CMPE110 Lecture 22

Custom Hardware Designs

Heiner Litz

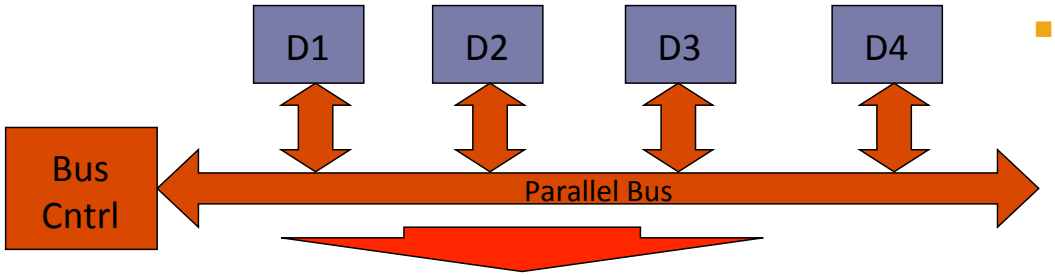
https://canvas.ucsc.edu/courses/12652

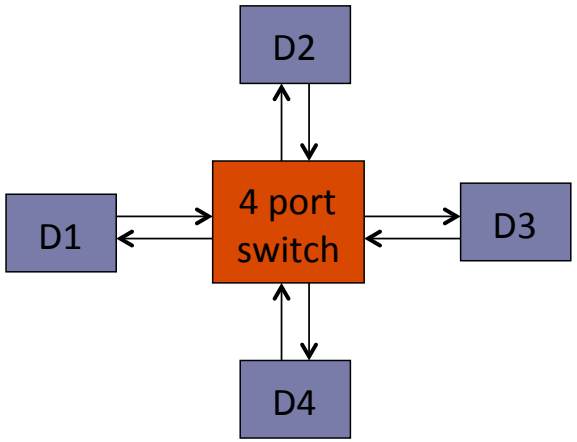
Announcements

- HA 5 Matrix Multiplication
  - Updated baseline implementation
  - Improves reproducibility/stability of results
  - Submission deadline moved to today
- HA 6 has been released
- Student Evaluations
  - Important for the quality of this course
  - Submissions are guaranteed to be **confidential**!
  - Important for me

Review

Alternative: Point-to-Point Links & Switches





- Serial point-to-point advantages
  - Faster links
  - Fewer chip package pins
  - Higher performance
  - Switch keeps arbitration on chip
- Many bus standards are moving to serial, point to point
- 3GIO, PCI-Express(PCI)
- Serial ATA (IDE hard disk)
- AMD Hypertransport, Intel QPI versus Intel Front Side Bus (FSB)

DMA Issues (1): Memory Pinning

- For DMA with physical addresses, memory pages must be pinned in DRAM - Why?

- OS should not page to disks pages involved with pending I/O

DMA Issues (2): Memory Translation

- If DMA uses physical addresses
  - Memory access across physical page boundaries may not correspond to contiguous virtual pages (or even the same application!)
- Solution 1: ≤ 1 page per DMA transfer
- Solution 1+: chain series of 1-page requests provided by the OS
  - Single interrupt at the end of the last DMA request in the chain
- Solution 2: DMA engine uses virtual addresses
  - Multi-page DMA requests are now easy
  - A TLB is necessary for the DMA engine
    - Who manages the TLB?

DMA Issues (3): Cache Coherence

- The data involved in DMA may reside in processor cache
  - If updated by I/O: must update or invalidate “old” cached copy
  - If read by I/O: must read latest value, which may be in the cache
  - Only a problem with write-back caches
- Another version of the “cache coherence” problem
  - Same problem in multi-core systems

DMA & Coherence

- Solution 1: OS flushes the cache before I/O reads or forces write-backs before I/O writes
  - Flush/write-back may involve selective addresses or whole cache
  - Can be done in software or with hardware (ISA) support
- Solution 2: Route memory accesses for I/O through the cache
  - Search the cache for copies and invalidate or write-back as needed
  - This hardware solution may impact performance negatively
    - While searching cache for I/O requests, it is not available to processor
- Cache coherent I/O device
  - Processor snoops I/O transactions and enforces write-back

Custom Hardware Designs

Efficiency

We care about multiple metrics

- Performance
  - ExecutionTime = IC * CPI * CCT, operations/second
- Power = C * Vdd^2 * F
- Energy = C * Vdd^2
- Cost
  - Area, design and testing complexity

Can we improve performance **without** increasing power, energy, and cost?

Reminder: Improving Performance without Increasing Power

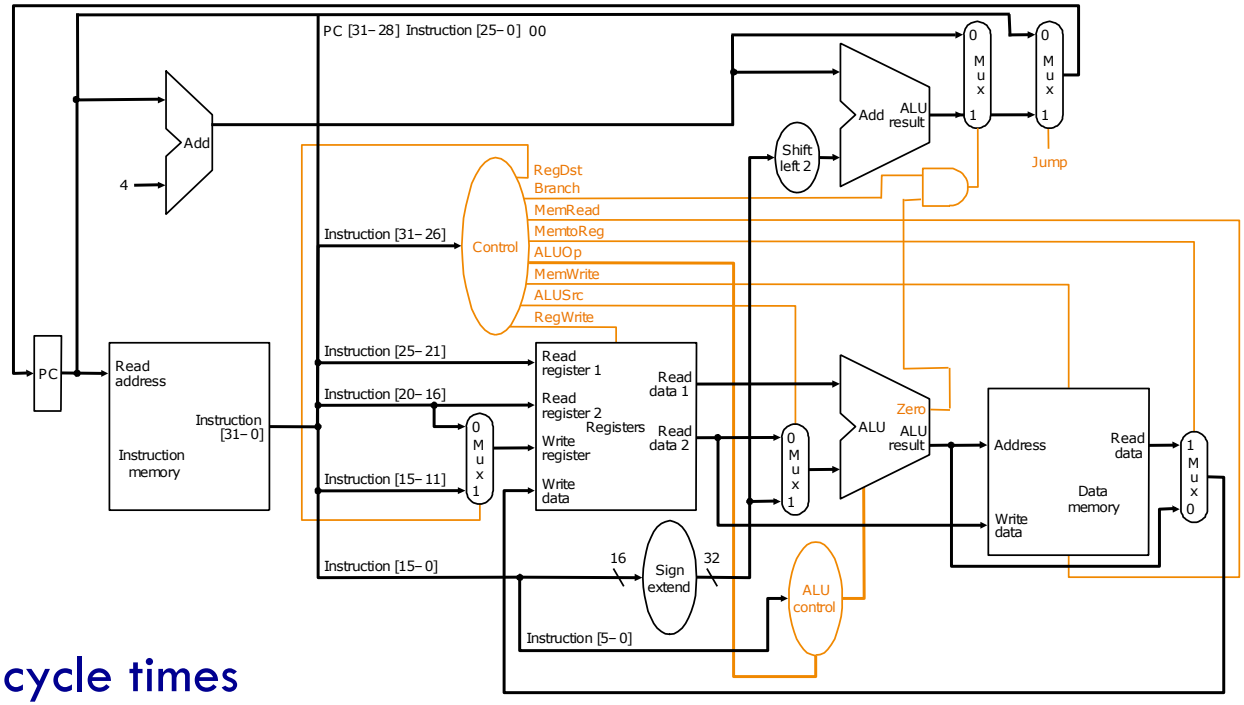
- So, how do we go about this?
  - Remember: **Power = C *Vdd^2*$F_{0\rightarrow1}$ + Vdd*$I_{leakage}$**

- What should we optimize processors for?
  - Answer: energy per instruction (EPI)

$$Power = \frac{energy}{second} = \frac{energy}{instruction} \times \frac{instructions}{second}$$

