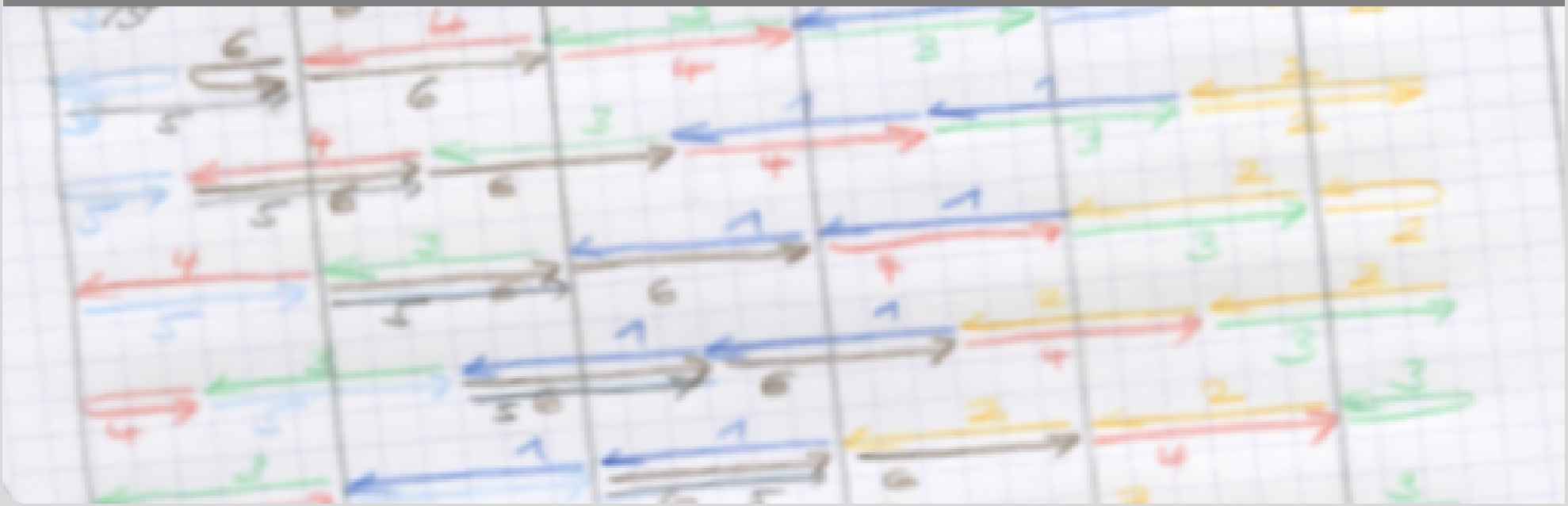
Karlsruhe Institute of Technology

Real-time sorting of binary numbers on one-dimensional CA

JAC 2010

Thomas Worsch and Hidenosuke Nishio | December 17

KARLSRUHE INSTITUTE OF TECHNOLOGY, INSTITUTE FOR CRYPTOGRAPHY AND SECURITY



Outline

1. Prerequisites
2. First algorithm
3. Second algorithm

Cellular Automata

- ⟨ insert your favorite definition here ⟩
  - one-dimensional
  - neighborhood radius 1
- synchronization of k cells
  - one general at one end: e. g. in time $2k - 2$
  - two generals, one at each end: e. g. in time k

Two theorems

Theorem 1

Sorting is an important problem.

Two theorems

Theorem 1

Sorting is an important problem.

Proof

several hundred publications on DBLP and CCSB
in 2006-2010 containing the word “sorting” in the title

Two theorems

Theorem 1

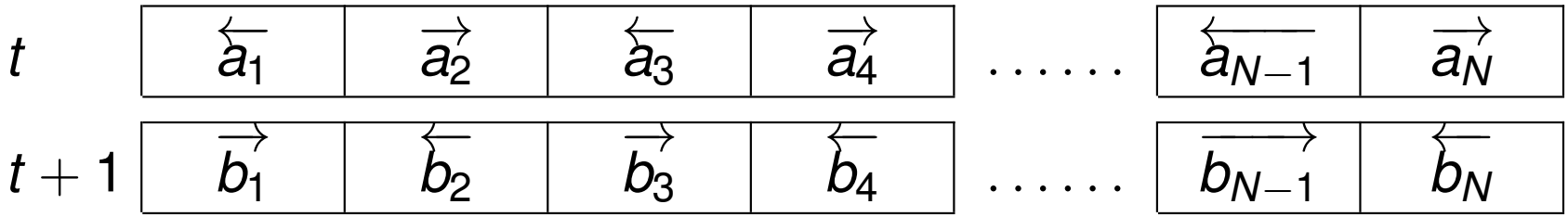
Sorting is an important problem.

Theorem 2

Odd-even Transposition Sort sorts n items in n steps.

Odd-even Transposition Sort

- parallel sorting algorithm
- one data item per processor
- processors numbered sequentially
  - each processor knows the parity of its number
- alternate looking to the left and looking to the right
  - Margolus “neighborhood”



$$\text{where } b_i = \begin{cases} \min(a_i, a_{i+1}) & \text{if } a_i \text{ points to the right} \\ \max(a_{i-1}, a_i) & \text{if } a_i \text{ points to the left} \end{cases}$$

OETS: an example

5	5	6	6	4	4	3	3	1	1	2	2
5	5	6	4	6	3	4	1	3	1	2	2
5	5	4	6	3	6	1	4	1	3	2	2
5	4	5	3	6	1	6	1	4	2	3	2
4	5	3	5	1	6	1	6	2	4	2	3
4	3	5	1	5	1	6	2	6	2	4	3
3	4	1	5	1	5	2	6	2	6	3	4
3	1	4	1	5	2	5	2	6	3	6	4
1	3	1	4	2	5	2	5	3	6	4	6
1	1	3	2	4	2	5	3	5	4	6	6
1	1	2	3	2	4	3	5	4	5	6	6
1	1	2	2	3	3	4	4	5	5	6	6

The problem

find a CA, independent of n and k, with the following properties

- inputs: n numbers (surrounded by quiescent states)
  - represented with k bits each
  - one bit per cell
  - most significant and least significant bits marked:
    input alphabet: $A = \{0, 1, \langle 0, \langle 1, 0|, 1|\}$.
  - e.g.: $\langle 0101|\langle 0110|\langle 0100|\langle 0011|\langle 0010|\langle 0001|$
- outputs: permuted and sorted sequence of numbers
  - e.g.: $\langle 0001|\langle 0010|\langle 0011|\langle 0100|\langle 0101|\langle 0110|$

Lower bound

Lemma

Any sorting CA needs at least $nk - 1$ steps for n k-bit numbers.

Proof

Consider

$$\begin{aligned} \text{input} \quad & w = \langle 01^{k-2}1|\langle 10^{k-2}0| \cdots \langle 10^{k-2}0|\langle 10^{k-2}0| \\ \text{output} \quad & w = \langle 01^{k-2}1|\langle 10^{k-2}0| \cdots \langle 10^{k-2}0|\langle 10^{k-2}0| \end{aligned}$$

Lower bound

Lemma

Any sorting CA needs at least $nk - 1$ steps for n k-bit numbers.

