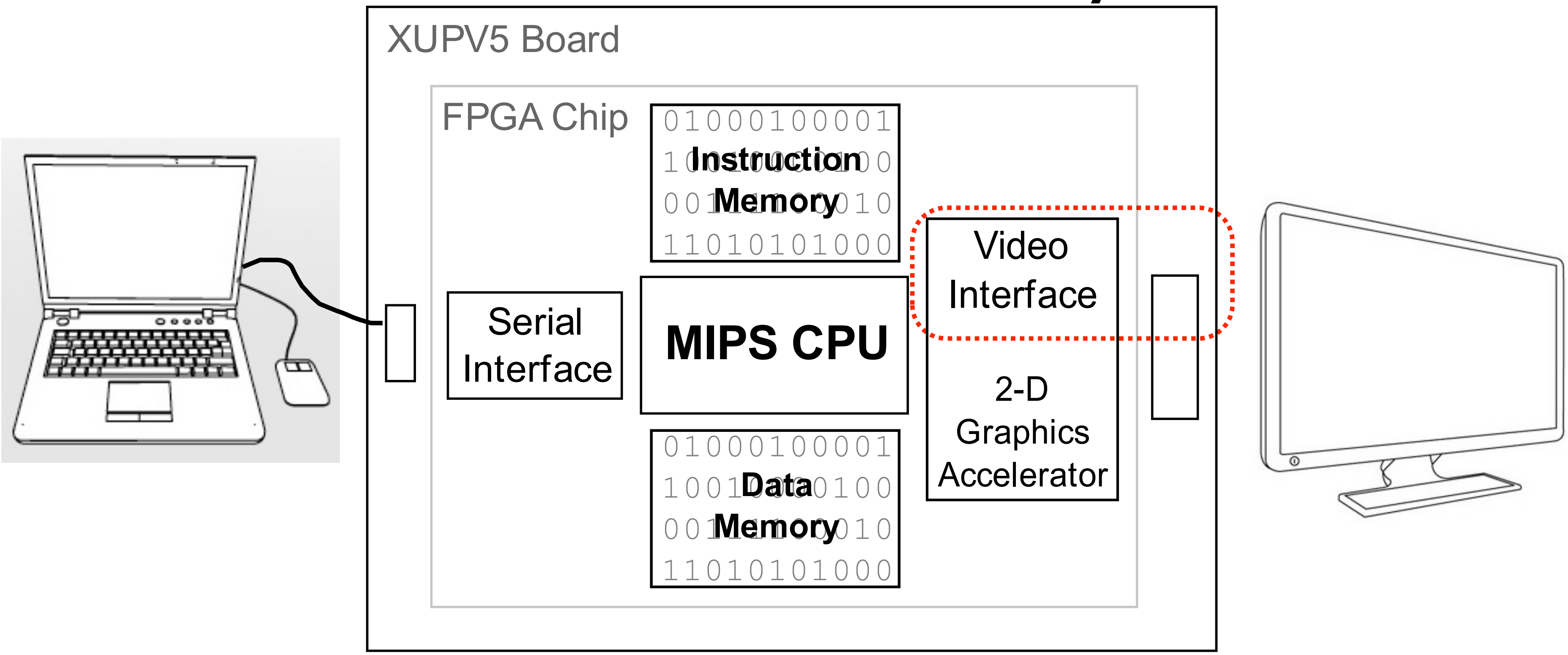
EECS150 - Digital Design

Lecture 15 - Video

March 6, 2011

John Wawrzynek

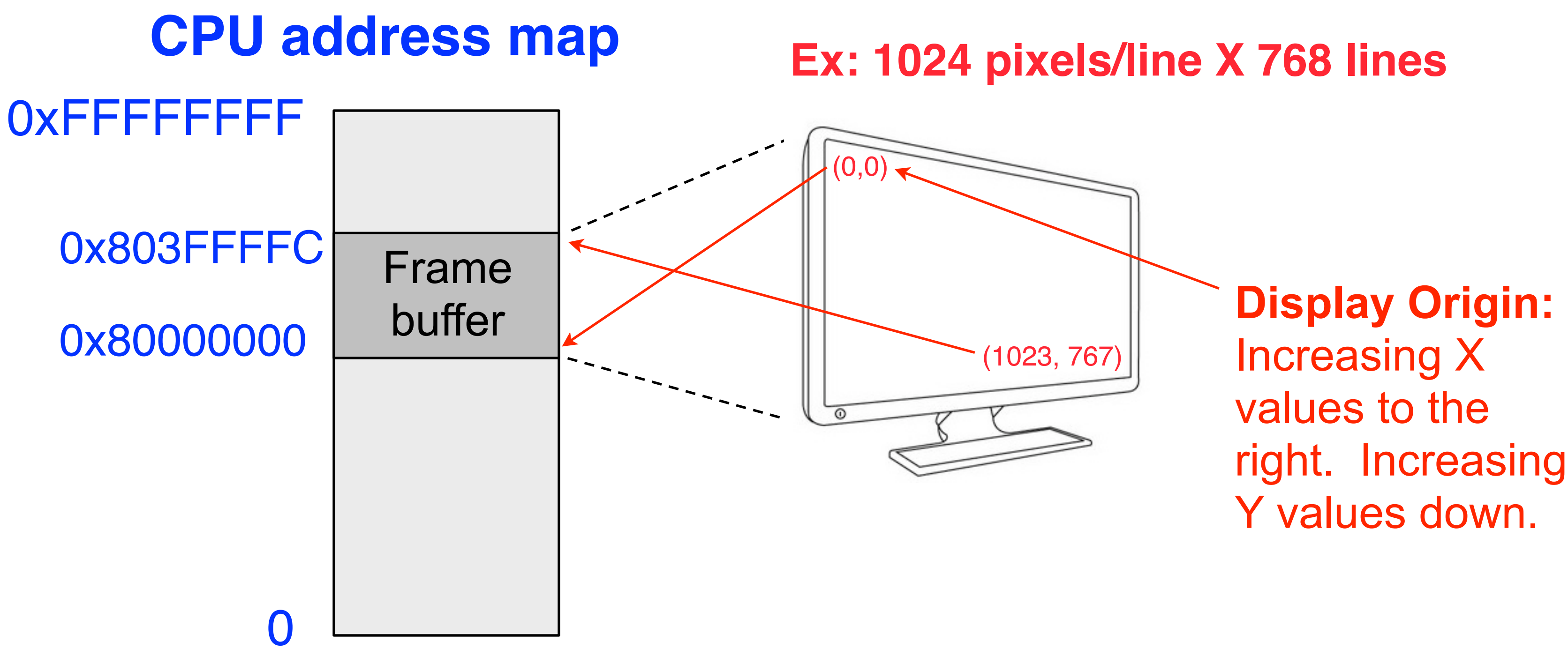
MIPS150 Video Subsystem



- Gives software ability to display information on screen.
- Equivalent to standard graphics cards:
  - Processor can directly write the display bit map
  - 2D Graphics acceleration

“Framebuffer” HW/SW Interface

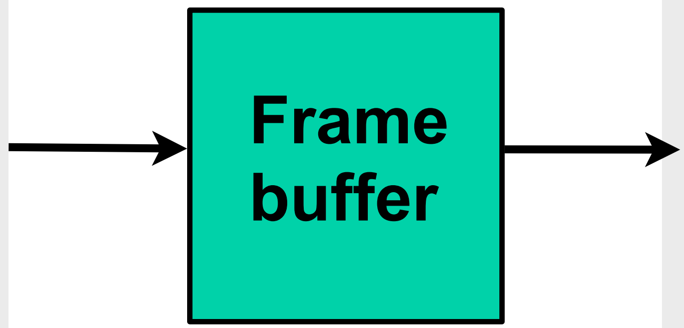
- A range of memory addresses correspond to the display.
- CPU writes (using sw instruction) pixel values to change display.
- No synchronization required. Independent process reads pixels from memory and sends them to the display interface at the required rate.