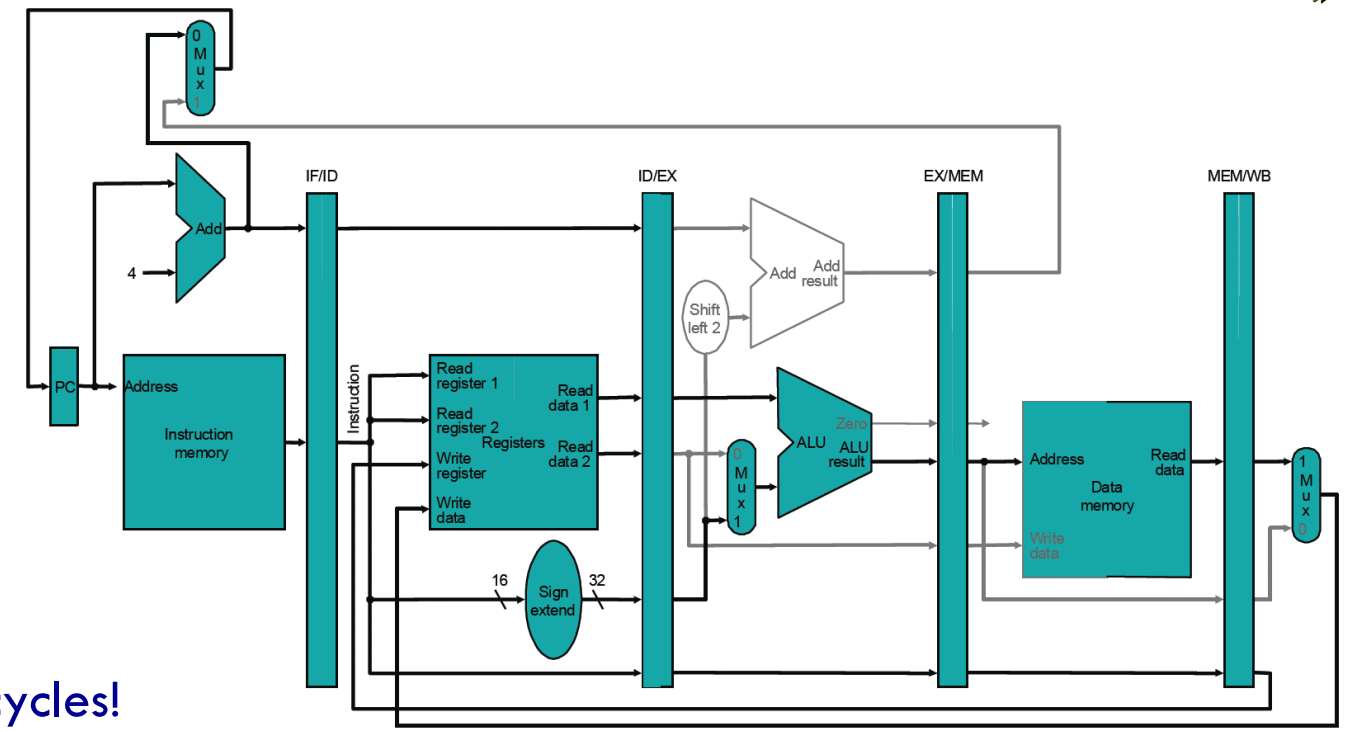
  - After minimizing EPI, tune performance & power as needed
    - Higher for server, lower for cellphone

Single-cycle Processor



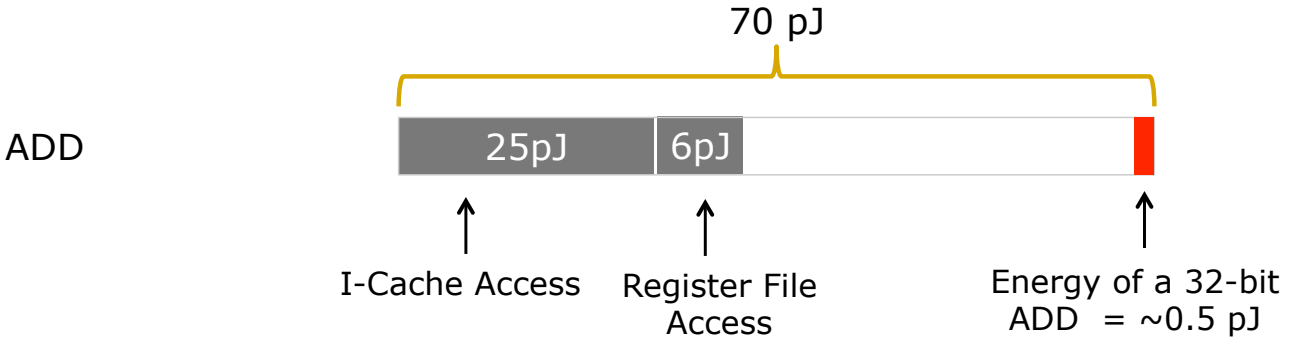
- Long cycle times
- Low reuse (area inefficient)
- Energy efficiency?

5-stage Pipeline



- Fast cycles!
- Lots of reuse!
- Energy efficiency?

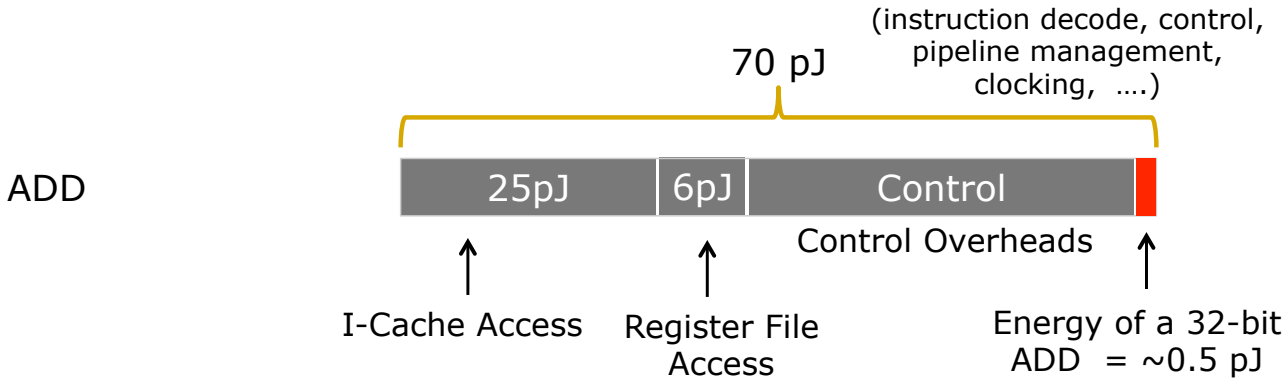
Anatomy of a RISC Instruction



0.7% energy goes to math!

* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

Anatomy of a RISC Instruction



* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

That's Not All!

ADD

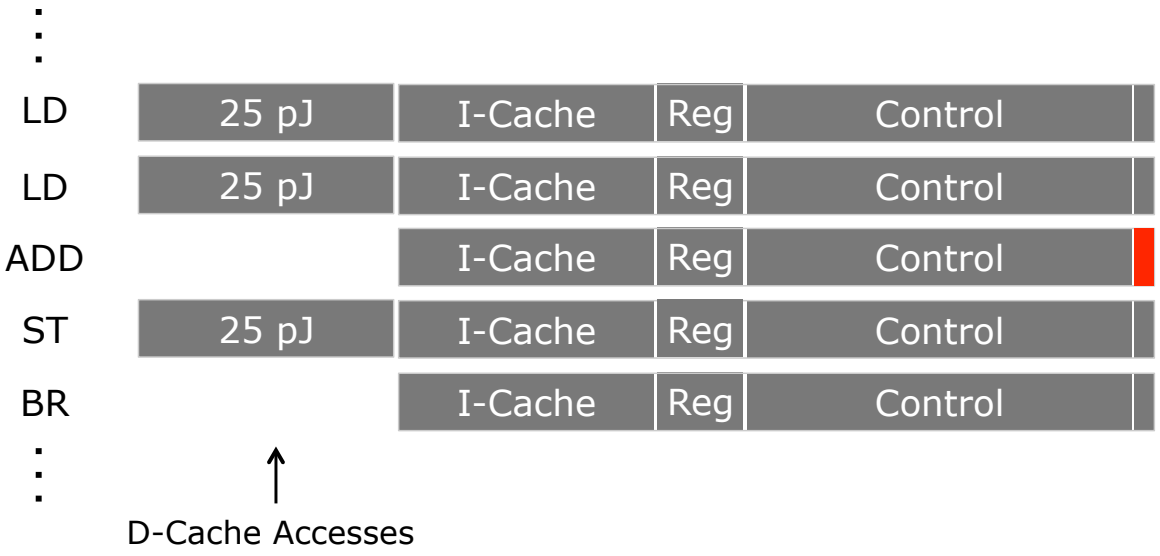
* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

Don't Forget the Overhead Instructions

⋮			
LD	I-Cache	Reg	Control
LD	I-Cache	Reg	Control
ADD	I-Cache	Reg	Control
ST	I-Cache	Reg	Control
BR	I-Cache	Reg	Control
⋮			

