write to tileset mem
           readdata = { 4'h 0, ts_dout };             //  Read lower 4 bits; pad upper
        end else if (address[13] == 1'b 0) begin      // Tilemap 00-------------
           tm_we    = write;                          //  Write to tilemap mem
           readdata = tm_dout;                        //  Read 8 bits
        end else if ( address[12:6] == 7'b 0_0000_00 ) // Palette 010000000------
           case (address[1:0])
             2'h 0 : begin readdata = palette_dout[7:0];   // Read red byte
                           creg_write[0] = write;          // creg <- red
                     end
             2'h 1 : begin readdata = palette_dout[15:8];  // Read green byte
                           creg_write[1] = write;          // creg <- green
                     end
             2'h 2 : begin readdata = palette_dout[23:16]; // Read blue byte
                           creg_write[2] = write;          // creg <- blue
                     end
             2'h 3 : begin readdata = 8'h 00;              // Always reads as 00
                           palette_we = write;             // mem <- creg
                     end
           endcase
   end

   always_ff @(posedge clk or posedge reset)
     if (reset) creg <= 24'b 0; else begin
        if (creg_write[0]) creg[7:0]   <= writedata;    // Write byte (color)
        if (creg_write[1]) creg[15:8]  <= writedata;    // to creg according to
        if (creg_write[2]) creg[23:16] <= writedata;    // creg_write bits
     end
endmodule
```

Figure 20: *vga_tiles.sv*: The tile generator component, which adds an Avalon MM Agent port to the video generation hardware



Name
VGA *Conduit*
VGA_B [8] *b*
VGA_BLANK_n [1] *blank_n*
VGA_CLK [1] *clk*
VGA_G [8] *g*
VGA_HS [1] *hs*
VGA_R [8] *r*
VGA_VS [1] *vs*
<<add signal>>
avalon *Avalon Memory Mapped Slave*
address [15] *address*
chipselect [1] *chipselect*
readdata [8] *readdata*
write [1] *write*
writedata [8] *writedata*
<<add signal>>
clock *Clock Input*
clk [1] *clk*
reset *Reset Input*
reset [1] *reset*
<<add signal>>
vga_clock *Clock Input*
vga_clk_in [1] *clk*
vga_reset *Reset Input*
VGA_RESET [1] *reset*
<<add signal>>
<<add interface>>

Name: avalon
Type: Avalon Memory Mapped Slave
Associated Clock: clock
Associated Reset: reset
Assignments:

Block Diagram: avalon — writedata[7..0] → writedata; chipselect → chipselect; write → write; address[14..0] → address; readdata[7..0] ← readdata

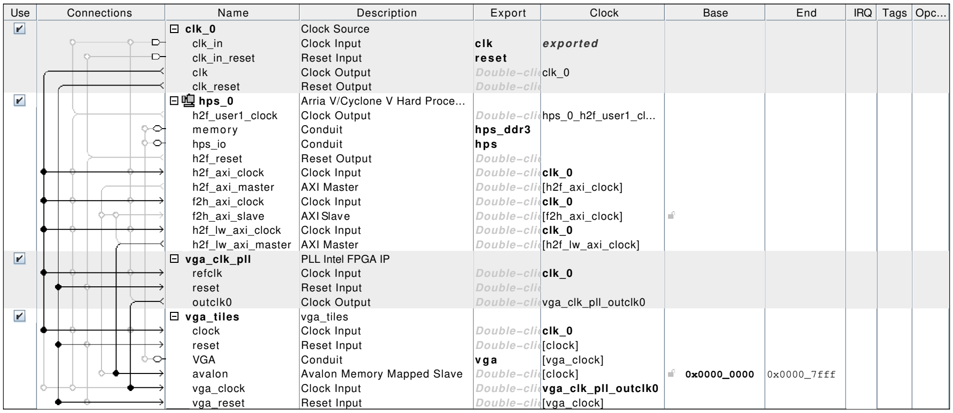
Parameters

Address units:	WORDS
Associated clock:	clock
Associated reset:	reset
Bits per symbol:	8
Burstcount units:	WORDS
Explicit address span:	00000000000000000000

Timing

Setup:	0
Read wait:	1
Write wait:	0
Hold:	0
Timing units:	Cycles

Figure 21: Signals & Interfaces for the *vga_tiles* component. This shows the configuration of the Avalon MM Agent port.



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opc...
✓		**clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-clic*	clk_0					
		clk_reset	Reset Output	*Double-clic*						
✓		**hps_0**	Arria V/Cyclone V Hard Proce...							
		h2f_user1_clock	Clock Output	*Double-clic*	hps_0_h2f_user1_cl...					
		memory	Conduit	**hps_ddr3**						
		hps_io	Conduit	**hps**						
		h2f_reset	Reset Output	*Double-clic*						
		h2f_axi_clock	Clock Input	*Double-clic*	**clk_0**					
		h2f_axi_master	AXI Master	*Double-clic*	[h2f_axi_clock]					
		f2h_axi_clock	Clock Input	*Double-clic*	**clk_0**					
		f2h_axi_slave	AXI Slave	*Double-clic*	[f2h_axi_clock]					
		h2f_lw_axi_clock	Clock Input	*Double-clic*	**clk_0**					
		h2f_lw_axi_master	AXI Master	*Double-clic*	[h2f_lw_axi_clock]					
✓		**vga_clk_pll**	PLL Intel FPGA IP							
		refclk	Clock Input	*Double-clic*	**clk_0**					
		reset	Reset Input	*Double-clic*						
		outclk0	Clock Output	*Double-clic*	vga_clk_pll_outclk0					
✓		**vga_tiles**	vga_tiles							
		clock	Clock Input	*Double-clic*	**clk_0**					
		reset	Reset Input	*Double-clic*	[clock]					
		VGA	Conduit	**vga**	[vga_clock]					
		avalon	Avalon Memory Mapped Slave	*Double-clic*	[clock]	**0x0000_0000**	0x0000_7fff			
		vga_clock	Clock Input	*Double-clic*	**vga_clk_pll_outclk0**					
		vga_reset	Reset Input	*Double-clic*	[vga_clock]					

Figure 22: The Platform Designer system for the tile generator

9 Making a Platform Designer System

I copied the *Makefile*, *soc_system.tcl*, *soc_system.qsys*, and *soc_system.srf* from Lab 3 and ran Platform Designer (*qsys-edit*).

I added Fig. 20 as an IP component in Platform Designer to connect it to the HPS. I followed the procedure in Lab 3: create a new *vga_tiles* component; add the *vga_tiles.sv*, *tiles.sv*, *vga_counters.sv*, and *twoportbram.sv* files; add a *VGA* conduit; move and rename the VGA signals to it; and ensure its read wait timing was set to 1, to allow time to read the synchronous BRAMs and its write wait to 0 (the defaults). Fig. 21 shows the signals.

I added the following lines to *vga_tiles_hw.tcl* to add the component to the device tree:

