

# **Data Flow on a Queue Machine**

*Bruno R. Preiss*

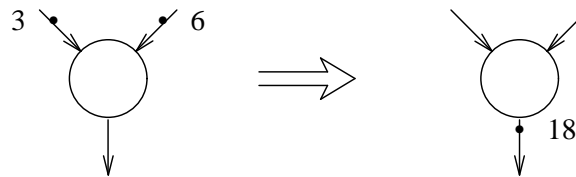
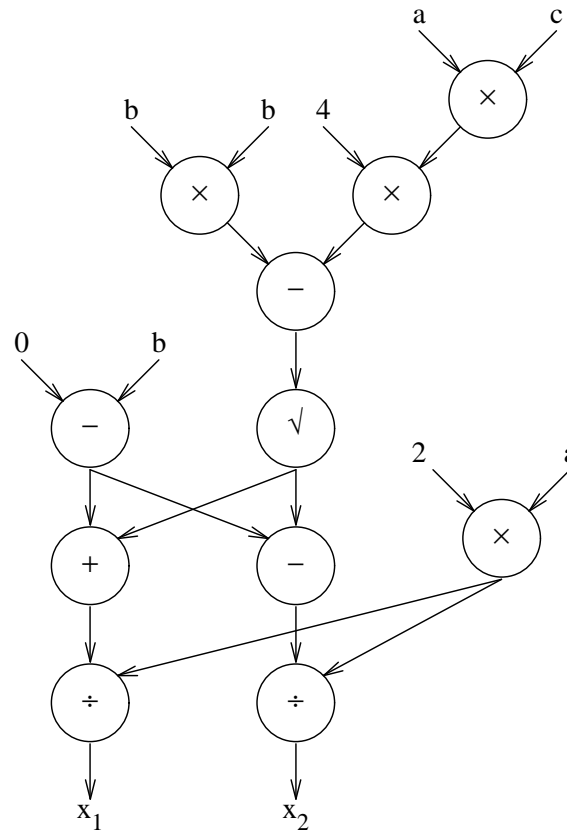
## Outline

- Genesis of data-flow architectures
- Static vs. dynamic data-flow architectures
- Pseudo-static data-flow execution model
- Some data-flow machines
- Simple queue machine
- Prioritized queue machine
- Program decomposition
- Queue machine processing element

## Genesis of Data Flow

- Data-flow principles: asynchrony and functionality

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}; \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$



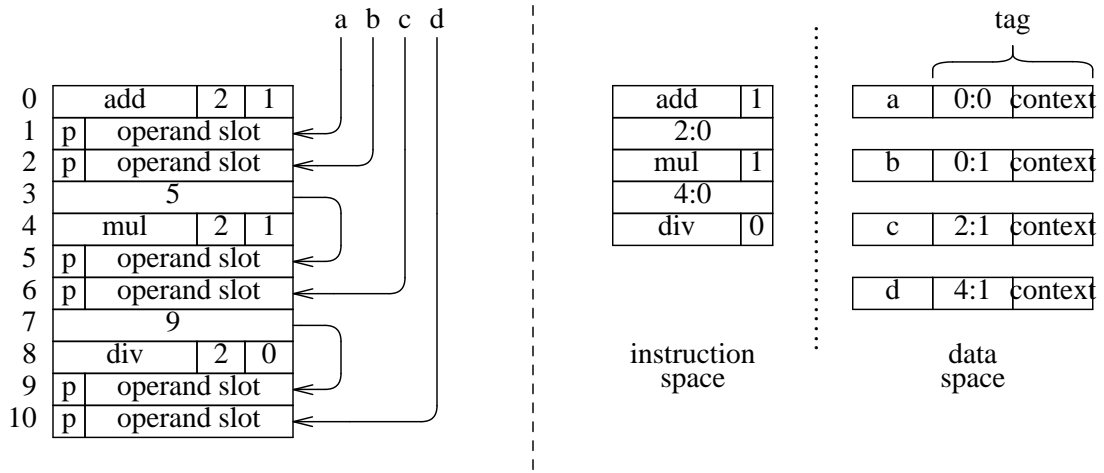
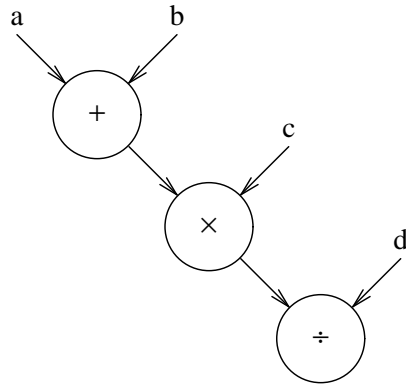
## **Data-Flow Execution Models**