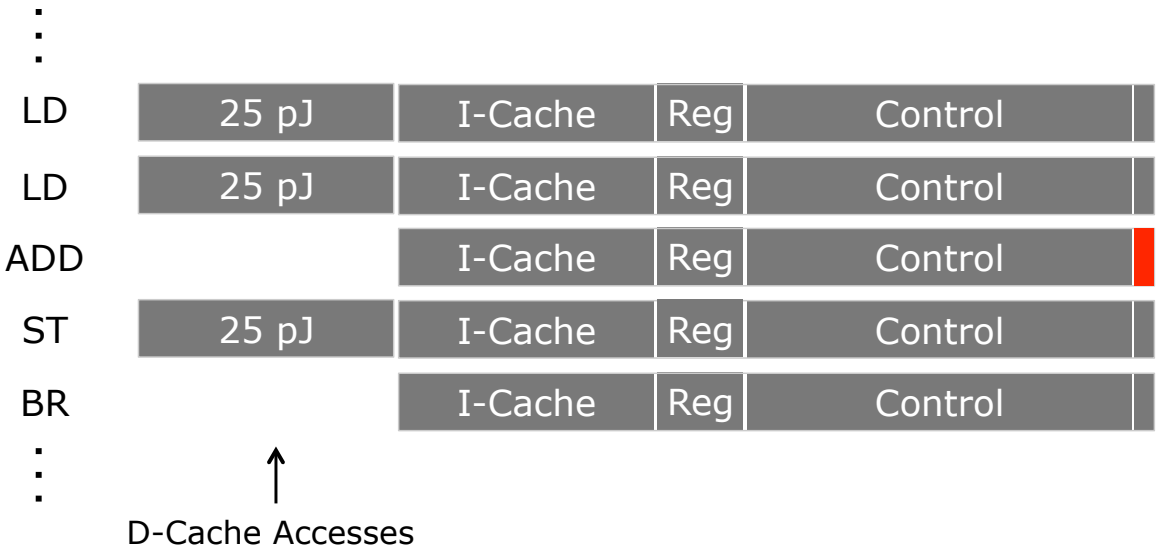
* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

Don't Forget the Overhead Instructions



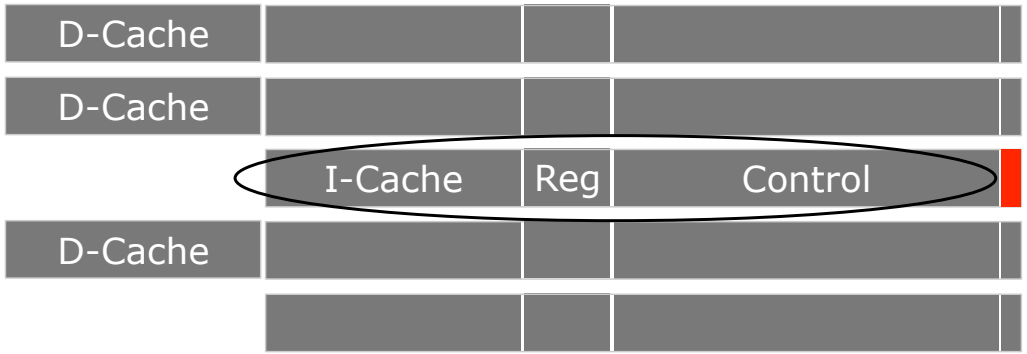
* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

How Do We Reduce This Waste?



* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

Reducing This Waste



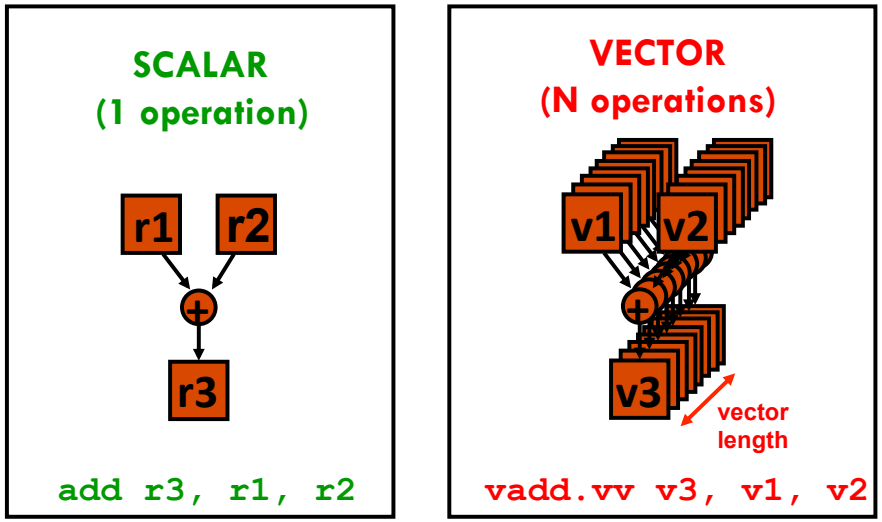
1. Perform a large number of operations per instruction

Requirements For Efficient Execution

1. Perform a large number of operations per instruction
2. Perform a large number of operations for each D-Cache access (locality)

Have We Done this Already?

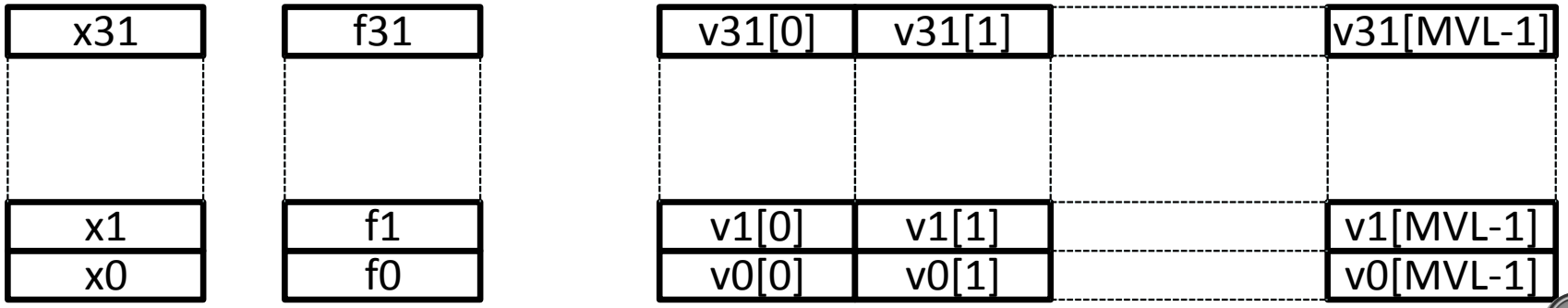
Data-level Parallelism (Vectors)



- Scalar instructions operate on single numbers (scalars)
- Vector instructions operate on vectors of numbers
  - Linear sequences of numbers
  - Stored in a vector register file

RISC-V Vector Extensions

- Add Vector Registers
- Variable size (depending on hardware)
- Add Vector Instructions (vsetvl, vld, vst, vadd,..)



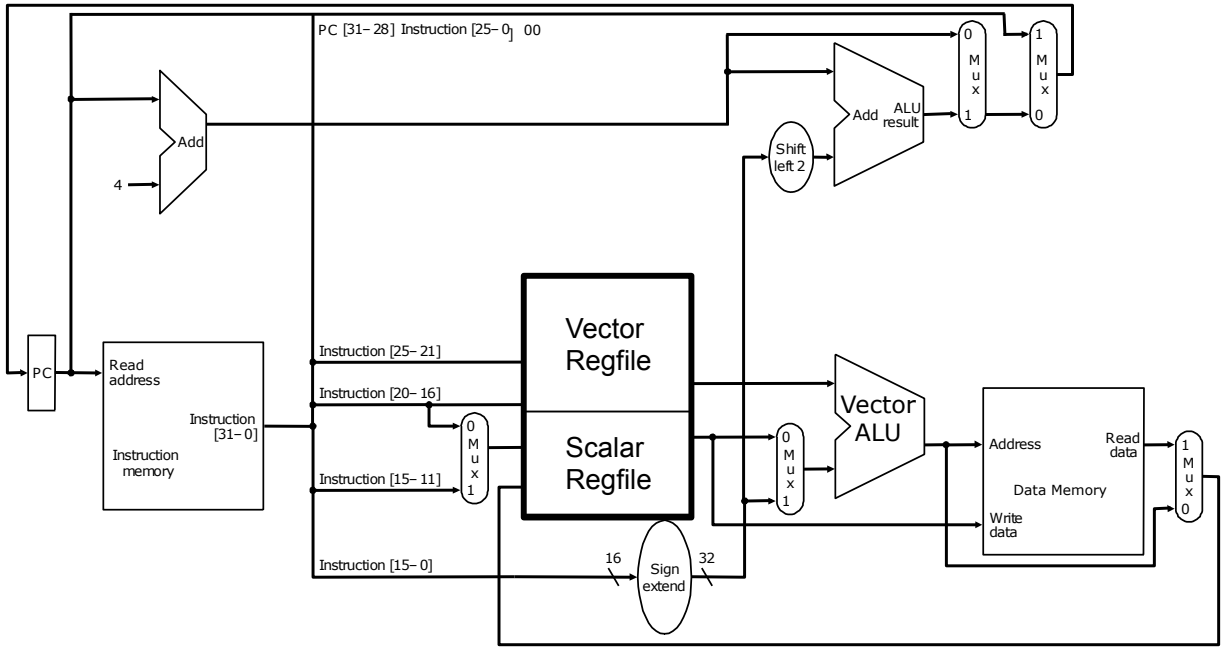
RISC-V Vector Example


```
stripmine:
  vsetvl t0, a0 # a0 holds vector length
  vld v0, a1    # Get first vector
  vld v1, a2    # Get second vector
  vadd v1, v0   # Add vectors
  vst v1, a3    # Store result vector
  sll t1,t0,2   # Multiply count by 4 to get byte
  add a1, t1    # Bump pointers
  add a2, t1
  add a3, t1
  sub a0, t0    # Subtract number done
  bnez a0, stripmine # Any more?
```

Benefits of Vector Instructions

- Each instruction specifies more parallel work
  - Can build a higher-performance processor
- Instruction overheads amortized better
  - Overhead: energy for fetch & decode instruction
  - Amortized over multiple arithmetic ops
- Fewer overhead instructions
  - Vector loops

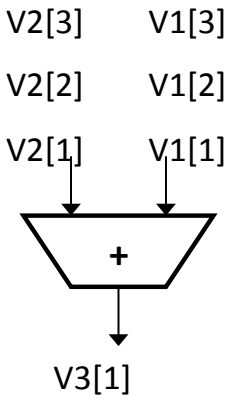
Simple Vector Processor



- Add vector register file
- How do we vectorize the ALU and the control?

