

# **18-643 Lecture 6: Good-for-HW Computation Models**

James C. Hoe

Department of ECE

Carnegie Mellon University

# Housekeeping

- Your goal today: see the temporal and spatial patterns of compute and data access in classic *good-for-HW* compute models
- Notices
  - Handout #4: lab 1, **due noon, 9/26**
  - Project status report due each Friday
- Readings (see lecture schedule online)
  - Wikipedia is a good starting point
  - for a textbook treatment see Ch 5 (+ Ch 8, 9, 10) of *Reconfigurable Computing* by Hauck and Dehon
  - for lab2, C. Zhang, et al., ISFPGA, 2015.



# Structural RTL: Low Level/Full Detailed

- Designer in charge
  - arbitrary control and datapath schemes
  - precise control—when, what, where—at the bit and cycle granularity

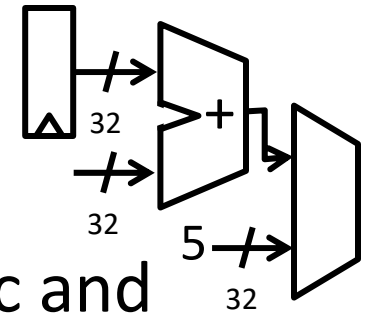
*With great power comes great responsibility . . .*

- RTL synthesis is literal (except comb. logic)
  - little room for timing and structural optimizations
  - faithful to both “necessary” and “artifacts”

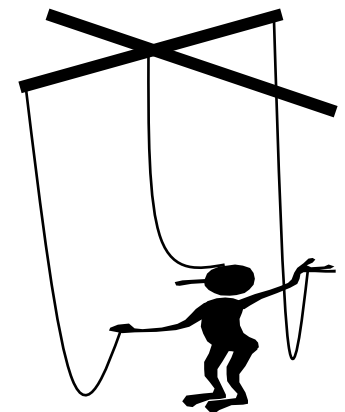
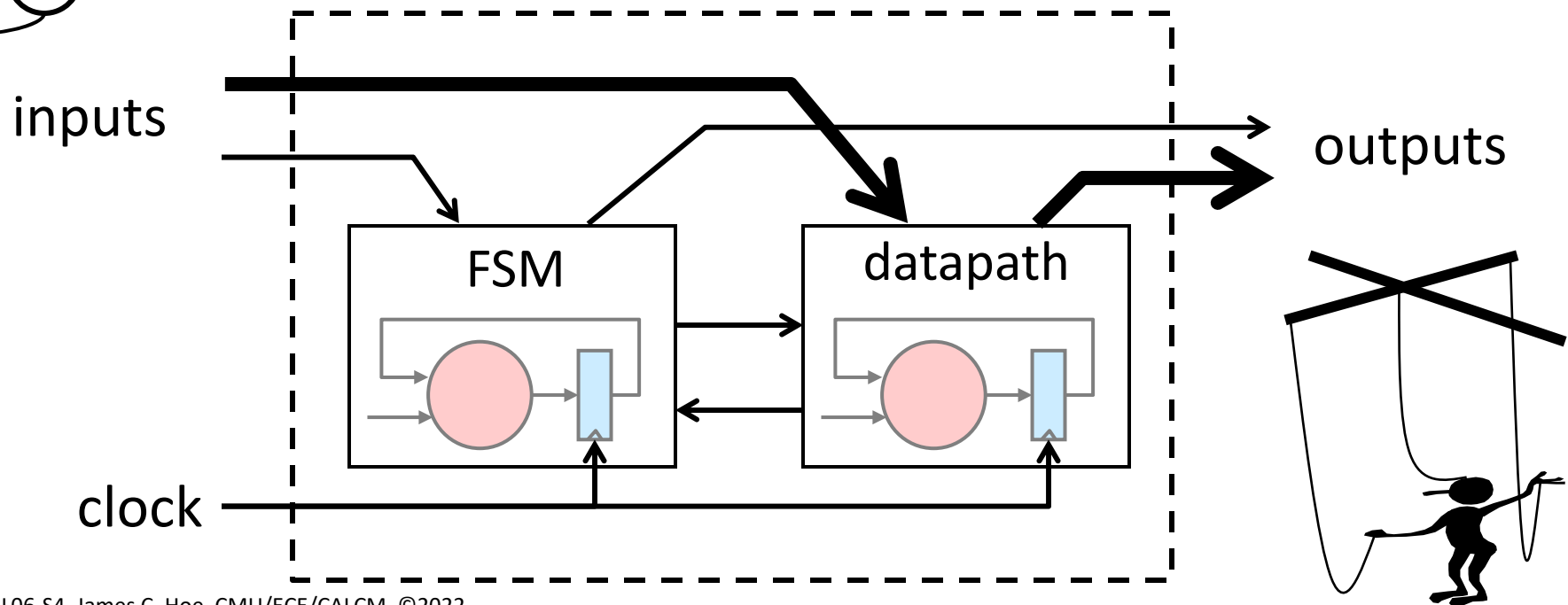
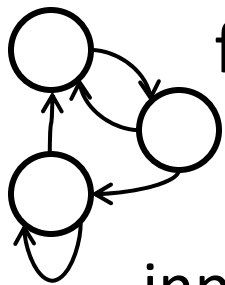
e.g., a and b mutually exclusive?

```
always@ (posedge c)
  if (a)
    o<=1;
  else if (b)
    o<=2;
```

# FSM-D “Design Pattern”



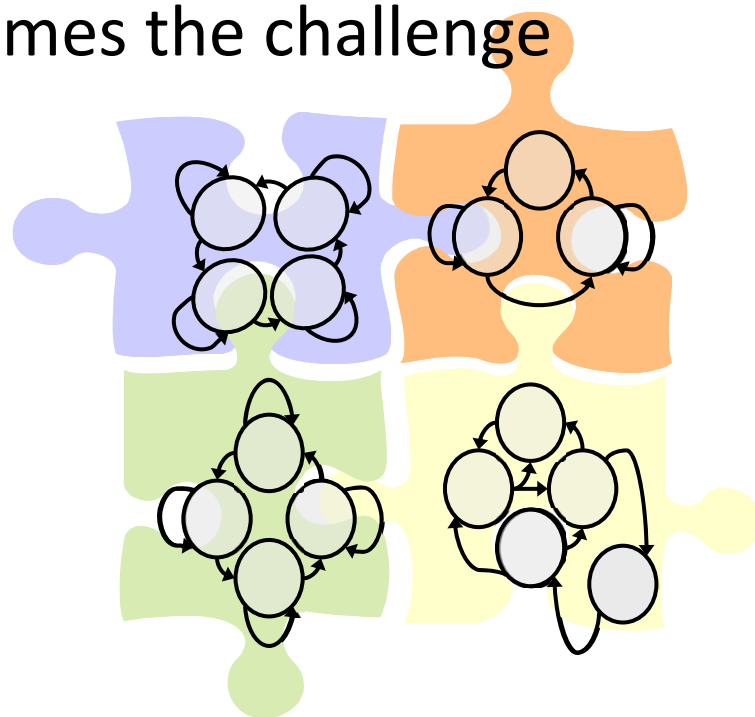
- datapath = “organized” combinational logic and registers to carry out computation (puppet)
- FSM = “stylized” combinational logic and registers for control and sequencing (puppeteer)





