Lecture 6:

Specializing Hardware for Image Processing

Visual Computing Systems
CMU 15-769, Fall 2016

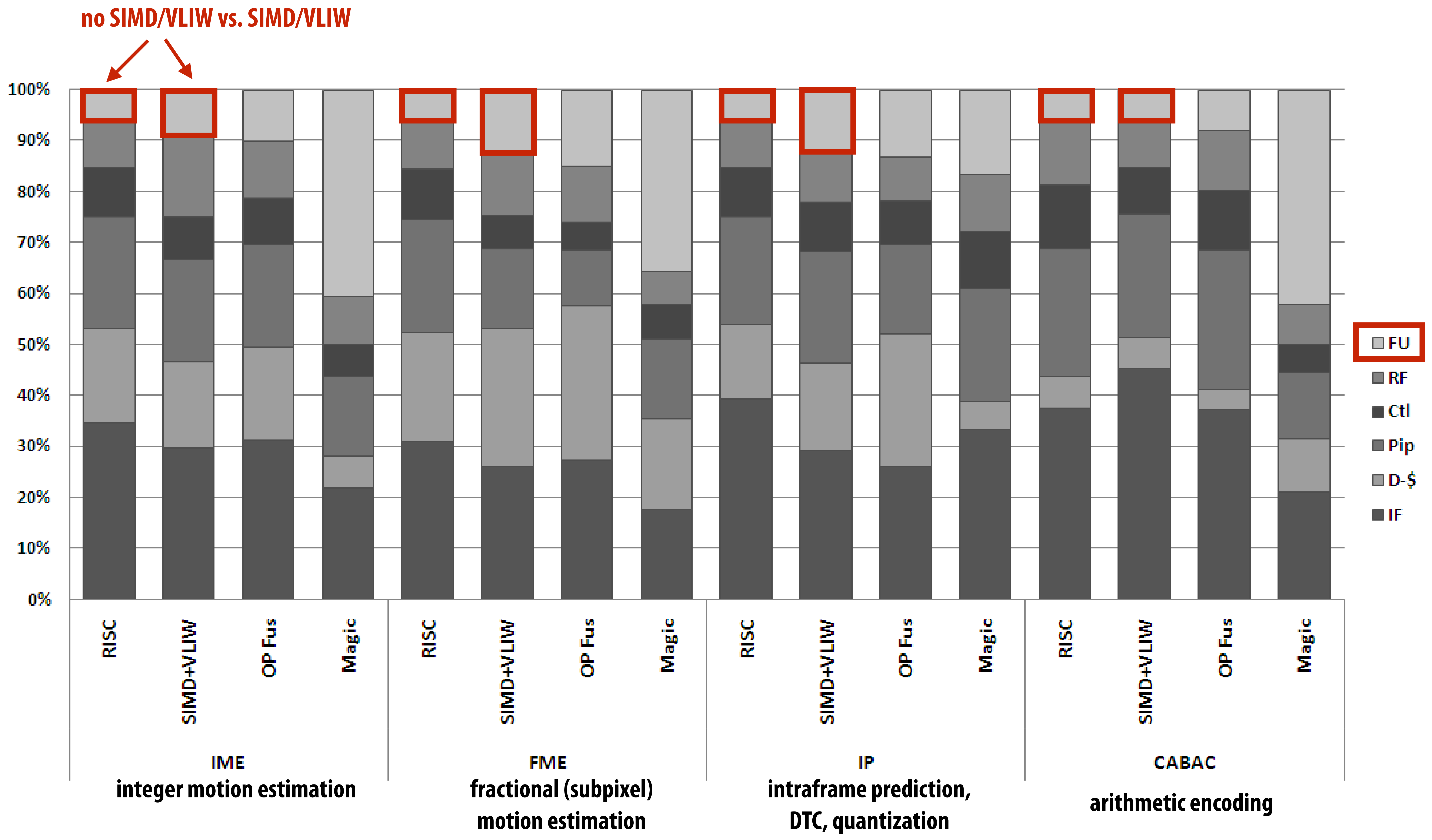
So far, the discussion in this class has focused on generating efficient code for multi-core processors such as CPUs and GPUs.

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]