Vector ALUs

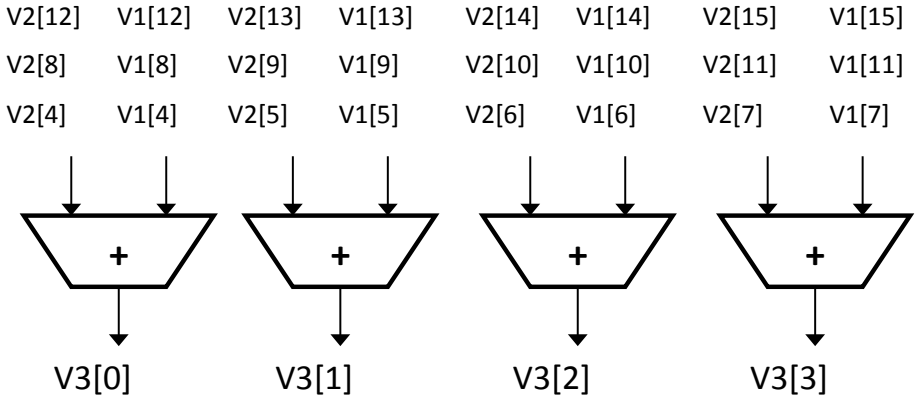
vadd v2,v1 (VL=N)



1 element/cycle

N cycles

1 adder



4 elements/cycle

N/4 cycles

4 adders

Vector/SIMD Summary

- Vector/SIMD can offer around 10x benefit for some apps
- Does SIMD help all applications?

How About Multi-core?

- How does multi-core compare?
  - Performance?
  - Energy?
  - Area?
- What if we can decrease total frequency and voltage?

Let’s go back to basics

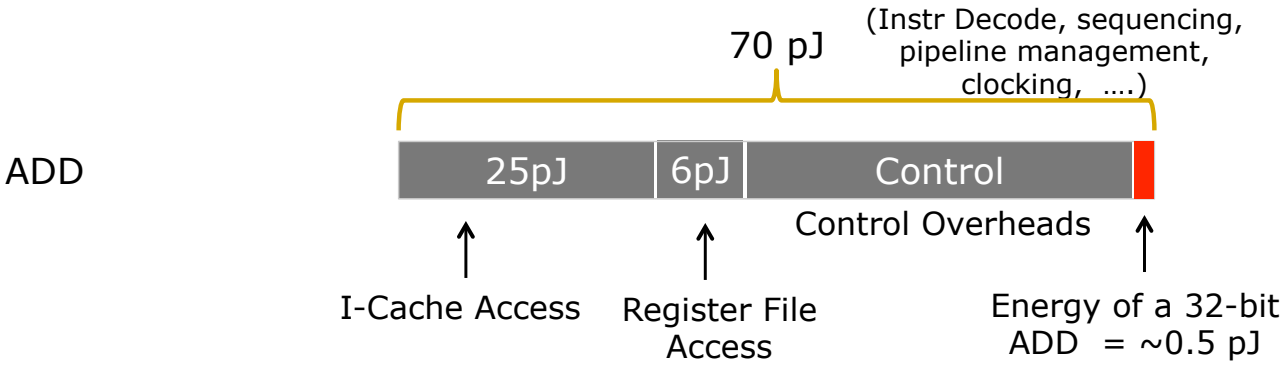
Programmability vs. Efficiency

- Processors are highly programmable
  - Can implement any algorithm you can program
  - Cost: Instruction overhead
- Custom hardware is highly efficient
  - Only build the hardware you need
  - Can only compute one algorithm

Custom Hardware

- Designed to run one algorithm really fast
- Contains algorithm or domain specific hardware
  - Specific mix of math and control units
  - Specific precision at each stage
  - Stages are cascaded
    - 1 instruction, 1 writeback

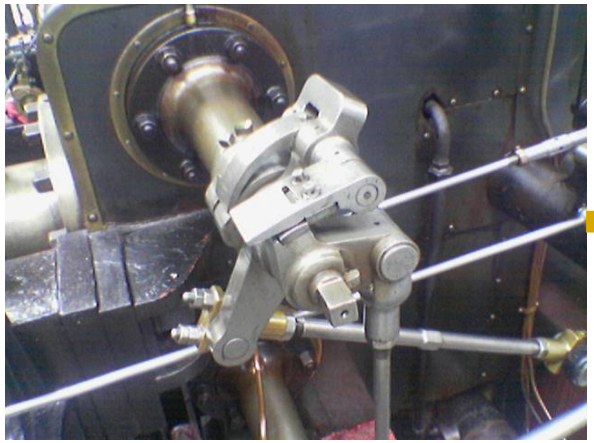
Custom Hardware

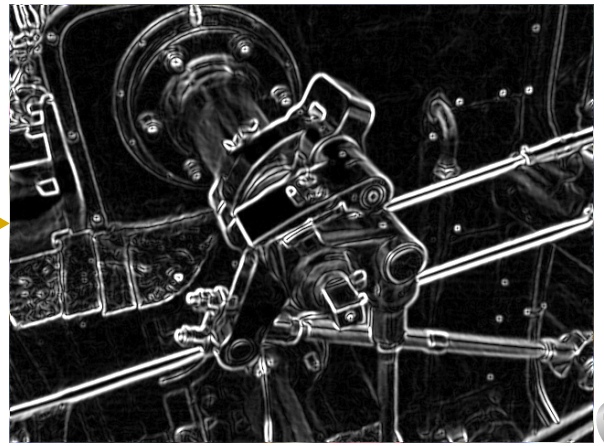


* Assuming a typical 32-bit embedded RISC in 45nm @ 0.9V technology

Sobel Filter

- Image processing and computer vision
- Performs edge detection
- 2D-Convolution with a filter





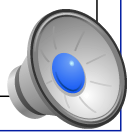
Sobel Filter

- Per pixel work
  - 15-16 arithmetic ops + some overheads
- Arithmetic operations
  - 4 shifts
  - 10 adds/sub (accumulate)
  - Value clamp (comparison)

