The FLEET Architecture

Ivan E. Sutherland

VP and Sun Fellow
Visiting Scholar, UC Berkeley

September 15, 2006

research.cs.berkeley.edu/fleet

This material is the product of a collaboration between
Sun and UC Berkeley. Use for any academic purpose.

UCIES# 2006-is34
14 September 2006

Four parts to this talk

Why MOVE
Concurrency
Computing
Programming

Part 1: Why MOVE

Technology has changed

THEN

Logic expensive - vacuum tubes

Communication cheap - copper wire

NOW

Logic cheap - transistors

Communication expensive

- area
- time
- energy

Architecture must also change

THEN

Logic expensive - program operations

Communication cheap - implicit

NOW

Logic cheap - implicit

Communication expensive

- put the programmer in charge

Only one instruction - MOVE

UCIES# 2006-is34
14 September 2006

MOVE is the **only** instruction

Where data go

controls

What happens

MOVE to adder

form sums

MOVE to write unit

record in memory

UCIES# 2006-is34
14 September 2006

We named it FLEET

for speed
but it became

Nautical

SHIP = logic unit

Computer = a FLEET of SHIPs
plus a "switch fabric"

Switch Fabric:

From SOURCE to DESTINATION

data are unchanged

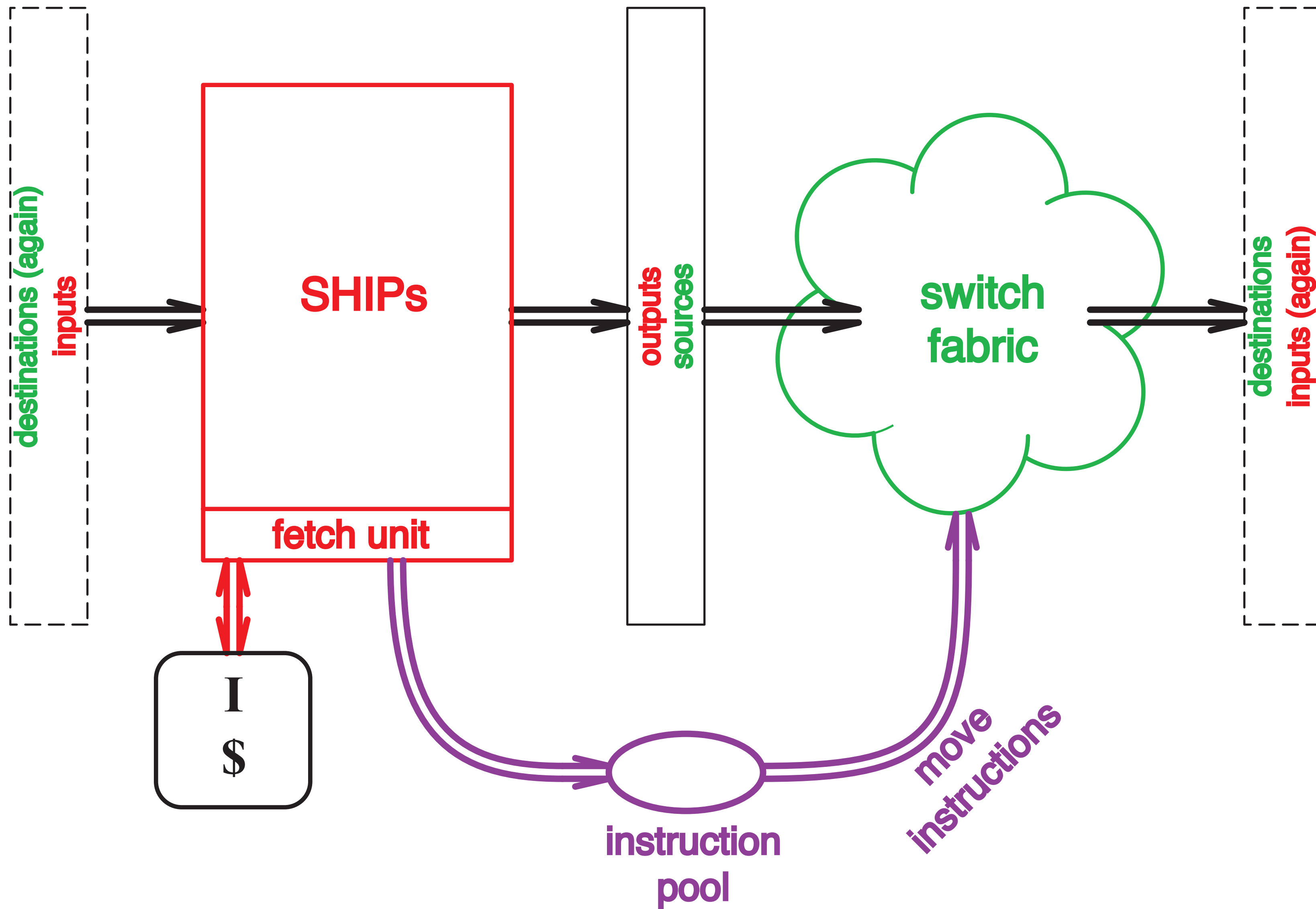
SHIP:

From INPUTS to OUTPUTS

data processed

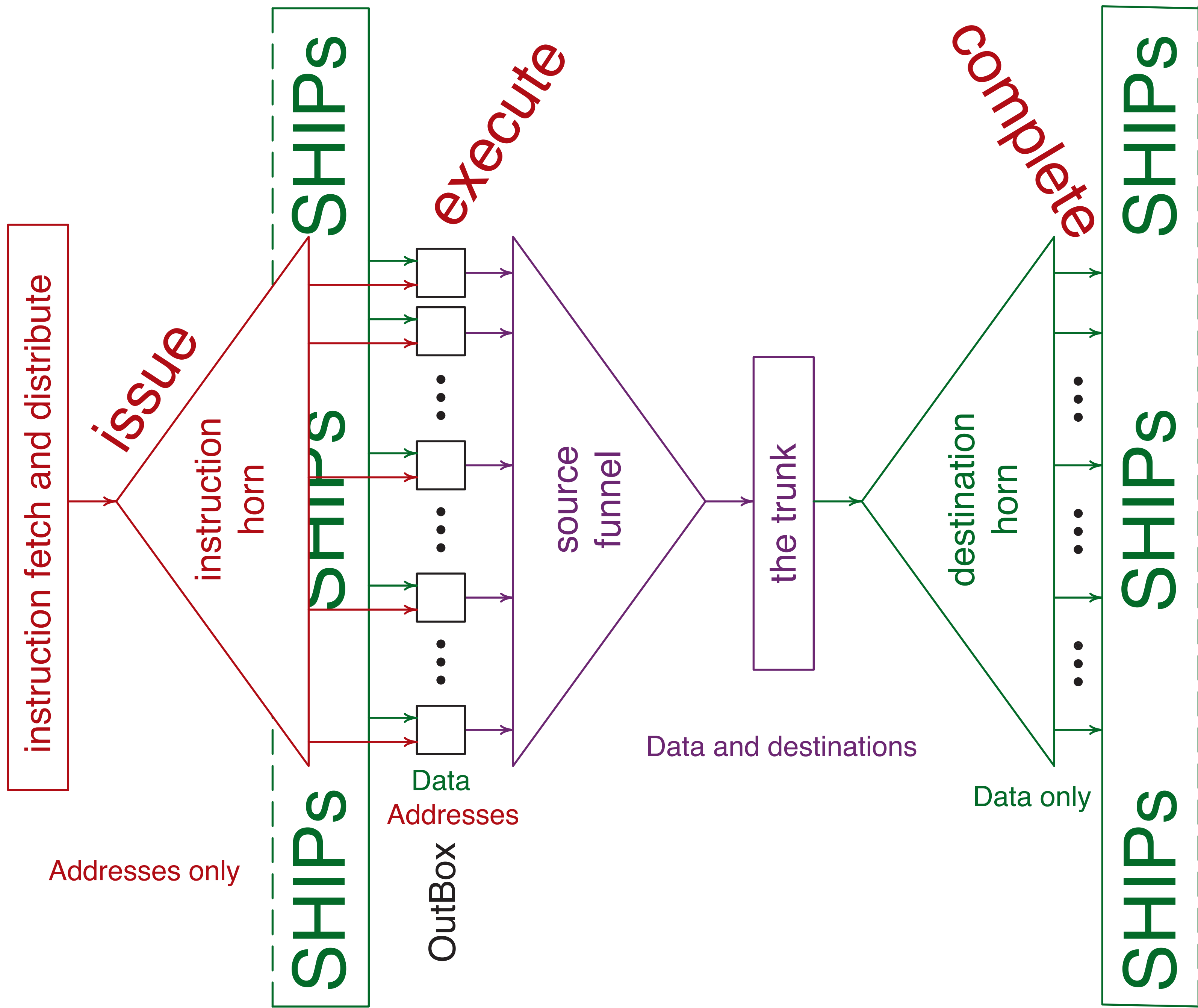
Source = a SHIP's Output

Destination = a SHIP's Input



Slide 8: An Abstract View of FLEET

UCIES# 2006-is34
14 September 2006



Slide 9: An Implementation of FLEET

Asynchronous MOVE

MOVE "issues" when
it enters the pool

MOVE "executes" when
data are available at its
source, i.e. a SHIP's output

MOVE "completes" when
space is available at its
destination, i.e. a SHIP's input

End of Part 1

Move machine

SHIPs + Switch Fabric

Asynchronous MOVEs

Part 2 - Concurrency

In the Switch Fabric

and for multiple MOVEs

Concurrent MOVEs !

Concurrent MOVEs

When MOVEs execute

MOVEs execute when data are available
Each MOVE waits as long as required
All waiting MOVEs are concurrent

Code Bags

Limited sequence inside bag
Zero or more successor bags
Each bag must fetch successor(s)
Each BAG's code enters
    the POOL concurrently

Sequence guarantees

Source sequence

MOVEs with same source
execute in issue sequence
Identical instructions complete in
execute sequence
Bag is a bag of lists with same source

Code bag sequence

All MOVEs in bag issue before
any from the "next" bag.

Two MOVE types

MOVE vs COPY

MOVE drains source data

COPY preserves source data

Plus

MOVEs with same source
  execute in issue sequence

COPY . . . MOVE duplicates

Code Bag

Unordered set of lists

Code Bag Descriptor (CBD)

Pointer to left of bag

Pointer to right of bag

Pointer to literals for this bag

JUMP replaced by:

(Code Bag Descriptor) => Fetch

Departed, R.I.P.