# Cooperating FSM-Ds

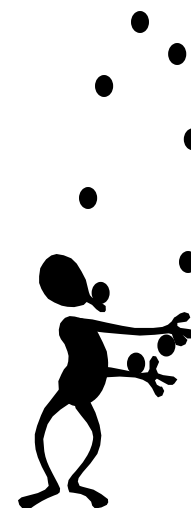
- Partitioning large design into manageable chunks
  - natural decomposition by functionalities
  - inherent concurrency and replications
- Correct decomposition leads to simpler parts but coordination of the parts becomes the challenge
  - synchronization: having two FSM-Ds in the right state at the right time
  - communication: exchange information between FSM-D (requires synchronization)





# Crux of RTL Design Difficulty

- We design concurrent FSM-Ds separately
  - liable to forget what one machine does when focused on another
- No language support for coordination
  - no explicit way to say how state transitions of two FSMs (i.e., control) must be related
- Coordination hardcoded into design implicitly
  - leave little room for automatic optimization
  - hard to localize design changes
  - (unless decoupled using request/reply-style handshakes)



**Lacks standard interfacing of SoC IP composition**



# What is High-Level?

- Abstract away detail/control from designer
  - pro: **need not** spell out every detail
  - con: **cannot** spell out every detail
- Missing details must be filled by someone
  - implied in the abstraction, and/or
  - filled in by the synthesis tool
- To be meaningful
  - reduce work, and/or
  - improve outcome

*In HW practice, low tolerance for degraded outcome regardless of ease*

# Good-for-HW Compute Model Examples

- Systolic Array
- Data Parallel
- Dataflow
- Stream Processing
- Commonalities
  - reduce design complexity/effort
  - supports scalable parallelism under simplified global coordination (by imposing a “structure”)
  - allows straightforward, efficient HW mapping
  - BUT, doesn’t work for all problems

**These models are not tied-to HW or SW**



# Good compute models distilled from good design patterns

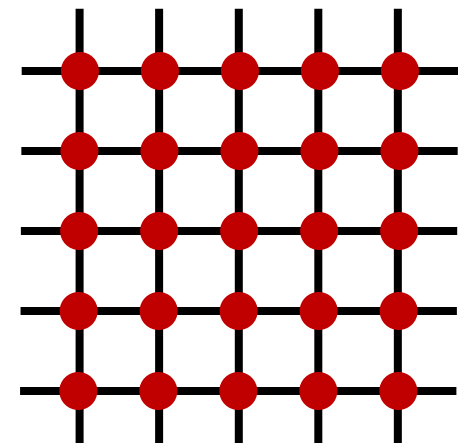
- Both temporal and spatial patterns in
  - computation
  - synchronization
  - data buffering
  - data movement

***What is allowed? uniformity? complexity?***

- What makes it good fit with hardware?
- What makes it good fit with application?
- What limits its generality?

# Systolic Array

- An array of PEs (imagine each an FSM or FSM-D)
  - strictly, PEs are identical; cannot know the size of the array or position in the array
  - could generalize to other structured topologies
- Globally synchronized by “pulses”; on each pulse
  - exchange bounded data with direct neighbors
  - perform bounded compute on fixed local storage
- Scope of design capture is a PE
  - do same thing in every position/pulse
  - localized interactions with identical neighbor



# E.g. Matrix-Matrix Multiplication

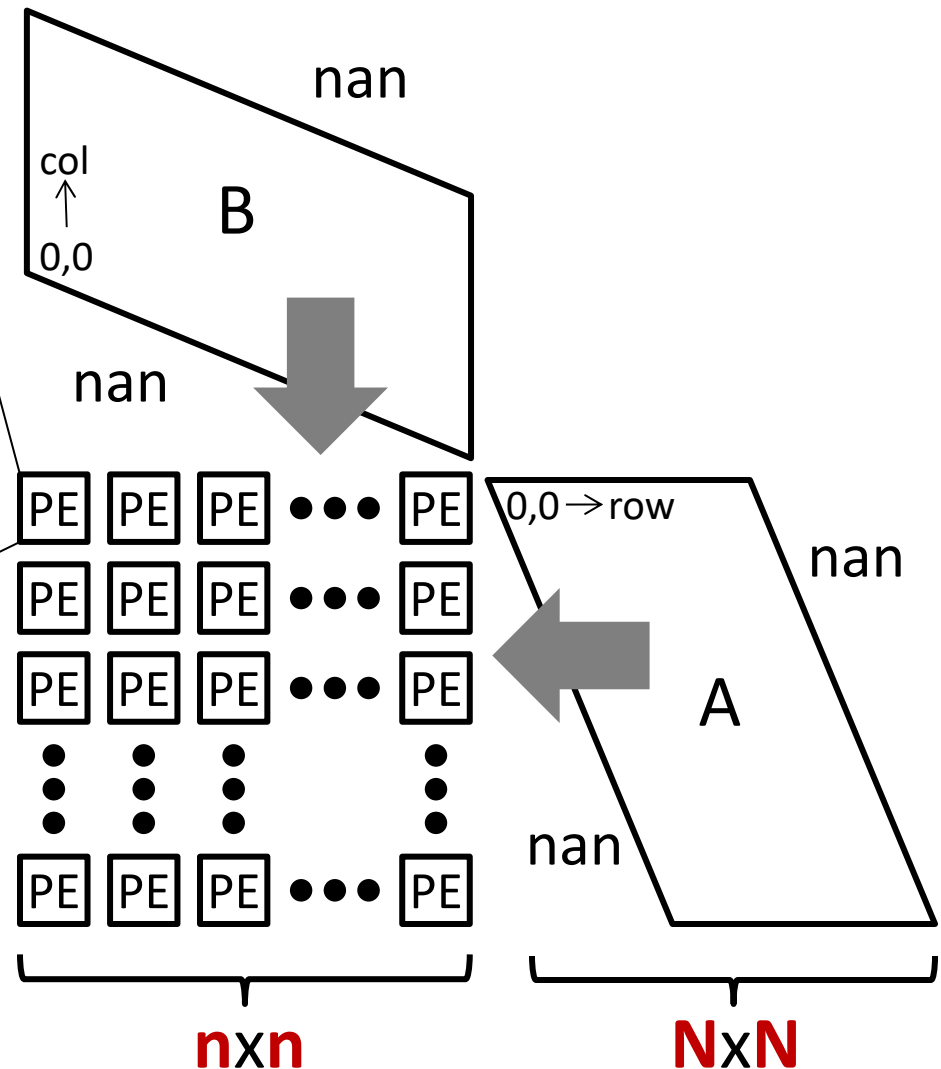
```

a=nan;
b=nan;
accum=0;

For each pulse {
  send-W(a); send-S(b);
  a=rcv-E(); b=rcv-N();
  if (a!=nan)
    accum=a*b+accum;
}