### **Static:**

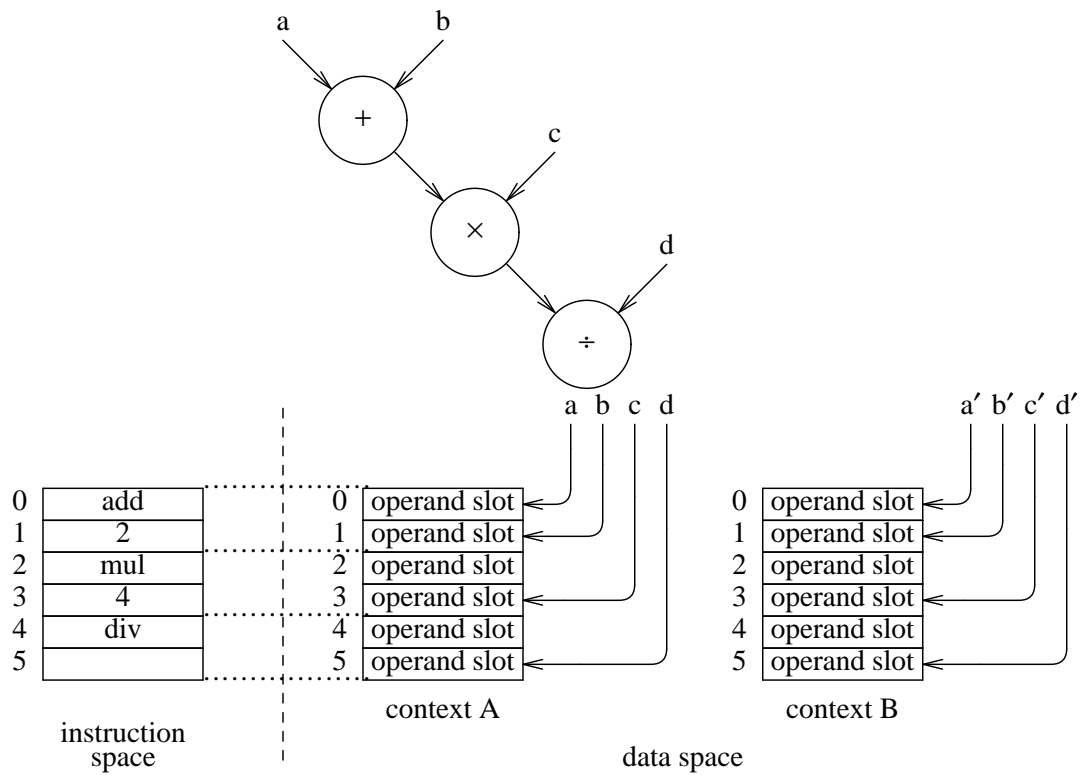
- Program loaded into memory in completed form before execution begins
- At most one instance of an actor can be enabled
- Same storage space used for instructions and data

### **Dynamic:**

- Program nodes can be instantiated at run time
- Several instances of an actor may be enabled
- Separate storage space used for instructions and data

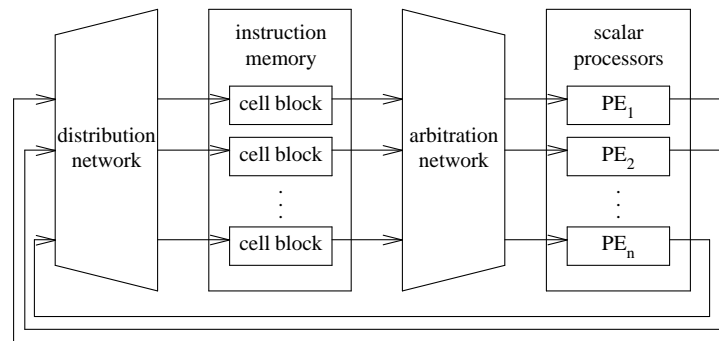


## Pseudo-static Data Flow



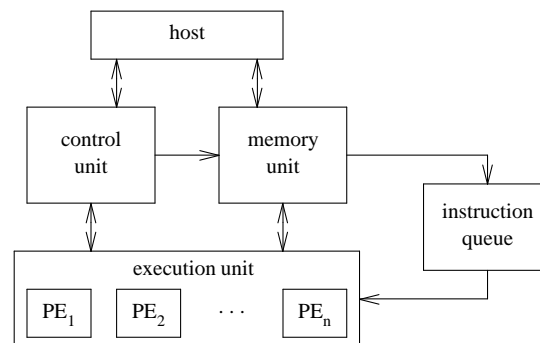
- Can associate several data spaces with one instruction space
- Reentrancy accomplished without code copying or tagged tokens

