EECS 151/251A
Fall 2019
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek

Lecture 14

Outline

- Accelerators

Motivation

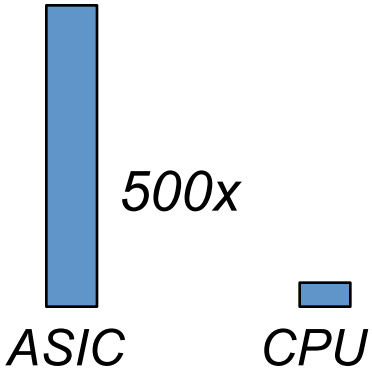
- 90/10 rule:
  - Often 90 percent of the program runtime and energy is consumed by 10 percent of the code (inner-loops).
  - Only small portions of an application become the performance bottlenecks.
  - Usually, these portions of code are data processing intensive with relatively fixed dataflow patterns (little control): cryptography, graphics, video, communications signal processing, networking, ...
  - The other 90 percent of the code not performance critical: UI, control, glue, exceptional cases, ...

    Hybrid processor-core hardware accelerator

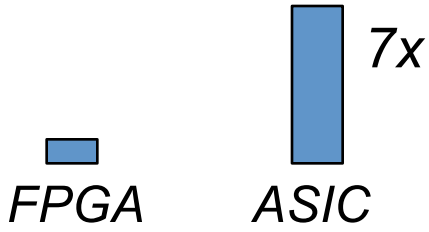
  - Hardware accelerator/economizer implements specialized circuits for inner-loops.
  - Processor packs the noncritical portions (90%), 10% of the computation into minimal space.

Energy Efficiency of CPU versus ASIC versus FPGA

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. SIGARCH Comput. Archit. News, 38:37–47, June 2010.



Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM

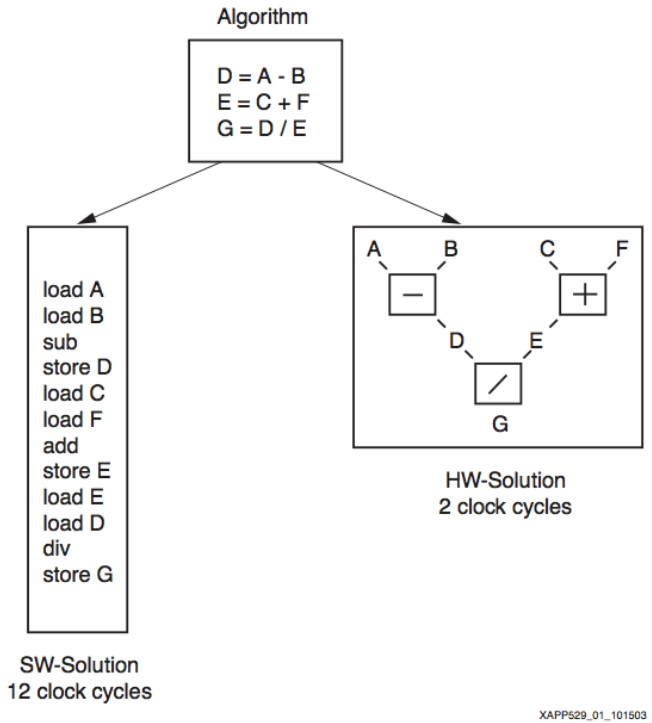


$\therefore$ *FPGA : CPU = 70x*

Similar story for performance efficiency

Why are accelerators more efficient than processors?

- Performance/cost or Energy/op
  1. exploit problem specific parallelism, at thread and instructions level
  2. custom "instructions" match the set of operations needed for the algorithm (replace multiple instructions with one), custom word width arithmetic, etc.
  3. remove overhead of instruction storage and fetch, ALU multiplexing



What about FPGAs?

"System on Chip" Example

- Three ARM cores, plus lots of accelerators
- Targets smart phones

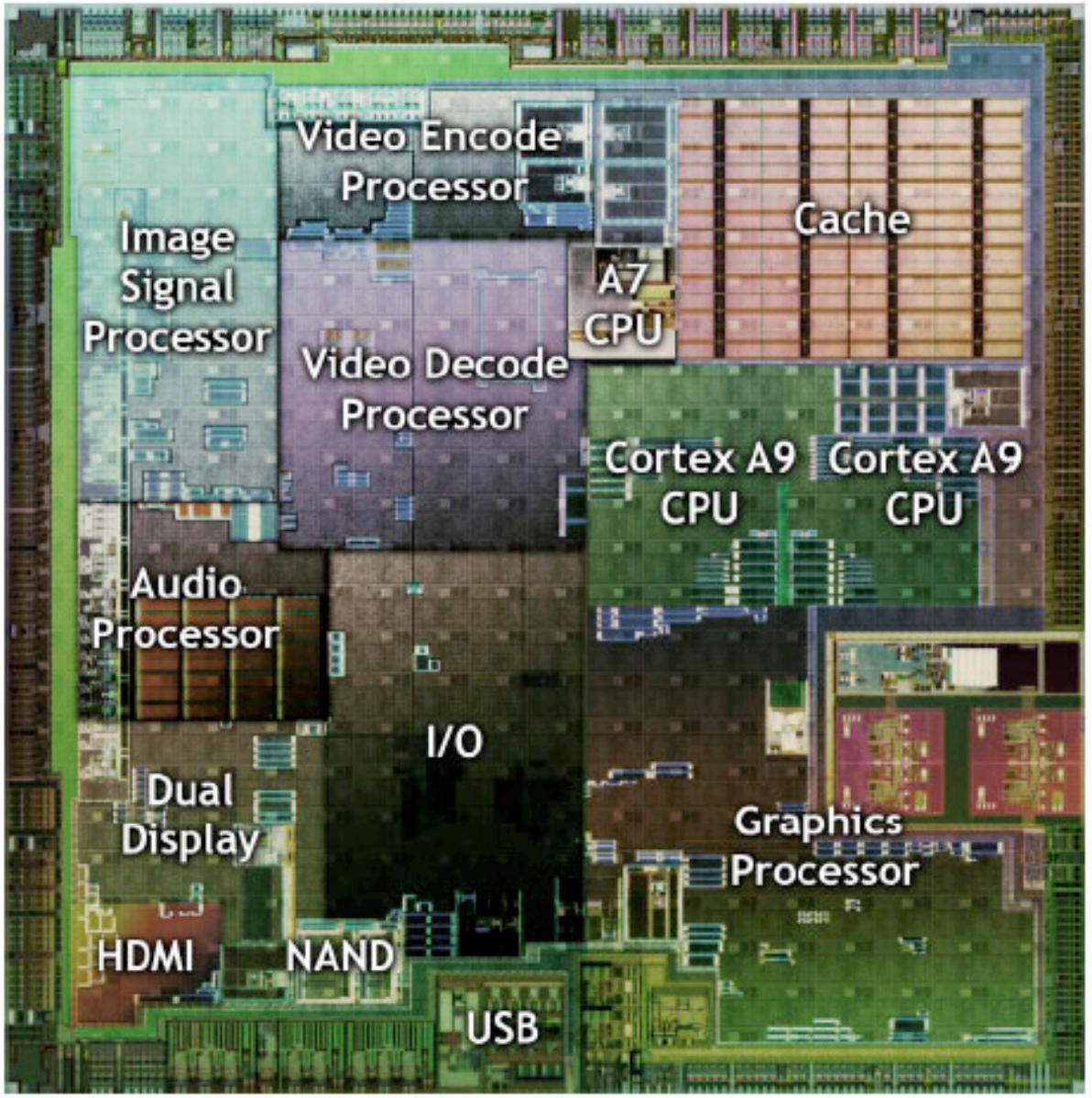
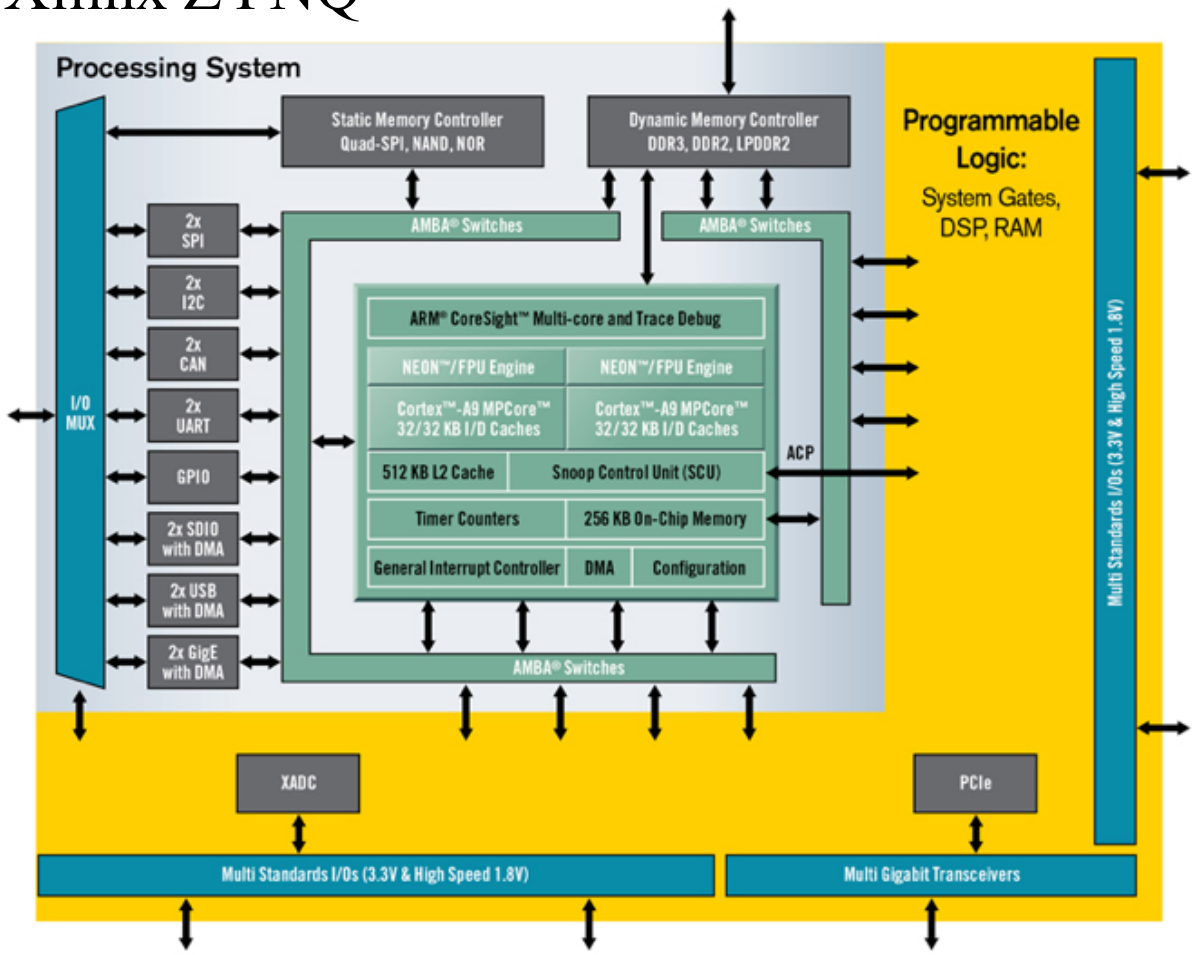