```

- Works for any **n**
- Only stores 3 vals per PE
- If **N**>**n**, emulate at **N**<sup>2</sup>/**n**<sup>2</sup> slowdown



# Does the last slide come to mind when you see??

```
float A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        for(int k=0; k<N; k++) {
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
    }
}
```

# Why systolic array good for HW?

- Parallel and scalable in nature
  - can efficiently emulate key aspects of stream processing and data-parallel
  - easy to build corresponding HW on VLSI (especially 1D and 2D arrays)
- No global communication, except for pulse
- Scope of design/analysis/debug is 1 FSM-D
- Great when it works
  - linear algebra, sorting, FFTs
  - works more often than you think
  - but clearly not a good fit for every problem

# Data Parallelism

- Abundant in matrix operations and scientific/numerical applications
- Example: DAXPY/LINPACK (inner loop of many linear algebra kernels)

$$Y = a * X + Y = \left\{ \begin{array}{l} \text{for } (i=0; i < N; i++) \{ \\ \quad Y[i] = a * X[i] + Y[i] \\ \} \end{array} \right.$$

–  $Y$  and  $X$  are vectors

✓ same operations repeated on each  $Y[i]$  and  $X[i]$

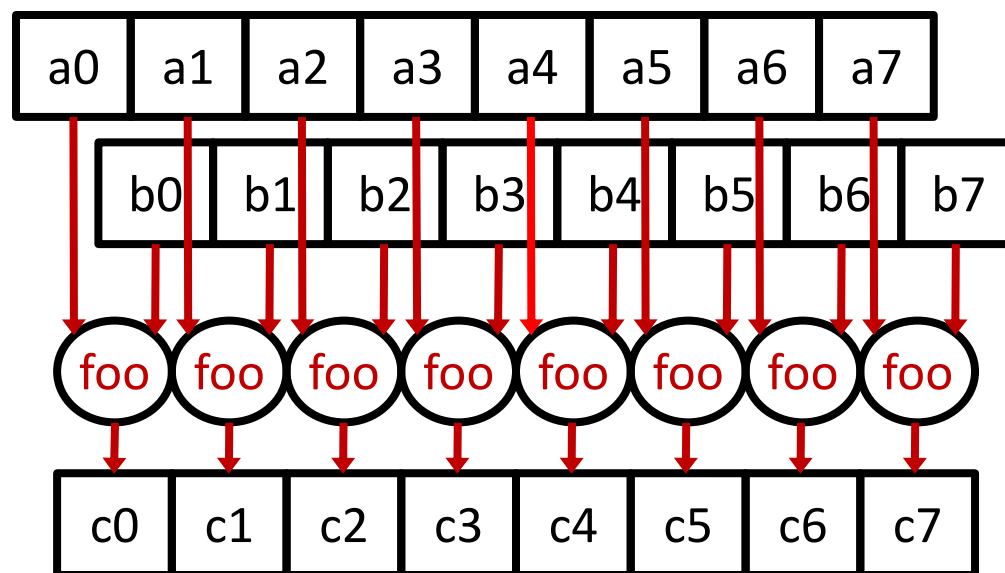
✓ no data dependence across iterations

***How to exploit data parallelism in hardware?***

# Data Parallel Execution

```
for (i=0; i<N; i++) {  
    C[i]=foo(A[i], B[i])  
}
```

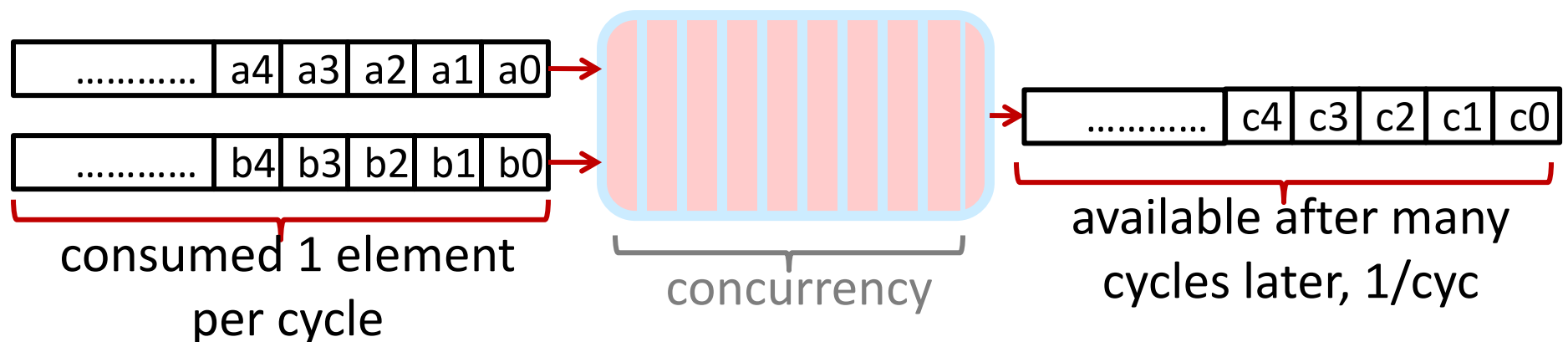
- Instantiate **k** copies of the hardware unit **foo** to process **k** iterations of the loop in parallel