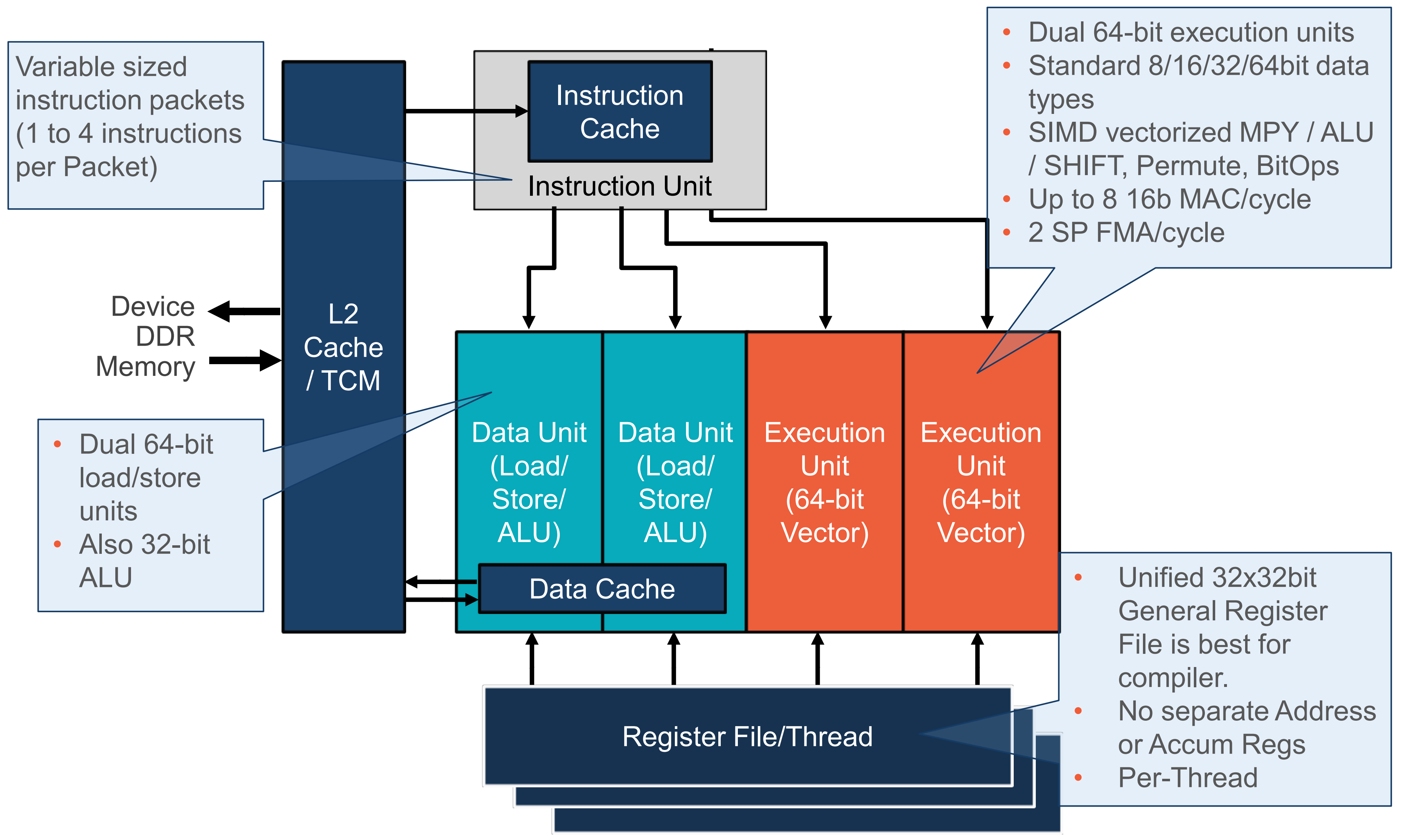
FU = functional units	Pip = pipeline registers (interstage)
RF = register fetch	D-$ = data cache
Ctrl = misc pipeline control	IF = instruction fetch + instruction cache

DSPs

- 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

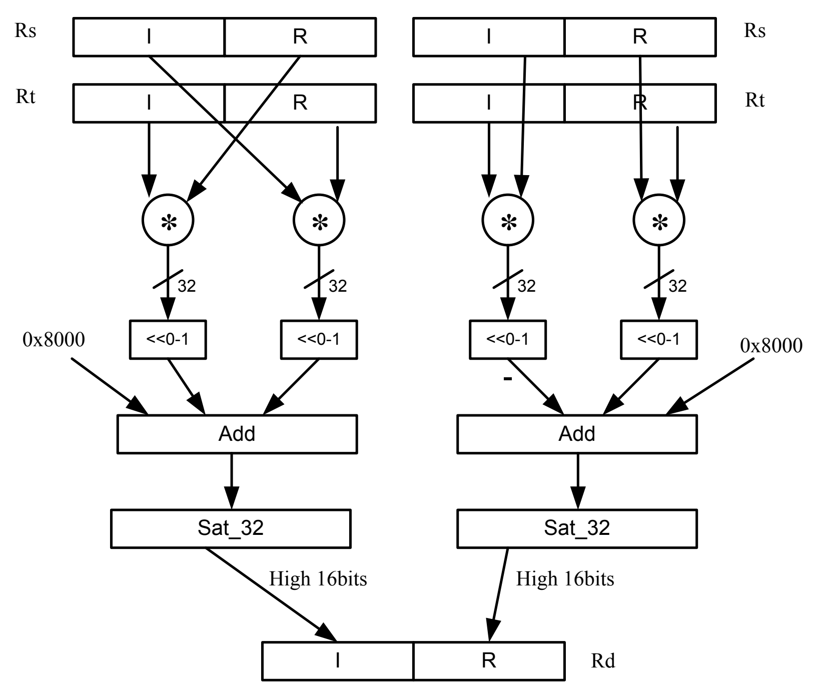

Variable sized instruction packets (1 to 4 instructions per Packet)
Instruction Cache
Instruction Unit
Dual 64-bit execution units
Standard 8/16/32/64bit data types
SIMD vectorized MPY / ALU / SHIFT, Permute, BitOps
Up to 8 16b MAC/cycle
2 SP FMA/cycle
Device DDR Memory
L2 Cache / TCM
Dual 64-bit load/store units
Also 32-bit ALU
Data Unit (Load/ Store/ ALU)
Data Unit (Load/ Store/ ALU)
Execution Unit (64-bit Vector)
Execution Unit (64-bit Vector)
Data Cache
Register File/Thread
Unified 32x32bit General Register File is best for compiler.
No separate Address or Accum Regs
Per-Thread

