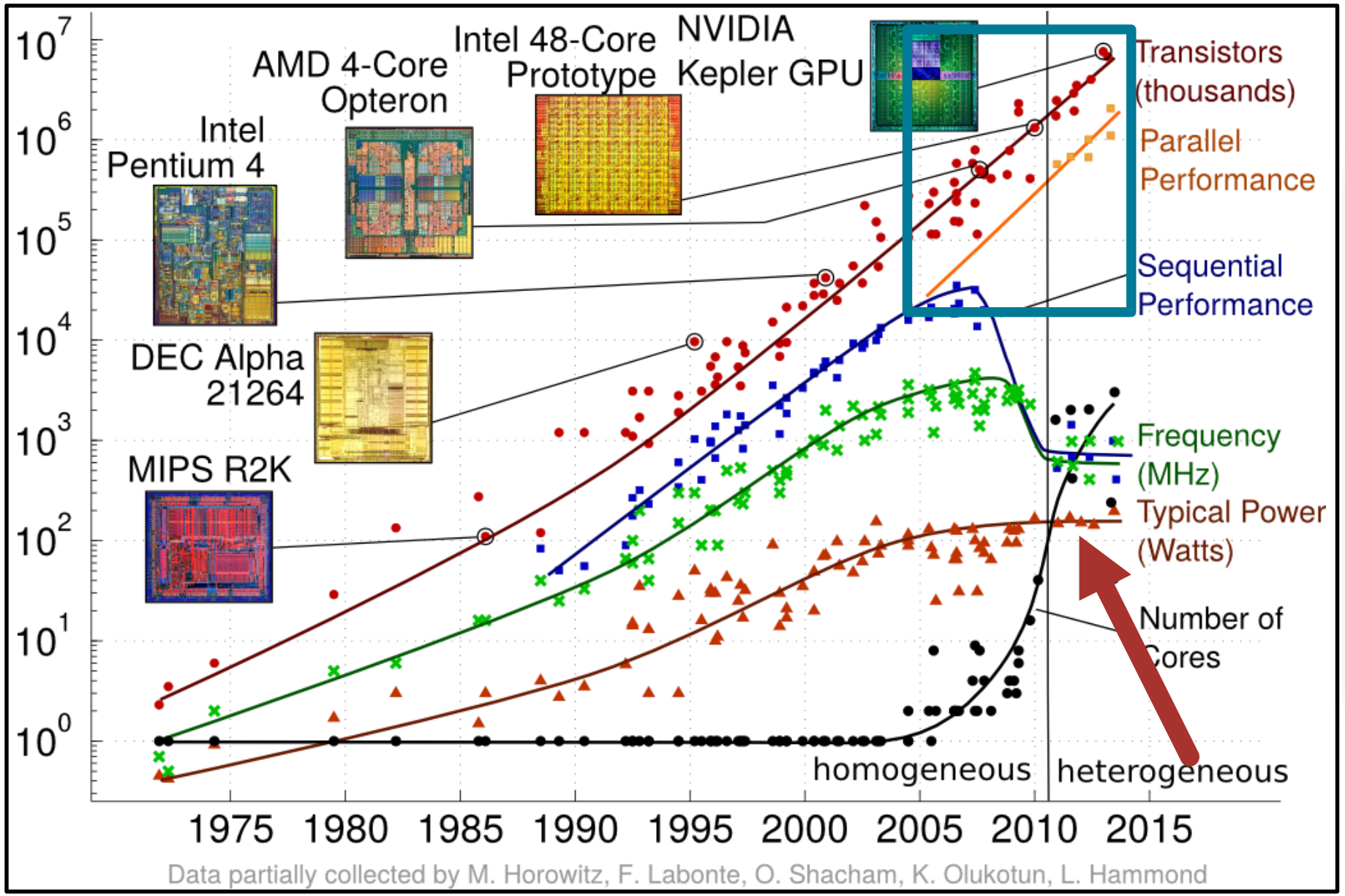
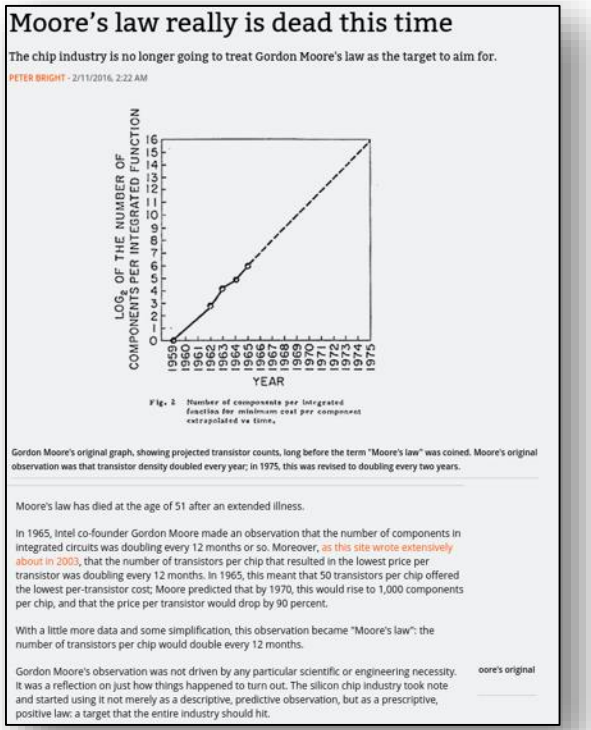
J. de Fine Licht, T. Hoefler

FPGAs and Beyond:

Specialized Hardware Architectures for HPC with High-Level Synthesis





Moore's law really is dead this time

The chip industry is no longer going to treat Gordon Moore's law as the target to aim for.