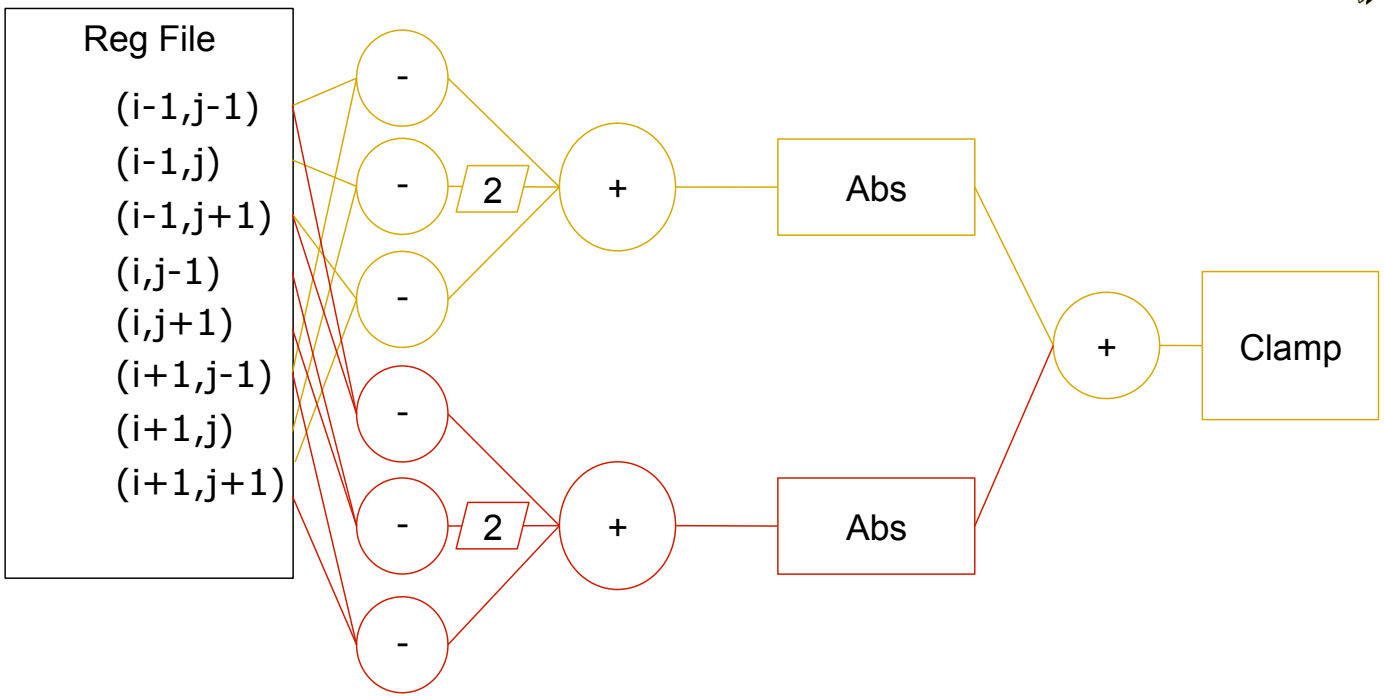
Sobel Filter

- Arithmetic operations
  - 4 shifts
  - 10 adds/sub (accumulate)
  - 2 abs
  - 1 value clamp (comparison)
- How many instructions?

```
//Calculate the x convolution
 for (int i=1; i<gray_img.rows; i++) {
   for (int j=1; j<gray_img.cols; j++) {
     sobel = abs(gray_img.data[IMG_WIDTH*(i-1) + (j-1)]-
              gray_img.data[IMG_WIDTH*(i+1) + (j-1)] +
              2*gray_img.data[IMG_WIDTH*(i-1) + (j)] -
              2*gray_img.data[IMG_WIDTH*(i+1) + (j)] +
              gray_img.data[IMG_WIDTH*(i-1) + (j+1)] -
              gray_img.data[IMG_WIDTH*(i+1) + (j+1)]);

    sobel += abs(gray_img.data[IMG_WIDTH*(i-1) + (j-1)] -
              gray_img.data[IMG_WIDTH*(i-1) + (j+1)] +
              2*gray_img.data[IMG_WIDTH*(i) + (j-1)] -
              2*gray_img.data[IMG_WIDTH*(i) + (j+1)] +
              gray_img.data[IMG_WIDTH*(i+1) + (j-1)] -
              gray_img.data[IMG_WIDTH*(i+1) + (j+1)]);

    sobel = (sobel > 255) ? 255 : sobel;

img_out.data[IMG_WIDTH*(i) + j] = sobel;
   }
 }
```

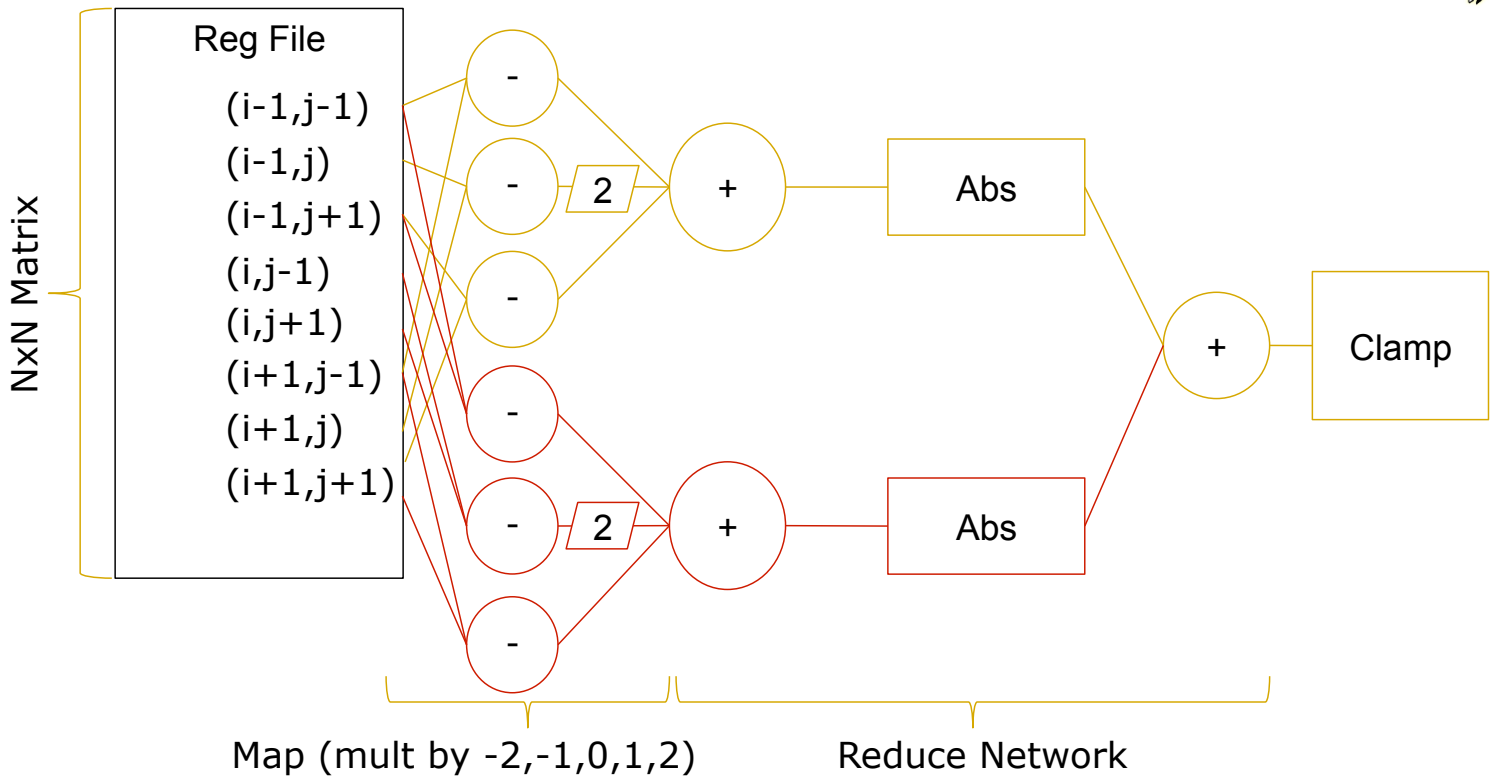


Custom HW for Sobel Filter

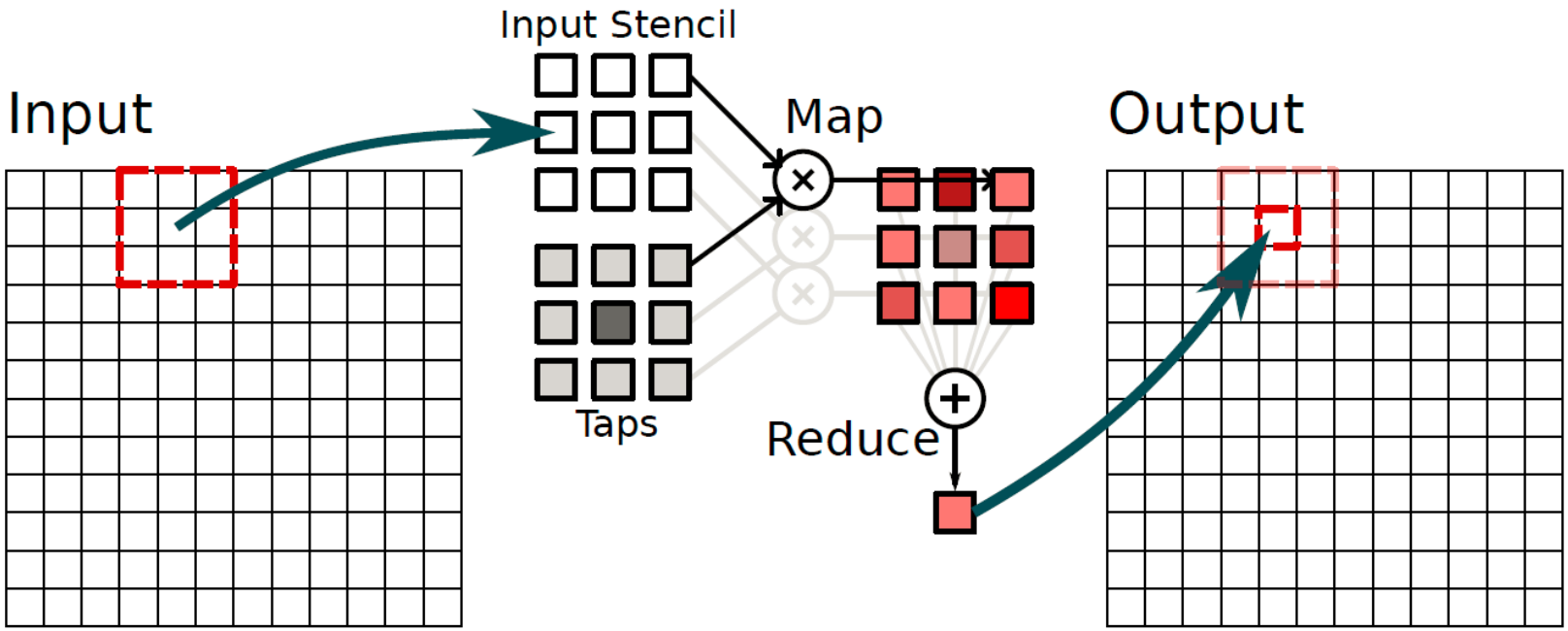


- 1 Instruction per pixel
- How general is this?