```
set_module_assignment embeddedsw.dts.group vga
set_module_assignment embeddedsw.dts.name vga_tiles
set_module_assignment embeddedsw.dts.vendor csee4840
```

Unlike Lab 3, this VGA video system runs at the VGA pixel clock frequency, which I generated with one of the six phased-locked loops (PLLs) on our FPGA. These are clever digital/analog circuits that drive a voltage-controlled oscillator from a phase detector in a feedback loop with a programmable clock divider M. The reference clock passes through a divider N before going to the phase detector; the output passes through another divider C. This produces a clock whose frequency is the reference clock's multiplied by $M \div (N \times C)$, where M, N, and C are integers (1–512). Our FPGA has a 50 MHz clock, and we want 25.175 MHz. Giving Platform Designer these two frequencies, it suggests 50 MHz × 215 ÷ (7 × 61) = 25.175644 MHz, which is close enough for a monitor.

I added the PLL and tile components and connected them as shown in Fig. 22. Note that the *refclk* for the PLL comes from the 50 MHz clock from the *clk_50* Clock Source and that the PLL's *outclk0* is sent to both the tile component and the *h2f_lw_axi_clock* input: the clock for the lightweight bus bridge on the HPS that is connected to the Avalon MM agent port on the Tile component. I also exported the VGA conduit to make a connection.

The rather fragile *sopc2dts* program does not recognize the *altera_pll* component and the clock it generates, so edit the *clocks* line in the *vga_tiles* component *soc_system.dts* file to remove the reference to *vga_clk_pll*, i.e.,

