18-742 Fall 2012
Parallel Computer Architecture
Lecture 22: Dataflow I

Prof. Onur Mutlu
Carnegie Mellon University

10/31/2012

New Review Assignments

- Were Due: Sunday, October 28, 11:59pm.
  - Das et al., "Aergia: Exploiting Packet Latency Slack in On-Chip Networks," ISCA 2010.
  - Dennis and Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," ISCA 1974.

- Was Due: Tuesday, October 30, 11:59pm.
  - Arvind and Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," IEEE TC 1990.

- Due: Thursday, November 1, 11:59pm.
  - Patt et al., "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.
  - Patt et al., "Critical issues regarding HPS, a high performance microarchitecture," MICRO 1985.

Other Readings

- Dataflow
  - Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
  - Lee and Hurson, "Dataflow Architectures and Multithreading," IEEE Computer 1994.

- Restricted Dataflow
  - Patt et al., "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.
  - Patt et al., "Critical issues regarding HPS, a high performance microarchitecture," MICRO 1985.
  - Sankaralingam et al., "Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture," ISCA 2003.
  - Burger et al., "Scaling to the End of Silicon with EDGE Architectures," IEEE Computer 2004.

Last Lecture

- Interconnects Research
  - Application-aware packet scheduling
  - Slack-driven packet scheduling
  - Scalable topologies
  - QoS and topology+QoS co-design for scalability

Today

- Start Dataflow

Data Flow

Readings: Data Flow (I)

- Dennis and Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," ISCA 1974.
- Treleaven et al., "Data-Driven and Demand-Driven Computer Architecture," ACM Computing Surveys 1982.
- Veen, "Dataflow Machine Architecture," ACM Computing Surveys 1986.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
- Arvind and Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," IEEE TC 1990.
- Patt et al., "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.
- Lee and Hurson, "Dataflow Architectures and Multithreading," IEEE Computer 1994.

Readings: Data Flow (II)

- Sankaralingam et al., "Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture," ISCA 2003.
- Burger et al., "Scaling to the End of Silicon with EDGE Architectures," IEEE Computer 2004.

Data Flow

- The models we have examined in 447/740 all assumed
  - Instructions are fetched and retired in sequential, control flow order

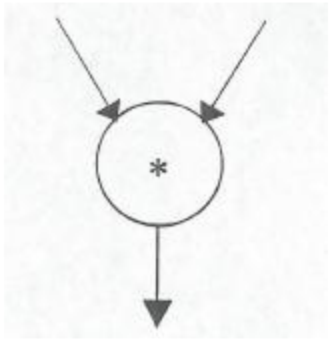
- This is part of the Von-Neumann model of computation
  - Single program counter
  - Sequential execution
  - Control flow determines fetch, execution, commit order

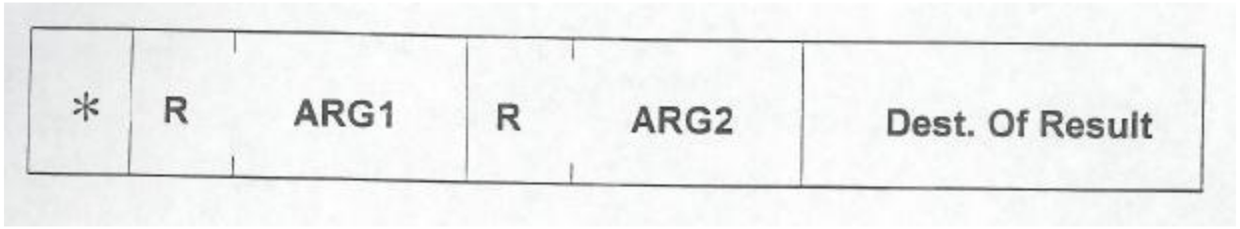
- What about out-of-order execution?
  - Architecture level: Obeys the control-flow model
  - Uarch level: A window of instructions executed in data-flow order → execute an instruction when its operands become available

Data Flow

- In a data flow machine, a program consists of data flow nodes
- A data flow node fires (fetched and executed) when all its inputs are ready
  - i.e. when all inputs have tokens

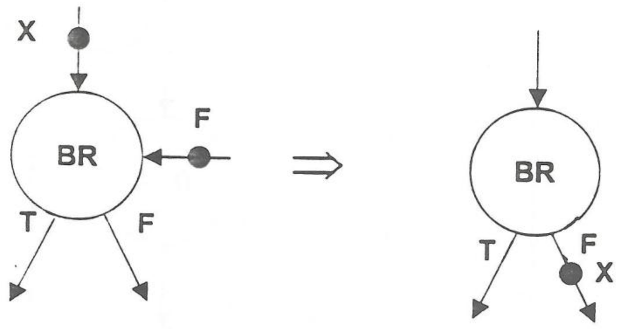
- Data flow node and its ISA representation



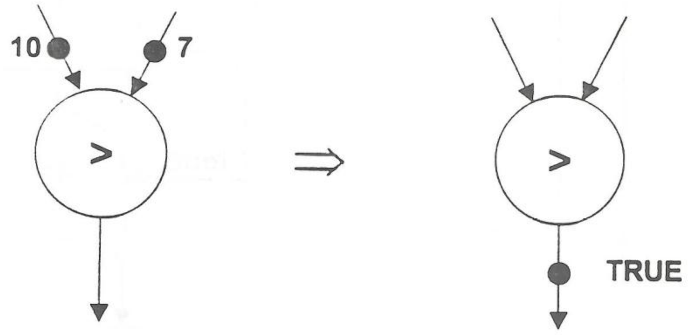


Data Flow Nodes

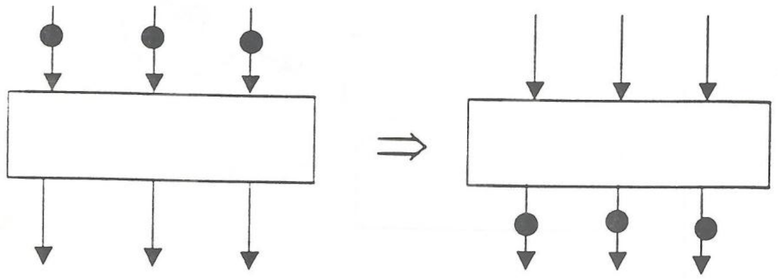
Conditional



Relational

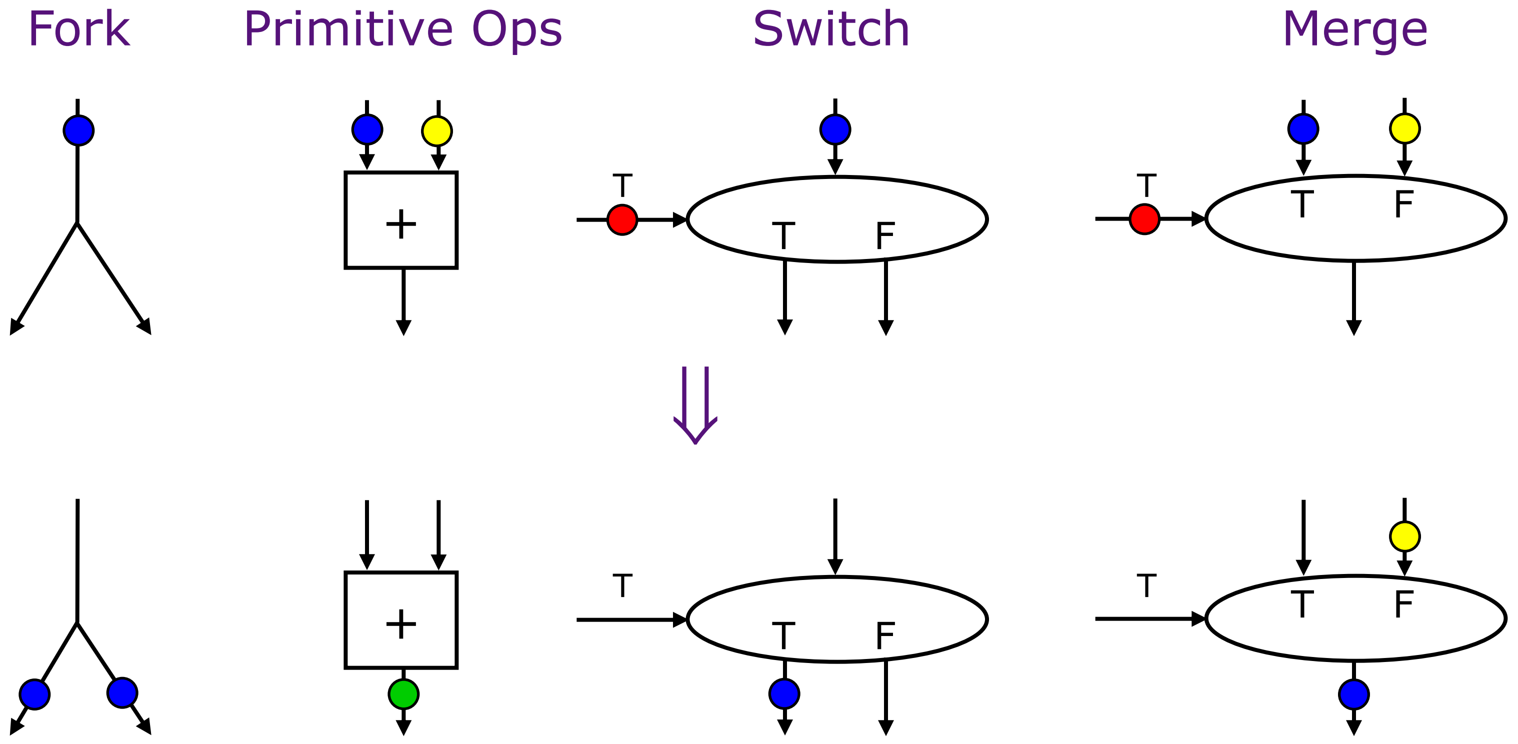


*Barrier Synch



Data Flow Nodes (II)