# Pipelined Execution

```
for (i=0; i<N; i++) {
    C[i]=foo (A[i], B[i])
}
```

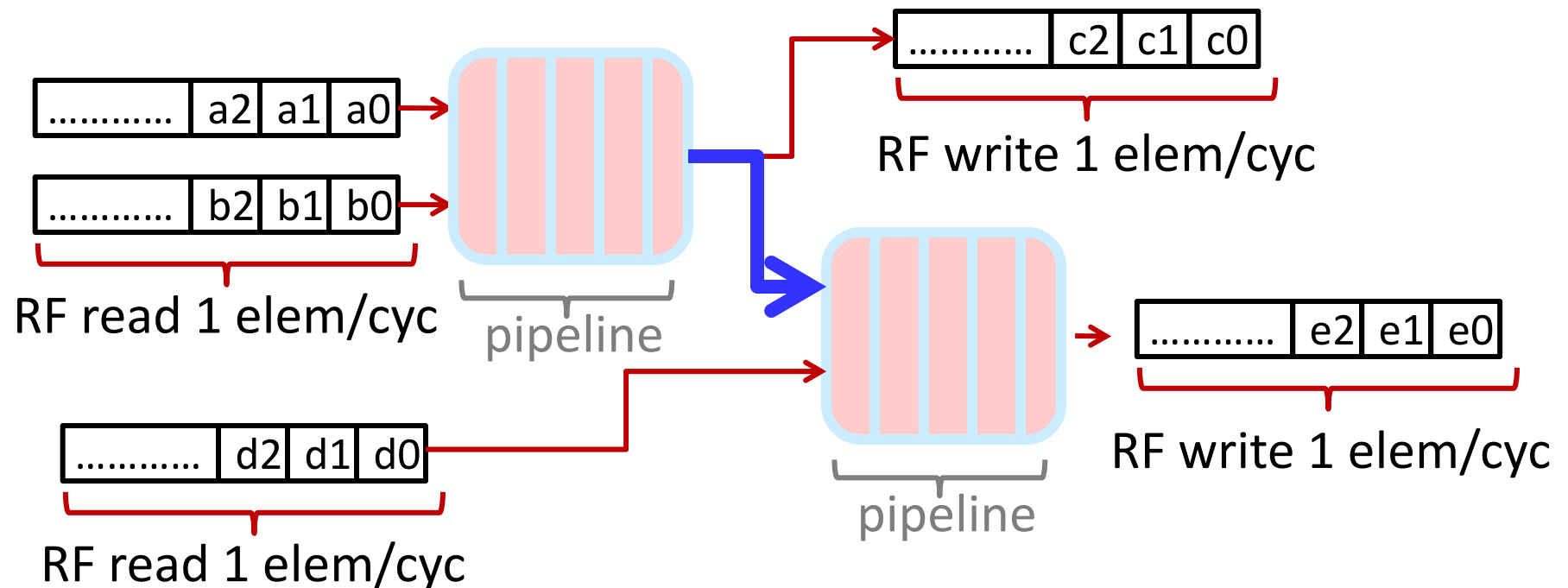
- Build a deeply pipelined (high-frequency) version of `foo()`



*Pipelining also works best when repeating identical and independent compute*



# Aside: Vector Chaining

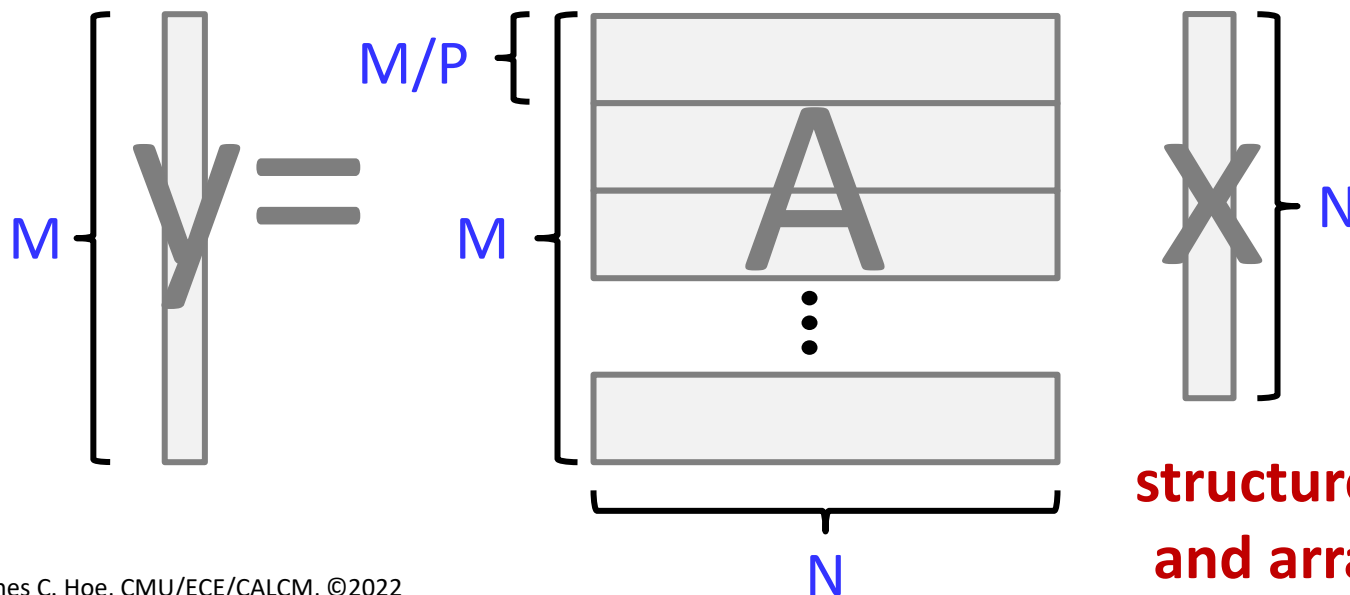


Visualize true (long) vectors “flowing” through the datapath as stream of elements, not as bulk objects

# E.g. SIMD Matrix-Vector Mult

```
// Each of the P threads is responsible for
// M/P rows of A; self is thread id
for(i=self*M/P;i<((self+1)*M/P);i++) {
    y[i]=0;
    for(j=0;j<N;j++) {
        y[i]+=A[i][j]*x[j];
    }
}
```