Program counter
JUMP instruction
Code BLOCKS

Sequential process

JUMP's epitaph

Confounded two ideas

- Pointer to next block
- End mark for this block

Both useful sooner

- Pointer - fetch while execute
- End - know how much to fetch

Why? Cost to store pointer

Concurrency vocabulary:

Concurrent = no order specified
i.e. any order is acceptable

Words to avoid for Bags

Beginning - End
First - Last - Next

Use sequence-free Words

Left side - Right side
Red part - Green part

Some implications:

Execution sequence not in code

Possible indeterminacy

Face to face with
  low level concurrency

Sequencing by SHIPs

Data flow sets execute sequence

Tokens are a useful data type

End of Part 2

MOVE & COPY

Asynchronous Switch Fabric

Code Bags - limited sequence

Part 3 - SHIPs

Asynchronous interfaces

Outputs can be empty

- MOVE will wait at source

Inputs can be full

- MOVE will wait at destination
  and may cause DEADLOCK

Data Types

- Token (zero bits)
- Boolean (one bit)
- Word (N bits)
- Code Bag Descriptor (? bits)

Record = 1 to 8 words

- Ordered
- Knows own length
- Atomic record MOVE

Source variants

COPY record - preserves it

MOVE record - drains it

COPY 1st word - preserves it

MOVE 1st word - record shorter

Destination variants

Partial - record remains open

Final - record sealed

$A_w \Rightarrow B_p$; $A_w \Rightarrow B_f$; $A \Rightarrow C$;

Simple SHIPs

Single address

- Bit bucket
- Constants: ZERO, TRUE, TOKEN
- Random number

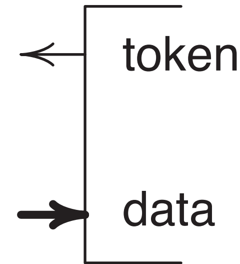
Record store

- One or more records
- First In First Out (FIFO)
- Capacity of FIFO

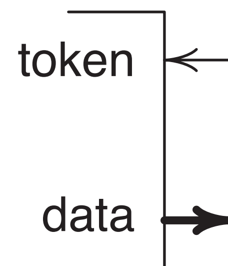
Interfaces

A token appears here after each data input, but only when the interface is ready for further input.

Data arriving here may never be overwritten. Record may fill the interface, blocking further input until the interface is ready.



Pipeline receiver interface



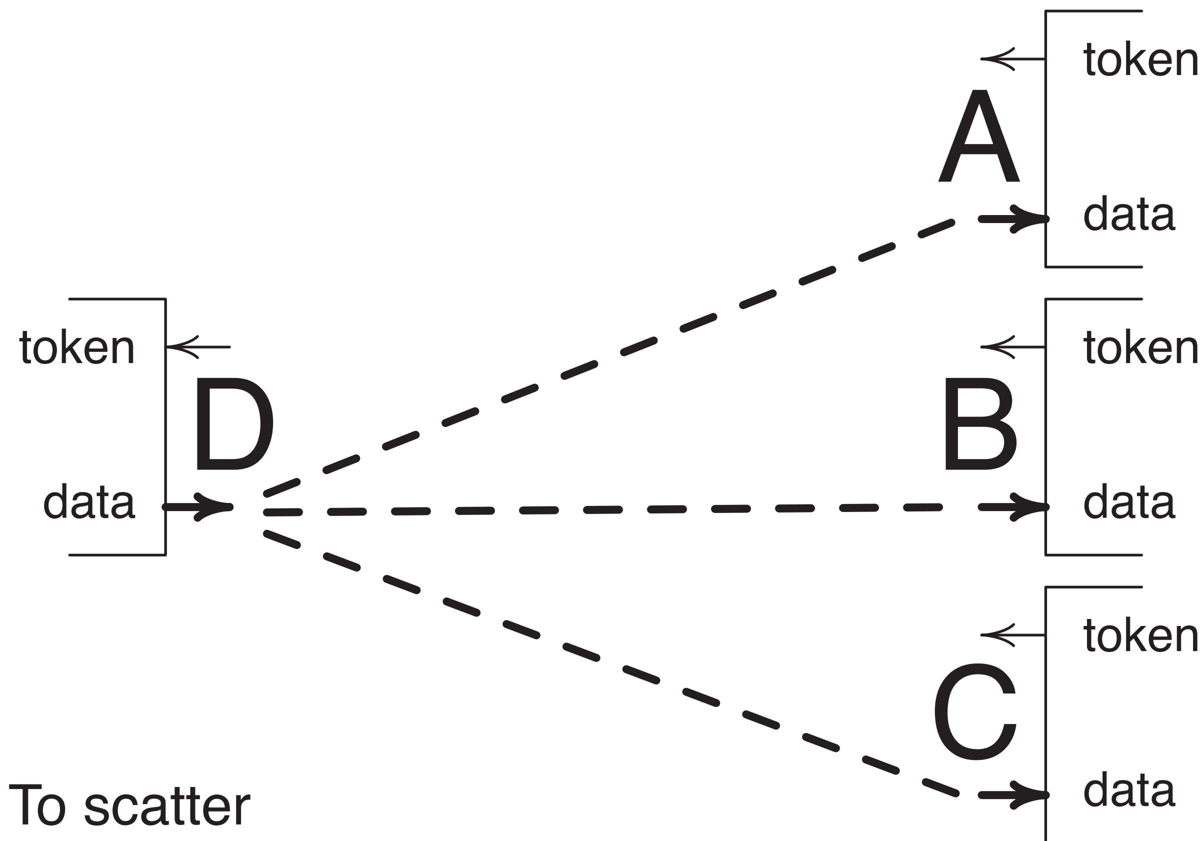
This destination must receive a fresh token for each new output data value.

