Contents

1 Introduction 2
2 Usage 2
2.1 Project installation 2
3 Reference 2
3.1 circuit 2
3.1.1 circuit 2
3.2 util 3
3.2.1 lpad 3
3.2.2 normalize-coord 3
3.2.3 opposite-anchor 3
3.2.4 port-shape-vertices 4
3.2.5 resolve-rel-coord 4
3.2.6 rotate-anchor 5
3.2.7 colors 5
3.3 wire 6
3.3.1 breakout 6
3.3.2 intersection 10
3.3.3 stub 11
3.3.4 wire 12
3.3.5 wire-styles 16
3.4 element 18
3.4.1 elmt 18
3.4.2 alu 24
3.4.3 block 24
3.4.4 extender 27
3.4.5 multiplexer 28
3.4.6 group 30
3.5 gates 35
3.5.1 gate 35
3.5.2 gate-and 37
3.5.3 gate-nand 37
3.5.4 gate-buf 38
3.5.5 gate-not 39
3.5.6 gate-or 39
3.5.7 gate-nor 40
3.5.8 gate-xor 40
3.5.9 gate-xnor 41
3.5.10 iec-gate 42
3.5.11 iec-gate-and 44
3.5.12 iec-gate-nand 44
3.5.13 iec-gate-buf 45
3.5.14 iec-gate-not 45
3.5.15 iec-gate-or 46
3.5.16 iec-gate-nor 47
3.5.17 iec-gate-xor 47
3.5.18 iec-gate-xnor 48

1 Introduction

This package provides a way to make beautiful block circuit diagrams using the CeTZ package.

2 Usage

Simply import Circuiteria and call the `circuit` function:

```
#import "@preview/circuiteria:0.2.1"
#circuiteria.circuit({
  import circuiteria: *
  ...
})
```

2.1 Project installation

If you have installed Circuiteria directly in your project, import src/lib.typ and call the `circuit` function:

```
#import "src/lib.typ" as circuiteria
#circuiteria.circuit({
  import circuiteria: *
  ...
})
```

3 Reference

3.1 circuit

- circuit()

3.1.1 circuit

Draws a block circuit diagram

This function is also available at the package root

Parameters

```
circuit(
  body: none array element,
  length: length ratio
) -> none
```

body `none` or `array` or `element`

A code block in which draw functions have been called

length `length` or `ratio`

Optional base unit

Default: `2em`

3.2 util

- lpad()
- normalize-coord()
- opposite-anchor()
- port-shape-vertices()
- resolve-rel-coord()
- rotate-anchor()

Variables

- colors

3.2.1 lpad

Pads a string on the left with 0s to the given length

```
#util.lpad("0100", 8)
```

```
00000100
```

Parameters

```
lpad(
  string: str,
  len: int
) -> str
```

string `str`

The string to pad

len `int`

The target length

3.2.2 normalize-coord

Normalizes a coordinate so that 2D points (x, y) are padded to (x, y, 0) as required by CeTZ's coordinate resolver. Also handles function-based and dictionary-based coordinates recursively.

Parameters

```
normalize-coord(coord: any) -> any
```

coord `any`

A CeTZ-compatible coordinate

3.2.3 opposite-anchor

Returns the anchor on the opposite side of the given one

```
#util.opposite-anchor("west")
```

east

Parameters

```
opposite-anchor(anchor: str) -> str
```

anchor `str`

The input anchor

3.2.4 port-shape-vertices

Returns the pentagon port shape vertices and the outer/inner anchor offsets for the given side and direction.

The pentagon straddles the block boundary: 1 unit outside, 1 unit inside; height ±0.5 units. For `kind: "in"` the tip points into the block; for `kind: "out"` it points away.

Parameters

```
port-shape-vertices(
  side: str,
  kind: str,
  u: length number
) -> dictionary: `(vertices: (..5 pairs..) outer: (dx dy) inner: (dx dy))`
```

side `str`

One of `"north"`, `"east"`, `"south"`, `"west"`

kind `str`

`"in"` or `"out"`

u `length` or `number`

The pentagon unit size

3.2.5 resolve-rel-coord

Resolves a relative coordinate shorthand of the form `(rel: <number>, to: <anchor>)` to a single scalar along the requested axis.

Used to implement the shared `(rel:, to:)` shorthand accepted by the `x` and `y` parameters of `elmt`, `gate` and `iec-gate`.

Parameters

```
resolve-rel-coord(
  ctx: dictionary,
  val: dictionary,
  axis: int
) -> number
```

ctx `dictionary`

The current CeTZ context

val `dictionary`

A dictionary with fields `rel` (number) and `to` (anchor)

axis `int`

`0` for x, `1` for y

3.2.6 rotate-anchor

Returns the anchor rotated 90 degrees clockwise relative to the given one

```
#util.rotate-anchor("west")
```

north

Parameters

```
rotate-anchor(anchor: str) -> str
```

anchor `str`

The anchor to rotate

3.2.7 colors

Predefined color palette

orange yellow green pink

purple blue

3.3 wire

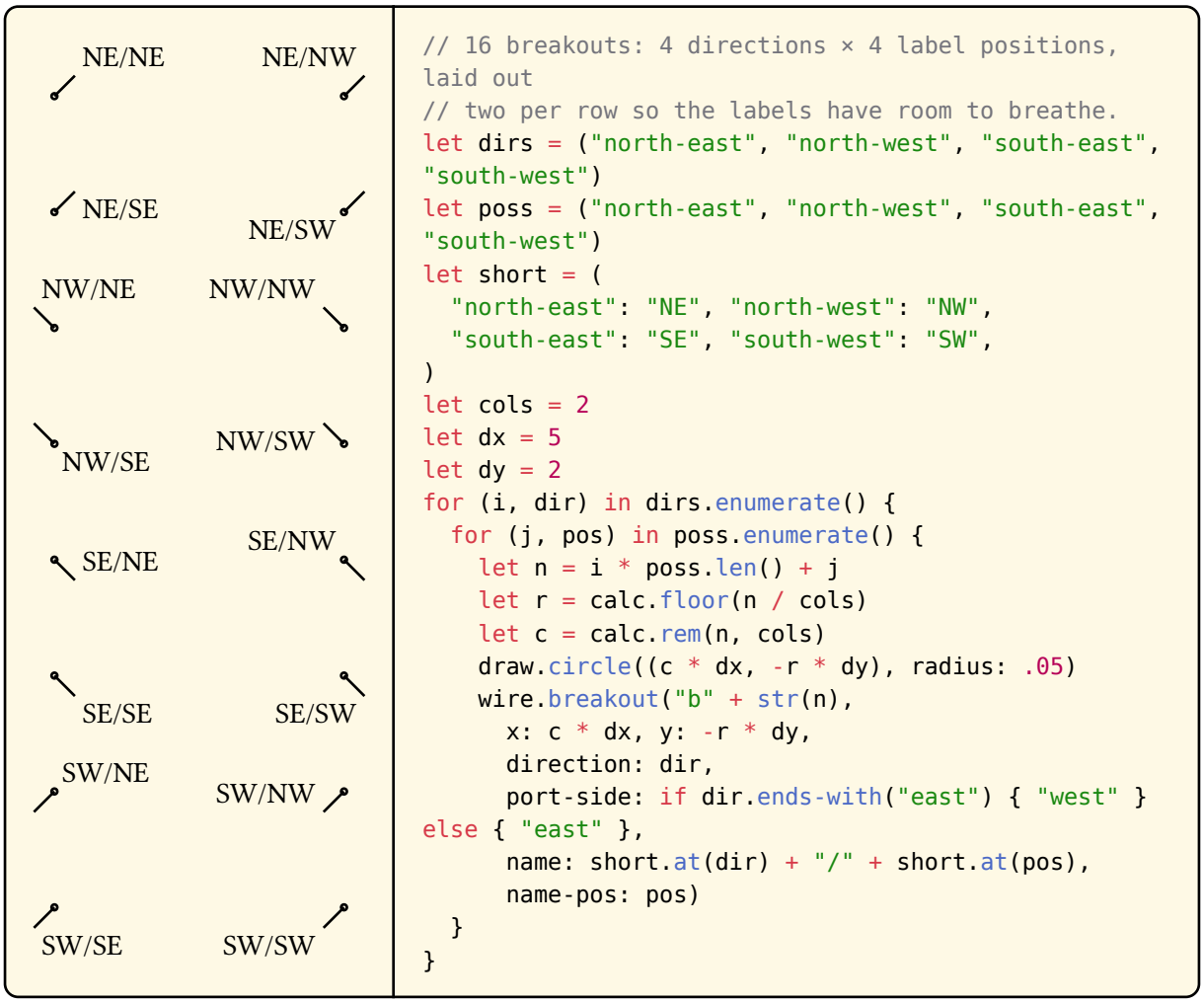
- breakout()
- intersection()
- stub()
- wire()