64-bit Load and

64-bit Store with post-update addressing

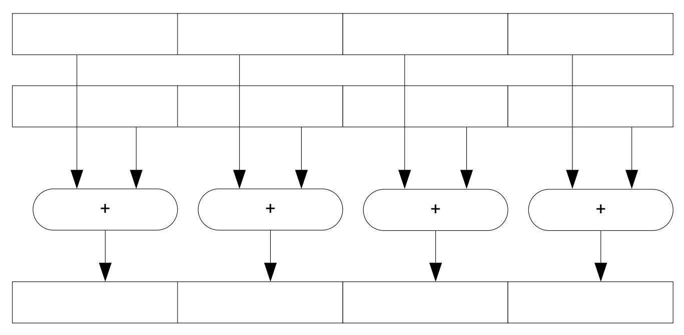
```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

Complex multiply with round and saturation



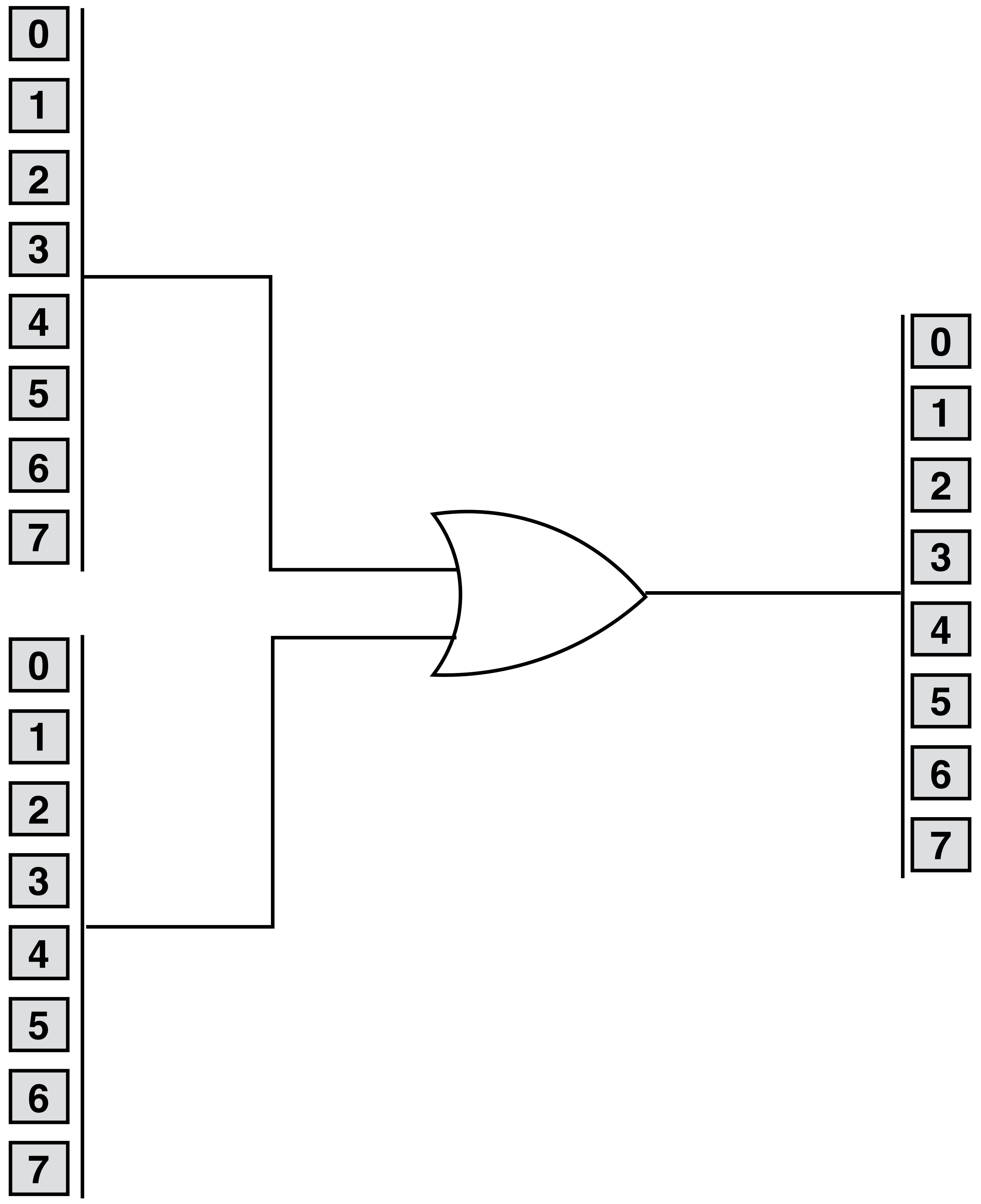
Zero-overhead loops
- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



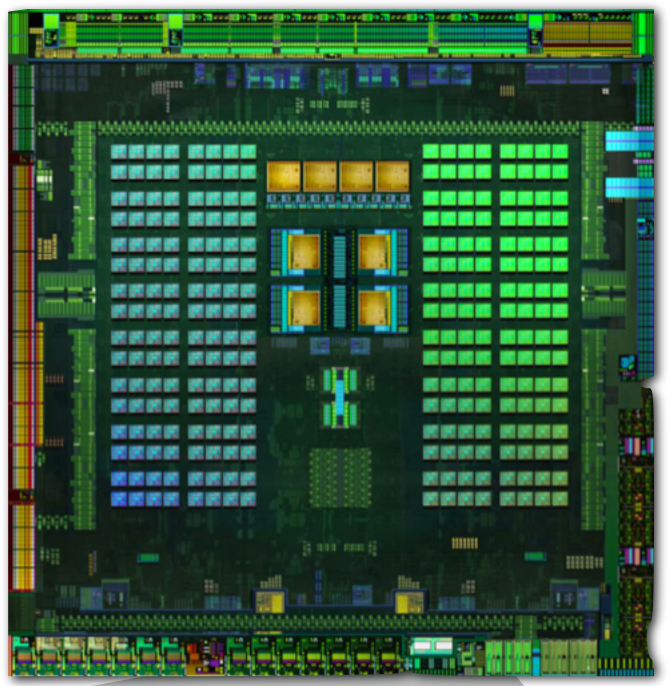
Contrast to custom circuit to perform the operation

