CS 152
Computer Architecture and Engineering
Lecture 21 -- Dataflow

2014-4-10

John Lazzaro

(not a prof - "John" is always OK)

TA: Eric Love

www-inst.eecs.berkeley.edu/~cs152/

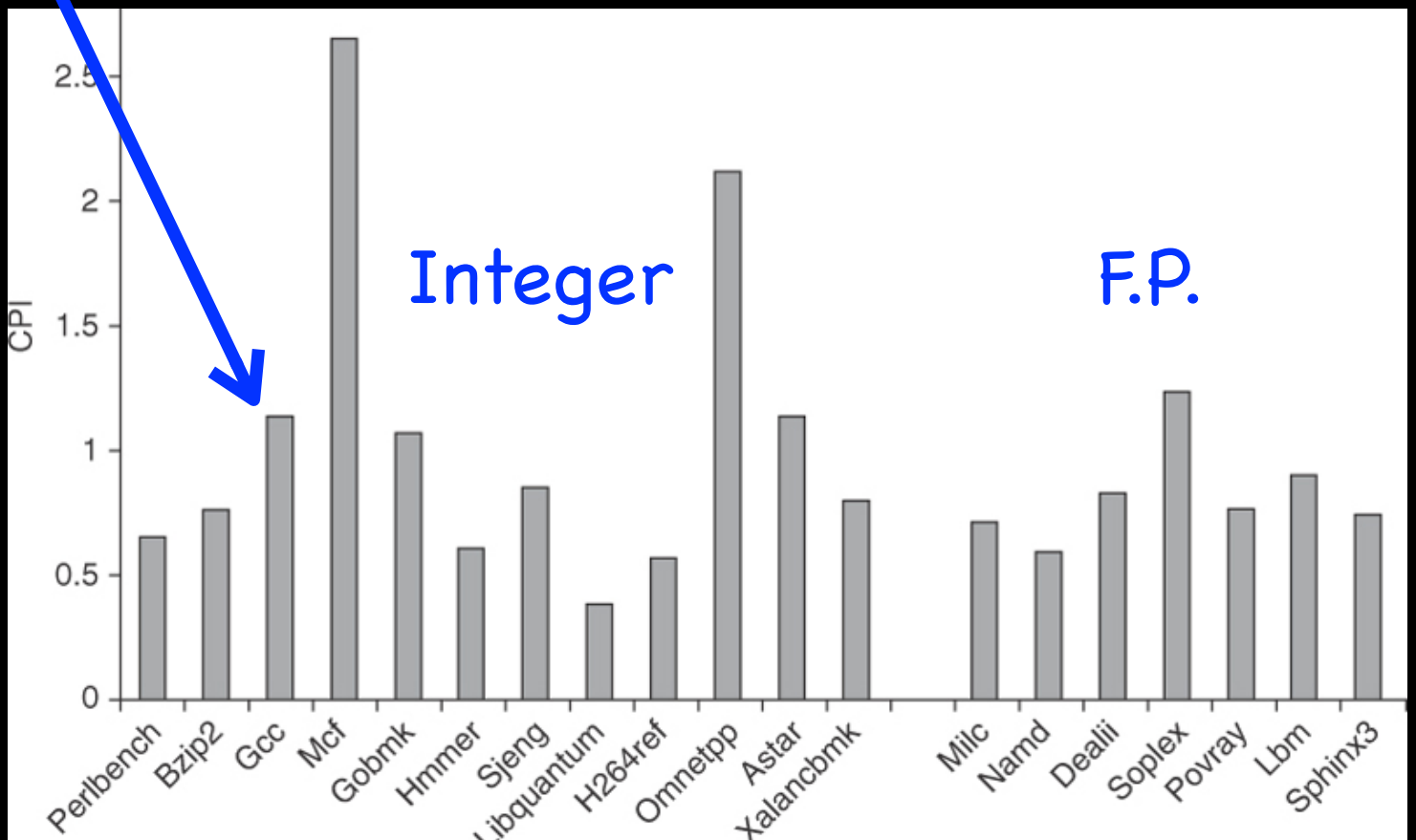
Play:

Actual CPIs on Intel Nehalem

Gcc: Perfect IPC was 55.

"Real-world" machines were 8-10.

Intel Nehalem was slightly under 1.



Maybe new ideas are needed ... for example, dataflow.

Von Neumann

Instructions execute in program order.

`i = i + 1;`
is ubiquitous in code.

Sequential model obscures parallelism.

Architected state.

Dataflow

An instruction may execute as soon as its operands are available.

Registers may only be written once.

Many instructions may execute concurrently.

Physical registers.

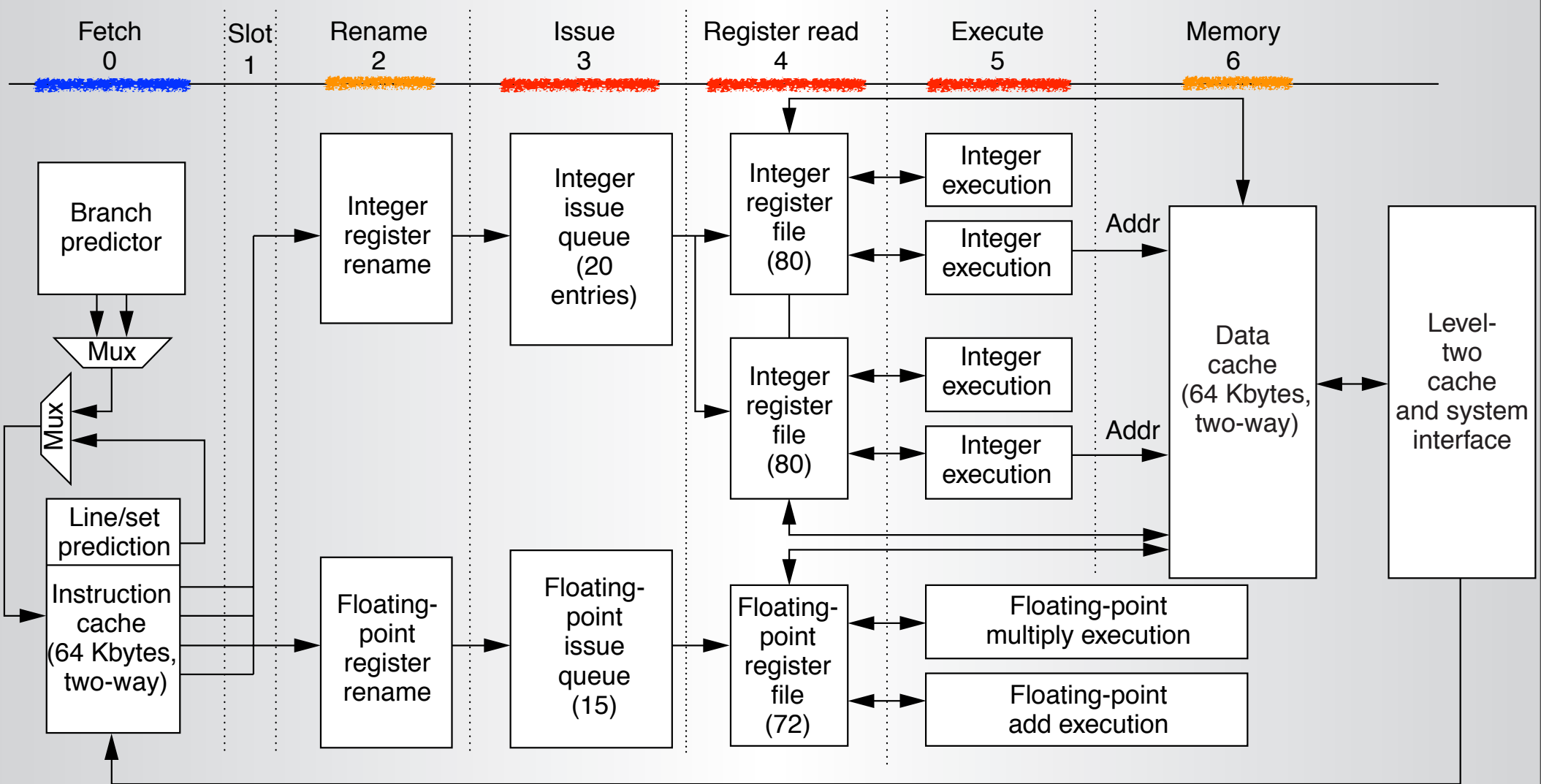
Both models of computation coexist in a dynamically-scheduled out-of-order processor.

21264 pipeline diagram

Von Neumann stages underlined in blue.

Dataflow stages underlined in red.

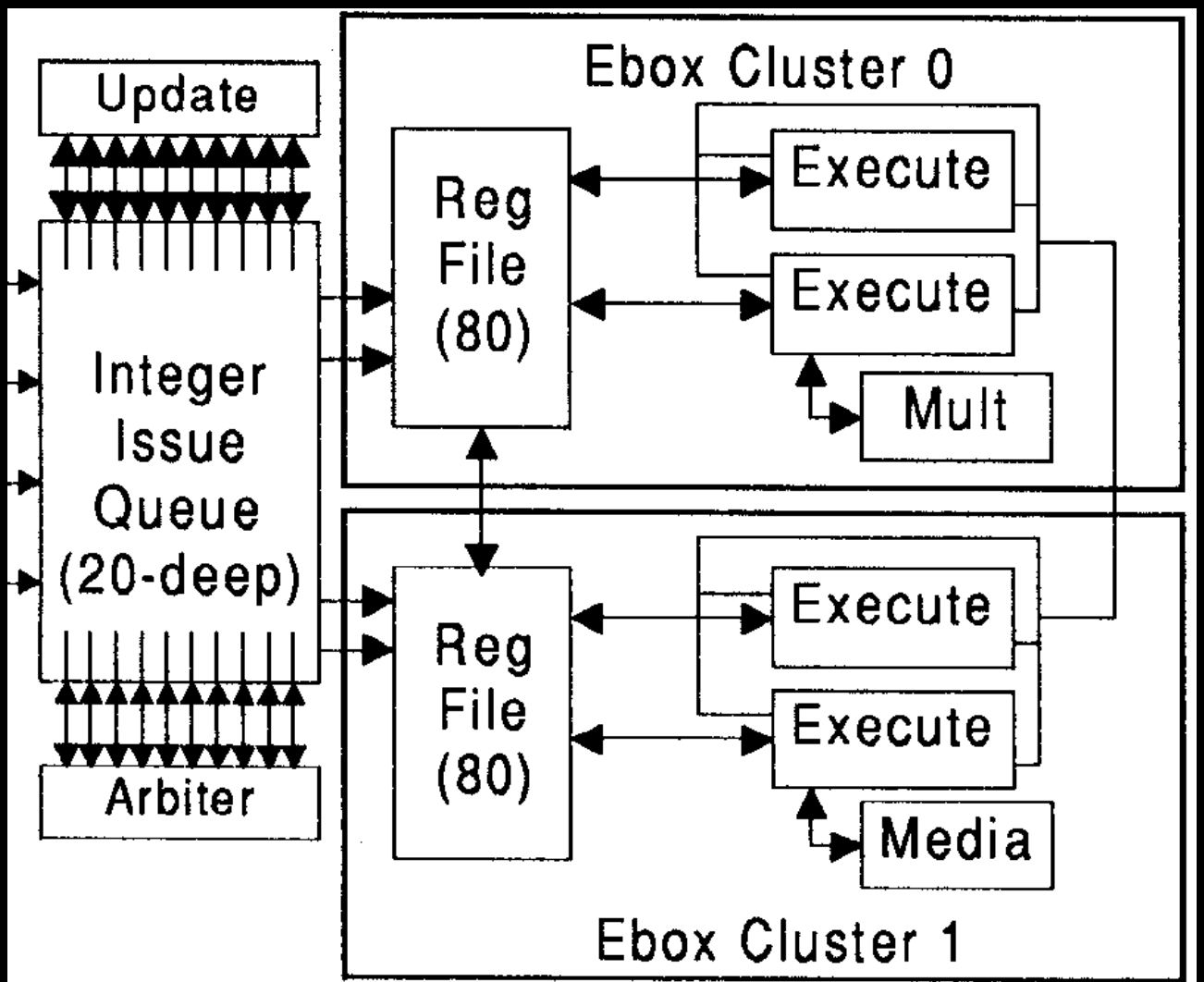
In orange stages, we translate between models.

