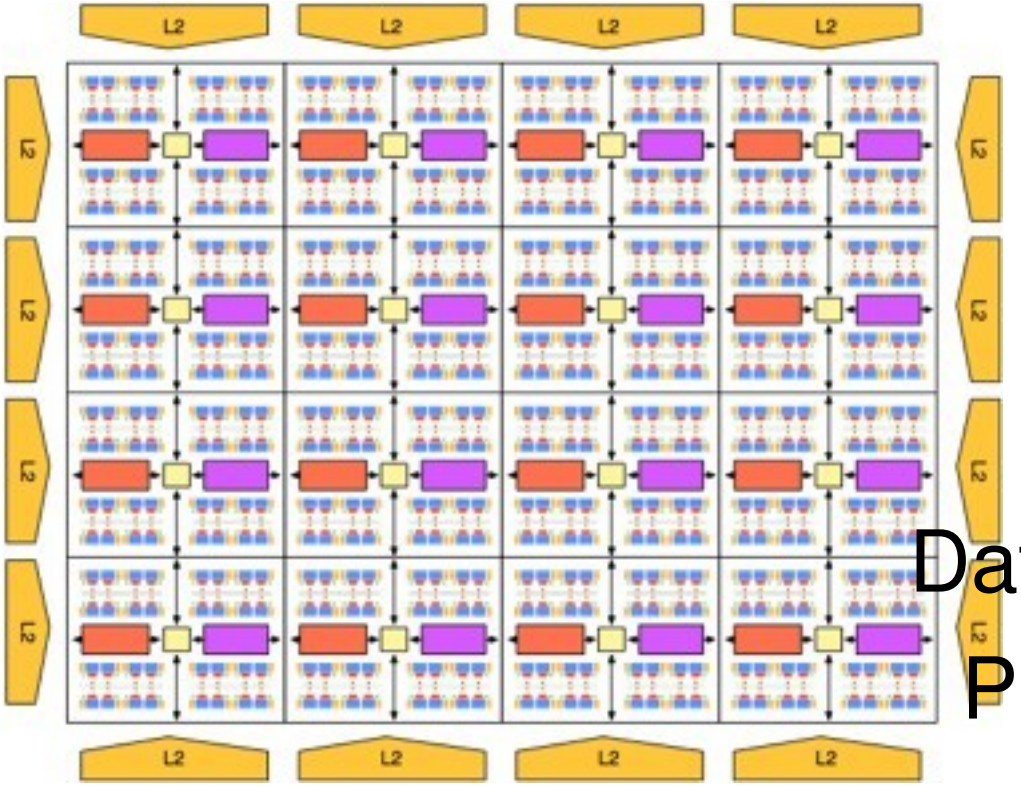
High Performance Architectures



Dataflow

Part 3

Dataflow Processors

- Recall from Basic Processor Pipelining:
- Hazards limit performance
  - Structural hazards
  - Data hazards due to
    - true dependences or
    - name (false) dependences: anti and output dependences
  - Control hazards
- **Name dependences** can be removed by:
  - compiler (register) renaming
  - renaming hardware $\rightarrow$ advanced superscalars
  - single-assignment rule $\rightarrow$ dataflow computers
- Data hazards due to **true dependences** and **control hazards** can be avoided if succeeding instructions in the pipeline stem from different contexts
  $\rightarrow$ dataflow computers, multithreaded processors

Dataflow Model of Computation

- *Enabling rule*:

  An instruction is enabled (i.e. executable) if all operands are available.

  - Von Neumann model: an instruction is enabled if it is pointed to by PC.

- The *computational rule* or *firing rule*, specifies when an enabled instruction is actually executed.

- *Basic instruction firing rule*:

  An instruction is fired (i.e. executed) when it becomes enabled.

  - The effect of firing an instruction is the consumption of its input data (operands) and generation of output data (results)

    Where are the structural hazards?

Dataflow languages

- Main characteristic: The *single-assignment rule*

  A variable may appear on the left side of an assignment only once within the area of the program in which it is active.

- Examples: VAL, Id, LUCID

- A dataflow program is compiled into a dataflow graph which is a directed graph consisting of named nodes, which represent instructions, and arcs, which represent data dependences among instructions
  - The dataflow graph is similar to a dependence graph used in intermediate representations of compilers.
- During the execution of the program, data propagate along the arcs in data packets, called *tokens*
- This flow of tokens enables some of the nodes (instructions) and fires them

Dataflow Architectures - Overview

- Pure dataflow computers:
  - static,
  - dynamic,
  - and the explicit token store architecture.
- Hybrid dataflow computers:
  - Augmenting the dataflow computation model with control-flow mechanisms, such as
    - RISC approach,
    - complex machine operations,
    - multi-threading,
    - large-grain computation,
    - etc.

