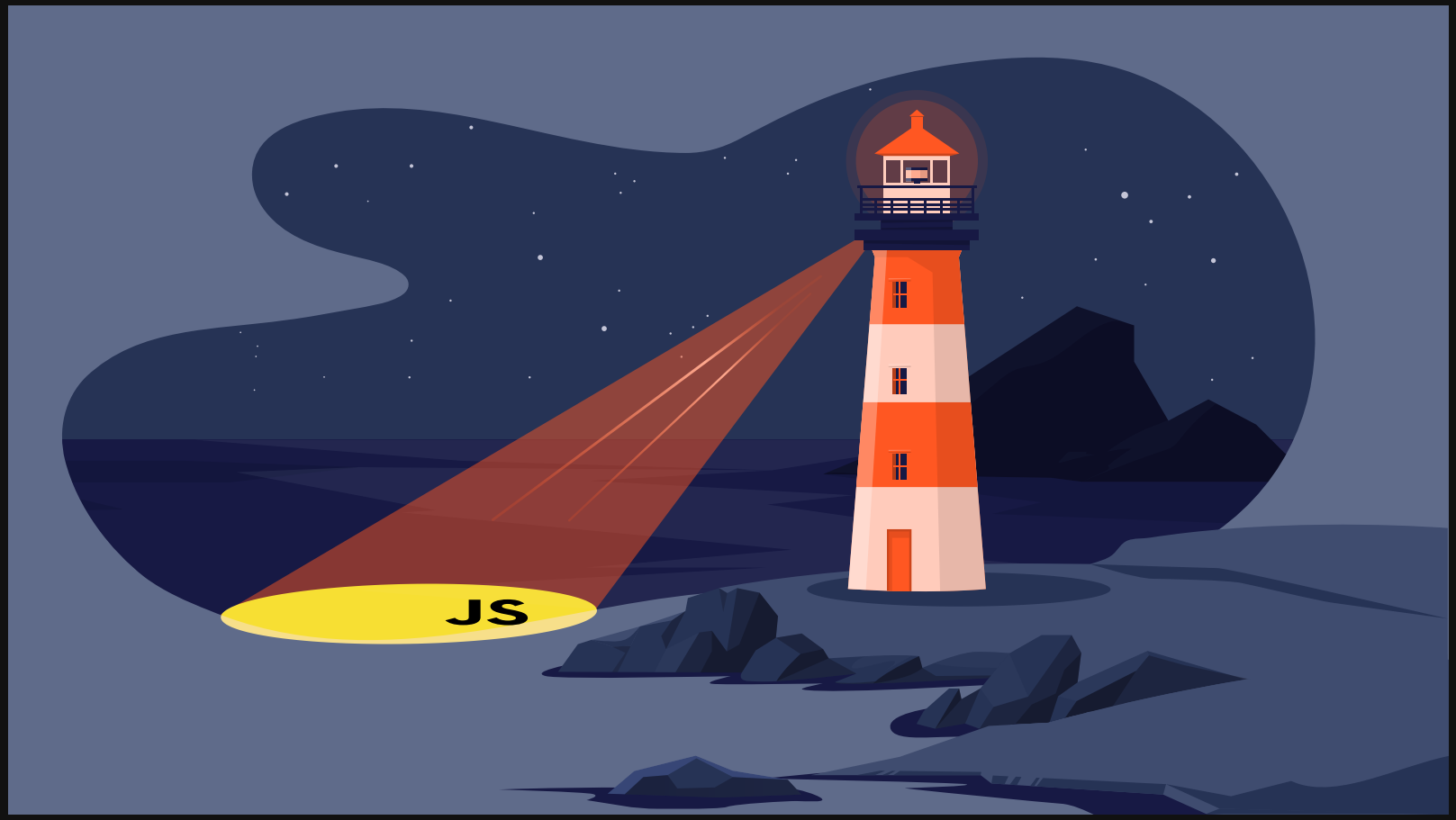
JS Signals

Use this QR Code to let me know what you thought of this talk!

adamlbarrett.bsky.social

BigAB



JS Signals?

What are JS Signals?

A not-so-new, poorly named, reactive state primitive

🙄 *...It's just a monoid in the category of endofunctors, what's the problem?*

- Anon Functional Programmer Bro

https://github.com/tc39/proposal-signals

README | Code of conduct | MIT license | Security

🚦 JavaScript Signals standard proposal 🚦

Stage 1 (explanation)

TC39 proposal champions: Daniel Ehrenberg, Yehuda Katz, Jatin Ramanathan, Shay Lewis, Kristen Hewell Garrett, Dominic Gannaway, Preston Sego, Milo M, Rob Eisenberg

Original authors: Rob Eisenberg and Daniel Ehrenberg

This document describes an early common direction for signals in JavaScript, similar to the Promises/A+ effort which preceded the Promises standardized by TC39 in ES2015. Try it for yourself, using a polyfill.

Similarly to Promises/A+, this effort focuses on aligning the JavaScript ecosystem. If this alignment is successful, then a standard could emerge, based on that experience. Several framework authors are collaborating here on a common model which could back their reactivity core. The current draft is based on design input from the authors/maintainers of Angular, Bubble, Ember, FAST, MobX, Preact, Qwik, RxJS, Solid, Starbeam, Svelte, Vue, Wiz, and more...

Differently from Promises/A+, we're not trying to solve for a common developer-facing surface API, but rather the precise core semantics of the underlying signal graph. This proposal does include a fully concrete API, but the API is not targeted to most application developers. Instead, the signal API here is a better fit for frameworks to build on top of, providing interoperability through common signal graph and auto-tracking mechanism.

The plan for this proposal is to do significant early prototyping, including integration into several frameworks, before advancing beyond Stage 1. We are only interested in standardizing Signals if they are suitable for use in

What makes then so special?

Developer
Experience

Performance

What are JS Signals?


A not-so-new, poorly named, reactive state primitive


A not-so-new, poorly
reactive state primitive
primitive

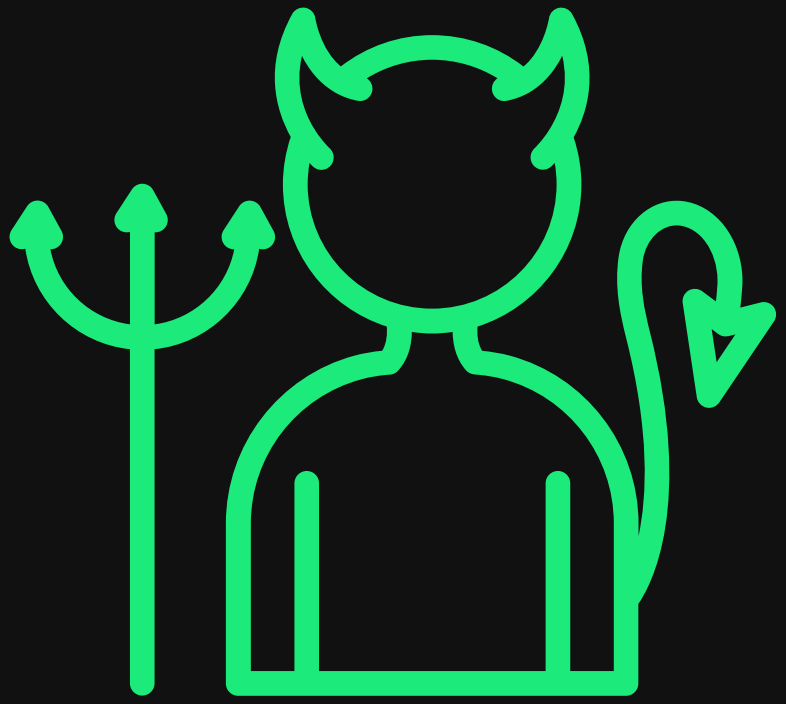
What do we mean by "reactive"?

We are **not** talking about:

- Systems that are **Responsive**, **Resilient**, **Elastic** and **Message Driven** (reactivemanifesto.org)
- Programming with asynchronous data streams
- An event-based model where data is pushed to the consumer, as it becomes available