## MIT Static Data-Flow Machine



- Static data-flow machine
- Packet communication based
- Networks are pipelined — provide queueing
- 4 processor prototype built

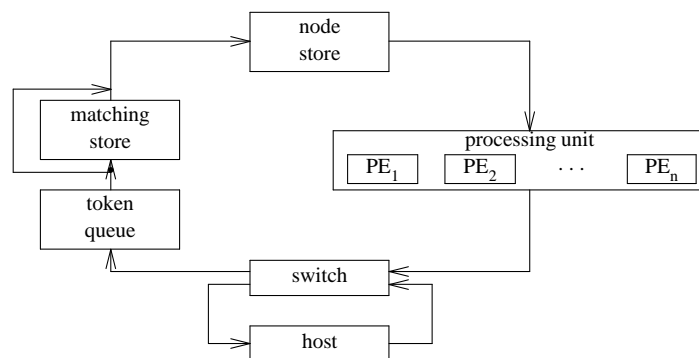
## LAU System Architecture



- Static data-flow machine
- Uses acyclic data-flow graphs

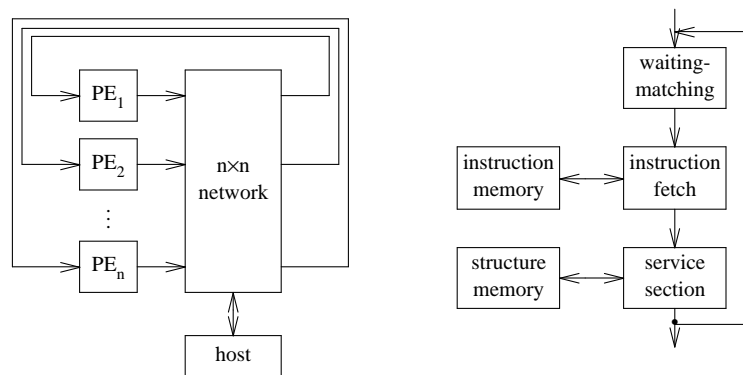
- Explicit external instruction queue
- 32 processor prototype built

## Manchester Data-Flow Architecture



- Dynamic data-flow architecture
- Tagged tokens
- Circular pipeline with token queue
- 15 processor prototype built

## MIT Dynamic Data-Flow Architecture



- Dynamic data-flow architecture

- Tagged tokens
- Circular pipeline with queueing in waiting/matching section

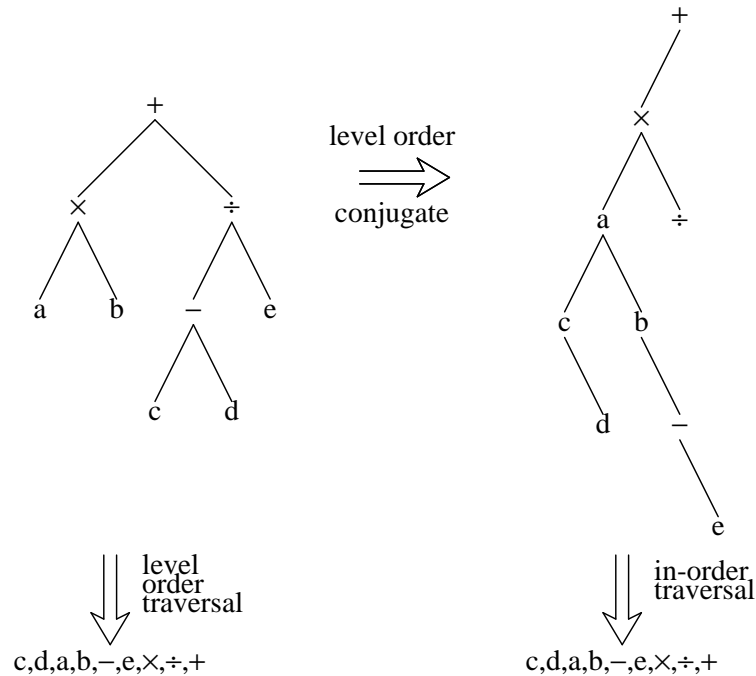
## Simple Queue Machine

- Uses a first-in, first-out (FIFO) queue for the manipulation of operands and results
- Its instructions implicitly reference an operand queue
- Analogous to a stack machine