PETER BRIGHT - 2/11/2016, 2:22 AM

Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Gordon Moore's original graph, showing projected transistor counts, long before the term "Moore's law" was coined. Moore's original observation was that transistor density doubled every year; in 1975, this was revised to doubling every two years.

Moore's law has died at the age of 51 after an extended illness.

In 1965, Intel co-founder Gordon Moore made an observation that the number of components in integrated circuits was doubling every 12 months or so. Moreover, as this site wrote extensively about in 2003, that the number of transistors per chip that resulted in the lowest price per transistor was doubling every 12 months. In 1965, this meant that 50 transistors per chip offered the lowest per-transistor cost; Moore predicted that by 1970, this would rise to 1,000 components per chip, and that the price per transistor would drop by 90 percent.

With a little more data and some simplification, this observation became "Moore's law": the number of transistors per chip would double every 12 months.

Gordon Moore's observation was not driven by any particular scientific or engineering necessity. It was a reflection on just how things happened to turn out. The silicon chip industry took note and started using it not merely as a descriptive, predictive observation, but as a prescriptive, positive law: a target that the entire industry should hit.

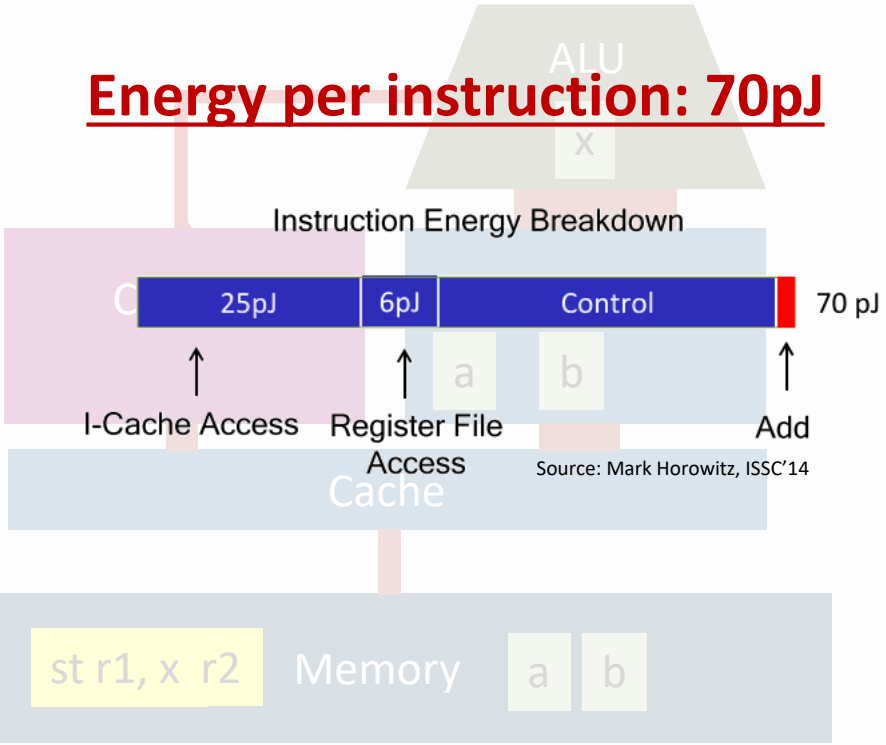
oore's original

Load-store vs. Dataflow

Load-store ("von Neumann")