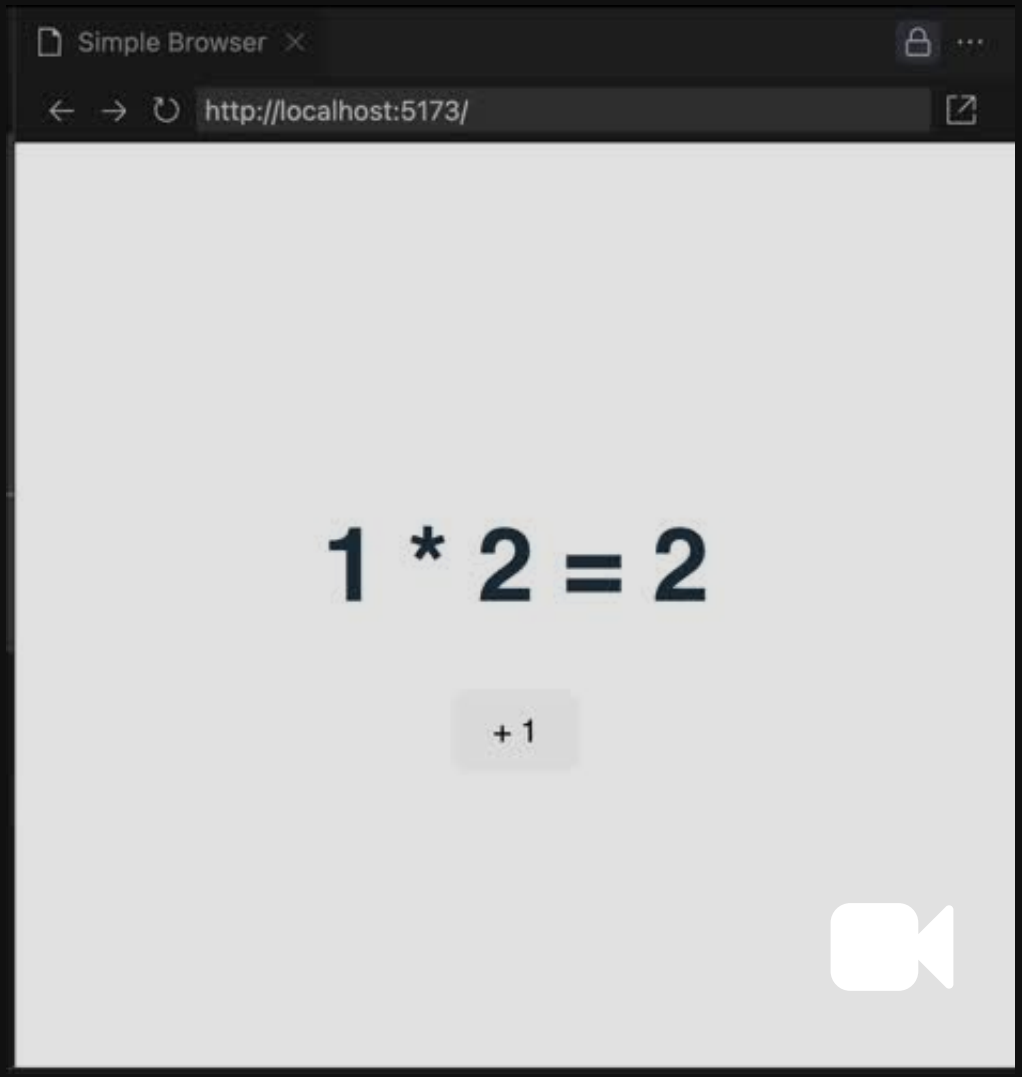
Reactive

- We have a value
- When that value changes...
- Anything "using" that value also changes ...**automatically**



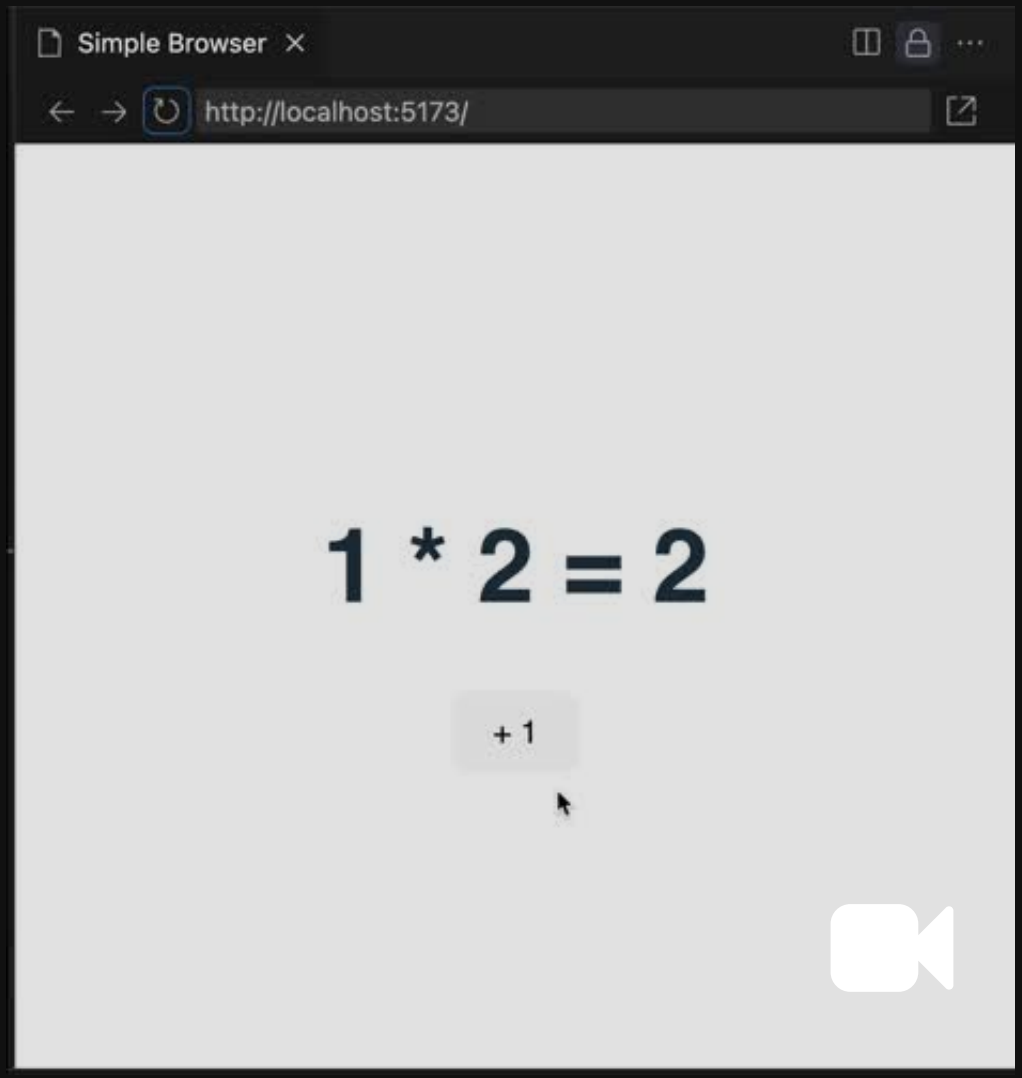
An Exploration of Reactvity in JavaScript

```
const button = document.querySelector('button');
const output = document.querySelector('output');

let count = 1;
let double = count * 2;

button.addEventListener('click', () => {
  count++;
});

const updateUI = () => {
  output.innerHTML = `
    <h1>${count} * 2 = ${double}</h1>
  `;
};
updateUI();
```


Simple Browser
http://localhost:5173/
1 * 2 = 2
+ 1

```
const button = document.querySelector('button');
const output = document.querySelector('output');

let count = 1;
let double = count * 2;

button.addEventListener('click', () => {
  count++;
});

const updateUI = () => {
  output.innerHTML = `
    <h1>${count} * 2 = ${double}</h1>
  `;
};
updateUI();
```

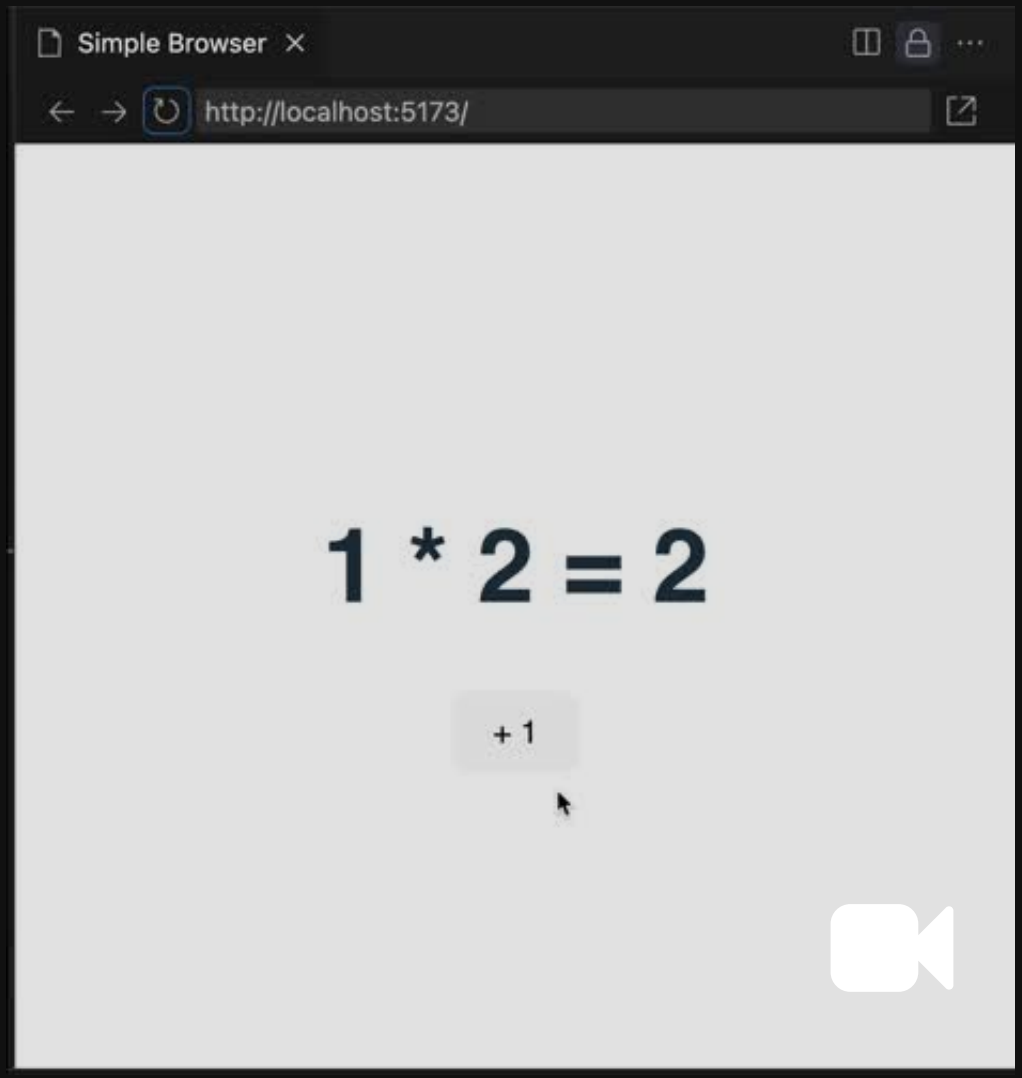
```
const button = document.querySelector('button');
const output = document.querySelector('output');

let count = 1;
let double = count * 2;

button.addEventListener('click', () => {
  count++;
  double = count * 2;
  updateUI();
});

const updateUI = () => {
  output.innerHTML = `
    <h1>${count} * 2 = ${double}</h1>
  `;
};
updateUI();
```


