Multithreaded Architectures
Lecture 3 of 4

Supercomputing '93 Tutorial

Friday, November 19, 1993

Portland, Oregon

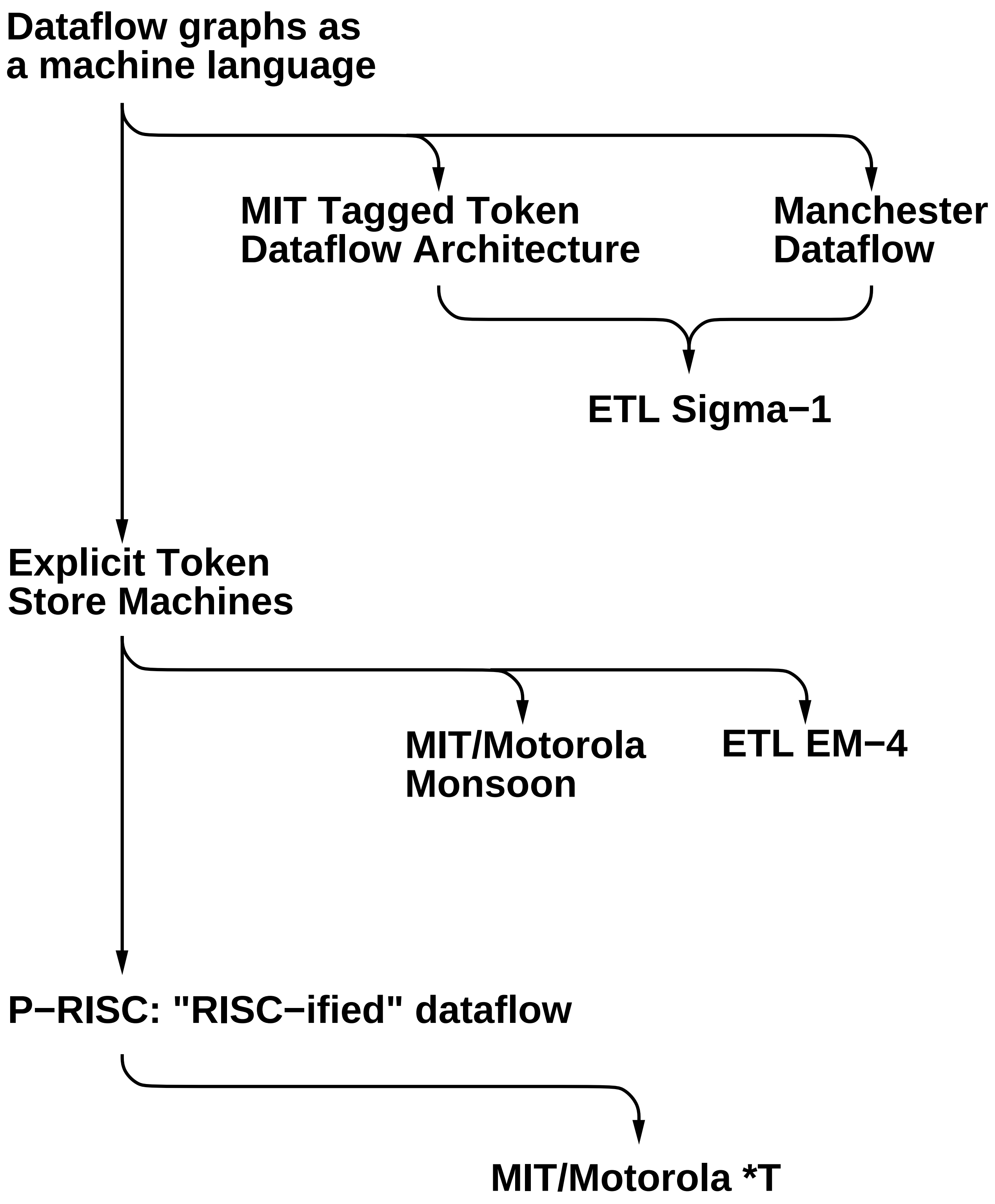
Rishiyur S. Nikhil

Digital Equipment Corporation

Cambridge Research Laboratory

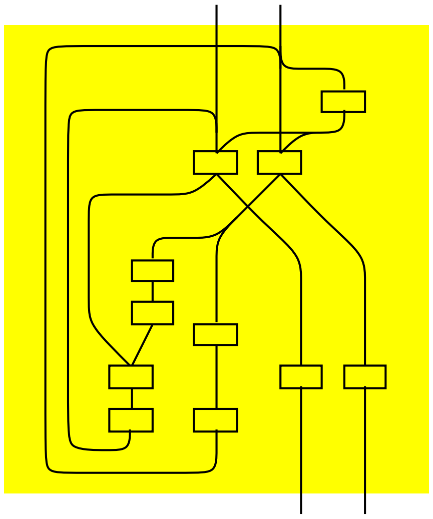
Overview of Lecture 3

Multithreading: a Dataflow Story


Dataflow graphs as a machine language
MIT Tagged Token Dataflow Architecture
Manchester Dataflow
ETL Sigma−1
Explicit Token Store Machines
MIT/Motorola Monsoon
ETL EM−4
P−RISC: "RISC−ified" dataflow
MIT/Motorola *T

Dataflow Graphs as a Machine Language

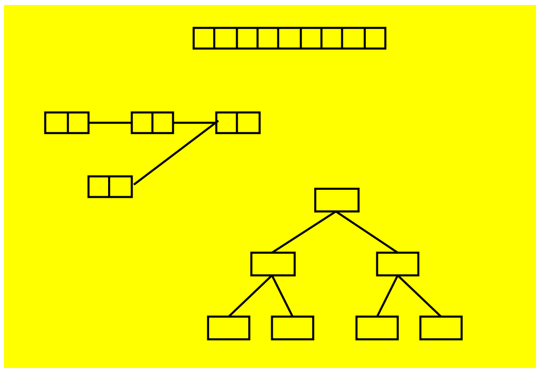
Code:
Dataflow Graph
(interconnection
of instructions)



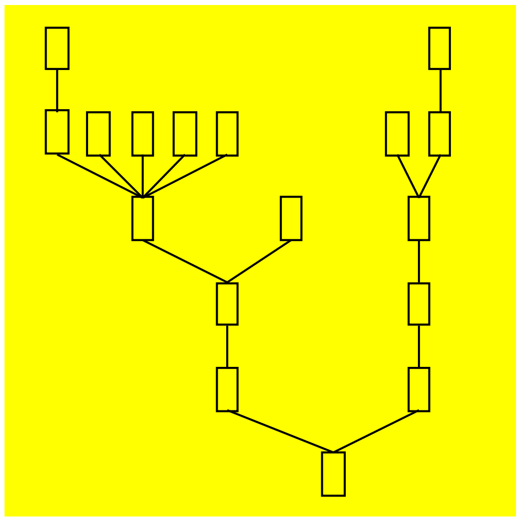
- **Conceptually, "tokens" carrying values flow along the edges of the graph.**
- **Values on tokens may be memory addresses**
- **Each instruction:**
  - **Waits for tokens on all inputs (and possibly a memory synchronization condition)**
  - **Consumes input tokens**
  - **Computes output values based on input values**
  - **Possibly reads/writes memory**
  - **Produces tokens on outputs**
- **No further restriction on instruction ordering**

Memories:
Two major components

Heap, for data structures
(e.g., I−structure memory)



"Stack"
(contexts, frames, ...)

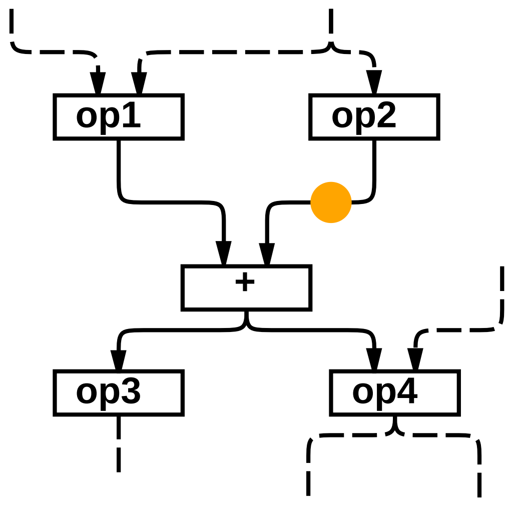


A common misconception:

"Dataflow graphs and machines cannot express side effects, and they only implement functional languages"

Encoding Dataflow Graphs and Tokens

Conceptual



Encoding of graph

Program memory:

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

Encoding of token:

A "packet" containing:

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

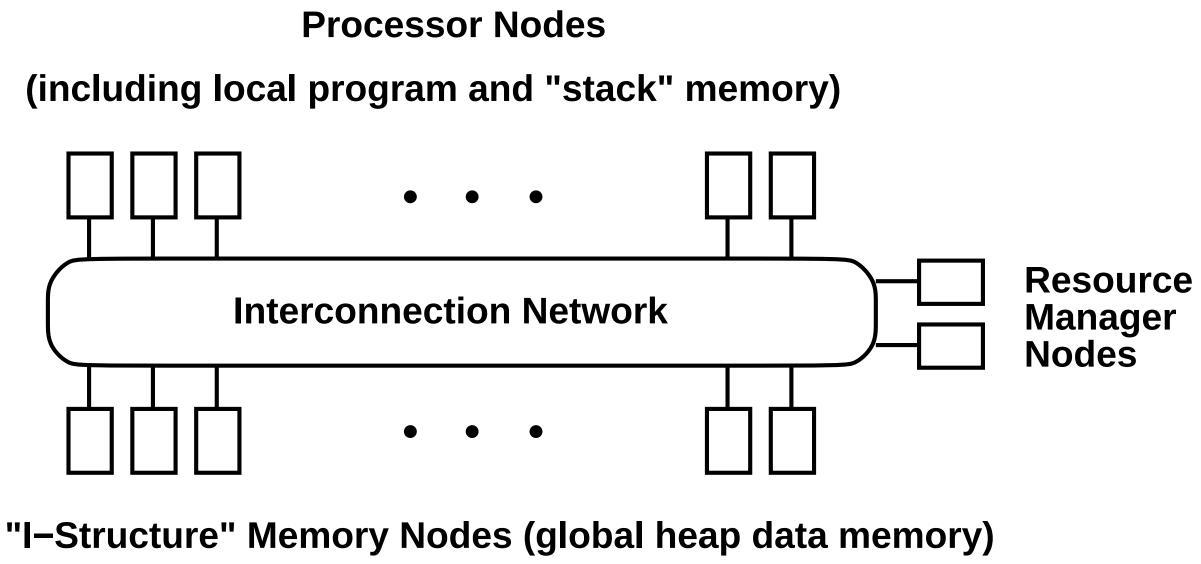
Re−entrancy ("dynamic" dataflow):

- 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

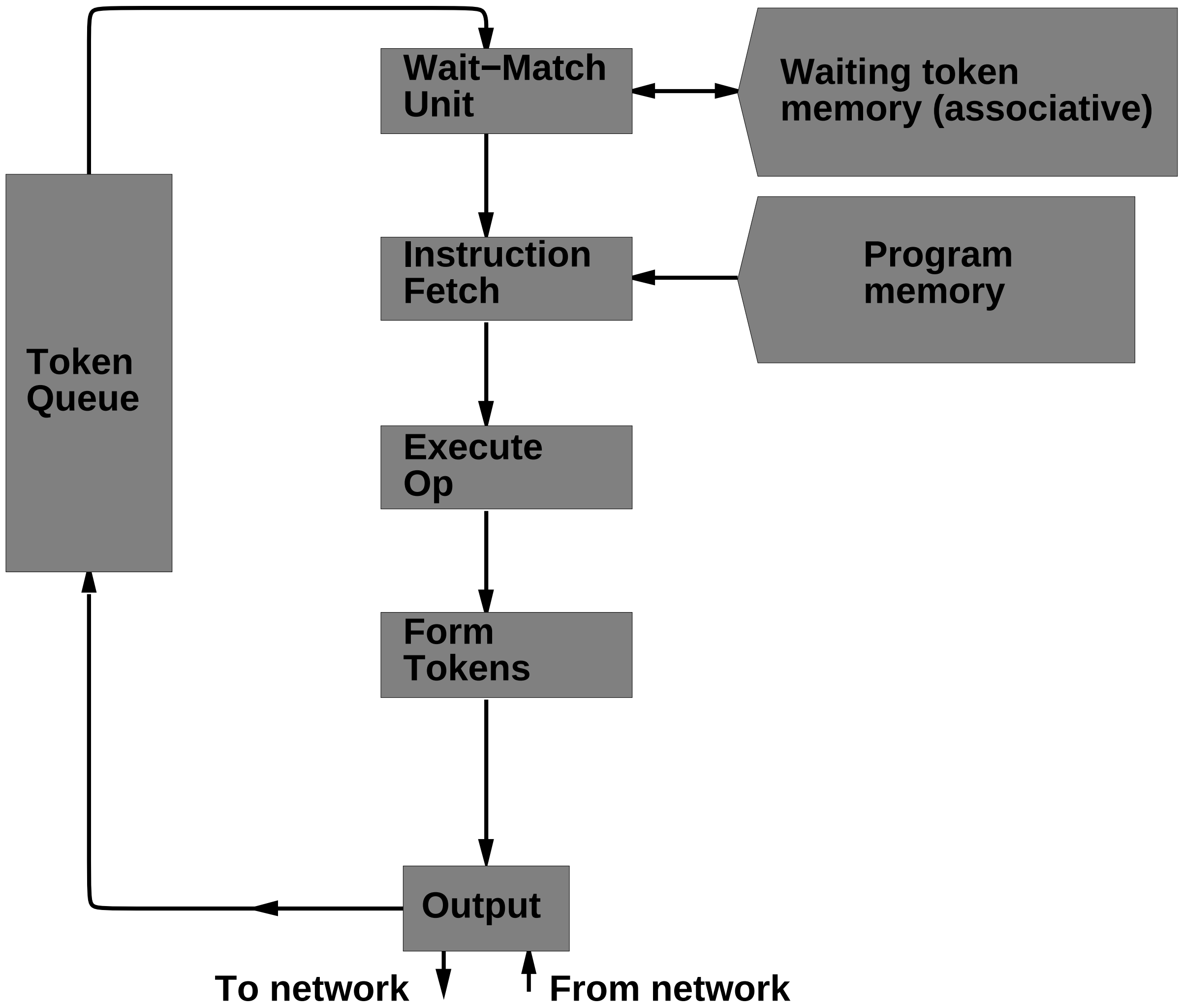
MIT Tagged Token Dataflow Architecture

- **Designed by Arvind et. al., MIT, early 1980's**
- **Simulated; never built**
- **Global view:**