$$x = a + b$$

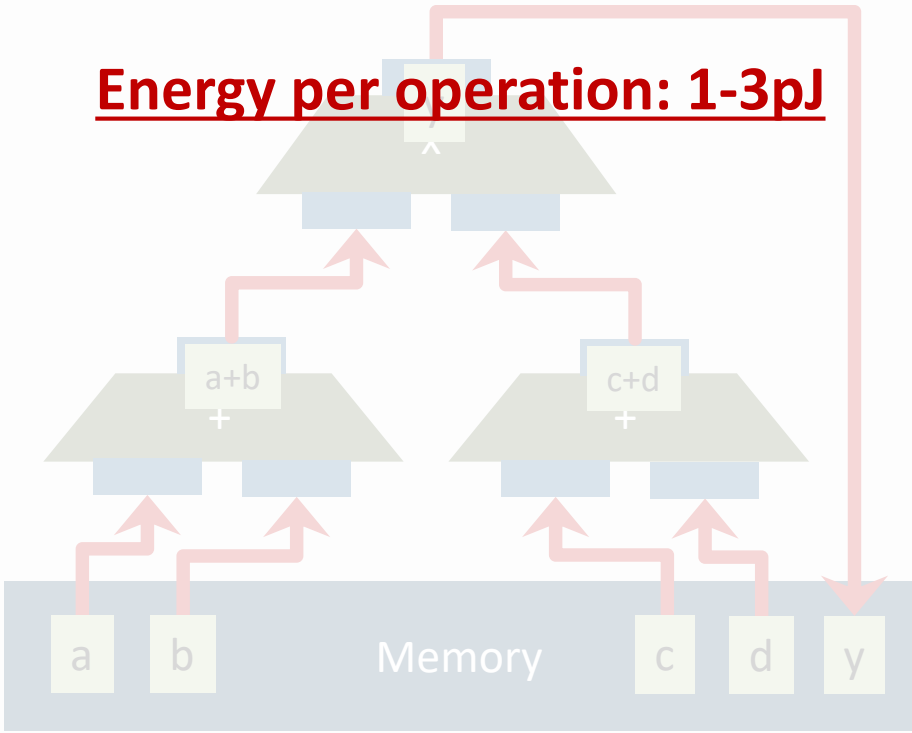
Energy per instruction: 70pJ



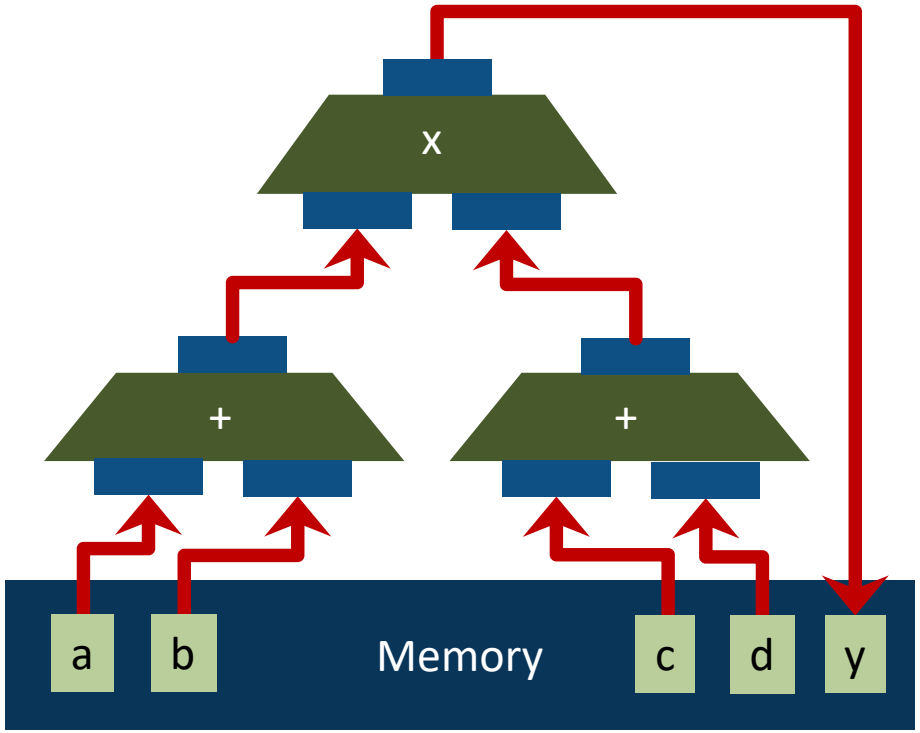
Static Dataflow ("non-von Neumann")