Simple Browser
http://localhost:5173/
1 * 2 = 2
+ 1

```
export const Observable = (fn) => ({ subscribe: fn });

export const observe = (element, event) => {
  return Observable((observer) => {
    element.addEventListener(event, observer);
    return () => {
      element.removeEventListener(event, observer);
    };
  });
};
```

```
export const map = (observable, fn) => {
  return Observable((observer) => {
    return observable.subscribe((value) => {
      observer(fn(value));
    });
  });
};

export const scan = (observable, fn, initialState) => {
  let state = initialState;
  let unsub;
  const observers = new Set();
  return Observable((observer) => {
    observers.add(observer);
    observer(state);
    if (observers.size === 1) {
      unsub = observable.subscribe((value) => {
        state = fn(state, value);
        observers.forEach((obs) => obs(state));
      });
    }
    return () => {
```

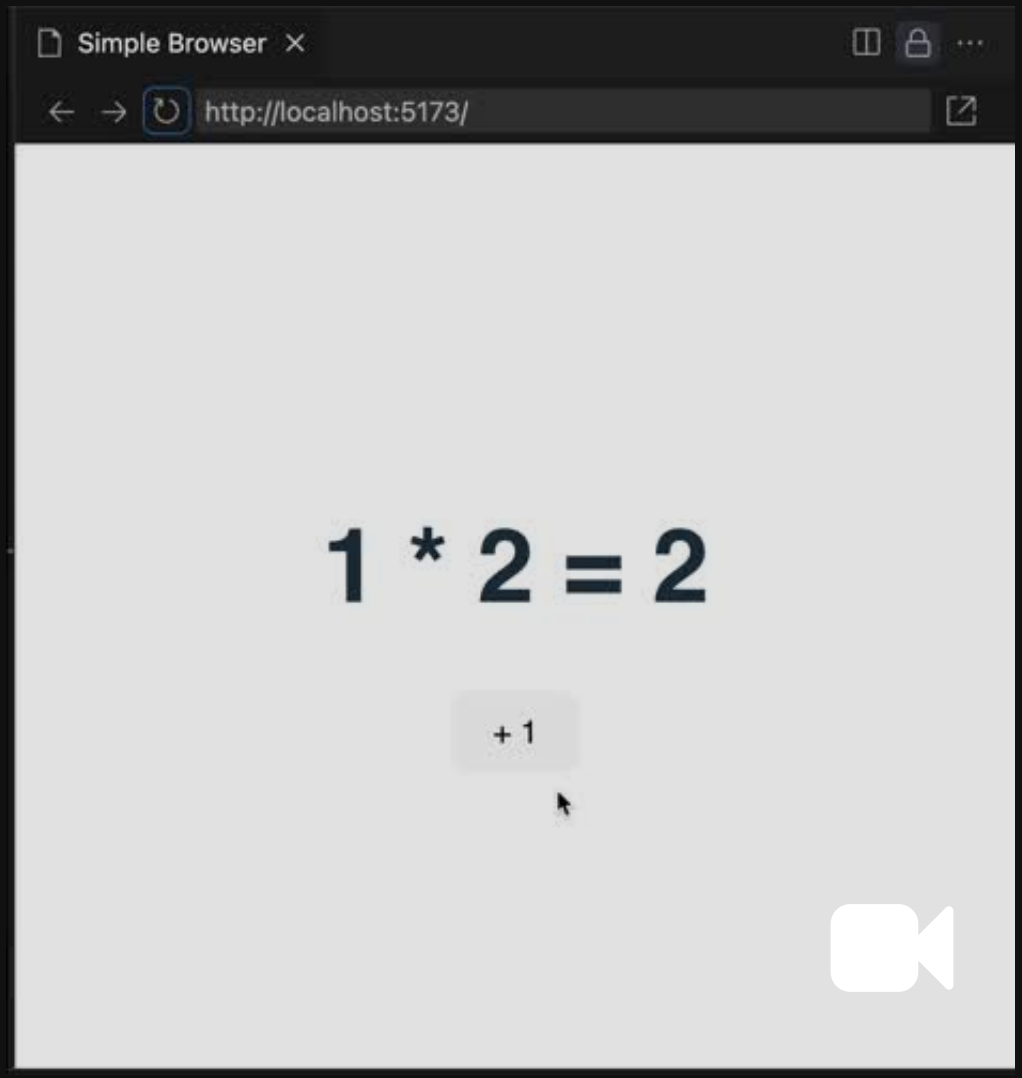
```
import { observe, scan, map, combine } from './observable';

const button = document.querySelector('button');
const output = document.querySelector('output');

const clicks = observe(button, 'click');

const count = scan(clicks, (v) => v + 1, 1);
const double = map(count, (n) => n * 2);

combine(count, double).subscribe(([n, dbl]) => {
  output.innerHTML = `
    <h1>${n} * 2 = ${dbl}</h1>
  `;
});
```


Simple Browser
http://localhost:5173/
1 * 2 = 2
+ 1

```
import { fromEvent, startWith, scan, map, zip } from 'rxjs';

const button = document.querySelector('button');
const output = document.querySelector('output');

const clicks = fromEvent(button, 'click');

const count = clicks.pipe(
  startWith(1),
  scan((count) => count + 1)
);
const double = count.pipe(map((count) => count * 2));

zip(count, double).subscribe(([count, double]) => {
  output.innerHTML = `
    <h1>${count} * 2 = ${double}</h1>
  `;
});
```

```
export const createStore = (state) => {
  let listeners = new Set();

  return {
    get: () => state,
    set: (updater = (v) => v) => {
      state = updater(state);
      listeners.forEach((listener) => listener(state));
    },
    subscribe: (listener) => {
      listeners.add(listener);
      listener(state);
      return () => {
        listeners.delete(listener);
      };
    },
  };
};
```

```
import { createStore, derived } from './store.js';

const button = document.querySelector('button');
const output = document.querySelector('output');

const count = createStore(1);
const double = derived([count], ([count]) => count * 2);

button.addEventListener('click', () =>
  count.set((count) => count + 1)
);

derived([count, double]).subscribe(([count, double]) => {
  output.innerHTML = `
    <h1>${count} * 2 = ${double}</h1>
  `;
});
```