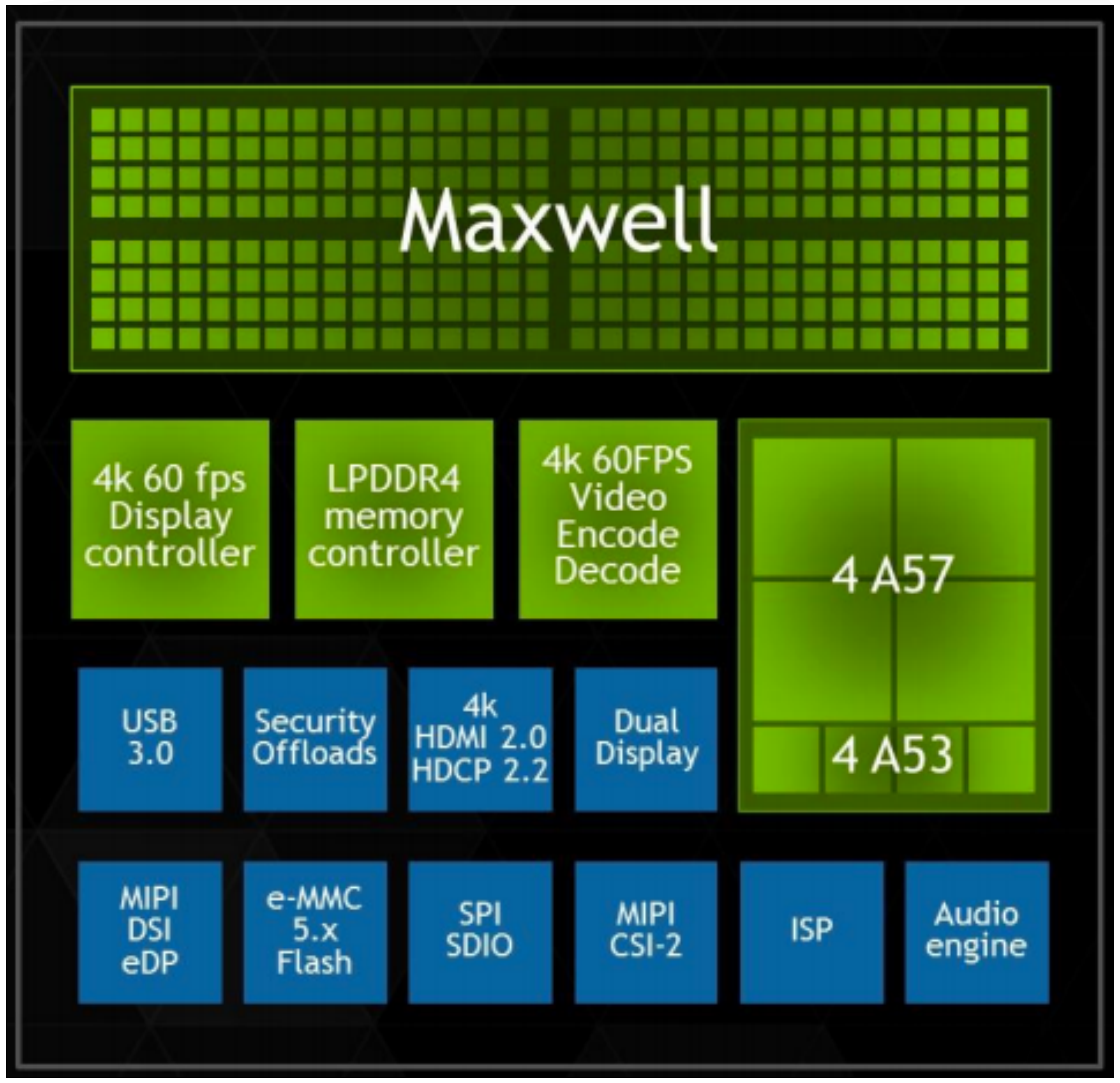
Example: 8-bit logical OR



Recall use of custom circuits in modern SoC

Example: NVIDIA Tegra X1



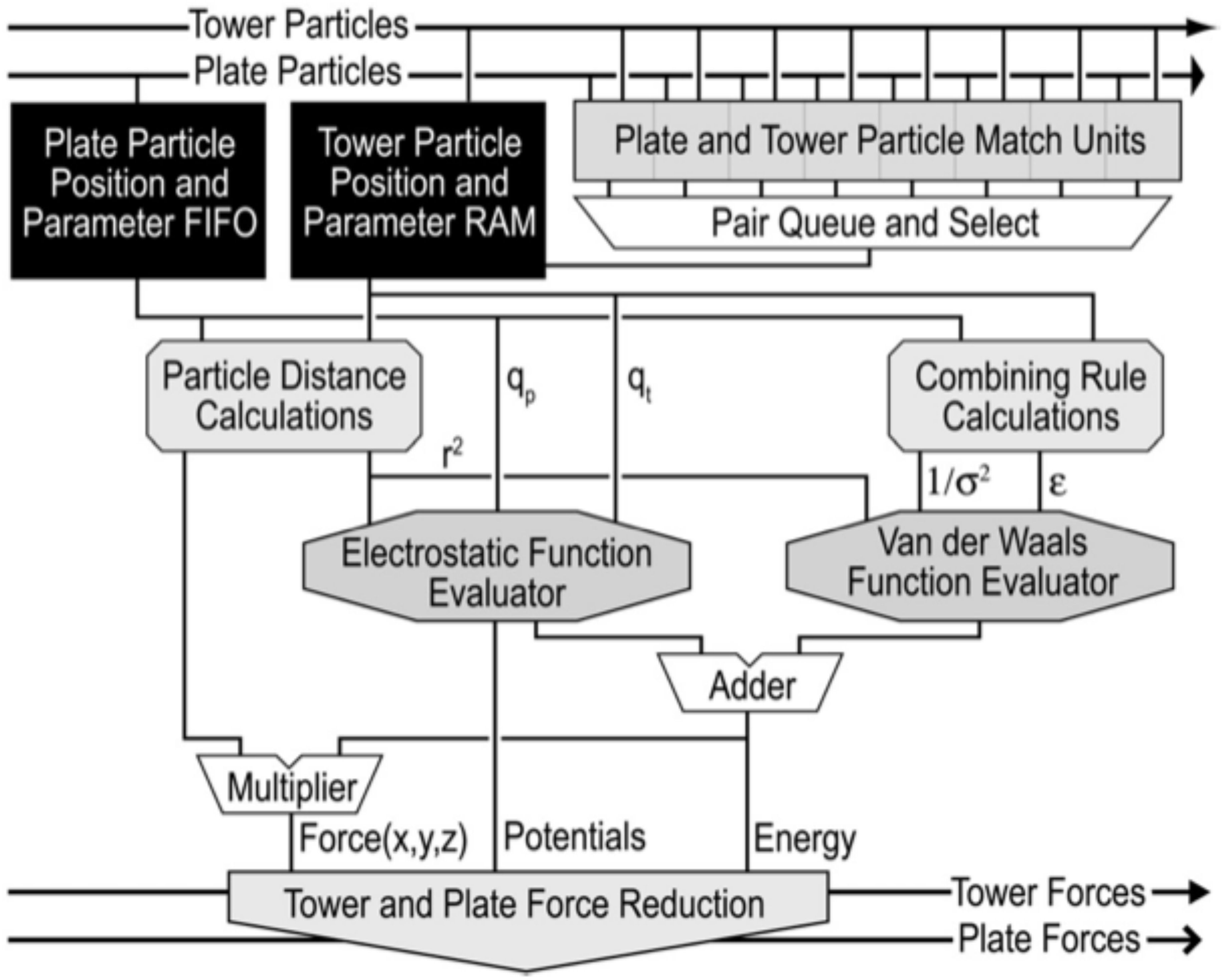


Audio encode/decode
Video encode/decode
High-frame rate camera RAW processing (ISP)
Data compression

Aside: Anton supercomputer

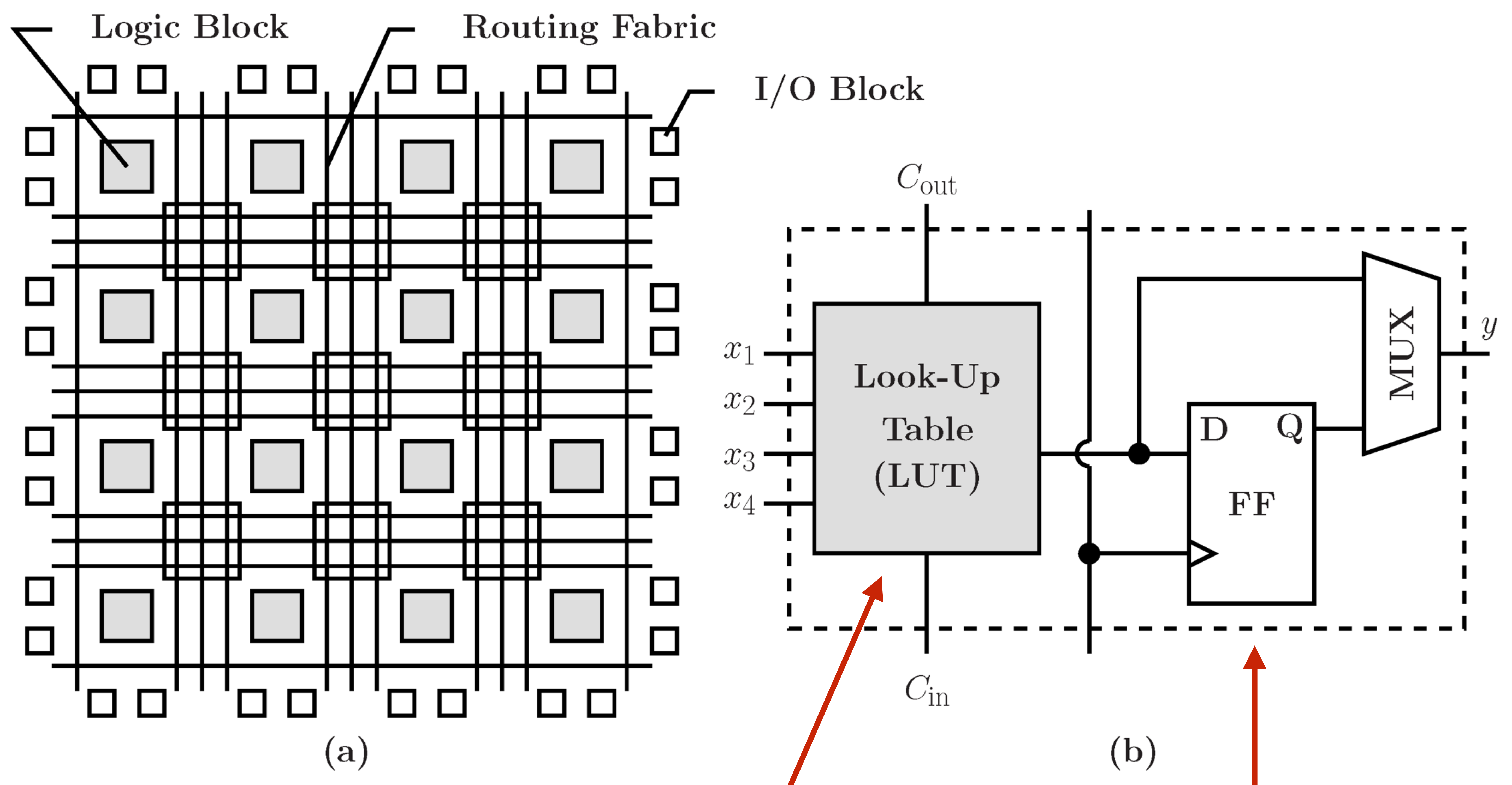