$f \leftarrow ab + \frac{(c-d)}{e}$			
Stack		Queue	
instruction sequence	stack contents	instruction sequence	queue contents
fetch $a$	$a$	fetch $c$	$c$
fetch $b$	$b, a$	fetch $d$	$c, d$
mul	$ab$	fetch $a$	$c, d, a$
fetch $c$	$c, ab$	fetch $b$	$c, d, a, b$
fetch $d$	$d, c, ab$	sub	$a, b, c-d$
sub	$c-d, ab$	fetch $e$	$a, b, c-d, e$
fetch $e$	$e, c-d, ab$	mul	$c-d, e, ab$
div	$\frac{(c-d)}{e}, ab$	div	$ab, \frac{(c-d)}{e}$
add	$ab + \frac{(c-d)}{e}$	add	$ab + \frac{(c-d)}{e}$
store $f$		store $f$	

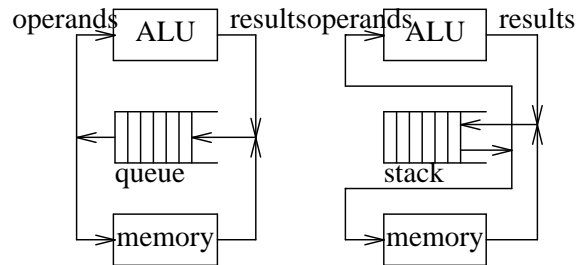
## Generating Instruction Sequences for a Simple Queue Machine

$$f \leftarrow ab + \frac{c-d}{d}$$



- LEMMA: Level-order traversal of expression parse tree gives queue machine instruction sequence
- Time complexity of level order conjugation is  $O(N)$
- Space complexity of level order conjugation is  $O(N)$

## Pipelined Execution: Stack vs. Queue



- Case 1: Non-overlapped operand fetch/execute
- Case 2: Overlapped operand fetch/execute

$$Speed-up^\dagger = \frac{StackMachineCycles}{QueueMachineCycles}$$

nodes in parse tree	number of trees	case 1	case 2
4	4	1.	1.
5	9	1.02	1.02
6	20	1.03	1.03
7	45	1.04	1.05
8	101	1.05	1.07
9	227	1.05	1.08
10	510	1.05	1.09
11	1146	1.06	1.10

†two-stage pipelined ALU

$$Speed-up^\dagger = \frac{StackMachineCycles}{QueueMachineCycles}$$

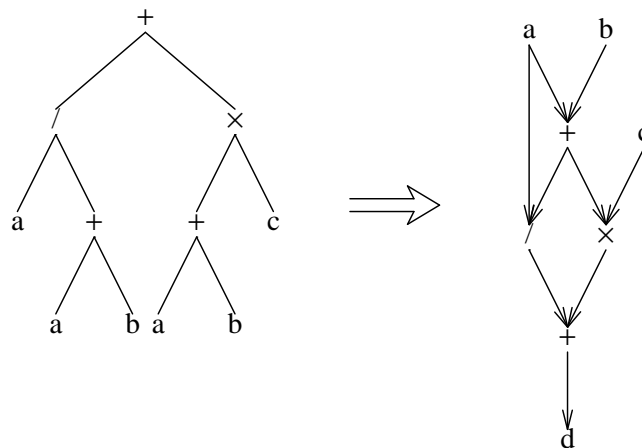
pipeline stages	assumption 1	assumption 2
1	1.	1.
2	1.06	1.10
3	1.08	1.09
4	1.09	1.08
5	1.10	1.07

†11 nodes in parse tree

## Prioritized Queue Machine

- Assign a “priority” to the result of each operation
- Place result in queue at position determined from priority