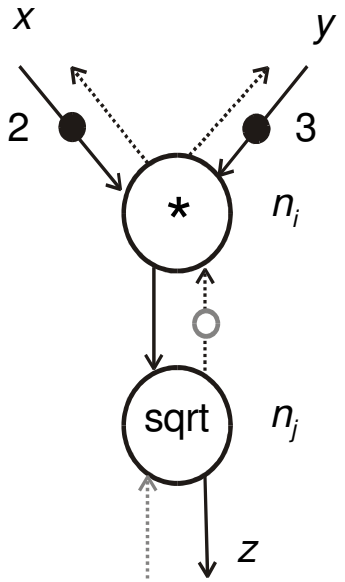
Pure Dataflow

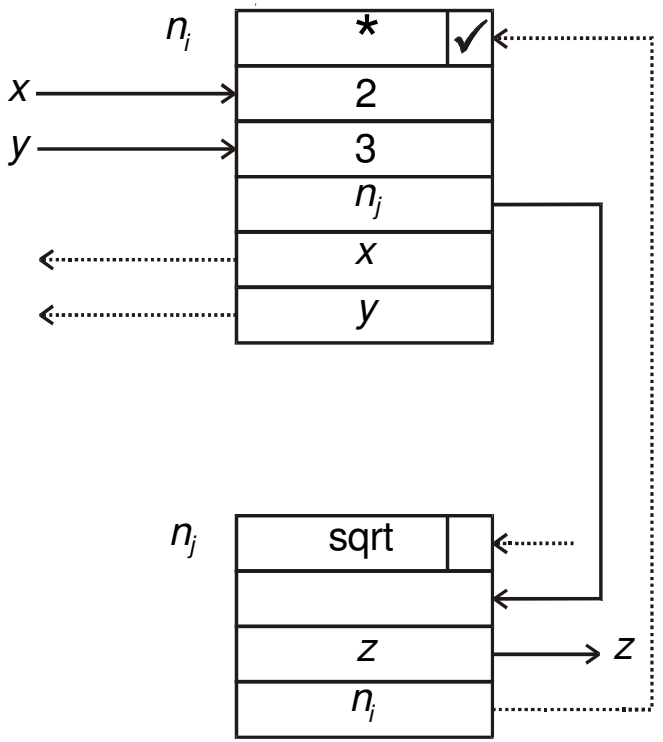
- A *dataflow computer* executes a program by receiving, processing and sending out *tokens*, each containing some *data* and a *tag*.
- Dependences between instructions are translated into *tag matching* and *tag transformation*.
- Processing starts when a set of *matched tokens* arrives at the execution unit.
- The instruction which has to be fetched from the instruction store (according to the tag information) contains information about
  - what to do with data
  - and how to transform the tags
- The *matching unit* and the execution unit are connected by an asynchronous pipeline, with queues added between the stages
- Some form of associative memory is required to support token matching
  - a real memory with associative access,
  - a simulated memory based on hashing,
  - or a direct matched memory

Static Dataflow

- A *dataflow graph* is represented as a collection of *activity templates*,
  each containing:
  - the opcode of the represented instruction,
  - operand slots for holding operand values,
  - and destination address fields, referring to the operand slots in sub-sequent activity templates that need to receive the result value.
- Each *token* consists only of a value and a destination address.

Dataflow graph and Activity template





data token

acknowledge signal

data arc

acknowledgement arc

Acknowledgement signals

- Notice, that different tokens destined for the same destination cannot be distinguished.
- Static dataflow approach allows at most one token on any one arc.

- *Extending the basic firing rule* as follows:

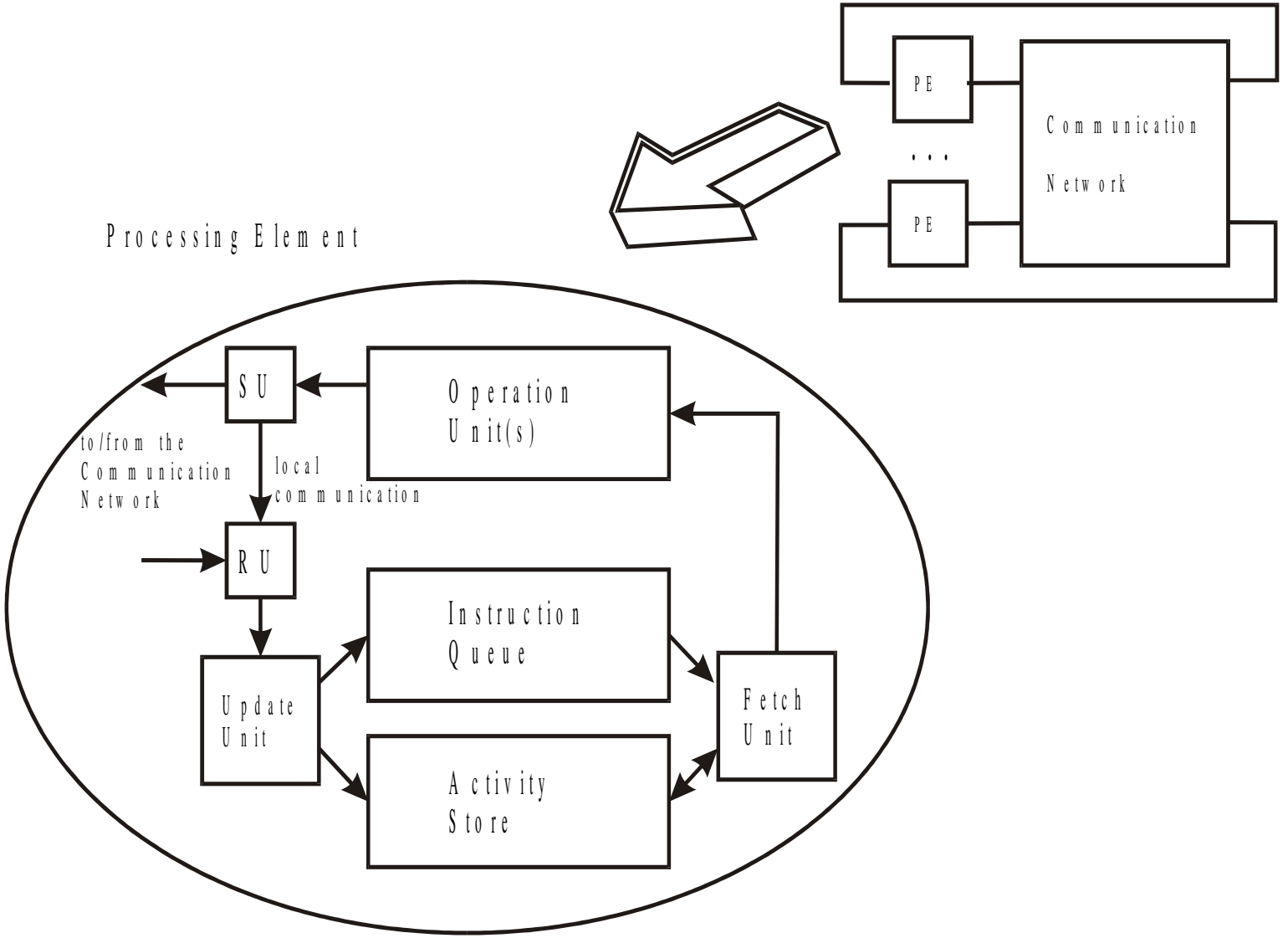
  An enabled node is fired if there is no token on any of its output arcs.

- Implementation of the restriction by *acknowledge signals* (additional tokens), traveling along additional arcs from consuming to producing nodes.
- Using acknowledgement signals, the *firing rule* can be changed to its original form:

  A node is fired at the moment when it becomes enabled.

- Again: structural hazards are ignored assuming unlimited resources!

MIT Static Dataflow Machine



Deficiencies of Static Dataflow

- Consecutive iterations of a loop can only be pipelined.
- Due to acknowledgment tokens, the token traffic is doubled.
- Lack of support for programming constructs that are essential to modern programming language
  - no procedure calls,
  - no recursion.