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



Dataflow Graphs

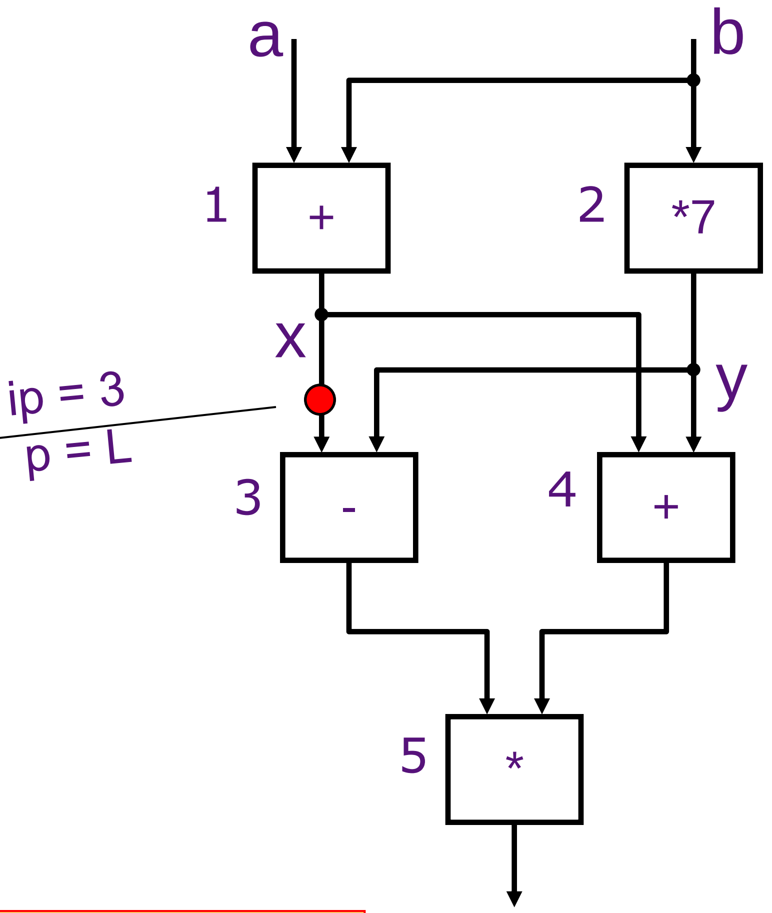
```
{x = a + b;
 y = b * 7
in
   (x-y) * (x+y)}
```

- Values in dataflow graphs are represented as tokens

token < ip , p , v >

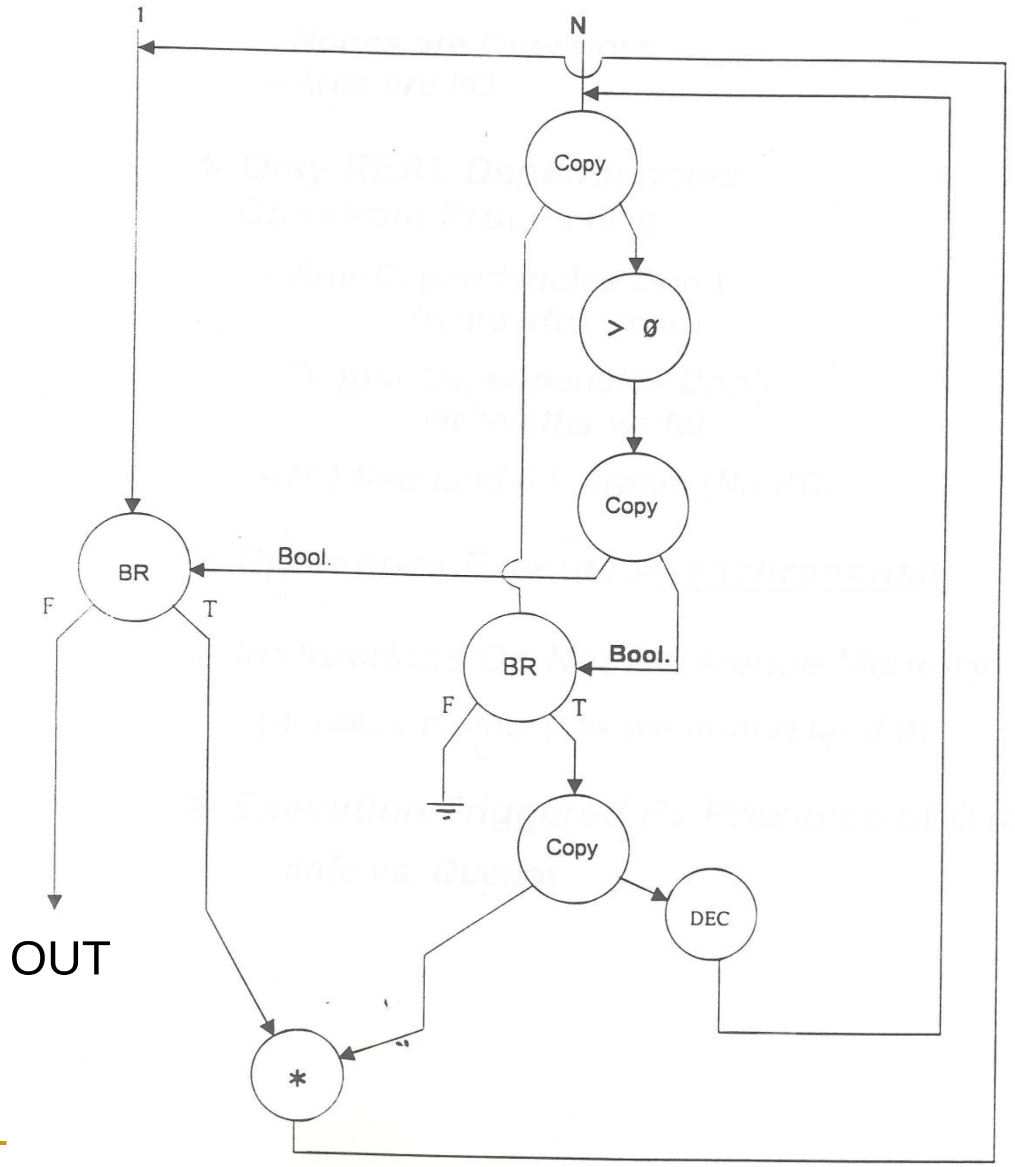
instruction ptr port data

- An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators



no separate control flow

Example Data Flow Program



Control Flow vs. Data Flow

```
a := x + y
b := a × a
c := 4 — a
```

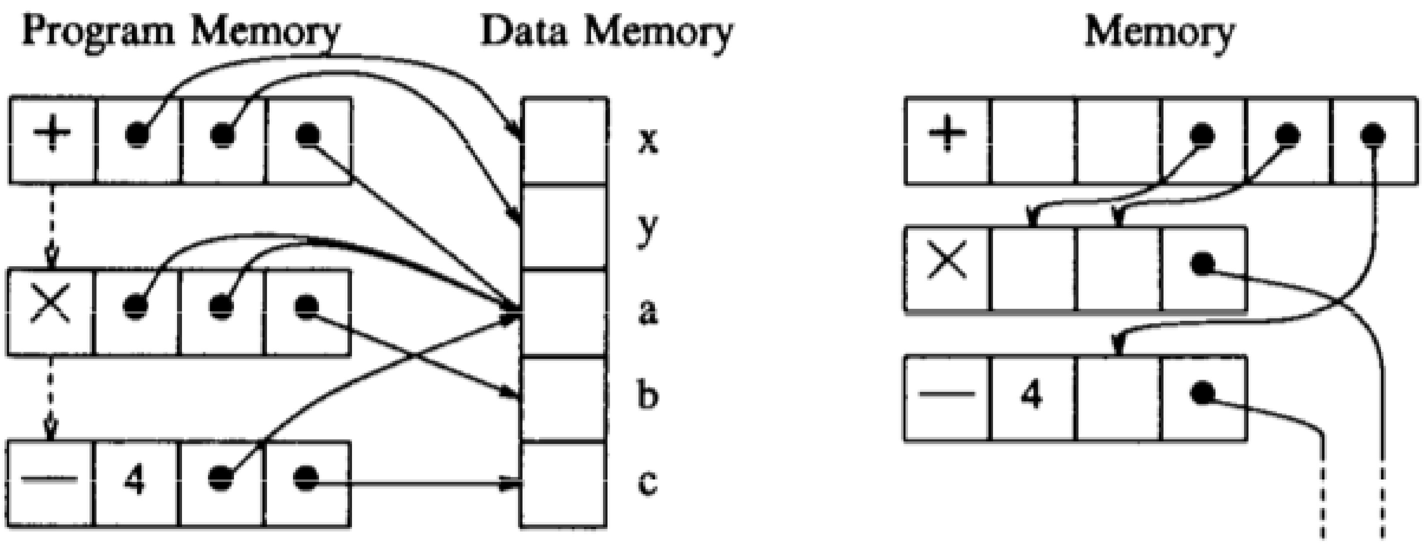


Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

Static Dataflow

- Allows only one instance of a node to be enabled for firing

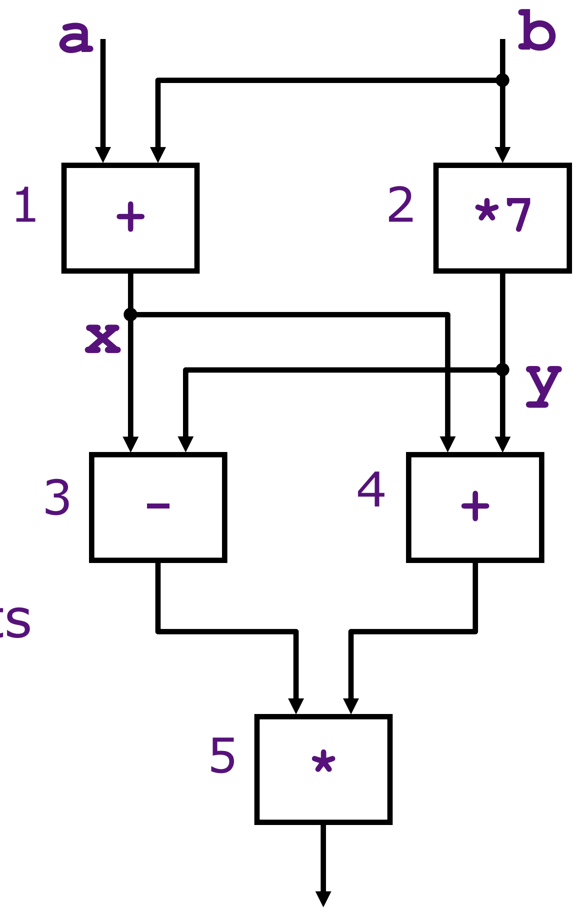
- A dataflow node is fired only when all of the tokens are available on its input arcs and no tokens exist on any of its its output arcs

- Dennis and Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," ISCA 1974.

