Behavioral description using message-passing

Rajit Manohar

Asynchronous VLSI and Architecture (AVLSI) Group
Computer Systems Lab, Yale University

https://avlsi.csl.yale.edu/
https://csl.yale.edu/~rajit/
https://avlsi.csl.yale.edu/act

Yale

AVLSI

Communicating Hardware Processes

- Behavioral language
  - CHP = Communicating hardware processes
    - Based on Tony Hoare's CSP (Communicating Sequential Processes) language
  - Simplified programming language
- Assignment-based language, but…
  - No memory allocation
  - No memory references
- Basic data types: Booleans and unsigned integers

```
bool x;

int y;

int<8> z;
```

Basic language constructs

- Simple statements
  - **skip** *statement that does nothing*
  - **x := E** *assignment*
    - "Evaluate expression on the right-hand side, assign it to the variable on the left-hand side" : just like a standard software programming language

```
y := y + x*2
```

```
b := b | ~z
```

```
skip
```

```
b-
```

```
b+
```

- **S_1;S_2** *Execute statement 1 until it is finished. Then execute statement 2*

```
y := y + x*2;
w := y - 3;
y := 0; skip
```

Arrays

- In hardware, an array results in an address-calculation mechanism

```
x[i] := x[i] + 1
```

- Array access is of two kinds
  - Standard array, where array index requires run-time information, or
  - Array index is a constant

```
x[0] := x[0] + 1
```

- Only use standard arrays when absolutely necessary!

Selection statements

- **Deterministic selection** statement: generalized if-statement

```
[   x > 10 -> y := 3
[]  x < 10 -> y := 4
]
```

  - ❖ If some condition (guard) is true, execute corresponding statement
  - ❖ If all guards are false, then **wait**
  - ❖ If multiple guards are true, error!

- **Non-deterministic selection**

```
[|  x > 10 -> y := 3
[]  x < 10 -> y := 4
[]  x > 8 -> y := 7
|]
```

Note: in CHP, semi-colon is used as a separator (no trailing semi-colon)

Yale

AVLSI

Loops

- **Deterministic loop** statement

```
*[  x > 10 -> y := 3; x:= x - 1
 [] x < 10 -> y := 4; x := x + 1
 ]
```

  - ❖ If some condition (guard) is true, execute corresponding statement and go back to the loop start
  - ❖ If all guards are false, **then exit**
  - ❖ If multiple guards are true, error!

```
 i := 0; j := 0;
*[ i < 10 ->
     j := j + i;
     i := i + 1
 ]
```

Yale

AVLSI

More constructs

- **S_1,S_2** *Execute non-interfering statements 1 and 2 in parallel*

```
*[  x > 10 -> y := 3, x:= x - 1
 [] x < 10 -> y := 4, x := x + 1
 ]
```

- Common short-hand
  - ❖ Infinite loop

```
*[ true -> STMTS ]
```

```
*[ STMTS ]
```

  - ❖ Wait for some condition

```
[ COND -> skip ]
```

```
[ COND ]
```

Yale

AVLSI

Communication

- Hardware modules exchange information via **communication channels**
- **Channel**
  - single-sender, single-receiver
  - a matching send and receive behaves as a *distributed assignment*

```
X!e
```

Evaluate "e" and
send it on output port X

```
Y?x
```

Receive value from input port Y
and assign it to variable "x"

  - If these two ports are connected, then this has the net effect of

```
x := e
```

  - Channels are **blocking***: a send waits for matching receive, and a receive waits for a matching send.*

Yale

AVLSI

Overall hardware description

- A *parallel* collection of *communicating hardware processes*
  - By default, no shared state
- Connections between processes via *channels to exchange information*
  - (General shared variables possible; ignoring for this summer school!)
- For this summer school, syntax for connections, type declarations, etc. in the ACT language
  - There are other examples of CSP-like languages (e.g. Occam)

Example: buffer

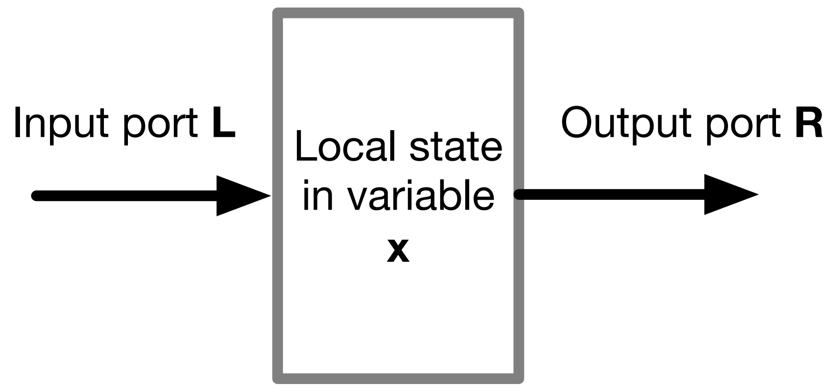
- One-place buffer, initially empty
  - Empty state
    - Only operation that is valid: read next input

      `L?x`

    - Final state: full
  - Full state
    - Only operation that is valid: send value on output

