EECS 151/251A
Fall 2017
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek and Nicholas Weaver

Lecture 14

Outline

- Accelerators
- Interfacing

Outline

- Accelerators
- Interfacing

Motivation

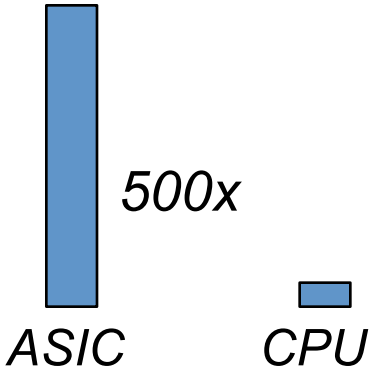
- 90/10 rule:
  - Often 90 percent of the program runtime and energy is consumed by 10 percent of the code (inner-loops).
  - Only small portions of an application become the performance bottlenecks.
  - Usually, these portions of code are data processing intensive with relatively fixed dataflow patterns (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

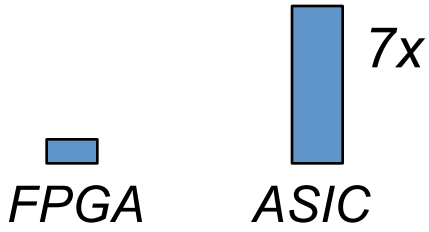
  - Hardware accelerator/economizer implements specialized circuits for inner-loops.
  - Processor packs the noncritical portions (90%), 10% of the computation into minimal space.

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

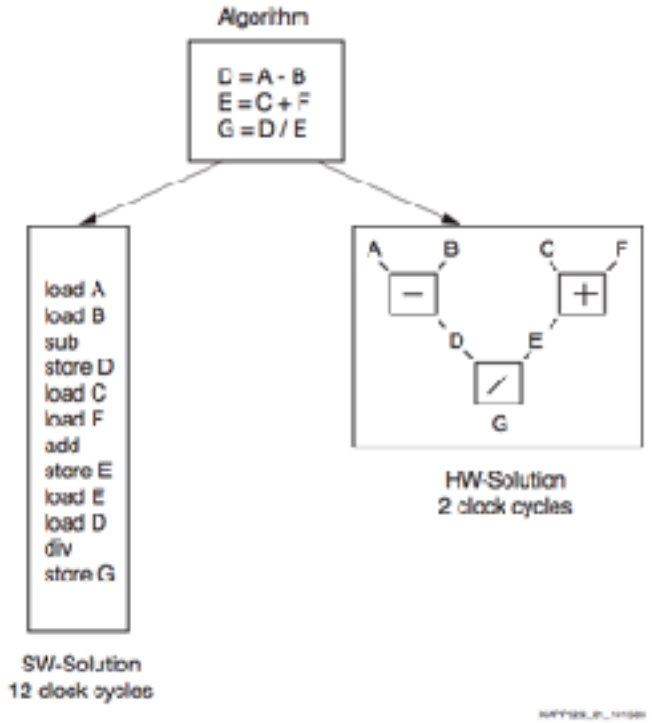


$\therefore$ *FPGA : CPU = 70x*

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

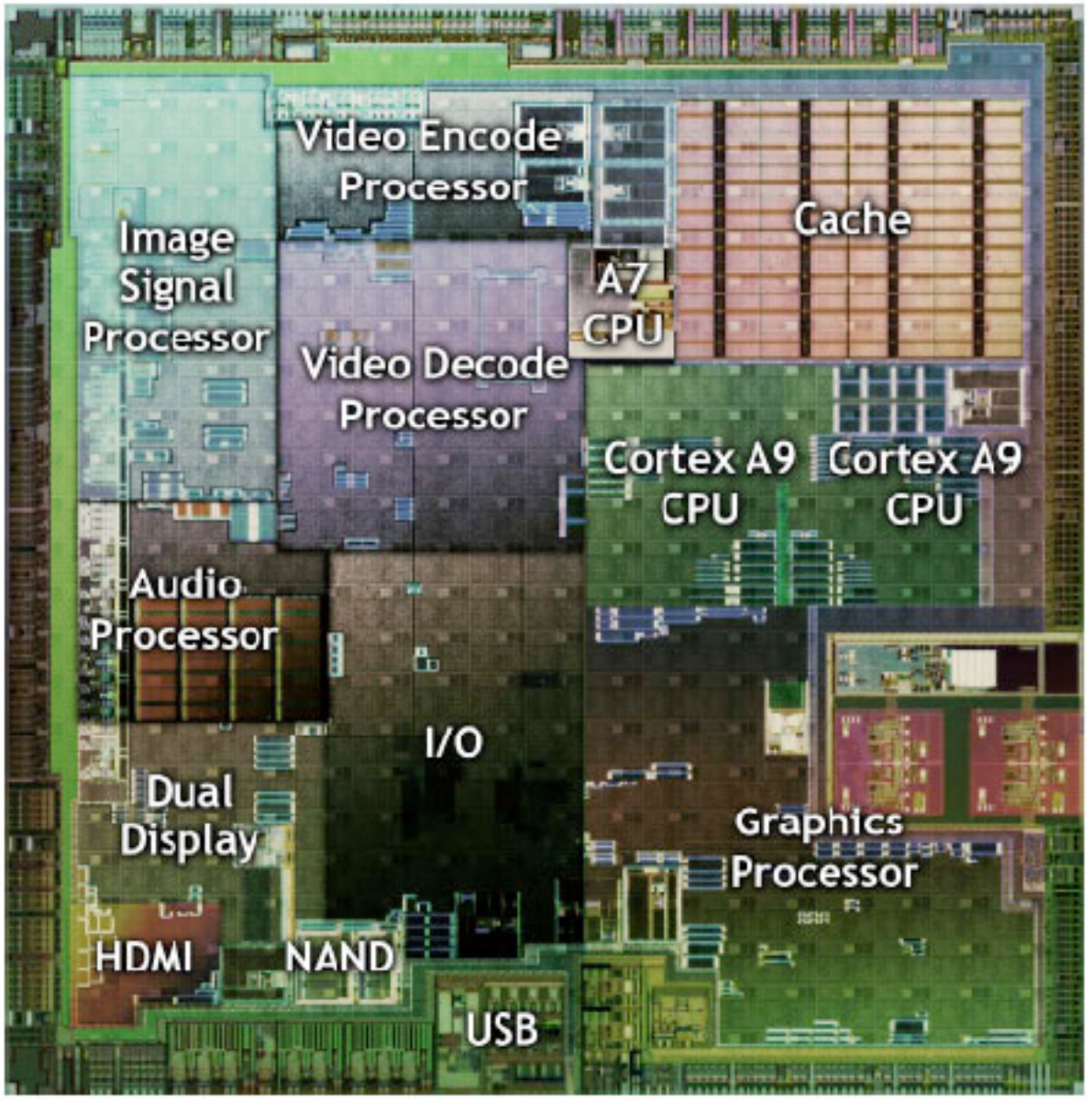
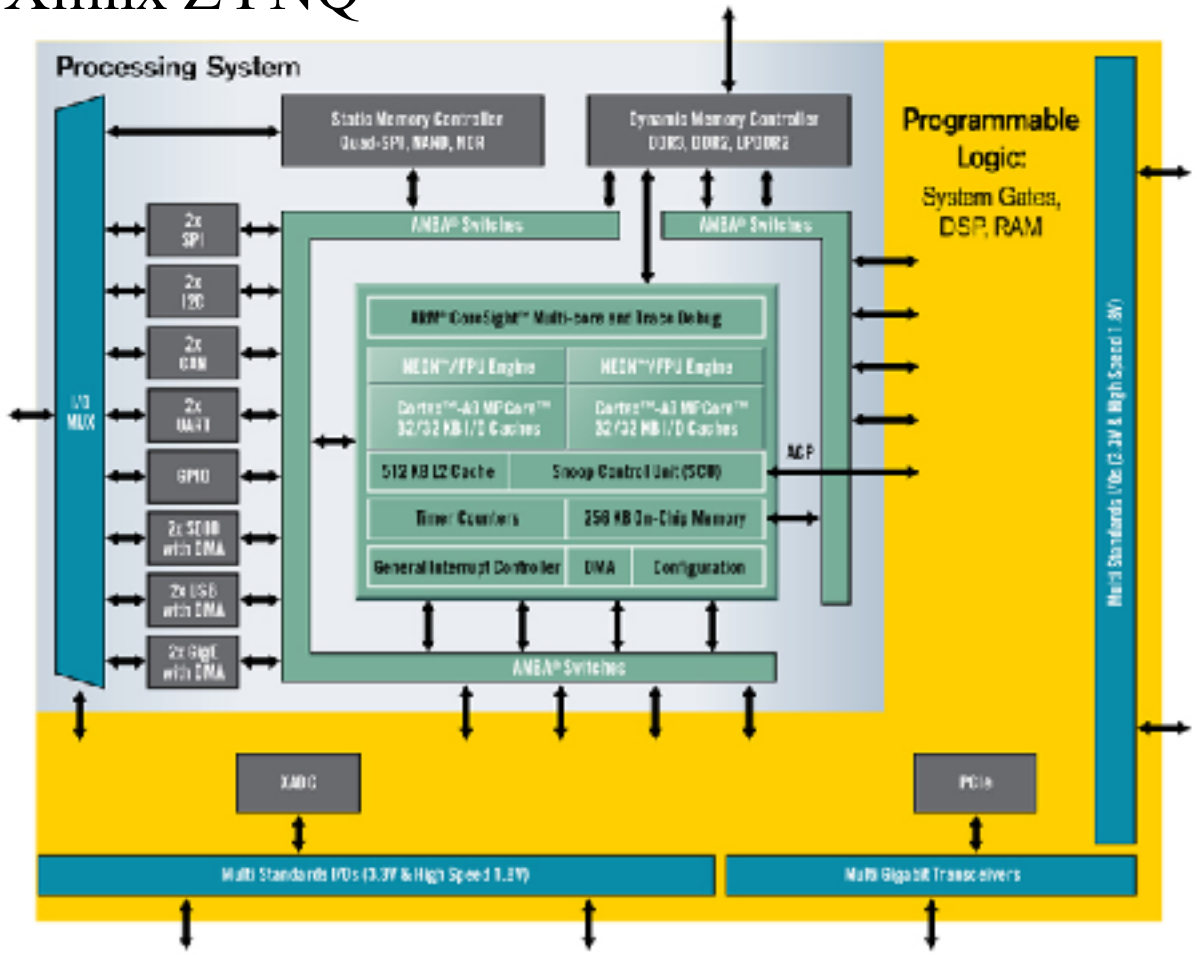


Figure 1 - NVIDIA Tegra 2 System on a Chip

Processors in FPGAs

Xilinx ZYNQ

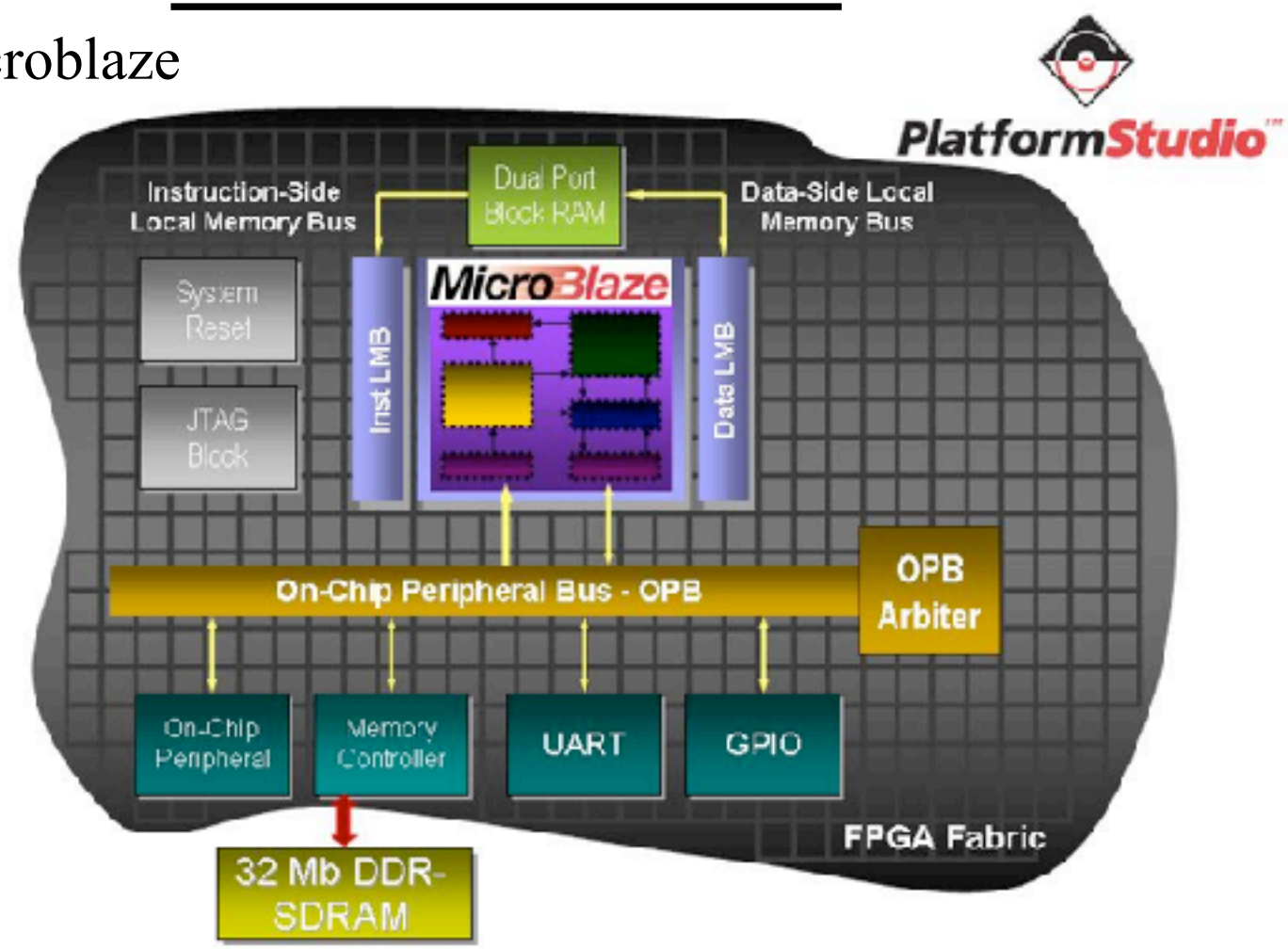


- Dual ARM Cortex™-A9 MPCore
  - Up to 800MHz
  - Enhanced with NEON Extension and Single & Double Precision Floating point unit
  - 32kB Instruction & 32kB Data L1 Cache
- Unified 512kB L2 Cache
- 256kB on-chip Memory
- DDR3, DDR2 and LPDDR2 Dynamic Memory Controller
- 2x QSPI, NAND Flash and NOR Flash 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:
  - 26k to 235k Logic Cells (approximately 430k to 3.5M of equivalent ASIC Gates)
  - 240kB to 1.86MB of Extensible Block RAM
  - 80 to 780 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**

