

# **CMPT 450/750: Computer Architecture**

## **Fall 2021**

### **Domain-Specific Architecture I**

**How did we get here ?**

**What are they ?**

*Alaa Alameldeen & Arrvindh Shriraman*



# Hardware Microbrewery ?

DSAs



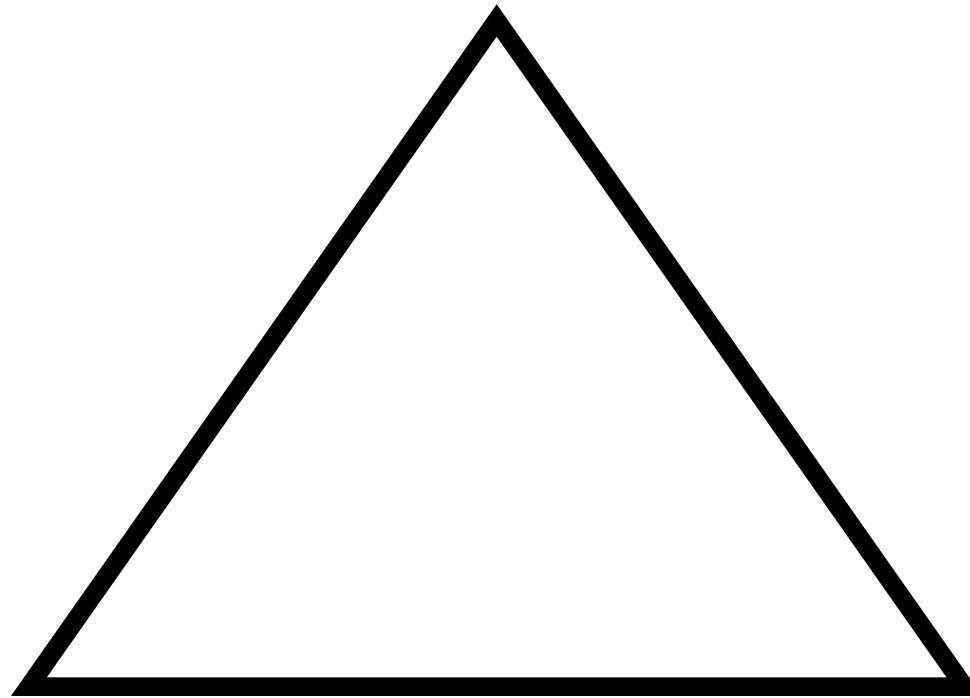
Out-of-Order Processors





**Abstractions/IR**

i.e., Does algorithm/kernel/app even have branches?



**Technology**

i.e., How many ns for table of size 1KB?  
What is the energy?

**Architecture**

i.e., What branch-predictor?  
How to organize tables?



The experiment is performed in a 4m x 6m room which includes:

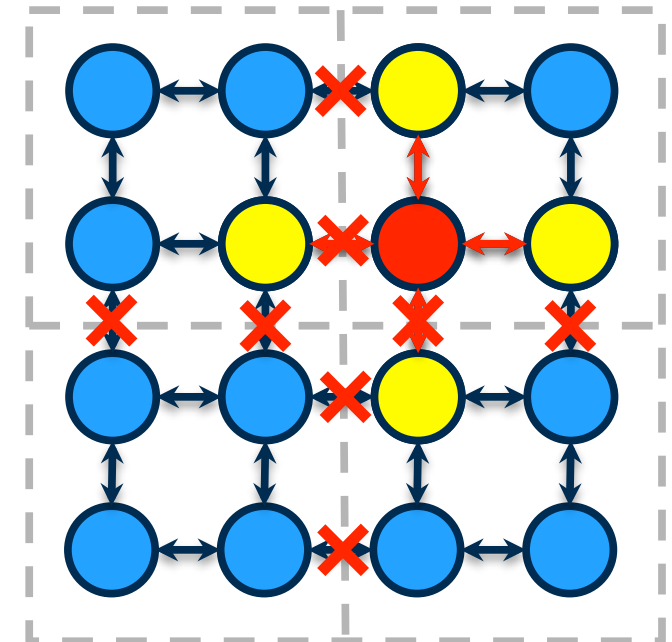
- 2 rectangle boxes as obstacles
- A robot car

One obstacle is static, while the other obstacle is constantly moved by a person.



Obstacles are tracked and localized with a Vicon system. Their positions are sent to an FPGA in order to compute a new value function.

Domain Pattern









# Google's TPU – DSAs in 2015

SFU



- **8-bit**  
(Intel 8080)
- **Systolic**  
(first pass)
- **24MB**  
(No cache)

>100X as many MACs vs CPU ■  
3.5X on-chip memory vs GPU  
80x energy efficiency vs CPU  
30x better performance vs CPU

Not to Scale



Information lost necessitating more complex hardware

---

### PYTHON

```
np.add(arr1, arr2)
```

### C/C++

```
for(i = 0; i < n; i++)  
    res[i] = arr1[i] + arr2[i]
```

### ISA

```
.Loop:  
lw    a5, 0(a2)    # *(arr1+i)  
lw    a6, 0(a3)    # *(arr2+i)  
add    a0, a5, a6  
sw    a0, 0(a4)  
# Bump pointers.  
addi   a2, a2, 4  
addi   a3, a3, 4  
addi   a4, a4, 4  
addi   a1, a1, 1  
bne    a1, a3, loop
```



Information lost necessitating more complex hardware

---

PYTHON

`np.add(arr1, arr2)`

C/C++

```
for(i = 0; i < n; i++)
    res[i] = arr1[i] + arr2[i]
```

ISA

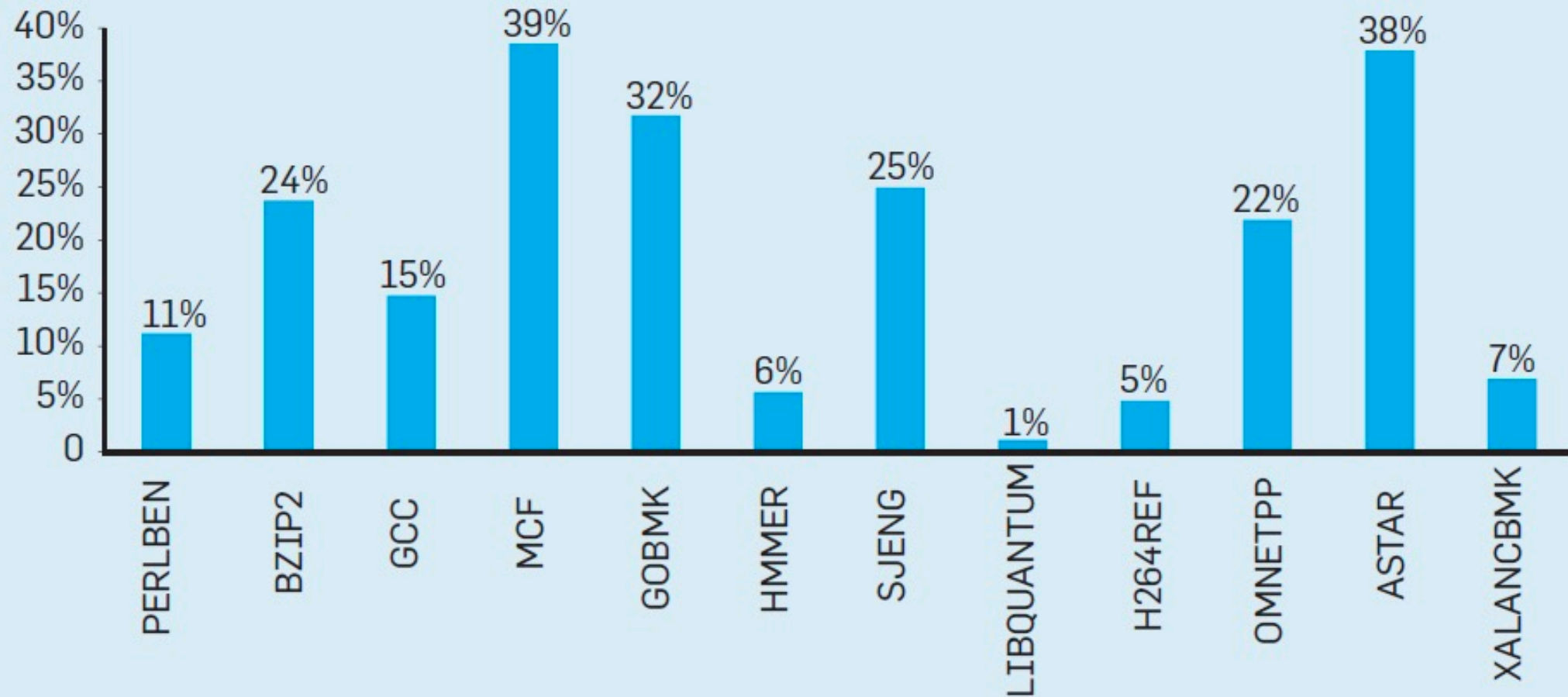
```
.Loop:
lw    a5, 0(a2)      # *(arr1+i)
lw    a6, 0(a3)      # *(arr2+i)
add   a0, a5, a6     # Global reg
sw    a0, 0(a4)
# Bump pointers.
addi  a2, a2, 4
addi  a3, a3, 4
addi  a4, a4, 4
addi  a1, a1, 1
bne   a1, a3, loop
```

Load/Store  
Queues

Branch  
Predictor to  
find loop parallelism



# Wasted instructions





# Why ISAs suck ?

```
#pragma clang unroll_count(10)
for(int i = 0; i < 10; i++)
    res[i] = arr1[i] + arr2[i];
}
```

```
res[0] = arr1[0] + arr2[0];
res[1] = arr1[1] + arr2[1];
...
res[9] = arr1[9] + arr2[9];
```

```
lw a6, 0(a0)
lw a4, 0(a1)
lw a5, 4(a0)
lw a3, 4(a1)
add a4, a4, a6
sw a4, 0(a2)
add a6, a3, a5
lw a7, 8(a0)
lw a5, 8(a1)
lw a3, 12(a0)
lw a4, 12(a1)
sw a6, 4(a2)
add a5, a5, a7
sw a5, 8(a2)
add a6, a4, a3
lw a7, 16(a0)
lw a5, 16(a1)
lw a3, 20(a0)
lw a4, 20(a1)
```



# Why ISAs suck ?

Register naming introduced dependencies

Need register  
renaming hardware

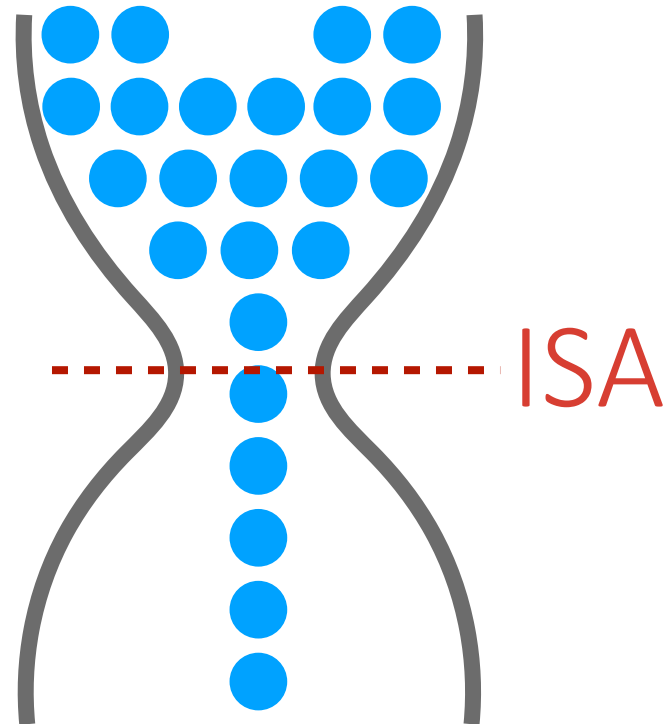
```
#pragma clang unroll_count(10)
for(int i = 0; i < 10; i++)
    res[i] = arr1[i] + arr2[i];
}
```

```
res[0] = arr1[0] + arr2[0];
res[1] = arr1[1] + arr2[1];
...
res[9] = arr1[9] + arr2[9];
```

```
lw a6, 0(a0)
lw a4, 0(a1)
lw a5, 4(a0)
lw a3, 4(a1)
add a4, a4, a6
sw a4, 0(a2)
add a6, a3, a5
lw a7, 8(a0)
lw a5, 8(a1)
lw a3, 12(a0)
lw a4, 12(a1)
sw a6, 4(a2)
add a5, a5, a7
sw a5, 8(a2)
add a6, a4, a3
lw a7, 16(a0)
lw a5, 16(a1)
lw a3, 20(a0)
lw a4, 20(a1)
```



# Why ISAs suck ?





**Why OOOs suck.**

**Is technology scaling dead/dying ?**

**Are DSAs/Accelerators The Solution?**



**Why OOOs suck.**

**Is technology scaling dead/dying ?**

**Are DSAs/Accelerators The Solution?**



# We had it all figured out!

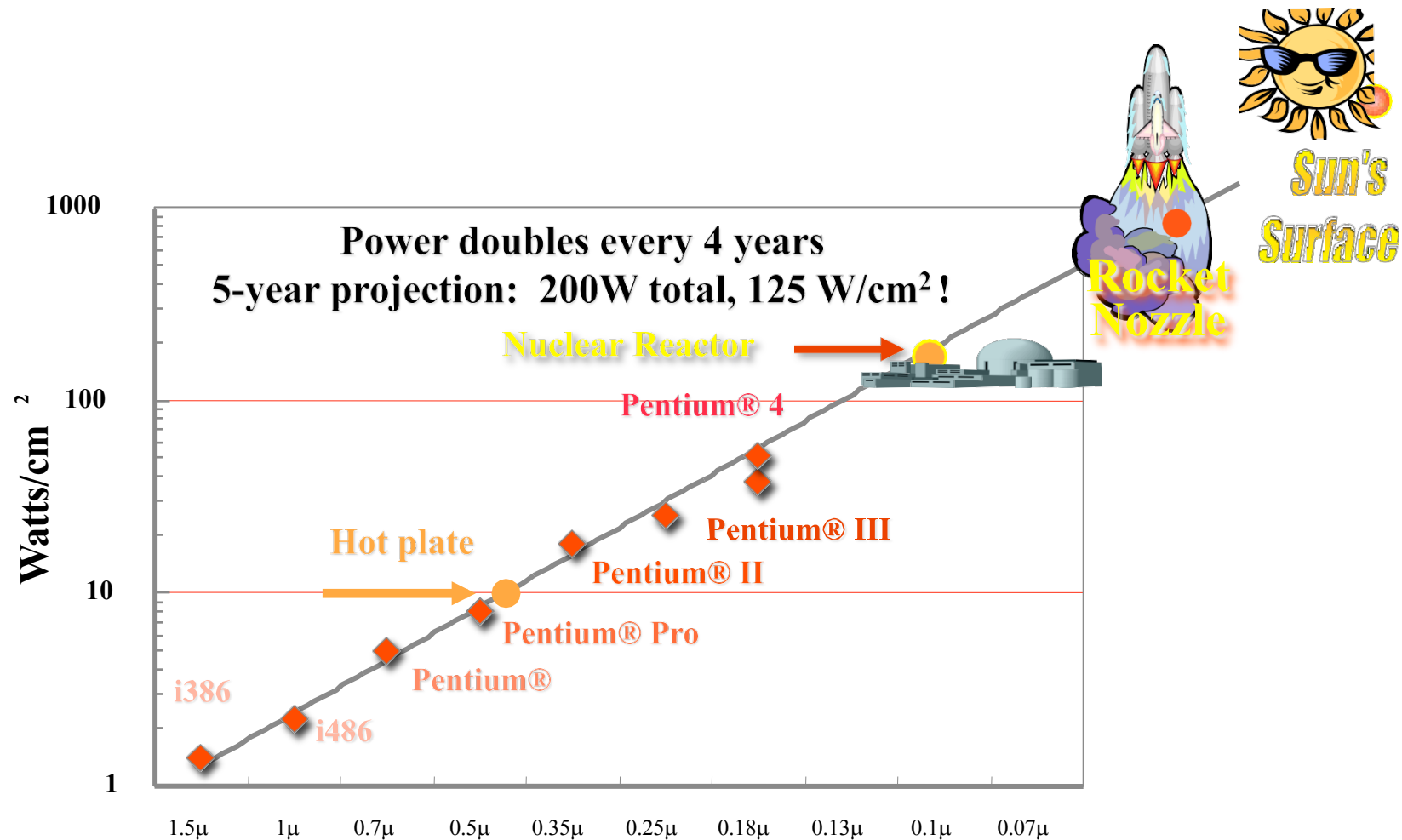
## ISCA 2002 Session I:

- The Optimum Pipeline Depth for a Microprocessor  
IBM (22-36 pipeline stages)
- The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays (~40 pipeline stages)  
Dec/Compaq/HP
- Increasing Processor Performance by Implementing Deeper Pipelines (~50-60 stages)  
Intel

Universal Conclusion: Frequency-Boosted Microarch == Future

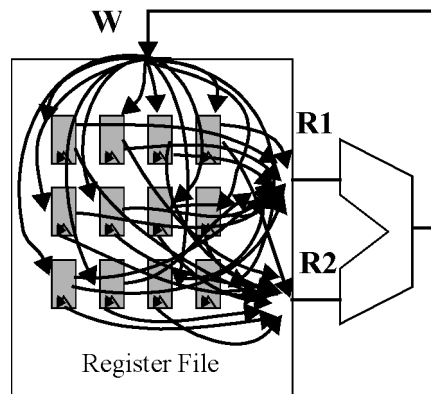


# Oops!

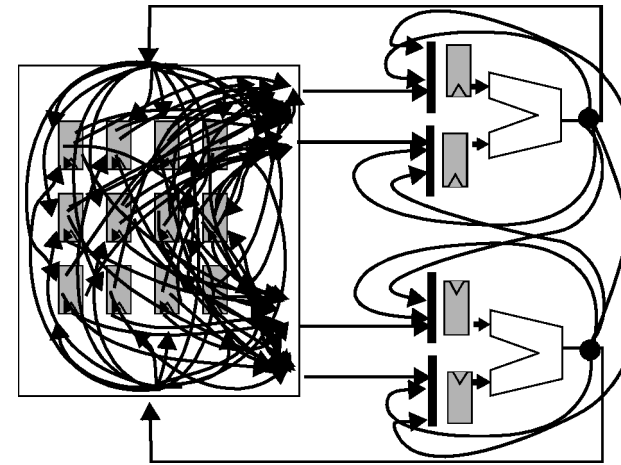


From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies"  
– Fred Pollack, Intel Corp. Micro32 conference key note - 1999.



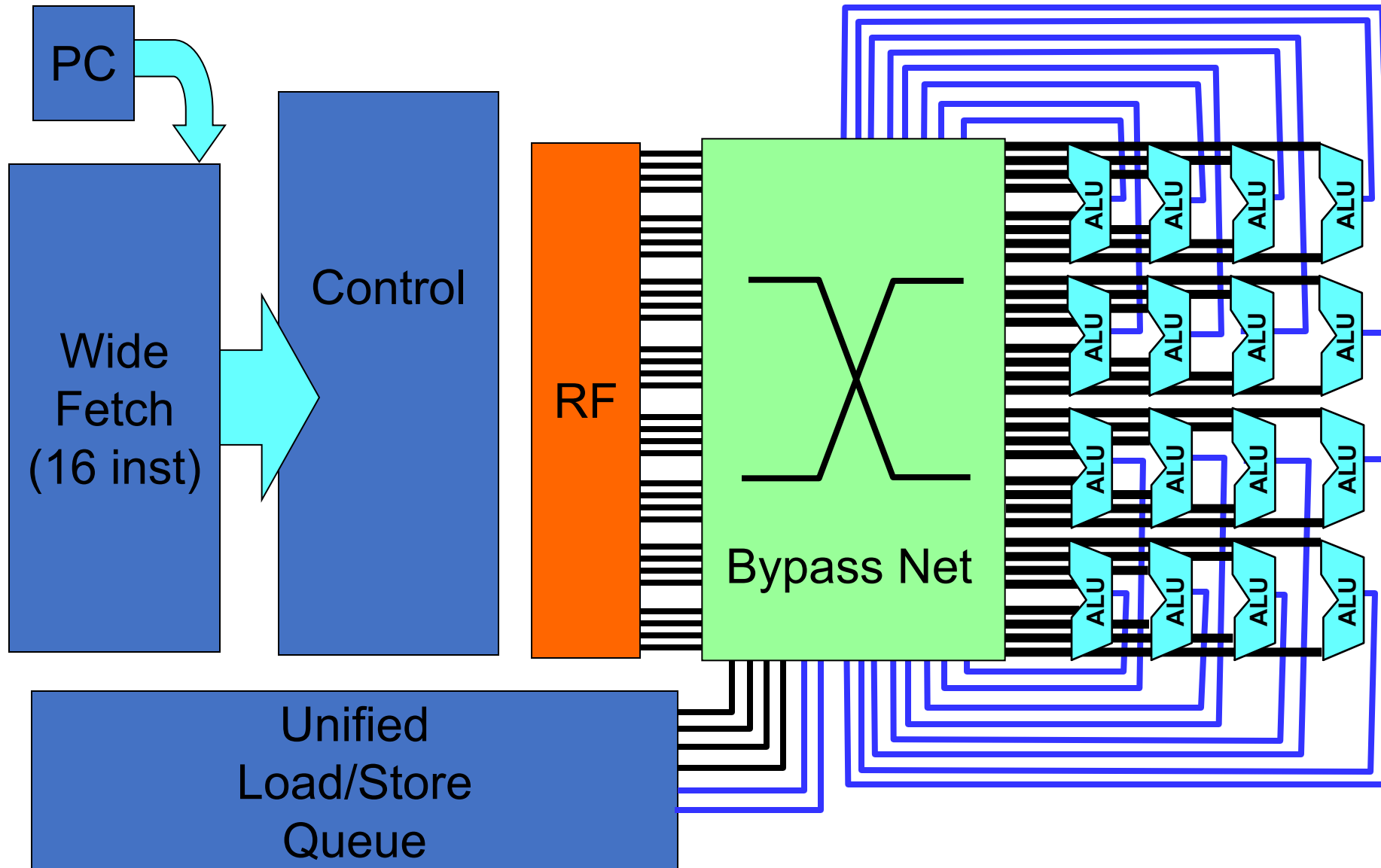


mul \$2,\$3,\$4  
add \$6,\$5,\$2



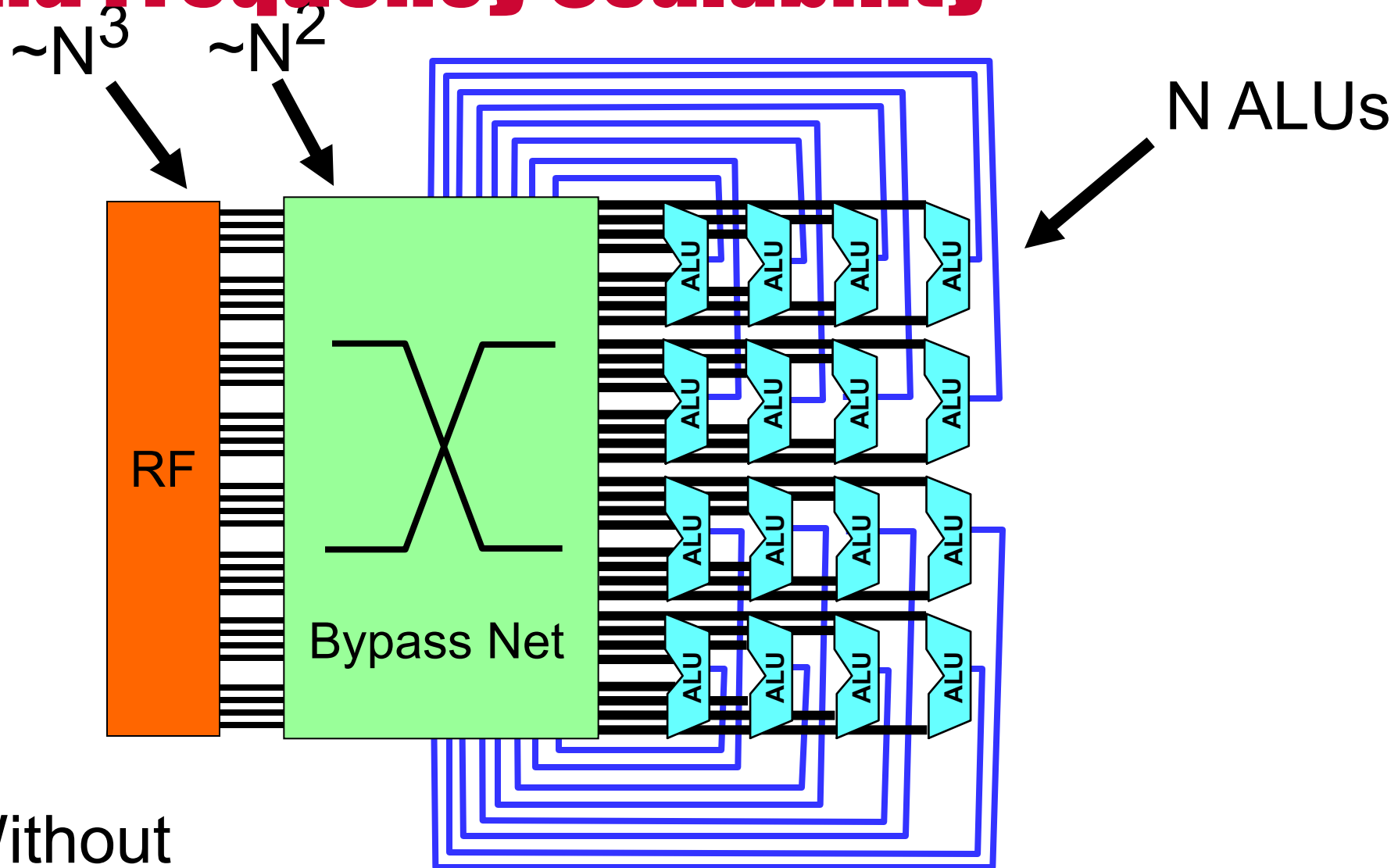
What's great about superscalar microprocessors? →  
**Fast low-latency tightly-coupled networks**  
(0-1 cycles of latency, no occupancy)







# Area and Frequency Scalability

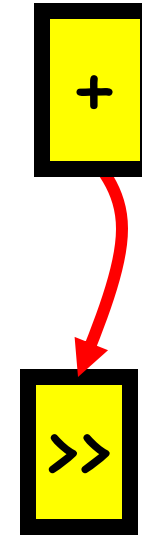
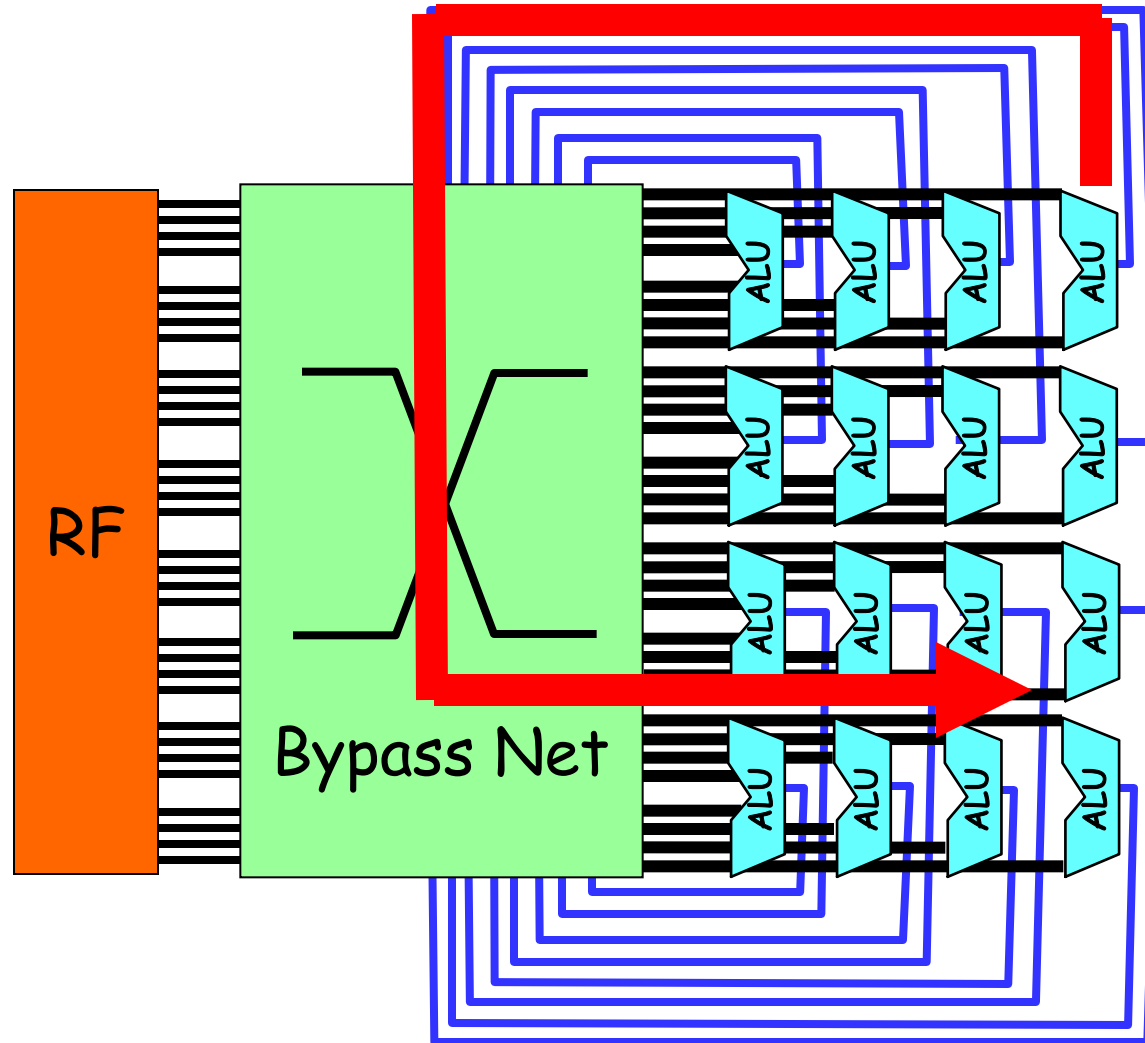


Without  
modification, freq decreases linearly or worse.



# Global Operand Routing

SFU





# Back to the future ...

**PPro/P3:**

**12 stages**

**P4 (b4 paper):**

**20 stages**

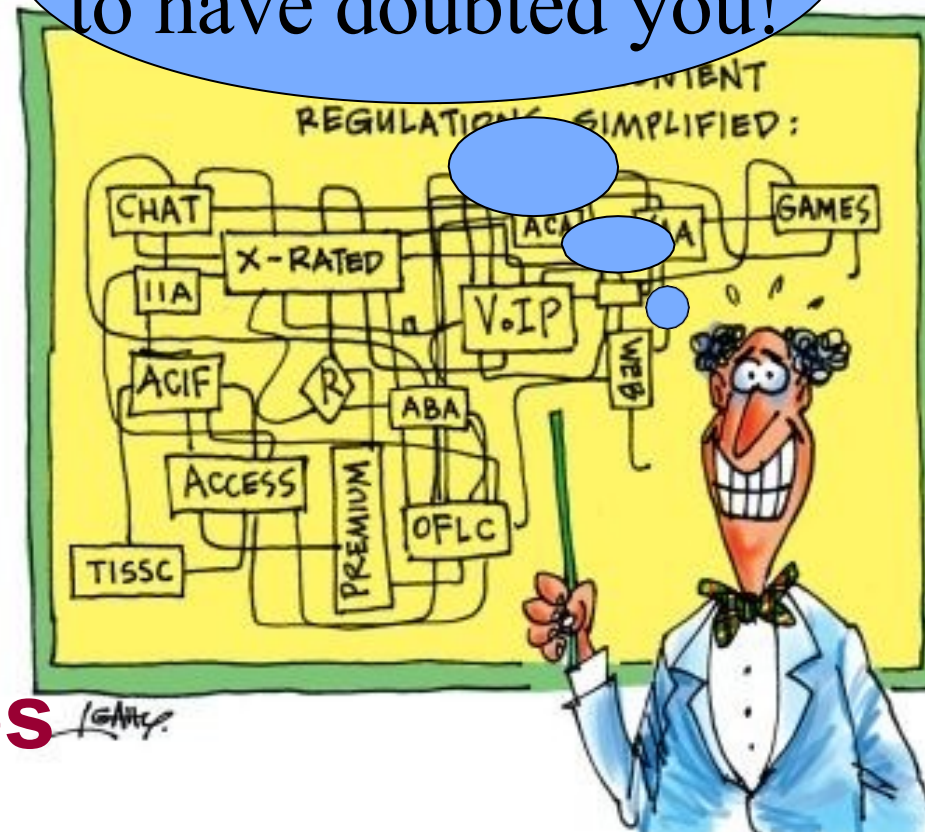
**P4/prescott:**

**31 stages**

**P5/Tejas:**

**>> 31 stages**

Oh P Pro, I'm sorry  
to have doubted you!





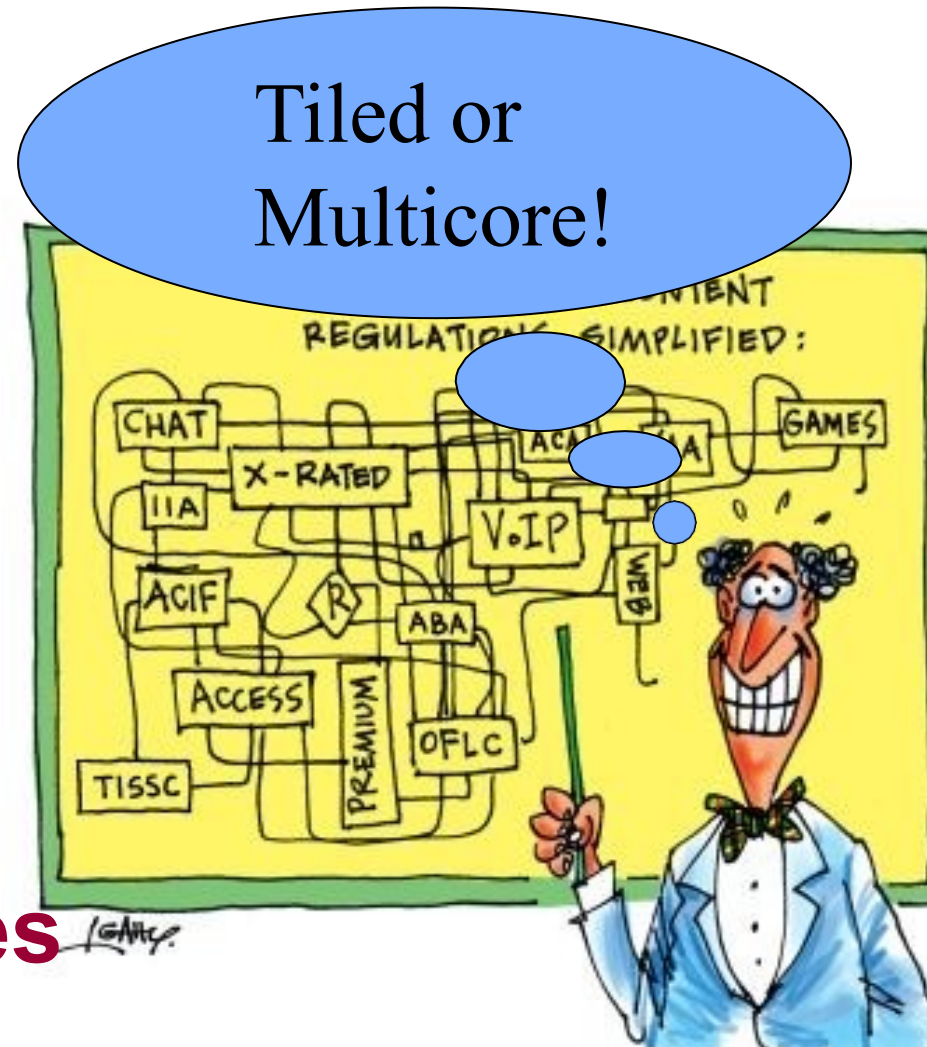
## And forward to multicore...