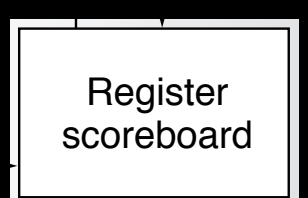


Dataflow stages of 21264

Idea: Write dataflow programs that reference physical registers, to execute on this machine.

Input: Instructions that reference physical registers.





Scoreboard: Tracks writes to physical registers.

Processor runs functions that follow dataflow rules.

Written once, on function invocation.

Functions may only access local variables.

Max # of variables + function arguments == number of physical registers.

```
int sum_squares(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a + b;
  b = y*y;

  return c;
}
```

Variables may only be written once.

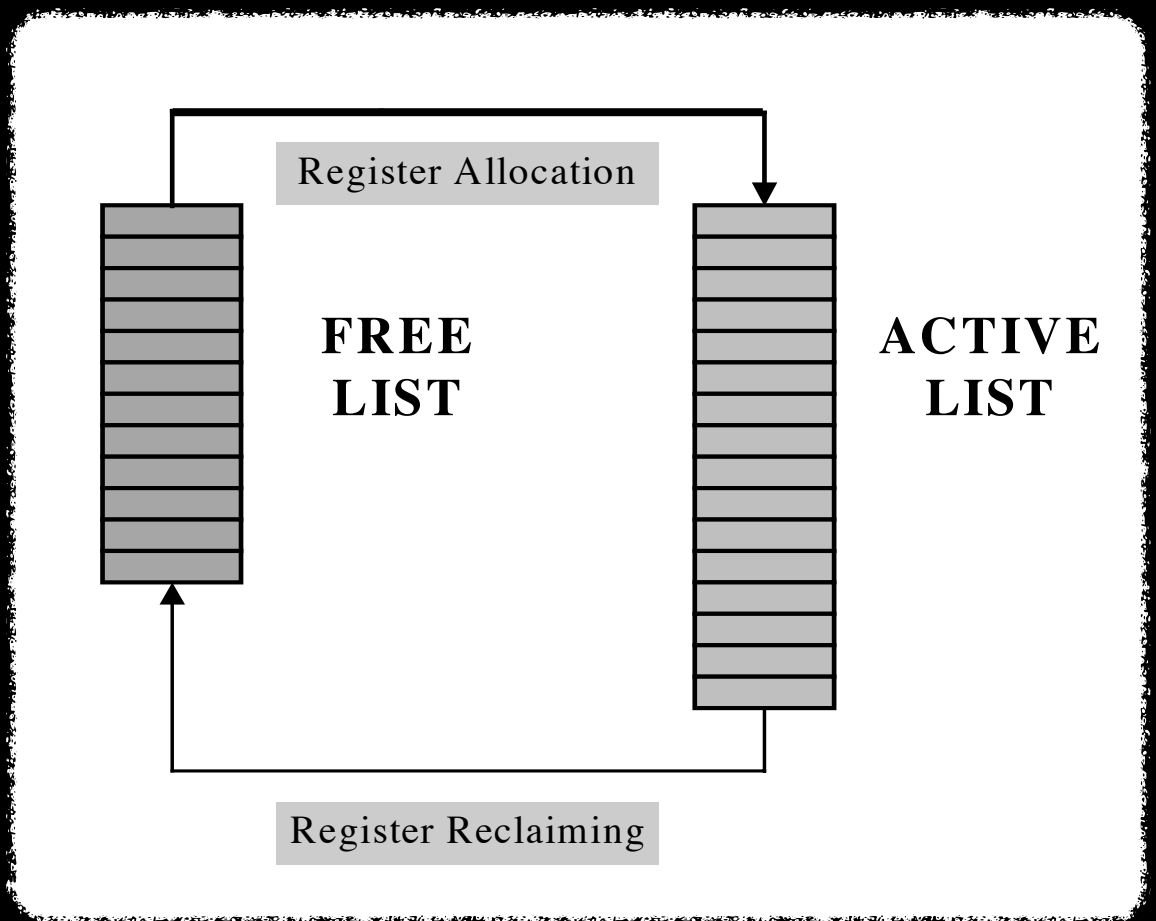
Not a bug! Statement order ignored.

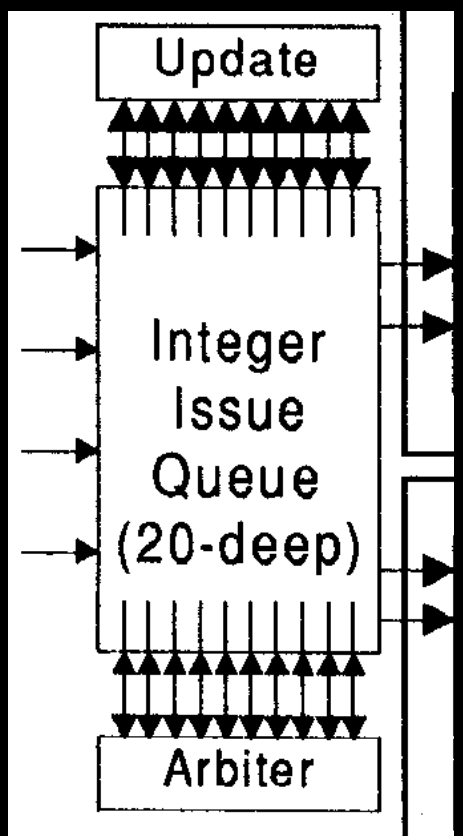
Outputs $x^2 + y^2$ to a network port, by writing to register R80.

Physical register manager.

When function is invoked, all registers are on the free list. As each instruction enters the queue, it's destination register moves to the active list. Thus, the "# of variables" limit.

Input: Instructions that reference physical registers.





Register scoreboard

On function exit, all active registers returned to free list.

Assembly code.

R0 = constant 0

```
int sum_squares(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a + b;
  b = y*y;

 return c;
}
```

Allocations: x = R1, y = R2, a = R3, b = R4, c = R80

Function invocation done by ADDIUs ...

```
ADDIU R1 R0 xval
ADDIU R2 R0 yval

MUL R3 R1 R1
ADD R80 R3 R5
MUL R5 R2 R2
```

Not a bug! Order ignored.

<empty line>

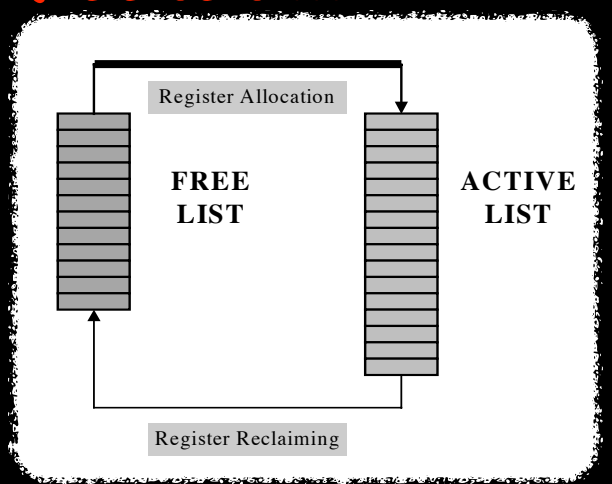
ISA convention: function exits when R80 written.

Dataflow stages of 21264

Since the function we wrote has fewer than 20 instructions, it will successfully execute.

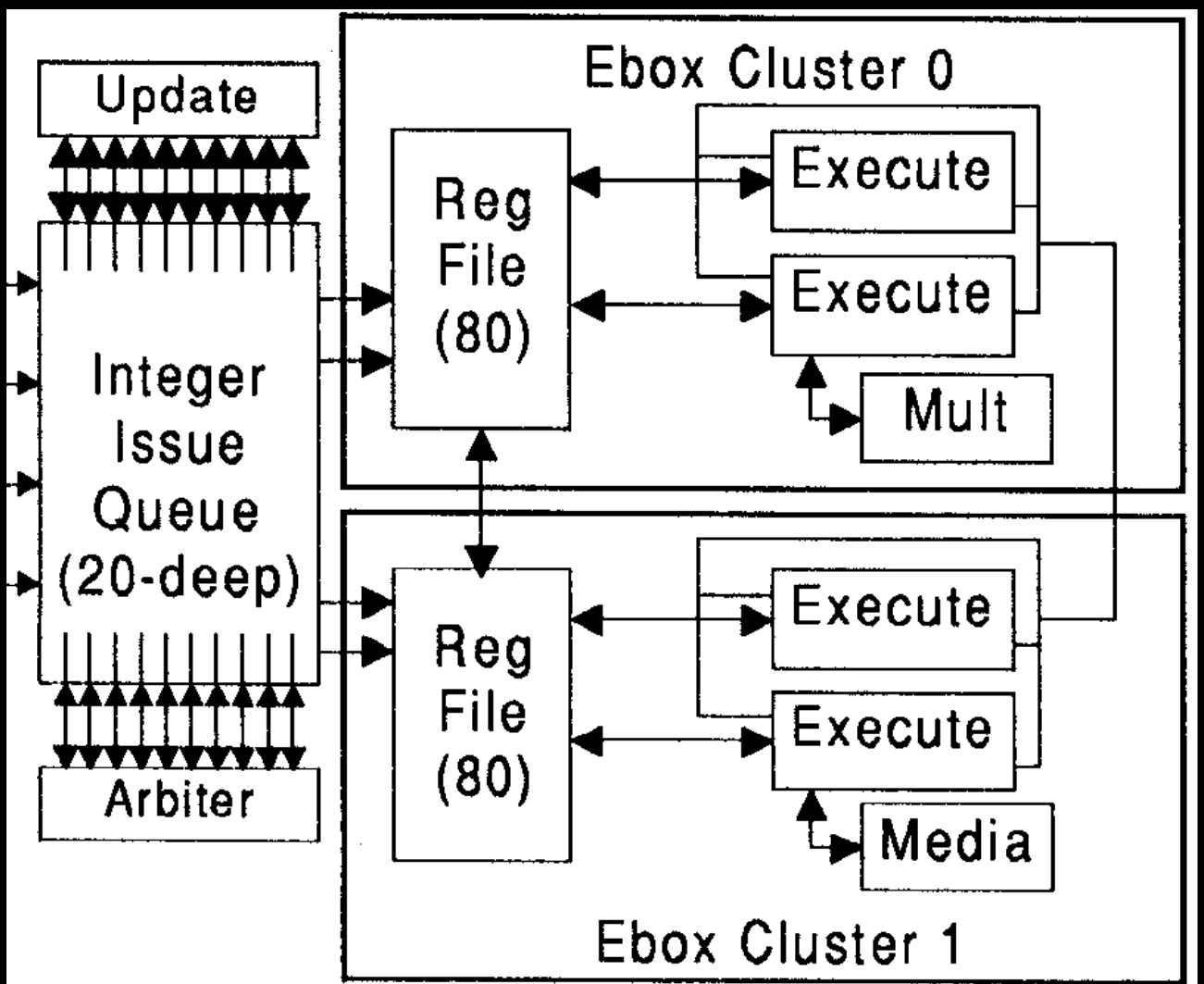
Why not 80?

Ordering rules are needed ...



If the first 20 instructions need instruction 21 to start executing?

Deadlock, given a 20-element queue!



Register scoreboard

Scoreboard: Tracks writes to physical registers.

Observations

- The basic idea seems to work.
  But it feels like a toy in so many ways ...
- Partial wish list ... while keeping dataflow nature.

Reusing registers during execution.

for(), while()
if/then, switch

Function calls within functions.

Bring back main memory!

Fix deadlock.

Efficient arrays.

These issues, and others, is what dataflow architecture in the 70-90s made progress on ... more after the break.