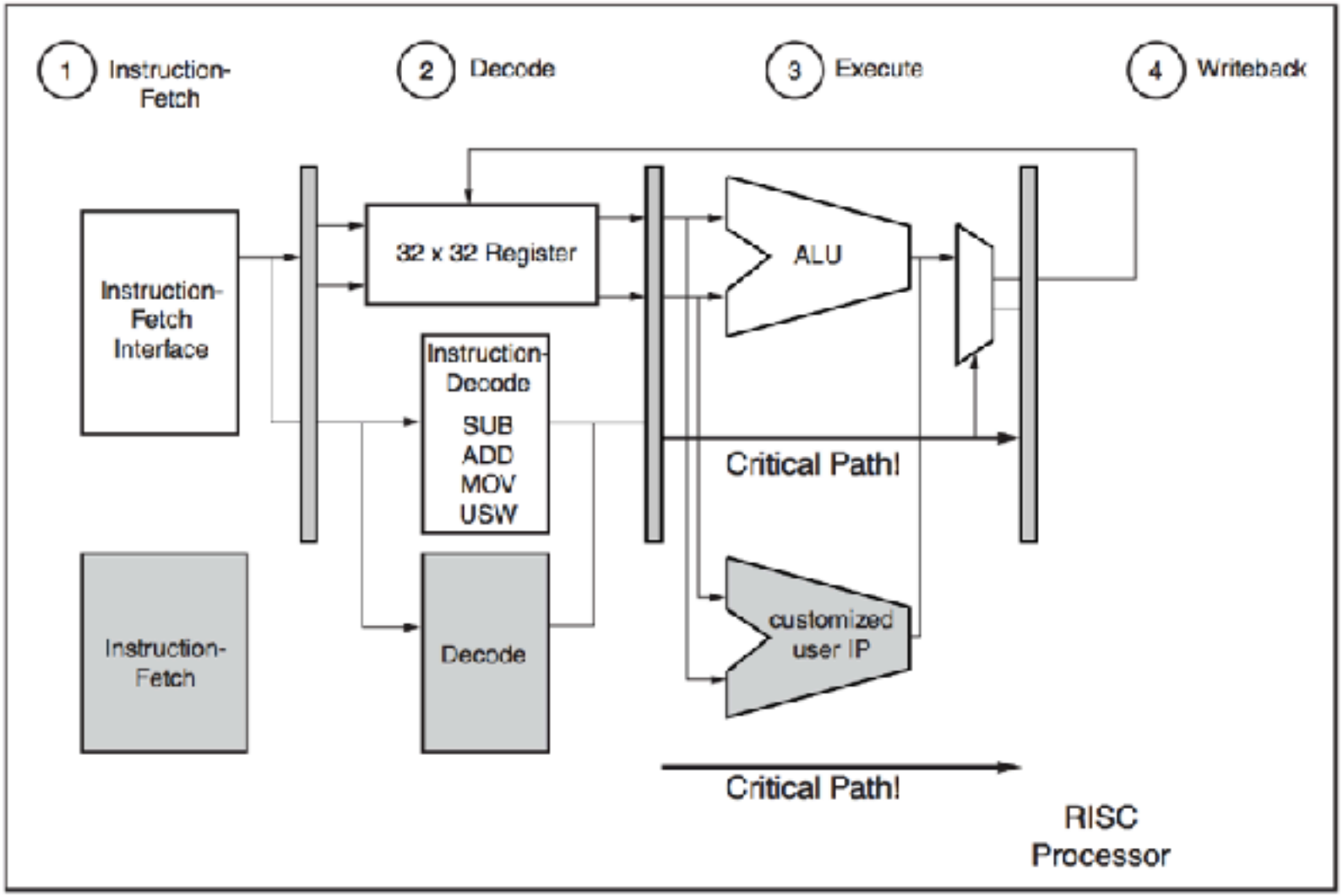
Soft Processors

Xilinx: Microblaze

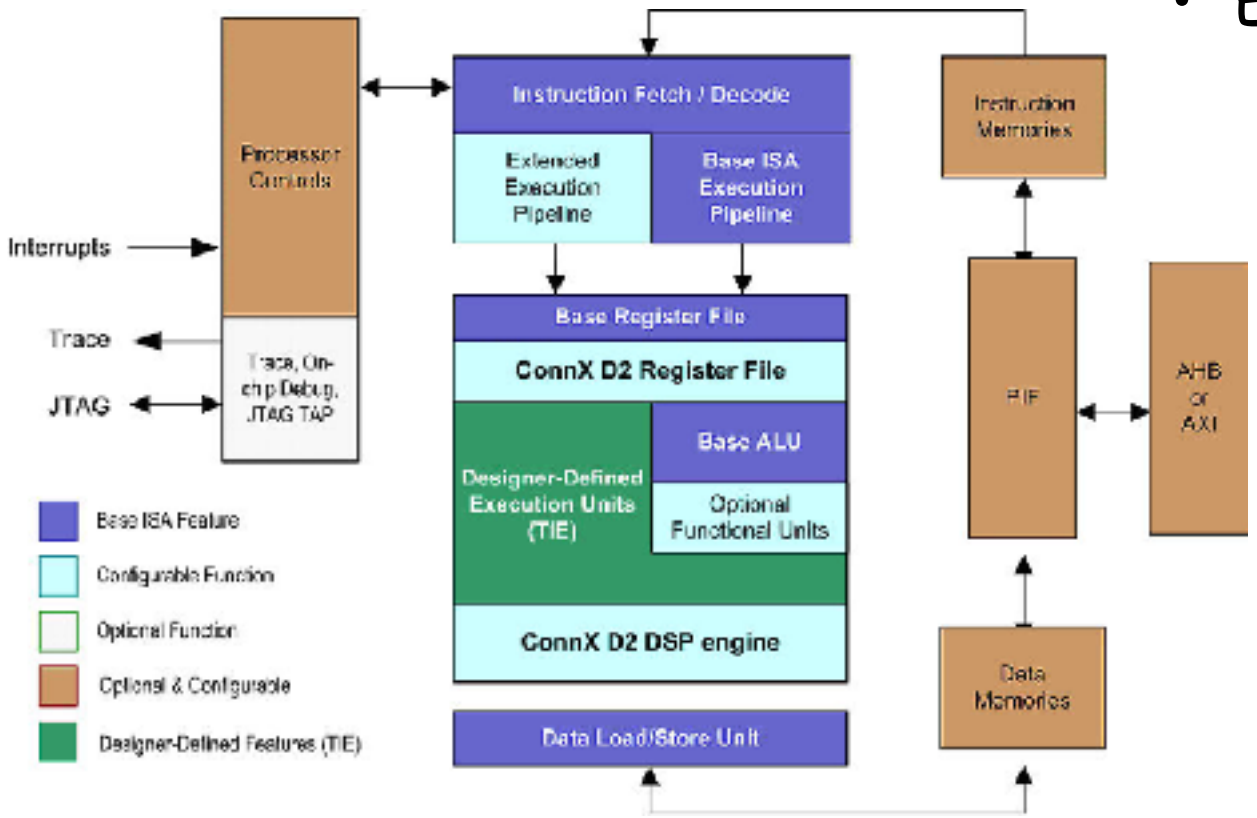


Intel/Altera: Nios

Custom Hardware in the Pipeline

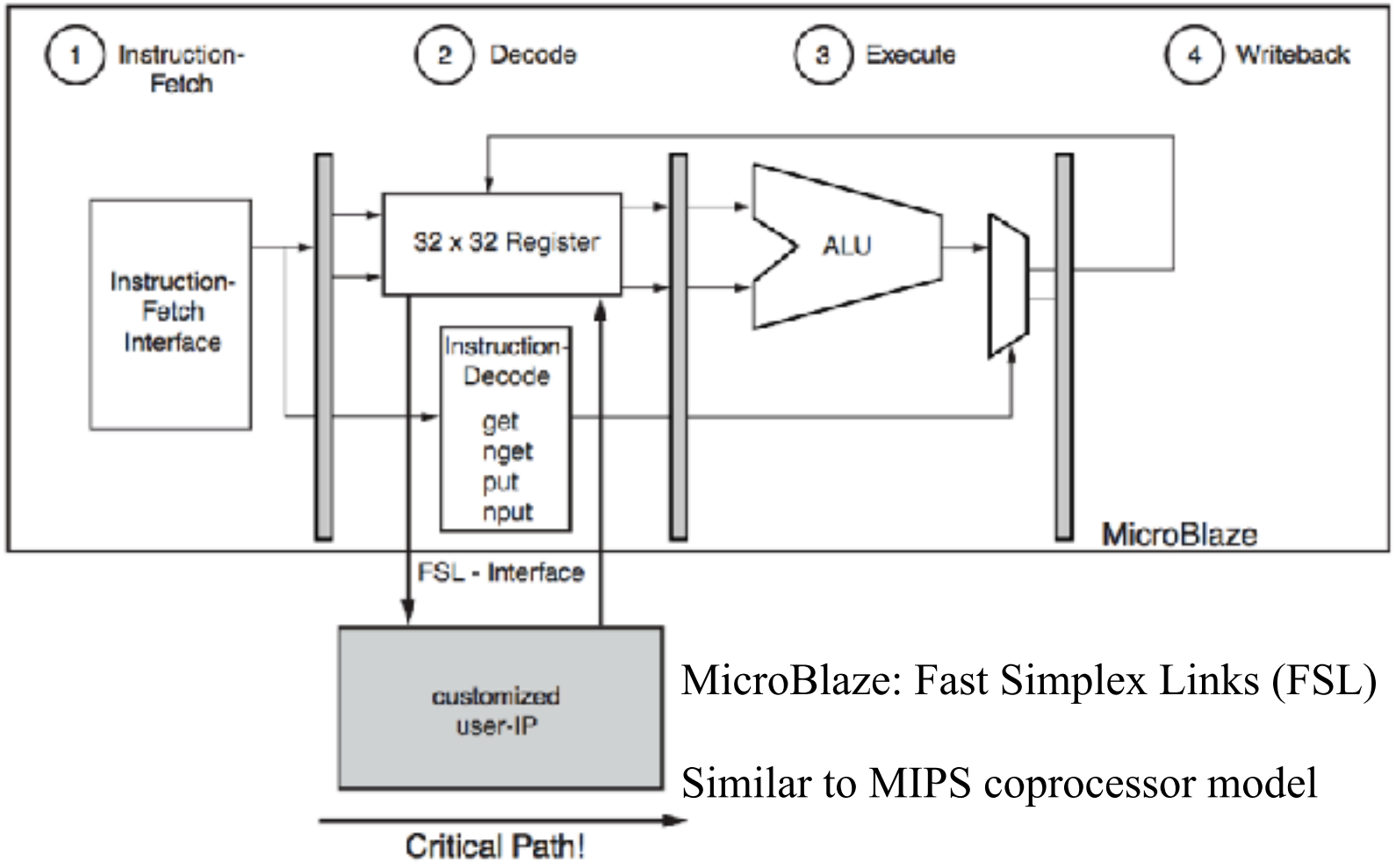


Custom Instructions



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

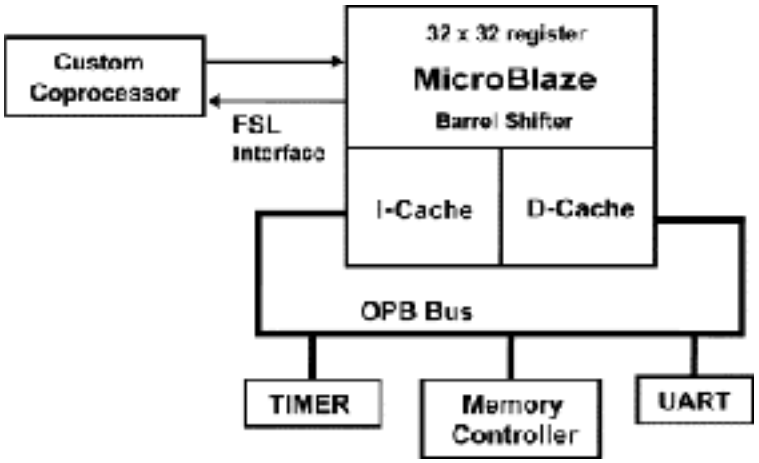
Tightly Coupled Co-processor



MicroBlaze: Fast Simplex Links (FSL)

Similar to MIPS coprocessor model

MicroBlaze Fast Simplex Links



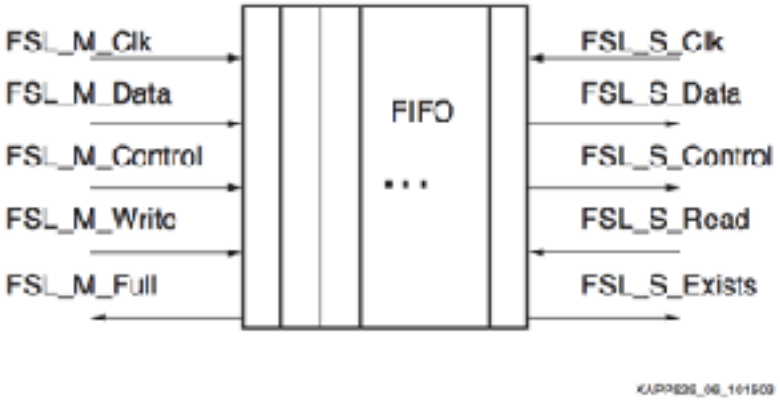
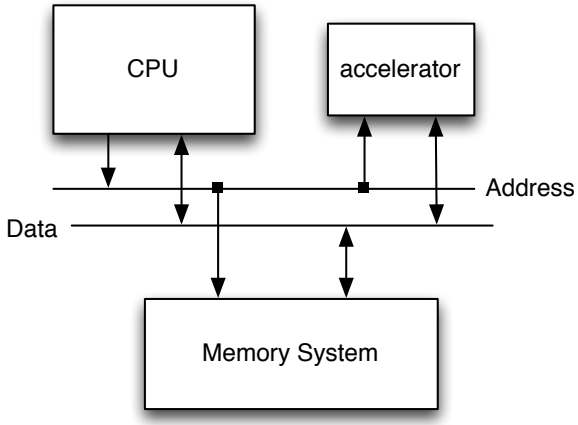