- Advantage: simple model

Dynamic Dataflow

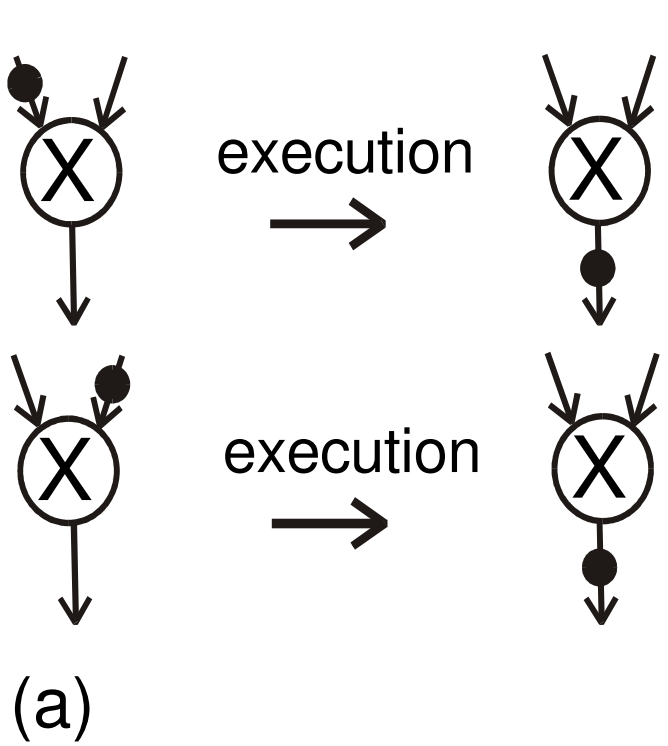
- Each loop iteration or subprogram invocation should be able to execute in parallel as a separate instance of a reentrant subgraph.
- The replication is only conceptual.
- Each *token* has a *tag*:
  - address of the instruction for which the particular data value is destined
  - and context information
- Each arc can be viewed as a bag that may contain an arbitrary number of tokens with different tags.

- The *enabling and firing rule* is now:

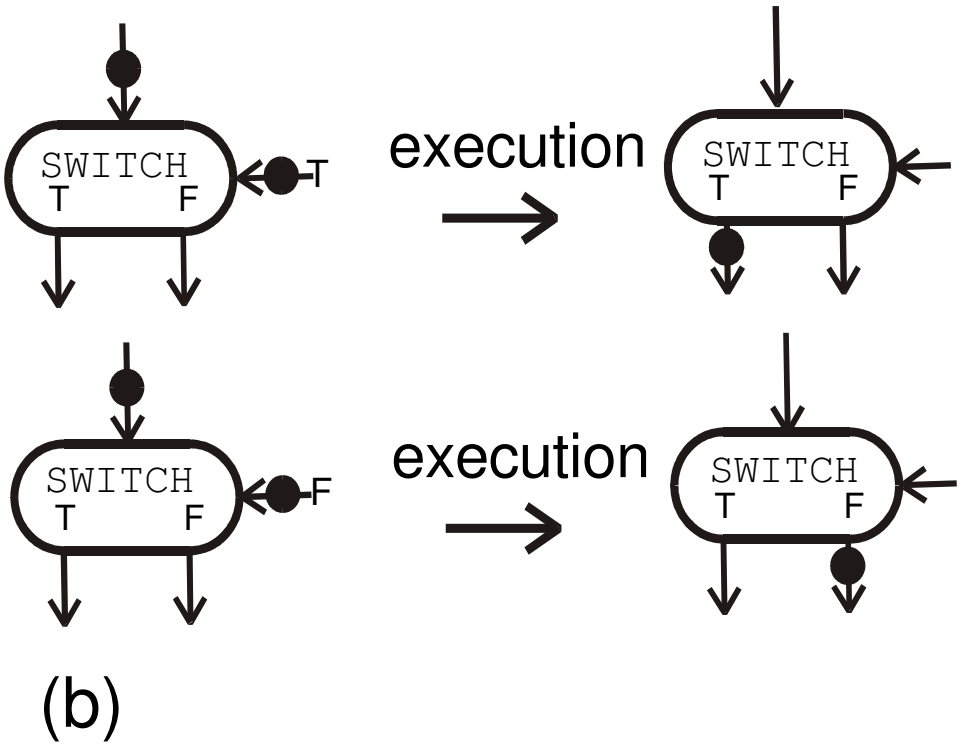
  A node is enabled and fired as soon as tokens with identical tags are present on all input arcs.

- Structural hazards ignored!

MERGE and SWITCH nodes

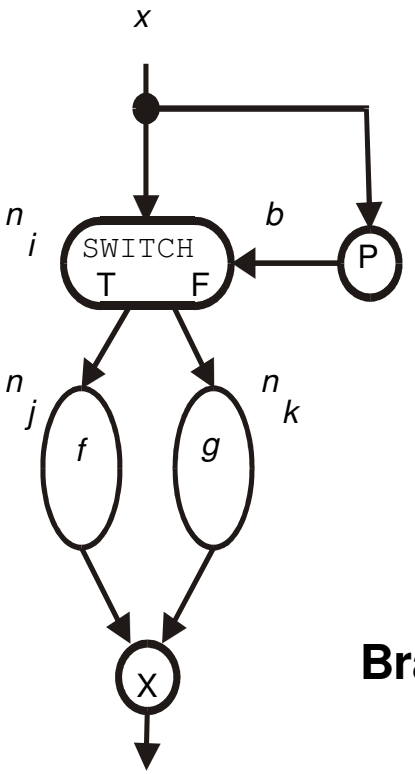


(a)

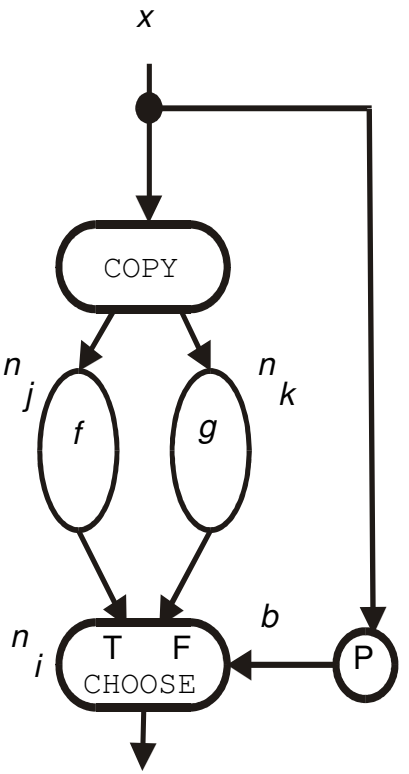


(b)