```
clocks = <&clk_0>;
```

10 Testing the VGA Tiles Component with U-Boot

While this component is designed to work with Linux, it is convenient to test it separately from device drivers. U-Boot, the first-stage bootloader, can do this since it provides a low-level command-line interface capable of writing to and reading from memory.

I copied the *soc_system.rbf* file over to the boot partition on the SD card and started booting Linux, which starts with loading the *rbf* file to the FPGA. This gave a blank display. Before Linux started completely, I rebooted by pressing the *HPS reset* button the board and pressed a key when prompted to *Hit any key to stop autoboot*.

First, I set a base address, which is added to all address specifications to reduce typing:

```
SOCFPGA_CYCLONE5 # base ff200000
Base Address: 0xff200000
```

Three commands are convenient for reading and modifying memory: *md.b*, which displays a range of memory; *mw.b*, which writes a single byte or a range of identical bytes to memory; and *mm.b*, which modifies a range of memory by allowing you to enter a new byte at each address.

The palette region (at offset 2000) should not change until a fourth byte is written. Note that in the below, the three color component values at 2000, 2001, and 2002 are not written until 2003 is written. This changed the screen to magenta since I modified color 0. In the tileset, tile 0 is all color 0, and the tilemap is all tile 0.

```
SOCFPGA_CYCLONE5 # md.b 2000 40
ff202000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
ff202010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
ff202020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
ff202030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
SOCFPGA_CYCLONE5 # mw.b 2000 ff
SOCFPGA_CYCLONE5 # md.b 2000 10
ff202000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
SOCFPGA_CYCLONE5 # mw.b 2001 80
SOCFPGA_CYCLONE5 # mw.b 2002 fe
SOCFPGA_CYCLONE5 # md.b 2000 10
ff202000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
SOCFPGA_CYCLONE5 # mw.b 2003 0
SOCFPGA_CYCLONE5 # md.b 2000 10
ff202000: ff 80 fe 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

I tested the Tilemap region (at offset 0) by writing a sequence of "random" bytes then verifying they could be read back.

```
SOCFPGA_CYCLONE5 # md.b 0 10
ff200000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
SOCFPGA_CYCLONE5 # mm.b 0
ff200000: 00 ? 1
ff200001: 00 ? 5
ff200002: 00 ? 10
ff200003: 00 ? 55
ff200004: 00 ? 43
ff200005: 00 ? 72
ff200006: 00 ? aa
ff200007: 00 ? ee
ff200008: 00 ? f0
ff200009: 00 ? 0f
ff20000a: 00 ? ff
ff20000b: 00 ? 00
ff20000c: 00 ? x
SOCFPGA_CYCLONE5 # md.b 0 10
ff200000: 01 05 10 55 43 72 aa ee f0 0f ff 00 00 00 00 00    ...UCr..........
```

I tested the Tileset region (at offset 4000) and verified only the top 4 bits of each byte were retained:

```
SOCFPGA_CYCLONE5 # mm.b 4000
ff204000: 00 ? 1
ff204001: 00 ? 2
ff204002: 00 ? 4
ff204003: 00 ? 8
ff204004: 00 ? a
ff204005: 00 ? F
ff204006: 00 ? 10
ff204007: 00 ? f0
ff204008: 00 ? ff
ff204009: 00 ? f1
ff20400a: 00 ? f3
ff20400b: 00 ? x
SOCFPGA_CYCLONE5 # md.b 4000 10
ff204000: 01 02 04 08 0a 0f 00 00 0f 01 03 00 00 00 00 00    ................
```

U-Boot is able to read into memory files from the FAT (Microsoft) filesystem on the SD card. The boot scripts do this as part of configuring the FPGA; it can also help test our peripheral. By design, the hardware uses the same data layout as the software prototype (Fig. 13) for the tilemap, tileset, and palette regions. In particular, palette colors are padded to align on 4-byte boundaries (see the software model in Fig. 19).