Variables

- wire-styles

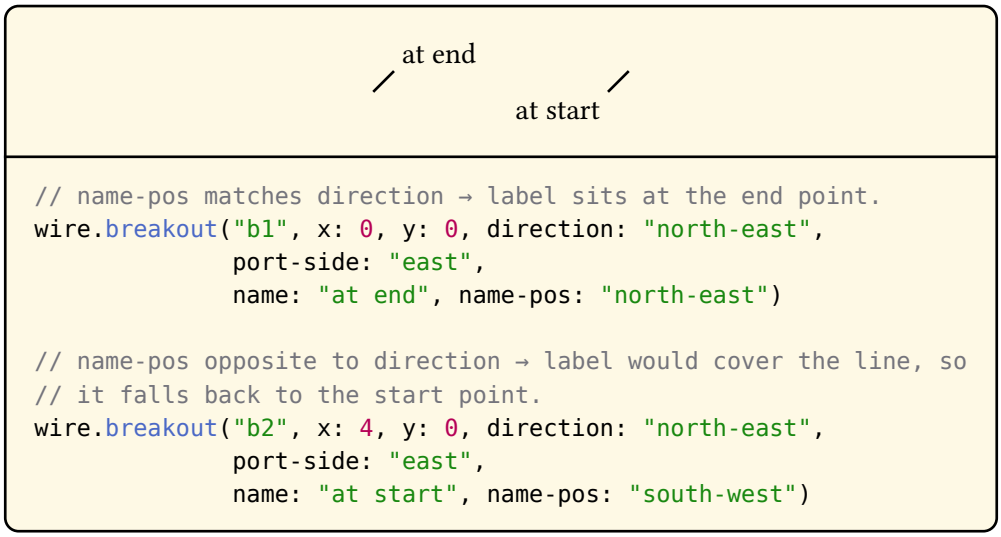
3.3.1 breakout

Draws a wire breakout: a 45° diagonal line starting at (x, y) and ending 1 unit away in one of the four diagonal directions. The end point exposes a port anchor named `<id>-port-<id>` so wires can attach to it as if it were a block port. A synthesized `<id>.center` anchor is placed opposite `port-side` so `wire()` with `style: "u-shape"` can auto-detect the port side.




```
// 16 breakouts: 4 directions × 4 label positions,
laid out
// two per row so the labels have room to breathe.
let dirs = ("north-east", "north-west", "south-east",
"south-west")
let poss = ("north-east", "north-west", "south-east",
"south-west")
let short = (
  "north-east": "NE", "north-west": "NW",
  "south-east": "SE", "south-west": "SW",
)
let cols = 2
let dx = 5
let dy = 2
for (i, dir) in dirs.enumerate() {
  for (j, pos) in poss.enumerate() {
    let n = i * poss.len() + j
    let r = calc.floor(n / cols)
    let c = calc.rem(n, cols)
    draw.circle((c * dx, -r * dy), radius: .05)
    wire.breakout("b" + str(n),
      x: c * dx, y: -r * dy,
      direction: dir,
      port-side: if dir.ends-with("east") { "west" }
else { "east" },
      name: short.at(dir) + "/" + short.at(pos),
      name-pos: pos)
  }
}
```

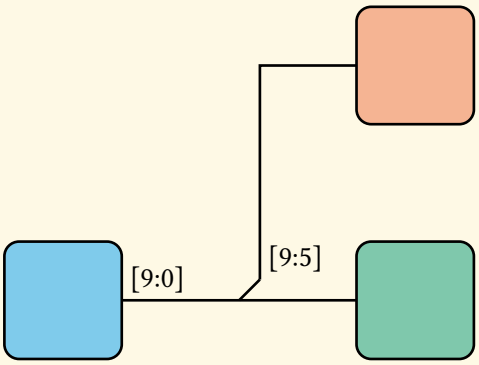
Label positions are anchored so that the label sits neatly next to the end point of the breakout. When the label direction matches the breakout direction (which would overlap the line) the label is automatically placed at the start point instead so the text does not cover the line.




```
// name-pos matches direction → label sits at the end point.
wire.breakout("b1", x: 0, y: 0, direction: "north-east",
              port-side: "east",
              name: "at end", name-pos: "north-east")

// name-pos opposite to direction → label would cover the line, so
// it falls back to the start point.
wire.breakout("b2", x: 4, y: 0, direction: "north-east",
              port-side: "east",
              name: "at start", name-pos: "south-west")
```

The port anchor makes it easy to fan out a bus:




```
// Bus fan-out: a horizontal wire from the
left block to the lower
// right block carries the full 10-bit bus.
A breakout in the middle
// of that wire taps bits [9:5] and routes
them to the upper right
// block via an L-shaped zigzag (up, then
right).
element.block(x: 0, y: 0, w: 2, h: 2, id:
"src",
  fill: util.colors.blue,
  ports: (east: ((id: "out"),)))
element.block(x: 6, y: 0, w: 2, h: 2, id:
"dstL",
  fill: util.colors.green,
  ports: (west: ((id: "in"),)))
element.block(x: 6, y: 4, w: 2, h: 2, id:
"dstU",
  fill: util.colors.orange,
  ports: (west: ((id: "in"),)))

// Full bus wire, labelled "[9:0]" on the
left.
wire.wire("wbus", ("src-port-out", "dstL-
port-in"),
          name: "[9:0]", name-pos: "start")

// Breakout tapped from the midpoint of the
bus.
wire.breakout("tap",
  x: (rel: 2, to: "src-port-out"),
  y: (from: "src-port-out"),
  direction: "north-east",
  port-side: "west",
  name: "[9:5]", name-pos: "north-east")

// L-shaped wire from the breakout up and
then right to the upper
// block's input.
wire.wire("wtap", ("tap-port-tap", "dstU-
port-in"),
          style: "zigzag", zigzag-dir:
"horizontal",
          zigzag-ratio: 100%)
```

Parameters

```
breakout(
  id: str,
  x: number dictionary,
  y: number dictionary,
  direction: str,
  length: length number,
  port-side: str,
  stroke: stroke,
  name: none str,
  name-pos: none str,
  debug: bool
)
```

id `str`

The breakout id (also used as the port id)

x `number` or `dictionary`

The starting x position (same forms as `element.block`)

Default: `none`

y `number` or `dictionary`

The starting y position (same forms as `element.block`)

Default: `none`

direction `str`

One of `"north-east"`, `"north-west"`, `"south-east"`, `"south-west"`

Default: `"north-east"`

length `length` or `number`

The diagonal length (default `1em`, matches port size)

Default: `1em`

port-side `str`

Cardinal side used for wire-attach behavior (`"north"`, `"east"`, `"south"`, `"west"`)

Default: `"east"`

stroke `stroke`

Line stroke (paint + thickness)

Default: `black + 1pt`

name `none` or `str`

Optional label displayed near the breakout

Default: `none`

name-pos `none` or `str`

Label position, one of `"north-west"`, `"north-east"`, `"south-west"`, `"south-east"`

Default: `none`

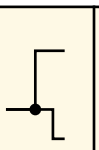
debug `bool`

If true, shows a red dot at the port anchor

Default: `false`

3.3.2 intersection

Draws a wire intersection at the given anchor



```
wire.wire("w1", ((0, 0), (1, 1)), style: "zigzag")
wire.wire("w2", ((0, 0), (1, -.5)),
          style: "zigzag", zigzag-ratio: 80%)
wire.intersection("w1.zig")
```

Parameters

```
intersection(
  pt: point,
  radius: number,
  fill: color
)
```

pt `point`

A CeTZ compatible point / anchor

radius `number`

The radius of the intersection

Default: `.1`

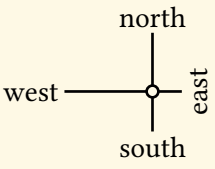
fill `color`

The fill color

Default: `black`

3.3.3 stub

Draws a wire stub (useful for unlinked ports)




```
draw.circle((0, 0), radius: .1, name: "p")
wire.stub("p", "north", name: "north", length: 1)
wire.stub("p", "east", name: "east", vertical: true)
wire.stub("p", "south", name: "south", length: 15pt)
wire.stub("p", "west", name: "west", length: 3em)
```