**PPro/P3:**  
**12 stages**

**P4 (b4 paper):**  
**~~20 stages~~**

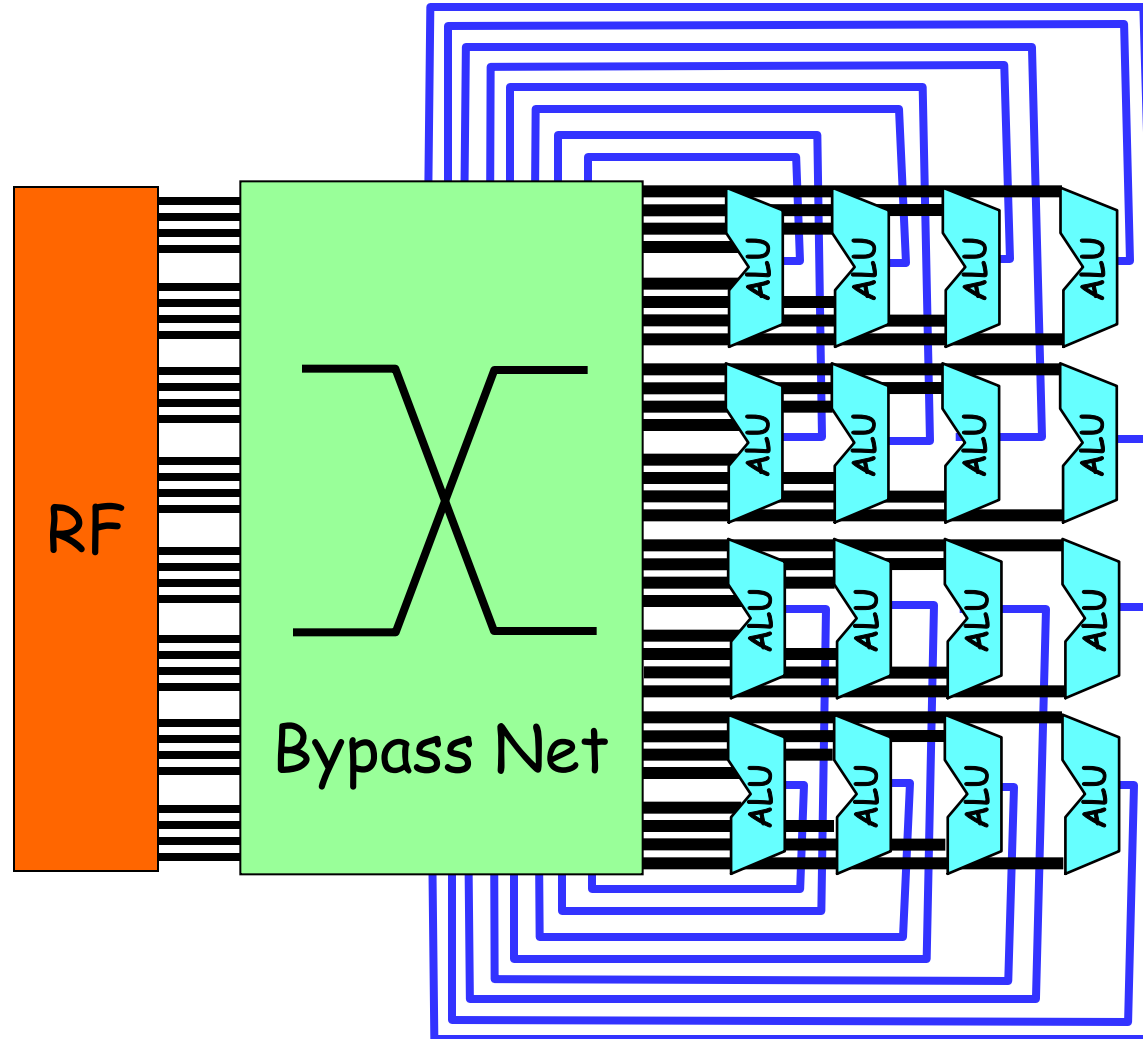
**P4/prescott:**  
**~~31 stages~~**

**P5/Tejas:**  
**~~>> 31 stages~~**





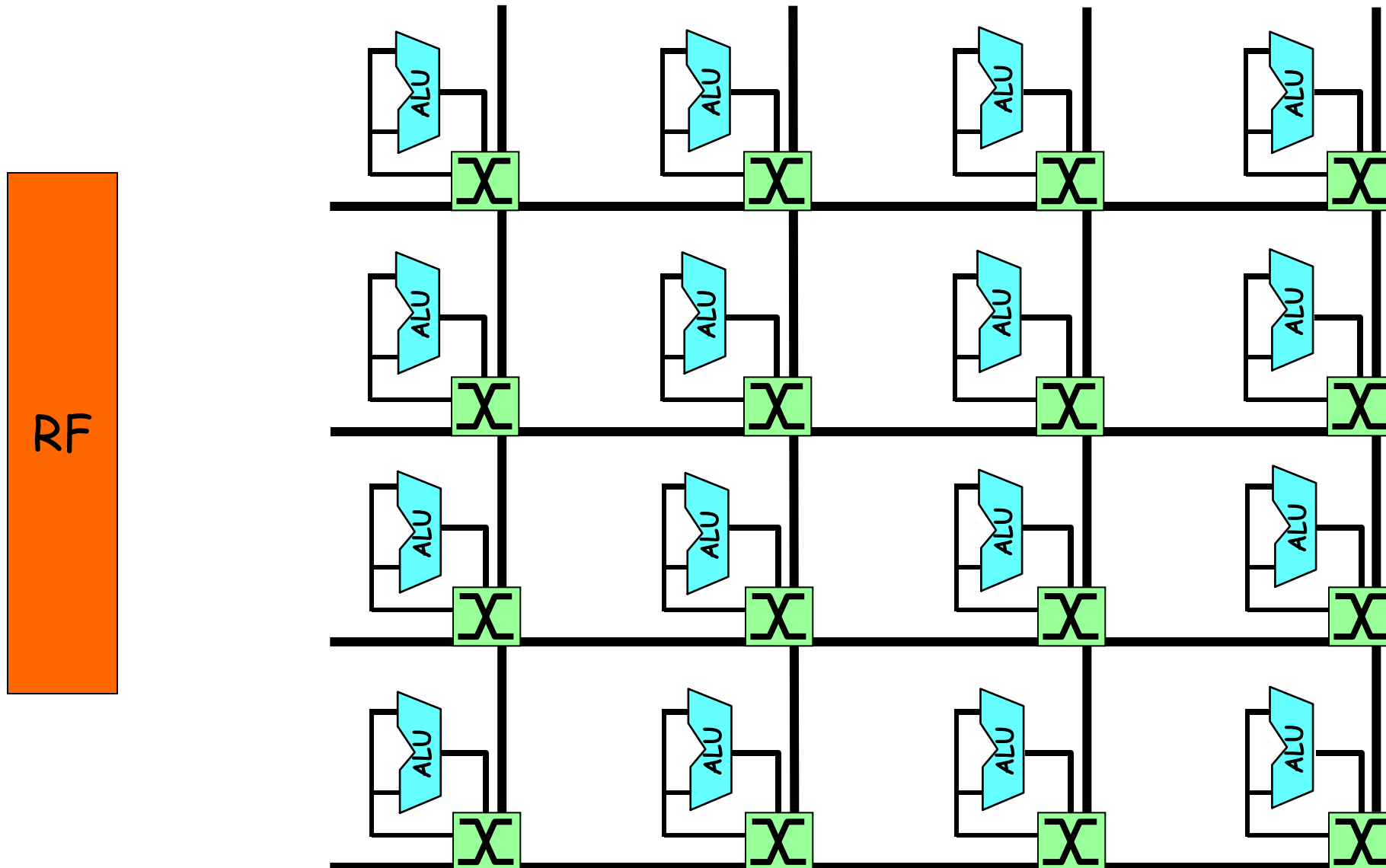
# Idea 1: Make operand routing local



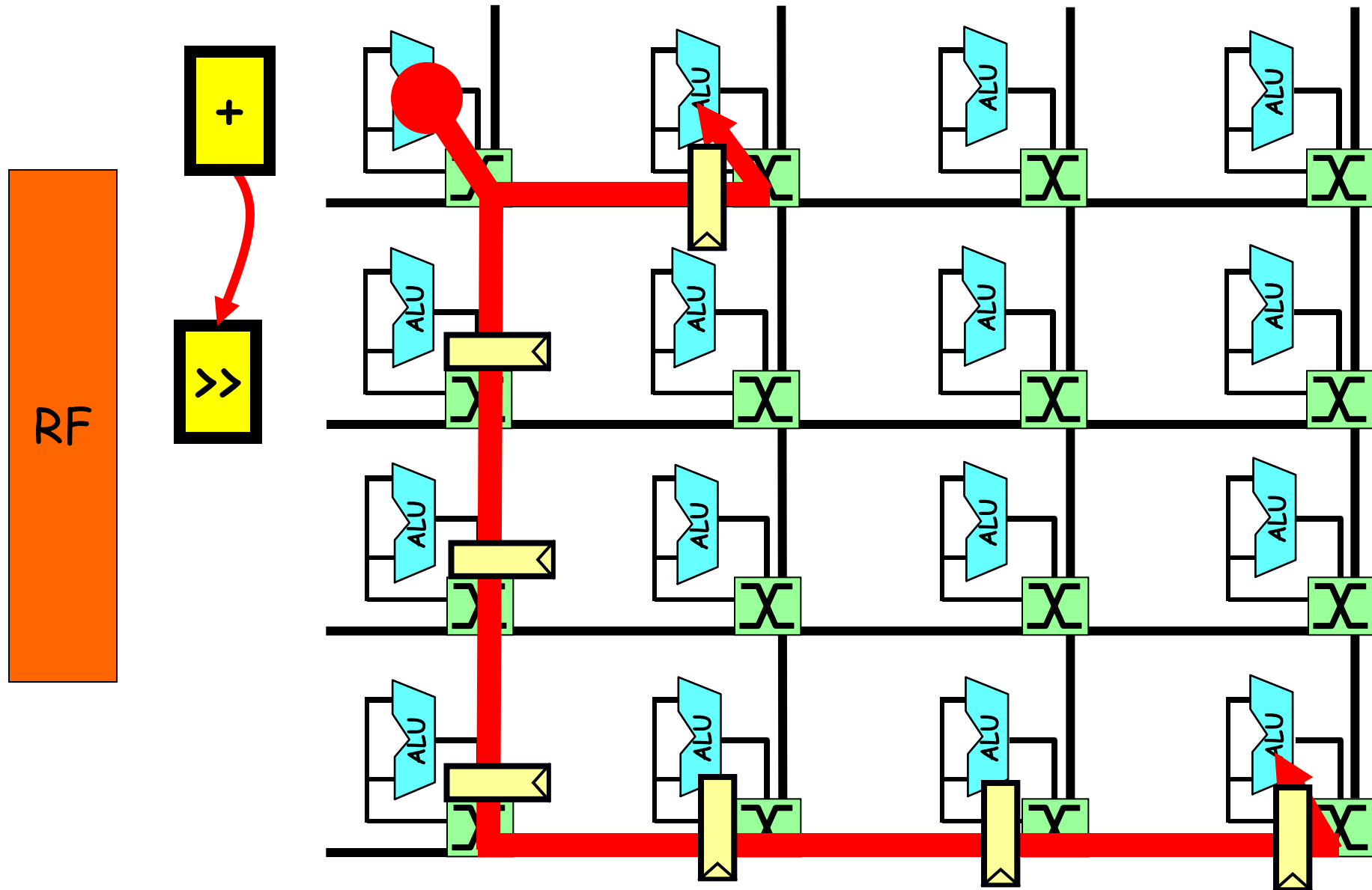


# Idea 1: Make operand routing local

SFU





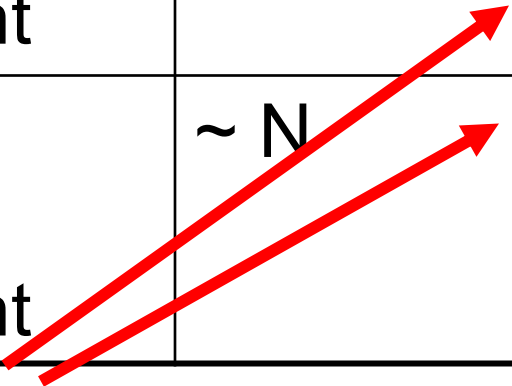




# Operand Latency

Time for operand to travel between instructions mapped to different ALUs.

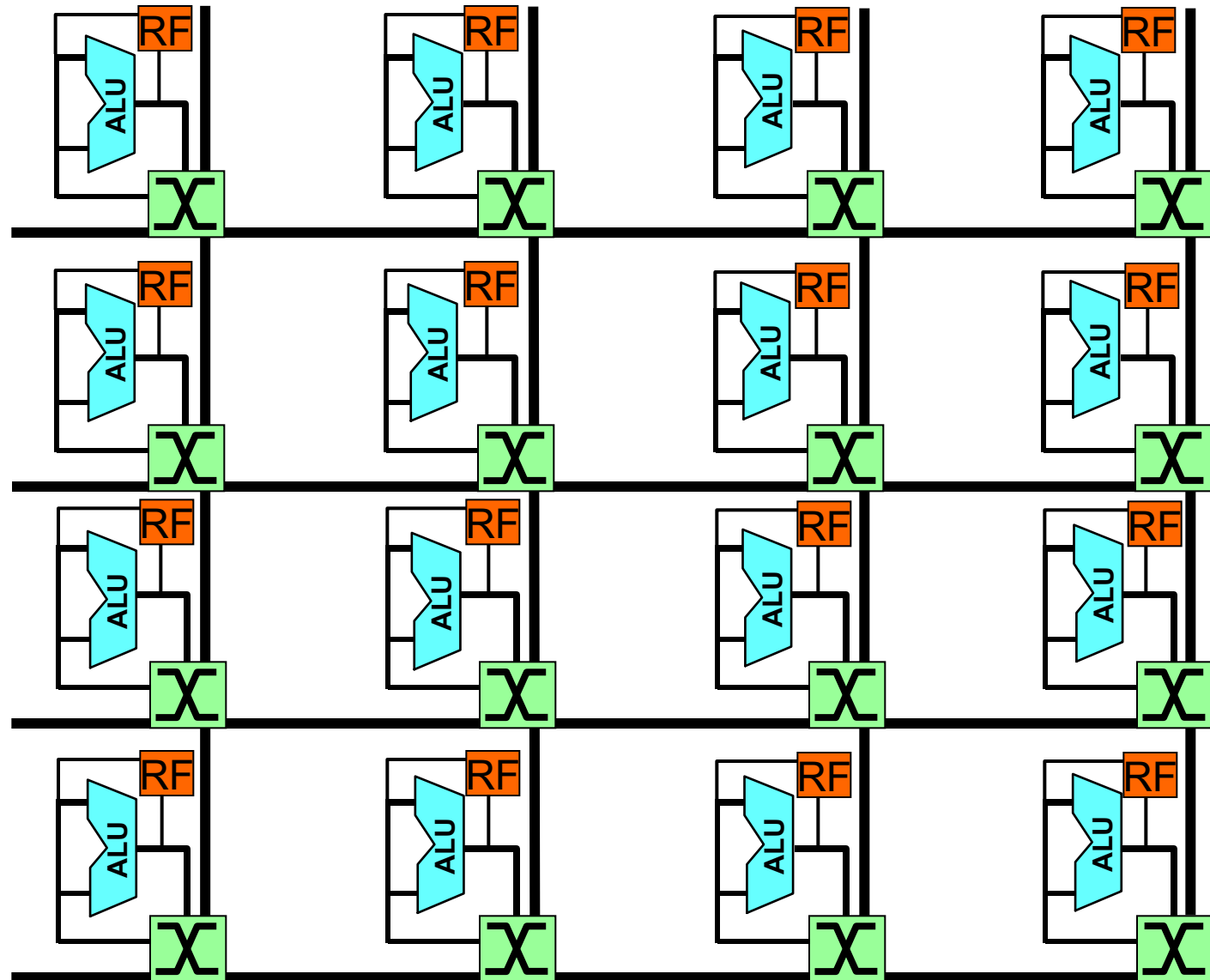
	Un-pipelined crossbar	Point-to-Point Routed Mesh Network
Non-local Placement	$\sim N$	$\sim N^{1/2}$
Locality-Driven Placement	$\sim N$	$\sim 1$



Latency bonus if we map communicating instructions nearby so communication is local.

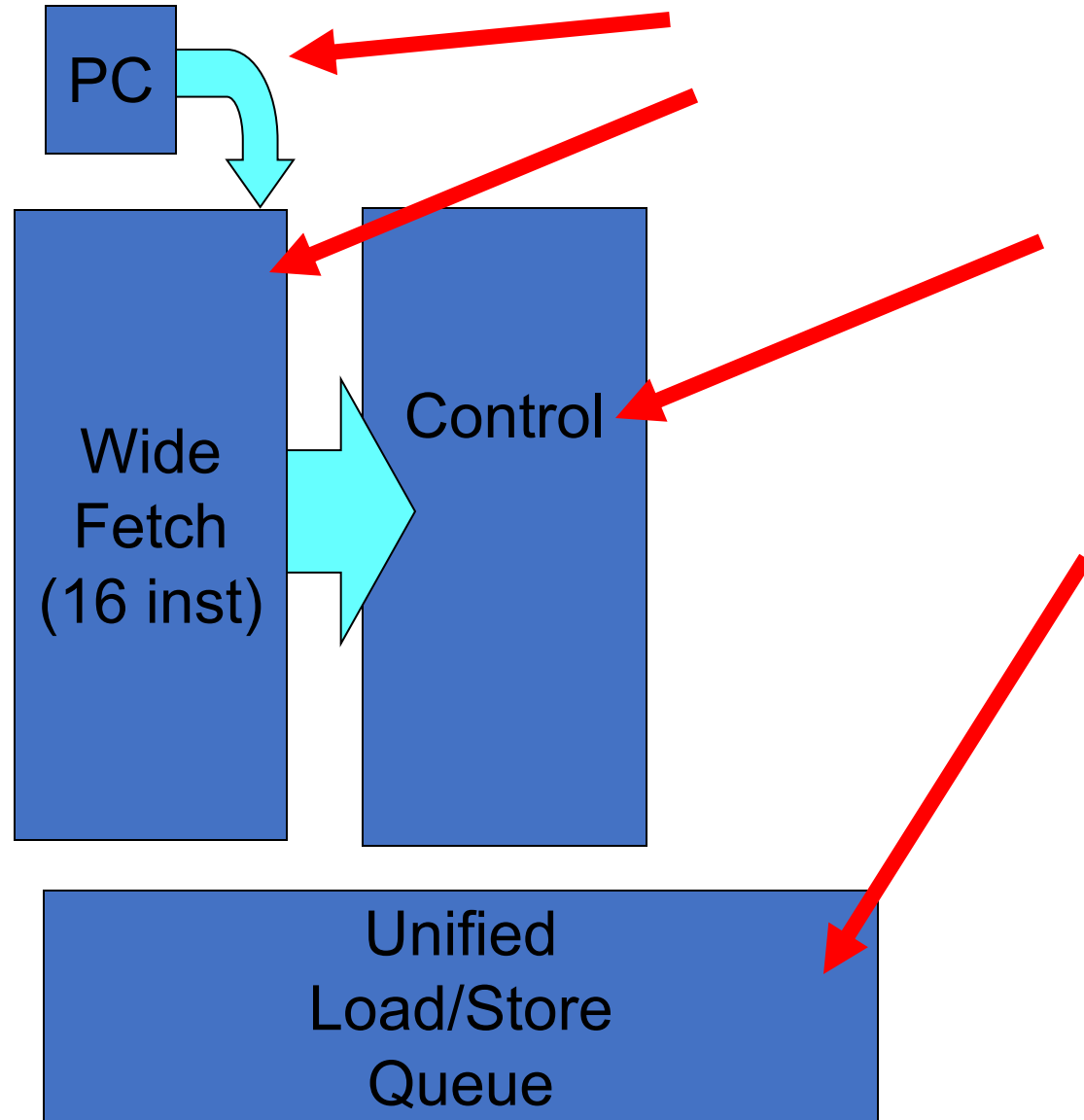


# Distribute the Register File



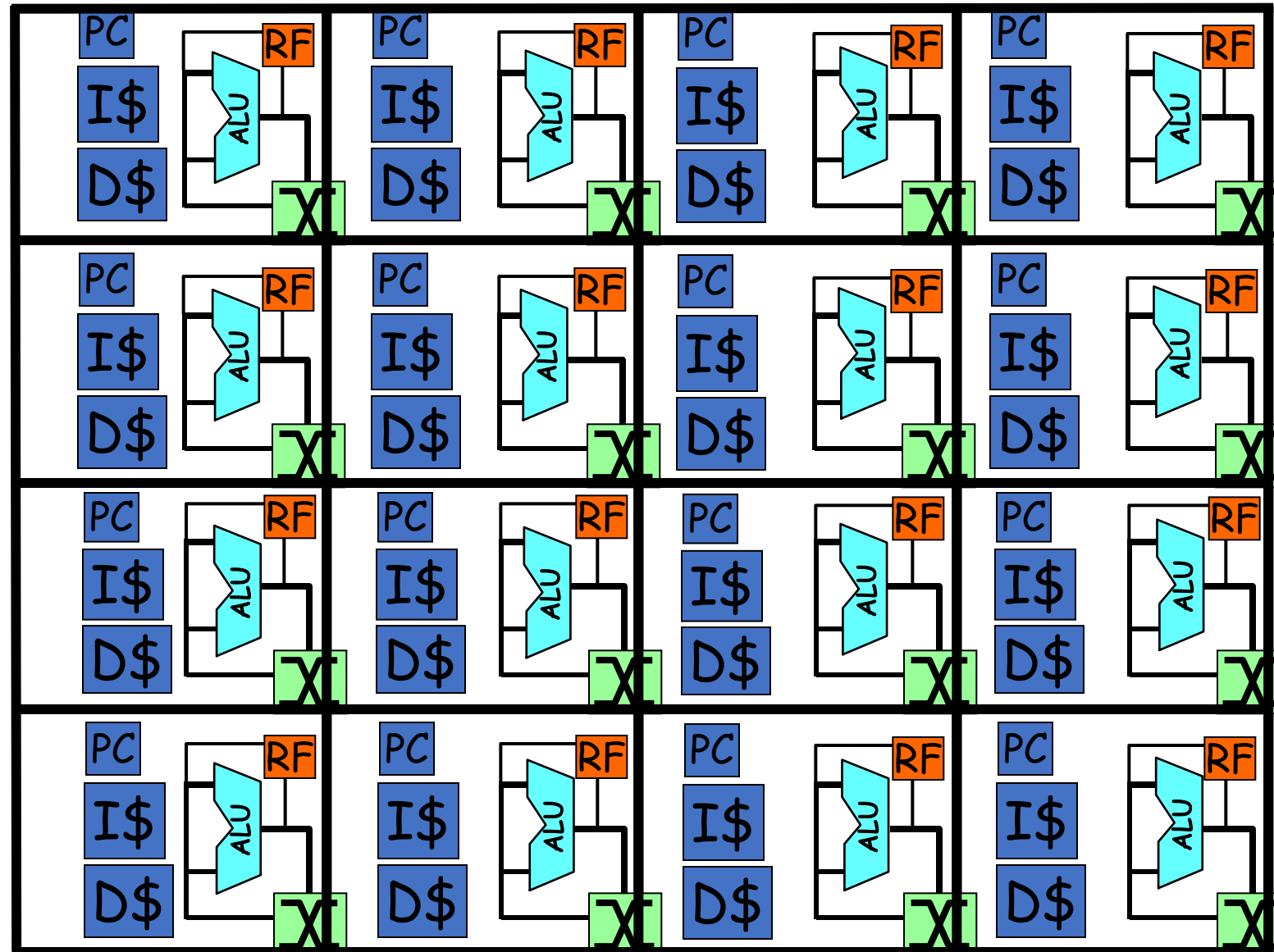


# More Scalability Problems





# Tiles (precursor to multicore)

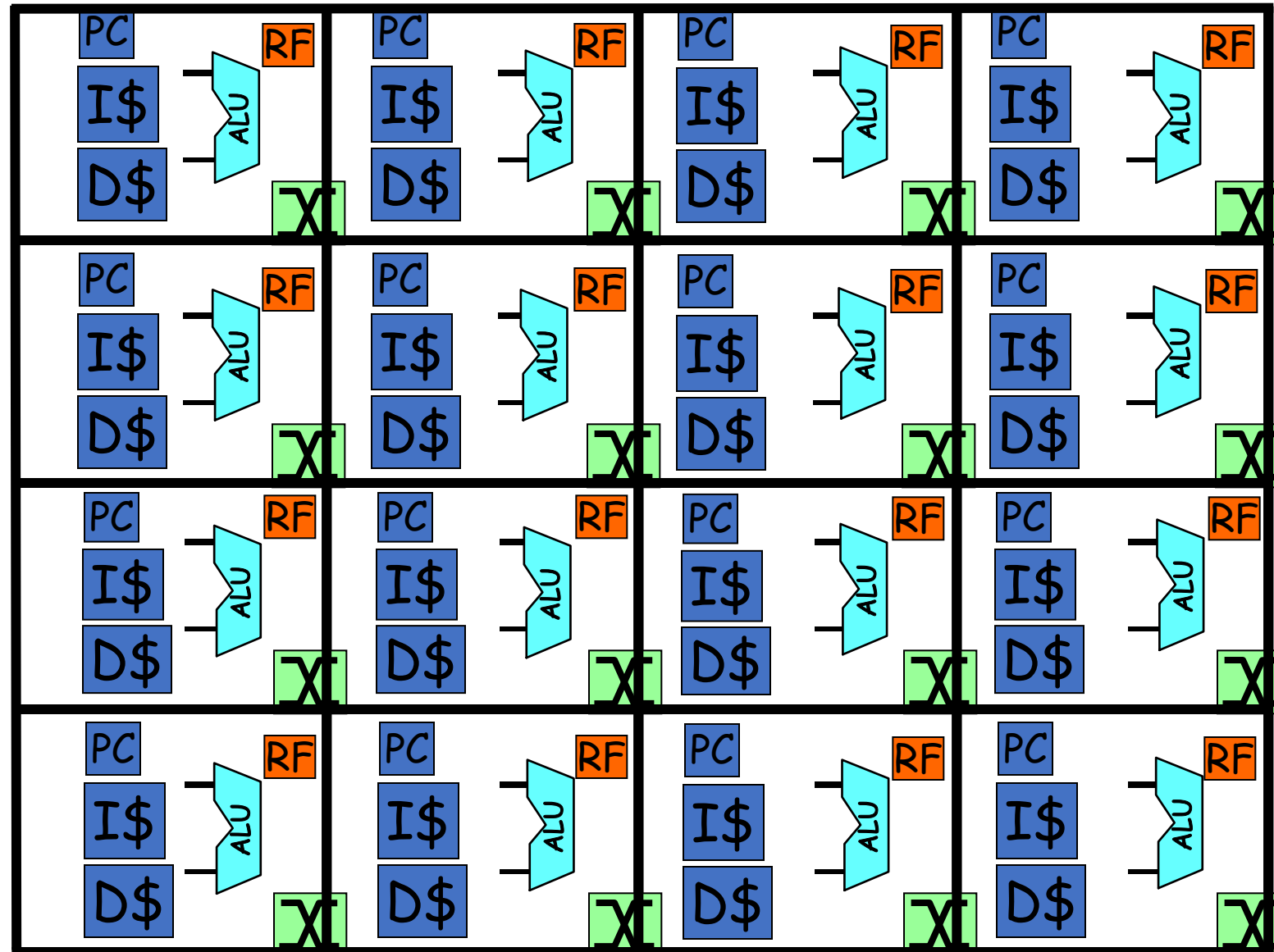




- [Complexity-effective superscalar processors](#), Subbarao Palacharla, Norman P Jouppi, and J E Smith In *PROC of the 24th ISCA*, Technical report , 1997.
- HP Clustered Processors** <https://www.hpl.hp.com/techreports/98/HPL-98-204.pdf>
- Scalar Operand Networks**,  
by Michael B Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal.  
*IEEE Transactions on Parallel and Distributed Systems*, February 2005



# Multicore (what was practical)

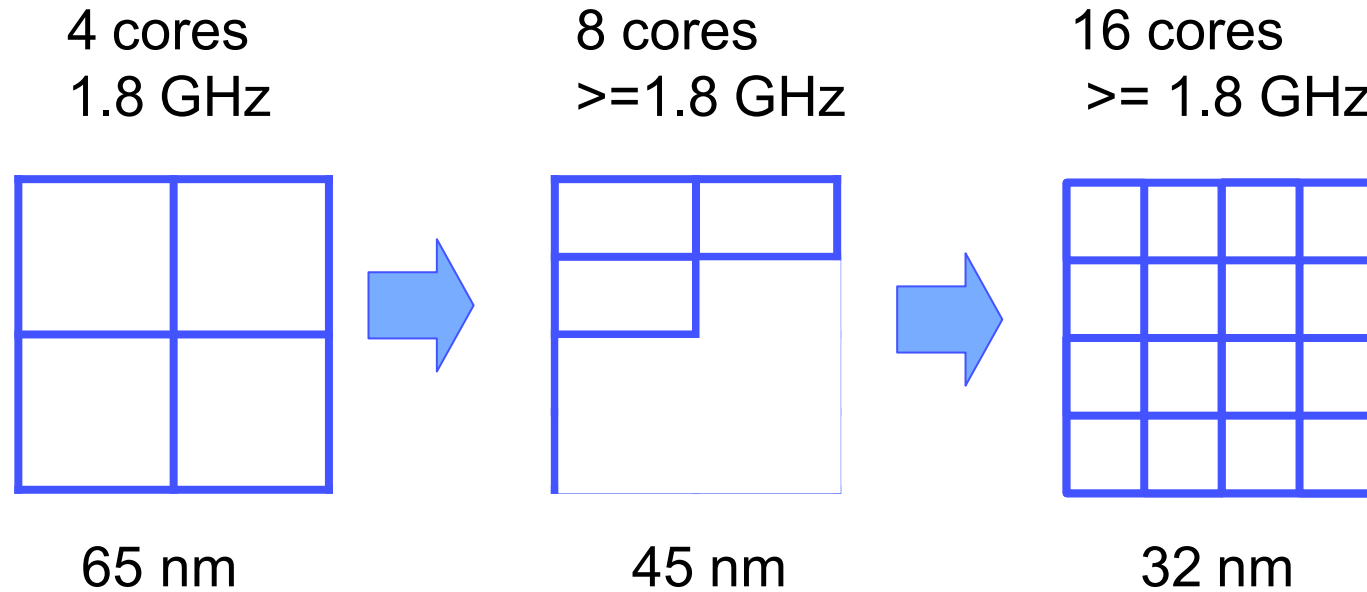




**Widespread Assumption: Microarchitecture was the cause of the power problem**



# The Scaling Promise of Multicore



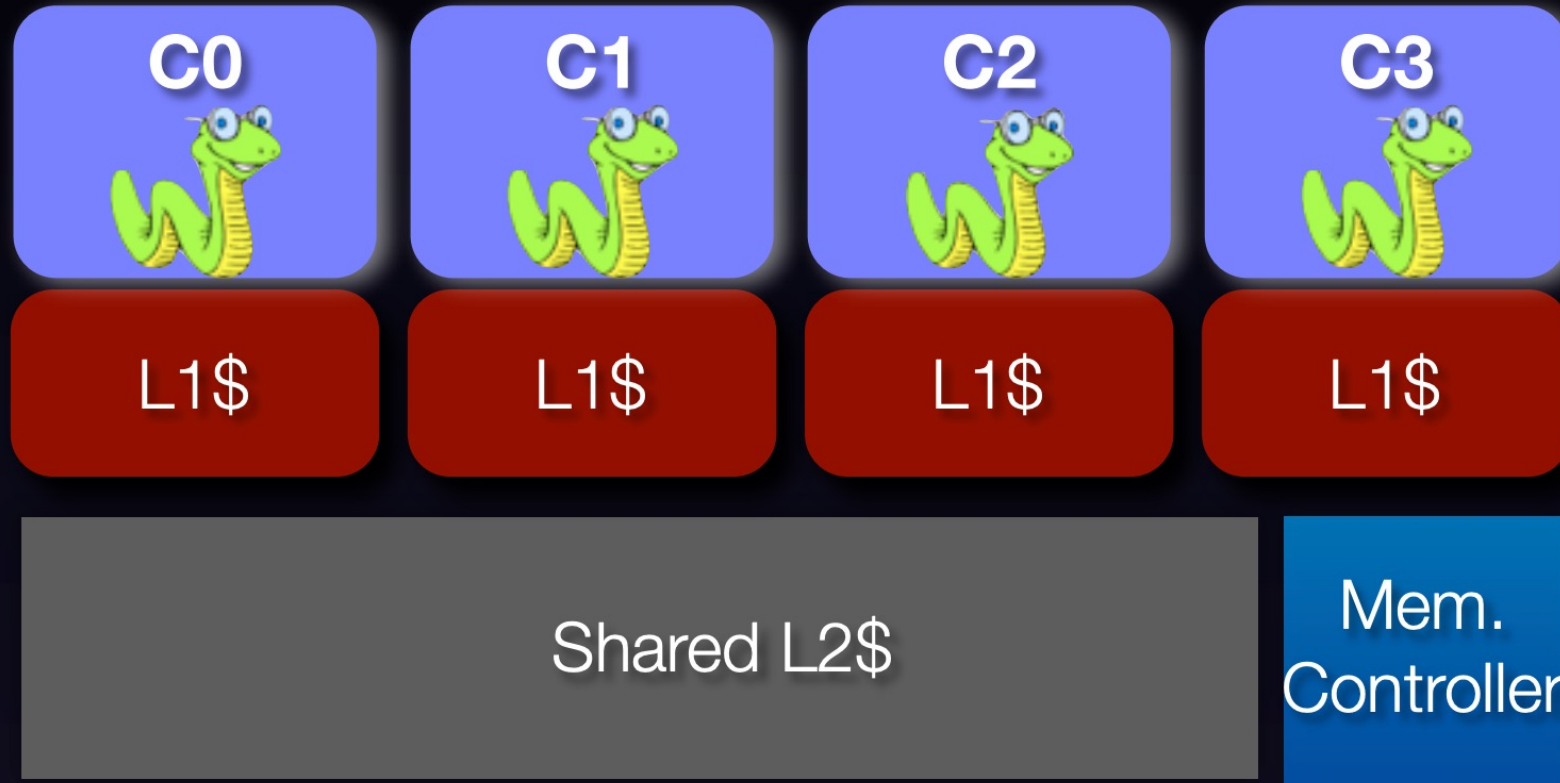
2x cores per generation,  
flat or slightly growing frequency



# More cores on a chip

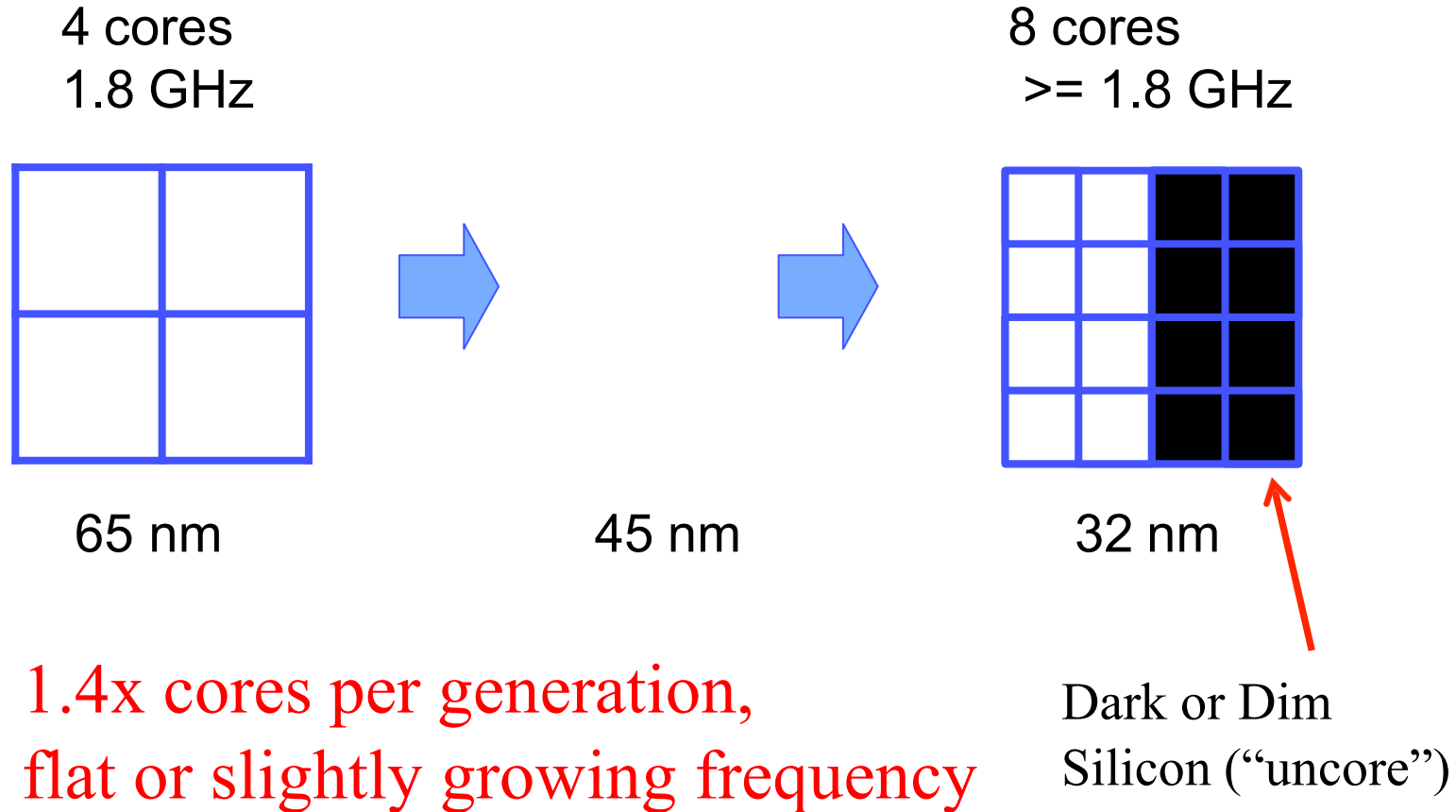
Each core ; 40% ↓ Ghz = 0.25x Power ↓

Overall Performance = 4 cores \* 0.6x/core = 2.4x





## But actually, that's not what's happening





**Why 000s suck.**

**Is technology scaling dead/dying ?**

**Are DSAs/Accelerators The Solution?**



# Scaling 101: Moore's Law

90 65 45 32 22 16 11 8 nm



$$S = \frac{22}{16} = \sim 1.4x$$



# Scaling 101:

## Transistors scale as $S^2$

180 nm

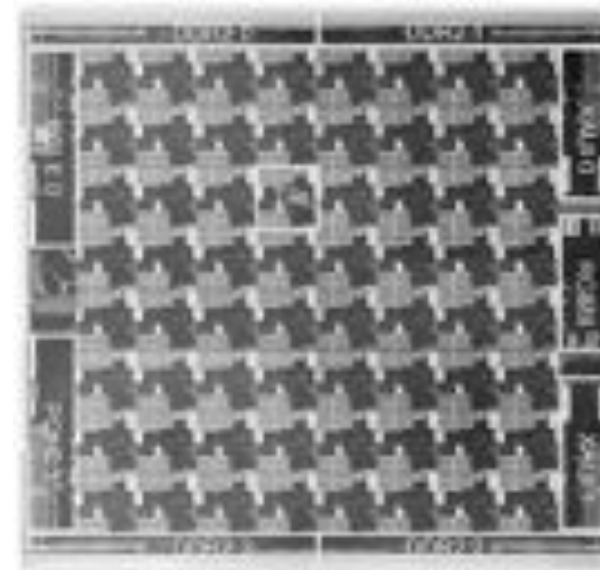
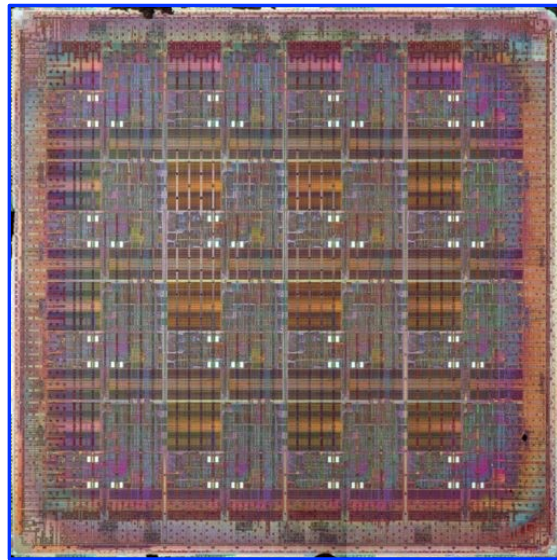
16 cores

$S = 2\times$

Transistors =  $4\times$

90 nm

64 cores







## Advanced Scaling:

If  $S=1.4\times \dots$

Scale by  **$S^3 = 2.8\times$**

$S^3$

$S^2$

$S$

1

Design of Ion-Implanted MOSFETs with Very Small Dimensions

Dennard et al, 1974

# Dennard: “Computing Capabilities





## Advanced Scaling:

If  $S=1.4\times \dots$

Scale by  **$S^3 = 2.8\times$**



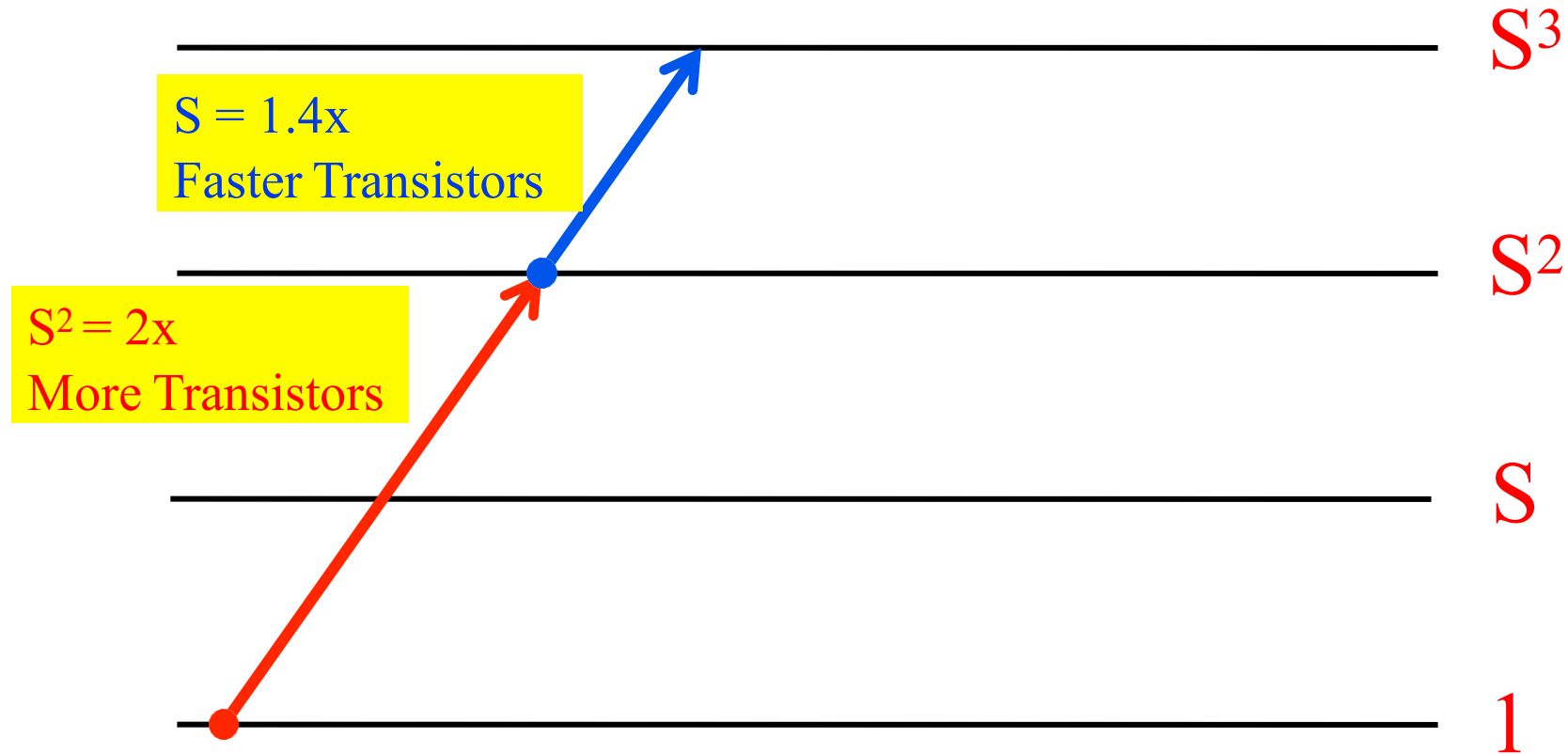
# Dennard: “Computing Capabilities



## Advanced Scaling:

If  $S=1.4x$  ...