[Developed by DE Shaw Research]

- Supercomputer containing custom circuits for molecular dynamics
  - Simulates time evolution of proteins
- ASIC for computing particle-particle interaction in a single cycle
  - Anton 1: 512 particle-particle interaction units


Tower Particles
Plate Particles
Plate Particle Position and Parameter FIFO
Tower Particle Position and Parameter RAM
Plate and Tower Particle Match Units
Pair Queue and Select
Particle Distance Calculations
q_p
q_t
Combining Rule Calculations
r^2
$1/\sigma^2$
ε
Electrostatic Function Evaluator
Van der Waals Function Evaluator
Adder
Multiplier
Force(x,y,z)
Potentials
Energy
Tower and Plate Force Reduction
Tower Forces
Plate Forces

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

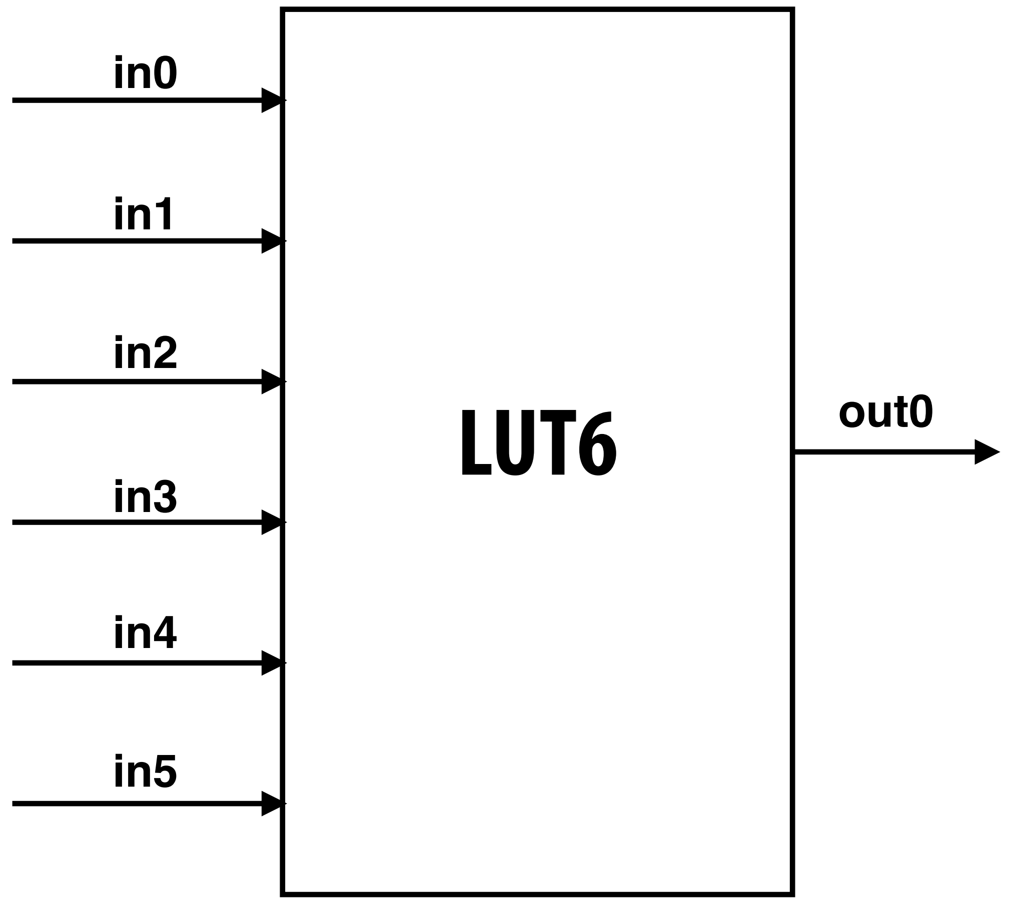


