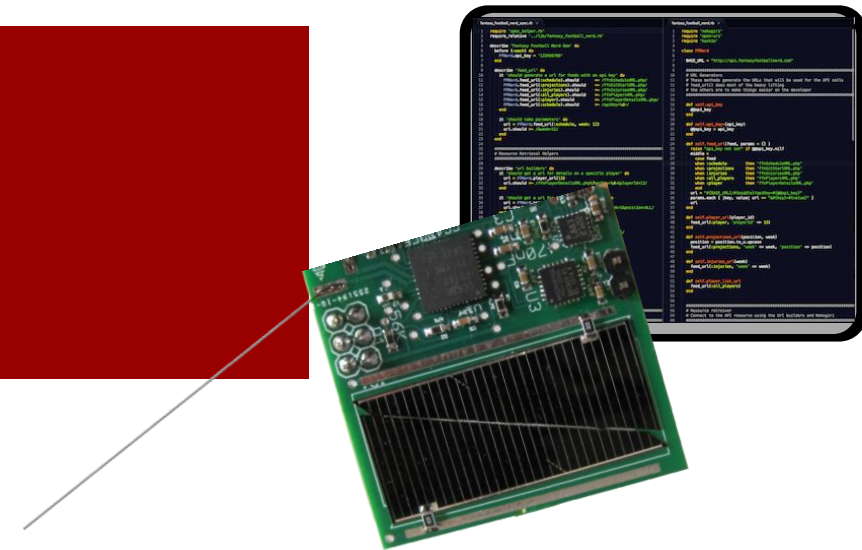




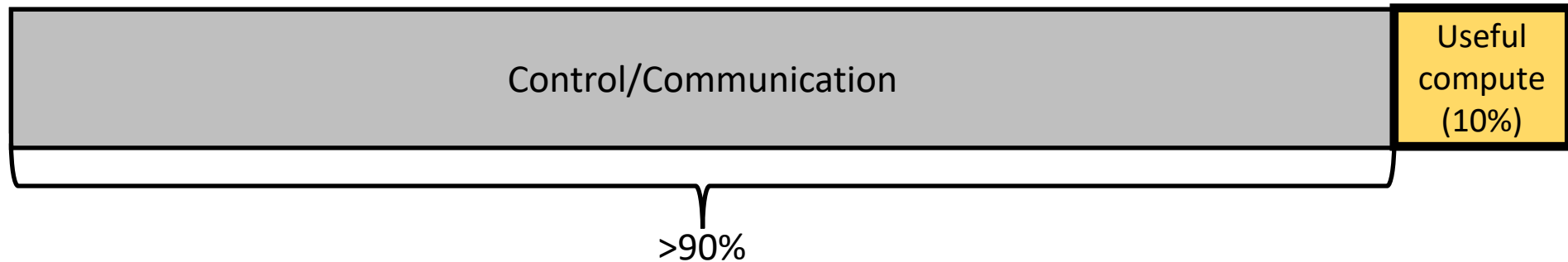
# Energy-minimal Computing

## Edge architectures for extreme efficiency

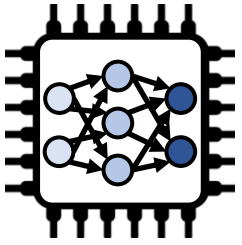


# Existing architectures are extremely inefficient

Instruction energy\* breakdown:



**Extreme Edge Computing Goal:**  
increase energy-efficiency and preserve programmability



# Where does all the energy go in existing computer architectures?

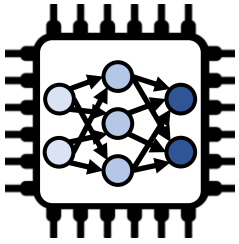
Something is fundamentally wrong here:

Instruction energy\* breakdown:

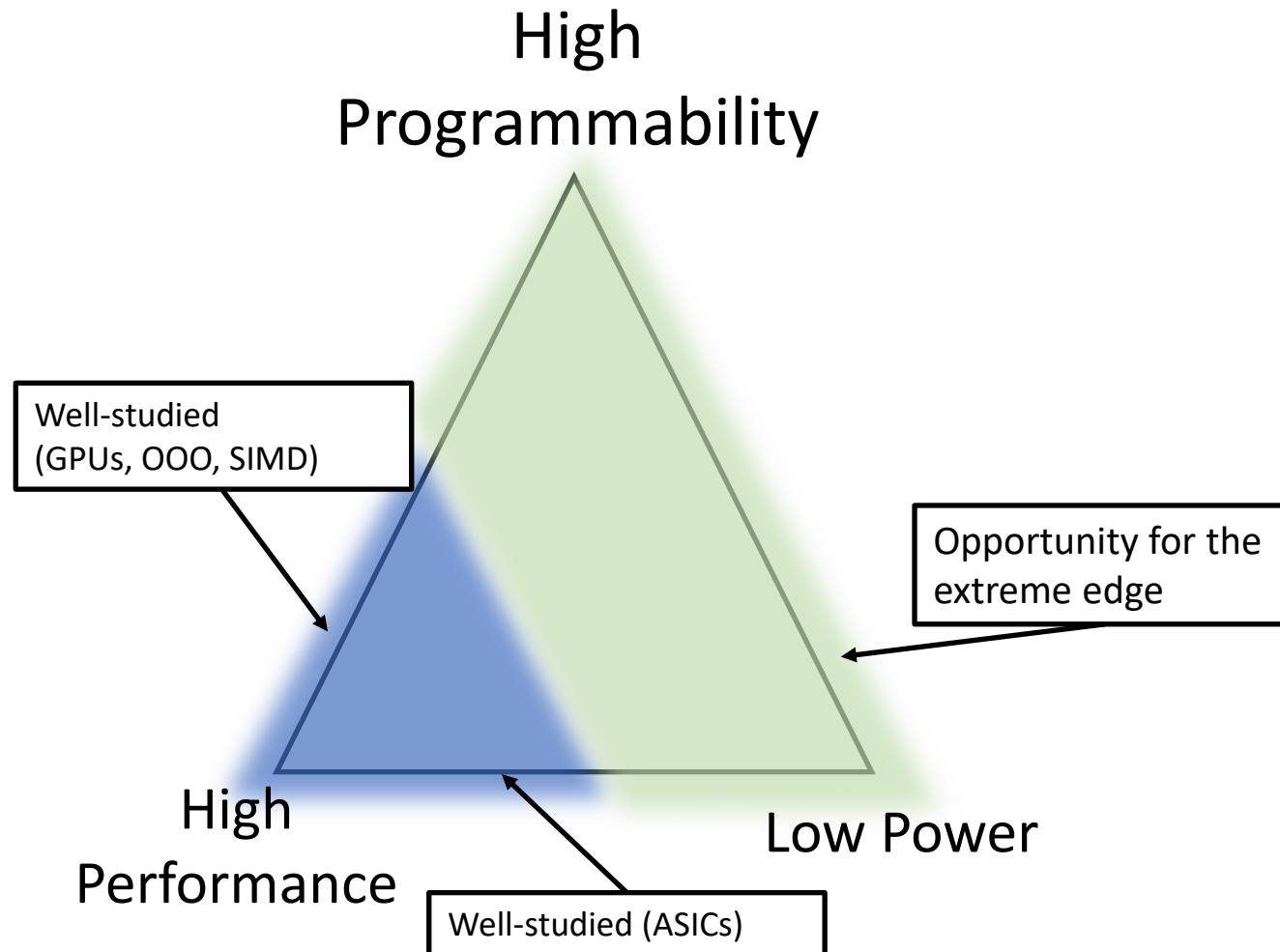


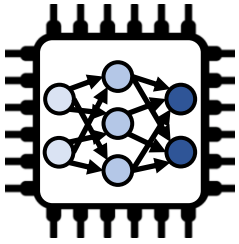
ASICs/Accelerators would improve this, but forfeit **programmability**

\*Horowitz ISSCC 2014 + measured values



# Fundamental extreme edge trade-offs



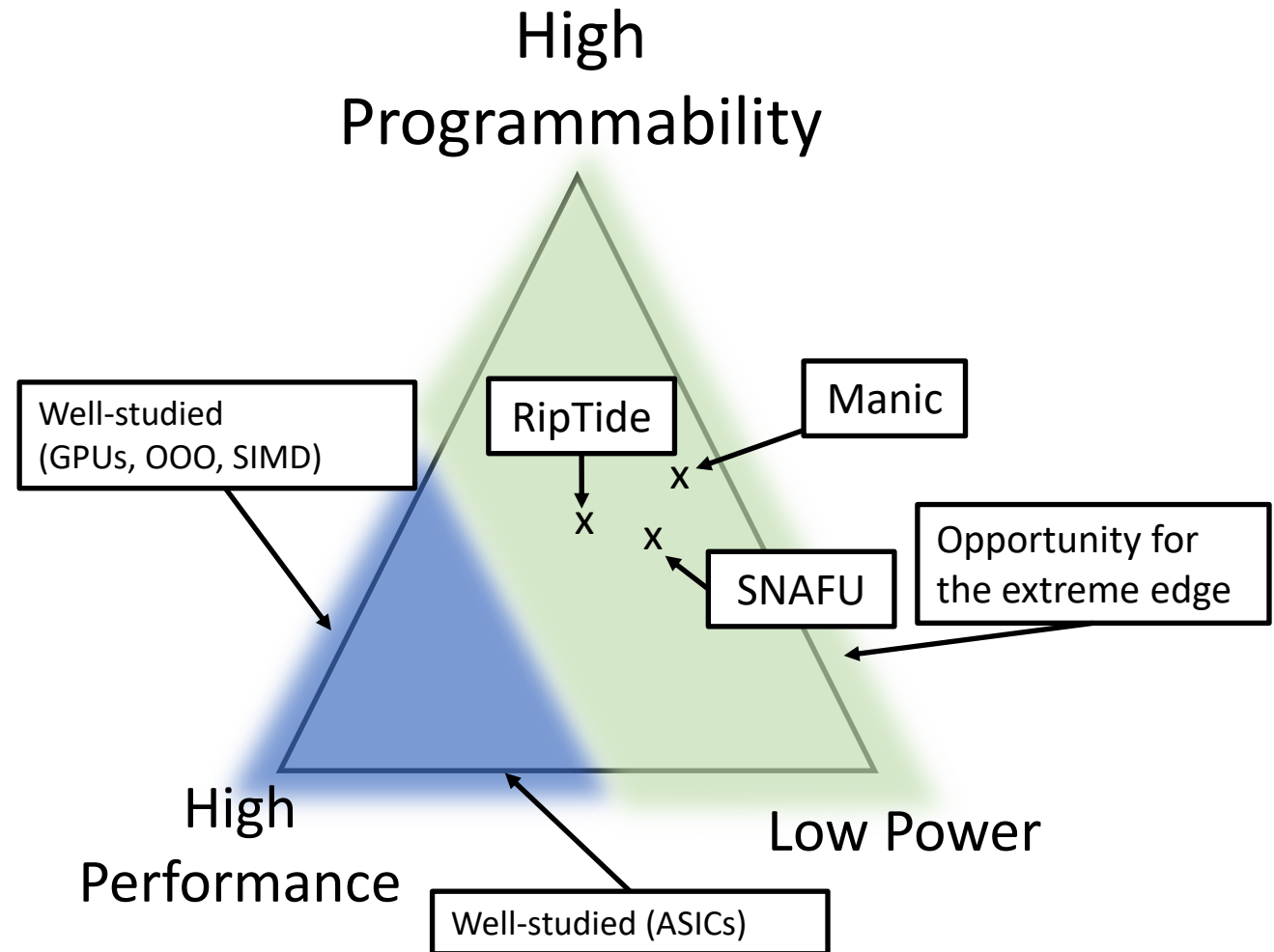


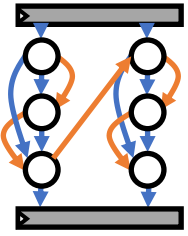
# Fundamental extreme edge trade-offs

## Key Idea:

Different architecture, different set of tradeoffs

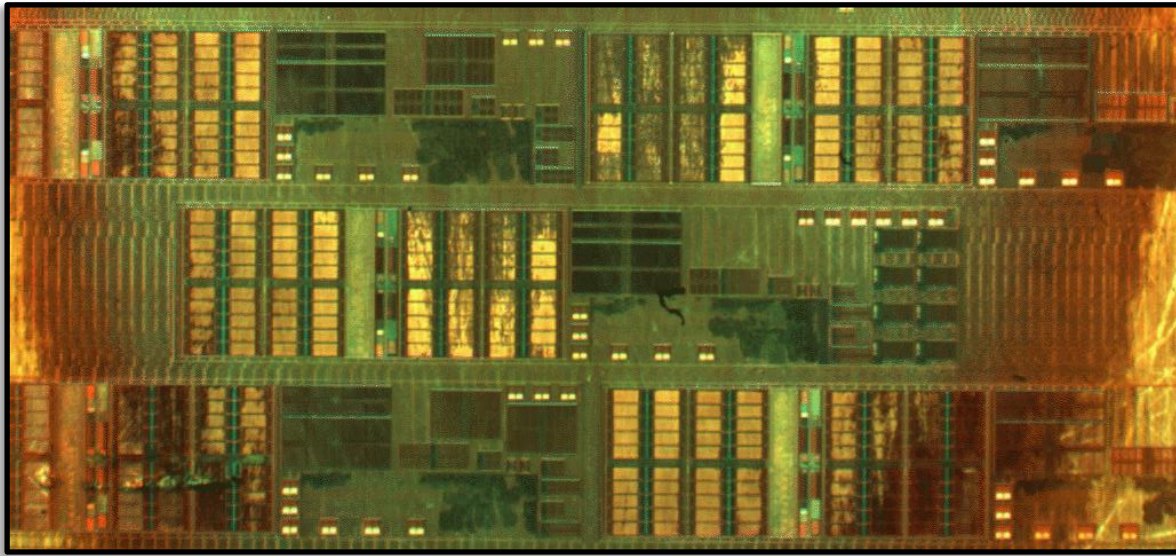
Extreme edge applications demand **programmable & energy-minimal** architectures



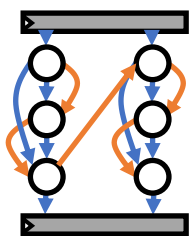


# MANIC: Extreme Edge Vector-dataflow processor

- Reduce instruction supply energy + VRF energy
- Maintain high-degree of programmability to support future kernels



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓



# Scalar execution model

## Example Program

for i in 0...3:

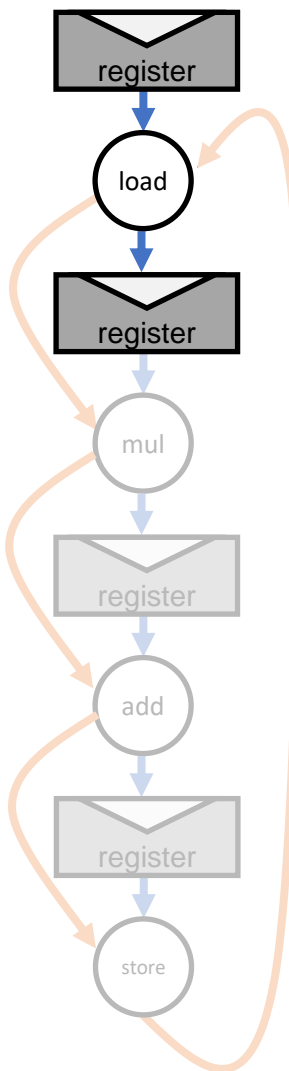
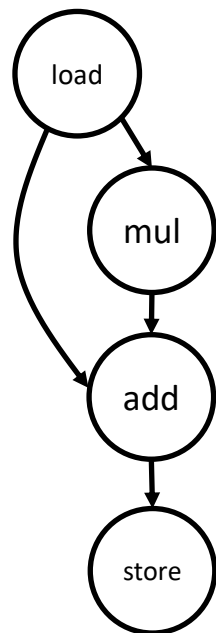
**load r0, &a[i]**

mul r1, r0, r0

add r2, r1, r0

store &b[i], r2

## Dataflow



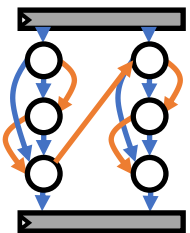
Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

 Dataflow

 Control-flow

Related: MSP430, ARM M0





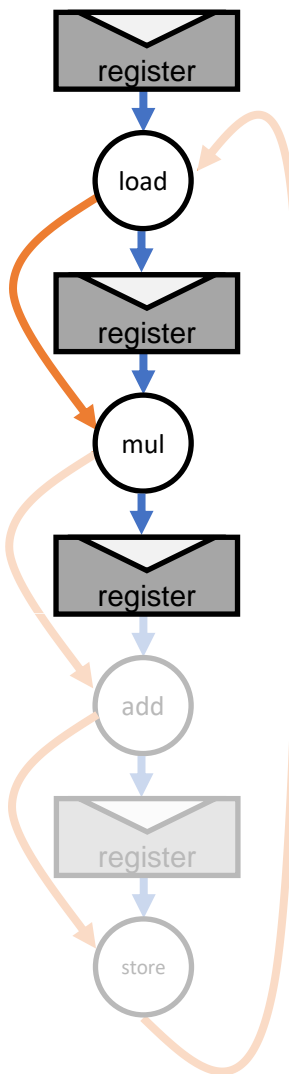
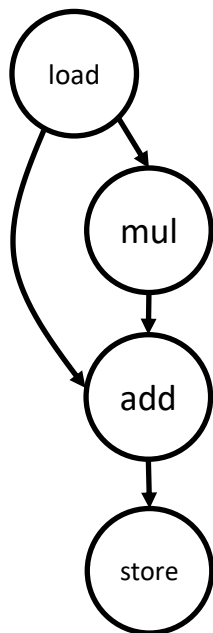
# Scalar execution model

## Example Program

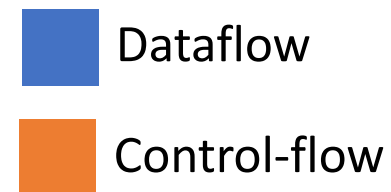
```

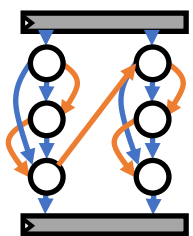
for i in 0...3:
  load r0, &a[i]
  mul r1, r0, r0
  add r2, r1, r0
  store &b[i], r2
  
```

## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓





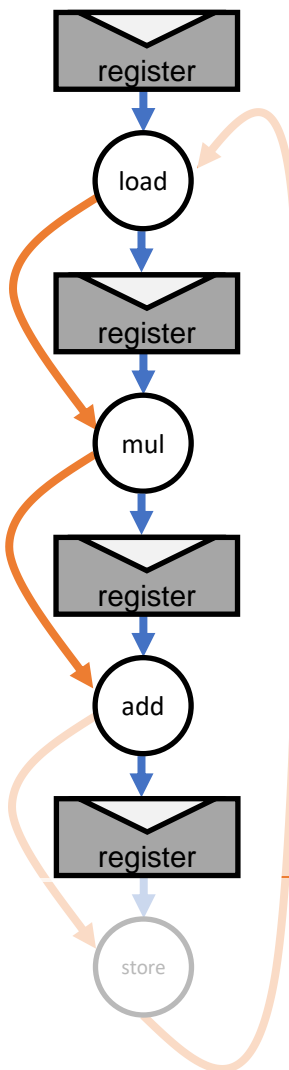
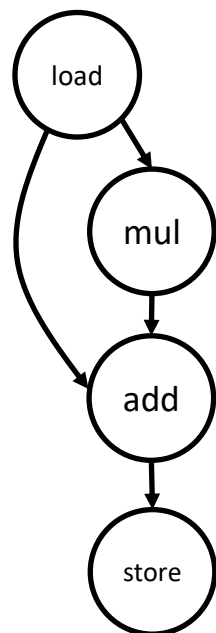
# Scalar execution model

## Example Program

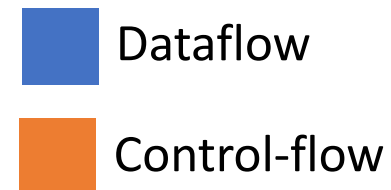
```

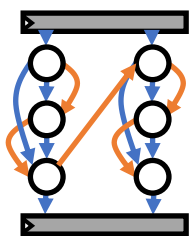
for i in 0...3:
  load r0, &a[i]
  mul r1, r0, r0
  add r2, r1, r0
  store &b[i], r2
  
```

## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓





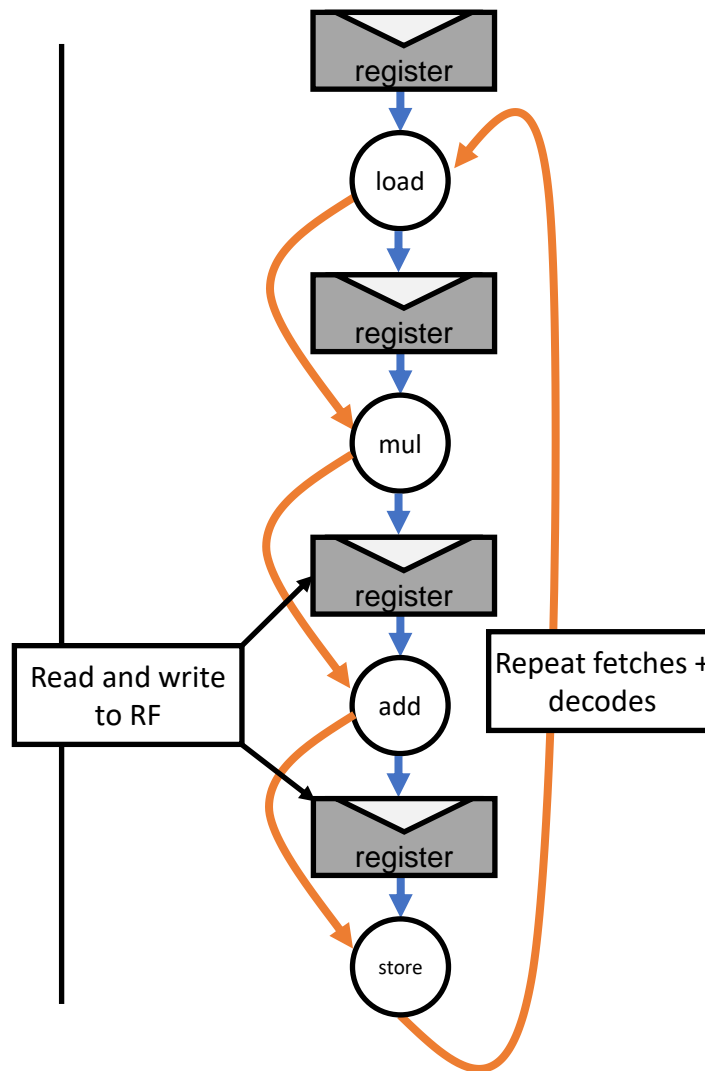
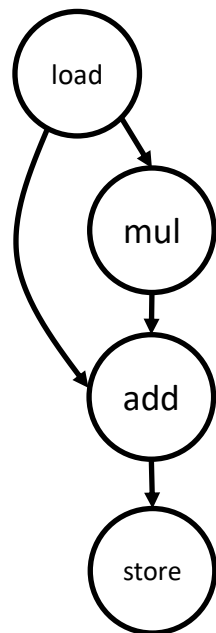
# Scalar execution model

## Example Program

```

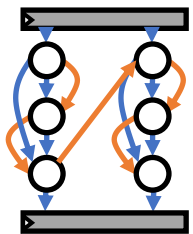
for i in 0...3:
  load r0, &a[i]
  mul r1, r0, r0
  add r2, r1, r0
  store &b[i], r2
  