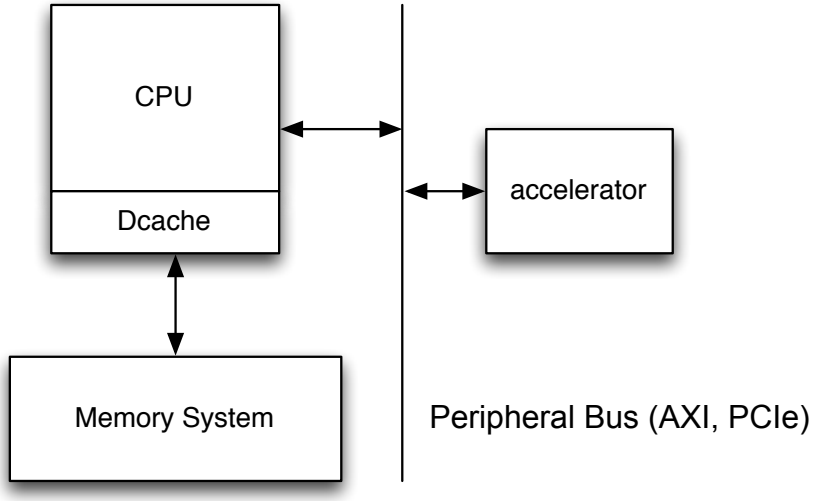
Figure 5: **FSL interface**

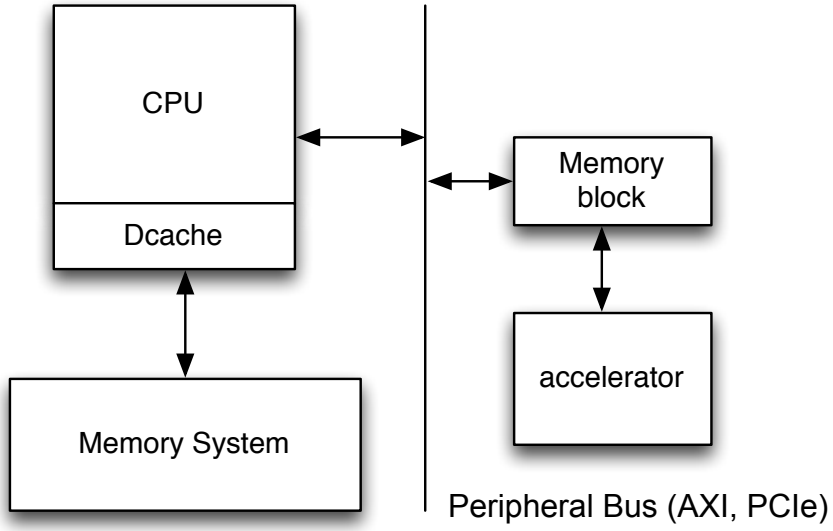
```
// Blocking Data Read and Write to Local Link no. id
microblaze_bread_datafsl(val, id)
microblaze_bwrite_datafsl(val, id)

// 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

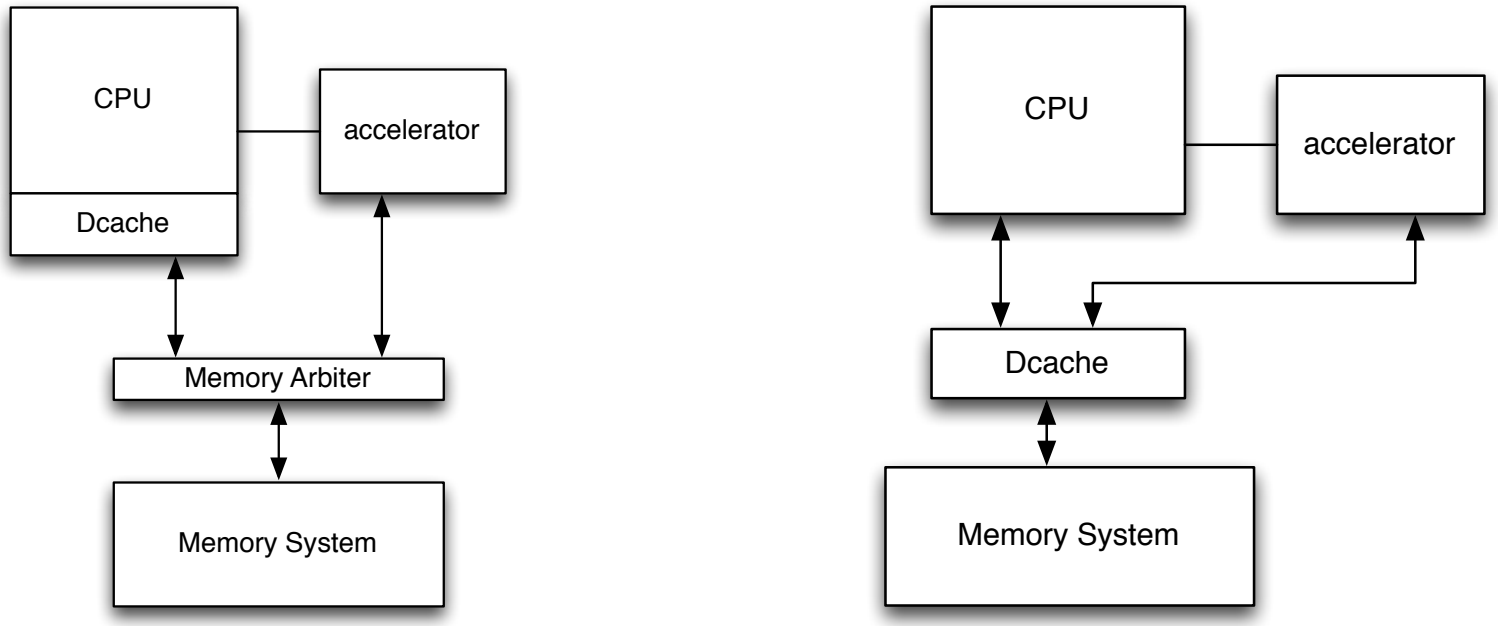


- Memory mapped control/ data registers



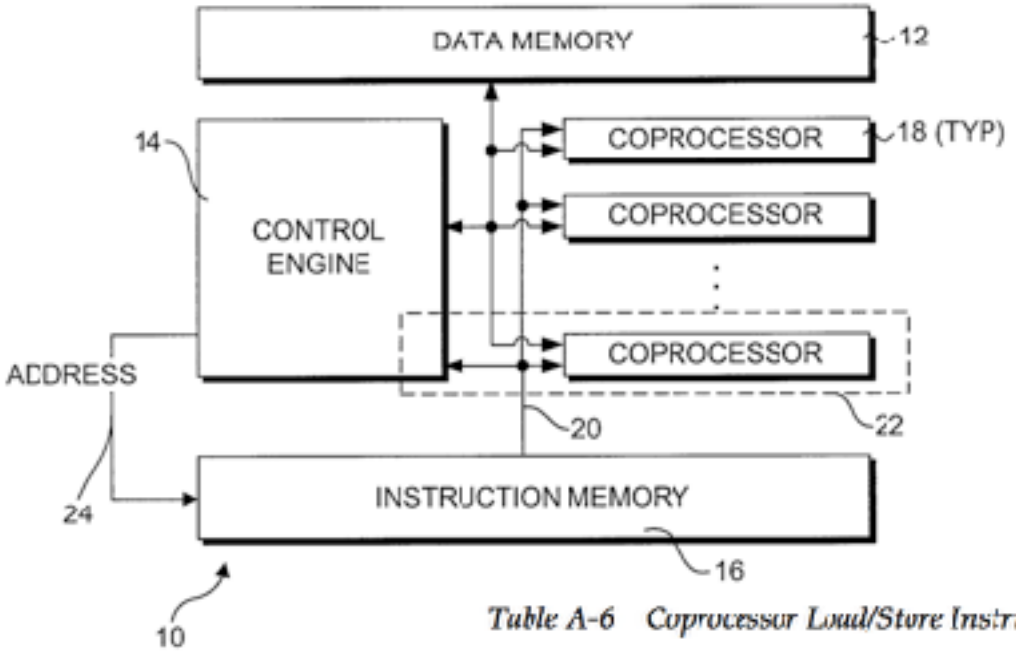


CPU/Accelerator Shared Memory



- 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?

Tightly Coupled Co-processor



- MIPS: load/store to/ from coprocessor, coprocessor op

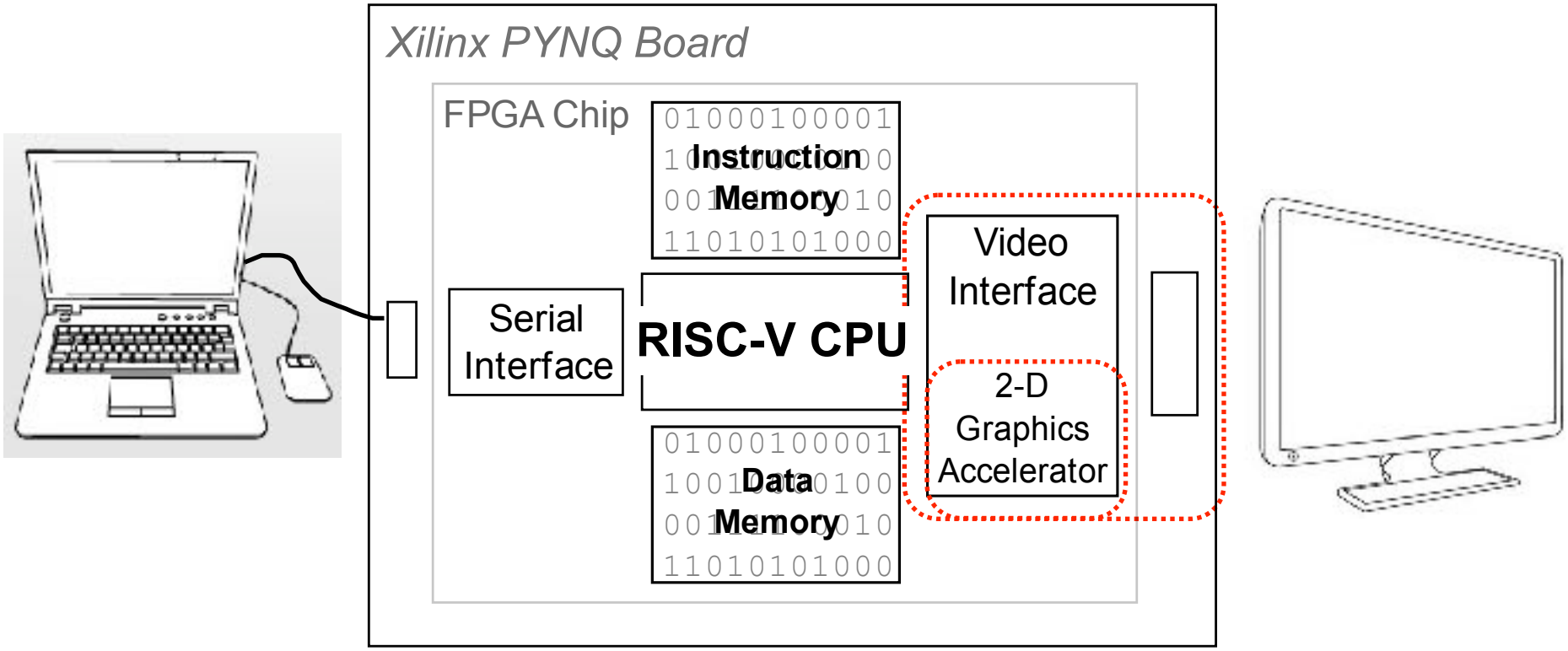
Table A-6 Coprocessor Load/Store Instructions

Mnemonic	Description	Defined in
LWCz	Load Word to Coprocessor-z	MIPS I
SWCz	Store Word from Coprocessor-z	I
LDCz	Load Doubleword to Coprocessor-z	II
SDCz	Store Doubleword from Coprocessor-z	II

Table A-7 FPU Load/Store Instructions Using Register + Register Addressing

Mnemonic	Description	Defined in
LWXC1	Load Word Indexed to Floating Point	MIPS IV
SWXC1	Store Word Indexed from Floating Point	IV
LDXC1	Load Doubleword Indexed to Floating Point	IV
SDXC1	Store Doubleword Indexed from Floating Point	IV

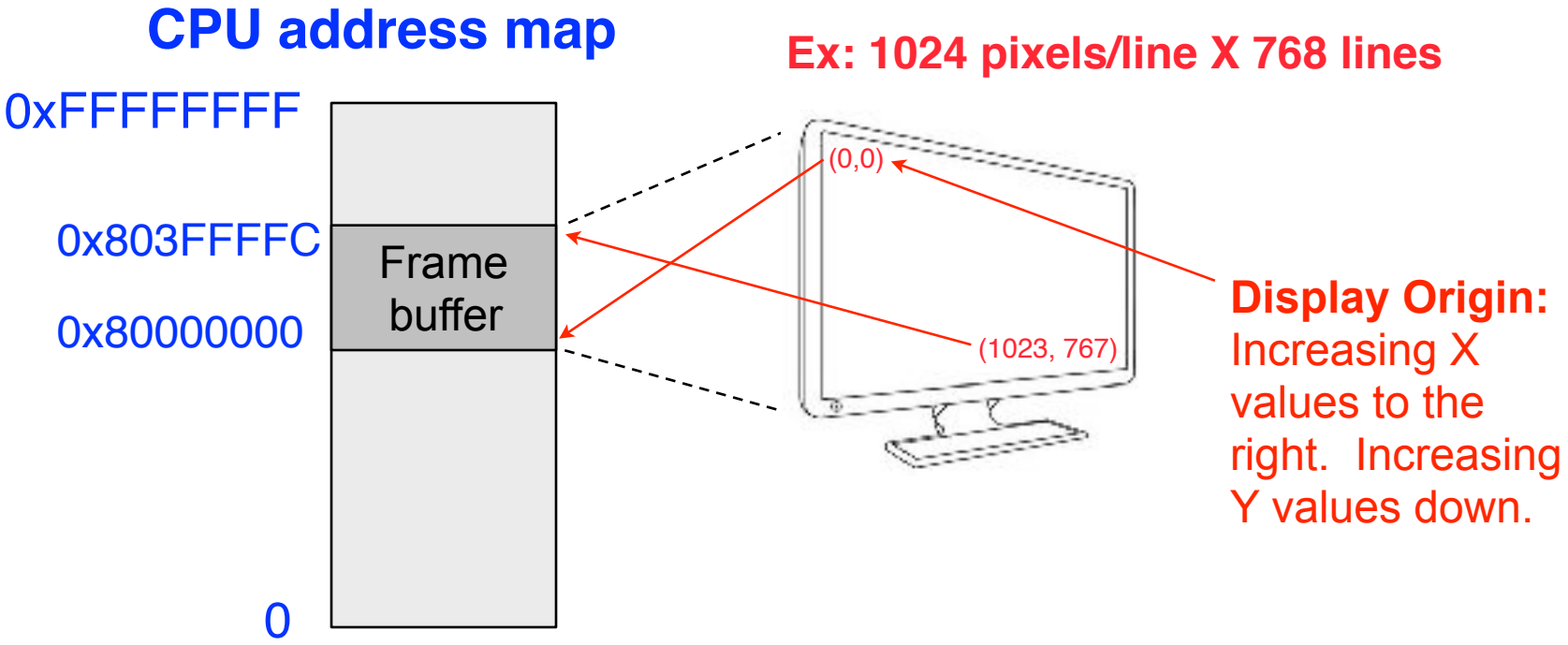
RISCV-151 Video Subsystem



- 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

