## Lecture 5:

# Frankencamera discussion + Specialized Image processing HW

**Visual Computing Systems**
**Stanford CS348V, Winter 2018**

# A Discussion of F-Cam

# (last night's reading)

## [Adams 2010]

# Key aspect in the design of any system:

# Choosing the "right" representations for the job

# Frankencamera: some 2010 context

- Cameras: becoming increasingly cheap and ubiquitous
- Significant processing capability available on cameras
- Many techniques for combining multiple photos to overcome deficiencies of traditional camera systems

# Multi-shot photography example: high dynamic range (HDR) images

Source photographs: each photograph has different exposure

Tone mapped HDR image

Credit: Debevec and Malik

# More multi-shot photography examples

**“Lucky” imaging**

**Take several photos in rapid succession: likely to find one without camera shake**

no-flash flash result

**Flash-no-flash photography [Eisemann and Durand]**
**(use flash image for sharp, colored image, infer room lighting from no-flash image)**

# Frankencamera: some 2010 context

- Cameras are cheap and ubiquitous
- Significant processing capability available on cameras
- Many emerging techniques for combining multiple photos to overcome deficiencies in traditional camera systems

- Problem: the ability to implement multi-shot techniques on cameras was limited by camera system programming abstractions
  - Programmable interface to camera was very basic
  - Influenced by physical button interface to a point-and-shoot camera:
    - `take_photograph(parameters, output_jpg_buffer)`
  - Result: on most implementations, latency between two photos was high, mitigating utility of multi-shot techniques (large scene movement, camera shake, between shots)

# Frankencamera goals

[Adams et al. 2010]

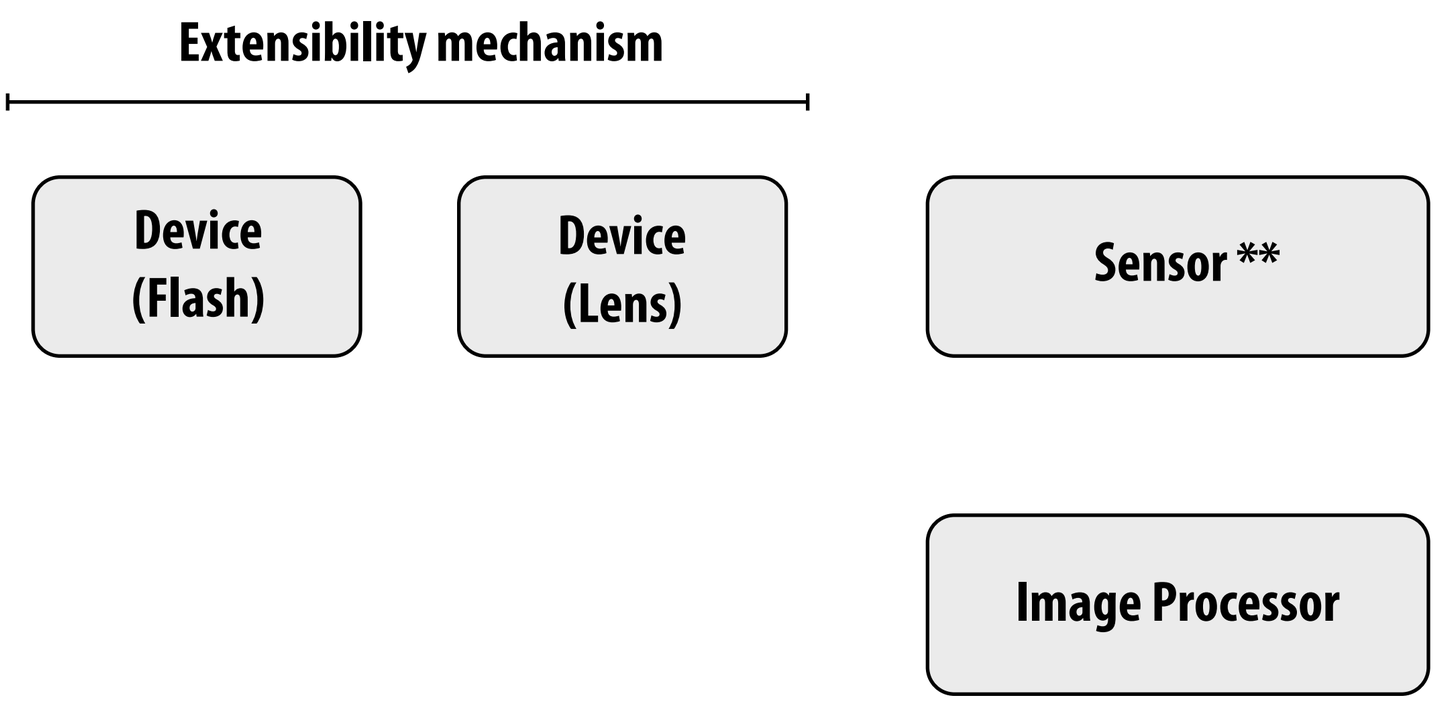
1. Create open, handheld computational camera platform for researchers
2. Define system architecture for computational photography applications
   - Motivated by impact of OpenGL on graphics application and graphics hardware development (portable apps despite highly optimized GPU implementations)
   - Motivated by proliferation of smart-phone apps

F2 Reference Implementation

Note: Apple was not involved in Frankencamera's industrial design. ;-)

Nokia N900 Smartphone Implementation

# F-cam components




** Sensor is really just a special case of a device

# What are F-Cam's key abstractions?

# Key concept: a shot

- A shot is a command
  - Actually it is a set of commands
  - Encapsulates both “set state” and “perform action(s)” commands
- Defines state (configuration) for:
  - Sensor
  - Image processor
  - Relevant devices
- Defines a timeline of actions
  - Exactly one sensor action: capture
  - Optional actions for devices
  - Note: timeline extends beyond length of exposure (“frame time”)

# Key concept: a shot

- Interesting analogy (for graphics people familiar with OpenGL)
  - An F-cam shot is similar to an OpenGL display list
  - A shot is really a series of commands (both action commands and state manipulation commands)
    - State manipulation commands specify the entire state of the system
    - But a shot defines precise timing of the commands in a shot (there is no OpenGL analogy for this)

# Key concept: a frame

- A frame describes the result of a shot

- A frame contains:
  - Reference to corresponding image buffer
  - Statistics for image (computed by image processor)
  - Shot configuration data (what was specified by application)
  - Actual configuration data (configuration actually used when acquiring image)
    - This actual may be different than shot configuration data

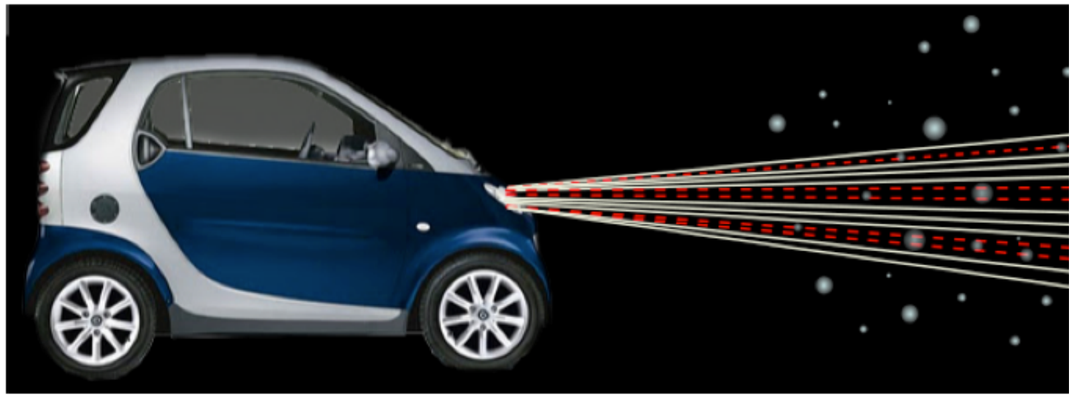
# Question

- F-cam tries to address a problem in conventional camera interface designs: was this a problem of throughput or latency?