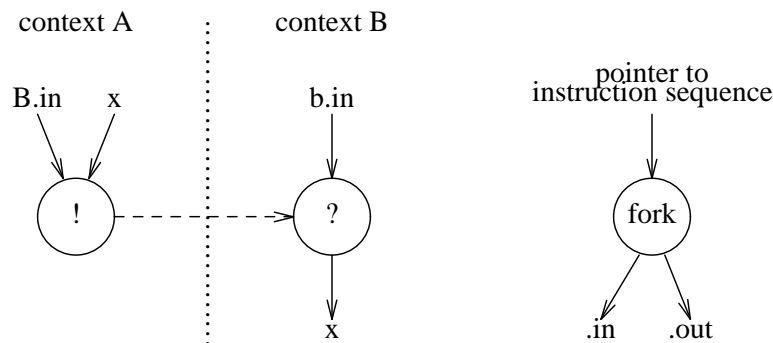
$d \leftarrow \frac{a}{a+b} + (a+b)c$		
Prioritized Queue Machine		
instruction sequence	result priorities	queue
fetch $a$	0,2	$a, \epsilon, a$
fetch $b$	1	$a, b, a$
plus	1,2	$a, a+b, a+b$
fetch $c$	3	$a, a+b, a+b, c$
div	2	$a+b, c, \frac{a}{a+b}$
mul	1	$\frac{a}{a+b}, (a+b)c$
add	0	$\frac{a}{a+b} + (a+b)c$
store $d$		



- LEMMA: Acyclic data-flow graphs “generate” valid prioritized queue machine instruction sequences

## Dynamic Data-Flow Graph Splicing

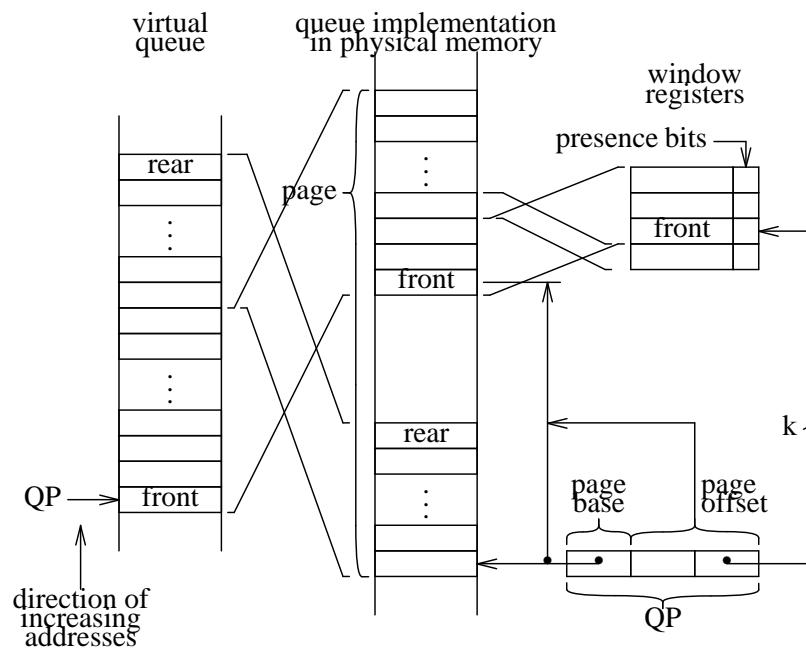
- Based on two concepts: *channels* and *contexts*
- Channel: Unidirectional communication path between two contexts
- Context: A process that evaluates an acyclic data-flow graph
- State of a context: An instruction sequence (and PC) and an operand queue
- Conditional execution, iteration, subroutine calls implemented by instructions for context creation (fork) and intercontext communication (send (!) and receive (?))