Parameters

```
stub(
  port-id: str,
  side: str,
  name: none str,
  vertical: bool,
  length: number,
  name-offset: number
)
```

port-id `str`

The port anchor

side `str`

The side on which the port is (one of “north”, “east”, “south”, “west”)

name `none` or `str`

Optional name displayed at the end of the stub

Default: `none`

vertical `bool`

Whether the name should be displayed vertically

Default: `false`

length `number`

The length of the stub

Default: `1em`

name-offset `number`

The name offset, perpendicular to the stub

Default: `0`

3.3.4 wire

Draws a wire between two points

Parameters

```
wire(
  id: str,
  pts: array,
  bus: bool,
  name: none str array,
  name-pos: str,
  slice: none array,
  color: color,
  dashed: bool,
  style: str,
  reverse: bool,
  guided-center: array,
  guided-margins: array,
  guided-sides: array,
  directed: bool,
  rotate-name: bool,
  zigzag-ratio: auto ratio,
  zigzag-dir: str,
  zigzag-at: none str array dictionary,
  dodge-y: number,
  dodge-sides: array,
  dodge-margins: array,
  u-side: auto str,
  u-depth: number
)
```

id `str`

The wire's id, for future reference (anchors)

pts `array`

The two points (as CeTZ compatible coordinates, i.e. XY, relative positions, ids, etc.)

bus `bool`

Whether the wire is a bus (multiple bits) or a simple signal (single bit)

Default: `false`

name `none` or `str` or `array`

Optional name of the wire. If it is an array, the first name will be put at the start of the wire, and the second at the end

Default: `none`

name-pos `str`

Position of the name. One of: “middle”, “start” or “end”

Default: `"middle"`

slice `none` or `array`

Optional bits slice (start and end bit indices). If set, it will be displayed at the start of the wire

Default: `none`

color `color`

The stroke color

Default: `black`

dashed `bool`

Whether the stroke is dashed or not

Default: `false`

style `str`

The wire’s style (see `wire-styles` for possible values)

Default: `"direct"`

reverse `bool`

If true, the start and end points will be swapped (useful in cases where the start point depends on the end point, for example with perpendiculars)

Default: `false`

guided-center `array`

the horizontal and vertical guides of the center guides (only with style “guided”)

Default: `(50%, 50%)`

guided-margins `array`

the start and end of guided margins of the wire (only with style “guided”)

Default: `(5%, 5%)`

guided-sides `array`

the side of start and end array (must be either “north”, “south”, “west”, “east”) (only work with style “guided”)

Default: `("east", "west")`

directed `bool`

If true, the wire will be directed, meaning an arrow will be drawn at the endpoint

Default: `false`

rotate-name `bool`

If true, the name will be rotated according to the wire’s slope

Default: `true`

zigzag-ratio `auto` or `ratio`

Position of the zigzag middle bend segment relative to the horizontal (or vertical) span, as a fraction between the two endpoints (only with style “zigzag”). Defaults to `auto`, which resolves to `50%`. Panics if combined with an explicitly set `zigzag-at`.

Default: `auto`

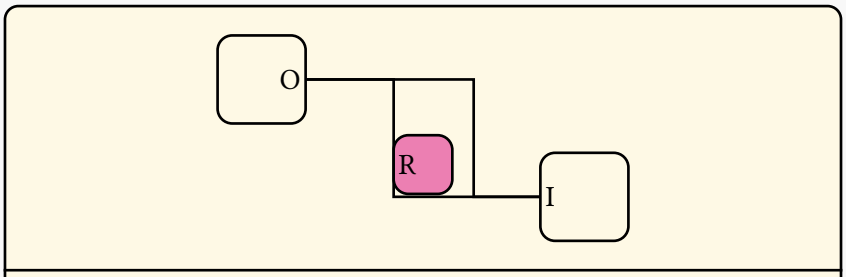
zigzag-dir `str`

The zigzag’s direction. As either “vertical” or “horizontal” (only with dstyle “zigzag”)

Default: `"vertical"`

zigzag-at `none` or `str` or `array` or `dictionary`

Optional. Anchor-based placement of the zigzag middle bend segment (only with style “zigzag”). When set, overrides `zigzag-ratio` (which must be left at its default `auto`). The middle segment sits on the axis extracted from the given coordinate: `x` for `zigzag-dir: "vertical"`, `y` for `"horizontal"`. Accepted forms: a raw CeTZ anchor name string (e.g. `"elmt.center"` or `"elmt-port-a"`); any native CeTZ coordinate (tuple, dict such as `(rel: (dx, dy), to: anchor)`); or the ergonomic scalar-offset form `(rel: <length>, to: <coord>)`, which is rewritten to a `(dx, 0)` offset for `"vertical"` and `(0, dy)` for `"horizontal"`. See




```
// `zigzag-at` places the middle bend of a zigzag wire
// on the axis of a referenced anchor, overriding
// `zigzag-ratio`.
element.block(x: 0, y: 0, w: 1.5, h: 1.5, id: "src",
  ports: (east: ((id: "o", name: "O"),),))
element.block(x: 3, y: -1.2, w: 1, h: 1, id: "ref",
  fill: util.colors.pink,
  ports: (west: ((id: "p", name: "R"),),))
element.block(x: 5.5, y: -2, w: 1.5, h: 1.5, id: "dst",
  ports: (west: ((id: "i", name: "I"),),))
// String form: anchor name of a port.
wire.wire("w1", ("src-port-o", "dst-port-i"),
          style: "zigzag",
          zigzag-at: "ref-port-p")
// Ergonomic offset: shift 8pt right of ref.east.
wire.wire("w2", ("src-port-o", "dst-port-i"),
          style: "zigzag",
          zigzag-at: (rel: 8pt, to: "ref.east"))
```

Default: `none`

dodge-y `number`

Y position to dodge the wire to (only with style “dodge”)

Default: `0`

dodge-sides `array`

The start and end sides (going out of the connected element) of the wire (only with style “dodge”)

Default: `("east", "west")`

dodge-margins `array`

The start and end margins (i.e. space before dodging) of the wire (only with style “dodge”)

Default: `(5%, 5%)`

u-side `auto` or `str`

The cardinal side both endpoints face (one of “east”, “west”, “north”, “south”). If `auto`, it is deduced from the endpoints when they are port anchors of the form `<elmt-id>-port-<port-id>` by comparing each port’s position with its element’s `.center` anchor. Auto-detection panics if the two endpoints resolve to different sides, or if neither endpoint is a port anchor (only with style “u-shape”)

Default: `auto`

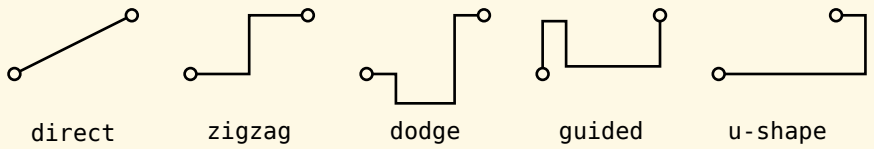
u-depth `number`

Distance between the outermost endpoint (in the `u-side` direction) and the U-bend line (only with style “u-shape”)

Default: `0.5`

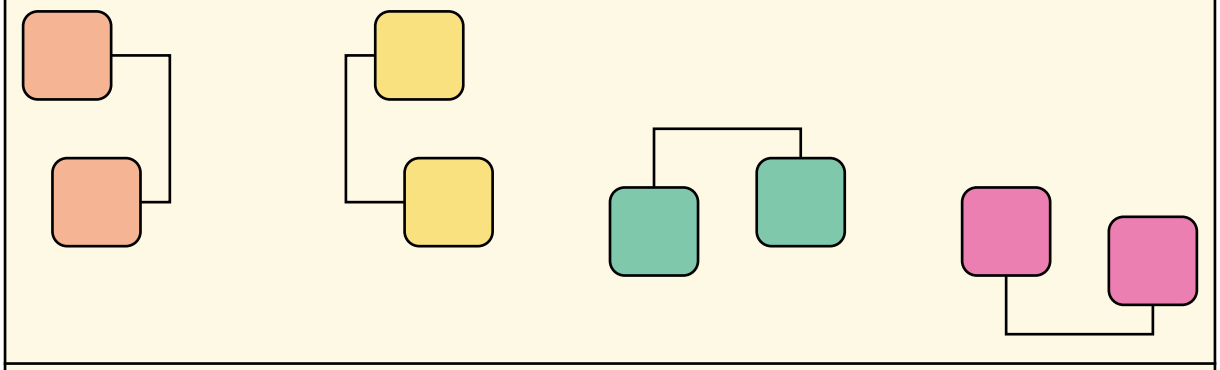
3.3.5 wire-styles

List of valid wire styles




```
for i in range(5) {
  draw.circle((i * 3, 0), radius: .1, name: "p" + str(i * 2))
  draw.circle((i * 3 + 2, 1), radius: .1, name: "p" + str(i * 2 + 1))
  draw.content((i * 3 + 1, -1), raw(wire.wire-styles.at(i)))
}
wire.wire("w1", ("p0", "p1"), style: "direct")
wire.wire("w2", ("p2", "p3"), style: "zigzag")
wire.wire("w3", ("p4", "p5"), style: "dodge",
          dodge-y: -0.5, dodge-margins: (0.5, 0.5))
wire.wire("w4", ("p6","p7"), style: "guided",
          guided-center:(20%, 40%), guided-margins: (90%,87%),
          guided-sides: ("north","south"))
wire.wire("w5", ("p8", "p9"), style: "u-shape",
          u-side: "east", u-depth: 0.5)
```