A new record appears here, if available, only after the previous record is drained and a fresh token has arrived. Move instructions may copy or drain data values.

Pipeline sender interface

Connect with two MOVEs

Scatter via source sequence

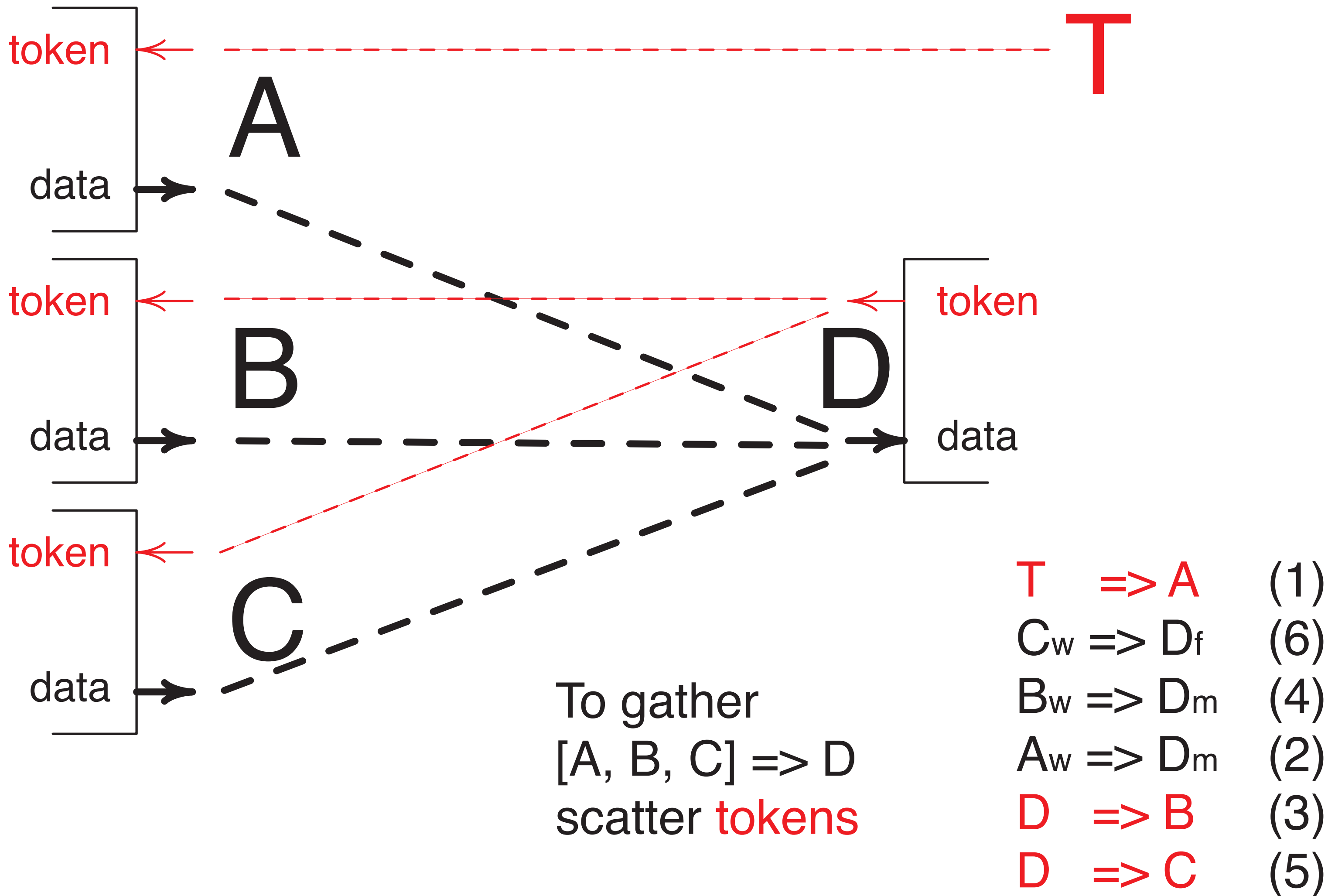


To scatter
[D1, D2, D3] => A, B, C
use source sequence

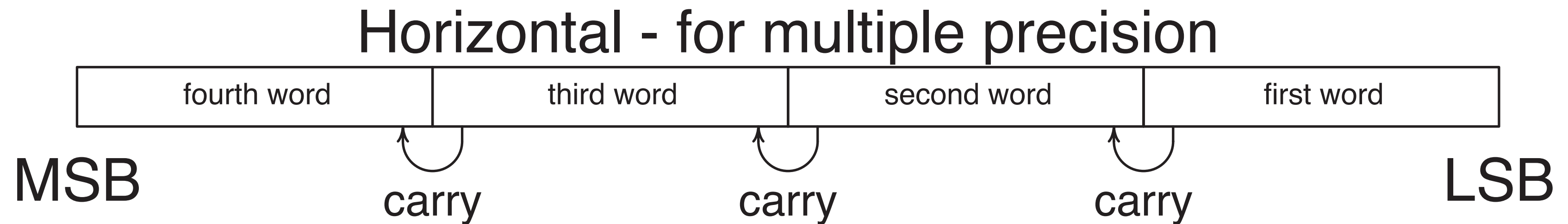
$D_w \Rightarrow A_f$ (1)
$D_w \Rightarrow B_f$ (2)
$D_w \Rightarrow C_f$ (3)