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

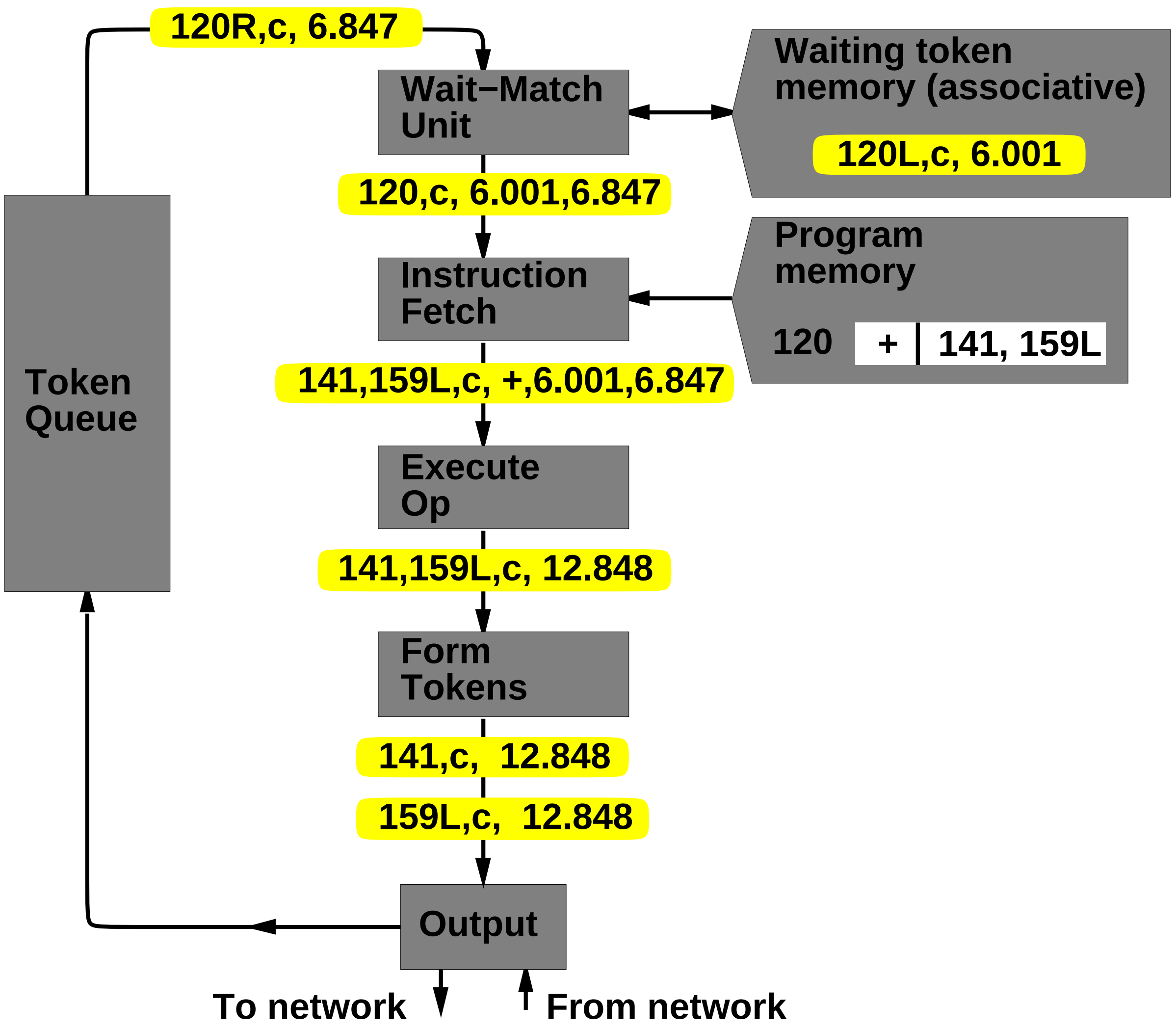
- **Resource Manager Nodes responsible for**
  - **Function allocation (allocation of context identifiers)**
  - **Heap allocation**
  - **etc.**
- **Stack memory and heap memory: globally addressed**

MIT Tagged Token Dataflow Architecture Processor



- **Wait−Match Unit:**
  - **Tokens for unary ops go straight through**
  - **Tokens for binary ops: try to match incoming token and a waiting token with same instruction address and context id**
    - **Success: Both tokens forwarded**
    - **Fail: Incoming token −−> Waiting Token Mem, Bubble (no−op) forwarded**

MIT Tagged Token Dataflow Architecture Processor Operation

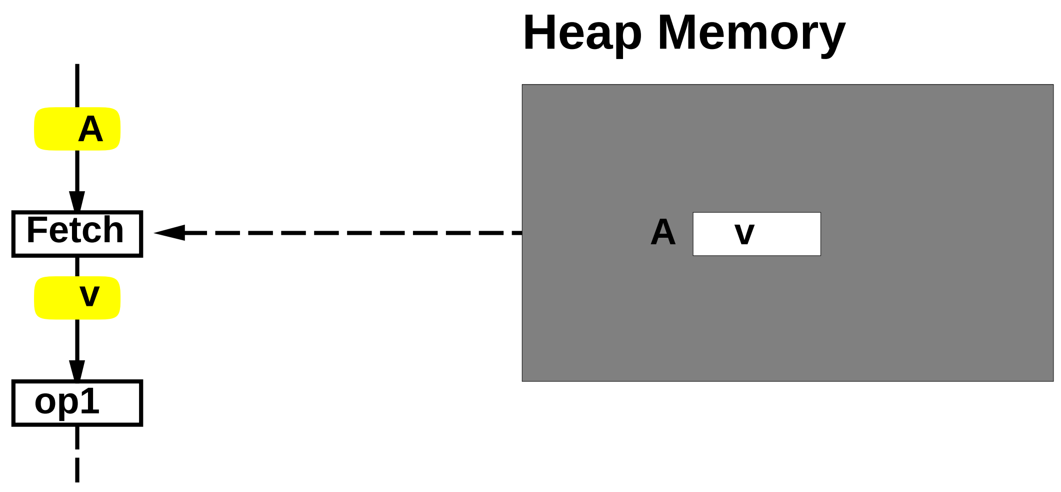


- Output Unit routes tokens:
  - Back to local Token Queue
  - To another Processor
  - To heap memory

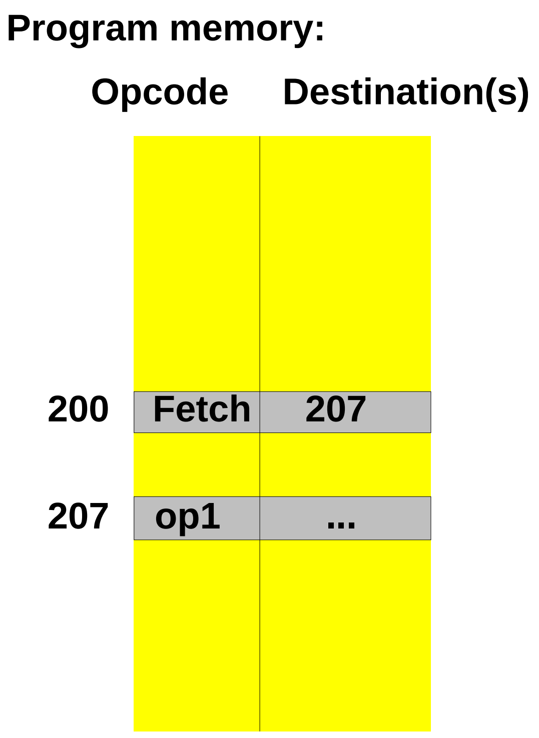
  based on the addresses on the token
- Tokens from network are placed in Token Queue

MIT Tagged Token Dataflow Architecture Support for "Remote Loads"

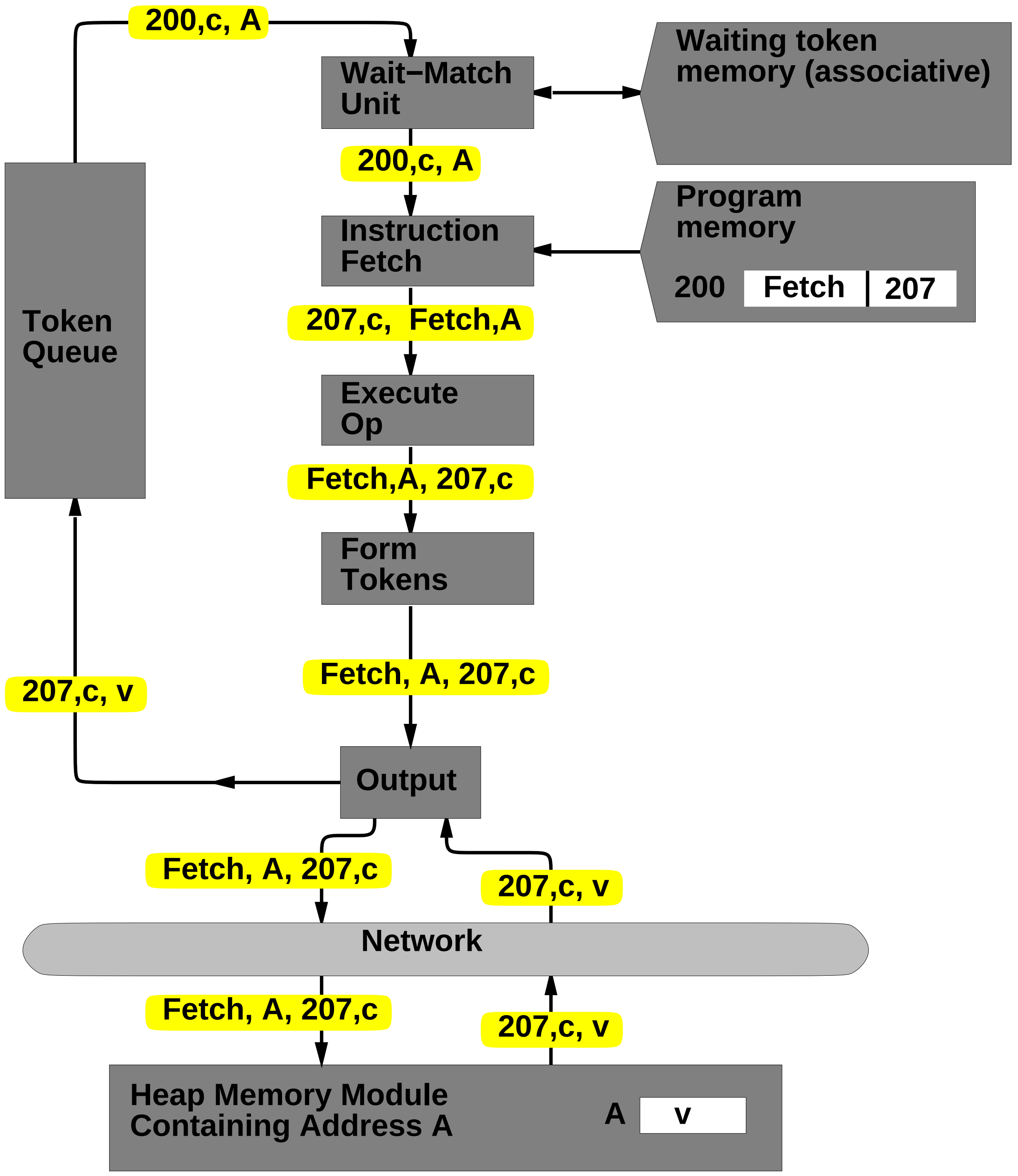
Conceptual:



Encoding of graph:



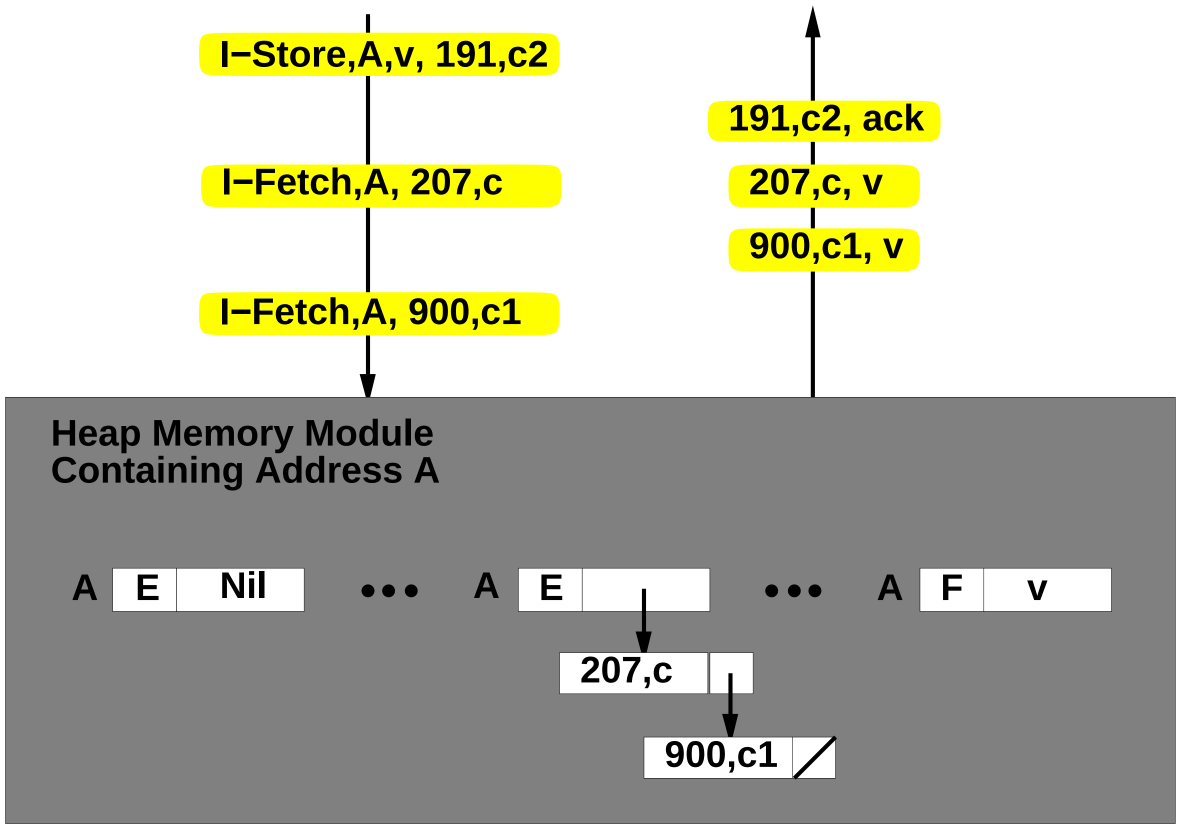
MIT Tagged Token Dataflow Architecture Support for "Remote Loads"



- Multiple remote loads are no problem:
  - Can be issued in parallel
  - "Join" of responses is implicit in Wait−Match