# Aside: latency in modern camera systems

- Often in this class our focus will be on achieving high throughput
  - e.g., pixels per clock, images/sec, triangles/sec
- But low latency is critical in many visual computing domains
  - Camera metering, autofocus, etc.
  - Multi-shot photography
  - Optical flow, object tracking

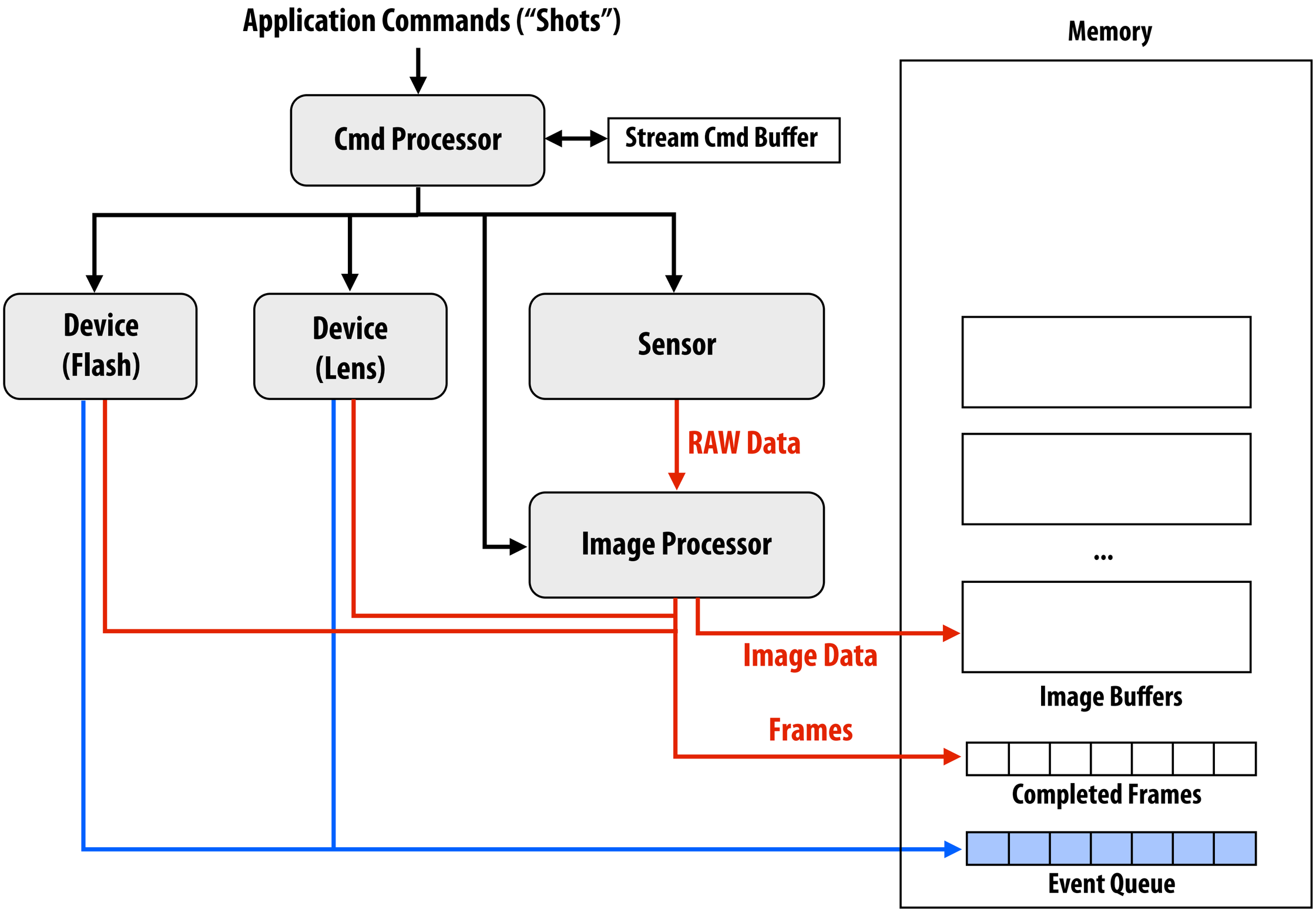
Extreme example:
CMU smart headlight project
[Tamburo et al. 2014]



# F-cam “streaming” mode

- System repeats shot (or series of shots) in infinite loop
- F-cam only stops acquiring frames when told to stop streaming by the application
- Example use case: “live view” (digital viewfinder) or continuous metering

# F-cam as an architecture



# F-cam scope

- F-cam provides a set of abstractions that allow for manipulating configurable camera components
  - Timeline-based specification of actions
  - Feed-forward system: no feedback loops

- F-cam architecture performs image processing, but…
  - This functionality as presented by the architecture is <u>not programmable</u>
  - Hence, F-cam does not provide an image processing language (it's like fixed-function OpenGL)
  - Other than work performed by the image processing stage, F-cam applications perform their own image processing (e.g., on smartphone/camera's CPU or GPU resources)

# Android Camera2 API

- Take a look at the documentation of the Android Camera2 API, and you'll see influence of F-Cam.