Proof

Consider

$$\begin{array}{ll} \text{input} & w = \langle 01^{k-2}1|\langle 10^{k-2}0| \cdots \langle 10^{k-2}0|\langle 10^{k-2}0| \\ \text{output} & w = \langle 01^{k-2}1|\langle 10^{k-2}0| \cdots \langle 10^{k-2}0|\langle 10^{k-2}0| \\ \text{input} & w = \langle 11^{k-2}1|\langle 10^{k-2}0| \cdots \langle 10^{k-2}0|\langle 10^{k-2}0| \end{array}$$

Lower bound

Lemma

Any sorting CA needs at least $nk - 1$ steps for n k-bit numbers.

Proof

Consider

$$\begin{array}{ll}
\text{input} & w = \langle 01^{k-2}1|\langle 10^{k-2}0|\cdots\langle 10^{k-2}0|\langle 10^{k-2}0| \\
\text{output} & w = \langle 01^{k-2}1|\langle 10^{k-2}0|\cdots\langle 10^{k-2}0|\langle 10^{k-2}0| \\
\text{input} & w = \langle 11^{k-2}1|\langle 10^{k-2}0|\cdots\langle 10^{k-2}0|\langle 10^{k-2}0| \\
\text{output} & w = \langle 10^{k-2}0|\langle 10^{k-2}0|\cdots\langle 10^{k-2}0|\langle 11^{k-2}1|
\end{array}$$

Upper bound

Result (Nishio, 1975)

CA can solve the sorting problem in time $3nk$.

Upper bound

Result (Nishio, 1975)

CA can solve the sorting problem in time $3nk$.

Speedups?

- standard speedup techniques don't help
  - unless you change the output convention
  - an analogue to increasing the neighborhood radius to 2
  - no no

Upper bound

Result (Nishio, 1975)

CA can solve the sorting problem in time $3nk$.

Speedups?

- standard speedup techniques don't help
  - unless you change the output convention
  - an analogue to increasing the neighborhood radius to 2
  - no no
- generalizing Nishio's construction seems to result in time $(1 + \varepsilon)nk$
  - at the expense of larger and larger set of states
  - no no

Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them

⟨1	1	0	1	0	0|
⟨1	1	1	0	1	0|
=					

Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them

⟨1	1	0	1	0	0|
⟨1	1	1	0	1	0|
	=				

Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them

⟨1	1	0	1	0	0|
⟨1	1	1	0	1	0|
		∧			

Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them

⟨1	1	0	1	0	0|
⟨1	1	1	0	1	0|
			∧		

Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them

⟨1	1	0	1	0	0|
⟨1	1	1	0	1	0|
				∧	

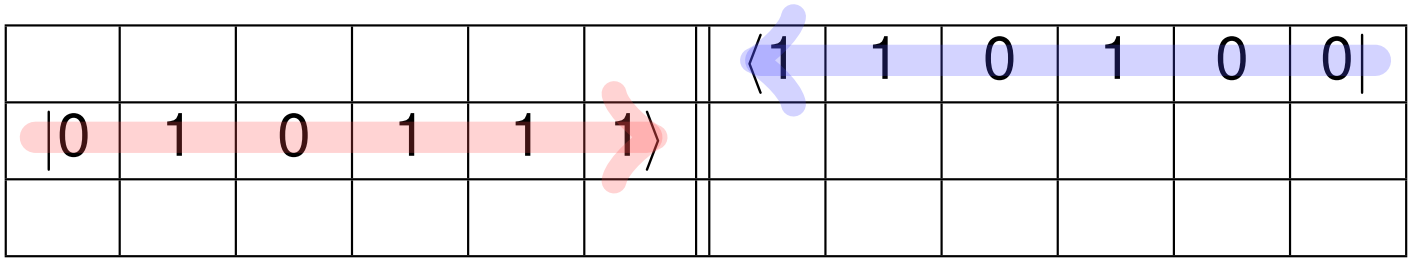
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them

⟨1	1	0	1	0	0|
⟨1	1	1	0	1	0|
					∧

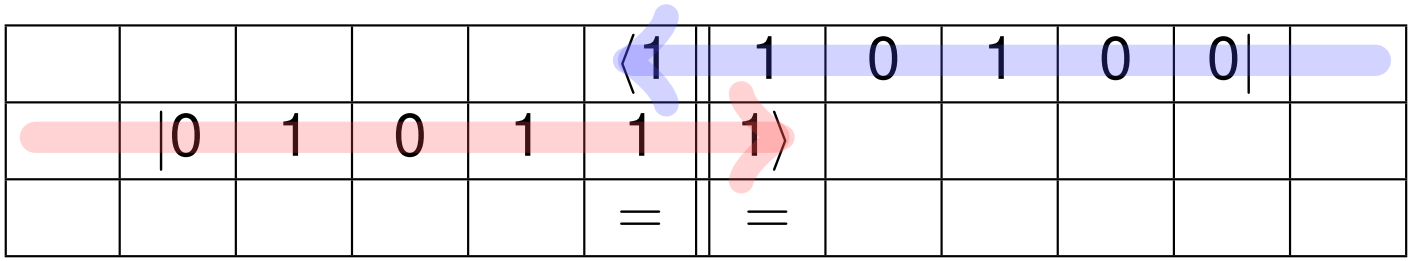
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



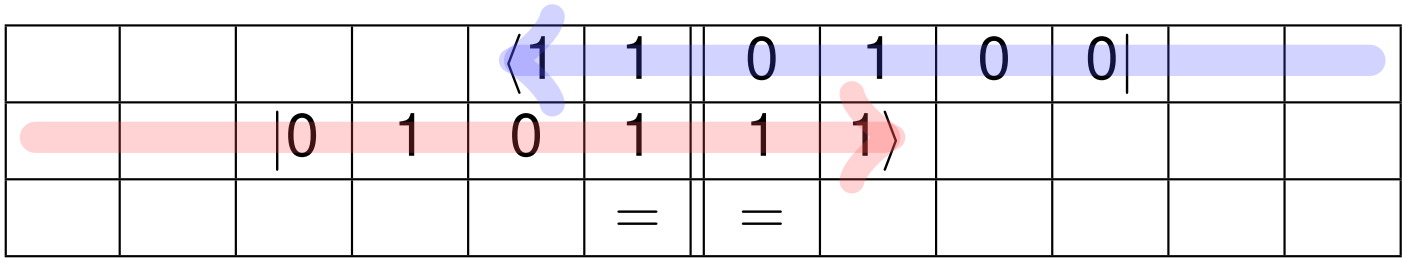
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