## Partitioning Programs

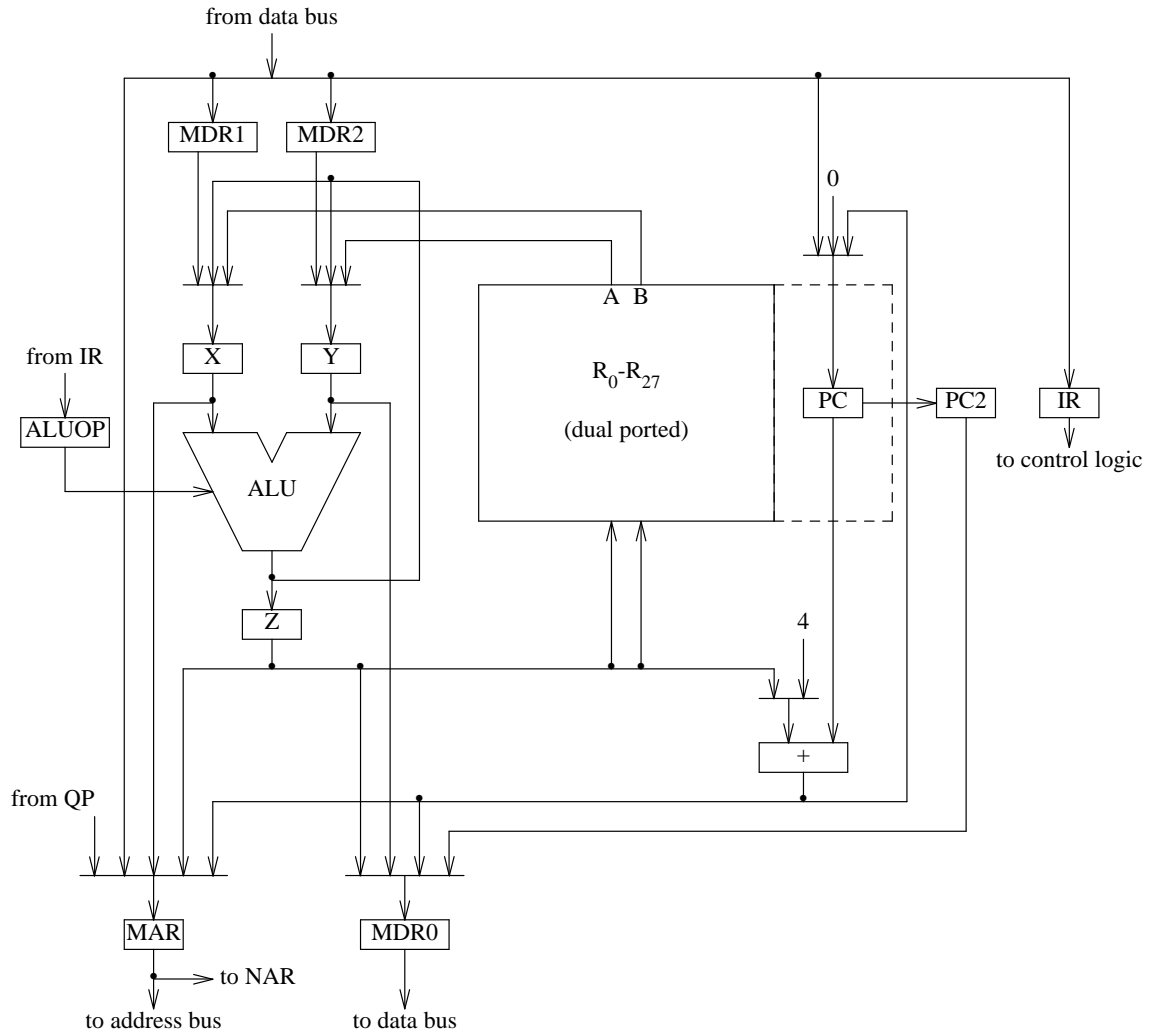
- Goal: exploit potential parallelism
- Use acyclic data-flow graph as the basic granule of computation
- Granule size trade-off:  
Small contexts → excessive intercontext communication and context generation overhead  
Large contexts → cannot exploit intracontext parallelism
- Context-based partition is a compromise between conventional data-flow and task- or process-based parallelism (Ada, Concurrent Euclid)
- Conventional data-flow architectures attempt to detect and exploit operator-level parallelism at execution time (costly and inefficient)
- Task- or process-based approach requires programmer to explicitly partition programs into granules of computation and to code communications between tasks
- Approach: Partition programs (automatically) into granules of computation more complex than single instructions, yet less complex than processor task, and to automatically exploit parallelism between contexts
- OCCAM compiler

## Queue Machine Processing Element



- QP: queue pointer register
- Allocate page of memory for queue of each context
- Use window registers as a queue cache
- Presence bits indicate validity of register contents

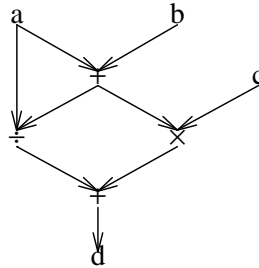
# Queue Machine Processing Element Architecture



## Queue Machine Assembly Language

- **Syntax:** *opcode*{+}[*src*<sub>1</sub>[,*src*<sub>2</sub>]][:*dst*<sub>1</sub>[,*dst*<sub>2</sub>]]

$$d \leftarrow \frac{a}{a+b} + (a+b)c$$



fetch #a: r0, r2

fetch #b: r1

plus++ r0,r1: r1, r2

fetch #c: r3

div++ r0,r1: r2

mul++ r0,r1: r1

plus++ r0,r1: r0

store+ #d,r0