Framebuffer Implementation

- Framebuffer like a simple dual-ported memory. Two independent processes access framebuffer:

CPU writes pixel locations. Could be in random order, e.g. drawing an object, or sequentially, e.g. clearing the screen.

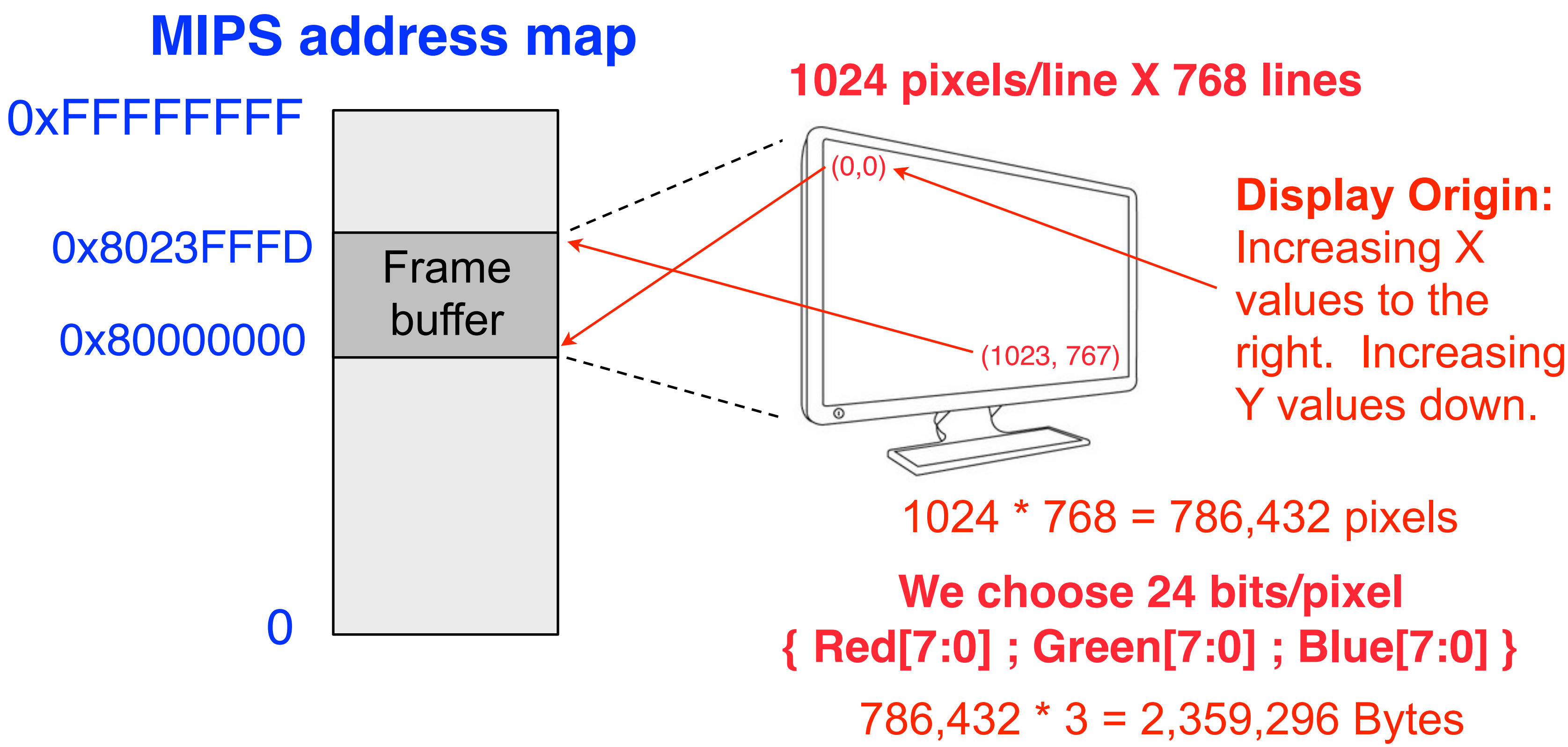


Video Interface continuously reads pixel locations in scan-line order and sends to physical display.

- How big is this memory and how do we implement it? For us:

1024 x 768 pixels/frame x 24 bits/pixel

Memory Mapped Framebuffer



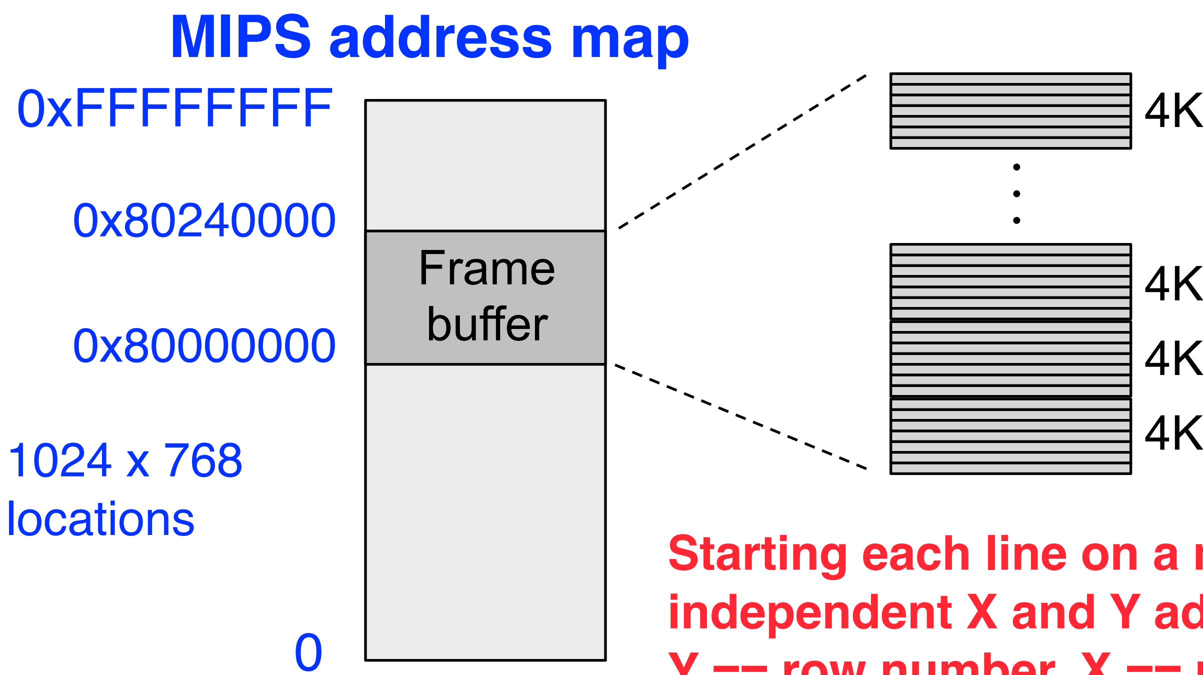
- Total memory bandwidth needed to support frame buffer?

Frame Buffer Implementation

- Which XUP memory resource to use?
- Memory Capacity Summary:
  - LUT RAM
  - Block RAM
  - External SRAM
  - External DRAM
- DRAM bandwidth:

Framebuffer Details

768 lines, 1024 pixels/line = 786,432 pixel locations



XUP DRAM memory capacity: 256 MBytes (in external DRAM).