Branch Implementations

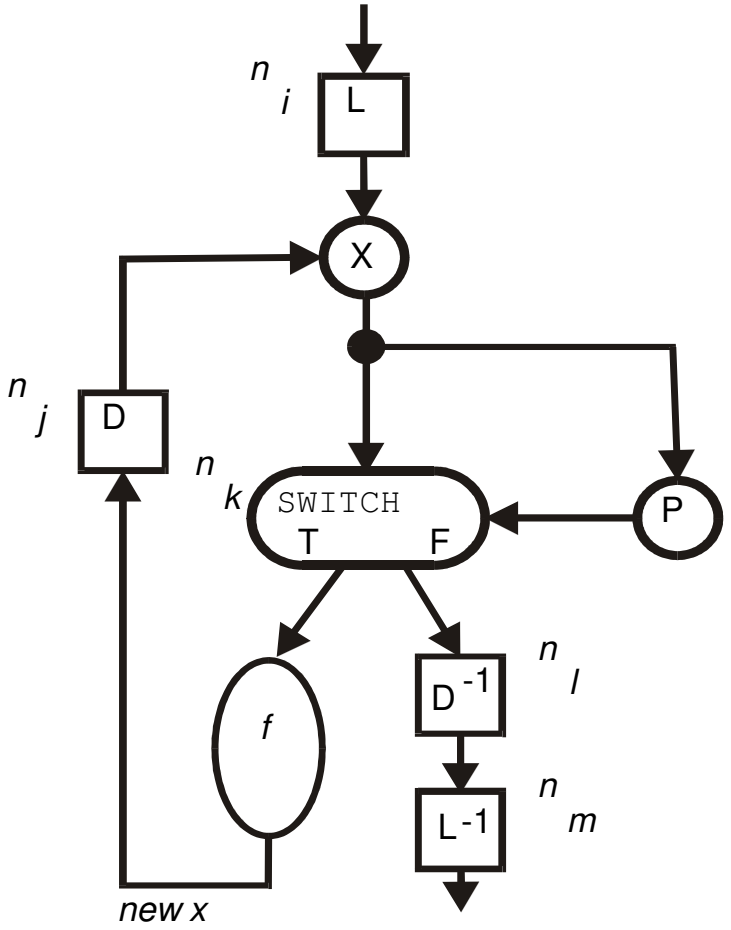


Branch



Speculative branch evaluation

Basic Loop Implementation



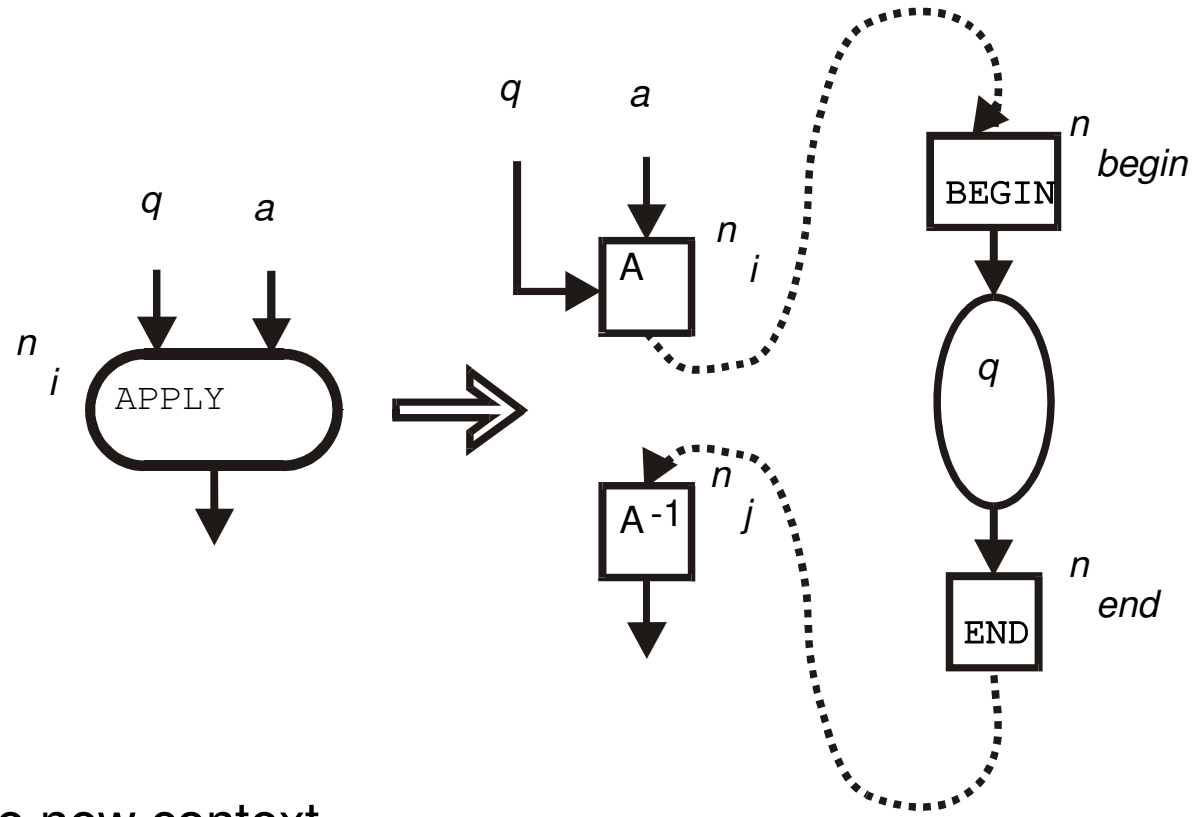
L: initiation, new loop context

D: increments loop iteration number