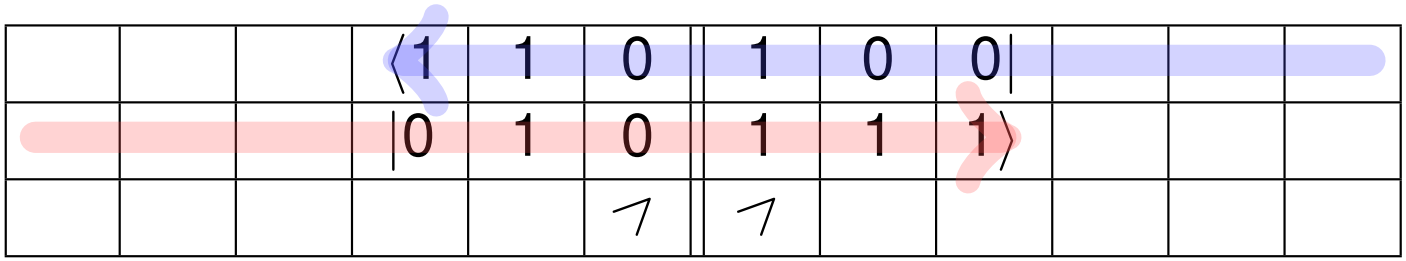
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



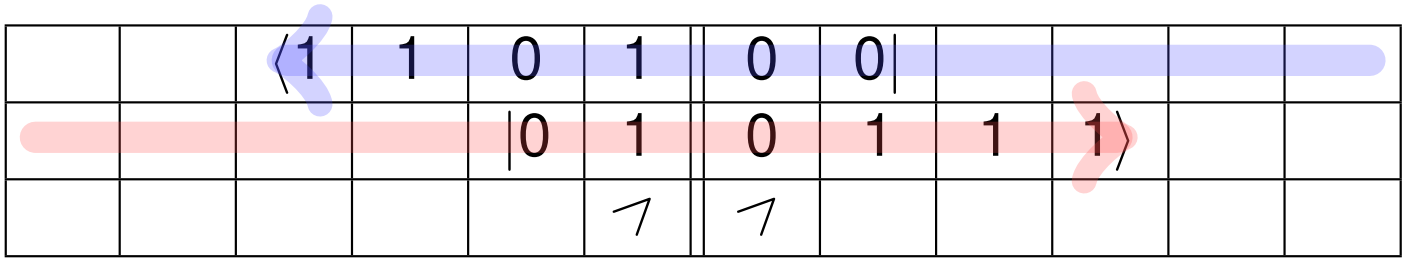
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



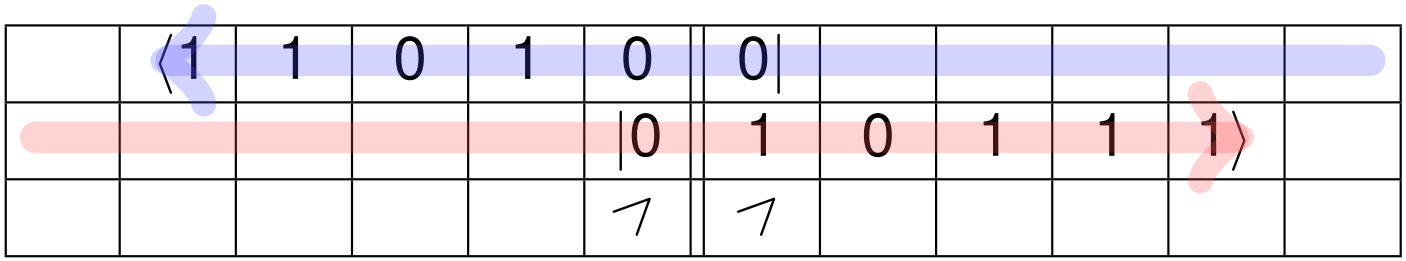
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



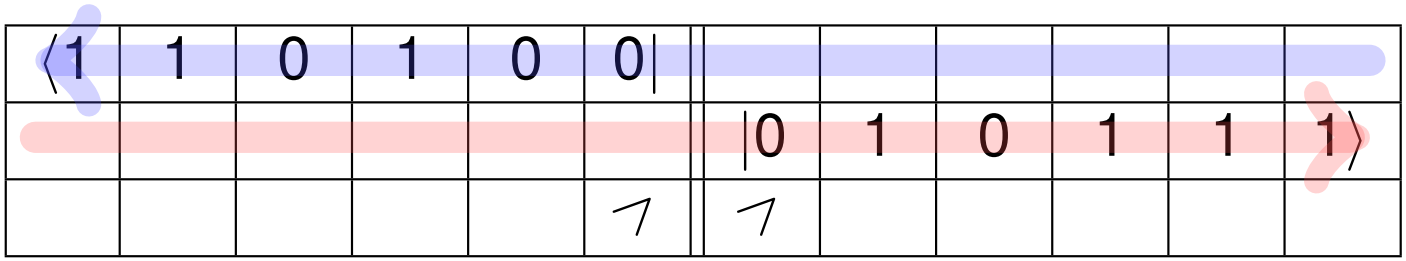
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



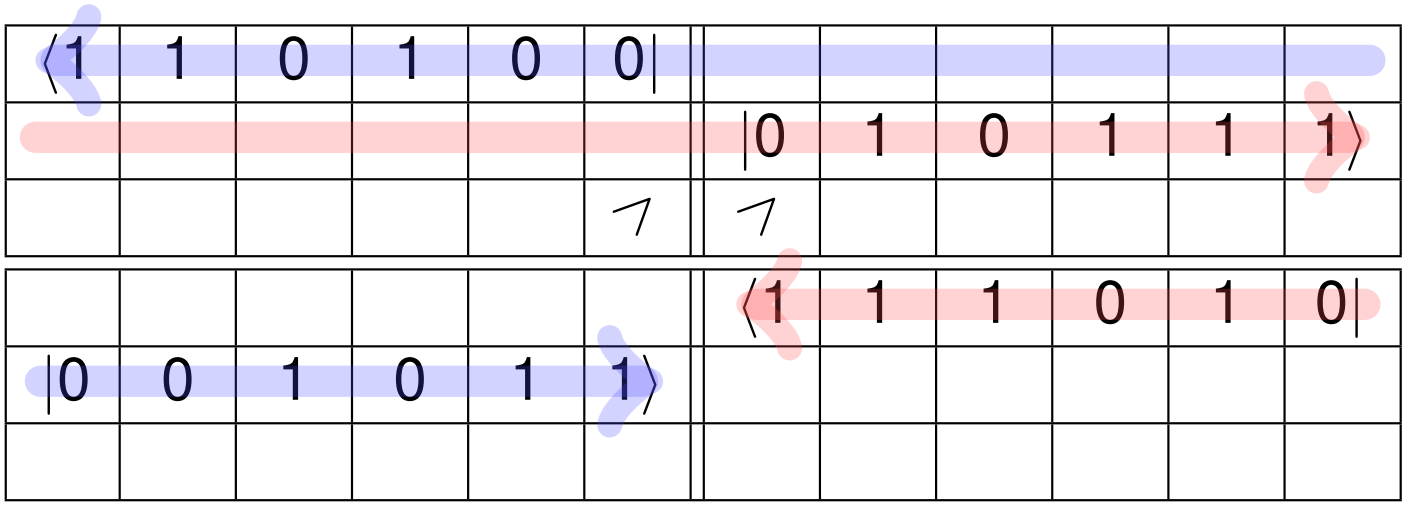
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