*seems wasteful to each thread to read each x[] M/P times*



**How to structure memory and array layout?**

# E.g. Vectorized Matrix-Vector Mult

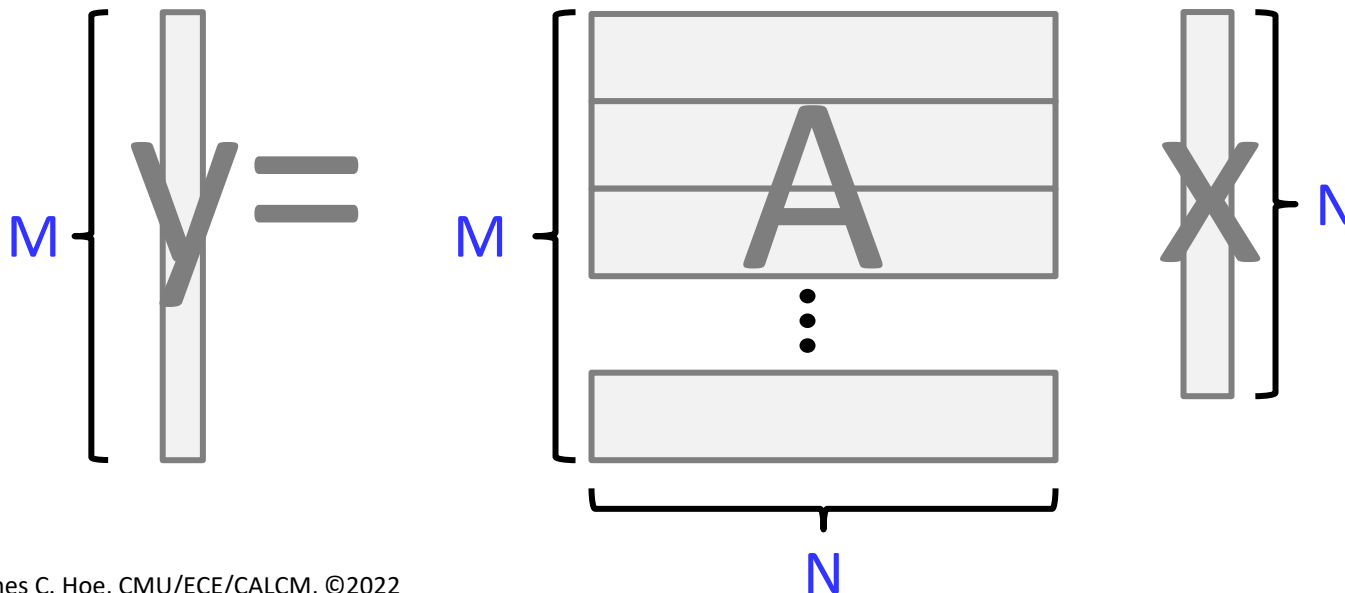
Repeat for each row of A

no such  
instruction  
allowed

```

LV V1, Rx           ; load vector x
LV V2, Ra           ; load i'th row of A
MULV V3, V2, V1     ; element-wise mult
"reduce" F0, V3 ; sum elements to scalar
S.D Ry, F0         ; store scalar result
  
```

(hint: is "reduce" data-parallel?  
what is II of MULV vs "reduce"?)

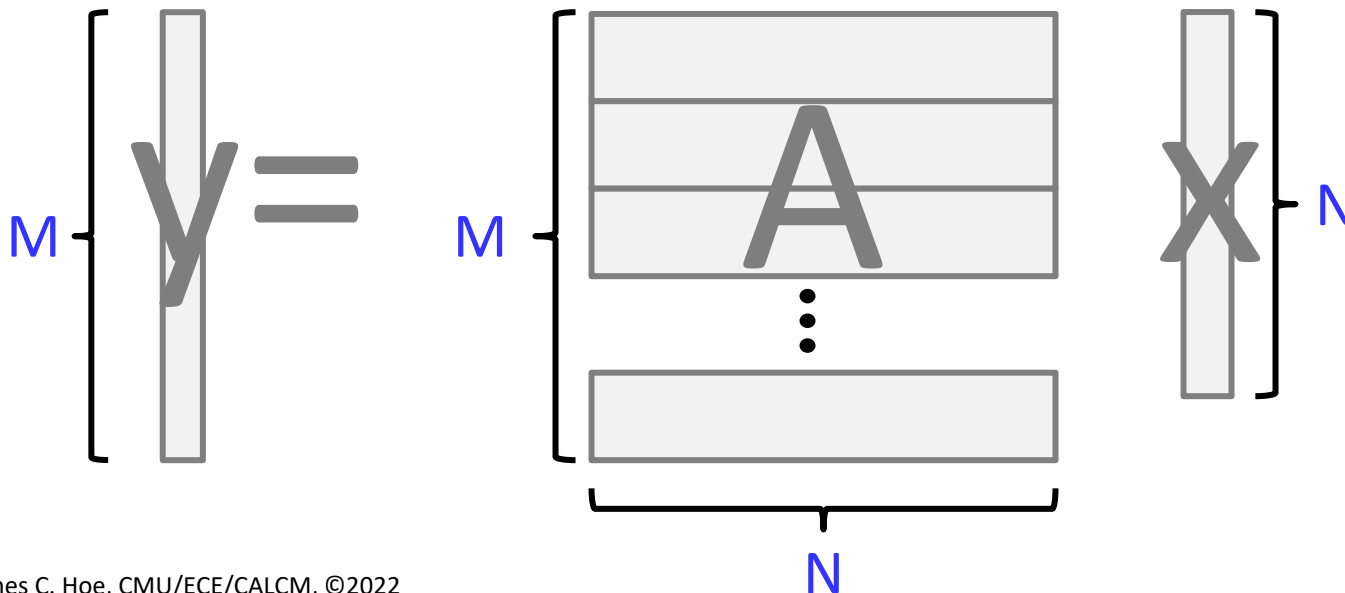


# E.g. Vectorized Matrix-Vector Mult

Repeat for each column of A

DAXPY {  
 LVWS V0, (Ra, Rs) ; load-strided i'th col of A  
 L.D F0, Rx ; load i'th element of x  
 MULVS.D V1, V0, F0 ; vector-scalar mult  
 ADDV.D Vy, Vy, V1 ; element-wise add

*Above is analogous (when/what/where) to the SIMD code*



# Why is data-parallel good-for-HW?

- Simplest but highly restricted parallelism
- Open to mixed implementation interpretations
  - SIMD parallelism +
  - (deep) pipeline parallelism
- Great when it works
  - important form of parallelism for scientific and numerical computing
  - but clearly not a good fit for every problem

# Dataflow Graphs

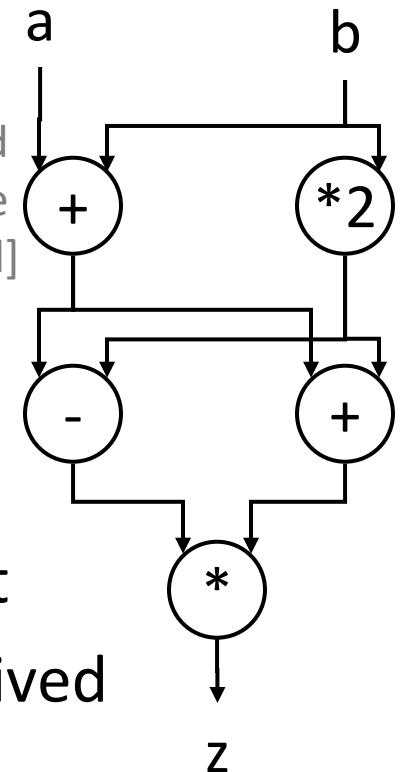
- Consider a von Neumann program
  - what is the significance of the program order?
  - what is the significance of the storage locations?

```

v := a + b;
w := b * 2;
x := v - w;
y := v + w;
z := x * y