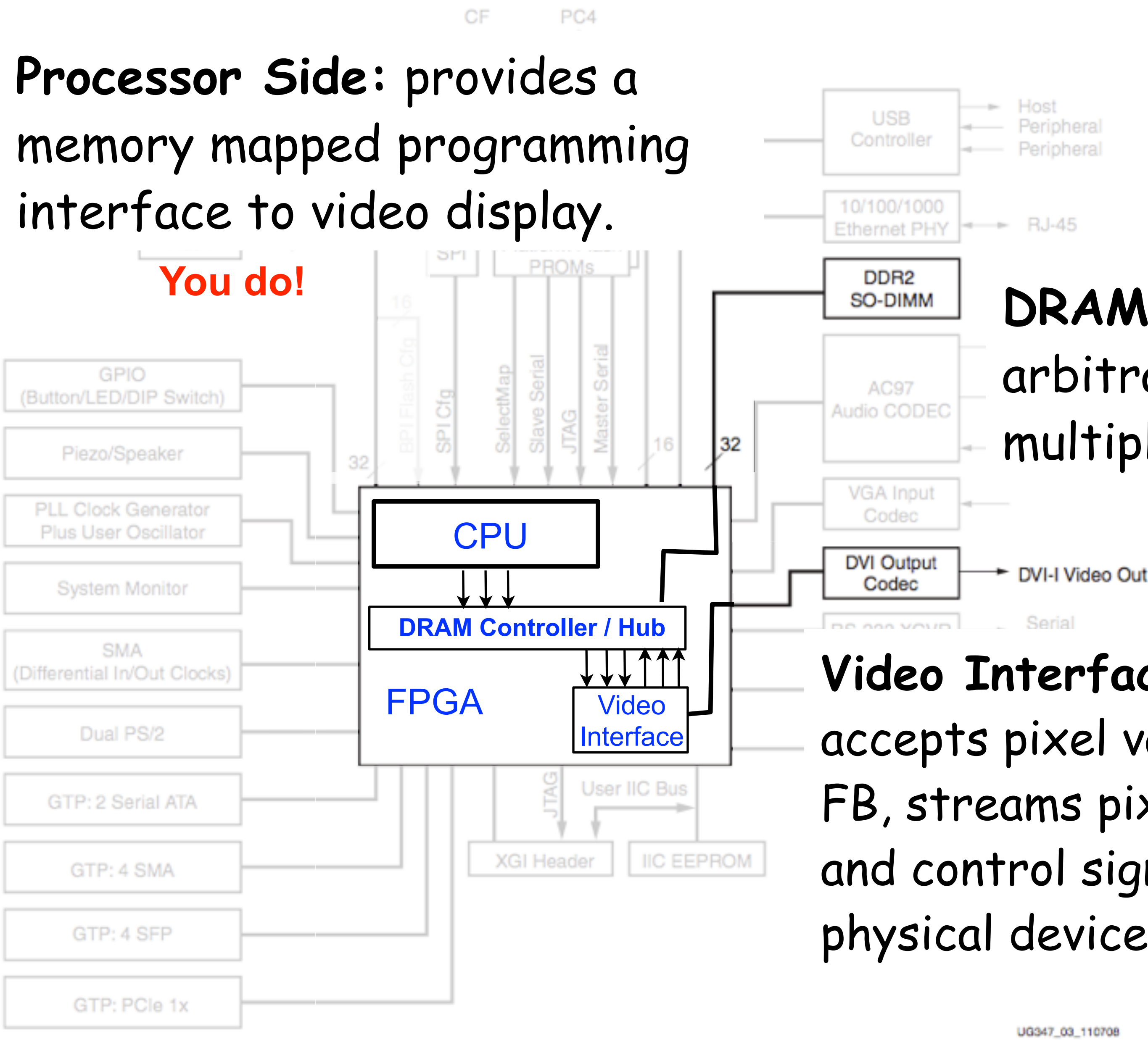
With Byte addressed memory, best to use 4 Bytes/ pixel

Starting each line on a multiple of 4K leads to independent X and Y address: {Y[9:0] ; X[11:2]}
Y == row number, X == pixel in row

Frame Buffer Physical Interface

Processor Side: provides a memory mapped programming interface to video display.

You do!



DRAM "Hub": arbitrates among multiple DRAM users.

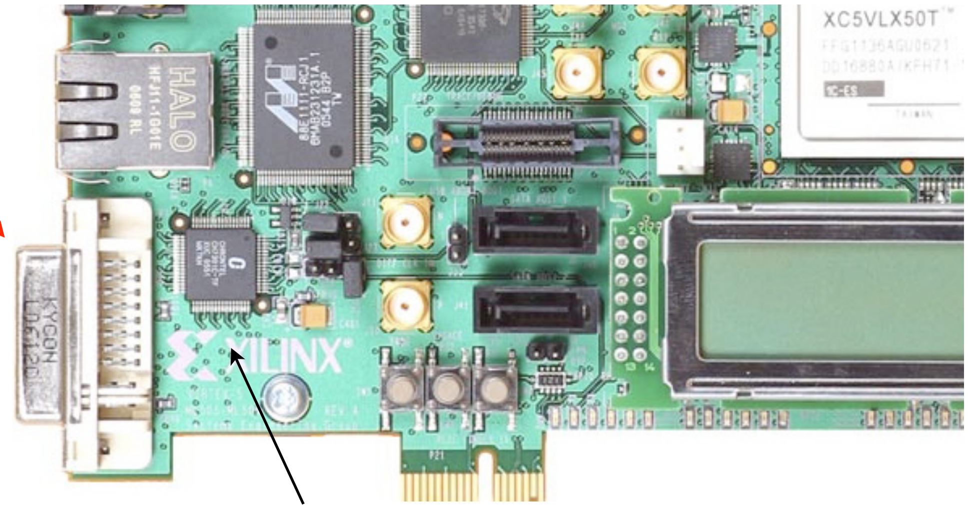
You do!

Video Interface Block: accepts pixel values from FB, streams pixels values and control signals to physical device.

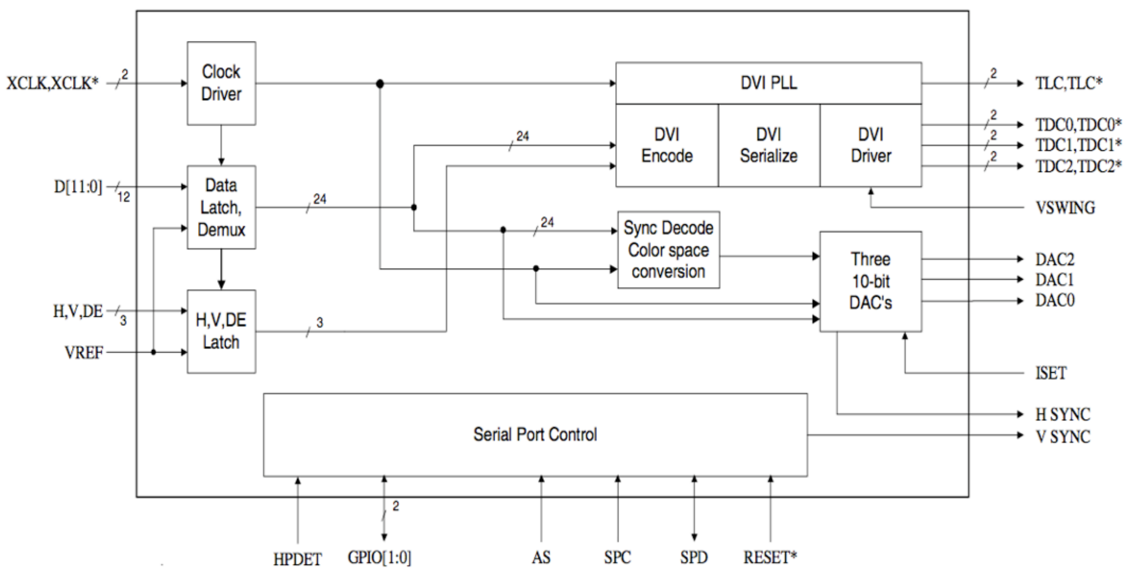
We do!