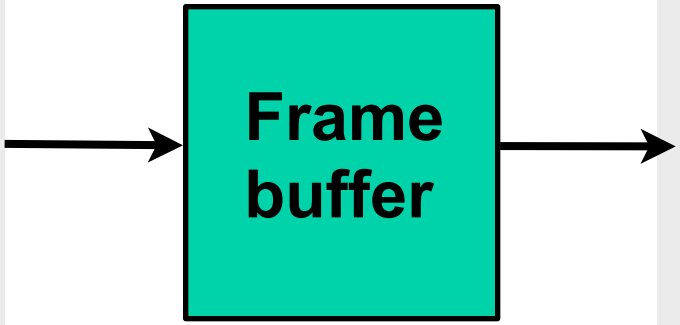
- A range of memory addresses correspond to the display.
- CPU writes (using sw instruction) pixel values to change display.
- No synchronization required. Independent process reads pixels from memory and sends them to the display interface at the required rate.



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.

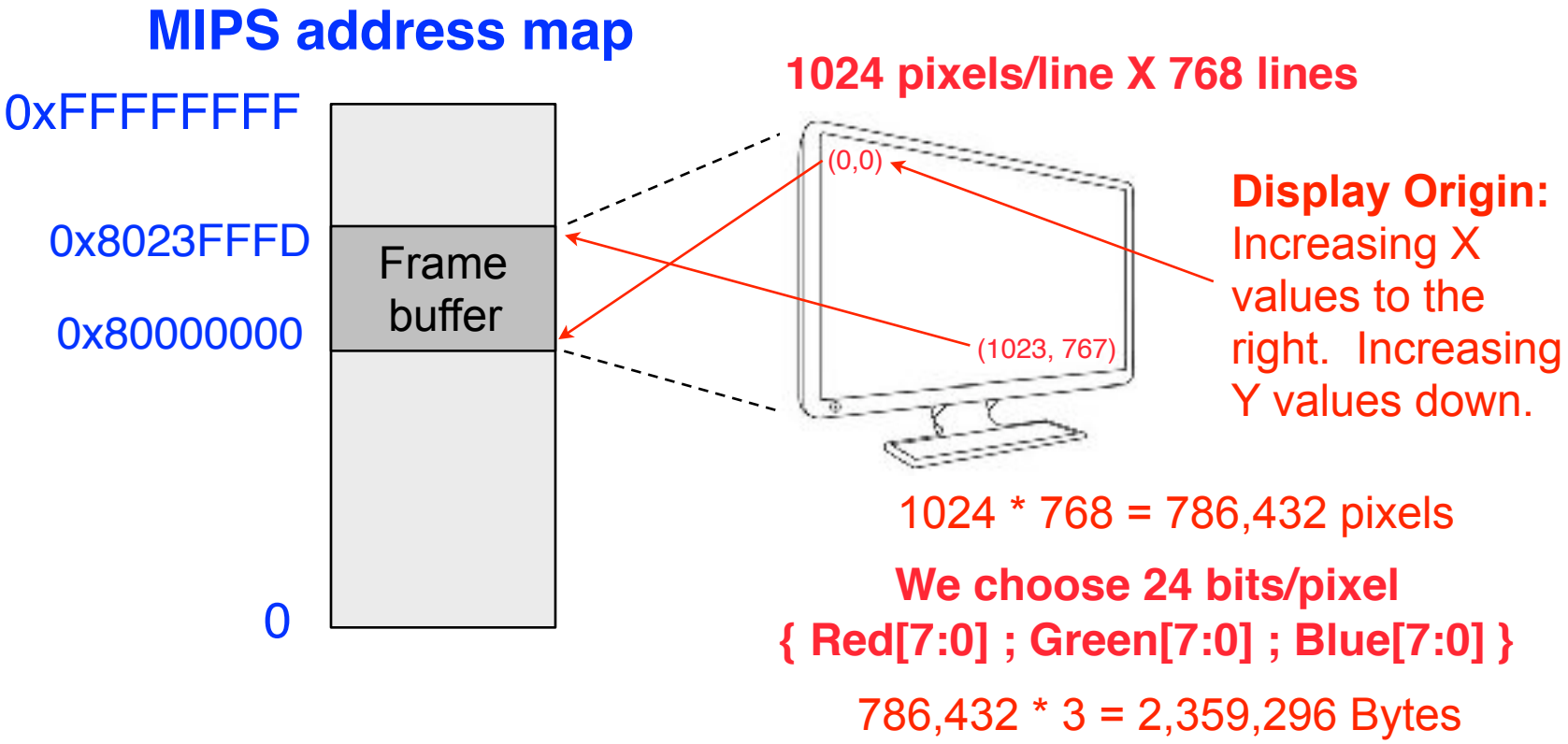


Video Interface continuously reads pixel locations in scan-line order and sends to physical display.

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

1024 x 768 pixels/frame x 24 bits/pixel

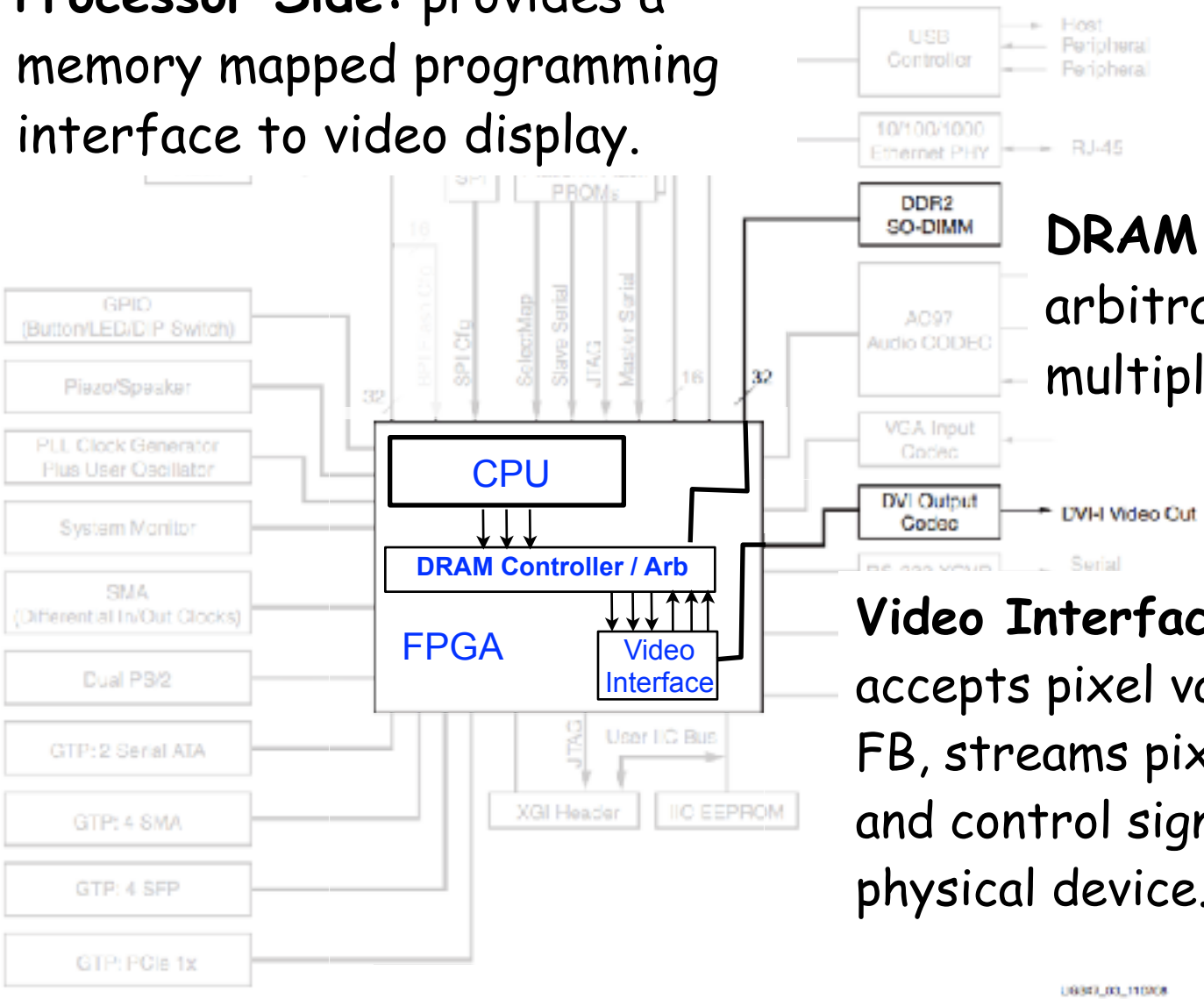
Memory Mapped Framebuffer



- Total memory bandwidth needed to support frame buffer?

Frame Buffer Physical Interface

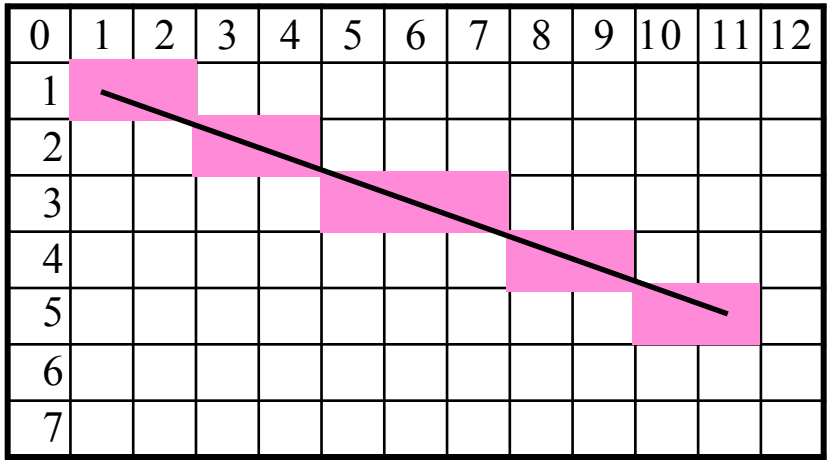
Processor Side: provides a memory mapped programming interface to video display.



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: slow or expensive!

Bresenham Line Drawing Algorithm

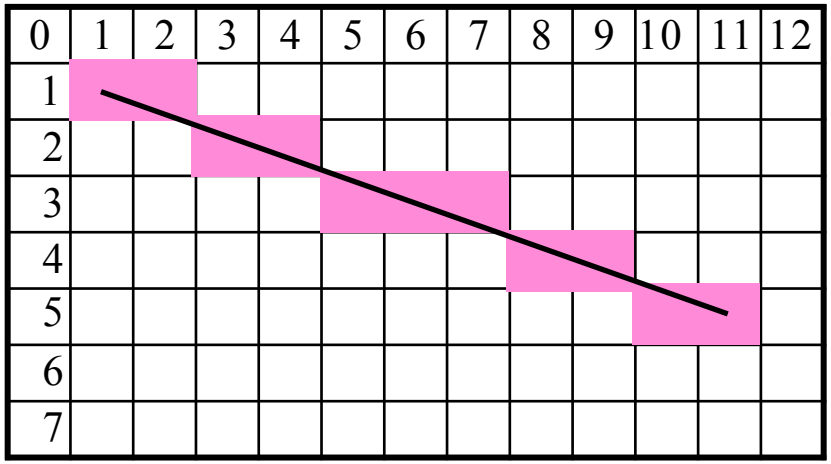
Developed by Jack E. Bresenham in 1962 at IBM.

"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

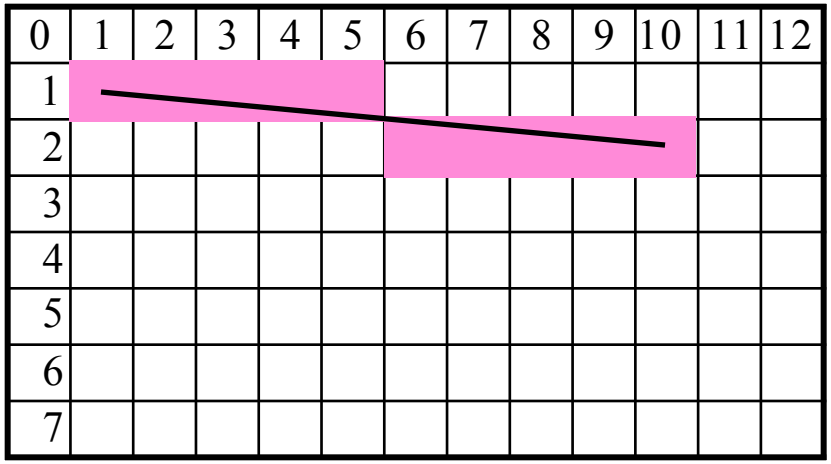
This version assumes: $x_0 < x_1,\ y_0 < y_1,$ slope =< 45 degrees



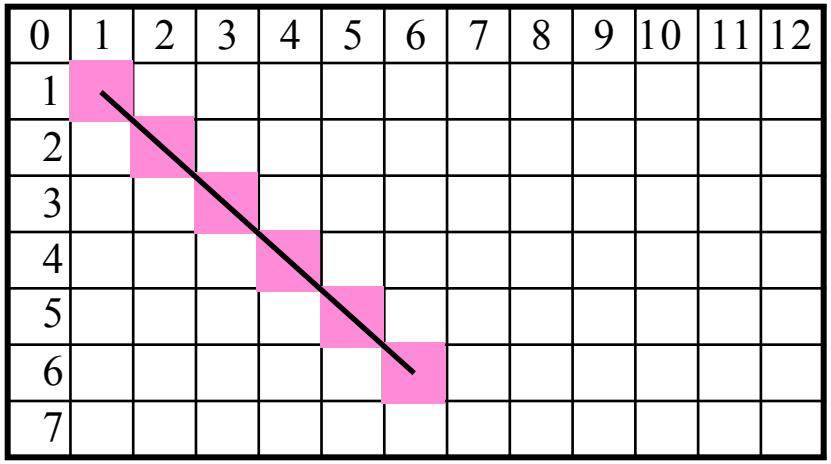