$$y = (a + b) \cdot (c + d)$$

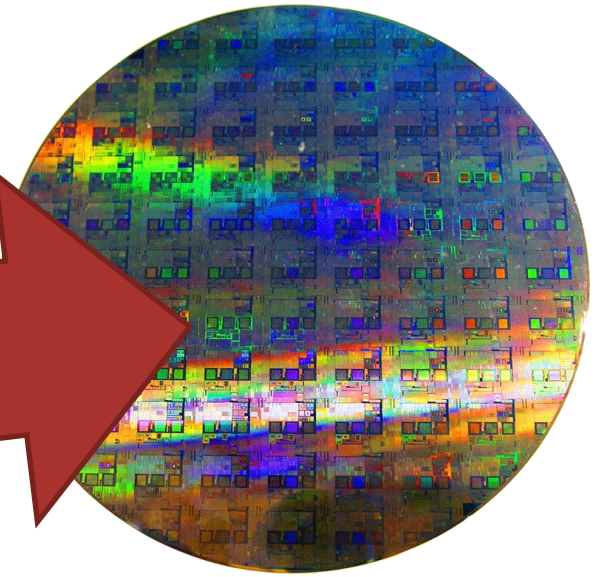
Energy per operation: 1-3pJ



Specialized hardware

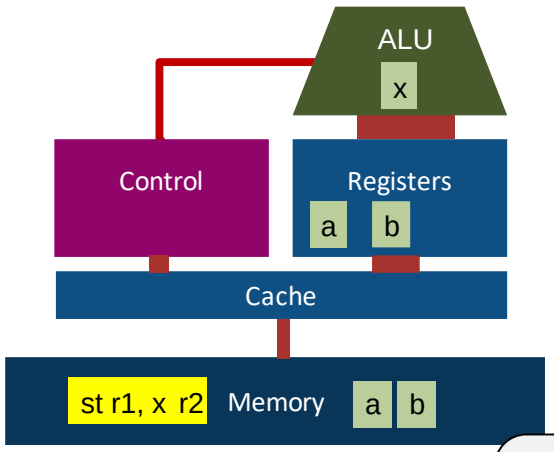


Option 1:
Print an ASIC

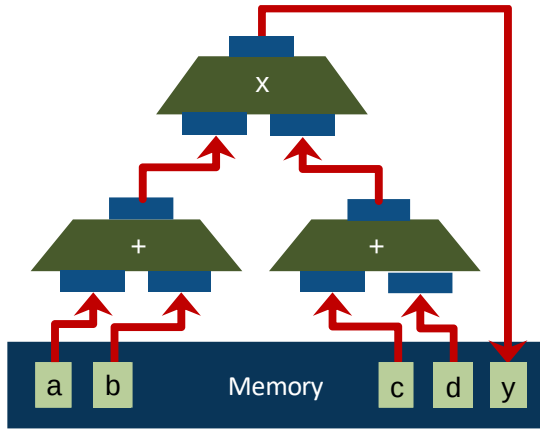


Option 2:
Reconfigurable hardware

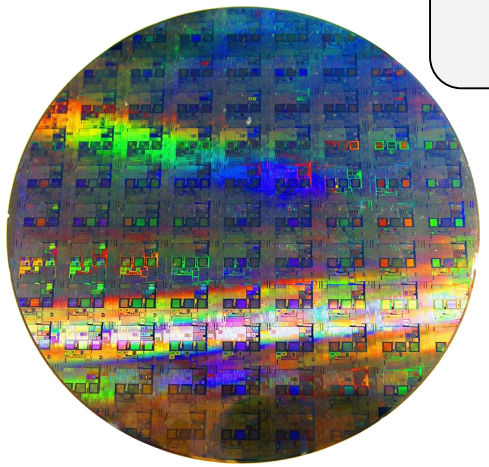
The paradox of FPGA efficiency



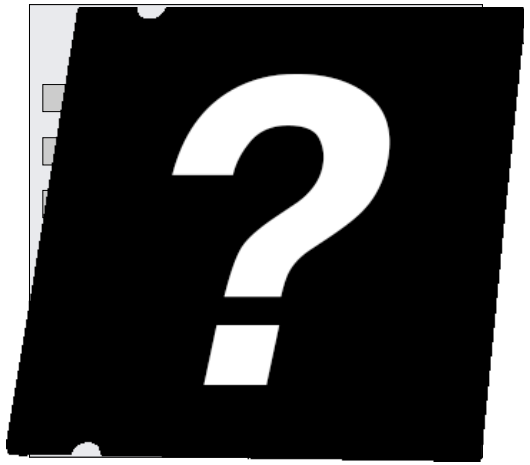
~100x



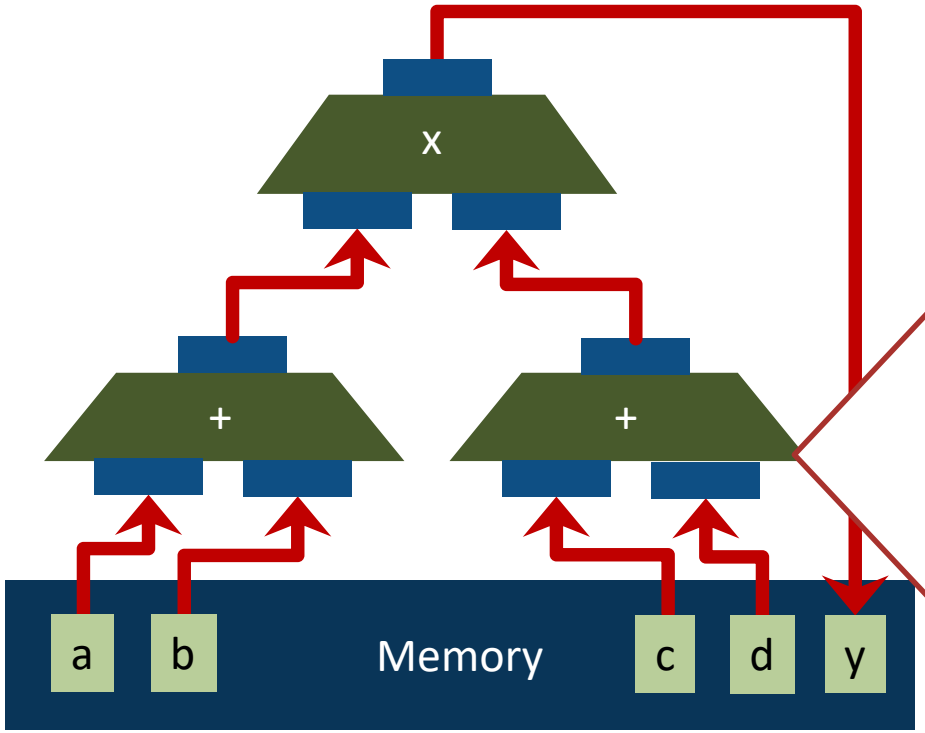
Comes from the *granularity*



?