Scale by  **$S^3 = 2.8x$**



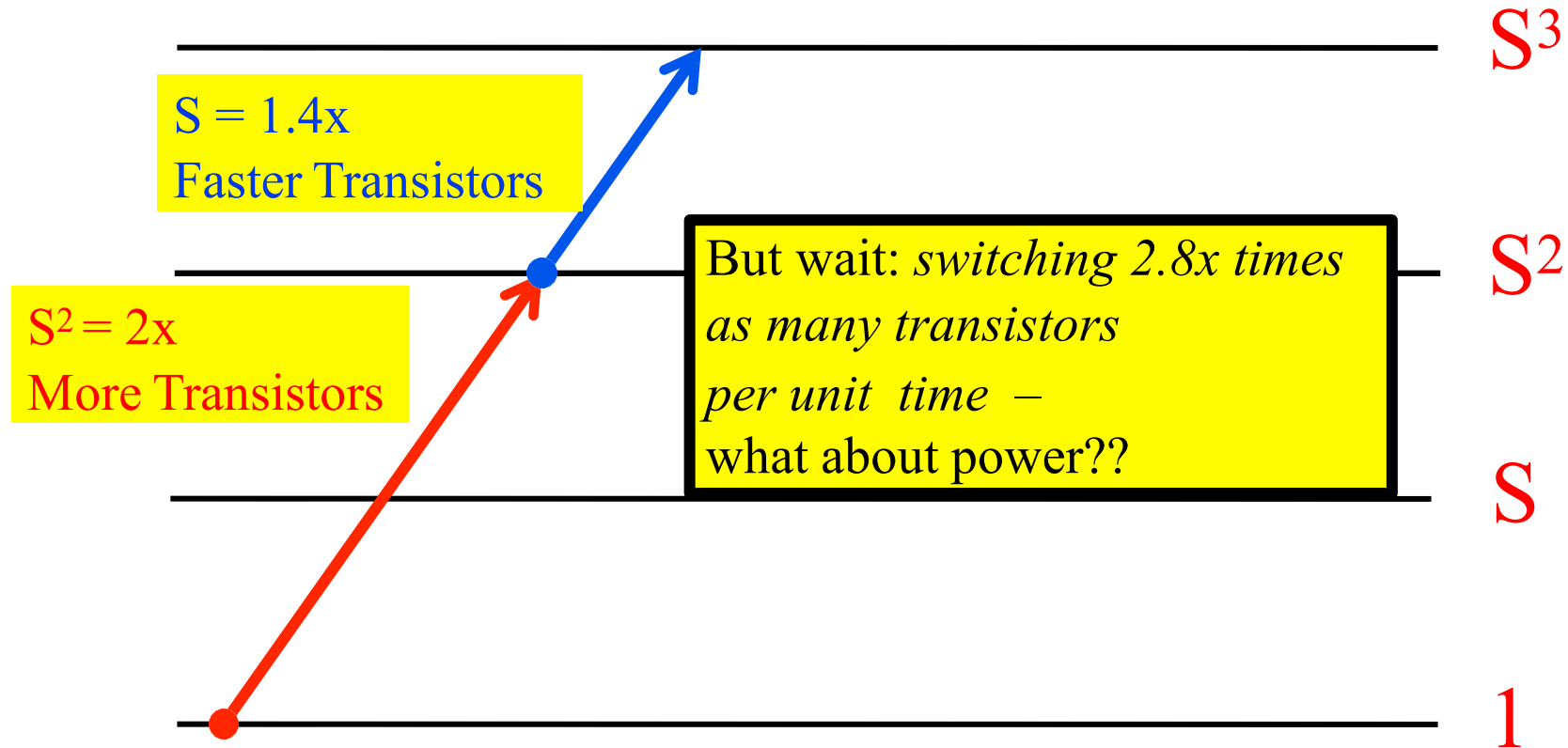
**Dennard: “Computing Capabilities**



## Advanced Scaling:

If  $S=1.4x$  ...

Scale by  **$S^3 = 2.8x$**

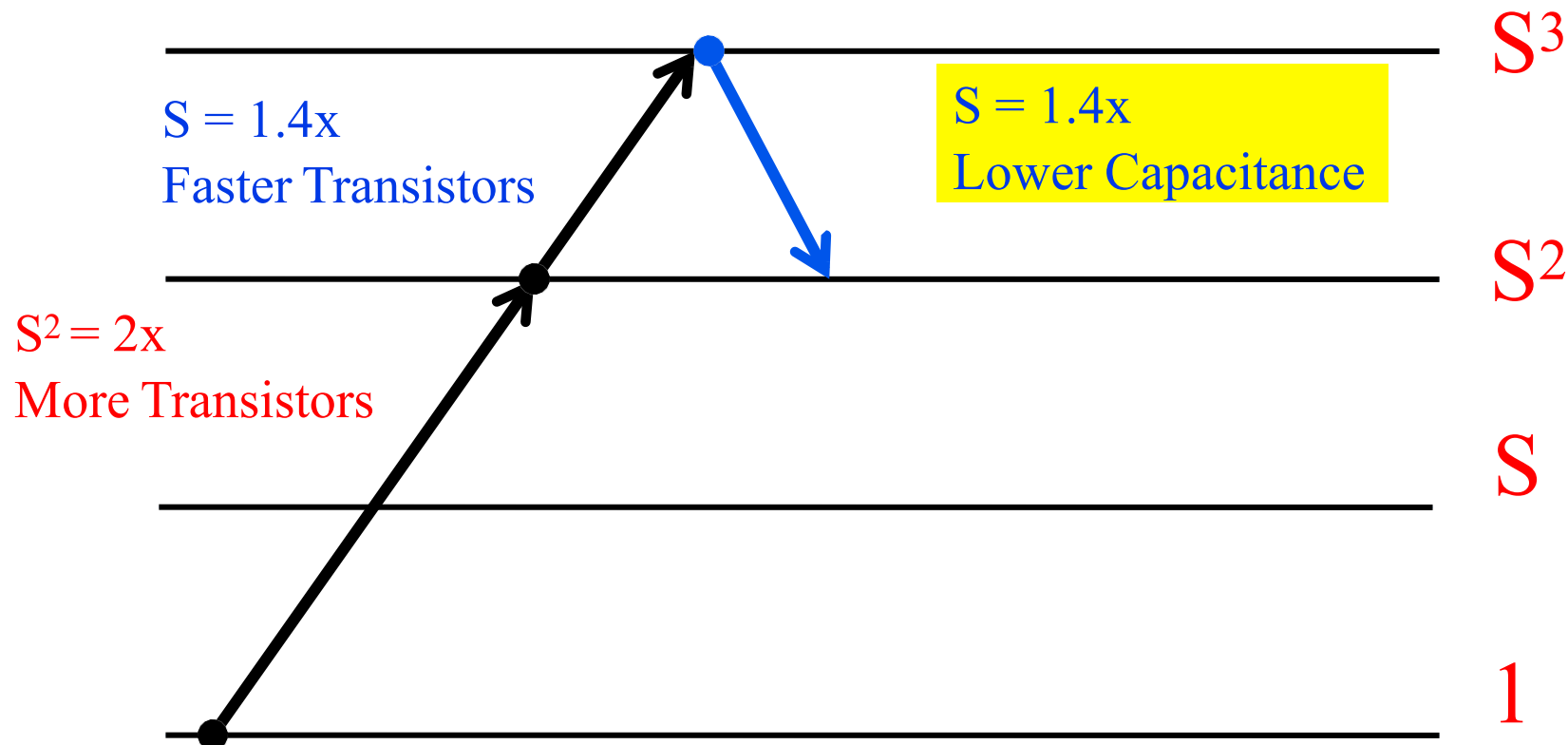


**“We can keep power consumption constant”**





## Dennard:

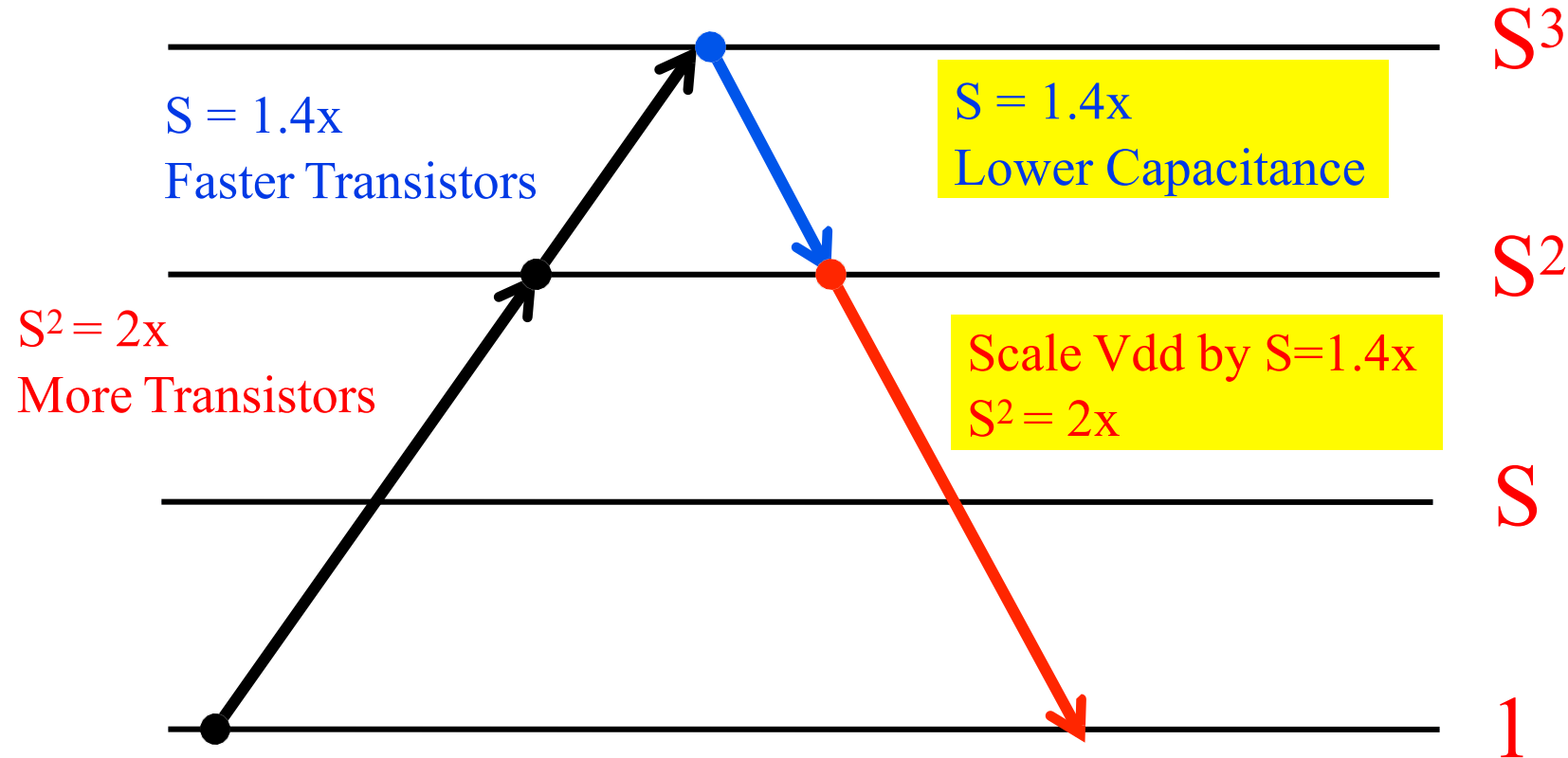


**“We can keep power consumption constant”**





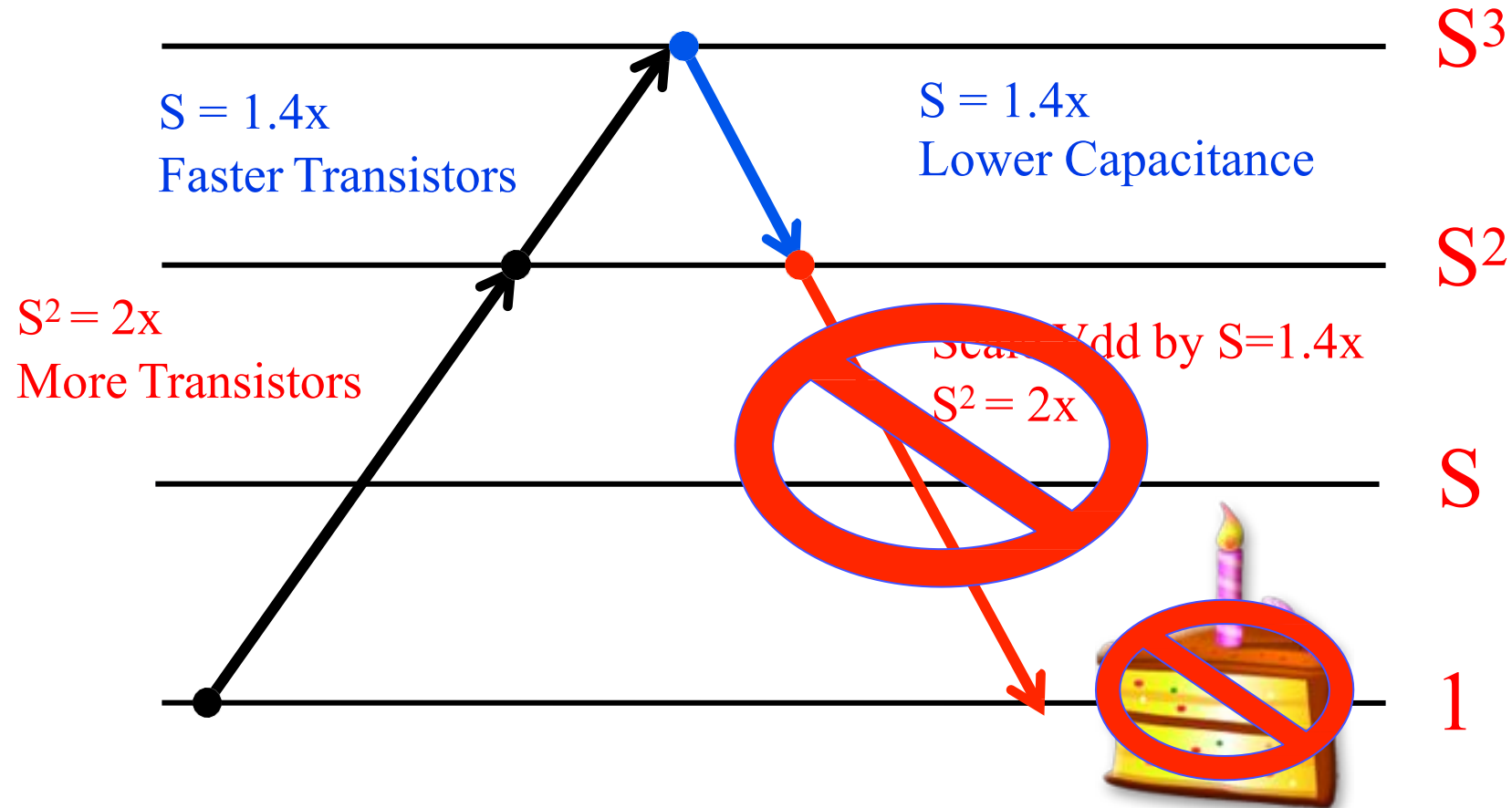
## Dennard:



**“We can keep power consumption constant”**



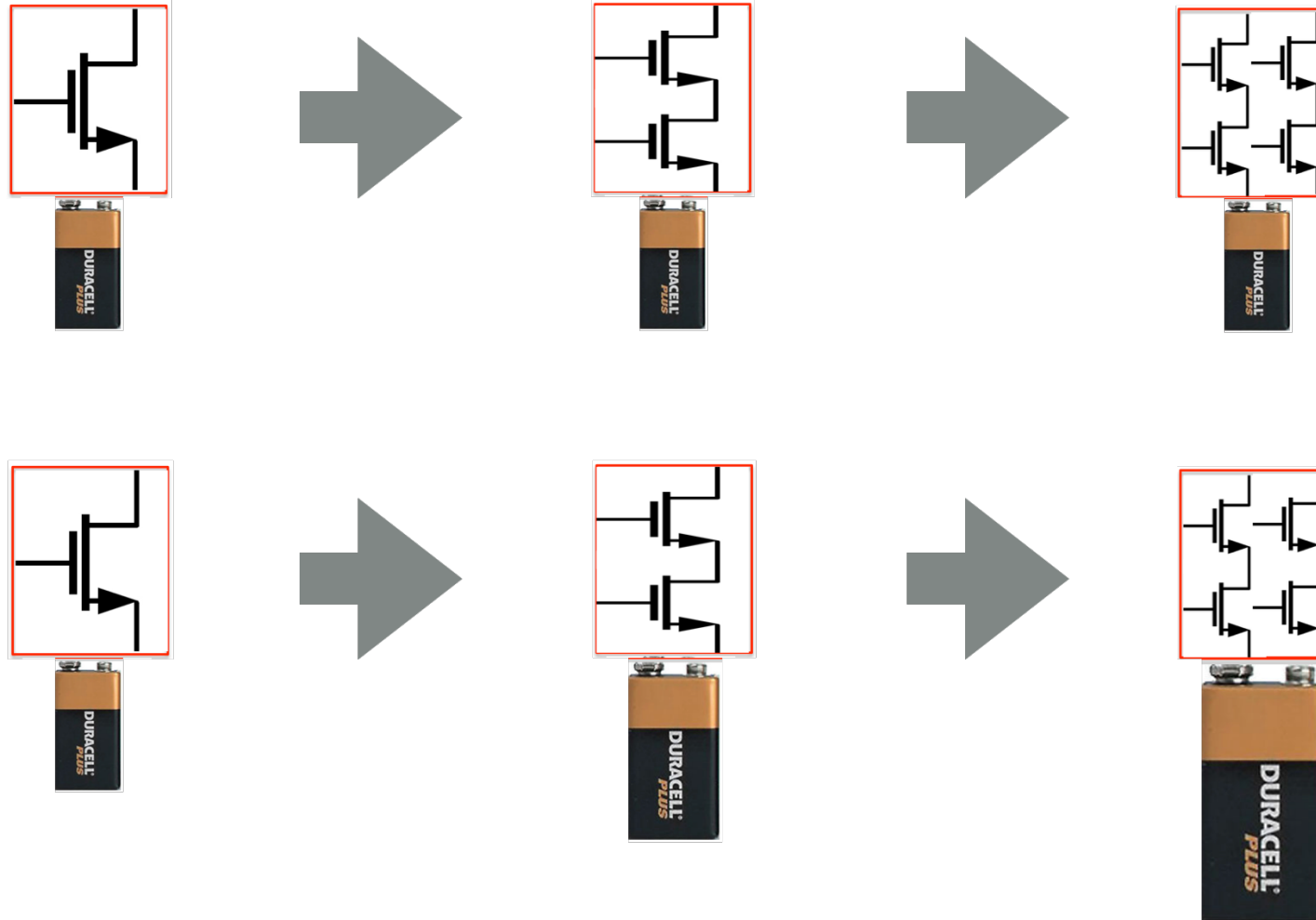
## Fast forward to 2005:



# Leakage Prevents Us From Scaling Voltage



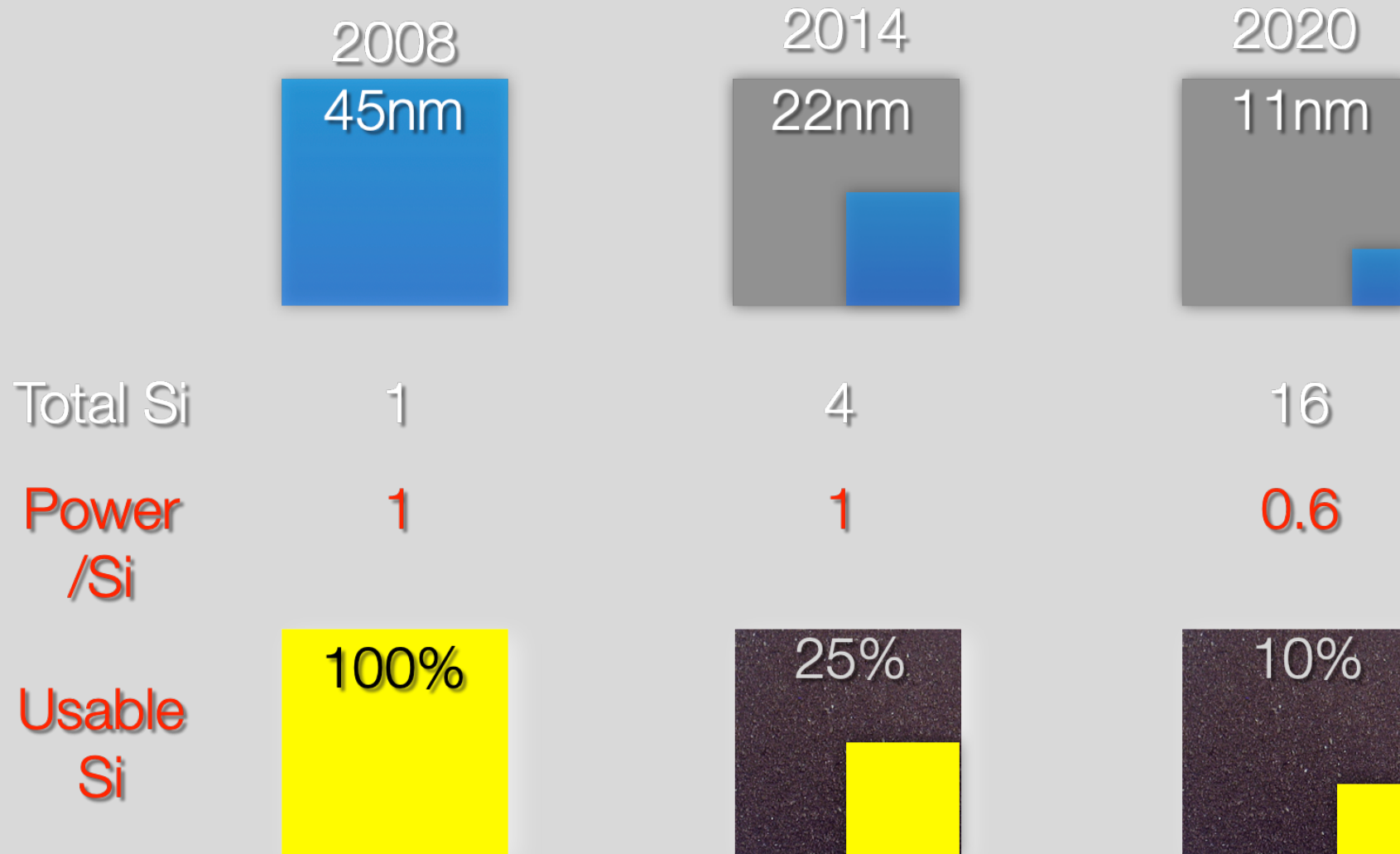
# Utilization Wall





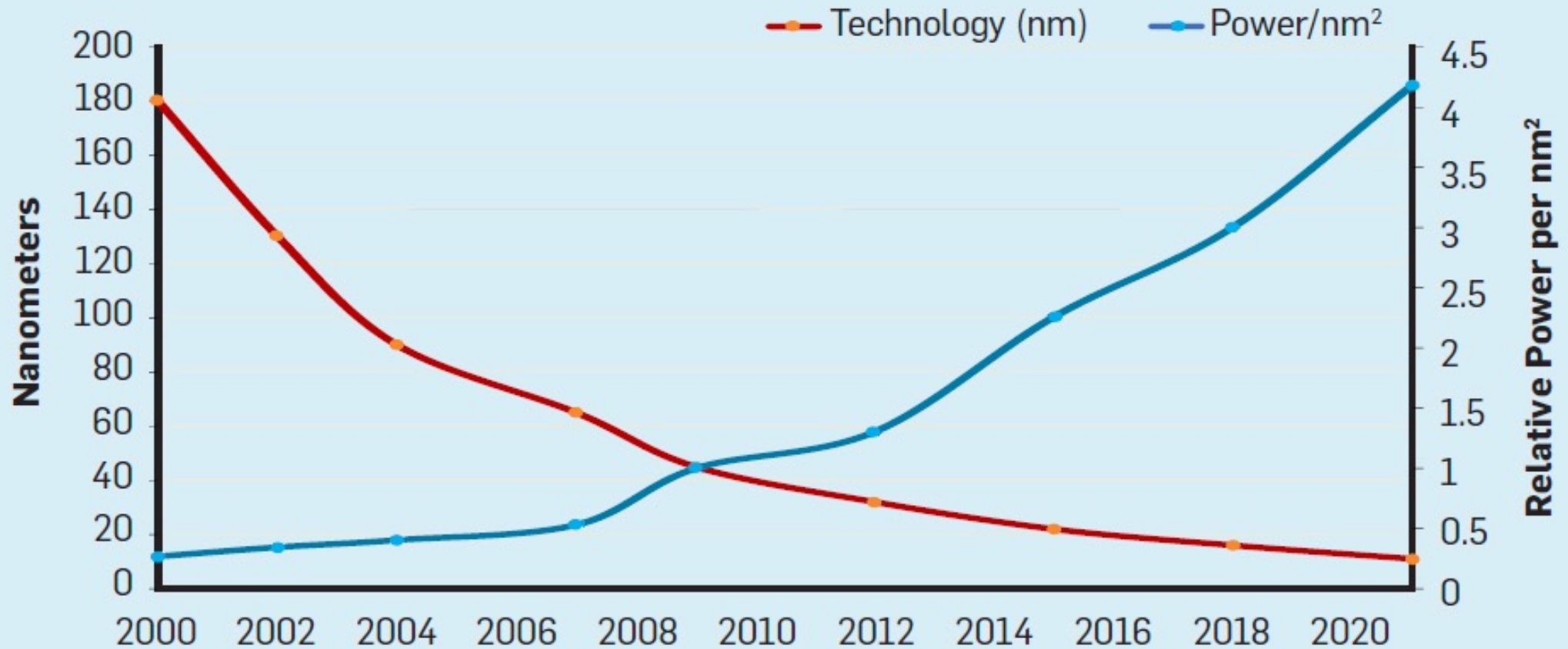
# We've Hit The Utilization Wall

*Utilization Wall: With each successive process generation, the percentage of a chip that can actively switch drops exponentially due to power constraints.*





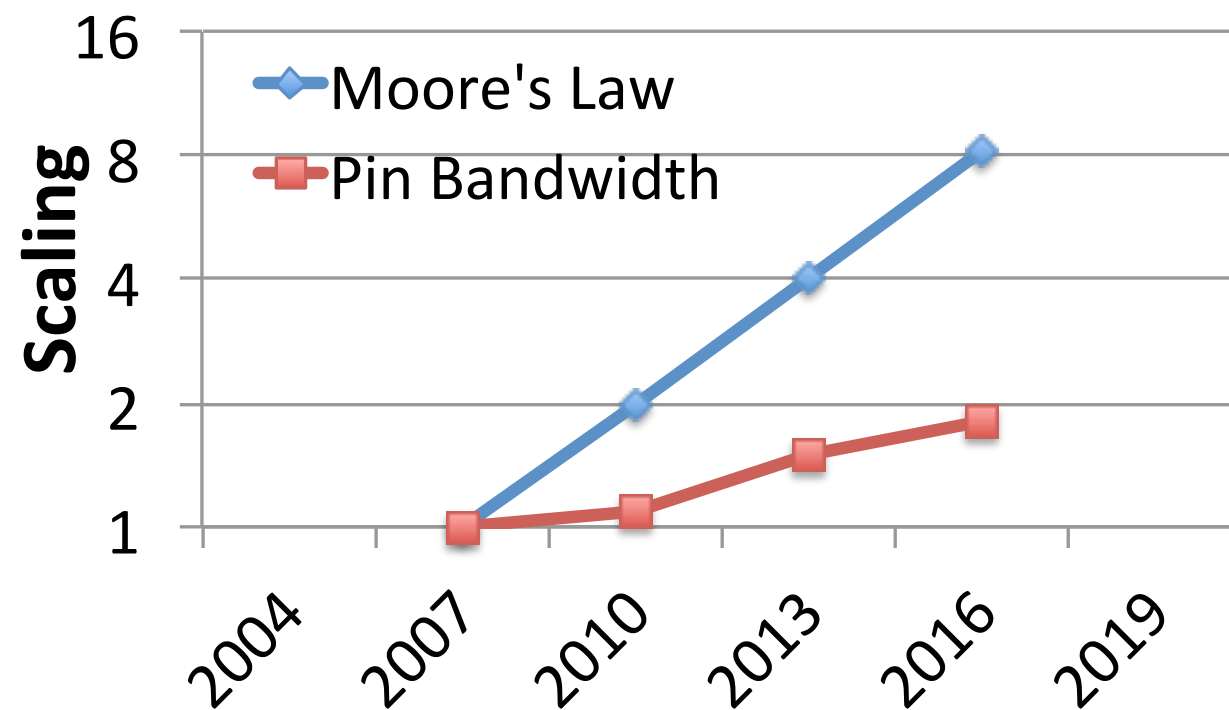
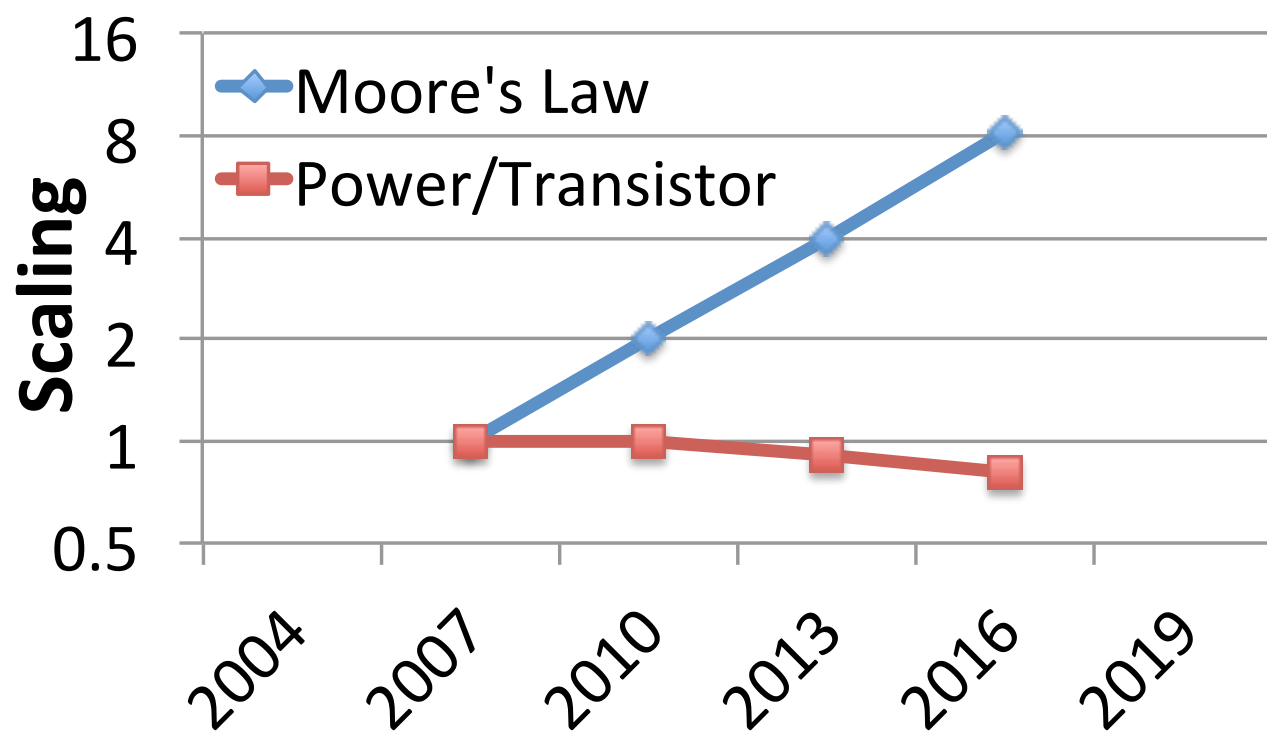
# Transistors vs Power



<https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext>

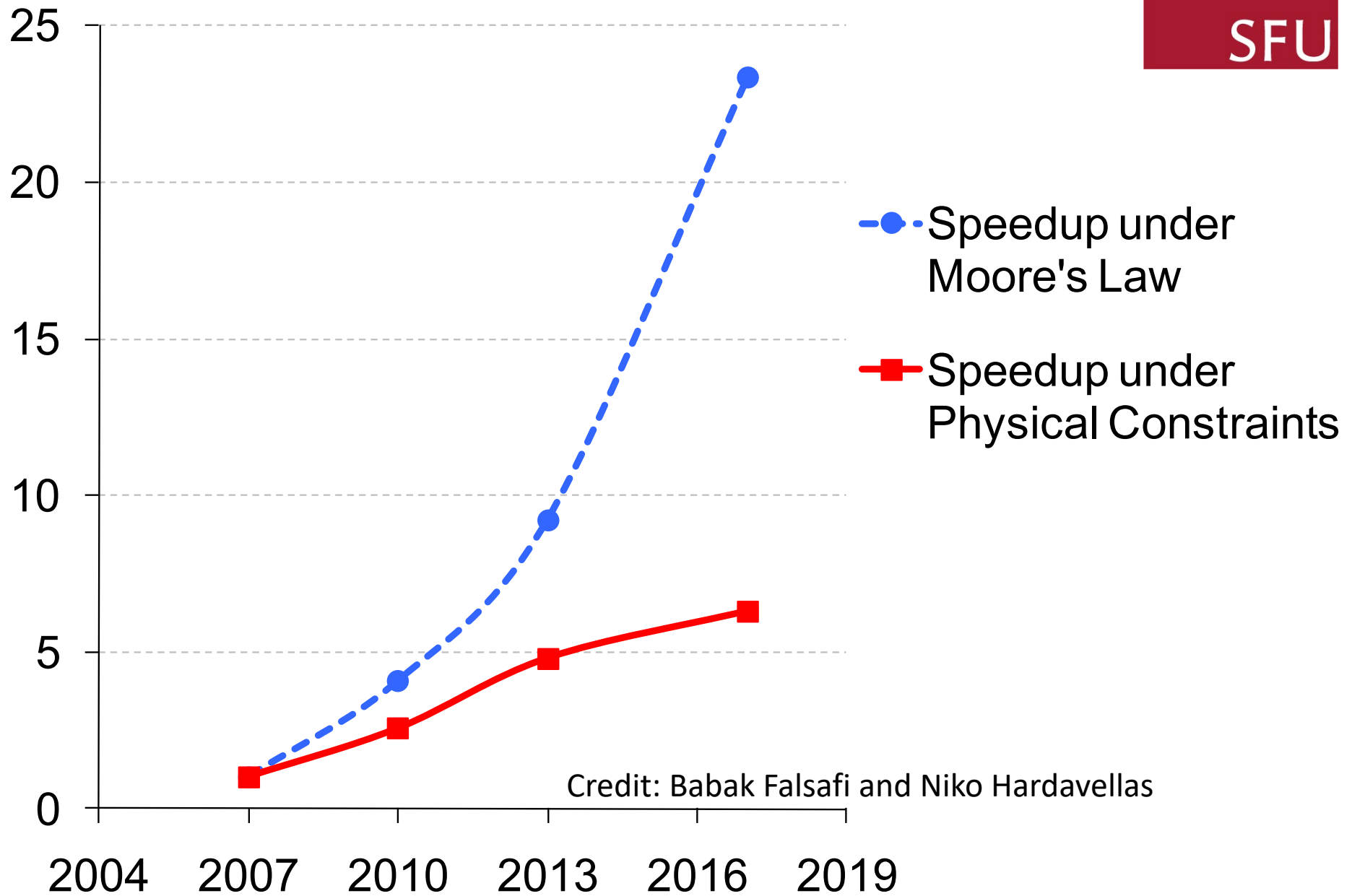


Credit: Babak Falsafi and Niko Hardavellas





Relative Performance



Credit: Babak Falsafi and Niko Hardavellas

Year of Technology Introduction

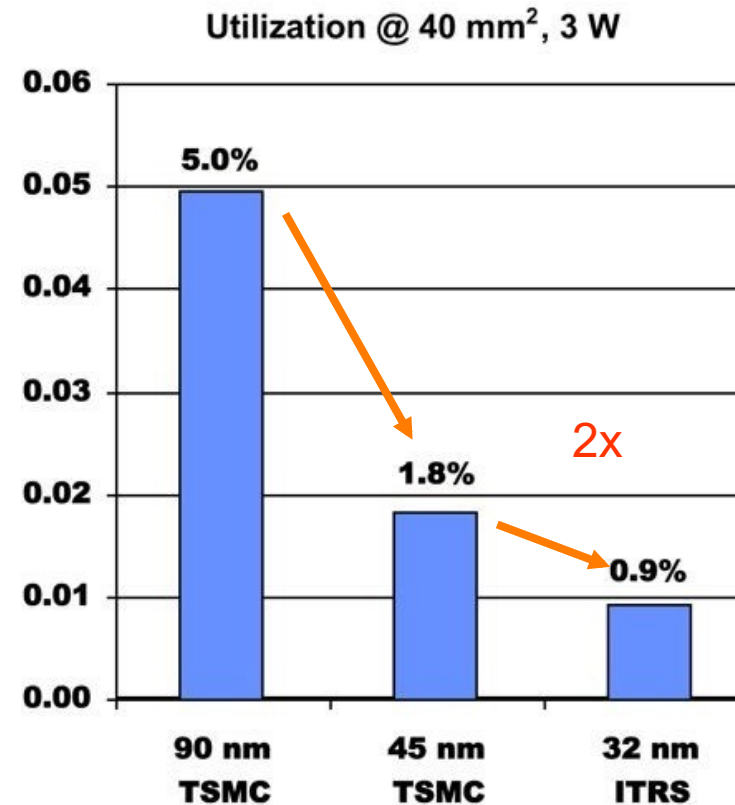


# Venkatesh [ASPLOS' 2010] – Utilization Wall

- Scaling theory
  - Transistor and power budgets are no longer balanced
  - Exponentially increasing problem!

- Experimental results
  - Replicated a small datapath
  - More "dark silicon" than active

- Observations in the wild
  - Flat frequency curve
  - "Turbo Mode"
  - Increasing cache/processor ratio



[Ganesh](#) et al:  
Conservation cores: reducing the  
energy of mature  
computations. [ASPLOS 2010](#): 205-218



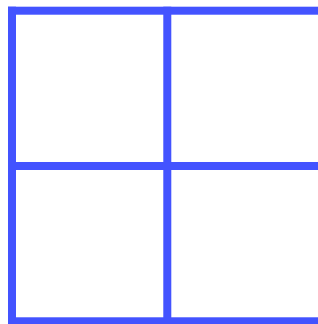
# Multicore hits the Utilization Wall

Spectrum of tradeoffs  
between # of cores and  
frequency

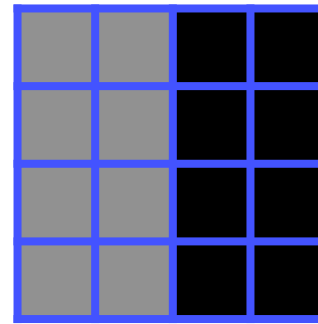
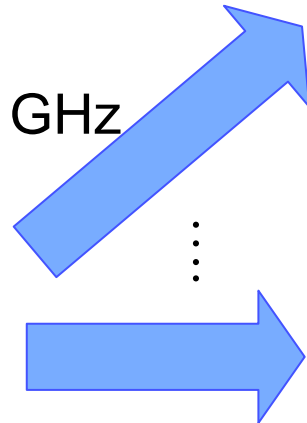
Example:

65 nm  $\rightarrow$  32 nm ( $S = 2$ )

4 cores @ 1.8 GHz

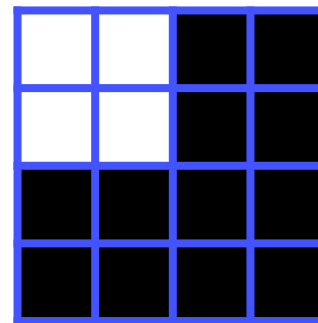


65 nm



4x4 cores @ .9 GHz  
(GPUs of future?)

2x4 cores @ 1.8 GHz  
(8 cores dark, 8 dim)



4 cores @ 2x1.8 GHz  
(12 cores dark)

32 nm

[Goulding, IEEE Micro 2011]  
[Esmaeilzadeh ISCA 2011]  
[Skadron IEEE Micro 2011]  
[Hardavellas, IEEE Micro 2011]



# Multicore has hit the Utilization Wall

Spectrum of tradeoffs  
between # of cores and  
frequency

⋮

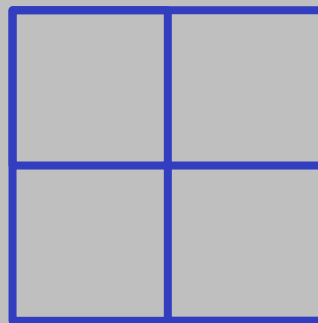


2x4 cores @ 1.8 GHz  
(m)

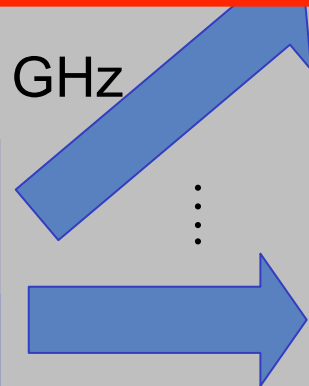
Exa  
65 n

The utilization wall will change the way  
everyone builds chips.