Flip flop (a register)

Programmable lookup table (LUT)

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)

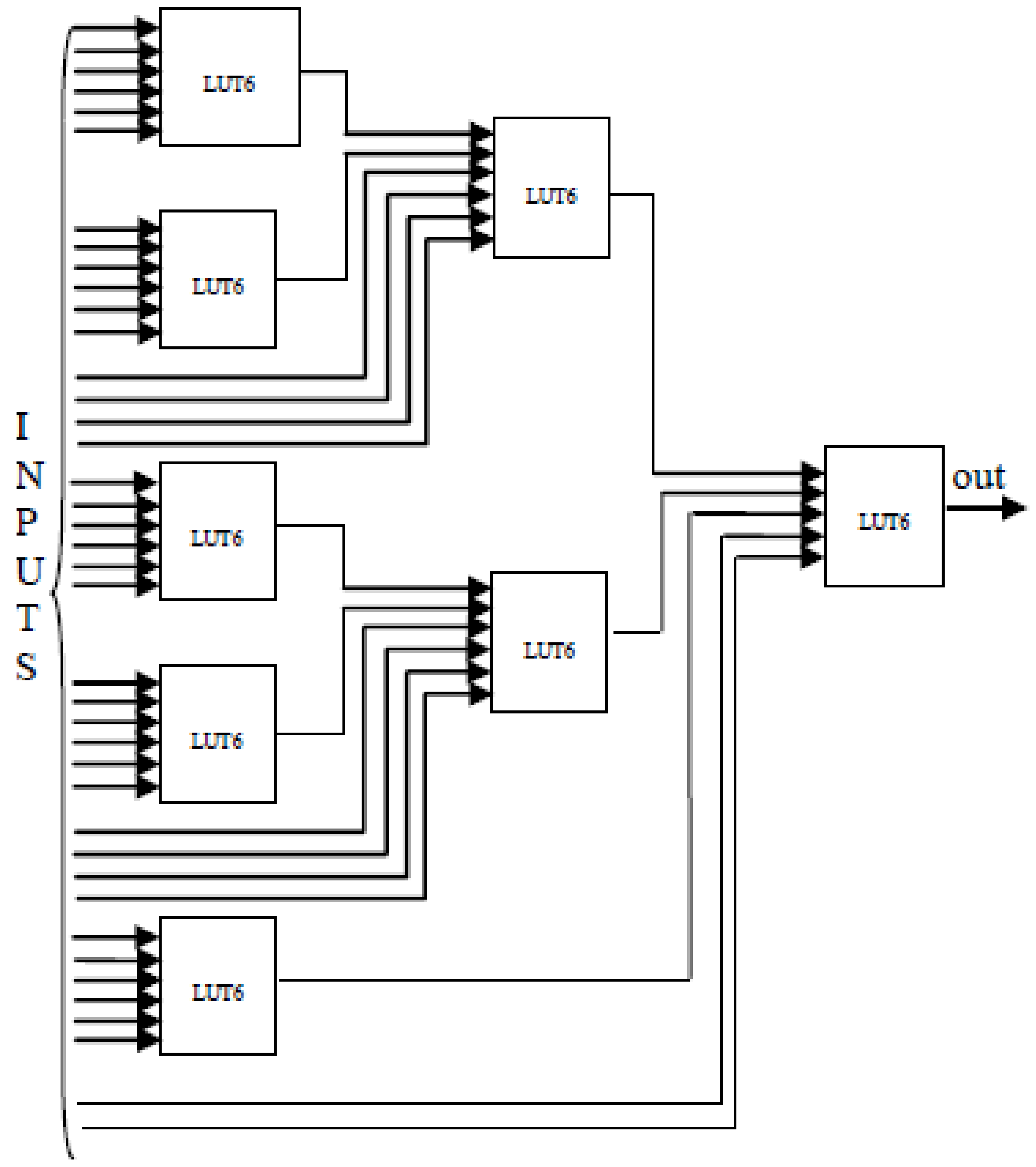


Image credit: [Zia 2013]

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)

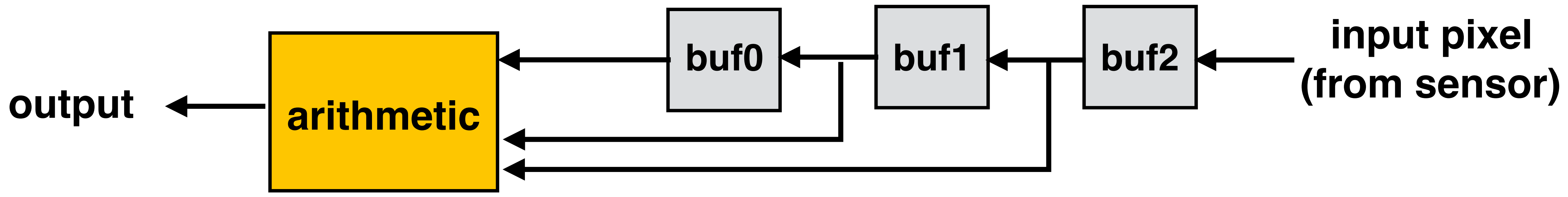
Consider 1D convolution:

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

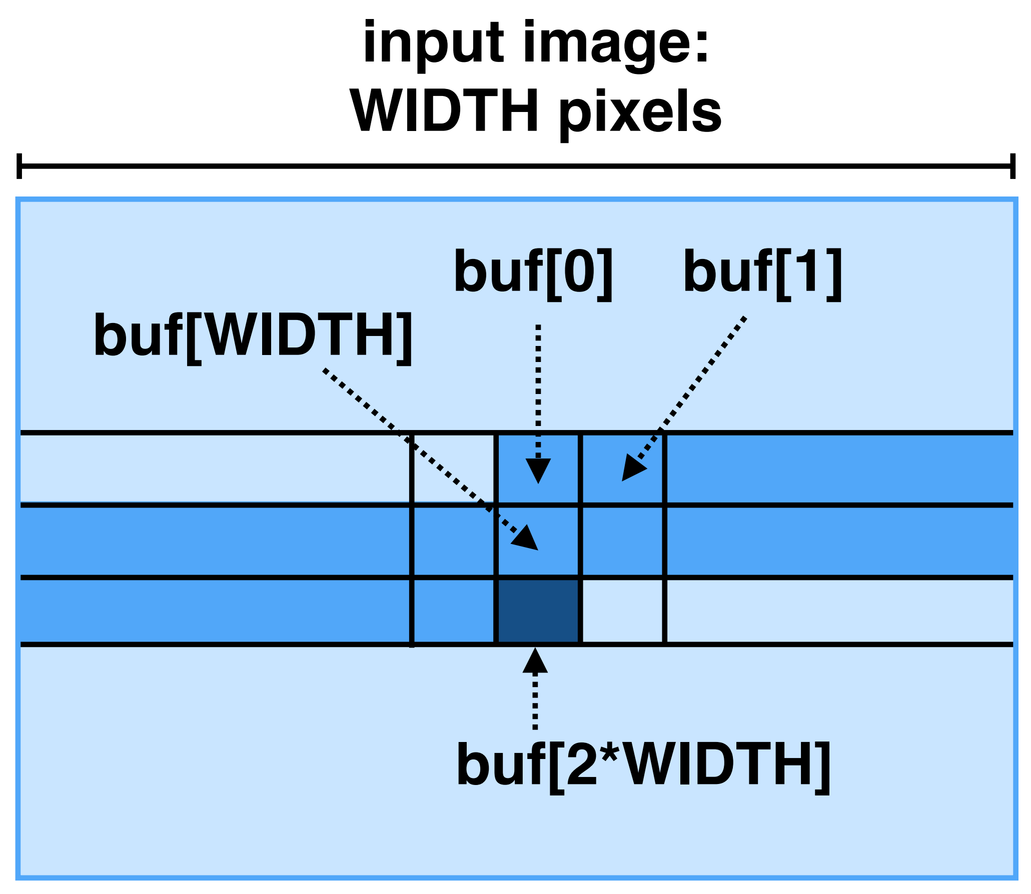