Break

Play:

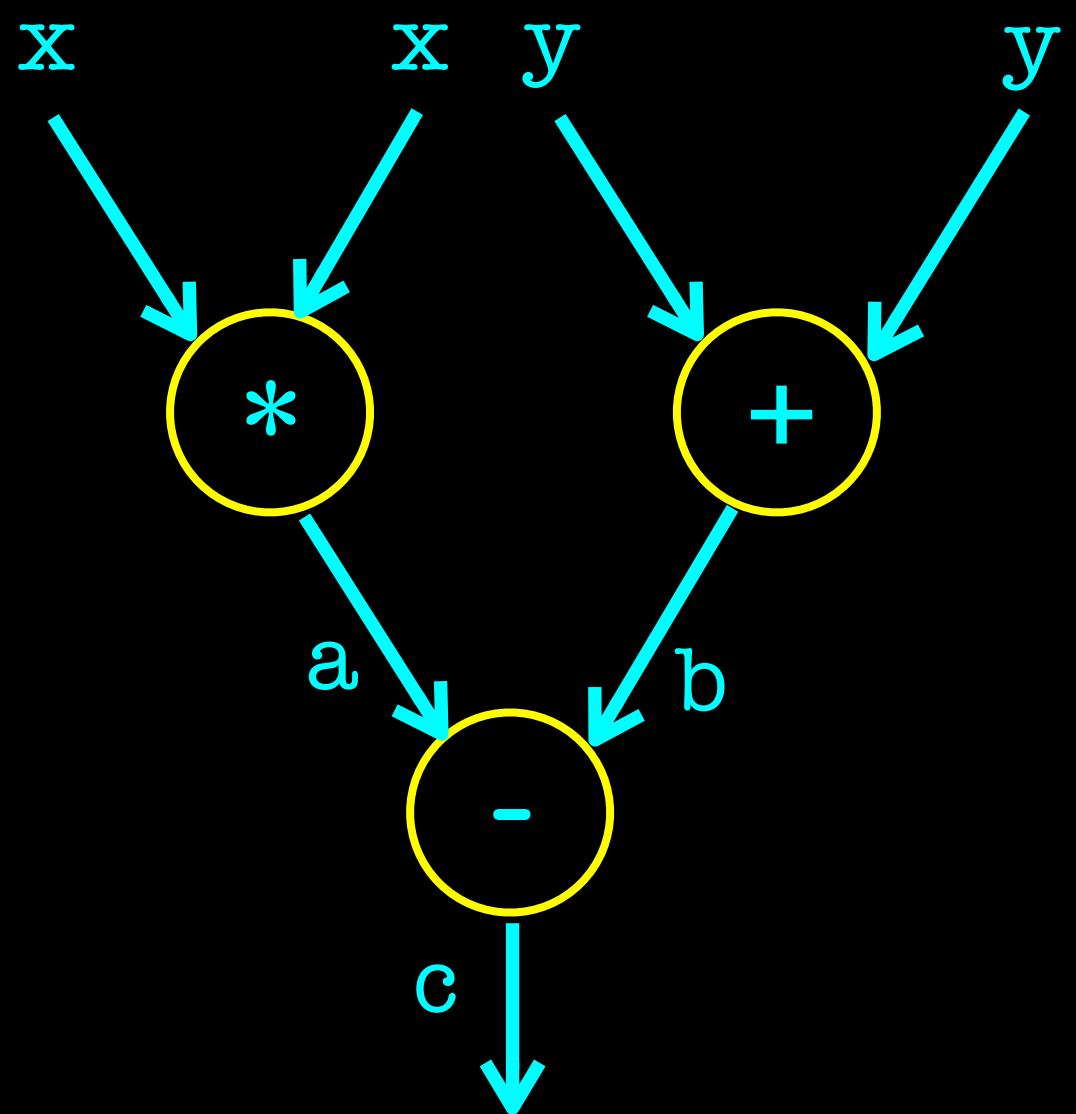
The Dataflow Graph

A graphical model of dataflow computation.
Circles are actors, arrows are arcs.
Jack Dennis, 1975

```
int fun(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a - b;
  b = y+y;

 return c;
}
```



Values flowing through graph == tokens

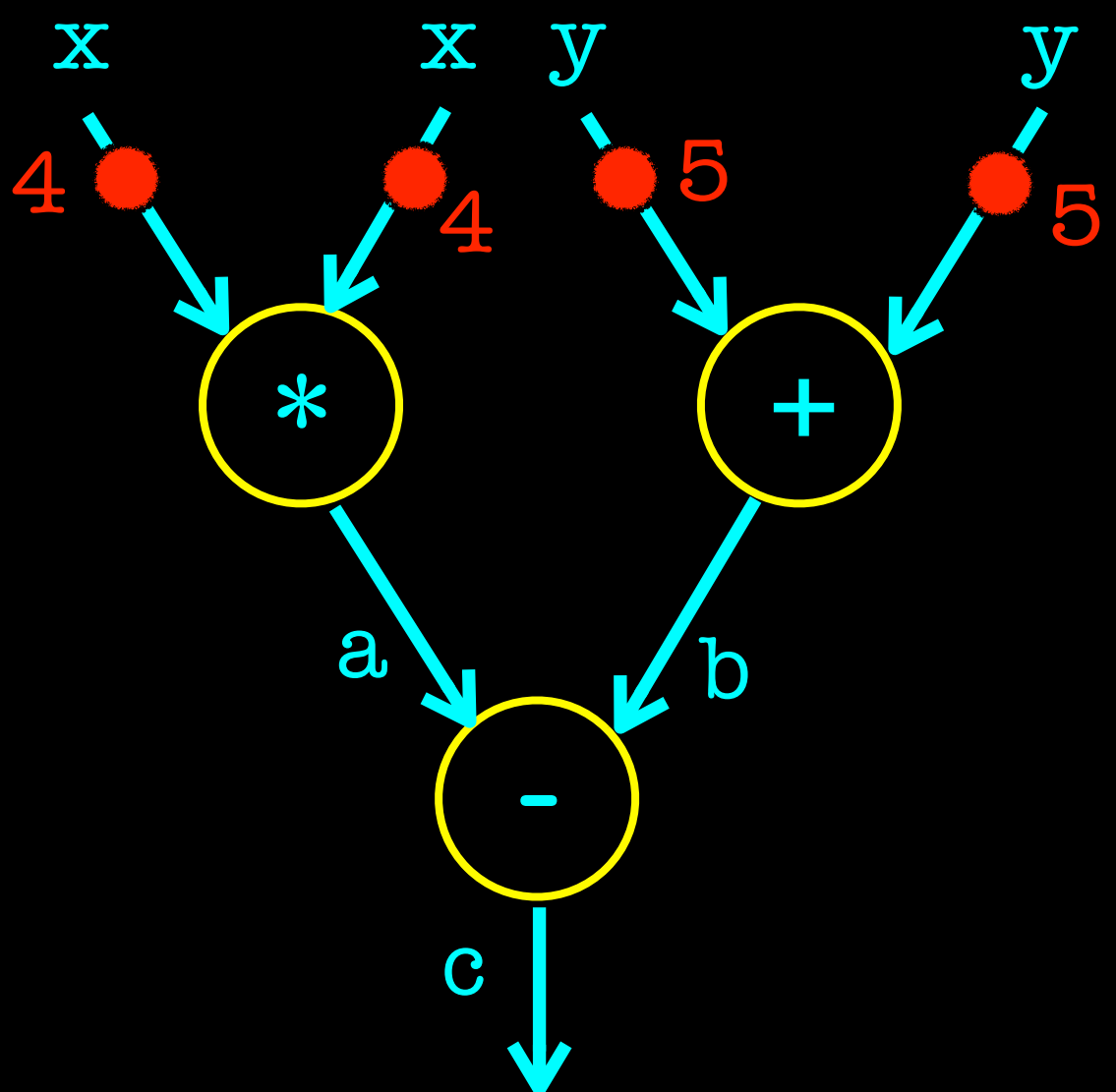
To call a function: put value tokens on input arcs

fun(x = 4, y = 5)

```
int fun(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a - b;
  b = y+y;

 return c;
}
```



The firing rules for actors

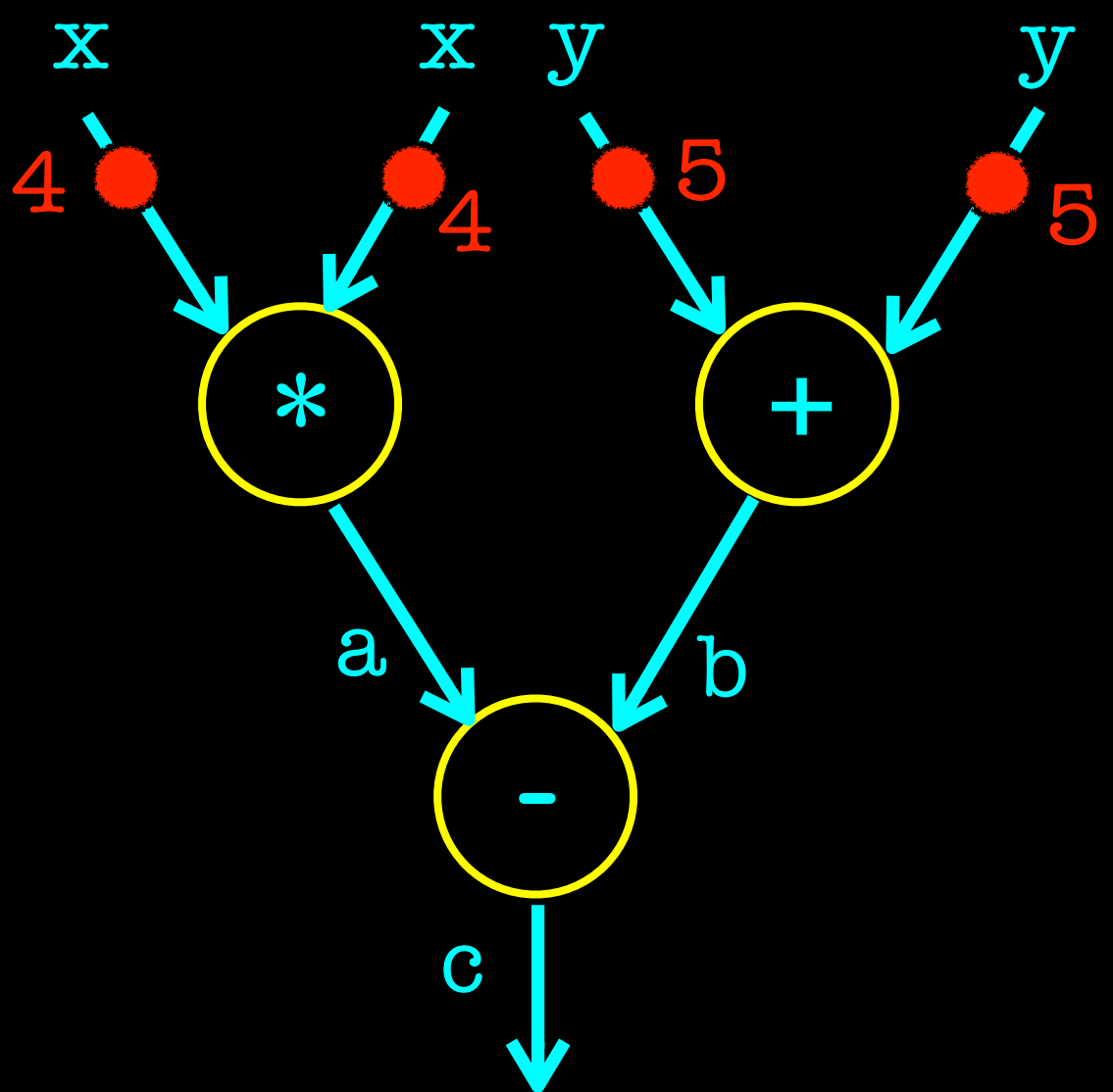
Both input arcs contain a token, and ...

... the output arc is empty (no token).

```
int fun(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a - b;
  b = y+y;

 return c;
}
```



What happens when an actor fires?