Figure 1 - NVIDIA Tegra 2 System on a Chip

Processors in FPGAs

Xilinx ZYNQ

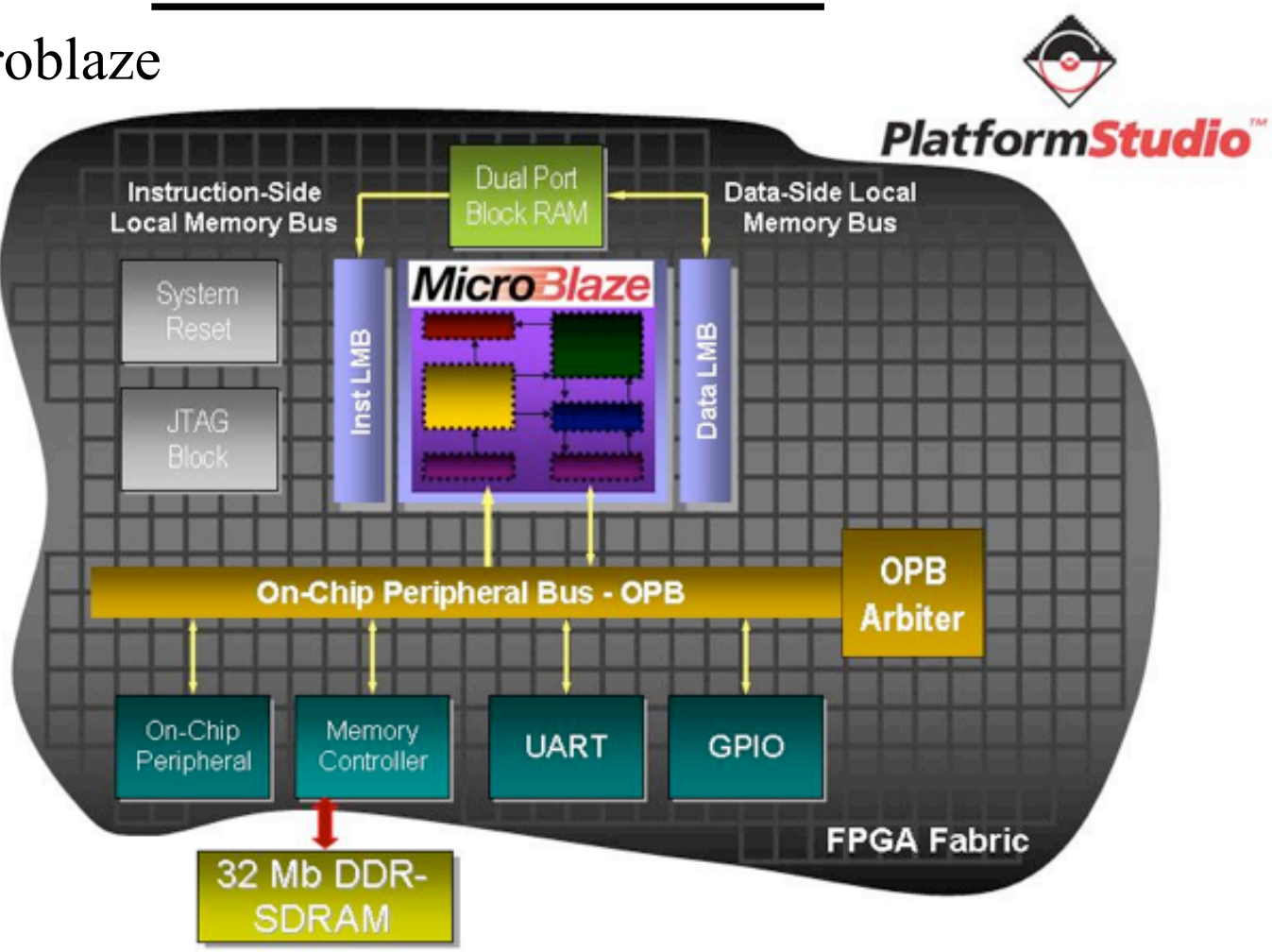


- Dual ARM Cortex™-A9 MPCore
  - Up to 800MHz
  - Enhanced with NEON Extension and Single & Double Precision Floating point unit
  - 32kB Instruction & 32kB Data L1 Cache
- Unified 512kB L2 Cache
- 256kB on-chip Memory
- DDR3, DDR2 and LPDDR2 Dynamic Memory Controller
- 2x QSPI, NAND Flash and NOR Flash Memory Controller
- 2x USB2.0 (OTG), 2x GbE, 2x CAN2,0B 2x SD/SDIO, 2x UART, 2x SPI, 2x I2C, 4x 32b GPIO
- AES & SHA 256b encryption engine for secure boot and secure configuration
- Dual 12bit 1Msps Analog-to-Digital converter
  - Up to 17 Differential Inputs
- Advanced Low Power 28nm Programmable Logic:
  - 28k to 235k Logic Cells (approximately 430k to 3.5M of equivalent ASIC Gates)
  - 240kB to 1.86MB of Extensible Block RAM
  - 80 to 760 18x25 DSP Slices (58 to 912 GMACS peak DSP performance)
- PCI Express® Gen2x8 (in largest devices)
- 154 to 404 User IOs (Multiplexed + SelectIO™)
- 4 to 12 12.5Gbps Transceivers (in largest devices)

