Lecture 1

Dataflow Model of Parallelism

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Oregon Programming Language Summer School (OPLSS)
Eugene, OR
July 14, 2018

Dataflow

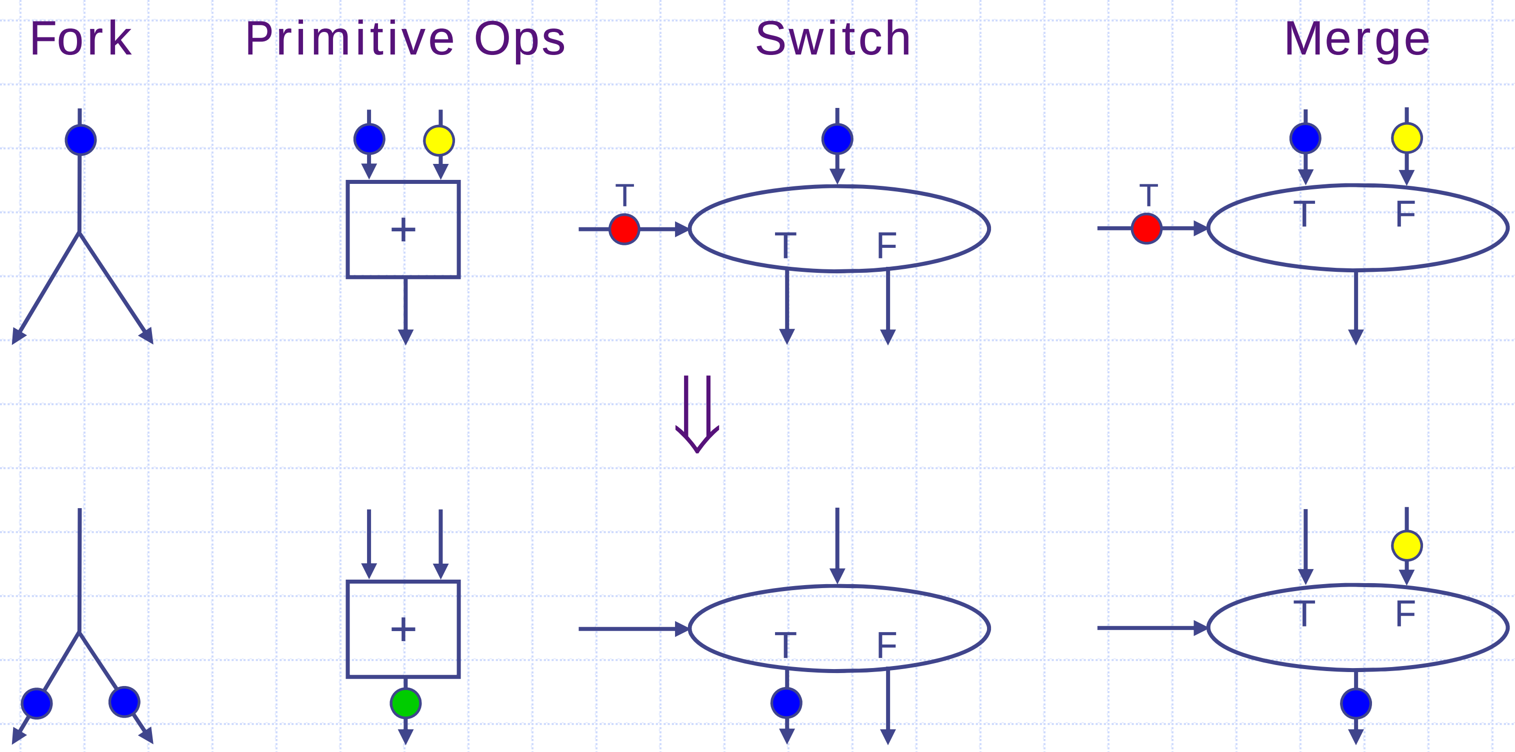
Jack Dennis 1969-1973

General purpose parallel machines based on a dataflow graph model of computation

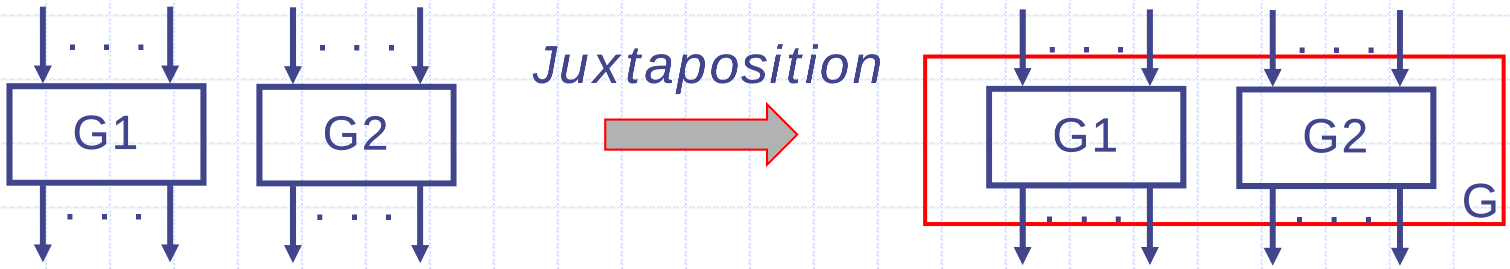
Inspired all the major players in dataflow during seventies and eighties, including Kim Gostelow and me @ UC Irvine

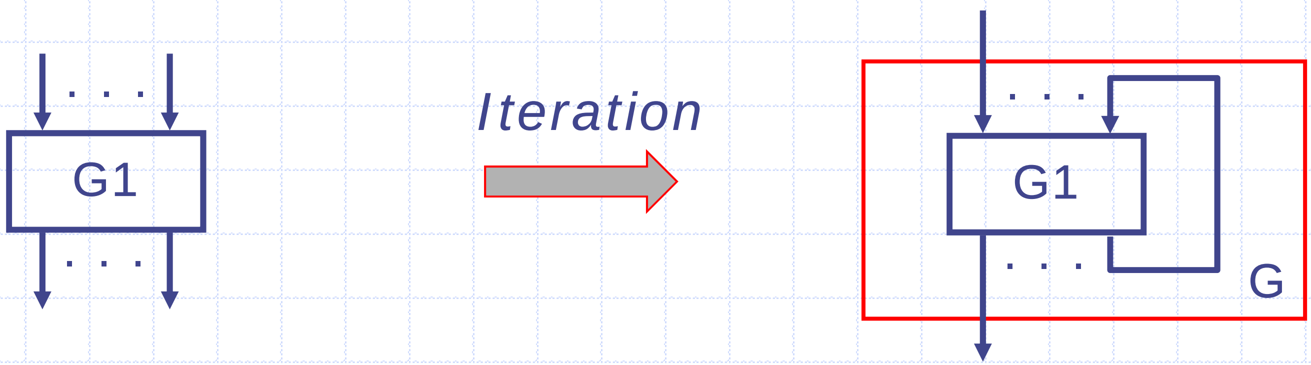
Dataflow Operators

- A small set of dataflow operators can be used to define a general programming language



Constructing Dataflow Graphs

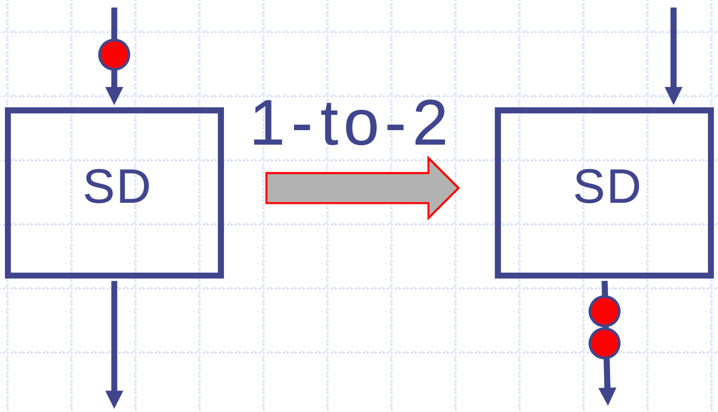


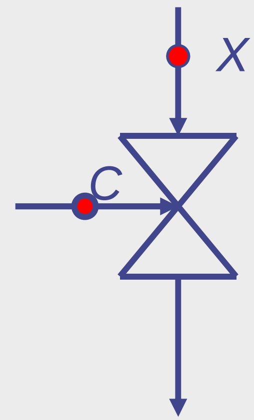


Exercise: Construct a graph for f(g(x)) using juxtaposition and iteration

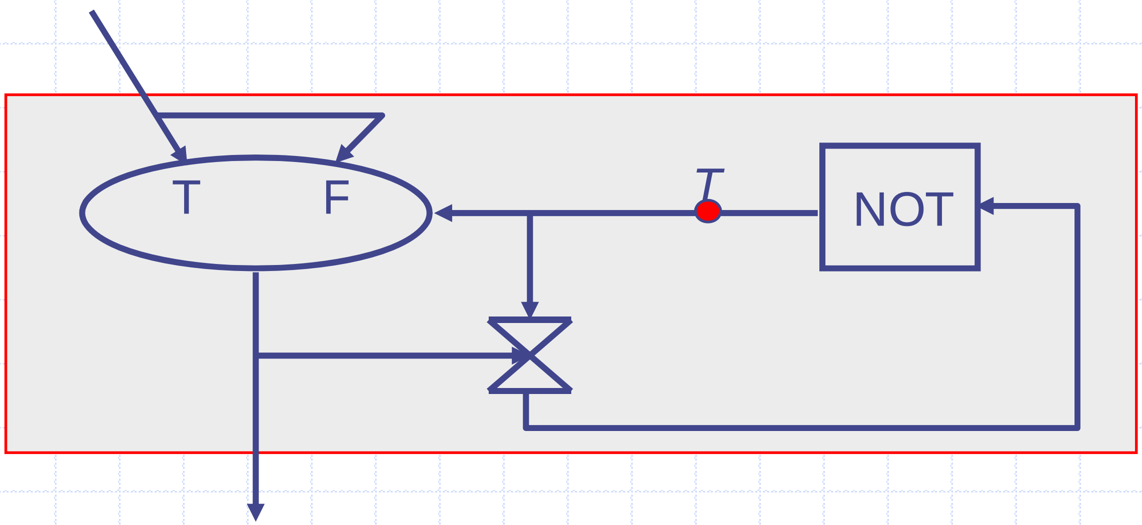
Example:

The Stream Duplicator



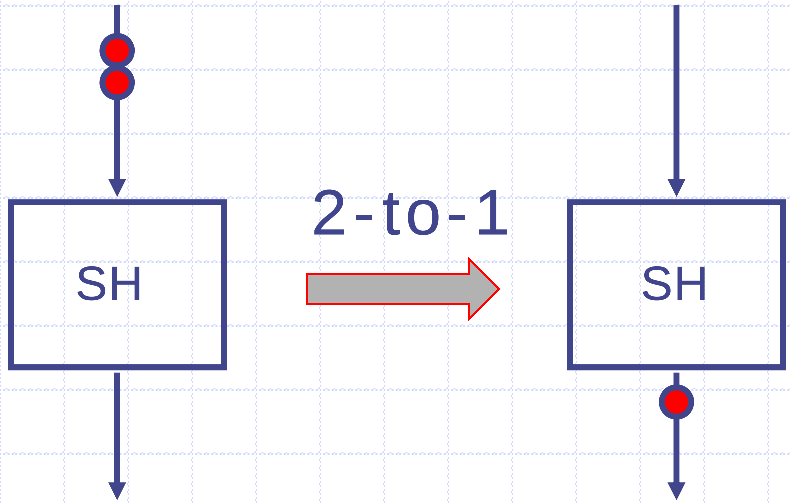


Gate: Lets X pass through only after C arrives



What happens if we don't use the gate?

The Stream Halver



Draw a graph that throws away every other token

Determinacy Property

- The values of output tokens are uniquely determined by the values of input tokens, i.e., *the behavior is time independent*

- Theorem: A dataflow graph formed by repeated *juxtaposition* and *iteration* of deterministic dataflow operators is deterministic

Proof?

Kahn Process Networks

Gilles Kahn 1973

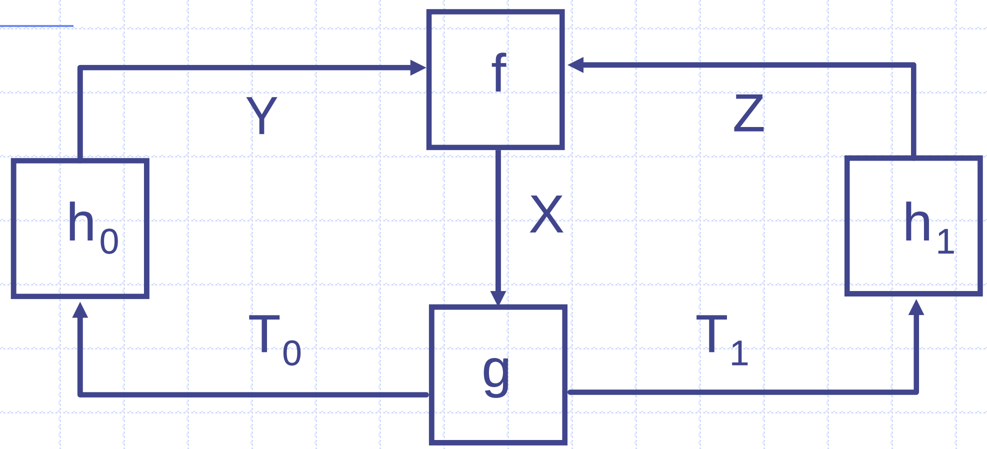
- Computing stations connected by *unbounded, FIFO channels*
- Each station executes a *sequential program*

  wait(ch): blocking read from a channel

  send(x,ch): non blocking

a station either *blocks* for an input on a specific channel or computes *(no test for emptyness)*

An Example



$X = f(Y,Z) \quad || \quad T_0 , T_1 = g(X) \quad || \quad Y = h_0(T_0) \quad || \quad Z = h_1(T_1)$