Static Dataflow Machine:
Instruction Templates

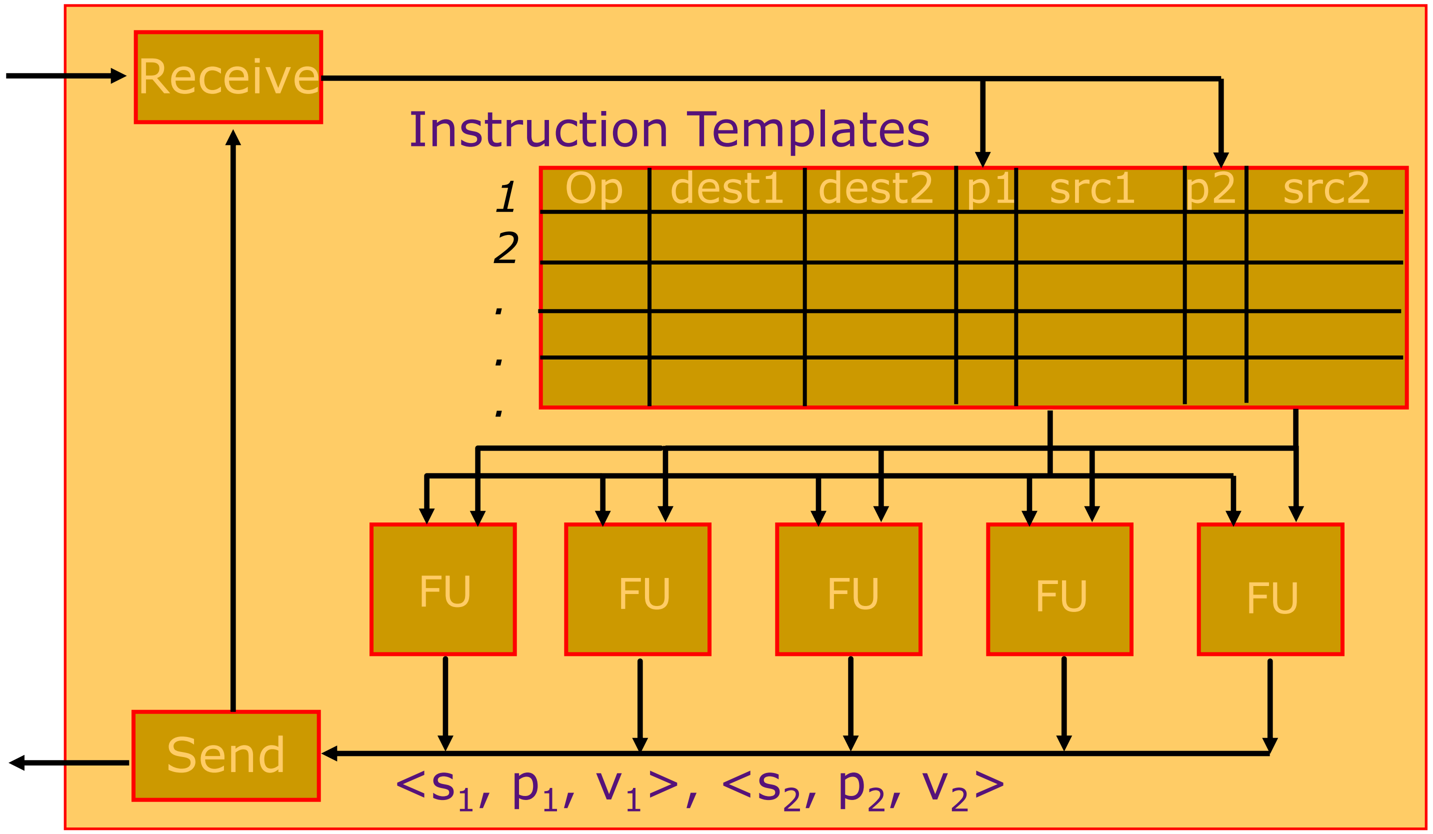
	Opcode	Destination 1	Destination 2		Operand 1		Operand 2
1	+	3L	4L				
2	*	3R	4R				
3	−	5L					
4	+	5R					
5	*	out					

Presence bits



Each arc in the graph has an operand slot in the program

Static Dataflow Machine (Dennis+, ISCA 1974)



- Many such processors can be connected together
- Programs can be statically divided among the processors

Static versus Dynamic Dataflow Machines

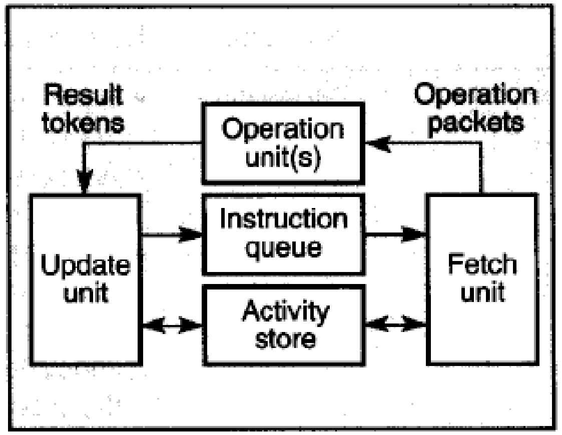


Figure 1. The basic organization of the static dataflow model.

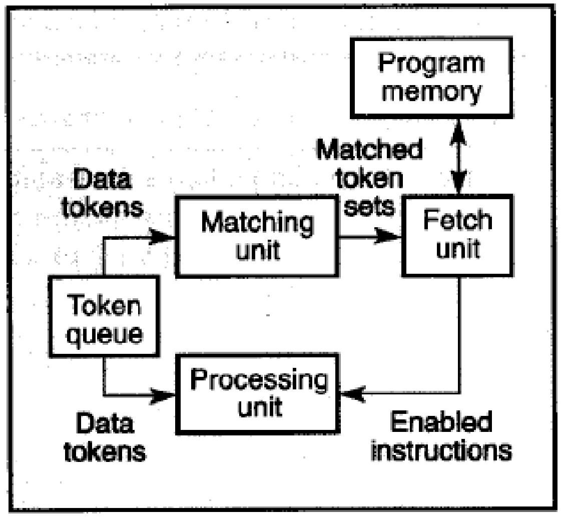
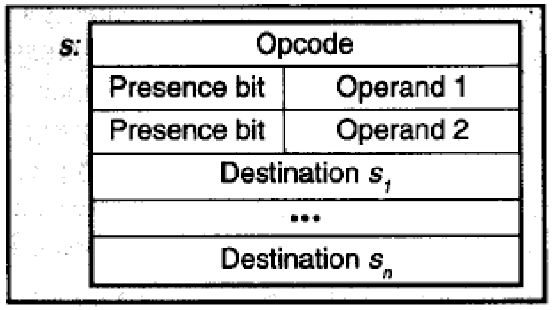


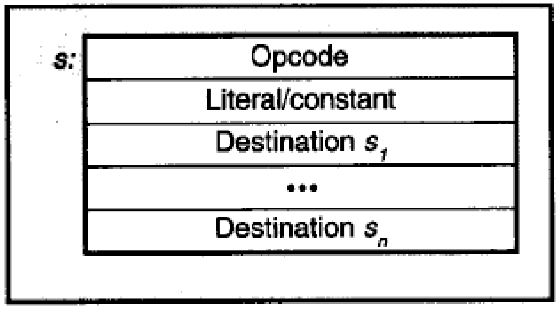
Figure 3. The general organization of the dynamic dataflow model.



s:

Opcode	
Presence bit	Operand 1
Presence bit	Operand 2
Destination s_1	
...	
Destination s_n	

Figure 2. An instruction template for the static dataflow model.



s:

Opcode
Literal/constant
Destination s_1
...
Destination s_n

Figure 4. An instruction format for the dynamic dataflow model.

Static Data Flow Machines

- Mismatch between the model and the implementation
  - The model requires *unbounded FIFO token queues* per arc but the architecture provides storage for one token per arc
  - The architecture *does not ensure FIFO* order in the reuse of an operand slot

- The static model *does not support*
  - Reentrant code
    - Function calls
    - Loops
  - Data Structures