MIT Tagged Token Dataflow Architecture Support for "Synchronizing Loads"

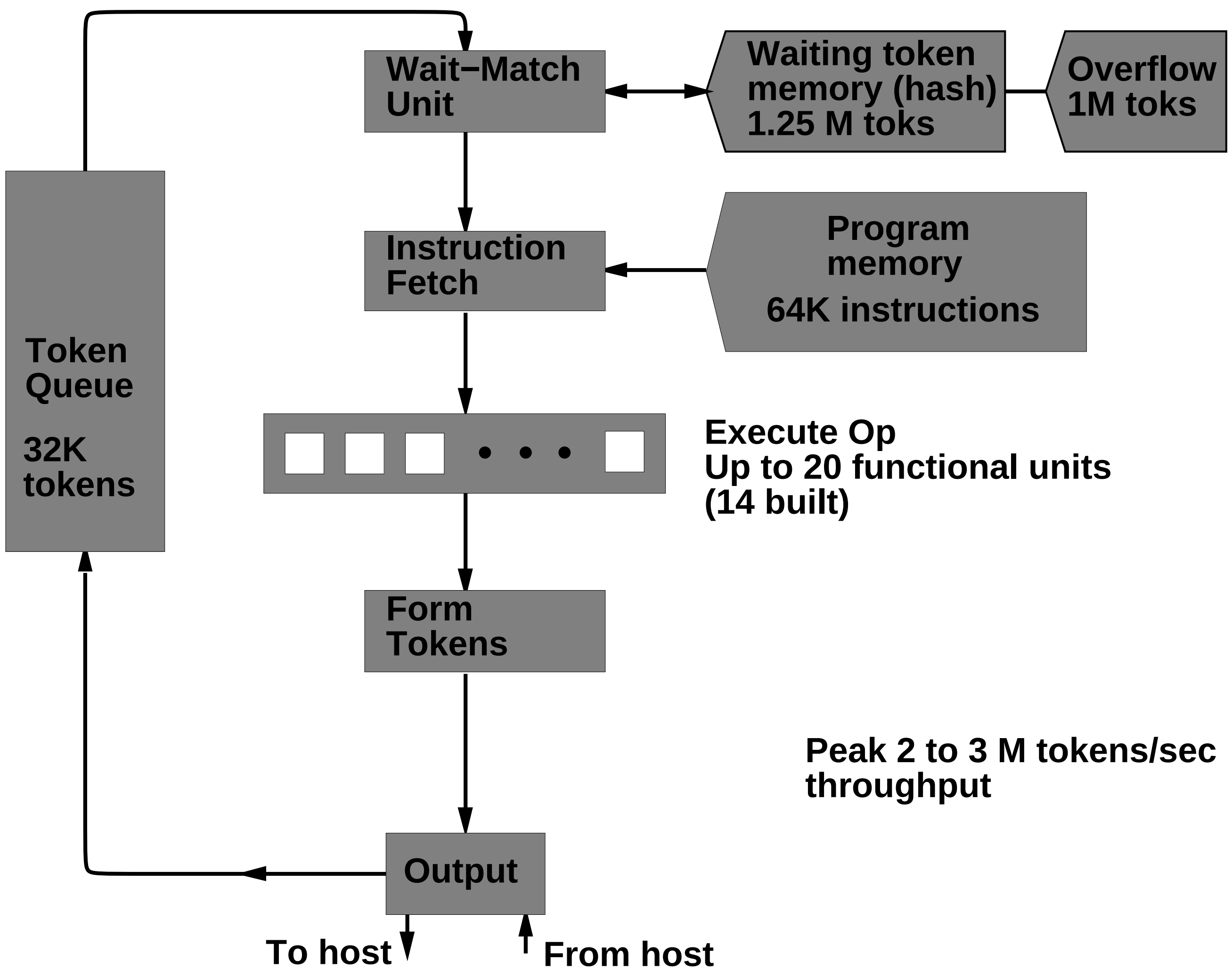
- **Heap memory locations have FULL/EMPTY bits**
- **When "I−Fetch" request arrives (instead of "Fetch"), if the heap location is EMPTY, heap memory module queues request at that location**
- **Later, when "I−Store" arrives, pending requests are discharged**
- **"I−structure semantics"**
  Note: no busy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



MIT Tagged Token Dataflow Architecture Assessment

- **Waiting Token Memory plays role of "stack memory"**
- **Wait−Match Unit problem:**
  - **Not easy to build large, fast, associative memories**
  - **Nevertheless, Manchester Dataflow and ETL Sigma−1 did so, using hardware hashing**
- **As in HEP, single−thread performance limited to 1/D (D = latency of processor pipeline)**
- **No access to "high speed" memory (registers)**
  - **This was fixed in Monsoon (in retrospect, it can also be done in TTDA)**

The Manchester Dataflow Machine



- John Gurd, et. al. at Manchester University, UK
- Operational early−mid 1980's
- Asynchronous (separate clock for each module)
- Performance comparable to Vax 11/780

The Manchester Dataflow Machine Wait−Match Operations

- **Normal binary matching; plus more:**
  - **"Sticky": Binary match, but leave left−partner in Waiting Token Memory**

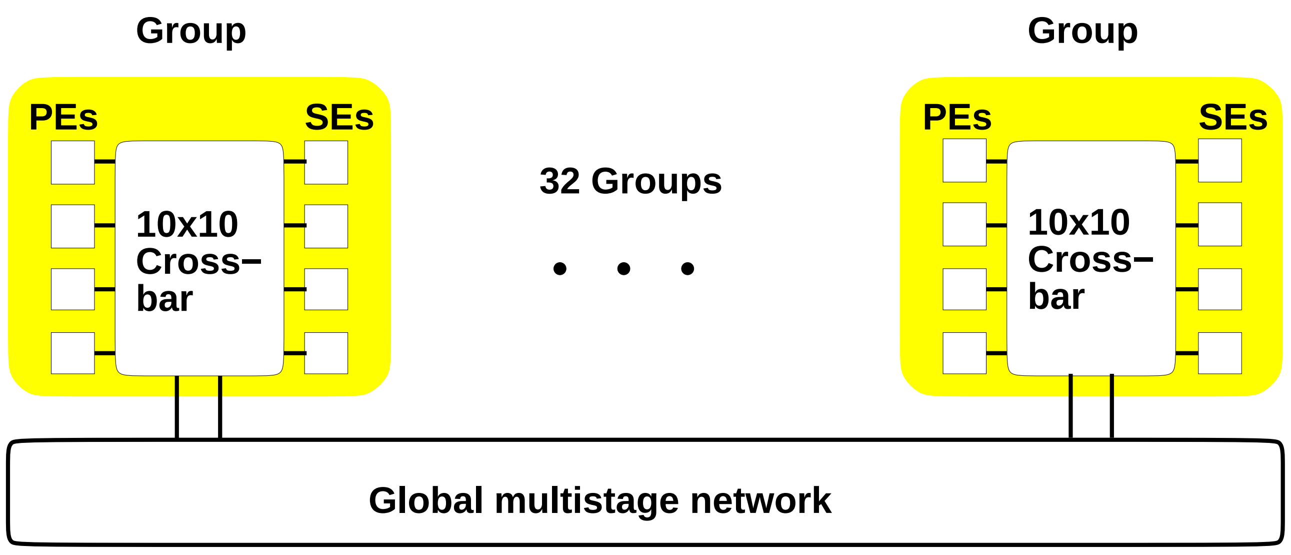
    E.g., to simulate I−structure memory
    - **Matching key is heap address**
    - **Left−partner is "write", carries data value**
    - **Right−partners are "reads", wait for write**

    In fact, since no structure memory was built for the Manchester machine, this was how it was done (at considerable overhead)
  - **"Constant": unary operations**
    - **"Constant Write": unconditionally store token into Waiting Token memory**
    - **"Constant Read": unconditionally read token from Waiting Token memory**

    E.g., to implement loop constants, avoiding overhead of circulating them
- **Similar mechanisms exist in TTDA, Sigma−1, Monsoon**

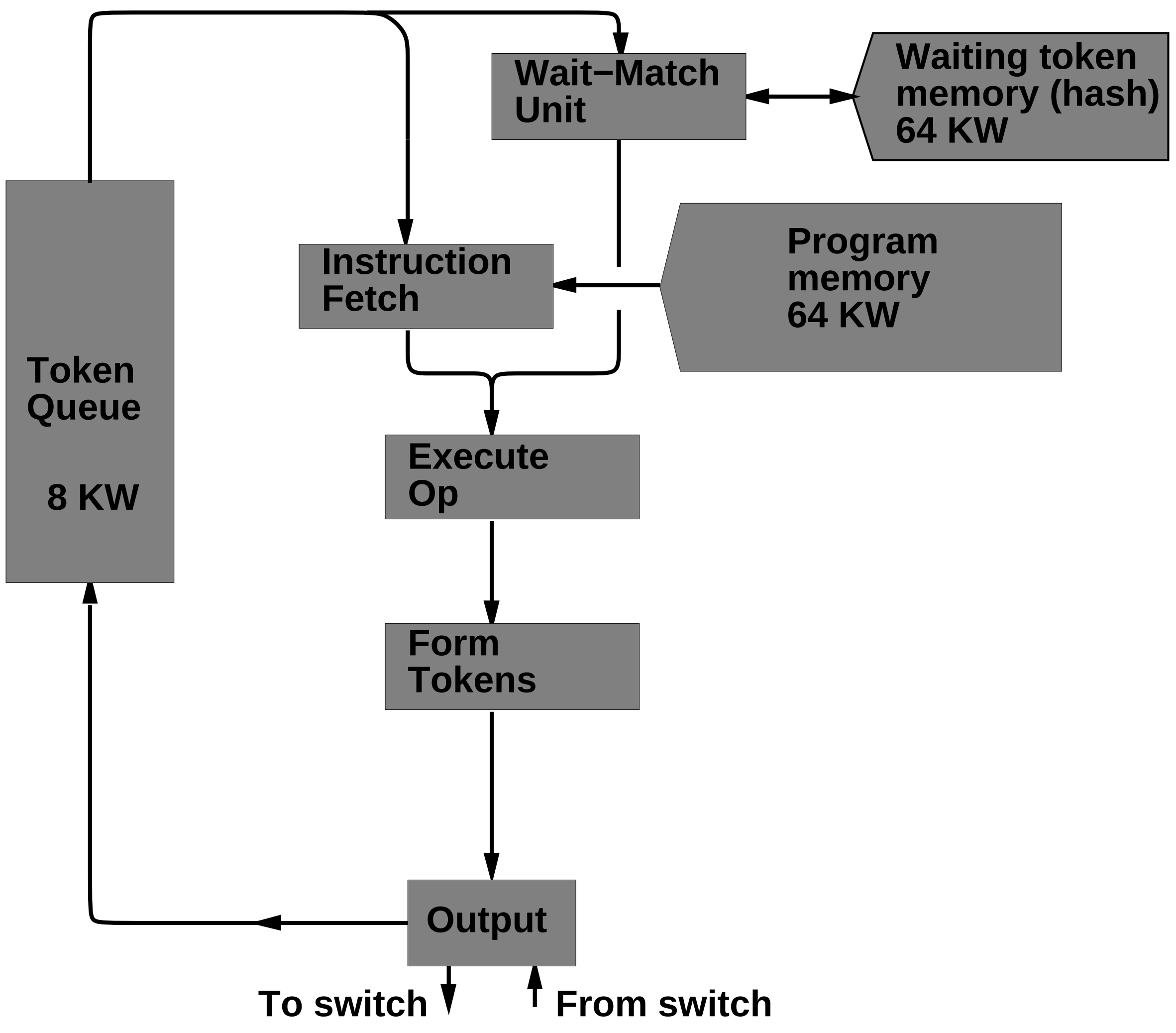
The ETL Sigma−1

- Kei Hiraki, Toshio Shimada, et. al.,
  Electrotechnical Laboratory, Tsukuba, Japan
- Global organization:



- Fully Synchronous, 10 MHz clock
- PE = "Processing Element"
- SE = "Structure Element" (I−structure Memory)
  256 KW each
- Prototype PE operational 1984
  4 PE, 4 Structure Memory Group operational 1986
  128 PE, 128 Structure Memory system operational 1987
- 128 PE system achieved 170 MFLOPS

The ETL Sigma−1 Processing Element Organization



- Instruction−fetch and Wait−match done in parallel

- Major problem: lack of high−level language

Explicit Token Store (ETS)

(Papadopoulos, Culler, Sakai, ...)

Basic ideas (elimination of associative Wait−Match):

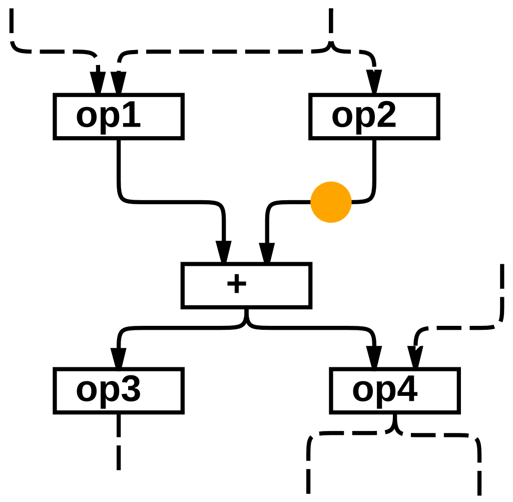
- **Waiting Token Memory directly addressed (like normal RAM), plus FULL/EMPTY bits**
- **Allocate a "frame" in Waiting Token Memory for each function invocation**
- **Frame Pointer (FP) is "context identifier"**
- **A token for a binary instruction meets (matches) its partner at a fixed offset in the frame**

  (offset determined at compile−time)
- **ETS is used in the MIT/Motorola Monsoon**

  and in the ETL EM−4 ("direct matching")

Explicit Token Store Encoding

Conceptual



Encoding of graph

Program memory:

	Op–code	Destination(s)	Rendezvous
109	op1	120L	1
113	op2	120R	
120	+	141, 159L	2
141	op3	...	
159	op4	... , ...	3