```
function line(x0, x1, y0, y1)
   int deltax := x1 - x0
   int deltay := y1 - y0
   int error := deltax / 2
   int y := y0
   for x from x0 to x1
      plot(x,y)
      error := error - deltay
      if error < 0 then
         y := y + 1
         error := error + deltax
```

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

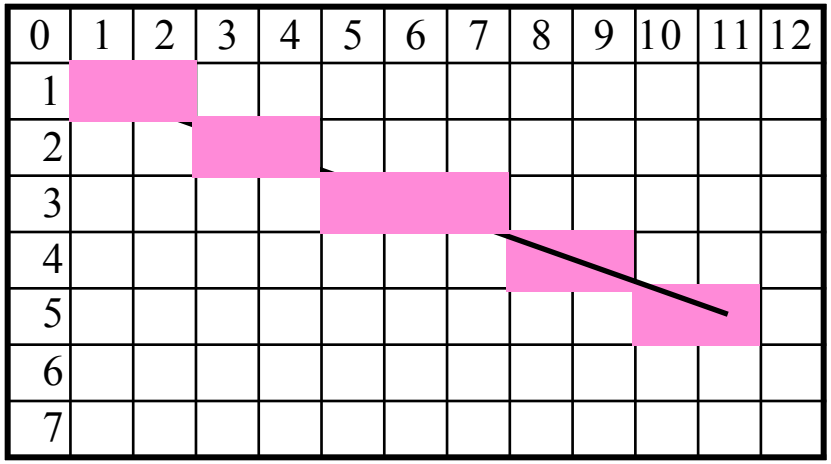


deltay = 1 (very low slope). y only gets incremented once (halfway between x0 and x1)



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

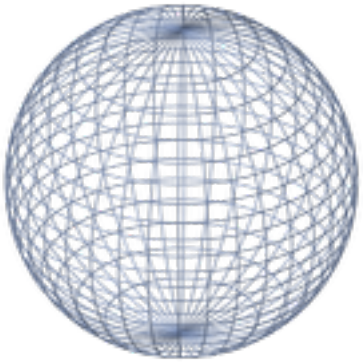
x = 4: plot(4,2)
error = 3 - 4 = -1
y = 2 + 1 = 3
error = -1 + 10 = 9

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.

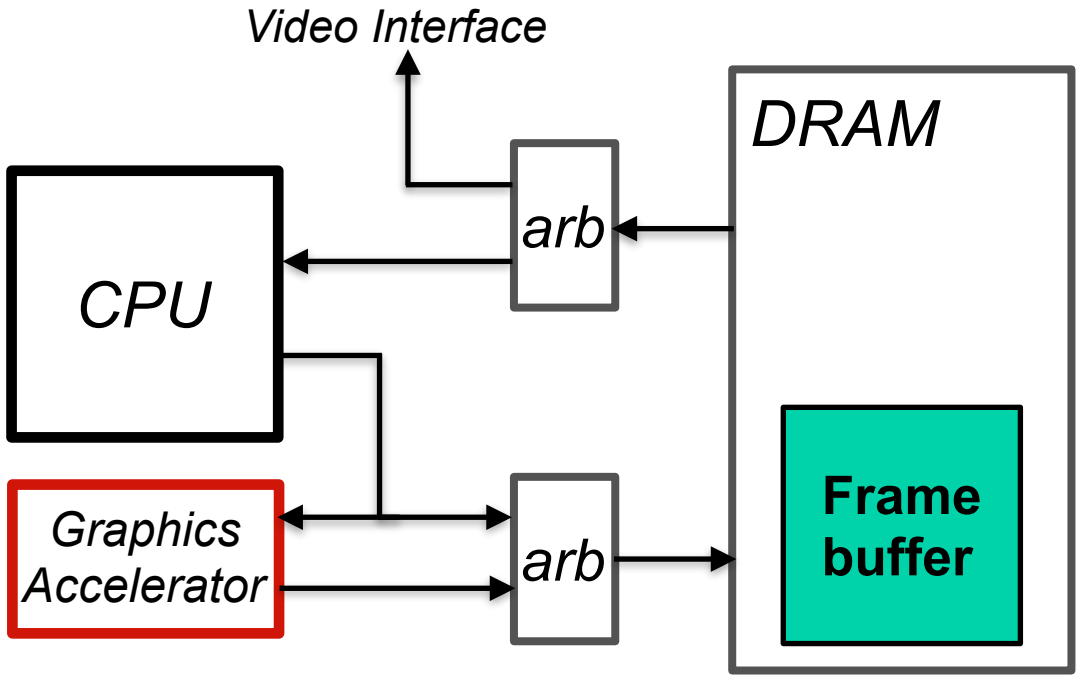
Estimate software performance (RISCV version)

What's needed to do it in hardware?

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

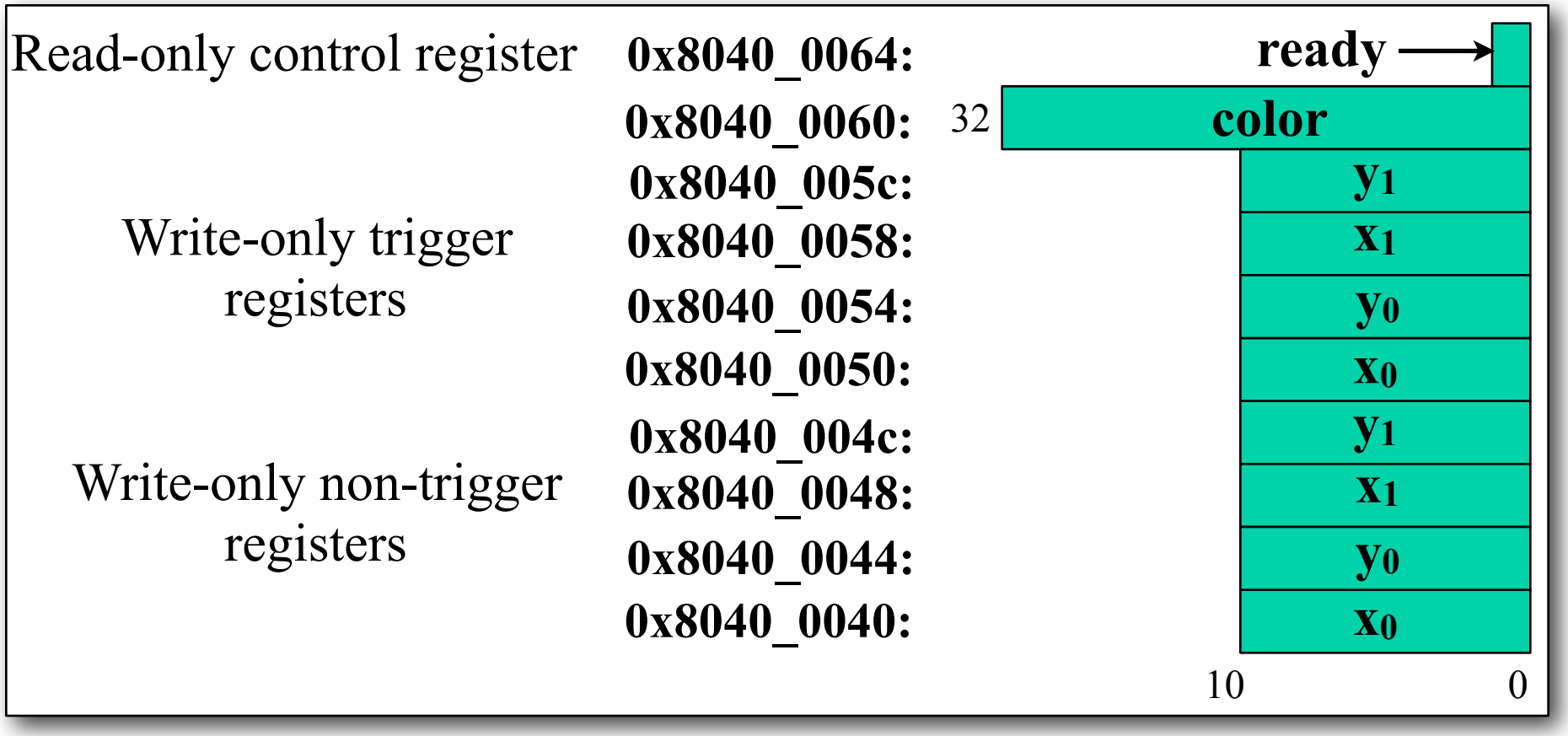
Accelerator Integration

- 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



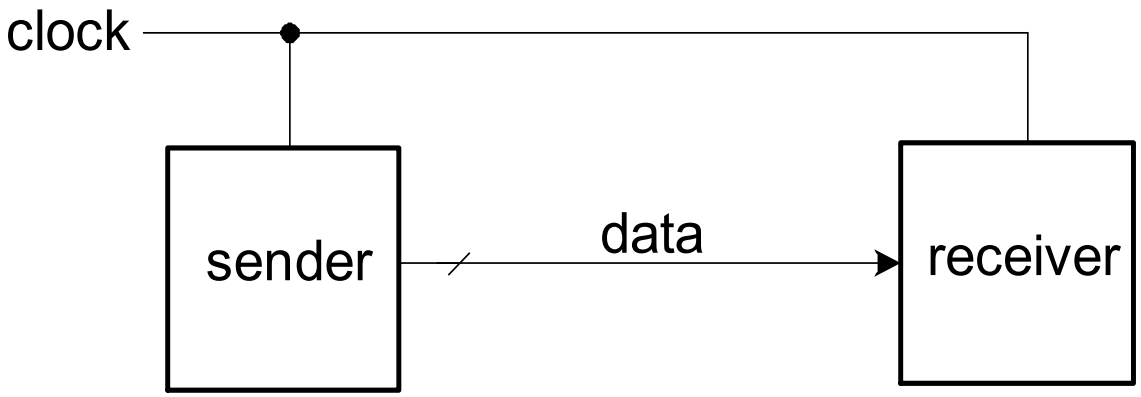
- CPU initializes line engine by sending pair of points and color value to use. Writes to "trigger" registers initiate line engine.

Outline

- Accelerators
- Interfacing
  - How do components (modules) communicate?

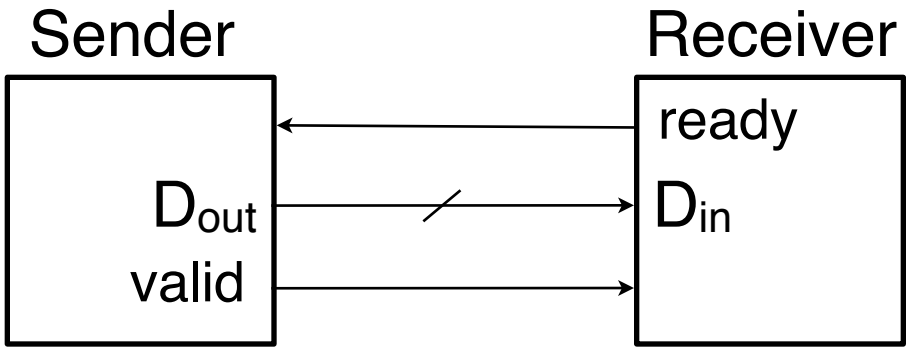
Synchronous Data Transfer

- In synchronous systems, the clock signal is used to coordinate the movement of data around the system.
- Take for example, transferring from module to module:



- By design, the clock period is sufficiently long to accommodate wire delay and time to get the data into the receiver.
- Assumes:
  - sender is ready to send data on each cycle & receiver is ready to receive data on each cycle
- What if the communication is sporadic?

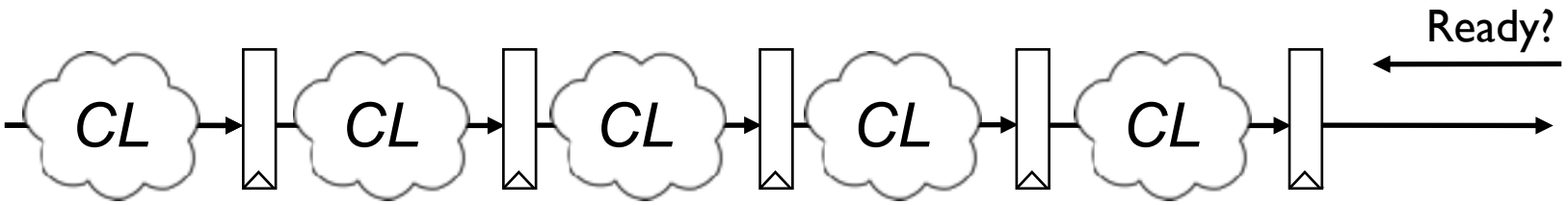
Data Transfer with Flow-Control



- After checking to see if the receiver is ready to receive, the sender asserts the "valid" signal to indicate that the data lines hold data for transferring
- As with synchronous transfer, the sender assumes that the transfer happens successfully
- Design constraint: the ready signal needs to be stable early enough in the cycle to allow the sender to respond with data and the valid signal before set-up time of receiver.
- Can we pipeline the control?

Output Stall Complication

Pipeline should only move data to right if output ready to accept next item

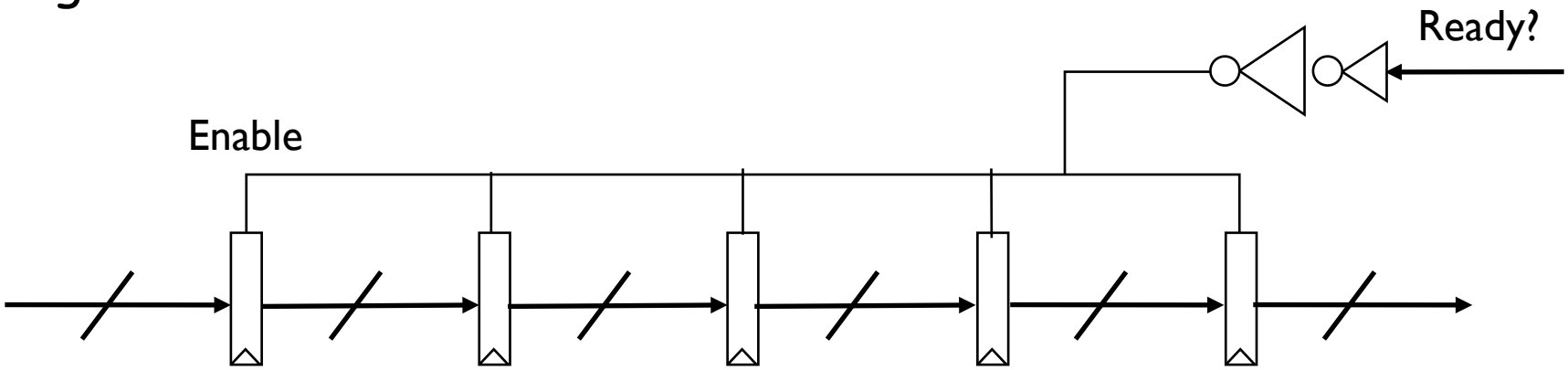


What complication does this introduce?

Need to fan out to enable signal on each flop

Stall Fan-Out Example

- 200 bits per shift register stage, 16 stages
- 3200 flip-flops
- How many fanout-of-four gate delays to buffer up ready signal?



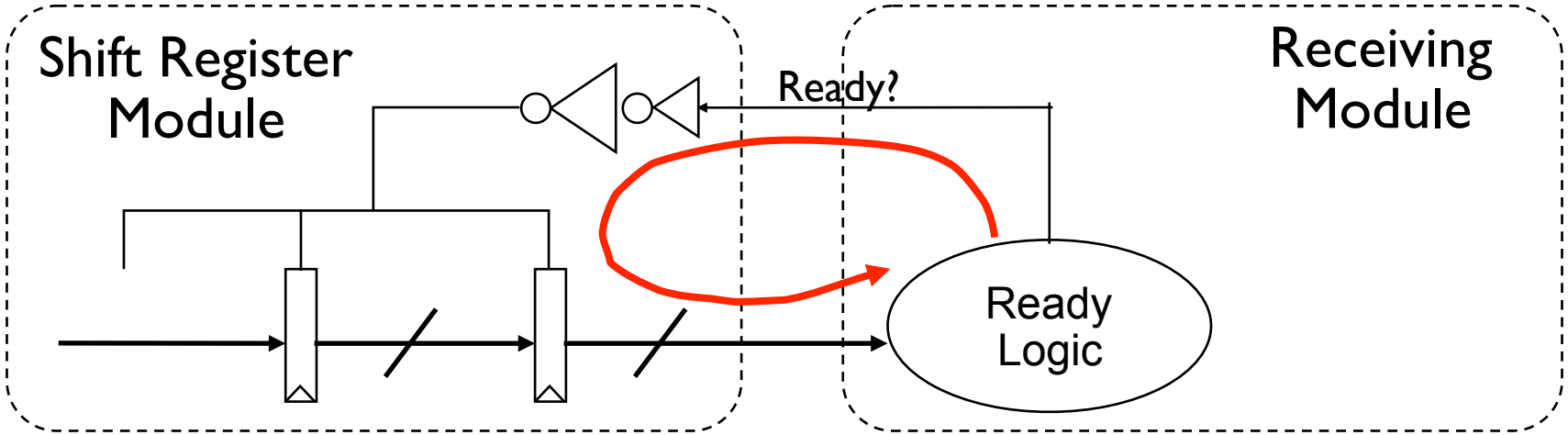
- $Log_4(3200) = 5.82$, ~ 6 FO4 delays!

This doesn't include any penalty for driving enable signal wires!

Loops Prevent Arbitrary Resizing

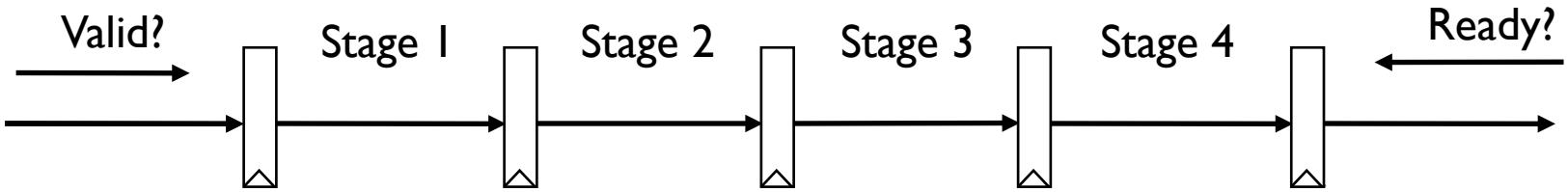
We could increase size of gates in ready logic block to reduce fan out required to drive ready signal to flop enables…