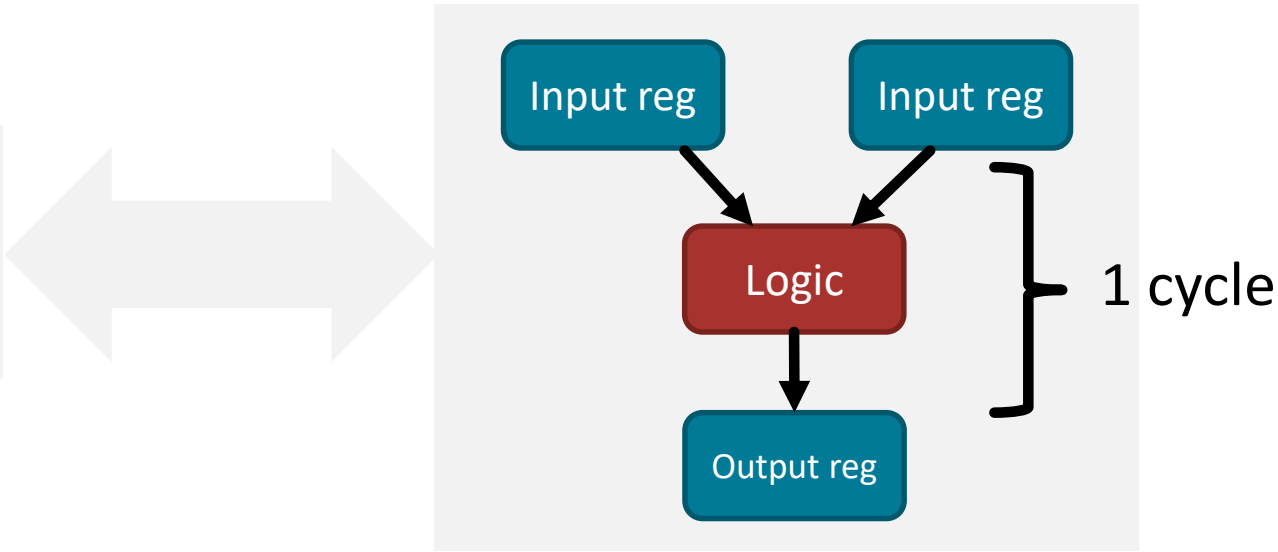
Specialized hardware



We must learn how to program these

Traditionally: register transfer level

```
always @(posedge clk)
  if (start) begin
      out <= in + 1;
  end
```