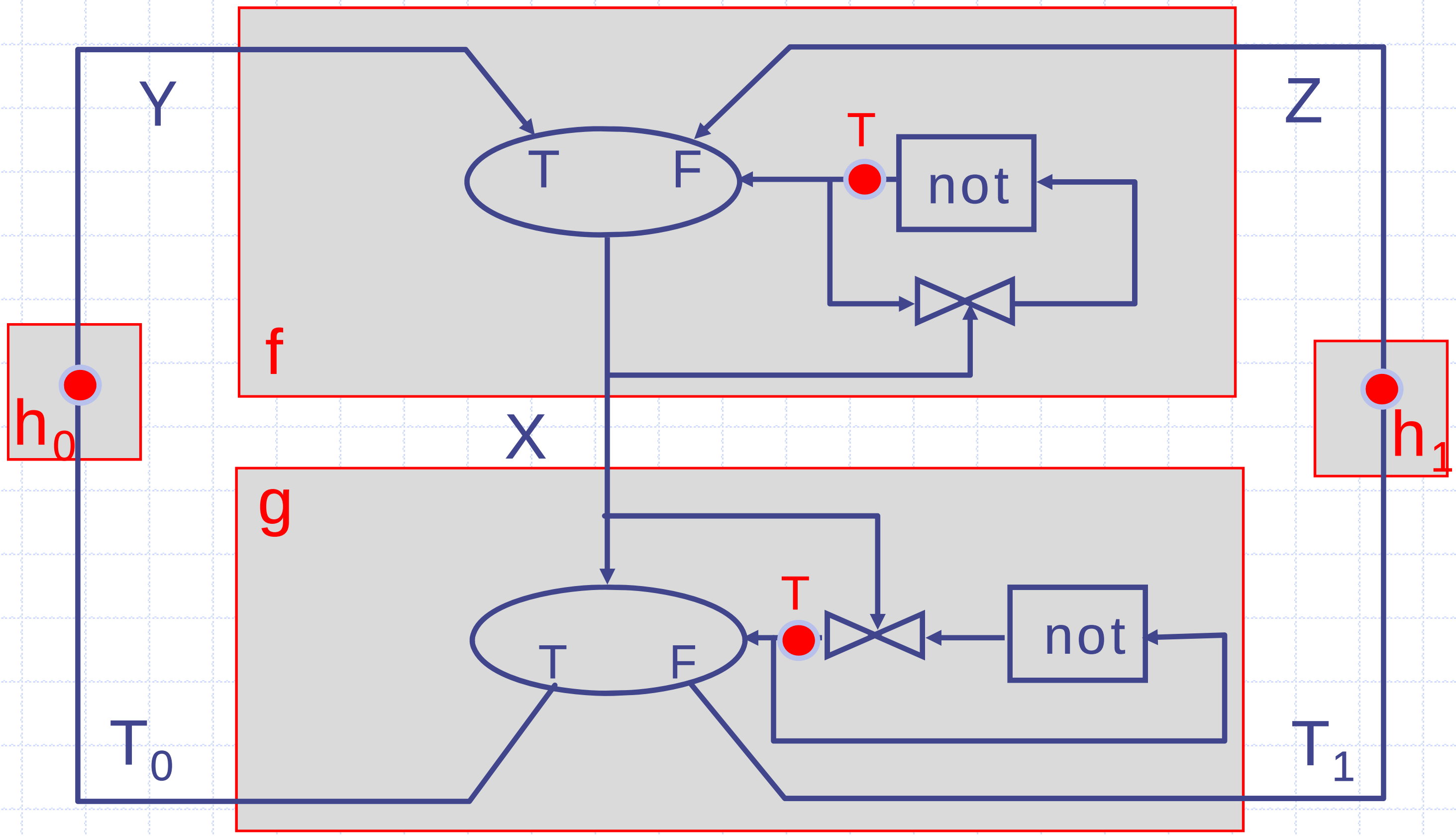
```
Process  f(U,V; W)
  {b= true;
   While true do
    {i := if b then wait(U)
             else wait(V);
     print(i);
     send(i,W);
     b := not b}}
```

```
Process  g(U ; V,W)
  {b= true;
   While true do
    {i := wait(U);
     if b then send(i,V)
     else send(i,W);
     b := not b }}
```

```
Process  hc (U ; V)
  {send(c,V);
   While true do
    {i := wait(U);
     send(i,V) }}
```

Functionality?

Kahnian Networks & Dataflow



Dataflow Graphs can Express any Kahnian Network
and vice versa

Determinacy

- A computing station in Kahnian network can be viewed as a function from sequences to sequences
- A Kahnian network can be viewed a systems of recursive equations, whose solutions characterize the I/O behavior of the network
- Kleene's Fixed Point Theorem: If each function in the network is *monotonic* and *continuous* then the set of recursive equations has a unique least fixed-point solution

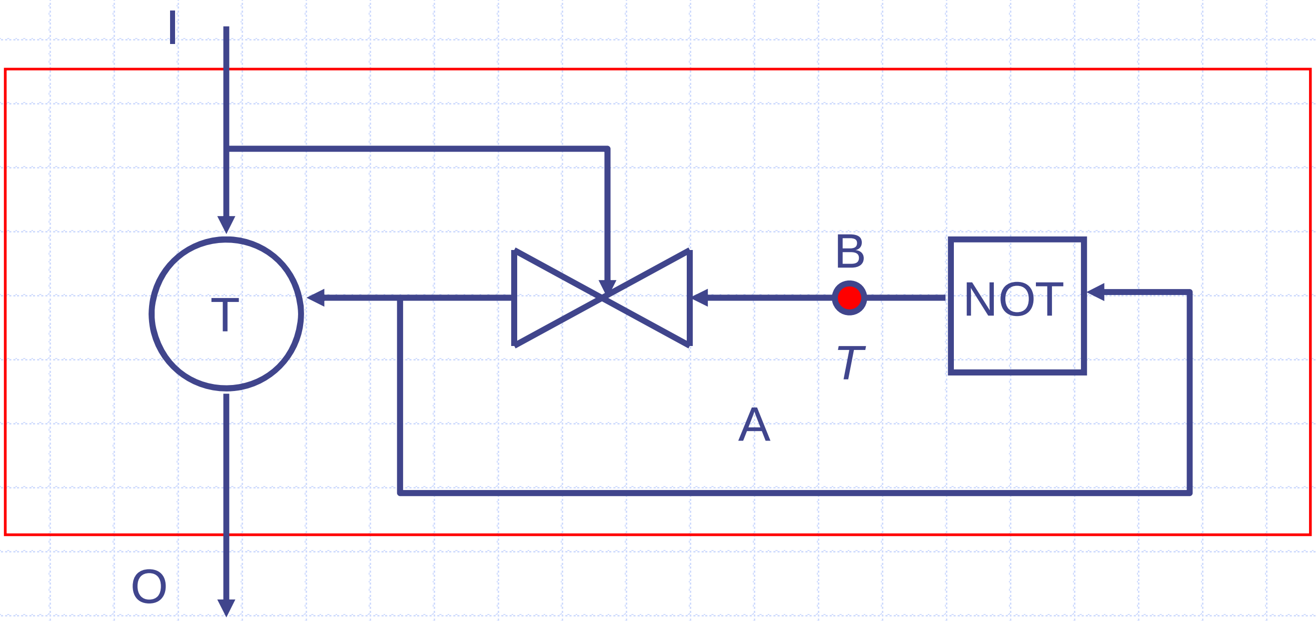
Dataflow Operators as Streams Functions

```
add(x:xs, y:ys)   =   +(x,y) : add(xs,ys)

T-gate (T:bs, x:xs)   =   x: T-gate(bs,xs)
T-gate (F:bs, x:xs)   =   T-gate(bs,xs)

merge(T:bs, x:xs, ys)   =   x: merge(bs,xs,ys)
merge(F:bs, xs, y:ys)   =   y: merge(bs,xs,ys)
```

Dataflow Graphs: A Set of Recursive Equations



O = T-gate (A,I);
A = gate (I,B);
B = T : Not (A) ;

G.Kahn: *Monotonicity* and *Continuity* of operators guarantee a unique solution, aka determinacy

Kahn Networks = Dennis networks

Domain of Sequences

- *Sequence:* $[x_1, \ldots, x_n]$
- *The least element:* [ ] (aka $\perp$)
- *The partial order ($\leq$):* prefix order on sequences
  - $[\,] \leq [x_1] \leq [x_1,x_2] \leq \ldots \leq [x_1, x_2, x_3 \ldots x_n]$
- *Monotonicity:* $x \leq y \Rightarrow f(x) \leq f(y)$
  - a monotonic operator on sequences can never retract a value that has been produced.
- *Continuity:* $f\,(\mathrm{U}_i\, X_i) = \mathrm{U}_i\, f\,(X_i)$
  - A continuous operator on sequences does not suddenly produce an output after consuming an infinite amount of input
- *Least fixed-point solution:* $f(f(\ldots(f(\perp))\ldots)$

Computing the solution

an example

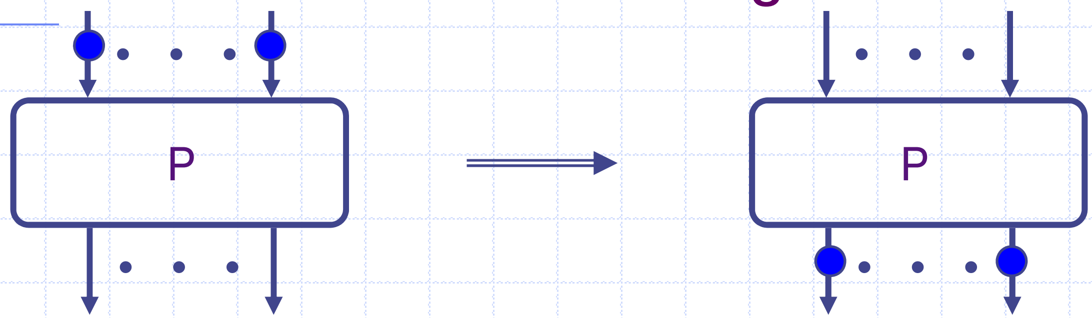
O = T-gate (A,I) ; A = gate (I,B) ; B = T:Not (A) ;

Assume edges without initial tokens are empty (⊥).
The least fixed point solution can be computed iteratively, by evaluating one operator at a time

I	[i1, i2, i3]	[i1, i2, i3]	[i1, i2, i3]
A	[ ]	[T]	[T]
B	[T]	[T]	[T, F]
O	[ ]	[ ]	[i1]