But this increases load on flops, so they have to get bigger --- a vicious cycle!

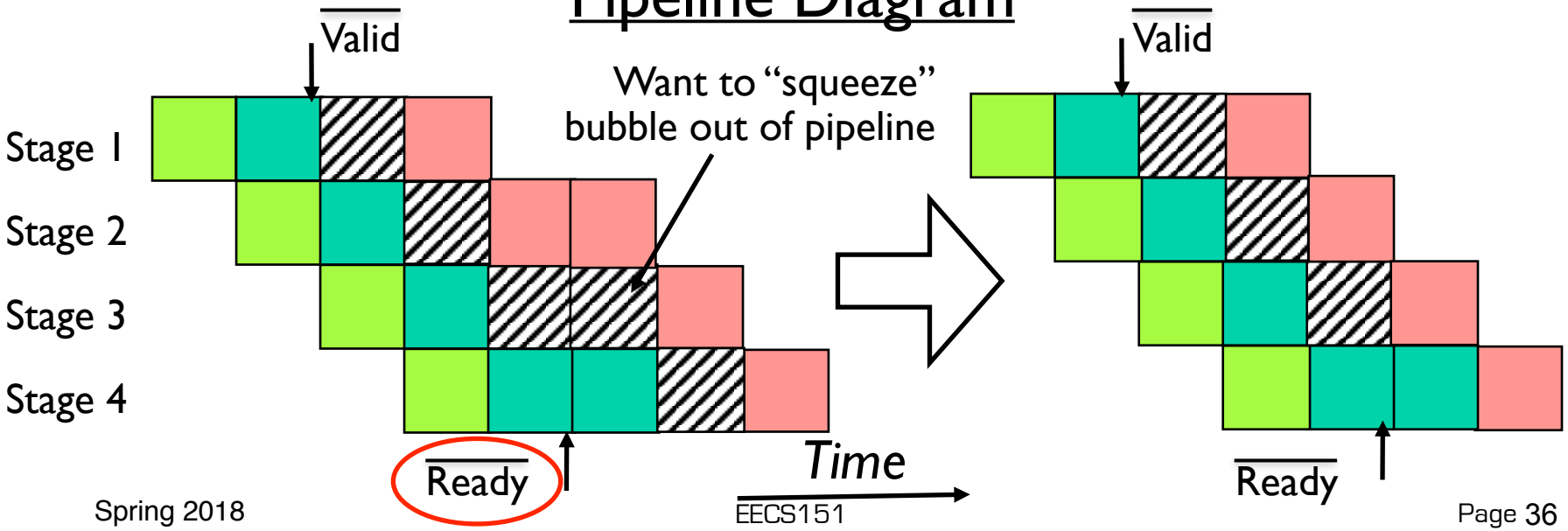


Second Complication: Input Bubbles

Sender doesn't have valid data every clock cycle, so empty "bubbles" inserted into pipeline

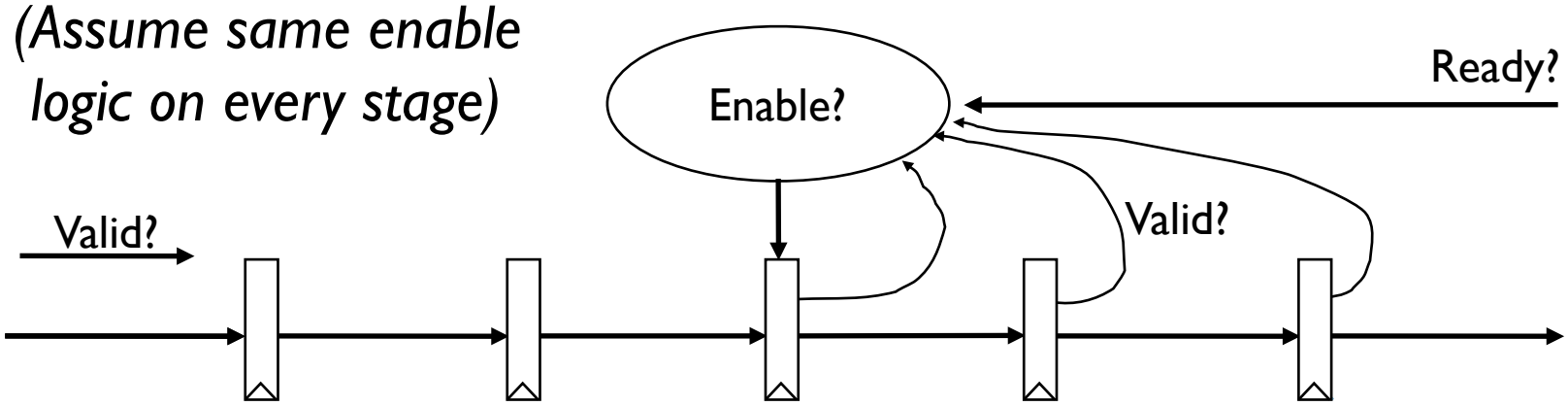


Pipeline Diagram



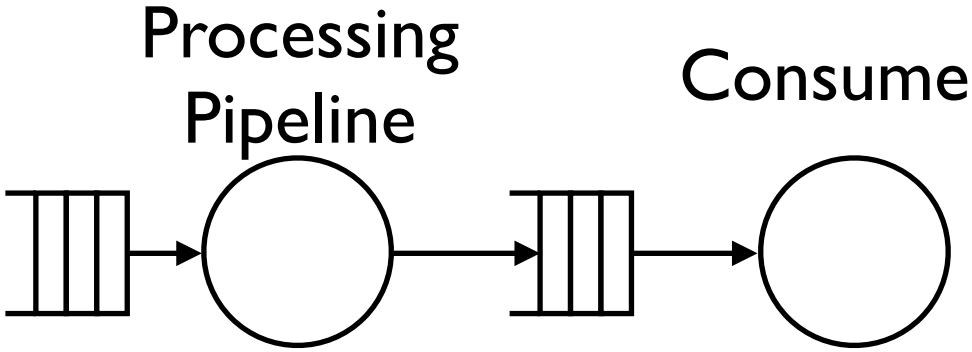
Logic to Squeeze Bubbles

Can move one stage to right if Ready asserted, or if there are any bubbles in stages to right of current stage



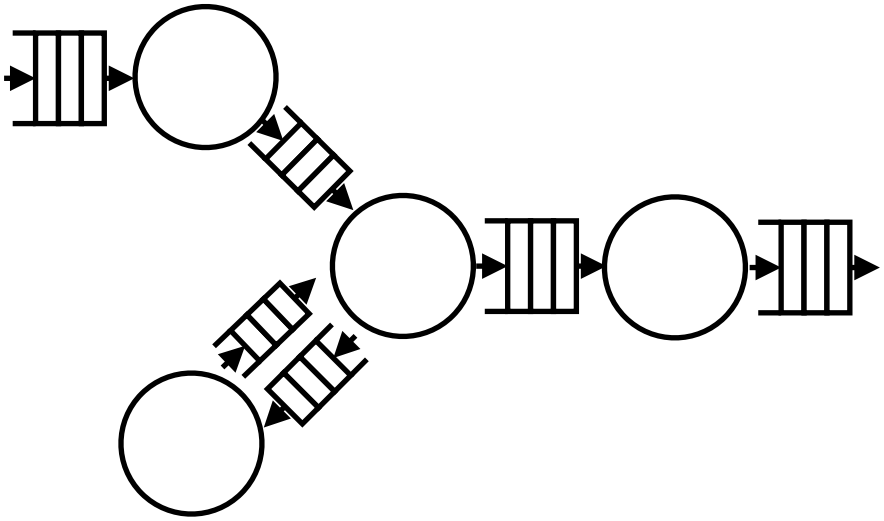
- Fan-in of number of valid signals grows with number of stages
- Fan-out of each stage's valid signal grows with number of stages
- Longer combinational paths as number of pipeline stages grows

Solution:
Break into Units and Decouple with FIFOs



- Pipeline only cares whether space in FIFO, not about whether consumer can take next value
  - Breaks combinational path between pipeline control logic and consumer control logic
- For full throughput with decoupling, need at least two elements in FIFO
  - With only one element, have to ping-pong between pipeline enqueue and consumer dequeue
  - Allowing both enqueue and dequeue in same cycle to single-element FIFO destroys decoupling (back to a synchronous connection)

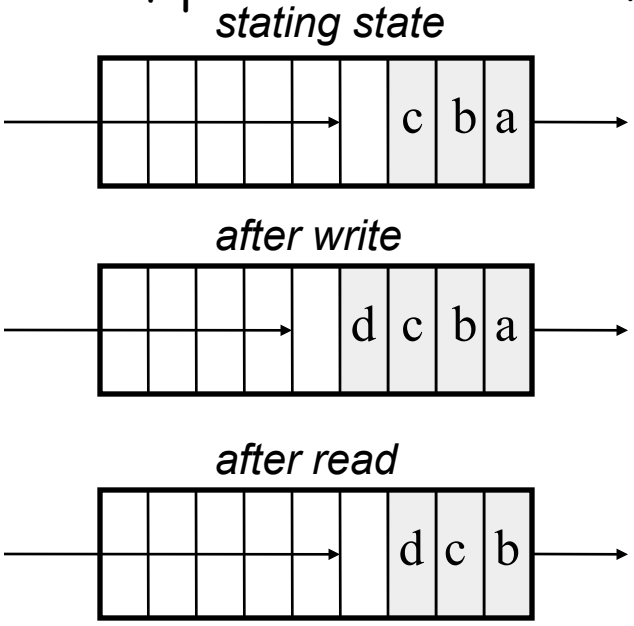
Decoupled Design Discipline



- Many large digital designs are divided into local synchronous pipelines, or units, connected via decoupling FIFOs
  - Approx. 10K-100K gates per unit
- Decoupled units may have different clocks (GALS)
  - In which case, need asynchronous FIFOs

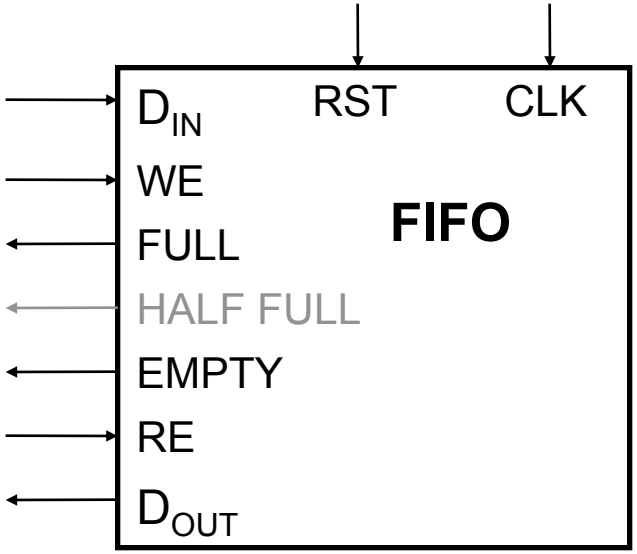
First-in-first-out (FIFO) Memory

- Used to implement *queues*.
- These find common use in computers and communication circuits.
- Generally, used to "decouple" actions of producer and consumer:



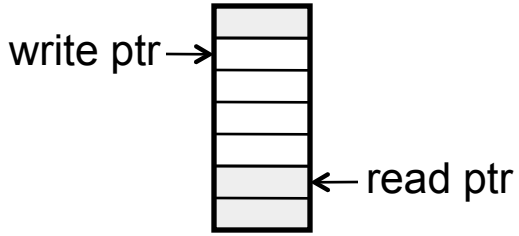
- Producer can perform many writes without consumer performing any reads (or vis versa). However, because of finite buffer size, on average, need equal number of reads and writes.
- Typical uses:
  - interfacing I/O devices. Example network interface. Data bursts from network, then processor bursts to memory buffer (or reads one word at a time from interface). Operations not synchronized.
  - Example: Audio output. Processor produces output samples in bursts (during process swap-in time). Audio DAC clocks it out at constant sample rate.

FIFO Interfaces

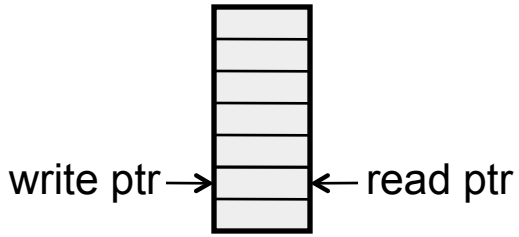


- After write or read operation, FULL and EMPTY indicate status of buffer.
- Used by external logic to control own reading from or writing to the buffer.
- FIFO resets to EMPTY state.
- HALF FULL (or other indicator of partial fullness) is optional.

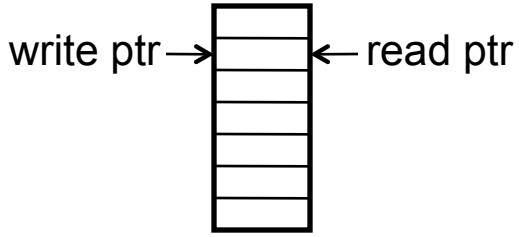
- Address pointers are used internally to keep next write position and next read position into a dual-port memory.



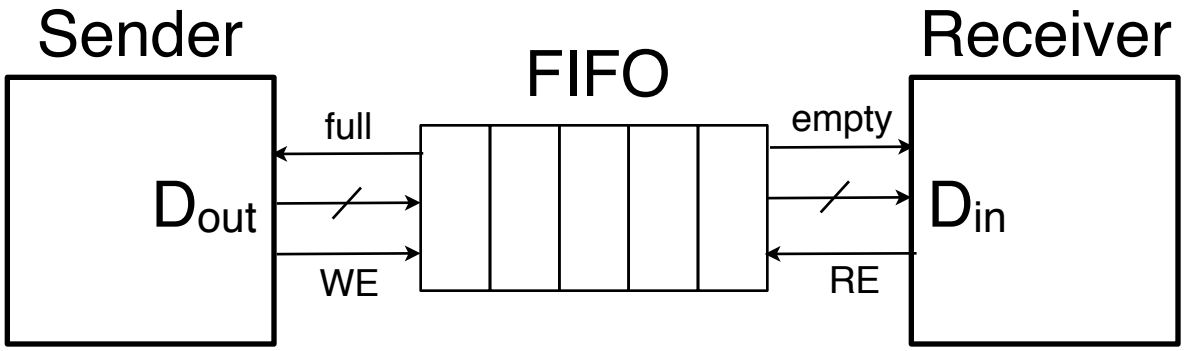