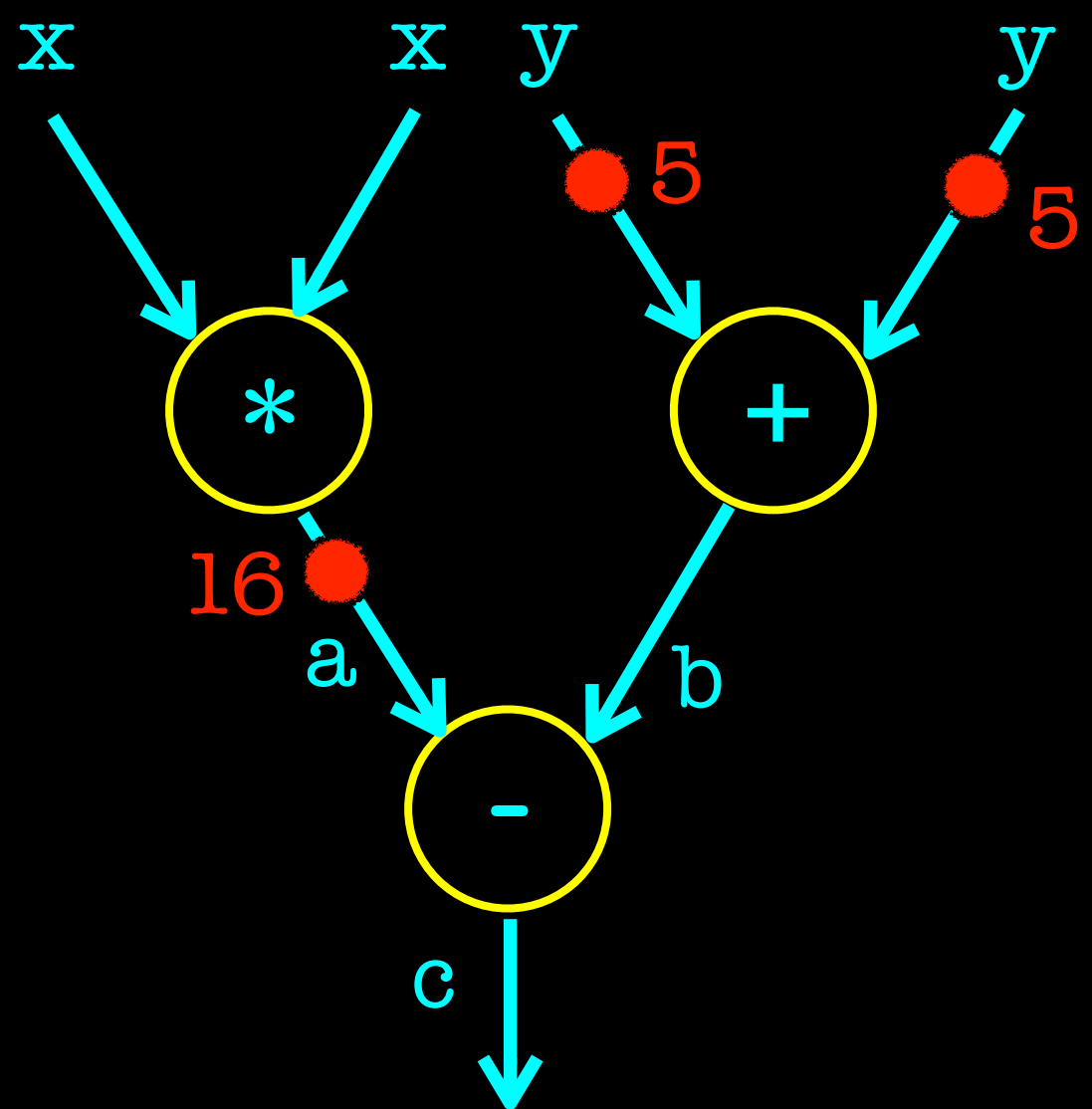
Tokens are removed from both input arcs ...

... the computed token is placed on output arc.

```
int fun(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a - b;
  b = y+y;

 return c;
}
```



What happens when an actor fires?

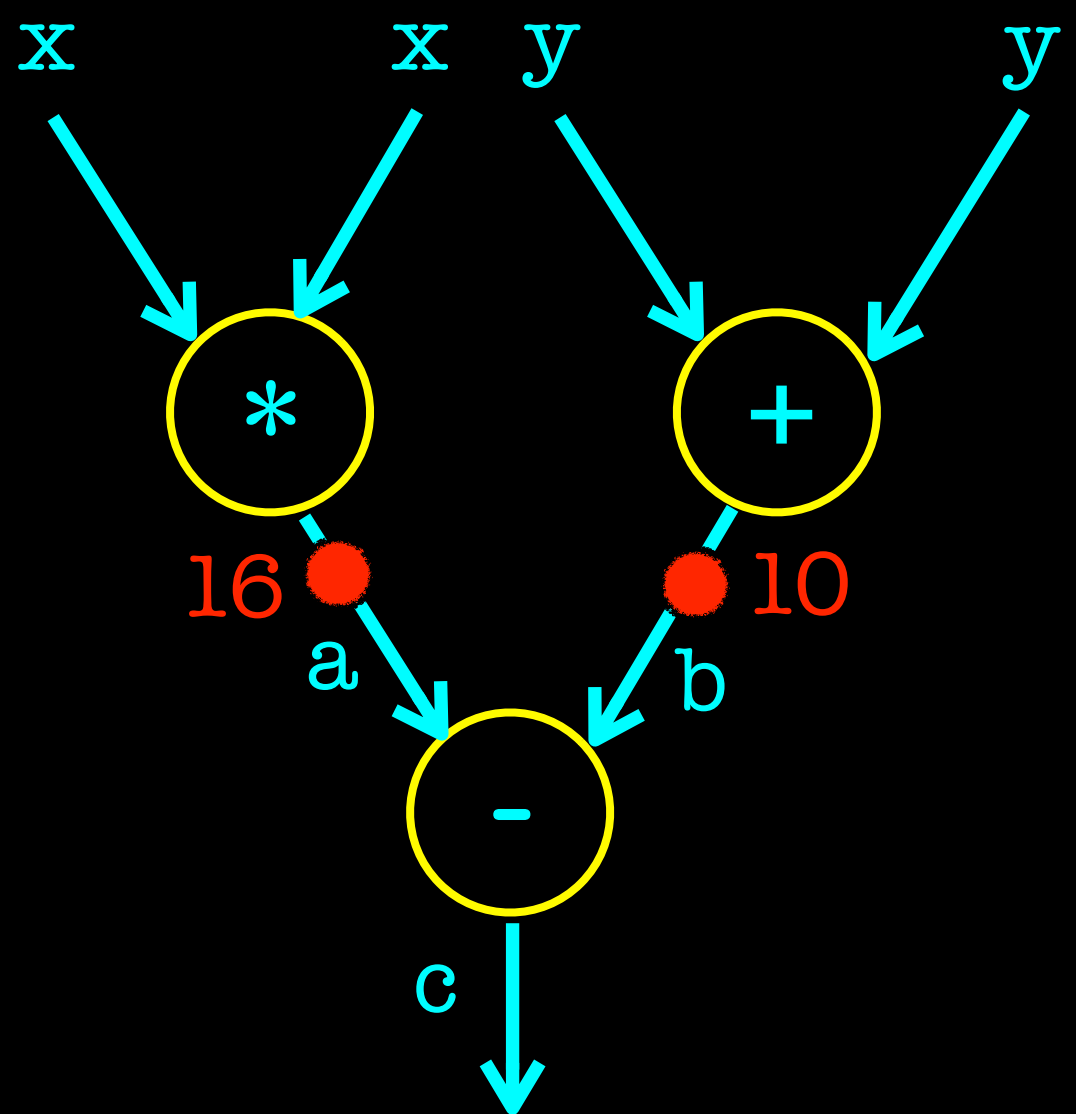
The * and + actors could have fired concurrently.

But the - actor had to wait for * and + to fire.

```
int fun(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a - b;
  b = y+y;

 return c;
}
```



The function call has ended

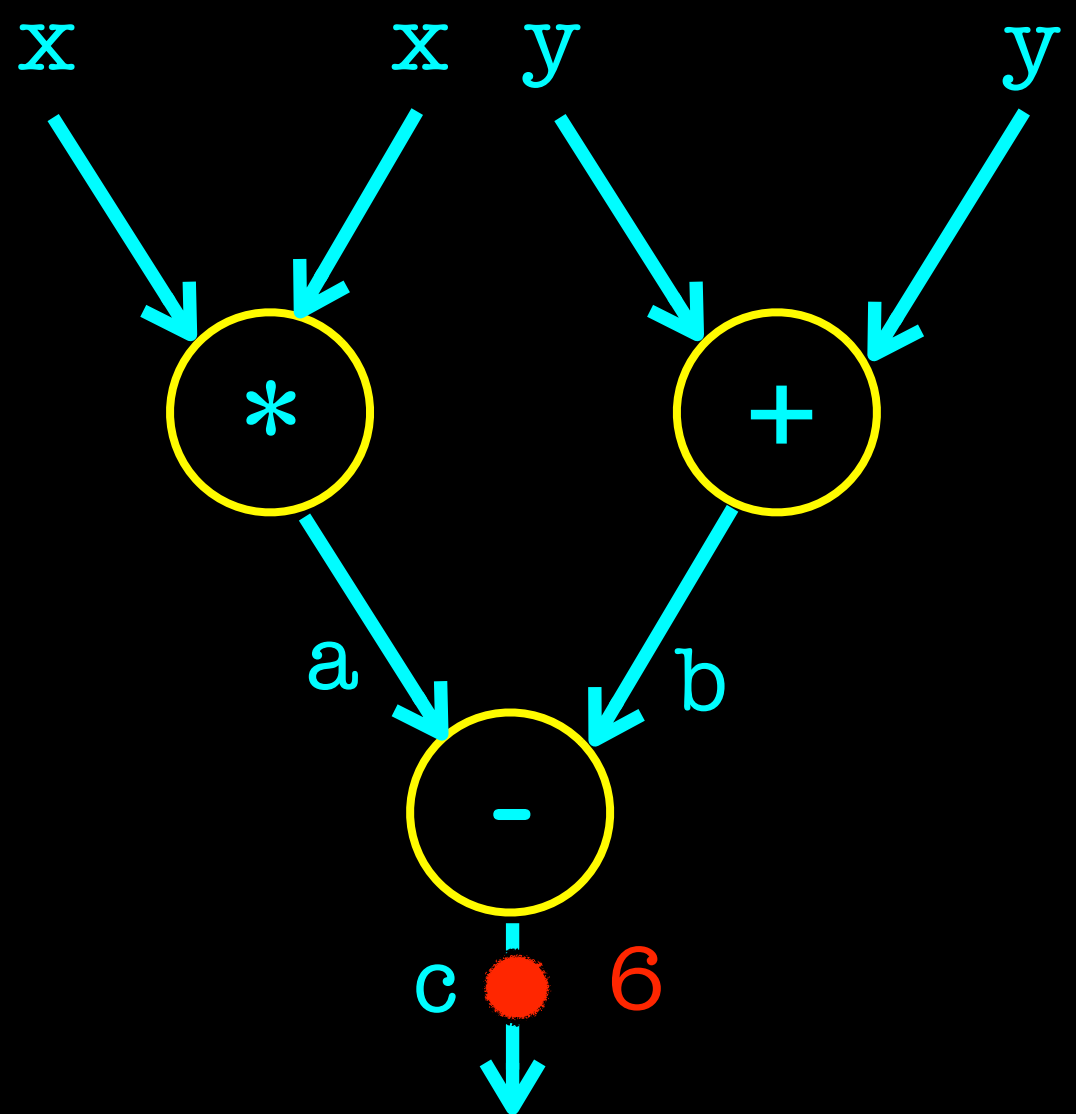
It has the "feel" of hardware ...