Altera: **Dual-Core ARM Cortex-A9 MPCore Processor**

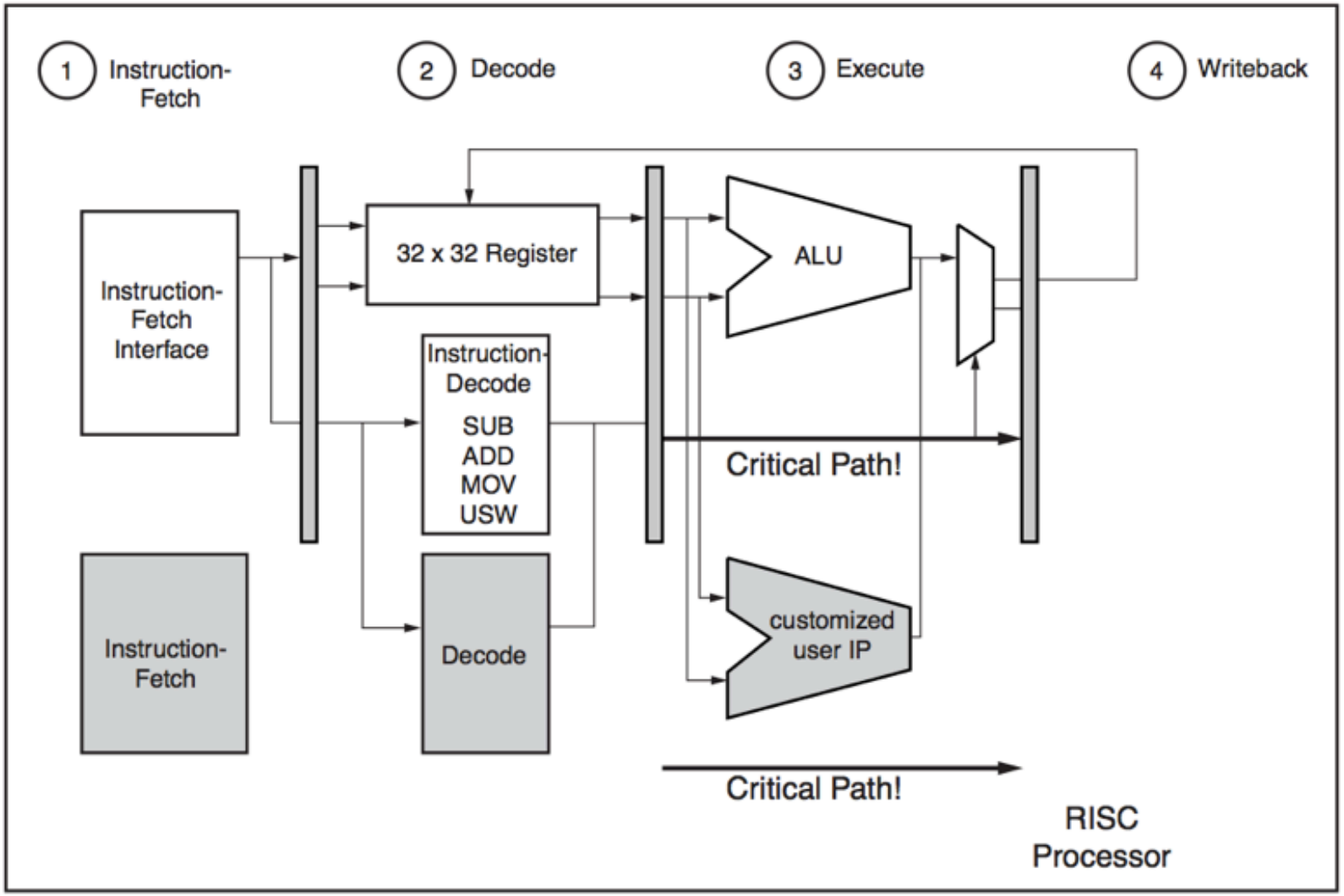
Soft Processors

Xilinx: Microblaze

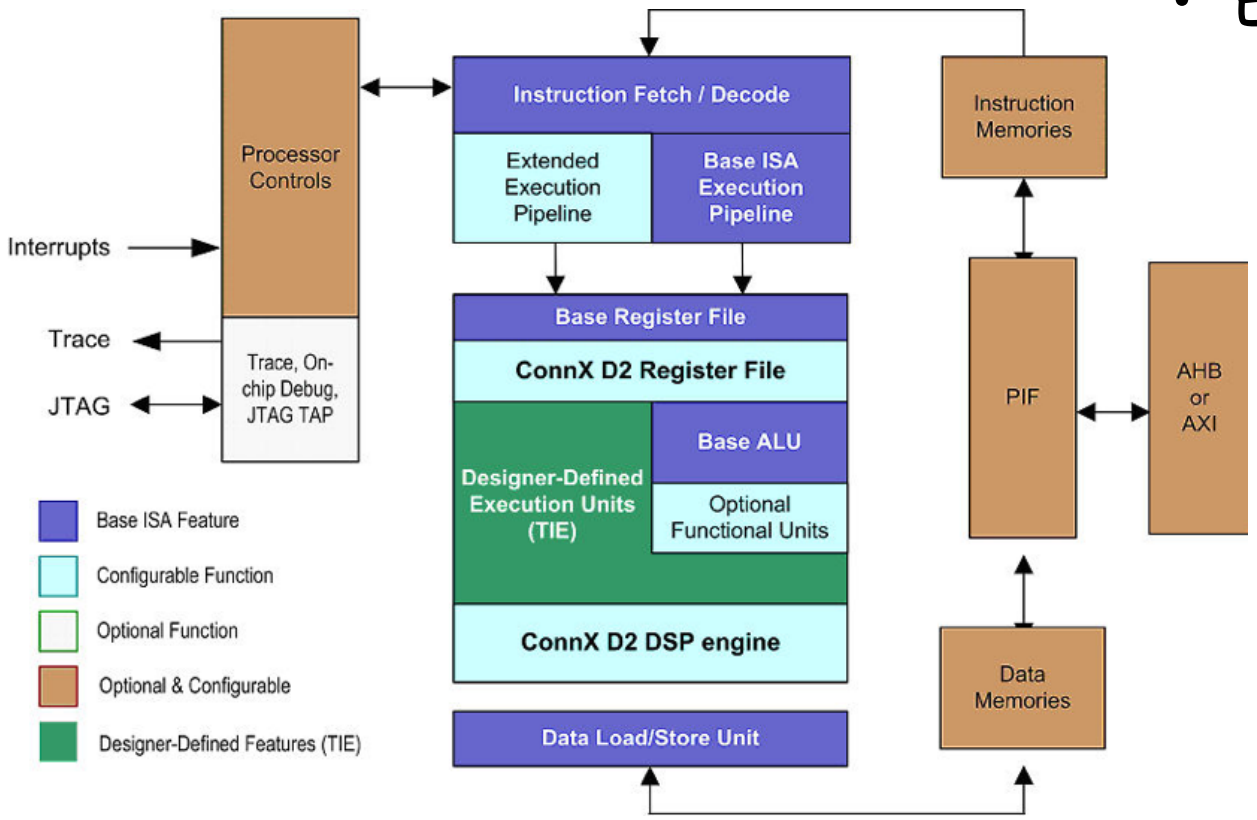


Intel/Altera: Nios

Custom Hardware in the Pipeline

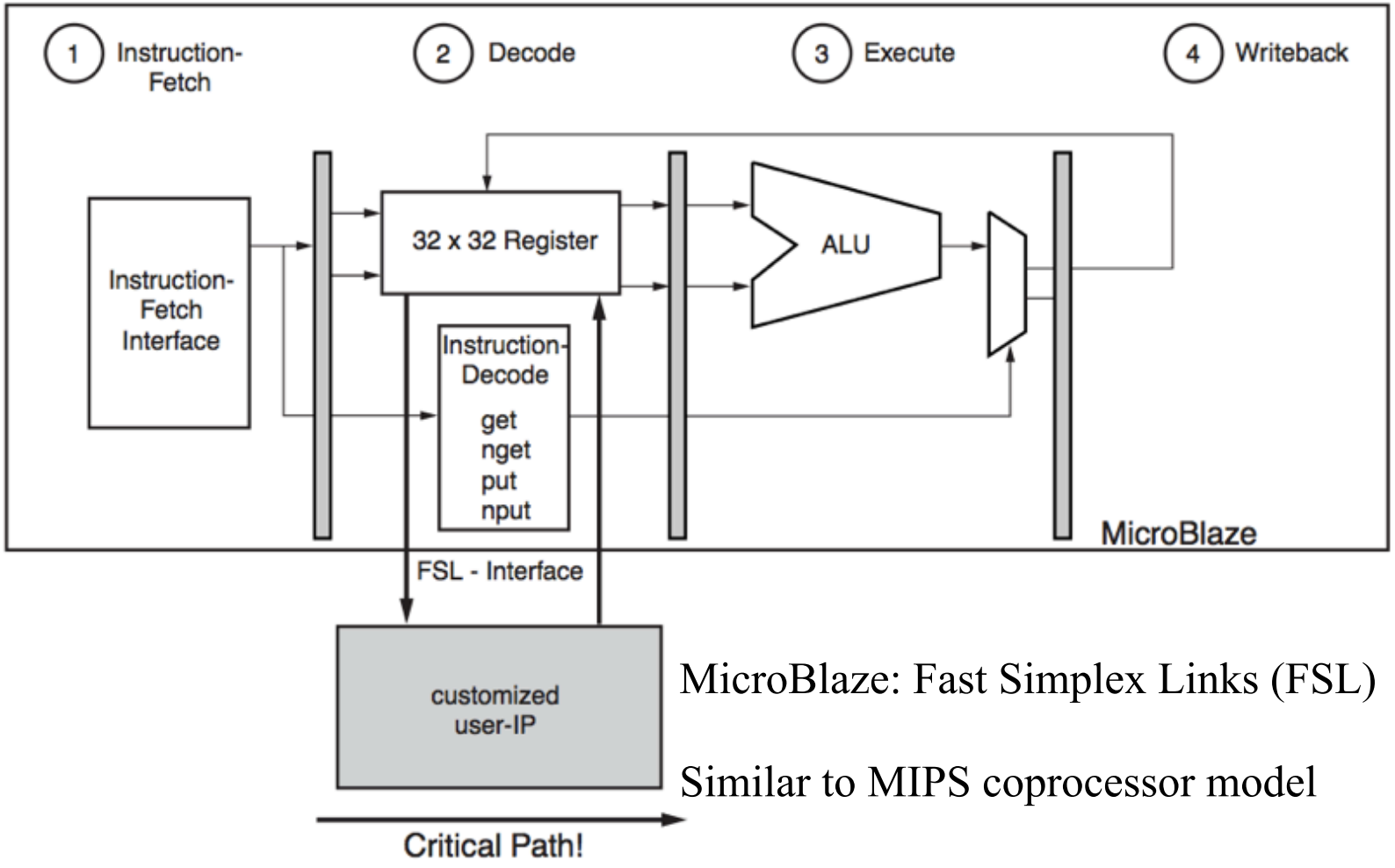


Custom Instructions



- Example: Tensilca
  - Special language TIE is used for defining special function units
  - Custom architecture automatically compiled
  - Compiler support challenging

Tightly Coupled Co-processor

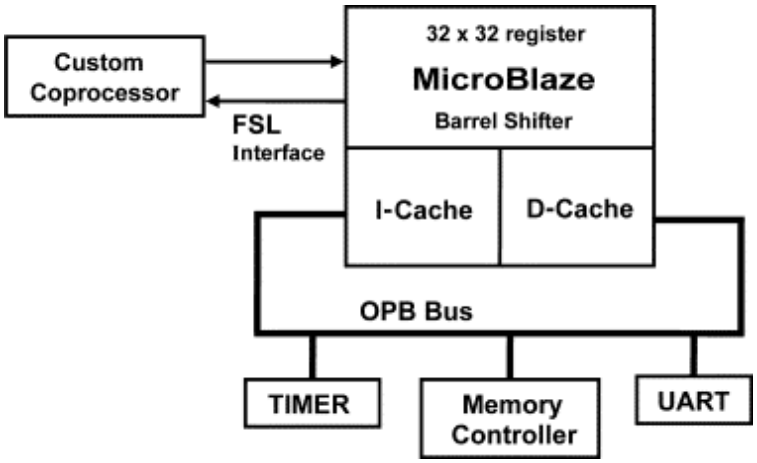


MicroBlaze: Fast Simplex Links (FSL)

Similar to MIPS coprocessor model

XAPP529_04_101503

MicroBlaze Fast Simplex Links



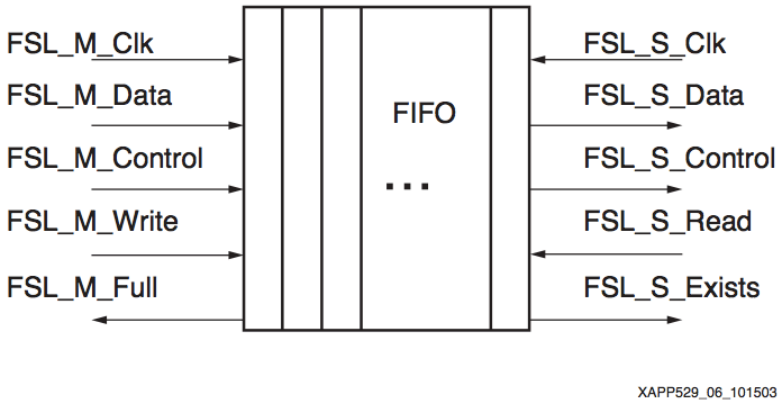