D^{-1}: reset loop iteration number to 1

L^{-1}: restore original context

Function application



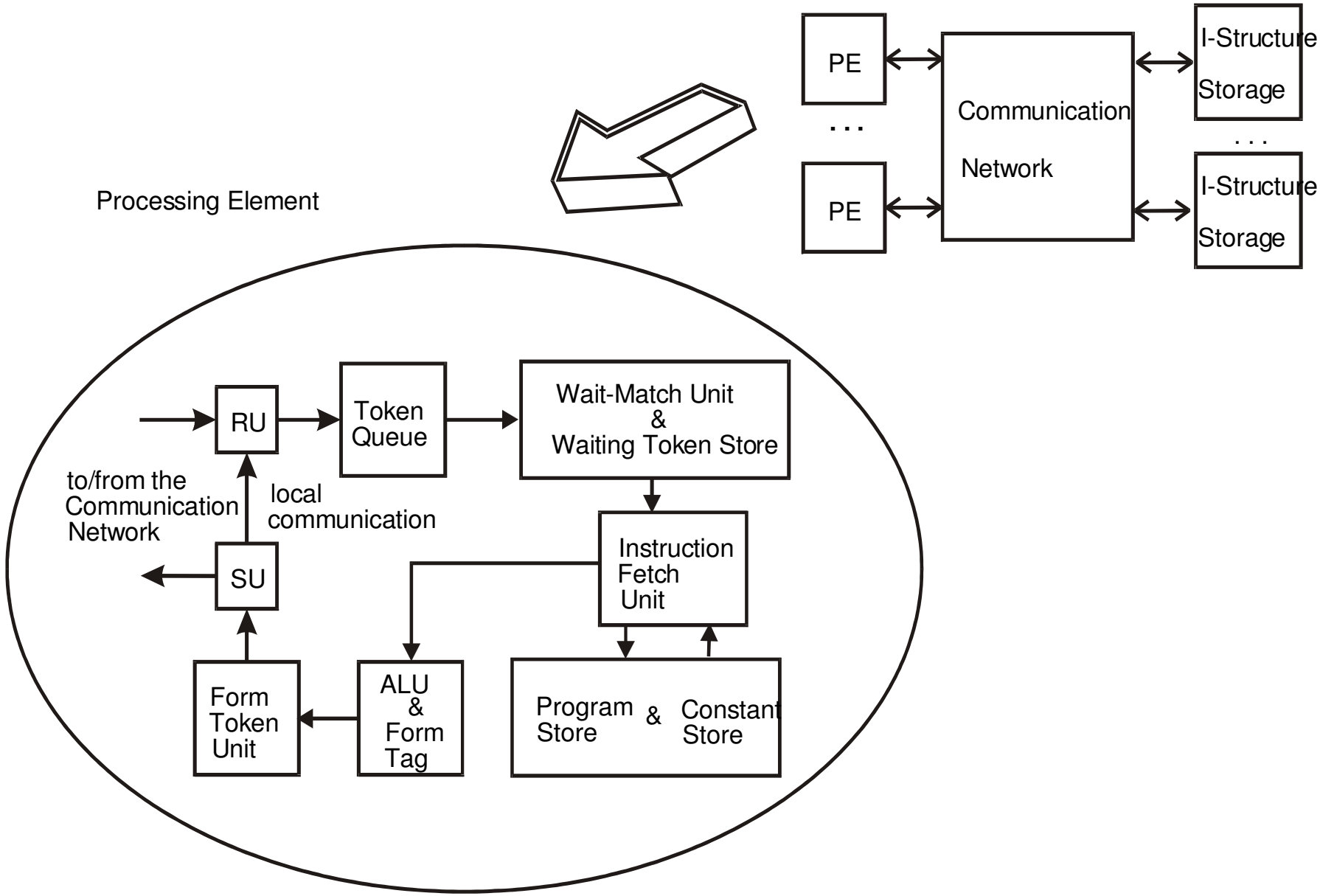
A: create new context

BEGIN: replicate tokens for each fork

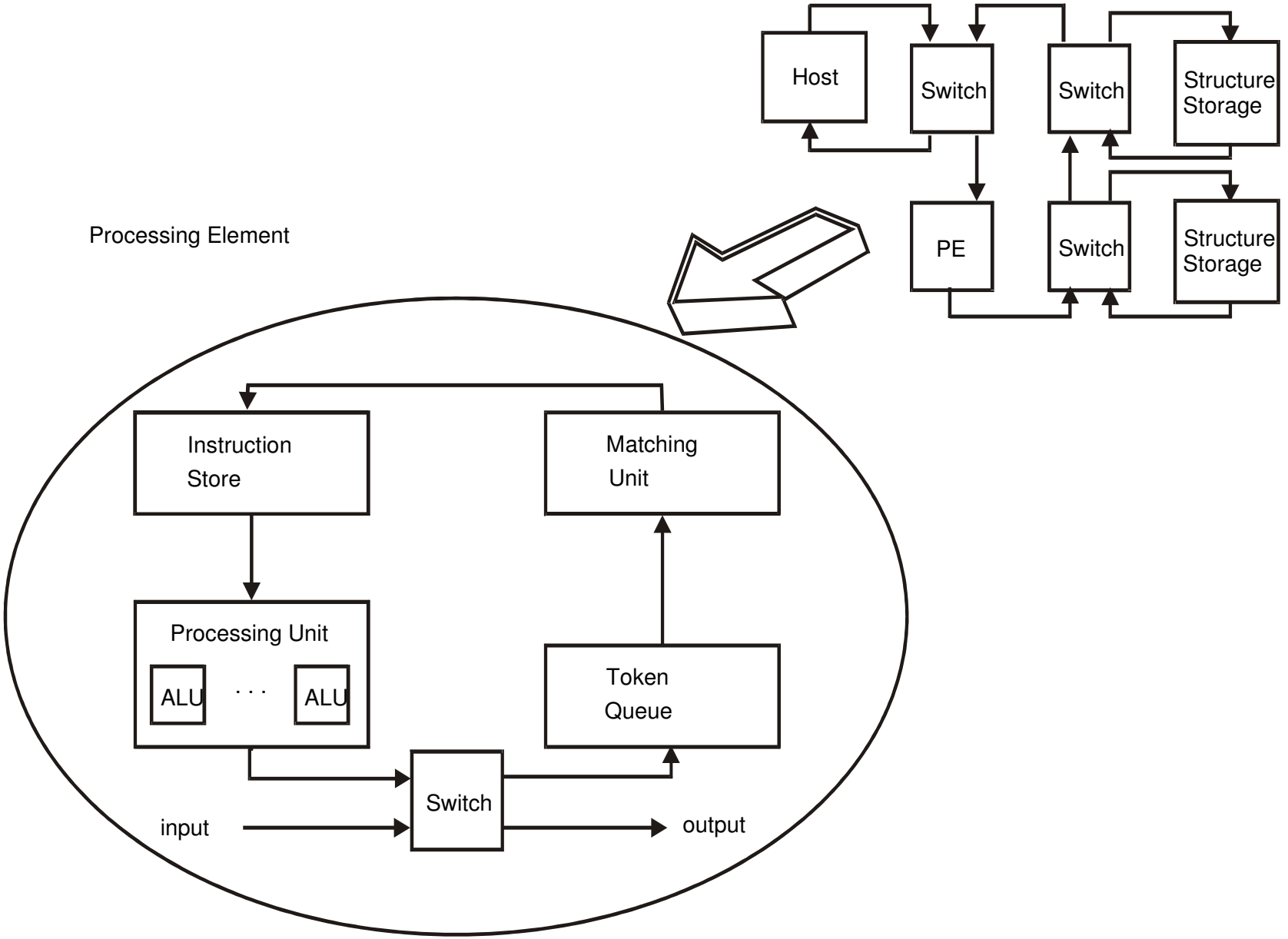
END: return results, unstack return address