```

## Dataflow



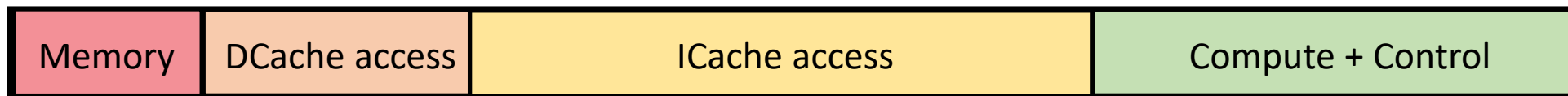
Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

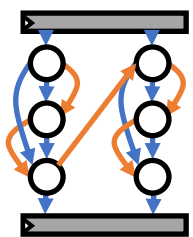




# Scalar execution is inefficient

- Energy wasted on instruction & data supply





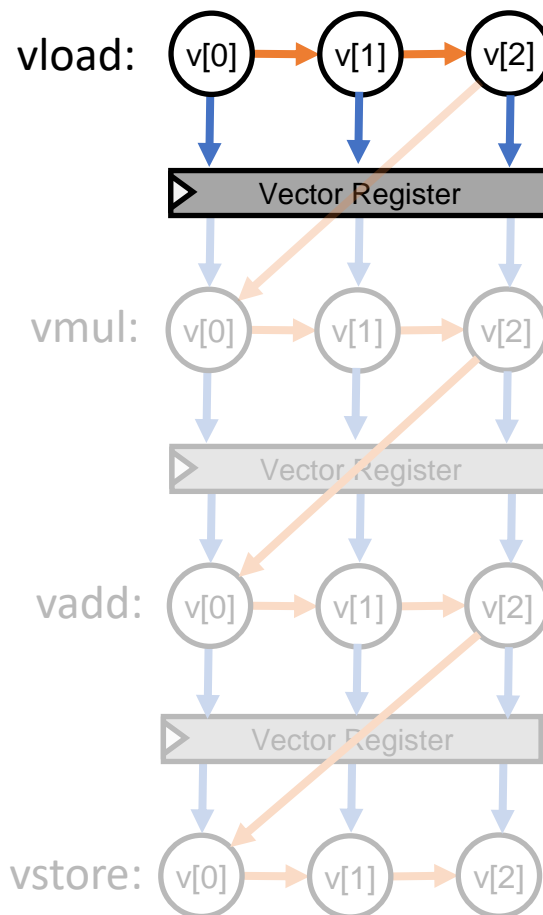
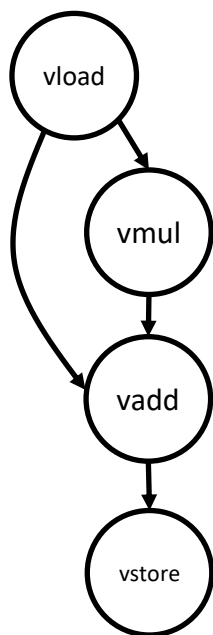
# Vector execution

## Example Program

```

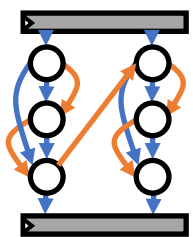
vload v0, &a
vmul v1, v0, v0
vadd v2, v1, v0
vstore &b, v2
  
```

## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

 Dataflow  
 Control-flow



# Vector execution

## Example Program

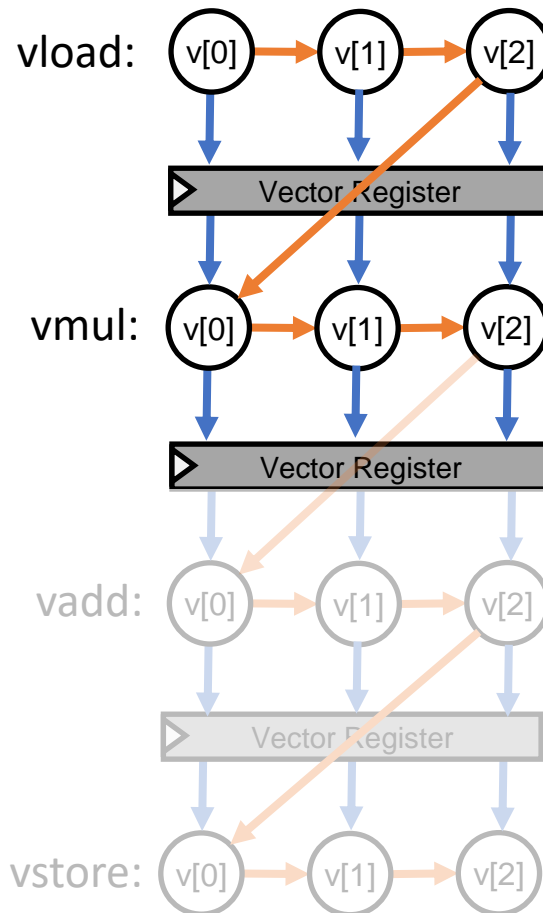
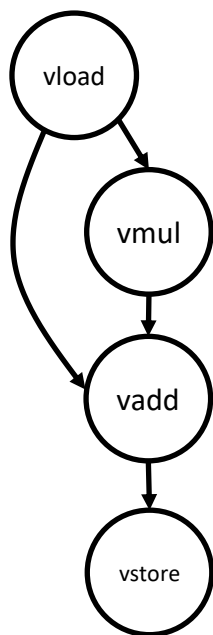
vload v0, &a

► **vmul v1, v0, v0**

vadd v2, v1, v0

vstore &b, v2

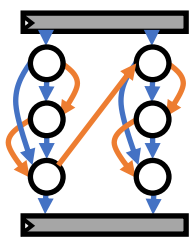
## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

■ Dataflow

■ Control-flow



# Vector execution

## Example Program

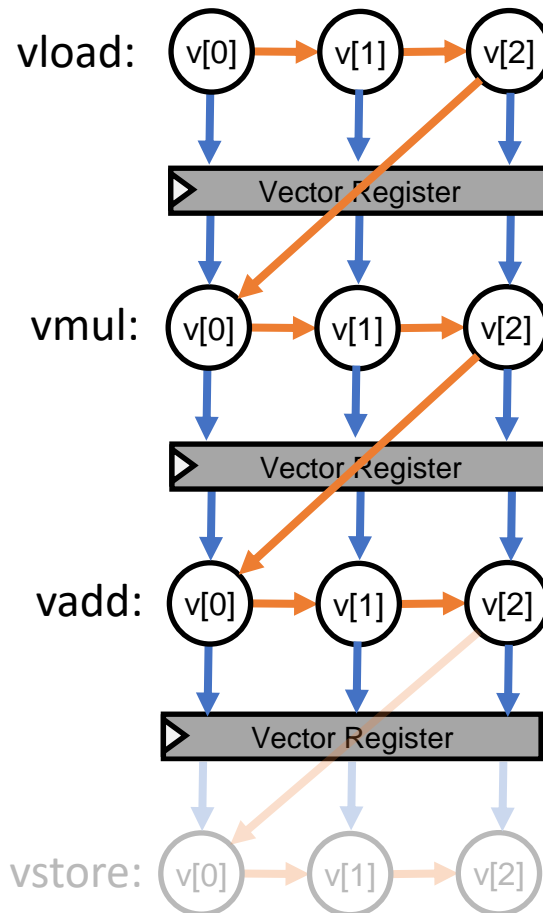
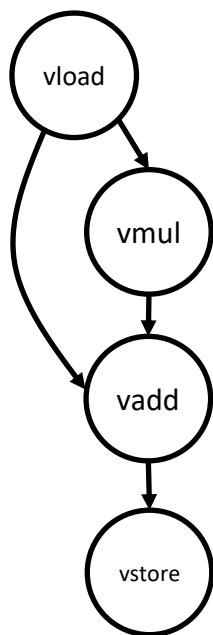
vload v0, &a

vmul v1, v0, v0

➡ **vadd v2, v1, v0**

vstore &b, v2

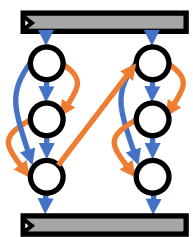
## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

■ Dataflow

■ Control-flow



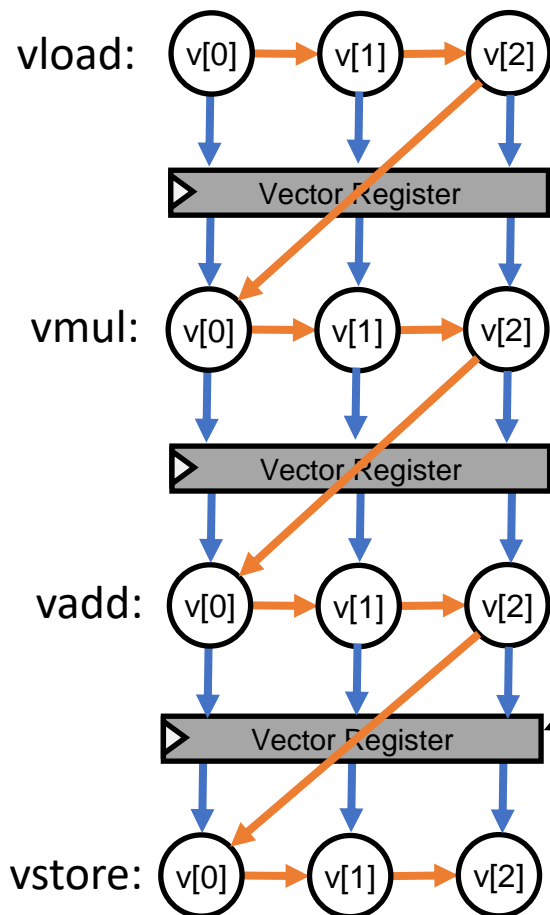
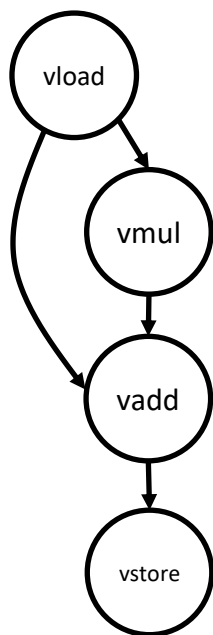
# Vector execution

## Example Program

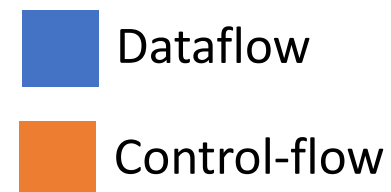
```

vload v0, &a
vmul v1, v0, v0
vadd v2, v1, v0
vstore &b, v2
  
```

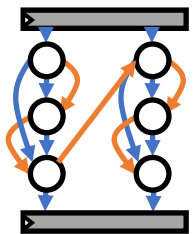
## Dataflow



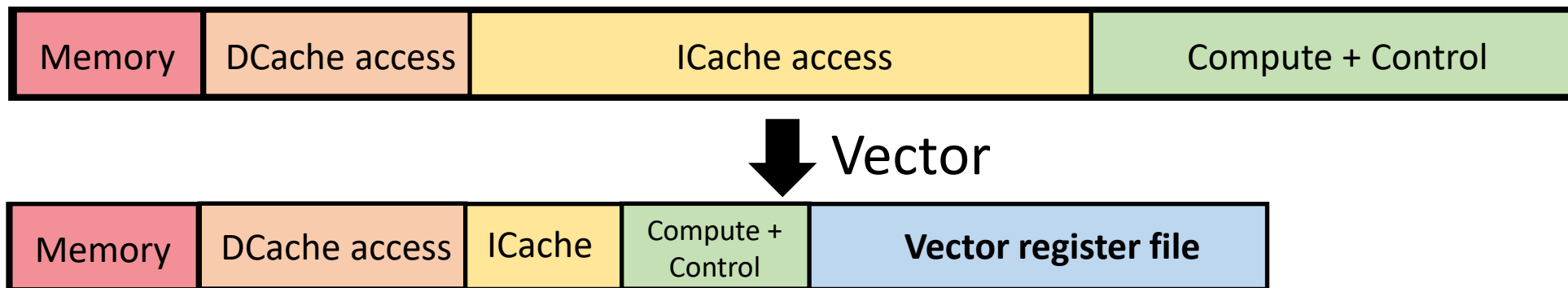
Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

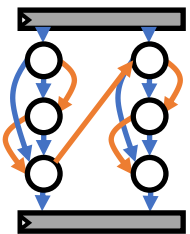






# Vector pays huge energy cost for VRF writes





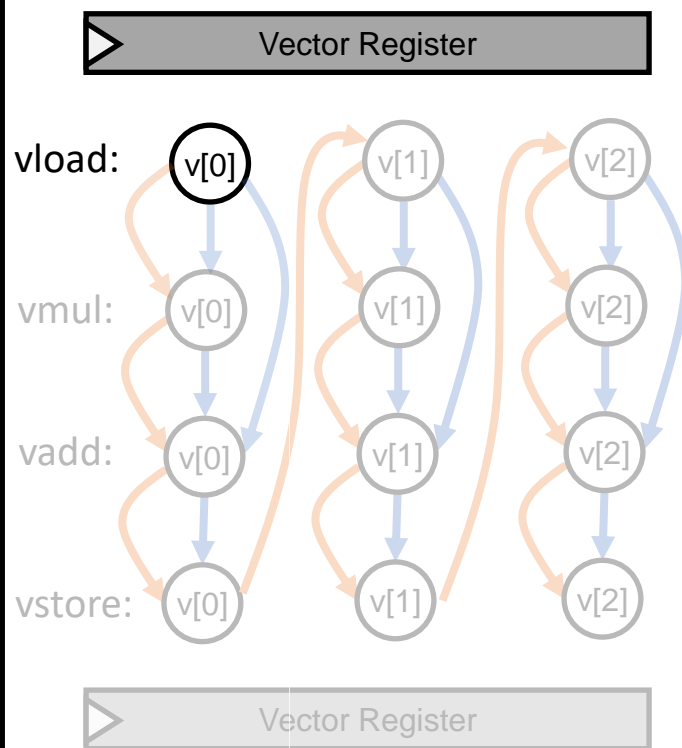
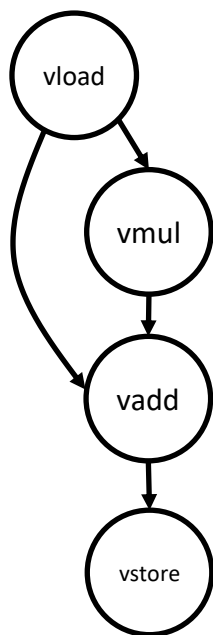
# MANIC's Vector-dataflow execution

## Example Program



```

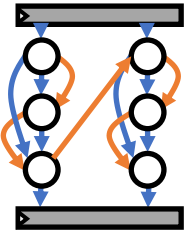
vload v0, &a
vmul v1, v0, v0
vadd v2, v1, v0
vstore &b, v2
  
```

## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

 Dataflow  
 Control-flow



# MANIC's Vector-dataflow execution

## Example Program

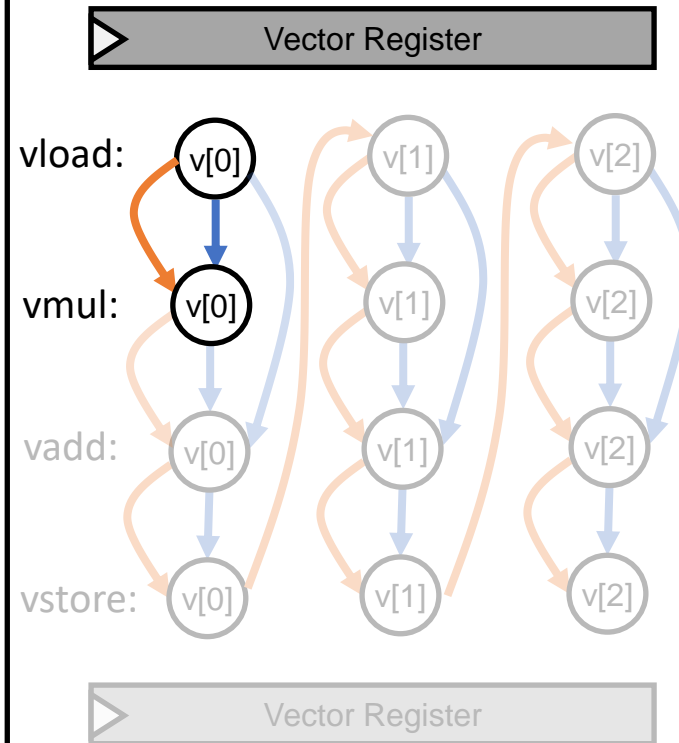
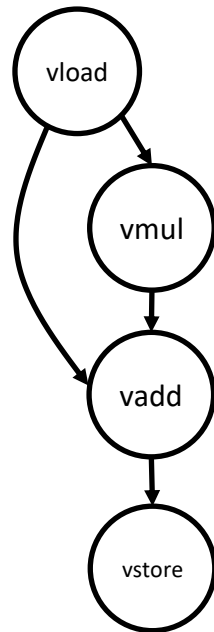
vload v0, &a

➡ **vmul v1, v0, v0**

vadd v2, v1, v0

vstore &b, v2

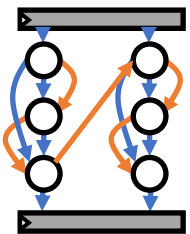
## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

■ Dataflow

■ Control-flow



# MANIC's Vector-dataflow execution

## Example Program

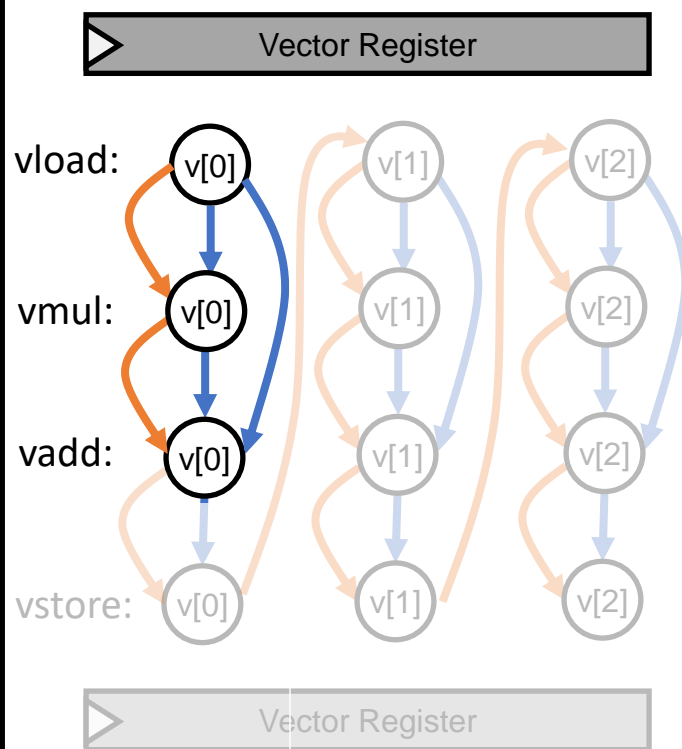
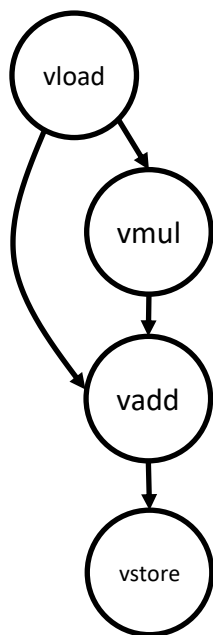
vload v0, &a

vmul v1, v0, v0

➡ **vadd v2, v1, v0**

vstore &b, v2

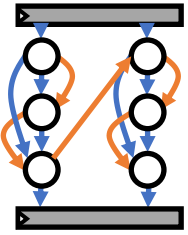
## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

■ Dataflow

■ Control-flow

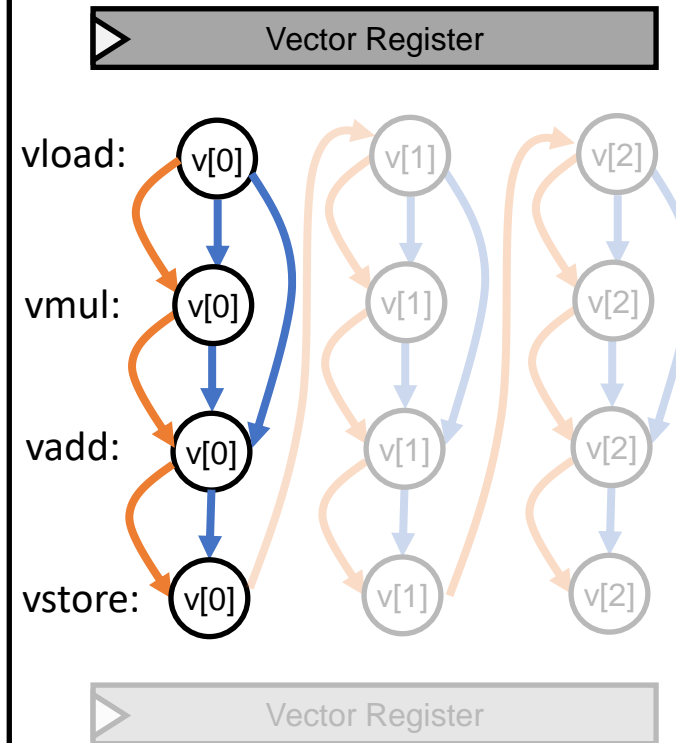
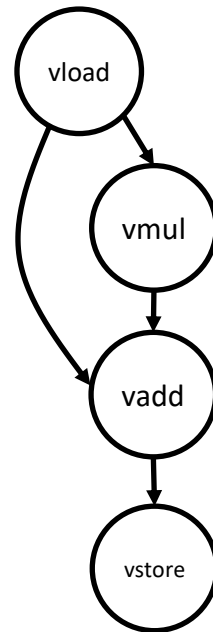


# MANIC's Vector-dataflow execution

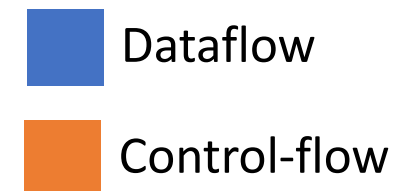
## Example Program

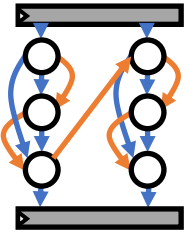
```
vload v0, &a
vmul v1, v0, v0
vadd v2, v1, v0
➡ vstore &b, v2
```

## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓





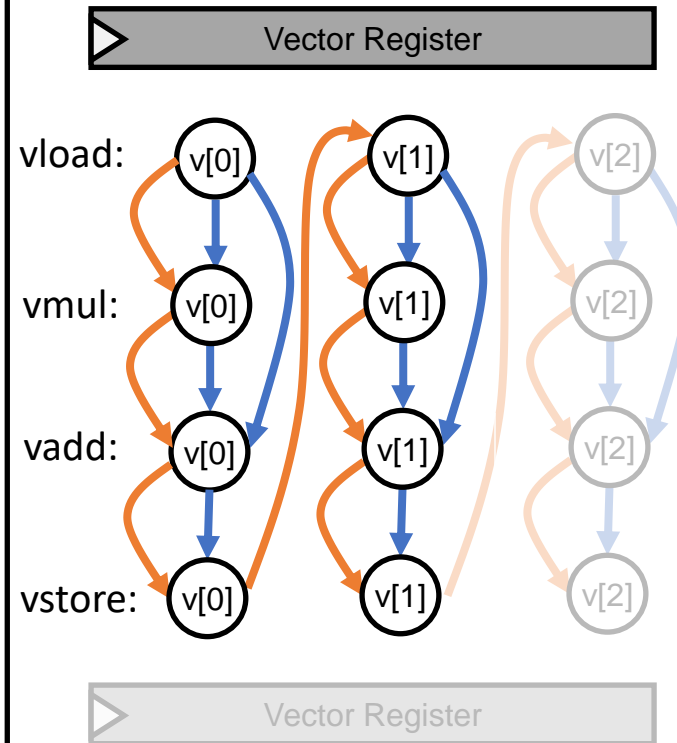
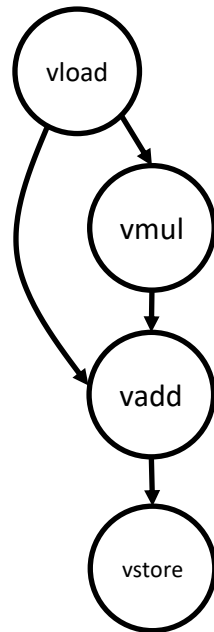
# MANIC's Vector-dataflow execution

## Example Program



```

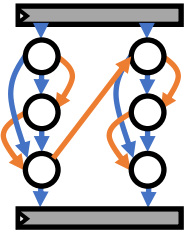
vload v0, &a
vmul v1, v0, v0
vadd v2, v1, v0
vstore &b, v2
  
```

## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

 Dataflow  
 Control-flow



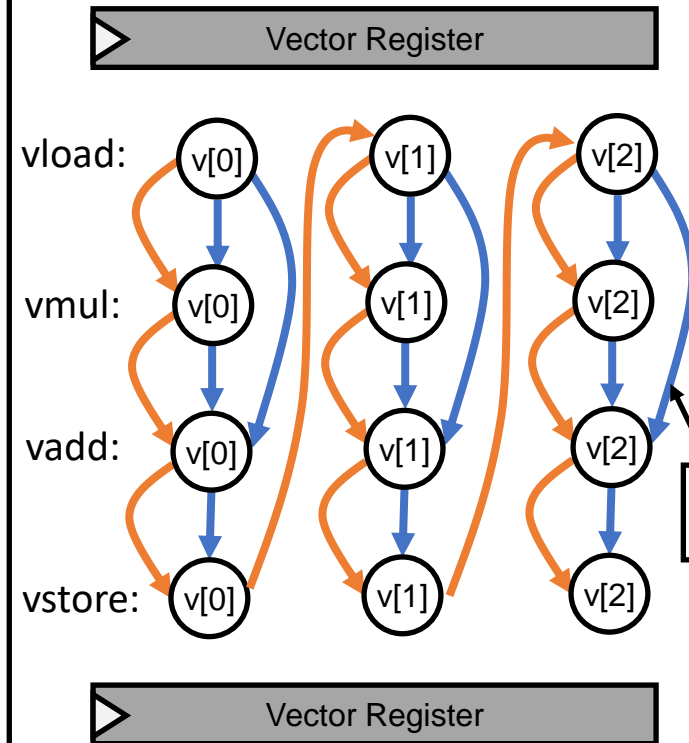
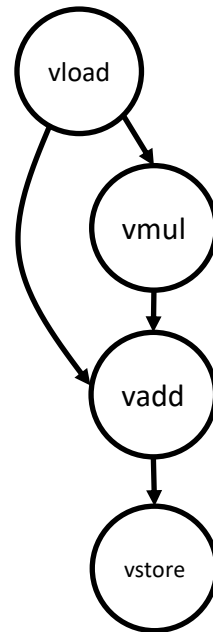
# MANIC's Vector-dataflow execution

## Example Program

```



vload v0, &a
vmul v1, v0, v0
vadd v2, v1, v0
vstore &b, v2
  
```

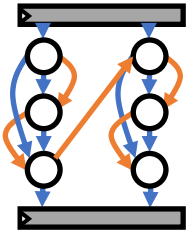
## Dataflow



Energy			
Model	Insns	RF Reads	RF Writes
Scalar	↑	↑	↑
Vector	↓	↑	↑
Vector-Dataflow	↓	↓	↓

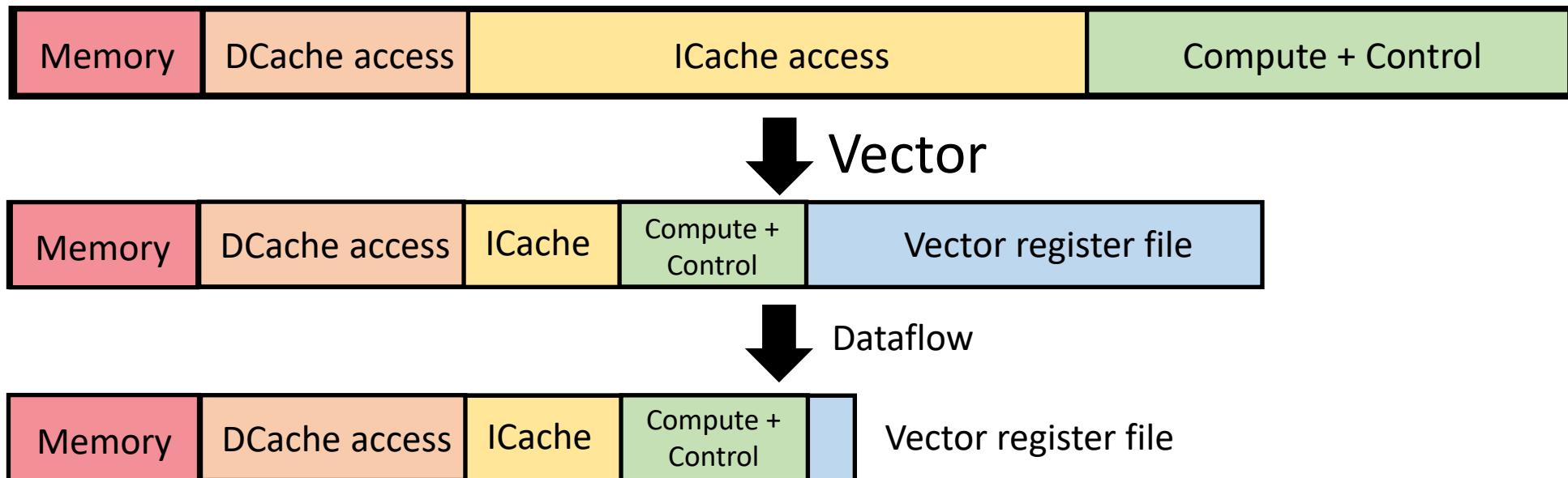
Dataflow Forwarding

 Dataflow  
 Control-flow



# Vector-dataflow reduces energy without costing programmability

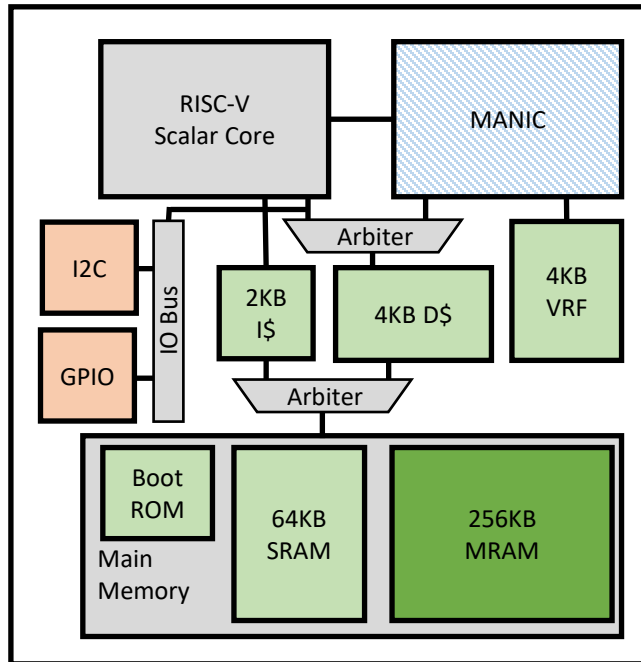
- Vector-dataflow execution
  - Vector execution **reduces instructions fetched**
  - Dataflow execution **eliminates VRF reads**
- Software support to **eliminate VRF writes**





# MANIC is an energy-minimal computer architecture implementing vector-dataflow

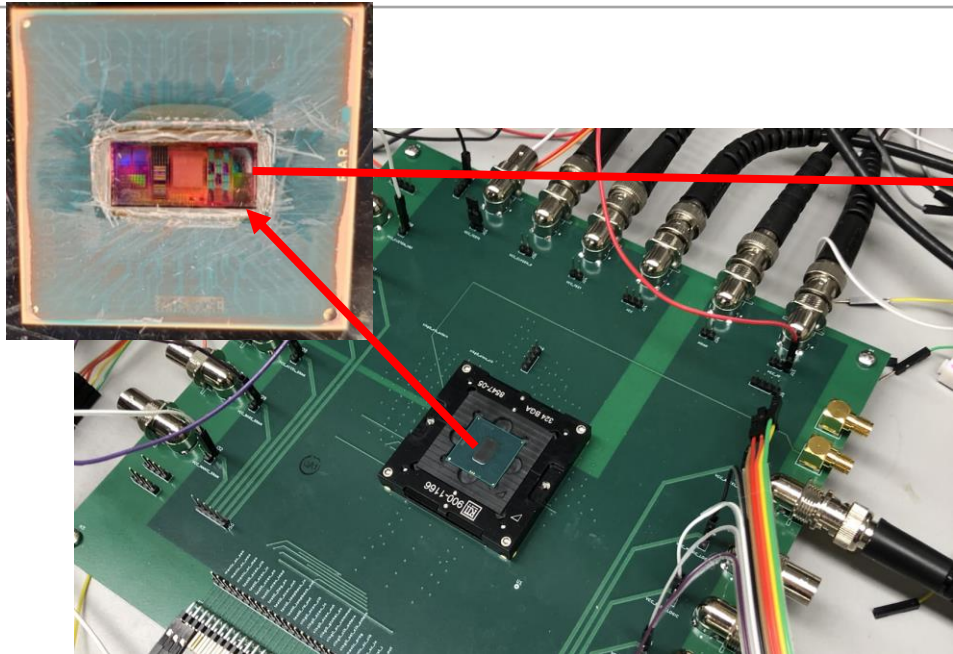
Block diagram



## Implementation Characteristics

- Complete standalone system
- Scalar, Vector & MANIC designs
- Intel 22nm bulk FinFet (HVT)
- Embedded MRAM
- SRAM, logic, MRAM power isolated

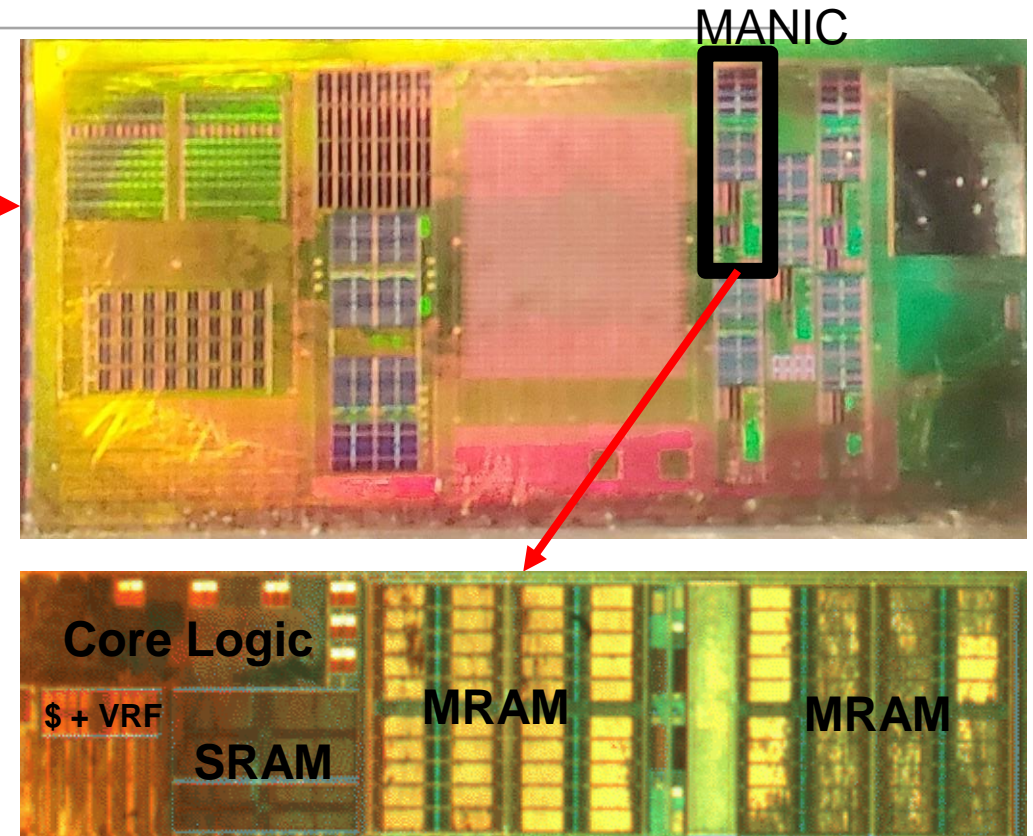
# Evaluating MANIC's efficiency in a silicon prototype



Intel 22nm FinFET 8 metal layers, MANIC + Vector + Scalar, 256kB MRAM + 64kB SRAM

**Evaluation Goals:** Energy characterization of first ever vector-dataflow chip.

**Key Result:** Power low & efficiency high enough to run on tiny solar panel indoors



**Operational Characteristics:**

**Frequency:** 4-50MHz

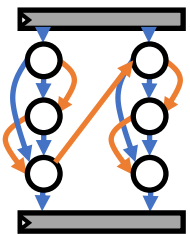
**Voltage:** 0.4-1.0V

**Power:** 19.1uW

**Efficiency:** 256 GOPS/W

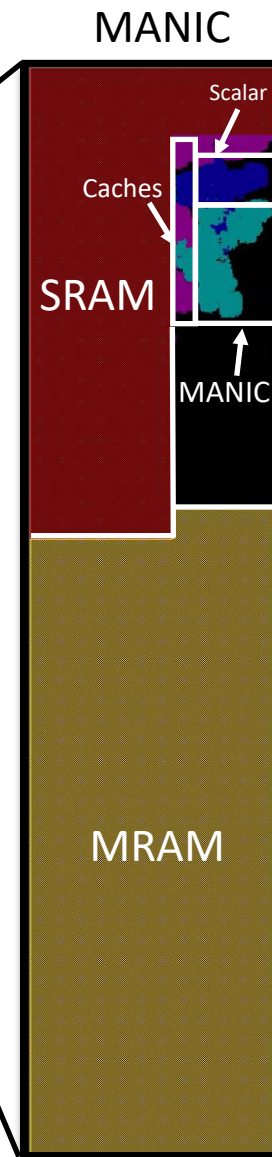
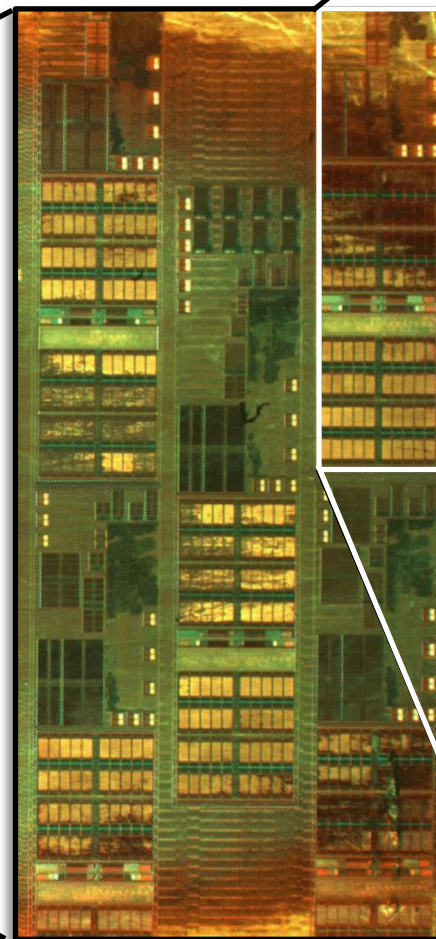
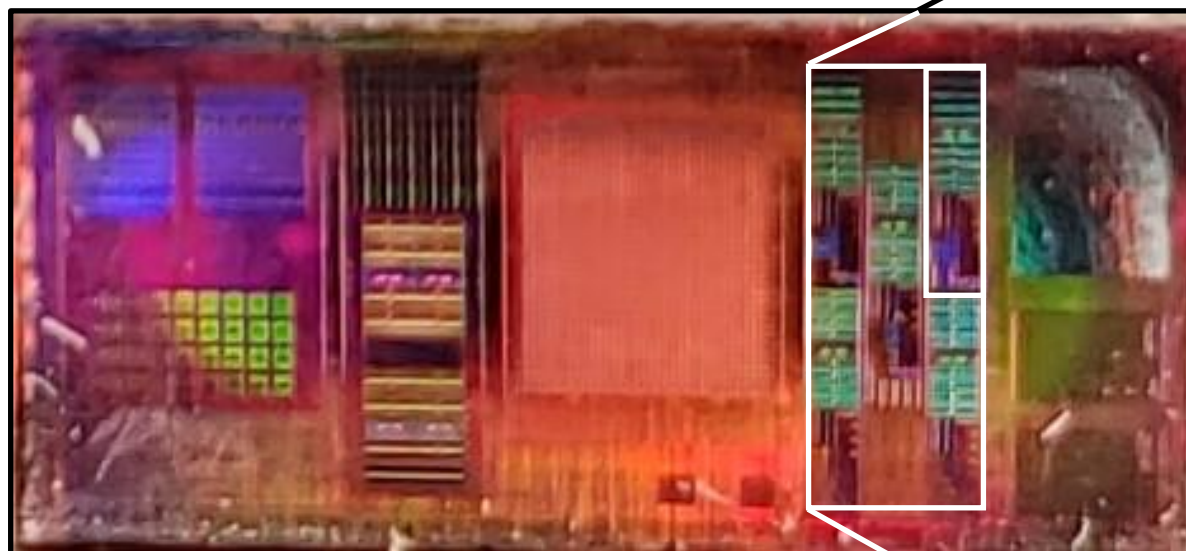
Carnegie Mellon University  
Electrical & Computer Engineering

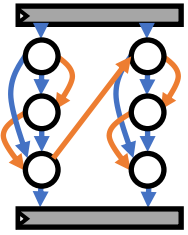




# More Pretty Chip Micrographs

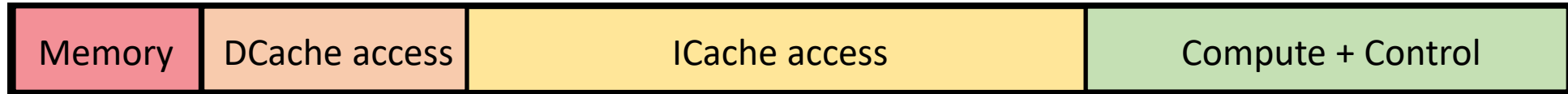
Courtesy CMU's Nanofabrication Laboratory & their electron microscope



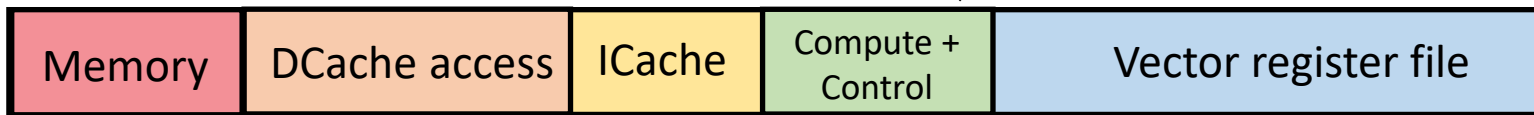


# Can we do even better?

## Let's eliminate all instruction control & caching costs!



↓ Vector



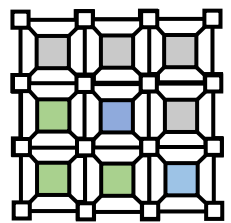
↓ Dataflow



↓ Coarse-grained Reconfigurable Array Architectures

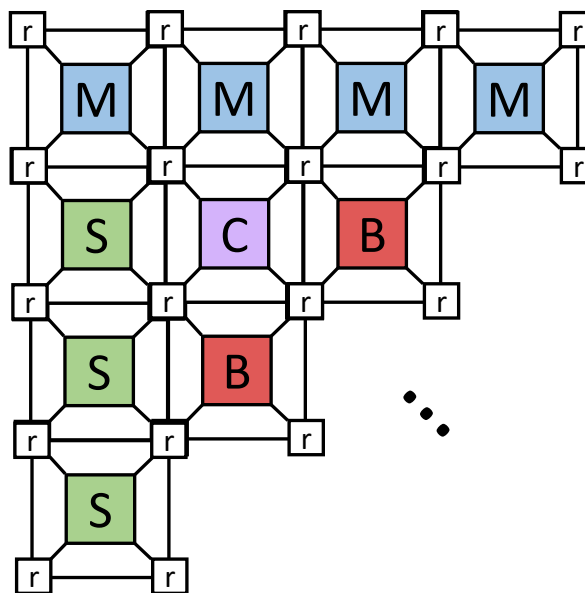


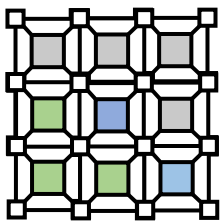
**Key idea: architecture eliminates instruction & control overheads**



# CGRA Overview

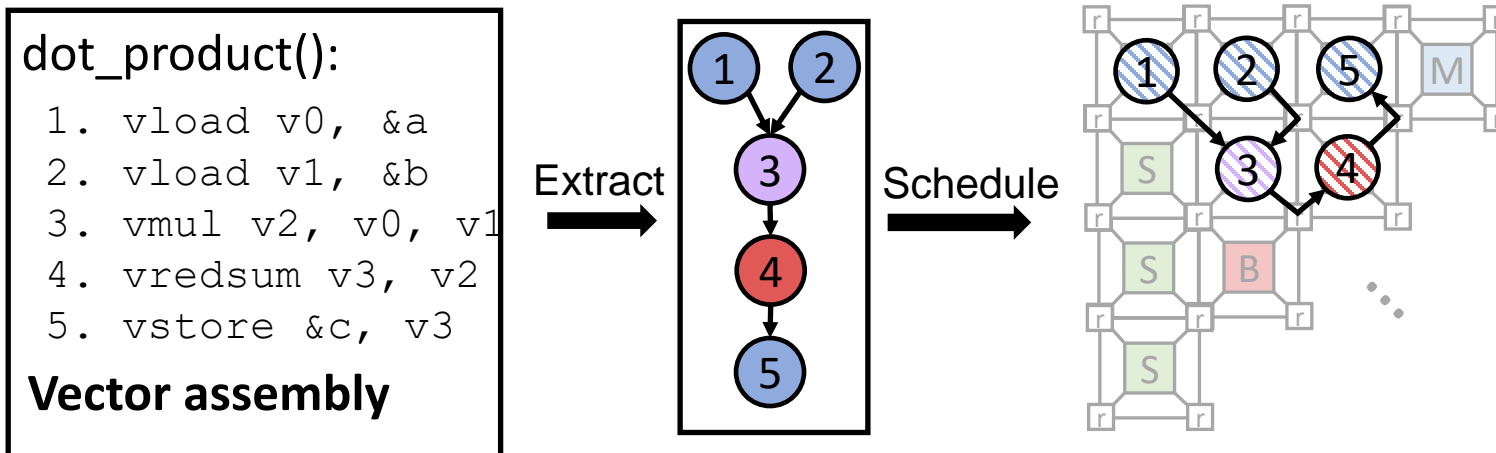
- Processing elements (PE) connected by Network-on-Chip (NoC)
  - Heterogenous PE capability
  - Connections configured by *software* compiler