# Class design challenge 1

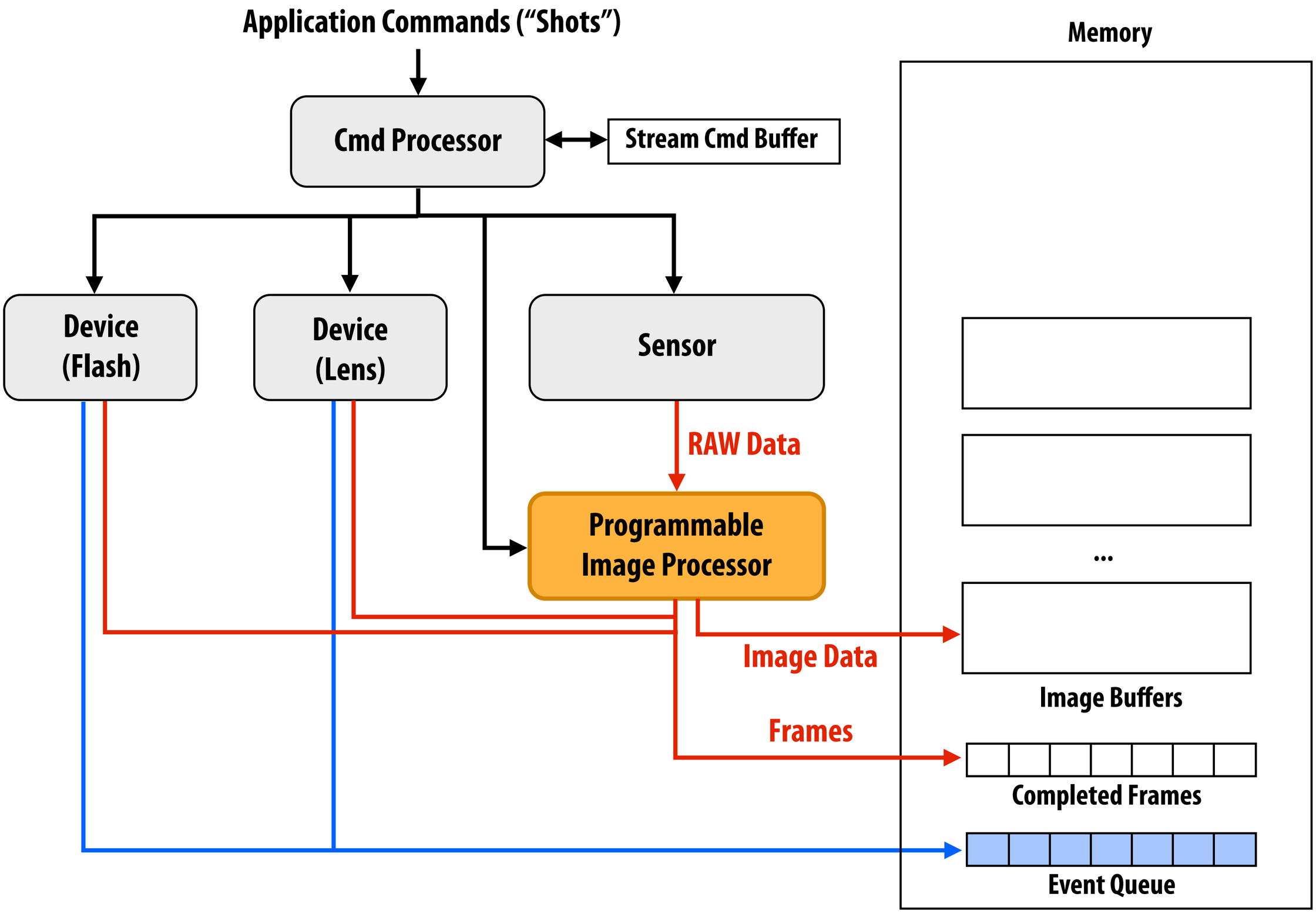
- Question: How is auto-focus expressed in F-cam?
  - Is autofocus part of F-cam?
  - Can you implement autofocus using F-cam?

- How might we extend the F-cam architecture to model a separate autofocus/metering sensor if the hardware platform contained them?

# Class design challenge 2

- Should we add a face-detection unit to the architecture?
- How might we abstract a face-detection unit?
- Or a SIFT feature extractor?

# Hypothetical F-cam extension: programmable image processing



# Class design challenge 3

- If there was a programmable image processor, application would probably seek to use it for more than just on data coming off sensor

- E.g., HDR imaging app

# Specialized Hardware for Image Processing

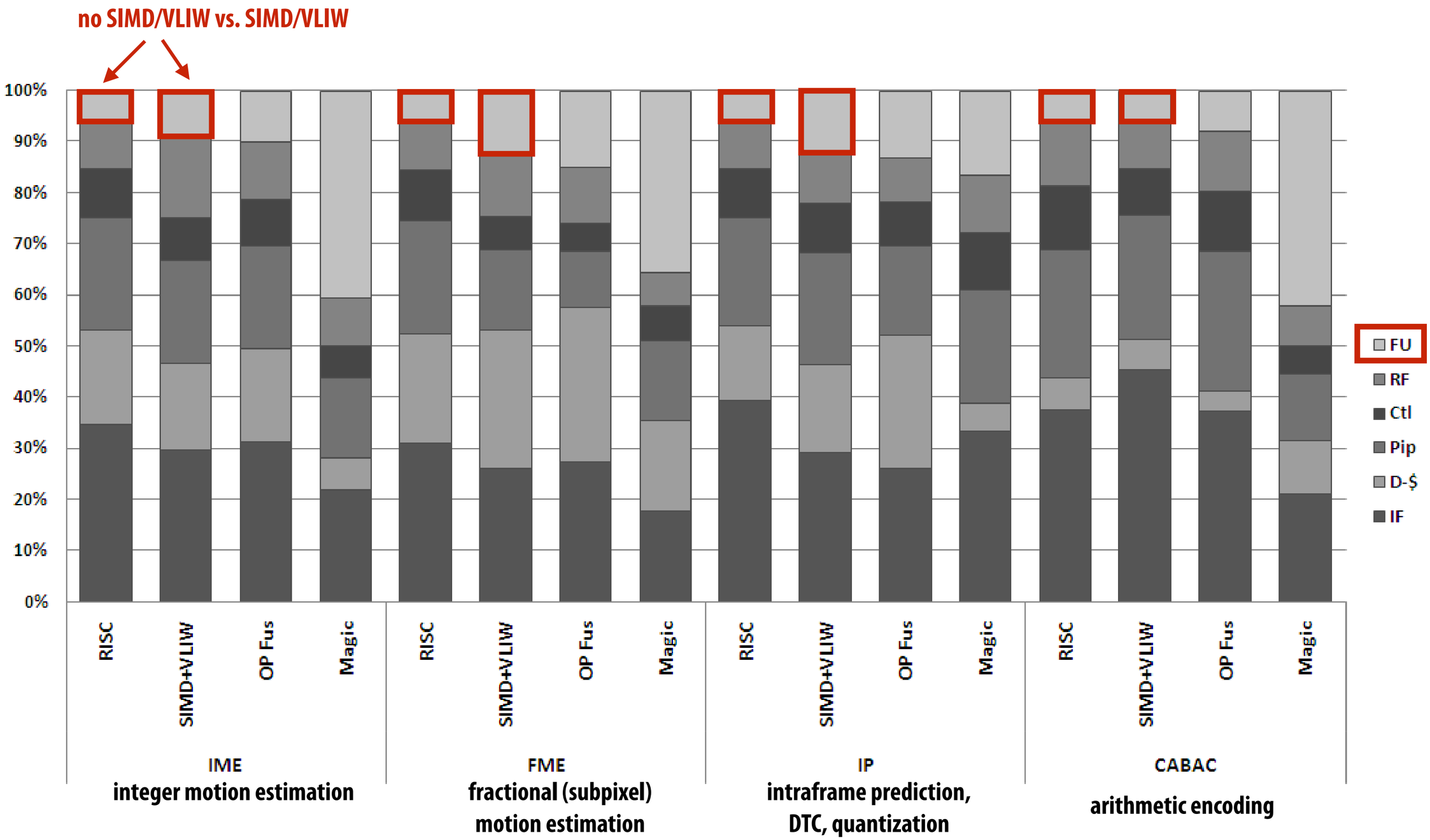
# So far, the discussion in this class has focused on generating efficient code for multi-core processors such as CPUs and GPUs.