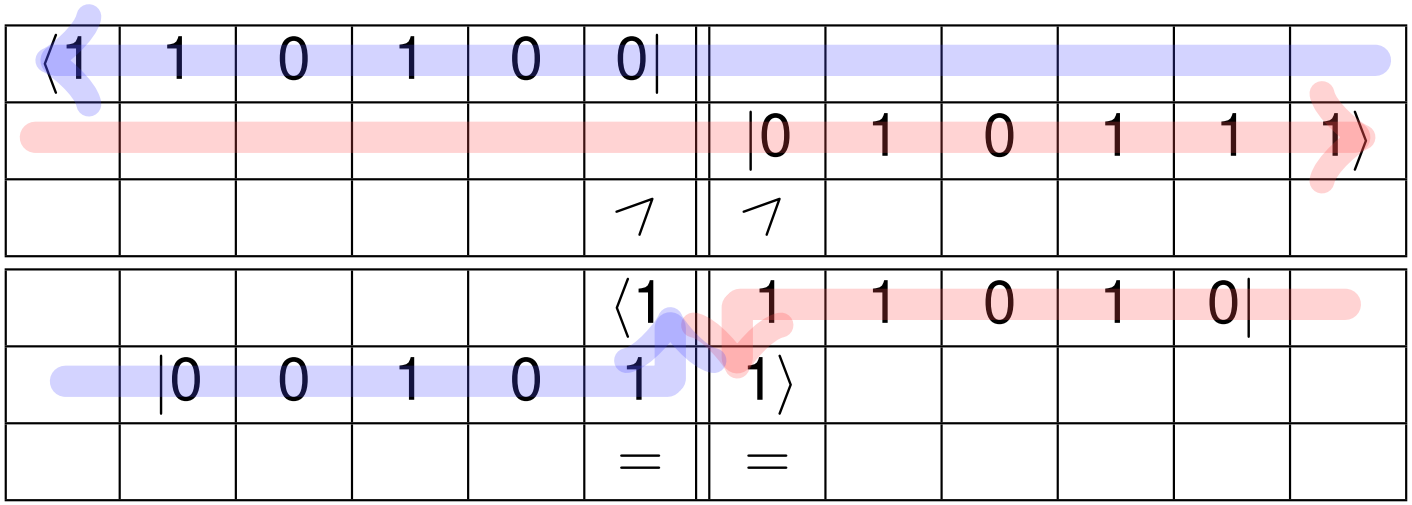
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



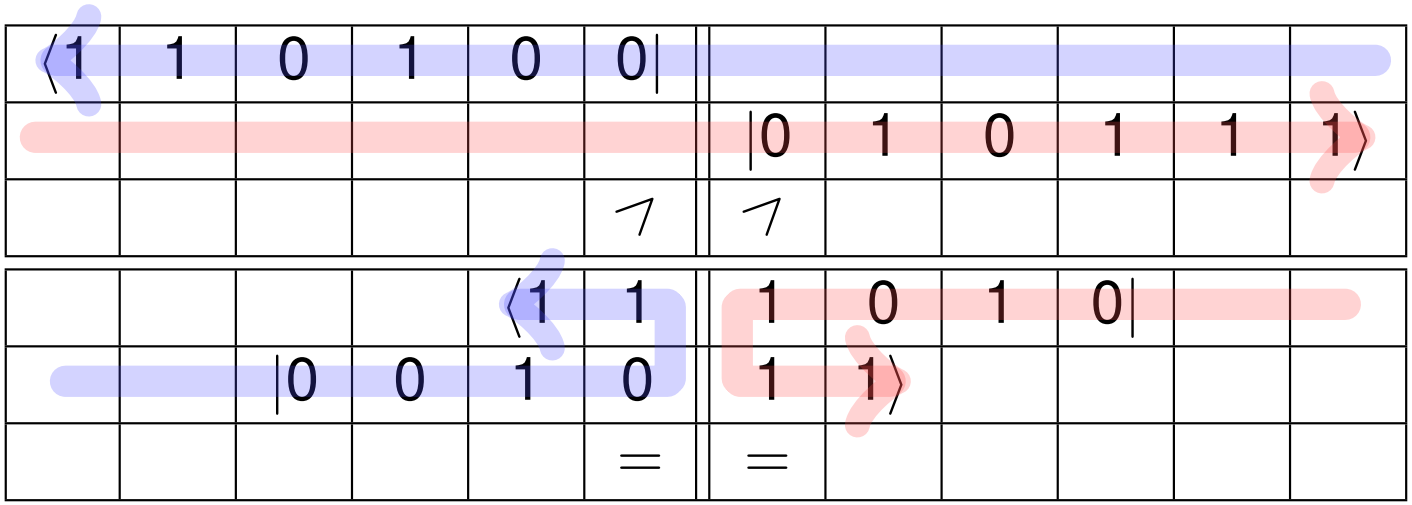
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



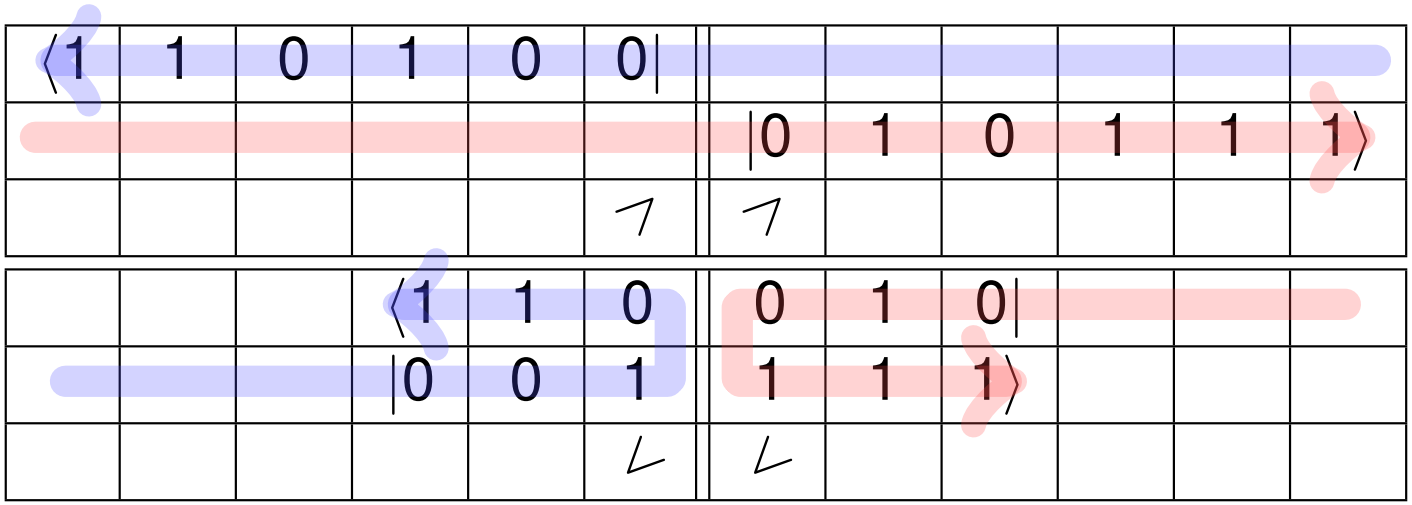
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