Domain Specific HW



Convolution Engine



Domain Specific Hardware

	Map	Reduce	Stencil Size	Data Flow
IME SAD	Abs Diff	Add	4x4	2D Convolution
FMW ½ pixel up-sample	Multiply	Add	6	1D Horizontal & vertical conv.
FME ¼ pixel up-sample	Average	None	--	2D Matrix operation
SIFT Gaussian blur	Multiply	Add	9, 13, 15	1D Horizontal & vertical conv.
SIFT DoG	Subtract	None	--	2D Matrix operation
SIFT Extreme	Compare	Logic AND	3	1D Horizontal & vertical conv.
Demosaic interpolation	Multiply	Complex	3	1D Horizontal & vertical conv.

- Find similar computations within a domain
- Sacrifice some efficiency for generality within domain

Memory Concerns: Bandwidth Matching

- Can only operate as fast as we can feed it
  - Memory too slow: Accelerator must stall
  - Accelerator too slow: Add more accelerators (or make yours bigger!)
- Match performance with available memory bandwidth

Memory Overheads

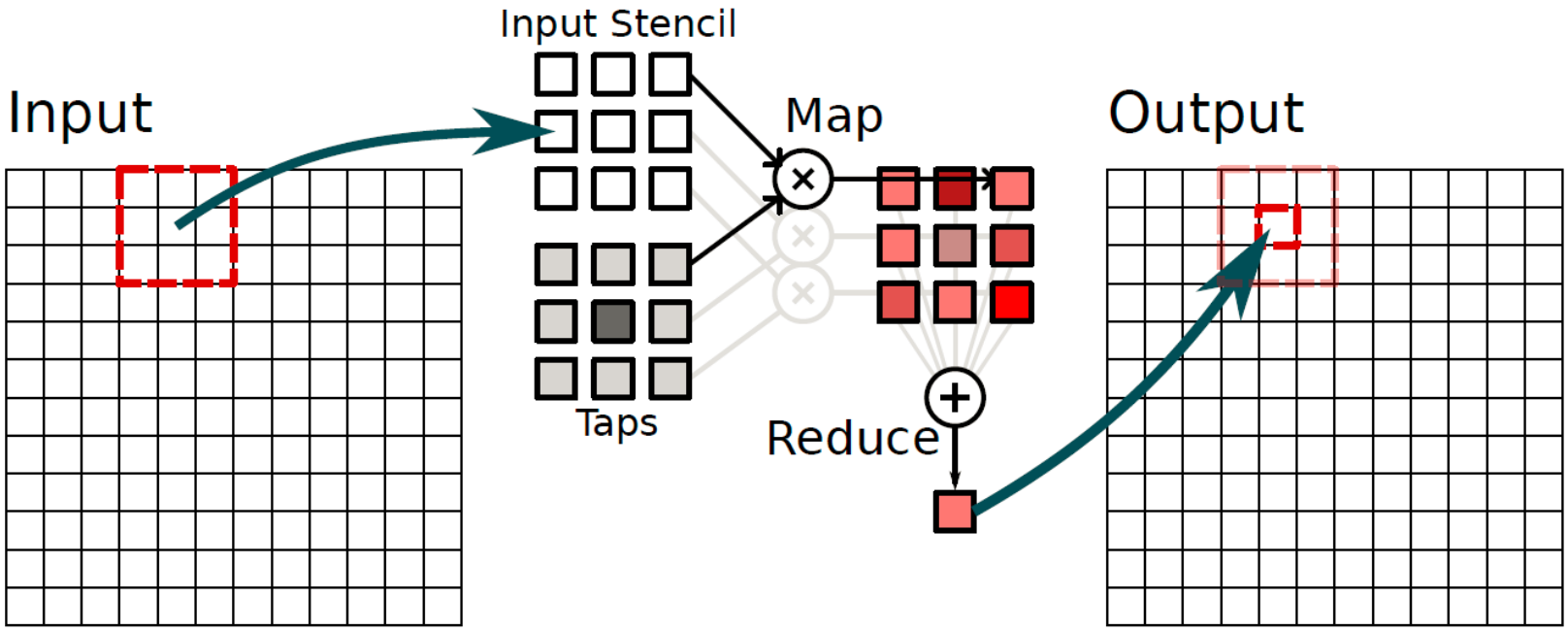
Operation	Energy	Scale
8-bit add	0.03 pJ	1
32-bit add	0.10 pJ	3
32-bit FP mult	4.00 pJ	133
RISC instruction	**70.00 pJ**	**2,300**
8KB cache access	10.00pJ	300
1MB cache access	**100.00 pJ**	**3,000**
DRAM access	**2,000.00 pJ**	**60,000**

Minimizing Memory Access

- Getting data to and from accelerators (and cores) is expensive
  - Especially if that data is in DRAM
- To minimize energy
  - Minimize read/writes by buffering operands and intermediates locally
  - Maximize data reuse (where do we find reuse in Sobel?)

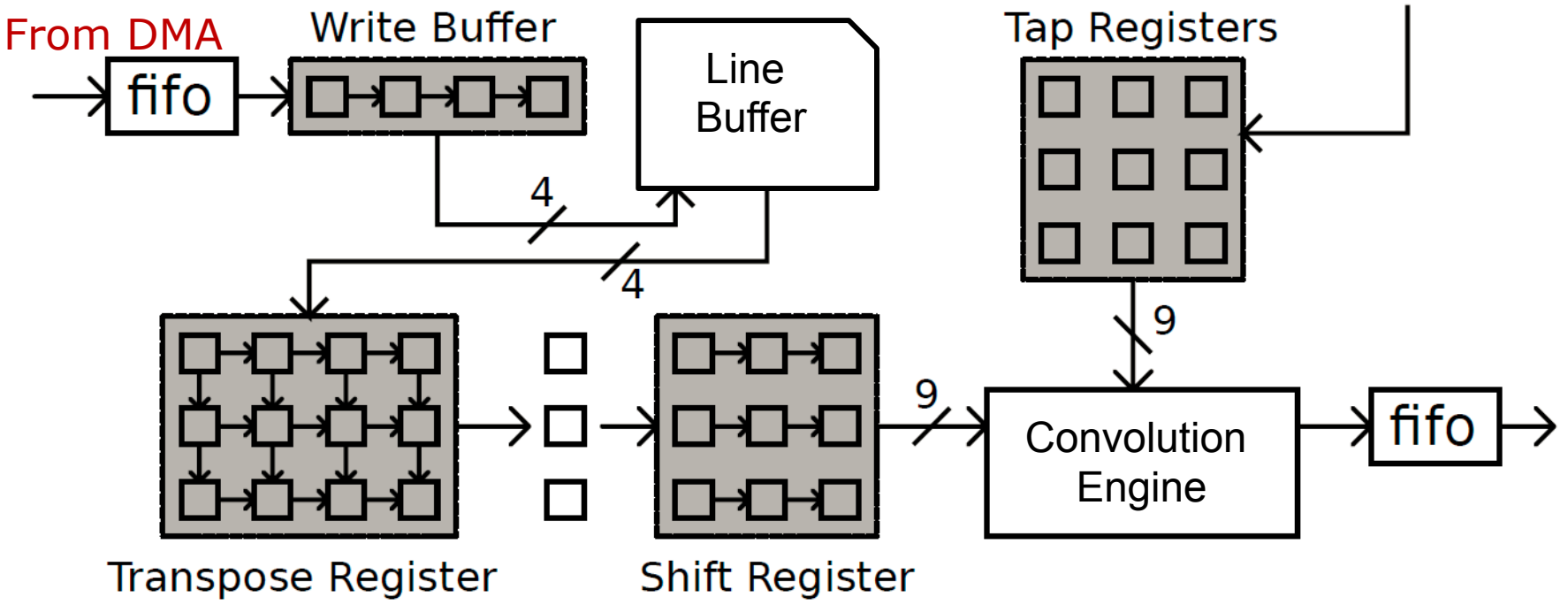
Read and write once, reuse often!

Convolution Engine



- How many rows of data do we need to buffer?
- How much new data do we need per pixel?

Convolution Datapath



- How many pixels per cycle?