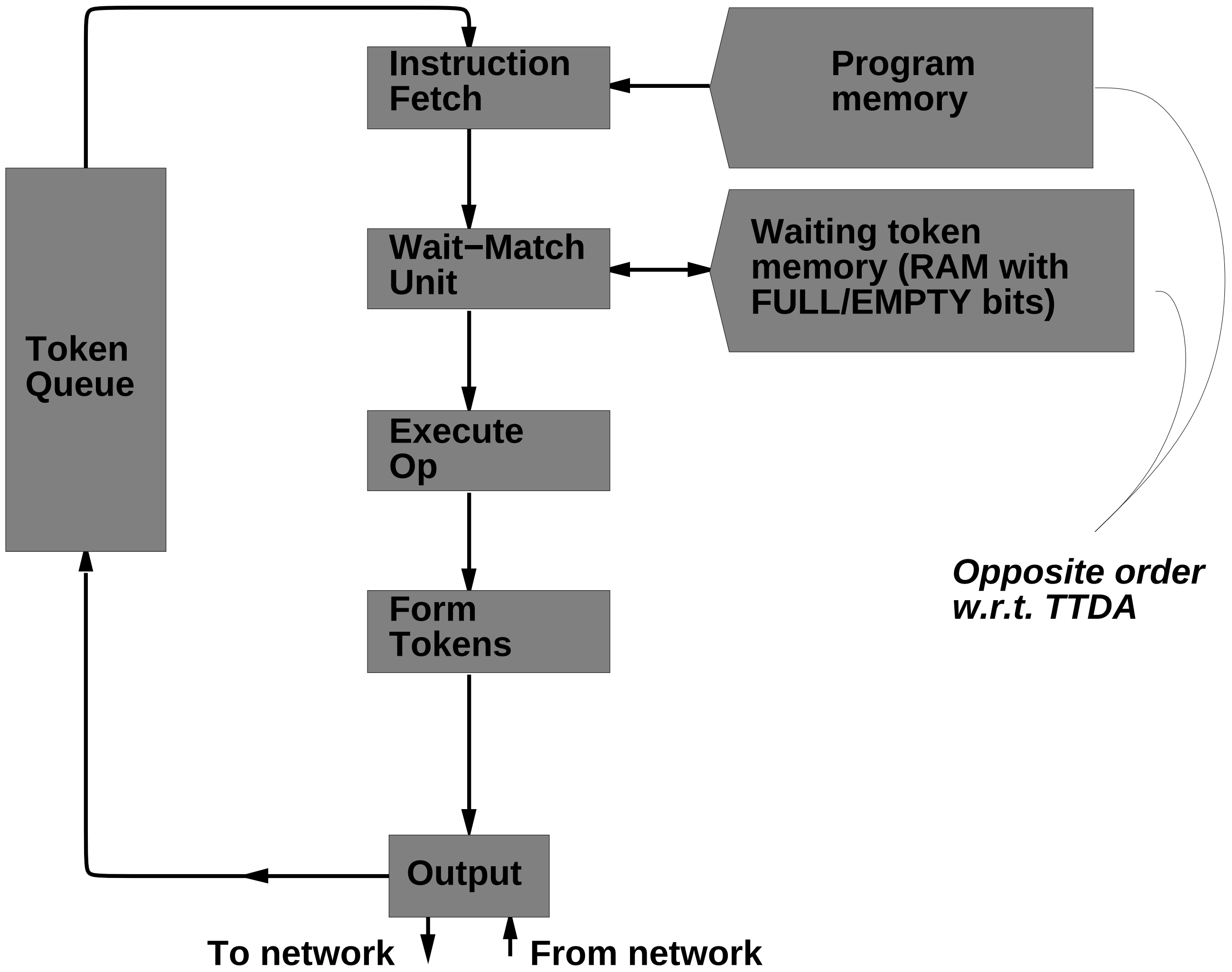
Same as TTDA

Encoding of token:

A "packet" containing:

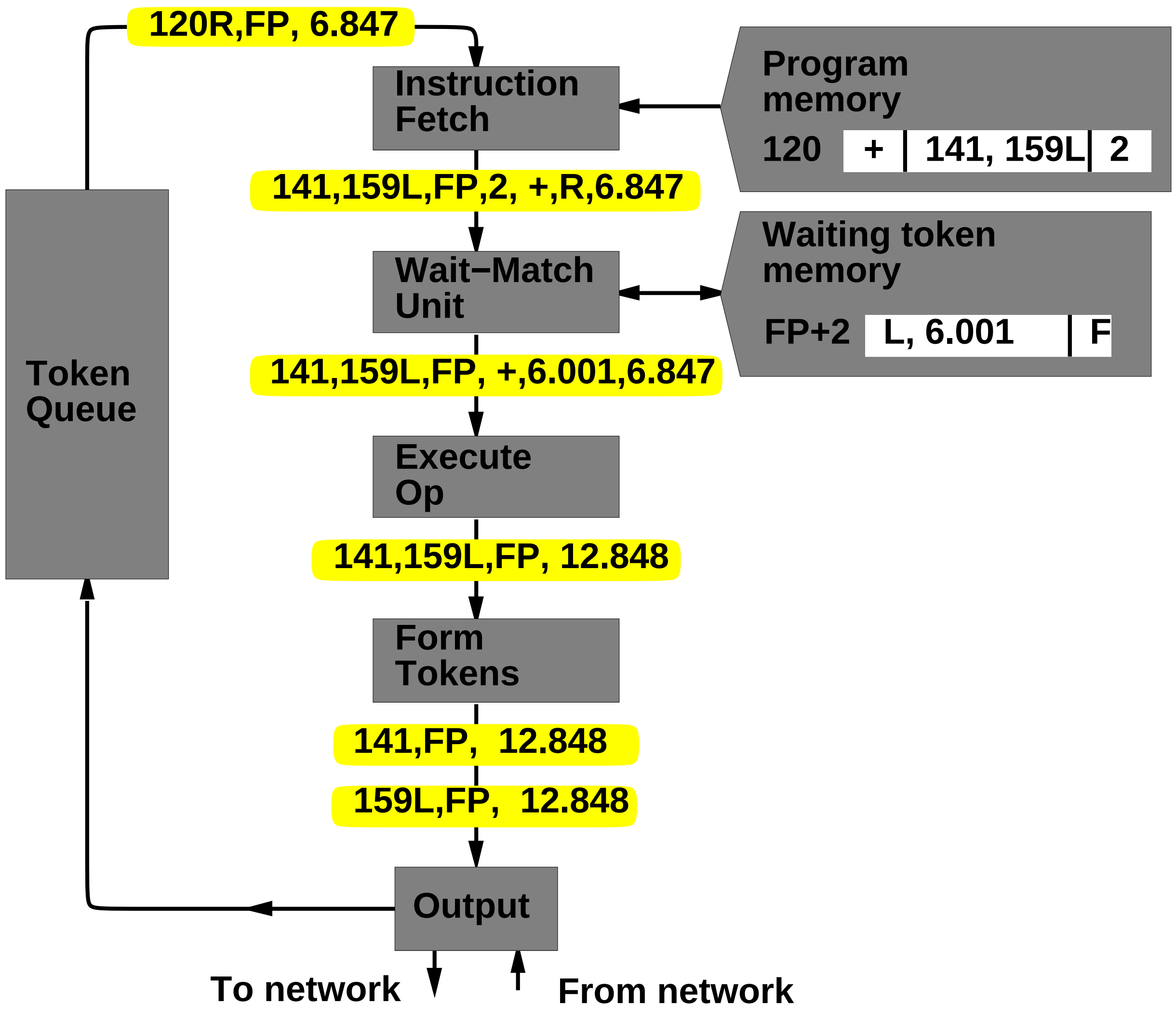
120R	Destination instruction address, Left/Right port
FP	Frame Pointer
6.847	Value

MIT/Motorola Monsoon Processor (simplified)



- **Wait−Match Unit:**
  - **Tokens for unary ops go straight through**
  - **Tokens for binary ops: FP (on incoming token) and rendezvous offset (in instruction) specify location where partner may be waiting:**
    - **FULL:** **Extract partner, both tokens forwarded**
    - **EMPTY:** **Incoming token −−> location, Bubble (no−op) forwarded**

MIT/Motorola Monsoon Processor Operation (simplified)

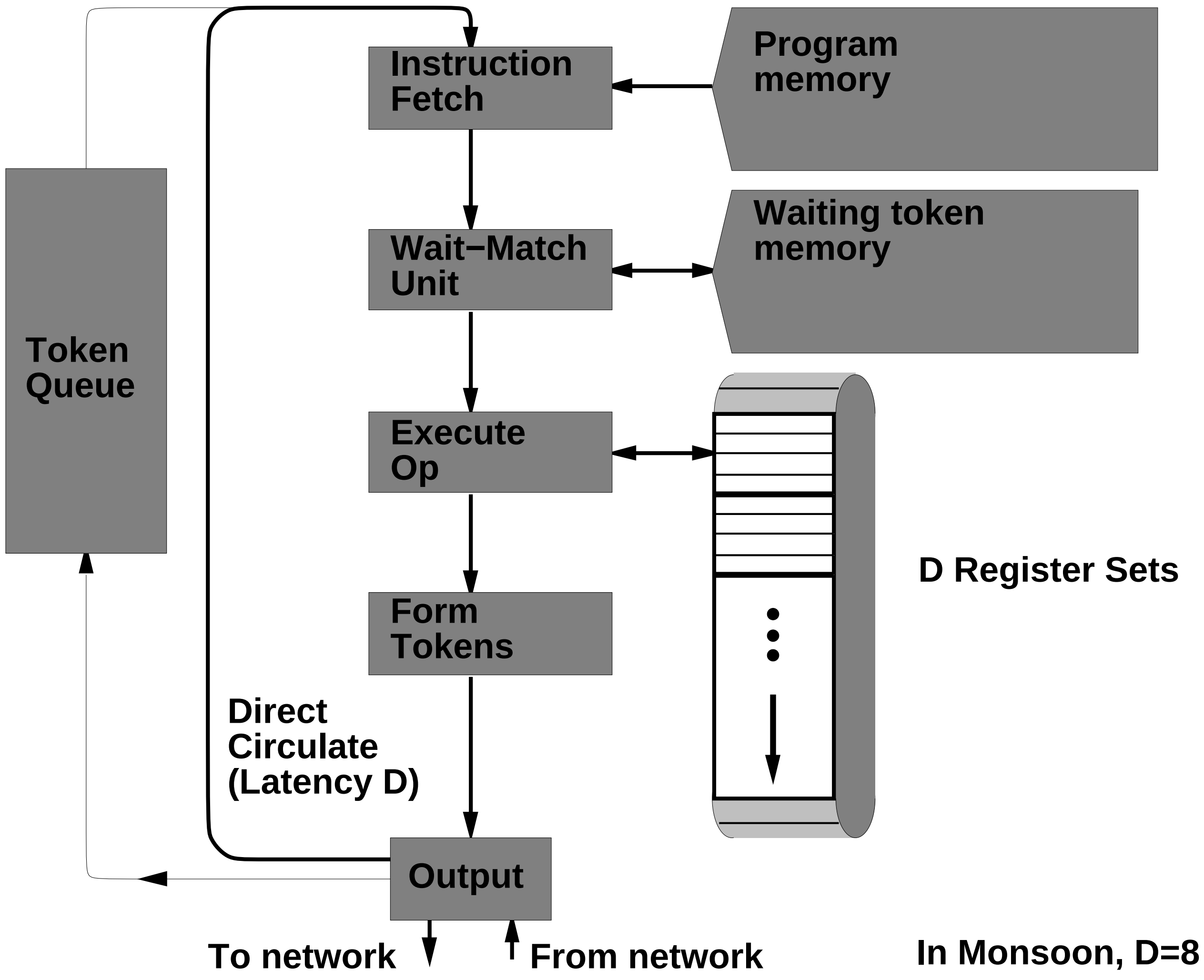


MIT/Motorola Monsoon Token Storage Issue

- **TTDA: size of allocated Waiting Token Memory = # of waiting tokens**
- **Monsoon: size of allocated Waiting Token Memory = # of allocated frames**

  (which may be sparsely populated with tokens)

MIT/Motorola Monsoon Threads and Thread−Registers

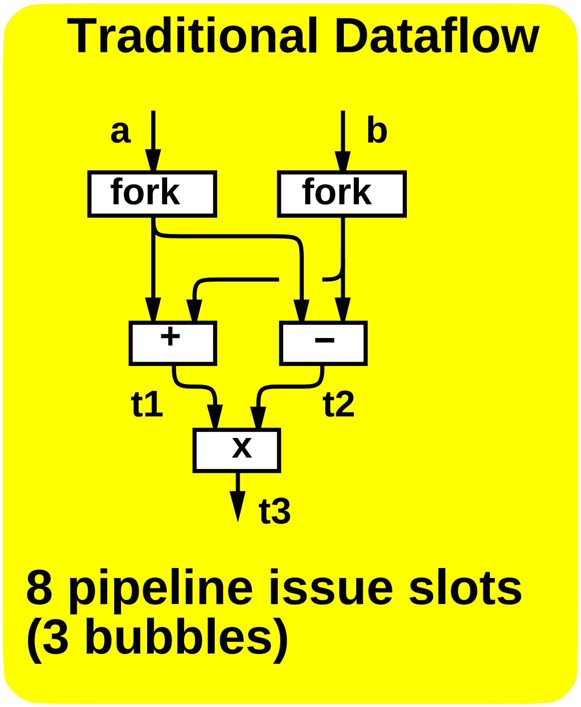


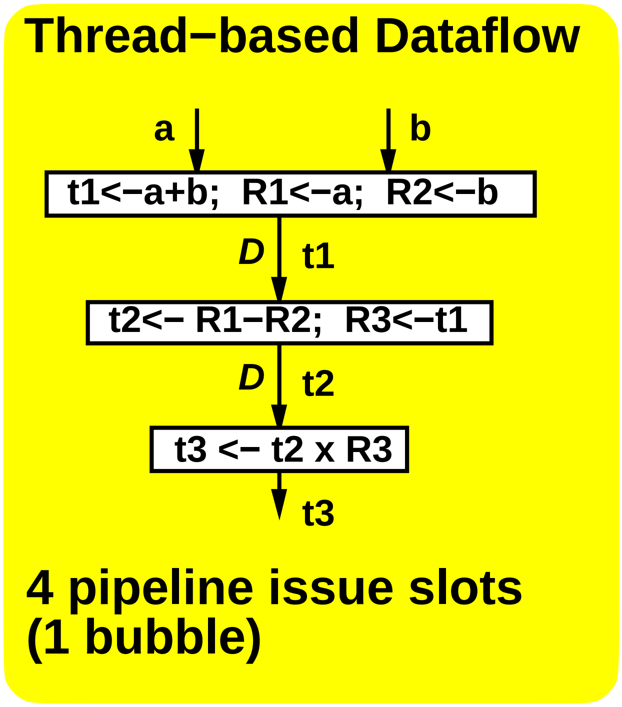
- Up to one local destination may be "Direct"
- Register sets circulate in synchrony with Direct Circu− late path; each of D threads has exclusive register set
- Opcodes generalized for register sources and dests (in addition to values on tokens)
- Registers not saved when thread evaporates (like Hybrid)

MIT/Motorola Monsoon Thread−based Programming (Concept)

- **Assume, for each instruction:**
  - **Four data sources (two token values, two regs)**