Physical Video Interface

DVI connector: accommodates analog and digital formats



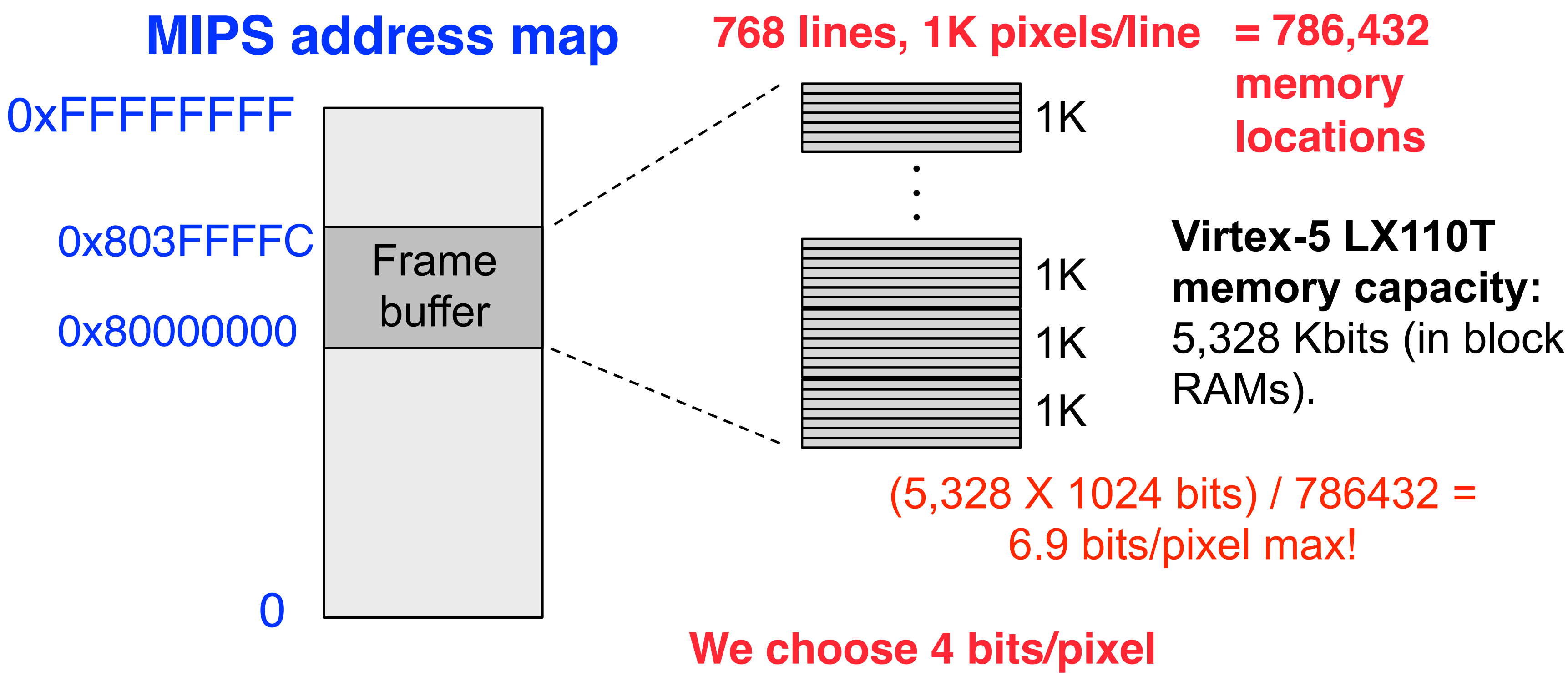
DVI Transmitter Chip, Chrontel 7301C.



Implements standard signaling voltage levels for video monitors. Digital to analog conversion for analog display formats.

Framebuffer Details 2009

- One pixel value per memory location.



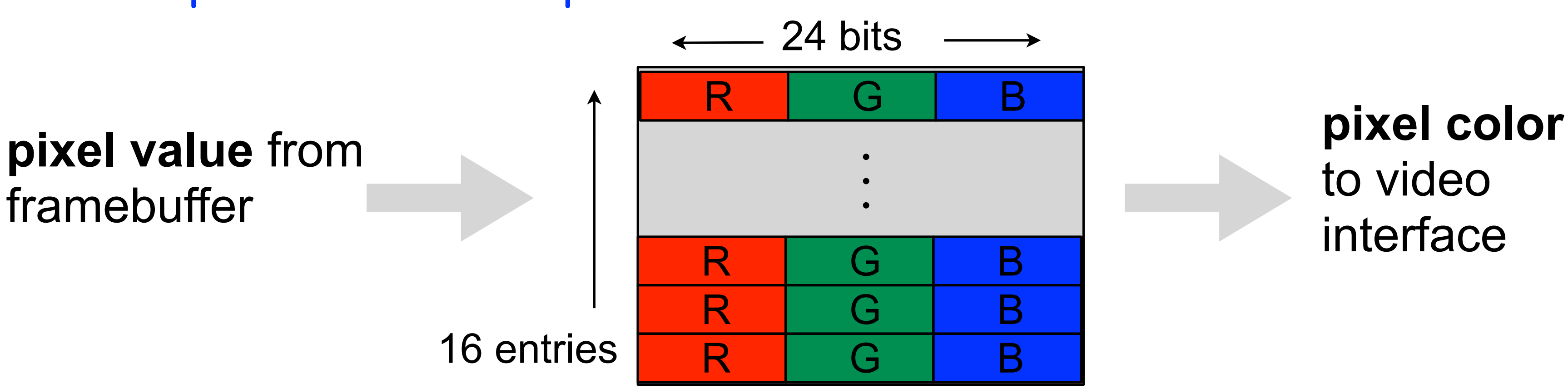
- Note, that with only 4 bits/pixel, we could assign more than one pixel per memory location. Ruled out by us, as it complicated software.

Color Map

4 bits per pixel, allows software to assign each screen location, one of 16 different colors.

However, physical display interface uses 8 bits / pixel-color. Therefore entire pallet is 2^{24} colors.