Problems with Re-entrancy

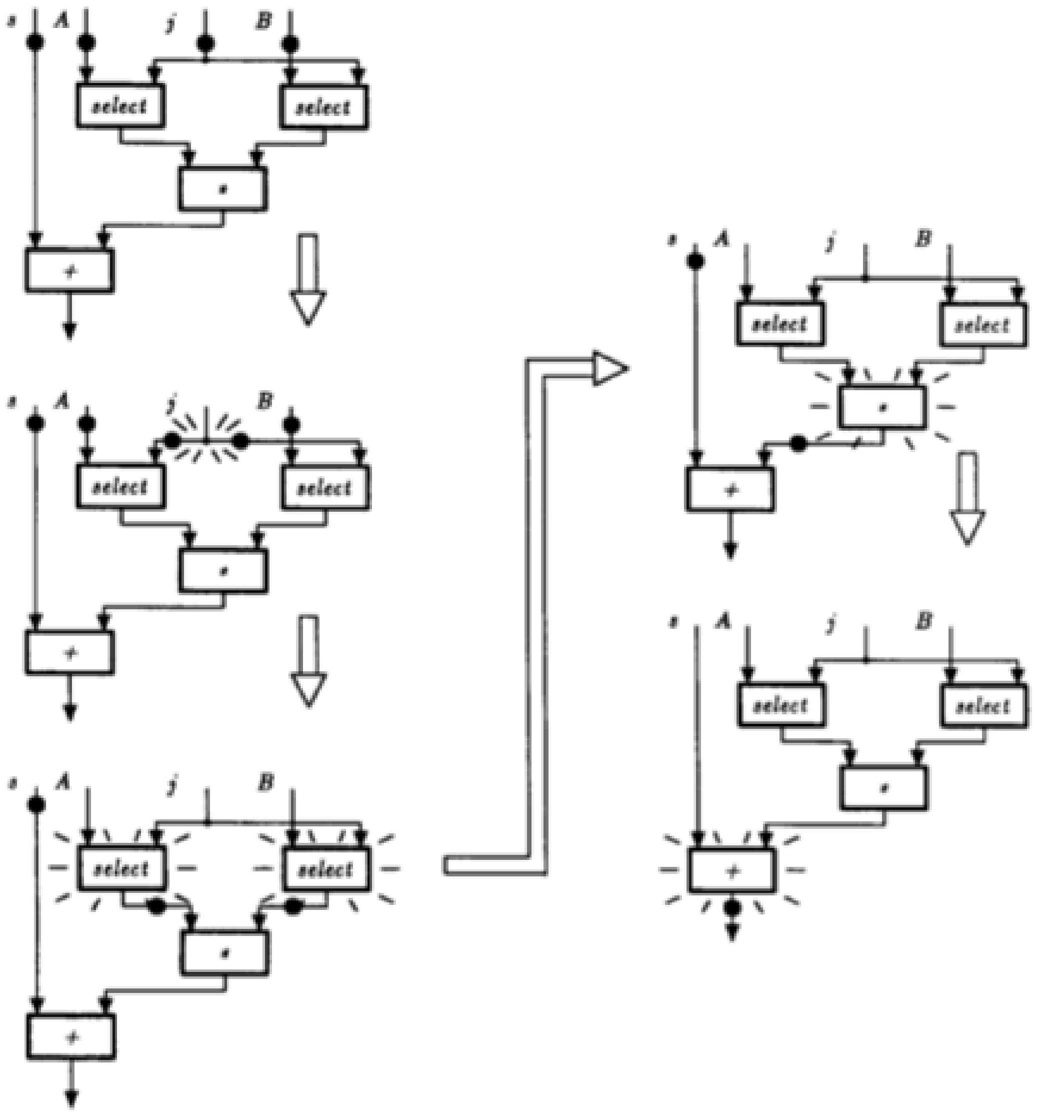


Fig. 3. A firing sequence for "s + A[i] * B[i]."

- Assume this was in a loop
- Or in a function
- And operations took variable time to execute
- How do you ensure the tokens that match are of the same invocation?

Dynamic Dataflow Architectures

- Allocate instruction templates, i.e., a frame, dynamically to support each loop iteration and procedure call
  - termination detection needed to deallocate frames

- The code can be shared if we separate the code and the operand storage

a token <fp, ip, port, data>

frame pointer

instruction pointer

A Frame in Dynamic Dataflow

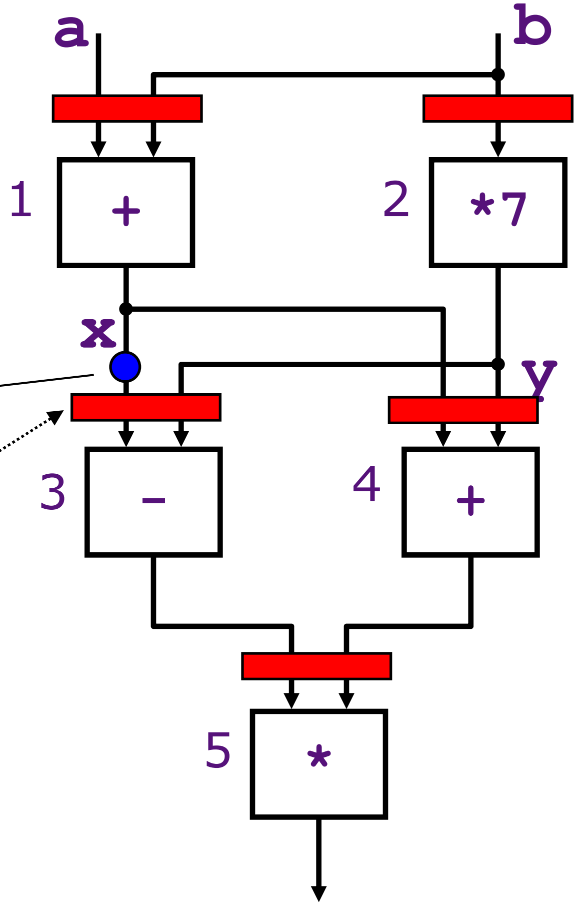
Program

1	+	*1*	3L, 4L
2	*	*2*	3R, 4R
3	-	*3*	5L
4	+	*4*	5R
5	*	*5*	out

<fp, ip, p , v>

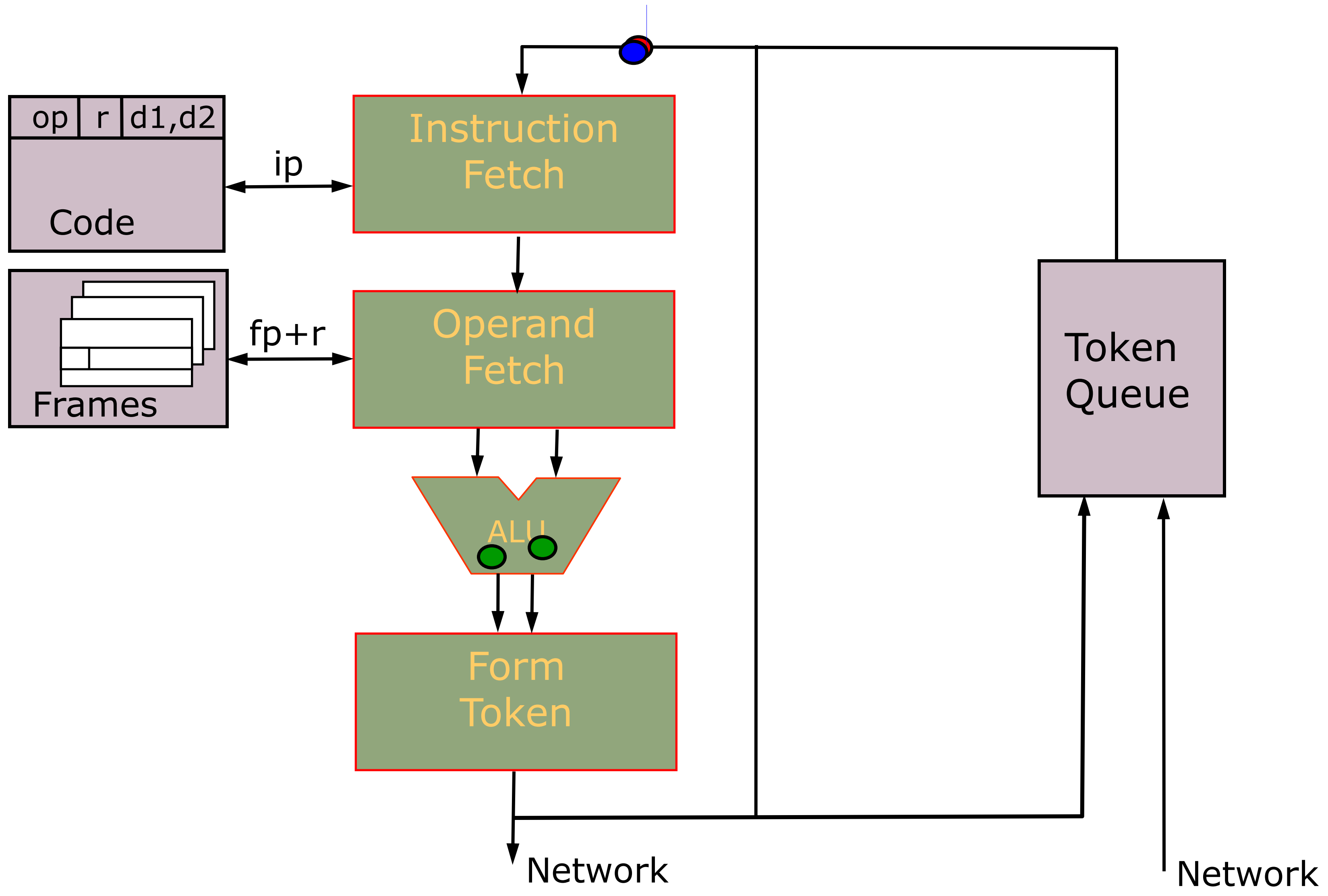
1		
2		
3	L	7
4		
5		

Frame



Need to provide storage for only one operand/operator

Monsoon Processor (ISCA 1990)