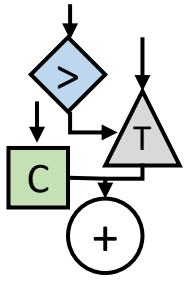


# CGRA Overview

- Collection of processing elements (PE) connected via NoC
  - Configure PE once, use many times: **no instruction fetch/control costs**
  - Data move directly PE to PE: **no RF/VRF/Cache costs**
  - Stream data through fabric: **Reduced memory costs**

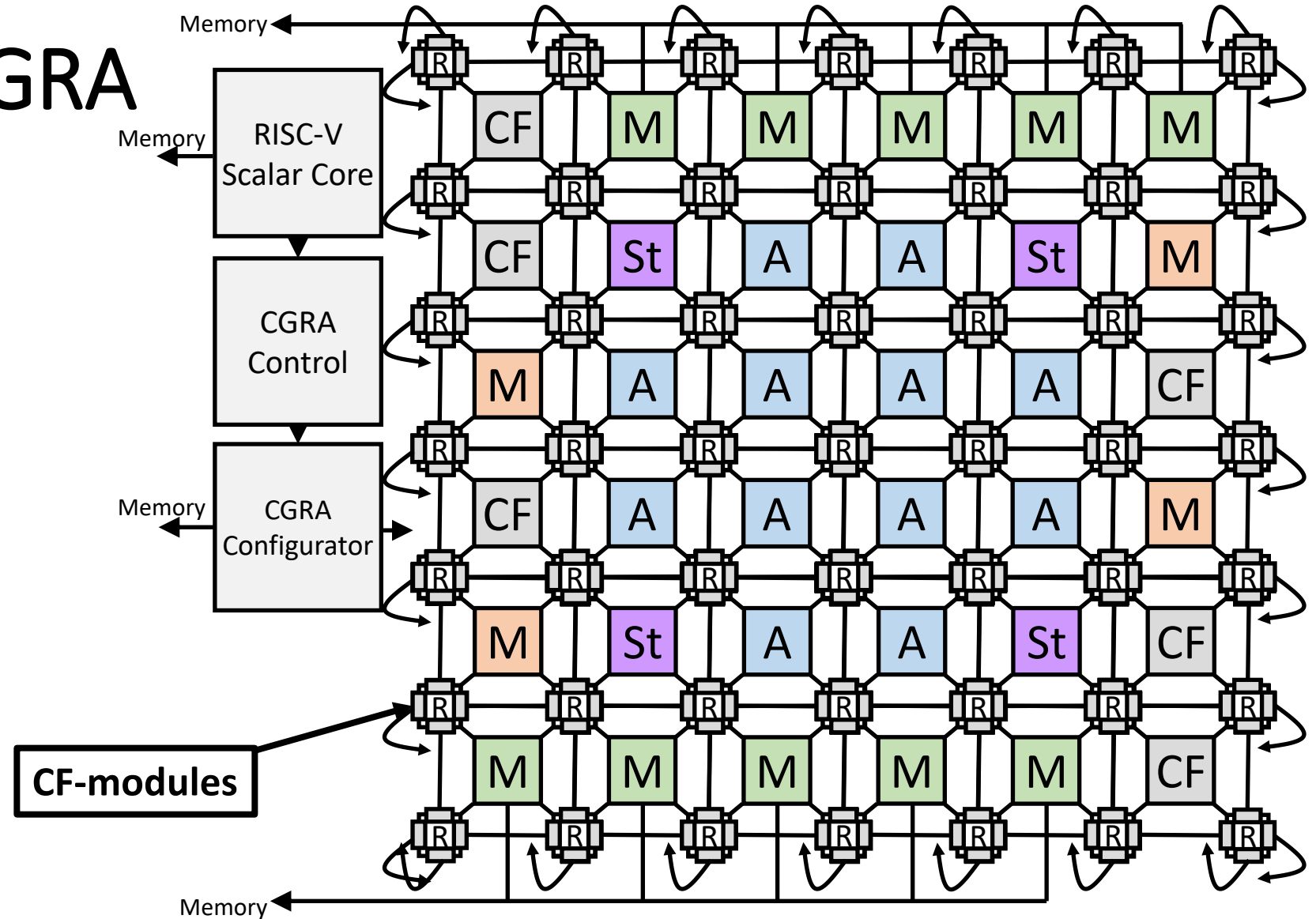


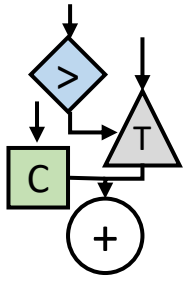
**Nearly all energy for actually useful computation!**



# RipTide CGRA

- M Memory
- M Multiplier
- St Stream
- A Arithmetic
- CF Control-flow





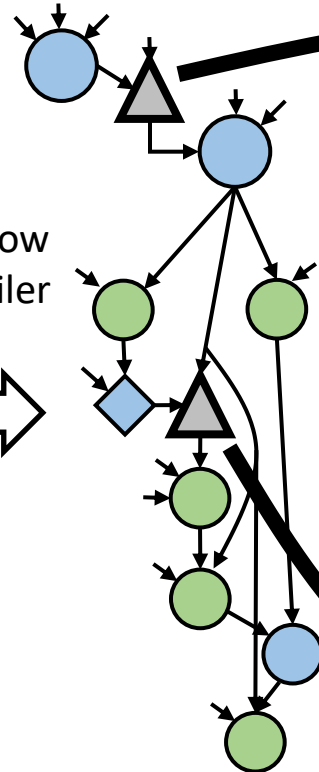
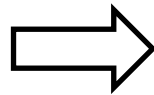
# CGRAs provide efficiency & programmability

**Dataflow** compiler support avoids the need for programmer acrobatics

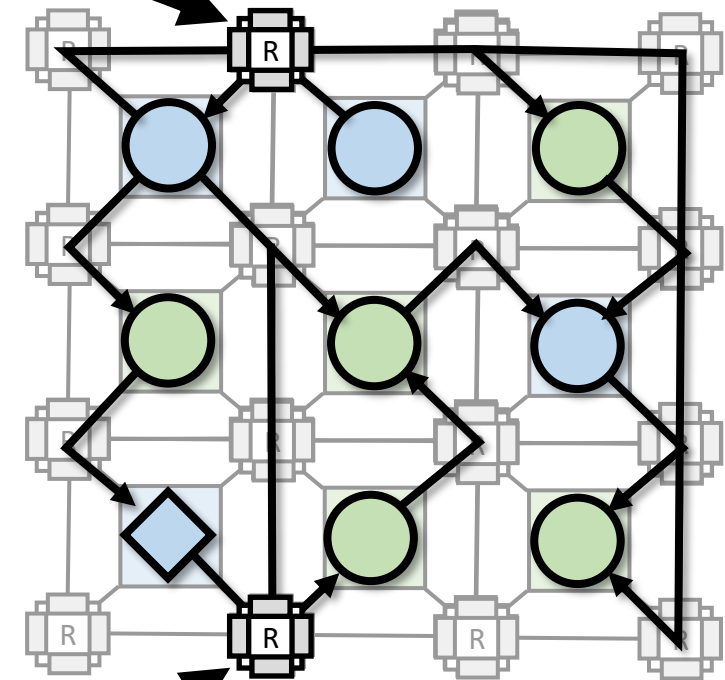
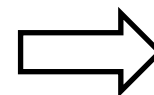
```
void test(int *a, int *b,  
         int *z, int n) {  
  for(int i = 0; i < n; i++) {  
    for(int j = 0; j < n; j++) {  
      if(a[j] < 0) {  
        z[j] = 0;  
      }  
      z[j] += b[j];  
    }  
  }  
}
```

C code

Dataflow  
Compiler



Optimized  
Dataflow Graph



RipTide CGRA



# Dataflow Architecture

LABORATORY FOR  
COMPUTER SCIENCE



MASSACHUSETTS  
INSTITUTE OF  
TECHNOLOGY

## The Varieties of Data Flow Computers

Computation Structures Group Memo 183-1  
August 1979  
Revised December 1979

Jack B. Dennis

# Dataflow Architecture: Dataflow Program Graphs

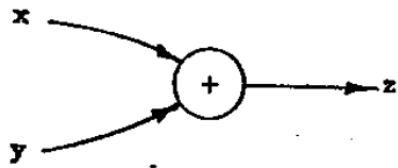


Fig. 2. Data flow actor.

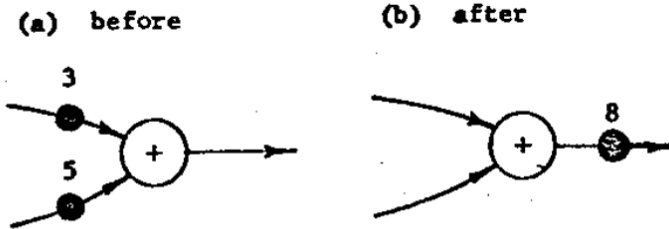


Fig. 3. Firing rule.

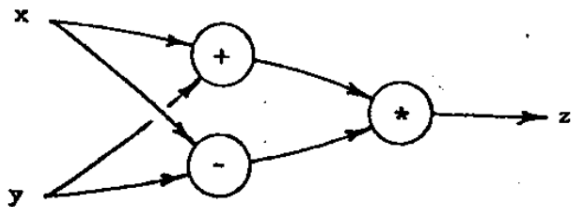


Fig. 4. Interconnection of operators.

```
if( x > 3 ){ x += 2 }
else{ x -= 1 }
y = x * 4
```

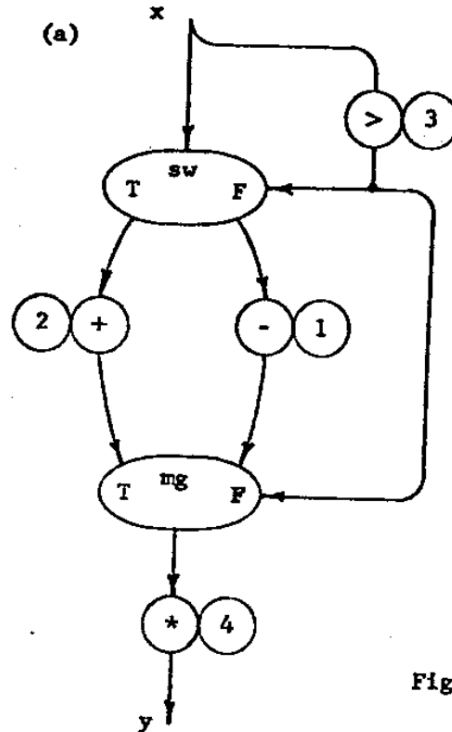
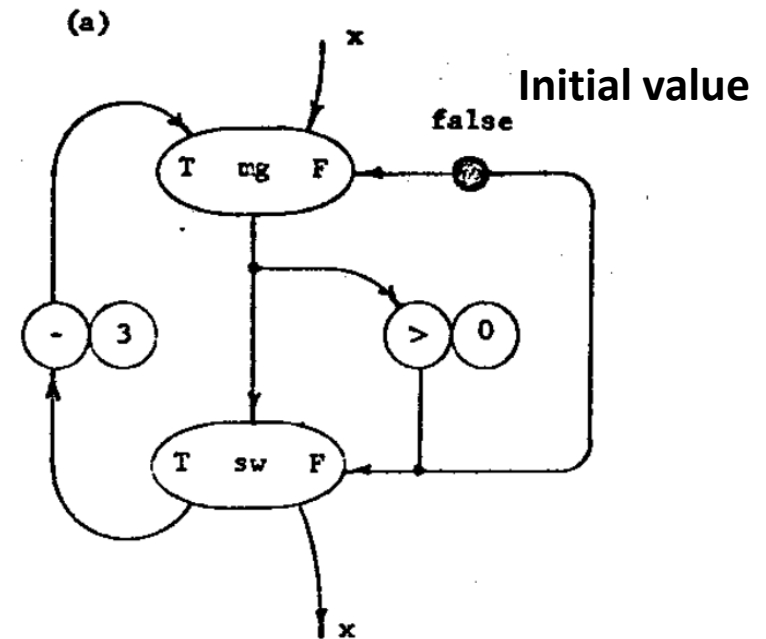


Fig.

Conditional operations  
represented as dataflow

```
while( x > 0 ){
    x -= 3
}
```



Loop represented as dataflow

# Dataflow: “Activity Template” implementation

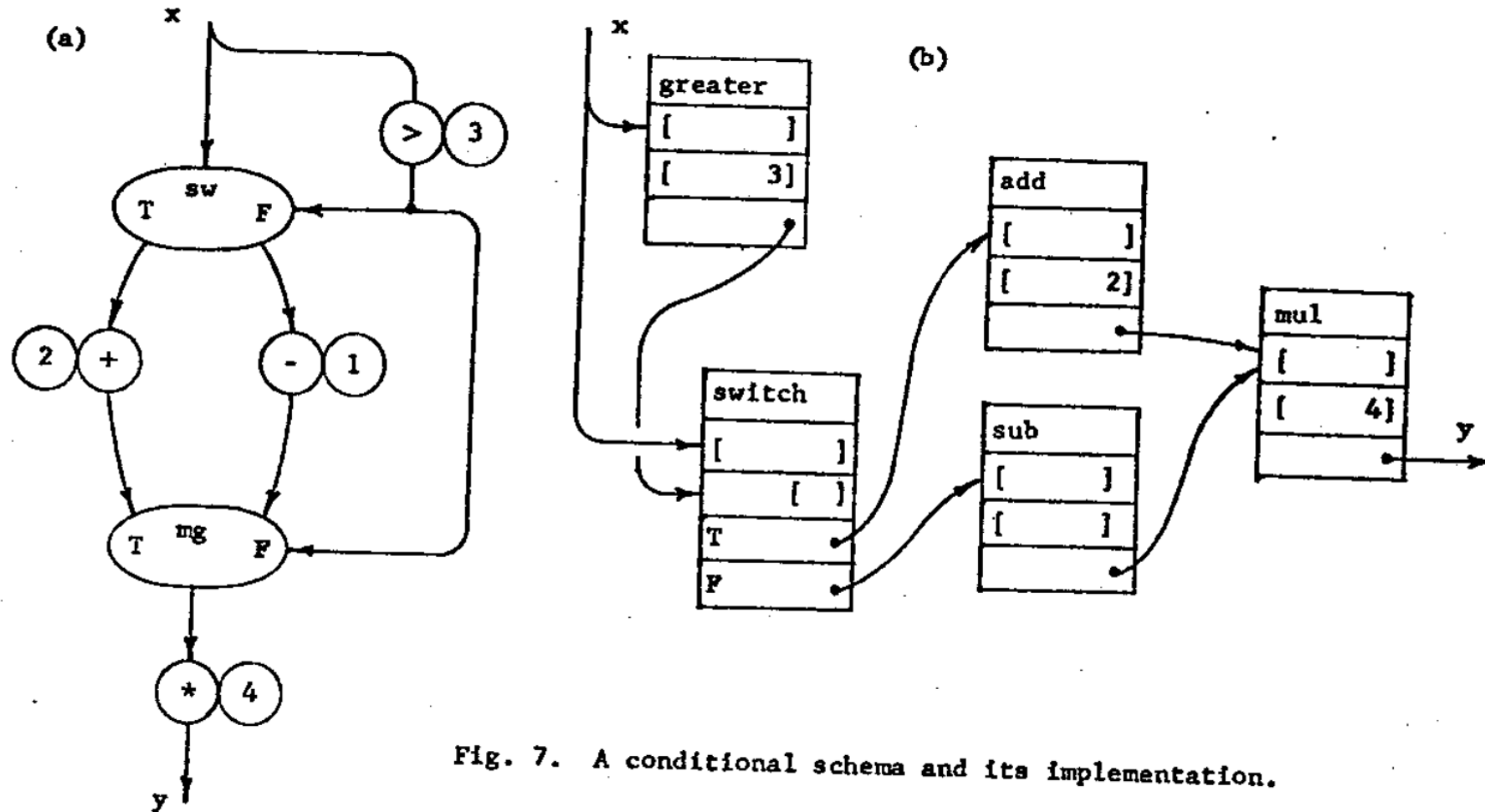
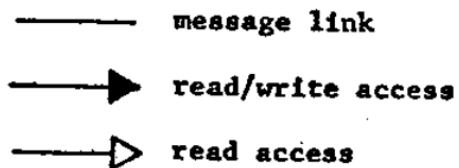
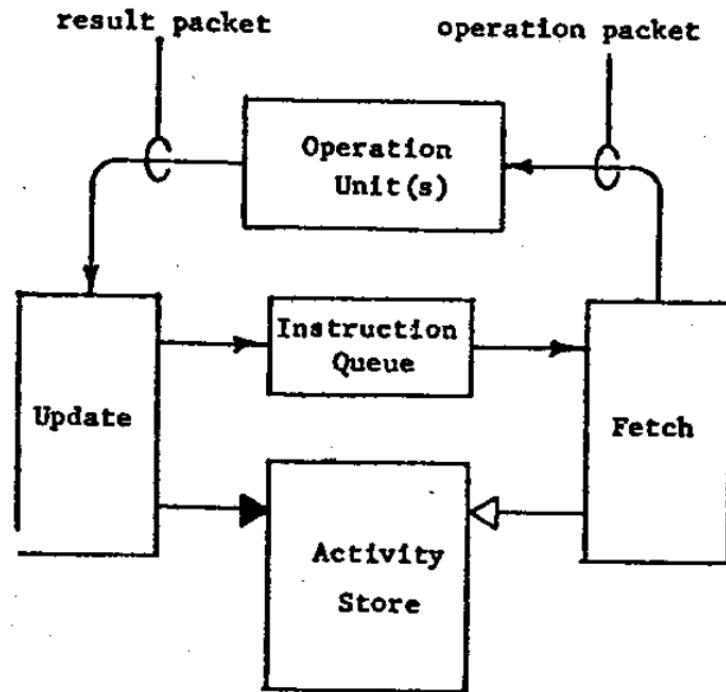
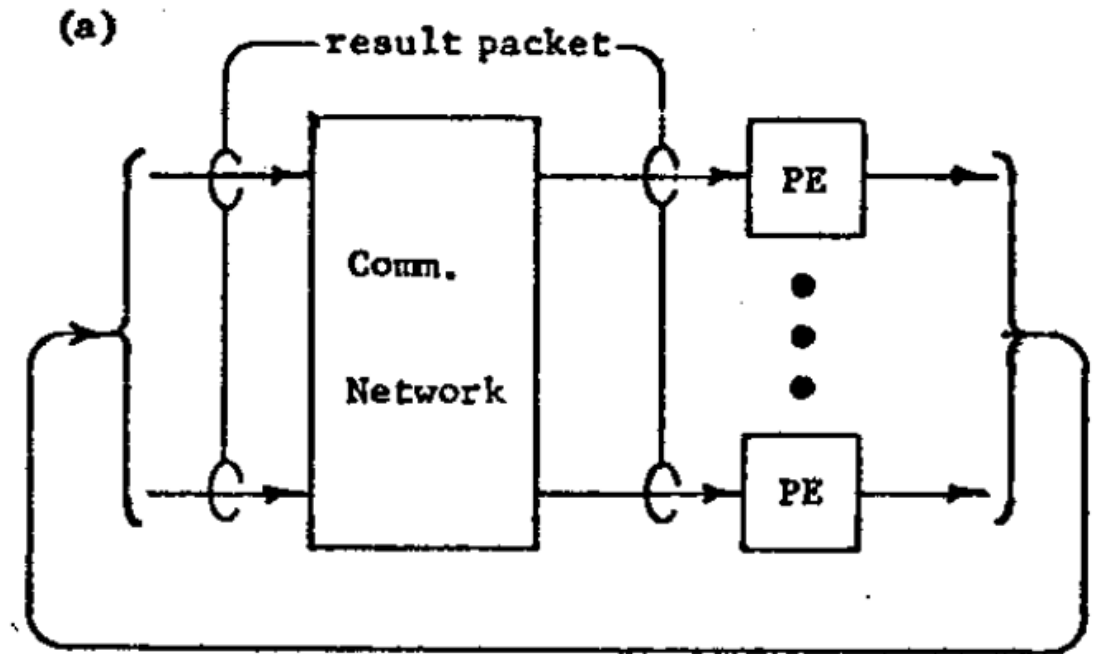


Fig. 7. A conditional schema and its implementation.

# Dataflow: Processing Element & Interconnect Arch.



Processing Element Architecture



Processing Element Interconnection Architecture

Question: what do we need to specify  
in this ISA?

# Dataflow: MIT Dataflow Architecture

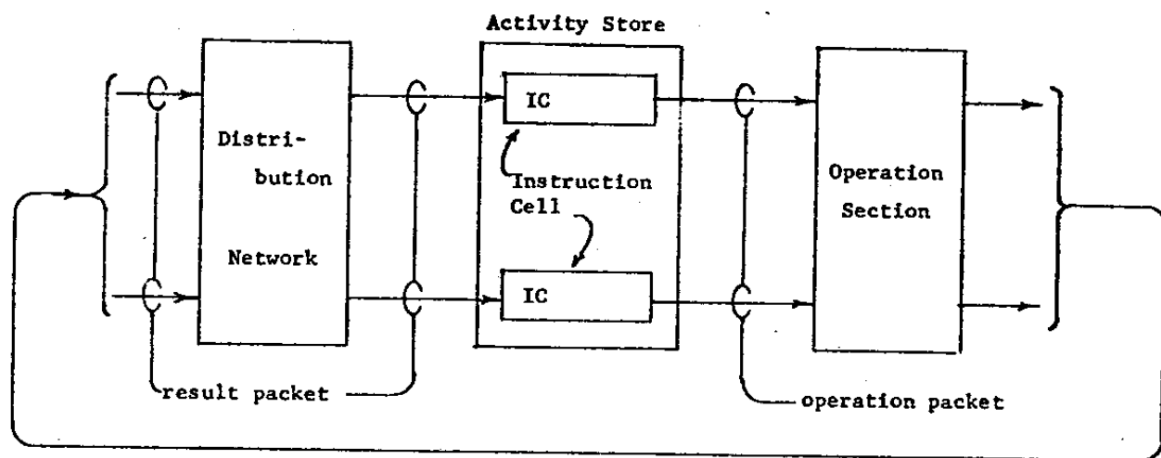


Fig. 14. MIT data flow processor.