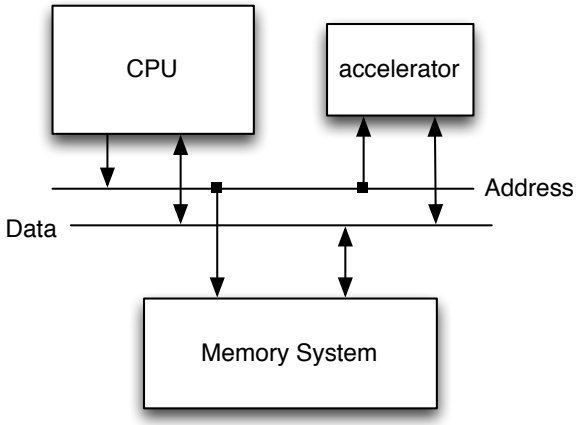


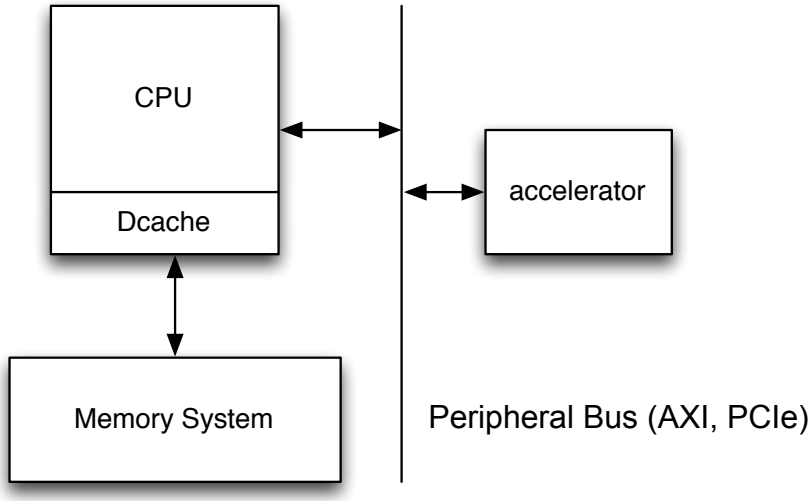
Figure 6: **FSL interface**

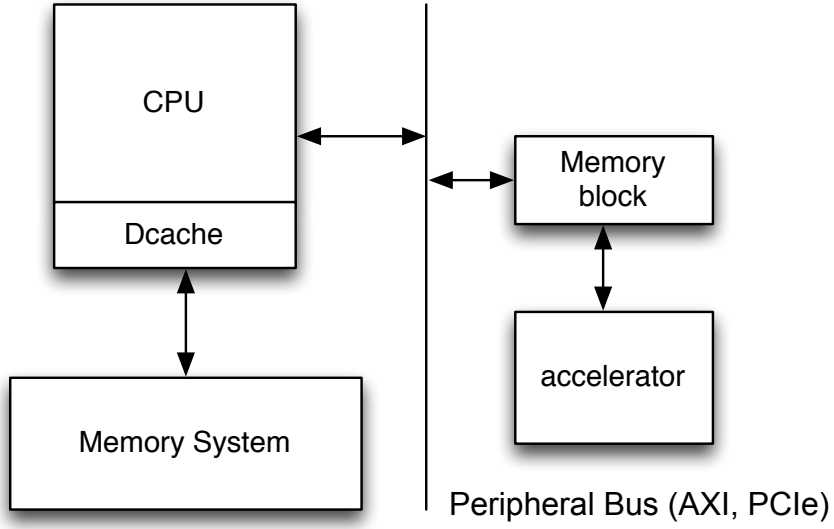
```
// Blocking Data Read and Write to Local Link no. id
microblaze_bread_datafsl(val, id)
microblaze_bwrite_datafsl(val, id)

// Non-blocking Data Read and Write to Local Link no. id
microblaze_nbread_datafsl(val, id)
microblaze_nbwrite_datafsl(val, id)

// Blocking Control Read and Write to Local Link no. id
microblaze_bread_cntlfsl(val, id)
microblaze_bwrite_cntlfsl(val, id)

// Non-blocking Control Read and Write to Local Link no. id
microblaze_nbread_cntlfsl(val, id)
microblaze_nbwrite_cntlfsl(val, id)
```