The boot partition on the SD card is `mmc 0:1` to U-Boot; *fatls* lists the files there:

```
SOCFPGA_CYCLONE5 # fatls mmc 0:1
   237800   u-boot.img
      226   u-boot.scr
  7007204   soc_system.rbf
    31245   soc_system.dtb
  4877224   zimage
       64   palette1.bin
     8192   tilemap1.bin
    16384   tileset1.bin
```

U-Boot supports environment variables. The *fpgadata* variable comes set an address to load file data. We will set three additional variables to the base of each region and stop using the *base* feature.

```
SOCFPGA_CYCLONE5 # tilemap=ff200000
SOCFPGA_CYCLONE5 # palette=ff202000
SOCFPGA_CYCLONE5 # tileset=ff204000
SOCFPGA_CYCLONE5 # base 0
Base Address: 0x00000000
```

The *fatload* command reads a file into memory; the *cp.b* command copies memory regions. Using this, we can load each of the binary files into memory, then into the peripheral:

```
SOCFPGA_CYCLONE5 # fatload mmc 0:1 $fpgadata palette1.bin
reading palette1.bin
64 bytes read in 4 ms (15.6 KiB/s)
SOCFPGA_CYCLONE5 # cp.b $fpgadata $palette 40

SOCFPGA_CYCLONE5 # fatload mmc 0:1 $fpgadata tilemap1.bin
reading tilemap1.bin
8192 bytes read in 6 ms (1.3 MiB/s)
SOCFPGA_CYCLONE5 # cp.b $fpgadata $tilemap 2000

SOCFPGA_CYCLONE5 # fatload mmc 0:1 $fpgadata tileset1.bin
reading tileset1.bin
16384 bytes read in 5 ms (3.1 MiB/s)
SOCFPGA_CYCLONE5 # cp.b $fpgadata $tileset 4000
```

Fig. 23 shows the image that appeared, which matched Fig. 12(a).

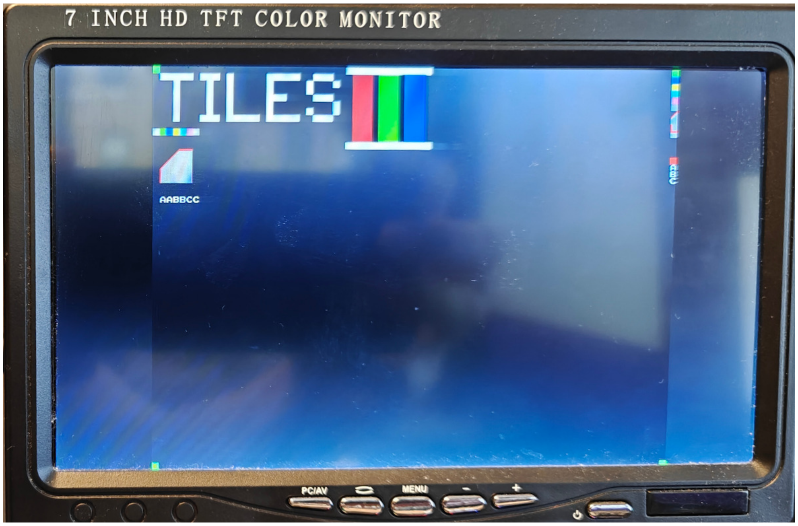


Figure 23: The tile generator displayed on a small VGA LCD monitor

Flow Status	Successful - Sun May 4 20:13:58 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	soc_system
Top-level Entity Name	soc_system_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	406 / 32,070 (1 %)
Total registers	598
Total pins	362 / 457 (79 %)
Total virtual pins	0
Total block memory bits	131,456 / 4,065,280 (3 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	1 / 4 (25 %)

Figure 24: Quartus compilation report. This confirms we used one PLL and the three memory regions (Tilemap, Tileset, and Palette) were implemented with block memory ($8K \times 8 + 16K \times 4 + 16 \times 24$). The logic use (406 ALMs) was minimal.