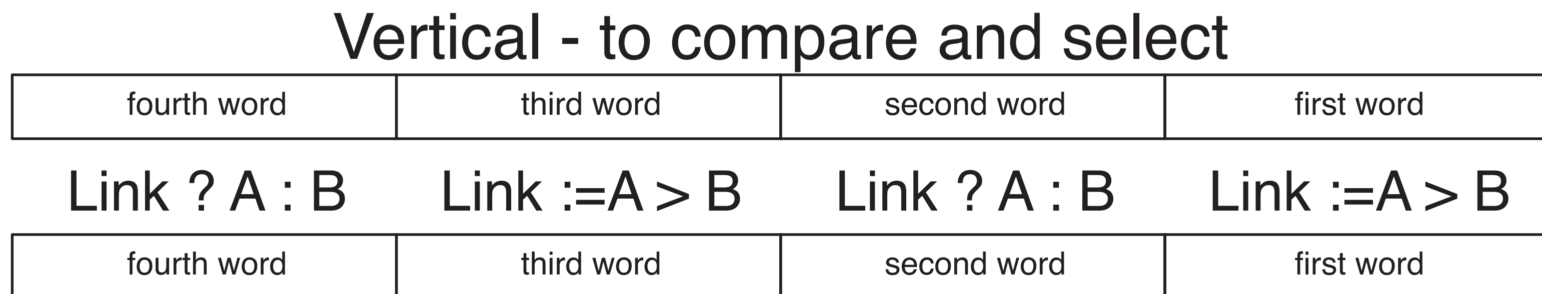
UCIES# 2006-is34
14 September 2006

Scatter tokens to gather data

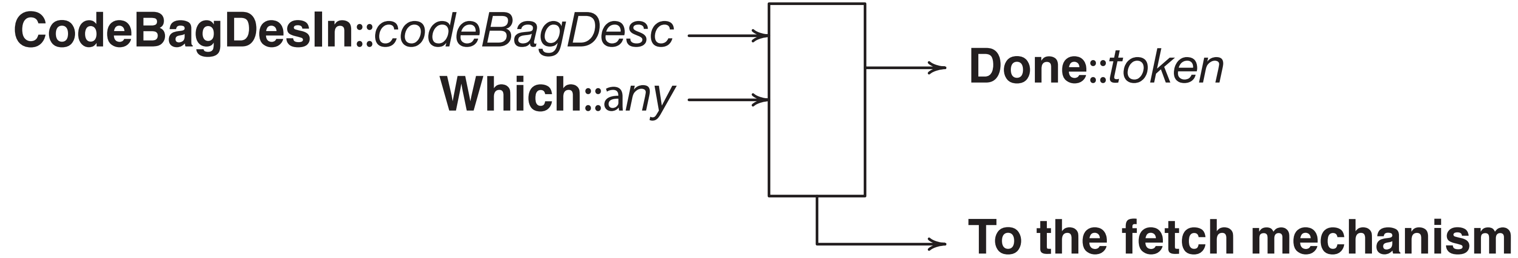


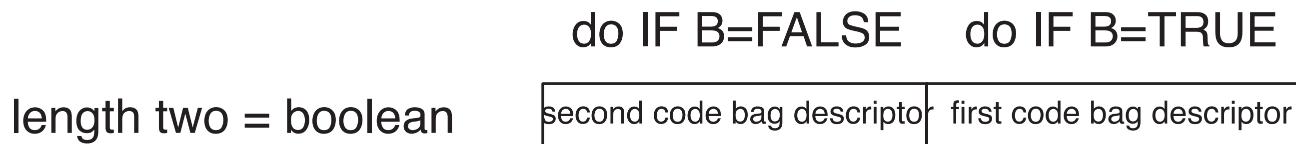
Uses of records