Simple Browser
http://localhost:5173/
1 * 2 = 2
+ 1

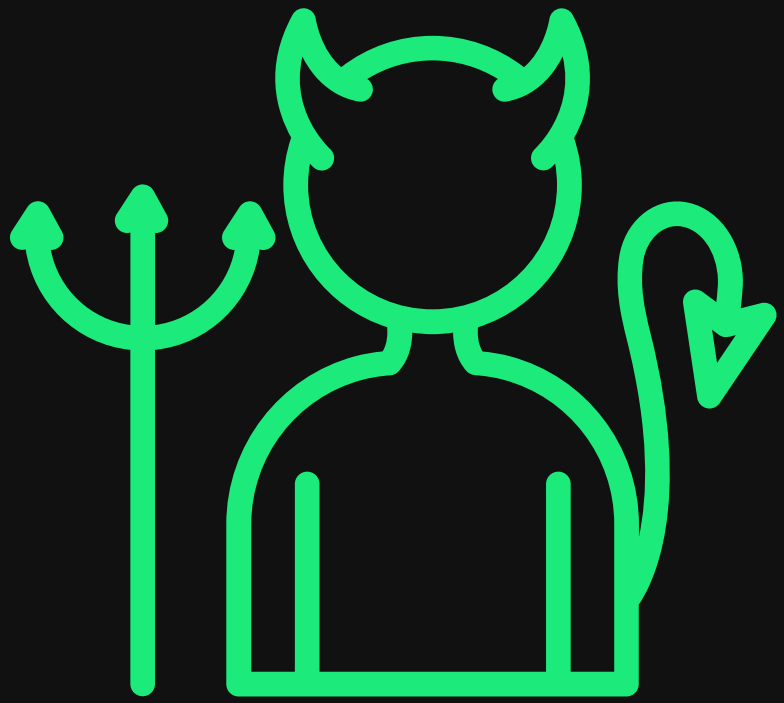
```
import { createStore, derived } from './store.js';

const button = document.querySelector('button');
const output = document.querySelector('output');

const count = createStore(1);
const double = derived([count], ([count]) => count * 2);

button.addEventListener('click', () =>
  count.set((count) => count + 1)
);

derived([count, double]).subscribe(([count, double]) => {
  output.innerHTML = `
    <h1>${count} * 2 = ${double}</h1>
  `;
});
```

```
import { useSyncExternalStore } from 'react';
import { createStore, derived } from './store.js';

const count = createStore(1);
const double = derived([count], ([count]) => count * 2);

const syncExternalStore = derived([count, double]);
const handleClick = () => count.set((v) => v + 1);

export default function App() {
  const [cnt, dbl] = useSyncExternalStore(
    syncExternalStore.subscribe,
    syncExternalStore.get
  );
  return (
    <>
      <output>
        <h1>
          {cnt} * 2 = {dbl}
        </h1>
      </output>
      <button onClick={handleClick}>+1</button>
```

```svelte
<script>
  import { writable, derived } from 'svelte/store';

  const count = writable(1);
  const double = derived(count, (n) => n * 2);

  const handleClick = () => $count += 1;
</script>

<output>
    <h1>
        {$count} * 2 = {$double}
    </h1>
</output>
<button on:click={handleClick}>+1</button>
```

```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => count.value++);

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

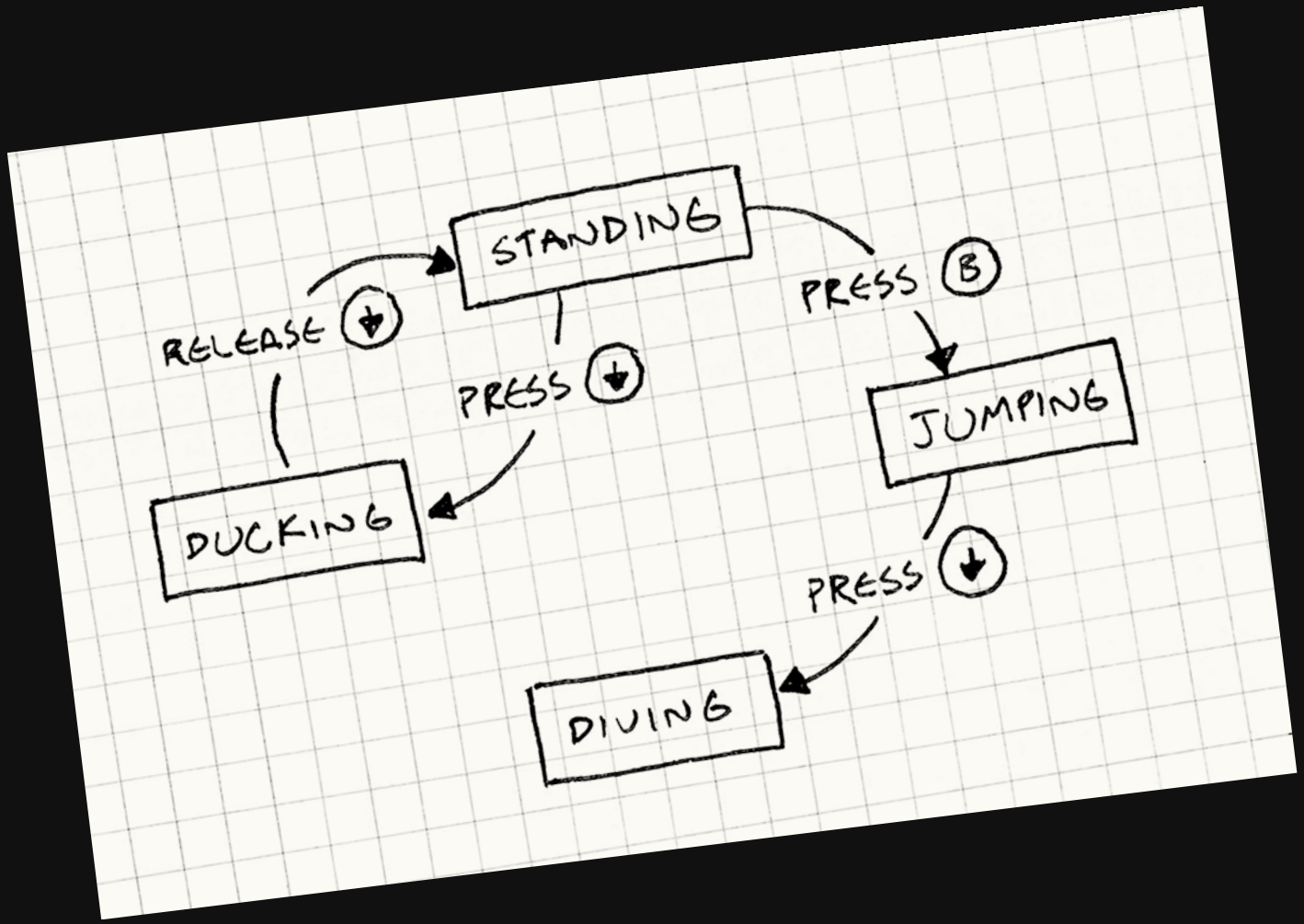
```jsx
import { useSignal, useComputed } from '@preact/signals-react'

export default function App() {
  const count = useSignal(0);
  const double = useComputed(() => count.value * 2);

  const handleClick = () => count.value++;
  return (
    <>
      <output>
        <h1>
          {count} * 2 = {double}
        </h1>
      </output>
      <button onClick={handleClick}>+1</button>
    </>
  );
}
```



So what do we mean by "reactive"?


A not-so-new, poorly
reactive state primitive
primitive

State


STANDING
RELEASE
PRESS
PRESS
B
DUCKING
JUMPING
PRESS
DIVING

Primitive


A not-so-new, new, poorly
named, reactive state
primitive


canjs
JavaScript MVC
Home
Download / Install
Tutorials
Live examples
Documentation
Simplify dynamic JavaScript UIs with the Model-View-View Model (MVVM) pattern
Download
v3.5.1 - 25kb min+gz
release notes
Knockout.
Key concepts
Declarative Bindings
Easily associate DOM elements with model data using a concise, readable syntax
Automatic UI Refresh
When your data model's state changes, your UI updates automatically
Dependency Tracking
Implicitly set up chains of relationships between model data, to transform and combine it
Templating
Quickly generate sophisticated, nested UIs as a function of your model data
New: Interactive tutorials
Get started with knockout.js quickly, learning to build single-page applications, custom bindings and more with these interactive tutorials.

Framework	since	Reactive State	Derived Values	Side Effect
Knockout	2010	observable	pureComputed	computed
CanJS	2012	observable	compute	compute.on()
Svelte	2016	let variable = ?	$: (Reactive Values)	$: (reactive statements)
Solid	2018	createSignal	createMemo	createEffect
Vue	2020	ref/reactive	computed	watch/watchEffect
Preact	2022	signal	computed	effect
Qwik	2022	useSignal/useStore	useComputed$ / useResource$	useTask$
Angular	2023	signal	computed	effect
Svelte 5	2024	$state	$derived	$effect

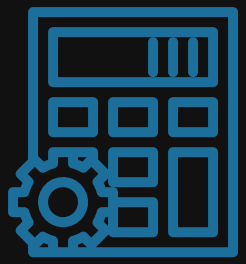
What are JS Signals?

What are JS Signals

```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => count.value++);

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

JS Signals

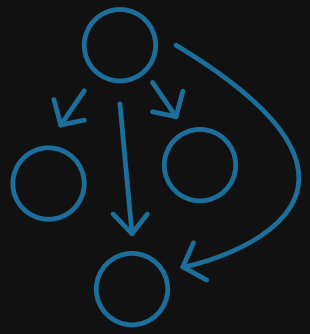
State



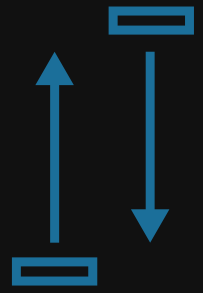
Derived

Effect

JS Signals



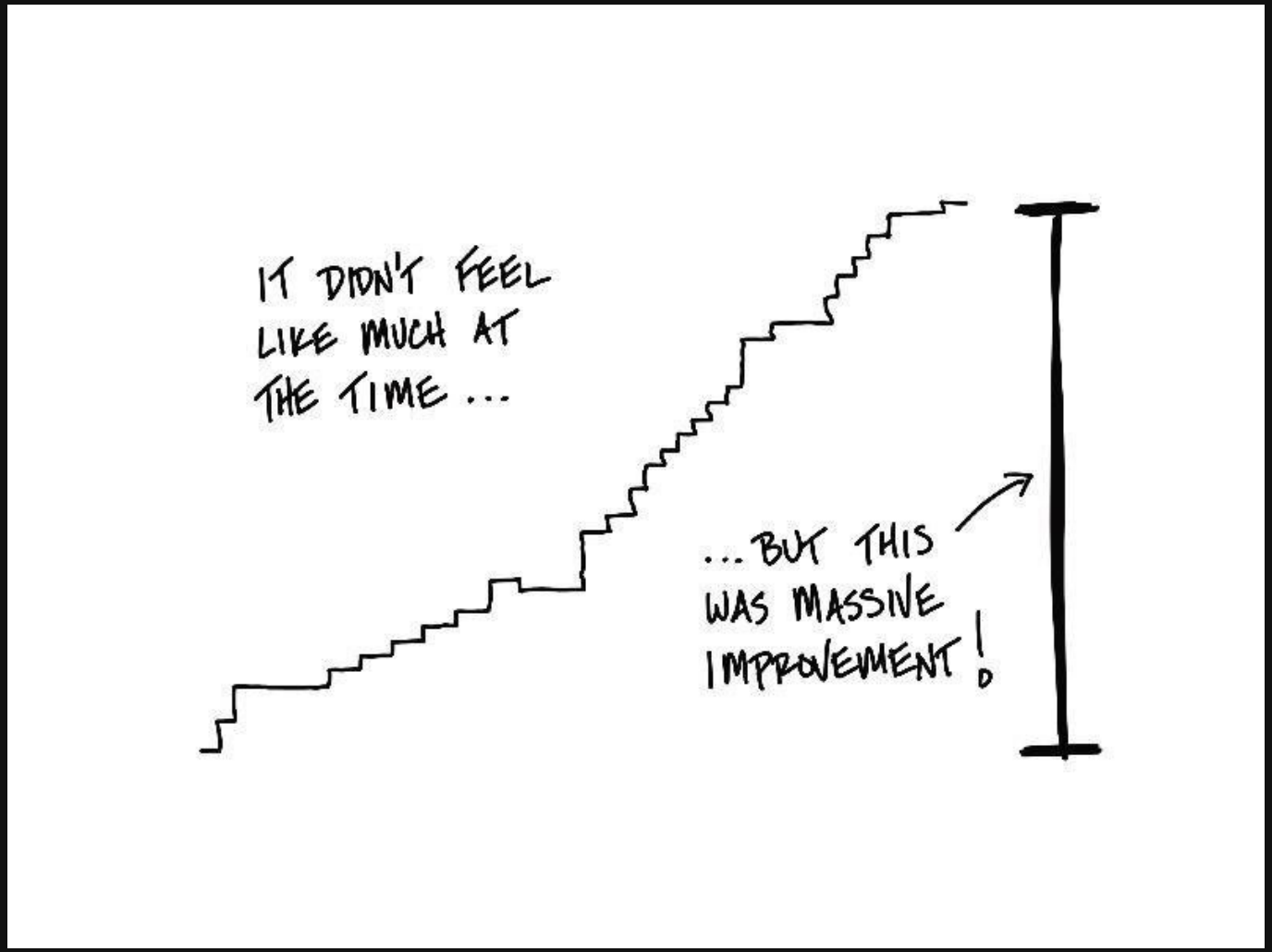
Acyclic



Push-then-Pull

Auto Tracking

Incremental Improvement




EventTarget
Observable
Stores
Signals


So how are they better?
(Better than stores)