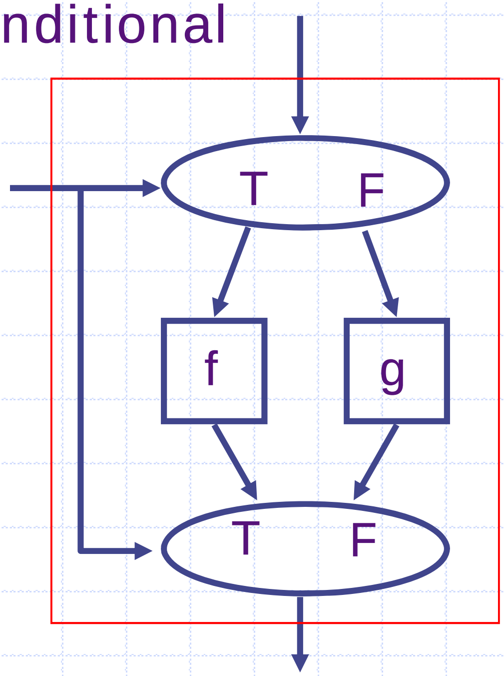
Well Behaved Schemas

one-in-one-out & self initializing

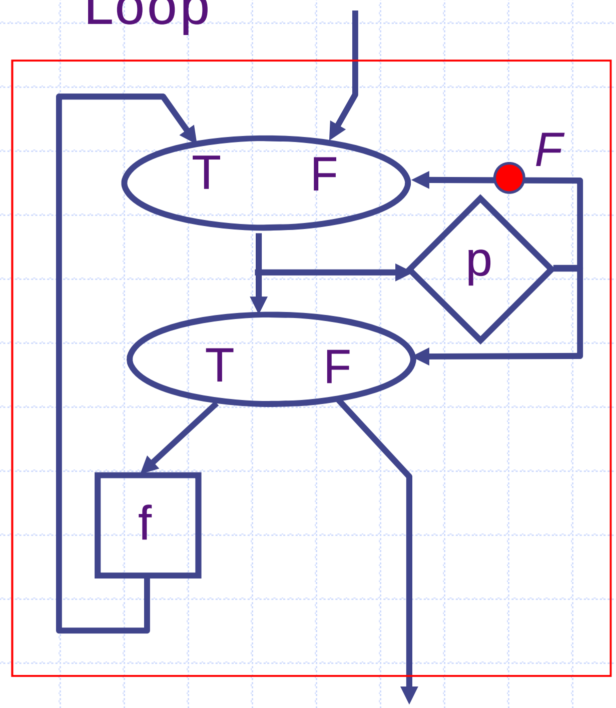


Before After

Conditional

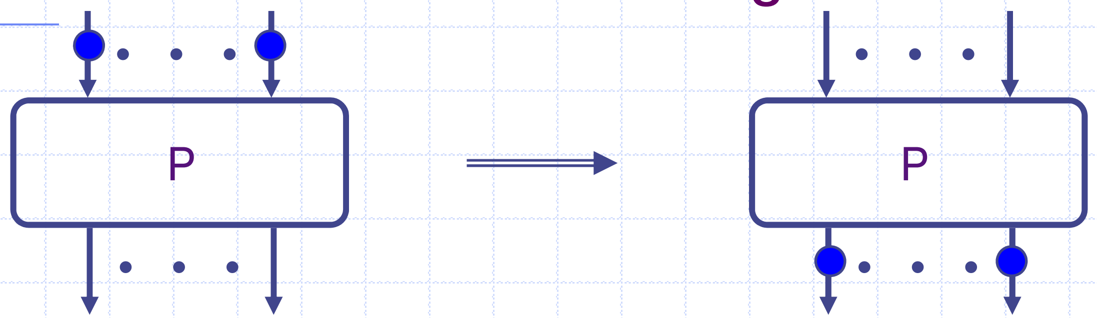


Loop



Well Behaved Schemas

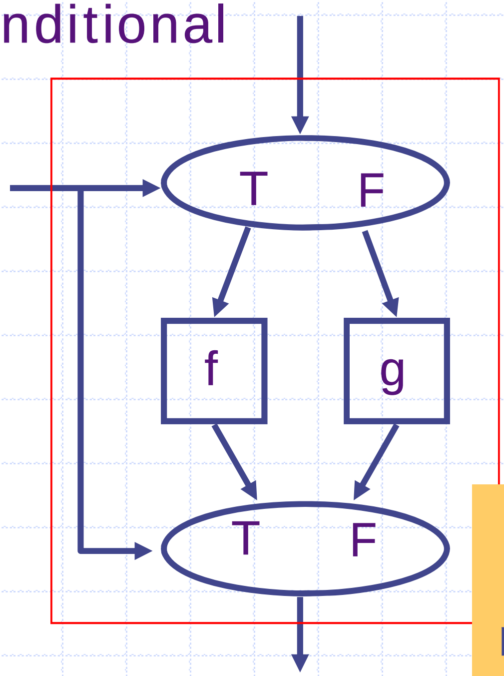
one-in-one-out & self initializing



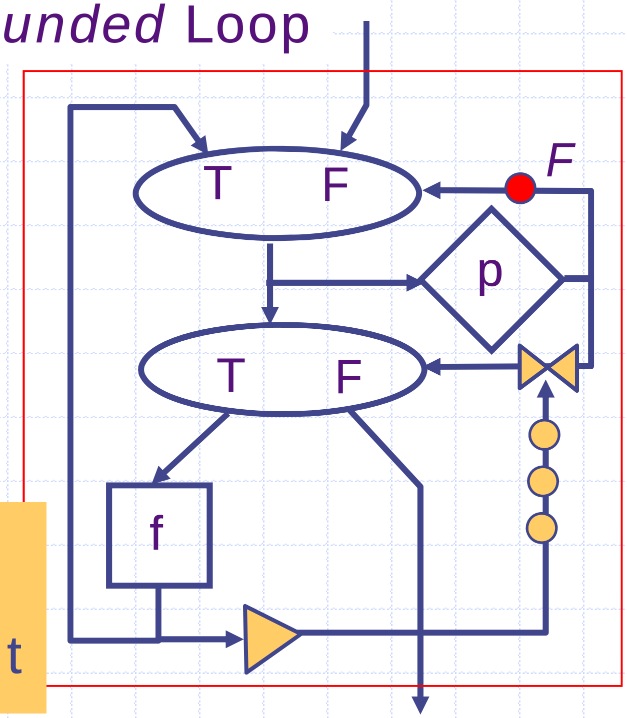
Before

After

Conditional

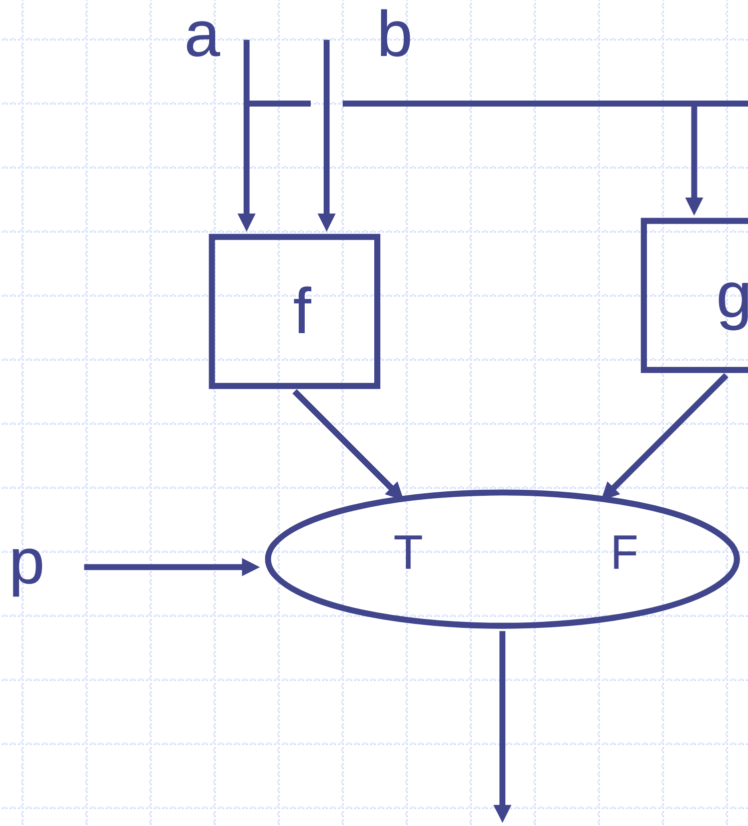


Bounded Loop



Needed for resource management

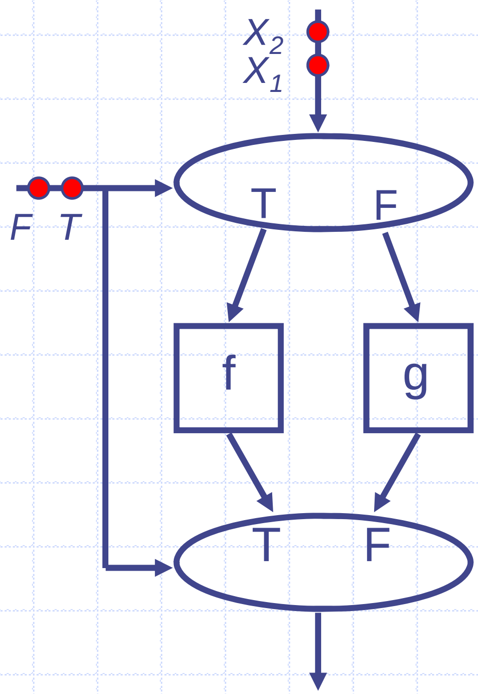
Another Conditional Schema

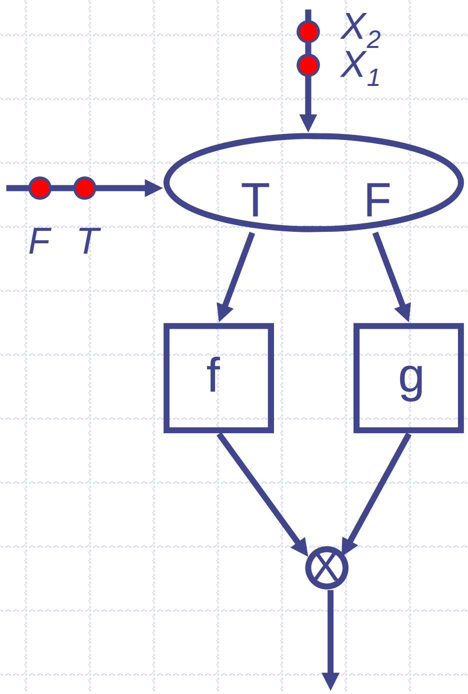


If p *then* f(a,b) *else* g(a,b)

Why is this schema not well behaved?

Yet another conditional schema





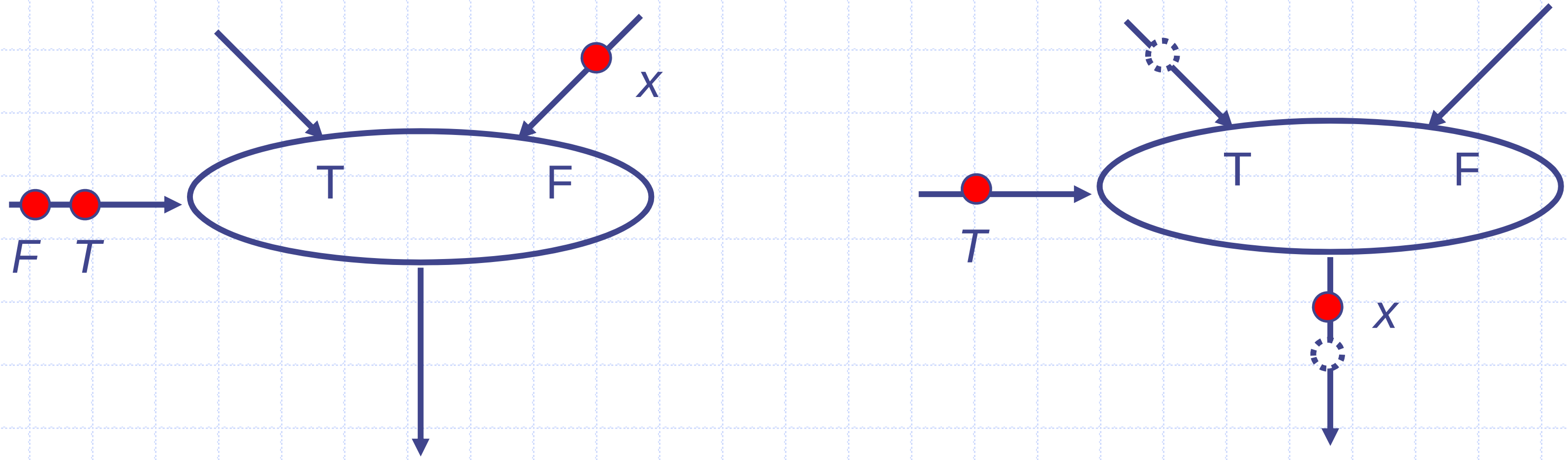
Suppose g(X2) computes much faster than f(X1).

Tokens will come out in the wrong order without the merge operator

Tagged-Token Dataflow

Increasing the concurrency

Missing Tokens and blocking behavior



x cannot be moved to the output because the token corresponding to T is missing.

How to model stream with “holes” ?

Another Interpretation of DFGs: *Streams with "holes"*

Streams with missing tokens

$$\{v_1, v_2, \perp, v_4, \perp, v_6, \ldots.\}$$

can be modeled by a set of tokens where tokens carry a tag designating their position in the stream

$$\{ <1, v_1>, <2, v_2>, <4, v_4>, <6, v_6> \}$$

Tagged Semantics of Operators

$$\text{add}(xs,ys) = \{\langle i,x+y\rangle \mid \langle i,x\rangle \in xs,\ \langle i,y\rangle \in ys\}$$

$$\text{T-gate}(bs,xs) = \{\langle k,x\rangle \mid \langle i,T\rangle \in bs,\ \langle i,x\rangle \in xs,\ \forall j \le i\ .\ \langle j,b_j\rangle \in bs\ ,\ k = \text{T-Count}(bs,i)\}$$

$$\begin{aligned}\text{merge}(bs,xs,ys) = {} & \{\langle i,x\rangle \mid \langle i,T\rangle \in bs,\ \langle k,x\rangle \in xs,\ \forall j \le i\ .\ \langle j,b_j\rangle \in bs,\ k = \text{T-Count}(bs,i)\} \\ & \cup \{\langle i,y\rangle \mid \langle i,F\rangle \in bs,\ \langle k,y\rangle \in ys,\ \forall j \le i\ .\ \langle j,b_j\rangle \in bs,\ k = \text{F-Count}(bs,i)\}\end{aligned}$$

$$D_a(xs) = \{\langle i+1,x\rangle \mid \langle i,x\rangle \in xs\} \cup \{\langle 1,a\rangle\}$$

needed wherever we place an initial token

Ordering on Streams with Holes

Stream with holes: $\{ \langle i, v_i \rangle, \langle j, v_j \rangle, \langle k, v_k \rangle \}$

The least element: $\{ \}$ (aka $\perp$)

The partial order ($\leq$): subset order

It is easy to show that all the operators under the tagged semantics are *monotonic* and *continuous*

$\Rightarrow$ *tagged semantics are also deterministic*

Tagged versus FIFO Interpretation

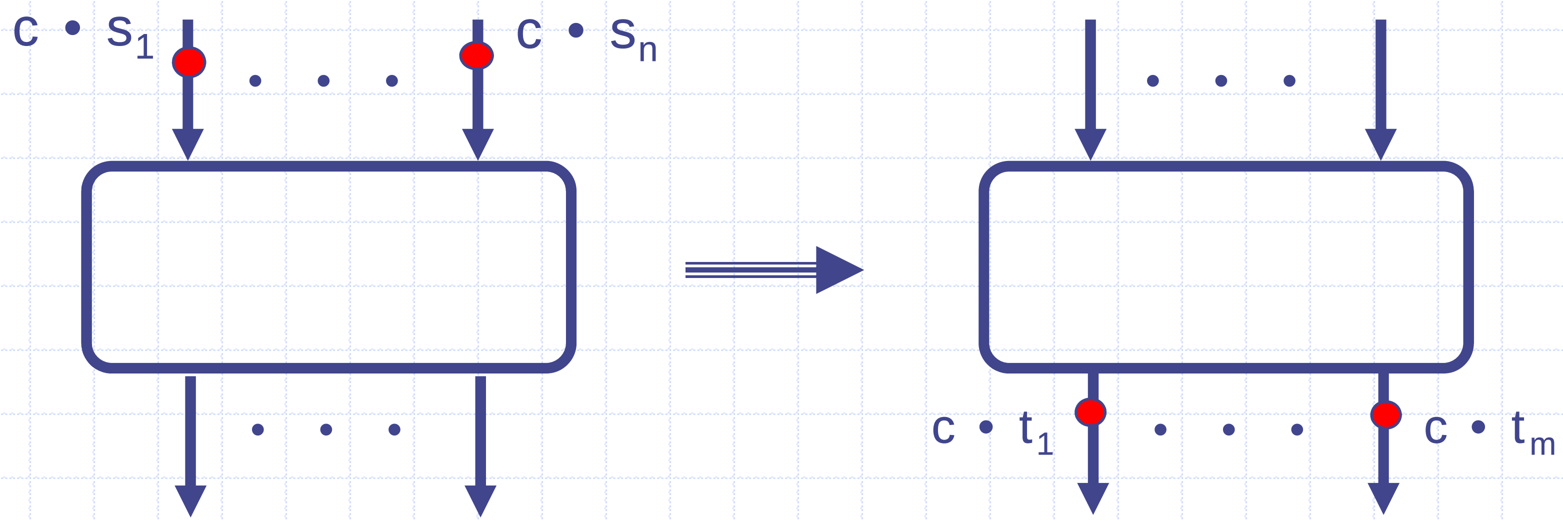
Theorem: Suppose the least fixed point of a dataflow program under the FIFO interpretation is X and under the tagged interpretation is Y then $X \leq Y$.