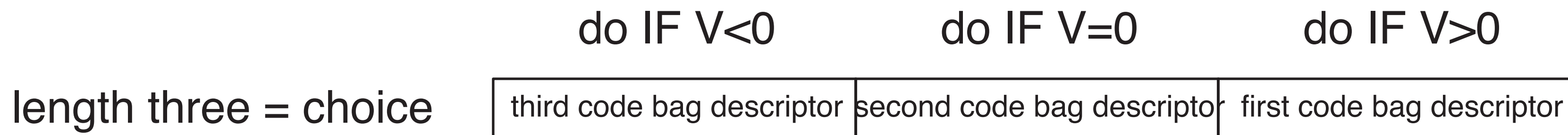




Fetch SHIP

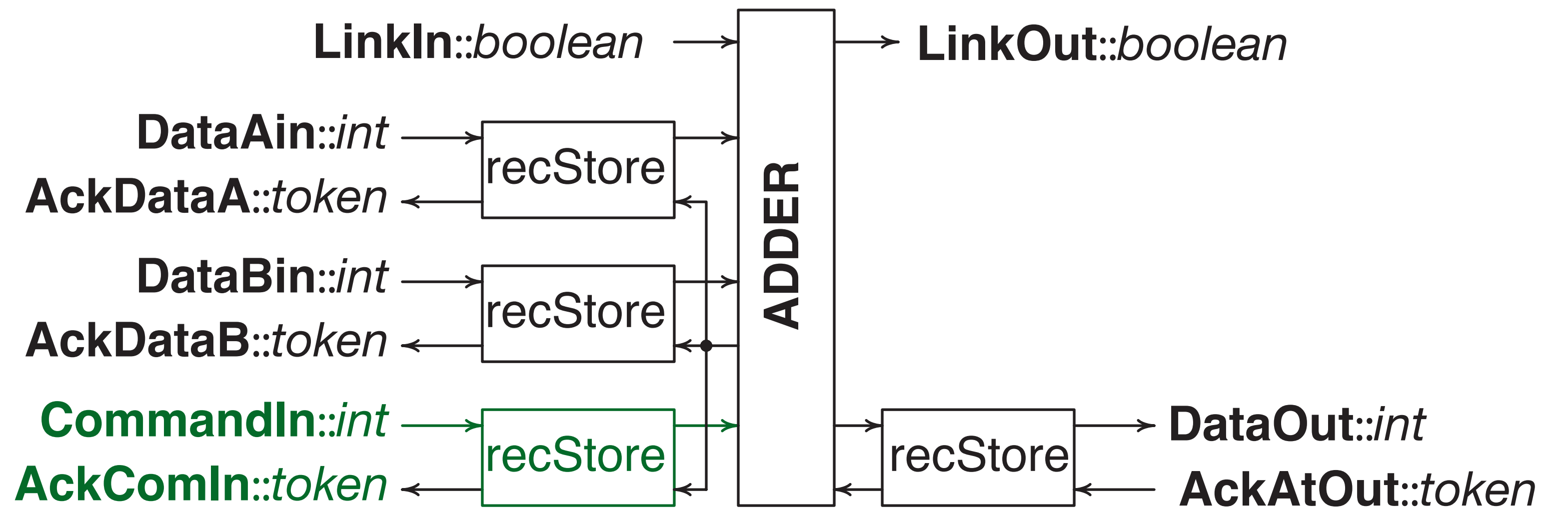


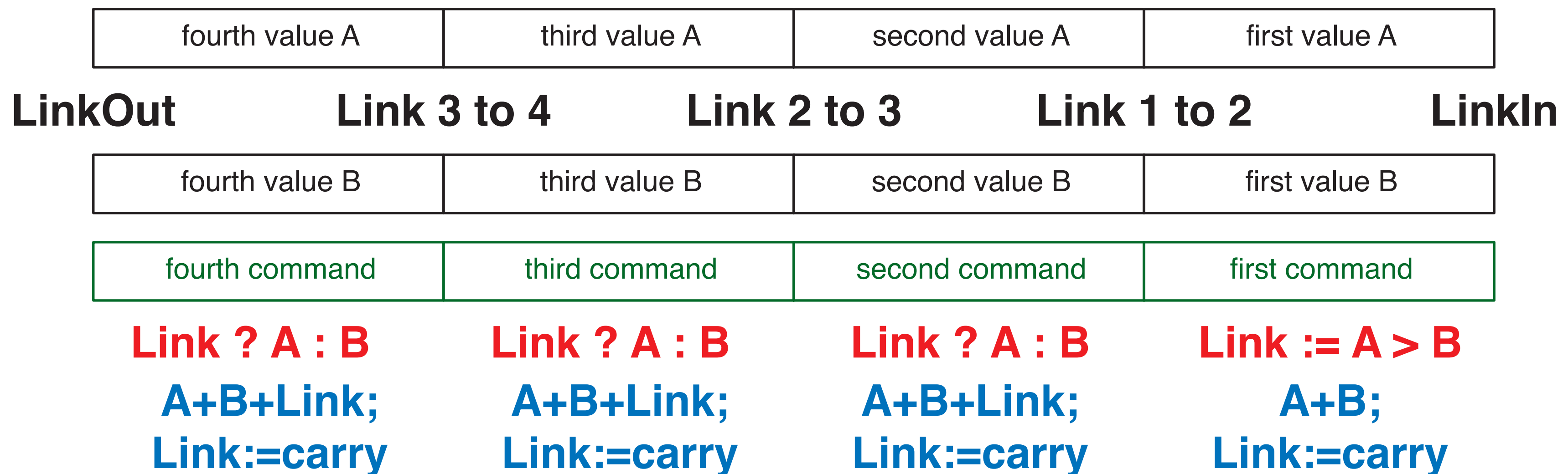




UCIES# 2006-is34
14 September 2006

Adder SHIP





Slide 29: Adder SHIP