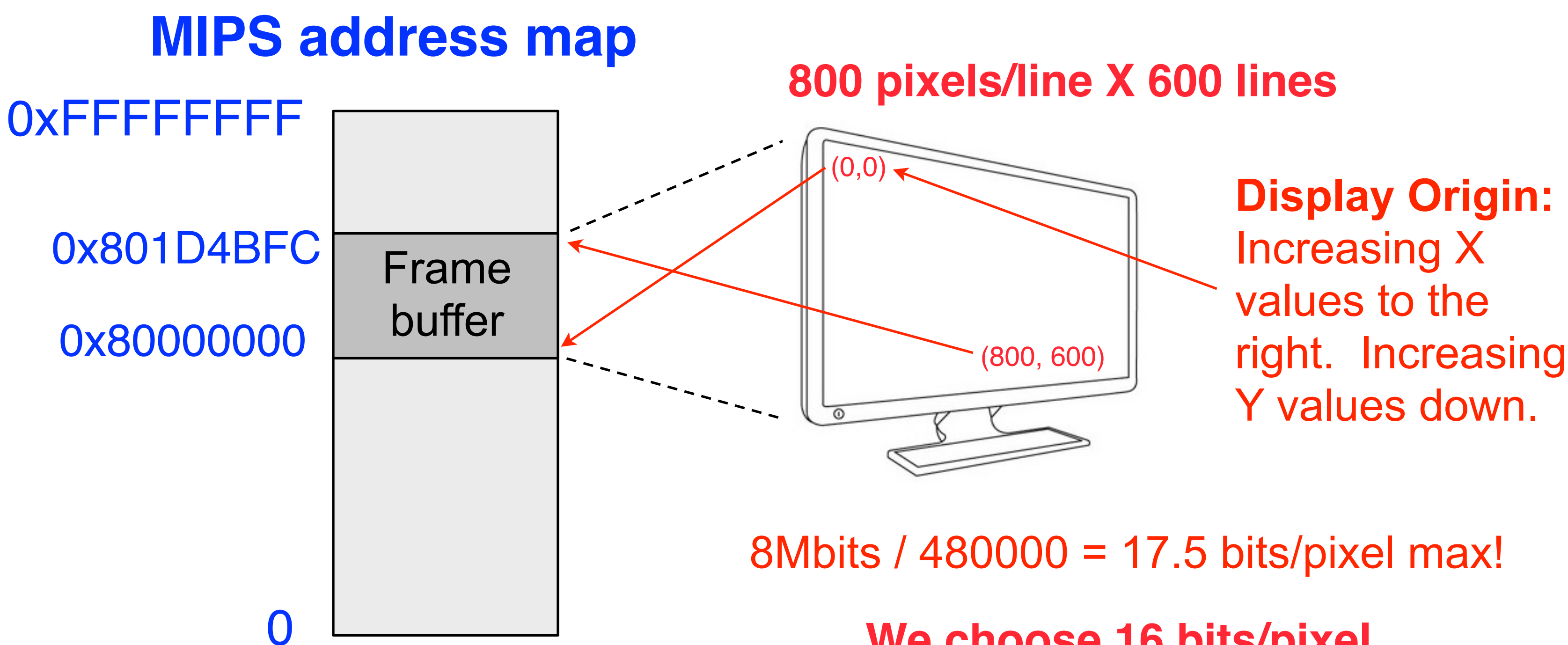
Color Map converts 4 bit pixel values to 24 bit colors.



Color map is memory mapped to CPU address space, so software can set the color table. Addresses: **`0x8040_0000`** **`0x8040_003C`**, one 24-bit entry per memory address.

Memory Mapped Framebuffer 2010

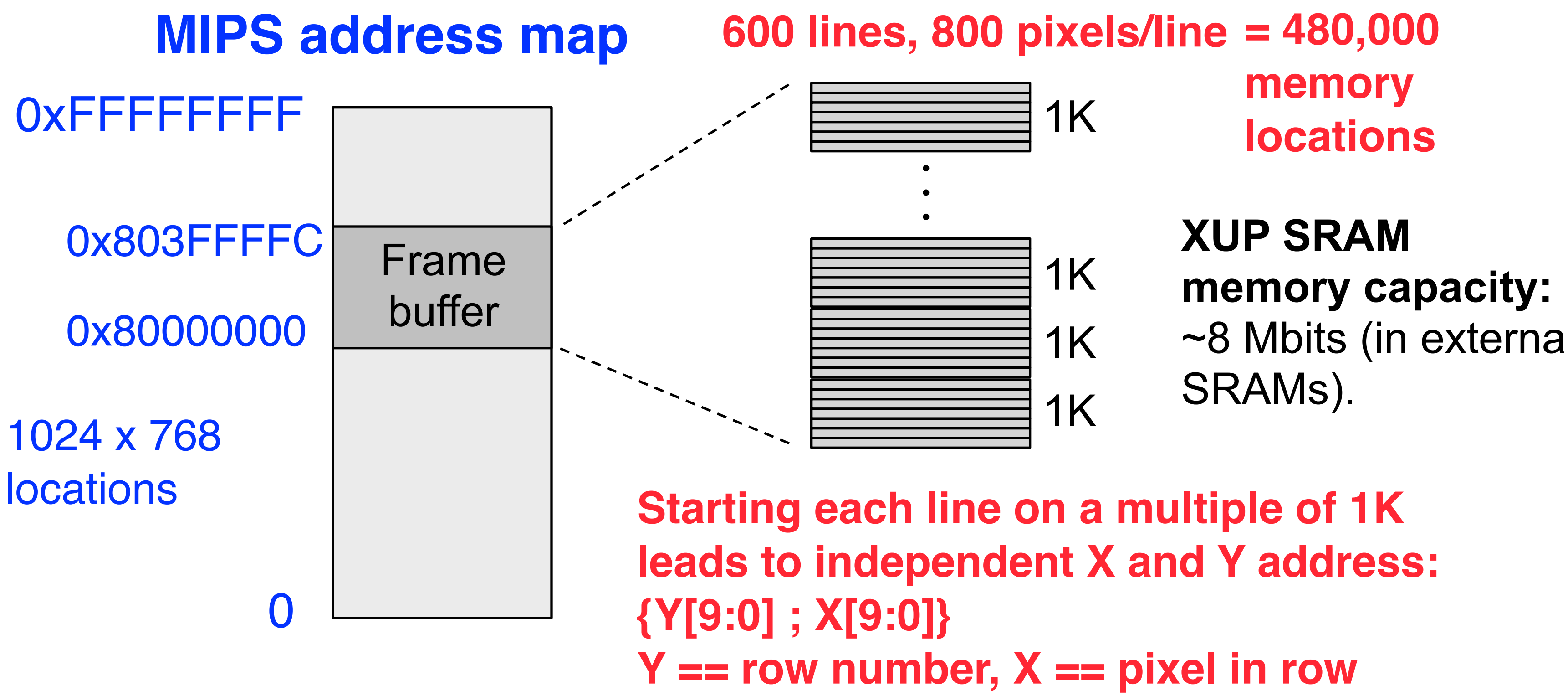
- A range of memory addresses correspond to the display.
- CPU writes (using sw instruction) pixel values to change display.
- No handshaking required. Independent process reads pixels from memory and sends them to the display interface at the required rate.



8Mbits / 480000 = 17.5 bits/pixel max!