```

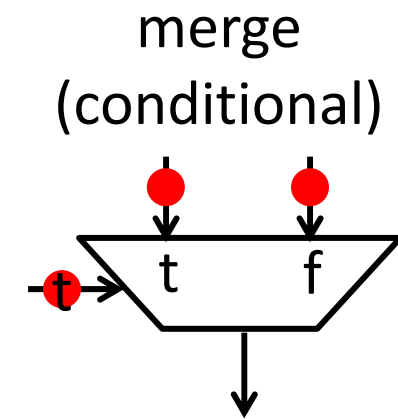
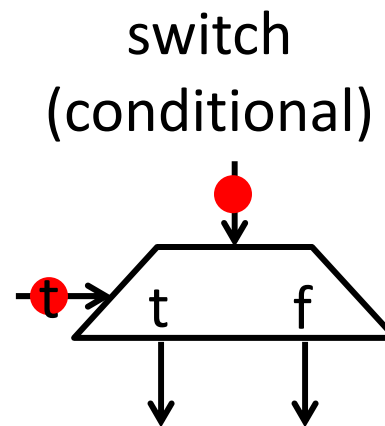
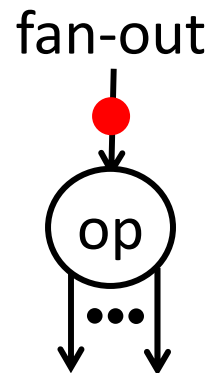
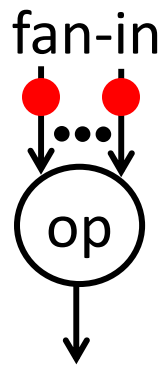
[figure and  
example  
from Arvind]



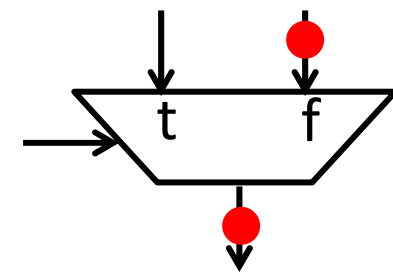
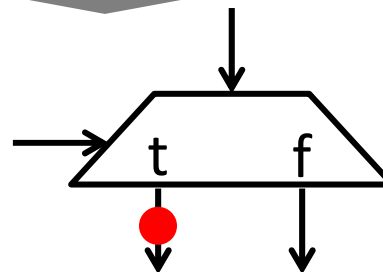
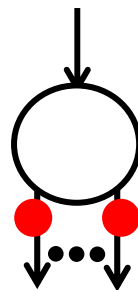
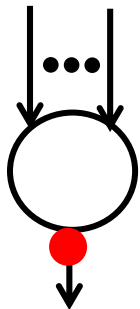
- Dataflow operation ordering and timing implied in data dependence
  - instruction specifies who receives the result
  - operation executes when all operands received
  - “source” vs “intermediate” representation

*(There is a lot more to this, e.g., loops, fxns)*

# Token Passing Execution



“fire” output tokens when  
all required input present

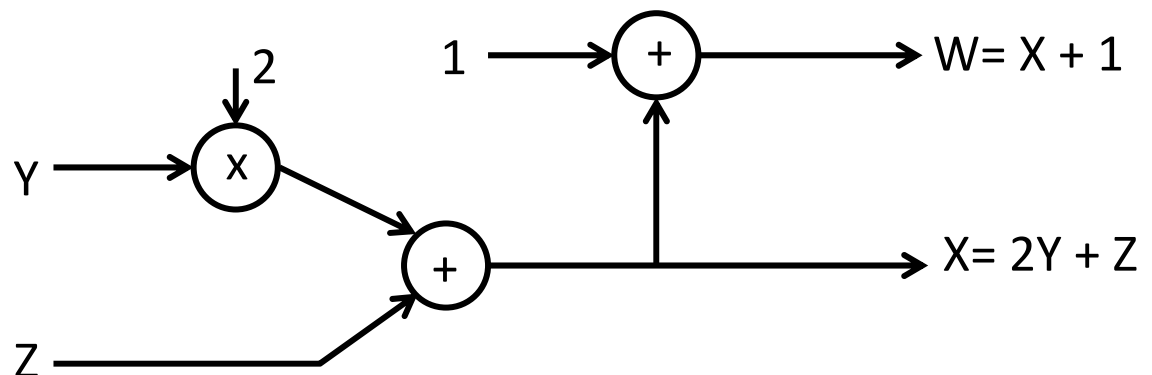


consider multi-, variable-cycle ops and links

# Synchronous Dataflow

- Operate on flows (sequence of data values)
  - i.e.,  $X = \{x_1, x_2, x_3, \dots\}$ , “1” =  $\{1, 1, 1, 1, \dots\}$
- Flow operators, e.g., switch, merge, duplicate
- Temporal operators, e.g.  $\text{pre}(X) = \{\text{nil}, x_1, x_2, x_3, \dots\}$