Managing the Accelerator

- Think of the accelerator as an I/O device
- Software manages the accelerator
  - Initiates Sobel computation
  - Provides input, receives output
  - Deals with errors

Moving Data Between CPU & Accelerator

- Direct Memory Access (DMA)
  - Streaming memory transfer mechanism
  - Takes in a set of source memory addresses
  - Takes in a MMIO target address
  - Streams source memory words to target

Writing to and from Custom HW

- Processor gathers image addresses
  - Ideally in a contiguous physical chunk
- Transmits addresses to DMA
- DMA writes data to accelerator
  - Signals when complete
- Accelerator computes on data
- Accelerator DMAs results back to allocated memory space

When To Build Custom?

- Custom can offer better performance or energy
- Why don't we build it for everything?

Custom Design Costs

- Design costs
  - RTL and verification
  - Driver software
- Area costs
  - Only handles one algorithm
  - Potentially limited use/reuse

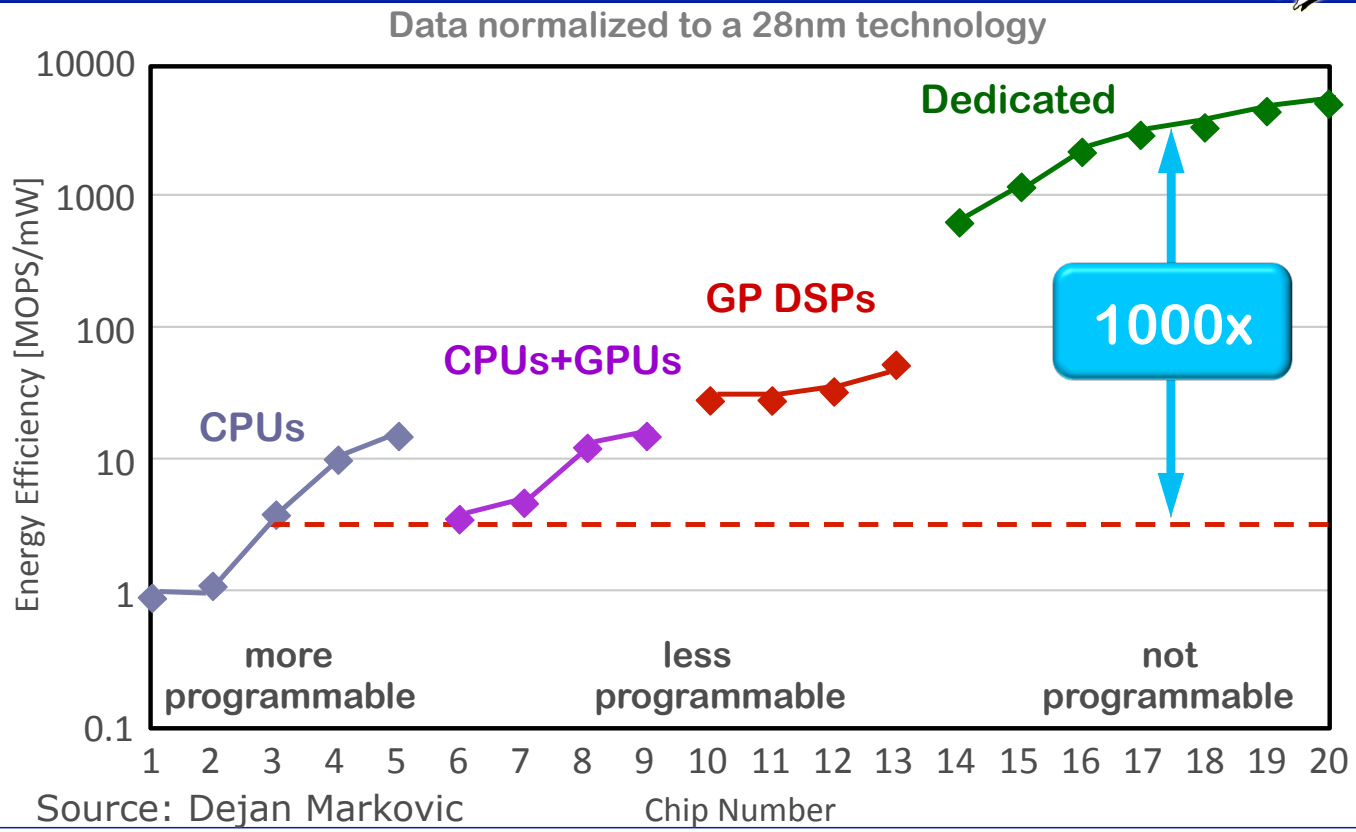
Is it worth it?

- How often will it be used?
  - May be able to generalize the accelerator at minor extra cost
- How much math does it do?
  - Are there enough ops to justify sending data to an accelerator?
  - Is the math lower/different precision?
- How much control flow does it have?
  - Control dominated applications may need processor-like flexibility
- What do memory accesses look like?
  - Are we spending most of our energy on DRAM accesses anyway?

Flexibility vs Efficiency

- Processors are programmable
  - Flexible – can target a large variety of problems
  - Inefficient – energy overheads of programmability
- Custom hardware is efficient
  - Efficient – does not incur overheads related to programmability
  - Inflexible – only does one thing, cannot be "reprogrammed" in a general way
- Can we do both?

Flexibility vs Efficiency



Source: Dejan Markovic

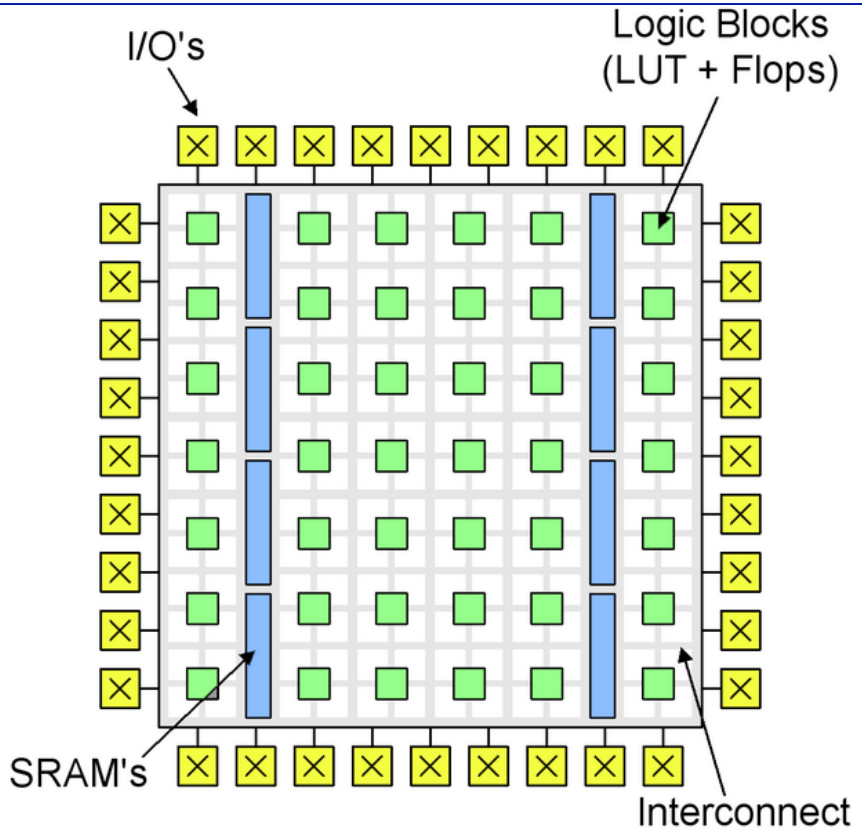
Reconfigurable Architectures

- Architecture composed of reconfigurable building elements
  - Compute, Memory, Interconnect
- Building blocks laid out as a 1-D or 2-D array on the chip
  - Hence the name "Spatial Architectures"
- Configuration bits controls operations of each element
  - Usually stored in dedicated SRAM cells that drive a MUX select signal to control a reconfigurable element
- Different configurations implement different "custom hardware"
  - Same underlying architecture, different config bits