# Consider the complexity of executing an instruction on a modern processor…

Read instruction ⟝ Address translation, communicate with icache, access icache, etc.
Decode instruction ⟝ Translate op to uops, access uop cache, etc.
Check for dependencies/pipeline hazards
Identify available execution resource
Use decoded operands to control register file SRAM (retrieve data)
Move data from register file to selected execution resource
Perform arithmetic operation
Move data from execution resource to register file
Use decoded operands to control write to register file SRAM

Question:
How does SIMD execution reduce overhead when executing certain types of computations?
What properties must these computations have?

# Fraction of energy consumed by different parts of instruction pipeline (H.264 video encoding) [Hameed et al. ISCA 2010]



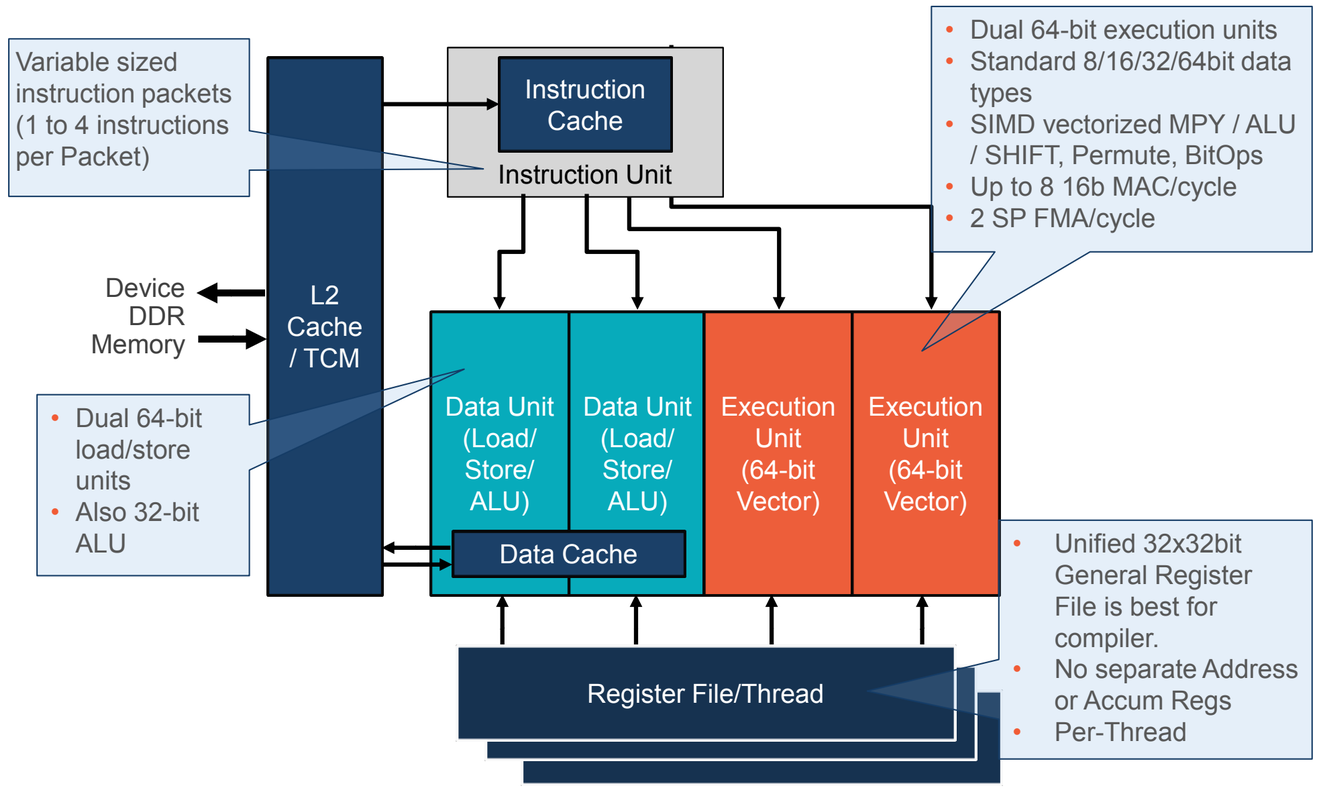

FU = functional units
RF = register fetch
Ctrl = misc pipeline control
Pip = pipeline registers (interstage)
D-$ = data cache
IF = instruction fetch + instruction cache

# DSPs: Qualcomm Hexagon

- Typically simpler instruction stream control paths
- Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction

**Example: Qualcomm Hexagon**
**Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors**

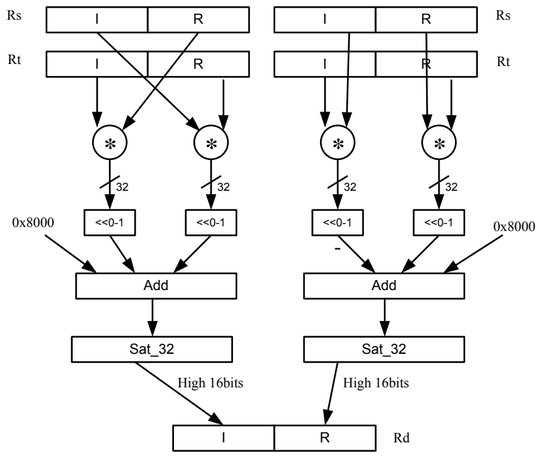
**Below: innermost loop of FFT**
**29 "RISC" ops per cycle**