- If pointers equal after write ⇒ FULL:



- If pointers equal after read ⇒ EMPTY:



Decoupled Communication



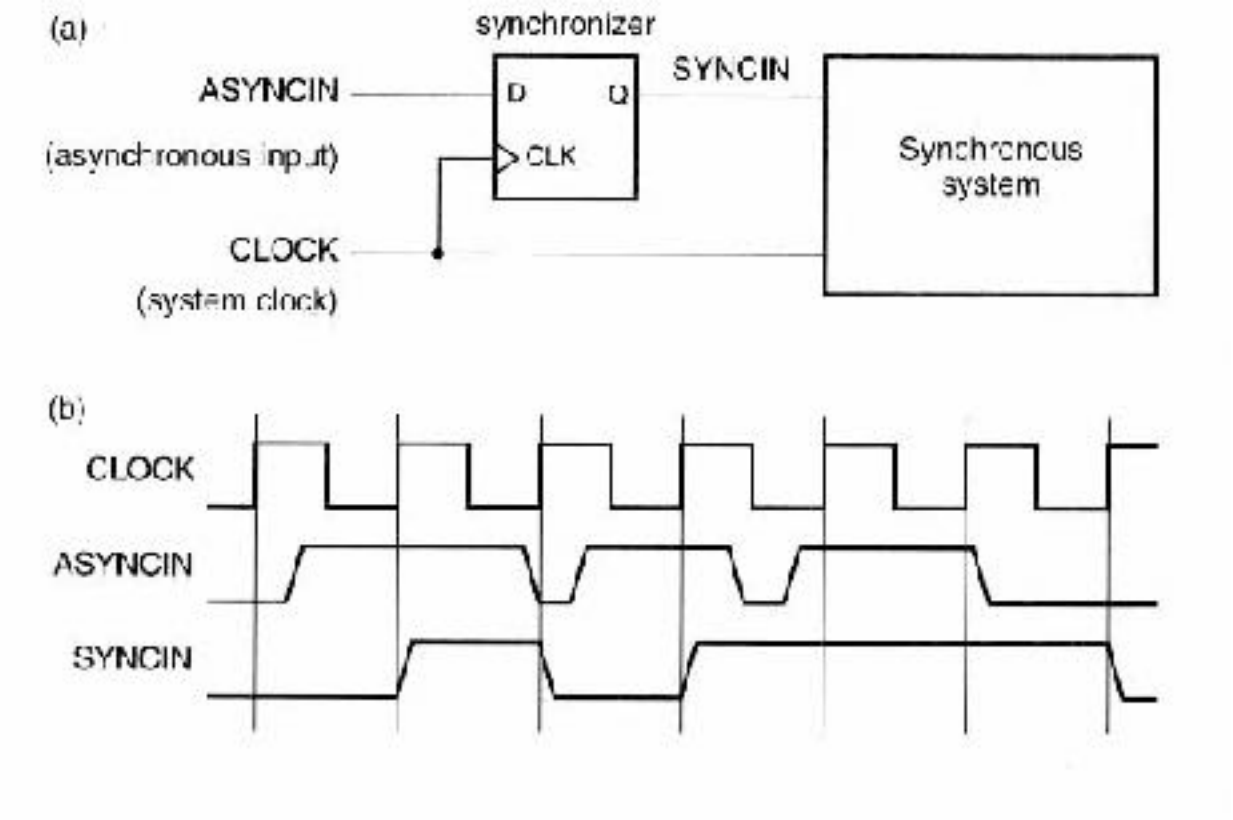
- Allow sender and receiver to transfer data independently
- Assumes, on average, equal number of sends and receives
- FIFO buffer must be large enough to accommodate instantaneous difference in send and receive rate
- Remaining Complication:
  - Latency in the feedback signal to sender ("full" in this case) or in Sender's ability to stop producing data can lead to receive buffer overrun!
  - Solutions:
    - Size buffers to accommodate worse case delay and send control signal early => "Skid buffering", or
    - "Credit-based" flow control

Communicating Across Clock Boundaries

- Many synchronous systems need to interface to asynchronous input signals:
  - Consider a computer system running at some clock frequency, say 1GHz with:
    - Interrupts from I/O devices, keystrokes, etc.
    - Data transfers from devices with their own clocks
      - Ethernet has its own 100MHz clock
      - PCI bus transfers, 66MHz standard clock.
  - These signals could have no known timing relationship with the system clock of the CPU.

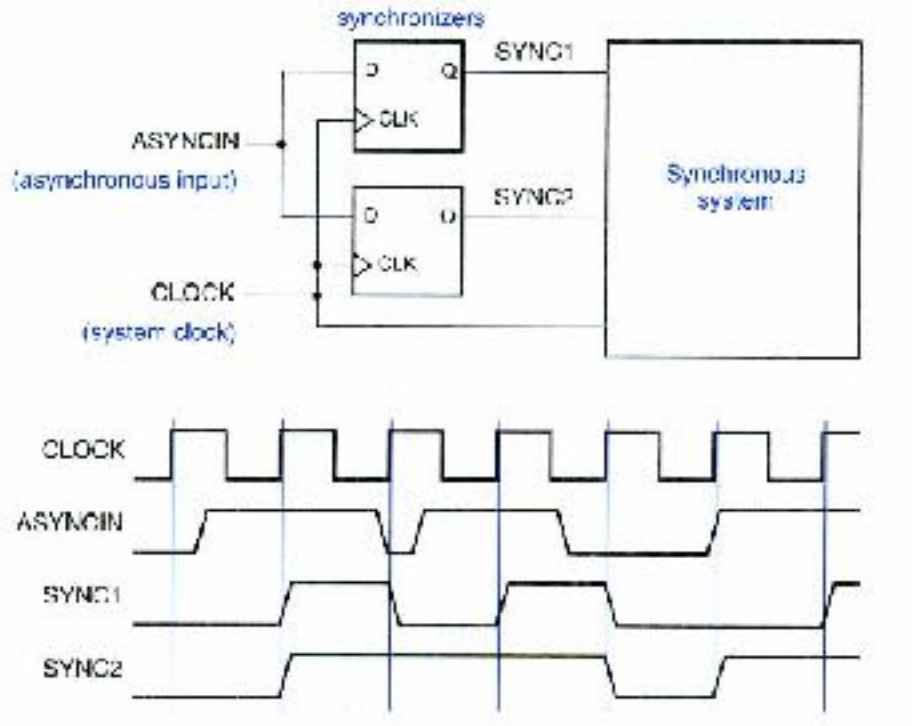
"Synchronizer" Circuit

- For a single asynchronous input, we use a simple flip-flop to bring the external input signal into the timing domain of the system clock:



"Synchronizer" Circuit

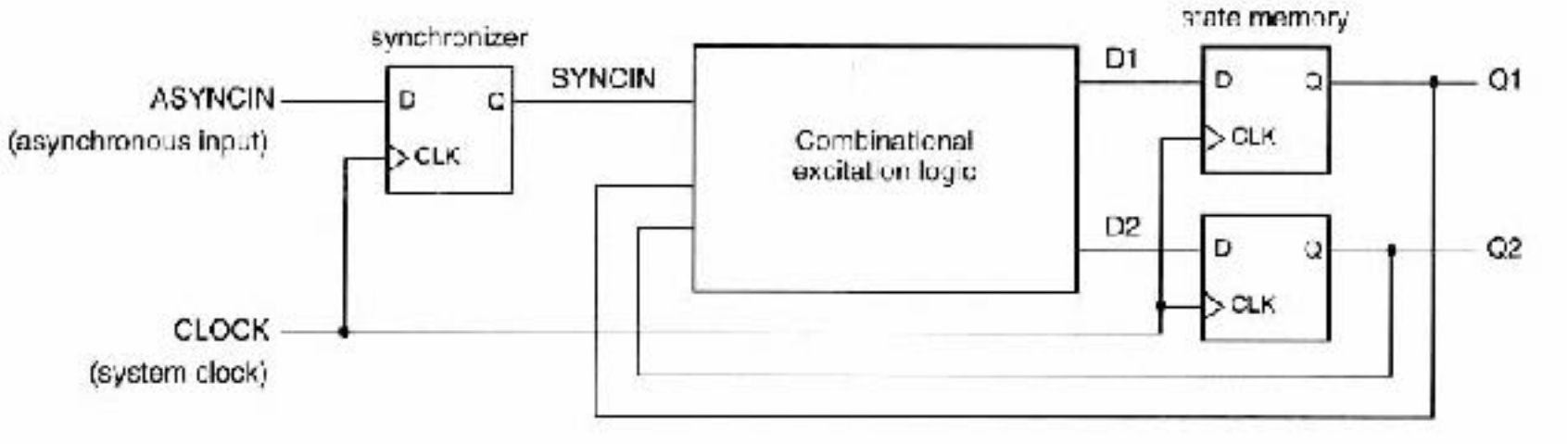
- It is essential for asynchronous inputs to be synchronized at only one place.



- Two flip-flops may not receive the clock and input signals at precisely the same time (clock *and* data skew).
- When the asynchronous changes near the clock edge, one flip-flop may sample input as 1 and the other as 0.

"Synchronizer" Circuit

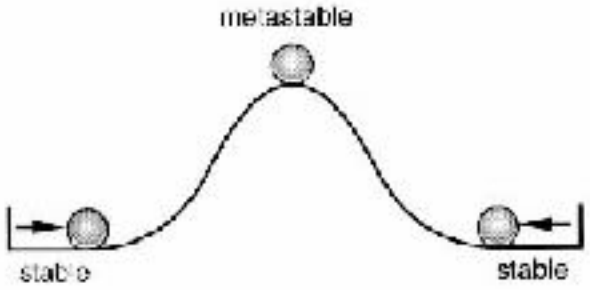
- Single point of synchronization is even more important when input goes to a combinational logic block (ex. FSM)
- The CL block can accidentally hide the fact that the signal is synchronized at multiple points.
- The CL magnifies the chance of the multiple points of synchronization seeing different values.

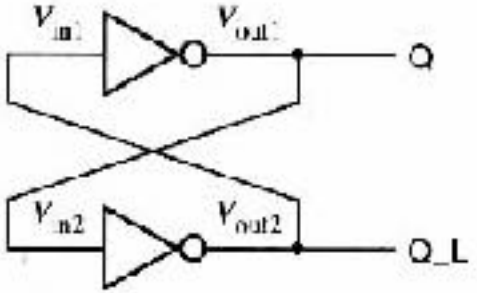


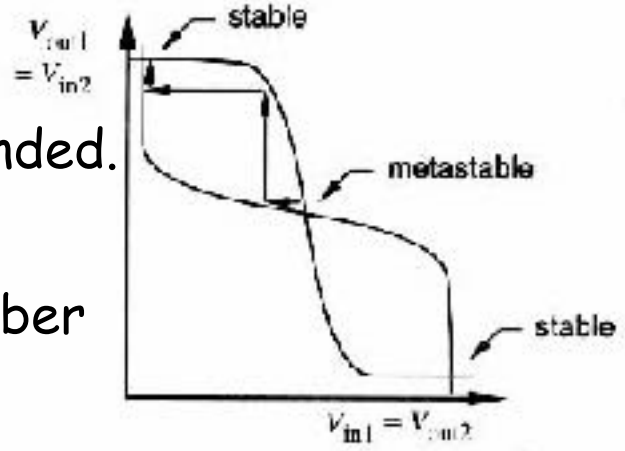
- Sounds simple, right?

Synchronizer Failure & Metastability

- We think of flip-flops having only two stable states - but all have a third *metastable* state halfway between 0 and 1.
- When the setup and hold times of a flip-flop are not met, the flip-flop could be put into the metastable state.
- Noise will be amplified and push the flip-flop one way or other.
- However, in theory, the time to transition to a legal state is unbounded.
- Does this really happen?
- The probability is low, but the number of trials is high!







Transfer function:

$V_{out1} = T(V_{in1})$

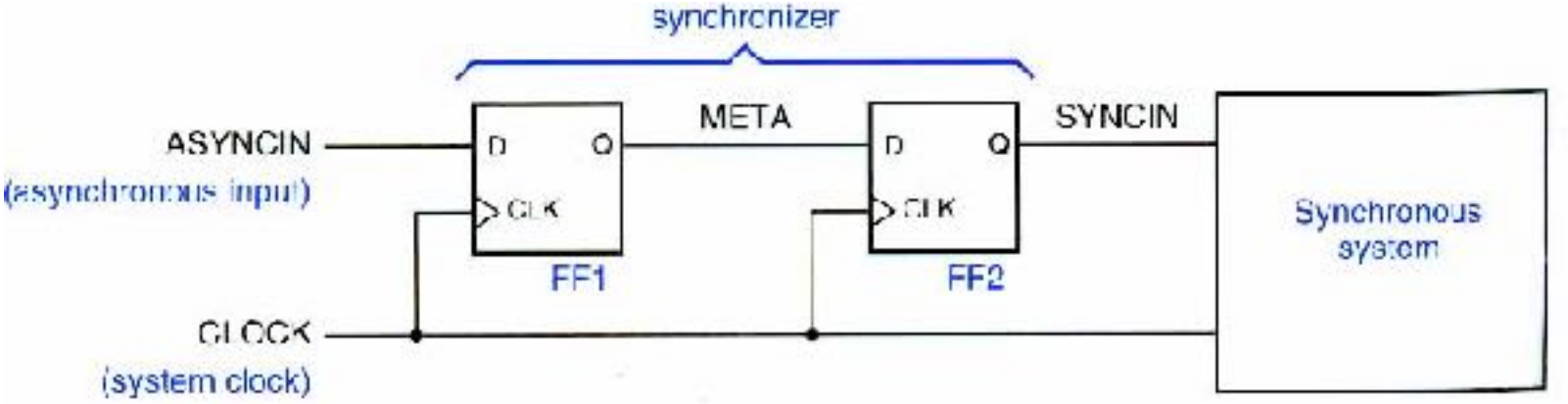
$V_{out2} = T(V_{in2})$

Synchronizer Failure & Metastability