Example - FPGAs

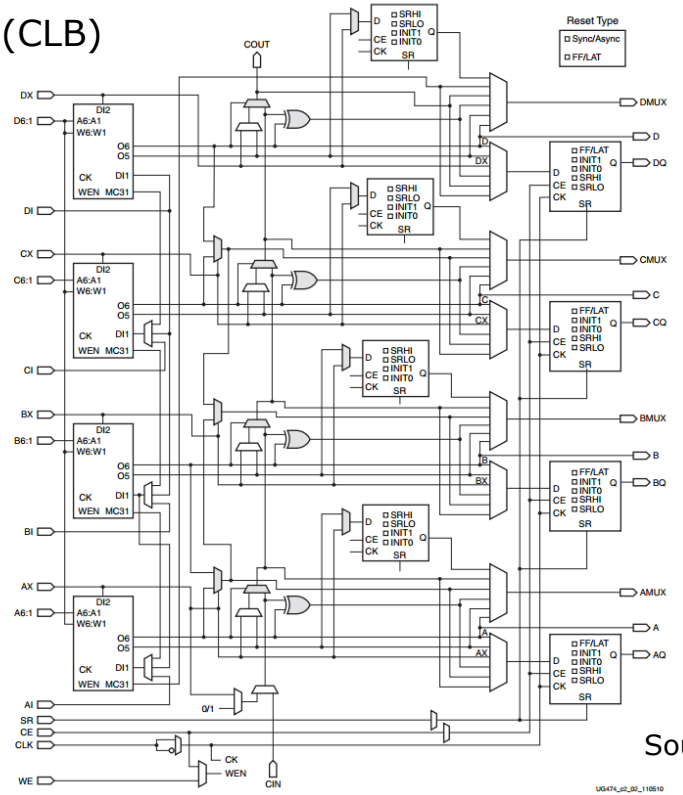
- Field-programmable Gate Array
- 2-dimensional array of reconfigurable logic elements
  - Look-up Tables (LUTs), Flip-flops, Adders, Block RAM (BRAMs), DSP blocks
- Static Programmable Interconnect
  - Switch box, connection box
- Input/Output interface at chip boundary
- Uses:
  - Digital circuit emulation
  - Custom hardware accelerators

Example - FPGAs



Reconfigurable Compute

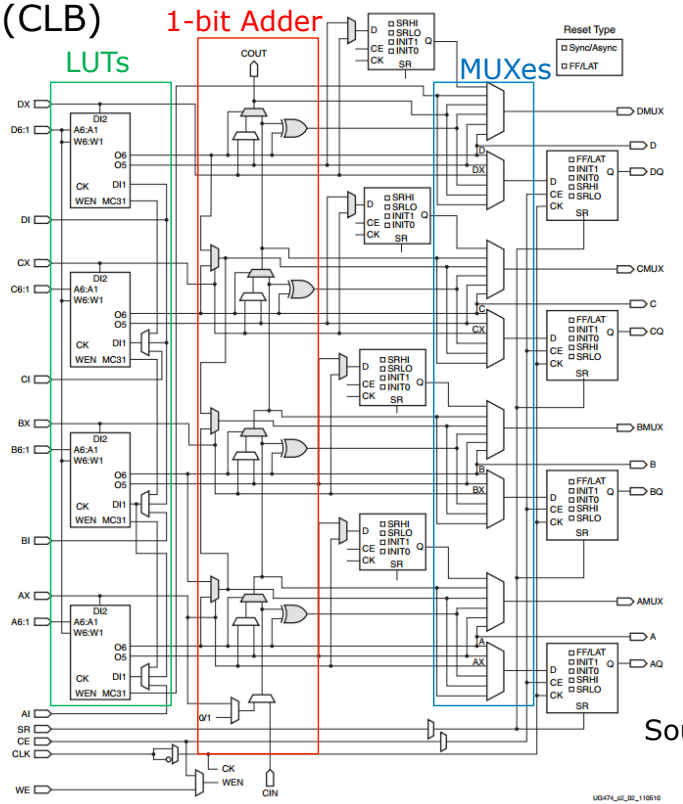
Xilinx Configurable Logic Block (CLB)



Source: 7 Series FPGAs CLB User Guide

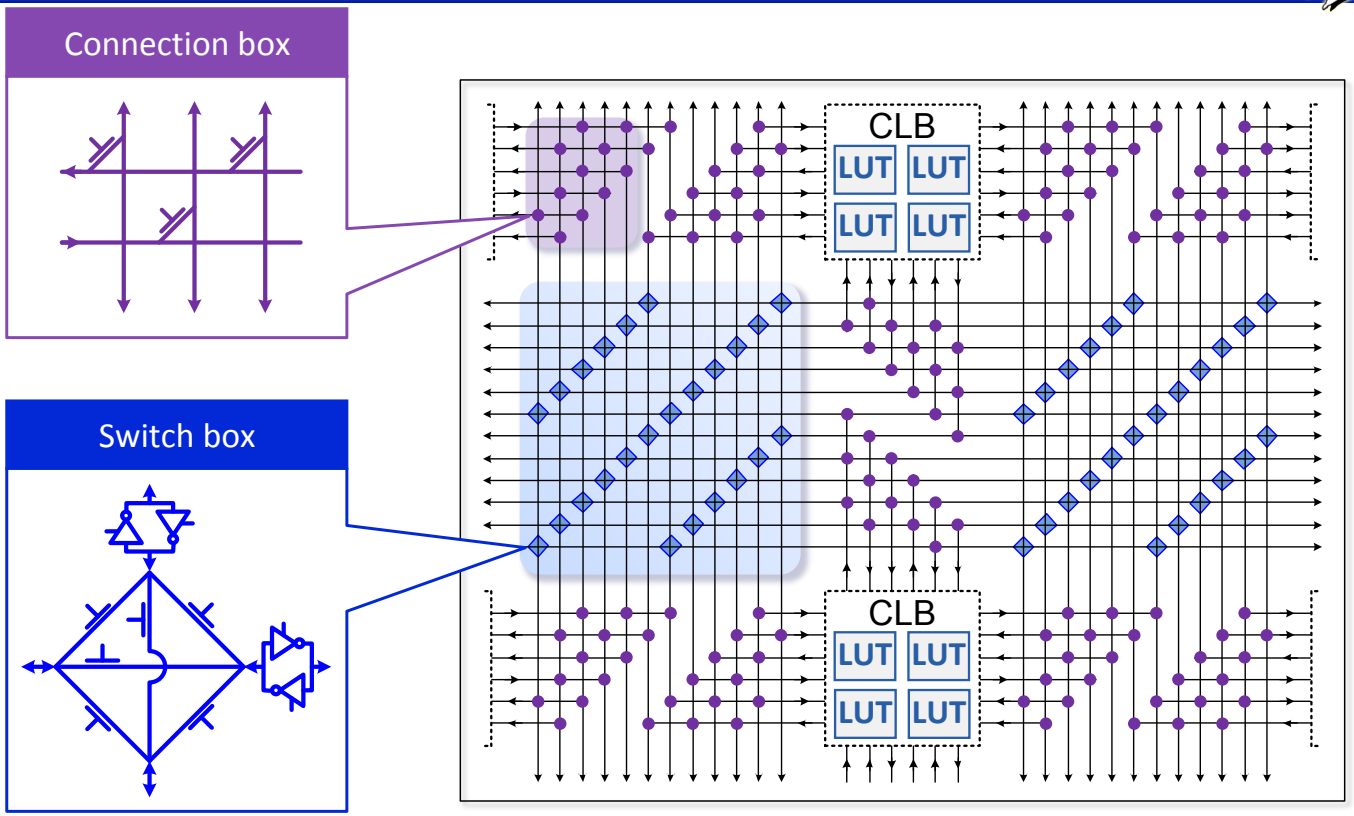
Reconfigurable Compute

Xilinx Configurable Logic Block (CLB)



Source: 7 Series FPGAs CLB User Guide

Static Programmable Interconnect

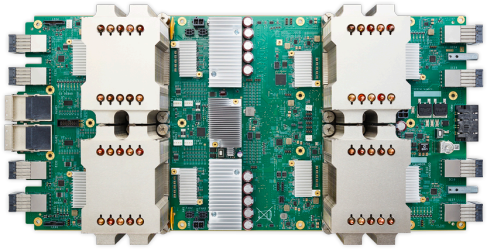


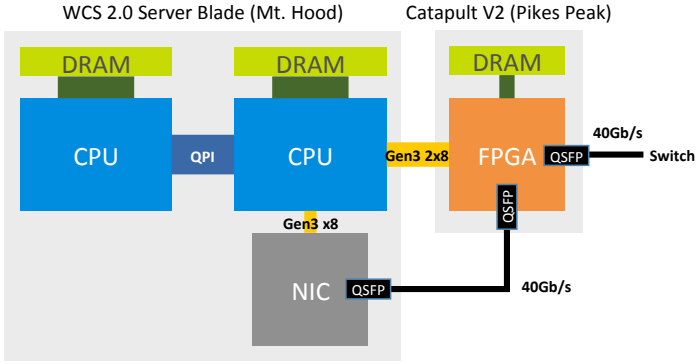
FPGAs – Pros and Cons

- Pros
  - More flexible than custom hardware
  - Cheaper than custom hardware – same chip is reused across designs
  - Can be more efficient than a general-purpose processor, especially Perf/W
  - Great for circuit emulation, validation
- Cons
  - Less efficient than custom hardware
  - Less programmable than general purpose processor
  - Slow programming toolchain (compare compile times for lab2 and lab3)

Acceleration in Industry

- Cloud
  - Google TPU & Cloud TPU
  - Amazon F-1 FPGA instances
  - Azure FPGA
- Mobile
  - Most chips include accelerators
  - Image, video, audio, ML, …





Summary

- Programmable hardware → generality, design reuse
- Custom hardware → efficiency

- How to achieve both
  - Use both programmable & custom hardware
  - Generalize custom hardware for a domain or class of algorithms