The `"u-shape"` style connects two ports that face the same cardinal side.



```
// east <-> east (u-side auto-detected)
element.block(x: 0,   y: 0,    w: 1.5, h: 1.5, id: "eA",
  fill: util.colors.orange,
  ports: (east: ((id: "out"),)))
element.block(x: 0.5, y: -2.5, w: 1.5, h: 1.5, id: "eB",
  fill: util.colors.orange,
  ports: (east: ((id: "out"),)))
wire.wire("uEE", ("eA-port-out", "eB-port-out"),
          style: "u-shape", u-depth: 0.5)

// west <-> west (auto-detected)
element.block(x: 6,   y: 0,    w: 1.5, h: 1.5, id: "wA",
  fill: util.colors.yellow,
  ports: (west: ((id: "in"),)))
element.block(x: 6.5, y: -2.5, w: 1.5, h: 1.5, id: "wB",
  fill: util.colors.yellow,
  ports: (west: ((id: "in"),)))
wire.wire("uWW", ("wA-port-in", "wB-port-in"),
          style: "u-shape", u-depth: 0.5)

// north <-> north (auto-detected)
element.block(x: 10,   y: -3,   w: 1.5, h: 1.5, id: "nA",
  fill: util.colors.green,
  ports: (north: ((id: "out"),)))
element.block(x: 12.5, y: -2.5, w: 1.5, h: 1.5, id: "nB",
  fill: util.colors.green,
  ports: (north: ((id: "out"),)))
wire.wire("uNN", ("nA-port-out", "nB-port-out"),
          style: "u-shape", u-depth: 0.5)

// south <-> south with explicit u-side override
element.block(x: 16,   y: -3,   w: 1.5, h: 1.5, id: "sA",
  fill: util.colors.pink,
  ports: (south: ((id: "out"),)))
element.block(x: 18.5, y: -3.5, w: 1.5, h: 1.5, id: "sB",
  fill: util.colors.pink,
  ports: (south: ((id: "out"),)))
wire.wire("uSS", ("sA-port-out", "sB-port-out"),
          style: "u-shape", u-side: "south", u-depth: 0.5)
```

3.4 element

- elmt()
- alu()
- block()
- extender()
- multiplexer()
- group()

3.4.1 elmt

Draws an element

Parameters

```
elmt(
  draw-shape: function,
  x: number dictionary,
  y: number dictionary,
  w: number,
  h: number,
  name: none str,
  name-anchor: str,
  ports: dictionary,
  ports-margins: dictionary,
  port-name-padding: number length,
  port-names-outside: bool auto,
  port-shape: none str,
  port-kind: str,
  port-size: length,
  fill: none color,
  stroke: stroke,
  id: str,
  auto-ports: bool,
  ports-y: dictionary,
  shell: bool,
  body,
  debug: dictionary
)
```

draw-shape `function`

Draw function

Default: `default-draw-shape`

x `number` or `dictionary`

The x position (bottom-left corner).

If it is a dictionary, it should be in the format `(rel: number, to: str)`, where `rel` is the offset and `to` the base anchor

Default: `none`

y `number` or `dictionary`

The y position (bottom-left corner).

If it is a dictionary, it can take one of two forms:
- `(rel: number, to: str)`: `rel` is the y offset and `to` the base anchor (analogous to `x`)
- `(from: str, to: str)`: `from` is the base anchor and `to` is the id of the port to align with the anchor

Default: `none`

w `number`

Width of the element

Default: `none`

h `number`

Height of the element

Default: `none`

name `none` or `str`

Optional name of the block

Default: `none`

name-anchor `str`

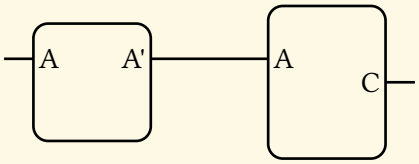
Anchor for the optional name

Default: `"center"`

ports `dictionary`

Dictionary of ports. The keys are cardinal directions (“north”, “east”, “south” and/or “west”). The values are arrays of ports (dictionaries) with the following fields:

- `id` (`str`): (Required) Port id
- `name` (`str`): Optional name displayed **in** the block
- `clock` (`bool`): Whether it is a clock port (triangle symbol)
- `vertical` (`bool`): Whether the name should be drawn vertically
- `align-with` (`str`, `dictionary`): Optional. Reuse another port’s placement coordinate (east/west: y, north/south: x). Accepted forms: `"<local-id>"` (same-element port; axis must match, transitively resolved, cycles panic), `(element: "<id>", port: "<id>")` (cross-element; axis not validated; target element must be drawn first), or `"<raw-anchor>"` (any unrecognised string is used as a CeTZ anchor name). See




```
// `align-with` reuses another port's coord
// (E/W: y, N/S: x). Forms:
//  1. "id"                       — local
//  2. (element:"e", port:"p")  — cross-elmt
//  3. "raw-anchor"               — CeTZ anchor
element.block(x: 0, y: 0, w: 2, h: 2, id: "b1",
  ports: (
    west: ((id: "a", name: "A", y: 30%),),
    east: ((id: "oa", name: "A'",
            "align-with": "a"),),
  ))
element.block(x: 4, y: -.3, w: 2, h: 2.6, id: "b2",
  ports: (
    west: ((id: "ia", name: "A",
            "align-with":
              (element: "b1", port: "oa")),),
    east: ((id: "o", name: "C"),),
  ))
wire.stub("b1-port-a", "west")
wire.wire("w", ("b1-port-oa", "b2-port-ia"),
          style: "direct")
wire.stub("b2-port-o", "east")
```

- `x` (length, ratio, function): Optional. North/south ports only. Explicit x-offset override measured from the left corner of the side. A `ratio` is used as a percentage along the side; a length/number is used as an absolute offset; a function is called with the element width and must return a length/number. Panics if used together with `align-with` or on an east/west port.
- `y` (length, ratio, function): Optional. East/west ports only. Analogous to `x`, measured from the top corner of the side and called with the element height when a function is passed.
- `shape` (`none`, `str`): Optional. Per-port override of the element-level `port-shape` (e.g. `"pentagon"`). Lets a single port on a block or group opt into (or out of) a shape independently of the element default. See the `port-shape` parameter below.
- `kind` (`str`): Optional. Per-port override of the element-level `port-kind` (`"in"` / `"out"`). Affects pentagon orientation.
- `size` (length): Optional. Per-port override of the element-level `port-size` (unit size of pentagon ports).
- `stroke` (`none`, `paint`, `stroke`): Optional. Per-port override of the pentagon stroke. Defaults to the element stroke.
- `fill` (`none`, color): Optional. Per-port override of the pentagon fill. Defaults to the element fill (or `gray` for clock ports with `shape: "pentagon"`).

Default: (:)

ports-margins `dictionary`

Dictionary of ports margins (used with automatic port placement). They keys are cardinal directions (“north”, “east”, “south”, “west”). The values are tuples of (<start>, <end>) margins (numbers)

Default: `(:)`

port-name-padding `number` or `length`

Padding between the port name label and the port anchor point (default `2pt`)

Default: `2pt`

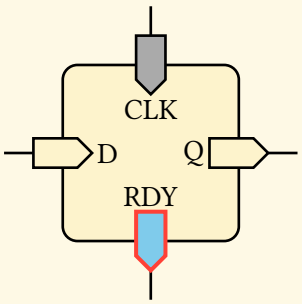
port-names-outside `bool` or `auto`

If true, port name labels are drawn on the outside of the block instead of the inside. When `auto` (default), follows the `shell` setting (outside if `shell: true`, inside otherwise)

Default: `auto`

port-shape `none` or `str`

Default port shape for all ports of this element. Set to `"pentagon"` to draw pentagon-shaped ports. Overridable per port via the port's `shape` field (see the ports dictionary above); a per-port `shape: none` also removes the shape for just that port. See




```
// Block with pentagon-shaped ports.
// - port-shape / port-kind / port-size are element-
level defaults.
// - Any port field (shape, kind, size, stroke,
fill) overrides per port.
// - kind: "in"  → tip points into the block
(default)
//   kind: "out" → tip points away from the block
// - clock ports with shape: "pentagon" drop their
triangle and default
//   to a gray fill.
element.block(
  id: "b", w: 3, h: 3, x: 0, y: 0,
  fill: util.colors.yellow.lighten(60%),
  port-shape: "pentagon",
  port-size: 1em,
  ports: (
    // west: default input
    west:  ((id: "in",  name: "D"),),
    // east: output — tip points away
    east:  ((id: "out", name: "Q", kind: "out"),),
    // north: clock port — pentagon replaces the
triangle, gray fill
    north: ((id: "clk", name: "CLK", clock: true),),
    // south: styled output with per-port fill and
stroke
    south: ((id: "rdy", name: "RDY", kind: "out",
             fill: util.colors.blue, stroke: red +
1.5pt),),
  ),
)
// Wires connect at the outer end of each pentagon.
wire.stub("b-port-in",  "west")
wire.stub("b-port-out", "east")
wire.stub("b-port-clk", "north")
wire.stub("b-port-rdy", "south")
```