Proof: Based on structural induction on the graph structure (*juxtaposition* and *iteration*)

Tagged interpretation has more parallelism

Arvind & Gostelow 1977

Well Behaved Graphs under tagging



1. *One-in-one-out property*

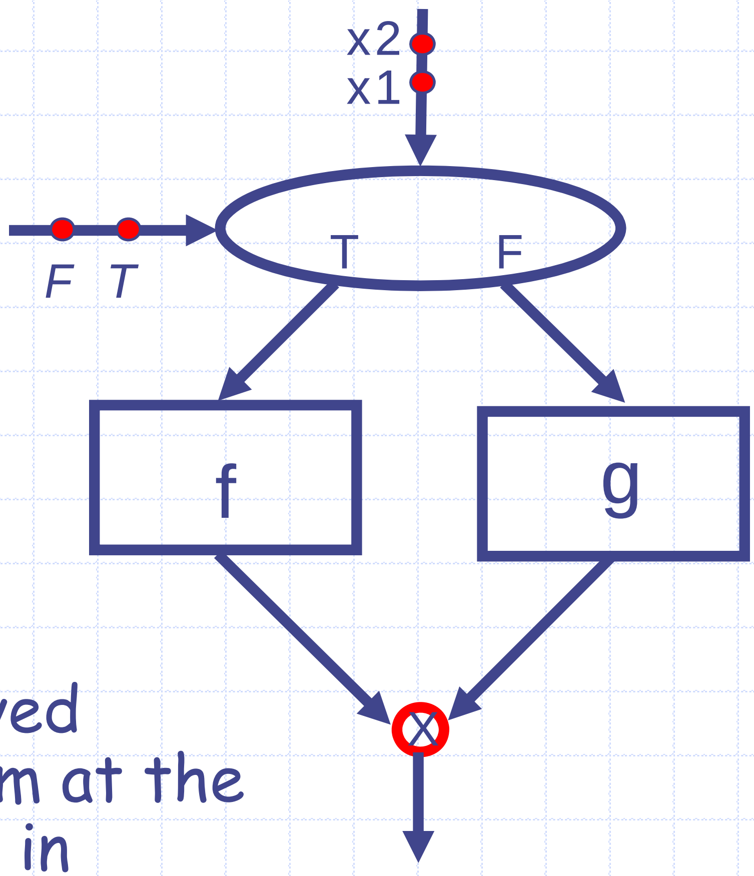
2. *Self Cleaning*: Initially the graph contains no tokens; when all the output tokens have been produced no tokens remain in the graph

Consequences of Tagging

- No need of *merge* in *well behaved* dataflow graphs

- Even if f(x1) completes after g(x2) the output tokens will carry the "correct" tag

- Tagging simplifies well-behaved schemas, increases parallelism at the cost of increased complexity in implementation



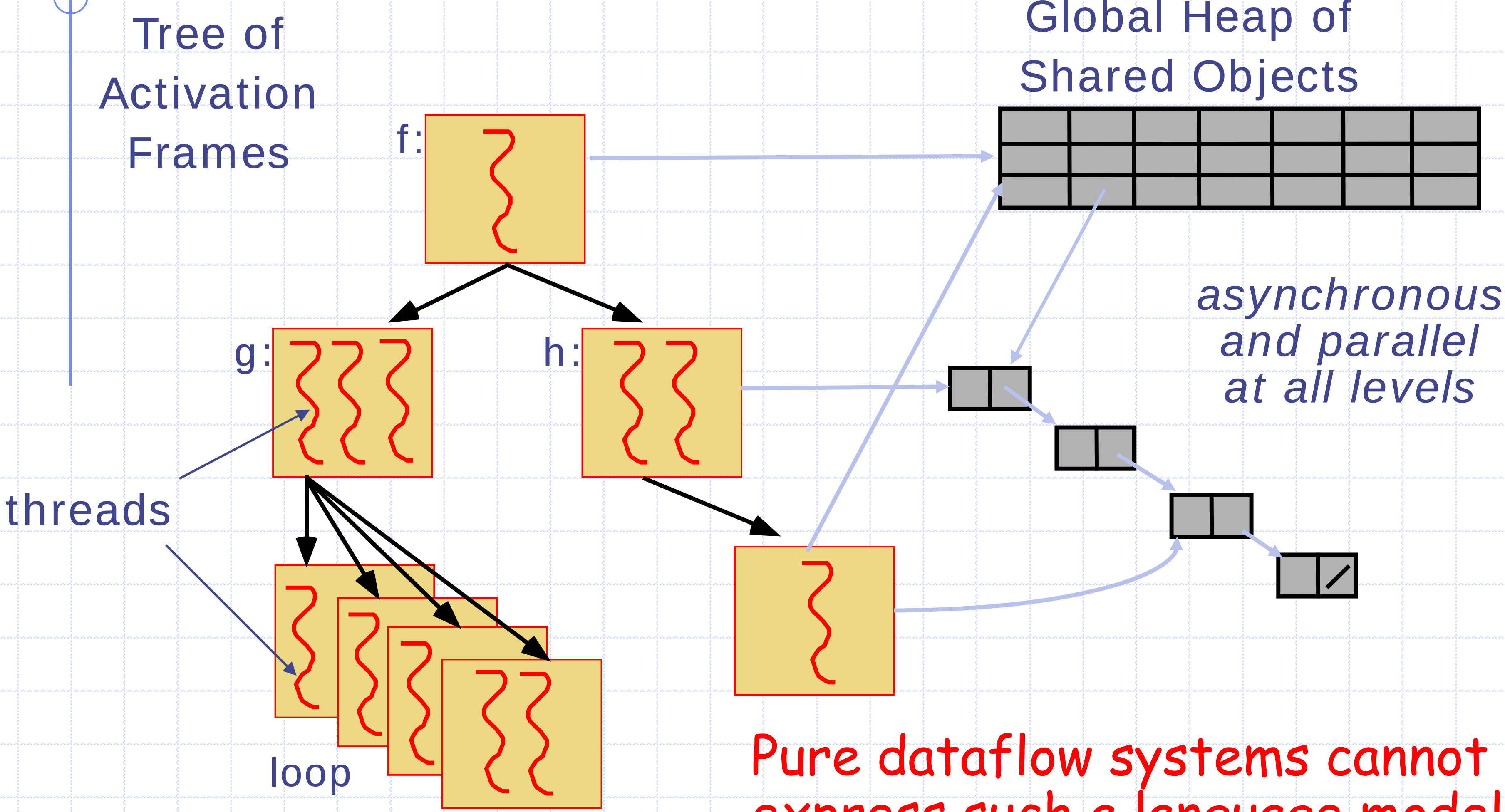
Data Structures

Data Structures

- Suppose we want to add two arrays, element by element
  - Option 1: the whole array is carried by a token (not practical in general)
  - Option 2: Array is represented as a stream of tokens
  - Option 3: Array is divided into several chunks and each chunk is fed to a different copy of the operator
  - Option 4: Some combination of options 2 and 3
- Such a choices of representation has to be made by the user or the compiler

These options are not suitable for handling complex or sparse data structures

A Parallel Multithreaded Language Model



Pure dataflow systems cannot express such a language model

Dynamic Dataflow Architectures and Languages (MIT 1979-94)

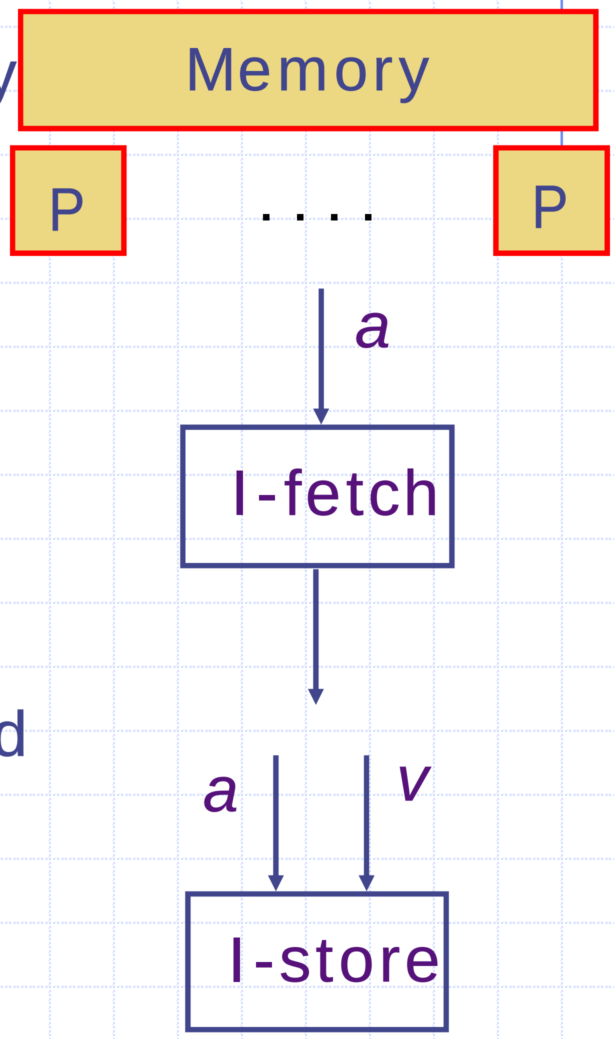
Extensions to the tagged dataflow model to support procedure calls and dynamically allocated heap objects

Data Structures in Dataflow

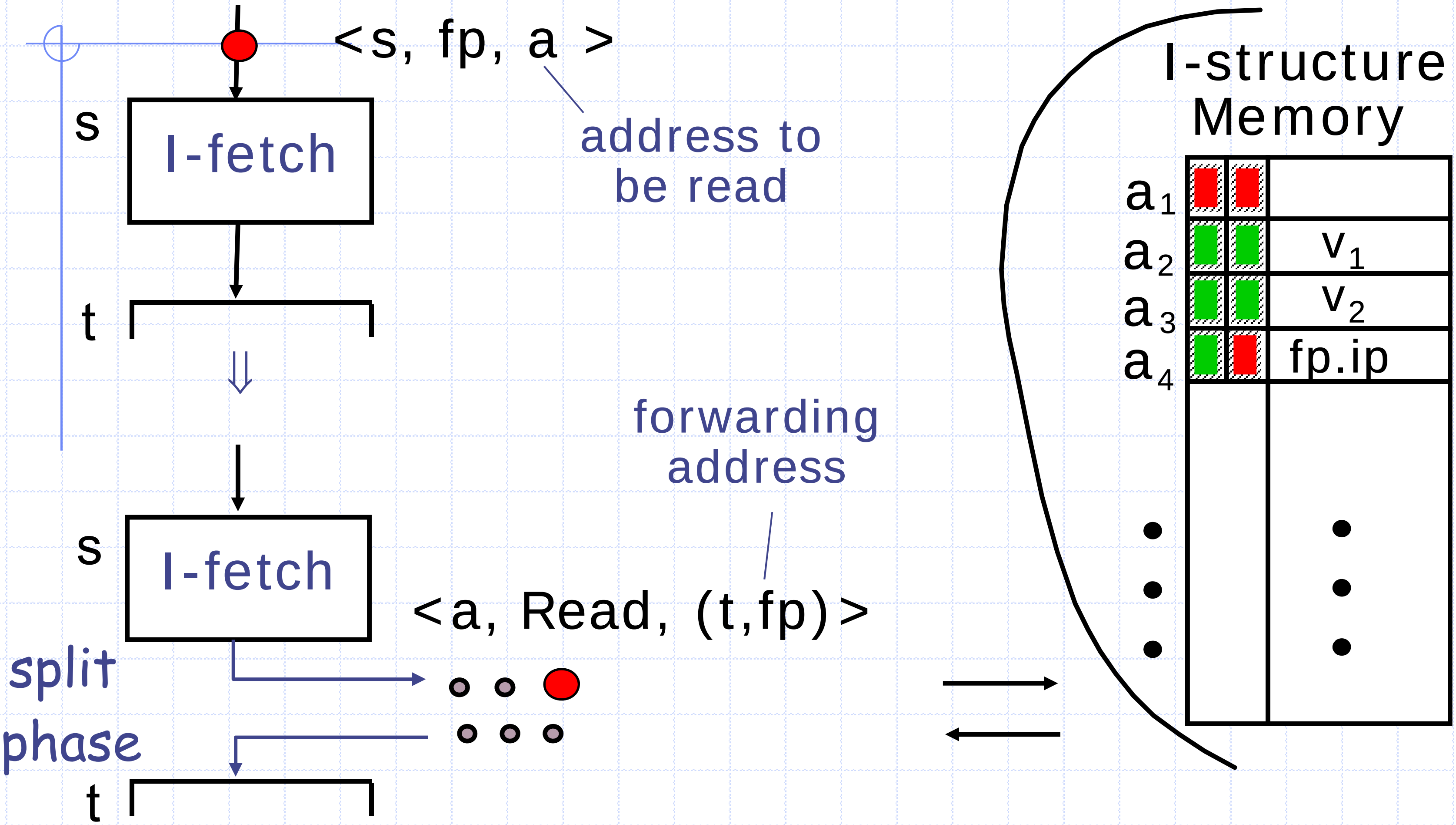
- Data structures reside in memory and tokens carry pointers
- I-structures
  - Write-once, Read multiple times

  allocate → write → read* → deallocate
- I-structures are like variables in logic languages (Pingali ...)
  - See I-vars, and M-vars in Haskell
- At the hardware level, I-fetch and I-store instructions

  No problem if a reader arrives before the writer at the memory location



I-Structure Storage:
Split-phase operations & Presence bits



Need to deal with multiple deferred reads;
Additional operations: take/put, clear

Language and Architecture to exploit *implicit parallelism*

Id (Functional language)

↓

Dataflow Graphs + I-Structures + . . .