End of Part 3

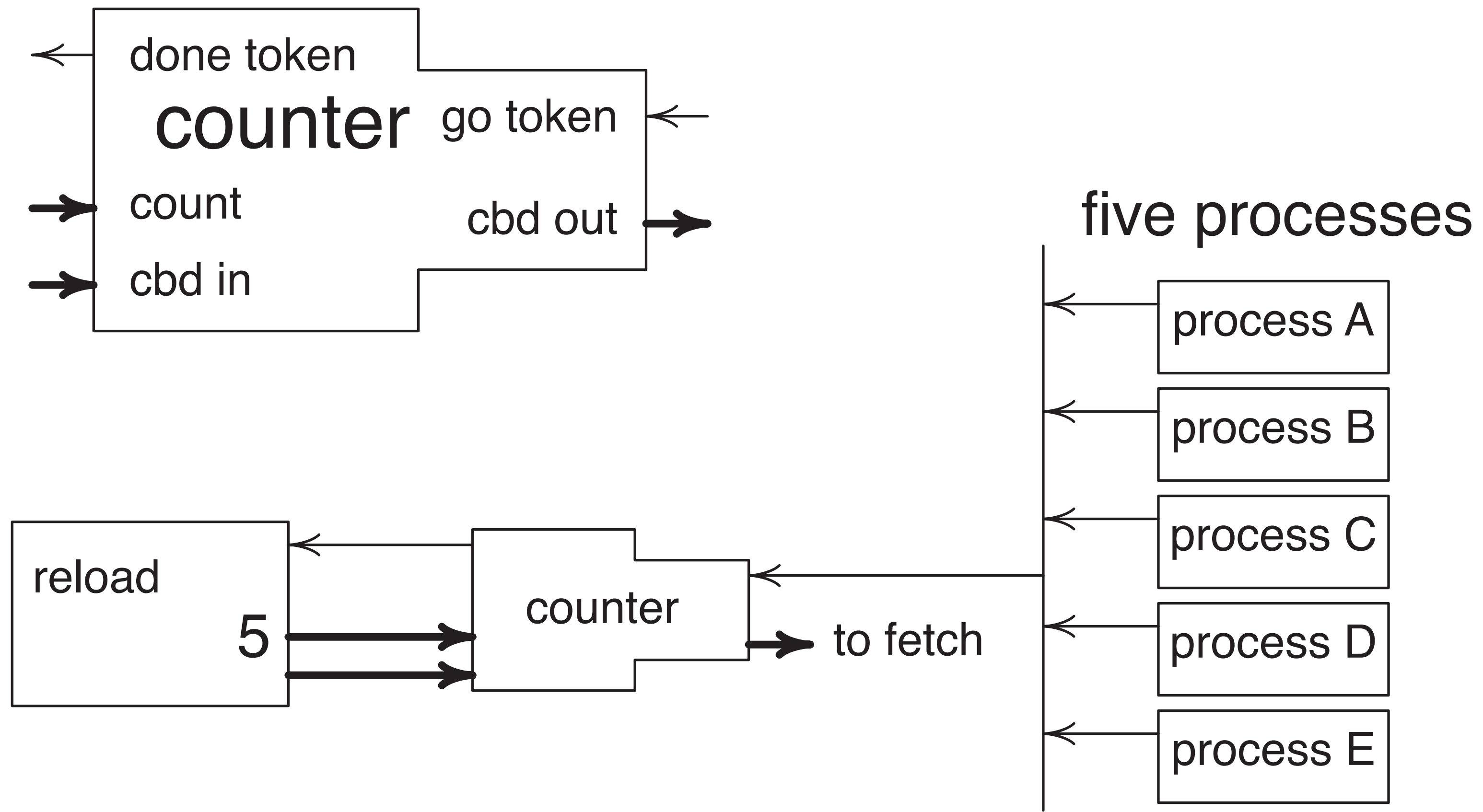
- SHIPs
- Asynchronous Switch Fabric
- Code Bags

Part 4 - Programming

- Pipelines are easy
- Close the loop via
  acknowledge token or
  via next code bag