4 cores @ 1.8 GHz

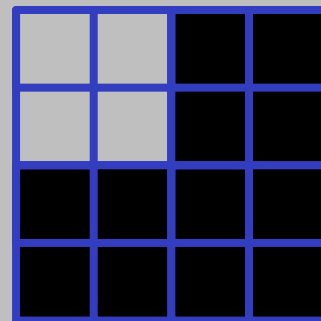


65 nm



⋮

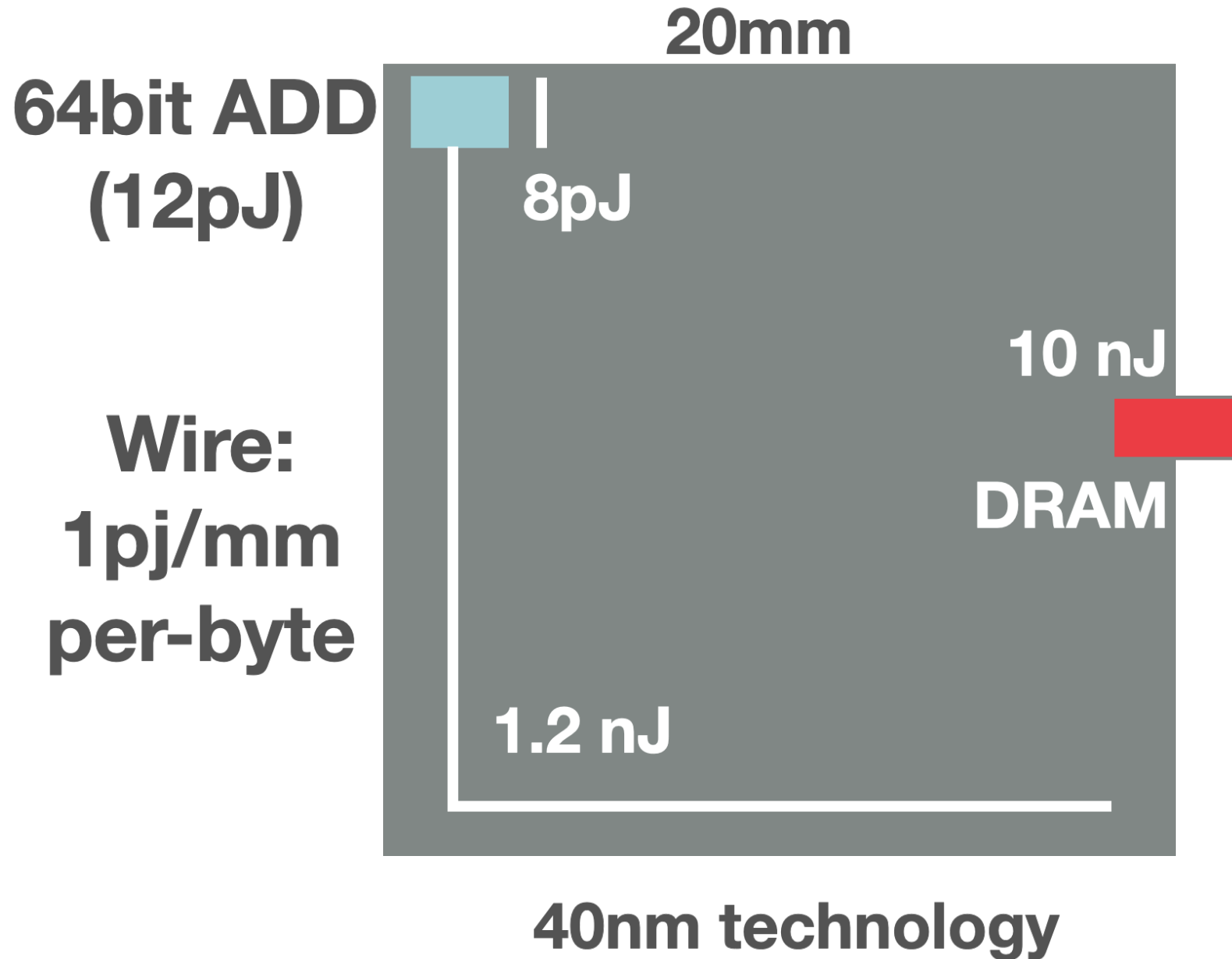
....



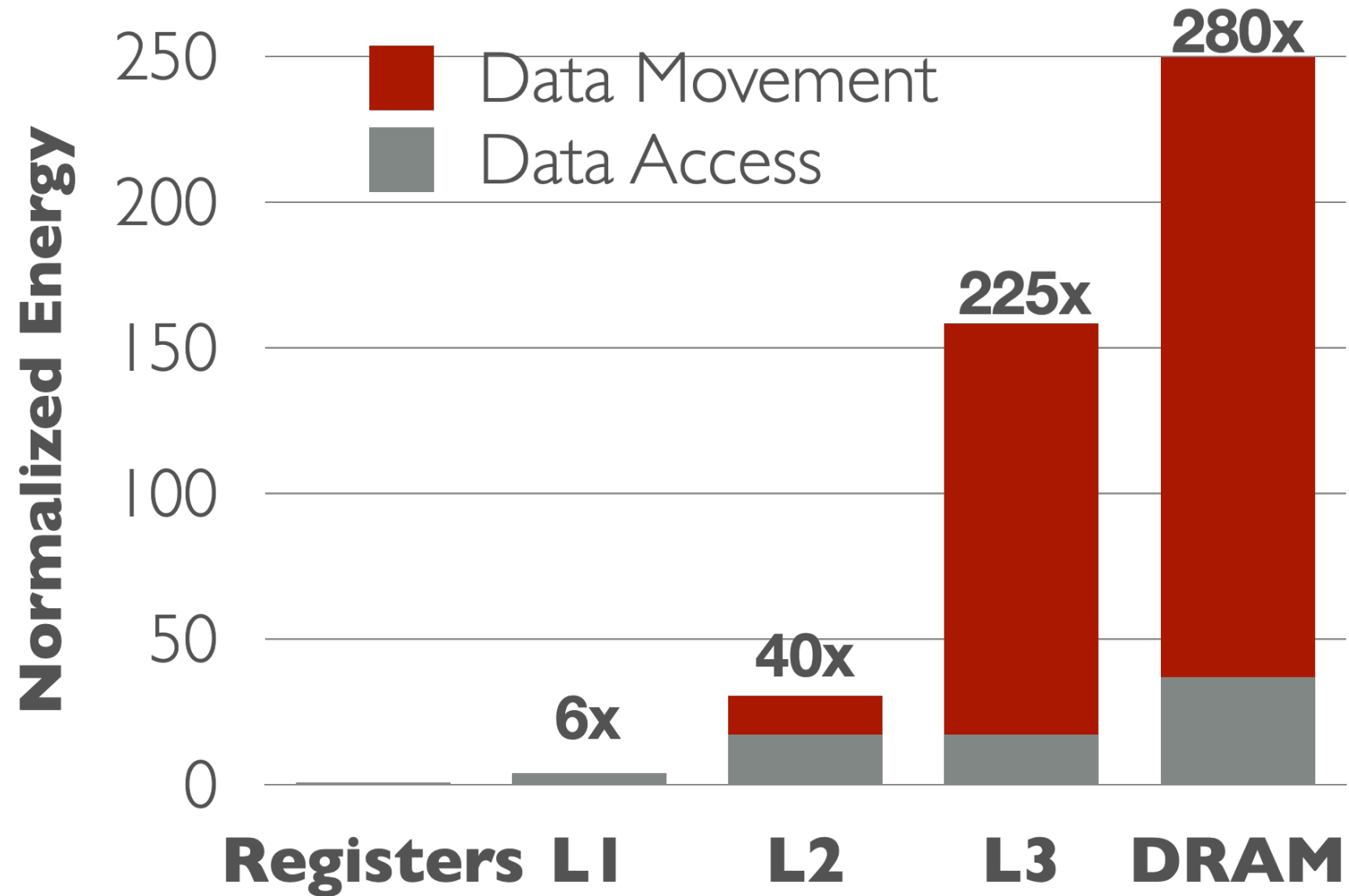
32 nm

4 cores @ 2x1.8 GHz  
(12 cores dark)









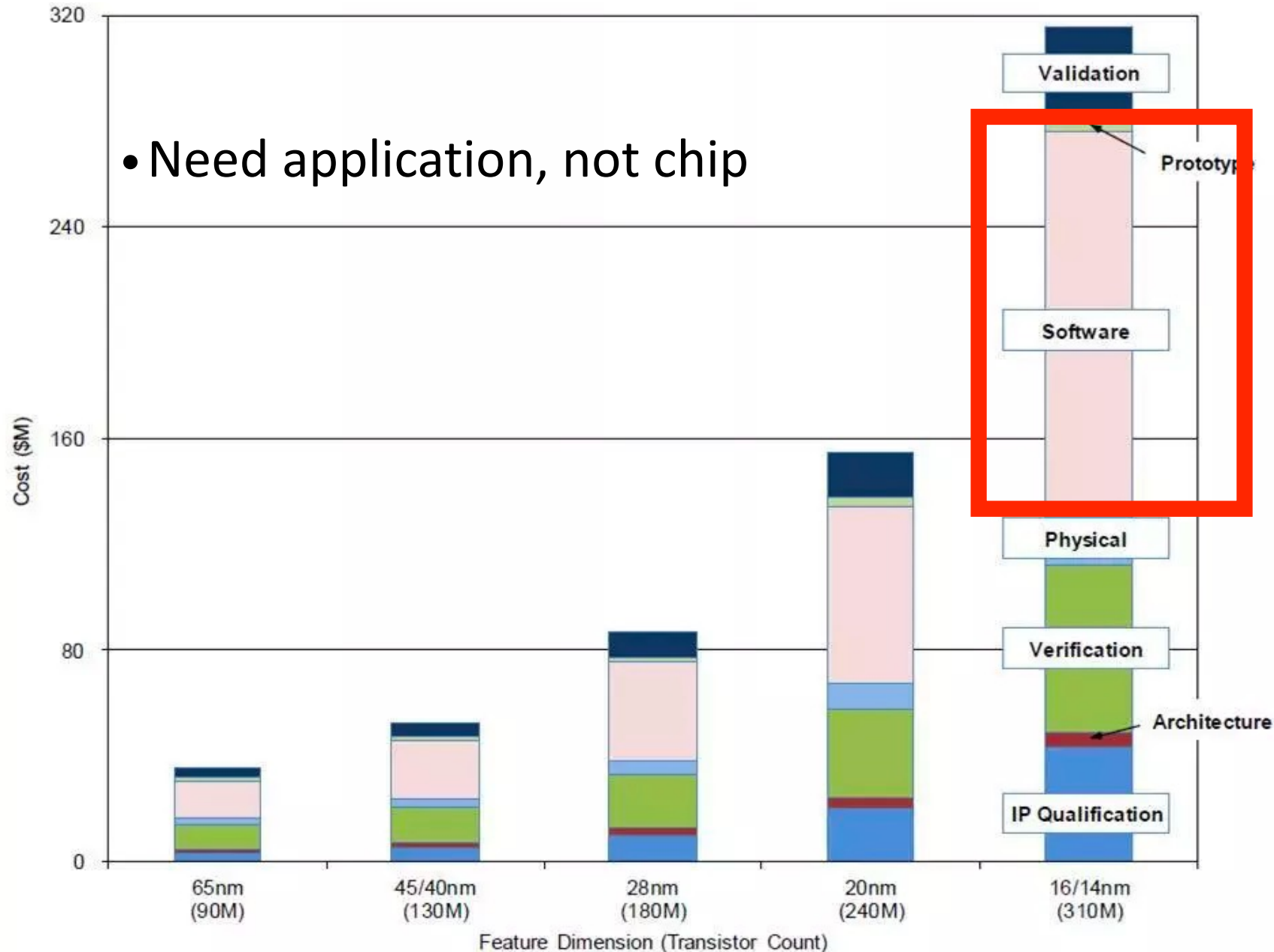


# Scaling : What's been going on ?

	2010	2017 (High Vdd)	2017 (Low Vdd)
Vdd	0.9V	0.75	0.65
Double prec. 64 bit FMA	50pJ	8.7pJ	6.5pJ <b>(8x)</b>
8KB SRAM (64bit read)	14pJ	2.4pJ	1.8pJ <b>(7x)</b>
<b>Wire energy (64 bits)</b>	<b>64pJ</b>	<b>40pJ</b>	<b>30pJ (2x)</b>

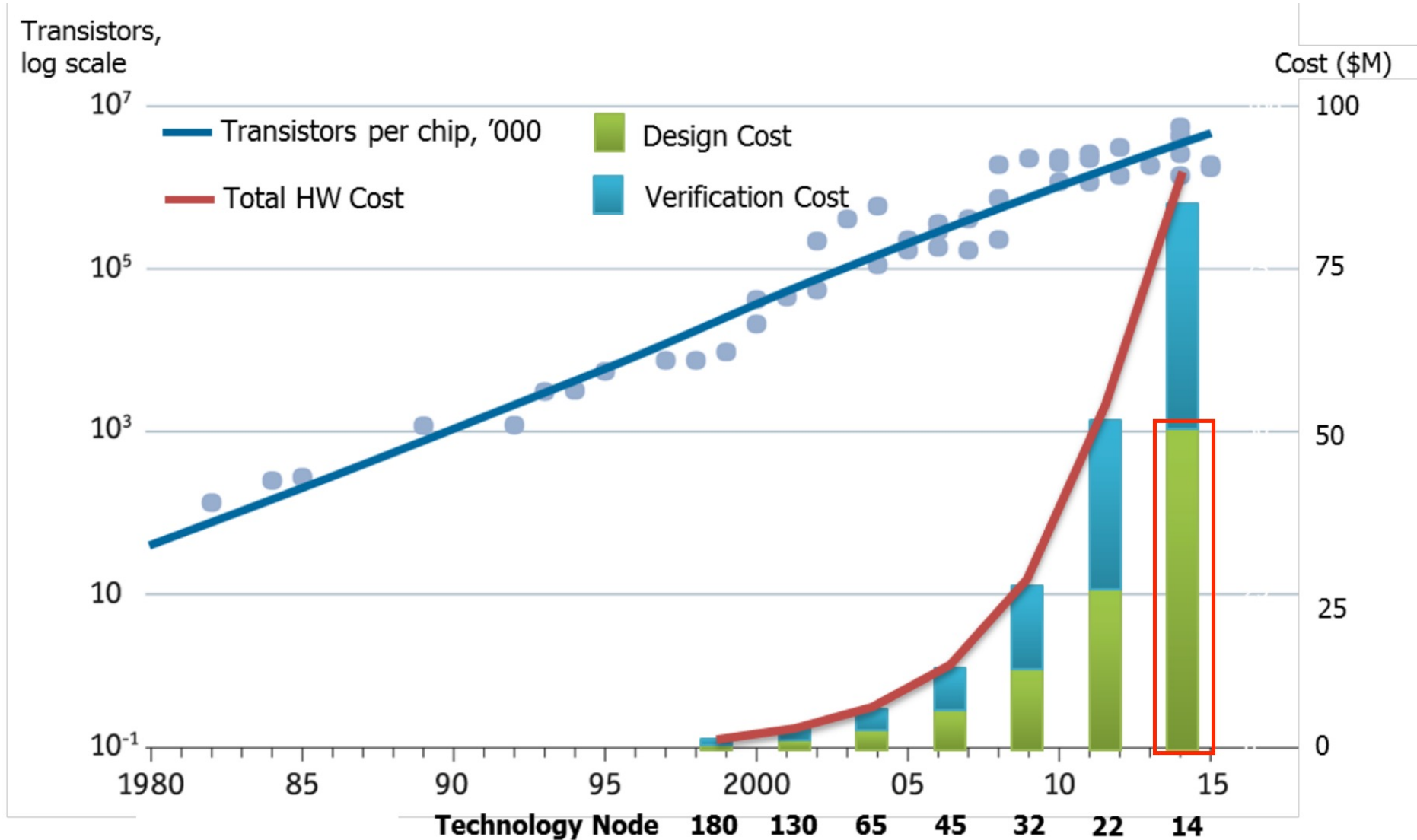


# Lesson 1: Its all about the software





# Lesson 2: Hardware design is hard





# Software Innovation Today



## Instagram

Proprietary Code  
500K-->13 people & \$1B

## Open Source

Python

Django

Memcached

Postgres/SQL

Redis

Apache

Linux

GNU \*

GCC

Credit: Michael Taylor  
Univ. of Washington





**Your Secret Sauce**

Where are the stdlibs ?

What are the APIs?

What are the  
abstractions?



Goal 1: Always keep the software-to-hardware flow

Goal 2: How do we lower the barrier to energy efficient chips?

Goal 3: Enable iterative HW exploration

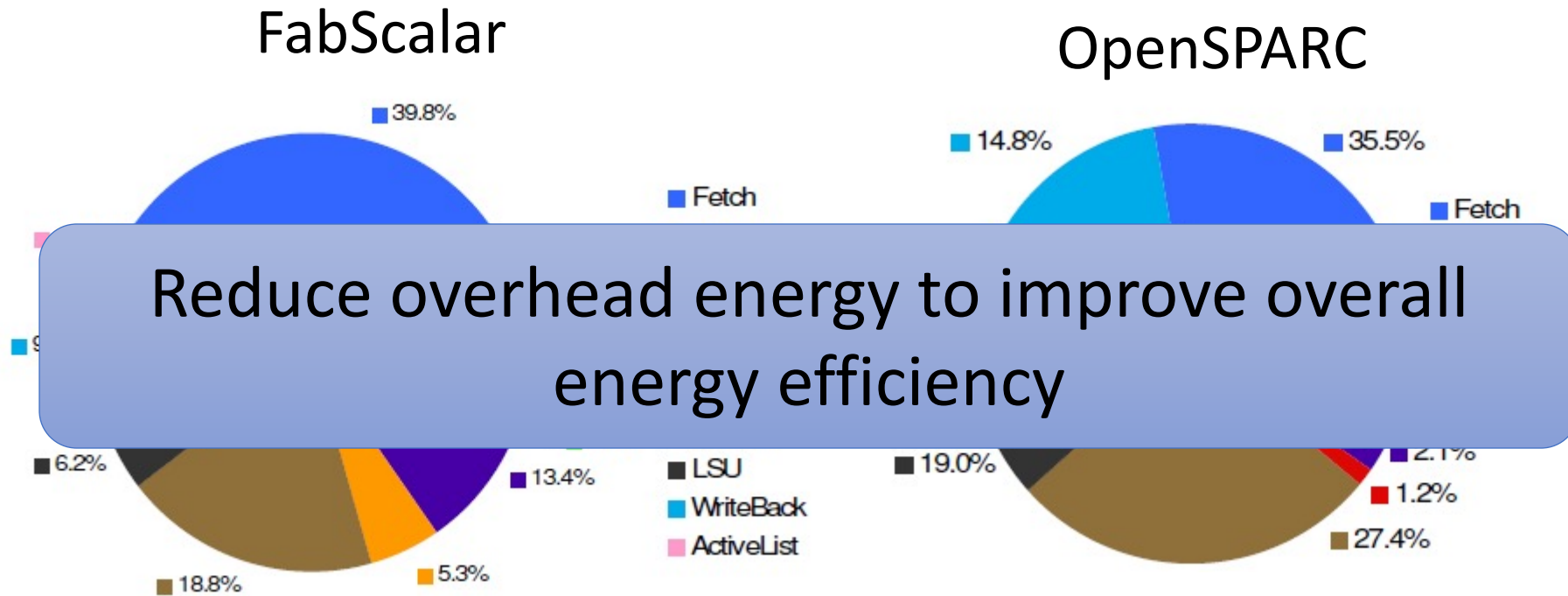
Goal 4: Full Application (multiple IPs)



## **Understanding the sources of inefficiency**



# Where energy consumed?



**Actual execution consumes only a fraction of energy**

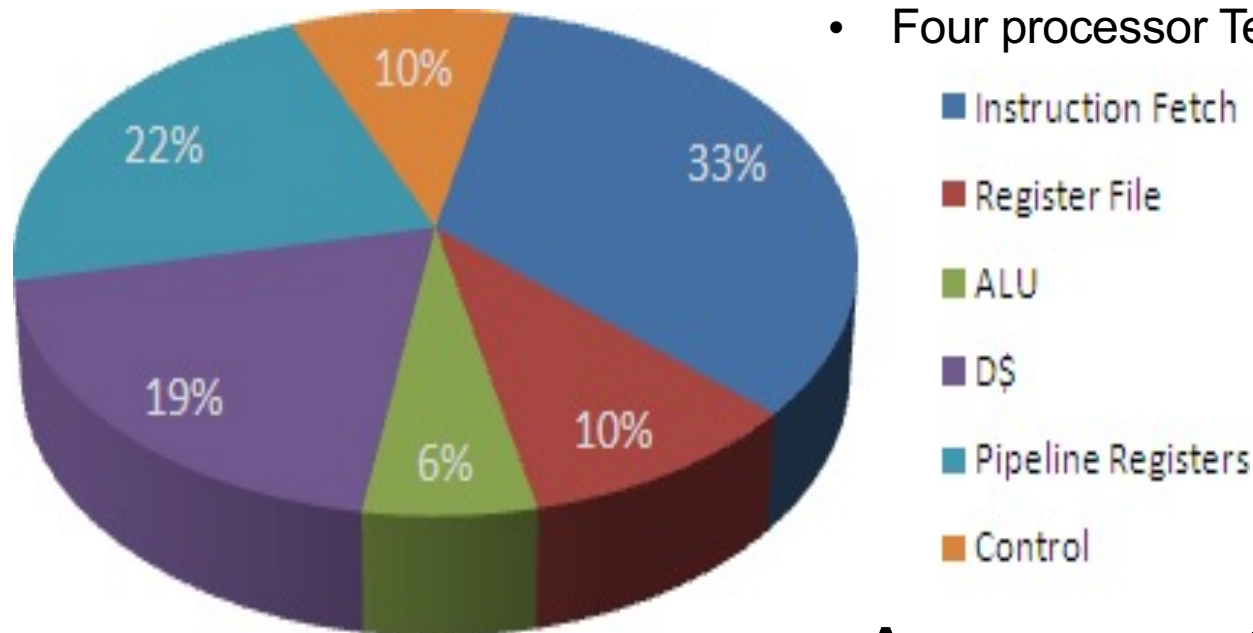
Data is from "Power balanced pipelines" Sartori et al. in HPCA 2012



# Multicore Energy Breakdown

## For HD H.264 encoder

- 2.8GHz Pentium 4 is 500x worse in energy\*
- Four processor Tensilica based CMP is also 500x worse in energy\*



## Assume everything but functional unit is overhead

- Only 20x improvement in efficiency

\* Chen, T.-C., et al., "Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder," Circuits and Systems for Video Technology, IEEE Transactions on, vol.16, no.6, pp. 673-688, June 2006.



# Achieving ASIC Efficiencies: Getting to 500x

## Need basic ops that are extremely low-energy

- Function units have overheads over raw operations
- 8-16 bit operations have energy of sub pJ
  - Function unit energy for RISC was around 5pJ

## And then don't mess it up

- “No” communication energy / op
  - This includes register and memory fetch
- Merging of many simple operations into mega ops
  - Eliminate the need to store / communicate intermediate results



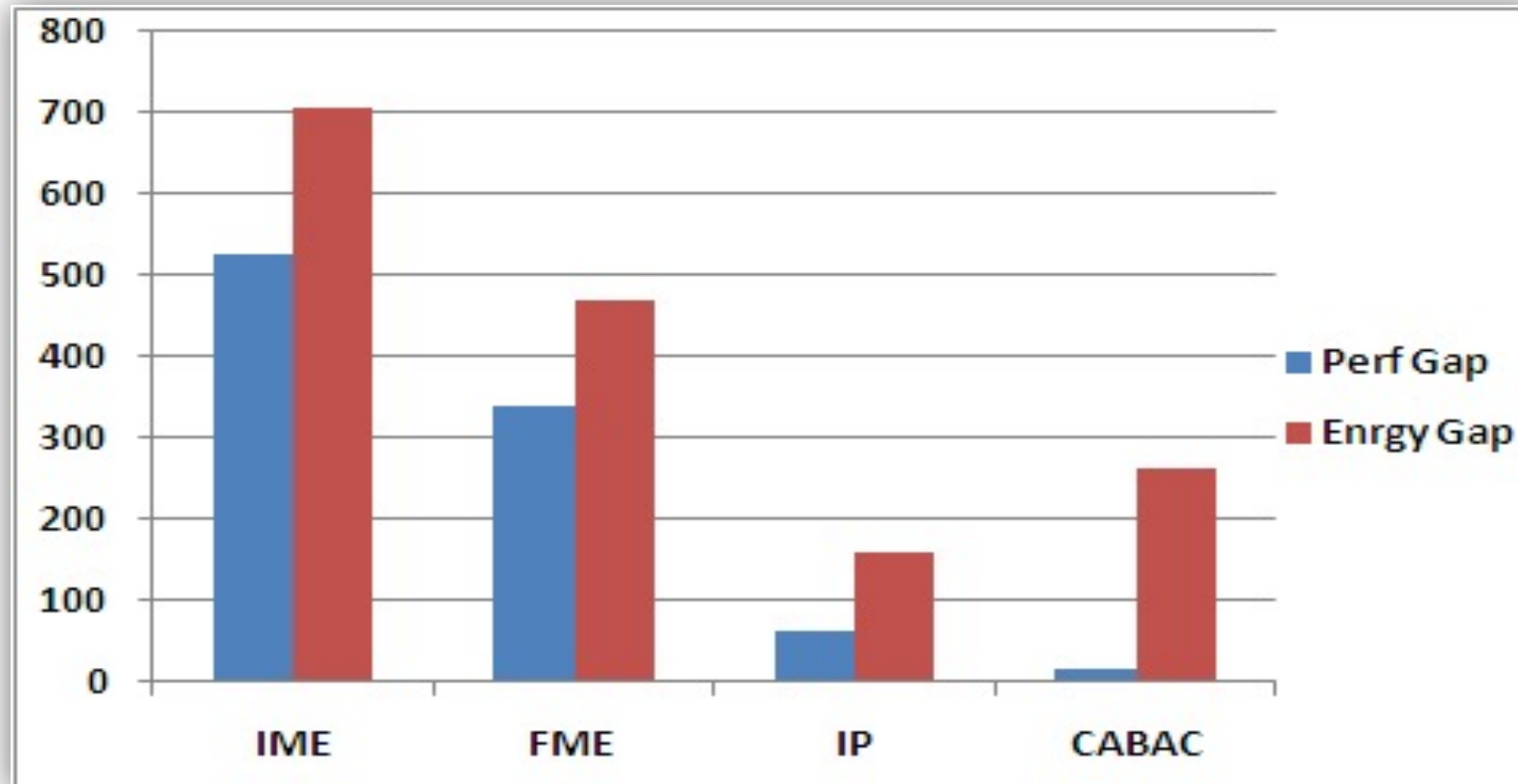
# Multicore vs. ASIC

## Huge efficiency gap

- 4-proc CMP 250x slower
- 500x extra energy

## Manycore doesn't help

- Energy/frame remains same
- Performance improves

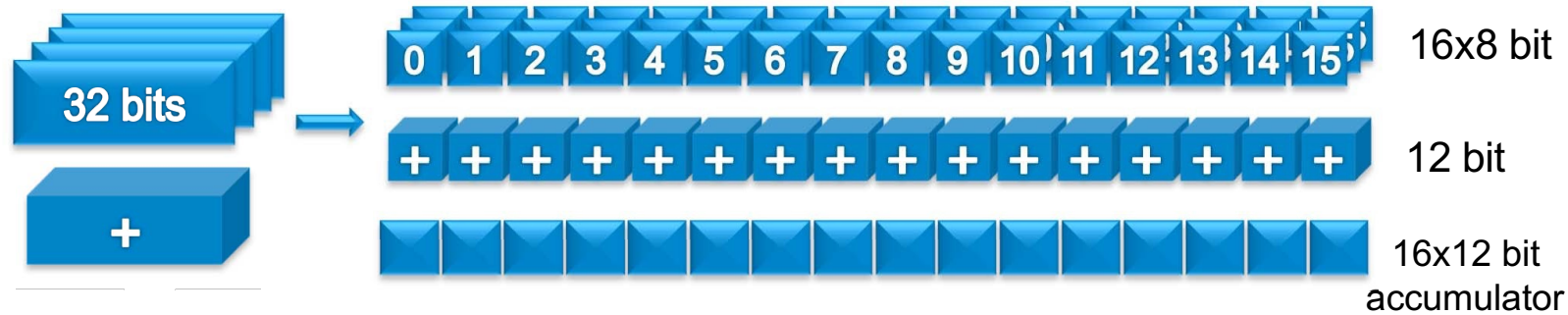




# Opt 1: SIMD, VLIW and Horizontal Fusion

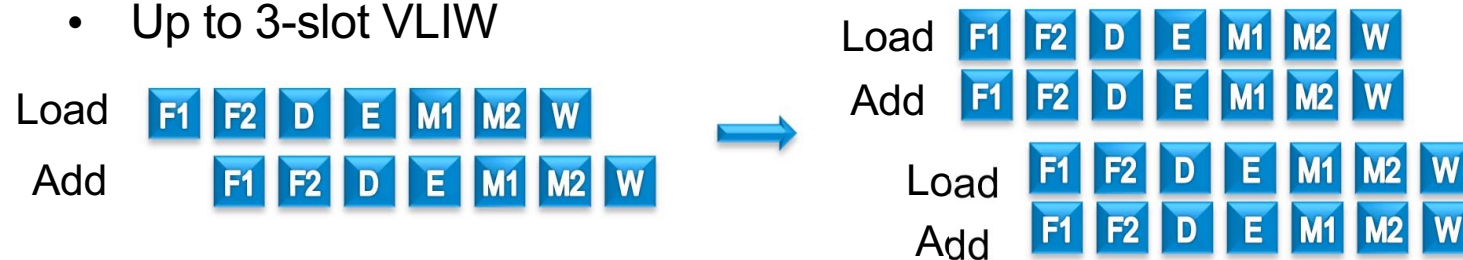
## SIMD

- Up to 18-way SIMD in reduced precision



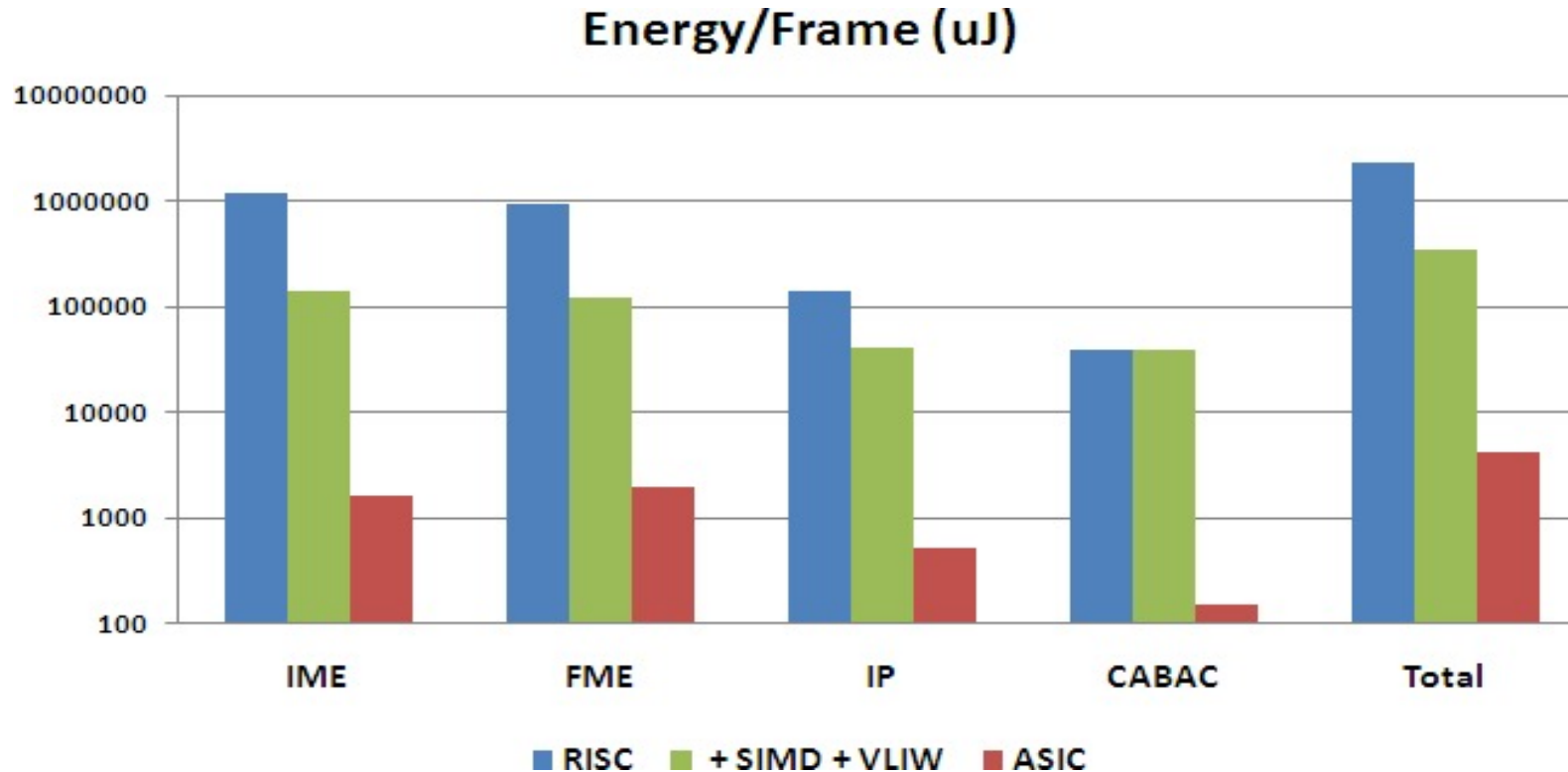
## VLIW

- Up to 3-slot VLIW





# SIMD and ILP - Results

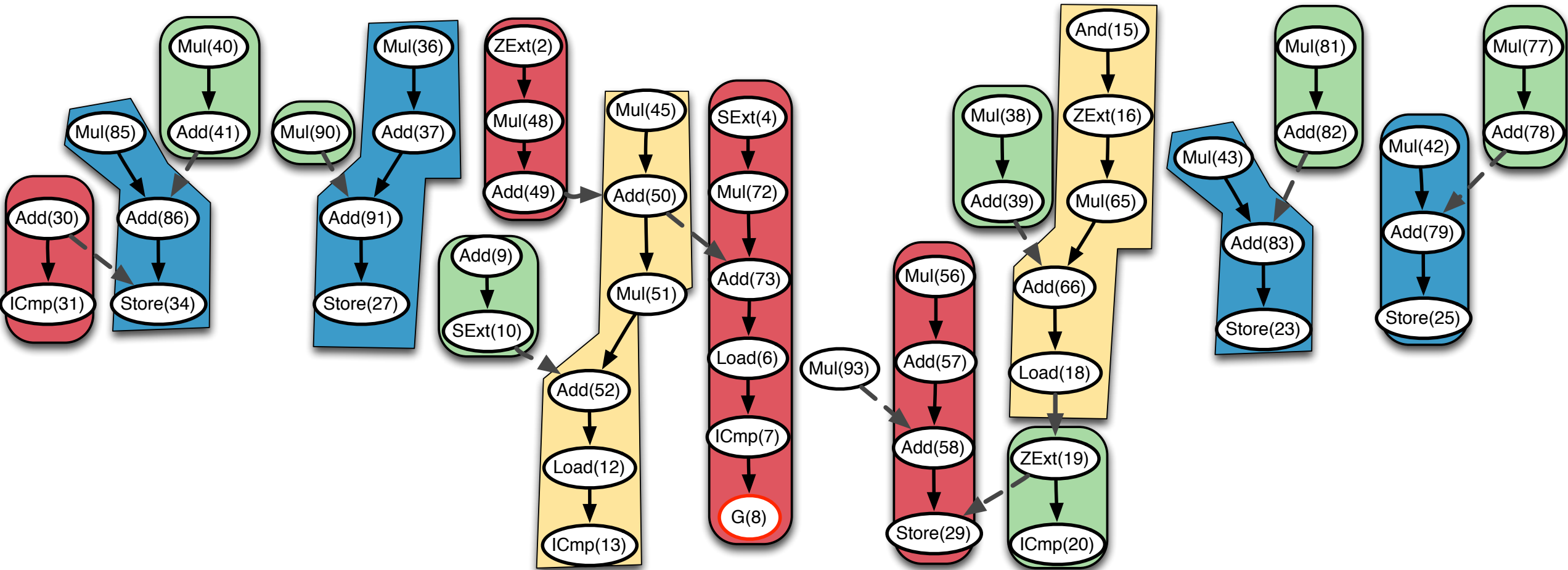


## Order of magnitude improvement in performance, energy

- For data parallel algorithms
- But ASIC still better by roughly 2 orders of magnitude

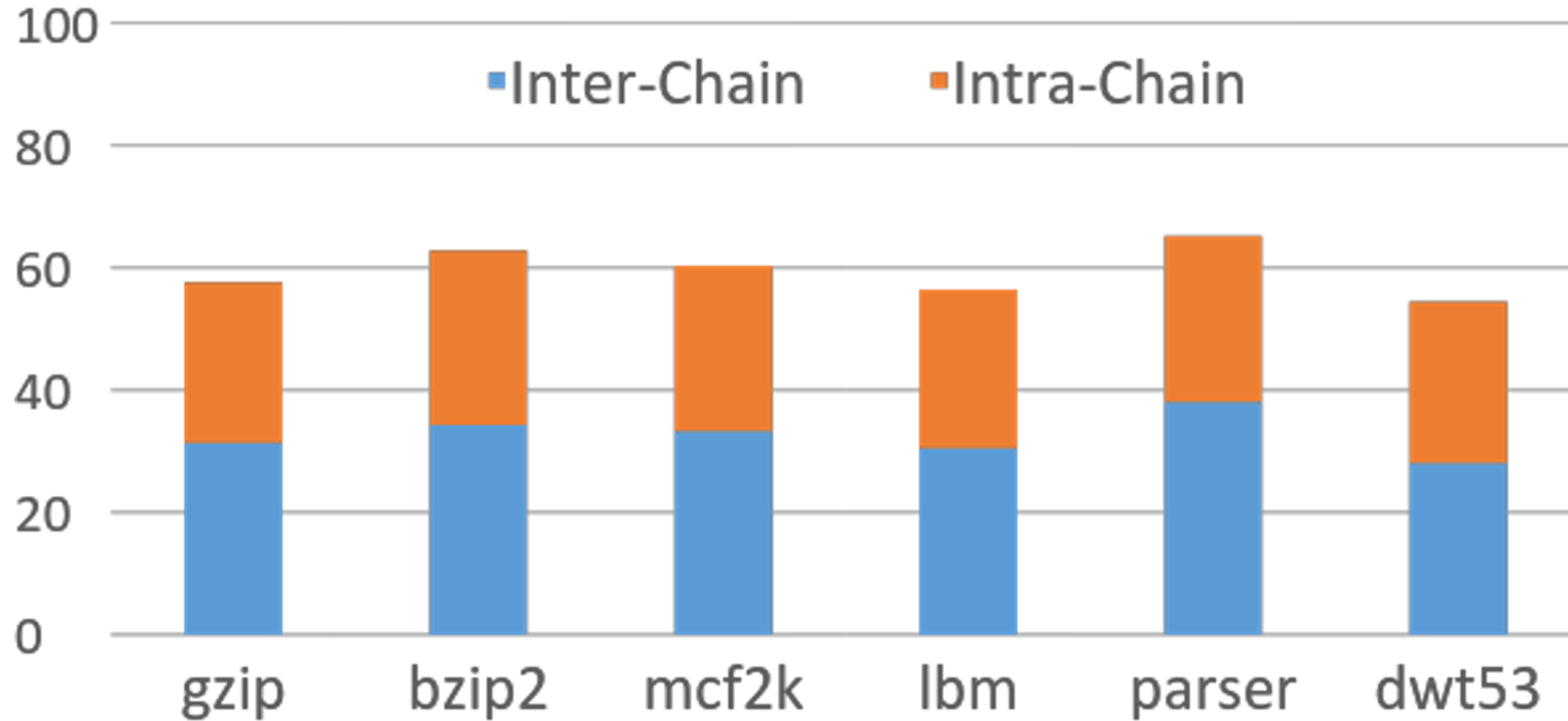


# Opt 2: Op Fusion





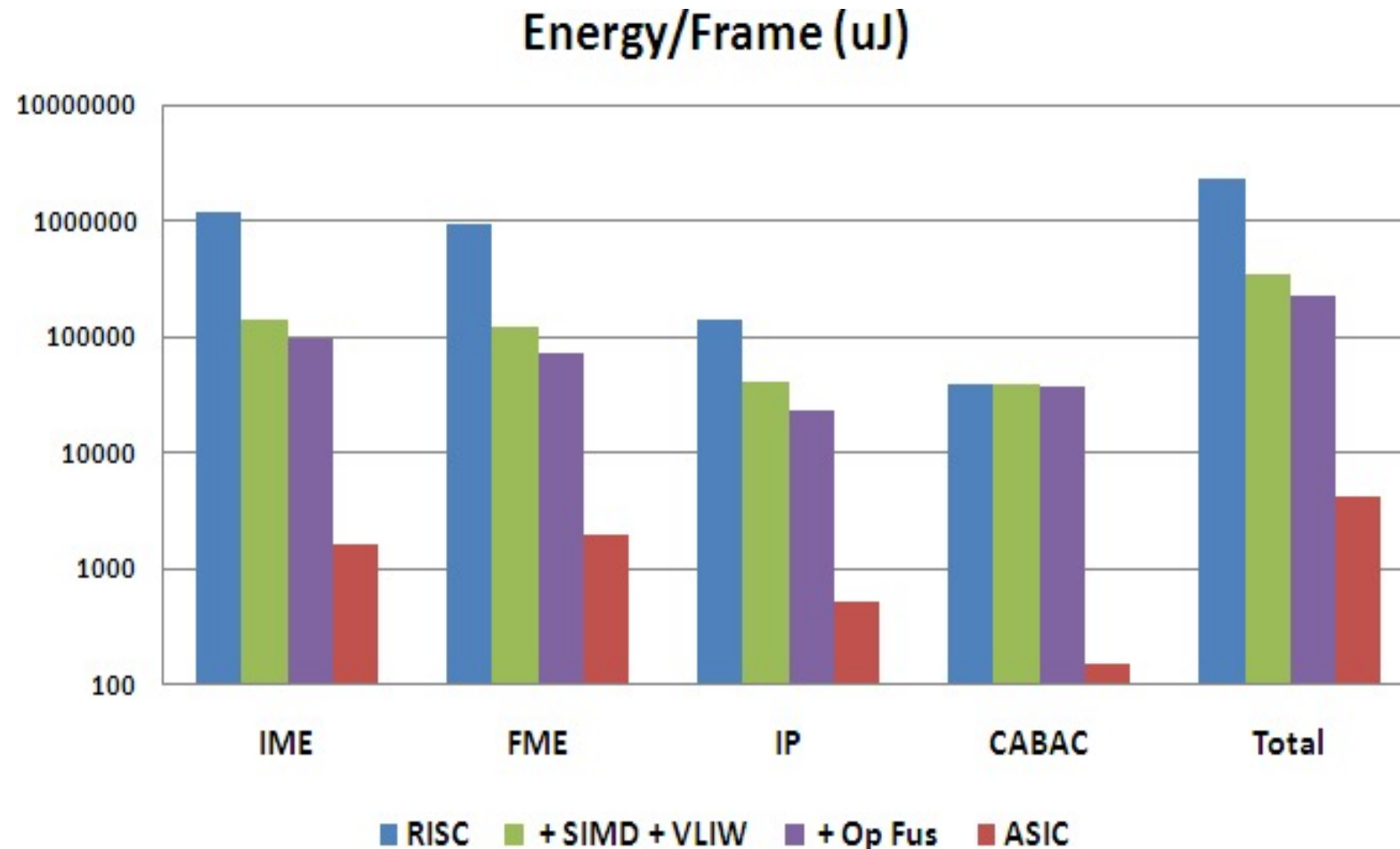
# Opt 2: Op Fusion



Reduces 40% of data movement energy



# Opt 2: Op Fusion



50x less energy efficient and 25x slower ASIC



# Summary: OPT-1 and 2

## Great for data parallel applications

- Improve energy efficiency by 10x over CPU
- Serial phases largely unaffected

## Overheads still dominate

- Basic operations are very low-energy
- Even with 15-20 operations per instruction, get 90% overhead
- Data movement dominates computation