Concept of Tagging

- Each invocation receives a separate tag

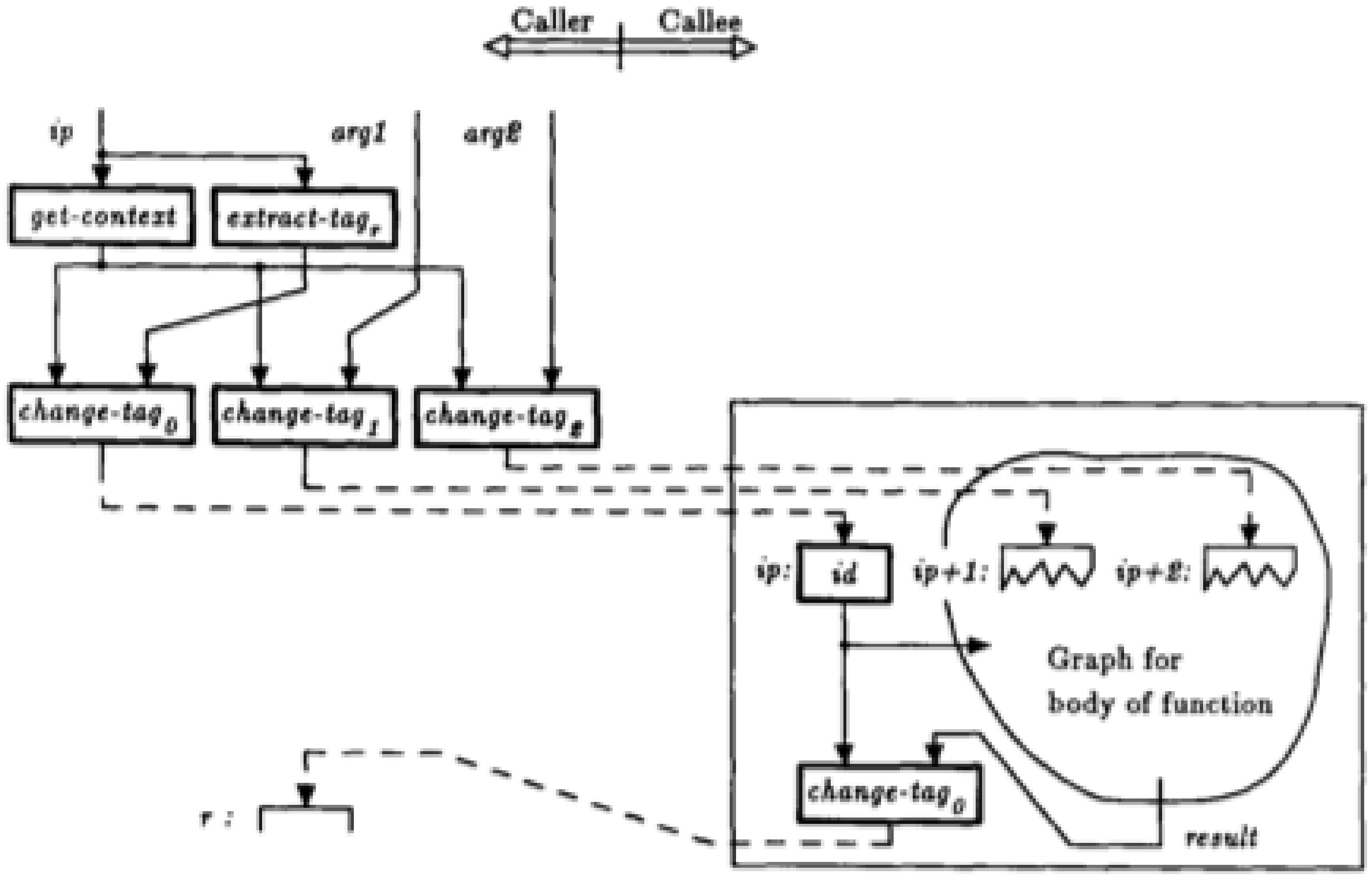
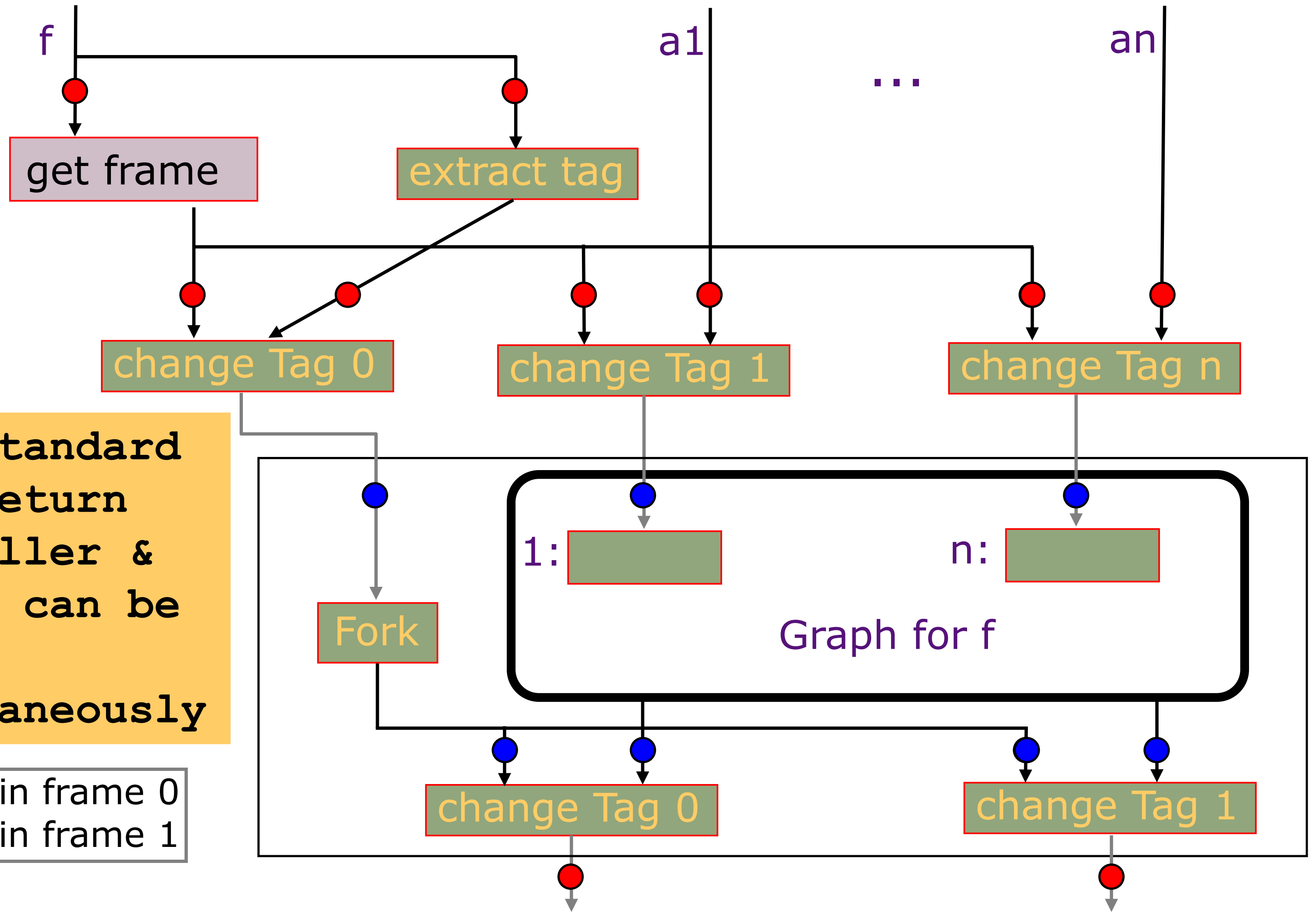


Fig. 6. Dataflow graph for function call and return linkage.

Procedure Linkage Operators



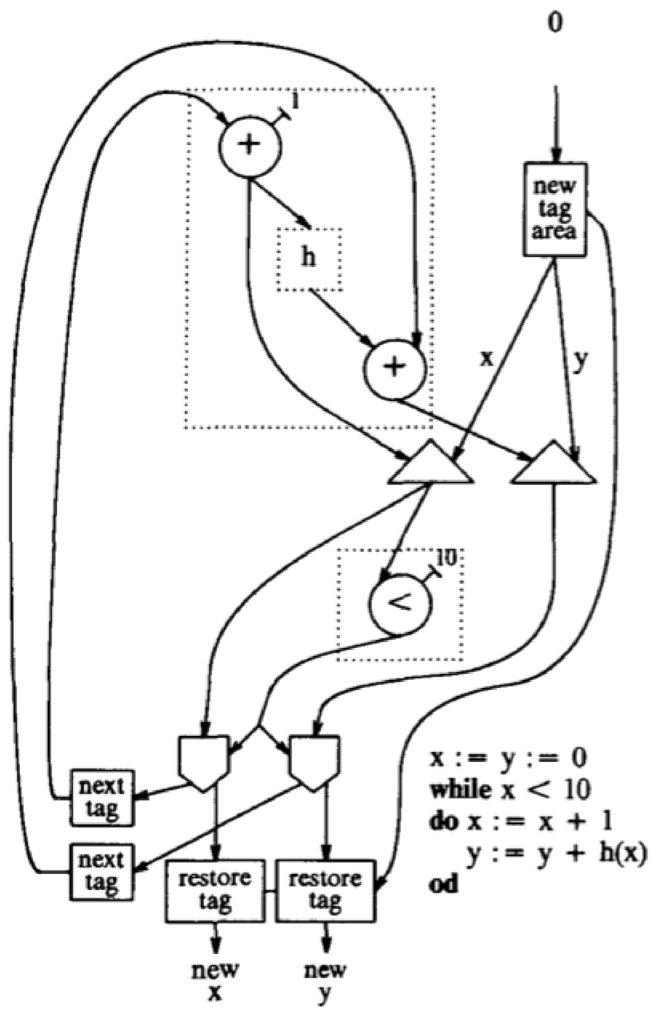
Like standard call/return but caller & callee can be active simultaneously

🔴 token in frame 0
🔵 token in frame 1

Function Calls

- Need extra mechanism to direct the output token of the function to the proper calling site
- Usually done by sending special token containing the return node address

Loops and Function Calls Summary



```
x := y := 0
while x < 10
do x := x + 1
   y := y + h(x)
od
```

Figure 10. An implementation of a loop using tagged tokens. At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations receive consecutive tags within this area. The tag from before the loop is restored on tokens that exit from the loop.

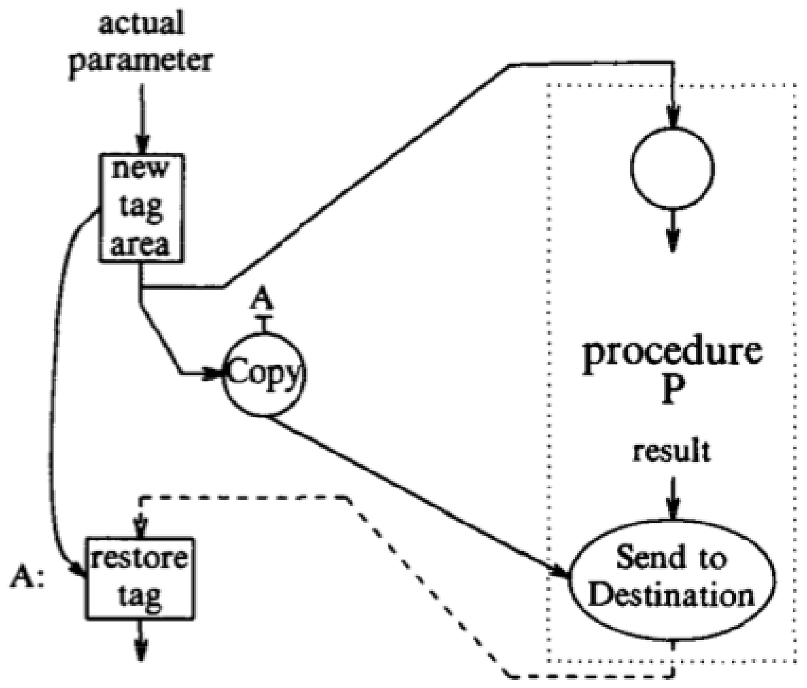


Figure 11. Interface for a procedure call. On the left a call of procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P and concurrently a token containing address A is sent to the output node. This SEND-TO-DESTINATION node transmits the other input token to a node of which the address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which restores the tag belonging to the calling expression.