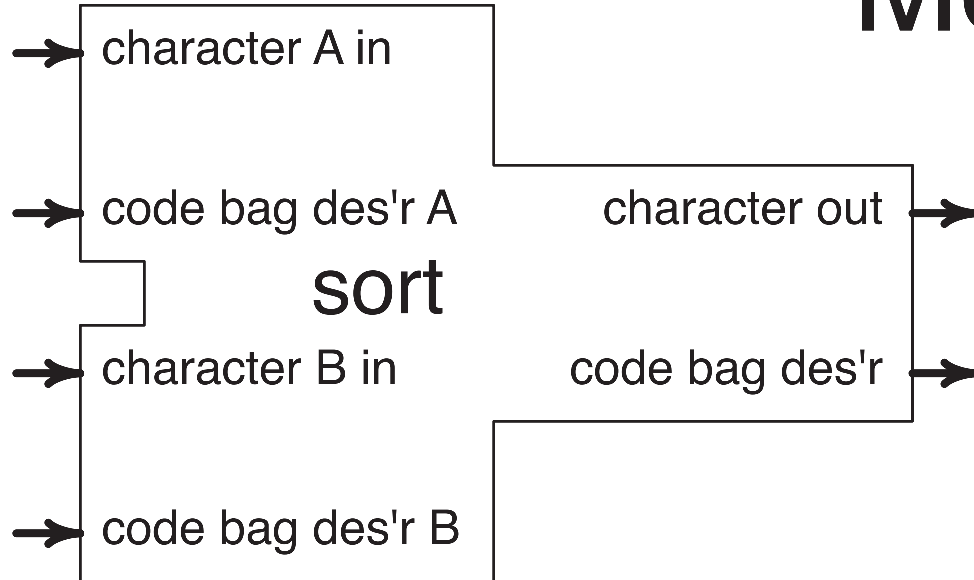
Token use example

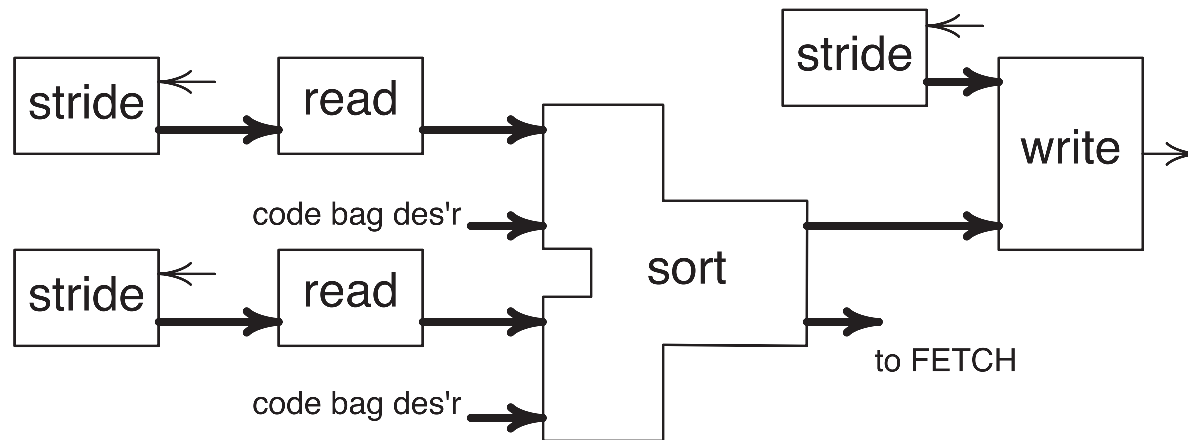
Finish five tasks



UCIES# 2006-is34
14 September 2006

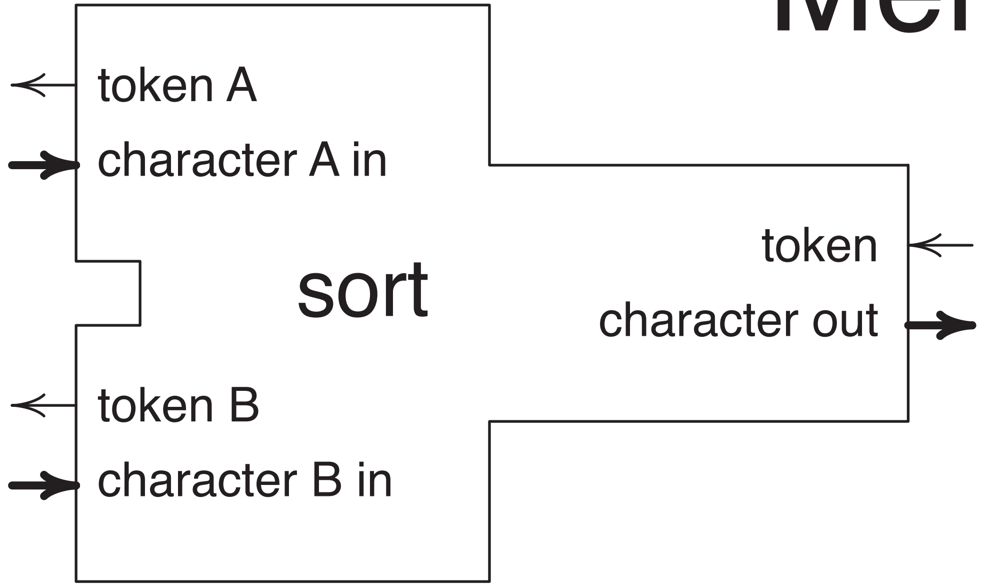
Merge sort

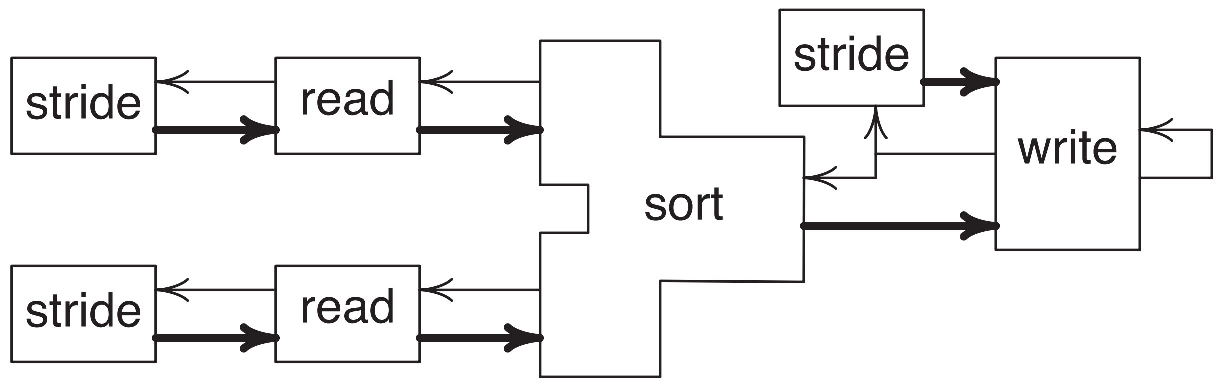




UCIES# 2006-is34
14 September 2006

Merge sort





Slide 33: Sort B

Programming lessons

Sequential machines

- Easy sequencing
- Painful concurrency

FLEET

- Basic concurrency
- Seek sequencing from
  pipelines, data flow, or tokens

Hardware issues

Master clear
Save and restore
Finding admirable SHIPs

Software issues

How to program a FLEET
Using records
Errors and Debugging
Avoiding Deadlock