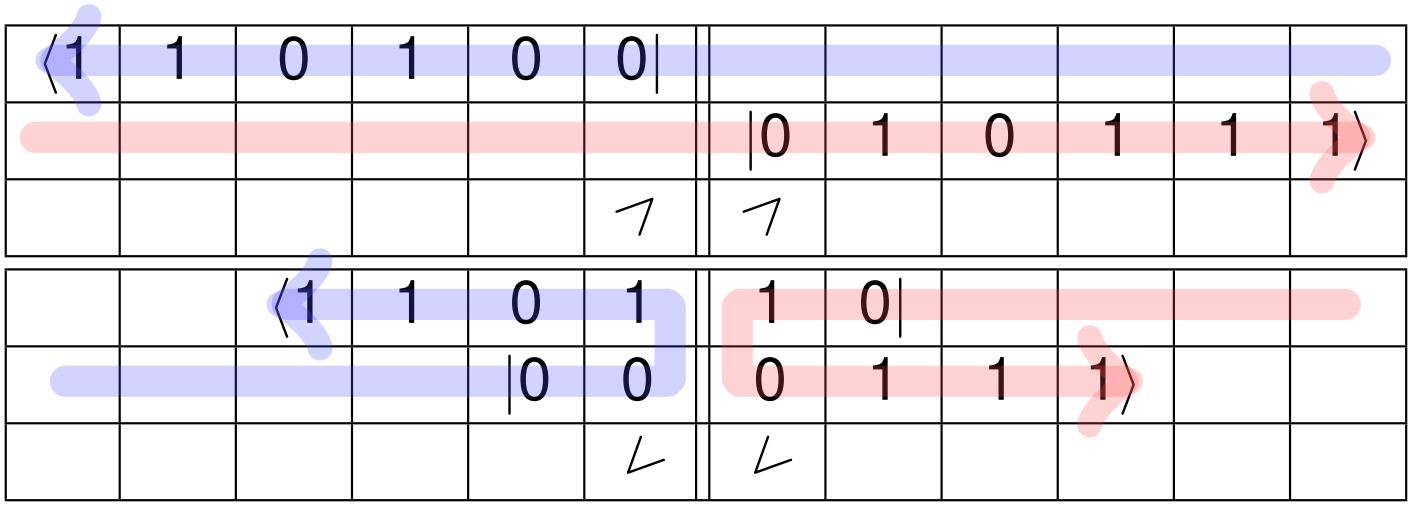
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



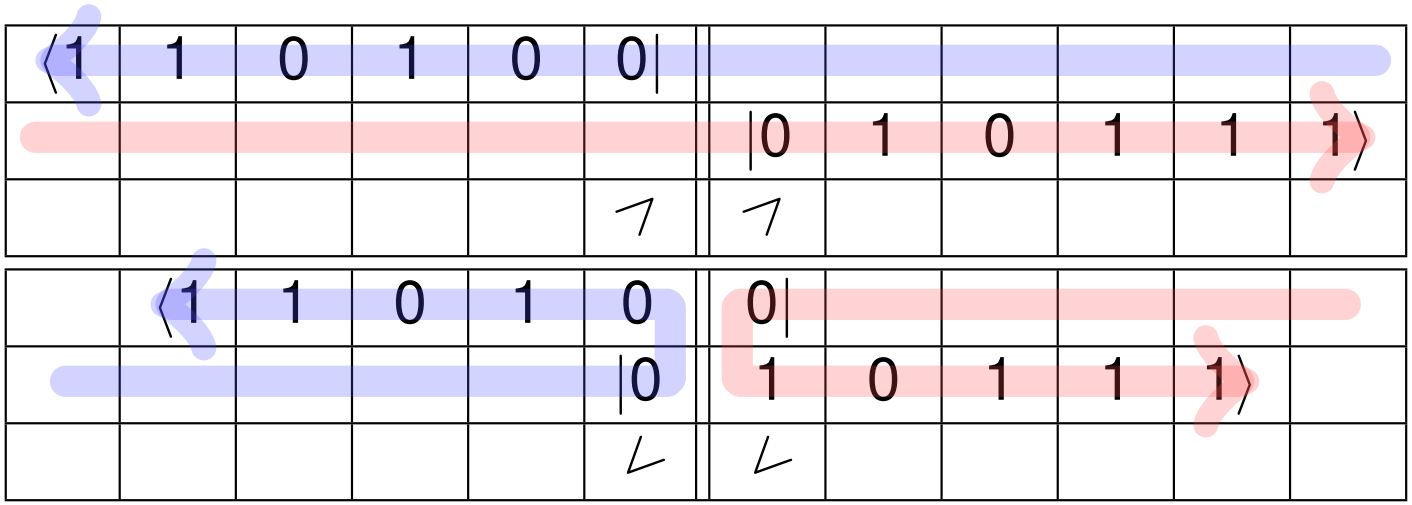
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



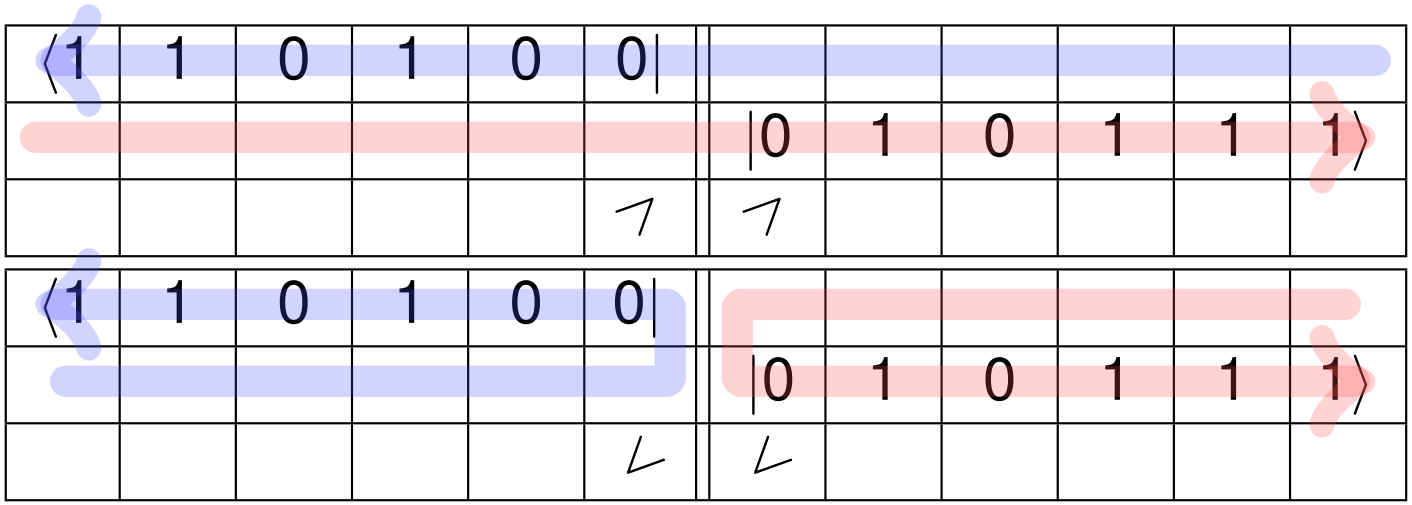
Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