Fig 1, Halbwachs, et al., The Synchronous Data Flow Programming Language LUSTRE



*Function vs Execution vs Implementation*

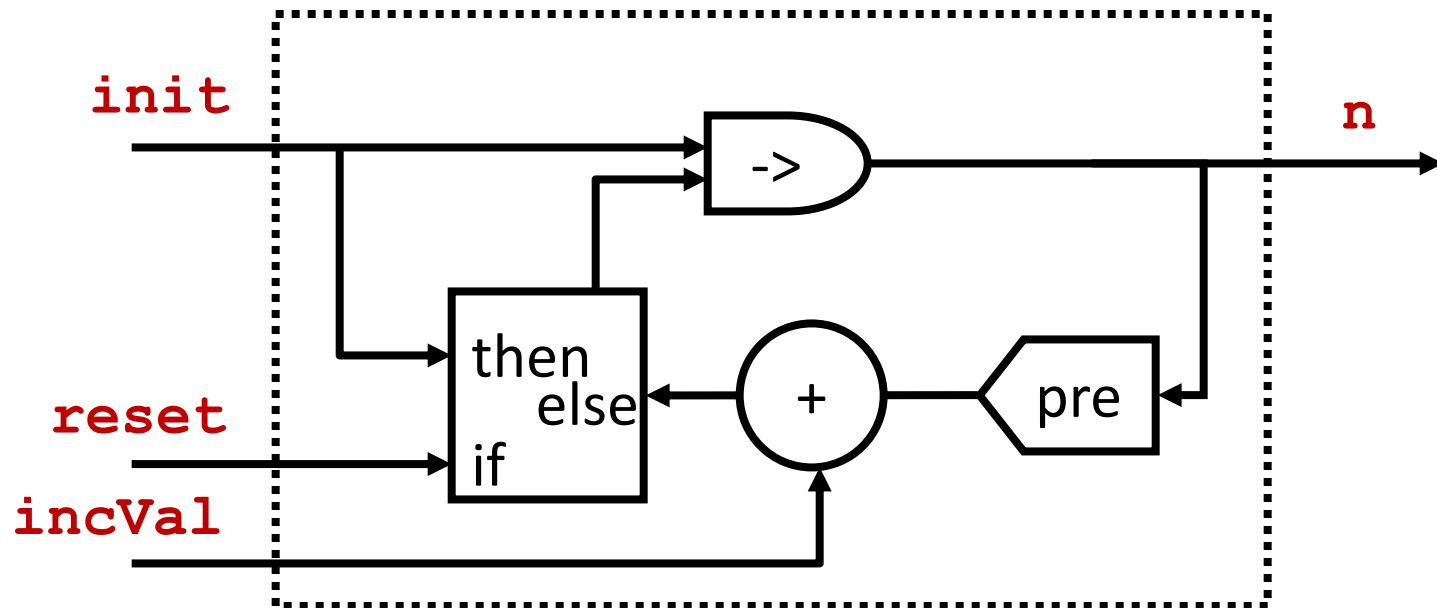


# What do you make of this?

```

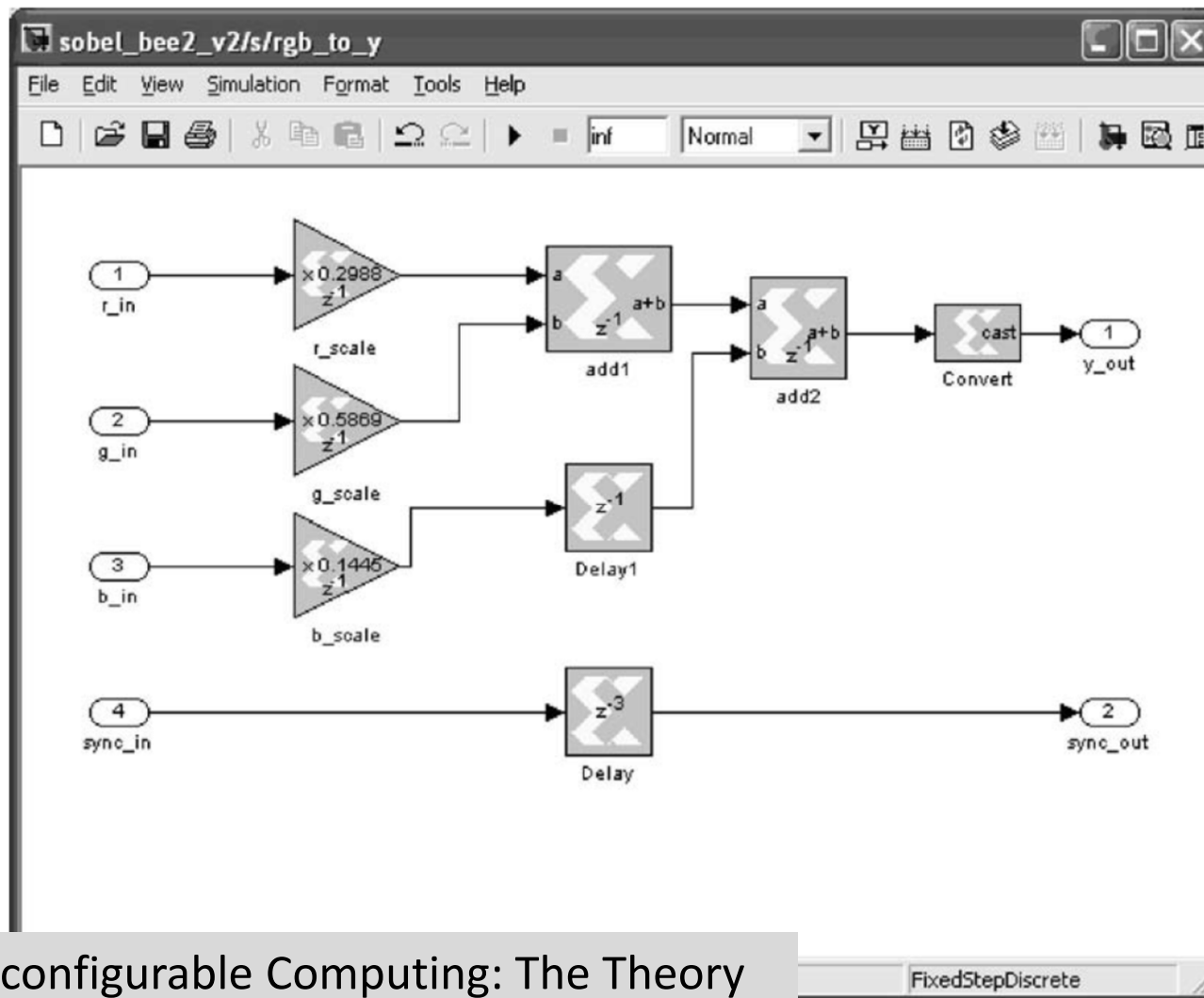
node ACCUM(init, incVal: int; reset: bool) returns
  (n: int);
let
  n = init -> if reset then init else pre(n) + incr
tel

```



$\text{pre}(\{e_1, e_2, e_3, \dots\})$  is  $\{\text{nil}, e_1, e_2, e_3, \dots\}$   
 $\{e_1, e_2, e_3, \dots\} \rightarrow \{f_1, f_2, f_3, \dots\}$  is  $\{e_1, f_2, f_3, f_4, \dots\}$

# Try Simulink in Vitis Model Composer



[Figure 8.1: “Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation”]

# Why is dataflow good-for-HW?

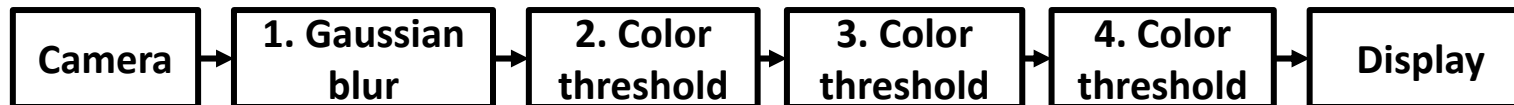
- Naturally express fine-grain, implicit parallelism
  - Many variations, asynchronous, dynamic, . . .
- Loose coupling between operators
  - synchronize by order in flow, not cycle or time
  - no imposed operation ordering
  - no global synchronization/communications
- Declarative nature permits implementation flexibilities
- Great when it works
  - excellent match with signal processing
  - but clearly not a good fit for every problem

# Stream Processing

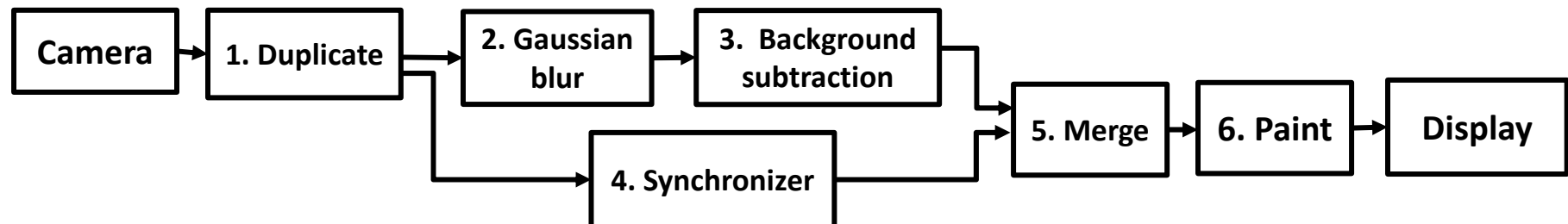
- Related to dataflow
  - operate on data in sequence (no random access)
  - repeat same operation on data in a stream
- Emphasis on IPs and their composition
  - design in terms of composing valid stream-to-stream transformations
  - simple, elastic, plug-and-play “interface”
- More flexible rules
  - input and output flows need not be synchronized
  - operator can have a fixed amount of memory
    - buffer/compute over a window of values
    - carry dependencies over values in a stream