64-bit Load and
64-bit Store with post-update addressing

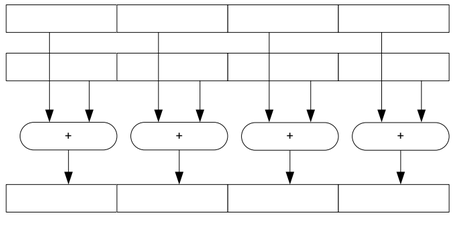
```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

Complex multiply with round and saturation



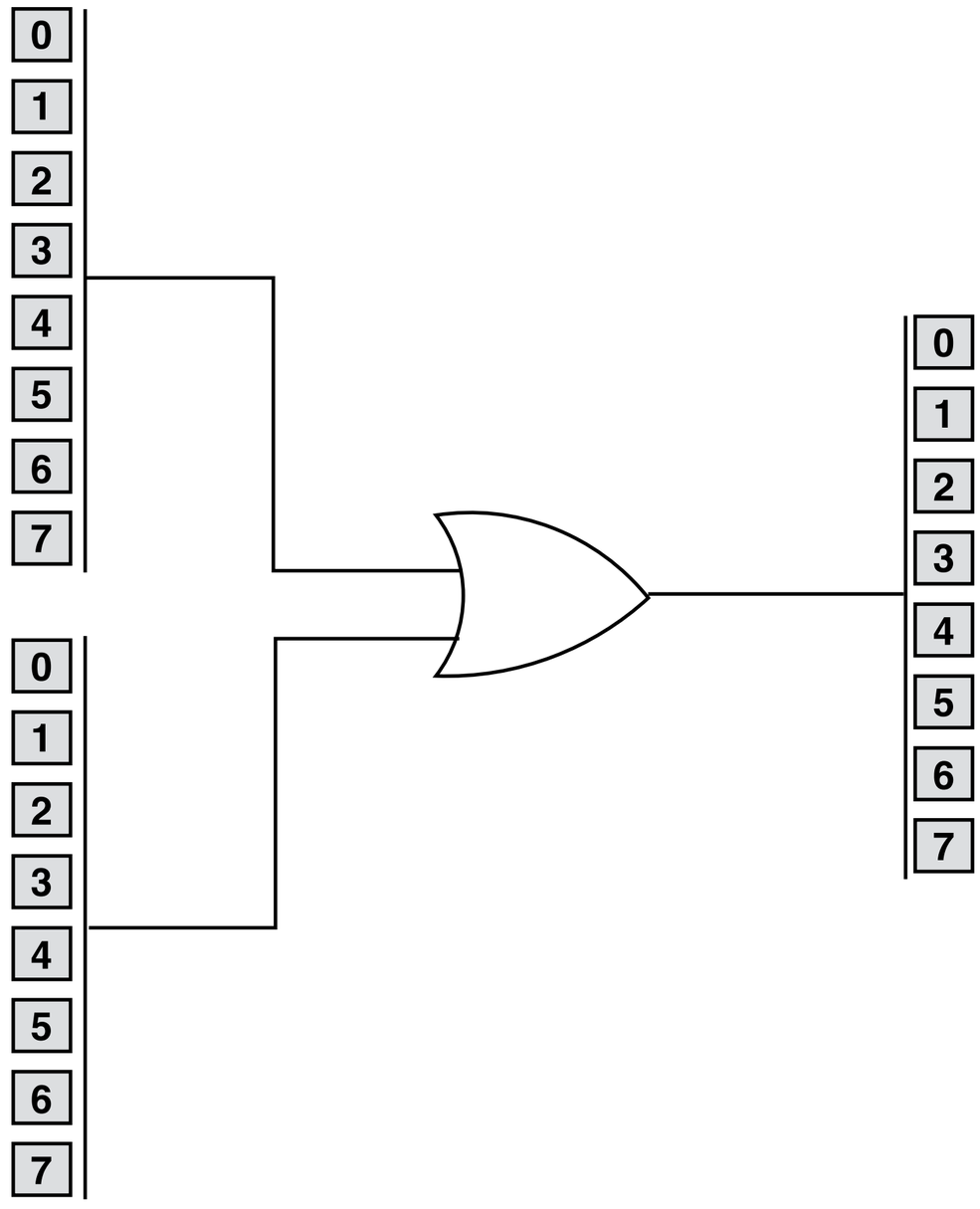

Zero-overhead loops
- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



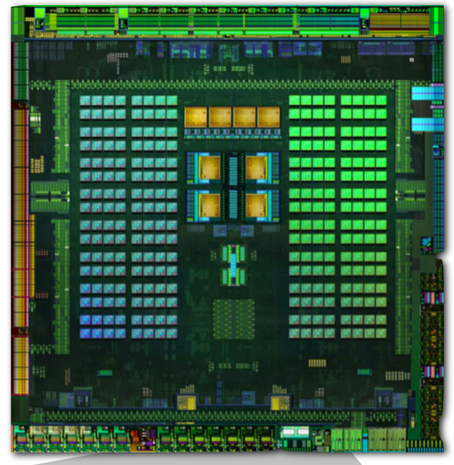
# Contrast to custom circuit to perform the operation

## Example: 8-bit logical OR



# Recall use of custom circuits in modern SoC

**Example: NVIDIA Tegra X1**

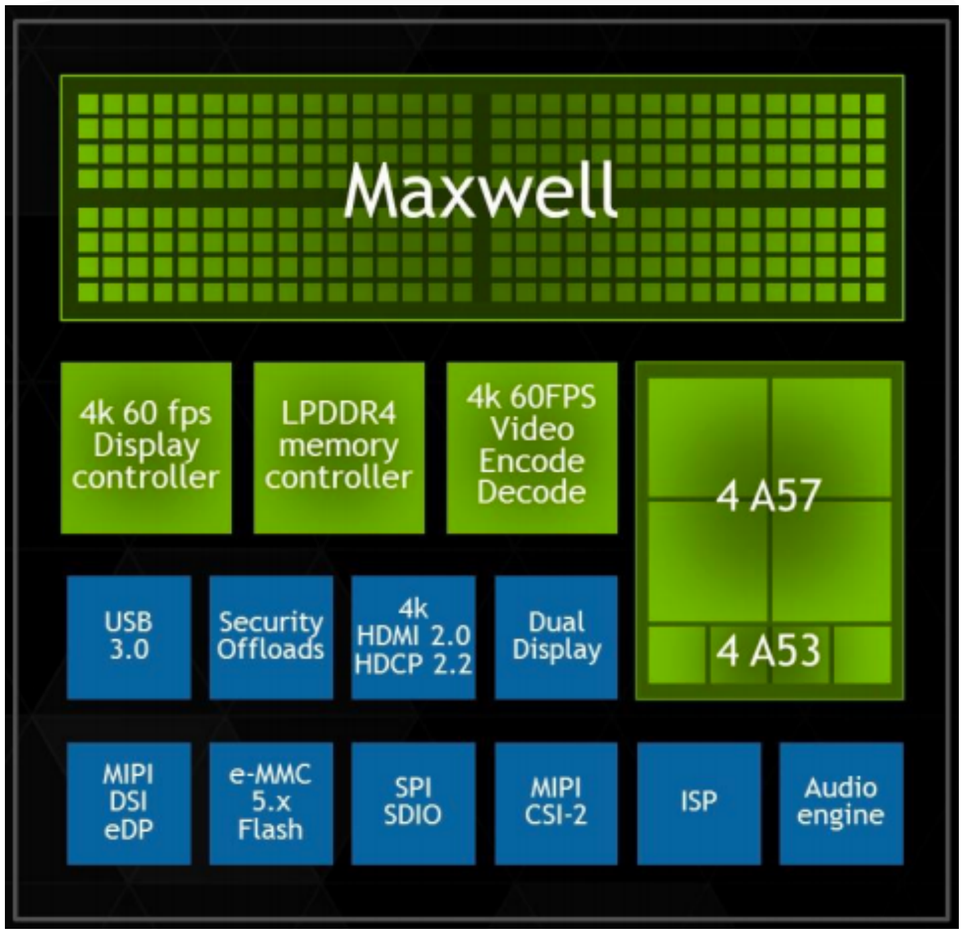


**Audio encode/decode**

**Video encode/decode**

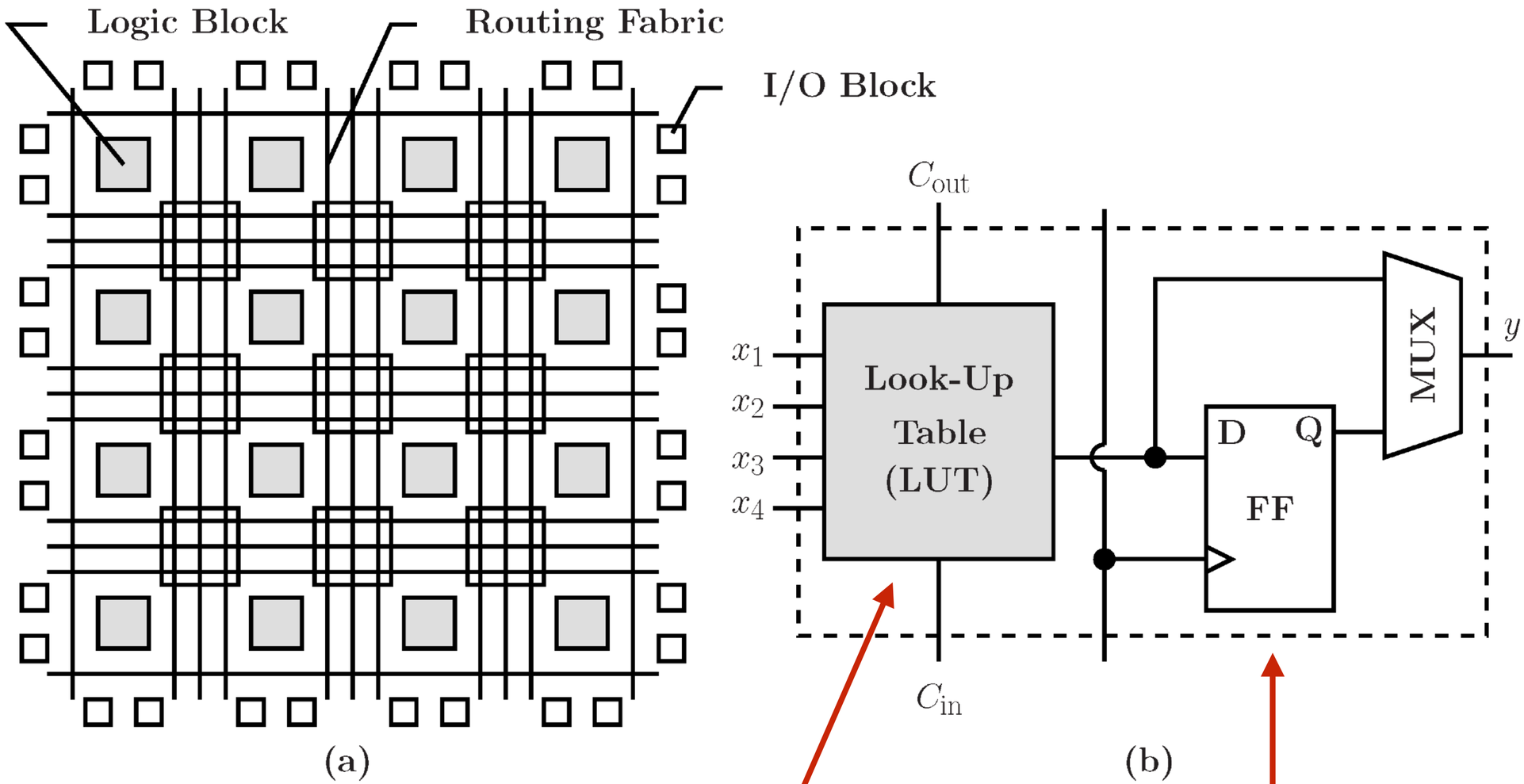
**High-frame rate camera RAW processing (ISP)**

**Data compression**



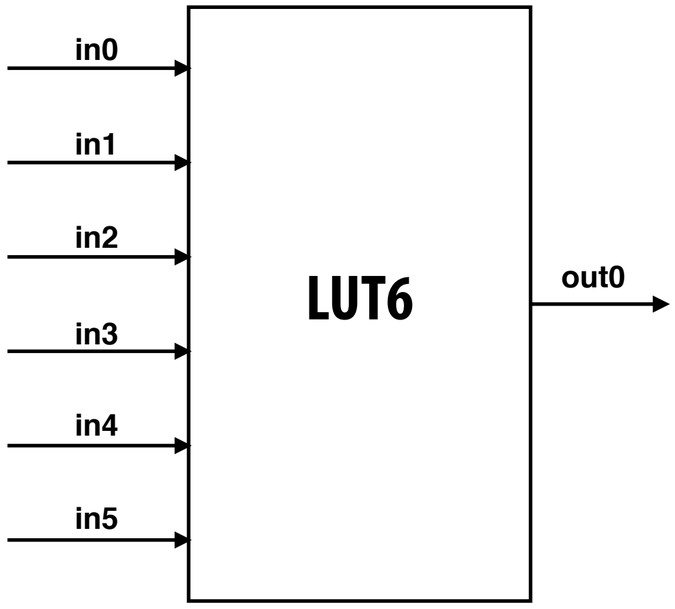
# FPGAs (field programmable gate arrays)

- FPGA chip provides array of logic blocks, connected by interconnect
- Programmer defines behavior of logic blocks via hardware description language (HDL) like Verilog or VHDL



# Specifying combinatorial logic via LUT

- Example: 6-input, 1 output LUT in Xilinx Virtex-7 FPGAs
  - Think of a LUT6 as a 64 element table




**Example: 6-input AND**

| In | Out |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| ⋮ | ⋮ |
| 63 | 1 |

**40-input AND constructed by chaining outputs of eight LUT6's (delay = 3)**

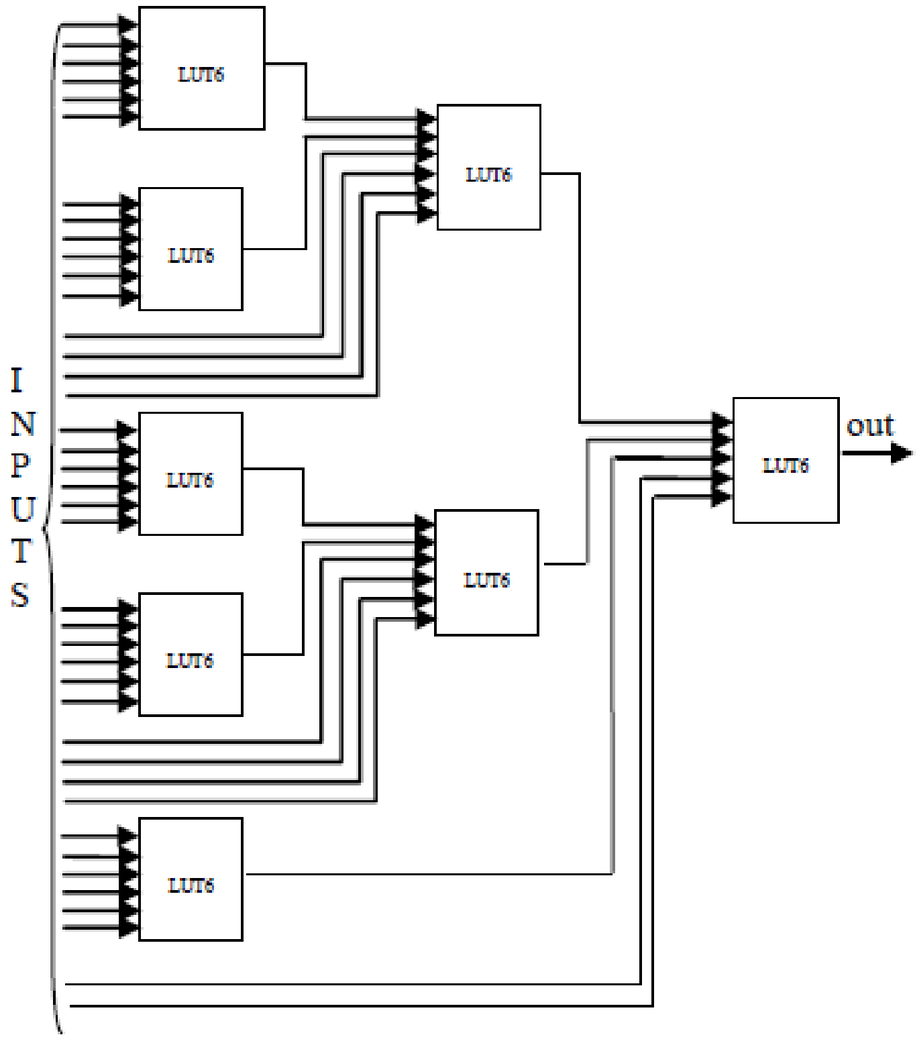



Image credit: [Zia 2013]

# Question

- What is the role of an ISA? (e.g., x86)

  Answer: interface between program definition (software) and hardware implementation

  Compilers produce sequence of instructions