    However, note:
    - **If there was a Wait−Match, then**
      - **Both token sources are valid**
      - **Registers contain garbage**
    - **If there was no Wait−Match, then**
      - **One token source is valid**
      - **Registers are valid if inside a "direct" thread**
  - **Three data destinations (one token value, two regs)**
  - **One ALU op, Two moves**
- **Now consider the expression: "(a+b)x(a−b)"**





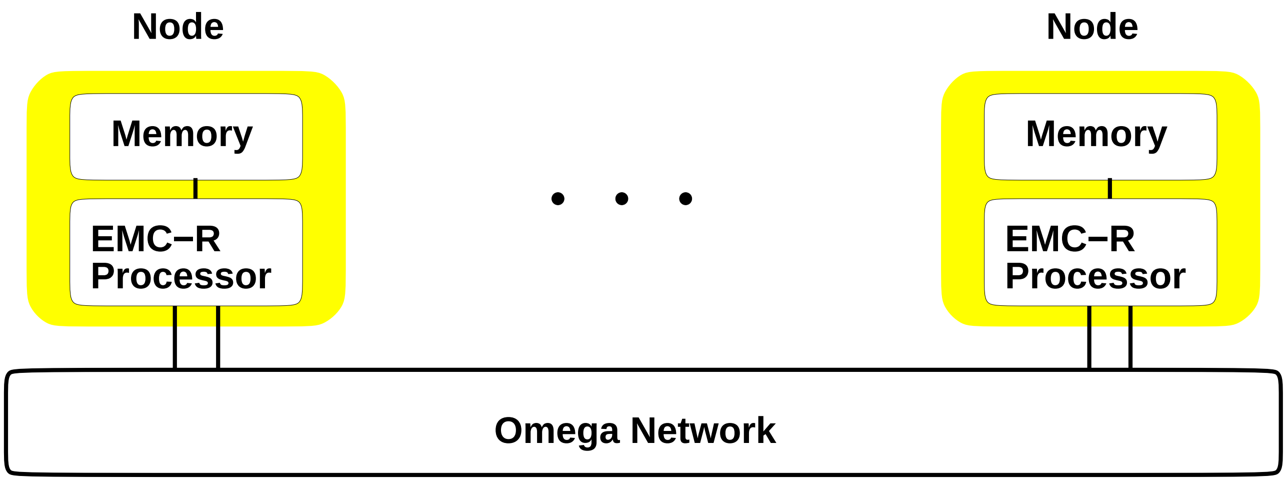
(each arrowhead = 1 issue slot, each join = 1 bubble)

MIT/Motorola Monsoon
Some Details

- **Principal architect: Greg Papadopoulos**
- **Word size: 64 bits data + 8 bits type tag**
- **Processor:**
  - **10 MHz clock**
  - **256 KW Program Memory (32b wide, max 1MW)**
  - **256 KW Waiting Token Memory (frames)**
    (word + 3 presence bits)
  - **Two x 32 Ktoken queues (system, user)**
  - **Each thread has 4 registers**
- **I−Structure Memory Modules:**
  - **4 MW (word + 3 presence bits, max 16 MW)**
  - **5 M requests/sec**
- **Network:**
  - **Multistage, pipelined**
  - **Packet Routing Chips (PaRC, 4 x 4 crossbar)**
  - **4 M tokens/sec/link (100 MB/sec)**
- **Max configuration of prototype: 8 x 8**
  (8 Processors, 8 I−Structure Memory Modules)
- **1 x 1 configurations operational Sept 1990**
  8 x 8 configurations operational Fall 1991

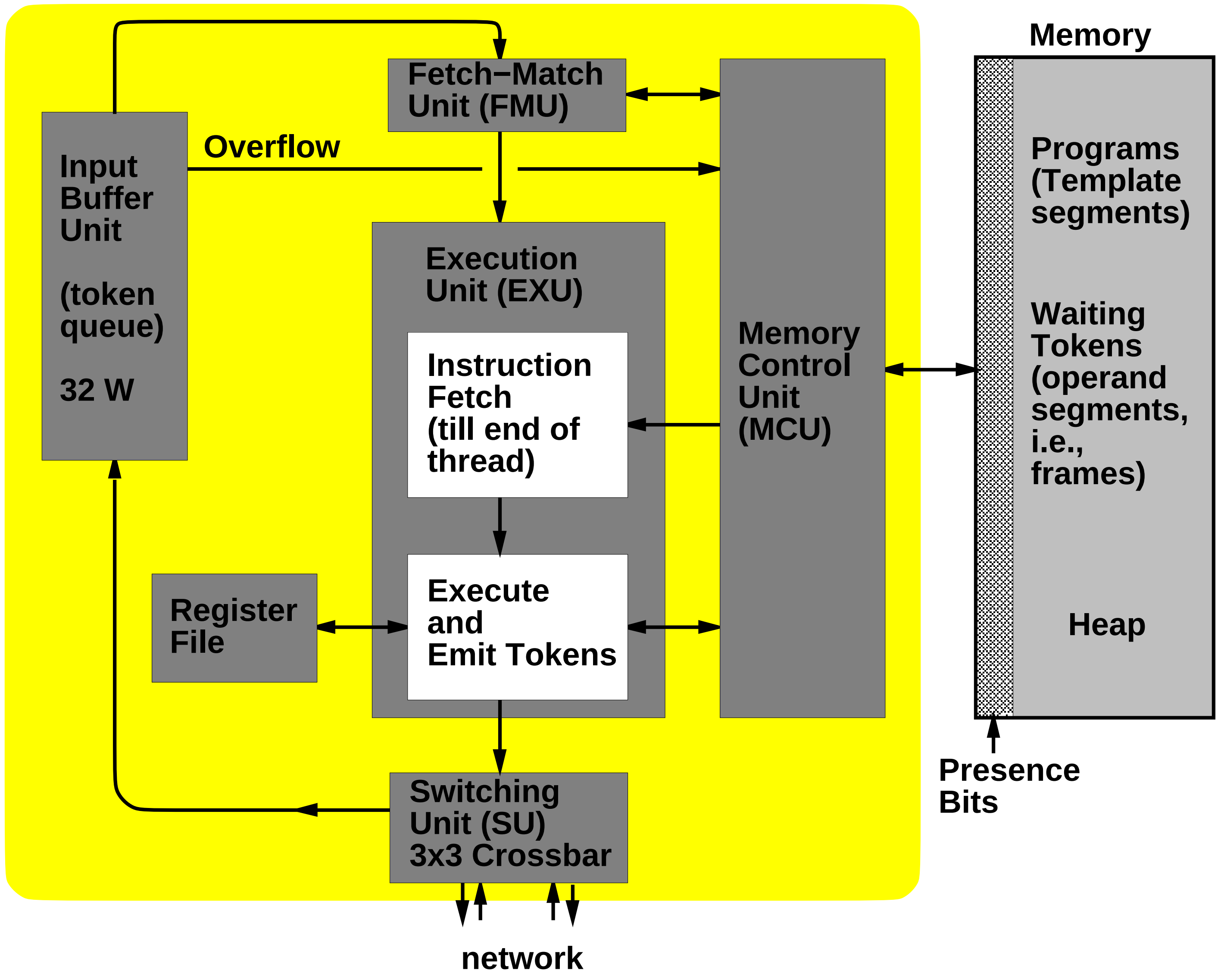
The ETL EM−4

- Sakai, Yamaguchi et. al.
  Electrotechnical Laboratory, Tsukuba, Japan
- Global organization:



- Fully Synchronous, 12.5 MHz clock
- EMC−R: single chip processor (no floating point)
  (includes one switch of the network)
- Each node also plays the role of I−structure Memory
  Mem/node: 1.31 MB SRAM (5.2 MB max)
- Designed for 1024 nodes
  80 node prototype operational since May 1990
- Prototype achieved 815 MIPS (80x80 matrix multiply)

The ETL EM−4
EMC−R Organization



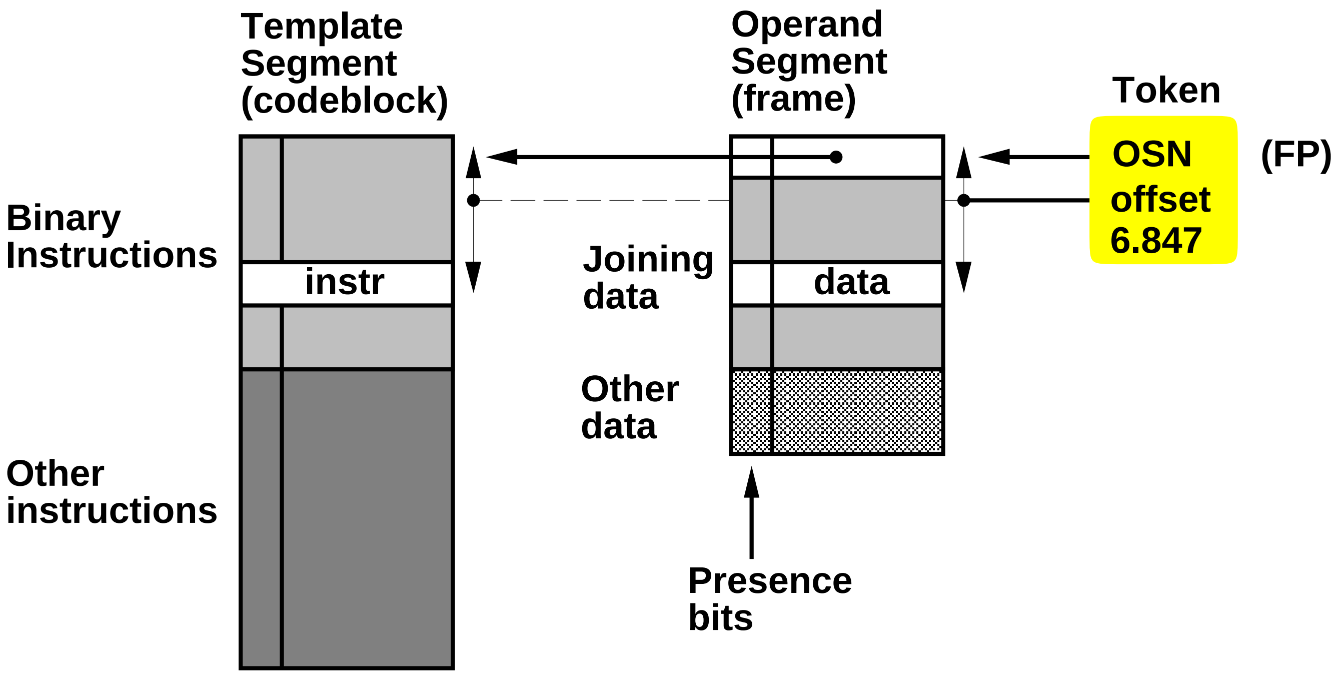
Fetch−Match Unit (FMU)
Overflow
Input Buffer Unit (token queue) 32 W
Execution Unit (EXU)
Instruction Fetch (till end of thread)
Execute and Emit Tokens
Register File
Memory Control Unit (MCU)
Memory
Programs (Template segments)
Waiting Tokens (operand segments, i.e., frames)
Heap
Presence Bits
Switching Unit (SU) 3x3 Crossbar
network

EXU executes "Strongly Connected Blocks" (threads)

- **When 1 or 2 tokens arrive from FMU:**
  - **Their values become sources for the instruction**
  - **Instruction can emit token or write to register**
  - **Instructions fetched continually using traditional sequencing (PC+1, Branch) until "stop" flag; Then, another pair of tokens is accepted from FMU**
  - **Each instruction in a thread specifies the two sources for the next instruction in the thread**

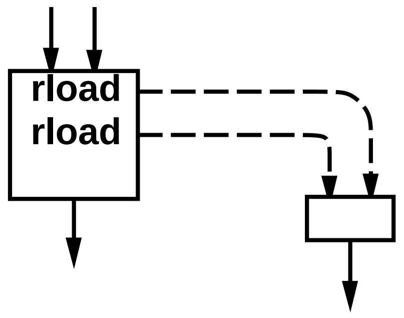
The ETL EM−4 Tokens and Matching

- **Same idea as in Monsoon; different encoding:**



- **Extra memory access to fetch instruction**
- **Sharing the offset prevents re−use of slots for joins.**

Example:



The two joins cannot overlap in time

- **Joins that don't carry data waste a word**
- **32 bit data words, 38 bit instruction words**

The ETL EM−4 "Remote Loads", "Synchronizing Loads"

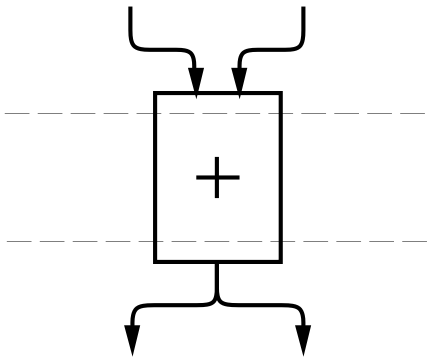
- **Suppose Node N1 performs a "Remote Load" or a "Synchronizing Load" on a location in Node N2**
  - **N1 behavior (issuing, handling response) is like Monsoon**
  - **N2 behavior is like J−Machine**
- **At Node N2:**
  - **No separate "Structure Memory": part of each node's memory is part of global heap**
  - **Incoming remote load request handled by EXU: triggers short thread that reads, emits response**
  - **Synchronizing loads: thread also tests FULL/EMPTY bits, enqueues request on EMPTY, if necessary (i.e., interpret I−structure semantics)**
  - **Latency affected by pending threads ahead of request (unbounded)**
- **At Node N1:**
  - **For multiple remote loads, "join" of responses handled with usual FMU synchronization (Unlike J−Machine, which uses CFUT traps)**

ETL EM−4
What next?