Control of Parallelism

- Problem: Many loop iterations can be present in the machine at any given time
  - 100K iterations on a 256 processor machine can swamp the machine (thrashing in token matching units)
  - Not enough bits to represent frame id

- Solution: Throttle loops. Control how many loop iterations can be in the machine at the same time.
  - Requires changes to loop dataflow graph to inhibit token generation when number of iterations is greater than N

Data Structures

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

- Data structures as we know of them (in imperative languages) are structures with persistent state
- Why do we want persistent state?
  - More natural representation for some tasks? (bank accounts, databases, …)
  - To exploit locality

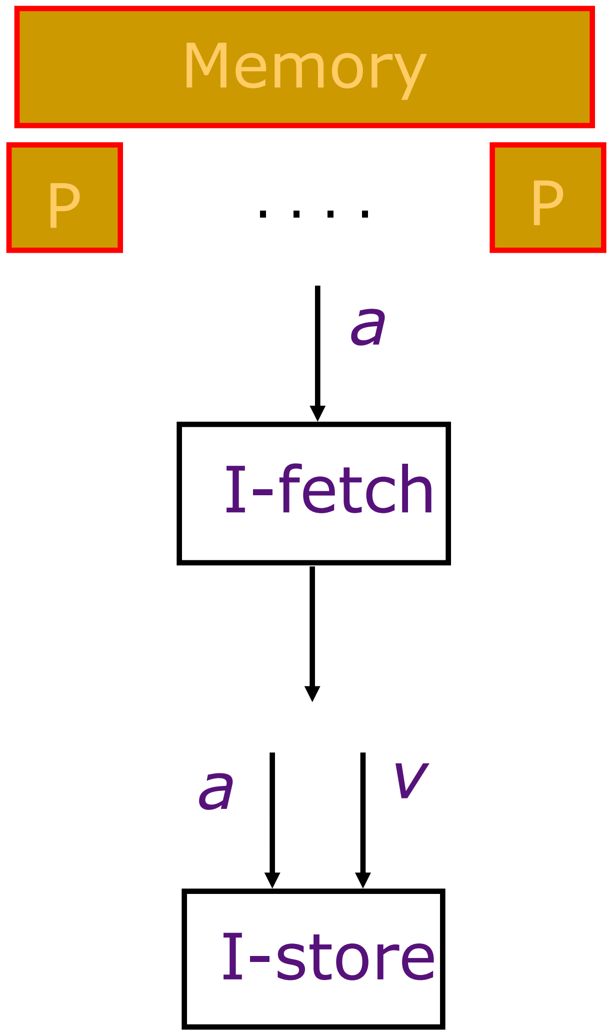
Data Structures in Dataflow

- Data structures reside in a structure store

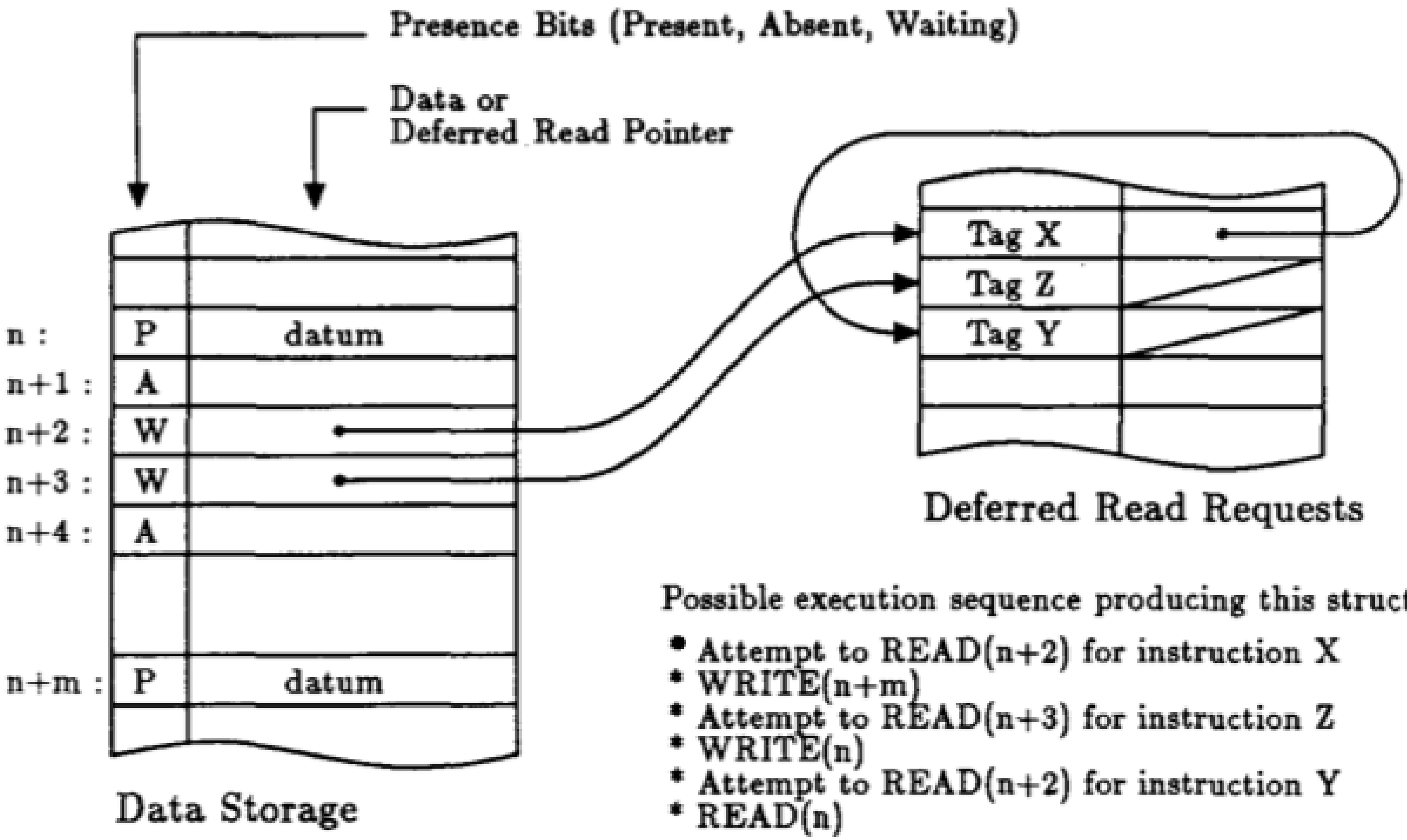
  ⇒ tokens carry pointers

- I-structures: Write-once, Read multiple times *or*
  - allocate, write, read, ..., read, deallocate

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



I-Structures



Possible execution sequence producing this structure:

- Attempt to READ(n+2) for instruction X
- WRITE(n+m)
- Attempt to READ(n+3) for instruction Z
- WRITE(n)
- Attempt to READ(n+2) for instruction Y
- READ(n)

Fig. 7. *I*-structure memory.

Dynamic Data Structures

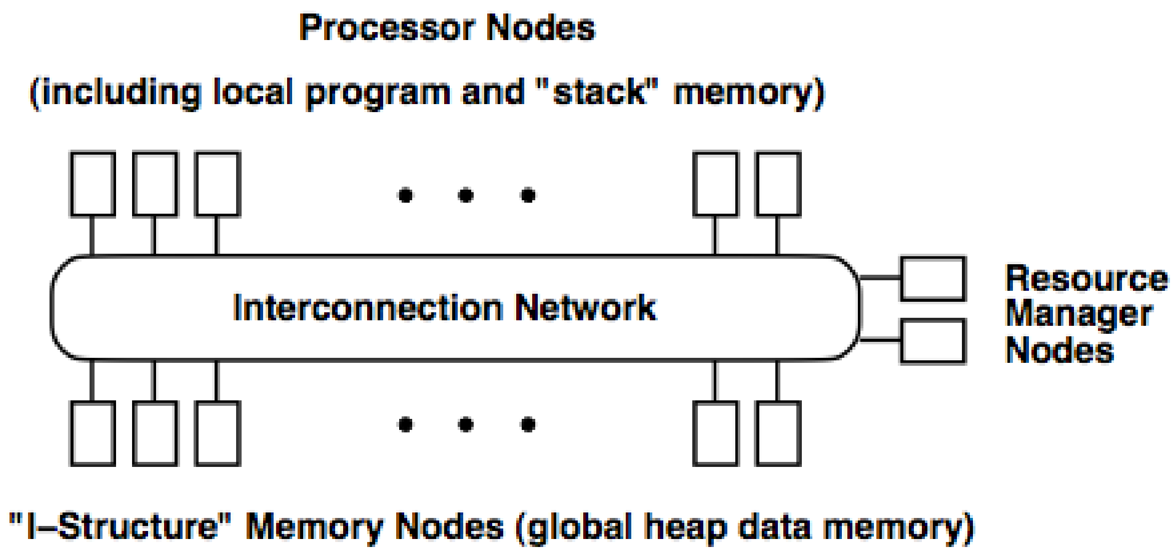
- Write-multiple-times data structures
- How can you support them in a dataflow machine?
  - Can you implement a linked list?

- What are the ordering semantics for writes and reads?