JS Signals

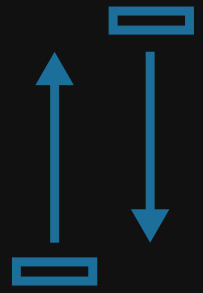
State

Derived

Effect

JS Signals

Acyclic



Push-then-Pull

Auto Tracking

Developer Experience

Auto Tracking

What are JS Signals

```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => {
  count.value++;
});

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

What are JS Signals

```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count() * 2);

button.addEventListener('click', () => {
  count.update((n) => n + 1)
});

effect(() => {
  output.innerHTML = `
    <h1>${count()} * 2 = ${double()}</h1>
  `;
});
```

What are JS Signals

```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.get() * 2);

button.addEventListener('click', () => {
  count.set(count.get() + 1)
});

effect(() => {
  output.innerHTML = `
    <h1>${count.get()} * 2 = ${double.get()}</h1>
  `;
});
```

```
<script>
  let count = $state(1);
  let double = $derived(count * 2);

  const handleClick = () => count++;
</script>

<output>
    <h1>
        {count} * 2 = {double}
    </h1>
</output>
<button onclick={handleClick}>+1</button>
```

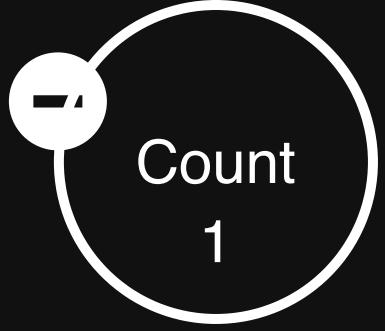
Performance

Push-then-Pull

What are JS Signals

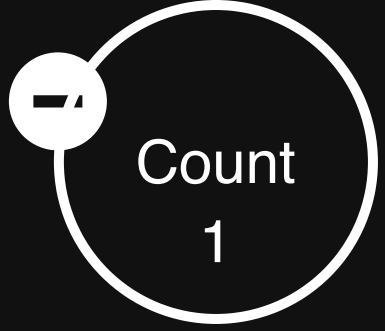
```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => {
  count.value++;
});

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

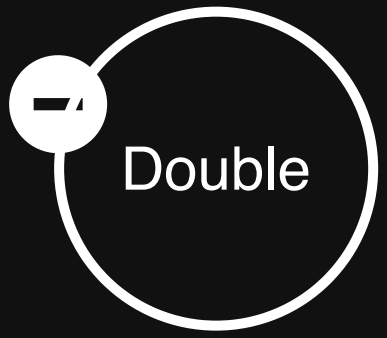

Count
1


Count
1

What are JS Signals

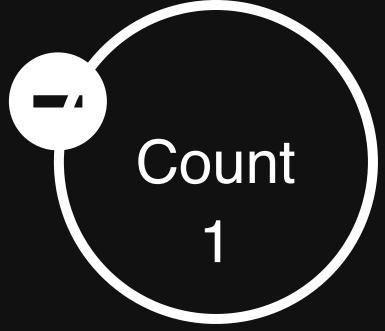
```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => {
  count.value++;
});

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

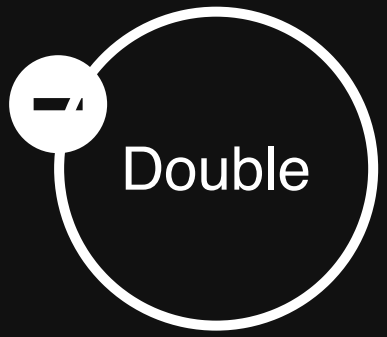

Count
1

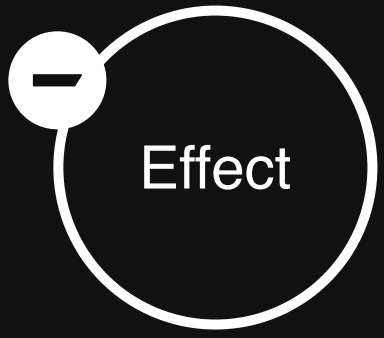

Double

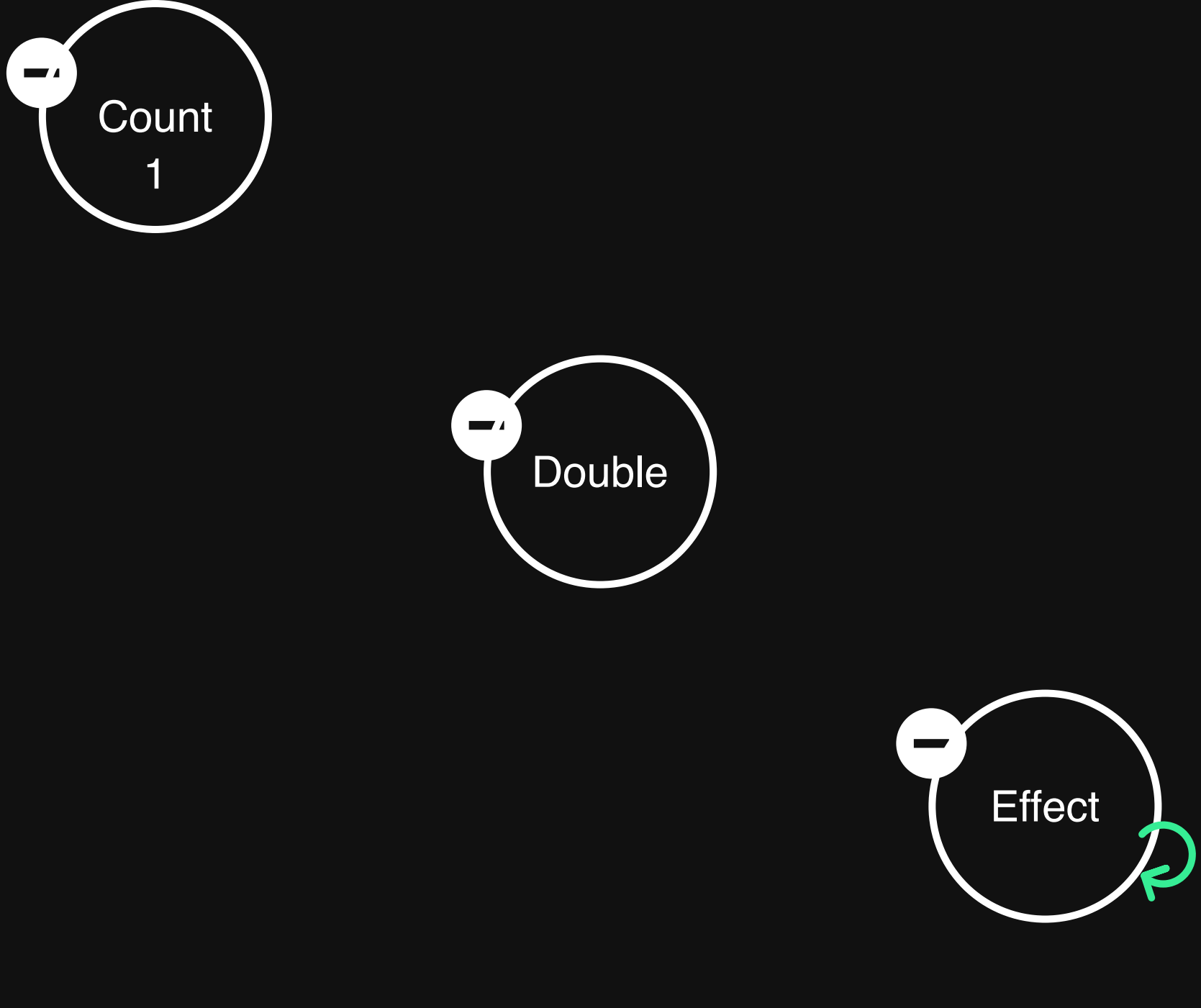
What are JS Signals

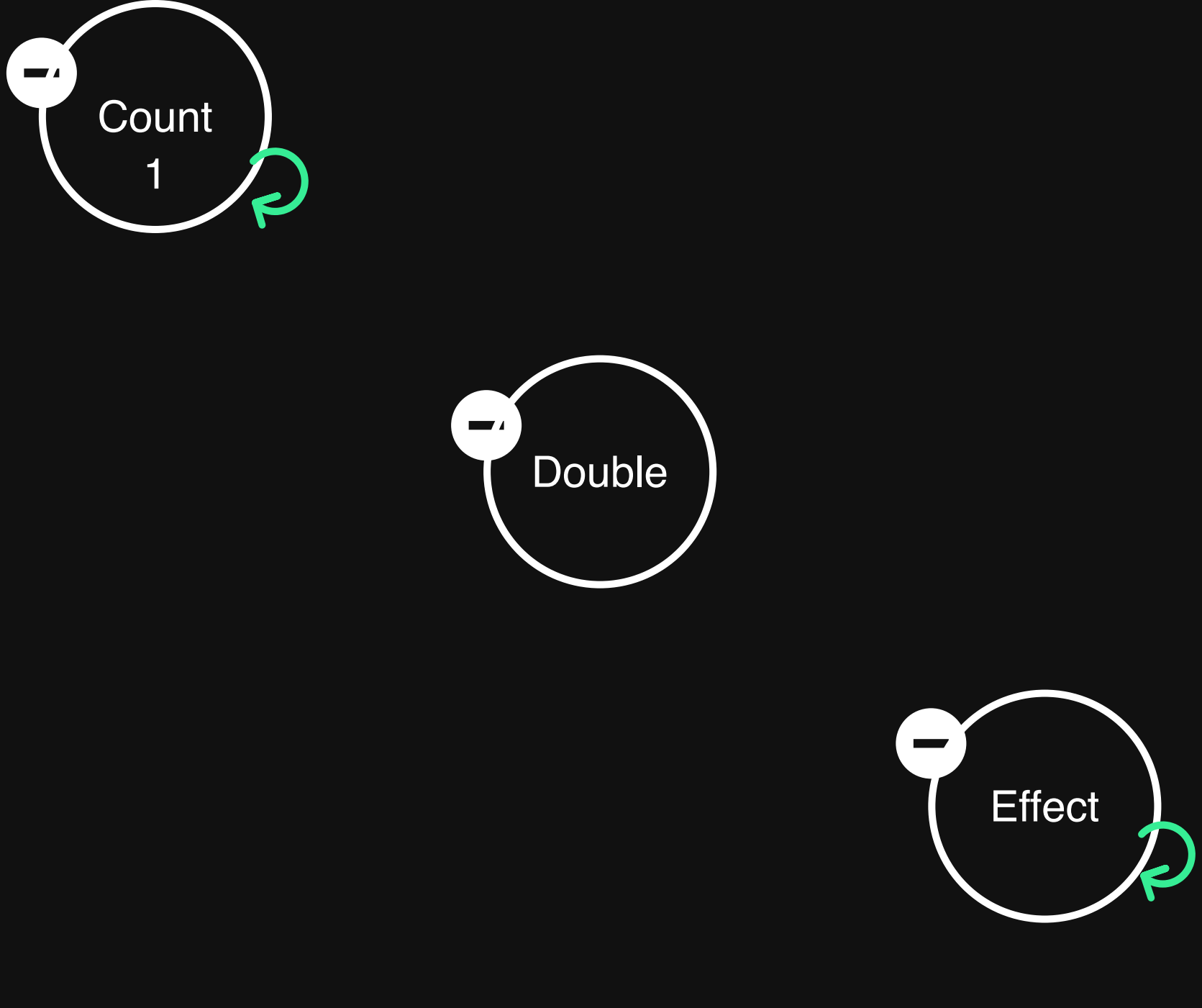
```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => {
  count.value++;
});