A^{-1}: replicate output for successors

MIT Tagged-Token Dataflow Architecture



Manchester Dataflow Machine



Advantages and Deficiencies of Dynamic Dataflow

- Major advantage: better performance (compared with static) because it allows multiple tokens on each arc thereby unfolding more parallelism.

- Problems:
  - efficient implementation of the *matching unit* that collects tokens with matching tags.
    - *Associative memory* would be ideal.
    - Unfortunately, it is not cost-effective since the amount of memory needed to store tokens waiting for a match tends to be very large.
    - All existing machines use some form of *hashing* techniques.
  - bad single thread performance (when not enough workload is present)
  - dyadic instructions lead to pipeline bubbles when first operand tokens arrive
  - no instruction locality $\rightarrow$ no use of registers

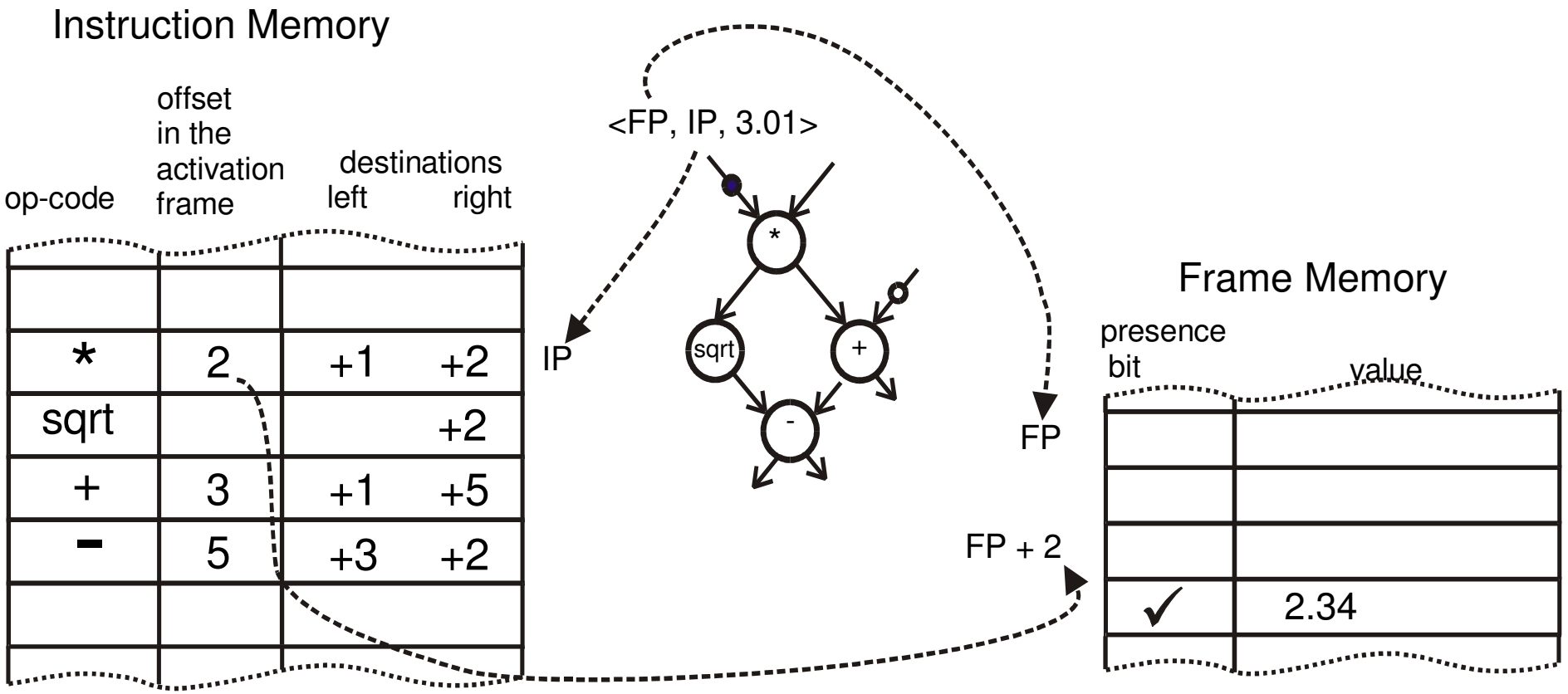
Explicit Token Store (ETS) Approach

- **Target**: efficient implementation of token matching.