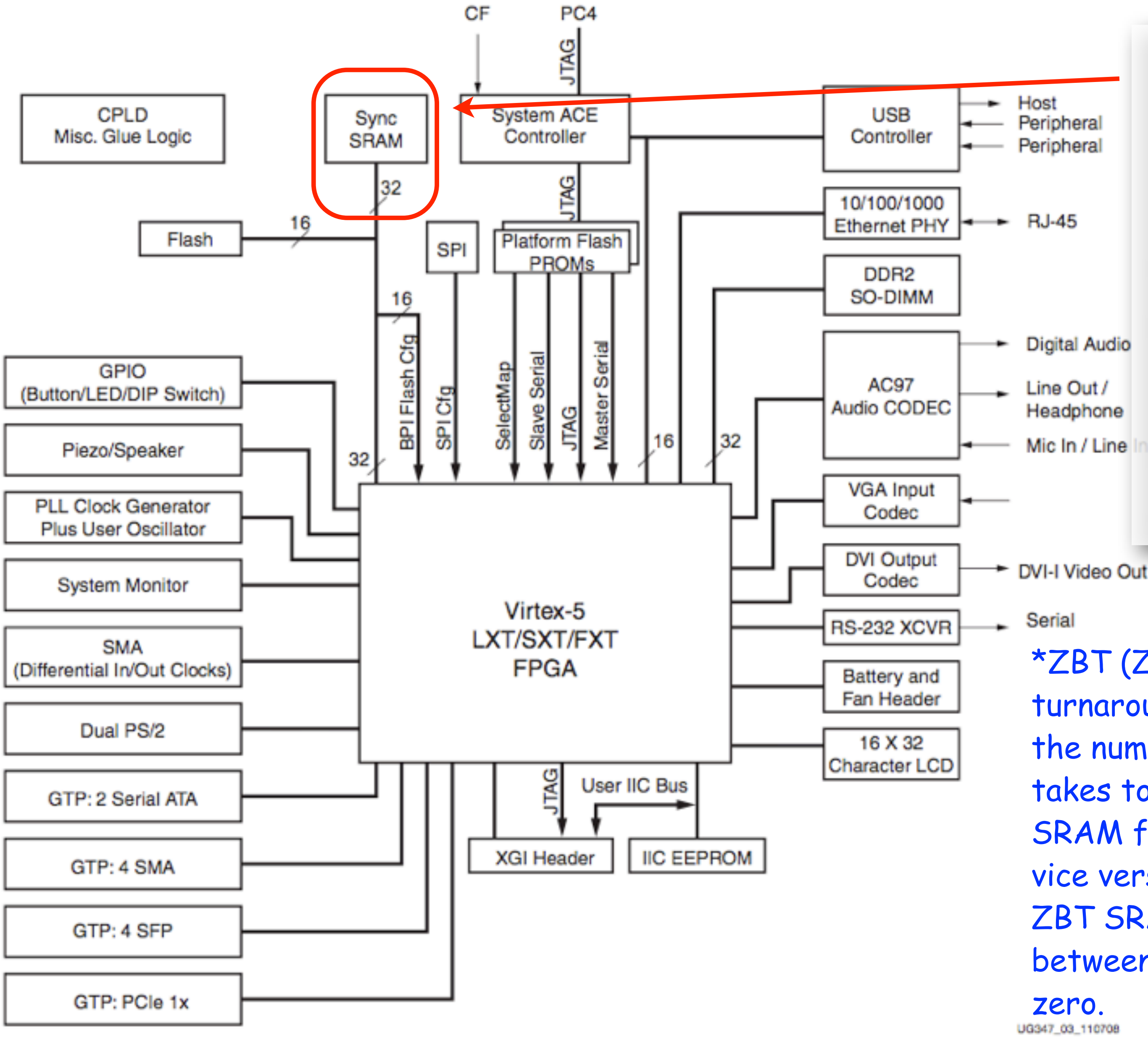
We choose 16 bits/pixel
{ Red[4:0] ; Green[5:0] ; Blue[4:0] }

Framebuffer Details 2010



- Note, that we assign only one 16 bit pixel per memory location.
- Two pixel address map to one address in the SRAM (it is 32bits wide).
- Only part of the mapped memory range occupied with physical memory.

XUP Board External SRAM

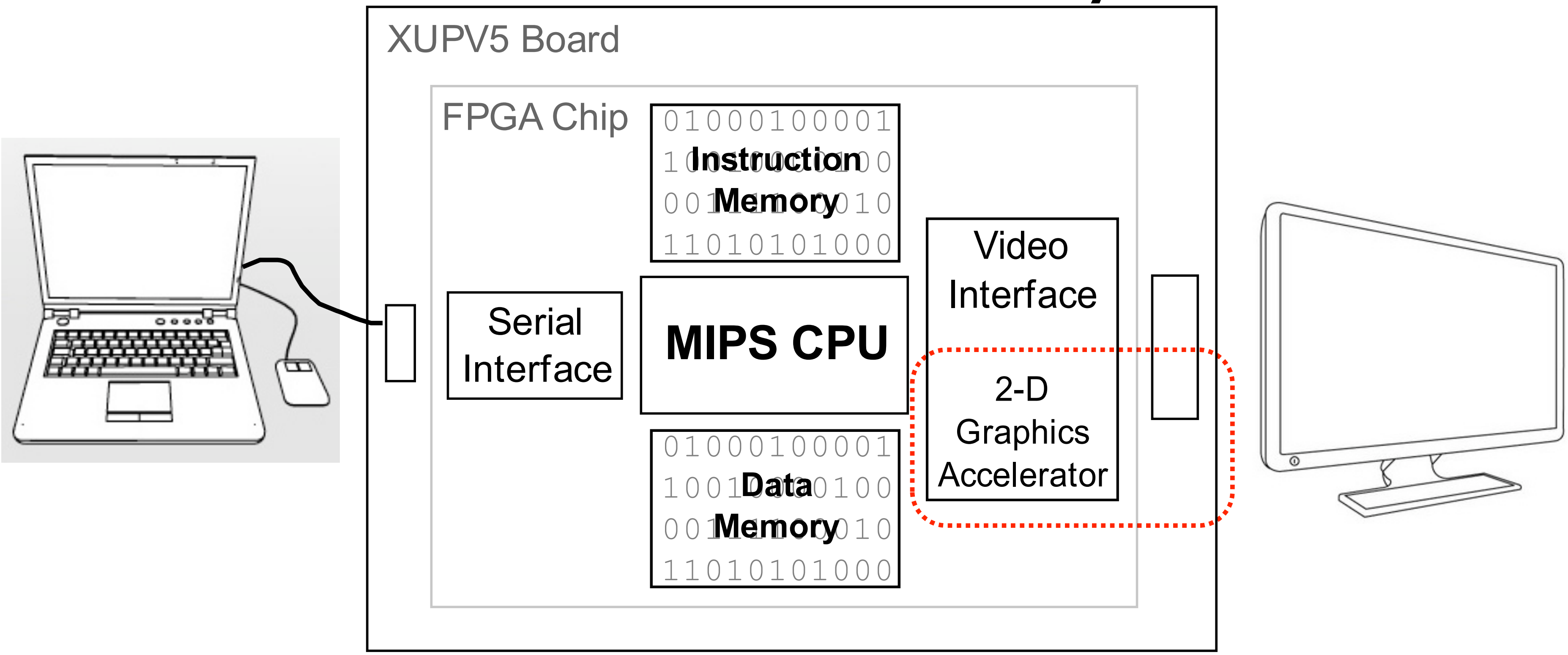


"ZBT" synchronous SRAM, 9 Mb on 32-bit data bus, with four "parity" bits

256K x 36 bits (located under the removable LCD)

*ZBT (ZBT stands for zero bus turnaround) — the turnaround is the number of clock cycles it takes to change access to the SRAM from write to read and vice versa. The turnaround for ZBT SRAMs or the latency between read and write cycle is zero.

MIPS150 Video Subsystem



- Gives software ability to display information on screen.
- Equivalent to standard graphics cards:
  - Processor can directly write the display bit map
  - 2D Graphics acceleration

Graphics Software

"Clearing" the screen - fill the entire screen with same color

Remember Framebuffer base address: **0x8000_0000**

Size: **1024 x 768**

```
clear:  # a0 holds 4-bit pixel color
        # t0 holds the pixel pointer
        ori     $t0, $0, 0x8000         # top half of frame address
        sll     $t0, $t0, 16            # form framebuffer beginning address
        # t2 holds the framebuffer max address
        ori     $t2, $0, 768            # 768 rows
        sll     $t2, $t2, 12            #  * 1K pixels/row * 4 Bytes/address
        addu    $t2, $t2, $t0           # form ending address
        addiu   $t2, $t2, -4            # minus one word address
        #
        # the write loop
L0:     sw      $a0, 0($t0)             # write the pixel
        bneq    $t0, $t2, L0            # loop until done
        addiu   $t0, $t0, 4             # bump pointer
        jr      $ra
```

How long does this take? What do we need to know to answer?

How does this compare to the frame rate?