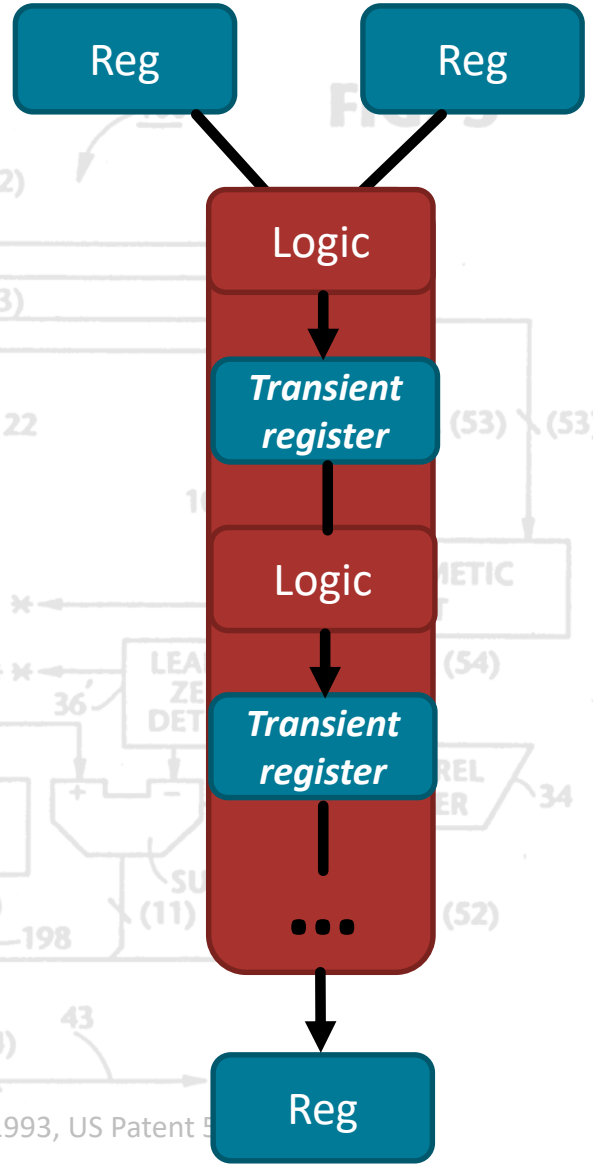



```
int c = a + b;
```

Single floating point operation

FPGA programming is **hard**[1]!

```
float c = a + b;
```



Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5

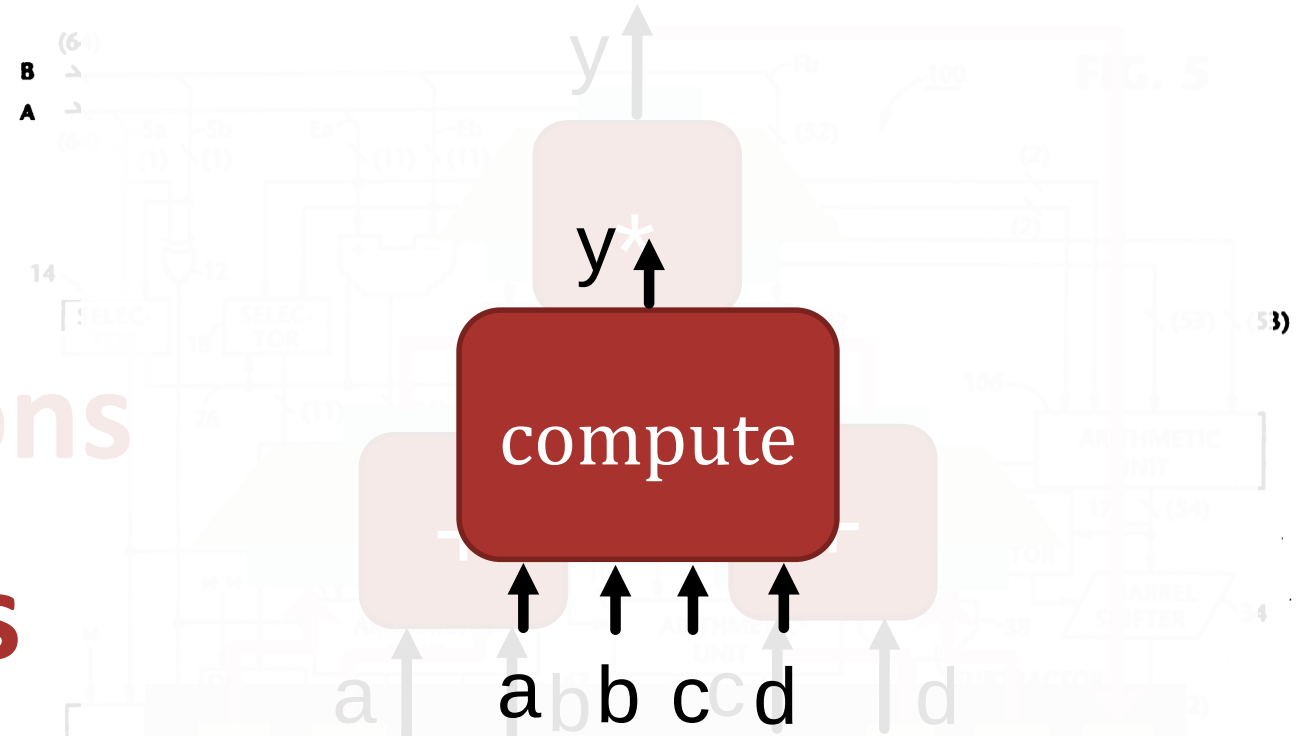
[1] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. Queue 11, 2, Pages 40 (February 2013), 13 pages.