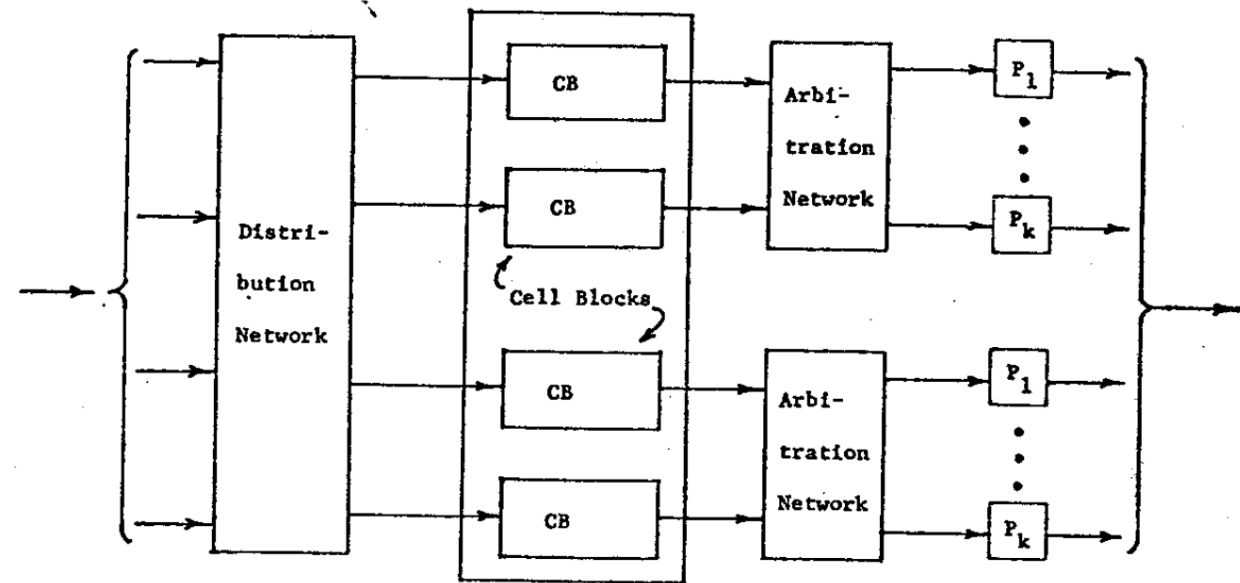
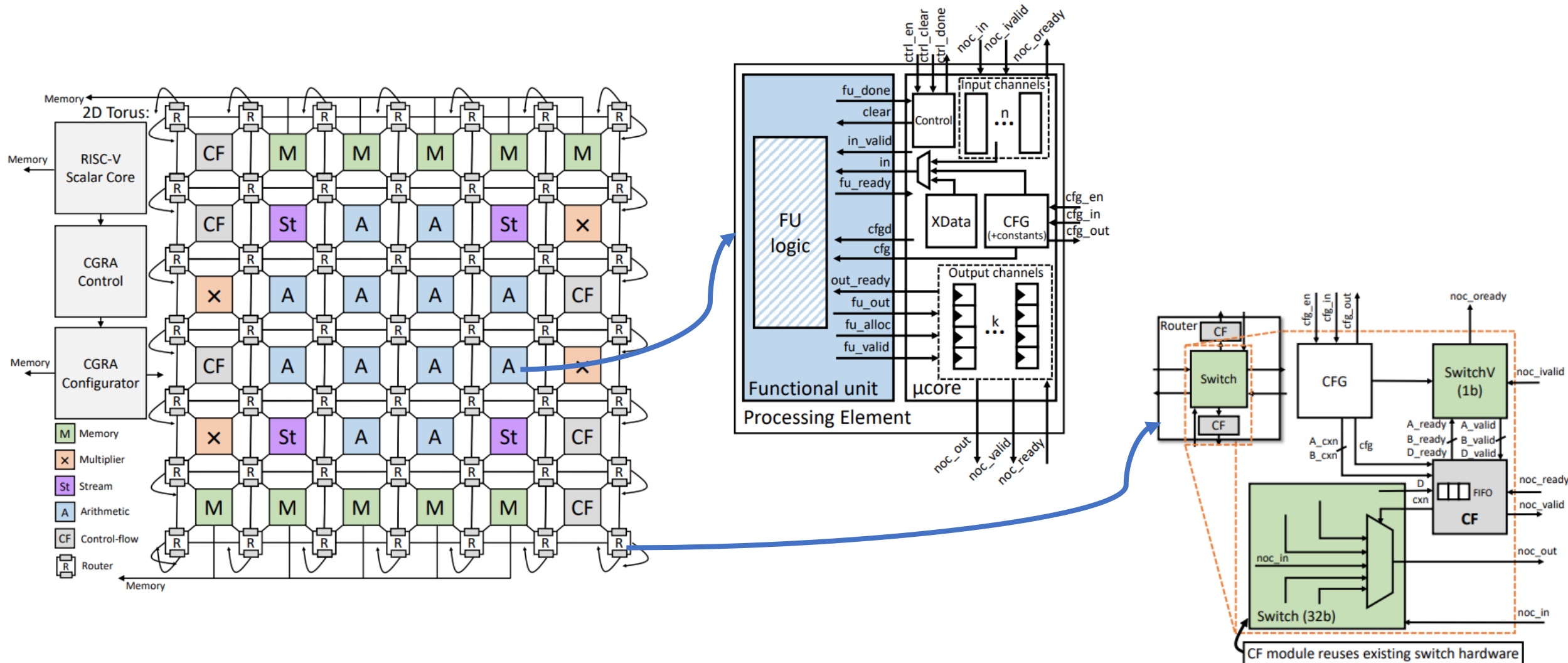


Fig. 15. Practical form of the MIT architecture.

What is the main difference in this architecture versus the "basic" architecture on the previous slide?

# Dataflow: The Riptide Ordered Dataflow Machine



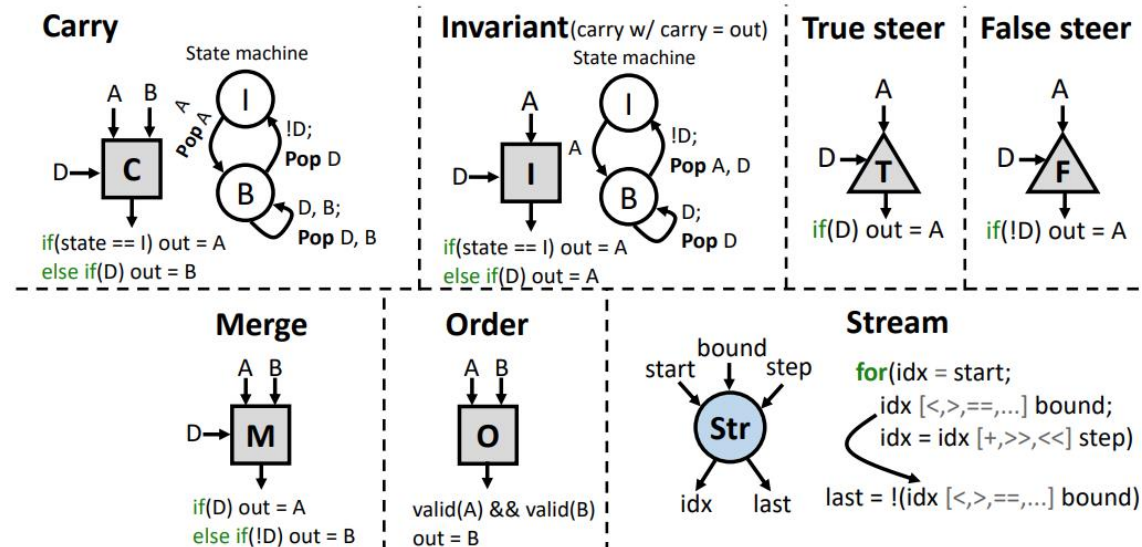
# Dataflow: The Riptide ISA

Operator(s)	Category	Symbol(s)	Semantics
Basic binary ops	Arithmetic	+, -, <<, !=, etc.	$a \text{ op } b$
Multiply, clip	Multiplier	*, clip	$a \text{ op } b$
Load	Memory	ld	$\text{ld base, idx}(, \text{dep})$
Store	Memory	st	$\text{st base, idx, val}(, \text{dep})$
Select	Control Flow	sel	$\text{cond} ? \text{val0} : \text{val1}$
Steer, carry, invariant	Control Flow	(T   F), C, I	See Fig. 3
Merge, order	Synchronization	M, O	See Fig. 3
Stream	Stream	STR	See Fig. 3

What program constructs do the **carry** and **invariant** ISA ops support?

What does the **order** ISA op do?

What program construct(s) does the **stream** ISA op support?



# Background: What is a Dataflow Machine?

A Preliminary Architecture for a Basic Data-Flow Processor\*

Jack B. Dennis and David P. Misunas  
Project MAC  
Massachusetts Institute of Technology

An example of a program in the elementary data-flow language is shown in Figure 1 and represents the following simple computation:

```
input a, b
y := (a+b)/x
x := (a*(a+b))+b
output y, x
```

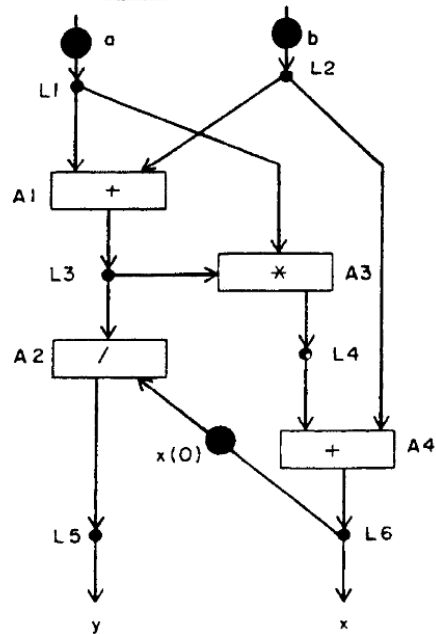


Figure 1. An elementary data-flow program.



Figure 6. Links of the basic data-flow language.

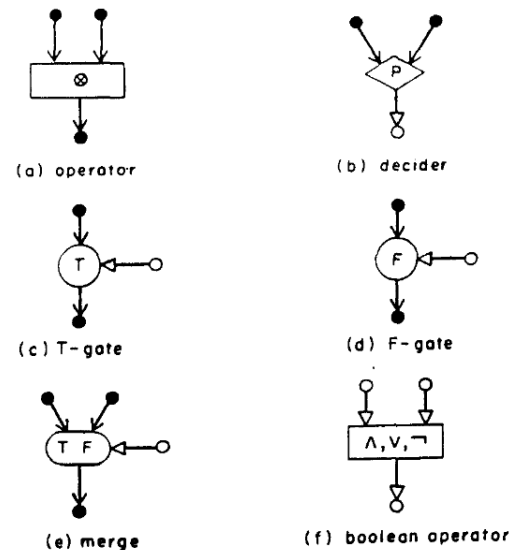


Figure 7. Actors of the basic data-flow language.

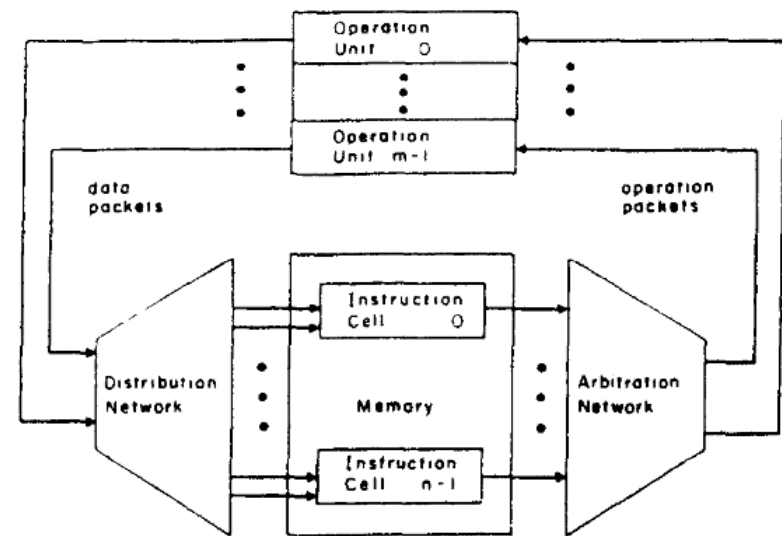
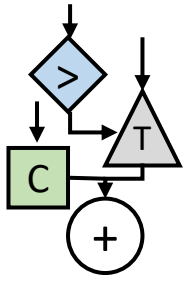


Figure 2. Organization of the elementary data-flow processor.





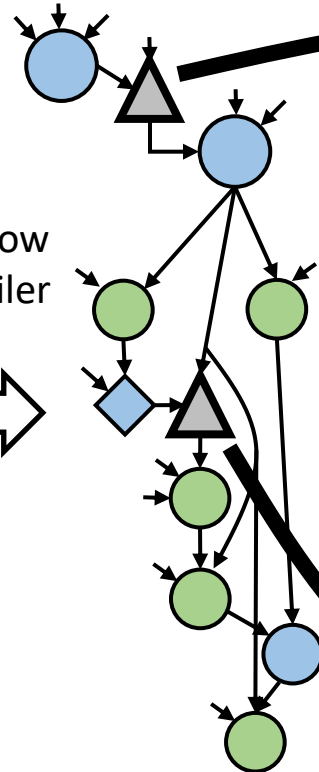
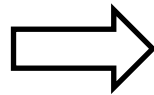
# Dataflow ISA matches CGRA Architecture

**Dataflow** compiler efficiently targets **reconfigurable dataflow** architecture!

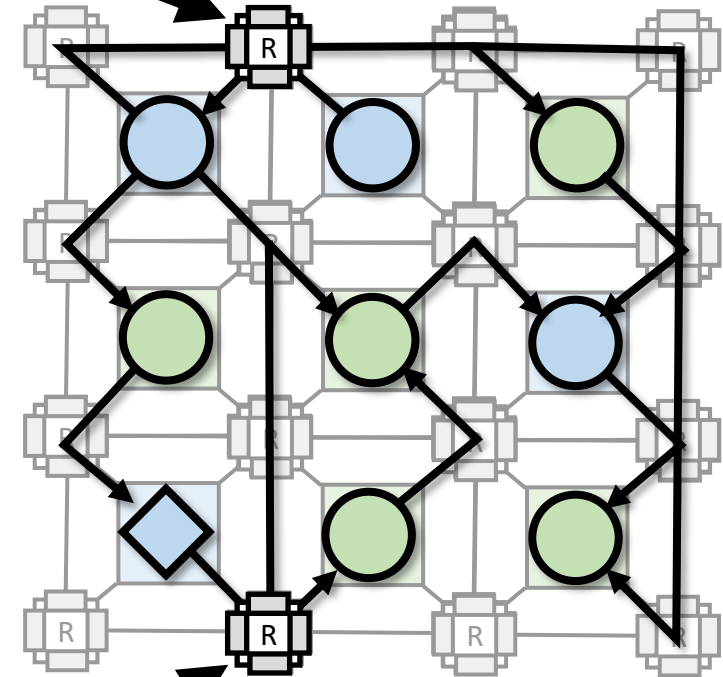
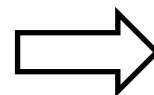
```
void test(int *a, int *b,  
         int *z, int n) {  
  for(int i = 0; i < n; i++) {  
    for(int j = 0; j < n; j++) {  
      if(a[j] < 0) {  
        z[j] = 0;  
      }  
      z[j] += b[j];  
    }  
  }  
}
```

C code

Dataflow  
Compiler



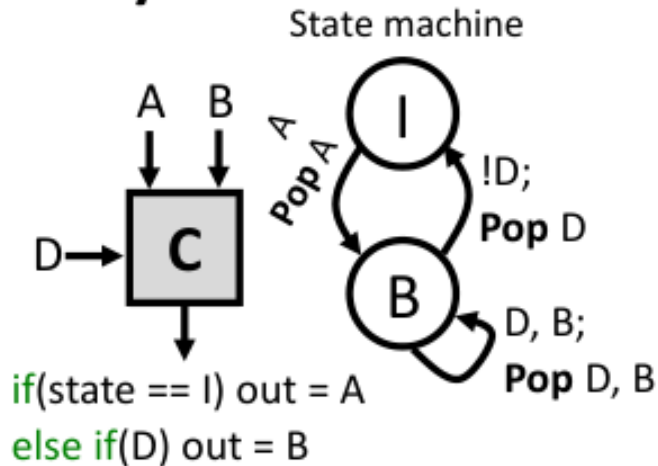
Optimized  
Dataflow Graph



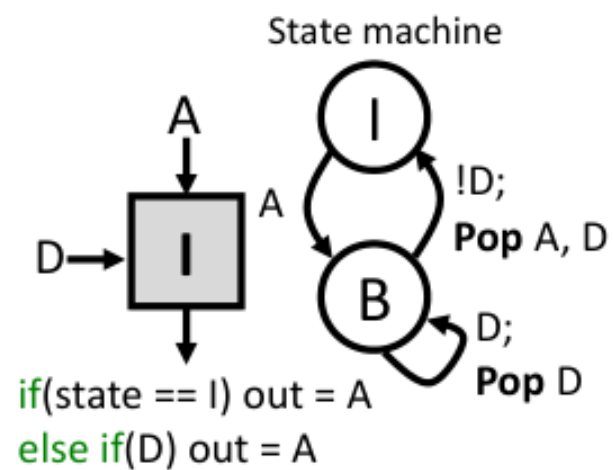
RipTide CGRA

# Intermediate representation for dataflow compilation

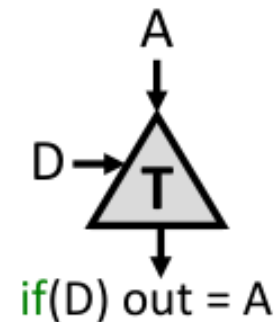
## Carry



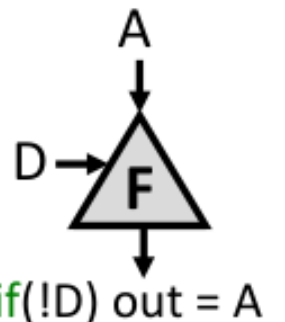
## Invariant (carry w/ carry = out)



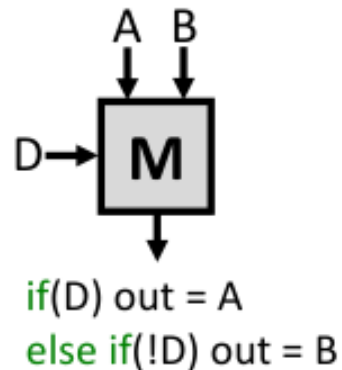
## True steer



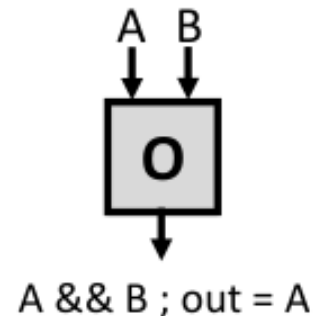
## False steer



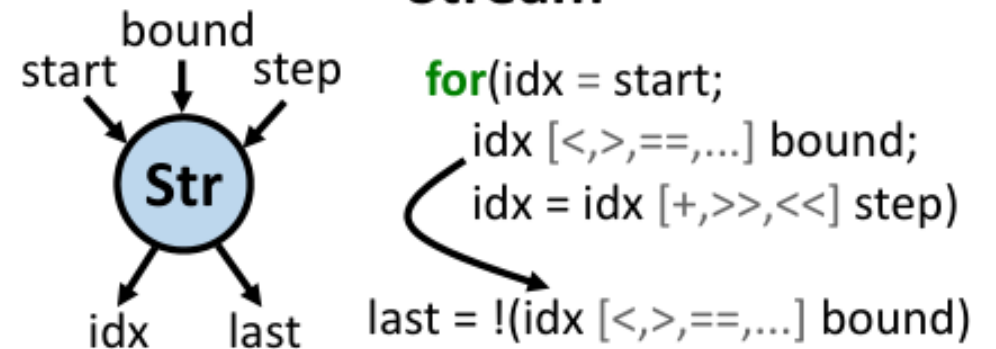
## Merge



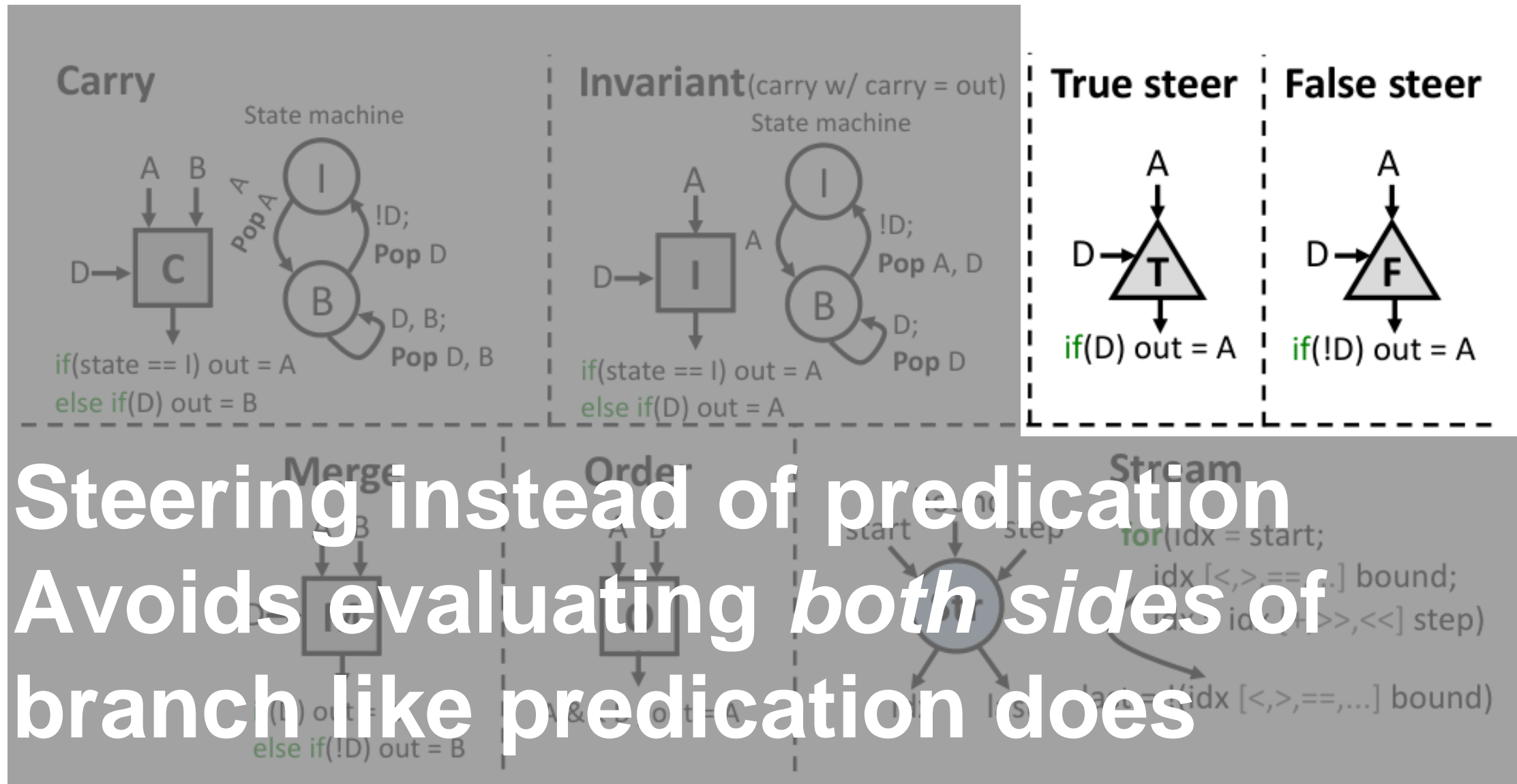
## Order



## Stream

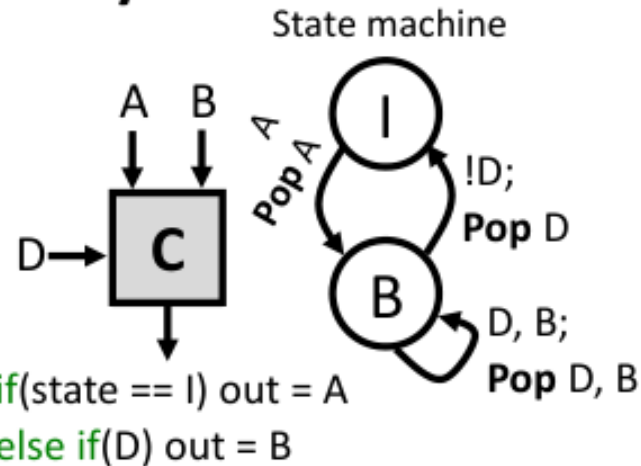


# Steering Sends Values Only Where They Are Needed

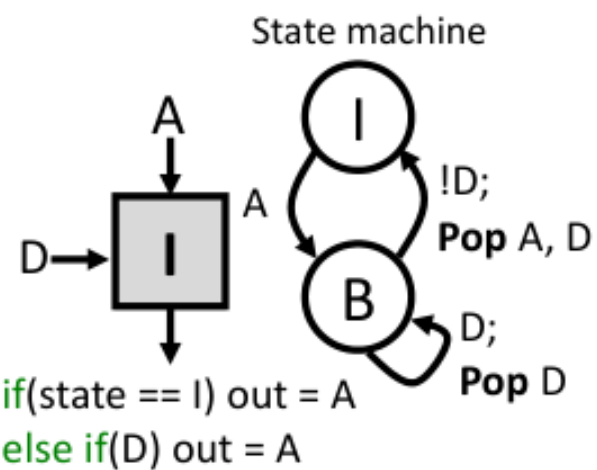


# Control-flow Operators Handle Loop-Carried Dependences

## Carry



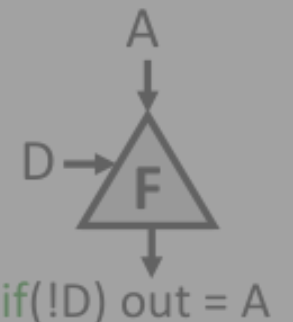
## Invariant (carry w/ carry = out)