↓

Monsoon Dataflow Machine

The Monsoon Project

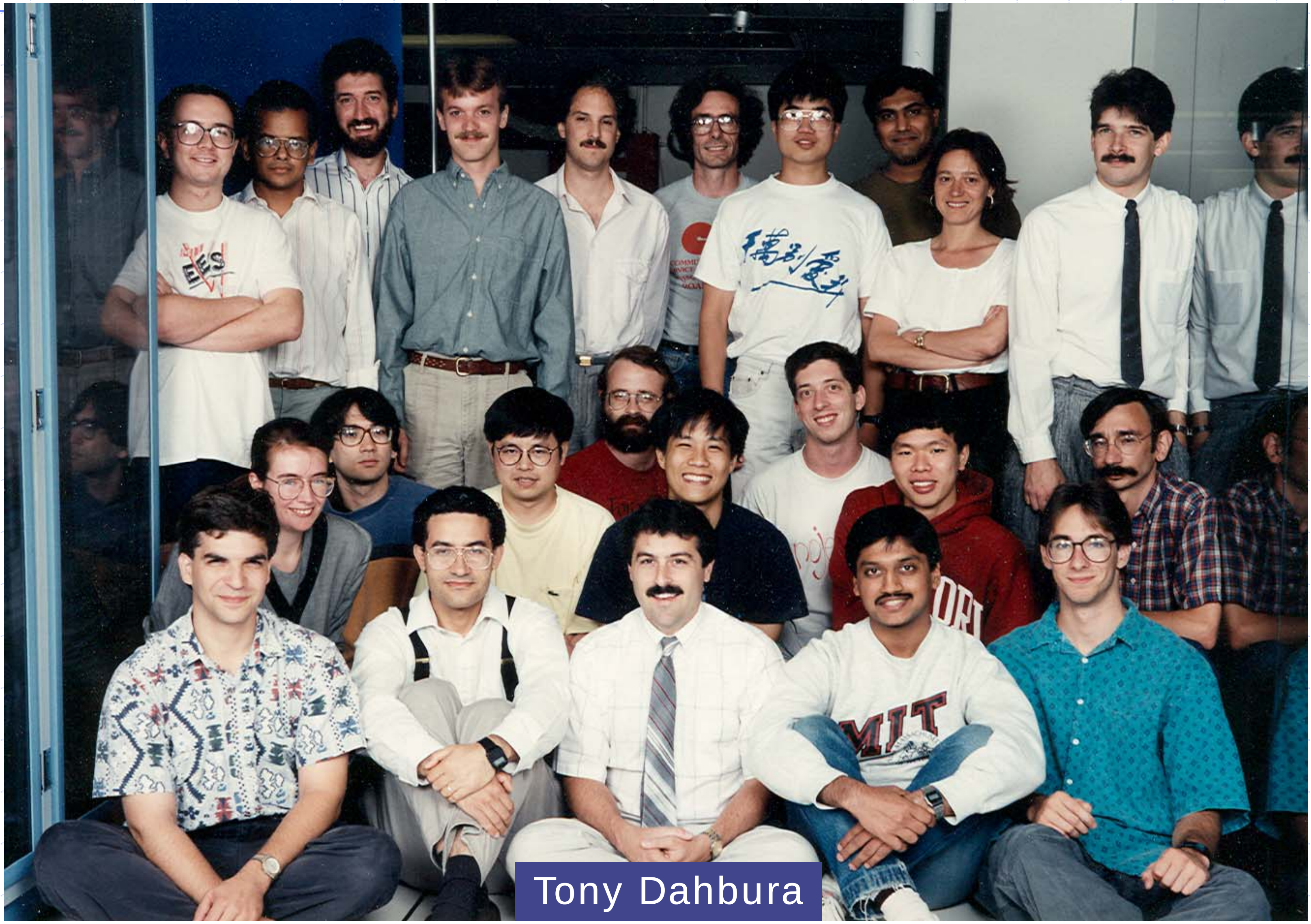
Motorola Cambridge Research Center + MIT (1988-91)

- Two 16-node systems
  - MIT, LANL
- Sixteen 2-node systems
  - Motorola, Colorado, USC, Oregon, McGill, …
- Id World Software

Displayed at Super Computing 1991

The Monsoon Project

Motorola Cambridge Research Center + MIT (1988-91)

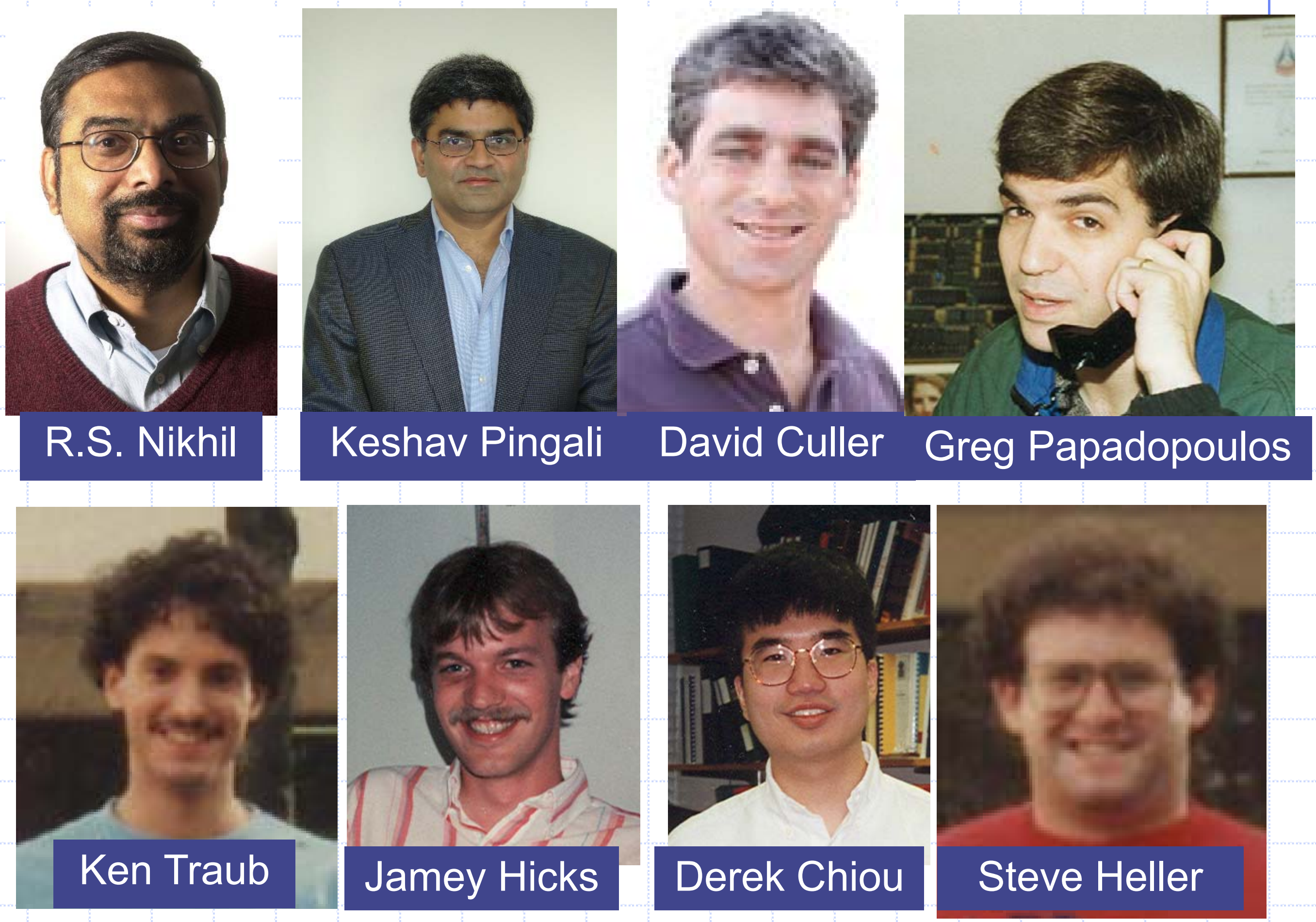


Tony Dahbura

Dataflow people @ MIT

- Rishiyur Nikhil,
- Keshav Pingali,
- Vinod Kathail,
- David Culler
- Greg Papadopolous
- Andy Boughton
- Chris Jeorge
- Ken Traub
- Steve Heller,
- Richard Soley,
- Dinart Mores
- Jamey Hicks,
- Alex Caro,
- Andy Shaw,
- Boon Ang
- Shail Anditya
- R Paul Johnson
- Jacj Costenza
- Paul Barth
- Jan Maessen
- Christine Flood
- Jonathan Young
- Derek Chiou
- Arun Iyangar
- Zena Ariola
- Mike Bekerle

- K. Eknadham (IBM)
- Wim Bohm (Colorado)
- Joe Stoy (Oxford)
- ...



Dataflow: An assessment

- Dataflow ideas are used extensively in modern microprocessors, but the Instructure Set Architecture (ISA) remains totally sequential
  - Processors use *speculation* to expose parallelism
- Compilers extract parallelism from sequential codes with varying degree of success
- Parallel programming remains difficult – *most* users have little interest in rewriting their programs to exploit parallelism
  - Ethos: *It is someone else's program*
- Yet, most microprocessors are multicores and most systems have many concurrent processes

Move towards domain specific solutions

Fine-grain vs Coarse-grain Dataflow Models

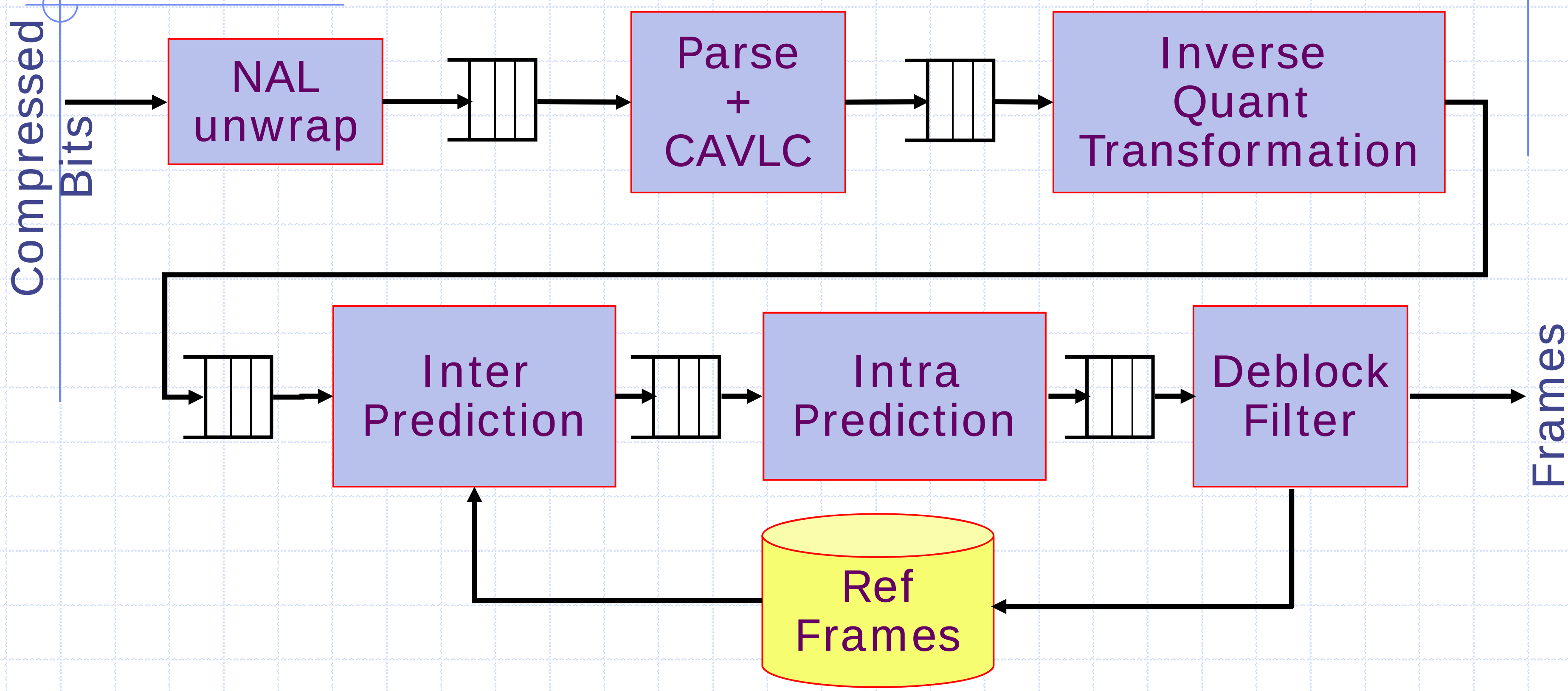
- Dataflow models say nothing about the *size of tokens* or the *complexity of computing nodes*
- In practice, *the cost of implementing fine-grain asynchronous parallelism is high*, so many systems use dataflow ideas at the macro-level and exploit fine-grain synchronous parallelism within the nodes

Applications

- Dataflow models are used extensively in many domains, especially for *hardware accelerators:*
  - Video and Audio codecs (e.g., H.264)
  - Network applications (e.g., packet filtering)
  - Wireless baseband processing (e.g., OFDM based protocols)
  - Deep Neural Networks (e.g., DNNs, RNNs)
  - Map-Reduce applications
  - …

H.264 Video Decoder

Chun-Chieh Lin, K Elliott Fleming [MEMOCODE 2008]



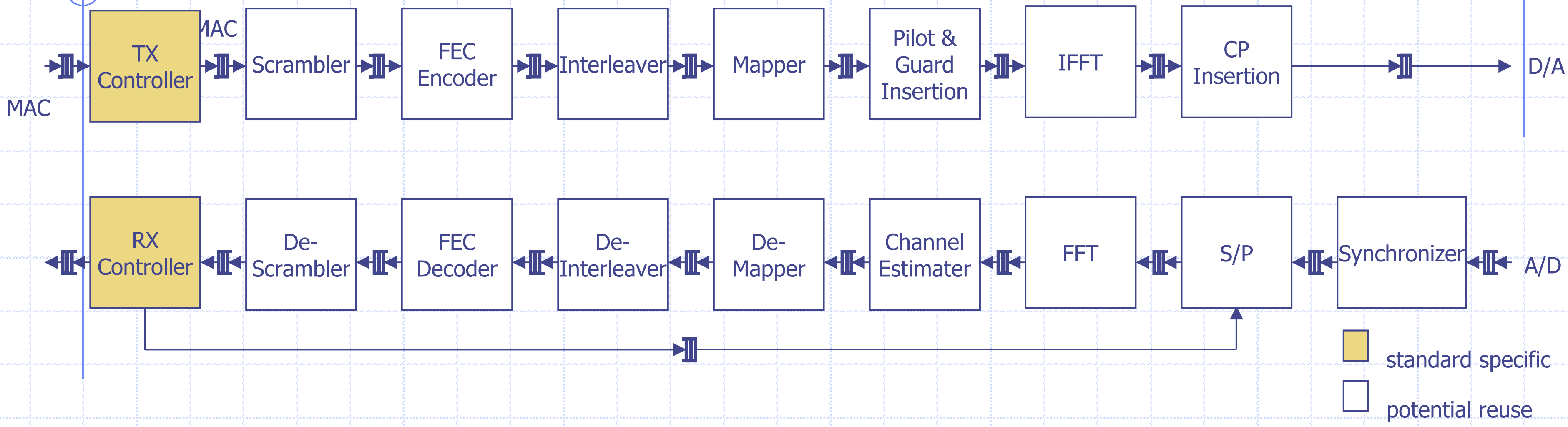
Different requirements for different environments

- QVGA 320x240p (30 fps)
- DVD 720x480p
- HD DVD 1280x720p (60-75 fps)

May be implemented in hardware or software depending upon ...