High-level synthesis

Circuits

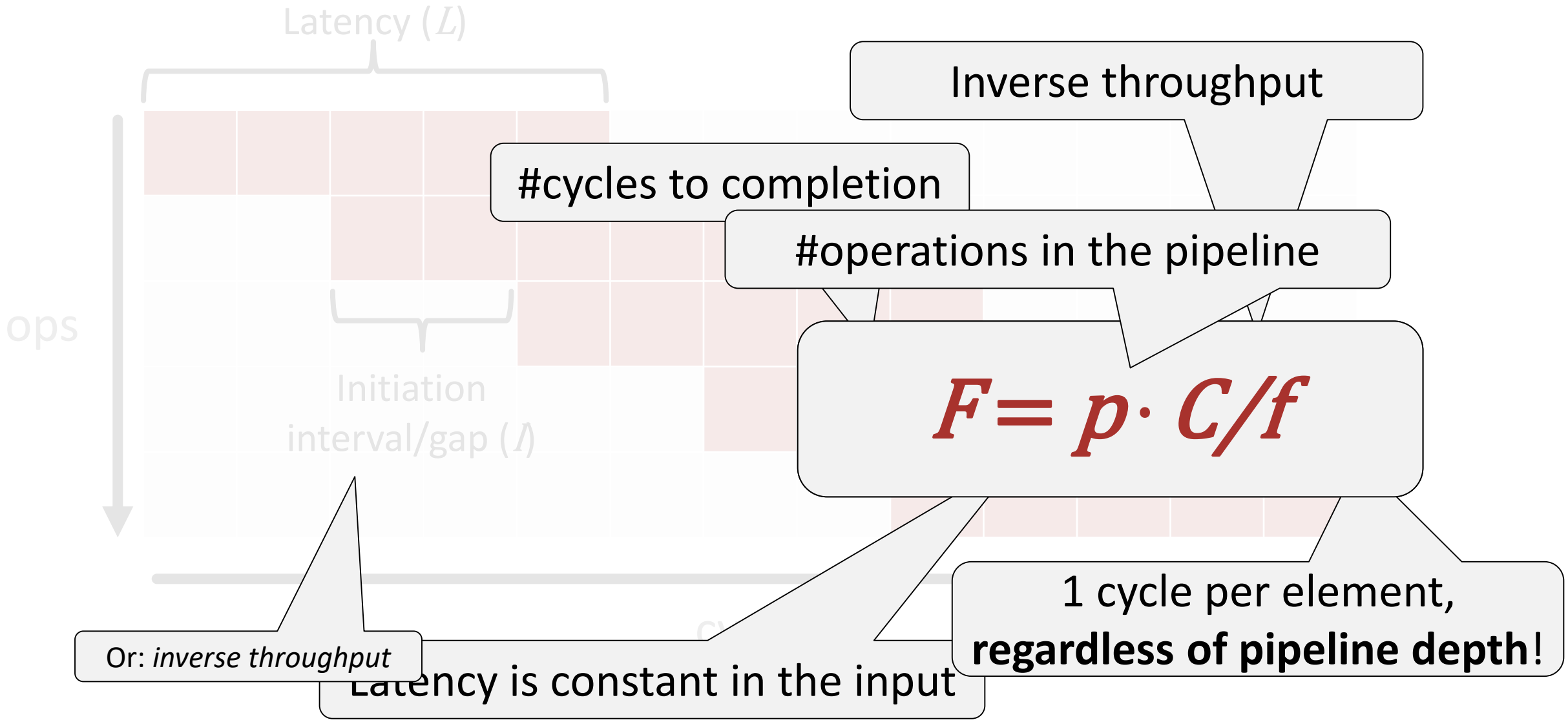
Operations

Pipelines



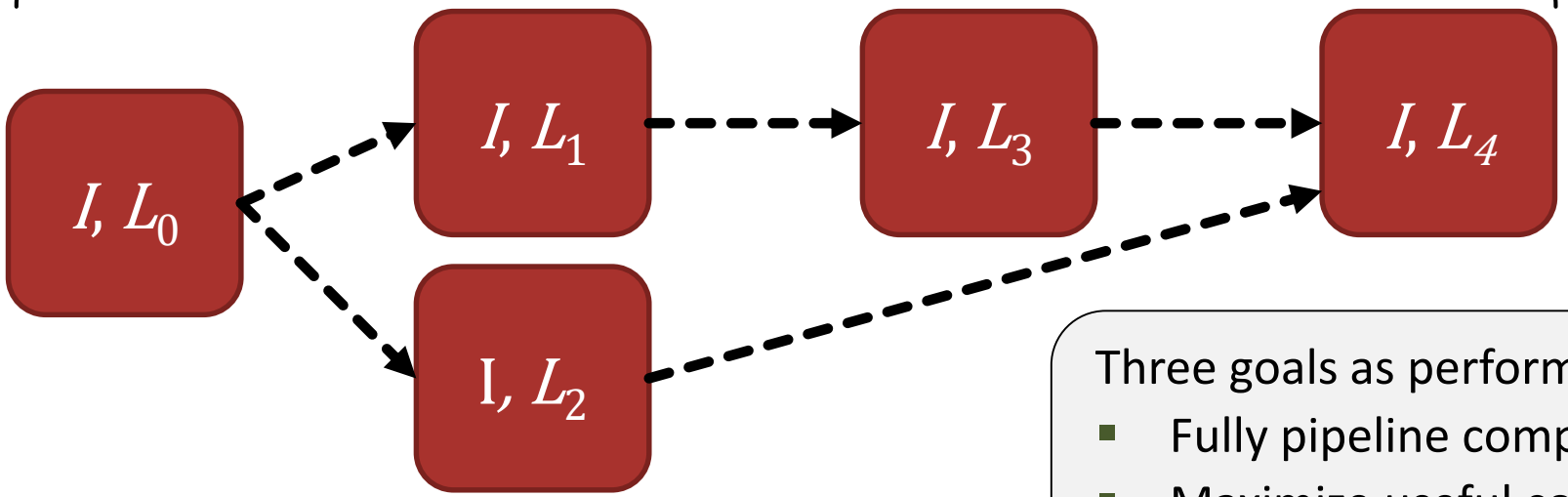

```
float y = (a + b) * (c + d);
```

Pipeline performance



No matter how deep the pipeline is a new result is produced **every cycle**

$$C = I\,N + L_0 + L_1 + L_3 \quad \approx I\,N \quad \text{for} \quad N \gg L$$



Three goals as performance engineers[1]:

- Fully pipeline compute (i.e., $I = 1$)
- Maximize useful computations **p**
- Saturate the pipeline

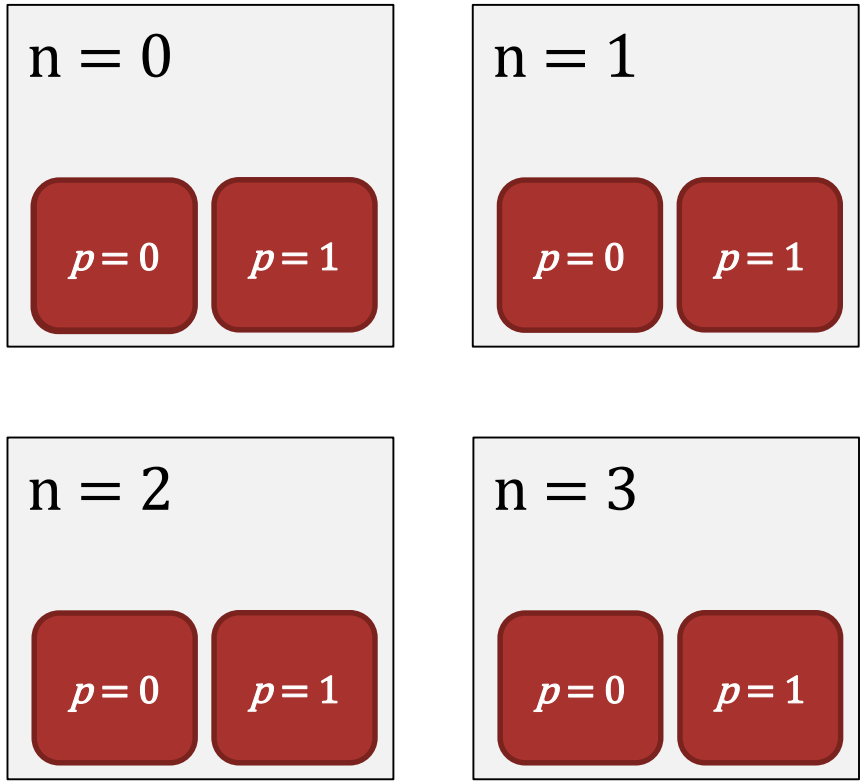
[1] de Fine Licht et al., "Transformations of High-Level Synthesis Codes for High-Performance Computing." *arXiv preprint arXiv:1805.08288* (2018).