Efficient hardware implementation: requires storage for 3 pixels in registers

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

"Shift" new pixel in



Line buffering



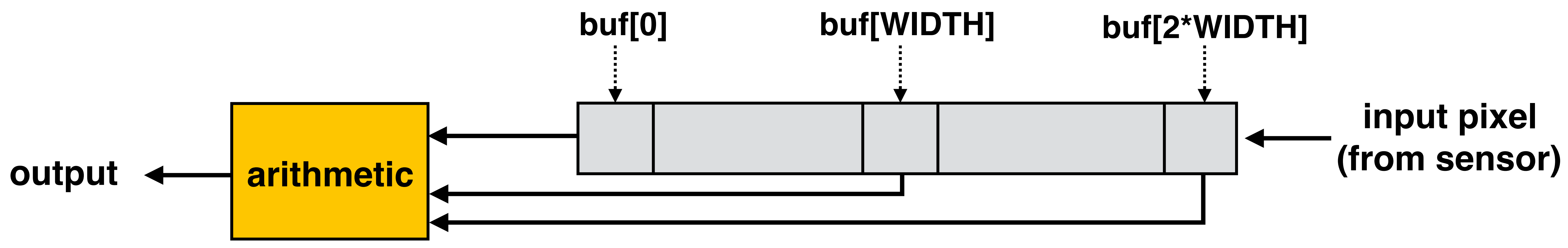
Consider convolution of 2D image in vertical direction:

```
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

Class discussion: Rigel

[Hegarty 2016]