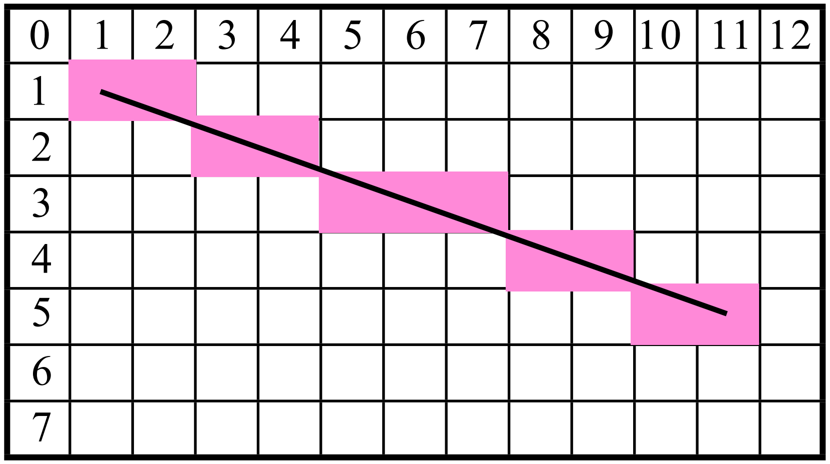
Optimized Clear Routine

Amortizing the loop overhead.

```
clear:
          .
          .
          .
        # the write loop
L0:     sw      $a0, 0($t0)             # write some pixels
        sw      $a0, 4($t0)
        sw      $a0, 8($t0)
        sw      $a0, 12($t0)
        sw      $a0, 16($t0)
        sw      $a0, 20($t0)
        sw      $a0, 24($t0)
        sw      $a0, 28($t0)
        sw      $a0, 32($t0)
        sw      $a0, 36($t0)
        sw      $a0, 40($t0)
        sw      $a0, 44($t0)
        sw      $a0, 48($t0)
        sw      $a0, 52($t0)
        sw      $a0, 56($t0)
        sw      $a0, 60($t0)
        bneq    $t0, $t2, L0            # loop until done
        addiu   $t0, $t0, 64            # bump pointer
        jr      $ra
```

What's the performance of this one?

Line Drawing



From (x_0,y_0) to (x_1,y_1)

Line equation defines all the points:

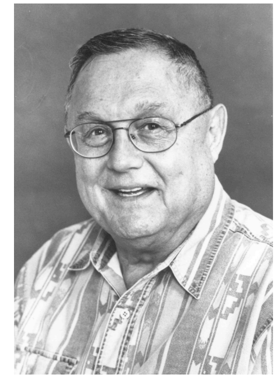
$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

For each x value, could compute y, with:
then round to the nearest integer y value.

$$\frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

Slope can be precomputed, but still requires floating point * and + in the loop: slow or expensive!

Bresenham Line Drawing Algorithm



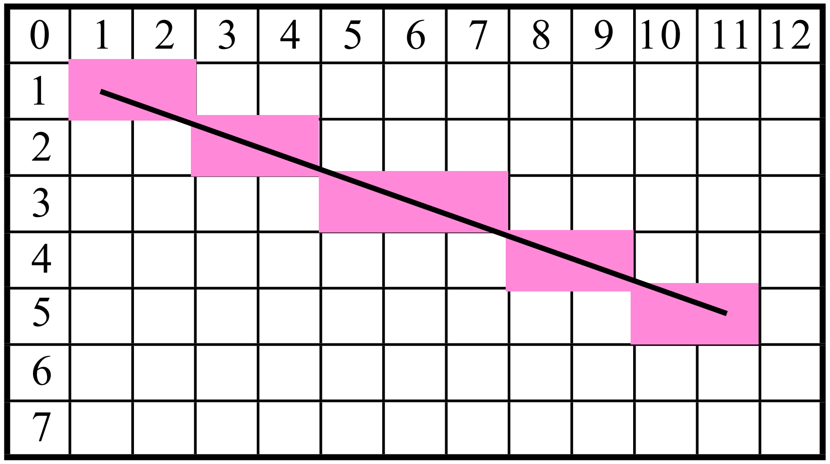
Developed by Jack E. Bresenham in 1962 at IBM.

"I was working in the computation lab at IBM's San Jose development lab. A Calcomp plotter had been attached to an IBM 1401 via the 1407 typewriter console. ...

- Computers of the day, slow at complex arithmetic operations, such as multiply, especially on floating point numbers.
- Bresenham's algorithm works with integers and without multiply or divide.
- Simplicity makes it appropriate for inexpensive hardware implementation.
- With extension, can be used for drawing circles.

Line Drawing Algorithm

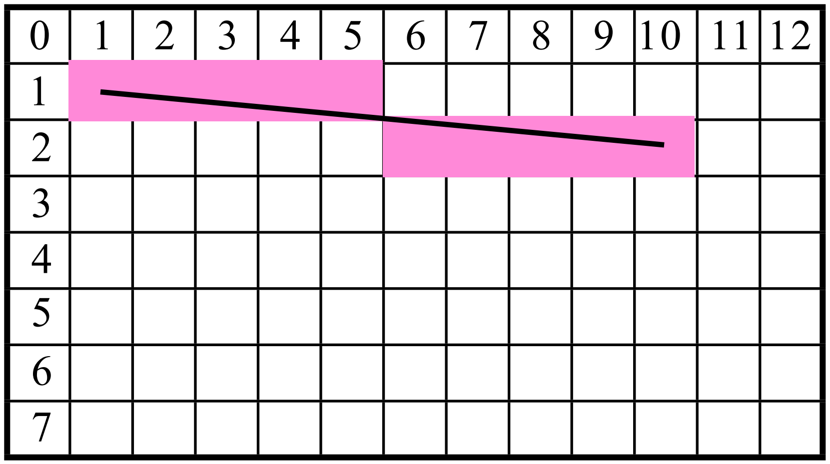
This version assumes: $x_0 < x_1$, $y_0 < y_1$, slope =< 45 degrees



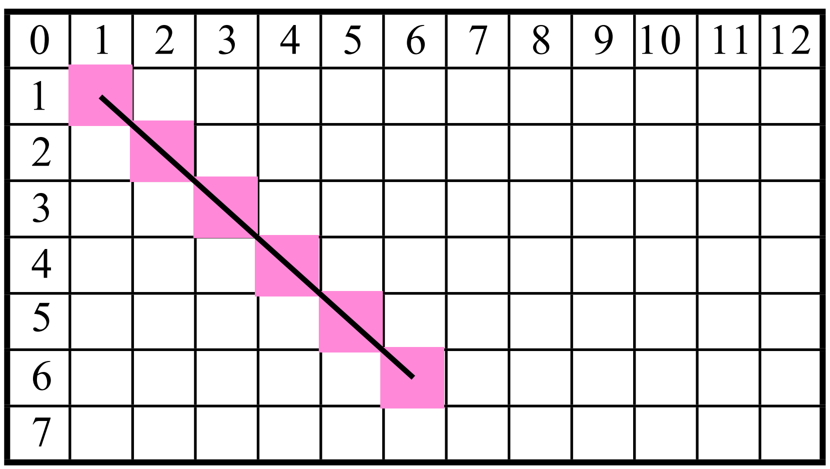

```
function line(x0, x1, y0, y1)
   int deltax := x1 - x0
   int deltay := y1 - y0
   int error := deltax / 2
   int y := y0
   for x from x0 to x1
      plot(x,y)
      error := error - deltay
      if error < 0 then
         y := y + 1
         error := error + deltax
```

Note: error starts at deltax/2 and gets decremented by deltay for each x, y gets incremented when error goes negative, therefore y gets incremented at a rate proportional to deltax/deltay.