- **EM−X: upgrade of EM−4**
  - **Newer technology, floating point**
  - **80 PEs, > 1 GFLOPS, expected October 1993**
- **Software**
  - **EM−C: full C + libraries for rloads, RPC, ...**
  - **ABCL obj. oriented language (U Tokyo)**
- **EM−5: design only, will not be built**
  - **16K PEs, 64−bit, 655 GIPS, 1.3 TFLOPS, 64−bit**
  - **Separate frame ptr, instruction ptr on tokens**
  - **EMC−G single processor chip (successor to EMC−R)**
  - **Off−chip network switch**
- **''Real World Computing''**
  - **New Japanese national project (1992 − 2002)**
  - **EM−4, EM−X, EM−5 will be major influences**
  - **RWC−1: 1K PEs, 500 GFLOPS, planned for 1995**
  - **RWC−2: 16K PEs, 10 TFLOPS, planned for 1997**

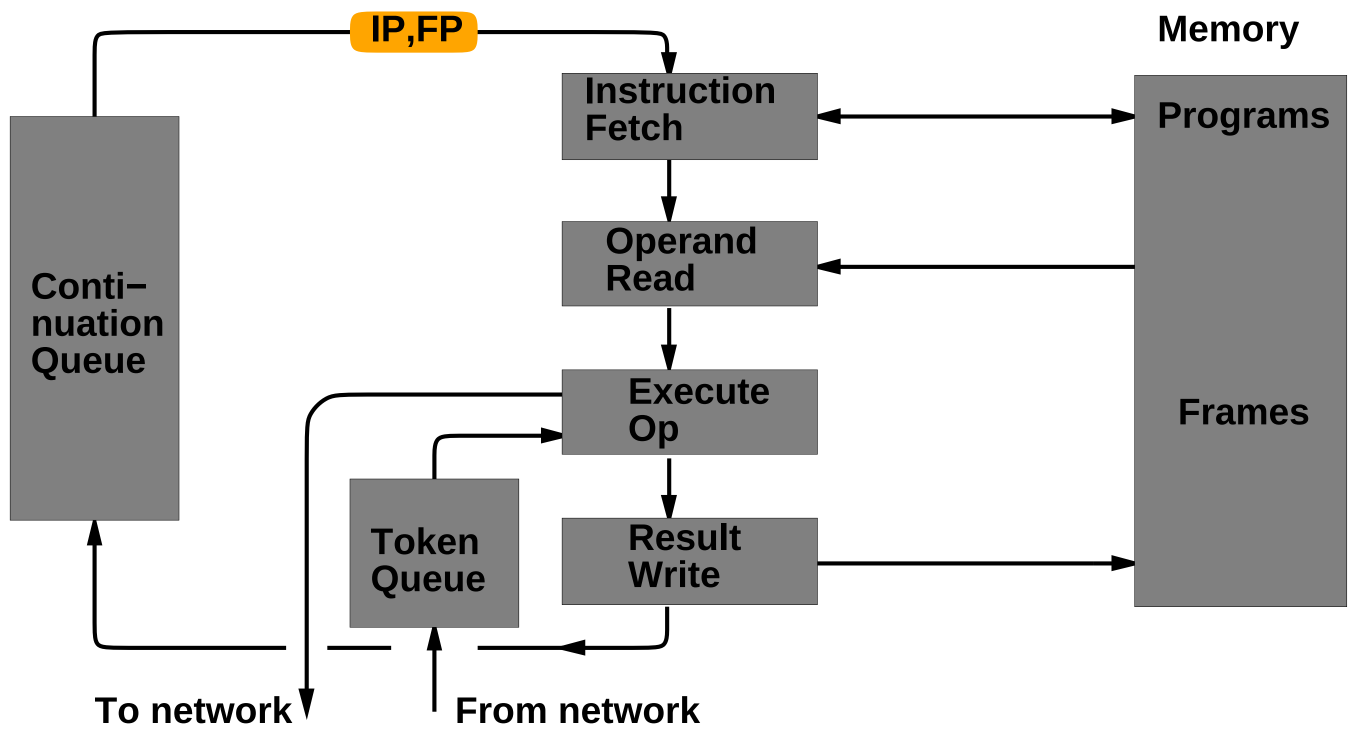
MIT P−RISC: "RISCifying" Dataflow

- **Nikhil, Arvind, 1988**
- **Split "complex" dataflow instruction into separate "simple" component instructions that can composed by the compiler:**
  - **Join**
  - **Operation**
  - **Fork**



  - **The compiler may choose to compose them differently (e.g., join, op, op, fork, op, op, op)**
- **Use traditional instruction sequencing (PC+1, Branch)**
  - **No more "destination" fields in instructions, except for control flow instructions−− branches, forks)**
- **Do all intra−processor communication via memory**
  - **No more data values on intra−processor tokens**
- **Do "joins" explicitly using memory locations**
  - **No more presence bits on frame locations**
  - **Generalizes to N−way joins (not just binary)**

MIT P−RISC: Basic Operation



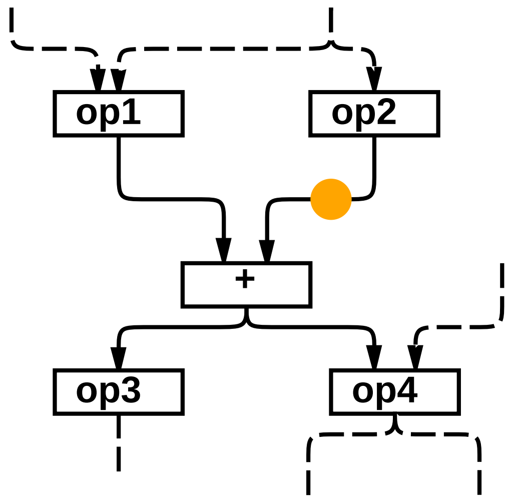
Instruction at IP	Frame Action	Intra−Processor token(s) emitted	Inter−node tokens
d <− s1+s2	FP[d]:=FP[s1]+FP[s2]	IP+1,FP	
jump L		L,FP	
fork L		IP+1, FP L,FP	

join N d	FP[d]++ *(atomically!)*	FP[d]=N: IP+1,FP
		else: —	

d <− rload s; L	read FP[s] FP[d]:=v	IP+1,FP L,FP	rload,FP[s]; L,FP,d L,FP; d,v

MIT P−RISC: Basic Operation Encoding

Conceptual



Encoding of continuations (intra−processor tokens):

A "packet" containing:

120	Destination instruction address
FP	Frame Pointer

Encoding of graph

Program Memory

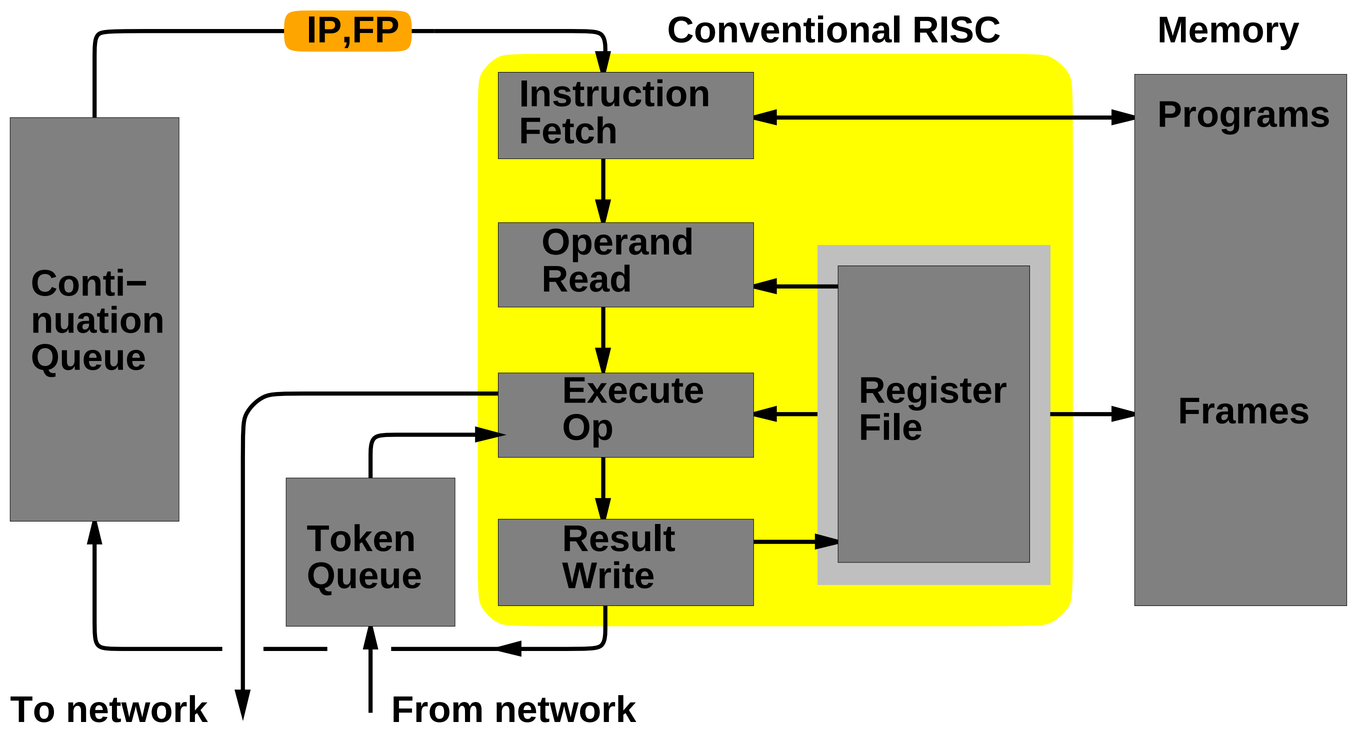
	Op−code	Operands
109	join 2	...
110	op1	d1, ...
111	jump	120
113	op2	d2, ...
	jump	120
120	join 2	dj
121	+	d,d1,d2
122	fork	159
123	jump	141
141	op3	... , d, ...
159	join 2	...
160	op4	... , d, ...

Tighter encodings of the graph now possible, producing longer "threads"

109	join 3	dj
110	jump	120
113	join 3	dj
114	jump	120
120	op1	d1, ...
121	op2	d2, ...
122	+	d, d1,d2
123	op3	... , d, ...
124	fork	159
125	jump	...
159	join 2	...
160	op4	... , d, ...

MIT P−RISC Alternate Implemenations

- To incorporate access to high−speed registers, could make it like Monsoon:
  - "Direct recirculate" continuations
  - D copies of registers (D = latency around pipeline)
  - etc.
- Or, could make it like EM−4:
  - Each continuation triggers a sequential thread that is executed to completion, before accepting the next continuation:

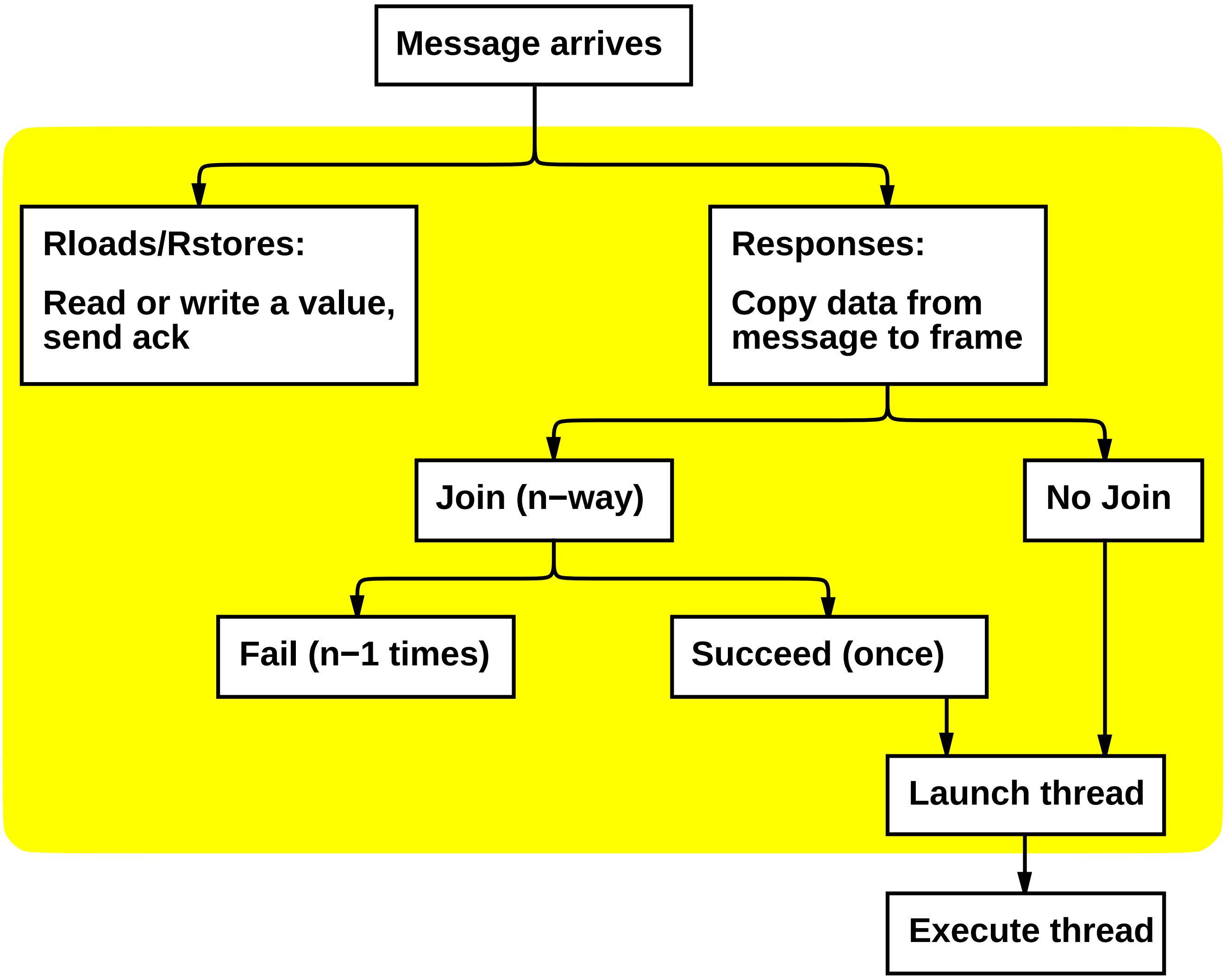


  - A thread ends at a failing JOIN instruction or an explicit HALT instruction

MIT *T: Message Triage

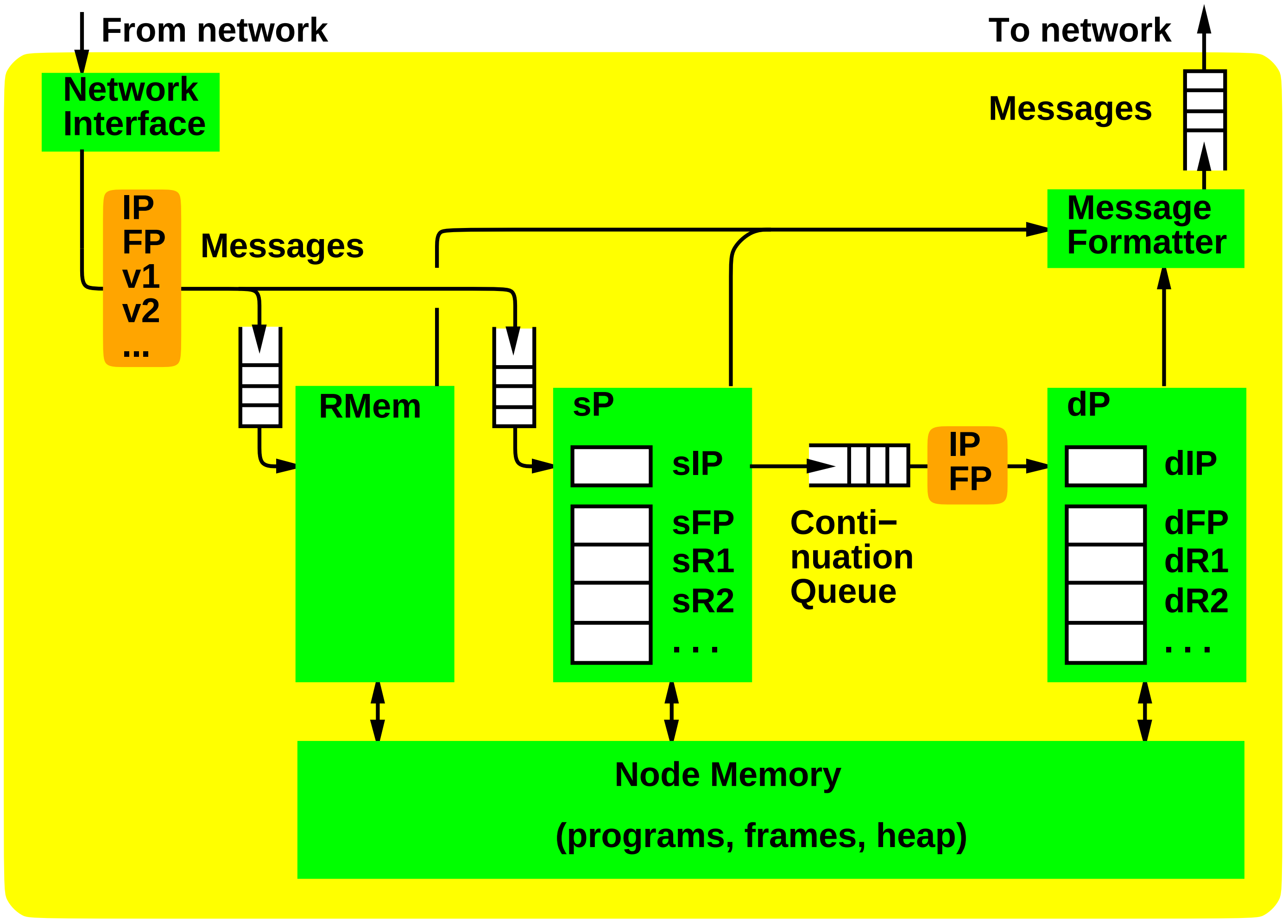
(*T is pronounced "start")