      `R!x`

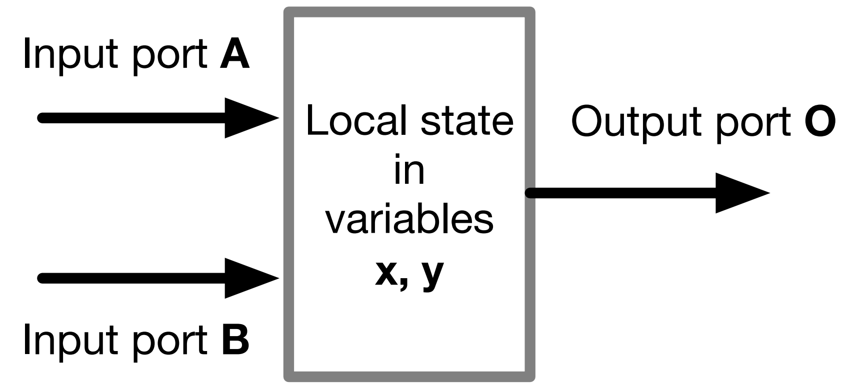
    - Final state: empty
  - Empty state to empty state: `L?x; R!x`



- Buffer repeats this forever: `*[ L?x; R!x ]`

Example: adder

- Read two operands
- Add them together
- Send the result on the output



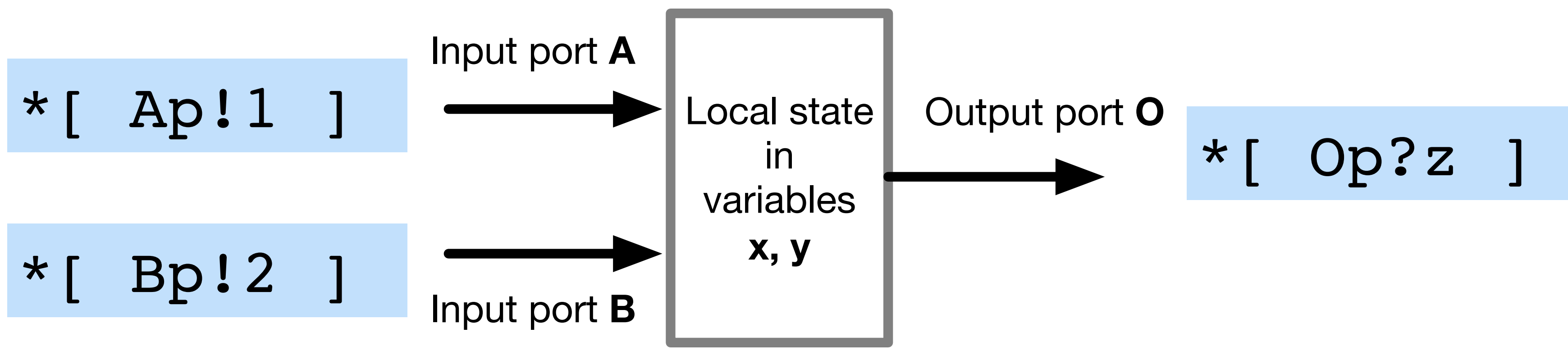

```
*[ A?x;B?y;
   O!(x+y)
 ]
```

- Parallel read

```
*[ A?x,B?y;
   O!(x+y)
 ]
```

Synchronization

```
*[ A?x;B?y; O!(x+y) ]
```



- Communication actions *synchronize* different parallel processes
  - ❖ Knowing where one process is in its local program can give you information about what other processes in the system are doing.

Non-deterministic constructs

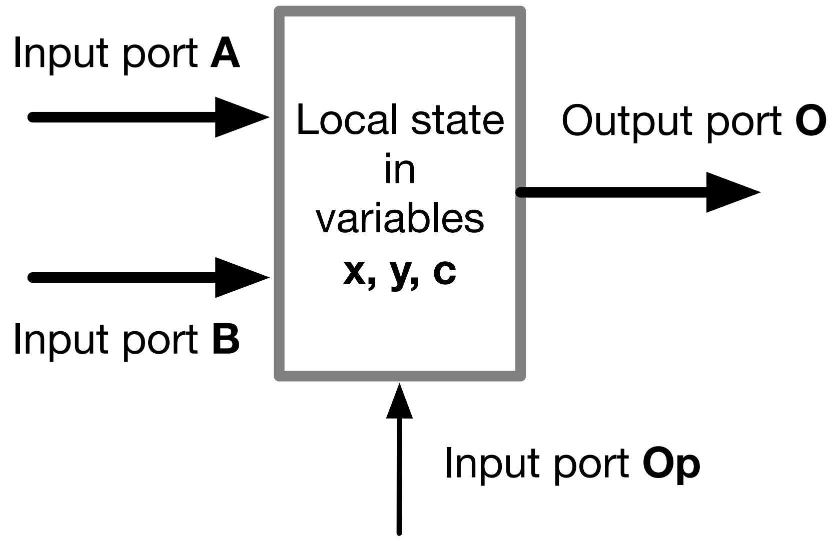
- Problem: two input ports **A** and **B** and one output **Z**
  - ❖ Receive the "next input" from either **A** or **B**
  - ❖ Send this value on the output **Z**

- We need some new syntax!
  - ❖ Probe: "is there a communication pending on this port?"

```
*[ [| #A -> A?x
   [] #B -> B?x
   |];
   Z!x
 ]
```

  - ❖ Use with care, and only when absolutely necessary

Example: add/subtract/multiply




```
*[ A?x, B?y, Op?c;
   [ c=0 -> O!(x+y)
   []c=1 -> O!(x-y)
   []c=2 -> O!(x*y)
   ]
 ]
```