... but by itself, it's not an implementation.

```
int fun(int x, y)
{
  int a, b, c;

  a = x*x;
  c = a - b;
  b = y+y;

 return c;
}
```

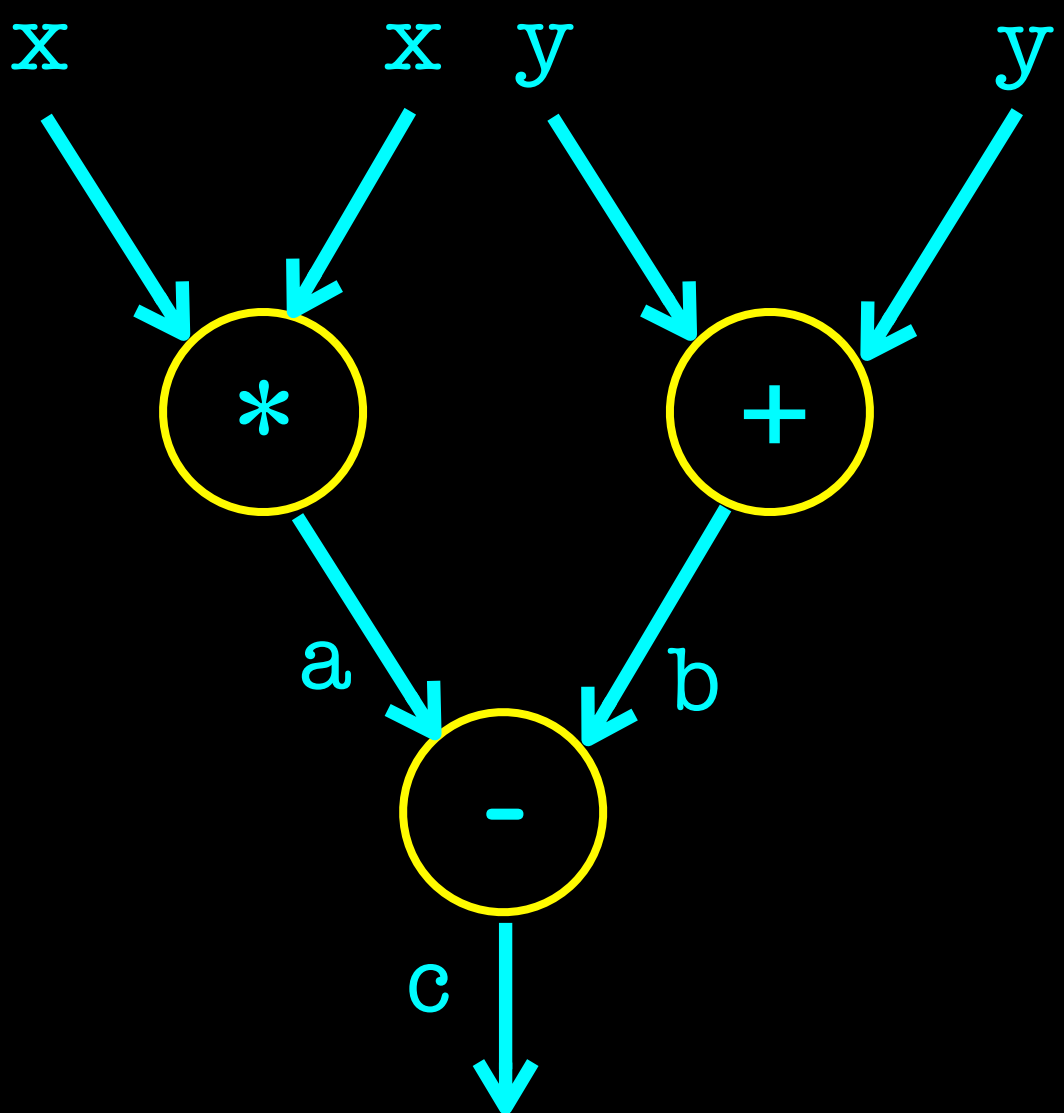


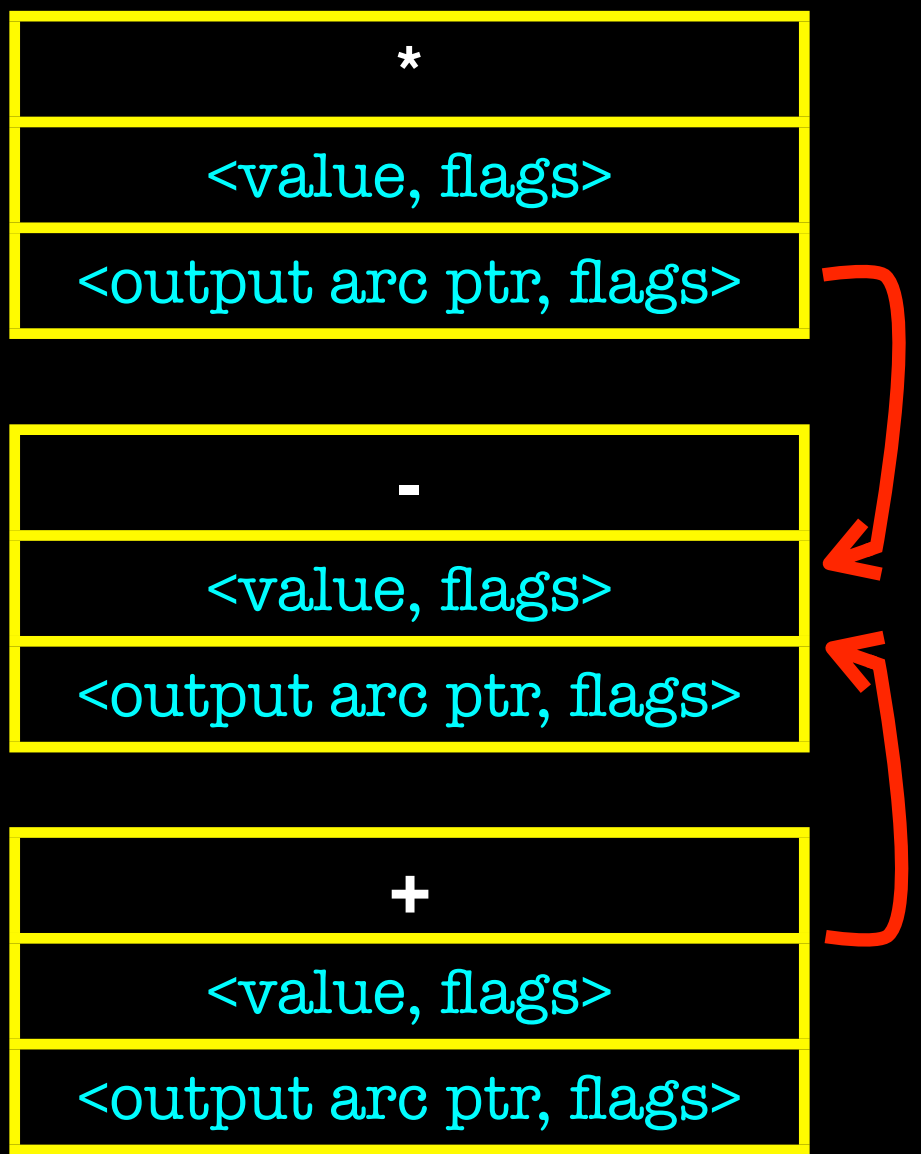
Monsoon: an Explicit Token-Store Architecture

Gregory M. Papadopoulos
Laboratory for Computer Science
Masachusetts Institute of Technology

David E. Culler
Computer Science Division
University of California, Berkeley

The activation frame of the function. Like a stack frame.