## True steer

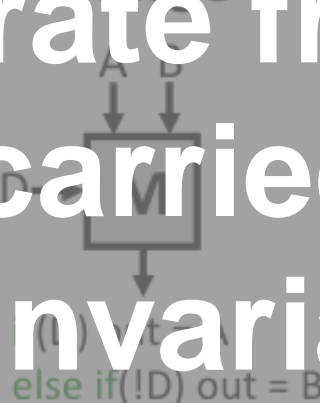


## False steer



Generate fresh value tokens for  
loop carried dependences and  
loop-invariant values

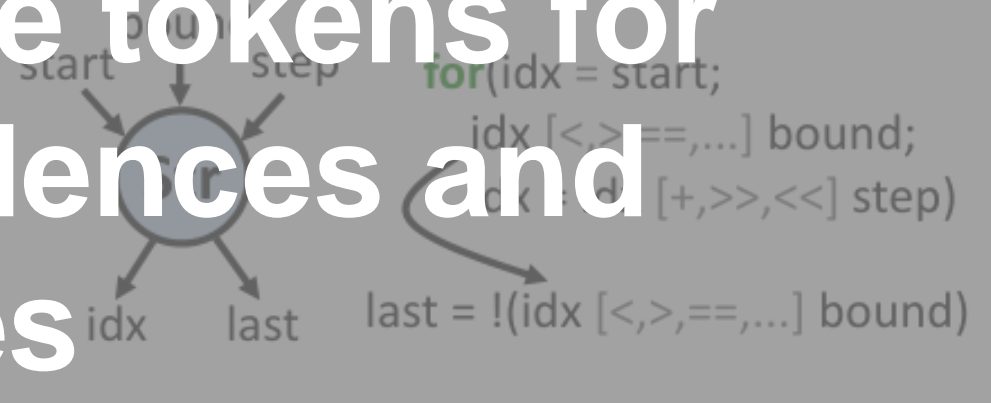
## Merge



## Order



## Stream

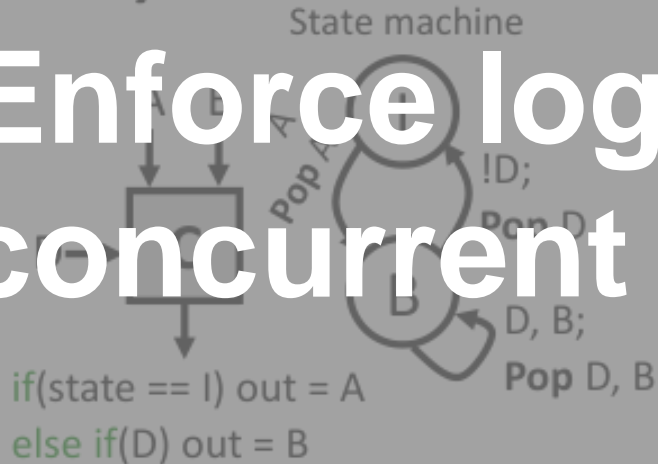




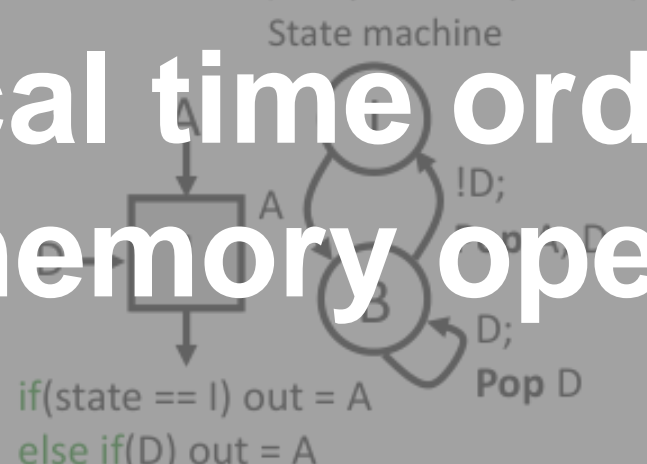
# Memory Ordering Gates Maintain Memory Consistency

Enforce logical time ordering of concurrent memory operations

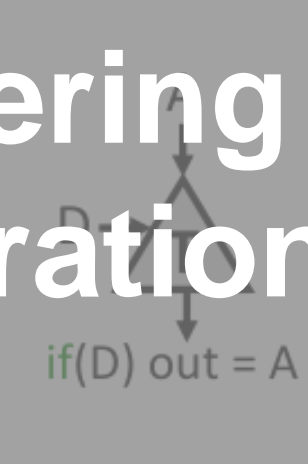
Carry



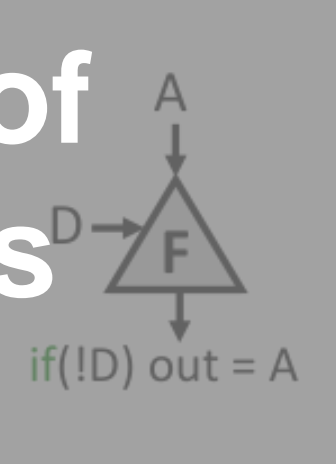
Invariant (carry w/ carry = out)



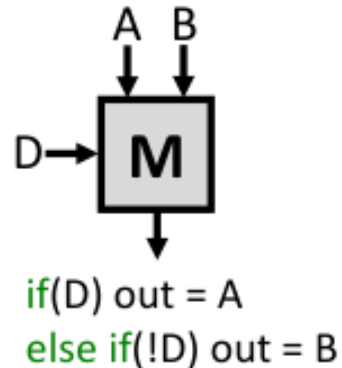
True steer



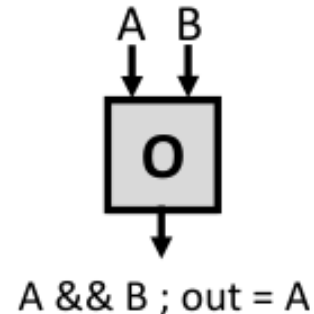
False steer



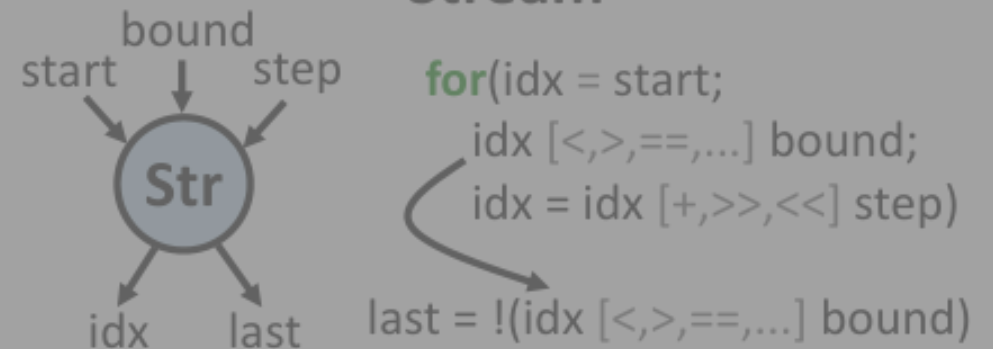
Merge



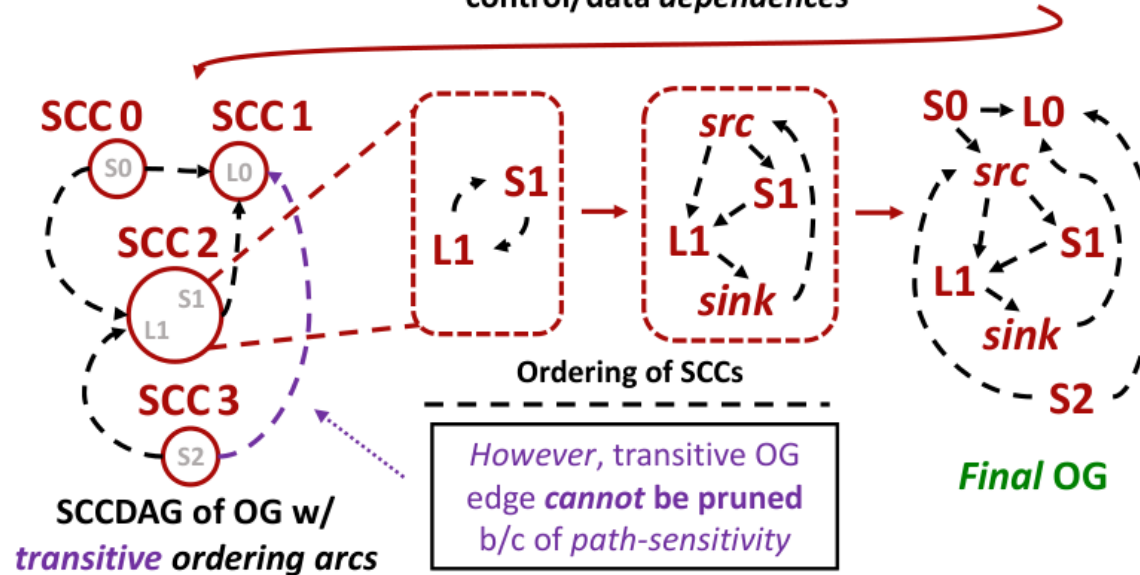
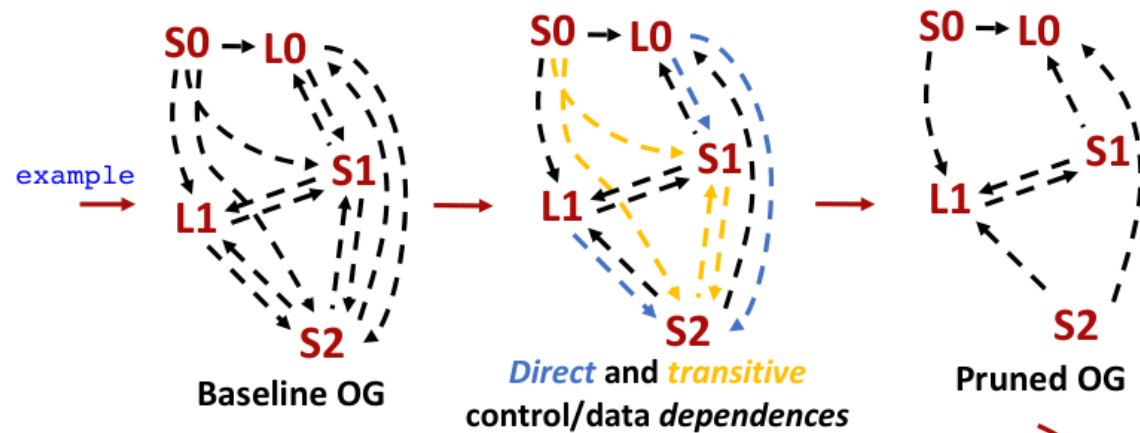
Order



Stream



# Memory Ordering Reduction Analysis

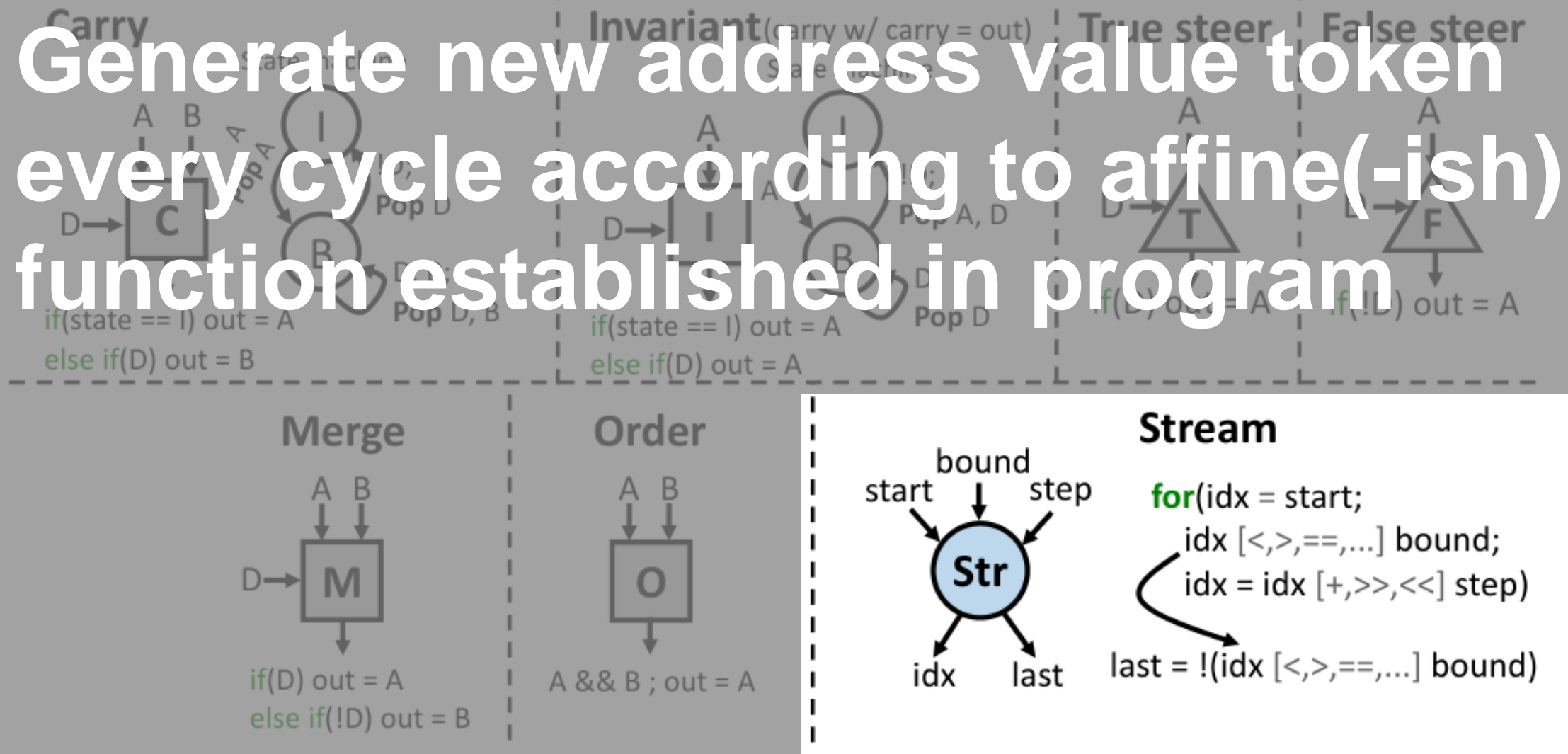


Existing memory ordering analyses rely on *transitive reduction* of ordering graph

Dataflow ordering reduction requires *path sensitivity* or cuts required orders.

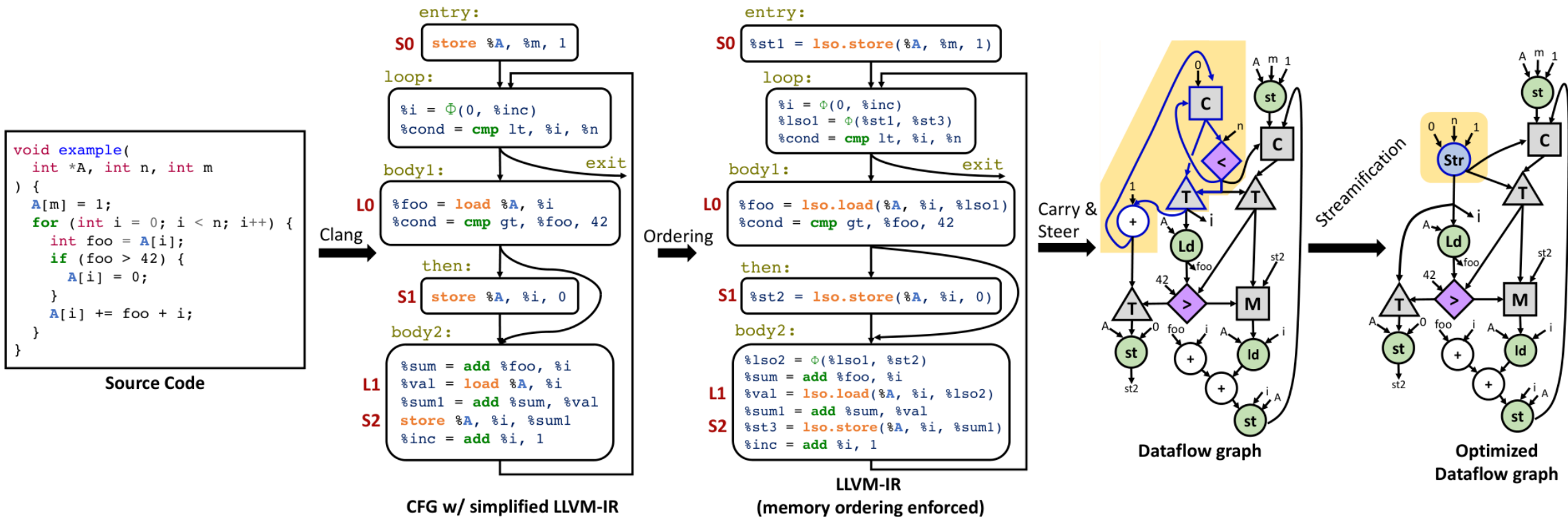
# Stream Gates Optimize Patterned Address Computation

Generate new address value token every cycle according to affine(-ish) function established in program





# End-to-end compilation flow

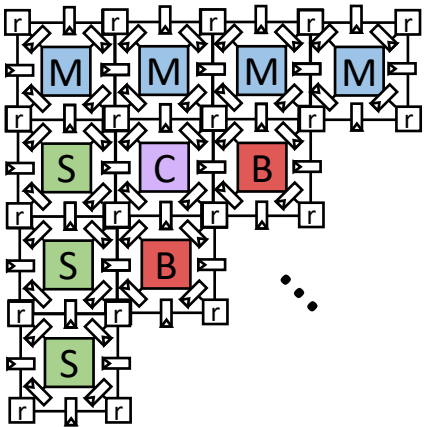




# Energy-Minimal Network-on-Chip Tricks: No Buffers & Control-Flow in NoC

## ✓ Multi-hop, bufferless NoC

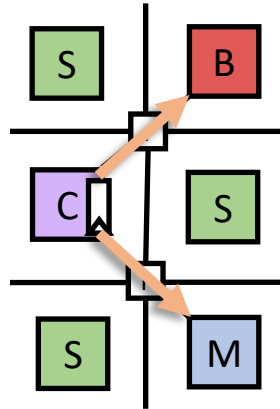
Buffer on every link



Duplicates data in multiple buffers

Buffers @ producer

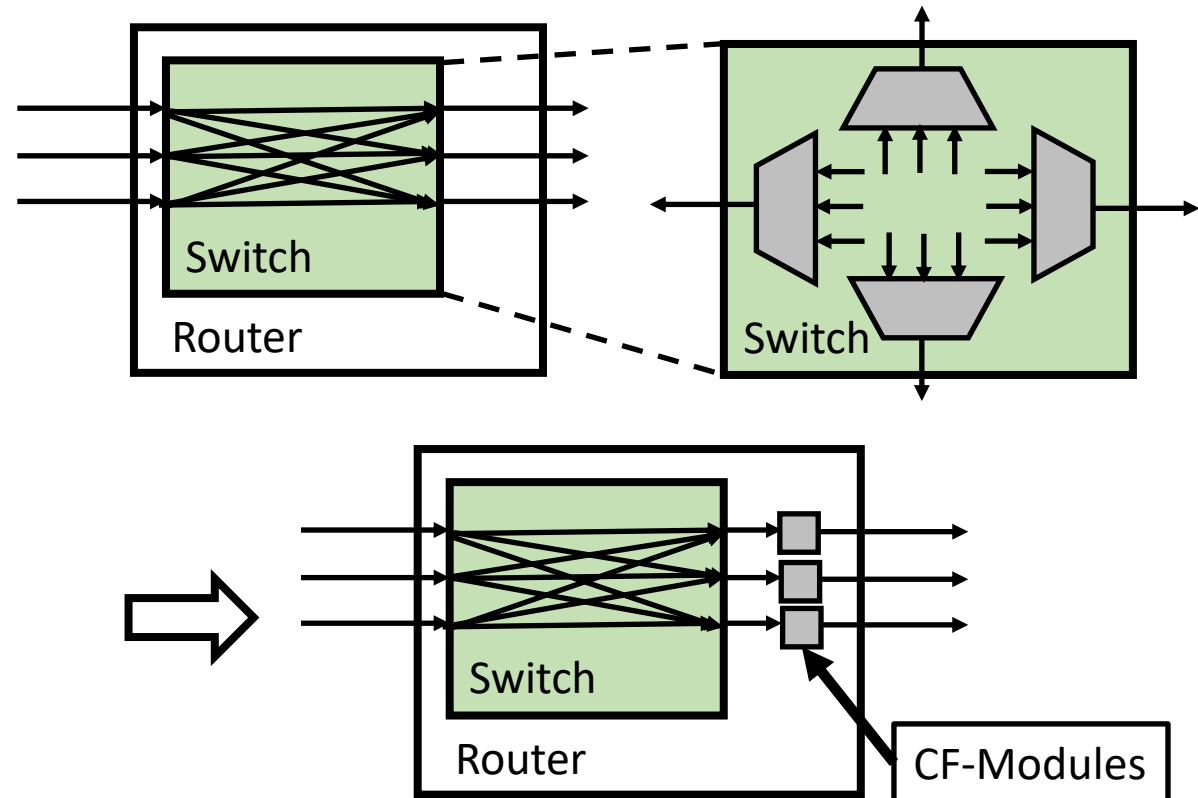
v.