Memory Mapped Accelerators

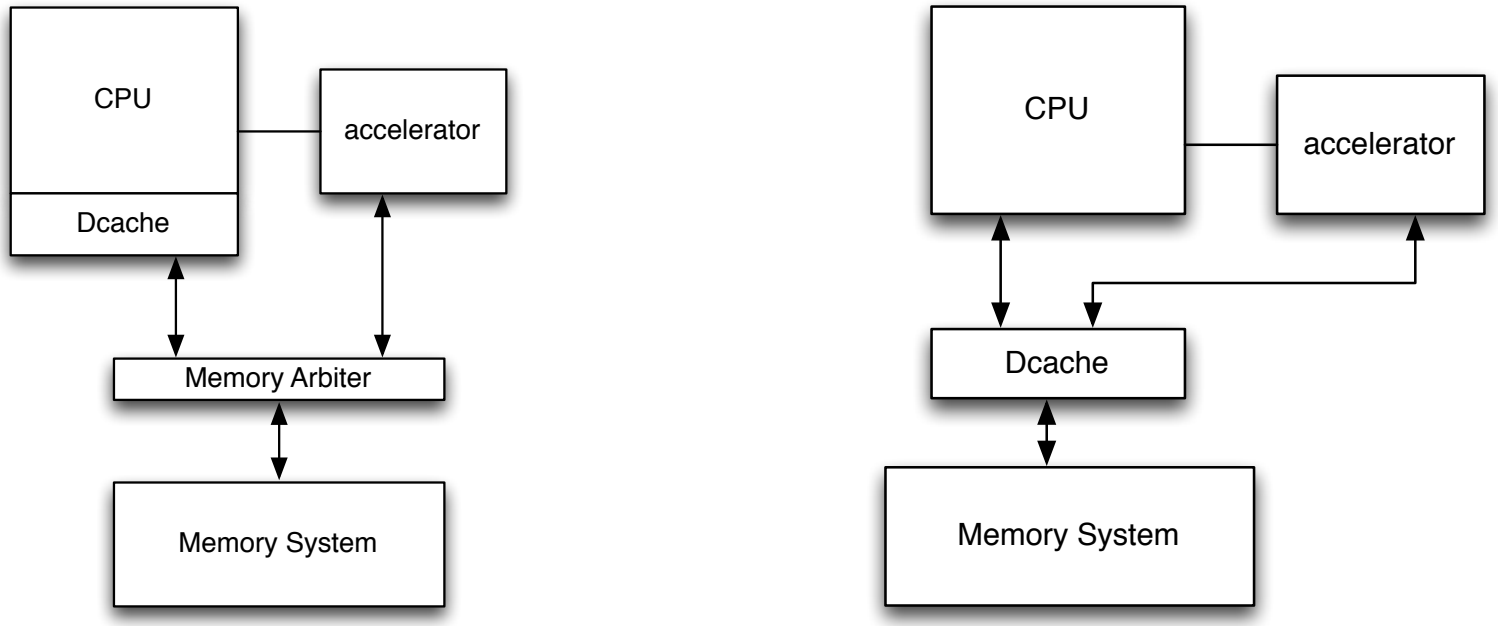


- Memory mapped control/ data registers



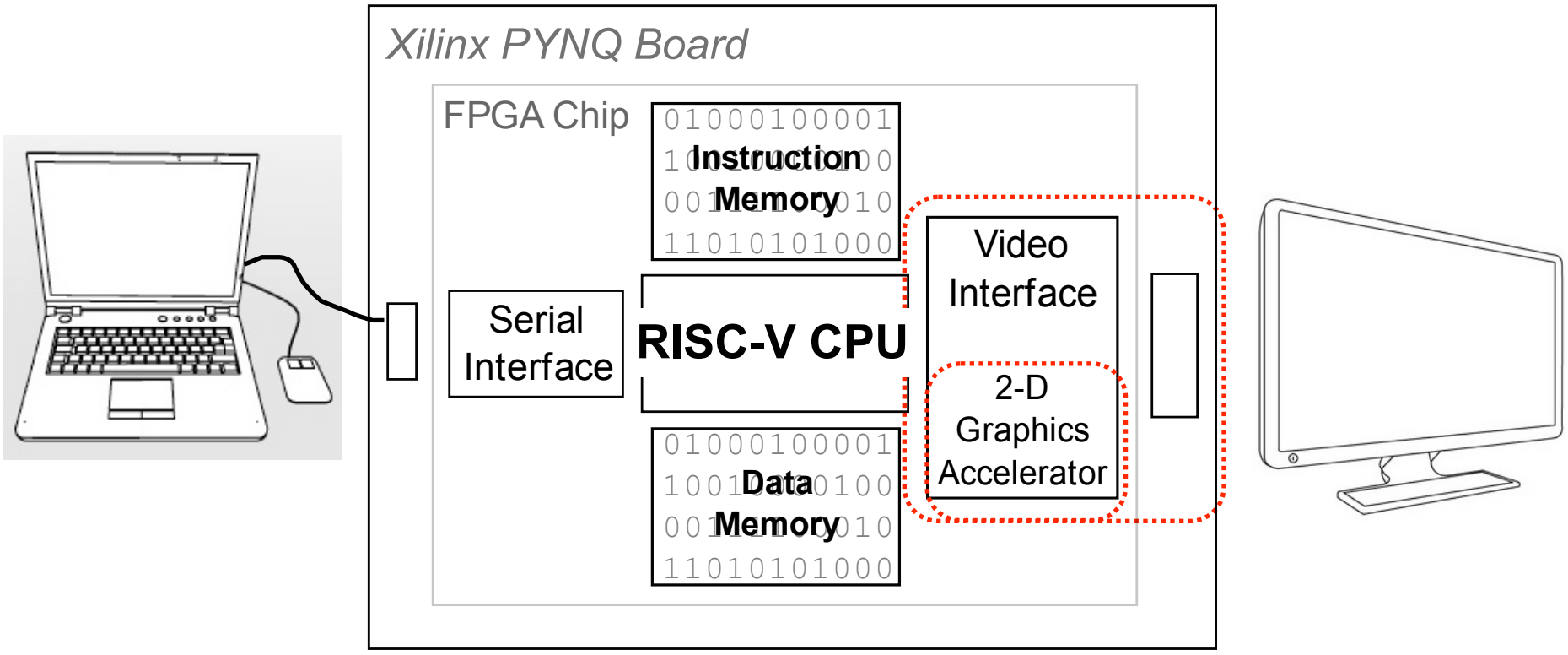


CPU/Accelerator Shared Memory



- Processor instructs accelerator to independently access memory and perform work
- How does processor synchronize with accelerator (how does it know when it is done)?
- Data Cache on CPU creates "coherency" issue
- What about a cache in the accelerator?