Comparison of two k-bit numbers

- numbers stored in parallel; move a signal compare them
- move numbers to comparison location(s)
  shift smaller number to the left, larger to the right



Outline

1 Prerequisites

2 First algorithm

3 Second algorithm

First algorithm: idea

- transform input

$\overleftarrow{w_1}$	$\overleftarrow{w_2}$	$\overleftarrow{w_3}$	$\overleftarrow{w_4}$	.....	$\overleftarrow{w_{n-1}}$	$\overleftarrow{w_n}$

- into

$\overleftarrow{w_1}$	$\overleftarrow{w_2}$	$\overleftarrow{w_3}$	$\overleftarrow{w_4}$	.....	$\overleftarrow{w_{n-1}}$	$\overleftarrow{w_n}$
$\overrightarrow{w_1}$	$\overrightarrow{w_2}$	$\overrightarrow{w_3}$	$\overrightarrow{w_4}$	.....	$\overrightarrow{w_{n-1}}$	$\overrightarrow{w_n}$

- sort the $N = 2n$ numbers using OETS

$\overleftarrow{w_{\sigma(1)}}$	$\overleftarrow{w_{\sigma(2)}}$	$\overleftarrow{w_{\sigma(3)}}$	$\overleftarrow{w_{\sigma(4)}}$	.....	$\overleftarrow{w_{\sigma(n-1)}}$	$\overleftarrow{w_{\sigma(n)}}$
$\overrightarrow{w_{\sigma(1)}}$	$\overrightarrow{w_{\sigma(2)}}$	$\overrightarrow{w_{\sigma(3)}}$	$\overrightarrow{w_{\sigma(4)}}$	.....	$\overrightarrow{w_{\sigma(n-1)}}$	$\overrightarrow{w_{\sigma(n)}}$

- in the end keep the upper row

$\overleftarrow{w_{\sigma(1)}}$	$\overleftarrow{w_{\sigma(2)}}$	$\overleftarrow{w_{\sigma(3)}}$	$\overleftarrow{w_{\sigma(4)}}$	.....	$\overleftarrow{w_{\sigma(n-1)}}$	$\overleftarrow{w_{\sigma(n)}}$

First algorithm: implementation

$\overleftarrow{A_1}$	$\overleftarrow{a_1}$	$\overleftarrow{A_2}$	$\overleftarrow{a_2}$	$\overleftarrow{A_3}$	$\overleftarrow{a_3}$	$\overleftarrow{A_4}$	$\overleftarrow{a_4}$	$\overleftarrow{A_5}$	$\overleftarrow{a_5}$
$\overrightarrow{a_1}$	$\overrightarrow{A_1}$	$\overrightarrow{a_2}$	$\overrightarrow{A_2}$	$\overrightarrow{a_3}$	$\overrightarrow{A_3}$	$\overrightarrow{a_4}$	$\overrightarrow{A_4}$	$\overrightarrow{a_5}$	$\overrightarrow{A_5}$

First algorithm: implementation

$\overleftarrow{A_1}$		$\overleftarrow{A_2}$		$\overleftarrow{A_3}$		$\overleftarrow{A_4}$		$\overleftarrow{A_5}$	
	$\overrightarrow{A_1}$		$\overrightarrow{A_2}$		$\overrightarrow{A_3}$		$\overrightarrow{A_4}$		$\overrightarrow{A_5}$

First algorithm: implementation

$\overleftarrow{A_1}$		$\overleftarrow{A_2}$		$\overleftarrow{A_3}$		$\overleftarrow{A_4}$		$\overleftarrow{A_5}$	
	$\overrightarrow{A_1}$		$\overrightarrow{A_2}$		$\overrightarrow{A_3}$		$\overrightarrow{A_4}$		$\overrightarrow{A_5}$

after $k/2$ steps:

	$\overleftarrow{B_2}$		$\overleftarrow{B_4}$		$\overleftarrow{B_6}$		$\overleftarrow{B_8}$		$\overleftarrow{B_{10}}$
$\overrightarrow{B_1}$		$\overrightarrow{B_3}$		$\overrightarrow{B_5}$		$\overrightarrow{B_7}$		$\overrightarrow{B_9}$	

First algorithm: implementation

$\overleftarrow{A_1}$	$\overleftarrow{a_1}$	$\overleftarrow{A_2}$	$\overleftarrow{a_2}$	$\overleftarrow{A_3}$	$\overleftarrow{a_3}$	$\overleftarrow{A_4}$	$\overleftarrow{a_4}$	$\overleftarrow{A_5}$	$\overleftarrow{a_5}$
$\overrightarrow{a_1}$	$\overrightarrow{A_1}$	$\overrightarrow{a_2}$	$\overrightarrow{A_2}$	$\overrightarrow{a_3}$	$\overrightarrow{A_3}$	$\overrightarrow{a_4}$	$\overrightarrow{A_4}$	$\overrightarrow{a_5}$	$\overrightarrow{A_5}$

after $k/2$ steps:

$\overleftarrow{a_1}$	$\overleftarrow{B_2}$	$\overleftarrow{a_2}$	$\overleftarrow{B_4}$	$\overrightarrow{b_3}$	$\overleftarrow{B_6}$	$\overrightarrow{b_4}$	$\overleftarrow{B_8}$	$\overrightarrow{b_5}$	$\overleftarrow{B_{10}}$
$\overrightarrow{B_1}$	$\overrightarrow{a_1}$	$\overrightarrow{B_3}$	$\overleftarrow{b_2}$	$\overrightarrow{B_5}$	$\overleftarrow{b_3}$	$\overrightarrow{B_7}$	$\overleftarrow{b_4}$	$\overrightarrow{B_9}$	$\overleftarrow{b_5}$

First algorithm: implementation

$\overleftarrow{A_1}$		$\overleftarrow{A_2}$		$\overleftarrow{A_3}$		$\overleftarrow{A_4}$		$\overleftarrow{A_5}$	
	$\overrightarrow{A_1}$		$\overrightarrow{A_2}$		$\overrightarrow{A_3}$		$\overrightarrow{A_4}$		$\overrightarrow{A_5}$

after $k/2$ steps:

	$\overleftarrow{B_2}$		$\overleftarrow{B_4}$		$\overleftarrow{B_6}$		$\overleftarrow{B_8}$		$\overleftarrow{B_{10}}$
$\overrightarrow{B_1}$		$\overrightarrow{B_3}$		$\overrightarrow{B_5}$		$\overrightarrow{B_7}$		$\overrightarrow{B_9}$	

First algorithm: implementation

$\overleftarrow{A_1}$		$\overleftarrow{A_2}$		$\overleftarrow{A_3}$		$\overleftarrow{A_4}$		$\overleftarrow{A_5}$	
	$\overrightarrow{A_1}$		$\overrightarrow{A_2}$		$\overrightarrow{A_3}$		$\overrightarrow{A_4}$		$\overrightarrow{A_5}$

after $k/2$ steps:

	$\overleftarrow{B_2}$		$\overleftarrow{B_4}$		$\overleftarrow{B_6}$		$\overleftarrow{B_8}$		$\overleftarrow{B_{10}}$
$\overrightarrow{B_1}$		$\overrightarrow{B_3}$		$\overrightarrow{B_5}$		$\overrightarrow{B_7}$		$\overrightarrow{B_9}$	

after k steps:

$\overleftarrow{C_1}$		$\overleftarrow{C_3}$		$\overleftarrow{C_5}$		$\overleftarrow{C_7}$		$\overleftarrow{C_9}$	
	$\overrightarrow{C_2}$		$\overrightarrow{C_4}$		$\overrightarrow{C_6}$		$\overrightarrow{C_8}$		$\overrightarrow{C_{10}}$

First algorithm: correctness

proof by induction shows:

- the positions of upper halves in sub-blocks
- coincides with the positions of the numbers in OETS

First algorithm: time complexity

- k steps for generating the mirrored numbers
- $2n$ phases with comparisons
  - k steps for each
  - BUT comparison of lower halves can be overlapped with comparison of upper halves of next phase
  - total time: nk
  - comparison of lower halves of last phase can be dropped (no-op)
- total: $k + nk$

Outline

1 Prerequisites

2 First algorithm

3 Second algorithm

Goal

- running time nk
- modify the first algorithm to save k steps
  - $k/2$ at the beginning
  - $k/2$ at the end
- resulting algorithm will not work

Goal

- running time nk
- modify the first algorithm to save k steps
  - $k/2$ at the beginning
  - $k/2$ at the end
- resulting algorithm will not work
  - correct results in the $n-1$ left blocks
  - something wrong at the rightmost block
    - fix by additional algorithm
    - running in parallel to the modified one

Observation

- How would you compute the mirrored numbers? like this:
  - shift bits to the left
  - reflect at the block boundary
- also shift the bits to the next block
  - can start comparisons after $k/2$ steps

⟨1	1	1	1	1	1|	⟨0	0	0	0	0	0|

Observation

- How would you compute the mirrored numbers? like this:
  - shift bits to the left
  - reflect at the block boundary
- also shift the bits to the next block
  - can start comparisons after $k/2$ steps

1	1	1	1	1|	⟨0	0	0	0	0	0|	
1⟩						0⟩					

Observation

- How would you compute the mirrored numbers? like this:
  - shift bits to the left
  - reflect at the block boundary
- also shift the bits to the next block
  - can start comparisons after $k/2$ steps

1	1	1	1|	⟨0	0	0	0	0	0|		
1	1⟩					0	0⟩				

Observation

- How would you compute the mirrored numbers? like this:
  - shift bits to the left
  - reflect at the block boundary
- also shift the bits to the next block
  - can start comparisons after $k/2$ steps

1	1	1|	⟨0	0	0	0	0	0|			
1	1	1⟩				0	0	0⟩			

Observation

- How would you compute the mirrored numbers? like this:
  - shift bits to the left
  - reflect at the block boundary
- also shift the bits to the next block
  - can start comparisons after $k/2$ steps

1	1	1|	⟨0	0	0	0	0	0|			
1	1	1⟩				0	0	0⟩			

Problem 1

- we need two copies of *each* input number

Problem 1

- we need two copies of *each* input number
  - produce two mirror images of the leftmost one

Problem 1

- we need two copies of *each* input number
  - produce two mirror images of the leftmost one
- sometimes three numbers have to be compared
  - send the smallest to the left
  - the other two to the right
  - happens only once in each sub-block

Problem 2

- the rightmost block is wrong at the end

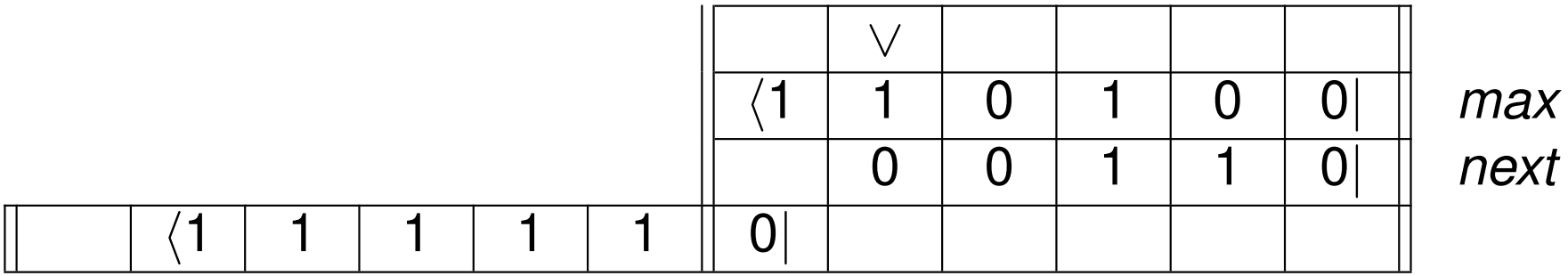
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left

						=						
						⟨1	1	0	1	0	0|	*max*
						⟨1	0	0	1	1	0|	*next*
⟨1	1	1	1	1	0|							

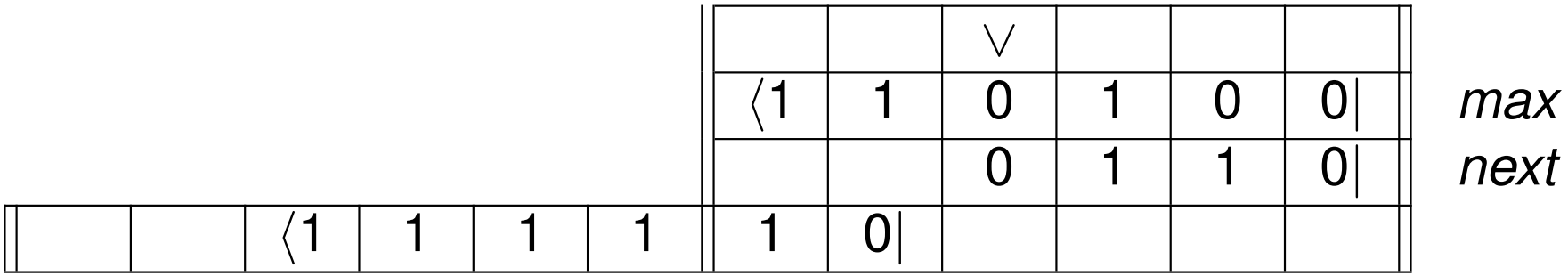
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