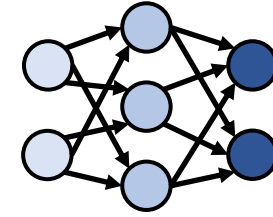
Allows broadcast to multiple consumers w/out duplication

## ✓ Control-flow in the NoC

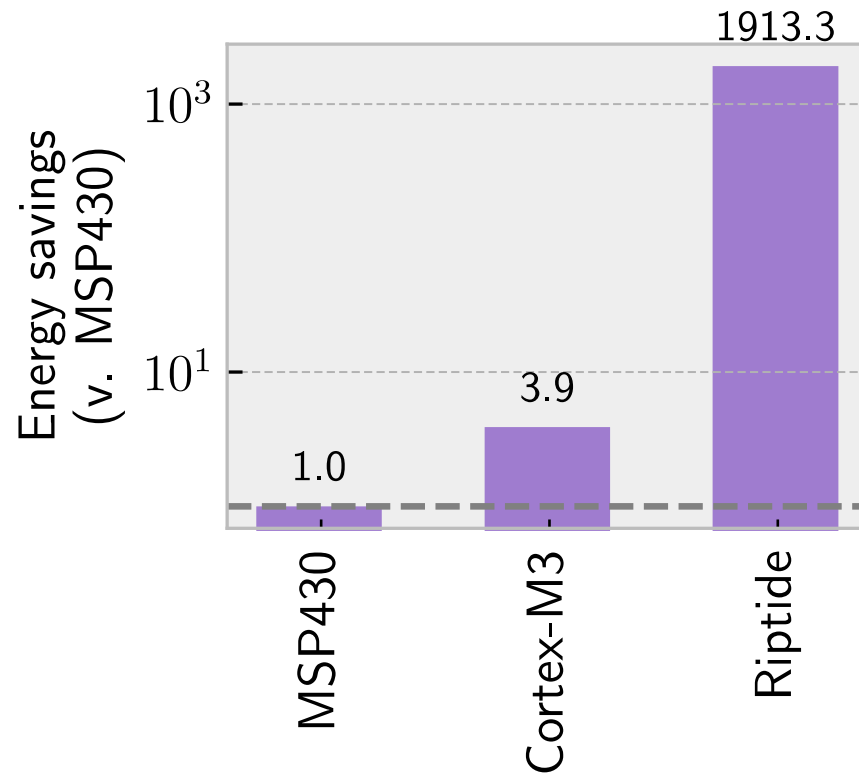
- Control-flow operations are numerous, but simple  
→ Wasteful to assign to PEs



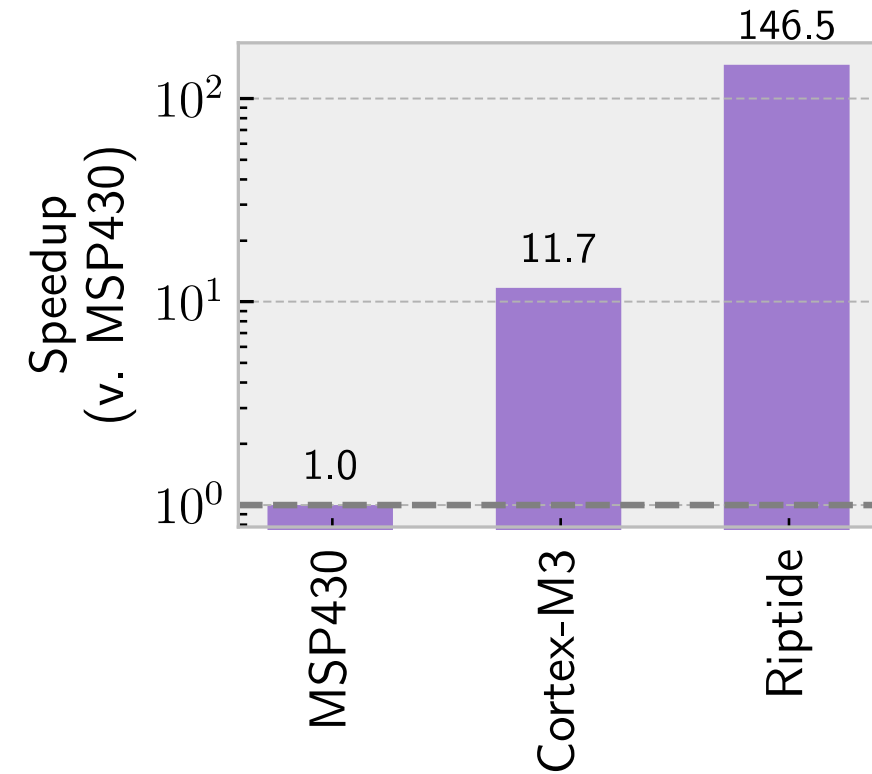
# Riptide vs. COTS Extreme Edge MCUs



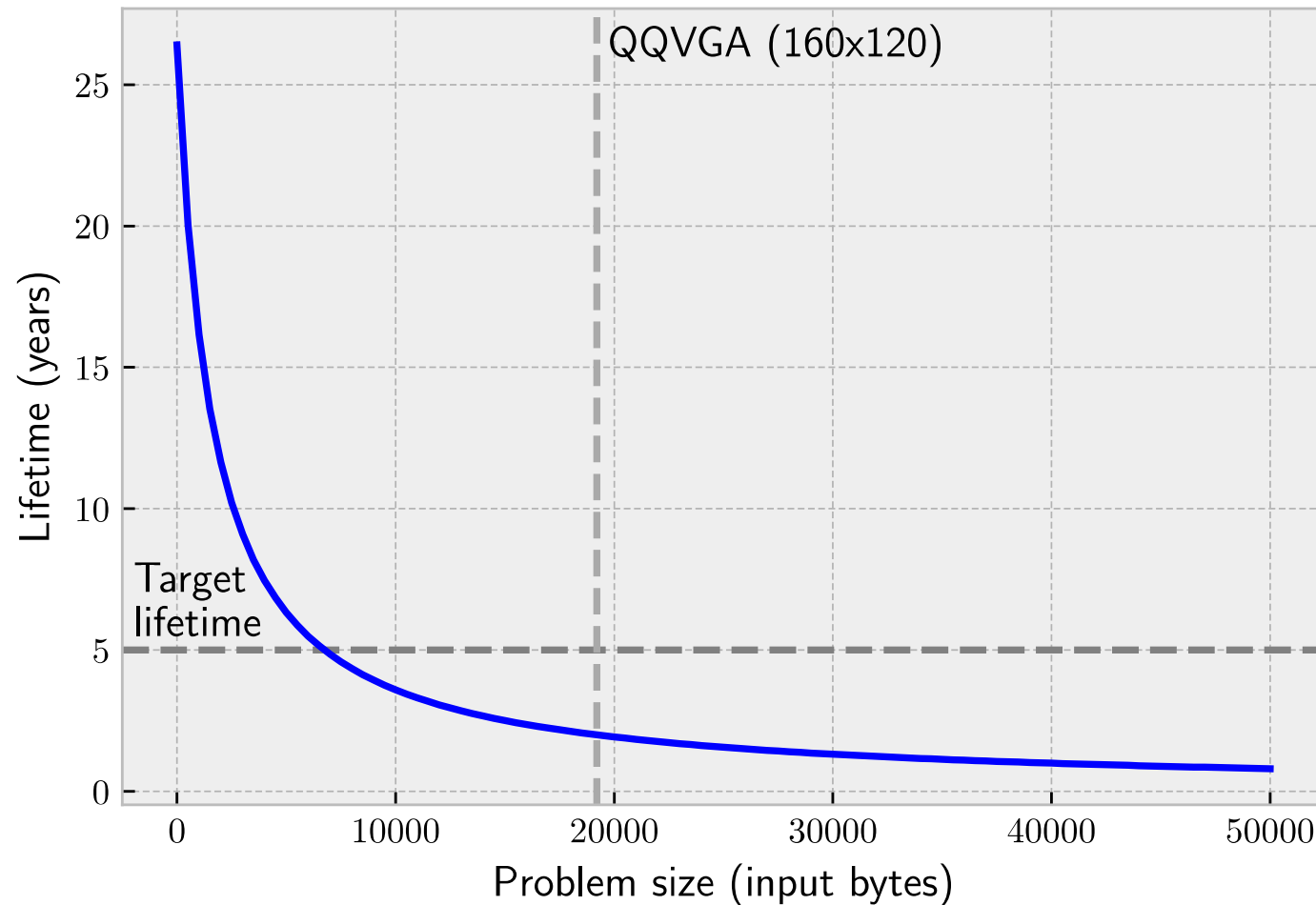
Energy savings



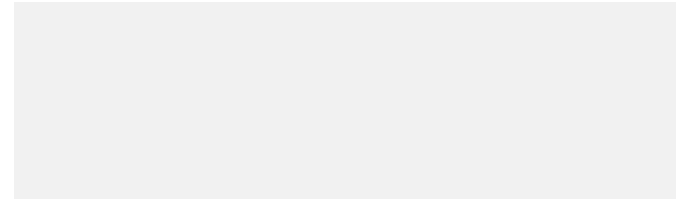
Speedup



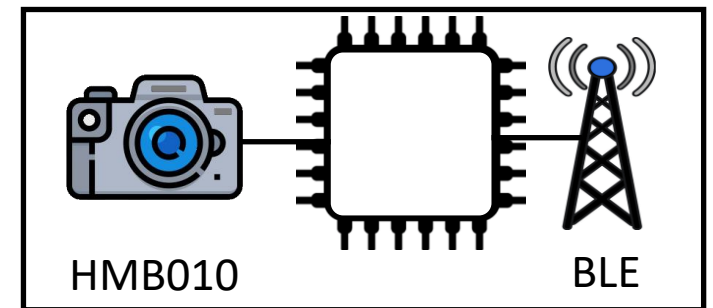
# Evaluating compute options for the extreme edge



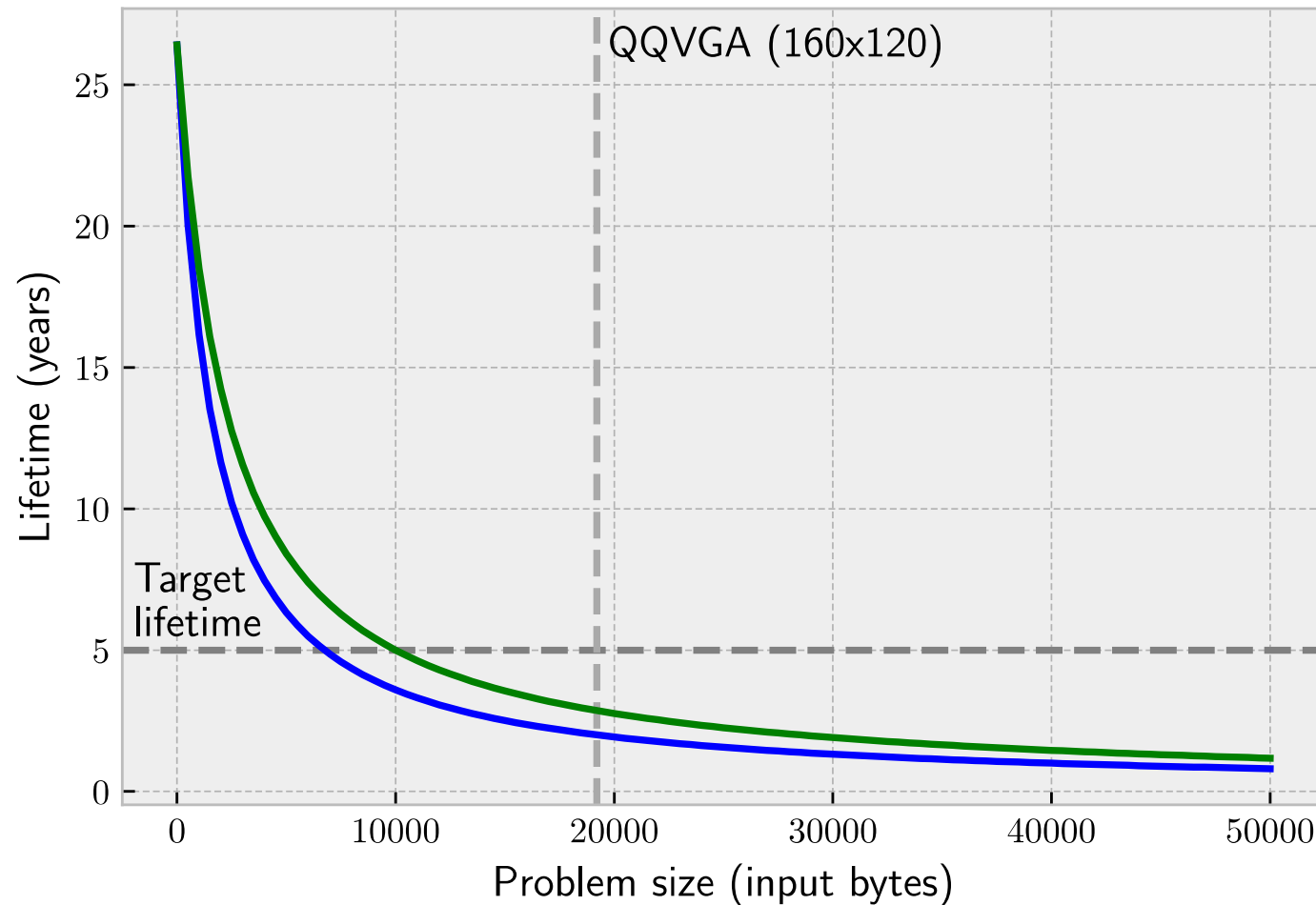
— Transmit always



System overview:

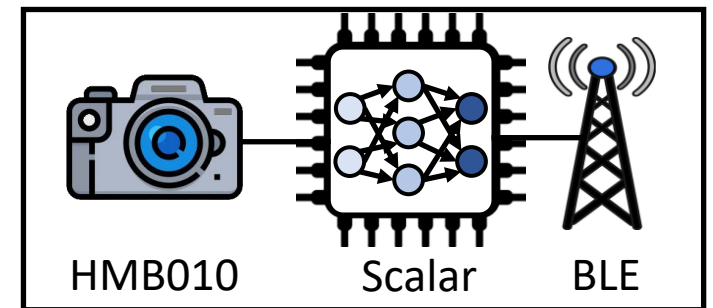


# Evaluating compute options for the extreme edge

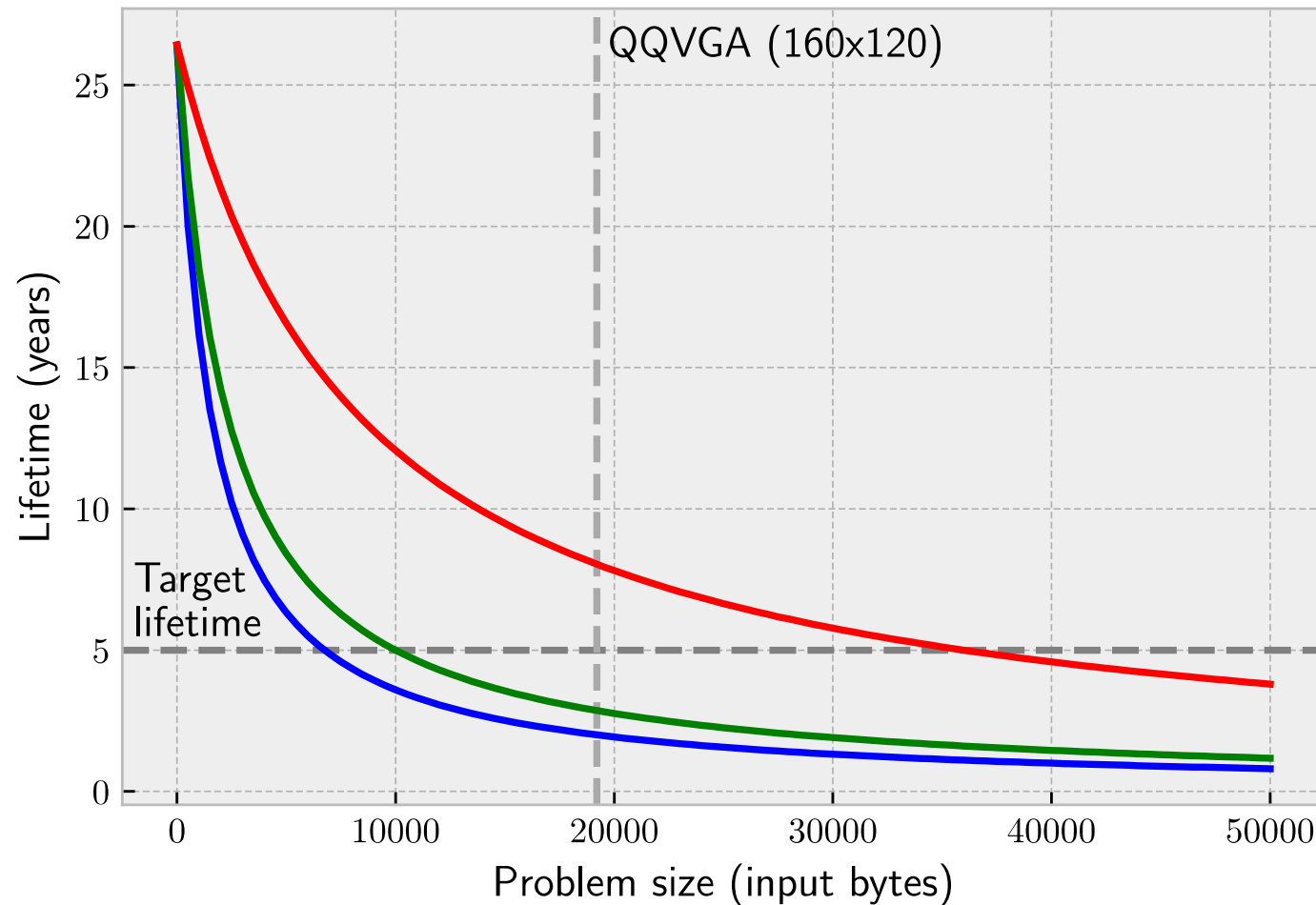


— Transmit always  
— Scalar

System overview:

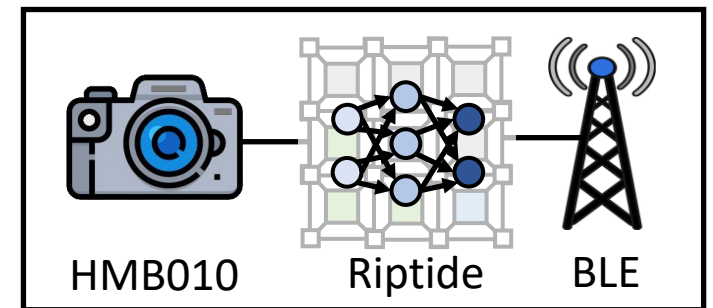


# Evaluating compute options for the extreme edge

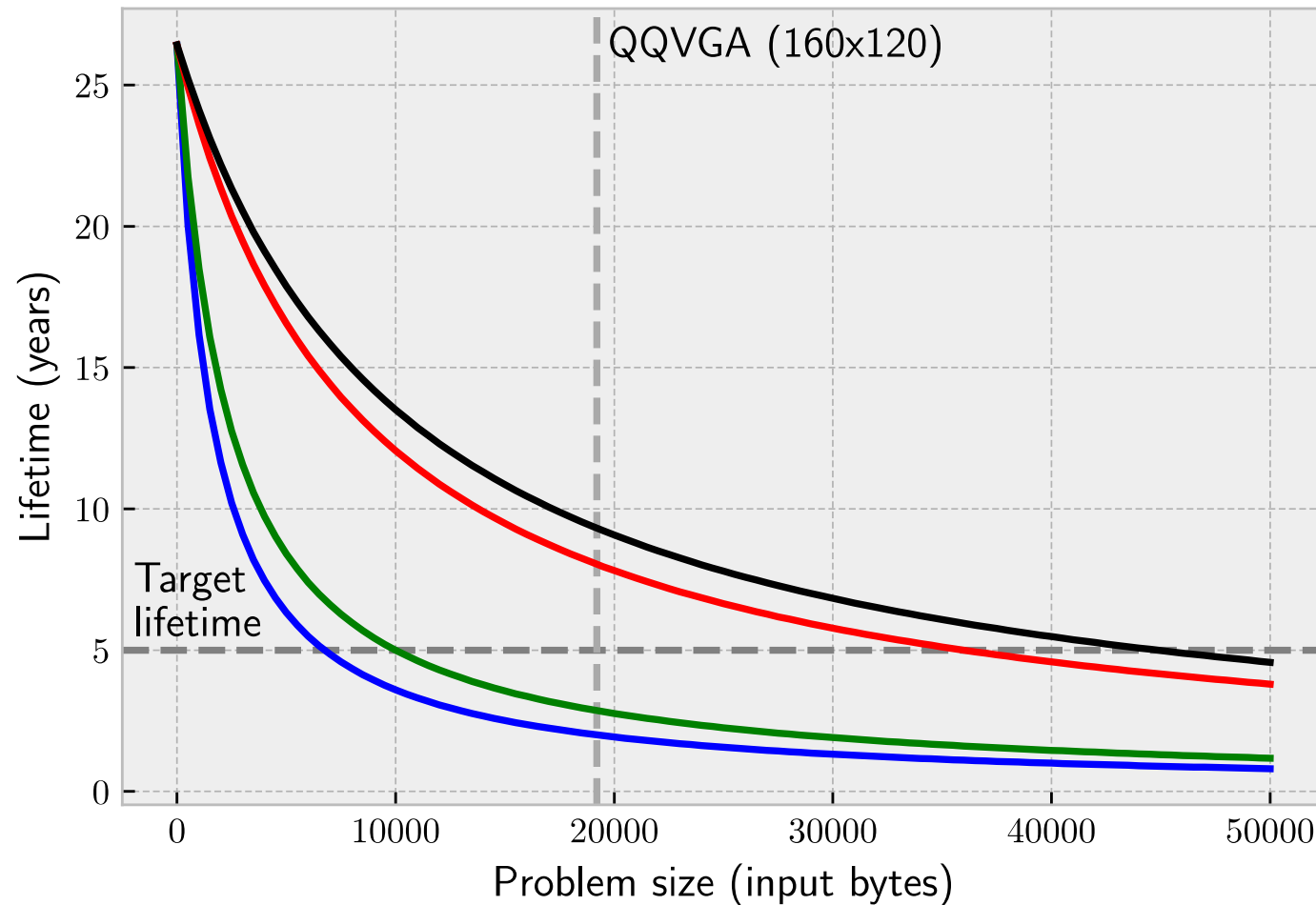


- Transmit always
- Scalar
- RipTide

System overview:

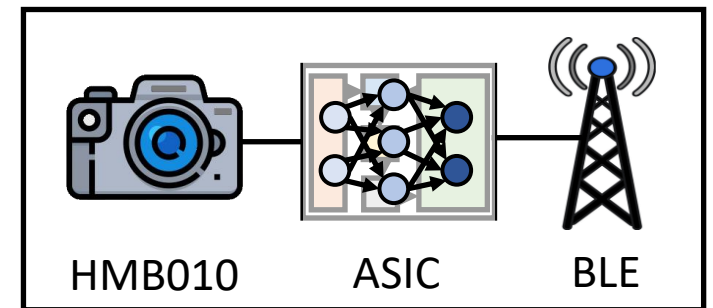


# Evaluating compute options for the extreme edge

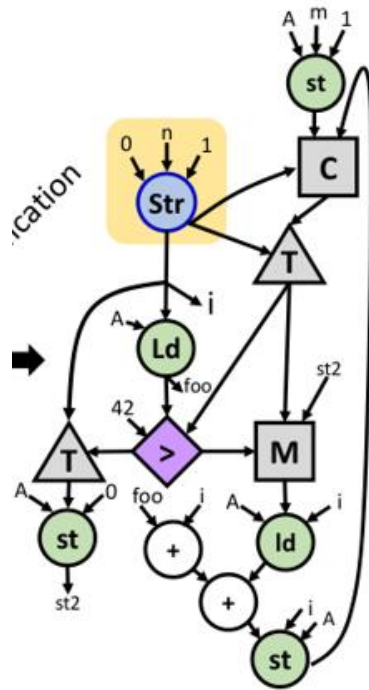


- Transmit always
- Scalar
- RipTide
- Hypothetical (10TOPS/W)

System overview:



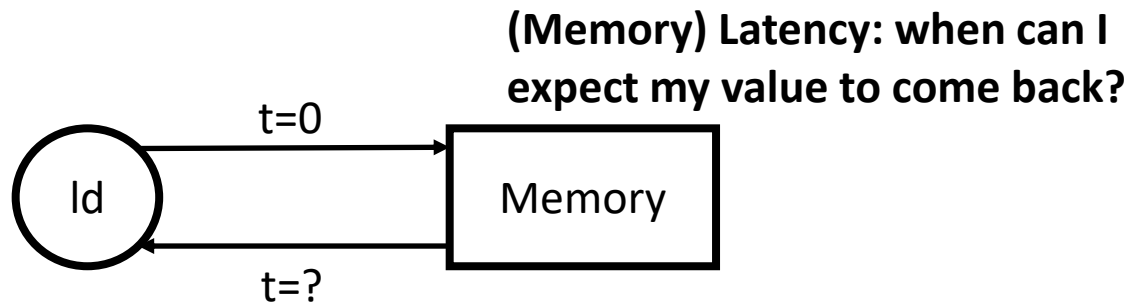
# That was “Ordered Dataflow”



Axiom: Tokens proceed through the graph in the order of their generation

How do we ensure that tokens flow through the dataflow graph in order?