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

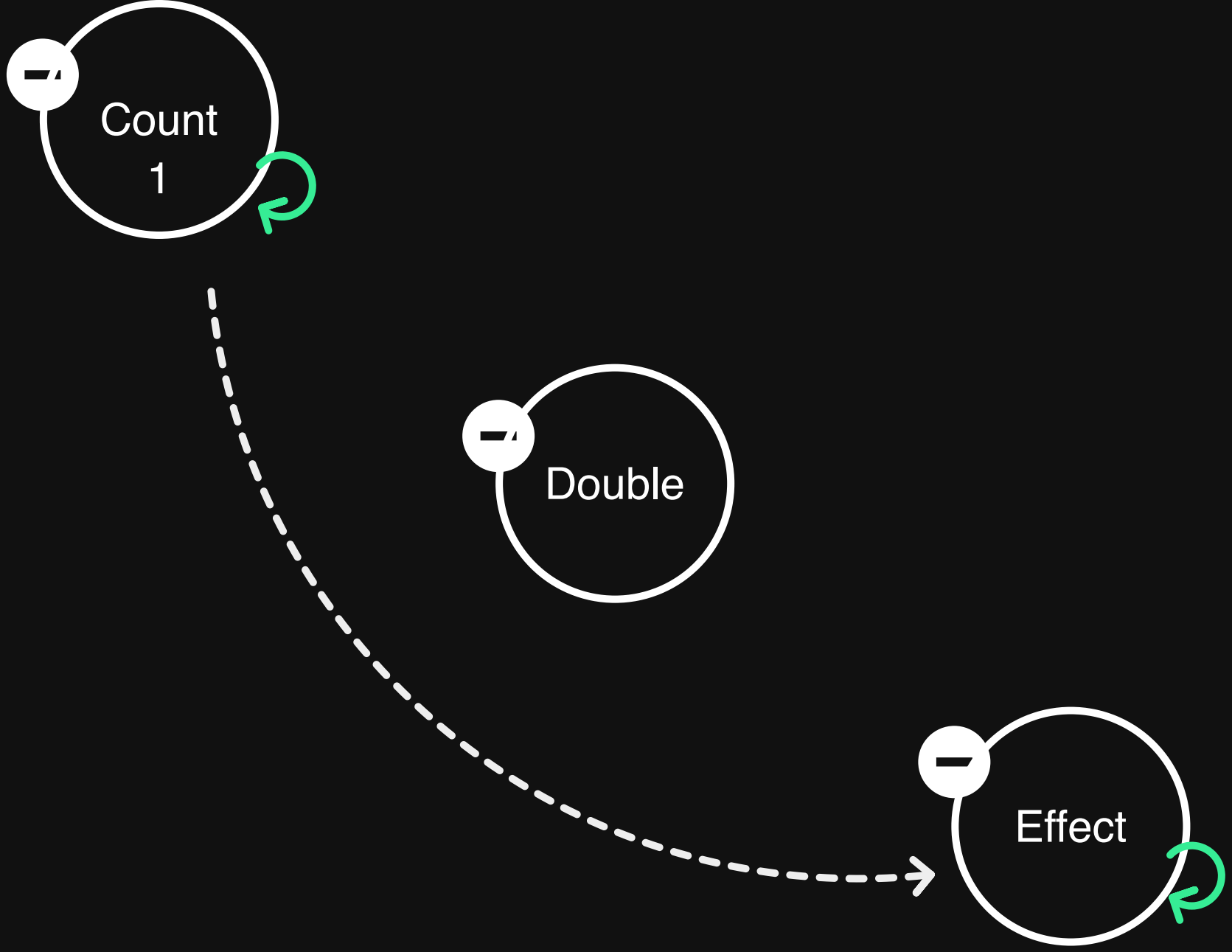

Count
1

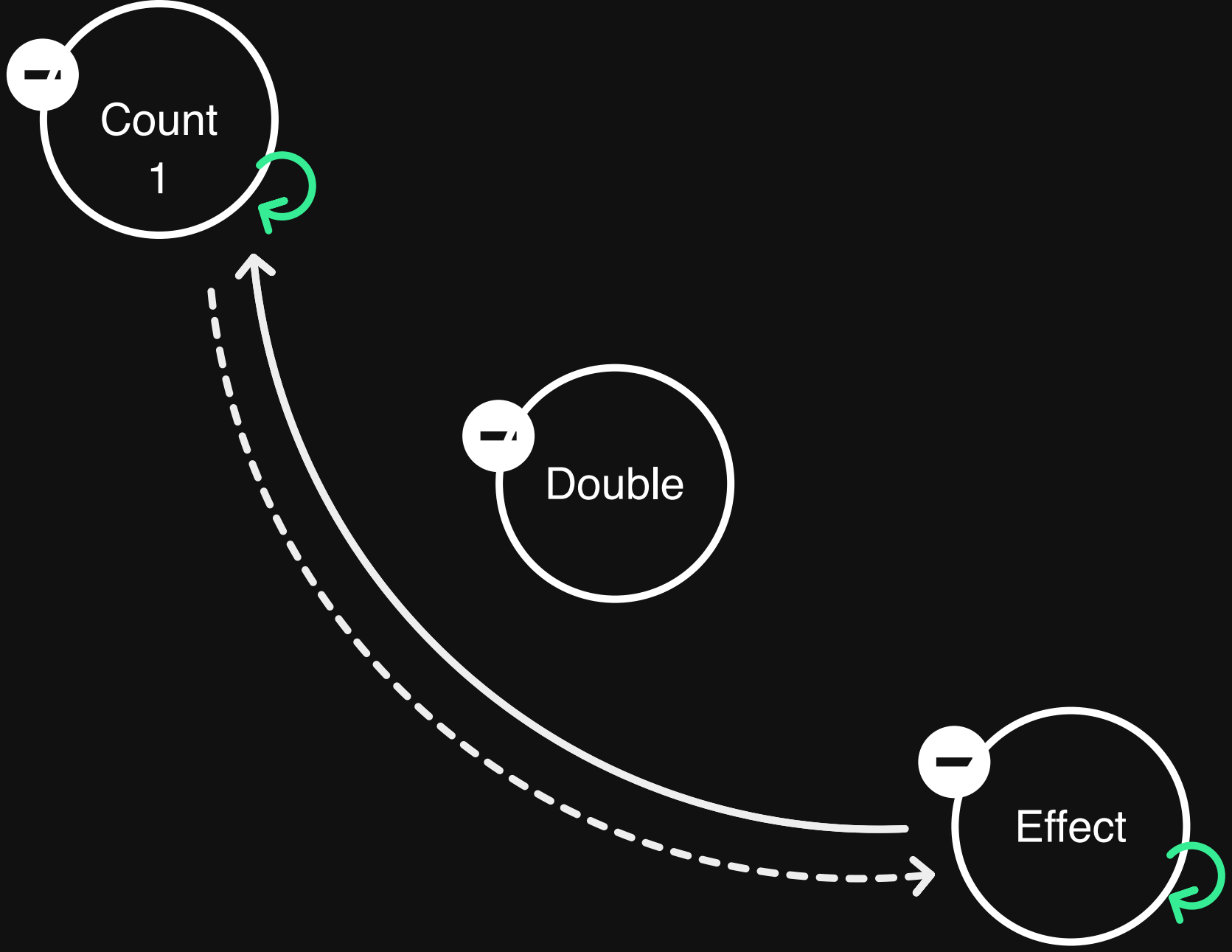

Double