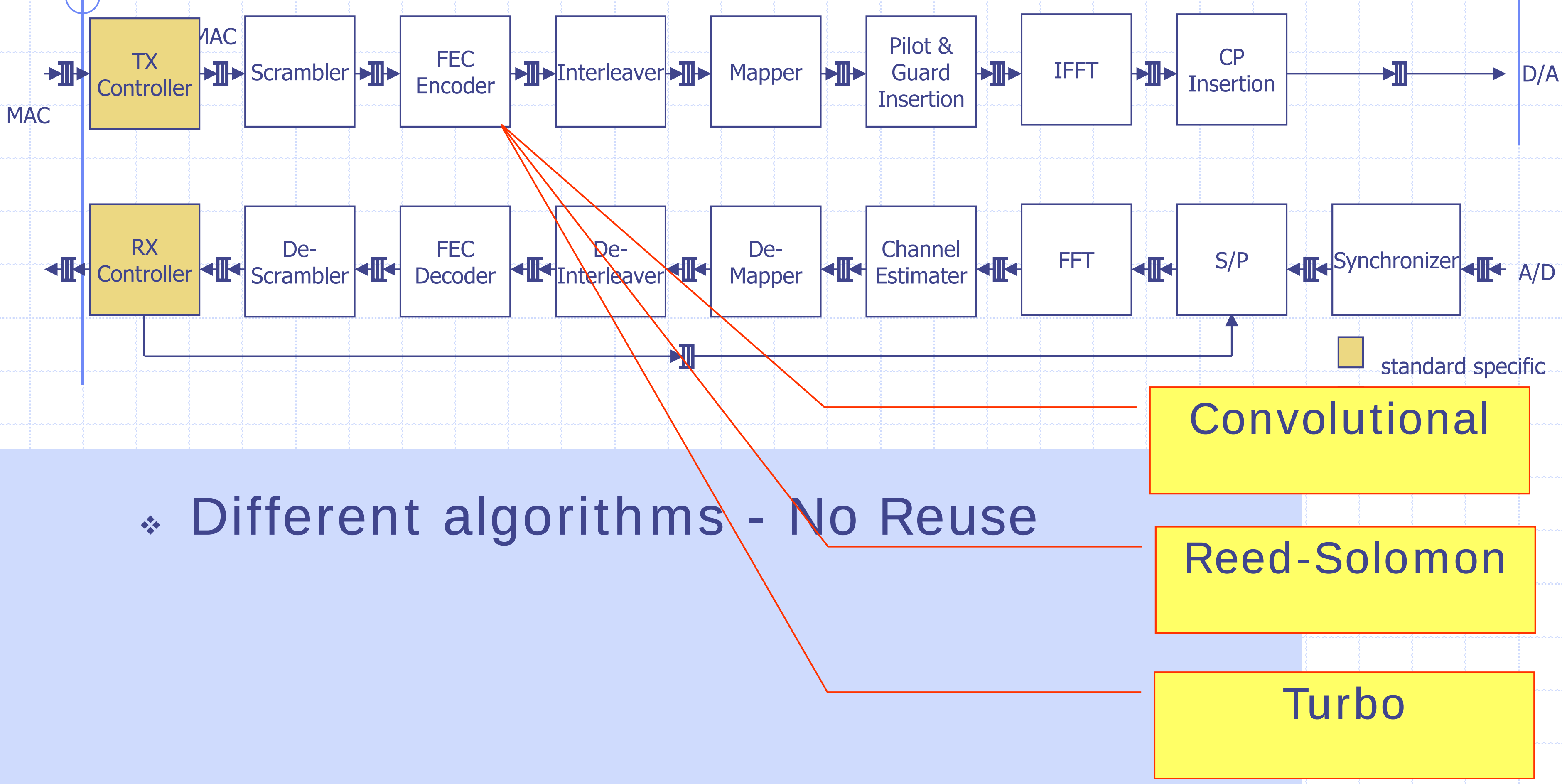
Wireless Baseband processing

OFDM TX/RX Blocks, Man cheuk Ng *et al* [2007]



Wireless Baseband processing

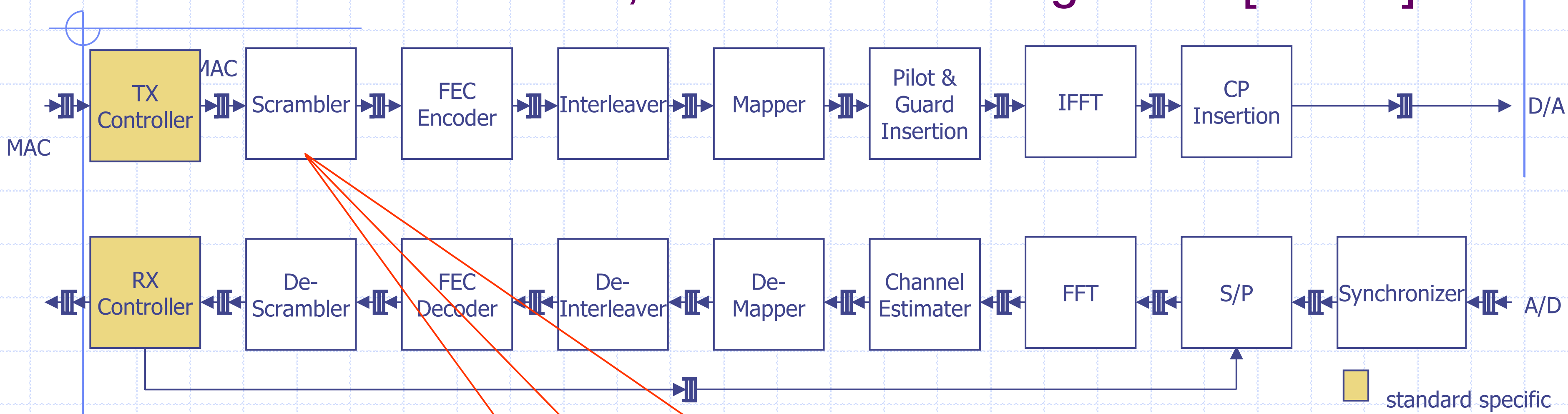
OFDM TX/RX Blocks, Man cheuk Ng *et al* [2007]



- Different algorithms - No Reuse

Wireless Baseband processing

OFDM TX/RX Blocks, Man cheuk Ng *et al* [2007]



WiFi: $x^7 + x^4 + 1$

WiMAX: $x^{15} + x^{14} + 1$

WUSB: $x^{15} + x^{14} + 1$

- Different algorithms - No Reuse
- Reusable algorithm with different parameter settings

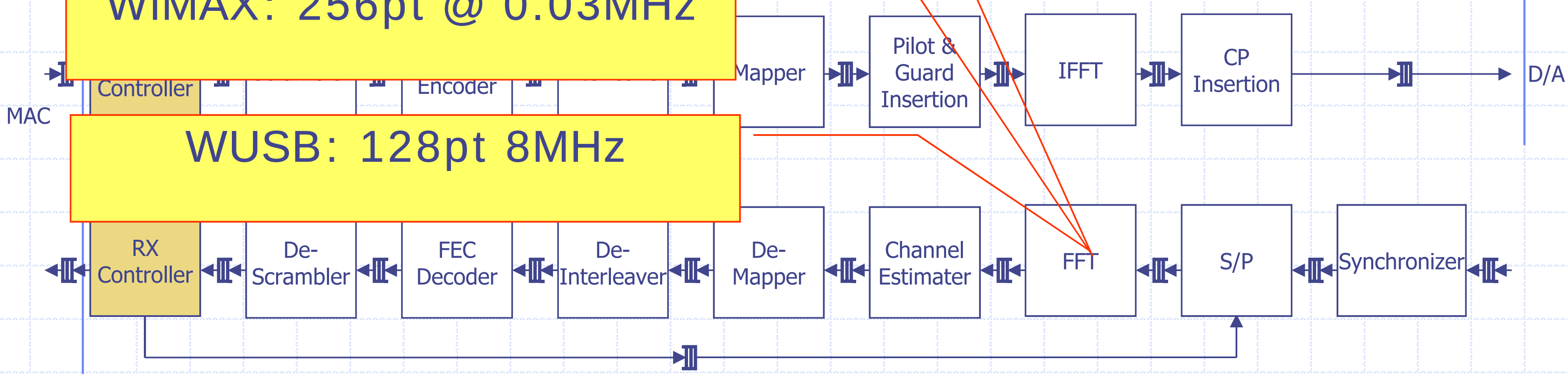
Wireless Baseband processing

OFDM TX/RX Blocks; Man cheuk Ng *et al* [2007]

WiFi: 64pt @ 0.25MHz

WiMAX: 256pt @ 0.03MHz

WUSB: 128pt 8MHz



- Different algorithms - No Reuse
- Reusable algorithm with different parameter settings
- Different throughput requirements

We want the same source description but different hardware!

There is a need to beyond dataflow models

- Dataflow models, like functional languages, obscure resource management issues which must be handled efficiently in any practical system
- Lack of shared state between nodes makes certain computations, e.g., graph algorithms, difficult to express
- *Determinacy*, though highly desirable in many situations, is ultimately quite limiting in *specifying* distributed software and hardware

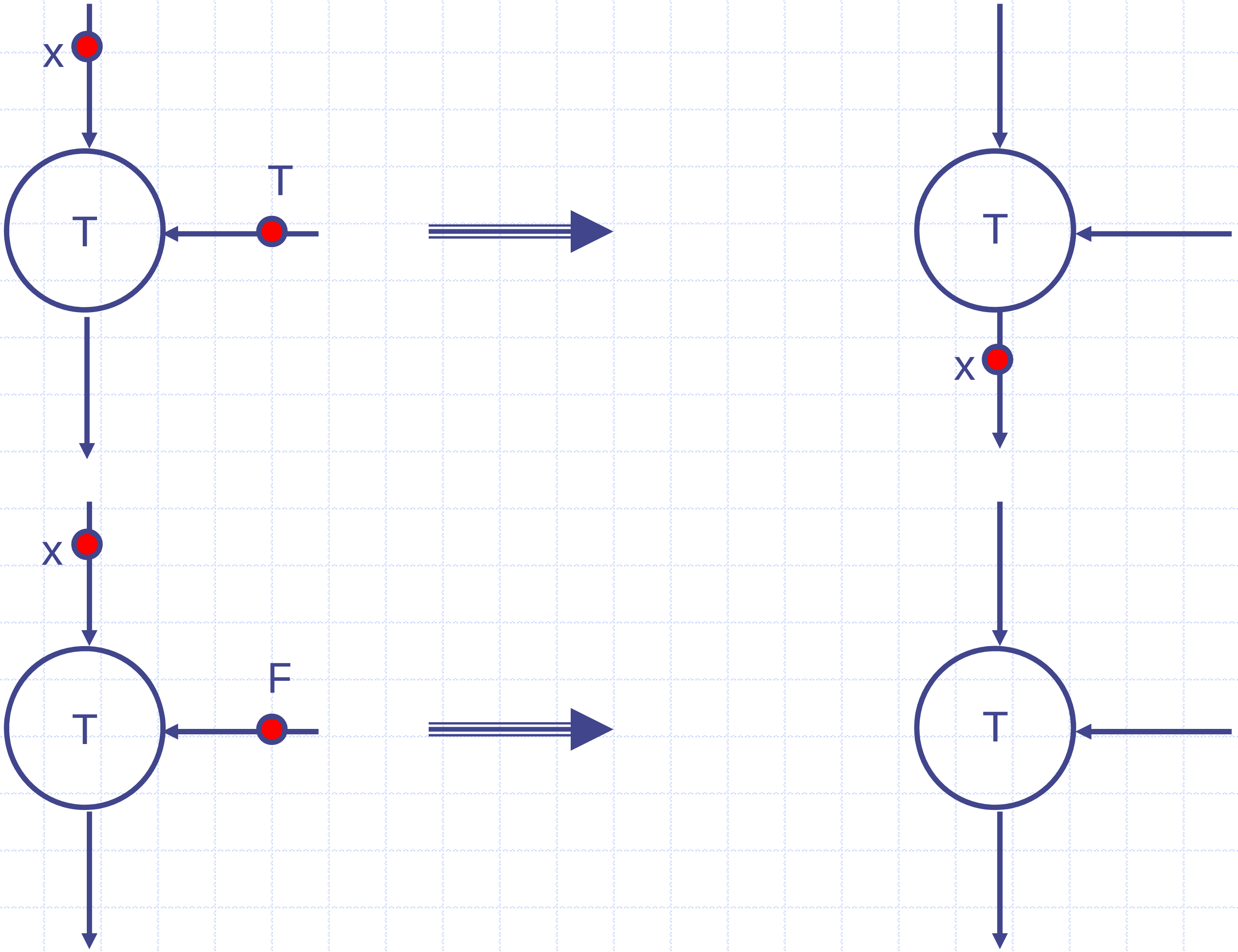
Alternative

- Serializable atomic actions
  - A general model of concurrency for both software and hardware
  - Can express most other models of concurrency
  - A specific model – *guarded atomic actions* – is also amenable to efficient hardware implementation

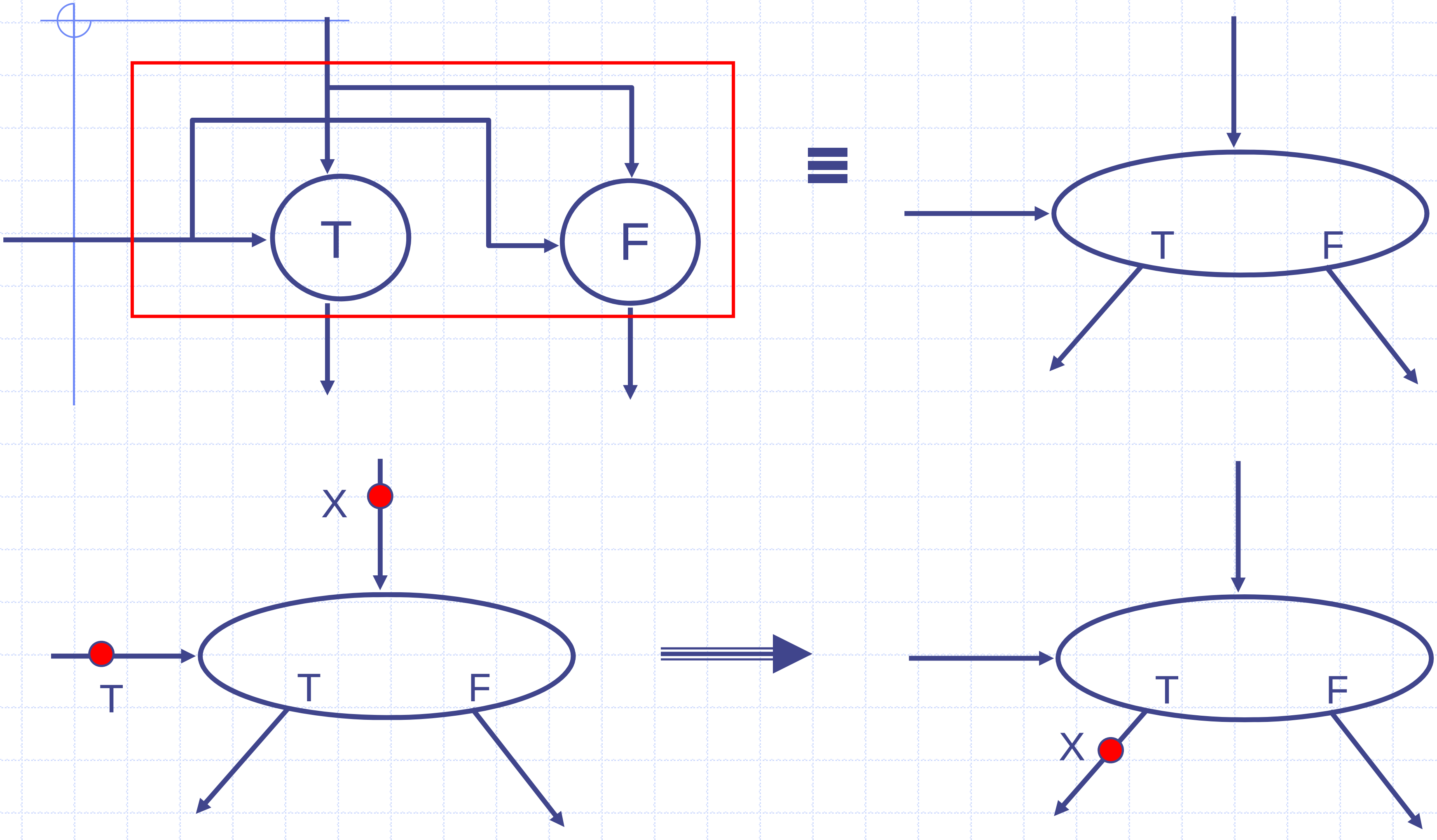
stay tuned ...