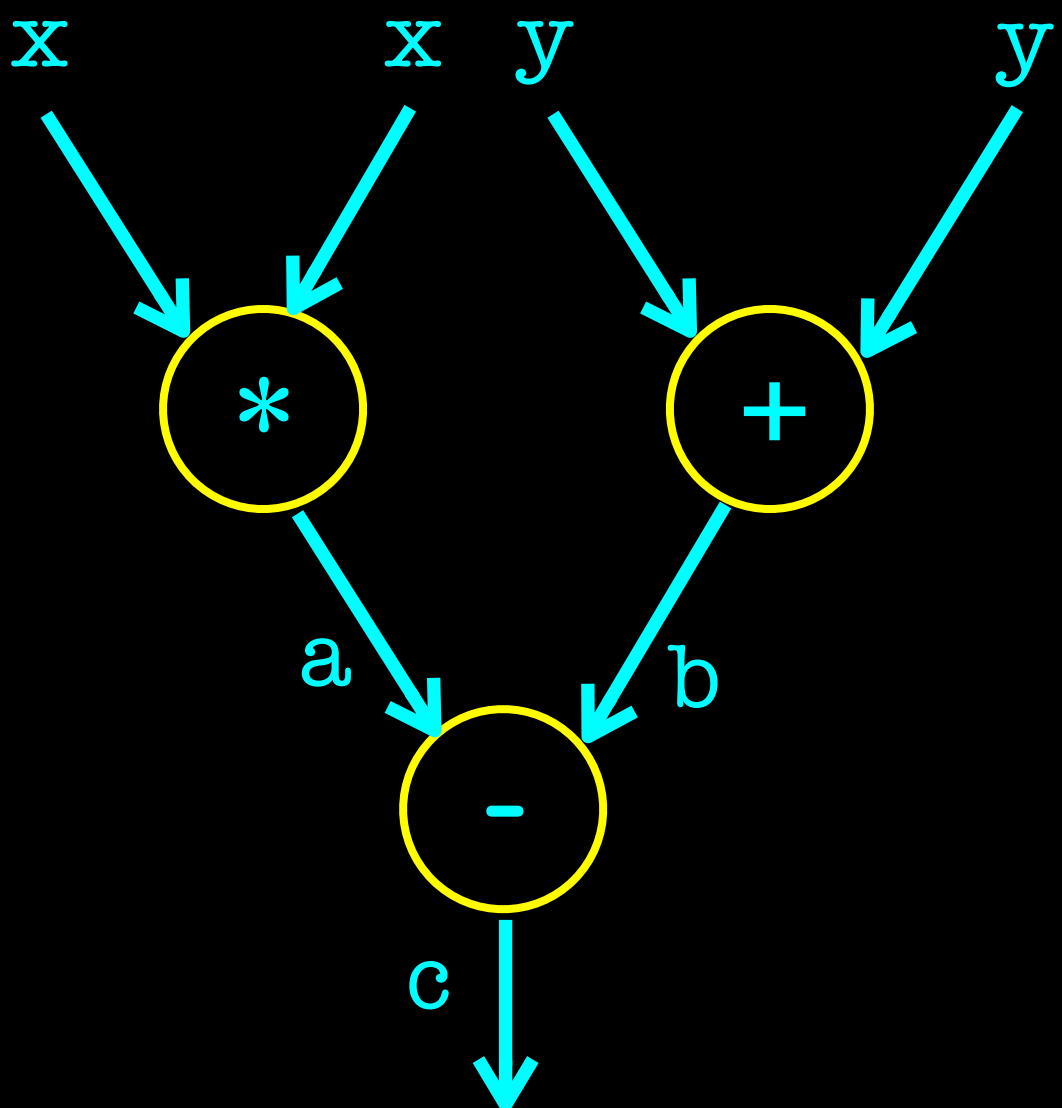


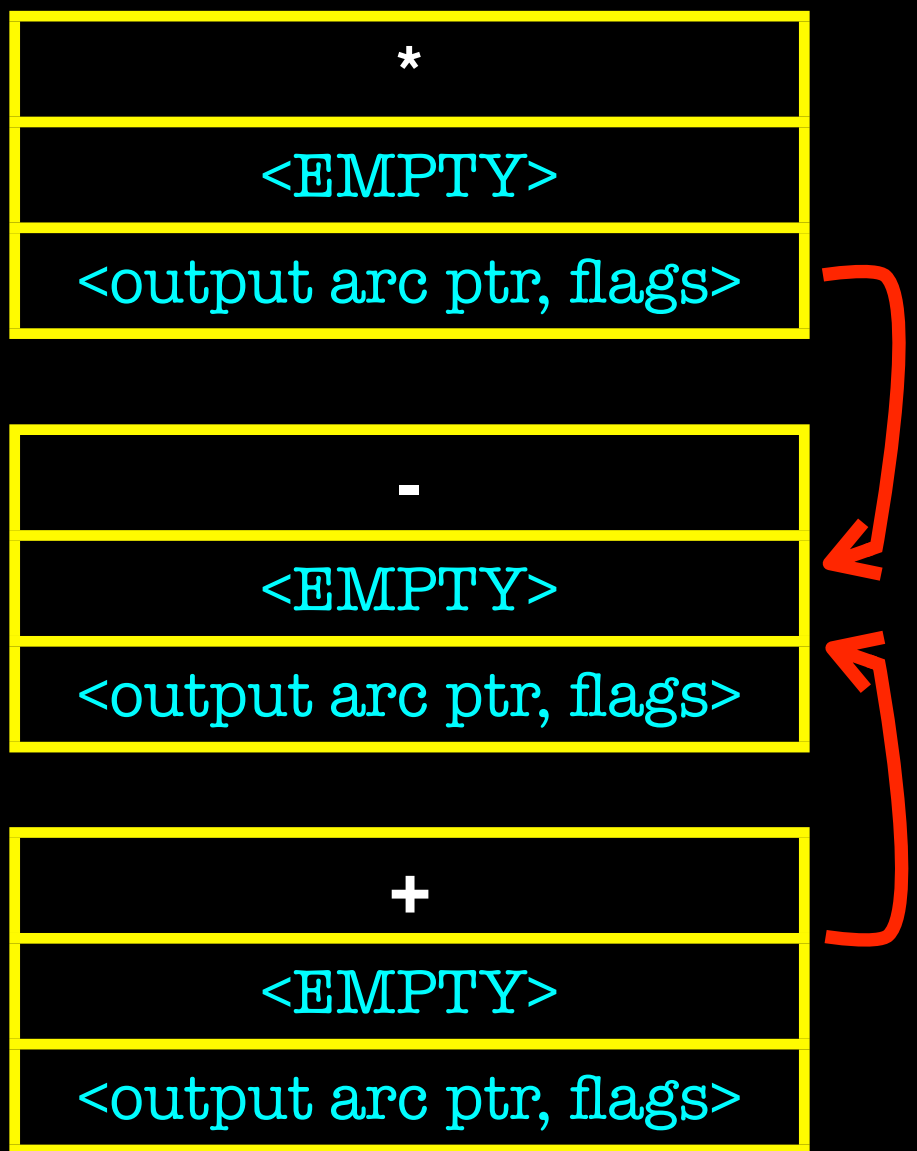


The intialization state

All values are empty.

Other fields reflect dataflow graph.



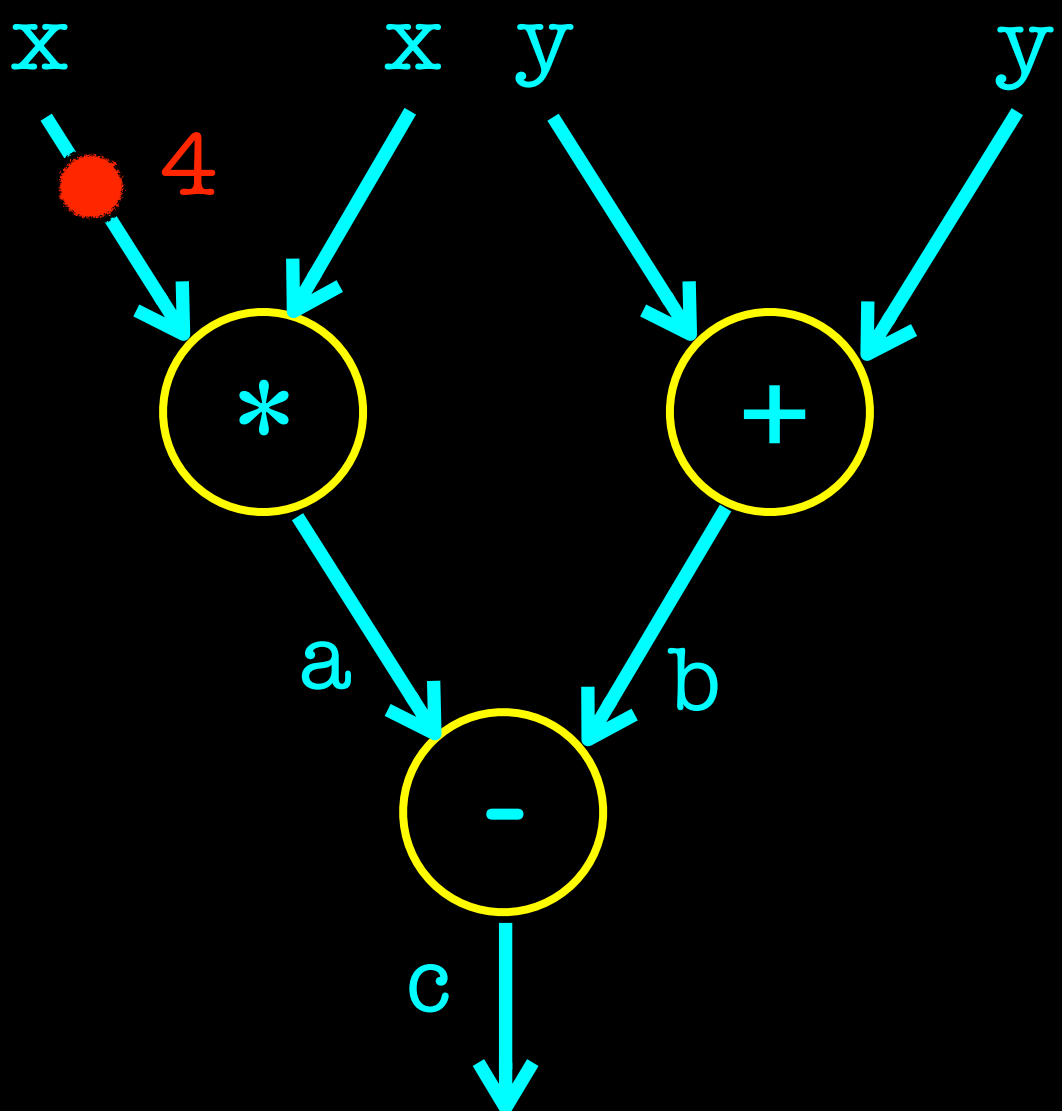


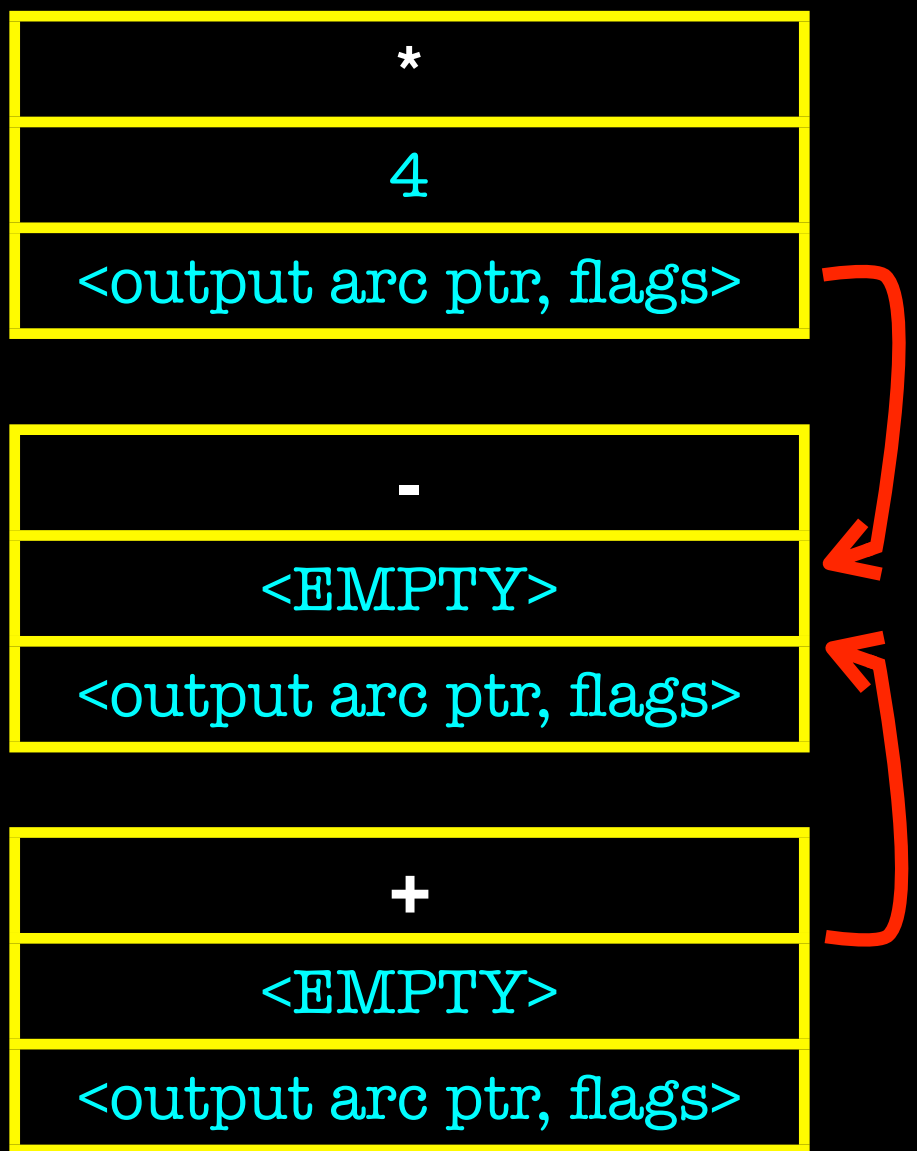
We process tokens one by one (left x=4)

A token has a pointer to an actor frame.

If frame is empty, it places its value in it.

Note: we don't remove red dot from graph.