Default: `none`

port-kind `str`

Default direction of ports (`"in"` or `"out"`). Overridable per port via the port's `kind` field

Default: `"in"`

port-size `length`

Default unit size of pentagon-shaped ports (default `1em`). Overridable per port via the port's `size` field

Default: `1em`

fill `none` or `color`

Fill color

Default: `none`

stroke `stroke`

Border stroke

Default: `black + 1pt`

id `str`

The block id (for future reference)

Default: `""`

auto-ports `bool`

Whether to use auto port placements or not. If false, `draw-shape` is responsible for adding the appropiate ports

Default: `true`

ports-y `dictionary`

Dictionary of the ports y offsets (used with `auto-ports: false`)

Default: `(:)`

shell `bool`

If true, the block behaves like a shell/container (analogous to a group): port name labels default to being drawn **outside** the block border, and wires attach to the **inside** tip of pentagon-shaped ports instead of the outside base. Useful for wrapping child elements without having ports overlap them (default `false`)

Default: `false`

debug `dictionary`

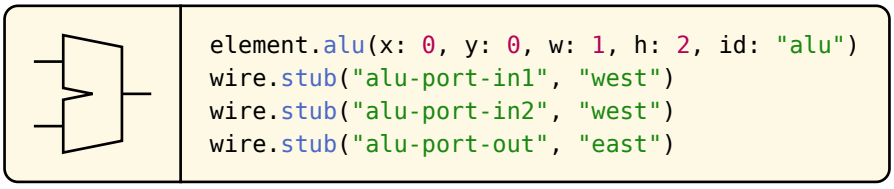
Dictionary of debug options.

Supported fields include:
- `ports`: if true, shows dots on all ports of the element

Default: `(`
```
    ports: false,
  )
```

3.4.2 alu

Draws an ALU with two inputs



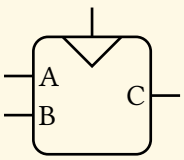
```
element.alu(x: 0, y: 0, w: 1, h: 2, id: "alu")
wire.stub("alu-port-in1", "west")
wire.stub("alu-port-in2", "west")
wire.stub("alu-port-out", "east")
```

For parameters description, see `elmt()`

Parameters
```
alu(
  x,
  y,
  w,
  h,
  name,
  name-anchor,
  fill,
  stroke,
  id,
  debug
)
```

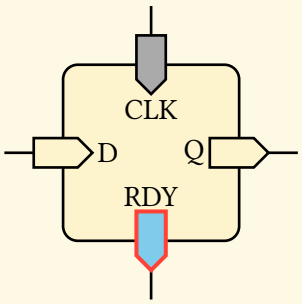
3.4.3 block

Draws a block element



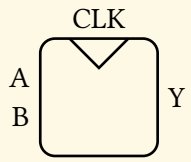

```
element.block(
  x: 0, y: 0, w: 2, h: 2, id: "block",
  ports: (
    north: ((id: "clk", clock: true),),
    west: ((id: "in1", name: "A"),
           (id: "in2", name: "B")),
    east: ((id: "out", name: "C"),)
  )
)
wire.stub("block-port-clk", "north")
wire.stub("block-port-in1", "west")
wire.stub("block-port-in2", "west")
wire.stub("block-port-out", "east")
```

Ports can be given a pentagon shape (with configurable direction, size, stroke and fill) via `port-shape` and per-port overrides:




```
// Block with pentagon-shaped ports.
// - port-shape / port-kind / port-size are element-
level defaults.
// - Any port field (shape, kind, size, stroke, fill)
overrides per port.
// - kind: "in"  → tip points into the block (default)
//   kind: "out" → tip points away from the block
// - clock ports with shape: "pentagon" drop their
triangle and default
//   to a gray fill.
element.block(
  id: "b", w: 3, h: 3, x: 0, y: 0,
  fill: util.colors.yellow.lighten(60%),
  port-shape: "pentagon",
  port-size: 1em,
  ports: (
    // west: default input
    west:  ((id: "in",  name: "D"),),
    // east: output — tip points away
    east:  ((id: "out", name: "Q", kind: "out"),),
    // north: clock port — pentagon replaces the
triangle, gray fill
    north: ((id: "clk", name: "CLK", clock: true),),
    // south: styled output with per-port fill and
stroke
    south: ((id: "rdy", name: "RDY", kind: "out",
             fill: util.colors.blue, stroke: red +
1.5pt),),
  ),
)
// Wires connect at the outer end of each pentagon.
wire.stub("b-port-in",  "west")
wire.stub("b-port-out", "east")
wire.stub("b-port-clk", "north")
wire.stub("b-port-rdy", "south")
```

Port name labels can be placed outside the block via `port-names-outside`:




```
element.block(
  x: 0, y: 0, w: 2, h: 2, id: "block",
  port-names-outside: true,
  port-name-padding: 4pt,
  ports: (
    north: ((id: "clk", name: "CLK", clock: true),),
    west: ((id: "in1", name: "A"),
           (id: "in2", name: "B")),
    east: ((id: "out", name: "Y"),)
  )
)
```

For other parameters description, see `elmt()`

Parameters

```
block(
  x,
  y,
  w,
  h,
  name,
  name-anchor,
  ports,
  ports-margins,
  port-name-padding,
  port-names-outside,
  port-shape,
  port-kind,
  port-size,
  fill,
  radius: number  length  ratio  dictionary,
  stroke,
  id,
  shell: bool,
  debug,
  ..args
)
```

radius `number` or `length` or `ratio` or `dictionary`

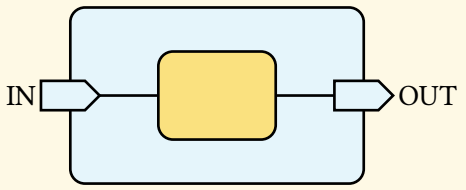
The corner radius of the block. See CeTZ documentation for more information

Default: `0.5em`

shell `bool`

If true, the block behaves like a shell/container (analogous to a group): port name labels default to being drawn outside the block, and wires attach to the inside tip of pentagon-shaped ports. See `elmt()`

Use `shell: true` to wrap child elements inside a bordered container without having ports overlap the interior:

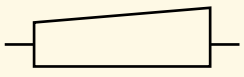



```
// A shell block acts as a container:
port name labels are drawn
// outside by default, and wires attach
at the *inside* tip of
// pentagon-shaped ports so child
elements can sit within.
element.block(
  id: "sh", w: 5, h: 3, x: 0, y: 0,
  shell: true,
  port-shape: "pentagon",
  fill: util.colors.blue.lighten(80%),
  ports: (
    west: ((id: "in",  name: "IN"),),
    east: ((id: "out", name: "OUT",
kind: "out"),),
  ),
  {
    element.block(id: "inner", w: 2, h:
1.5, x: 1.5, y: 0.75,
      fill: util.colors.yellow,
      ports: (west: ((id: "a"),), east:
((id: "b"),)))
  }
)
wire.wire("w-in",  ("sh-port-in",
"inner-port-a"))
wire.wire("w-out", ("inner-port-b", "sh-
port-out"))
```

Default: `false`

3.4.4 extender

Draws a bit extender



```
element.extender(
  x: 0, y: 0, w: 3, h: 1,
  id: "extender"
)
wire.stub("extender-port-in", "west")
wire.stub("extender-port-out", "east")
```

For other parameters description, see `elmt()`

Parameters

```
extender(
  x,
  y,
  w,
  h,
  name,
  name-anchor,
  fill,
  stroke,
  id,
  h-ratio: ratio,
  align-out: bool,
  debug
)
```

h-ratio `ratio`

The height ratio of the left side relative to the full height

Default: `75%`

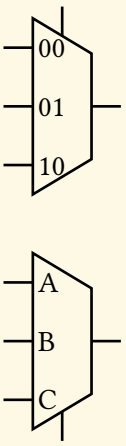
align-out `bool`

If true, the output and input ports are aligned, otherwise, the output port is centered on the right side

Default: `true`

3.4.5 multiplexer

Draws a multiplexer




```
element.multiplexer(
  x: 0, y: 0, w: 1, h: 3,
  id: "multiplexer",
  entries: 3
)
wire.stub("multiplexer.north", "north")
wire.stub("multiplexer-port-out", "east")

element.multiplexer(
  x: 0, y: -4, w: 1, h: 3,
  id: "multiplexer2",
  entries: ("A", "B", "C")
)
wire.stub("multiplexer2.south", "south")
wire.stub("multiplexer2-port-out", "east")

for i in range(3) {
  wire.stub("multiplexer-port-in" + str(i), "west")
  wire.stub("multiplexer2-port-in" + str(i), "west")
}
```

For other parameters description, see `elmt()`

Parameters

```
multiplexer(
  x,
  y,
  w,
  h,
  name,
  name-anchor,
  entries: int array,
  h-ratio: ratio,
  fill,
  stroke,
  id,
  debug
)
```

entries `int` or `array`

If it is an integer, it defines the number of input ports (automatically named with their binary index). If it is an array of strings, it defines the name of each input.