  Hardware executes sequences of instructions as efficiently as possible

  (As shown earlier in lecture, many circuits used to implement/preserve this abstraction, not execute the computation needed by the program)

# New ways of defining hardware

- Verilog/VHDL present very low level programming abstractions for modeling circuits (RTL abstraction: register transfer level)
  - Combinatorial logic
  - Registers

- Due to need for greater efficiency, there is significant modern interest in making it easier to synthesize circuit-level designs
  - Skip the ISA, directly synthesize circuits needed to compute the tasks defined by a program.
  - Raise the level of abstraction of direct hardware programming

- Examples:
  - C to HDL (e.g., ROCCC, Vivado)
  - Bluespec
  - CoRAM [Chung 11]
  - Chisel [Bachrach 2012]

# Enter domain specific languages

# Compiling image processing pipelines directly to FPGAs

- Darkroom [Hegarty 2014]
- Rigel [Hegarty 2016]

- Motivation:
  - Convenience of high-level description of image processing algorithms (like Halide)
  - Energy-efficiency of FPGA implementations

    (particularly important for high-frame rate, low-latency, always on, embedded/robotics applications)

# Optimizing for minimal buffering

- Recall: scheduling Halide programs for CPUs/GPUs
  - Key challenge: organize computation so intermediate buffers fit in caches
- Scheduling for FPGAs:
  - Key challenge: minimize size of intermediate buffers (keep buffered data spatially close to combinatorial logic)

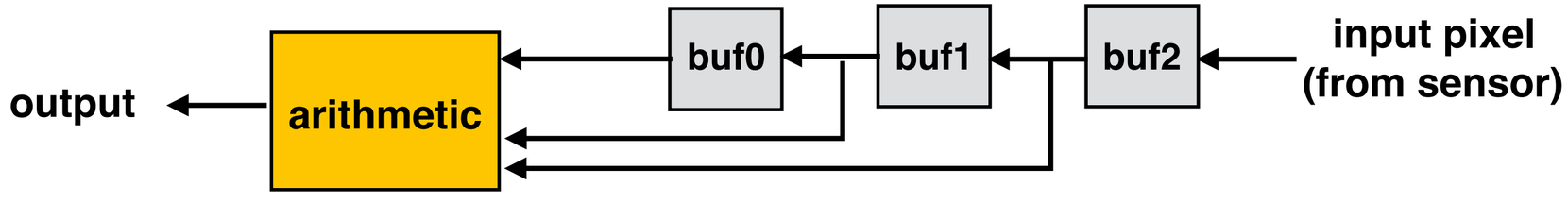
Consider 1D convolution:

```
out(x) =  (in(x-1) + in(x) + in(x+1)) / 3.0
```

Efficient hardware implementation: requires storage for 3 pixels in registers

```
out_pixel = (buf0 + buf1 + buf2) / 3
buf0 = buf1
buf1 = buf2
buf2 = in_pixel
```

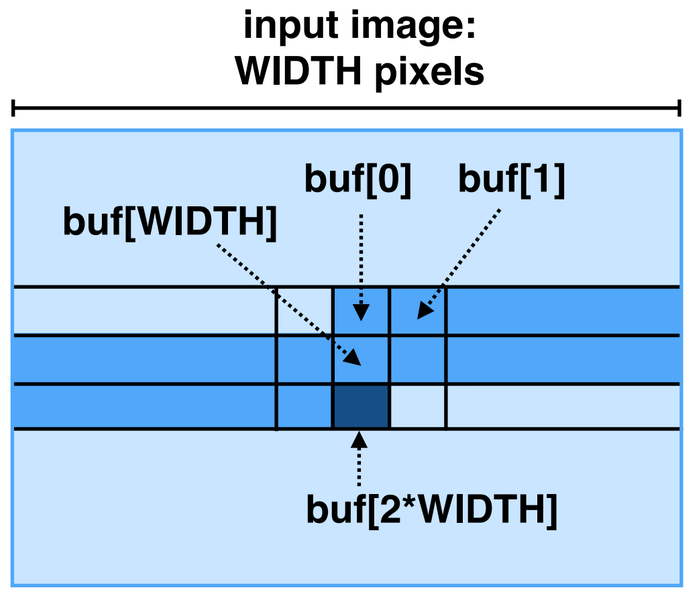
"Shift" new pixel in



# Line buffering

## Consider convolution of 2D image in vertical direction:

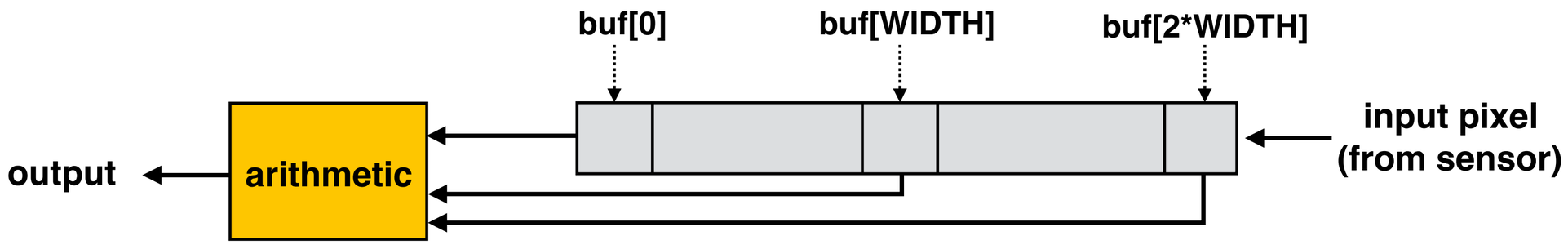
```
out(x,y) = (in(x,y-1) + in(x,y) + in(x,y+1)) / 3.0
```




## Efficient hardware implementation:

**let buf be a shift register containing 2*WIDTH+1 pixels (“line buffer”)**