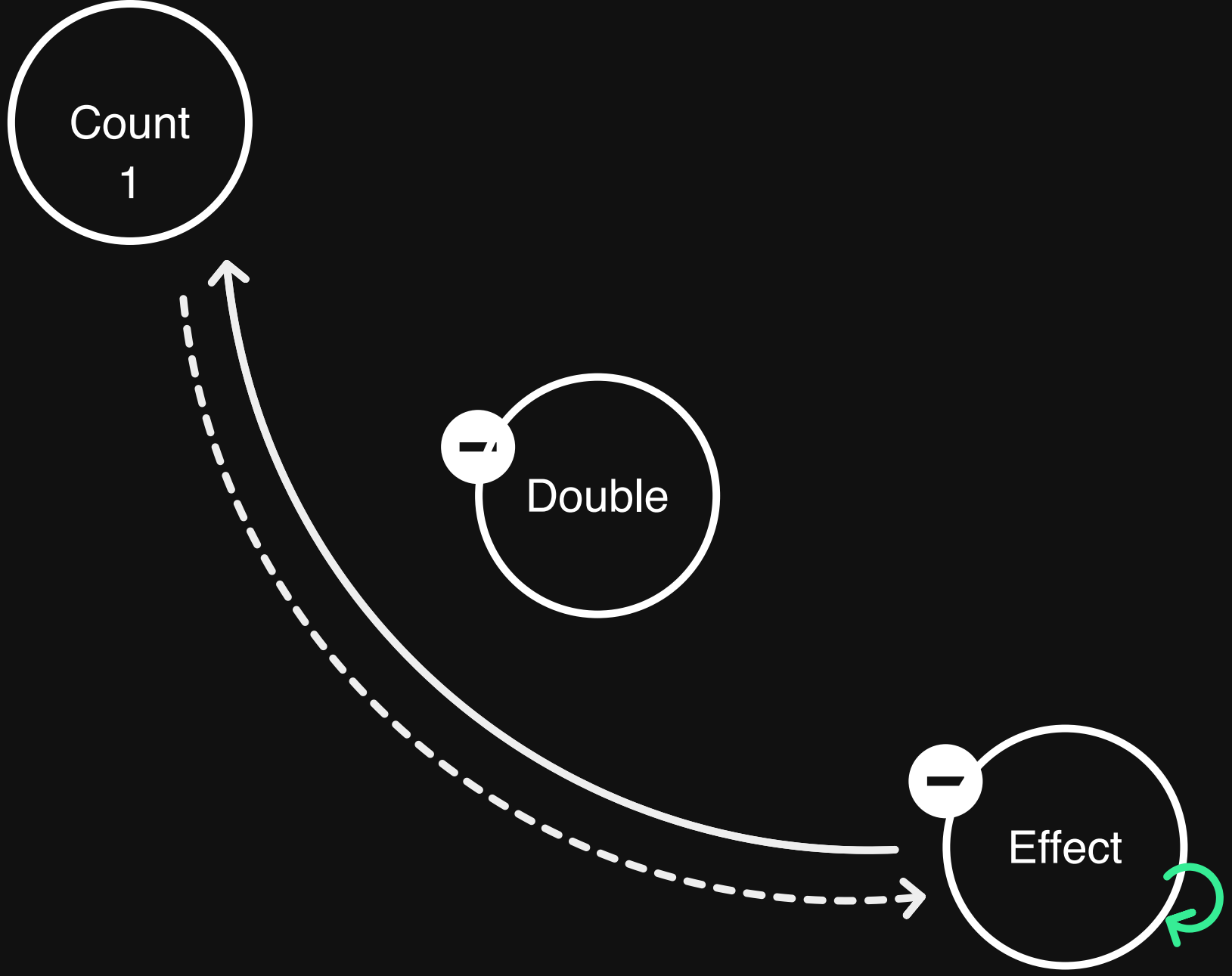

Effect

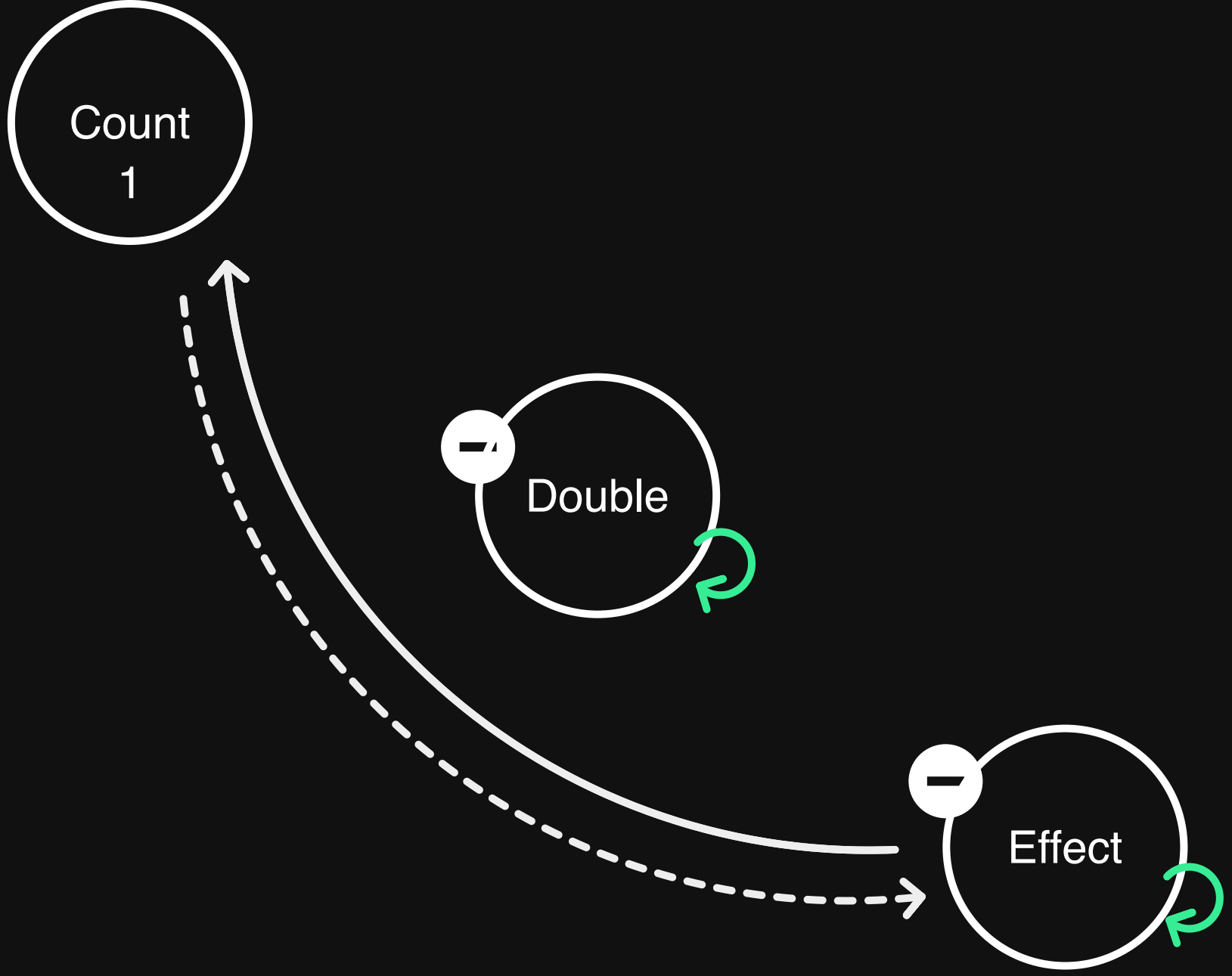

Count
1
Double
Effect

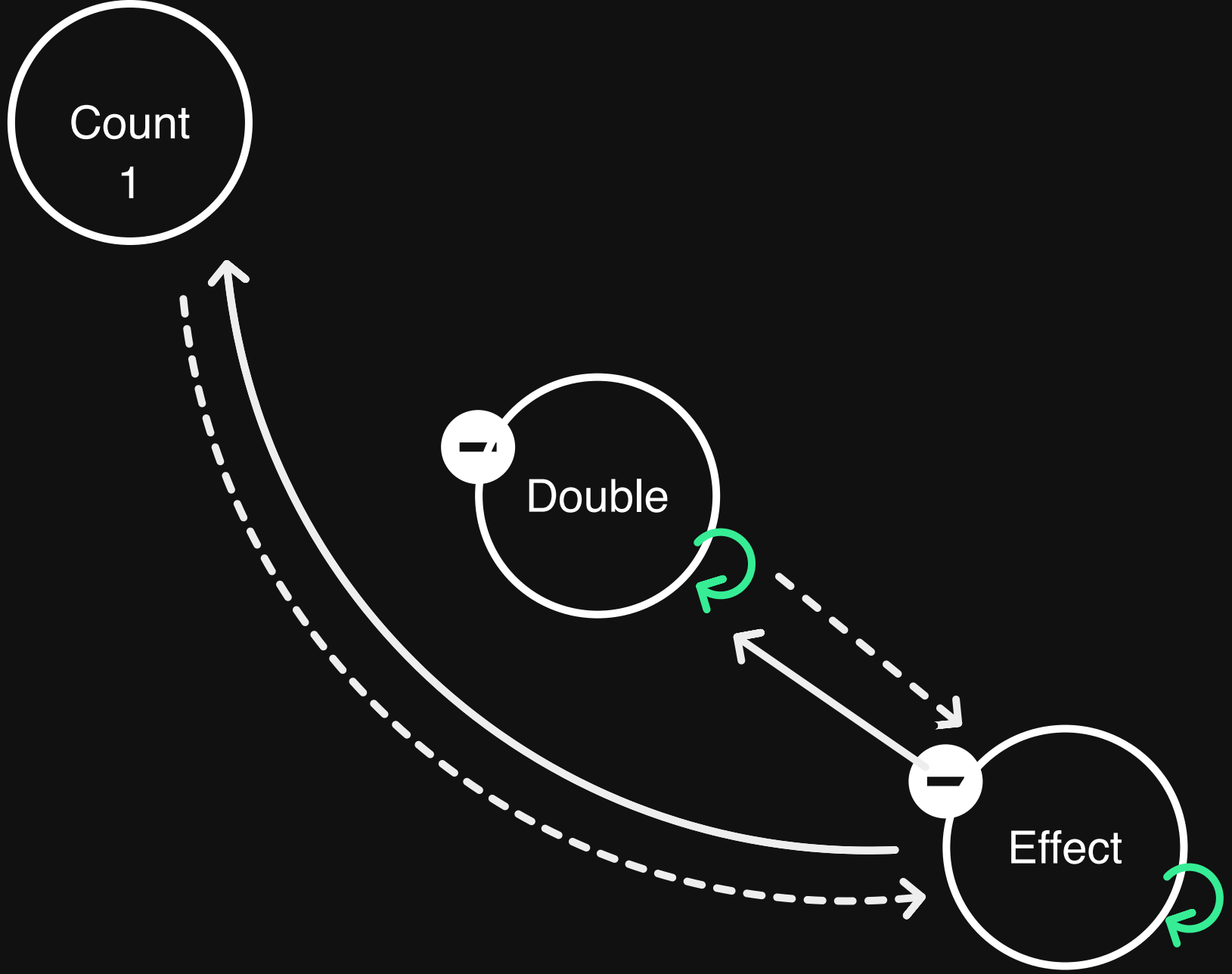