- Imperative vs. functional languages
  - Side effects and mutable state

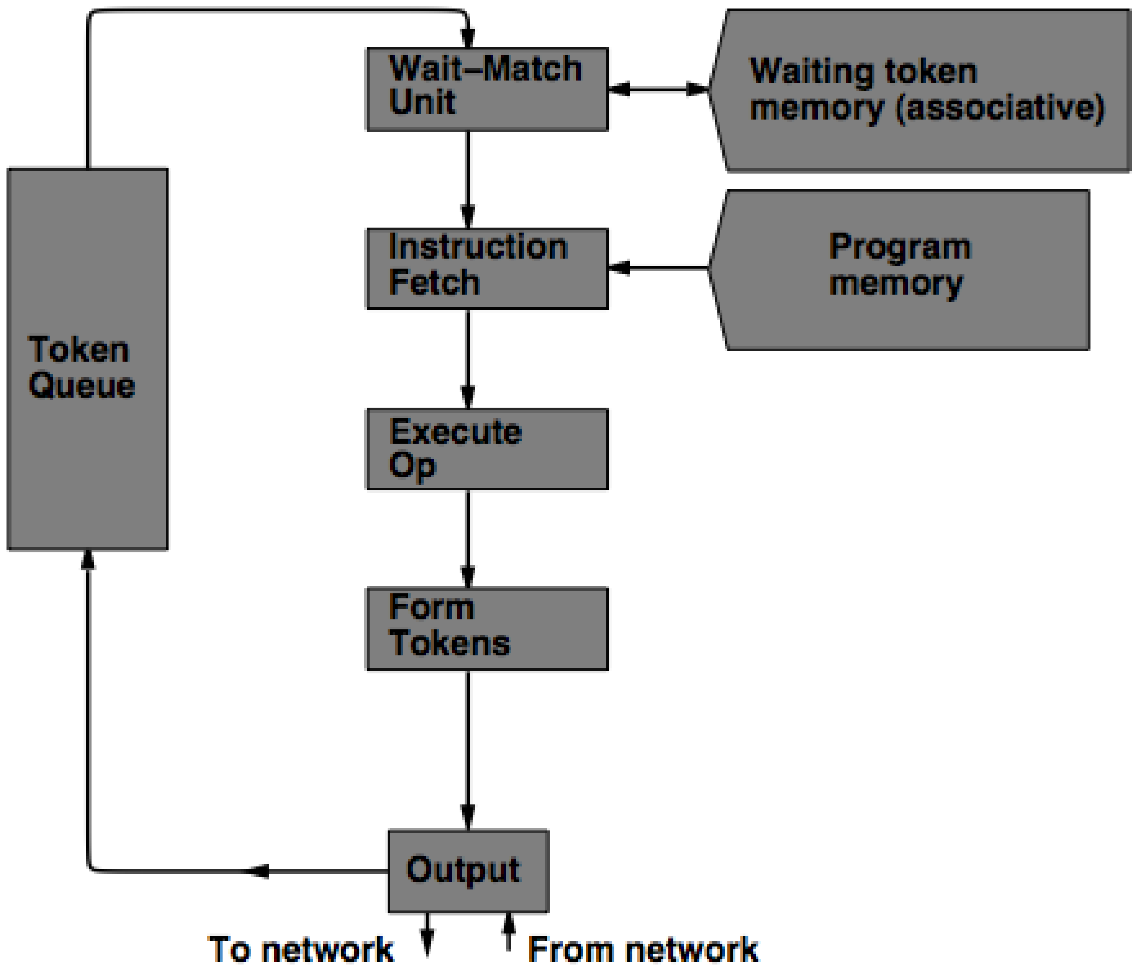
    vs.
  - No side effects and no mutable state

MIT Tagged Token Data Flow Architecture


Processor Nodes
(including local program and "stack" memory)
Interconnection Network
Resource Manager Nodes
"I–Structure" Memory Nodes (global heap data memory)

- Resource Manager Nodes
  - responsible for Function allocation (allocation of context/frame identifiers), Heap allocation, etc.

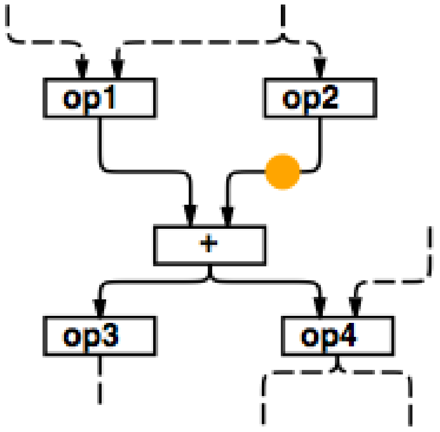
MIT Tagged Token Data Flow Architecture



- Wait–Match Unit: try to match incoming token and context id and a waiting token with same instruction address
  - Success: Both tokens forwarded
  - Fail: Incoming token --> Waiting Token Mem, bubble (no-op forwarded)

TTDA Data Flow Example

Conceptual



Encoding of graph

Program memory:

	Op–code	Destination(s)
109	op1	120L
113	op2	120R
120	+	141, 159L
141	op3	...
159	op4	... , ...

Re–entrancy ("dynamic" dataflow):

- Each invocation of a function or loop iteration gets its own, unique, "Context"
- Tokens destined for same instruction in different invocations are distinguished by a context identifier

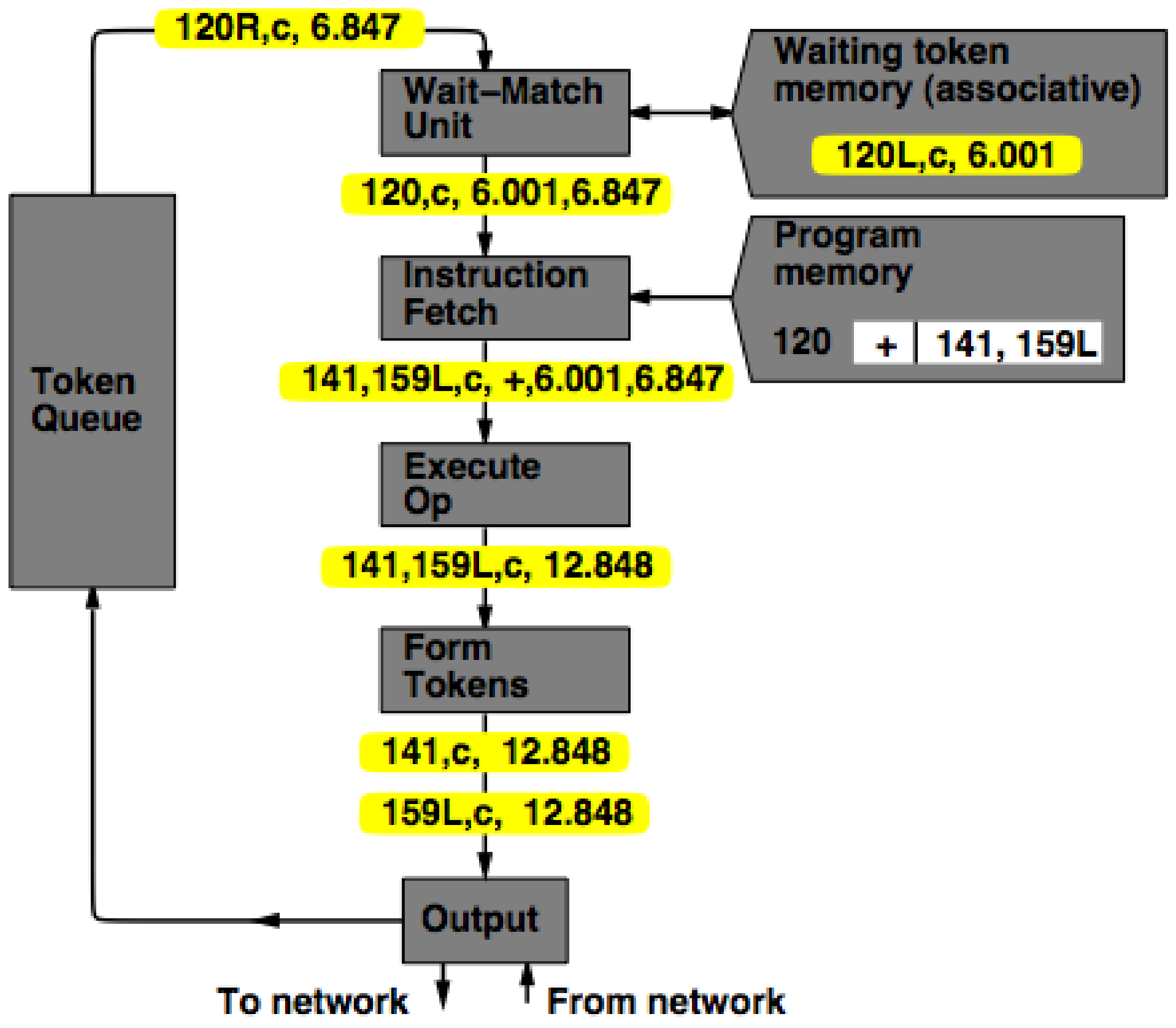
120R	Destination instruction address, Left/Right port
Ctxt	Context Identifier
6.847	Value

Encoding of token:

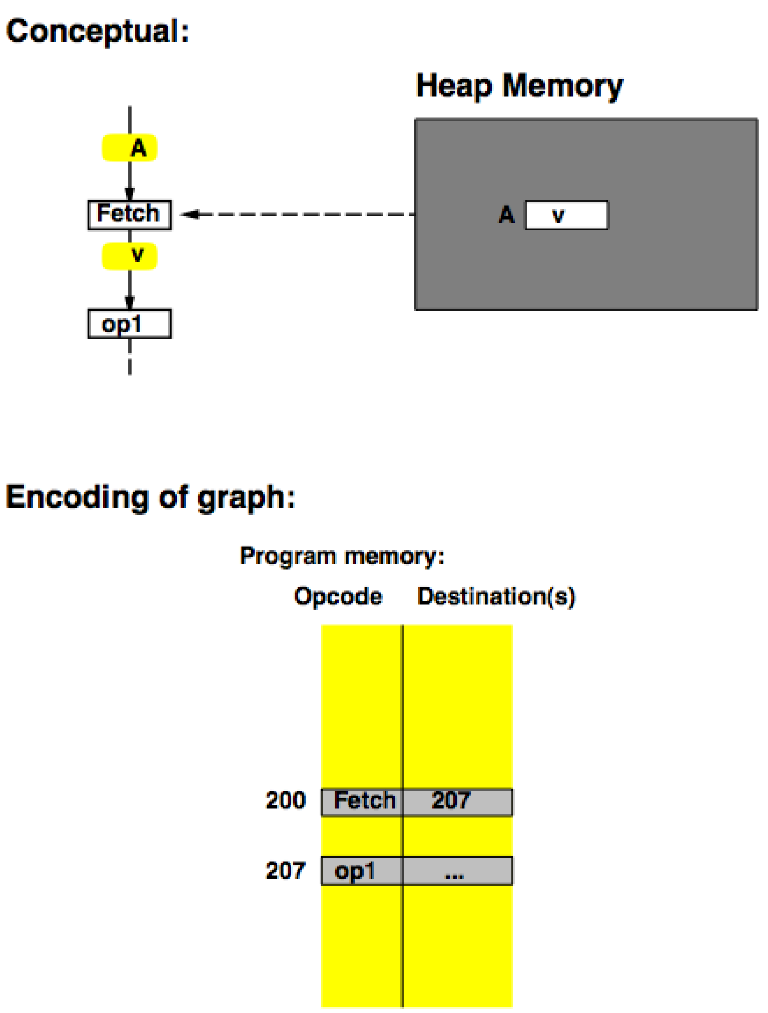
A "packet" containing:

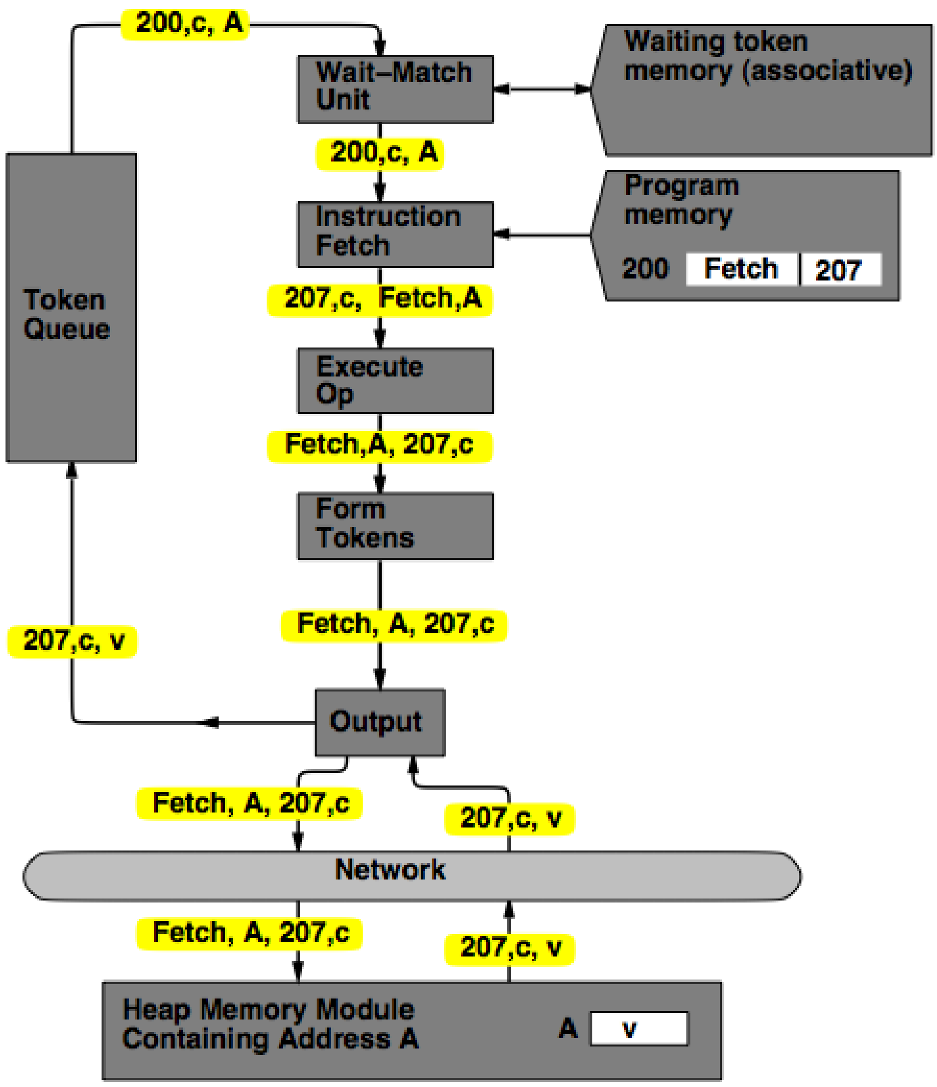
120R	Destination instruction address, Left/Right port
6.847	Value

TTDA Data Flow Example



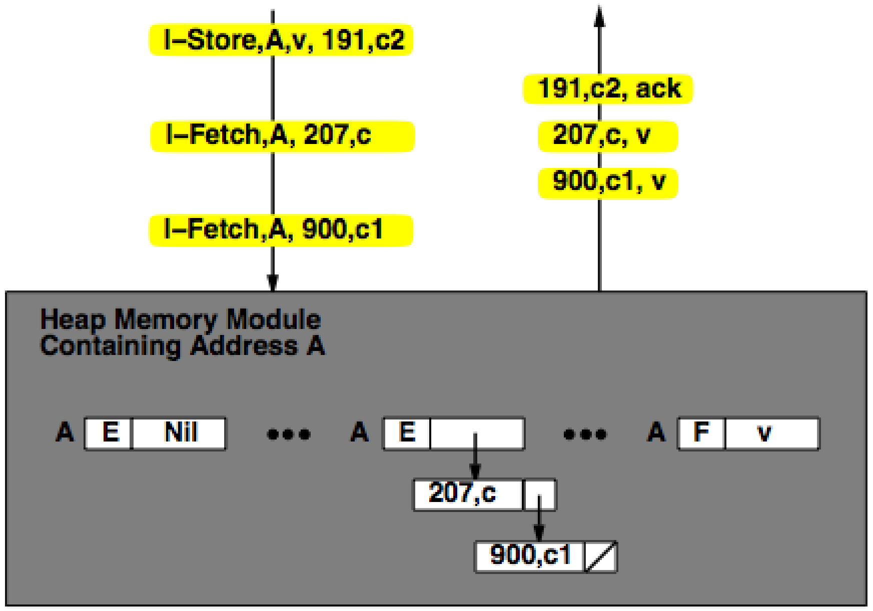
TTDA Data Flow Example



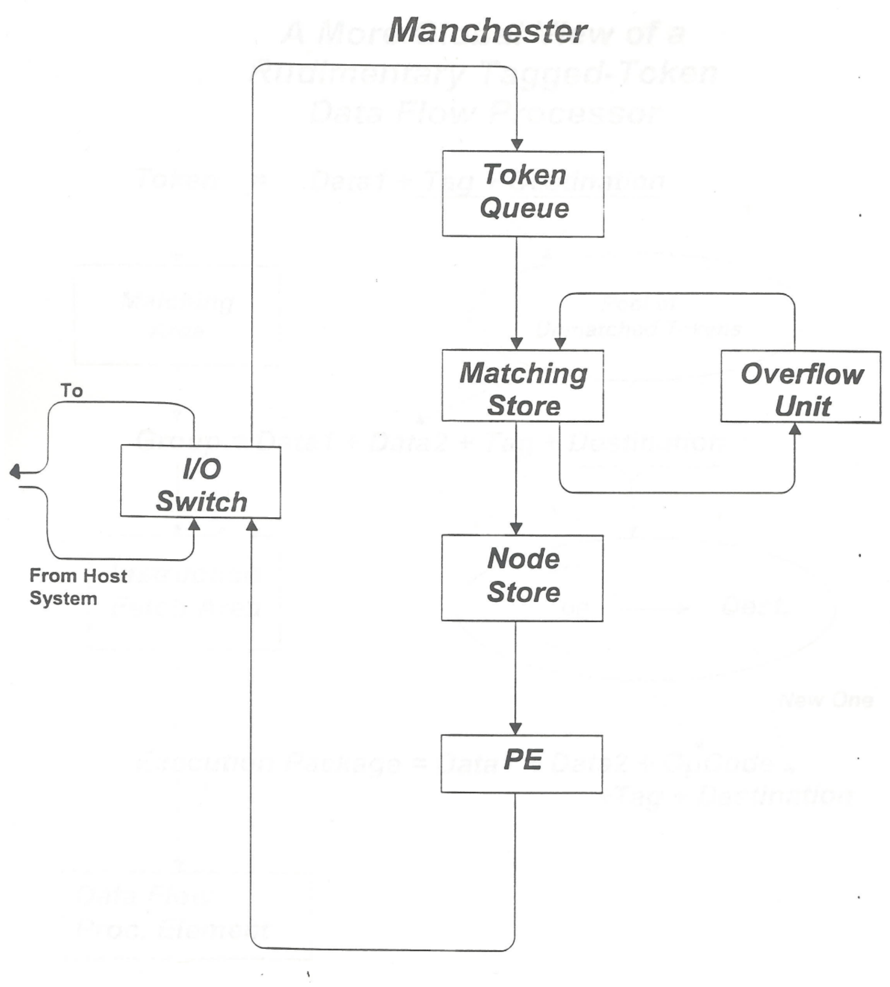


TTDA Synchronization

- Heap memory locations have FULL/EMPTY bits
- if the heap location is EMPTY, heap memory module queues request at that location When "I−Fetch" request arrives (instead of "Fetch"),
- Later, when "I−Store" arrives, pending requests are discharged
- No busy waiting
- No extra messages



Manchester Data Flow Machine



- Matching Store: Pairs together tokens destined for the same instruction
- Large data set → overflow in overflow unit
- Paired tokens fetch the appropriate instruction from the node store

Data Flow Summary

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)

- Data Flow at the ISA level has not been (as) successful

- Data Flow implementations under the hood (while preserving sequential ISA semantics) have been successful
  - Out of order execution
  - Hwu and Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," ISCA 1986.

Data Flow Characteristics

- Data-driven execution of instruction-level graphical code
  - Nodes are operators
  - Arcs are data (I/O)
  - As opposed to control-driven execution
- Only real dependencies constrain processing
- No sequential I-stream
  - No program counter
- Operations execute asynchronously
- Execution triggered by the presence of data
- Single assignment languages and functional programming
  - E.g., SISAL in Manchester Data Flow Computer
  - No mutable state

Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Implementing dynamic data structures difficult in pure data flow models
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Instruction cycle is inefficient (delay between dependent instructions), memory locality is not exploited

Combining Data Flow and Control Flow

- Can we get the best of both worlds?

- Two possibilities
  - Model 1: Keep control flow at the ISA level, do dataflow underneath, preserving sequential semantics
  - Model 2: Keep dataflow model, but incorporate control flow at the ISA level to improve efficiency, exploit locality, and ease resource management
    - Incorporate threads into dataflow: statically ordered instructions; when the first instruction is fired, the remaining instructions execute without interruption