Default: `2`

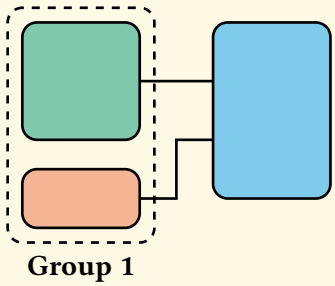
h-ratio `ratio`

The height ratio of the right side relative to the full height

Default: `60%`

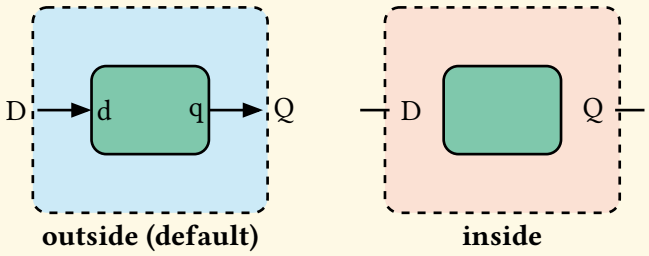
3.4.6 group

Draws a group of elements




```
element.group(
  id: "g1", name: "Group 1", stroke: (dash: "dashed"),
  {
    element.block(id: "b1", w: 2, h: 2,
      x: 0, y: 1.5,
      ports: (east: ((id: "out"),)),
      fill: util.colors.green
    )
    element.block(id: "b2", w: 2, h: 1,
      x: 0, y: 0,
      ports: (east: ((id: "out"),)),
      fill: util.colors.orange
    )
  }
)
element.block(id: "b3", w: 2, h: 3,
  x: (rel: 1, to: "g1.east"),
  y: (from: "b1-port-out", to: "in1"),
  ports: (west: ((id: "in1"), (id: "in2"))),
  fill: util.colors.blue
)
wire.wire("w1", ("b1-port-out", "b3-port-in1"))
wire.wire("w2", ("b2-port-out", "b3-port-in2"),
          style: "zigzag")
```

Ports can also be defined on a group, just like on a block. By default port labels are drawn **outside** the group border so they do not overlap child elements. Set `outside-port-names: false` to draw them inside.




```
// outside-port-names: true (default) — labels outside the border
element.group(
  id: "g1", name: "outside (default)",
  stroke: (dash: "dashed"),
  fill: util.colors.blue.lighten(60%),
  padding: 2em,
  outside-port-names: true,
  port-name-padding: 4pt,
  ports: (
    west:  ((id: "in",  name: "D"),),
    east:  ((id: "out", name: "Q"),),
  ),
  { element.block(id: "b1", w: 2, h: 1.5, x: 0, y: 0,
      fill: util.colors.green,
      ports: (west: ((id: "d", name: "d"),),
              east: ((id: "q", name: "q"),))) }
)
wire.wire("wG1in",  ("g1-port-in", "b1-port-d"), directed: true)
wire.wire("wG1out", ("b1-port-q",  "g1-port-out"), directed: true)

// outside-port-names: false — labels inside the border (block style)
// with extra padding via port-name-padding
element.group(
  id: "g2", name: "inside",
  stroke: (dash: "dashed"),
  fill: util.colors.orange.lighten(60%),
  padding: 2em,
  outside-port-names: false,
  port-name-padding: 0.4em,
  ports: (
    west:  ((id: "in",  name: "D"),),
    east:  ((id: "out", name: "Q"),),
  ),
  { element.block(id: "b2", w: 2, h: 1.5, x: 6, y: 0,
      fill: util.colors.green,
      ports: (west: ((id: "d"),), east: ((id: "q"),))) }
)
wire.stub("g2-port-in",  "west")
wire.stub("g2-port-out", "east")
```

Parameters

```
group(
  body: elements function,
  id: str,
  name: str,
  name-anchor: str,
  fill: color,
  stroke: stroke,
  padding: float length array dictionary,
  radius: number,
  ports: dictionary,
  ports-margins: dictionary,
  outside-port-names: bool,
  port-name-padding: number length,
  port-shape: none str,
  port-kind: str,
  port-size: length,
  debug: dictionary
)
```

body `elements` or `function`

Elements to group

id `str`

see `elmt()`

Default: `""`

name `str`

The group's name

Default: `none`

name-anchor `str`

The anchor for the name.
Note: the name will be placed on the **outside** of the group

Default: `"south"`

fill `color`

see `elmt()`

Default: `none`

stroke `stroke`

see `elmt()`

Default: `black + 1pt`

padding `float` or `length` or `array` or `dictionary`

The inside padding:
- float / length: same for all sides
- array: either (<all>,), (<vertical>, <horizontal>) or (<top>, <right>, <bottom>, <left>)
- dictionary: valid keys are “top”, “right”, “bottom” and “left”

Default: `0.5em`

radius `number`

The corner radius

Default: `0.5em`

ports `dictionary`

see `elmt()`

Default: `(:)`

ports-margins `dictionary`

see `elmt()`

Default: `(:)`

outside-port-names `bool`

If `true` (default), port name labels are drawn outside the group border so they do not overlap child elements. Set to `false` to draw labels inside, matching the block behaviour

Default: `true`

port-name-padding `number` or `length`

Padding between the port name label and the group border (default `2pt`). Accepts anything CeTZ accepts as `padding` (number or length)

Default: `2pt`

port-shape `none` or `str`

see `elmt()`

Default: `none`

port-kind `str`

see `elmt()`

Default: `"in"`

port-size `length`

see `elmt()`

Default: `1em`

debug `dictionary`

see `elmt()`

Default:
```
(
    ports: false
)
```

3.5 gates

- gate()
- gate-and()
- gate-nand()
- gate-buf()
- gate-not()
- gate-or()
- gate-nor()
- gate-xor()
- gate-xnor()
- iec-gate()
- iec-gate-and()
- iec-gate-nand()
- iec-gate-buf()
- iec-gate-not()
- iec-gate-or()
- iec-gate-nor()
- iec-gate-xor()
- iec-gate-xnor()

3.5.1 gate

Draws a logic gate. This function is also available as `element.gate()`

Parameters

```
gate(
  draw-shape: function,
  x: number dictionary,
  y: number dictionary,
  w: number,
  h: number,
  inputs: int,
  fill: none color,
  stroke: stroke,
  id: str,
  inverted: str array,
  inverted-radius: number,
  debug: dictionary
)
```

draw-shape `function`

see `elmt()`

Default: `default-draw-shape`

x `number` or `dictionary`

see `elmt()`

Default: `none`

y `number` or `dictionary`

see `elmt()`

Default: `none`

w `number`

see `elmt()`

Default: `none`

h `number`

see `elmt()`

Default: `none`

inputs `int`

The number of inputs

Default: `2`

fill `none` or `color`

see `elmt()`

Default: `none`

stroke `stroke`

see `elmt()`

Default: `black + 1pt`

id `str`

see `elmt()`

Default: `""`

inverted `str` or `array`

Either “all” or an array of port ids to display as inverted

Default: `()`

inverted-radius `number`

The radius of inverted ports dot

Default: `0.1`

debug `dictionary`

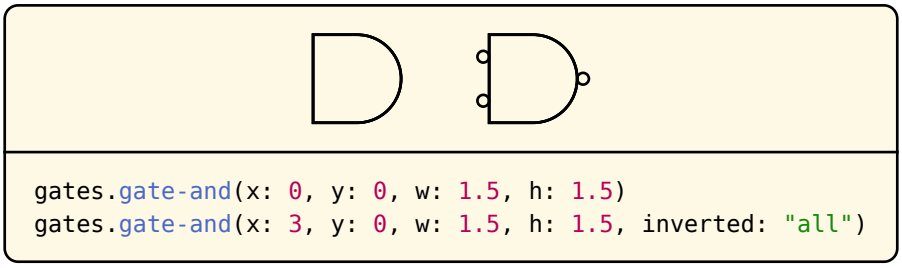
see `elmt()`

Default: `(`
```
    ports: false
  )
```

3.5.2 gate-and

Draws an AND gate. This function is also available as `element.gate-and()`

For parameters, see `gate()`

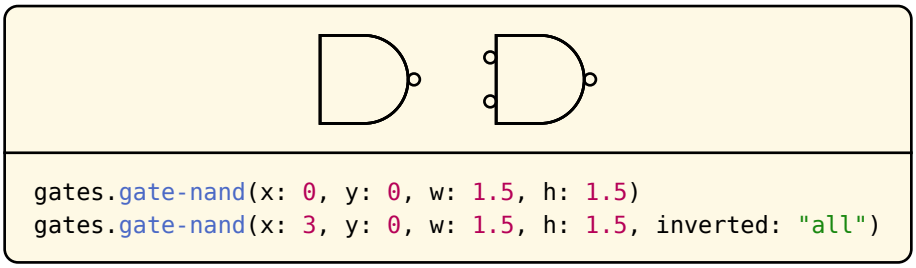


Parameters
```
gate-and(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.3 gate-nand

Draws an NAND gate. This function is also available as `element.gate-nand()`

For parameters, see `gate()`