Extras

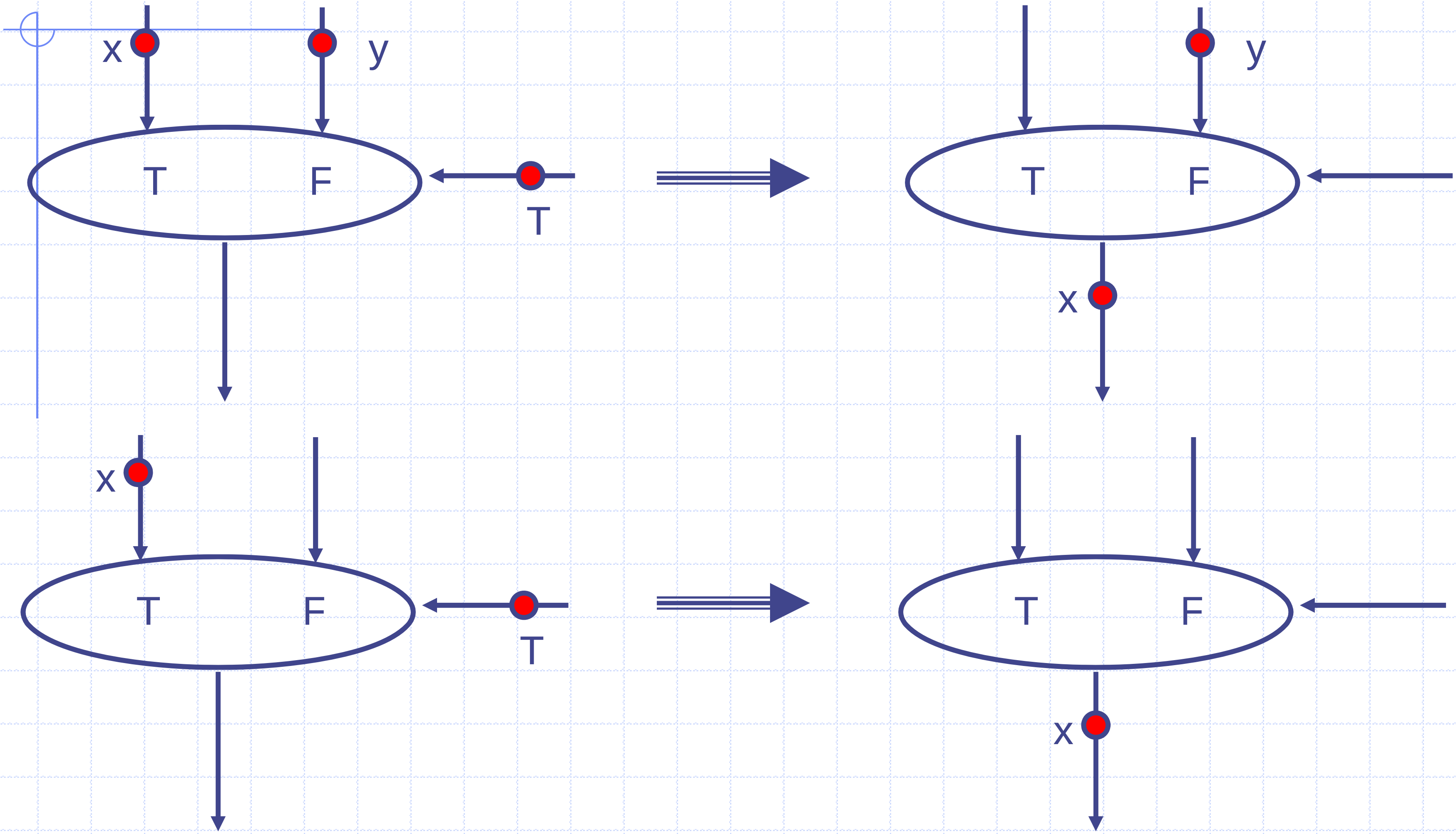
Firing Rules: T-Gate



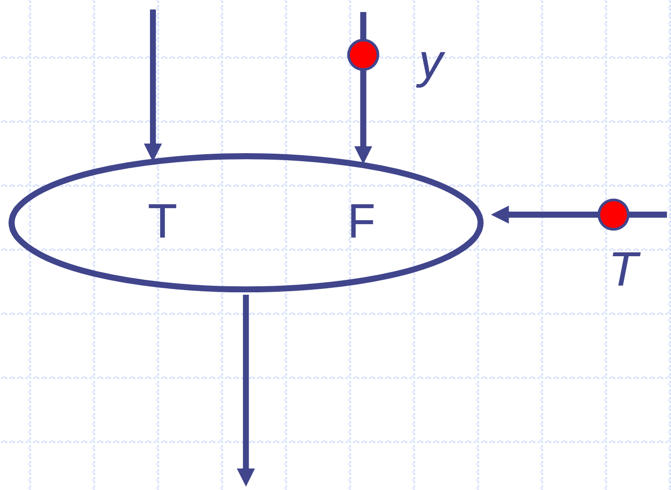
The Switch Operator



Firing Rules: Merge

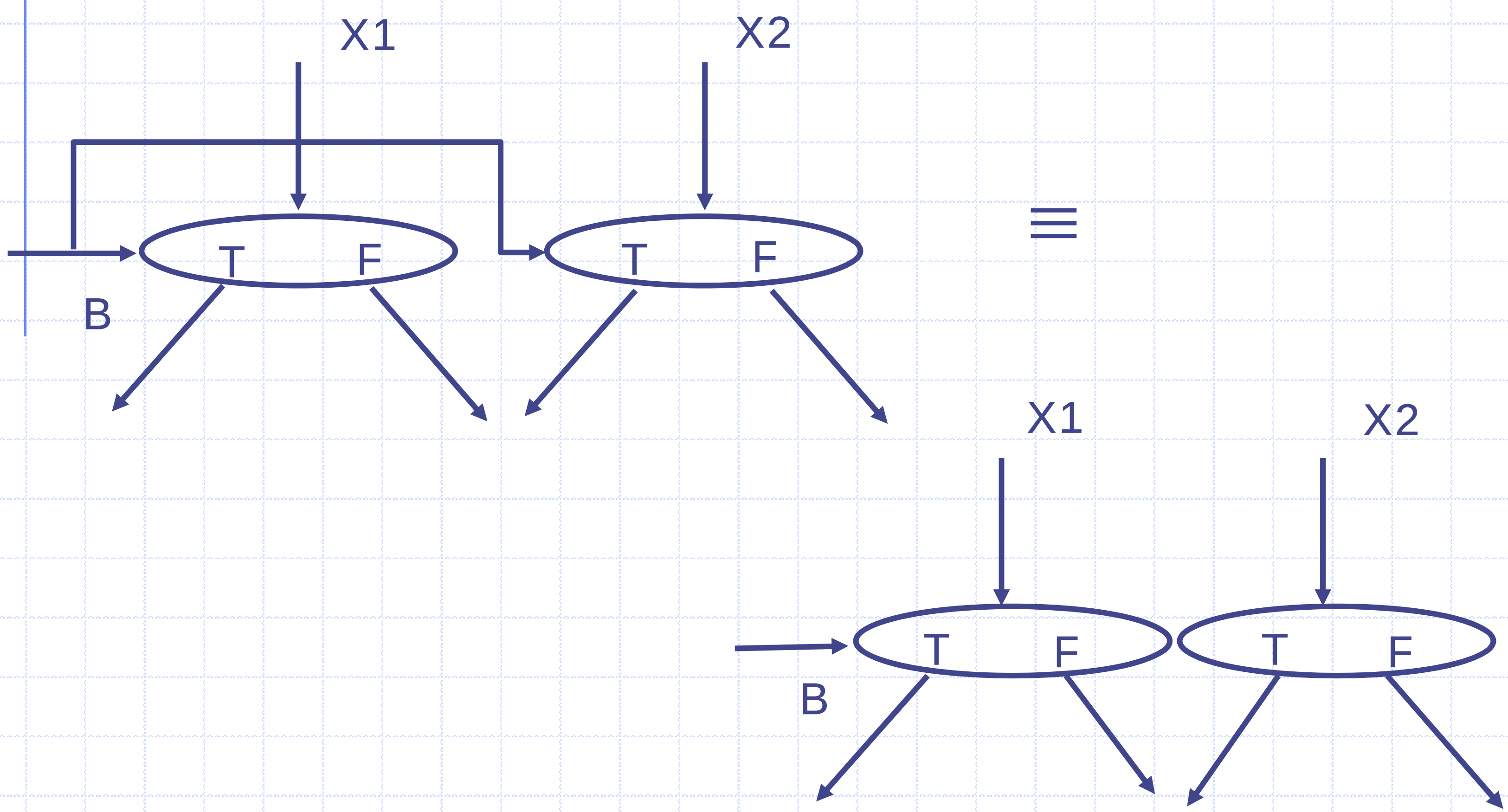


Firing Rules: Merge *cont*

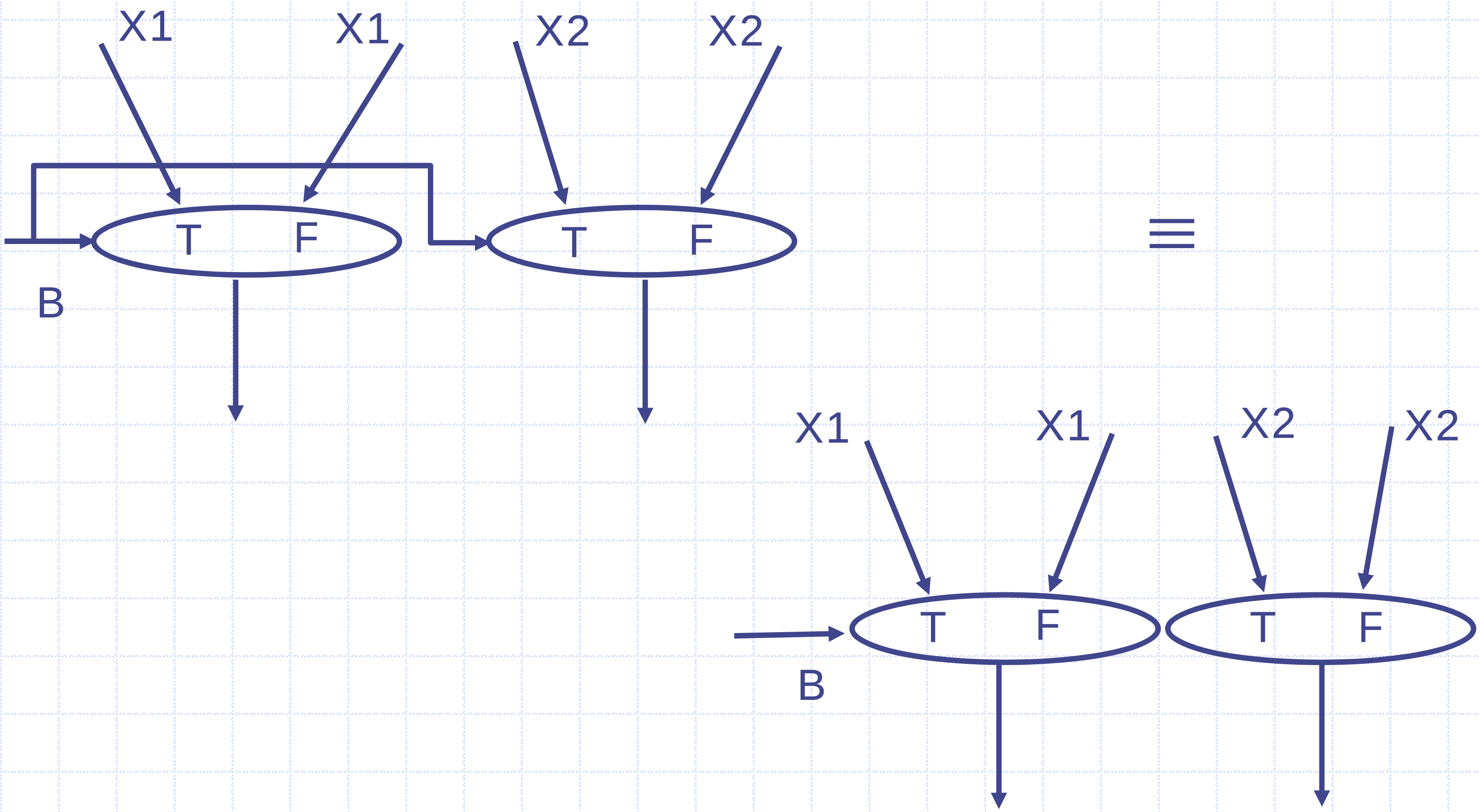


not ready
to fire

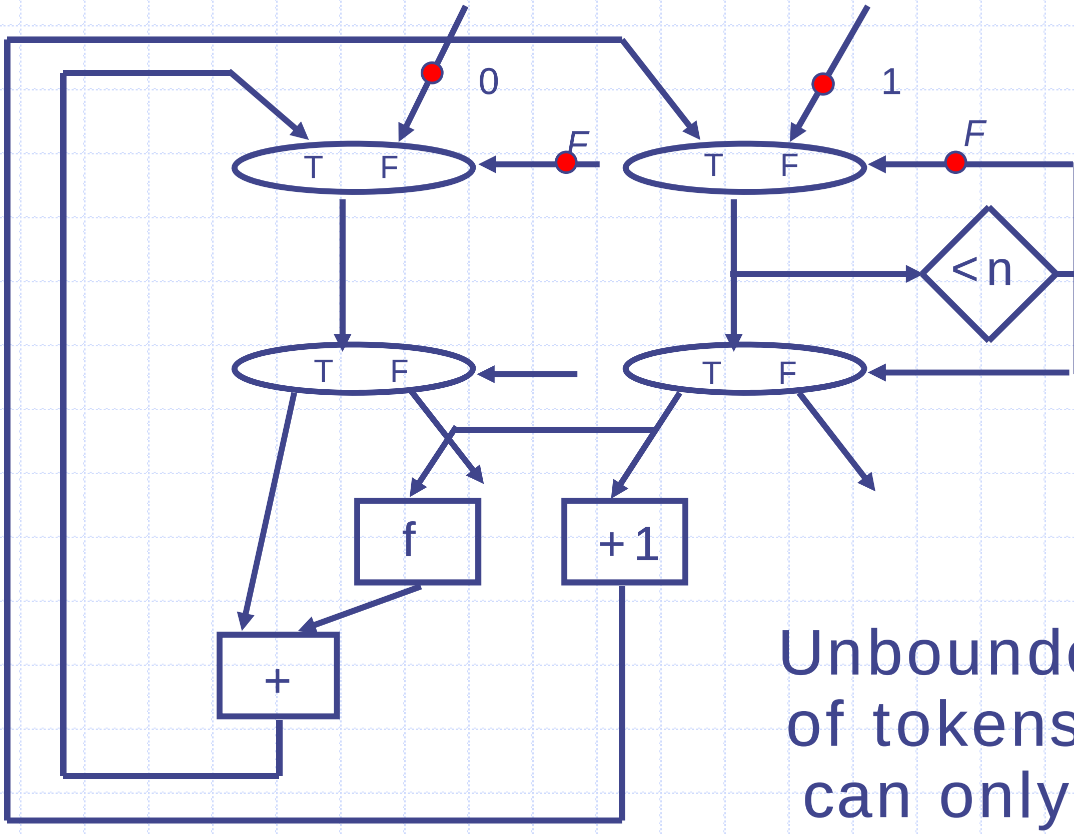
Some Conventions



Some Conventions *Cont.*

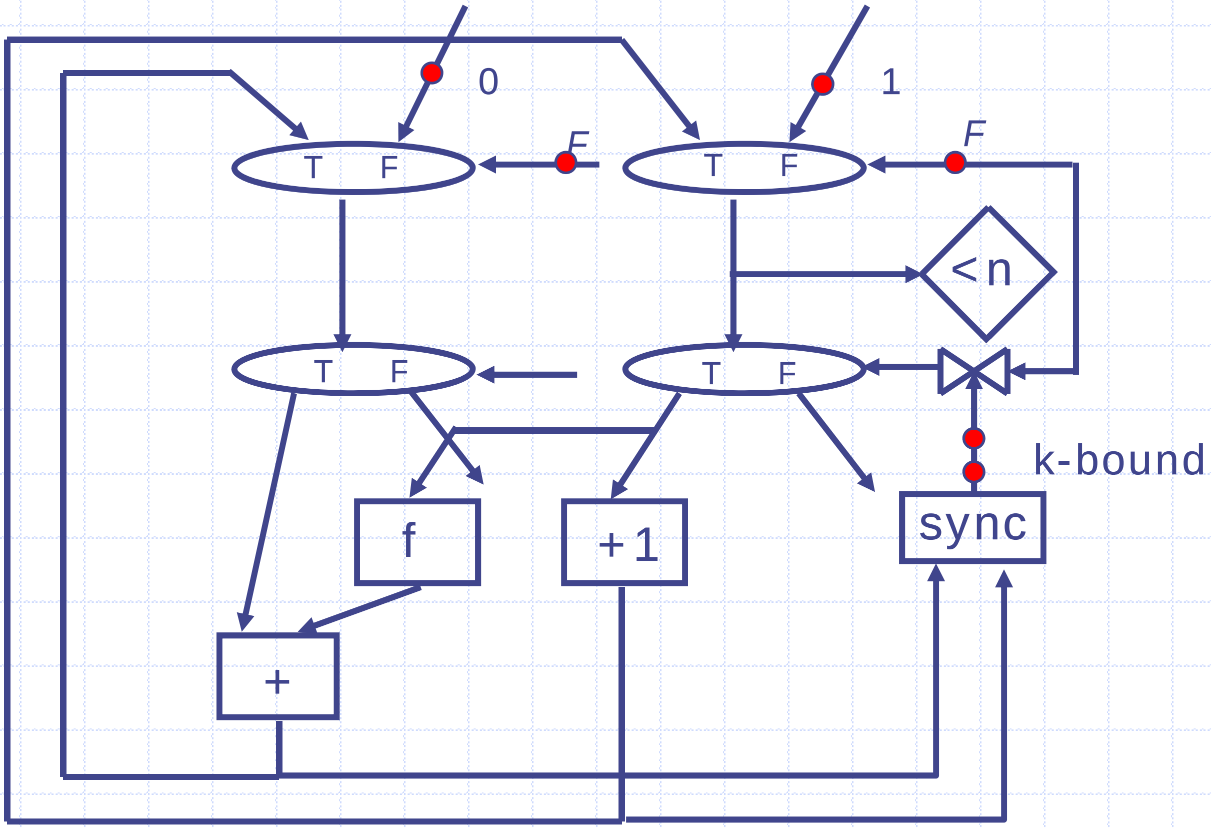


Unbounded Cyclic Graphs

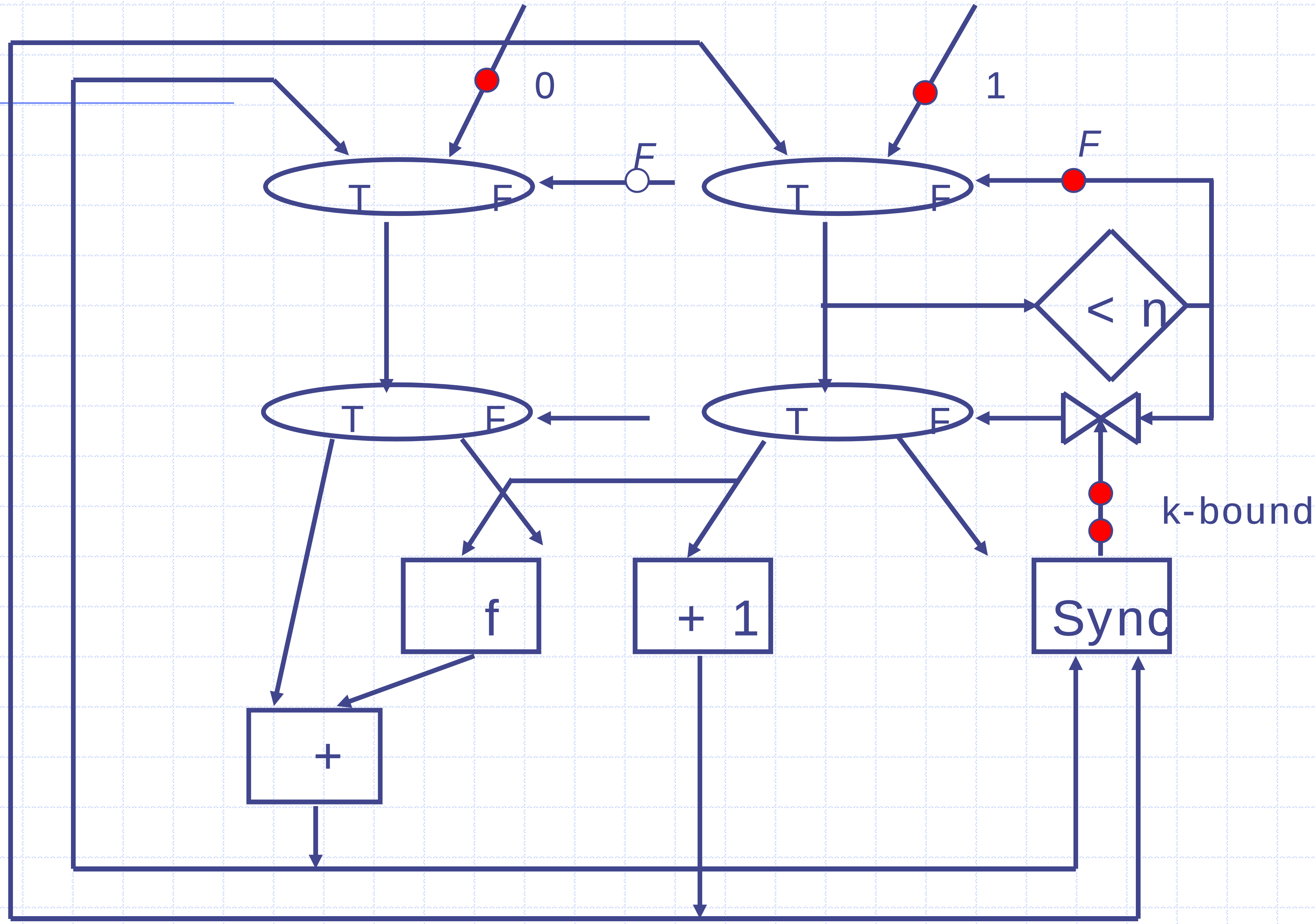


Unbounded number of tokens on an arc can only arise due to cycles.

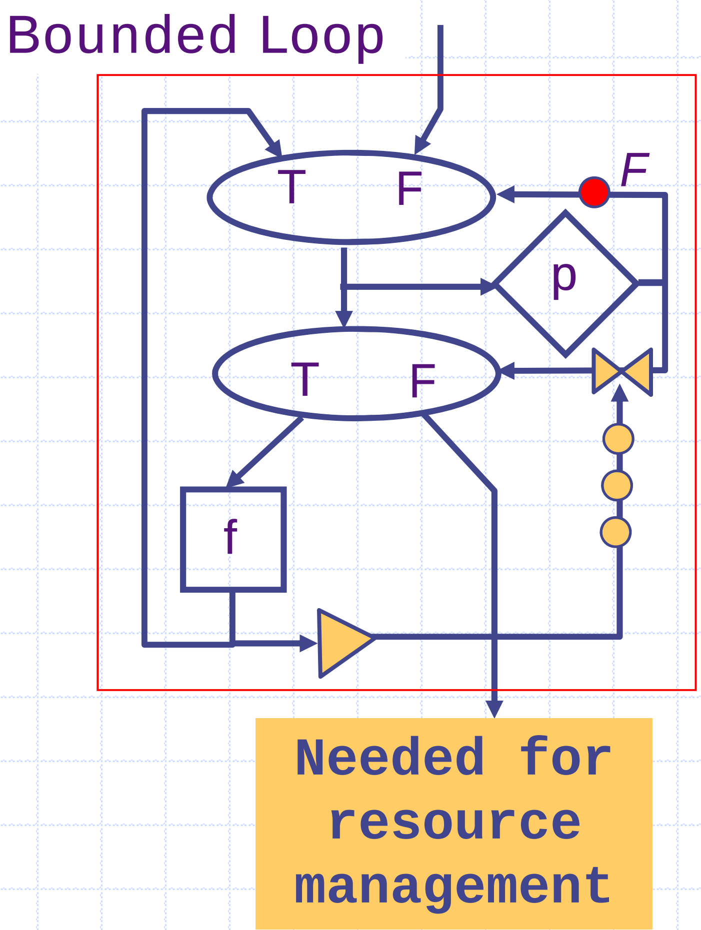
Bounded Cyclic Graphs



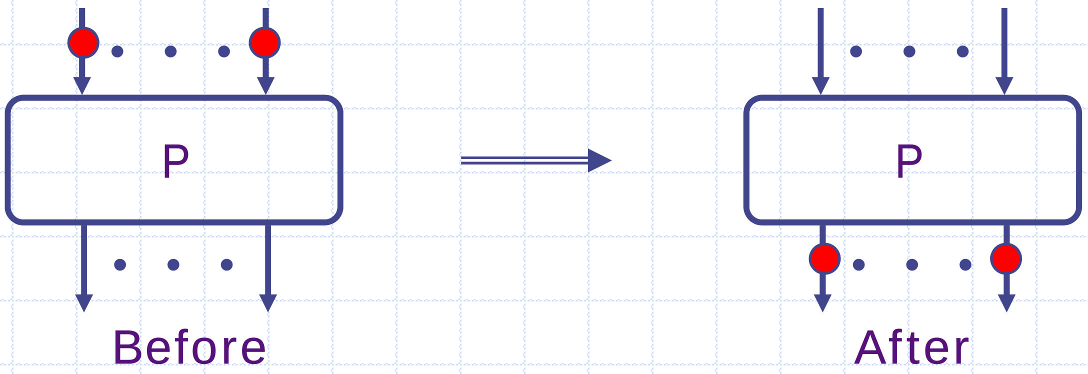
Bounded Cyclic Graphs



Well Behaved Schemas



New Definition of *Well Behavedness*



1. One token on each input arc produces exactly one token on each output arc.
2. The initial distribution of tokens on the arcs is restored.
3. No arc can have an unbounded buildup of tokens.

Bounded Cyclic Graphs are *Well Behaved*

- Initial number of tokens at the *gate* input determines the maximum number of tokens on any arc.
- However, loop bounding can alter the "meaning" of a graph, i.e., can cause *deadlock.*
- In general, restricting the number of tokens on an arc causes deadlock.

Can this program deadlock if the number of tokens per arc is restricted?