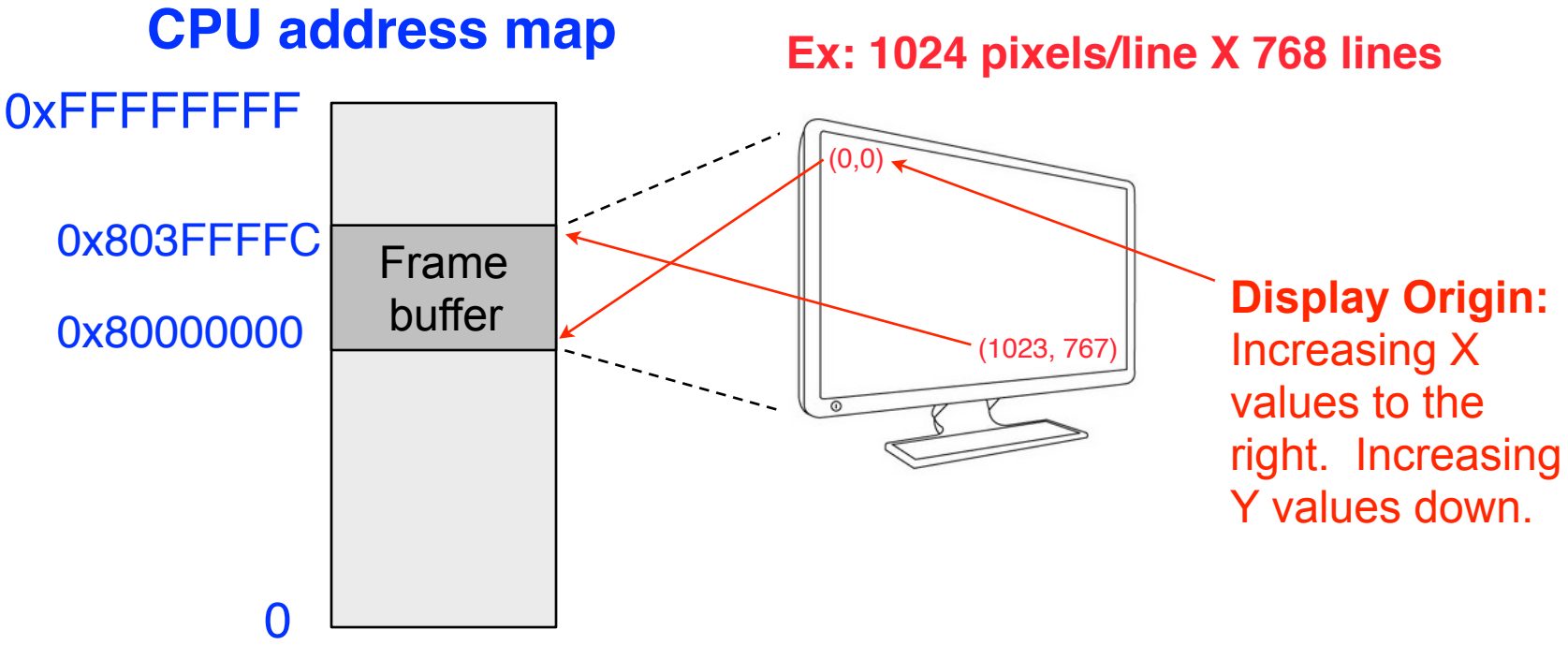
RISCV-151 Video Subsystem



- Gives software ability to display information on screen.
- Also, similar to standard graphics cards:
  - 2D Graphics acceleration to offload work from processor

"Framebuffer" HW/SW Interface

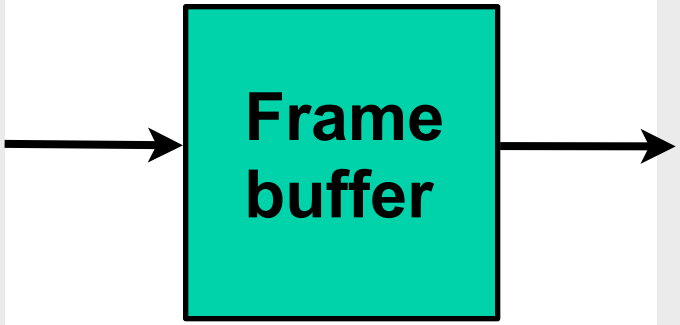
- A range of memory addresses correspond to the display.
- CPU writes (using sw instruction) pixel values to change display.
- No synchronization required. Independent process reads pixels from memory and sends them to the display interface at the required rate.



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.

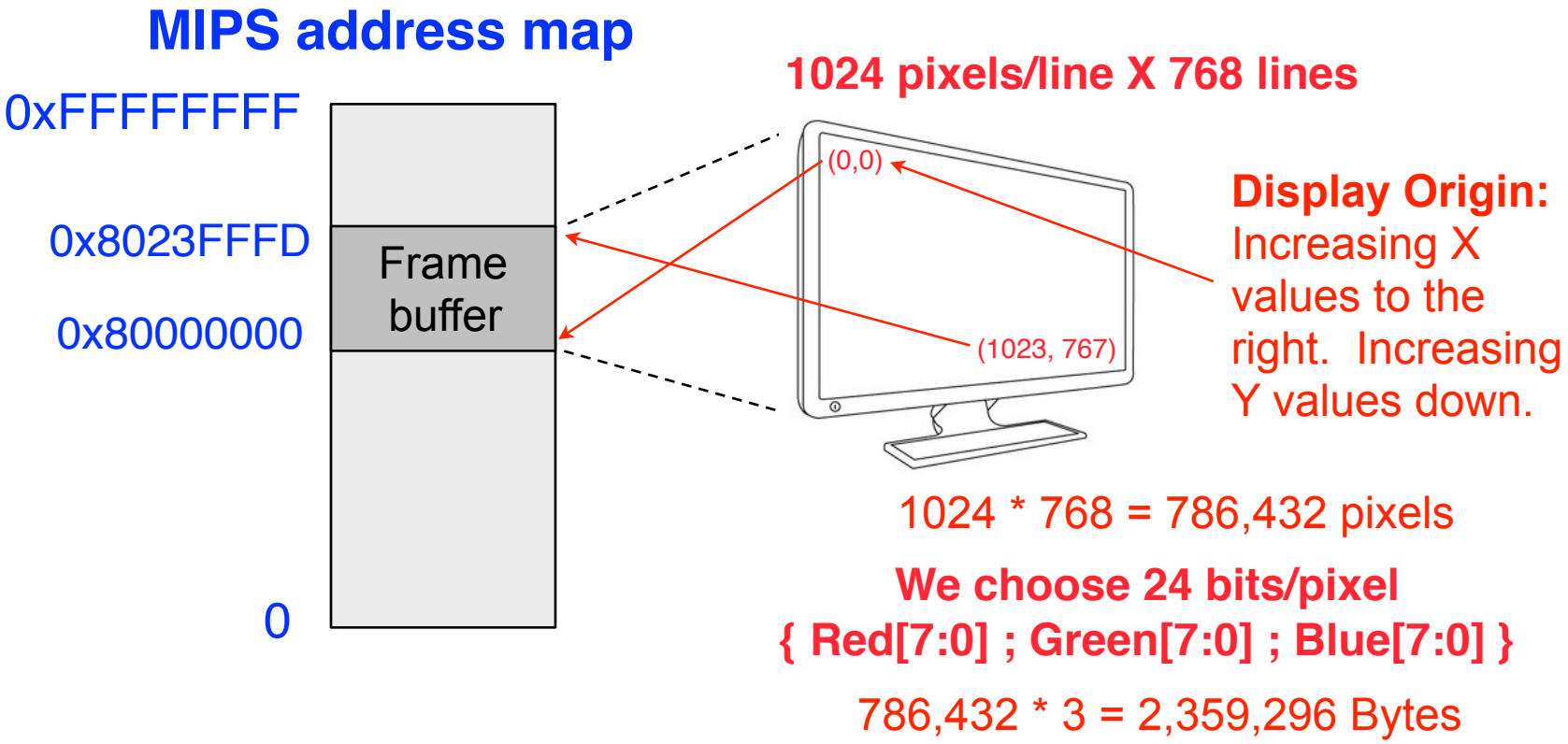


Video Interface continuously reads pixel locations in scan-line order and sends to physical display.