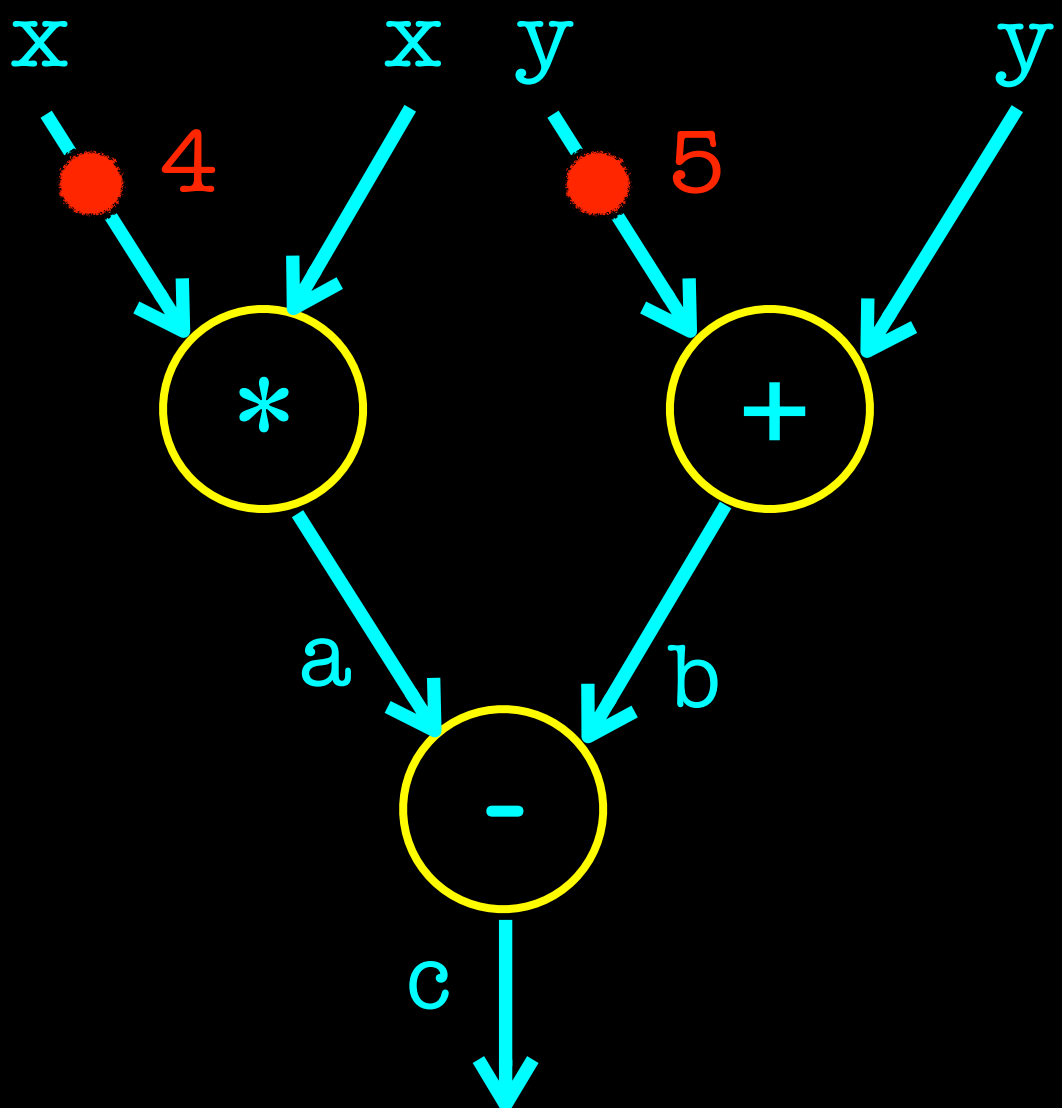


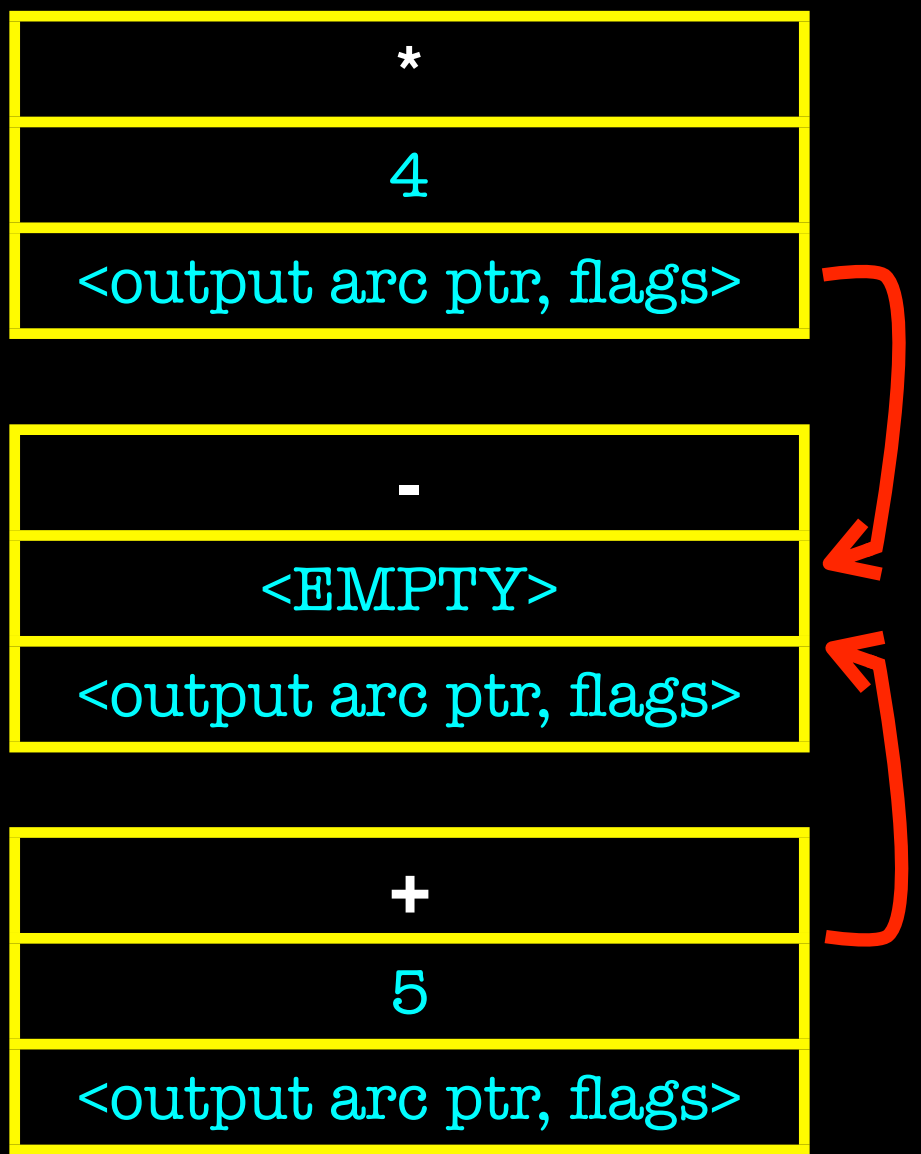
We process tokens one by one (left y=5)

A token has a pointer to an actor frame.

If frame is empty, it places its value in it.

Note: we don't remove red dot from graph.

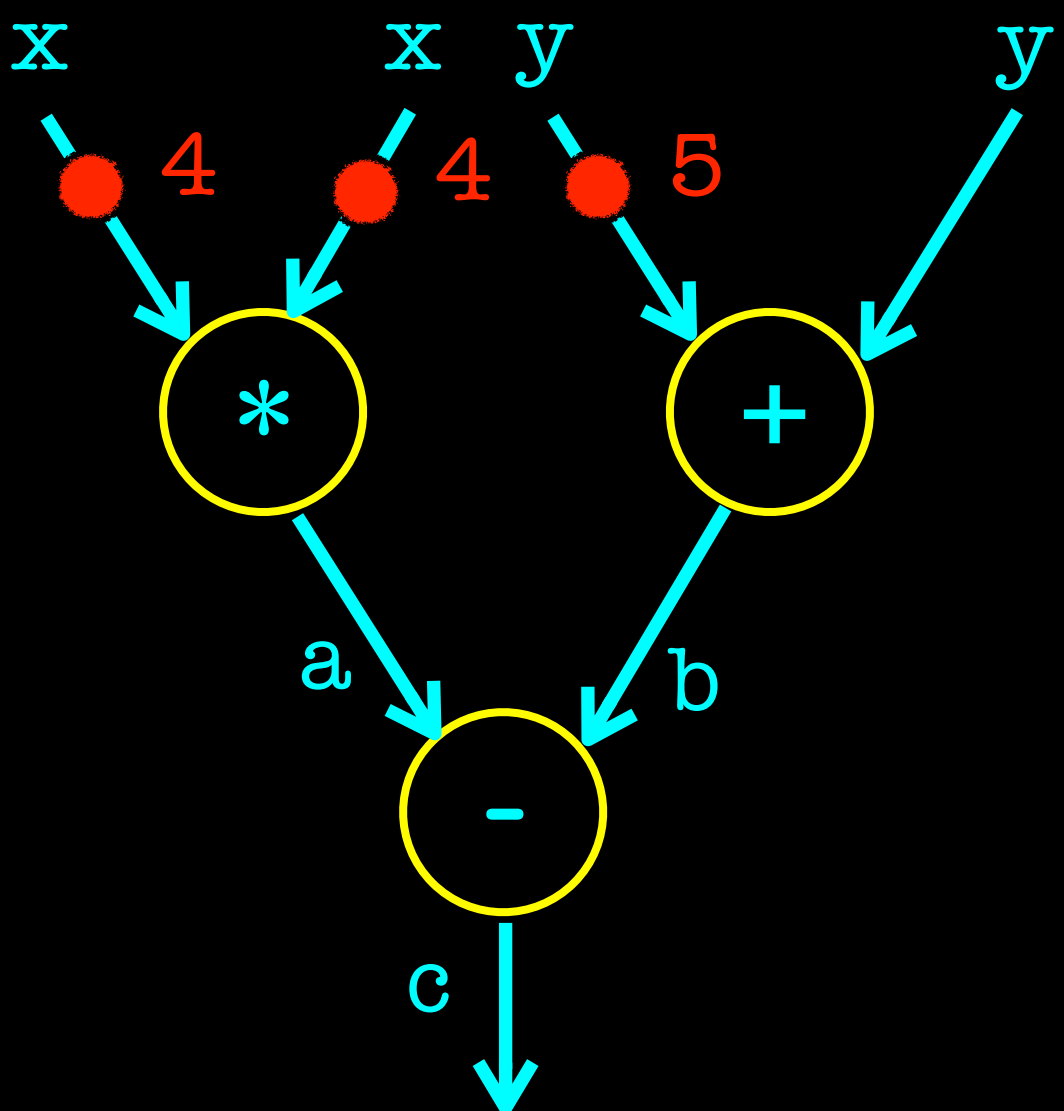


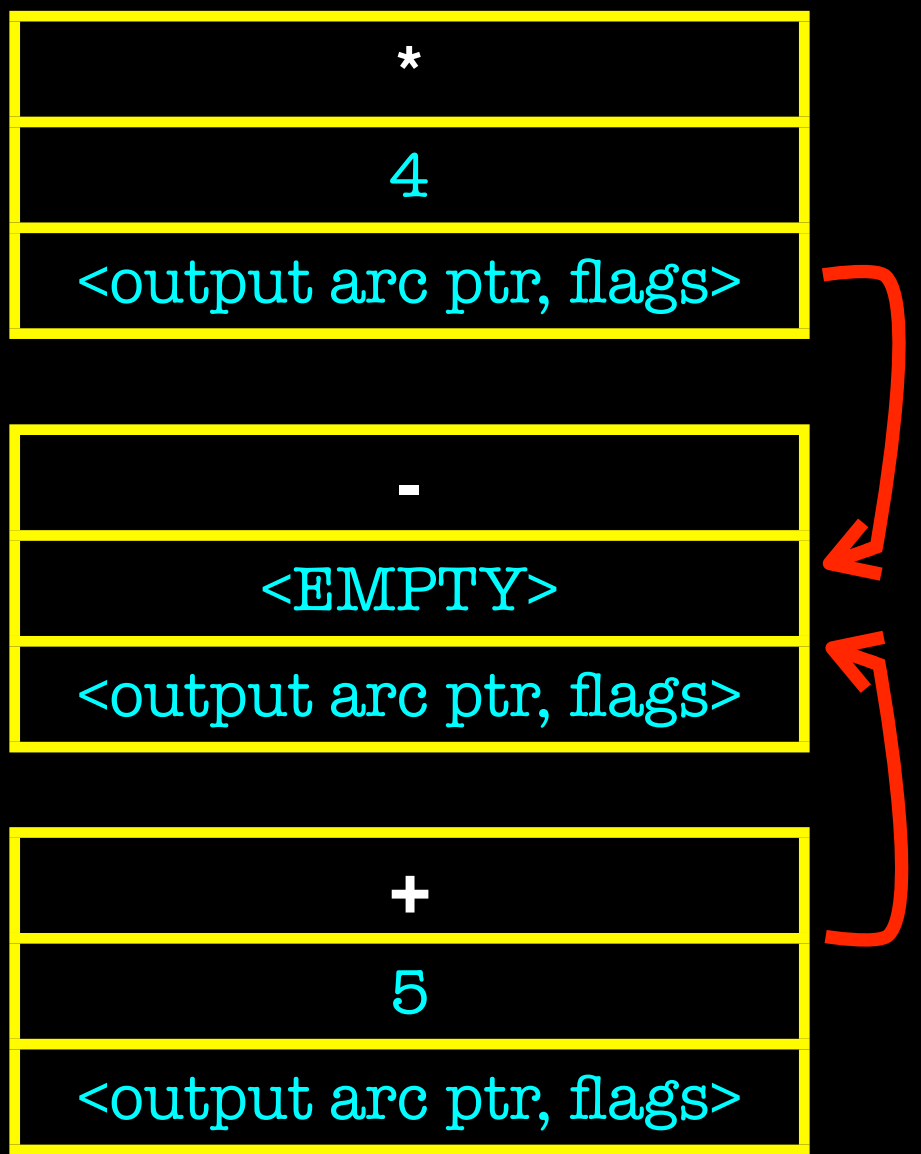


We process tokens one by one (right x=4)

If frame is full, token empties it, and computes.