- If the system uses a synchronizer output while the output is still in the metastable state ⇒ synchronizer failure.
- Initial versions of several commercial ICs have suffered from metastability problems - effectively synchronization failure:
  - AMD9513 system timing controller
  - AMD9519 interrupt controller
  - Zilog Z-80 Serial I/O interface
  - Intel 8048 microprocessor
  - AMD 29000 microprocessor
- To avoid synchronizer failure wait long enough before using a synchronizer's output. *"Long enough", according to Wakerly, is so that the mean time between synchronizer failures is several orders of magnitude longer than the designer's expected length of employment!*
- In practice all we can do is reduce the probability of failure to a vanishing small value.

Reliable Synchronizer Design

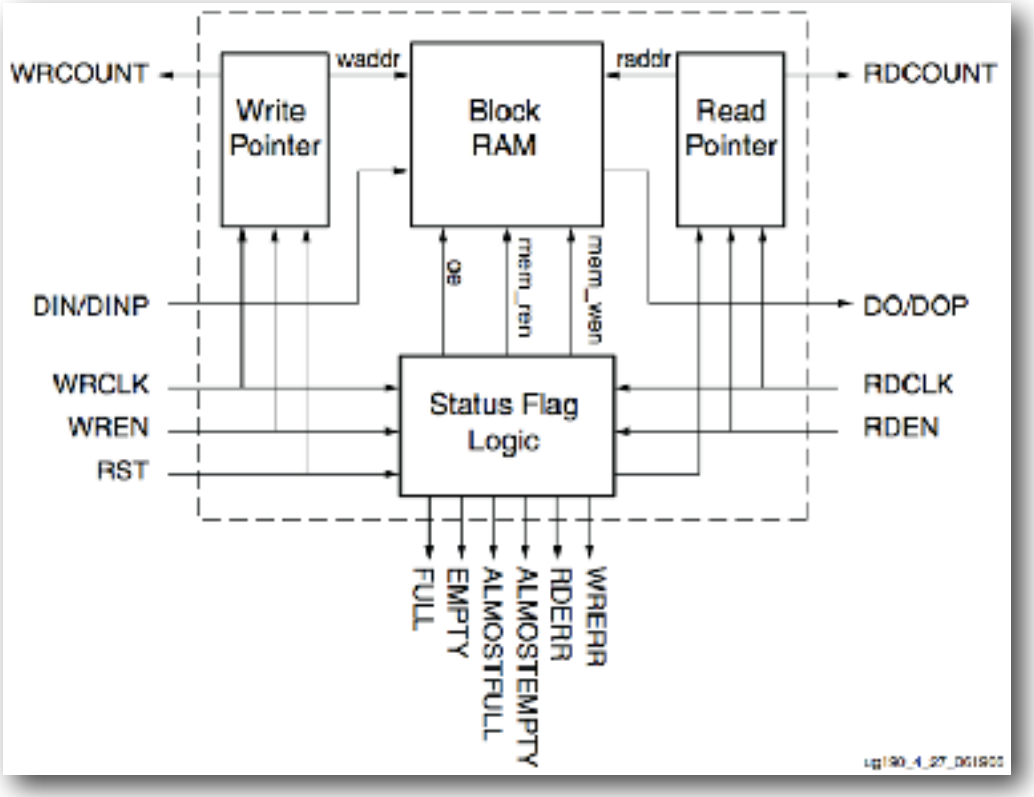
- **The probability that a flip-flop stays in the metastable state decreases exponentially with time.**
- Therefore, any scheme that delays using the signal can be used to decrease the probability of failure.
- In practice, delaying the signal by a cycle is usually sufficient:



- If the clock period is greater than metastability resolution time plus FF2 setup time, FF2 gets a synchronized version of ASYNCIN.
- Multi-cycle synchronizers (using counters or more cascaded flip-flops) are even better – but often overkill.

Xilinx FPGA FIFOs

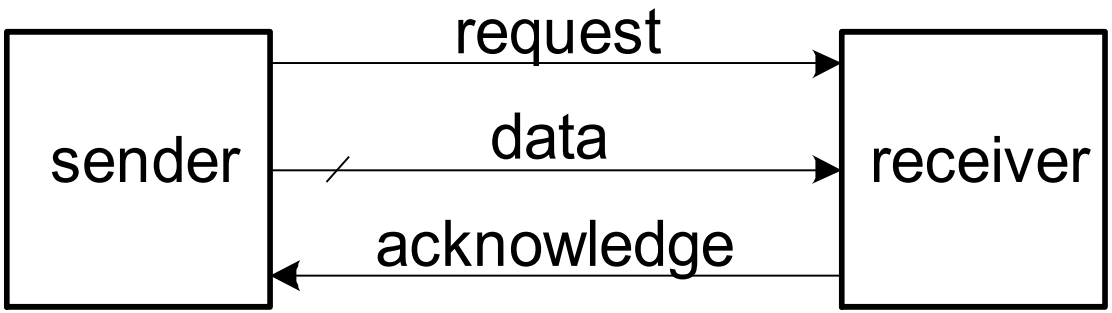
- BlockRAMS include dedicated circuits for FIFOs (details in User Guide (ug190)).
- Takes advantage of separate dual ports and **independent ports clocks**.



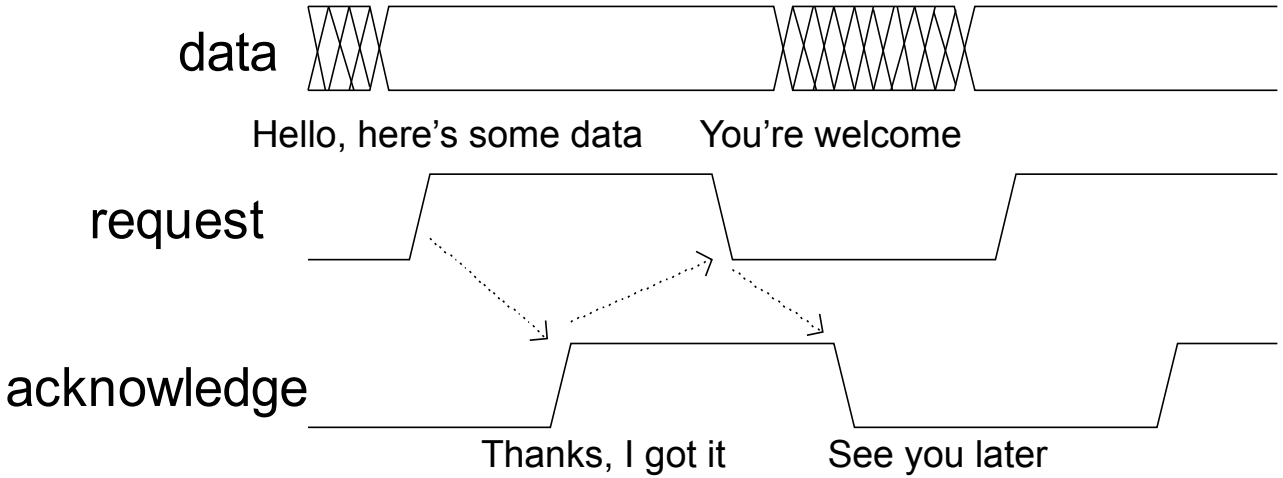
Asynchronous Circuits

- Many researchers (and a few industrial designers) have proposed a variety of circuit design methodologies that **eliminate the need for a globally distributed clock**.
- They cite a variety of important potential advantages over synchronous systems.
- To date, these attempts have remained mainly in Universities.
- A few commercial asynchronous chips/systems have been build.
- Sometimes, asynchronous blocks sometimes appear inside otherwise synchronous systems.
  - Asynchronous techniques have long been employed in DRAM and other memory chips for generation internal control without external clocks. (Precharge/ sense-amplifier timing based on address line changes.
- In GALS (globally asynchronous locally synchronous) systems, independently synchronous islands communicate asynchronously.

Delay Insensitive (self-timed transfer)



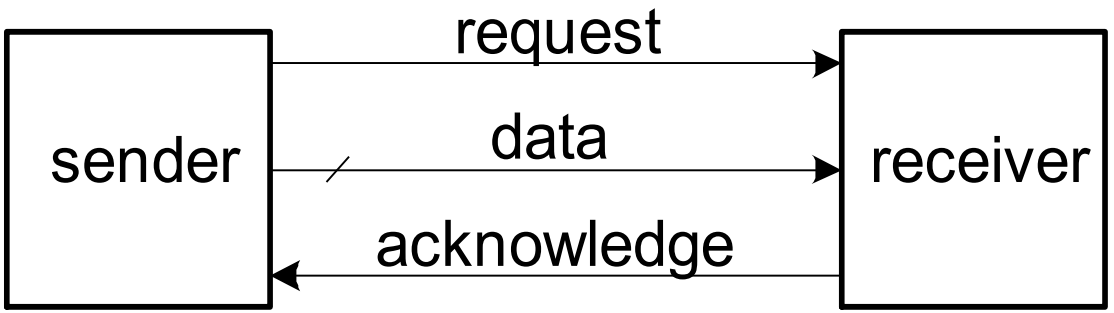
- Request/acknowledge “handshake” signal pair used to coordinate data transfer.



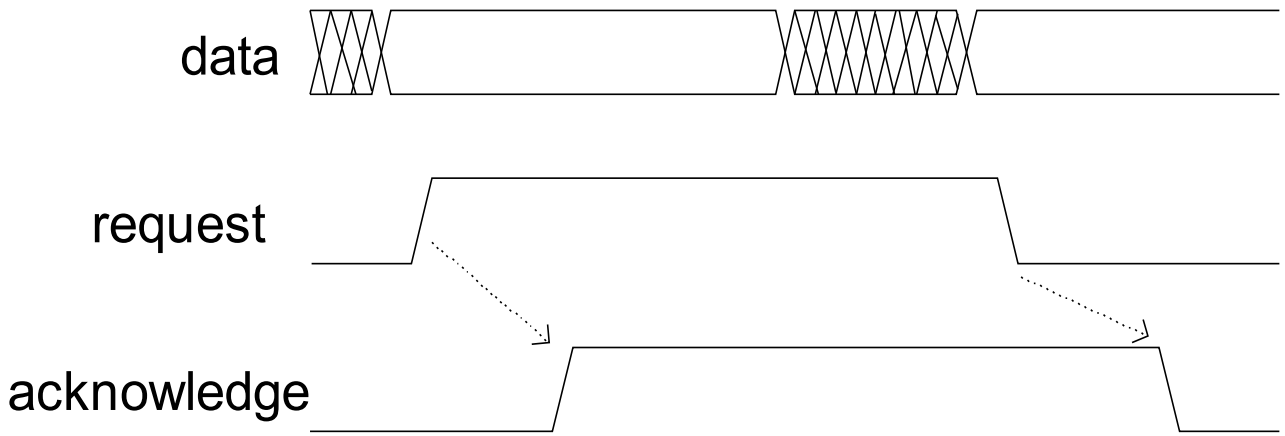
4-cycle (“return-to-zero”) signaling

- Note, transfer is insensitive to any delay in sending and receiving.

Delay Insensitive (self-timed transfer)



2-cycle ("non-return-to-zero") signaling



- Only two transitions per transfer. Maybe higher performance.
- More complex logic. 4-cycle return to zero can usually be overlapped with other operations.