Line Drawing, Examples

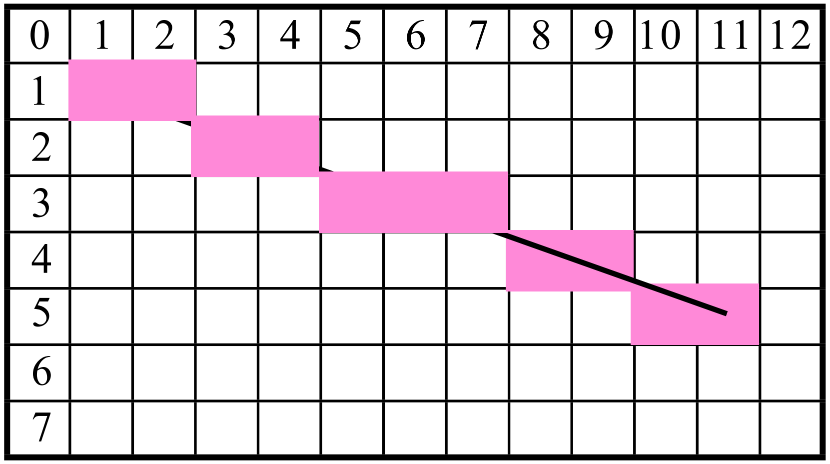


deltay = 1 (very low slope). y only gets incremented once (halfway between x0 and x1)



deltay = deltax (45 degrees, max slope). y gets incremented for every x

Line Drawing Example




```
function line(x0, x1, y0, y1)
    int deltax := x1 - x0
    int deltay := y1 - y0
    int error := deltax / 2
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error - deltay
        if error < 0 then
            y := y + 1
            error := error + deltax
```

(1,1) -> (11,5)

deltax = 10, deltay = 4, error = 10/2 = 5, y = 1

x = 1: plot(1,1)
error = 5 - 4 = 1

x = 2: plot(2,1)
error = 1 - 4 = -3
  y = 1 + 1 = 2
  error = -3 + 10 = 7

x = 3: plot(3,2)
error = 7 - 4 = 3

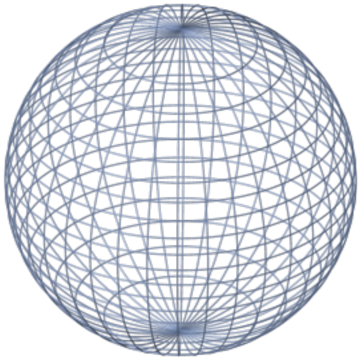
x = 4: plot(4,2)
error = 3 - 4 = -1
  y = 2 + 1 = 3
  error = -1 + 10 = 9

x = 5: plot(5,3)
error = 9 - 4 = 5

x = 6: plot(6,3)
error = 5 - 4 = 1

x = 7: plot(7,3)
error = 1 - 4 = -3
  y = 3 + 1 = 4
  error = -3 + 10 -= 7

C Version



```
#define SWAP(x, y) (x ^= y ^= x ^= y)
#define ABS(x) (((x)<0) ? -(x) : (x))

void line(int x0, int y0, int x1, int y1) {
  char steep = (ABS(y1 - y0) > ABS(x1 - x0)) ? 1 : 0;
  if (steep) {
    SWAP(x0, y0);
    SWAP(x1, y1);
  }
  if (x0 > x1) {
    SWAP(x0, x1);
    SWAP(y0, y1);
  }
  int deltax = x1 - x0;
  int deltay = ABS(y1 - y0);
  int error = deltax / 2;
  int ystep;
  int y = y0
  int x;
  ystep = (y0 < y1) ? 1 : -1;
  for (x = x0; x <= x1; x++) {
    if (steep)
      plot(y,x);
    else
      plot(x,y);
    error = error - deltay;
    if (error < 0) {
      y += ystep;
      error += deltax;
    }
  }
}
```

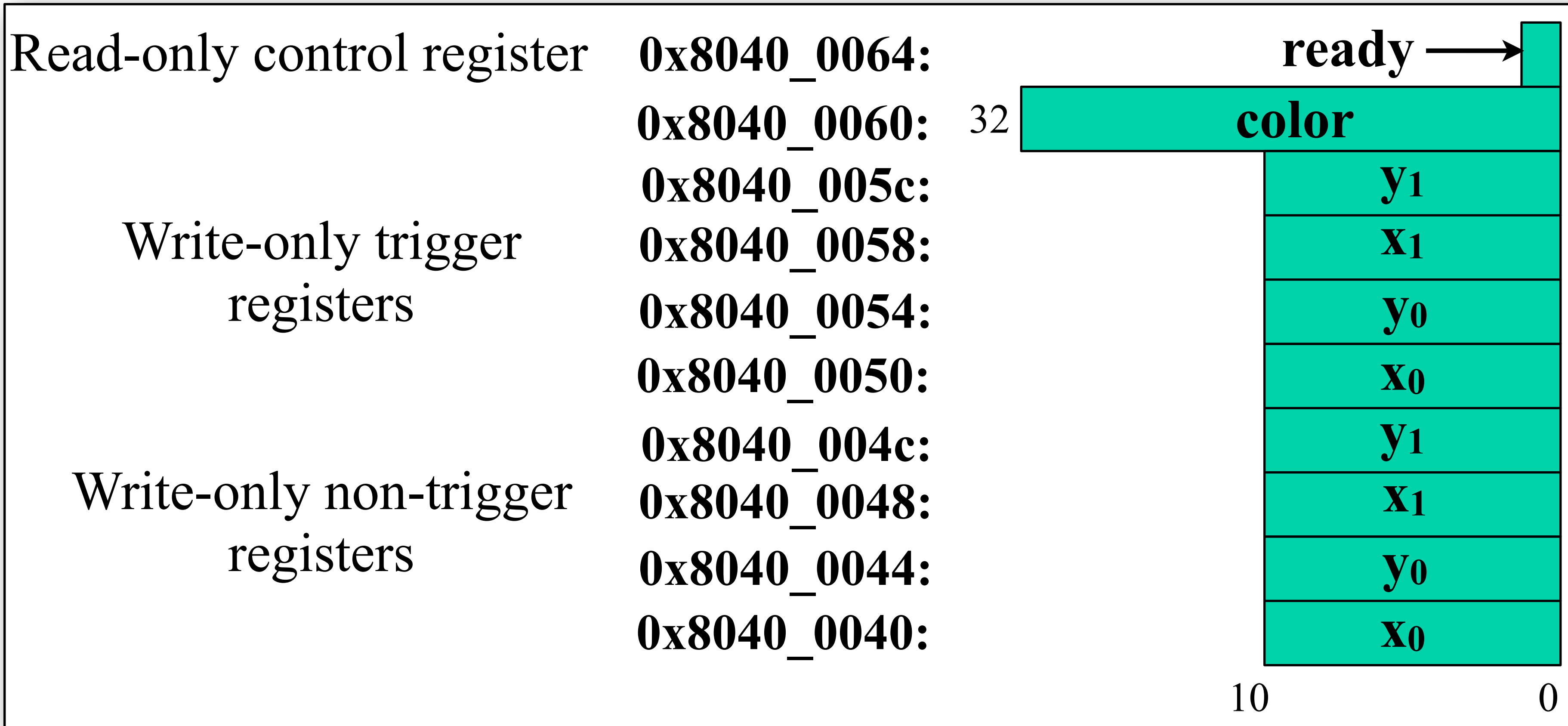
Modified to work in any quadrant and for any slope.

Estimate software performance (MIPS version)

What's needed to do it in hardware?

Goal is one pixel per cycle. Pipelining might be necessary.

Hardware Implementation Notes



- CPU initializes line engine by sending pair of points and color value to use. Writes to 0x8040_005* trigger engine.
- Framebuffer has one write port - Shared by CPU and line engine. Priority to CPU - Line engine stalls when CPU writes.