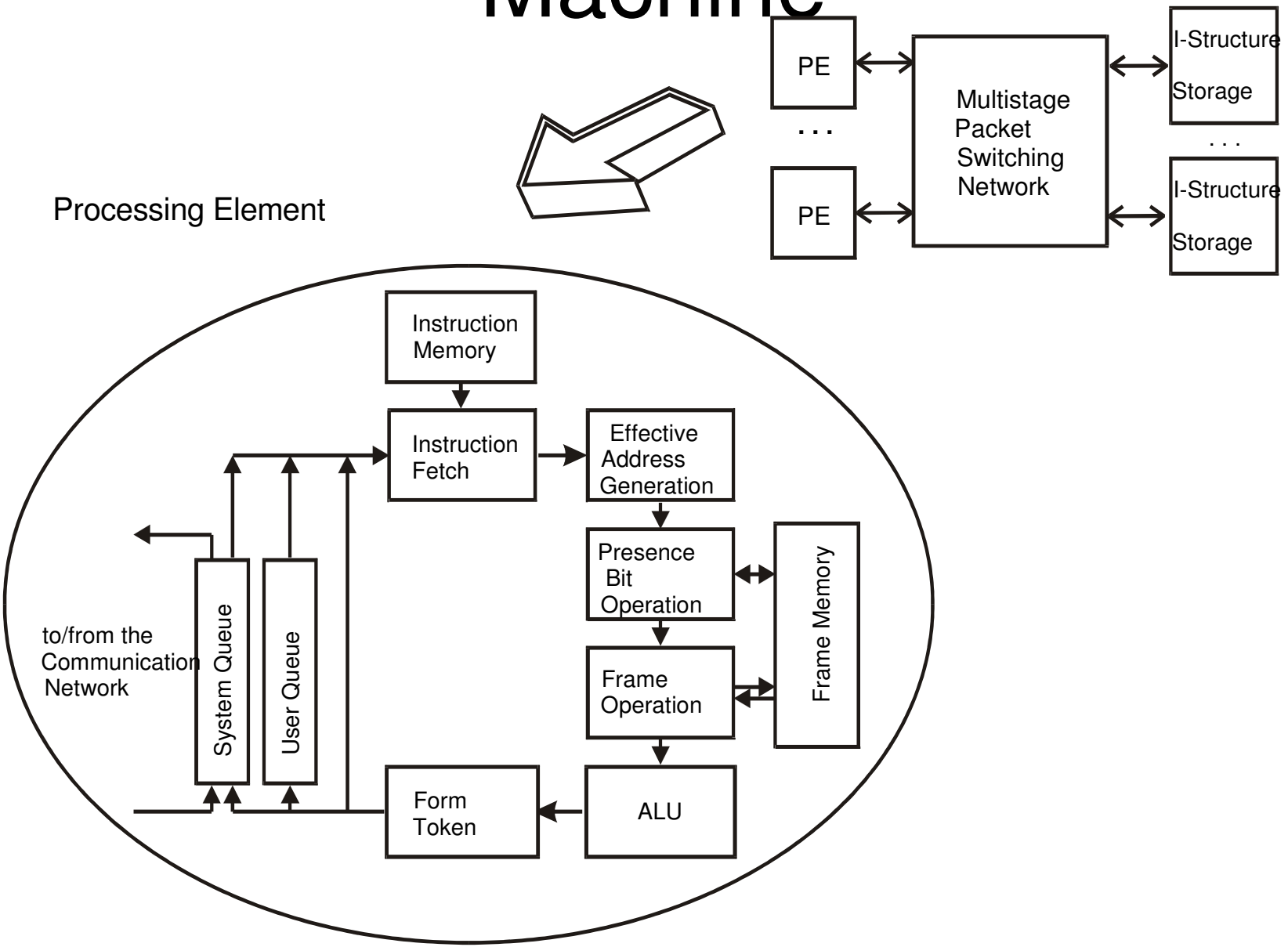
- **Basic idea**: allocate a *separate frame* in a *frame memory* for each active loop iteration or subprogram invocation.

- A *frame* consists of slots; each *slot* holds an operand that is used in the corresponding activity.
- *Access to slots* is direct (i.e. through offsets relative to the frame pointer)
  $\rightarrow$ no associative search is needed.