## To get ASIC efficiency need more compute/overhead

- Find functions with large compute/low communication
- Aggregate work in large chunks to create highly optimized FUs
- Merge data-storage and data-path structures



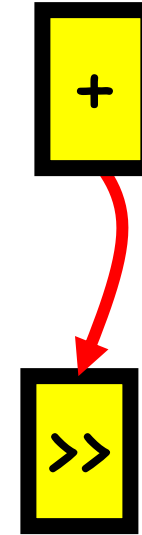




# “Magic” Instructions

## Create specialized data storage structures

- Require modest memory bandwidth to keep full
- Internal data motion is hard wired
- Use all the local data for computation



**Arbitrary new low-power compute operations   Large effect on energy efficiency and performance**

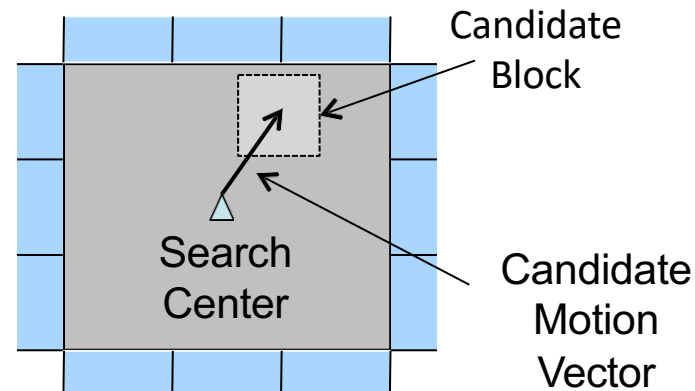


# Magic Instructions – SAD Example

$$sum = sum + abs(x_{ref} - x_{cur})$$

## Looking for the difference between two images

- Hundreds of SAD calculations to get one image difference
  - Need to test many different position to find the best
- Data for each calculation is nearly the same

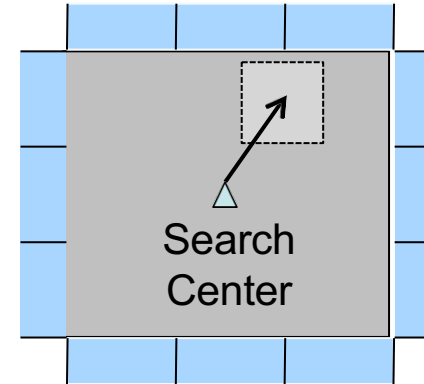




# Magic Instructions – SAD Example

## SIMD implementation

- Limited to 16 operations per cycle
- Horizontal data-reuse requires many shift operations
- No vertical data reuse means wasted cache energy
- Significant register file access energy

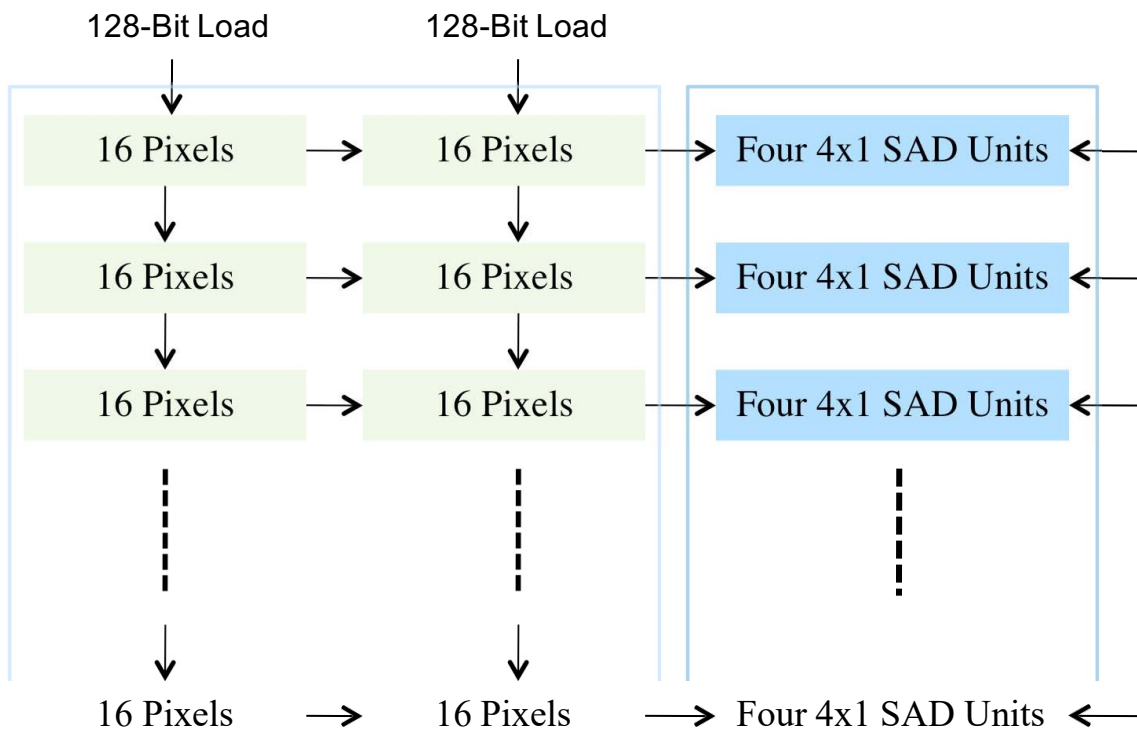


## Magic – *Serial in, parallel out structure*

- Enables 256 SADs/cycle which reduces fetch energy
- Vertical data-reuse results in reduced DCache energy
- Dedicated paths to compute reduce register access energy



# Custom SAD instruction Hardware



## Reference Pixel Registers:

Horizontal and vertical shift with parallel access to all rows

256 SAD Units

## Optimization strategy similar across all algorithms

- Closely couple data storage and data path structures
- Maximize data reuse inside the datapath

## Commonly used hardware structures and techniques

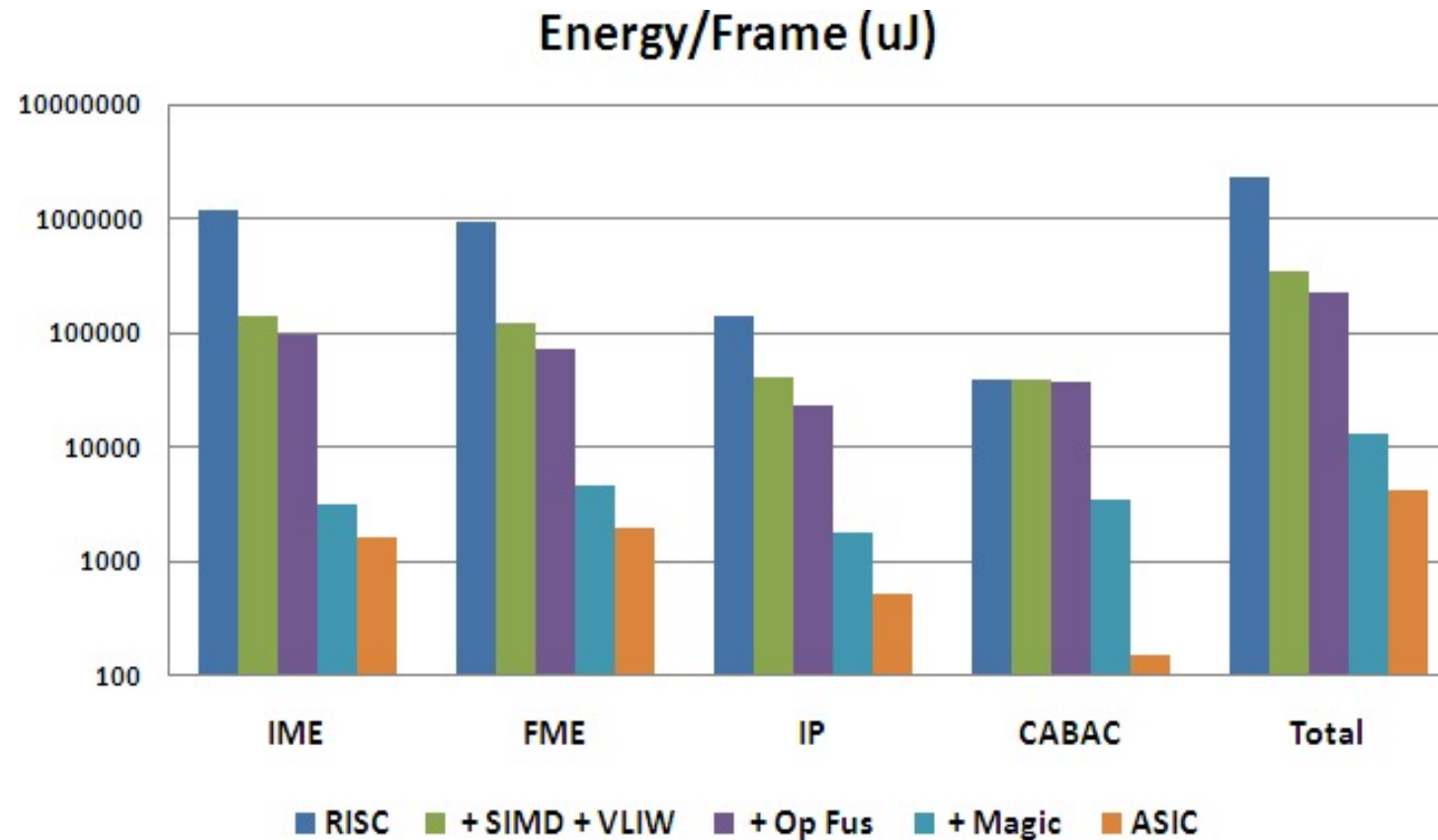
- Shift registers with parallel access to internal values
- Direct computation of the desired output
  - Eliminate the generation (and storage) of intermediate results

**Hundreds of extremely low-power ops per instruction**

**Works well for both data parallel and sequential algorithms**



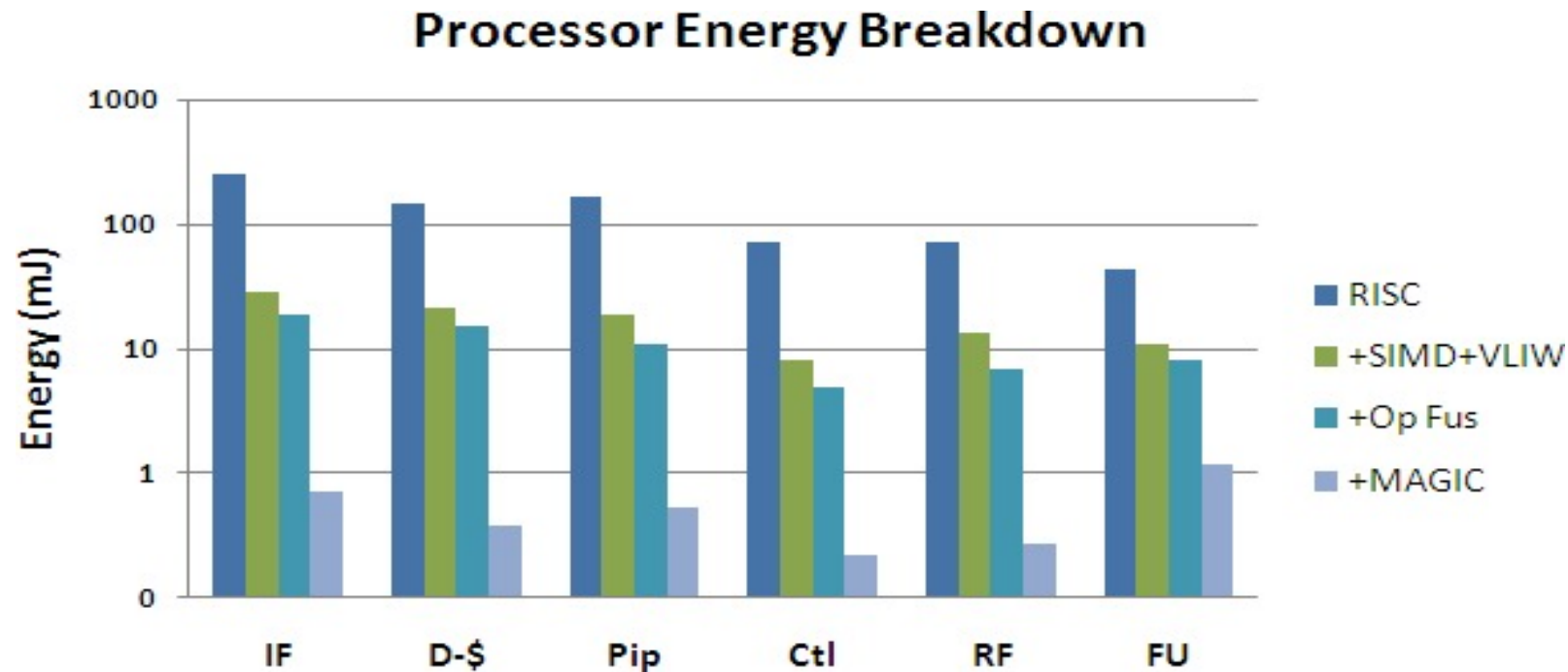
# Magic Instructions - Energy



Efficiency orders of magnitude better than GP  
Within 3X of ASIC energy efficiency



# Magic instructions - Results



## Over 35% energy now in ALU

- Overheads are well-amortized – *up to 256 ops / instruction*
- More data re-use within the data-path

**Most of the code involves magic instructions**



**Why OOOs suck.**

**Is technology scaling dead/dying ?**

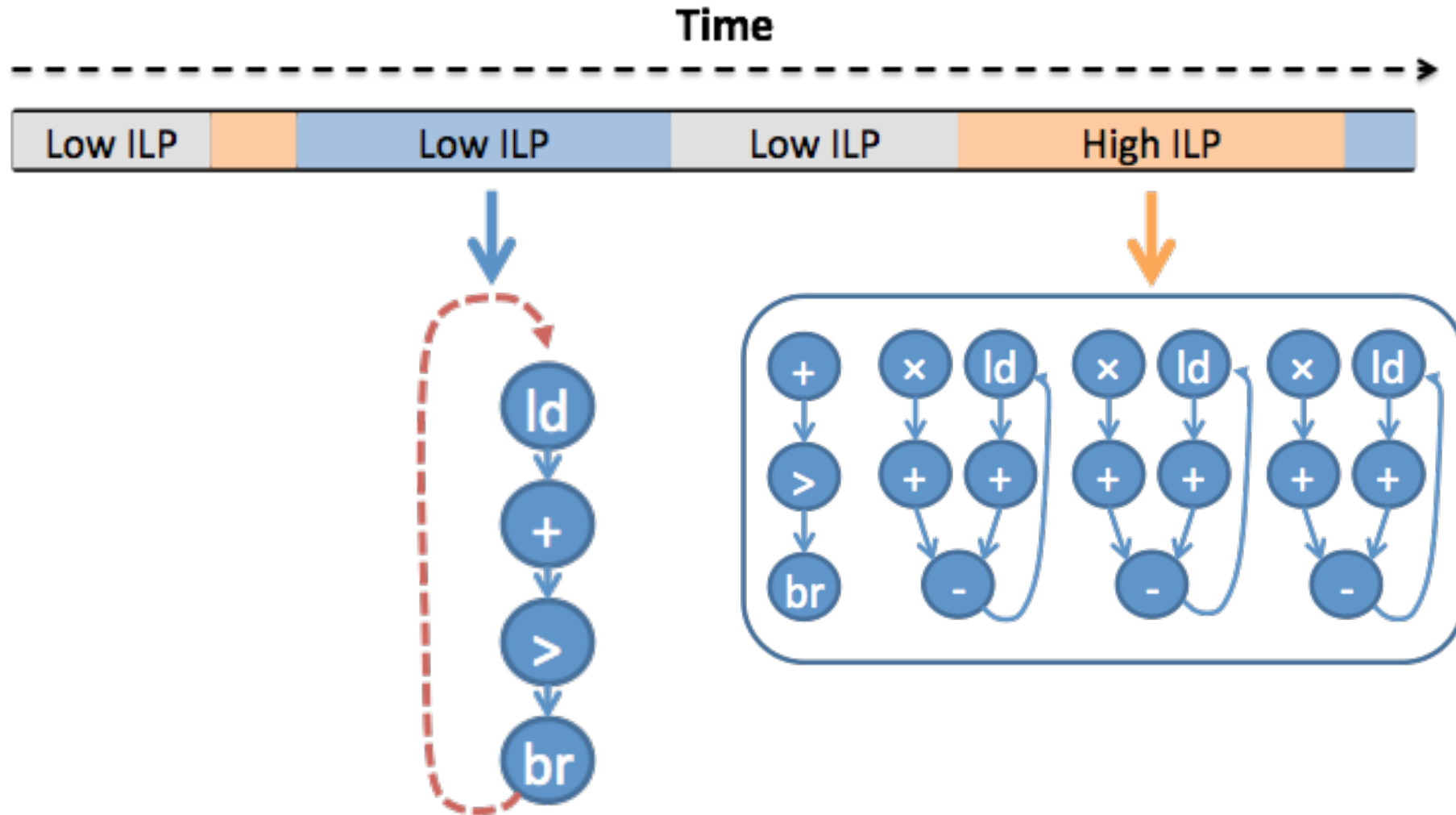
**Are DSAs/Accelerators The Solution?**



# What are Accelerators?

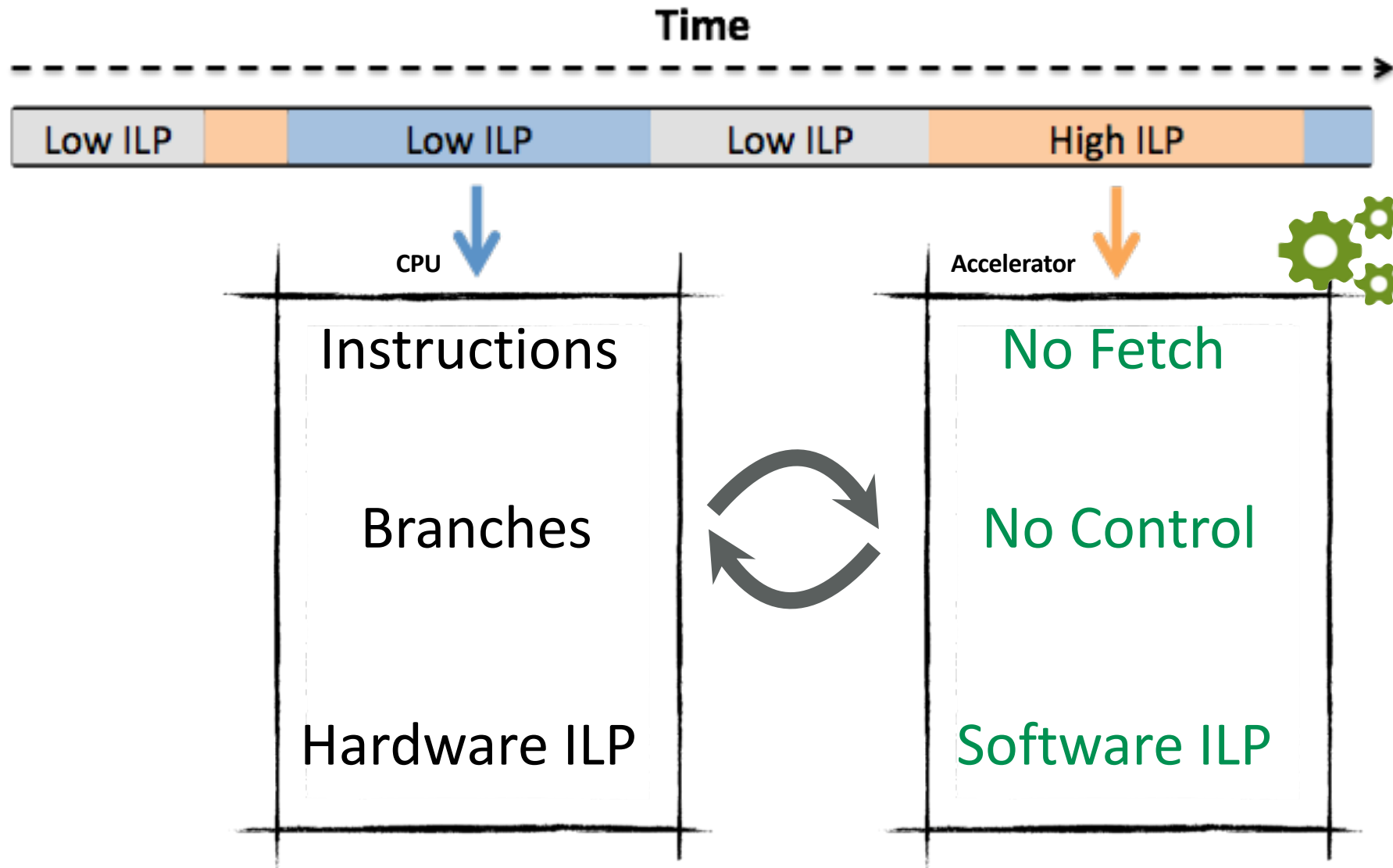


# What are Accelerators?





# What are Accelerators?

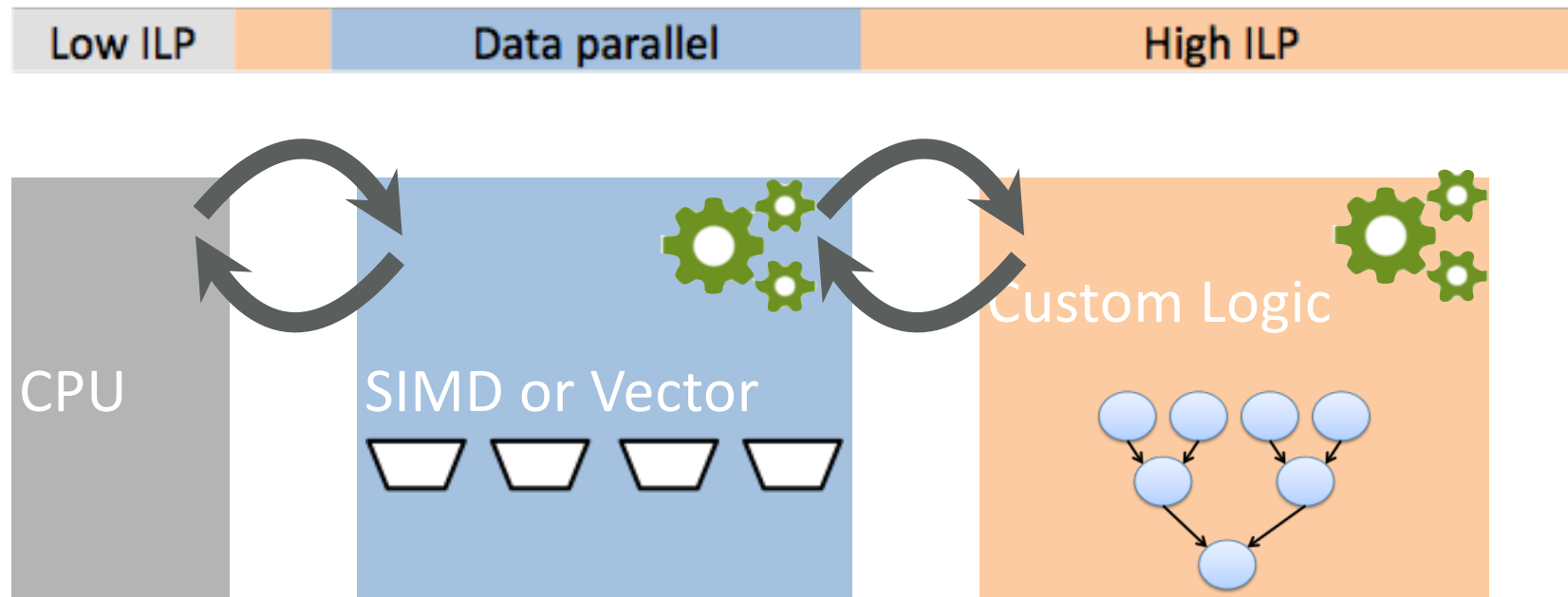


[Intel Harp, IBM CAPI, ARM Big-Little, BERET, DYSER, CCORE]



# Accelerator Execution

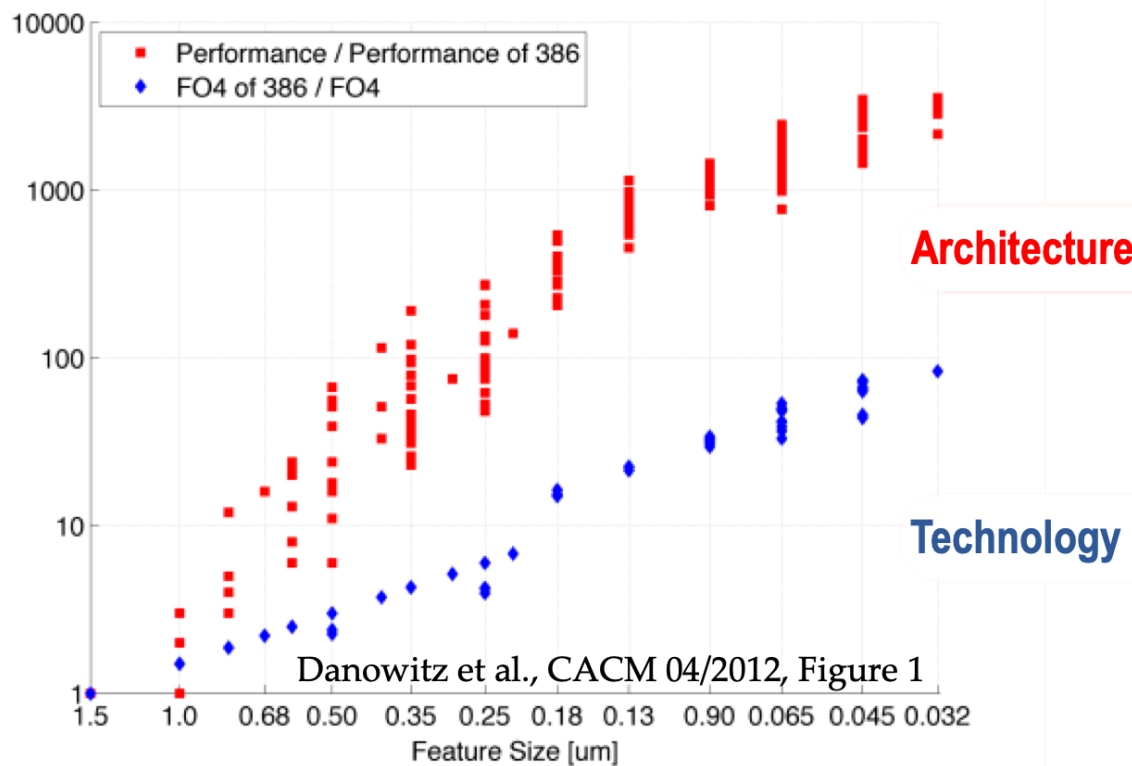
- Hope! Large acceleratable program regions





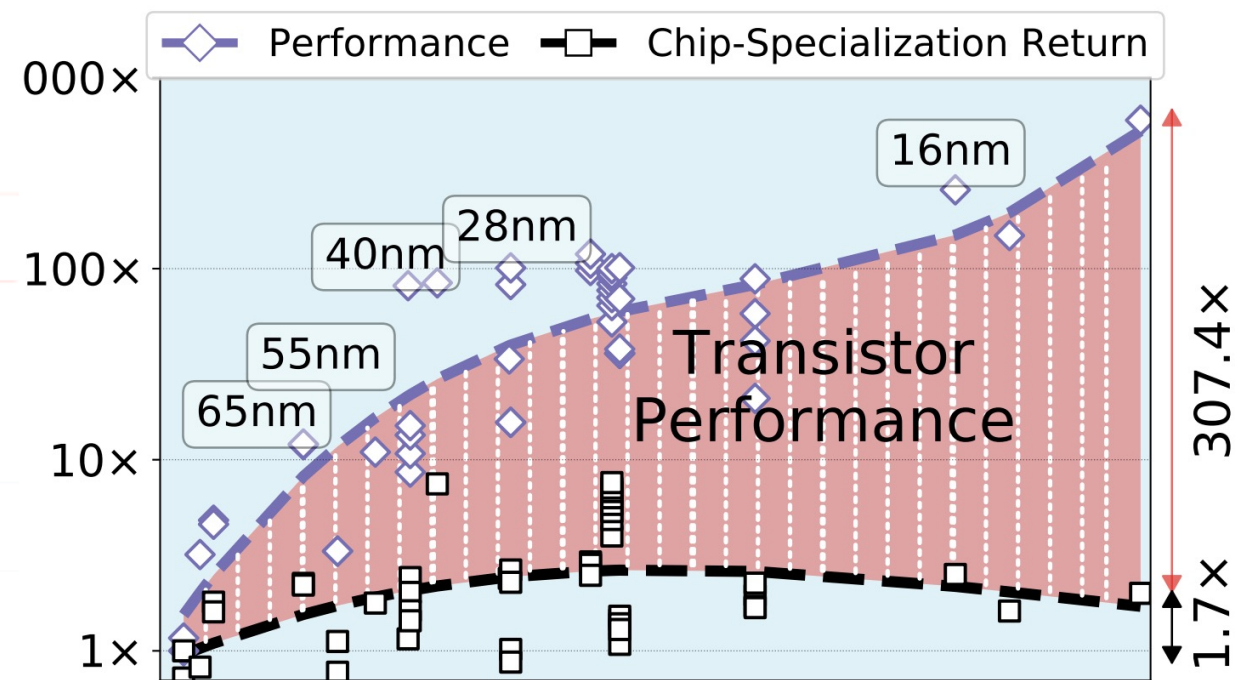
# We are just getting started

## Pre-Moore



~50% every two years

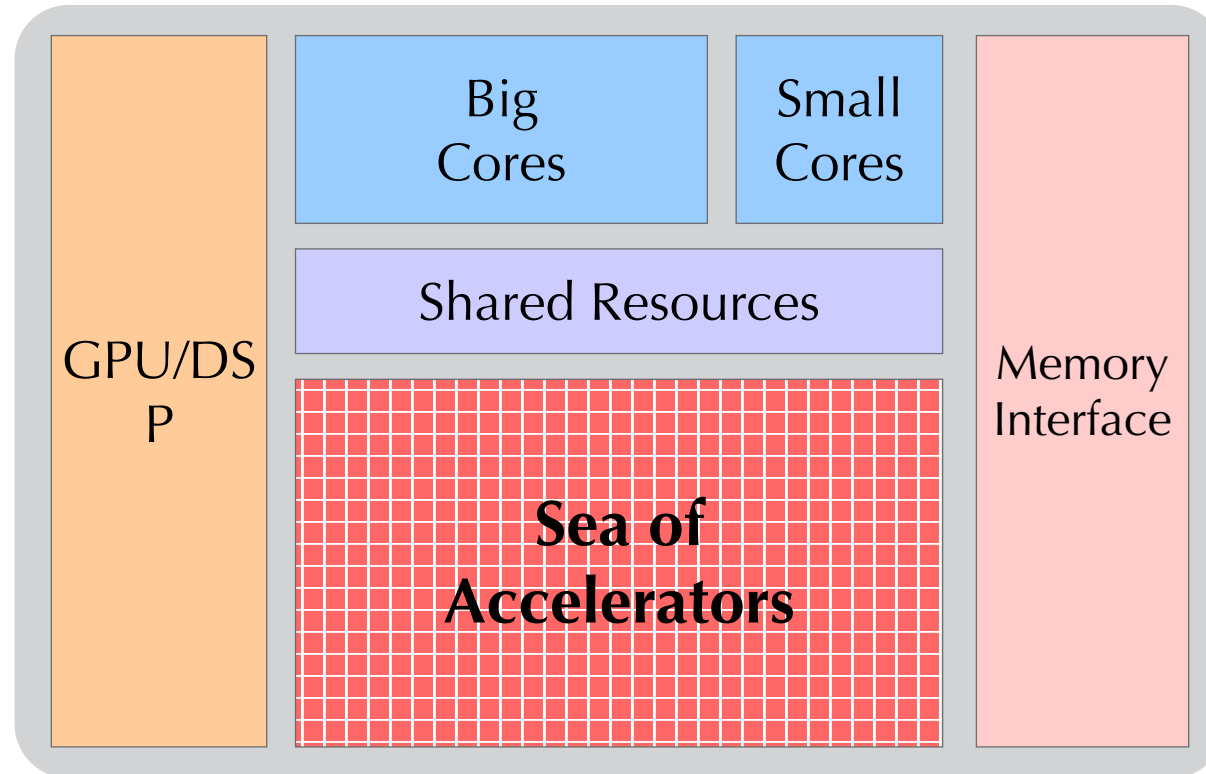
## Post-Moore



~20% every two years  
after initial 100x



# Future Accelerator-Centric Architectures



How to decompose applications into accelerators?

How to rapidly design lots of accelerators?

How to design and manage the shared resources?

↑ Flexibility

↓ Design Cost

↑ Programmability



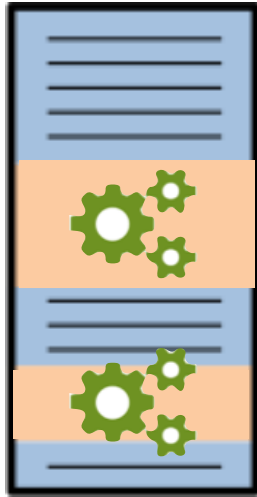
# Why does it work?

- Applications execute in phases
- Applications follow 90-10 rule
  - 10% of code-region contributes to 90% of run time
- *Creating specialization for such code-regions amortizes the overheads*
- Removing instructions from main pipeline
  - Less use of Instruction Queue, ROB, Register File
  - Effectively larger instruction window
- Decoupled Execution
  - Concurrency between main processor and CGRA
  - Many FUs -> High Potential ILP
- Benefits of Vectorization
  - Fewer memory access instructions
  - Explicit pipelining of CGRA



# How can software help accelerators?

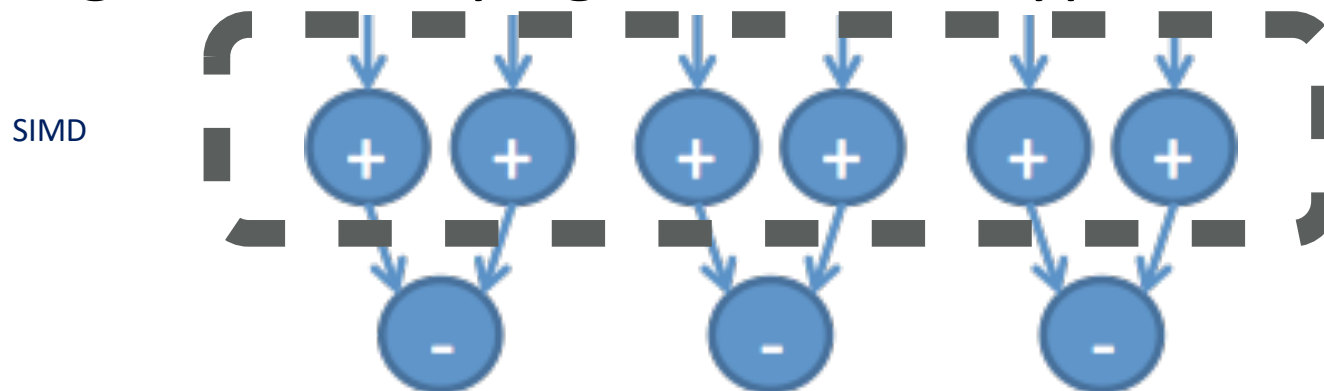
- Challenge 1: Find acceleratable programs regions



Control not supported (need SW help)

Mem. ops not supported (self prophecy?)

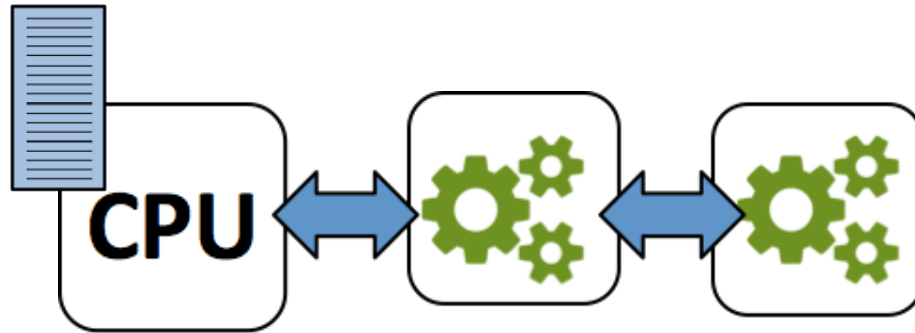
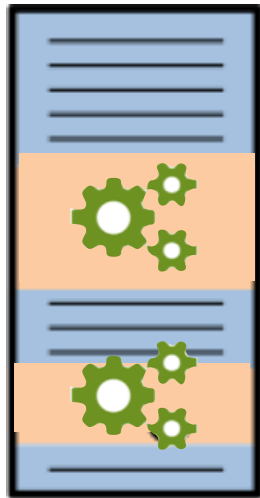
- Challenge 2: Identifying accelerator types





# How can software define accelerators?

- Challenge 3: How to compose accelerators?