- **Nikhil, Papadopoulos, Arvind [1991]**
- **Many aspects of message−handling are short, simple activities, only a few of which result in execution of a non−trivial thread:**



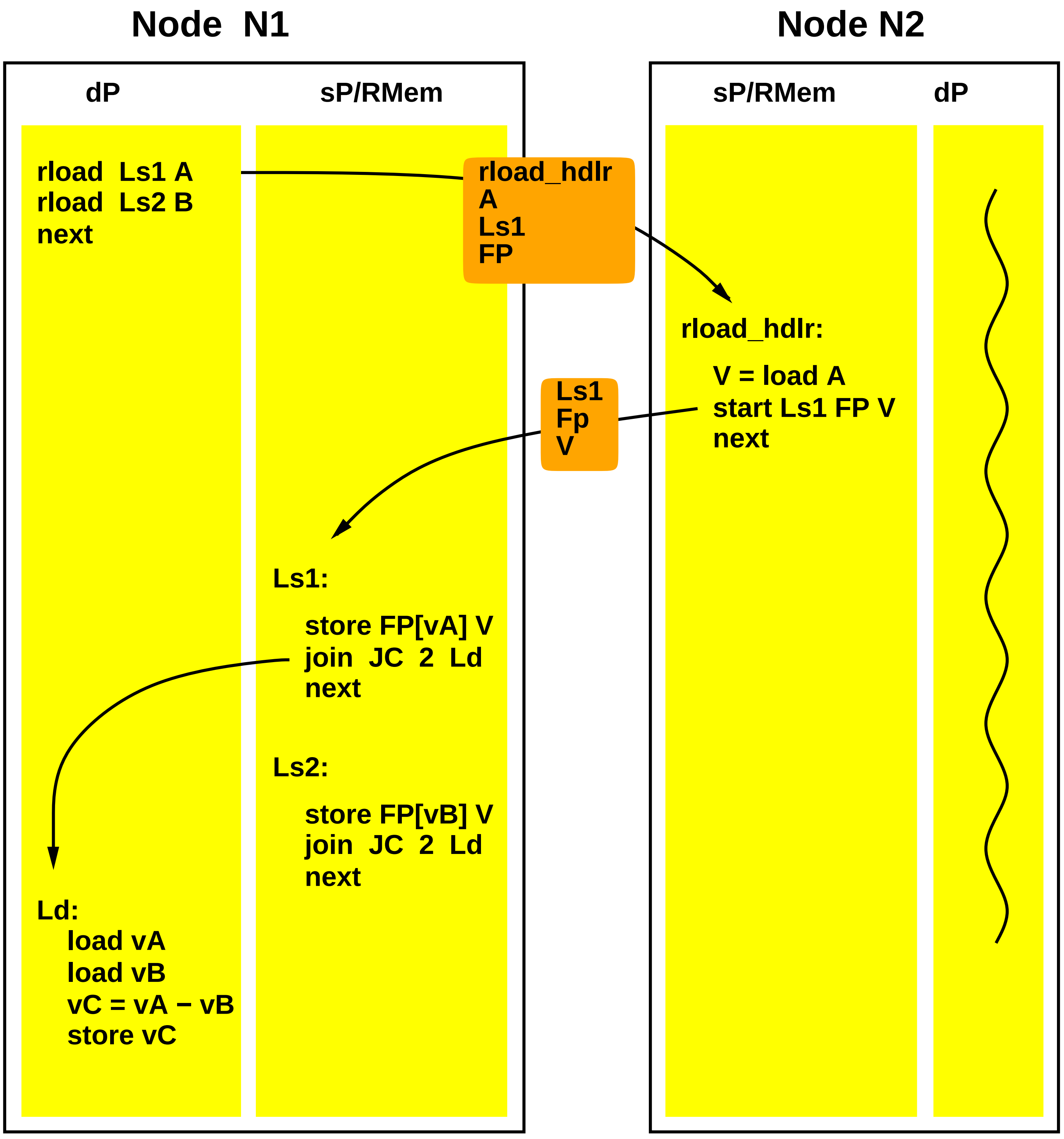
- ***T node: offload these simple (but disruptive) tasks from main processor (which executes long threads) into separate coprocessors**

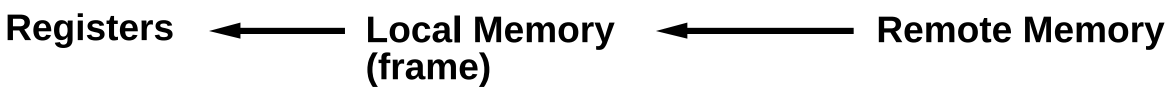
MIT *T Node Architecture (Concept)



- ***RMem: Remote Memory Request Processor***
  - Handles rload/rstore requests
- ***sP: Synchronization CoProcessor***
  - Optimized for simple, short threads
  - Fast load: incoming message –> sIP, sFP, sR1, ...
  - Handles rload/rstore responses, joins, ...
- ***dP: Data Processor***
  - Conventional RISC (optimized for long threads)
  - Fast load: incoming continuation –> dIP, dFP

*T: "Remote Loads" Code Structure



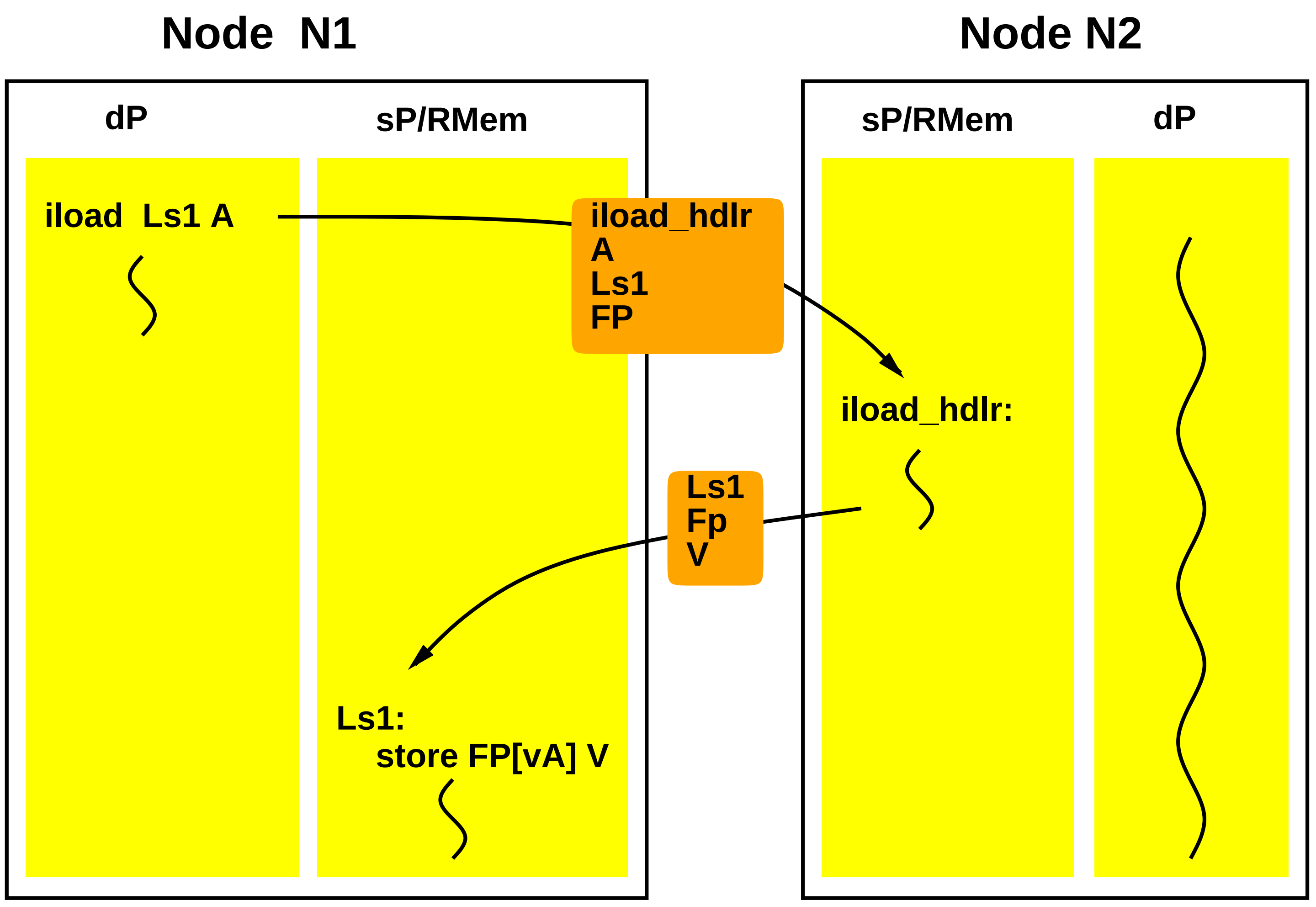


Data Movement

MIT *T: "Synchronizing Loads"

Requesting node (N1): Use "iload" instruction

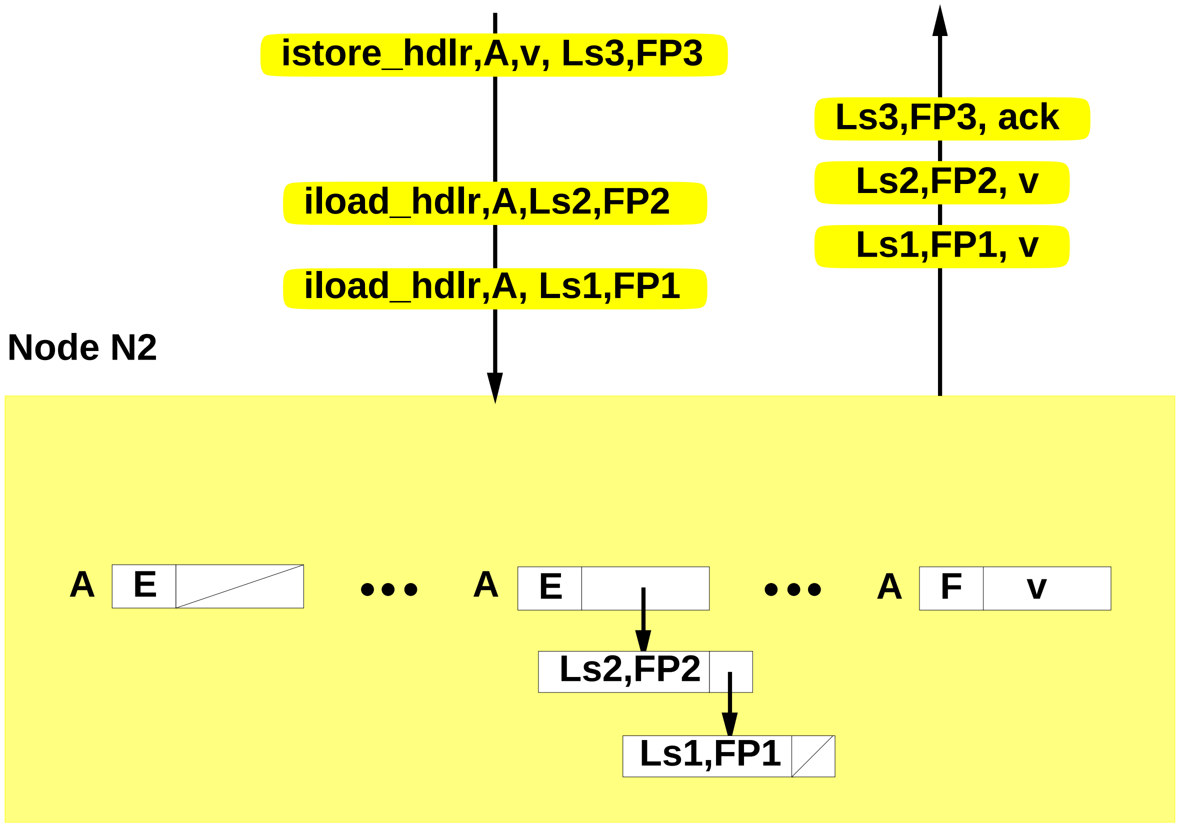
Interface is exactly like "rload":



MIT *T: "Synchronizing Loads"

Replying node (N2) implements "I−Structure" semantics:

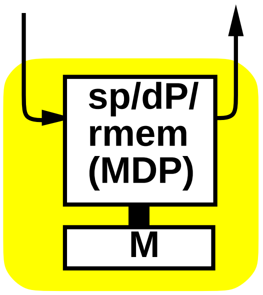
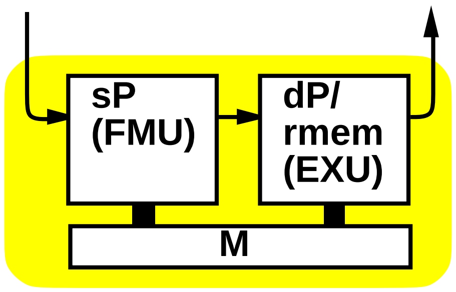
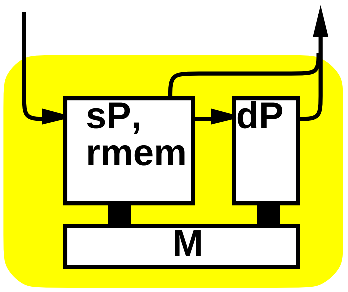
- **FULL/EMPTY bits on producer−consumer variables**
- **"iload" requests to EMPTY locations are queued**
- **"istore" stores value, releases queued requests**



- **No busy−waiting**
- **No extra messages**

(because messages carry virtual continuations, not physical continuations)

J−Machine, EM−4 and *T
Some Comparisons

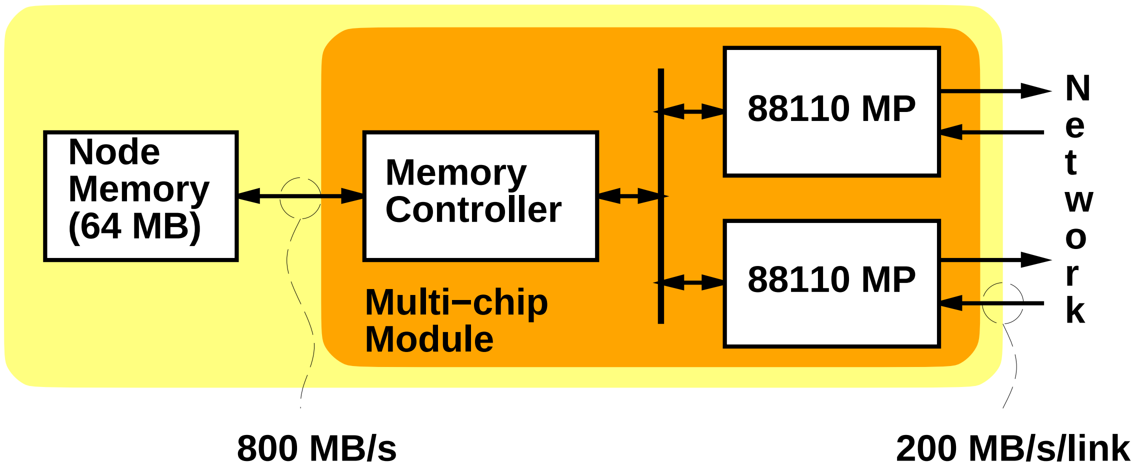
	 J−Machine	 EM−4	 *T
Joins:	N−ary, in software	Binary, in hardware in sP N−ary can be done in software in dP	N−ary, in software in sP (hardware support possible)
Remote Load request handling latency	Depends on threads ahead of request (unless handled at higher priority)	Depends on threads ahead of request	Independent of threads ahead of request (unless continuation queue is full)
Remote Load request args (return conti−nuation)	Travel through memory	Stay in registers	Stay in registers

MIT/Motorola *T Prototype

Papadopoulos (MIT), Greiner (Motorola), ... (1991 −> ...)

- **Expected to be operational 1994**

- **Node architecture**



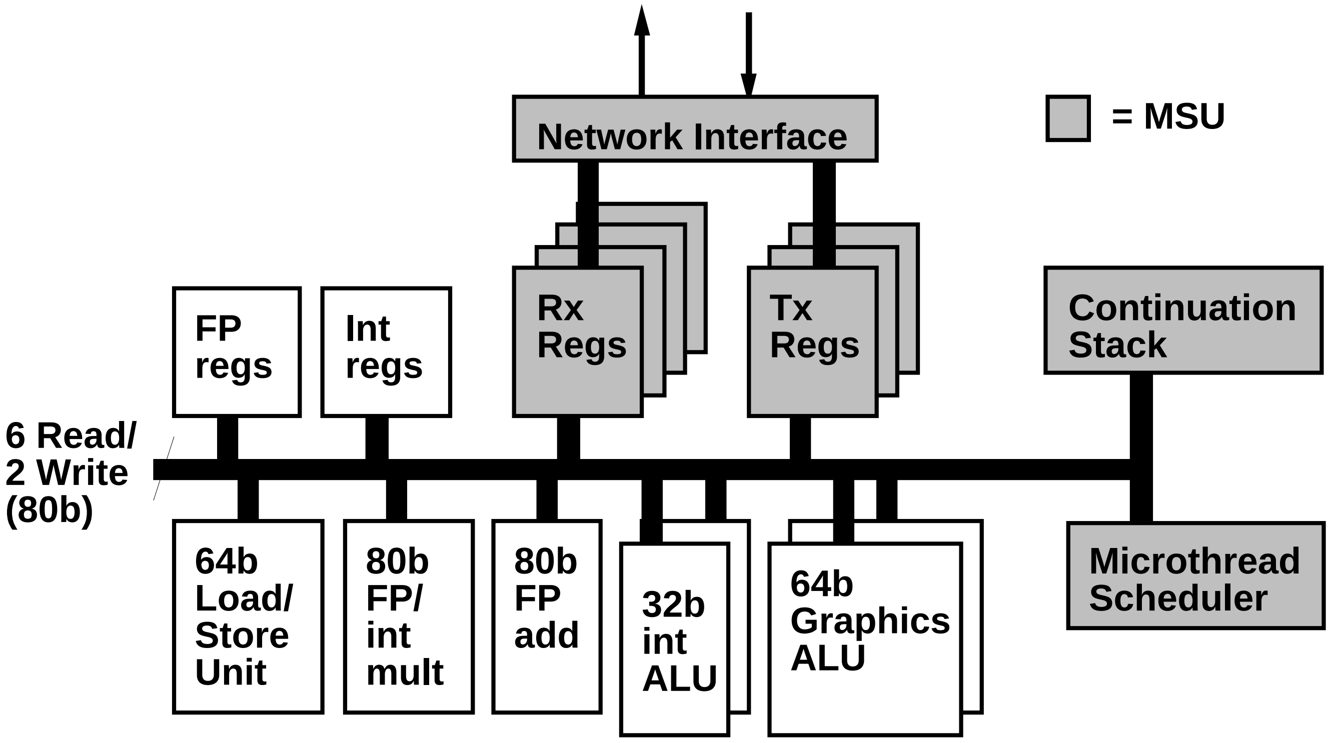
- **88110 MP =**
  - **Standard Motorola 88110 32−bit RISC 50 MHz, 2x superscalar (100 MIPS)**
  - **MSU (Message and Synchronization Unit) Implemented as an 88K on−chip SFU***

*** 88K family allows for additional on−chip SFUs, with**

- **Reserved opcode space**
- **Common instruction−issue logic, caches etc.**
- **Direct access to processor registers**
- **Other example SFUs: Floating point, graphics ...**

MIT/Motorola *T Prototype 88110 MP Communications

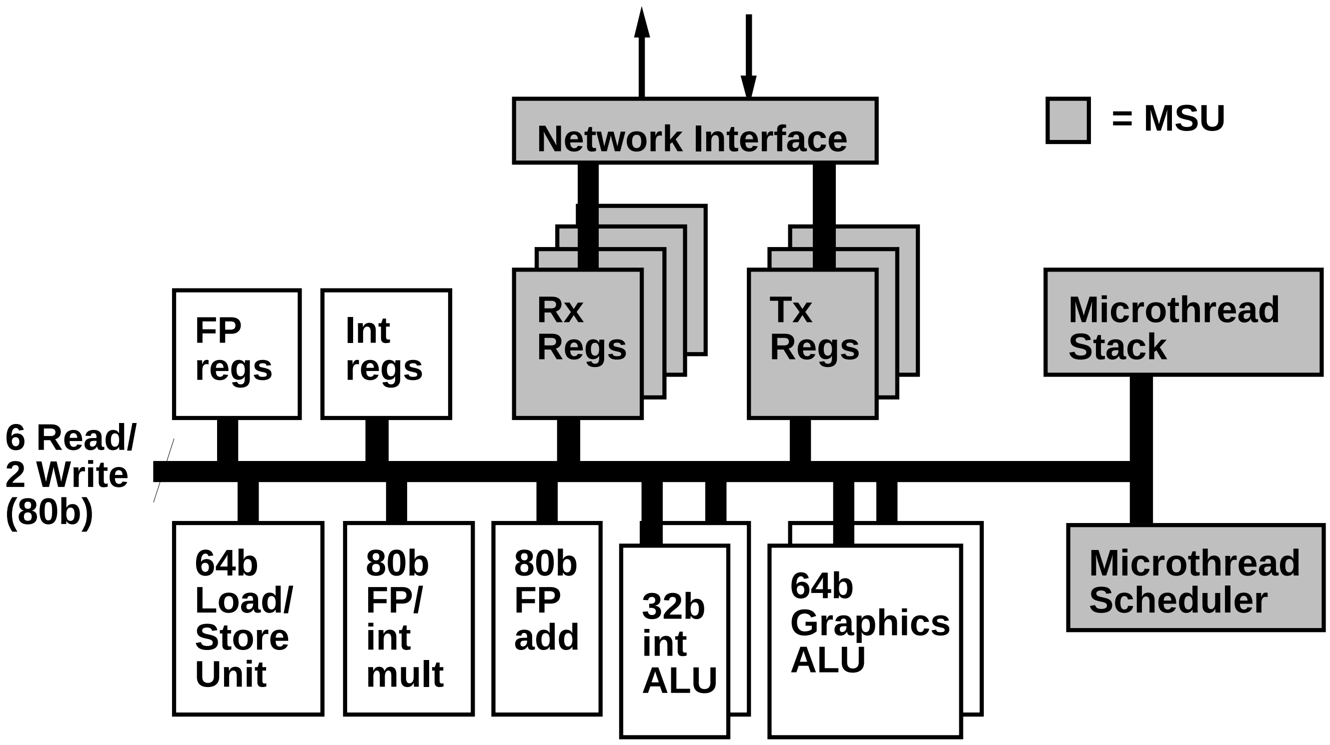
(Greiner, Papadopoulos, ..., 1991/1992)



- **Current incoming message: 24 Rx regs (r/w)**
  16 message buffers
- **Current outgoing message: 24 Tx regs (r/w)**
  4 message buffers
- **New instructions:**
  - **Move data between Tx, Rx regs and normal regs (4 x 64b/tick bandwidth)**
  - **Launch Tx message**
  - **Poll for, get next Rx message**

MIT/Motorola *T Prototype 88110 MP Microthread Scheduling

(Greiner, Papadopoulos, ..., 1991/1992)



- **Microthread descriptor: Instr. and Data ptrs (64 b total)**
- **New instructions:**
  - **Push descriptor on microthread stack (64 deep)**
  - **Get descriptor from microthread scheduler. Chooses from, e.g.,**
    - **Fixed high priority microthread**
    - **Microthread in head of next message**
    - **Microthread stack**
    - **Tracing microthread**
    - **Background microthread**

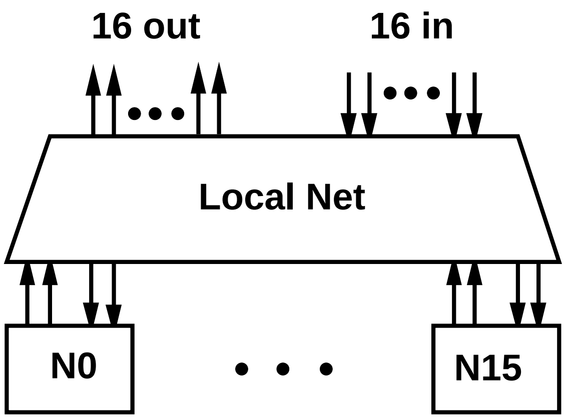
    etc.
  - **Pack, unpack microthread descrs. into messages**

MIT/Motorola *T Prototype System View (Tentative)

Global addressing (software sees single address space)

A "brick" (16 nodes):

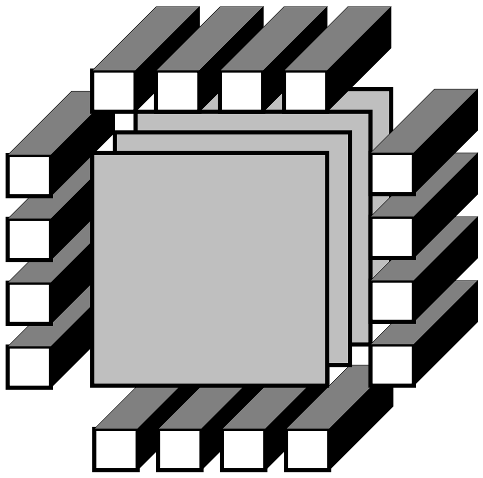
- **3,200 MIPS**
- **3.2 GFLOPS**
- **1 GB dram**
- **6.4 GB/s I+O**
- **about 9" x 9" x 9"**



256−node machine (16 bricks):

- **50,000 MIPS**
- **50 GFLOPS**
- **16 GB dram**
- **25 GB/s Bisection BW**
- **1.5 meter cube**

(limited by connector−pin density)



Each of the four layers implements a 16x16 switch on a board (no cables!)

No communication between layers

MIT/Motorola *T Prototype Network

- Messages:

Logical Node # (4 bytes)
Length (4 bytes)
Data (8 to 88 Bytes)

- Bandwidth at 88110 pins:
  - 4 Bytes/tick (200 MB/s), in and out
- 3 ticks (60 nsec) latency, minimum, for:
  - Transmit:
    Transmit instrucion to appearance of first flit
  - Receive:
    Arr. of last flit to exec of 1st handler instruction
- Switching chip: PaRC II (Packet Routing Chip)
  - 8 x 8 crossbar
  - 200 MB/sec/link
  - Pipelined link buffers, cut−through
  - 2nd generation (Monsoon's PaRC was 1st)
- Topology: Fat Tree

Multithreading: a Dataflow Story Final Comments

- **The system−level view has stayed constant, from the TTDA through *T:**
  - **Information on inter−node tokens:**
    Full virtual continuations (tags) and values
  - **Semantics of inter−node communication:**
    Split−phase transactions
    Non−blocking sends
  - **Very tight, smooth integration of message−handling with computation pipeline**
  - **Intra−node multithreading to tolerate latency**

- **The various designs differ in internal node architecture, with trends towards**
  - **Removal of intra−node synchronization**
  - **Longer threads**
  - **Use of high−speed registers**
  - **Compatibility with conventional machine codes**

- ***T "unifies" dataflow, von Neumann ideas, supporting:**
  - **Scalable shared memory programming model**
  - **Existing SIMD/SPMD codes**