This slide shows pre-computation state.

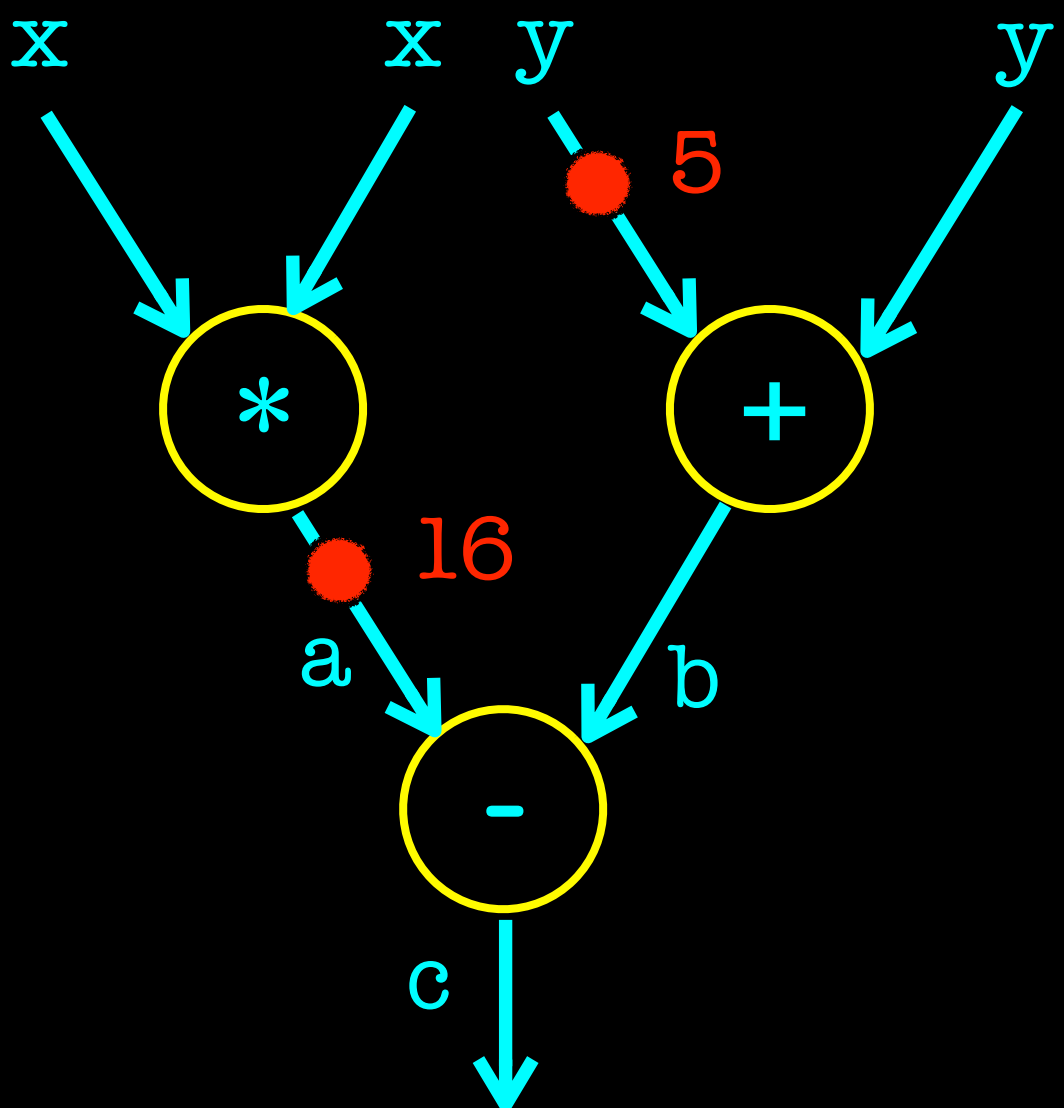


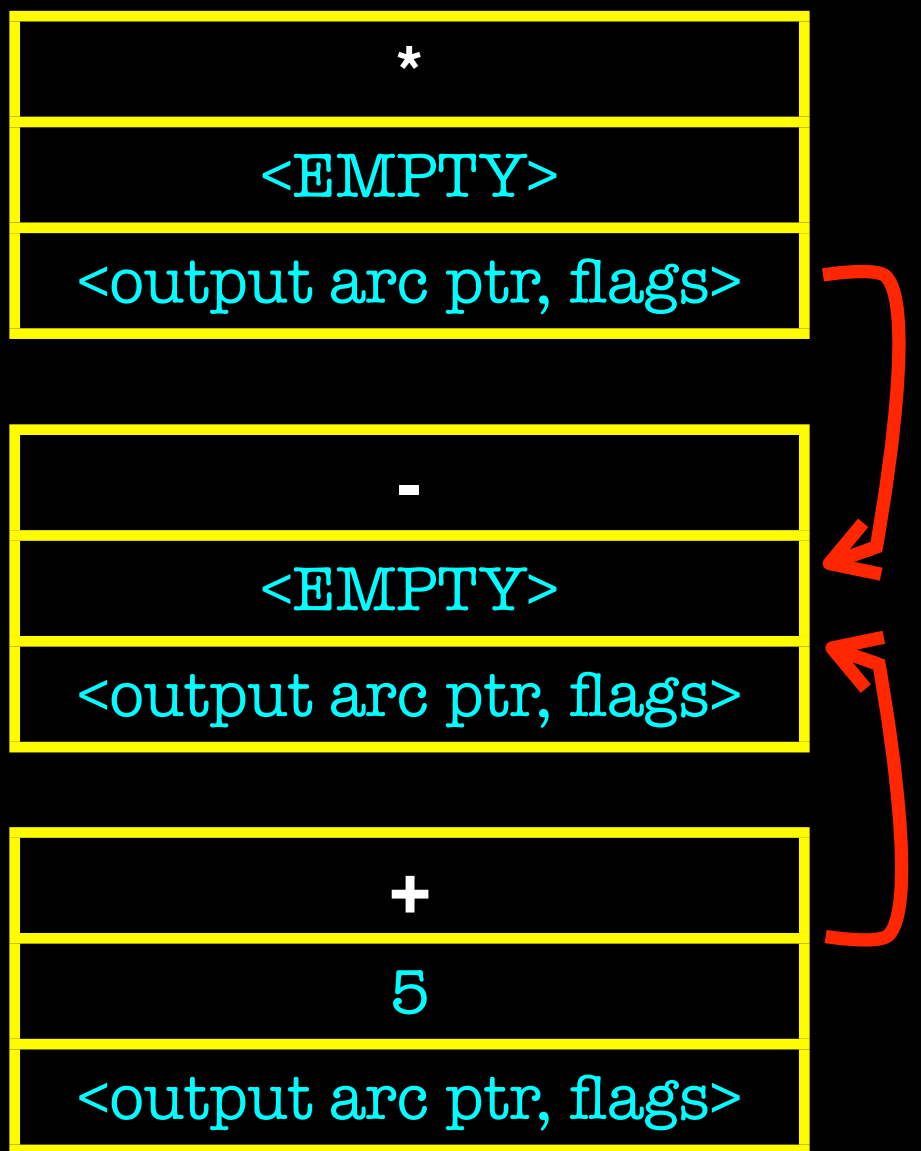


We process tokens one by one (right x=4)

Remove * input dots for x from graph, add * output dot.

Repeat token processing until data graph is token-free.

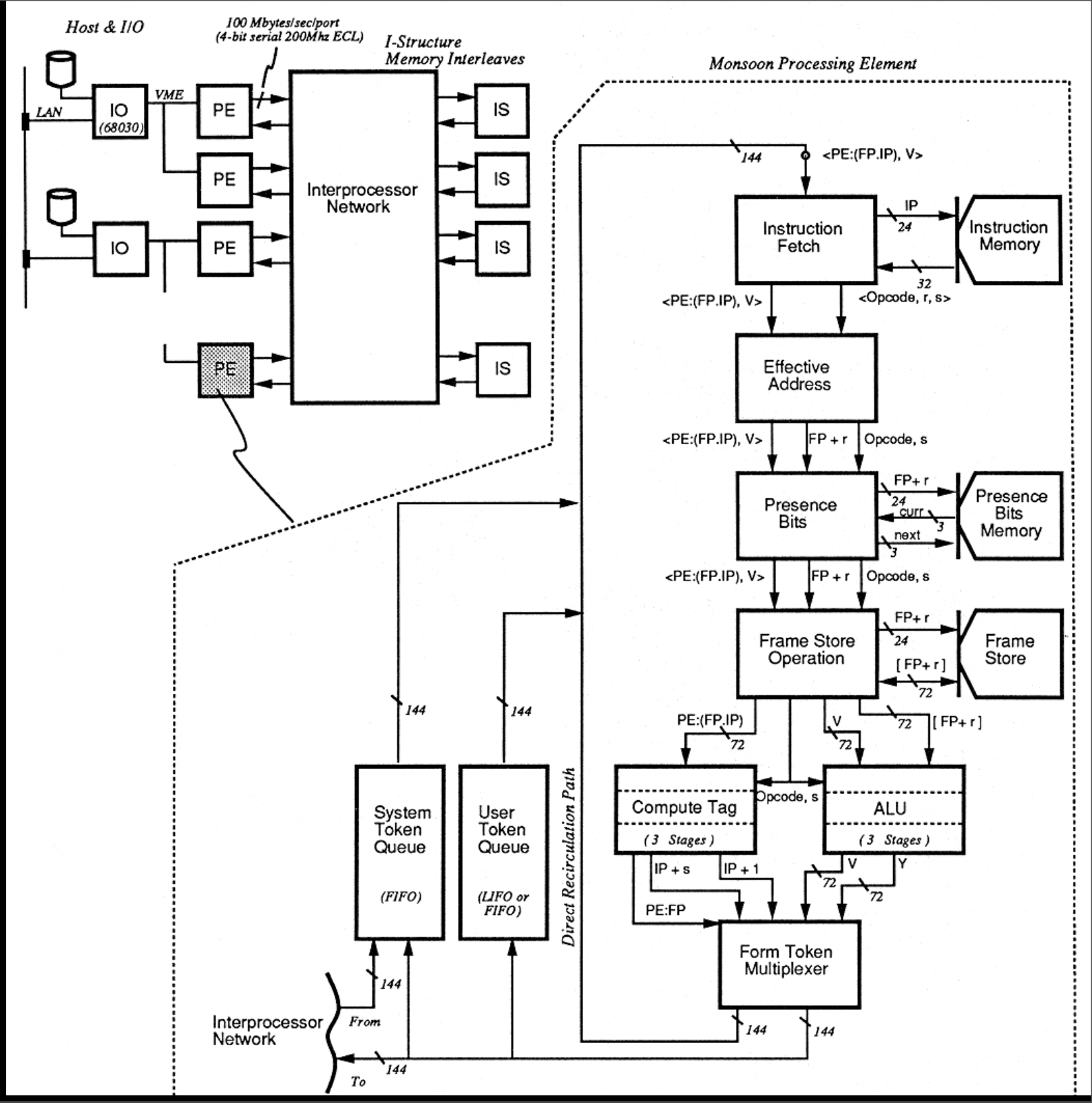




How compute happens.

Pipelined hardware

Monsoon 1990.



Observations

- The basic idea seems to work.
  But it feels like a toy in so many ways ...
- Partial wish list ... while keeping dataflow nature.

Reusing registers during execution.

for(), while()
if/then, switch

Function calls within functions.

Bring back main memory!

Fix deadlock.

Efficient arrays.

This is what research looks like ...

On Tuesday

Graphics week ...

	T 4/15	GPU + SIMD + Vectors I		Chapter 4.2, 4.4-6.

Have a good weekend !