```
gates.gate-nand(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-nand(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

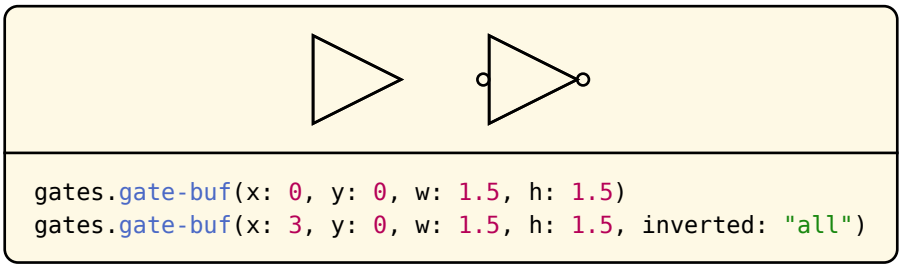
Parameters

```
gate-nand(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.4 gate-buf

Draws a buffer gate. This function is also available as `element.gate-buf()`

For parameters, see `gate()`



```
gates.gate-buf(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-buf(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

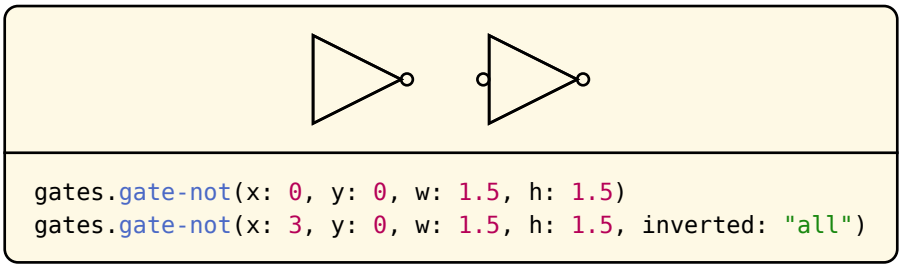
Parameters

```
gate-buf(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.5 gate-not

Draws a NOT gate. This function is also available as `element.gate-not()`

For parameters, see `gate()`



```
gates.gate-not(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-not(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

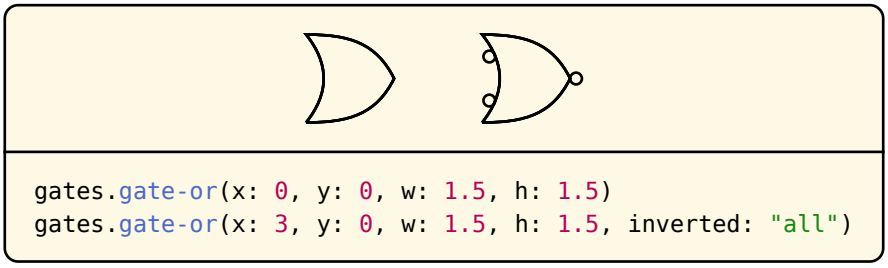
Parameters

```
gate-not(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.6 gate-or

Draws an OR gate. This function is also available as `element.gate-or()`

For parameters, see `gate()`



```
gates.gate-or(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-or(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

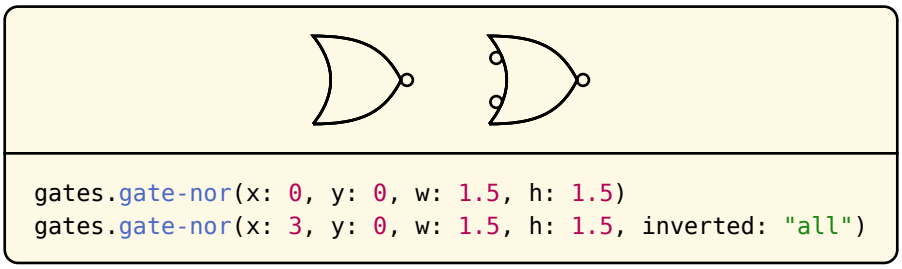
Parameters

```
gate-or(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.7 gate-nor

Draws a NOR gate. This function is also available as `element.gate-nor()`

For parameters, see `gate()`



```
gates.gate-nor(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-nor(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

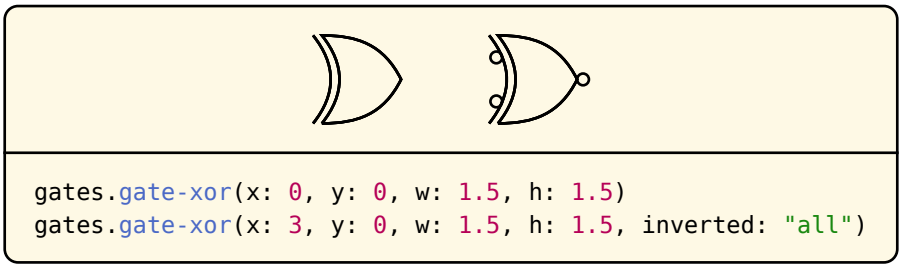
Parameters

```
gate-nor(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.8 gate-xor

Draws a XOR gate. This function is also available as `element.gate-xor()`

For parameters, see `gate()`



```
gates.gate-xor(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-xor(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

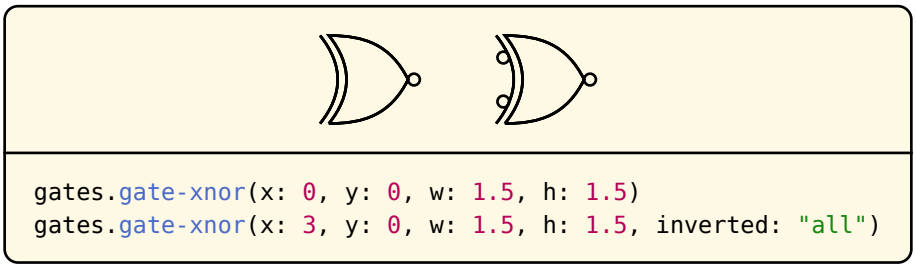
Parameters

```
gate-xor(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.9 gate-xnor

Draws a XNOR gate. This function is also available as `element.gate-xnor()`

For parameters, see `gate()`



```
gates.gate-xnor(x: 0, y: 0, w: 1.5, h: 1.5)
gates.gate-xnor(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

Parameters

```
gate-xnor(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.10 iec-gate

Draws a logic gate. This function is also available as `element.iec-gate()`

Parameters

```
iec-gate(
  draw-shape: function,
  x: number dictionary,
  y: number dictionary,
  w: number,
  h: number,
  inputs: int,
  fill: none color,
  stroke: stroke,
  id: str,
  inverted: str array,
  inverted-radius: number,
  debug: dictionary,
  symbol: str
)
```

draw-shape `function`

see `elmt()`

Default: `default-draw-shape`

x `number` or `dictionary`

see `elmt()`

Default: `none`

y `number` or `dictionary`

see `elmt()`

Default: `none`

w `number`

see `elmt()`

Default: `none`

h `number`

see `elmt()`

Default: `none`

inputs `int`

The number of inputs

Default: `2`

fill `none` or `color`

see `elmt()`

Default: `none`

stroke `stroke`

see `elmt()`

Default: `black + 1pt`

id `str`

see `elmt()`

Default: `""`

inverted `str` or `array`

Either “all” or an array of port ids to display as inverted

Default: `()`

inverted-radius `number`

The radius of inverted ports dot

Default: `0.1`

debug `dictionary`

see `elmt()`

Default:
```
(
    ports: false
)
```

symbol `str`

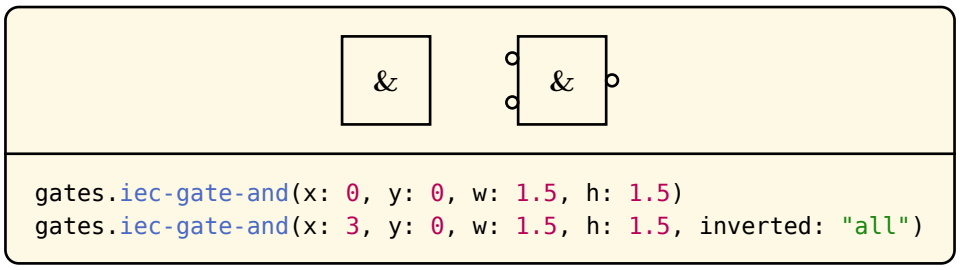
The symbol to display at the center of the gate

Default: `""`

3.5.11 iec-gate-and

Draws an IEC-AND gate. This function is also available as `element.iec-gate-and()`

For parameters, see `iec-gate()`