# Accelerator Granularity

FPGA

Algorithm

GPUs

Threads

---

**Onchip-FPGA**

Extended Basic Blocks

**Loop  
Accelerators**

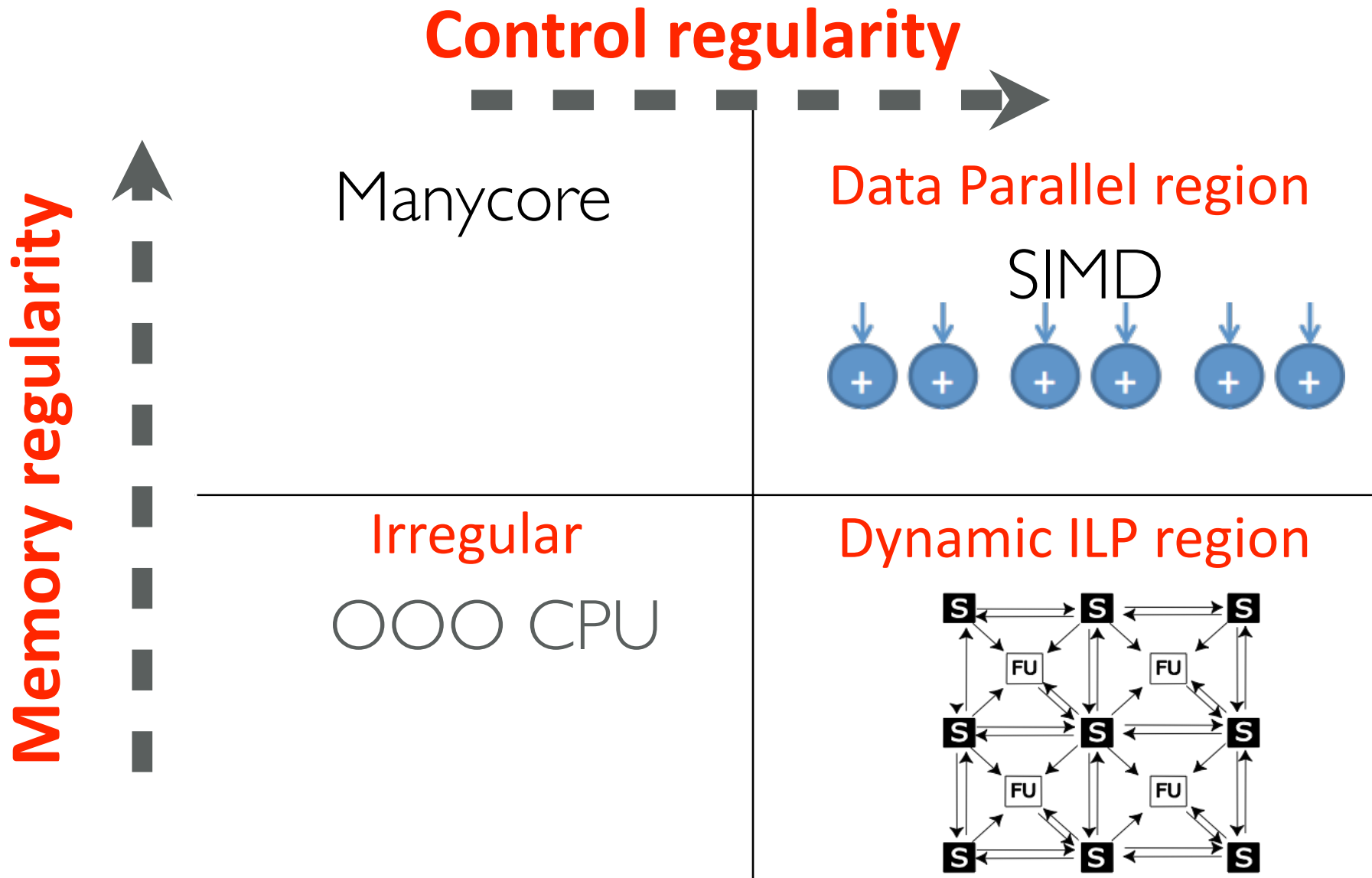
Program loops

**SIMD**

Instructions



# Types of Accelerators?





**Domain Specific Architecture =  
Compiler-Driven Spatial Hardware**



# Our History

MICRO'16

ISCA'15

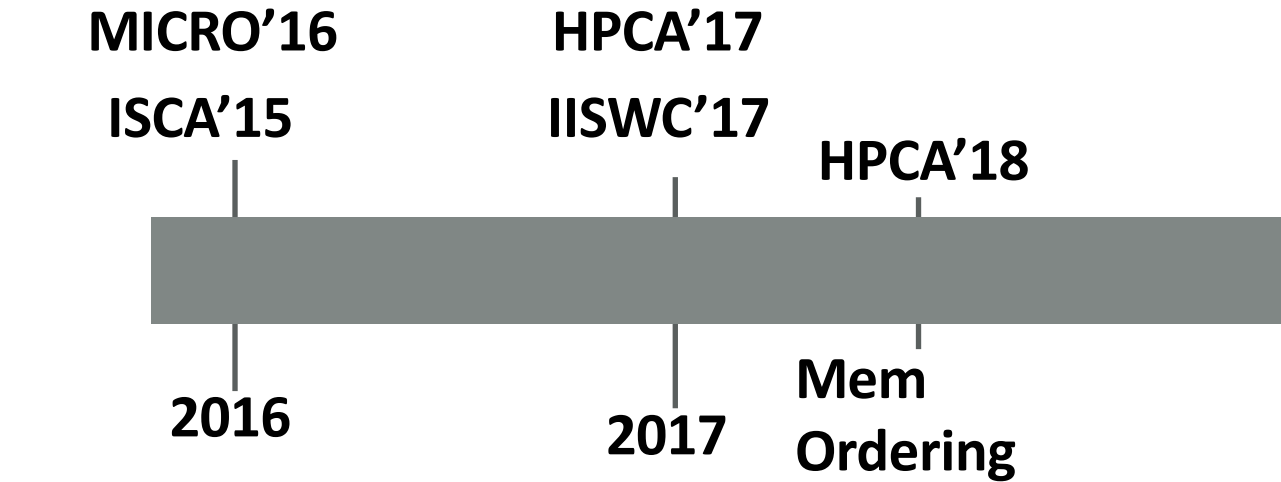


2016

Lets accelerate  
something



# Our History



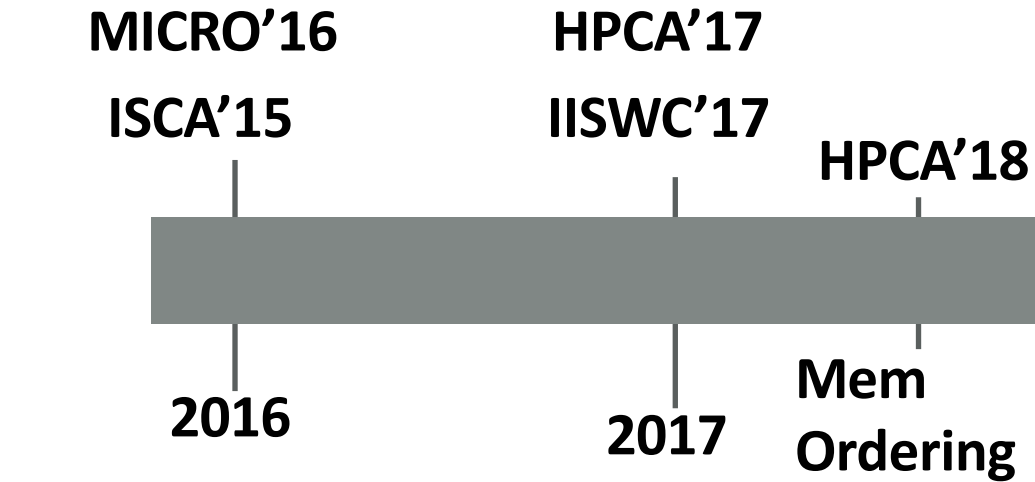
Lets accelerate  
something

What should  
we accelerate?

Compiler front-end work  
for FPGA toolchains



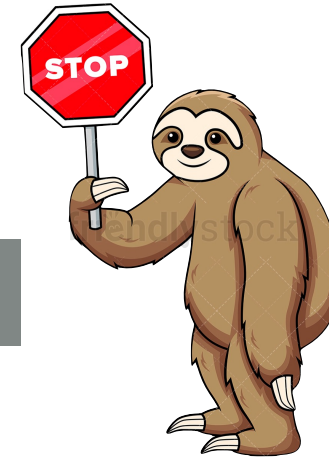
# Our History



Lets accelerate  
something

What should  
we accelerate?

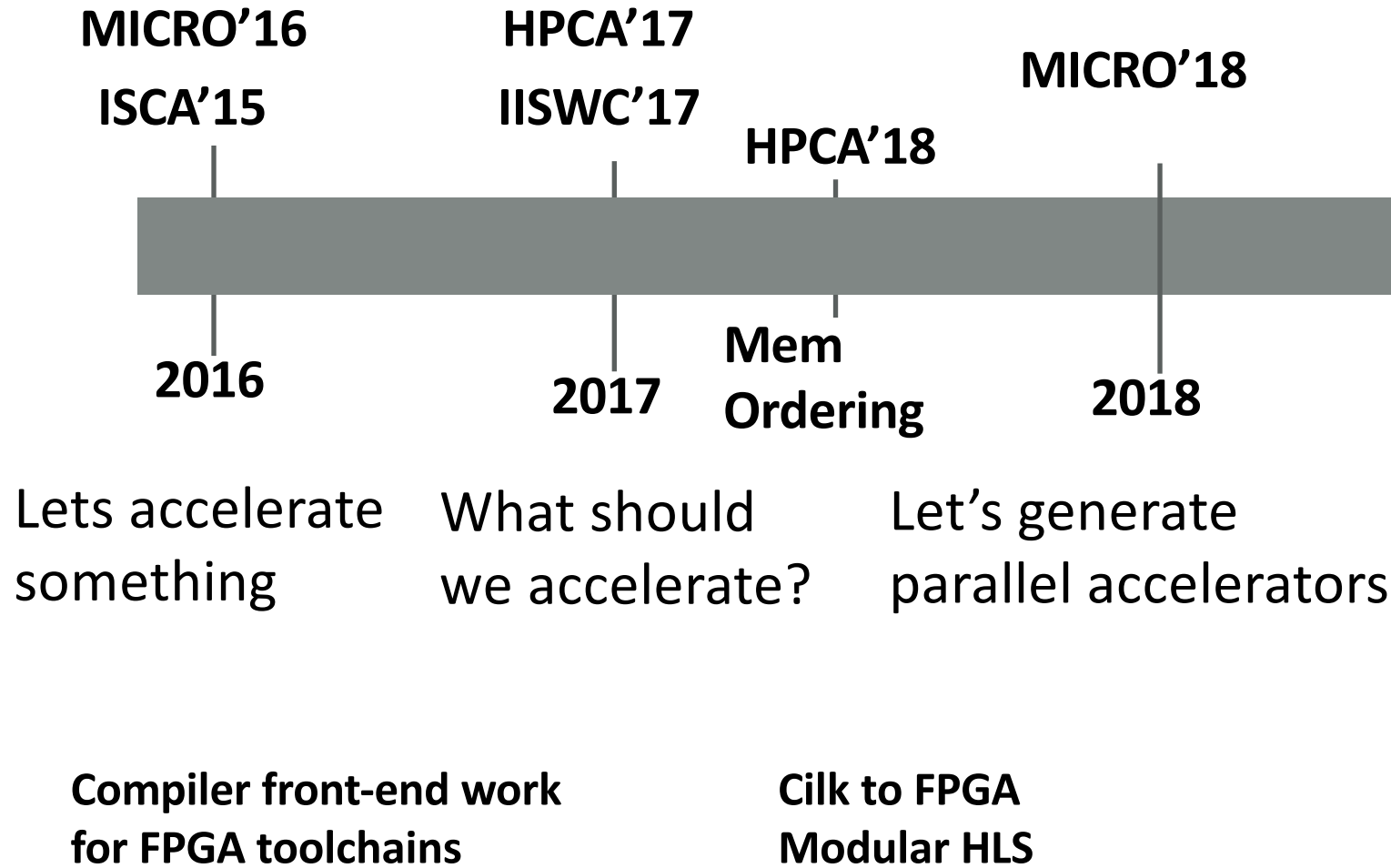
Compiler front-end work  
for FPGA toolchains



Community moving  
towards DSLs

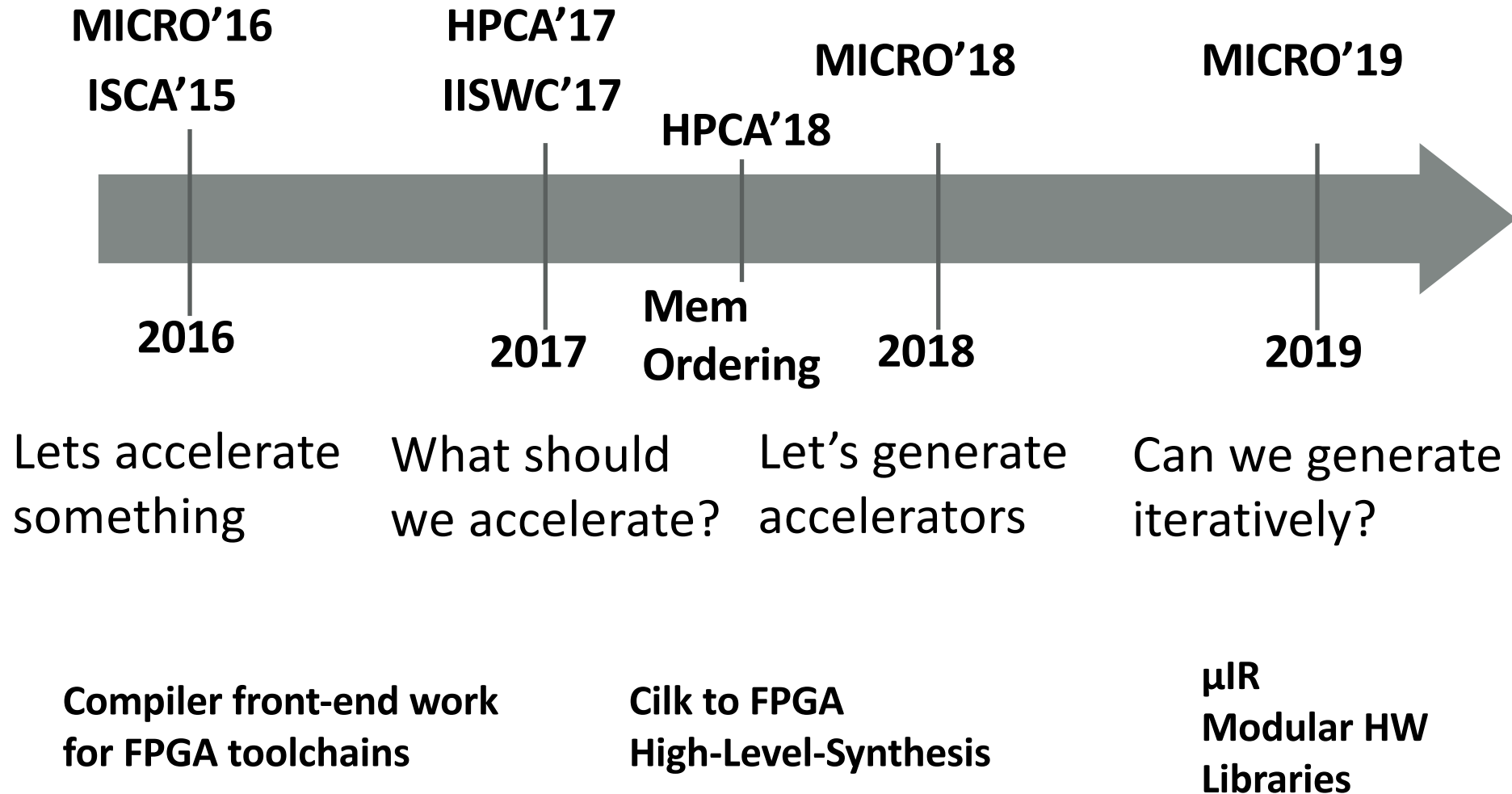


# Our History



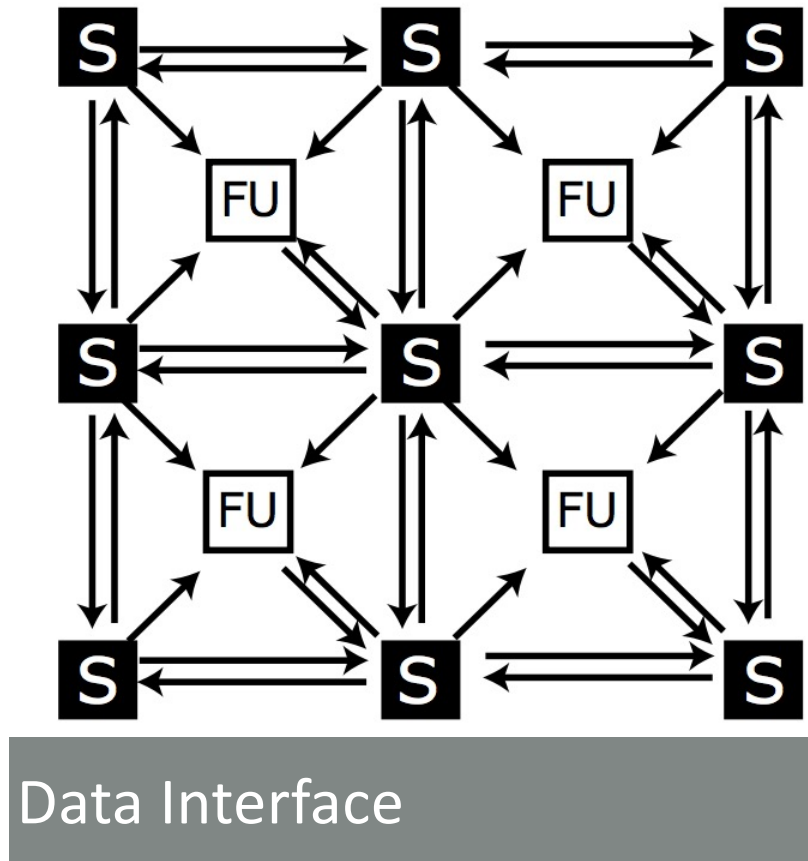


# Our History





# Architecture Template 1 (Dataflow)



- Target: Line-rate processing
- Spatial Dataflow
  - No control (e.g., branches, loops )
- Instruction grain - ALU ops
- Stateless fabric
  - (no register file)
- Logically single line-buffer

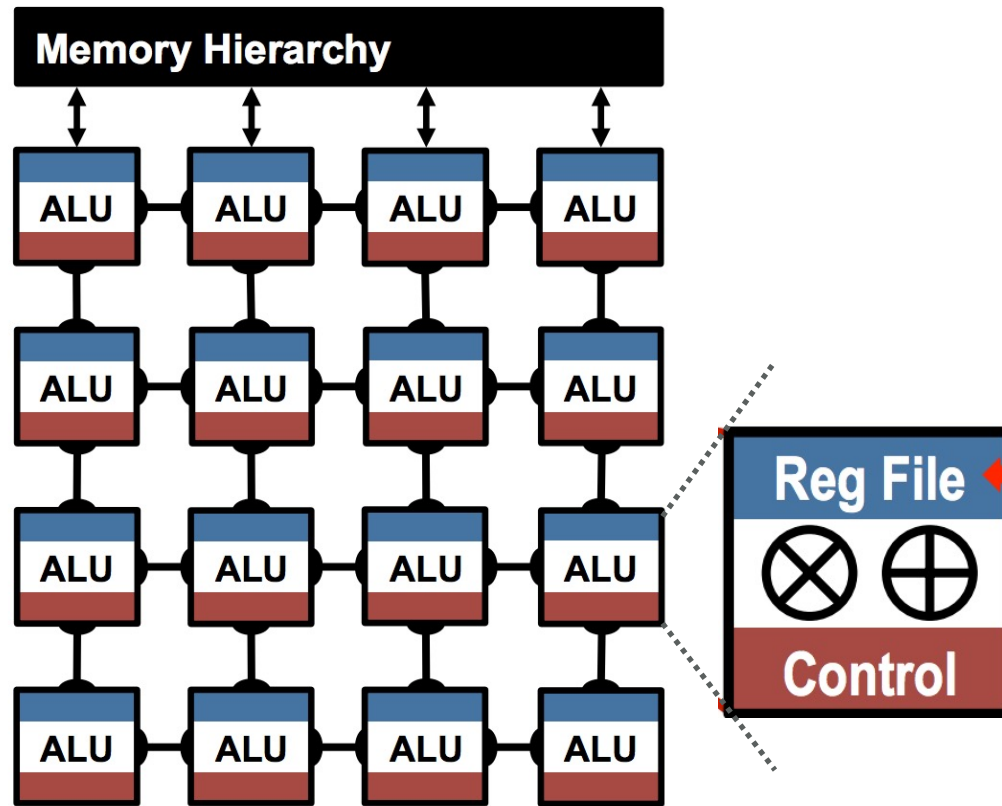
Related:

Academia: CGRA-ME, Dyser, CGRA-express,

Industry: Xilinx CGRA overlays, Wireless baseband?



# Template 2 (CGRA-based)



- Target: Image-Processing, Machine Learning, DNN,
- Spatio-Temporal Dataflow
  - Rich control (if-else, loops, nested loops )
- Kernel Grain
- Stateful fabric
  - Rich register hierarchy
- Multiple line-buffers

**Industry:** <https://ieeexplore-ieee-org/stamp/stamp.jsp?tp=&arnumber=7818353>

Intel early-CGRA (<https://arxiv.org/pdf/1711.07606.pdf>)

Xilinx Versal (DSP/Vector Processor Fabric: [https://www.xilinx.com/support/documentation/data\\_sheets/ds950-versal-overview.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf))

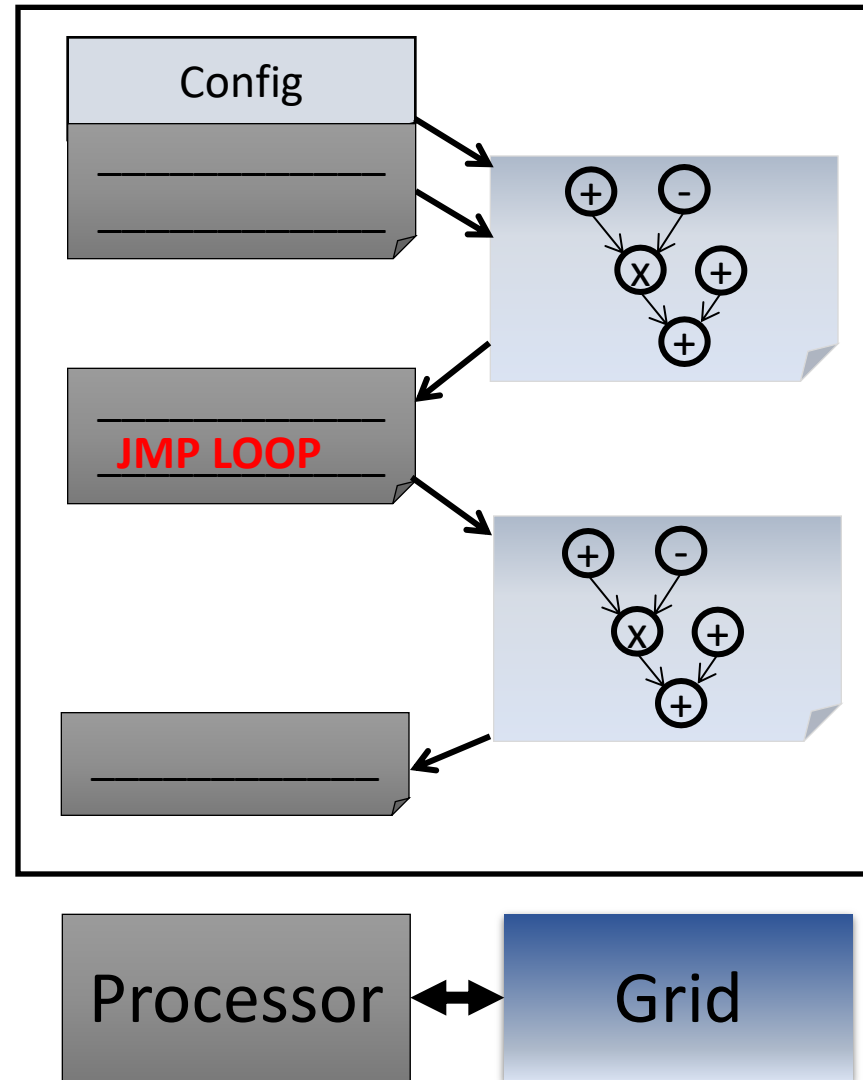
Eyeriss/MIT: <https://arxiv.org/pdf/1807.07928.pdf>

Sambanova and Spatial (<https://www.sambanovasystems.com/news/>)



# Execution Model: Decoupled Access/Execute Model

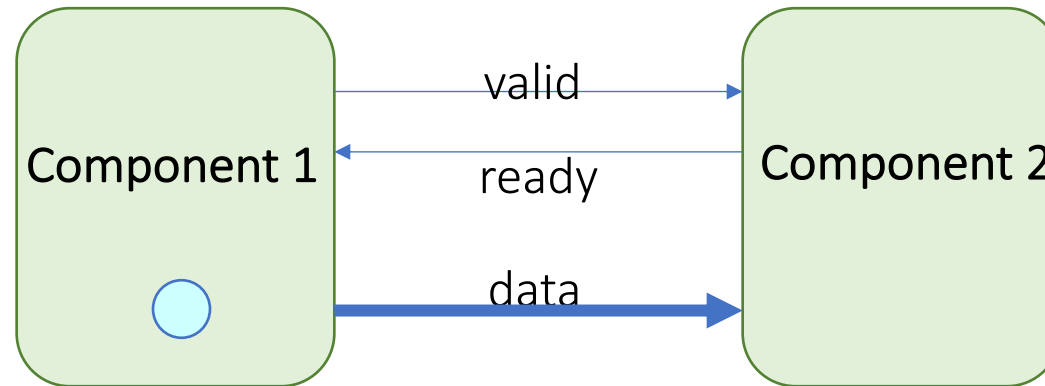
- Memory access instructions execute in processor pipeline
  - Address Calculation, Loads, and Stores
  - Configure
  - Send Data
  - Recv Data
  - Loop control
- Computation executes





# Dataflow Execution

- Implement **dynamic scheduling**
- Every component communicates via a pair of handshake signals
- The data is propagated from component to component as soon as memory and control dependencies are resolved





# Dataflow Execution Model

- Dataflow by nature has write-once semantics
  - Each arc (token) represents a data value
  - An arc (token) gets transformed by a dataflow node into a new arc (token)
- No persistent state...

Eliminates per instruction overheads

No fetch, decode etc.,

No expensive register reads etc.,

High performance itself leads to energy savings

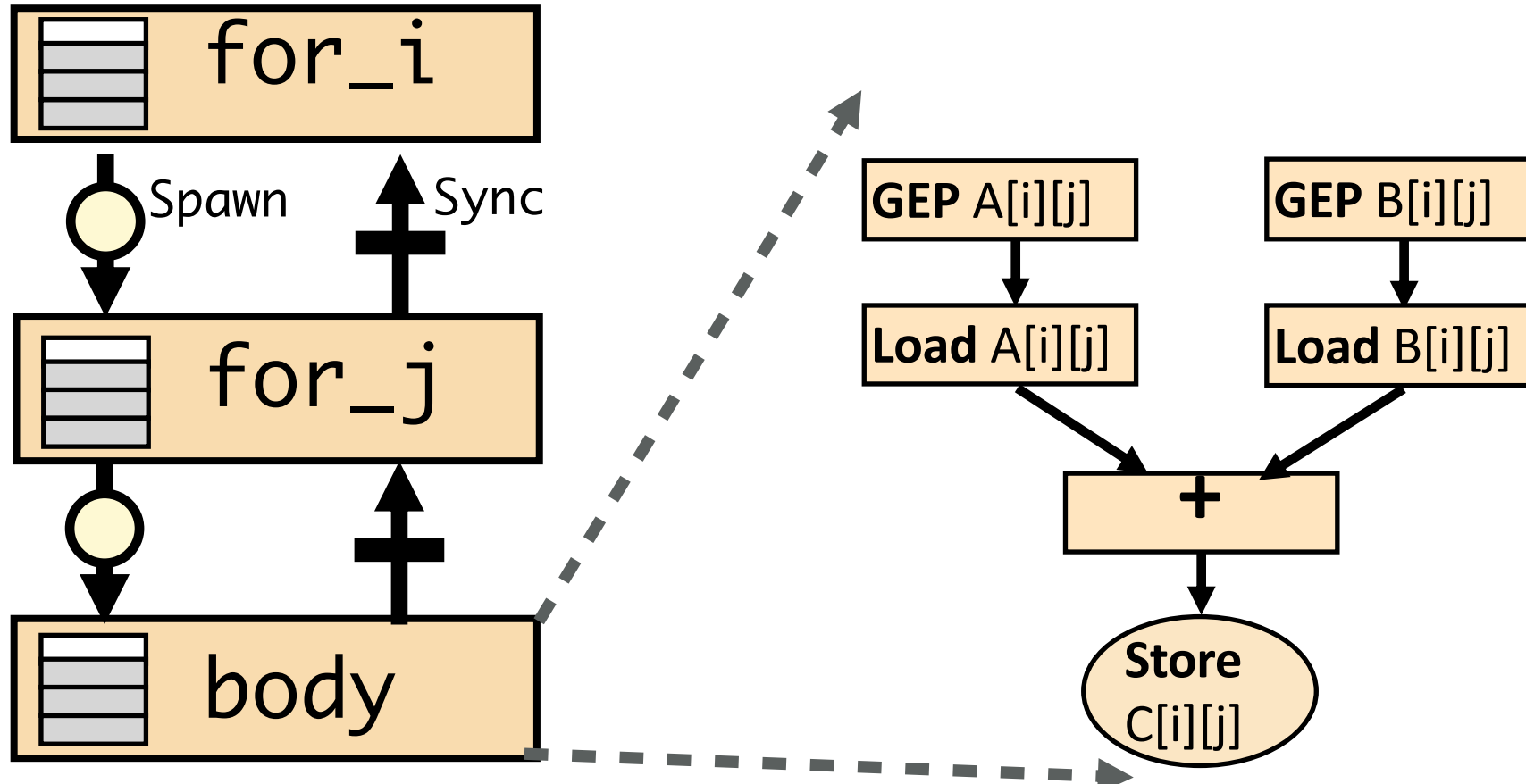
No additional power-hungry structures



# DSAs on one slide

```
parallel_for(i = 0 until n)  
  parallel_for(j = 0 until n)  
    c[i][j] = a[i][j] + b[i][j];
```

Hierarchical Data + Control Dynamic Graph

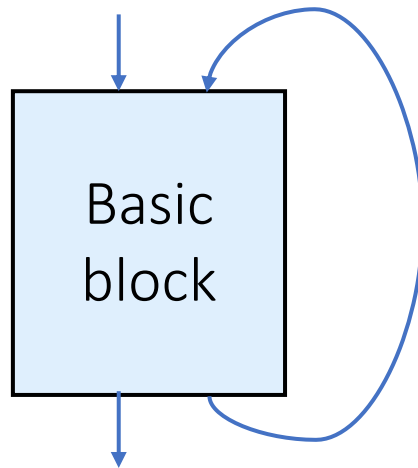




# Dataflow Circuits

- C to intermediate graph representation
  - LLVM compiler framework

```
for (i = 0; i < 100; i++) {  
    a[x[i]] = a[x[i]] * w[i];  
}
```

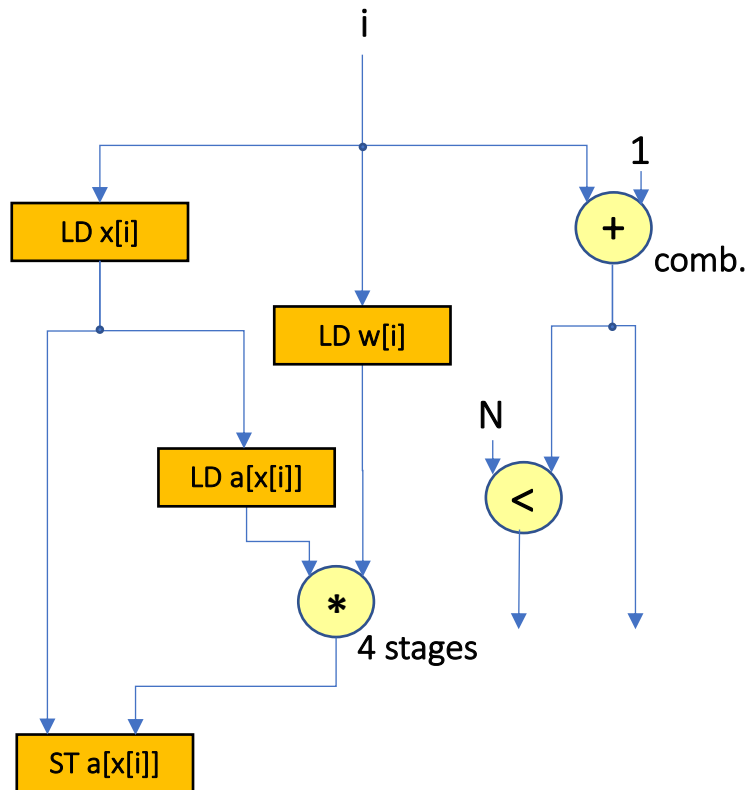


1. **TAPAS: Generating Parallel Accelerators from Parallel Programs**. Steven Margerm, Amirali Sharifian, Apala Guha, Gilles Pokam and Arrvindh Shriraman., **MICRO**, 2018.
2. **μIR -An intermediate representation for transforming and optimizing the microarchitecture of application accelerators**. Amirali Sharifian, Reza Hojabr, Navid Rahimis, Sihao Liu, Apala Guha, Tony Nowatzki and Arrvindh Shriraman. **MICRO**, 2019.
3. **Dynamically Scheduled High-level Synthesis** Josipović, Ghosal, and lenne.. FPGA 2018



# Dataflow Circuits

- Constructing the datapath

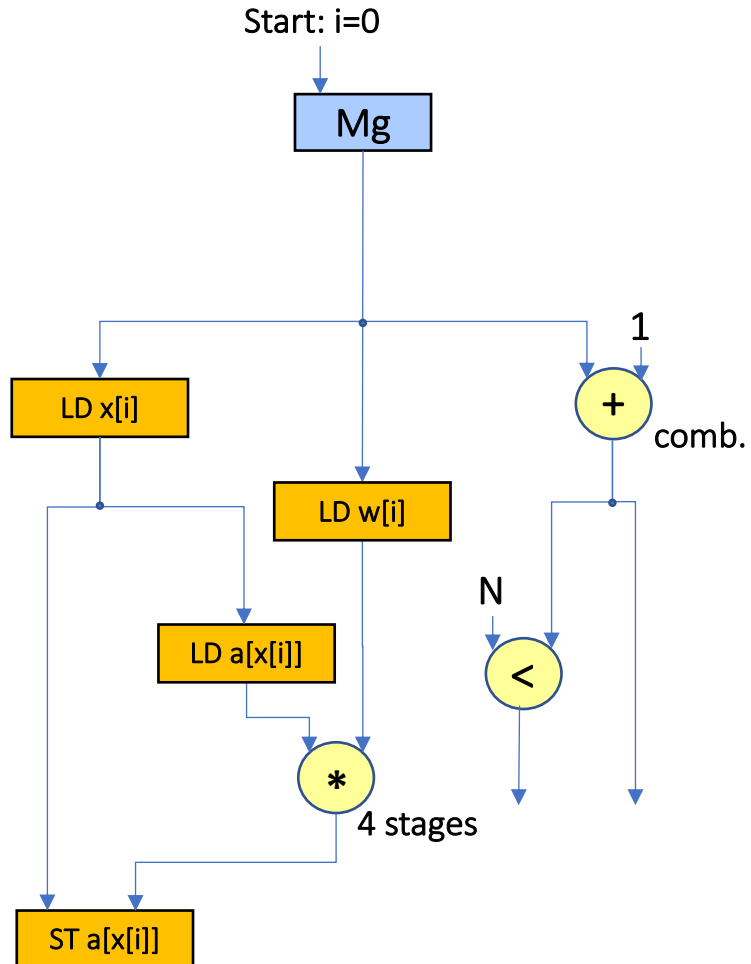


```
for (i = 0; i < 100; i++) {  
    a[x[i]] = a[x[i]] * w[i];  
}
```

**Each operator corresponds to  
a functional unit**



# Dataflow Circuits

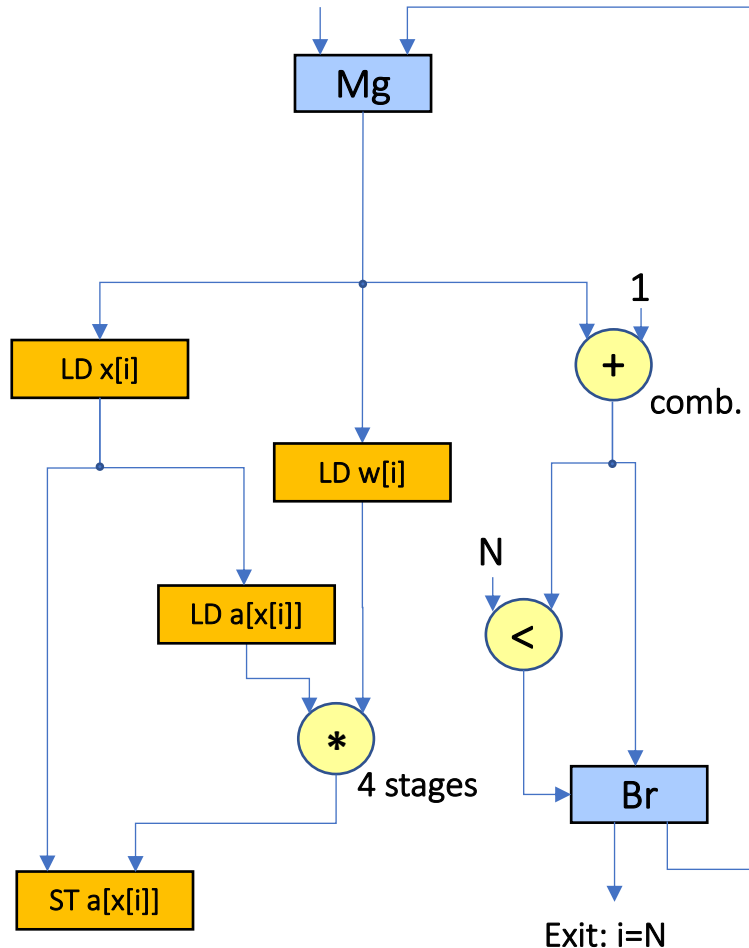


```
for (i = 0; i < 100; i++) {  
    a[x[i]] = a[x[i]] * w[i];  
}
```

**A Merge for each variable  
entering the BB**



# Synthesizing Dataflow Circuits

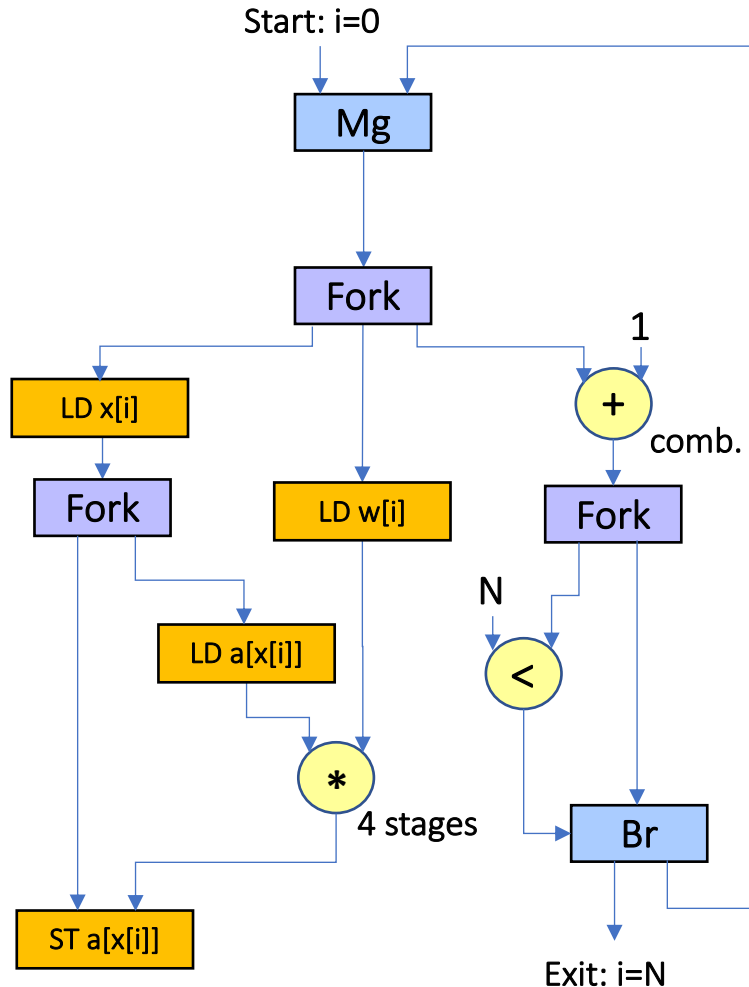


```
for (i = 0; i < 100; i++) {  
    a[x[i]] = a[x[i]] * w[i];  
}
```

**A Branch for each variable  
exiting the BB**



# Synthesizing Dataflow Circuits



```

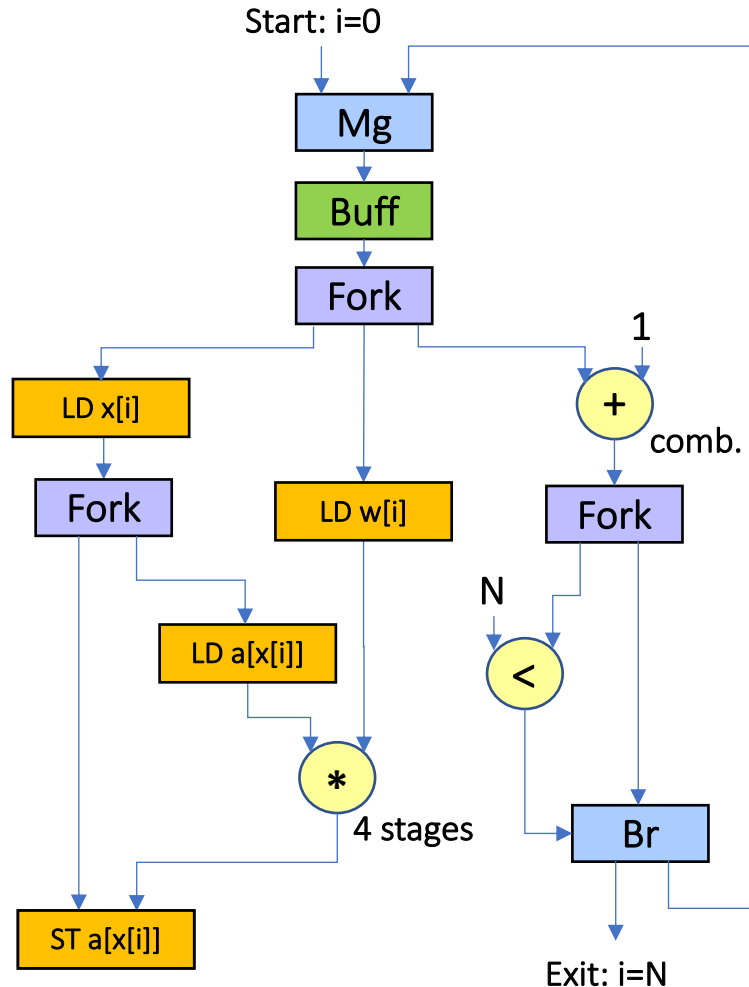
for (i = 0; i < 100; i++) {
    a[x[i]] = a[x[i]] * w[i];
}