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

1024 x 768 pixels/frame x 24 bits/pixel

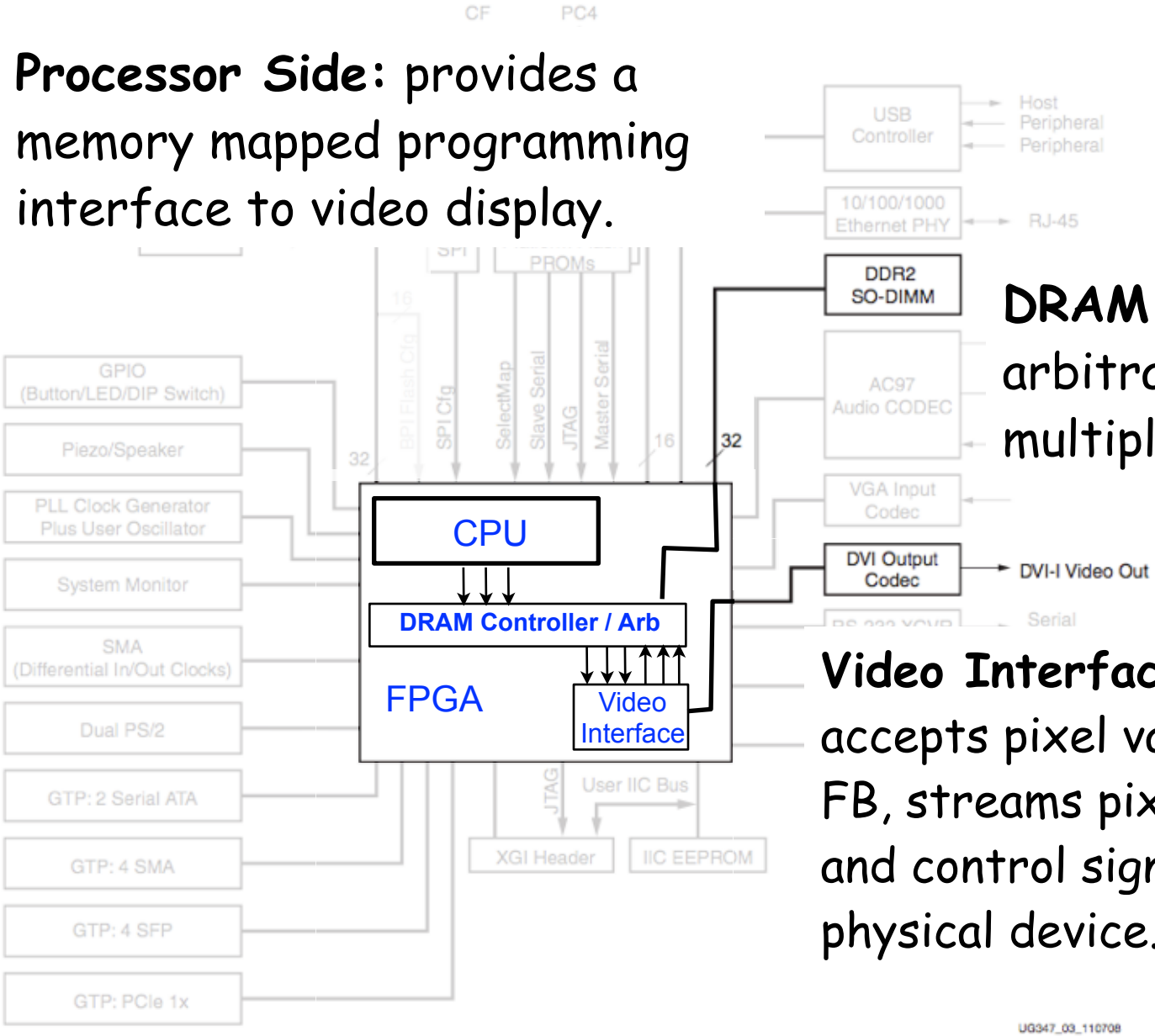
Memory Mapped Framebuffer



- Total memory bandwidth needed to support frame buffer?

Frame Buffer Physical Interface

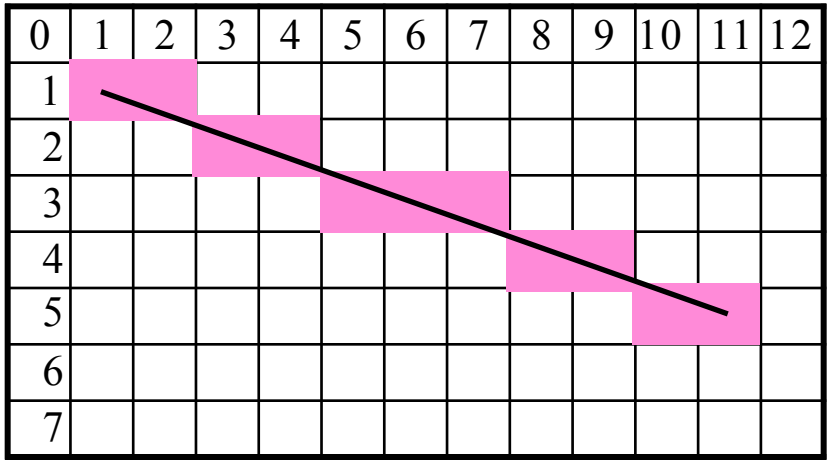
Processor Side: provides a memory mapped programming interface to video display.



DRAM "Arb": arbitrates among multiple DRAM users.

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

Line Drawing Acceleration



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:

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

then round to the nearest integer y value.

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

Bresenham Line Drawing Algorithm

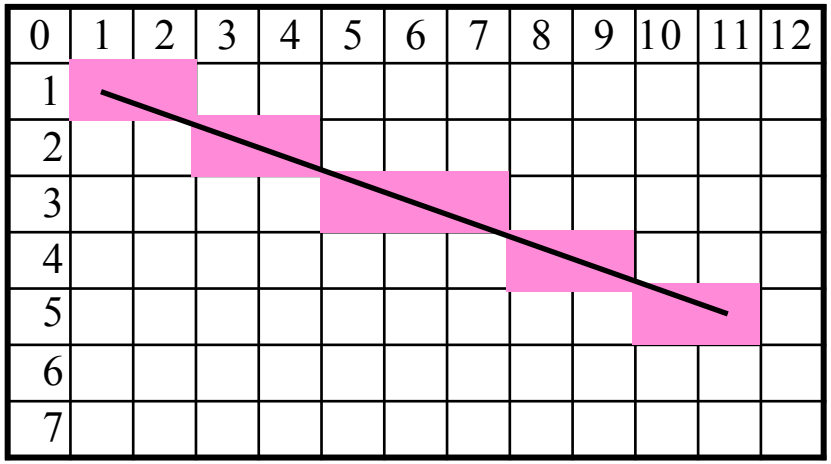
Developed by Jack E. Bresenham in 1962 at IBM.

"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

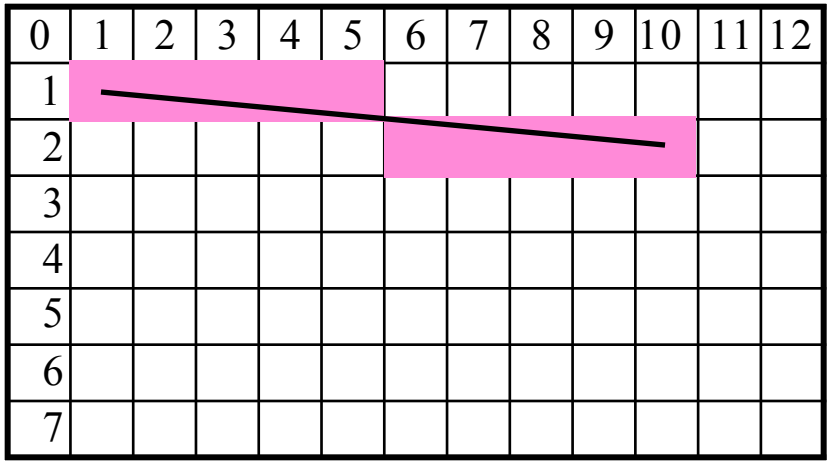
This version assumes: $x_0 < x_1,\ y_0 < y_1,$ slope =< 45 degrees



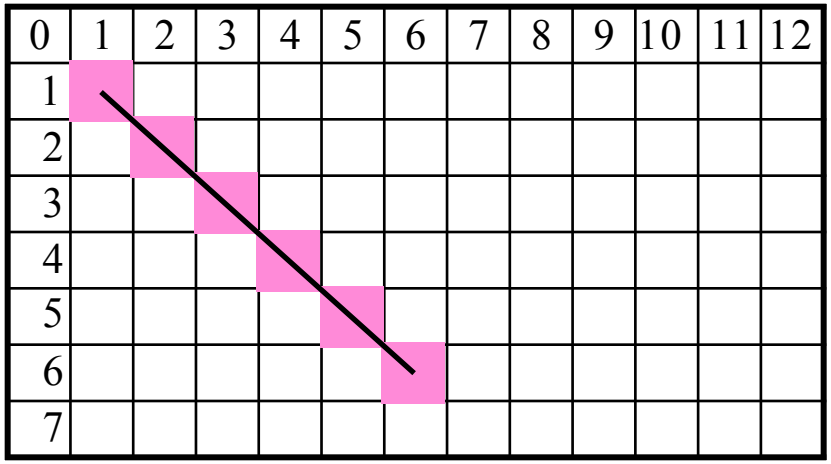

```
function line(x0, x1, y0, y1)
   int deltax := x1 - x0
   int deltay := y1 - y0
   int error := deltax / 2
   int y := y0
   for x from x0 to x1
      plot(x,y)
      error := error - deltay
      if error < 0 then
         y := y + 1
         error := error + deltax
```