End goal: peak!

Parallelizing hardware in HLS

for_parallel $n \leftarrow 1$ **to** N **do**
  for_pipelined $m \leftarrow 1$ **to** M **do**
    for_parallel $p \leftarrow 1$ **to** P **do**
      $C[n,p] \leftarrow C[n,p] + A[n,m] \cdot B[m,p]$

This is now the **full runtime** of the algorith



Parallel loops are **removed** from the iteration space!

$$C \approx IM$$

$$F = fNP$$

Performance corresponds to parallel hardware

Parallelizing hardware in HLS

Hardware optimization vs. software optimization?

Scalability transformations

- Vectorization
- Replication
- Streaming dataflow

- Accumulation interleaving
- Cyclic buffering
- Pipeline colaescing
- ...

- Condition flattening
- Type demotion
- ...

We can implement massively parallel specialized hardware with HLS[1]!

...but, like GPU-programming, we must be architecture-aware.

[1] de Fine Licht et al., "Transformations of High-Level Synthesis Codes for High-Performance Computing." *arXiv preprint arXiv:1805.08288* (2018).

Hardware scaling

For HLS, the base case is **bad** ☹

	Perf. [GOp/s]	Speedup Relative	Speedup Cumulative
Naive	0.02	1×	–
Buffered [§2.5]	0.8	40×	–
Vectorized [§3.1, §4.2, §4.3, §4.4]	6.4	8×	320×
Replicated [§3.2, §3.3, §3.4]	227.8	36×	11,400×

[2D stencil]

	Perf. [GOp/s]	Speedup Relative	Speedup Cumulative
Naive	0.01	1×	–
Fused [§2.1, §2.6, §2.7, §4.2]	0.4	40×	–
Vectorized [§3.1]	3.2	8×	320×
Replicated [§3.2, §3.3, §3.4]	184.1	58×	18,410×

[Matrix multiplication]

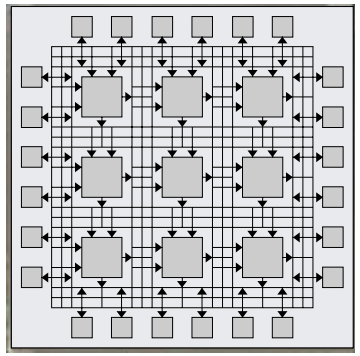
Luckily, there are patterns ☺

	Perf. [GOp/s]	Speedup Relative	Speedup Cumulative
Initial [§4.2, §4.3]	0.9	1×	–
Interleaved [§2.2.1]	6.0	7×	–
Replicated [§3.2, §3.3]	231.9	39×	258×

[N-body simulation]

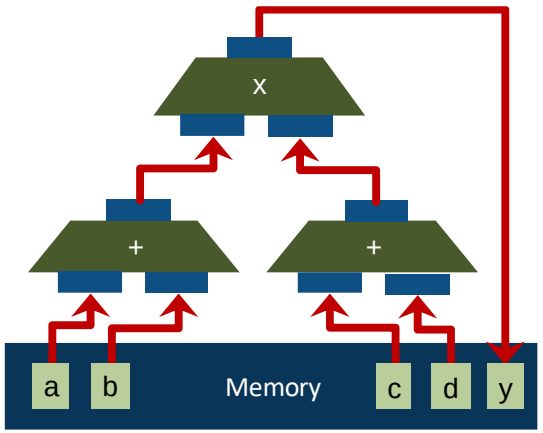
From de Fine Licht et al., "Transformations of High-Level Synthesis Codes for High-Performance Computing." *arXiv preprint arXiv:1805.08288* (2018).

HLS for FPGAs...

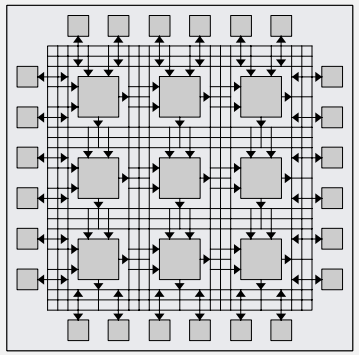


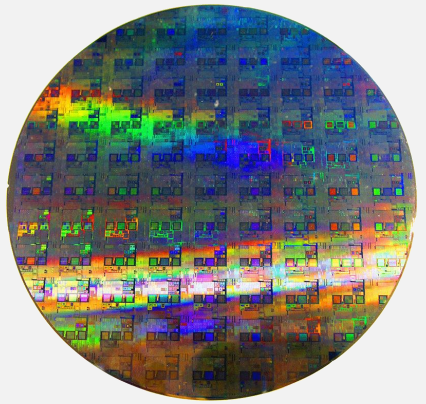
for_parallel $n \leftarrow 1$ **to** N **do**
  for_pipelined $m \leftarrow 1$ **to** M **do**
    for_parallel $p \leftarrow 1$ **to** P **do**
      $C[n,p] \leftarrow C[n,p] + A[n,m] \cdot B[m,p]$

...and beyond?



on





HLS++

Thank you for your attention*!

*For more, see:

”Transformations of High-Level Synthesis Codes for High-Performance Computing” [arXiv 1805.08288]

ETH zürich