```
// assume: no output until shift register fills
out_pixel = (buf[0] + buf[WIDTH] + buf[2*WIDTH]) / 3.0
shift(buf);  // buf[i] = buf[i+1]
buf[2*WIDTH] = in_pixel
```




## Note: despite notation, line buffer *is not* a random access SRAM, it is a shift register

# Halide to hardware

- Reinterpret common Halide scheduling primitives
  - unroll() —> replicate hardware
- Add new primitive accelerate()
  - Defines granularity of accelerated task
  - Defines throughput of accelerated task

```
Func unsharp(Func in) {
  Func gray, blurx, blury, sharpen, ratio, unsharp;
  Var x, y, c, xi, yi;

  // The algorithm
  gray(x, y) = 0.3*in(0, x, y) + 0.6*in(1, x, y) + 0.1*in(2, x, y);
  blury(x, y) = (gray(x, y-1) + gray(x, y) + gray(x, y+1)) / 3;
  blurx(x, y) = (blury(x-1, y) + blury(x, y) + blury(x+1, y)) / 3;
  sharpen(x, y) = 2 * gray(x, y) - blurx(x, y);
  ratio(x, y) = sharpen(x, y) / gray(x, y);
  unsharp(c, x, y) = ratio(x, y) * input(c, x, y);

  // The schedule
  unsharp.tile(x, y, xi, yi, 256, 256).unroll(c)
         .accelerate({in}, xi, x)
         .parallel(y).parallel(x);
  in.fifo_depth(unsharp, 512);
  gray.linebuffer().fifo_depth(ratio, 8);
  blury.linebuffer();
  ratio.linebuffer();

  return unsharp;
}
```

**Parallel units for rgb**

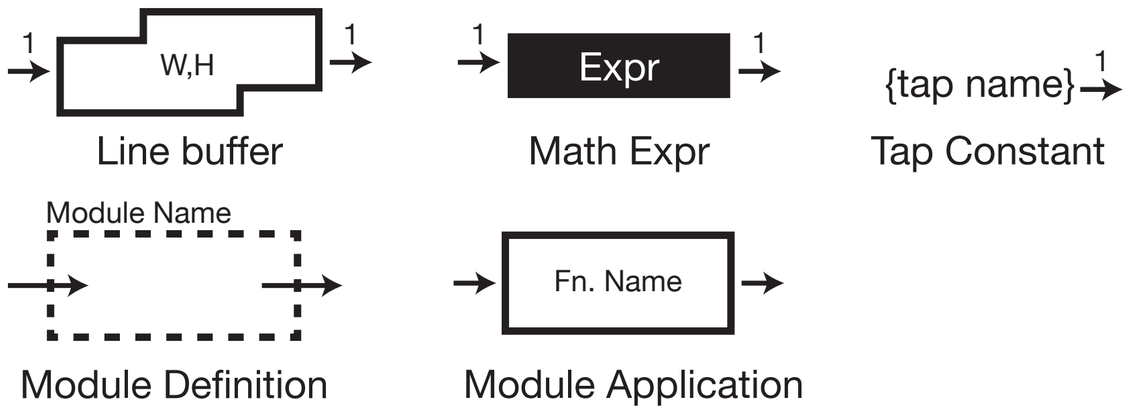
**Unit of work done per cycle: one pixel (one iteration of xi loop)**

**Unit of work given to accelerator: 256x256 tile (one iteration of x loop)**

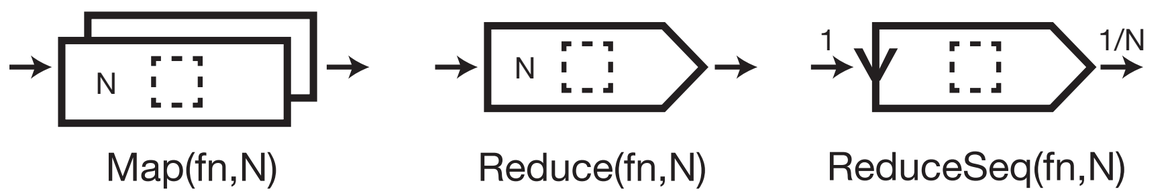
# Rigel

- **Provides set of well-defined building hardware blocks which can be assembled into image processing data flow graphs**
- **Provides programmer service of gluing modules in a dataflow graph together to form a complete implementation**

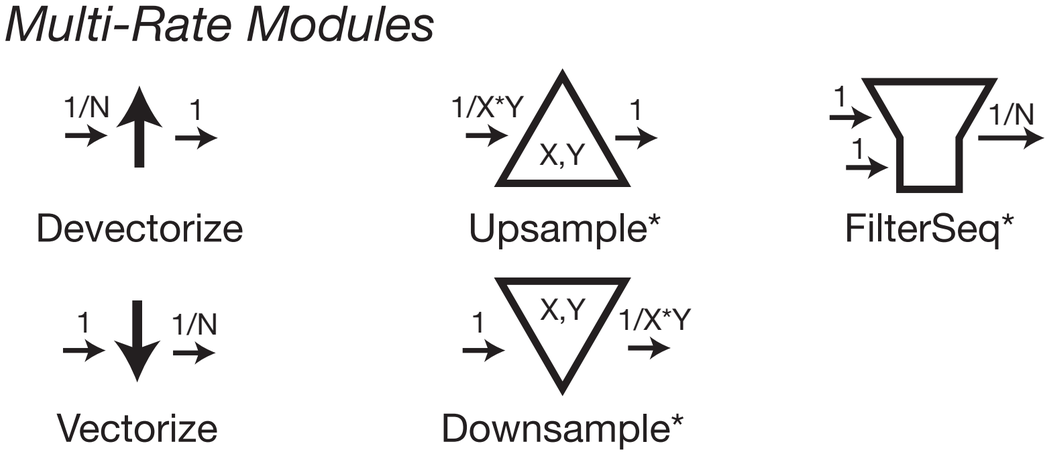
*Core Modules*




*Higher-Order Modules*




*Multi-Rate Modules*




## Example: 4x4 convolution in Rigel