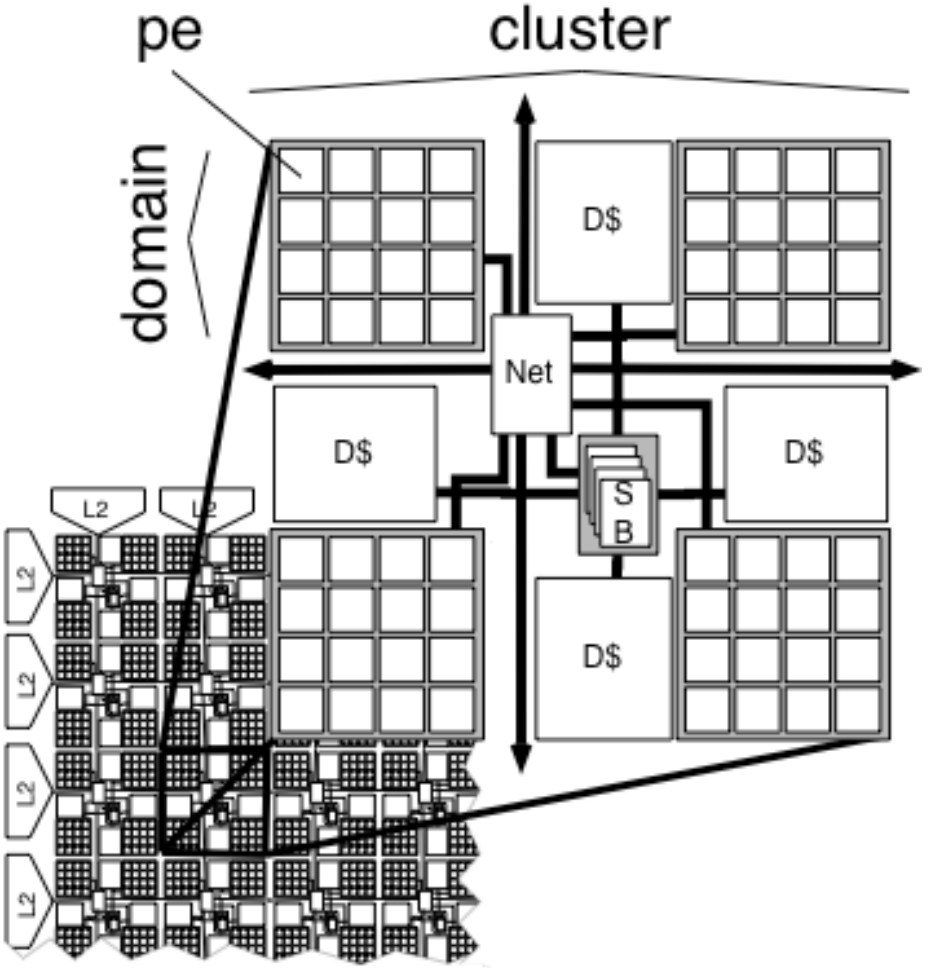
Explicit Token Store



Monsoon, an Explicit Token Store Machine

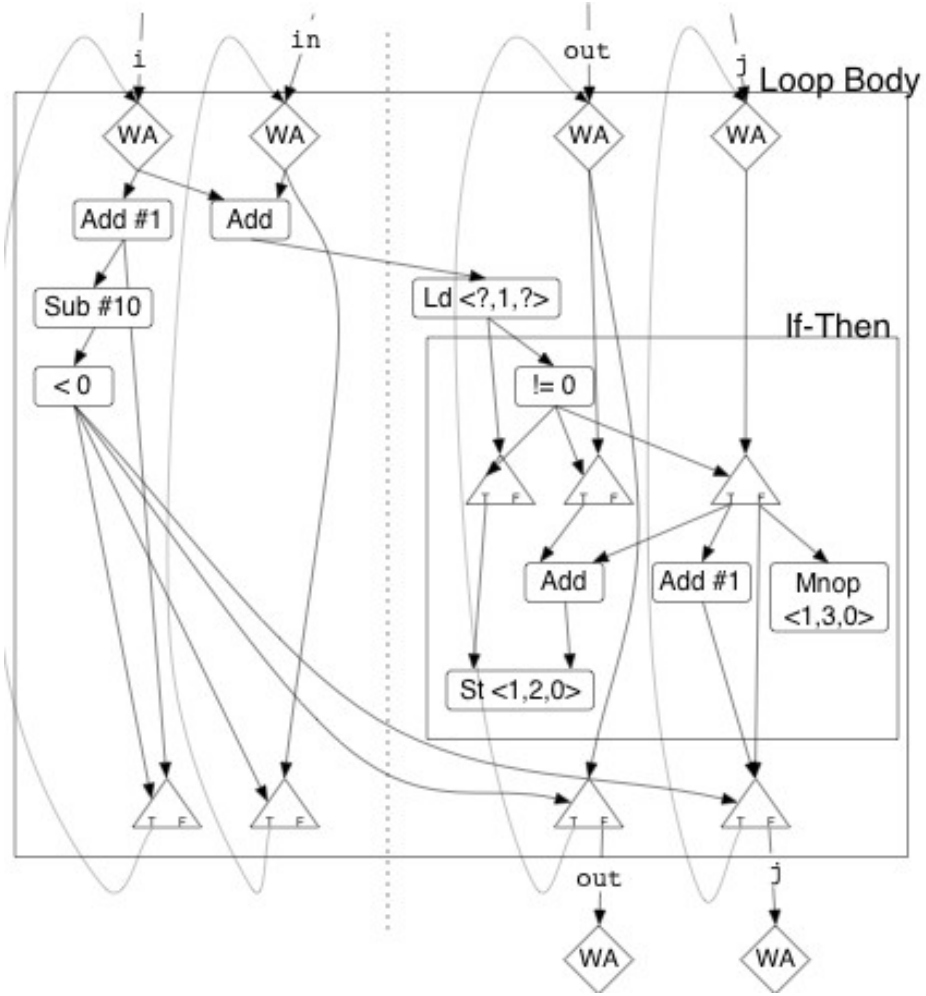


WaveCache: A Dataflow Processor



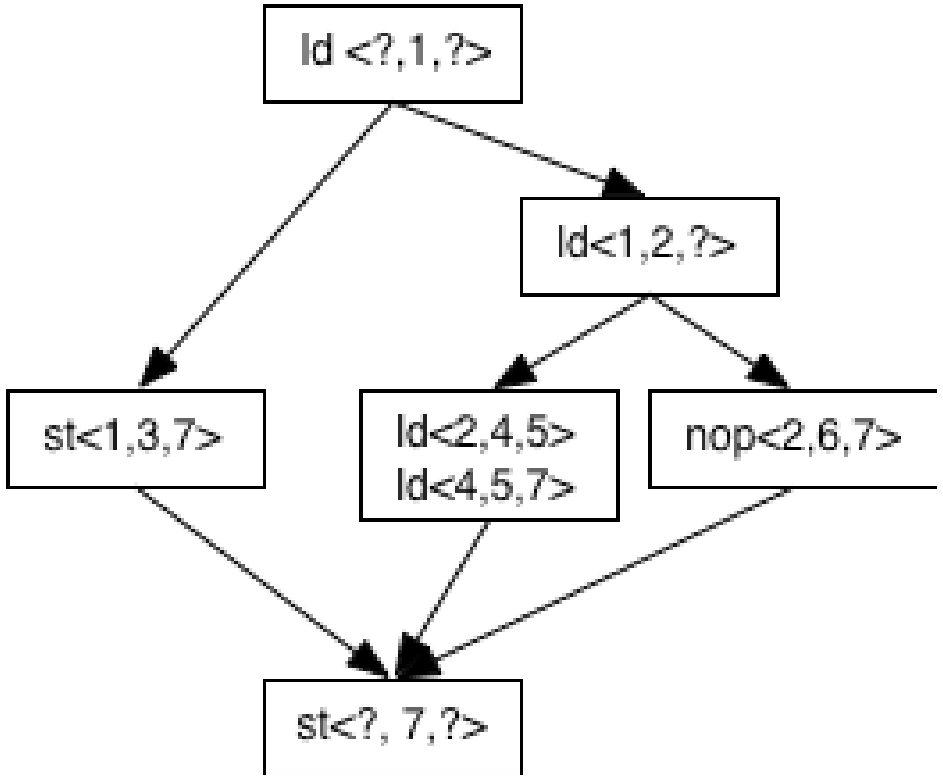
- WaveScalar is an ISA of a dataflow processor named WaveCache
- The WaveCache is a grid of approximately 2K processing elements (PEs) arranged into clusters of 16

WaveCache: A Dataflow Processor



- A WaveScalar executable contains an encoding of the program dataflow graph
- The instructions explicitly send data values to the instructions that need them instead of broadcasting them via the register file
- The potential consumers are known at compile time, but depending on control flow, only a subset of them should receive the values at run-time

WaveCache: A Dataflow Processor



- Traditional imperative languages provide the programmer with a model of memory known as total load-store ordering
- WaveScalar brings load-store ordering to dataflow computing using wave-ordered memory
- Wave-ordered memory annotates each memory operation with its location in its wave and its ordering relationships (defined by the control flow graph) with other memory operations in the same wave