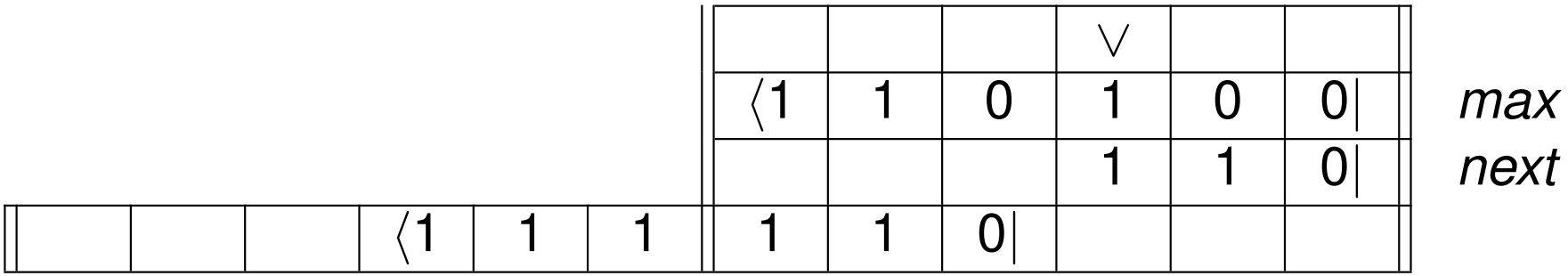
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



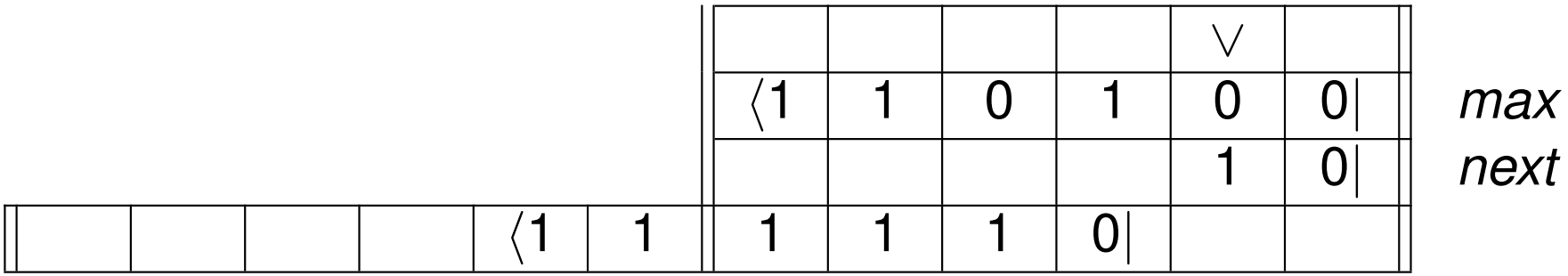
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



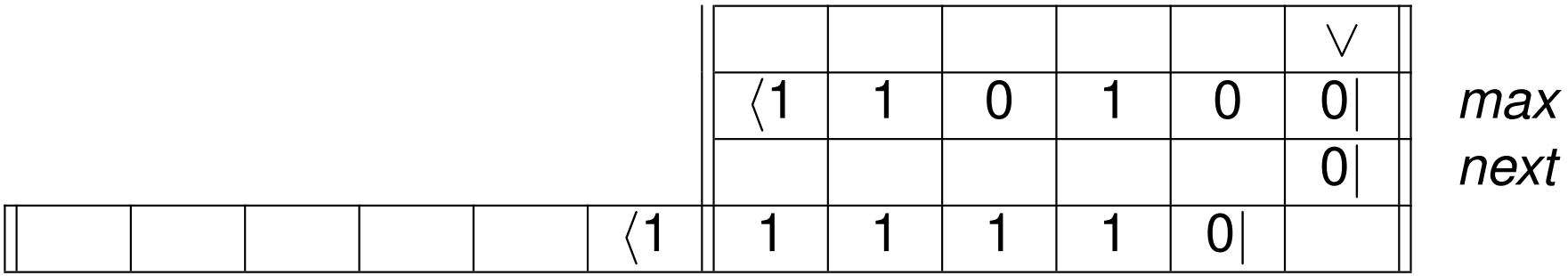
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



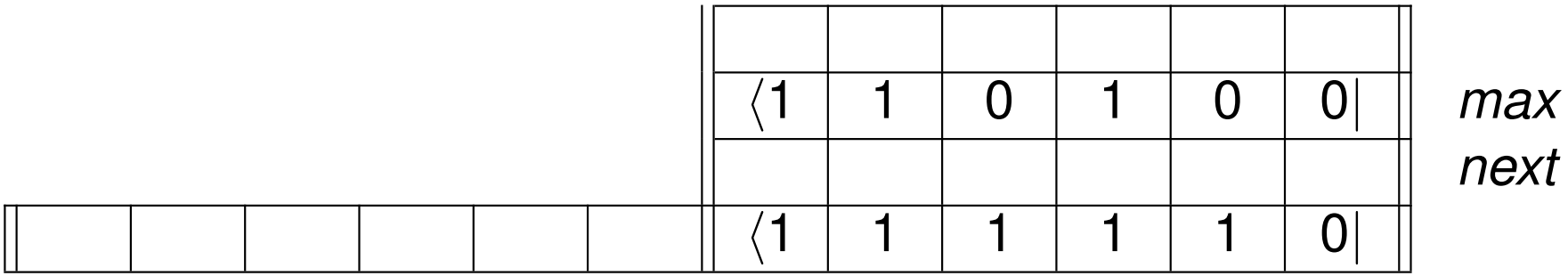
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



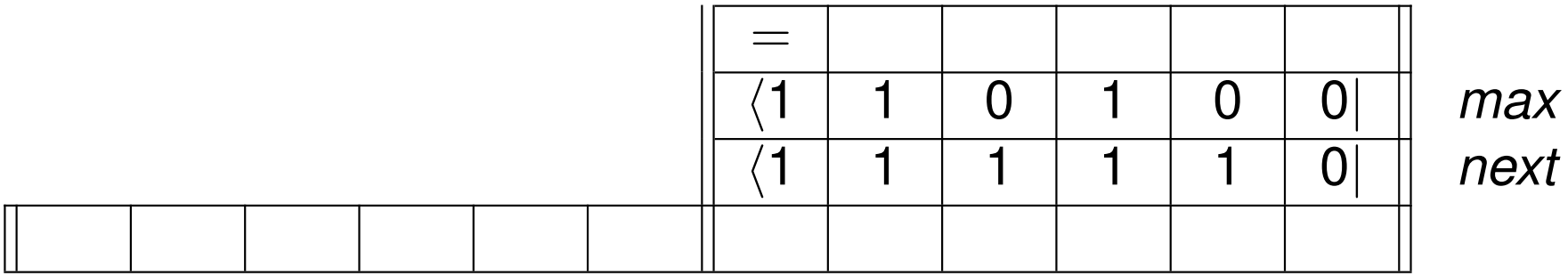
Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



Problem 2

- the rightmost block is wrong at the end
- fix by explicitly computing the largest number there
  - shift all numbers to the right
  - when one number has reached the last block
  - copy to a separate register,
    compare with current maximum and compute new one
  - while receiving the next number from the left



Summary and Outlook

- sorting time nk
- only 1 step more than a lower bound

- less ugly “uniform” algorithm?
- sorting in 2 dimensions?
- less states?

Summary and Outlook

- sorting time nk
- only 1 step more than a lower bound

- less ugly “uniform” algorithm?
- sorting in 2 dimensions?
- less states?

Thank you very much for your attention