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

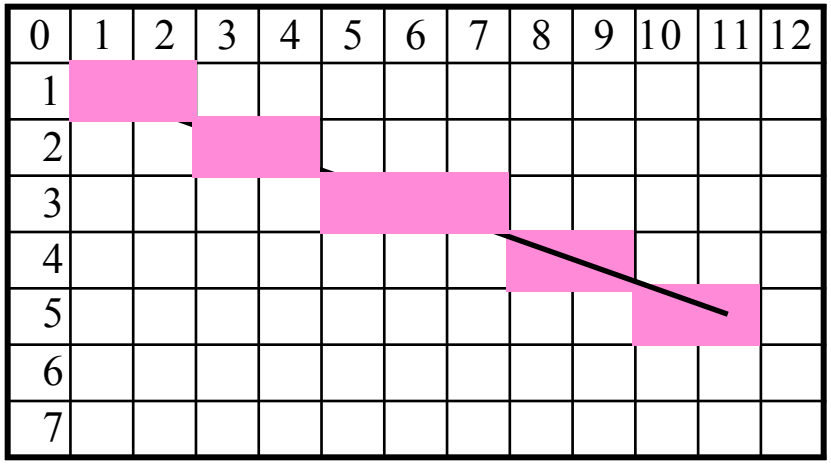


deltay = 1 (very low slope). y only gets incremented once (halfway between x0 and x1)



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

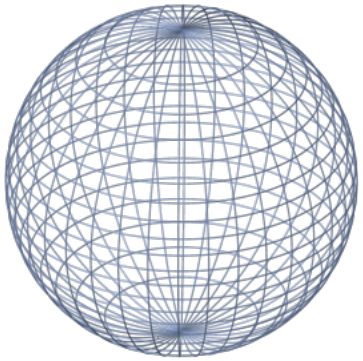
x = 4: plot(4,2)
error = 3 - 4 = -1
y = 2 + 1 = 3
error = -1 + 10 = 9

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.

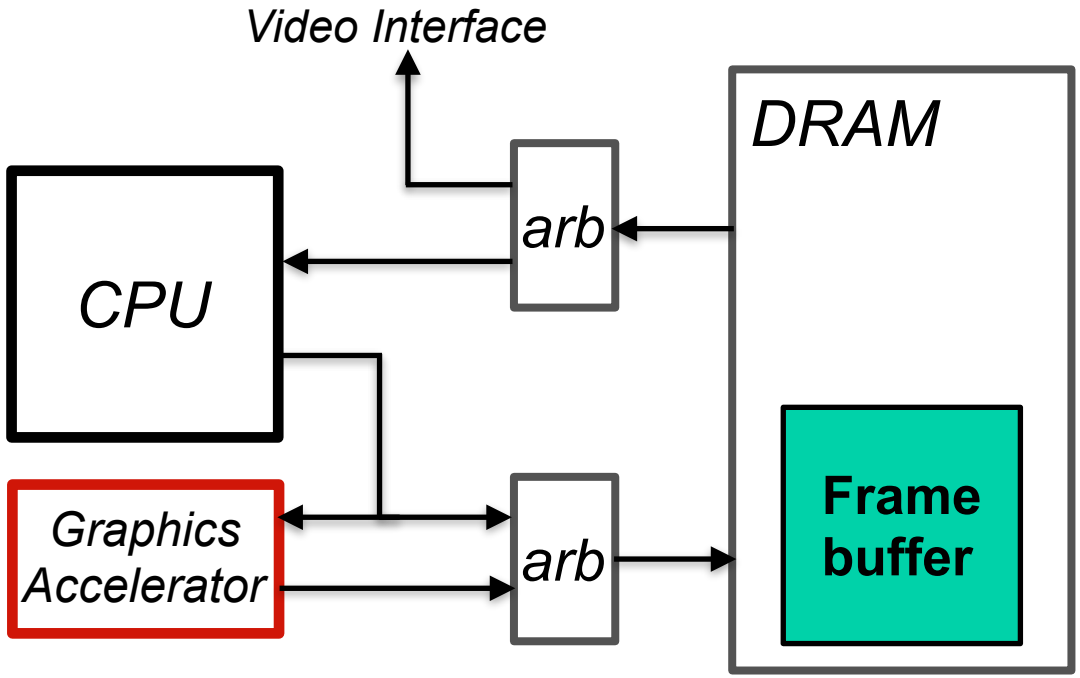
Estimate software performance (RISCV version)

What's needed to do it in hardware?

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

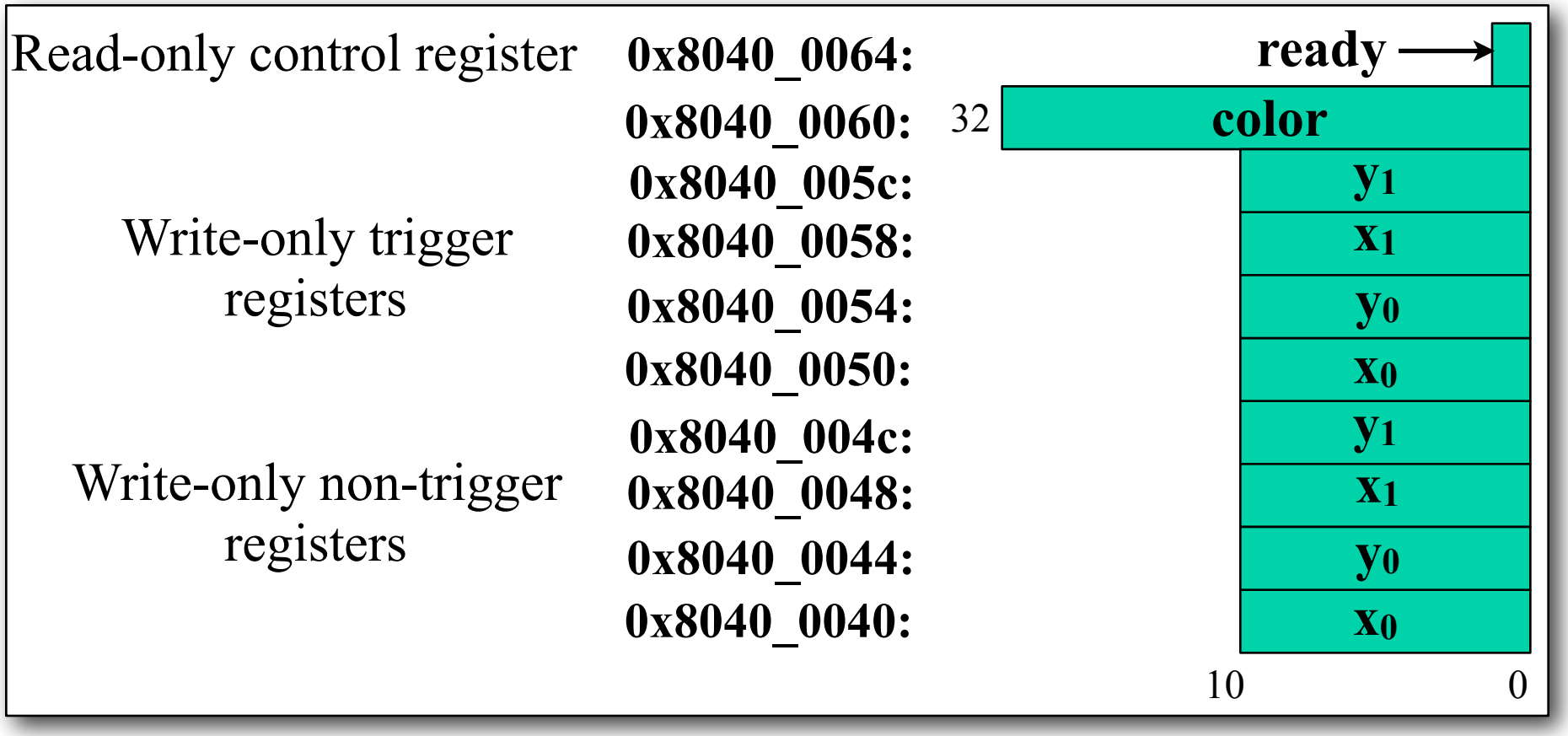
Accelerator Integration

- Arbiters control access to/from DRAM



- CPU initializes line engine by sending pair of points and color value to use. Writes to "trigger" registers initiate line engine.
- Framebuffer (DRAM) has one write port - Shared by CPU and line engine. Priority to CPU - Line engine stalls when CPU writes.

Hardware Implementation Notes



- CPU initializes line engine by sending pair of points and color value to use. Writes to "trigger" registers initiate line engine.