# Regular and Data-Independent: E.g., Vision Processing Pipeline

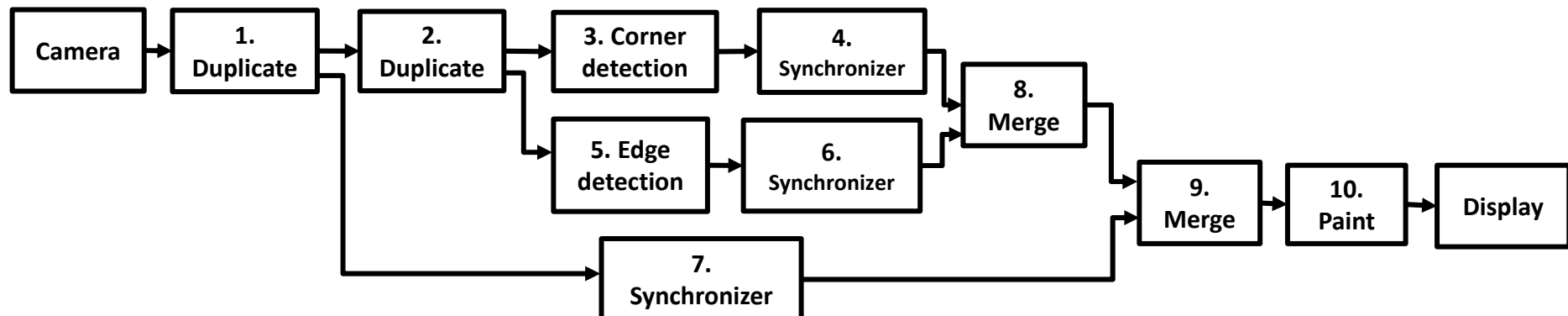
Color-based object tracking (linear pipeline, 4 stages)



Background subtraction (2-branch pipeline, 6 stages)

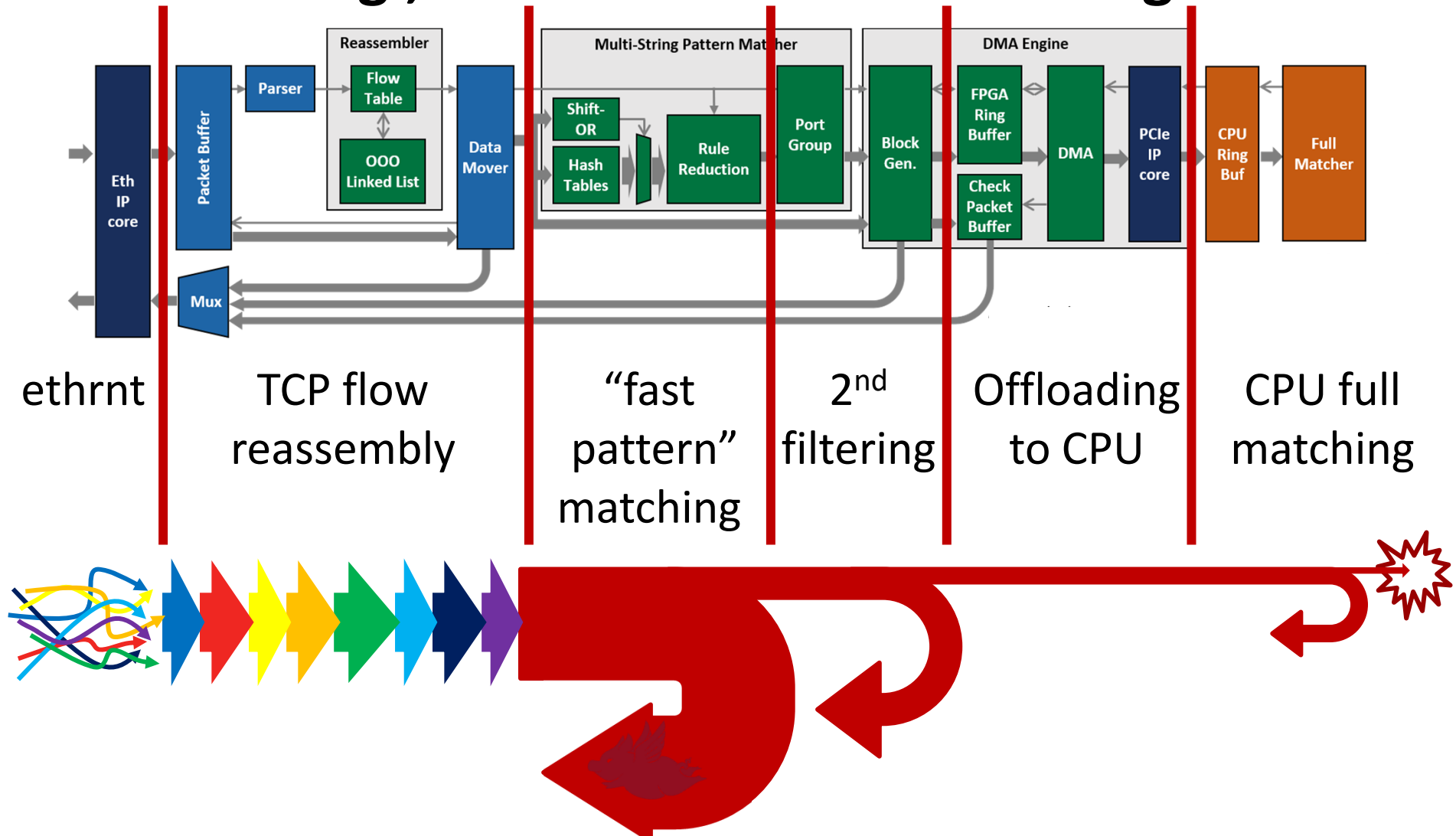


Corner + edge detection (3-branch pipeline, 10 stages)



# Irregular and Data-Dependent

## E.g., Network Packet Processing



<https://github.com/cmu-snap/pigasus>



# Commonalities Revisited

- Parallelism under simplified global coordination
  - enforced regularity
  - asynchronous coupling
- Straightforward efficient mapping to hardware
  - low performance overhead
  - low resource overhead
  - high resource utilization
- Simplify design without interfering with quality
- But only works on specific problem patterns



# Parting Thoughts:

## Conflict between High-Level and Generality

insist on quality