```
gates.iec-gate-and(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-and(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

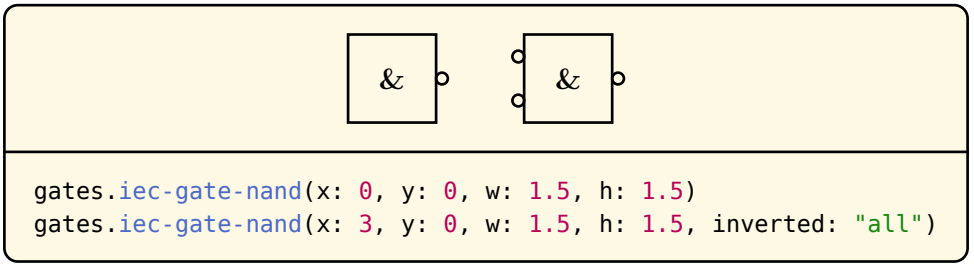
Parameters

```
iec-gate-and(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.12 iec-gate-nand

Draws an IEC-NAND gate. This function is also available as `element.iec-gate-nand()`

For parameters, see `iec-gate()`




```
gates.iec-gate-nand(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-nand(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

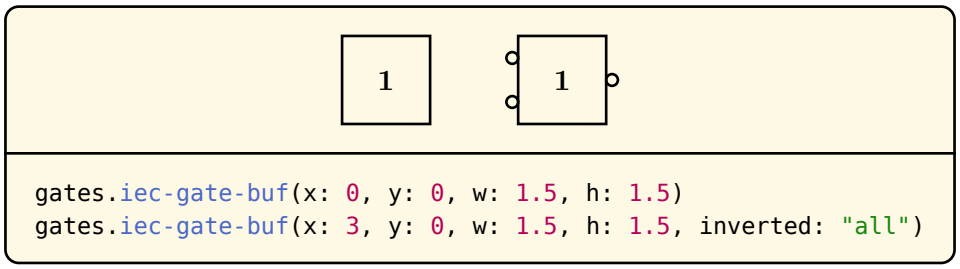
Parameters

```
iec-gate-nand(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.13 iec-gate-buf

Draws an IEC buffer gate. This function is also available as `element.iec-gate-buf()`

For parameters, see `iec-gate()`



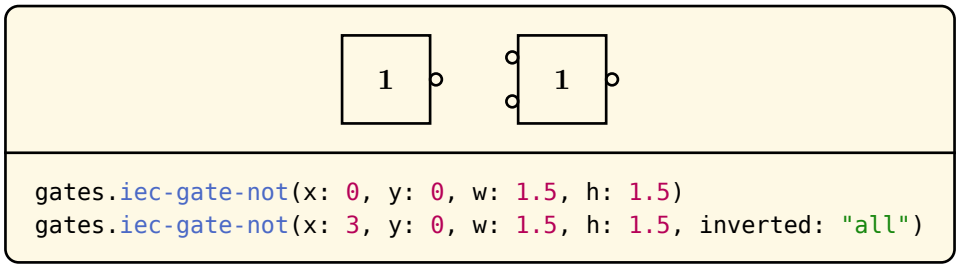
Parameters

```
iec-gate-buf(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.14 iec-gate-not

Draws an IEC NOT gate. This function is also available as `element.iec-gate-not()`

For parameters, see `iec-gate()`




```
gates.iec-gate-not(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-not(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

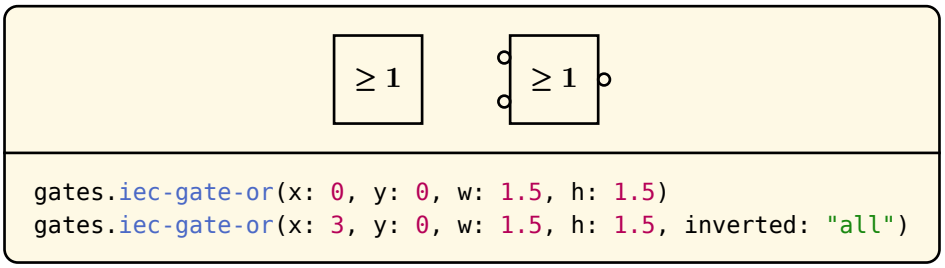
Parameters

```
iec-gate-not(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.15 iec-gate-or

Draws an IEC-OR gate. This function is also available as `element.iec-gate-or()`

For parameters, see `iec-gate()`




```
gates.iec-gate-or(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-or(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

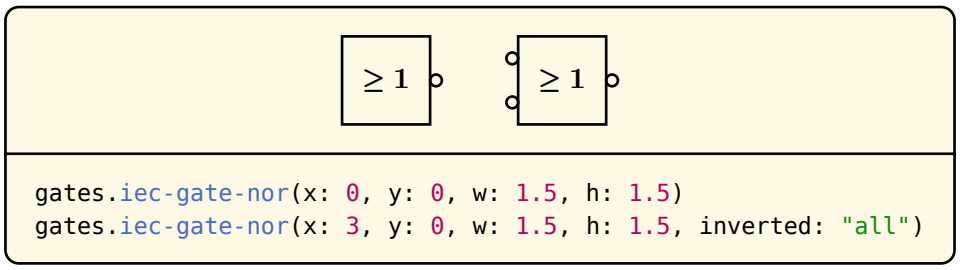
Parameters

```
iec-gate-or(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.16 iec-gate-nor

Draws an IEC-NOR gate. This function is also available as `element.iec-gate-nor()`

For parameters, see `iec-gate()`




```
gates.iec-gate-nor(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-nor(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

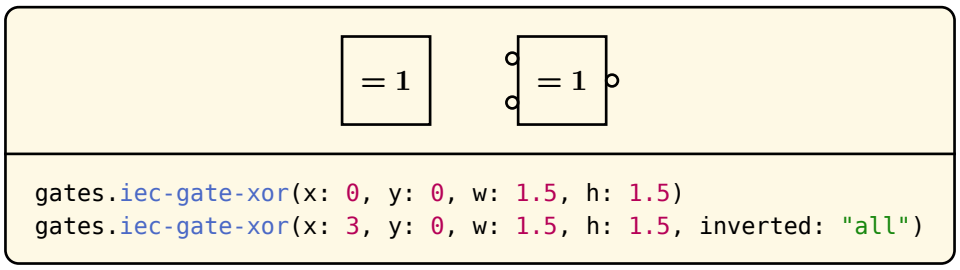
Parameters

```
iec-gate-nor(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.17 iec-gate-xor

Draws an IEC-XOR gate. This function is also available as `element.iec-gate-xor()`

For parameters, see `iec-gate()`




```
gates.iec-gate-xor(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-xor(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

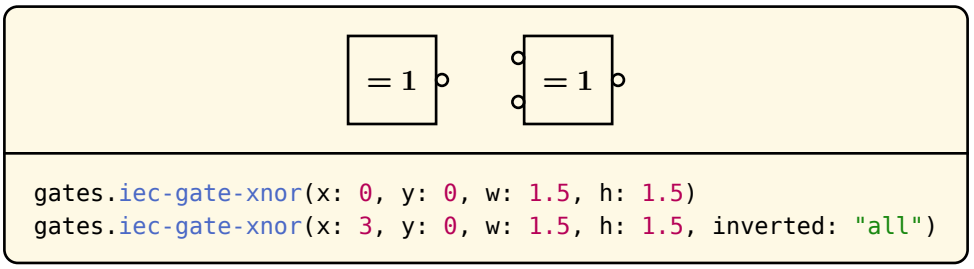
Parameters

```
iec-gate-xor(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```

3.5.18 iec-gate-xnor

Draws an IEC-XNOR gate. This function is also available as `element.iec-gate-xnor()`

For parameters, see `iec-gate()`




```
gates.iec-gate-xnor(x: 0, y: 0, w: 1.5, h: 1.5)
gates.iec-gate-xnor(x: 3, y: 0, w: 1.5, h: 1.5, inverted: "all")
```

Parameters

```
iec-gate-xnor(
  x,
  y,
  w,
  h,
  inputs,
  fill,
  stroke,
  id,
  inverted,
  debug
)
```