# What about allowing token reordering?

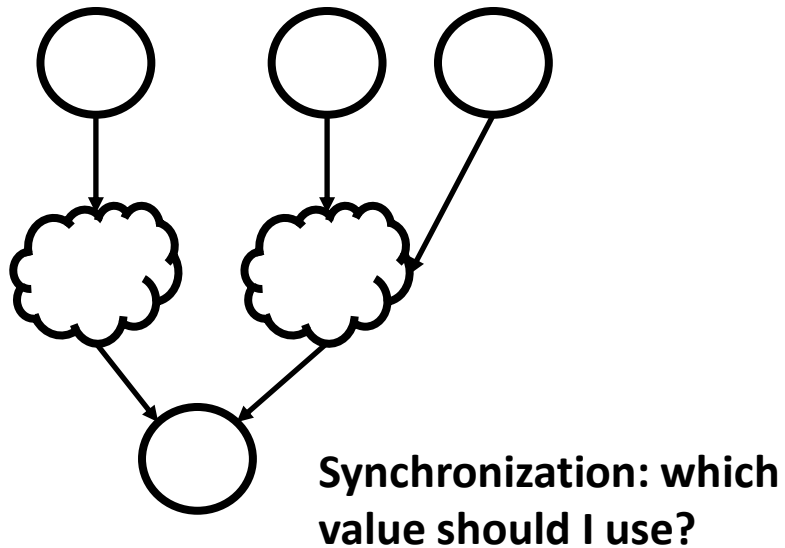


“Tagged-token dataflow architectures”

Two issues: Latency & Synchronization

Latency: time between when operation is issued and when completes

Synchronization: need to assure data properly written before read





# Monsoon: an Explicit Token-Store Architecture

Gregory M. Papadopoulos  
Laboratory for Computer Science  
Massachusetts Institute of Technology

David E. Culler  
Computer Science Division  
University of California, Berkeley

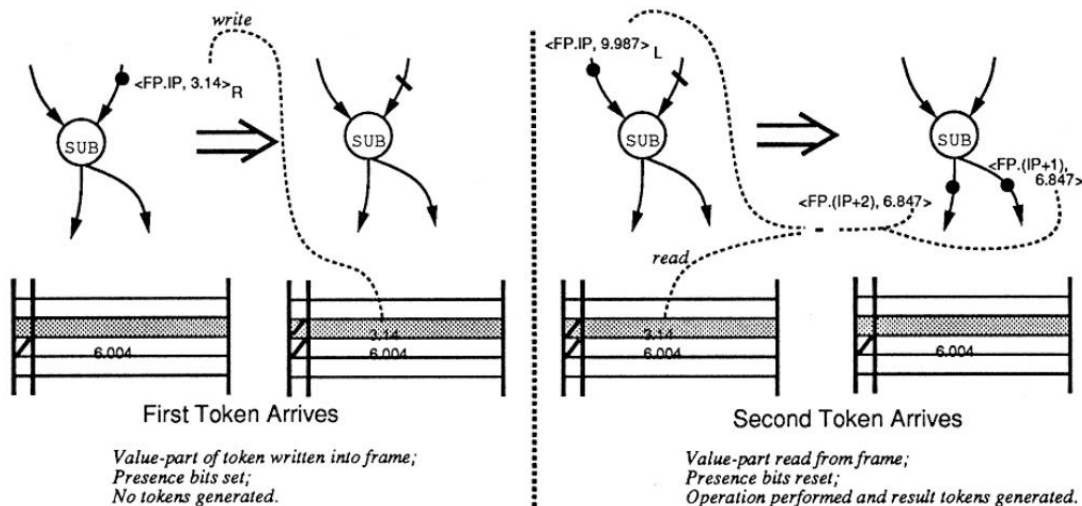


Figure 1: ETS Representation of an Executing Dataflow Program

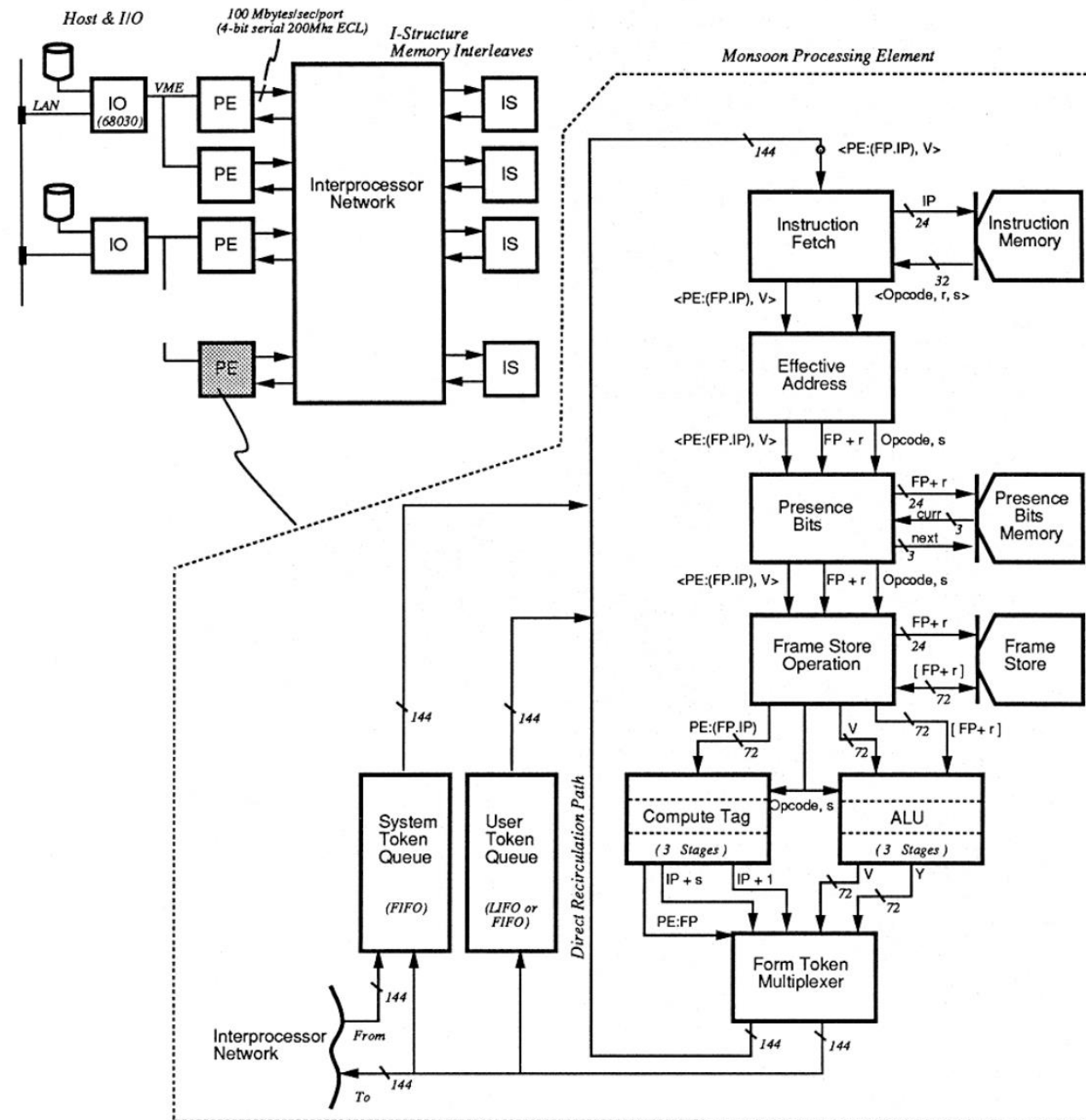


Figure 2: Monsoon Processing Element Pipeline

# Tagged-token Dataflow Architecture

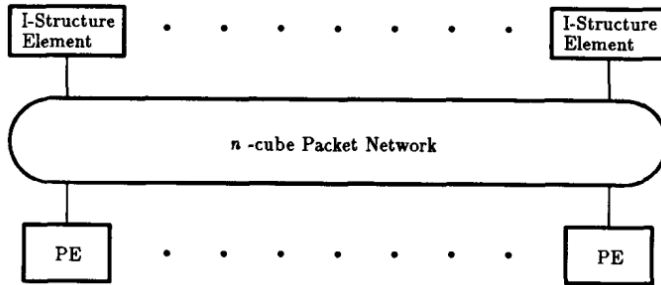


Fig. 17. Top-level view of TTDA.

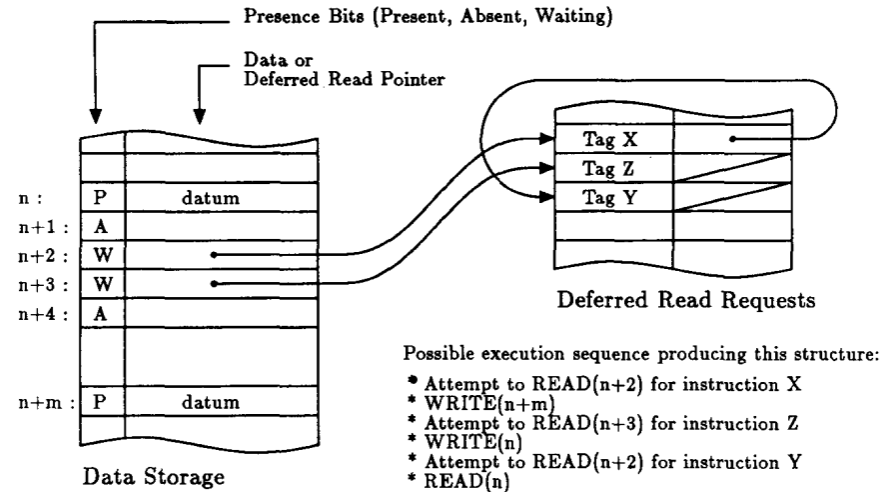


Fig. 7. I-structure memory.

**I-structures:** latency-tolerant memory  
**I-fetch** send rd tok w/ addr+continuation  
***P:** read & run; A/W: queue*  
***I-store:** send wr tok to populate table*  
***A/W are non-blocking (why?)***  
***I-allocate:** make storage for fetch/stores*

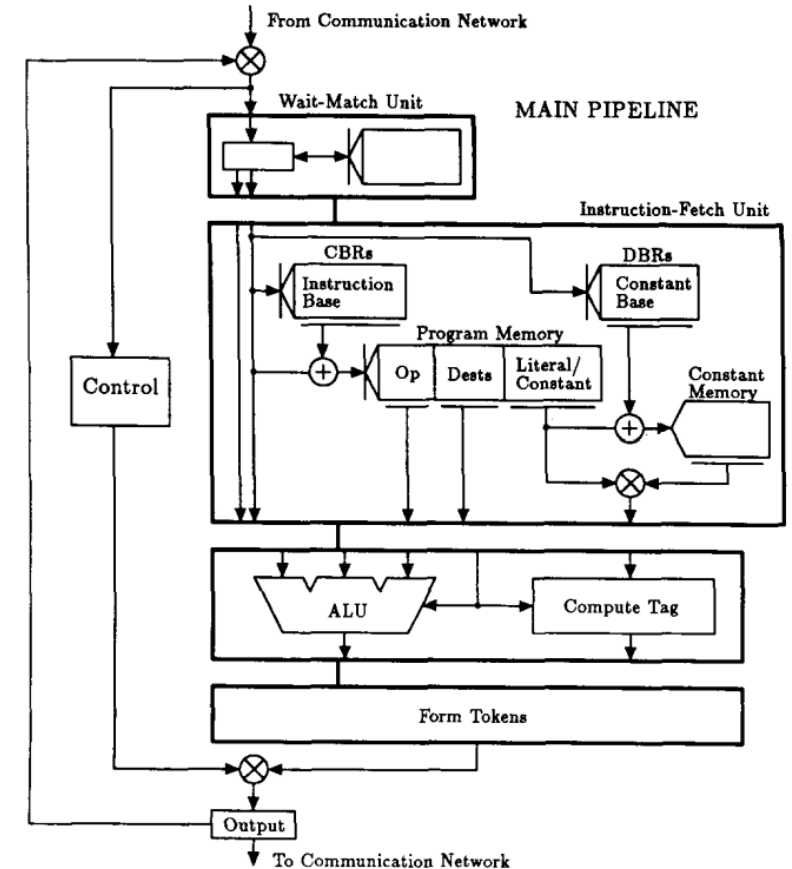
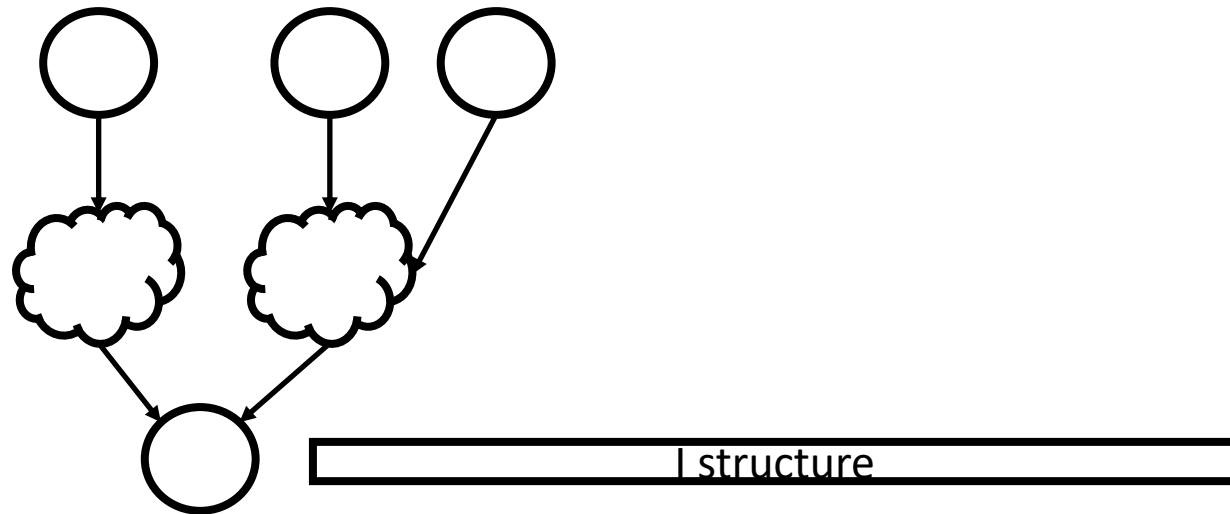


Fig. 18. A processing element.

**Token matching:** synchronization of out-of-order inputs, using i-structs

# Parallelism is a resource congestion problem



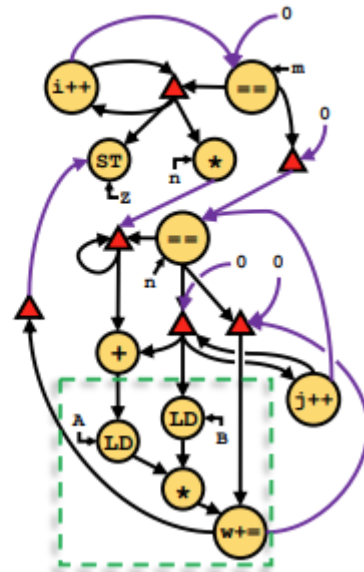
Synchronization: which value should I use? Many options accumulating over time

# Varieties of Dataflow Execution

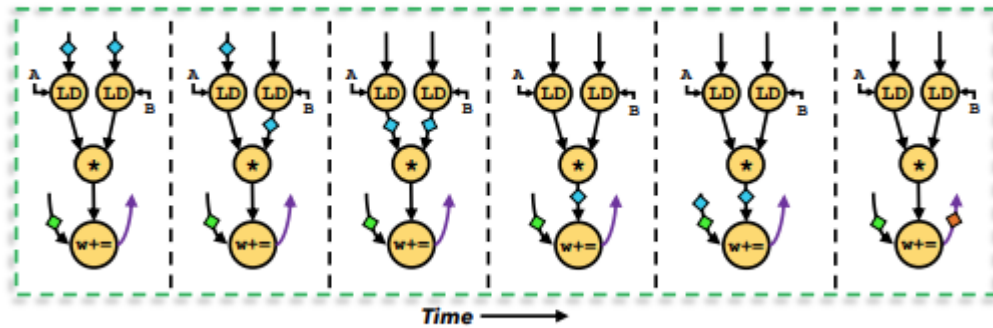
**Figure 3:** A running example: dense matrix-vector multiplication. We zoom in on the innermost loop body (green box) to demonstrate dataflow execution.

```
def dmv (A, B, Z, m, n):  
  for i = 0..m:  
    w = 0  
    for j = 0..n:  
      w += A[i,j] * B[j]  
    Z[i] = w
```

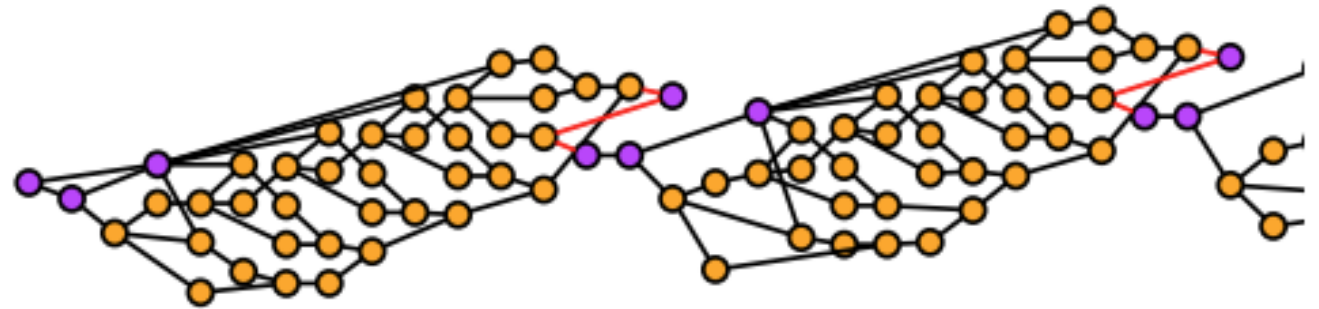
(a) Pseudocode.



(b) Dataflow graph (DFG).



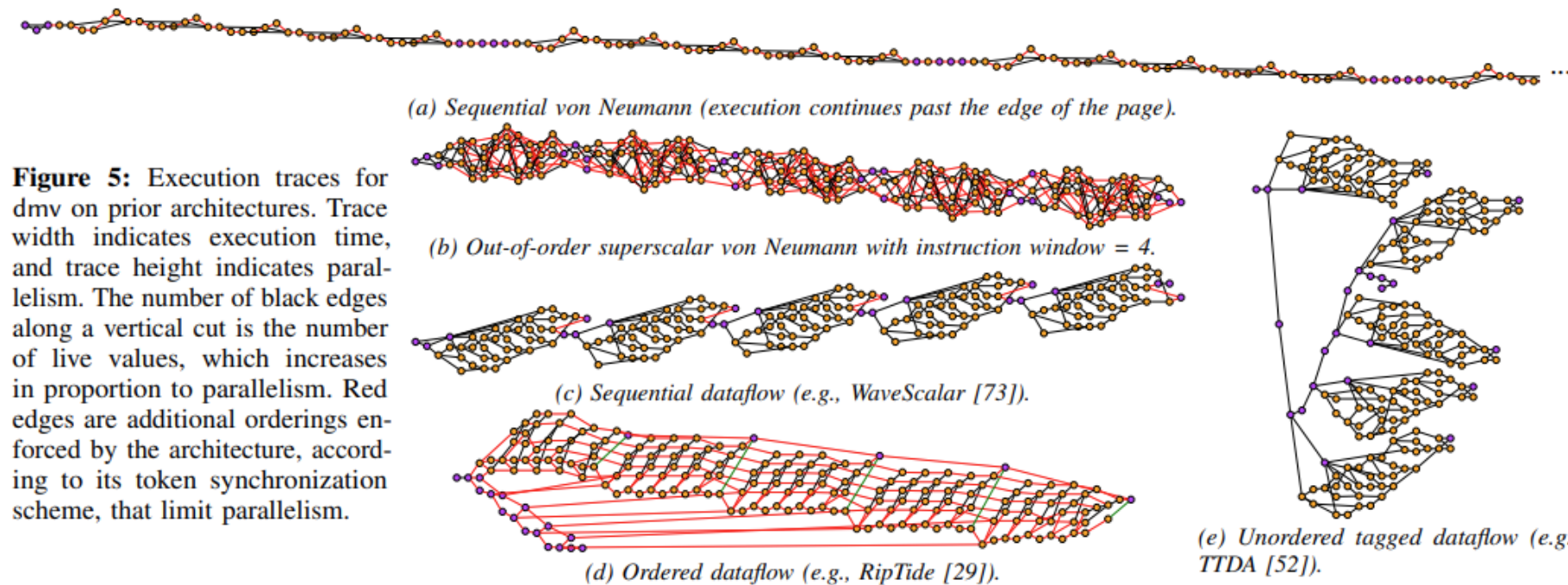
(c) Execution trace of innermost loop.



**Figure 4:** Partial execution trace of dmv.



# Varieties of Dataflow Execution



**Figure 5:** Execution traces for dmV on prior architectures. Trace width indicates execution time, and trace height indicates parallelism. The number of black edges along a vertical cut is the number of live values, which increases in proportion to parallelism. Red edges are additional orderings enforced by the architecture, according to its token synchronization scheme, that limit parallelism.

sense, vN's sequential ordering restricts execution to a "depth-first" traversal of a program's full dynamic execution graph (Fig. 1). The result is minimal parallelism, so that program execution takes a long time (the graph is wide) but live state is minimized (the graph is short).

Parallelism can be increased by having multiple vN execution streams, i.e., multithreading. Token synchronization now includes a token's thread id, in addition to its vN ordering. Multithreading punts all of the problems of parallelism.