```

**A Fork after every node with multiple successors**



# Synthesizing Dataflow Circuits



```

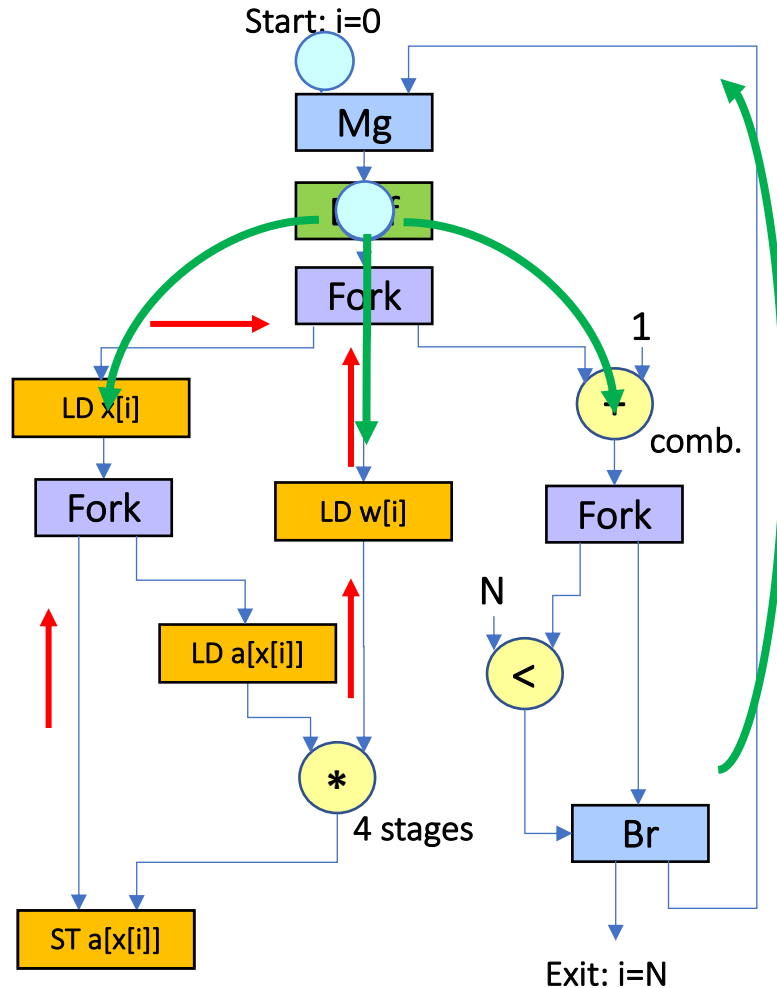
for (i = 0; i < 100; i++) {
    a[x[i]] = a[x[i]] * w[i];
}

```

**Use buffers to break  
combinational loops**



# Synthesizing Dataflow Circuits

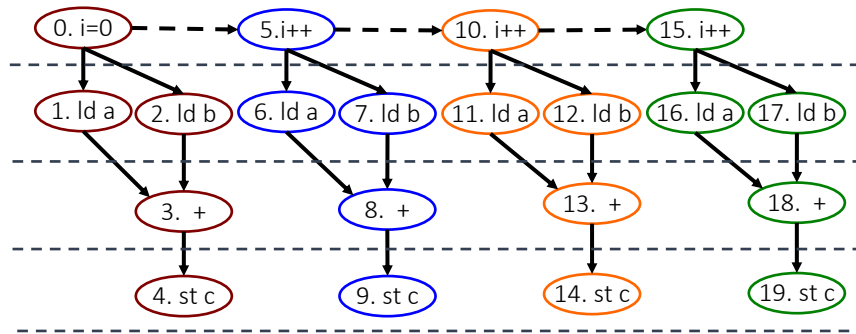


```
for (i = 0; i < 100; i++) {
    a[x[i]] = a[x[i]] * w[i];
}
```



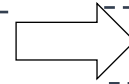
# Scheduling 101

Idealistic DDDG

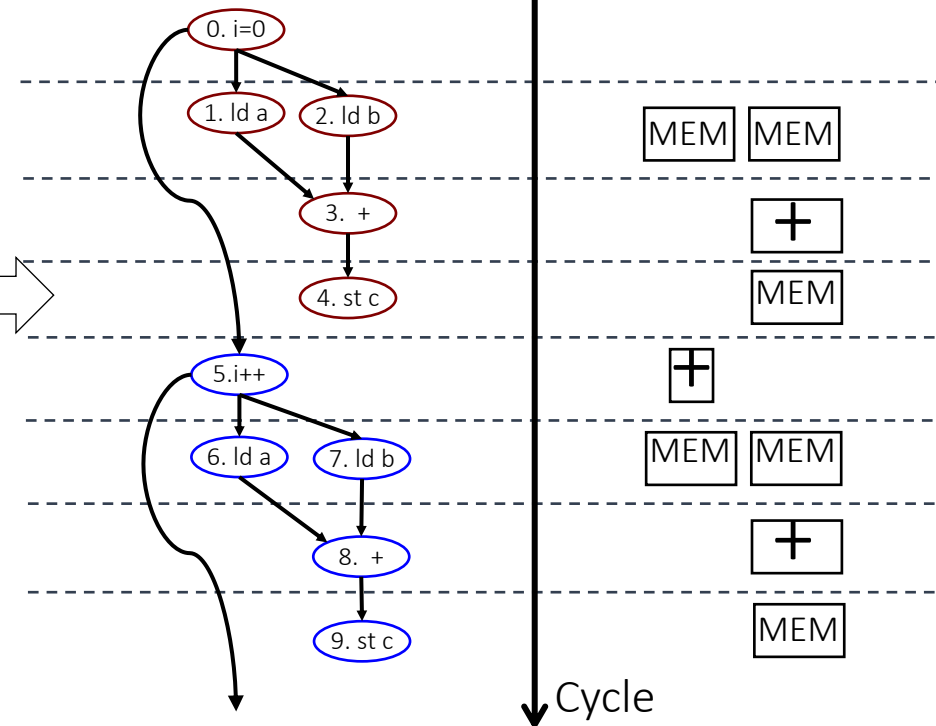


Acc Design Parameters:

- ✓ Memory BW ≤ **2**
- ✓ **1** Adder



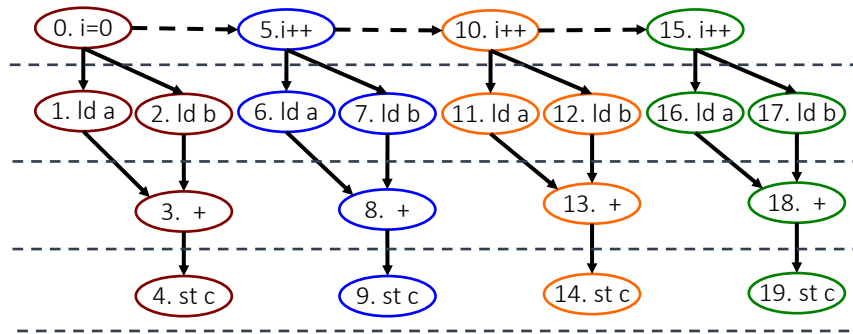
Resource Activity





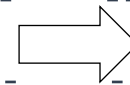
# Scheduling 101

Spatio-Temporal Dataflow

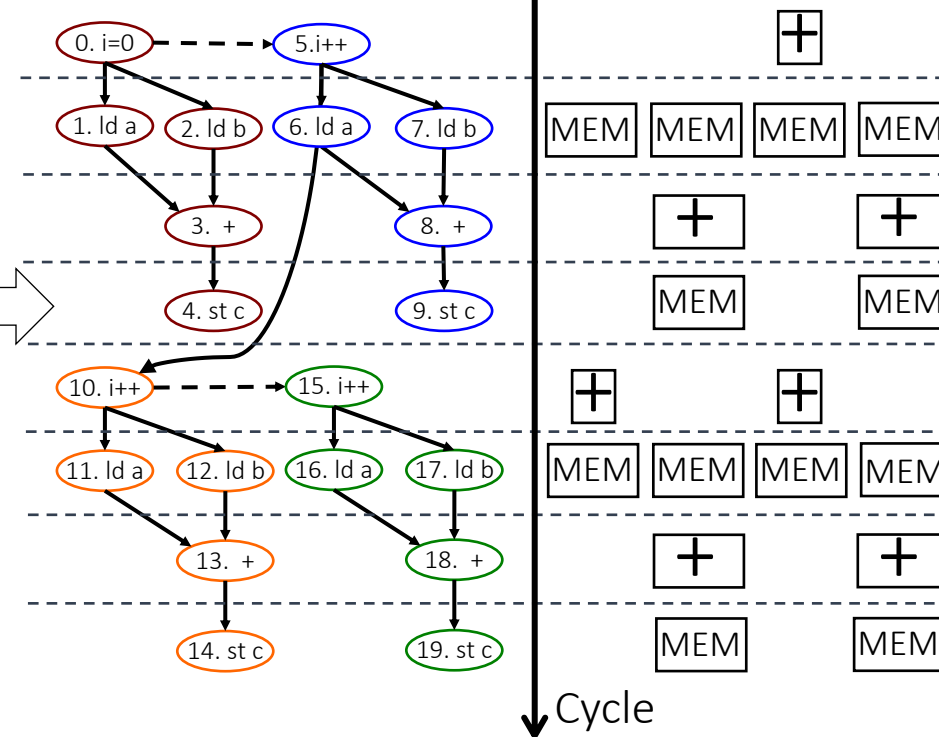


Acc Design Parameters:

- ✓ Memory BW  $\leq 4$
- ✓ **2** Adders



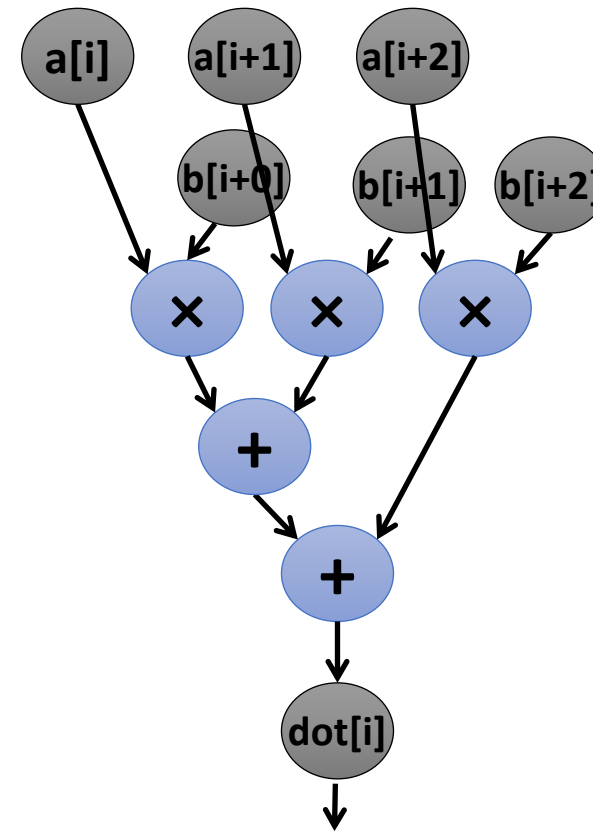
Resource Activity





# CGRA Vector Interface

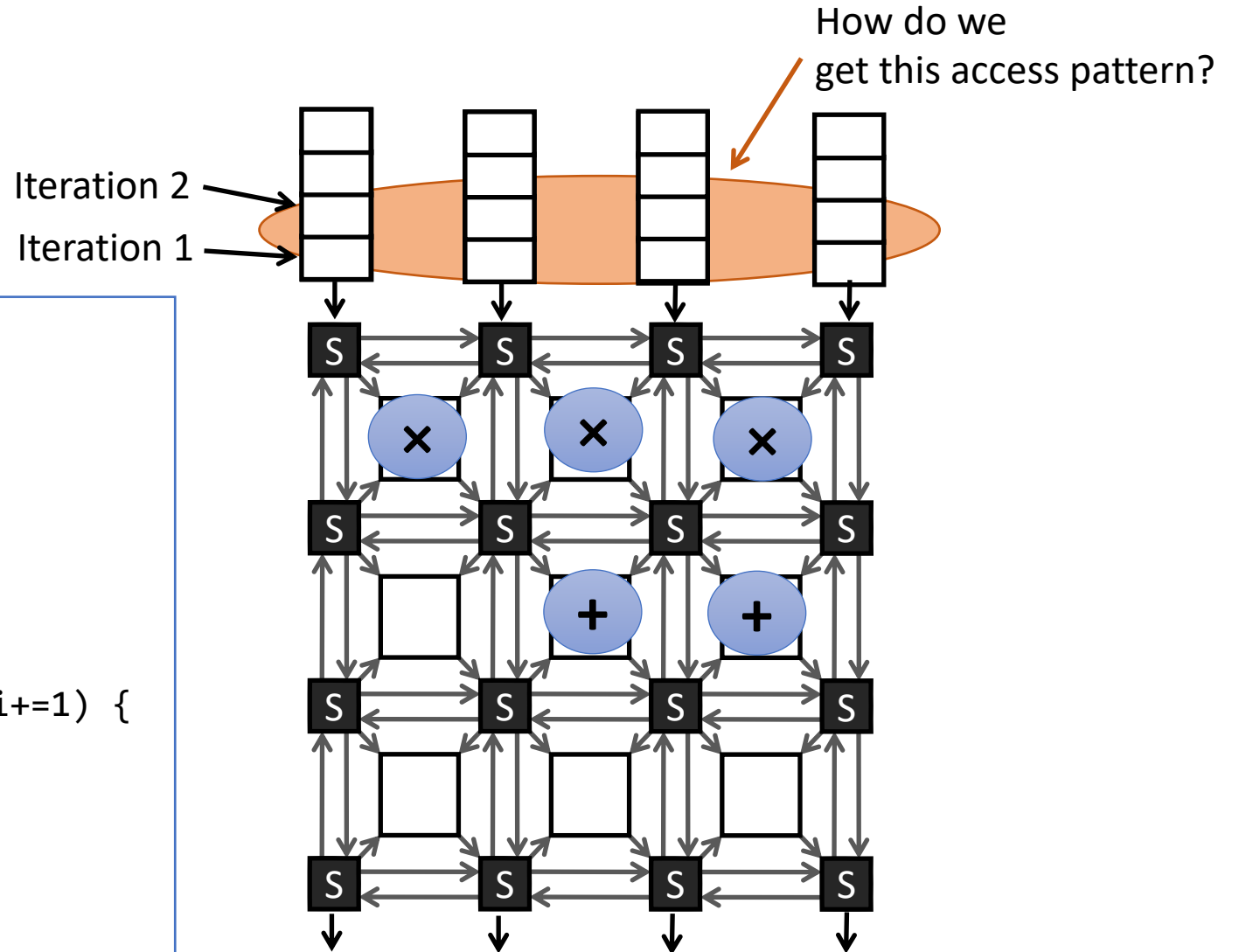
```
struct vec {  
    float x, y, z;  
    float q;  
}  
vec A[], B[];  
float *a = A, *b = B;  
float dot[];  
for(int i =0; i < LEN; i+=1) {  
    dot[i]=A[i].x*B[i].x  
        +A[i].y*B[i].y  
        +A[i].z*B[i].z;  
}
```





# CGRA Vector Interface

```
struct vec {  
    float x, y, z;  
    float q;  
}  
vec A[], B[];  
float *a = A, *b = B;  
float dot[];  
for(int i =0; i < LEN; i+=1) {  
    dot[i]=A[i].x*B[i].x  
        +A[i].y*B[i].y  
        +A[i].z*B[i].z;  
}
```



Ports shown only for a[]



**Domain Specific Architecture =  
Compiler-Driven Spatial Hardware**



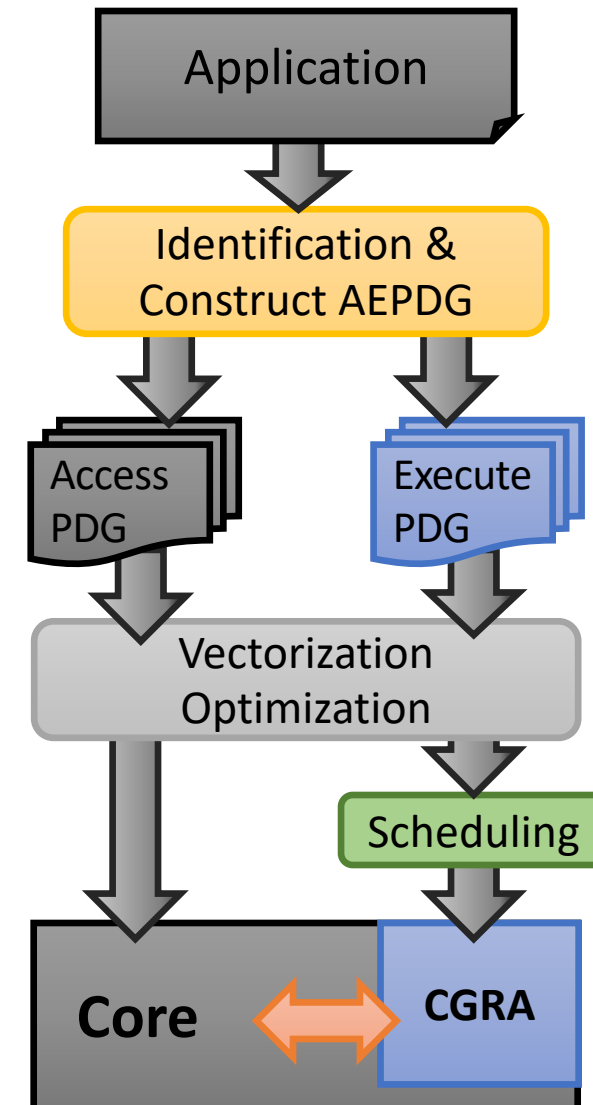
# Compiler Intermediate Representation

- Makes it easier to optimize for target architecture
- A suitable IR should
  - Model the architecture, accurately if possible
  - Capture the dependencies between the operations
  - Generate code for the architecture with ease



# Compilation Tasks

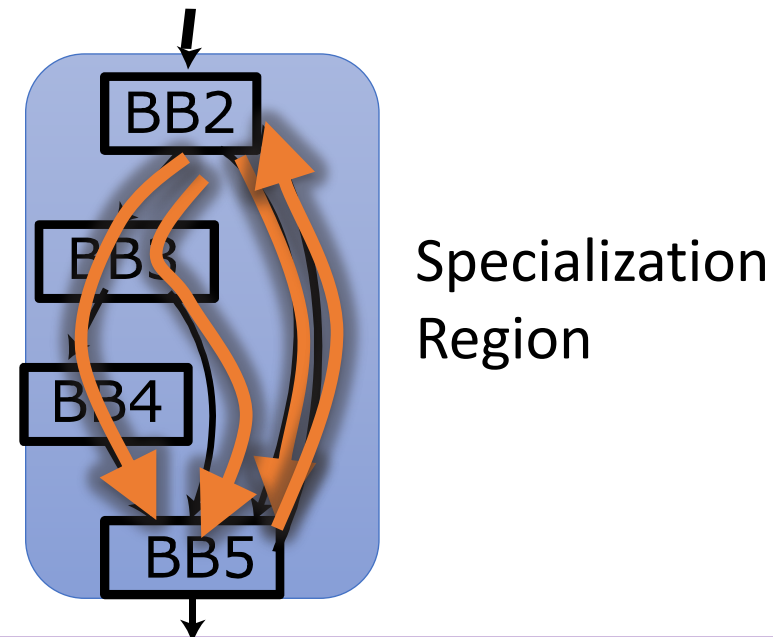
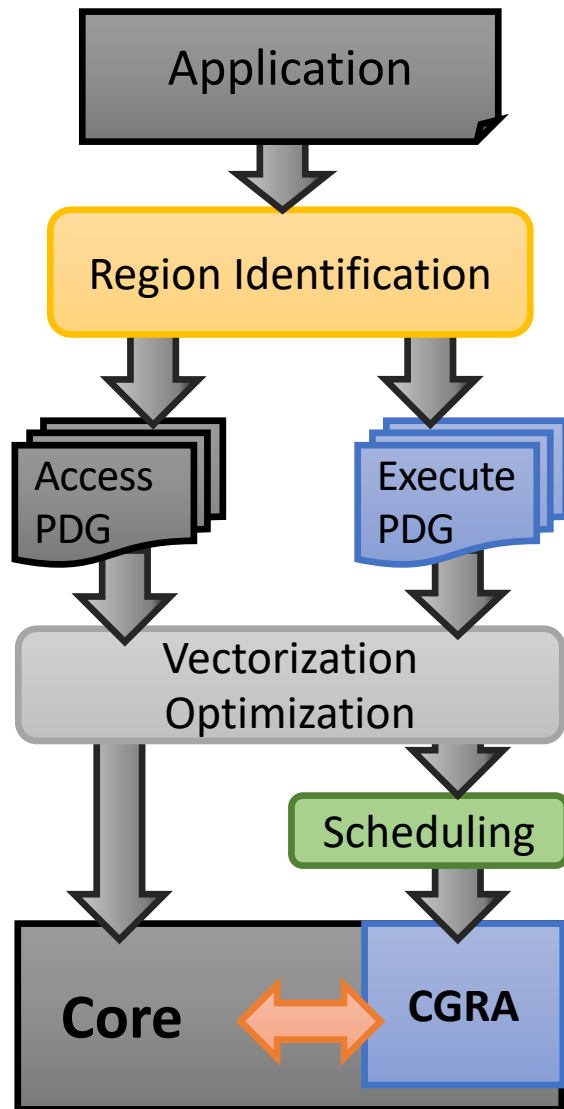
- Identify code-regions/loops to specialize
- Construct AEPDG
  - Access PDG
  - Execute PDG
- Perform Vectorization/ Optimizations
- Schedule
  - Execute PDG to CGRA
  - Access PDG to core



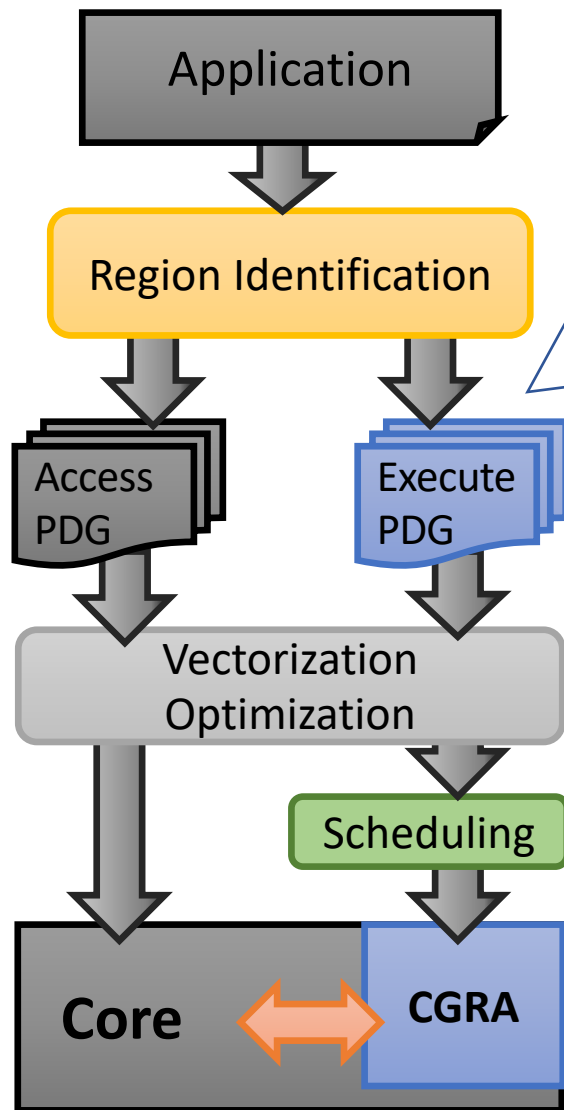


# Region Identification

- Identify code-regions to specialize
  - Path Profiling
  - Utilize Loops
- Need Single-Entry / Single Exit Region







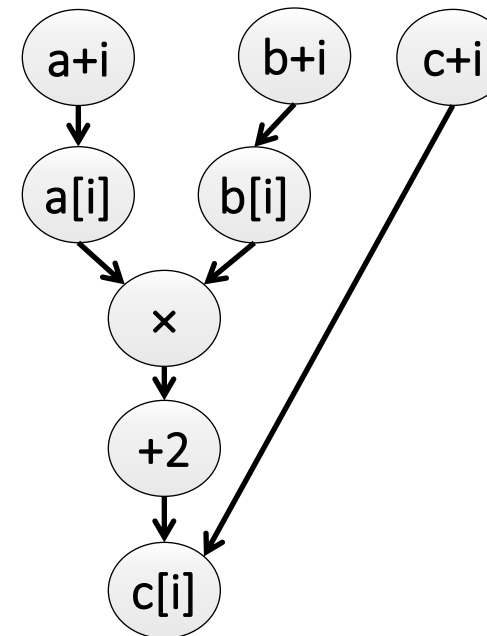
# Construct AEPDG

- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

Address Calc:

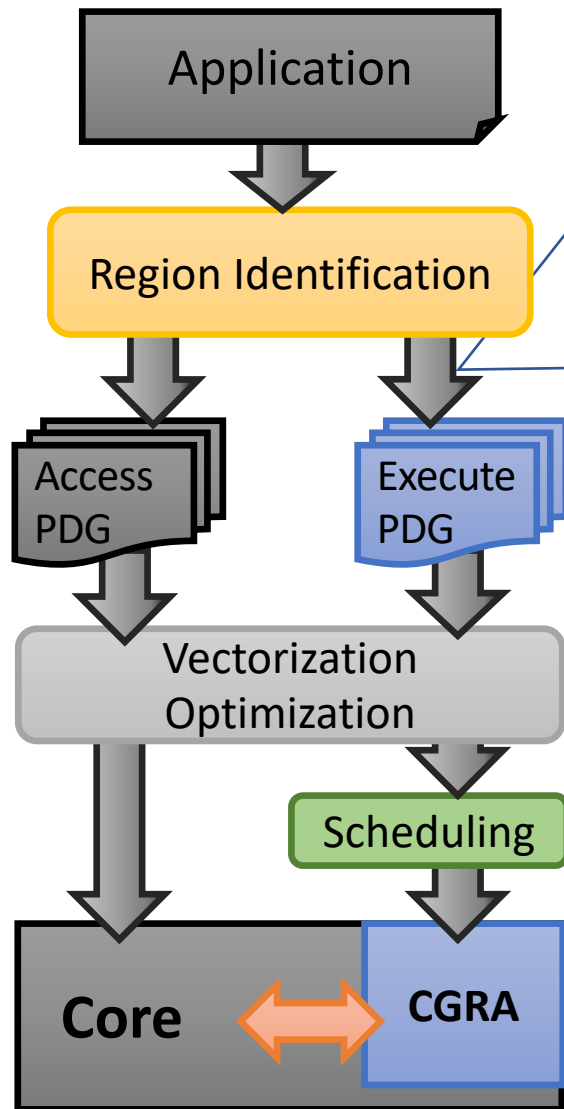
Loads:

Store:





# Construct AEPDG

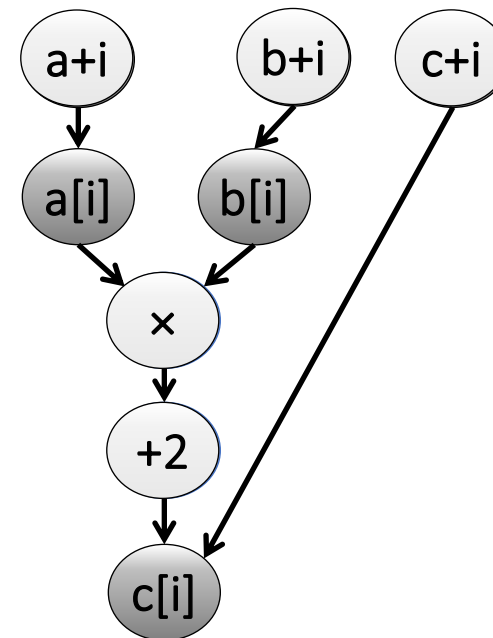


- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

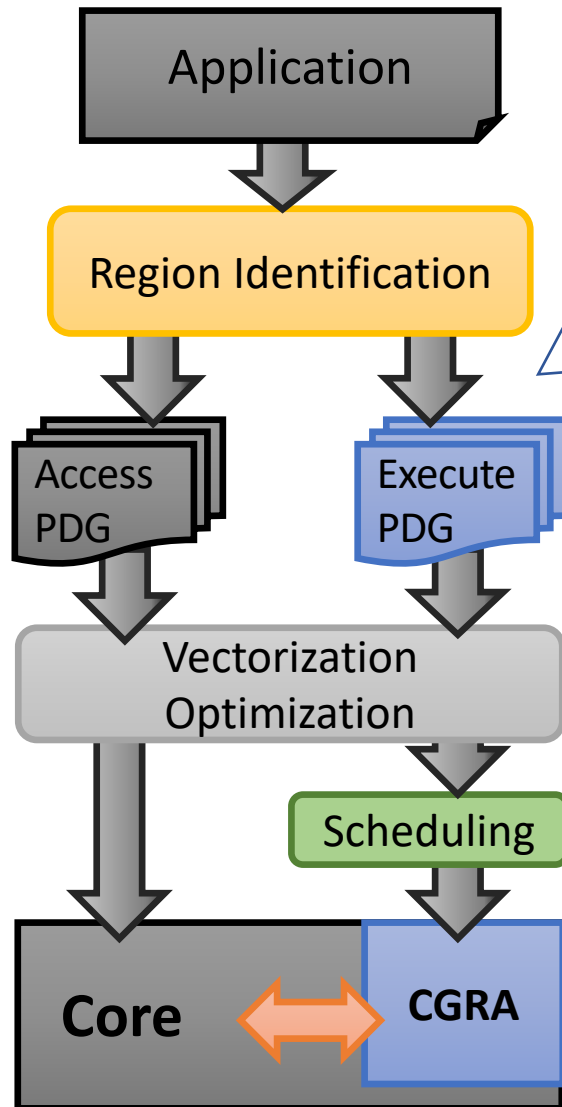
Address Calc:

Loads:

Store:







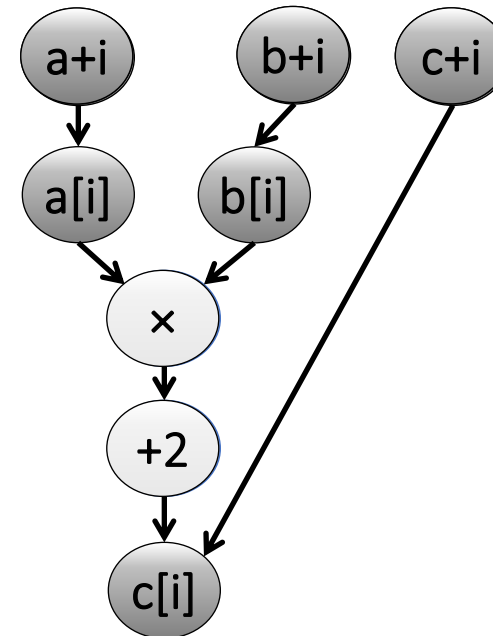
# Construct AEPDG

- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

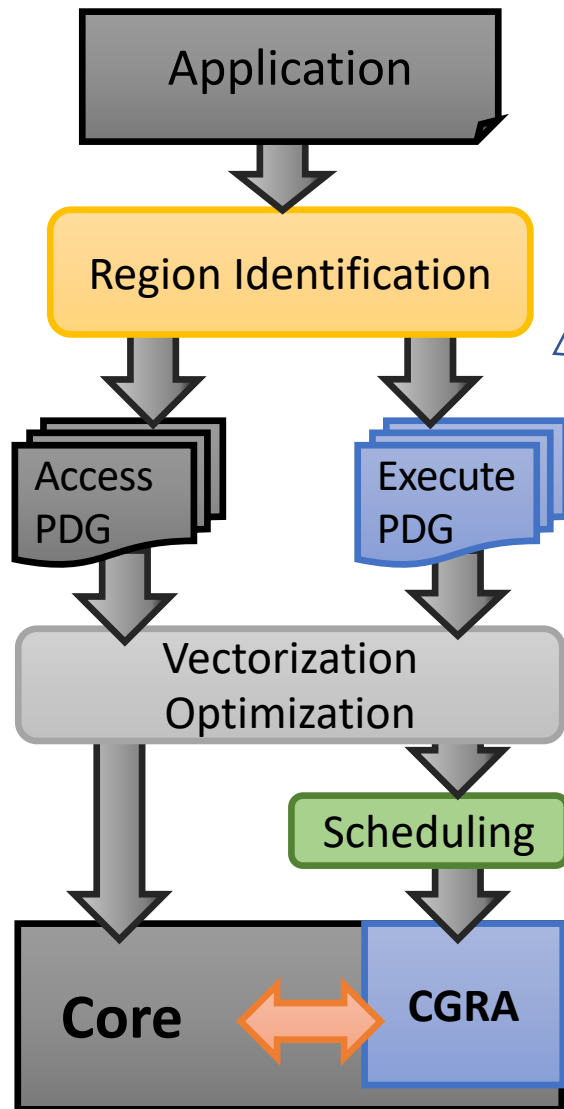
Address Calc:

Loads:

Store:







# Construct AEPDG

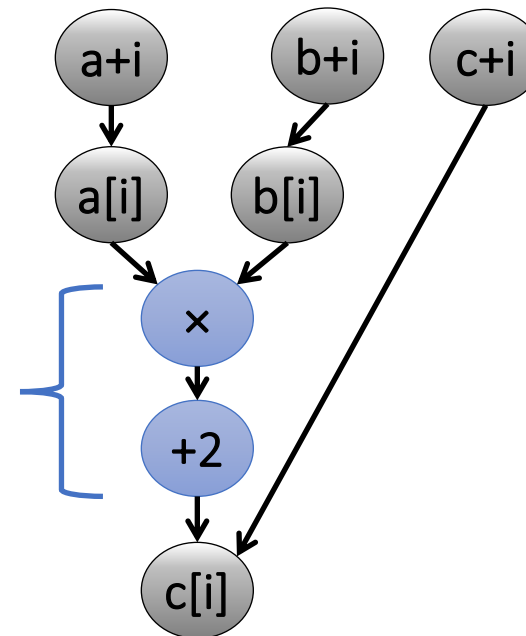
- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

Address Calc:

Loads:

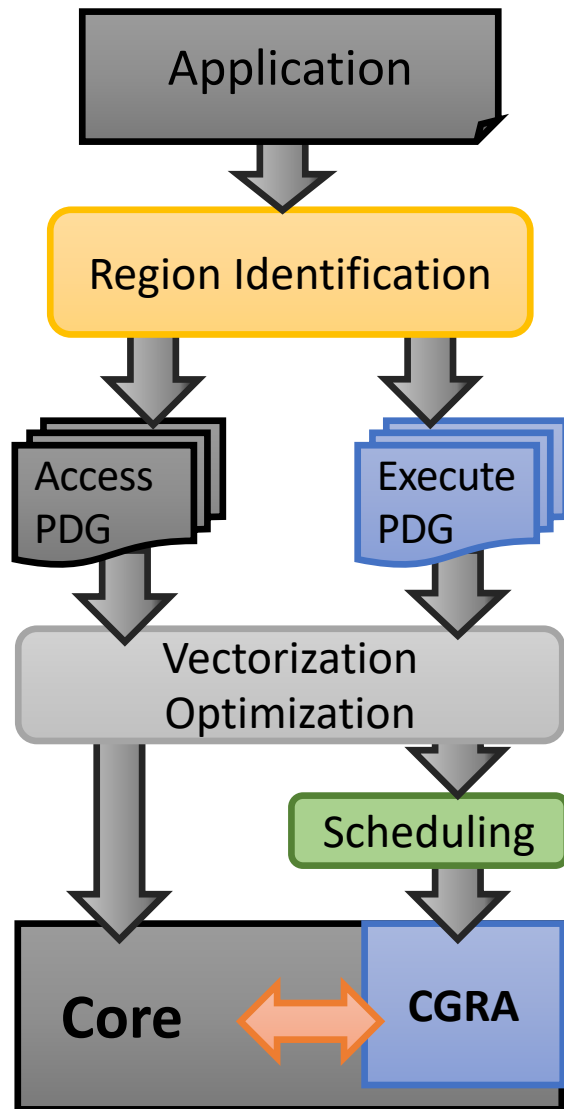
Execute Subregion

Store:

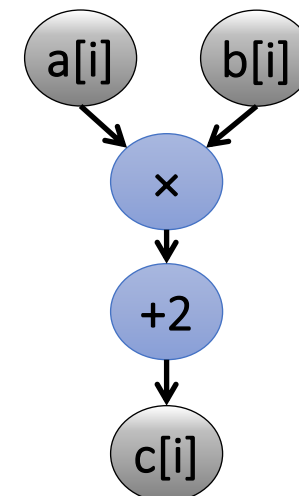




# Vectorization

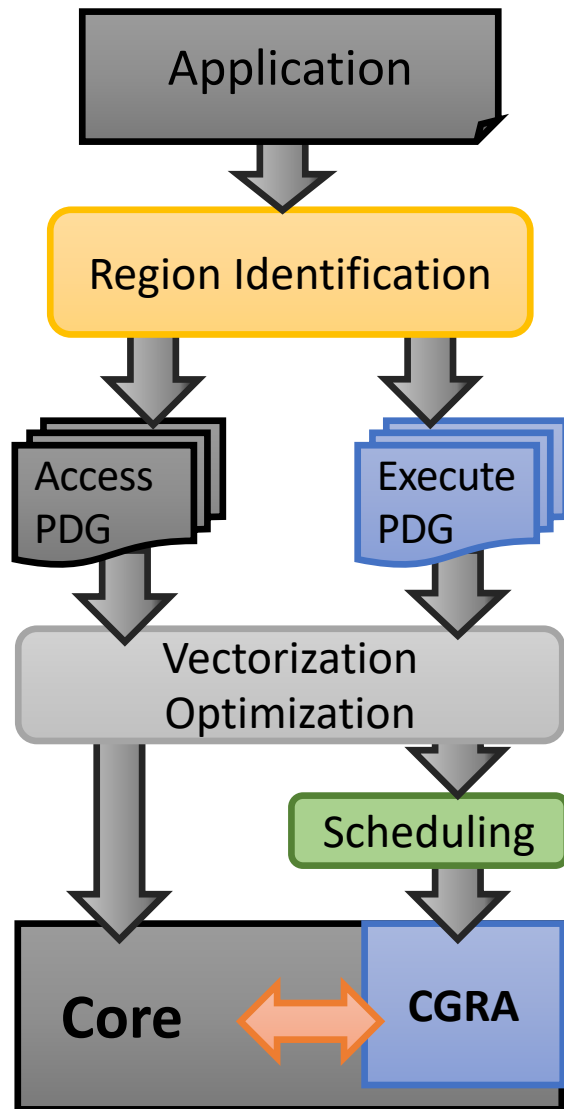


- Similar to SIMD Techniques, loops must have:
  - Independent Iterations
  - Must be no Store/Load Aliasing
- Memory Access: No gather/scatter
- Perform Loop Control
  - Modify trip count/peel scalar loop

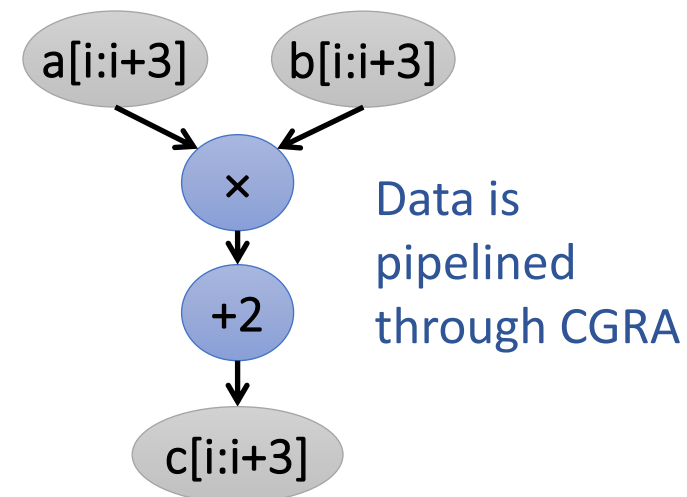




# Vectorization

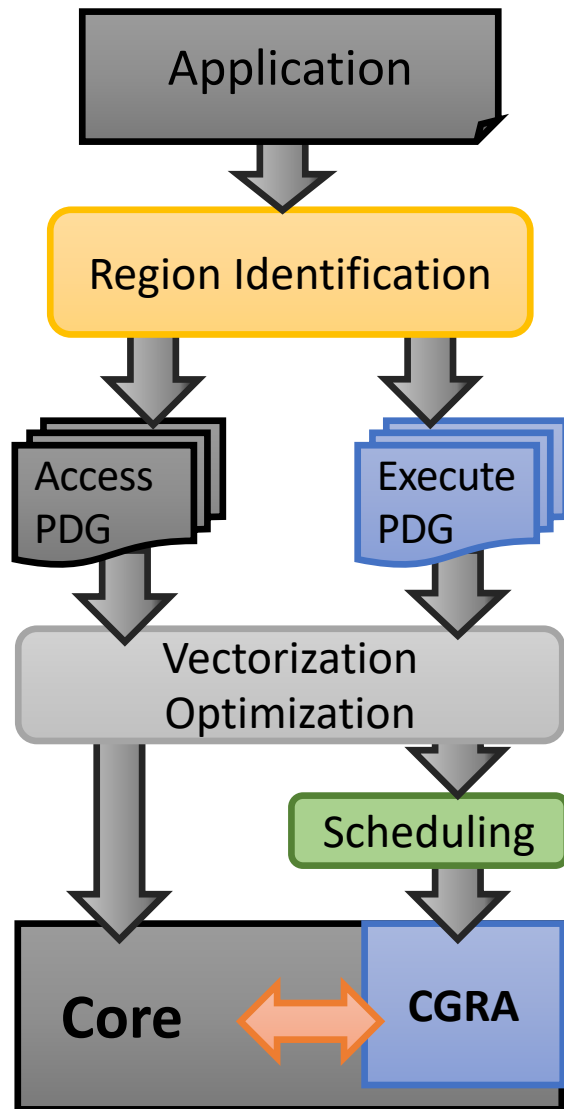


- Similar to SIMD Techniques, loops must have:
  - Independent Iterations
  - Must be no Store/Load Aliasing
- Memory Access: No gather/scatter
- Perform Loop Control
  - Modify trip count/peel scalar loop

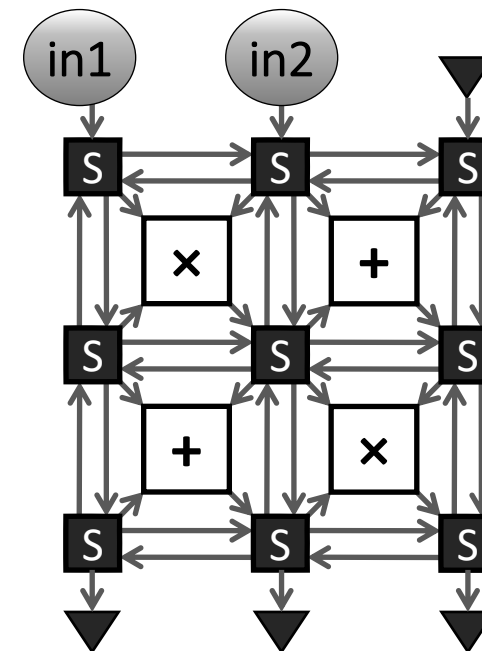
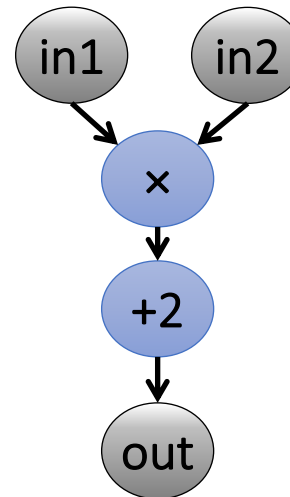




# Scheduling

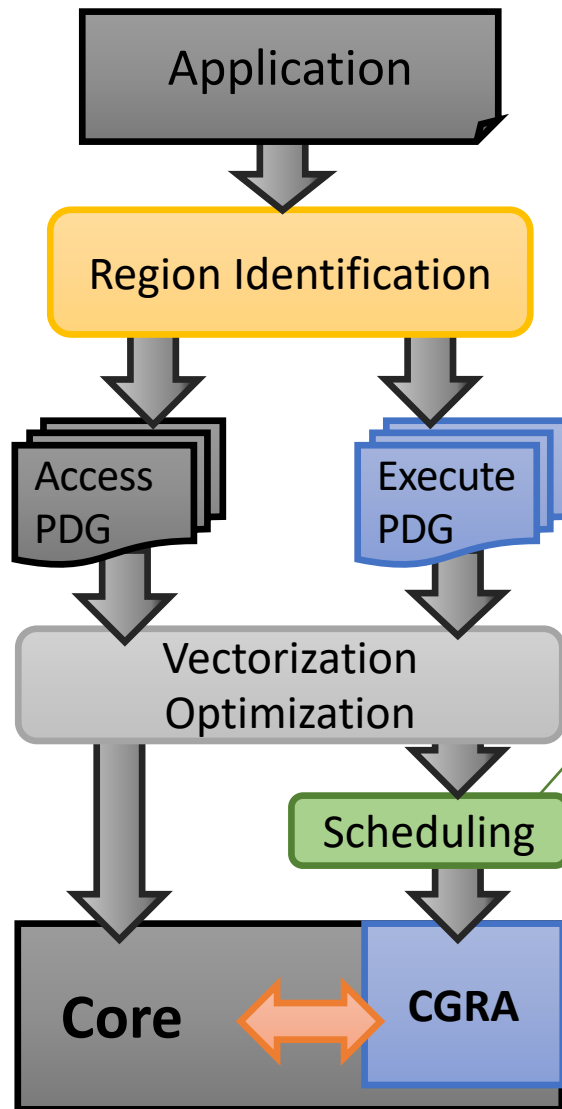


- Map Execute Subregion
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

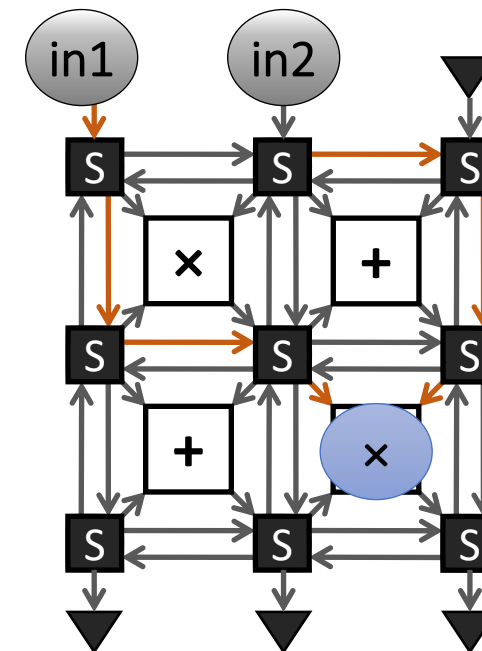
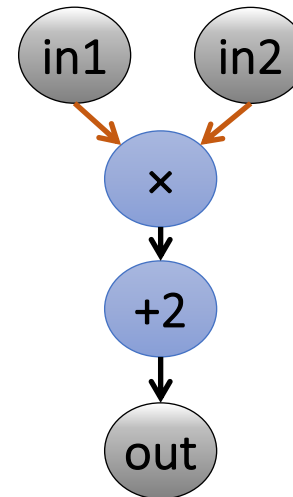




# Scheduling

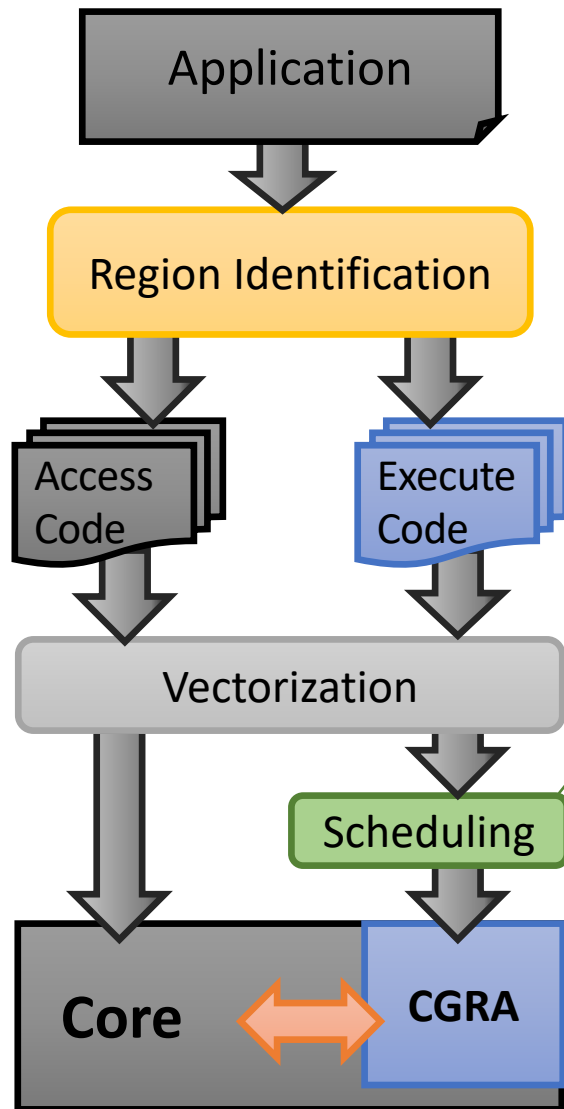


- Map Execute Subregion to CGRA
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

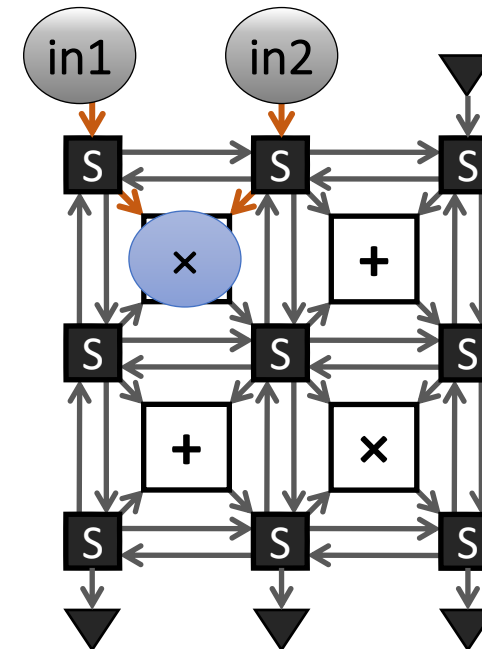
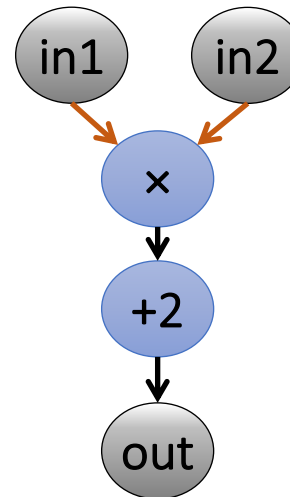




# Scheduling

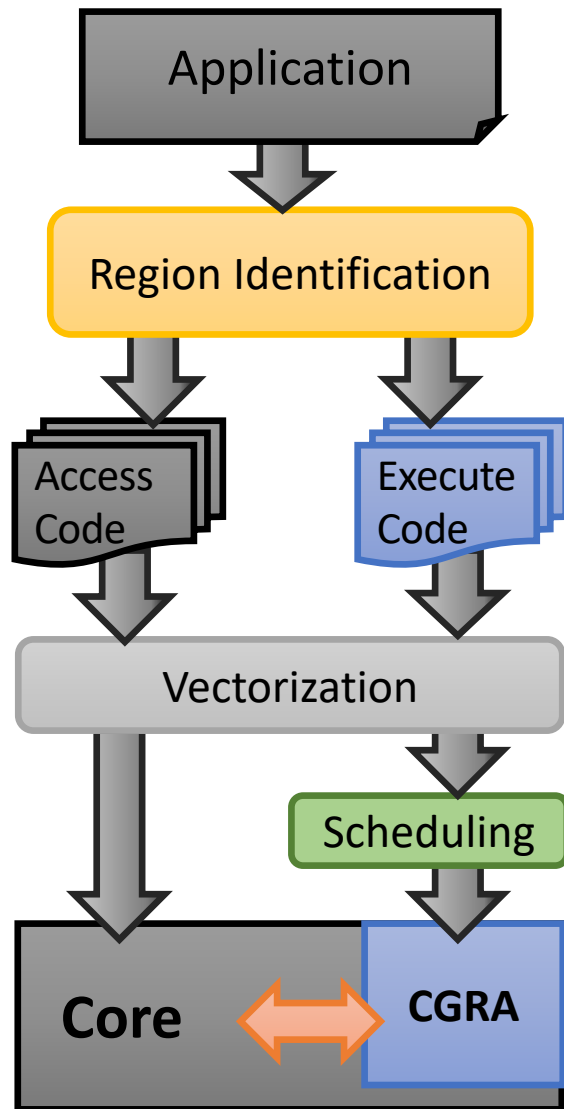


- Map Execute Subregion
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

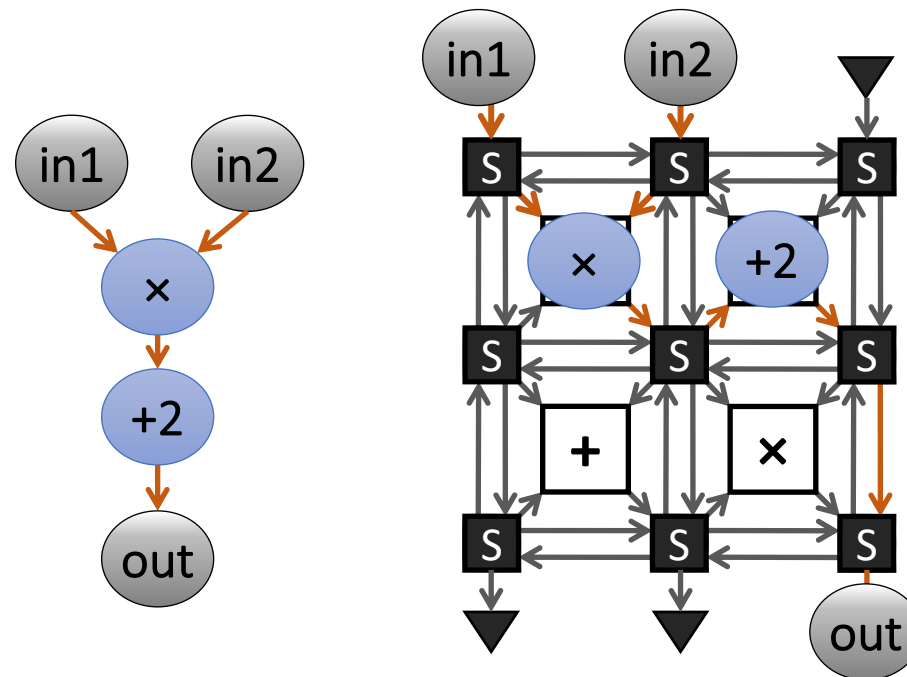




# Scheduling

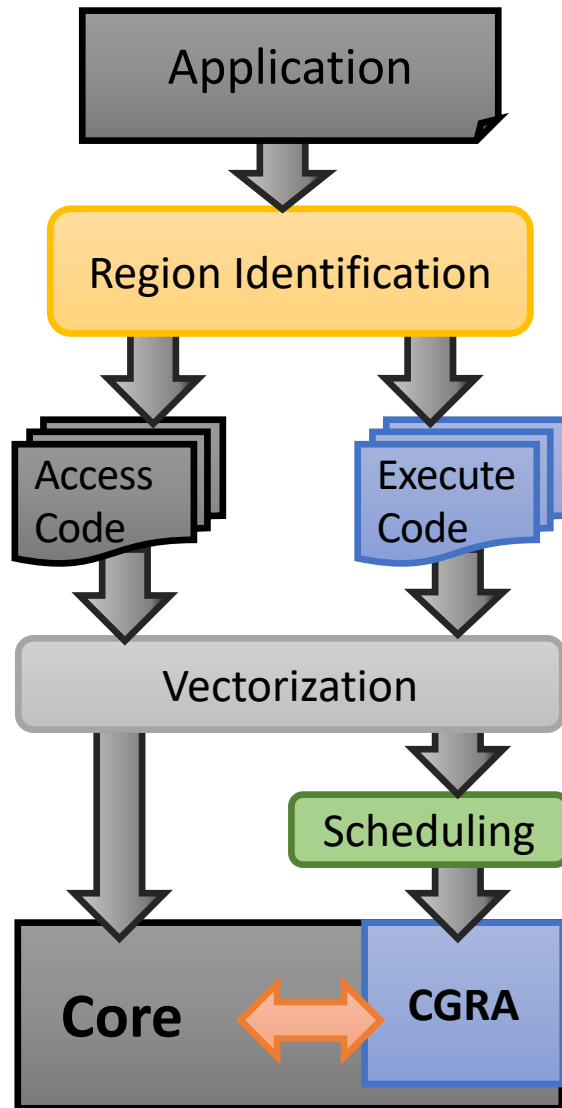


- Map Execute Subregion
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

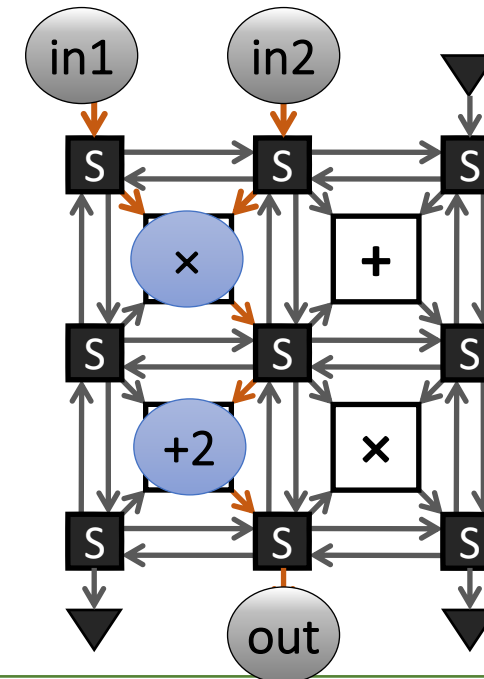
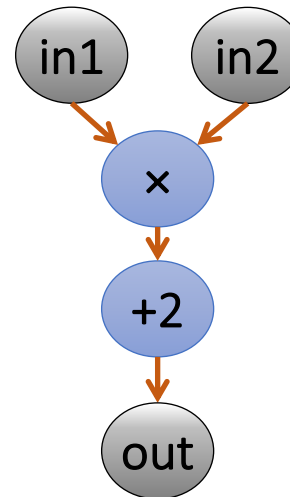




# Scheduling



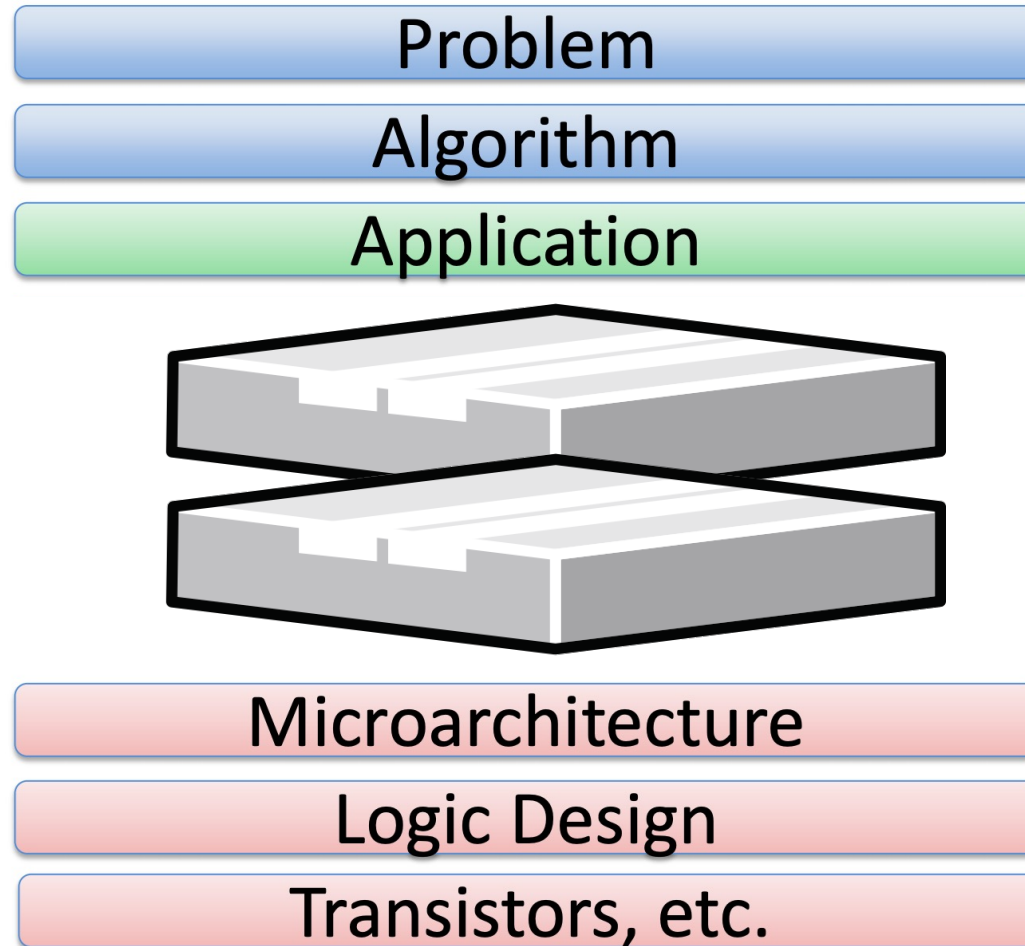
- Map Execute Subregion to CGRA
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes





# The MicroArchitectural IR ( $\mu$ IR) Flow

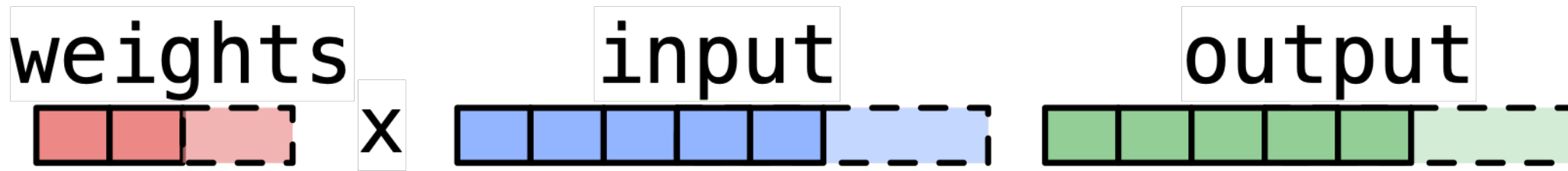
SFU



Extensible  
software  
and hardware  
intermediate representation



# 1D Convolution Hardware

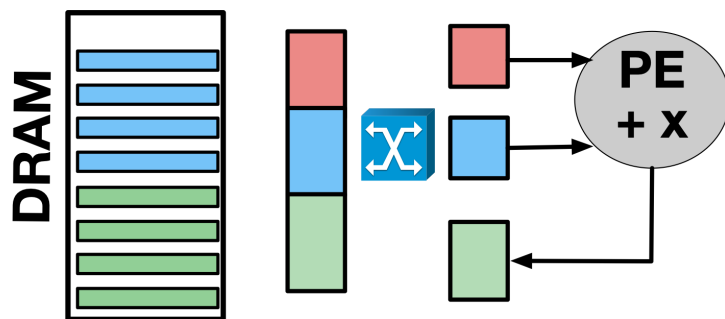


```
cilk_for (i=0; i < (M-W); i++) {  
    cilk_reduce (j = 0; j < W; j++) {  
        output[i] += input[i+j]*  
                    weight[j];  
    }  
}
```

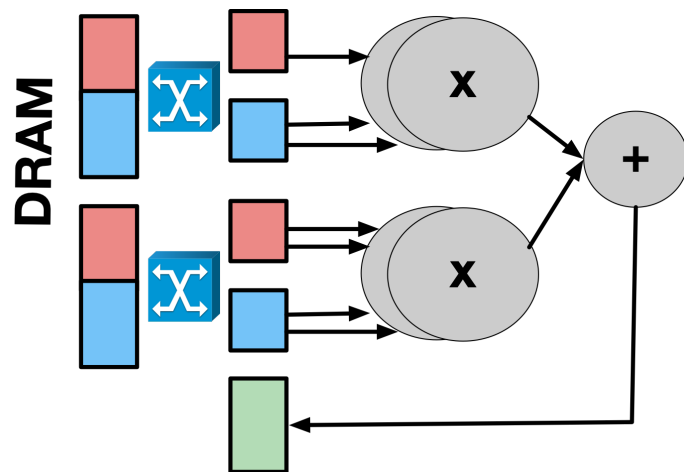


# 1D Convolution Hardware

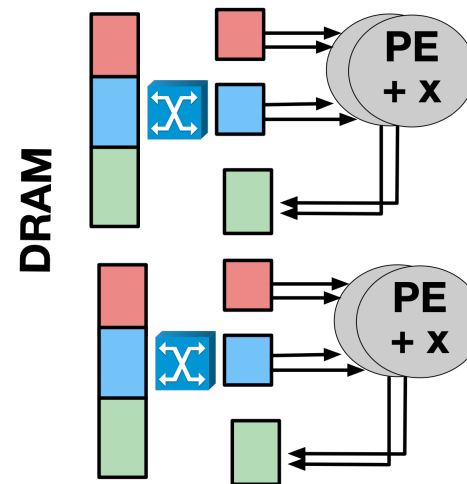
## 1: Register promotion



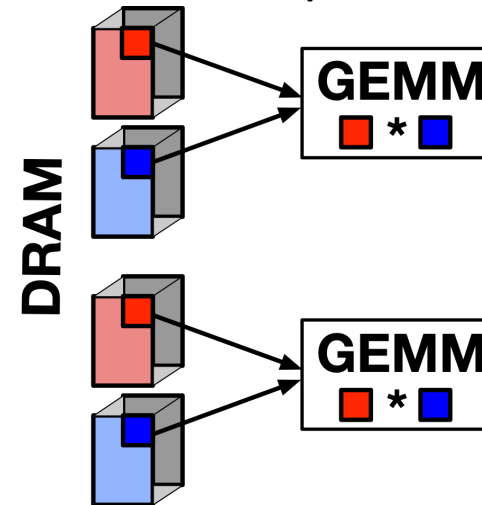
## 2: Pipeline



## 3: Parallelism

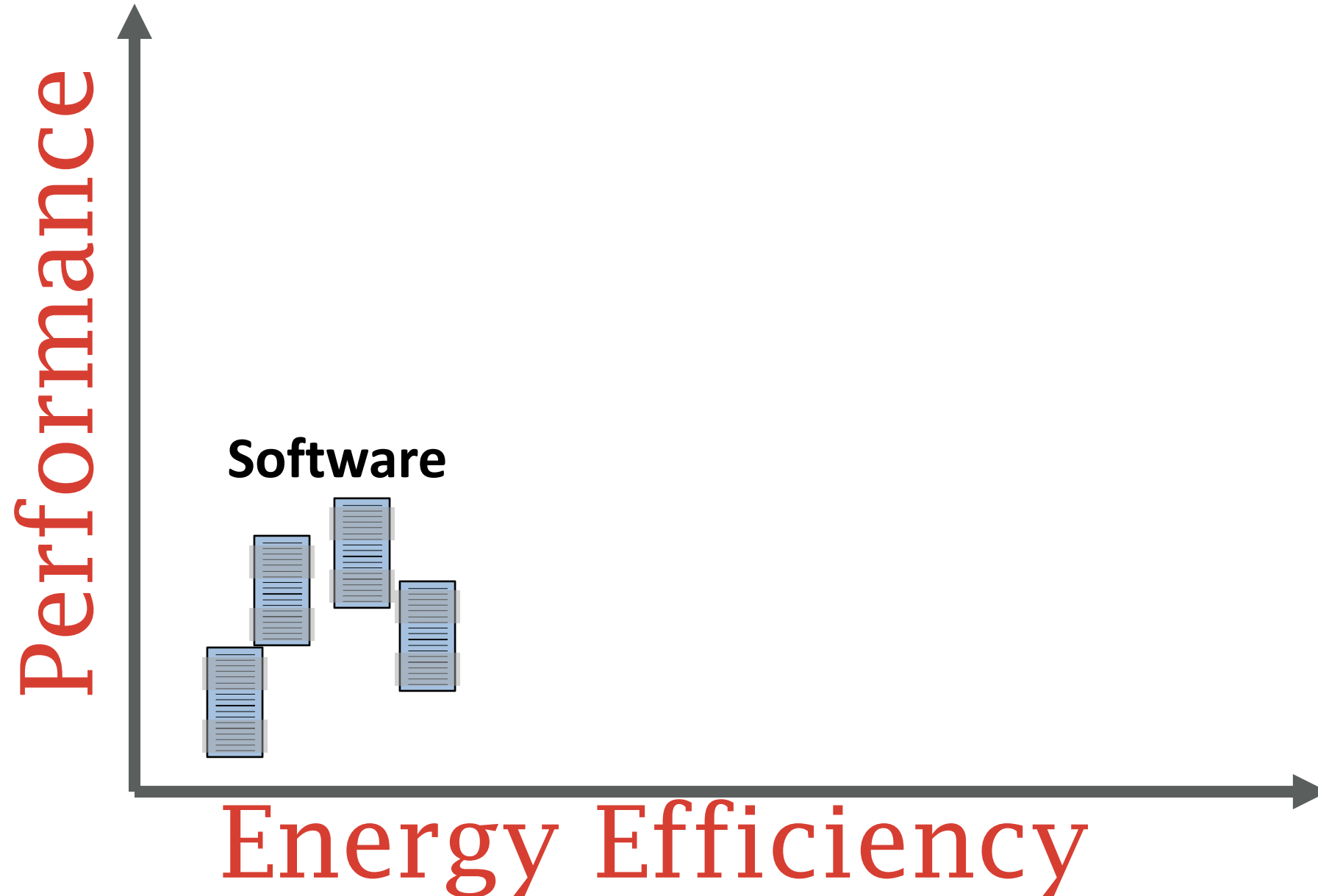


## 4: Domain-specific



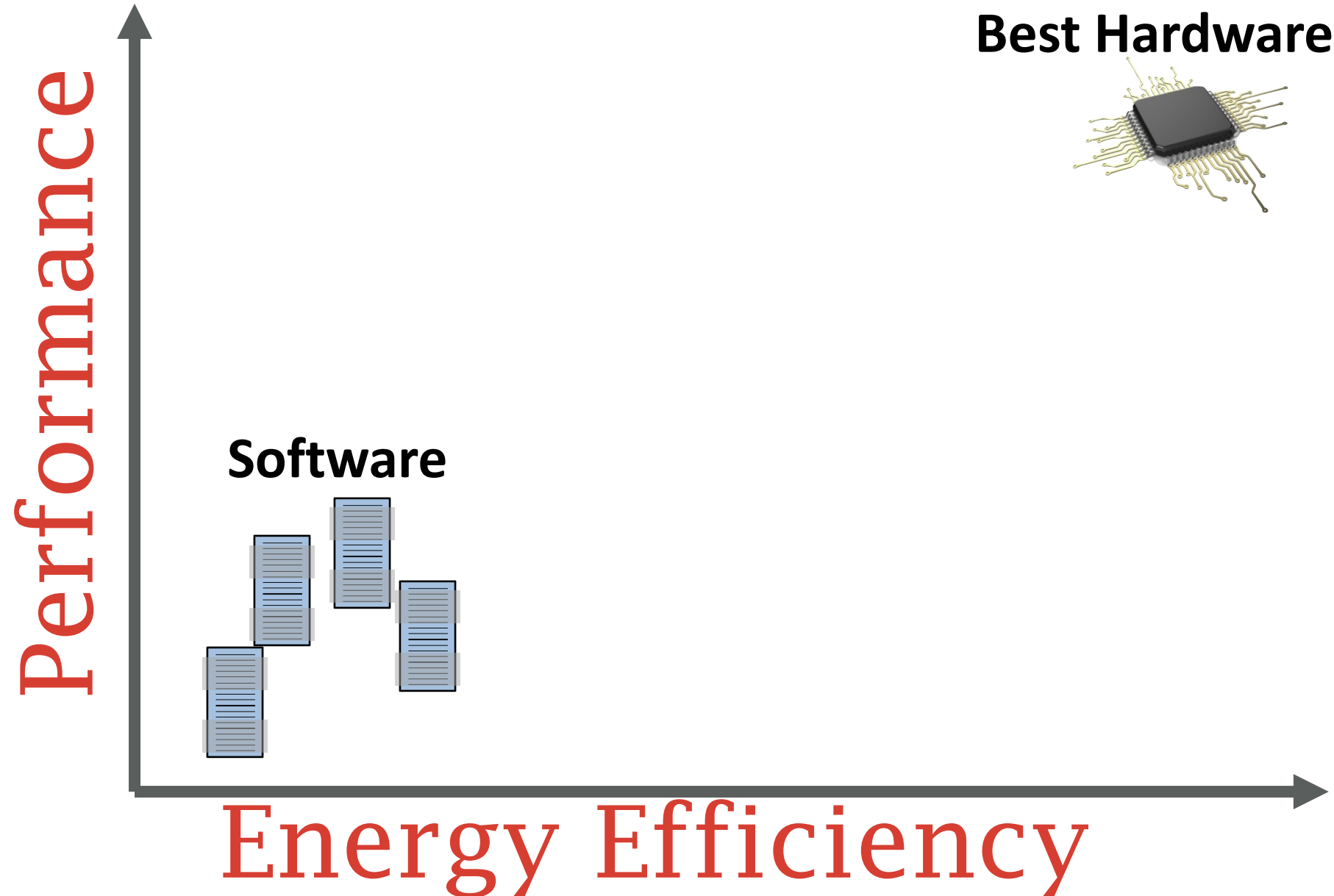


# We have a vision



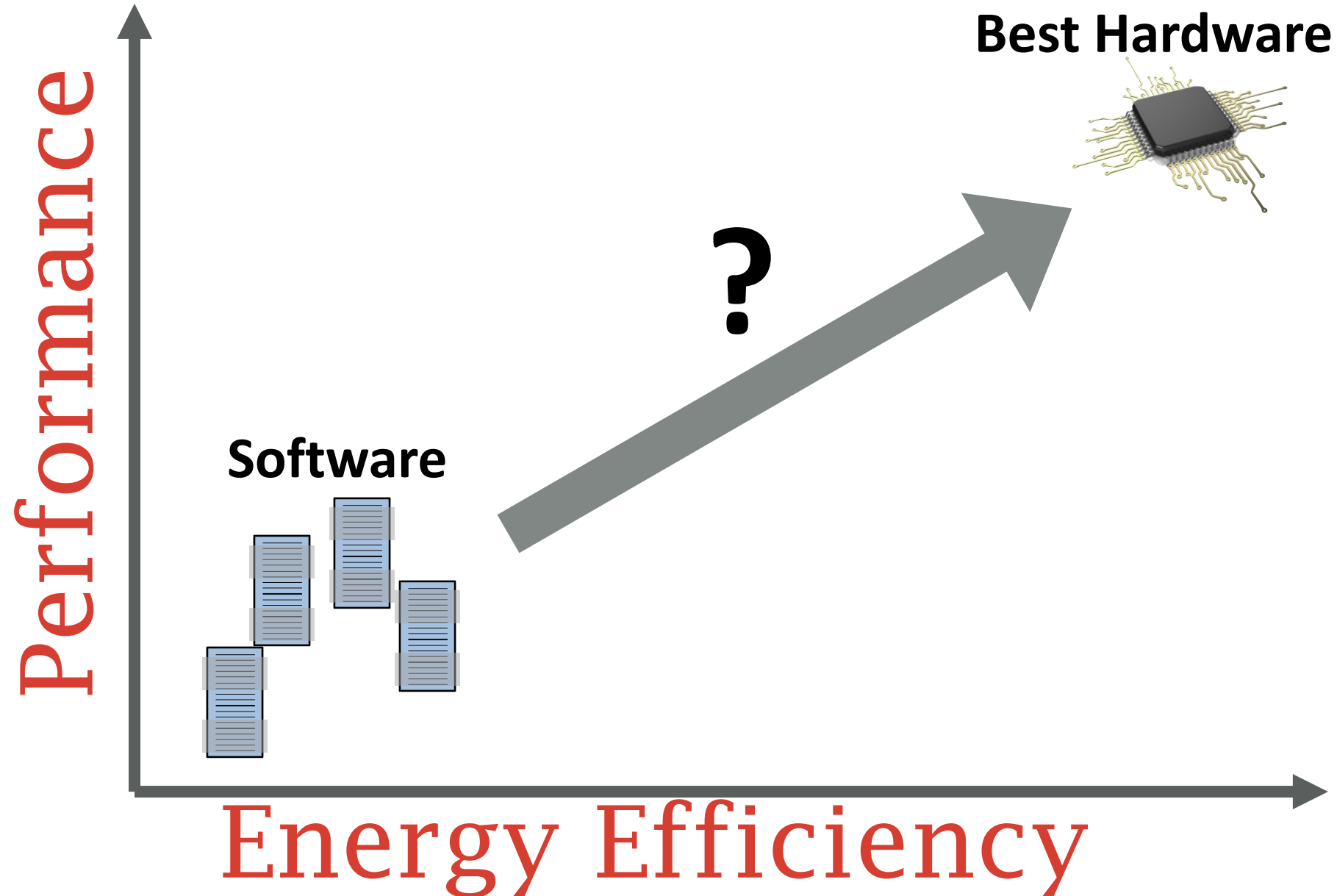


# We have a vision



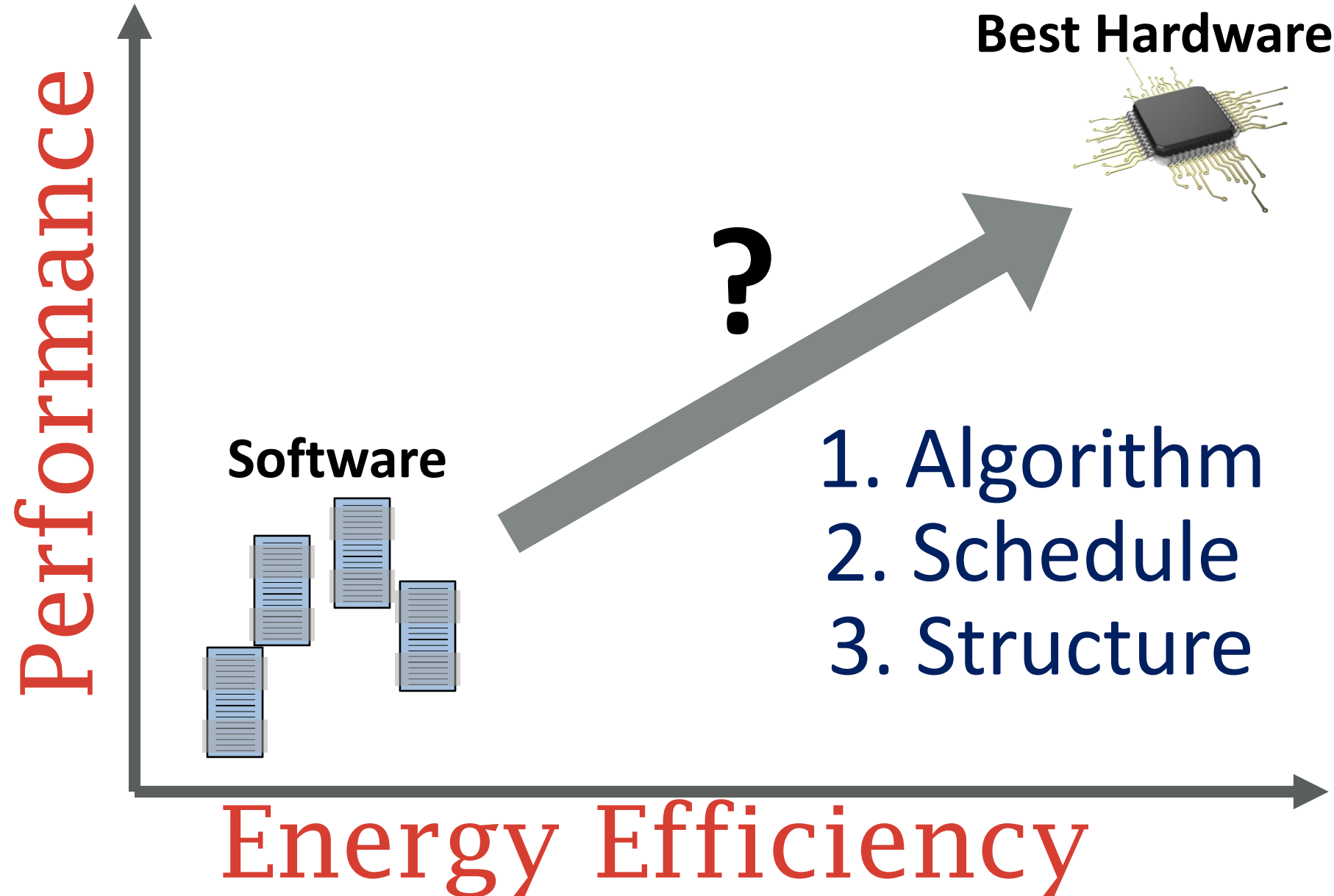


# We have a vision





# We have a vision





# Iron Law of Hardware