Count
1
Double
Effect

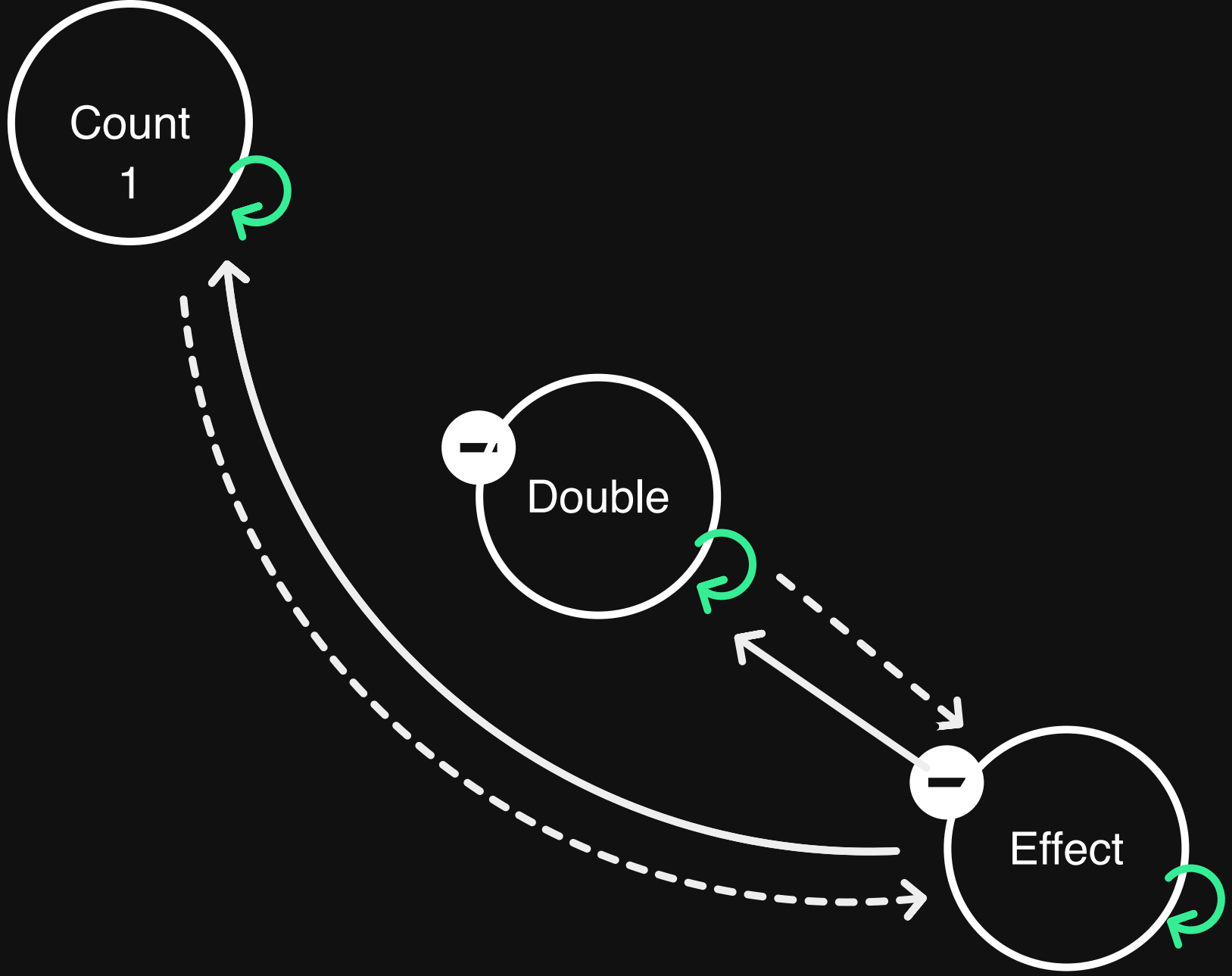

Count
1
Double
Effect

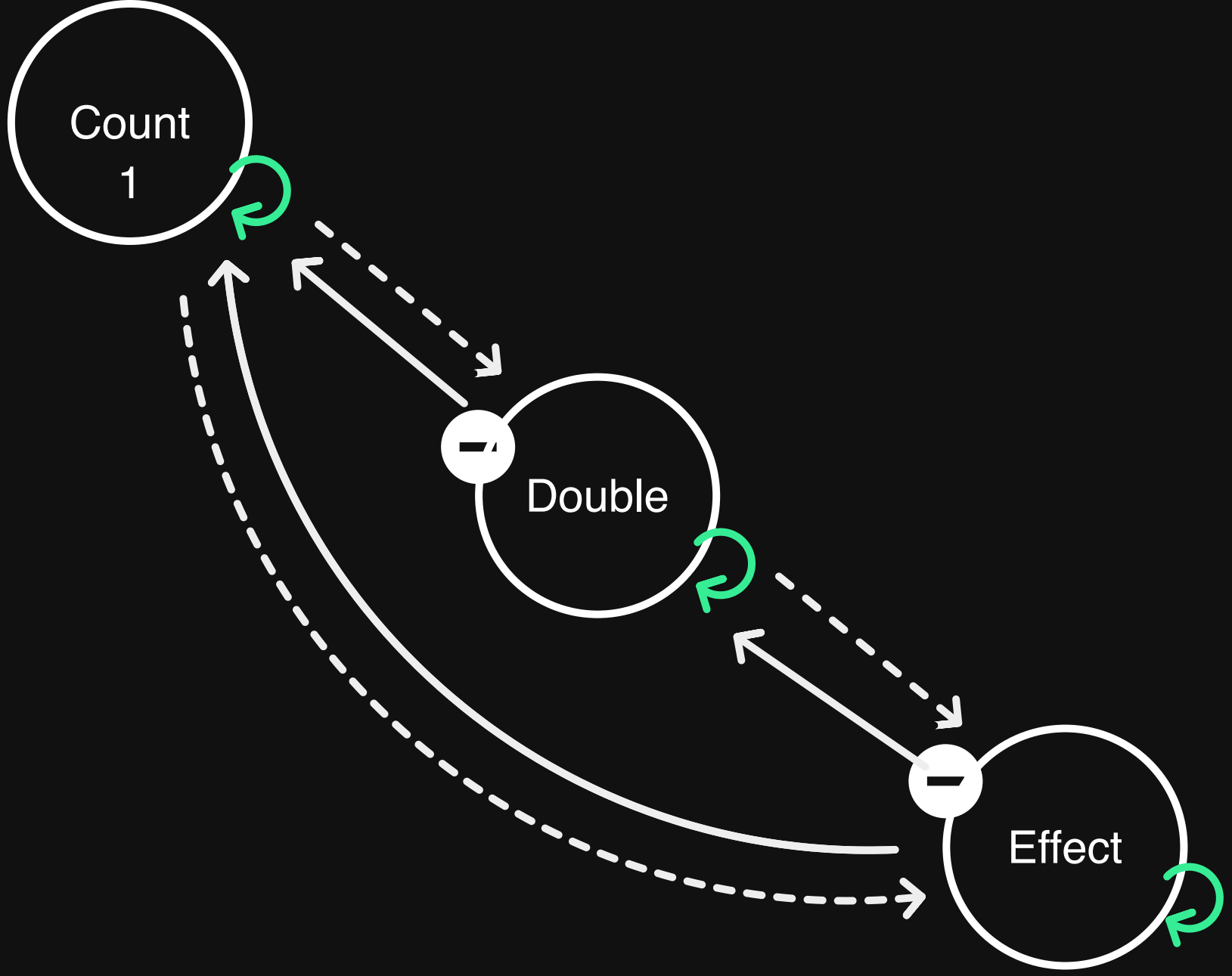

Count
1
Double
Effect

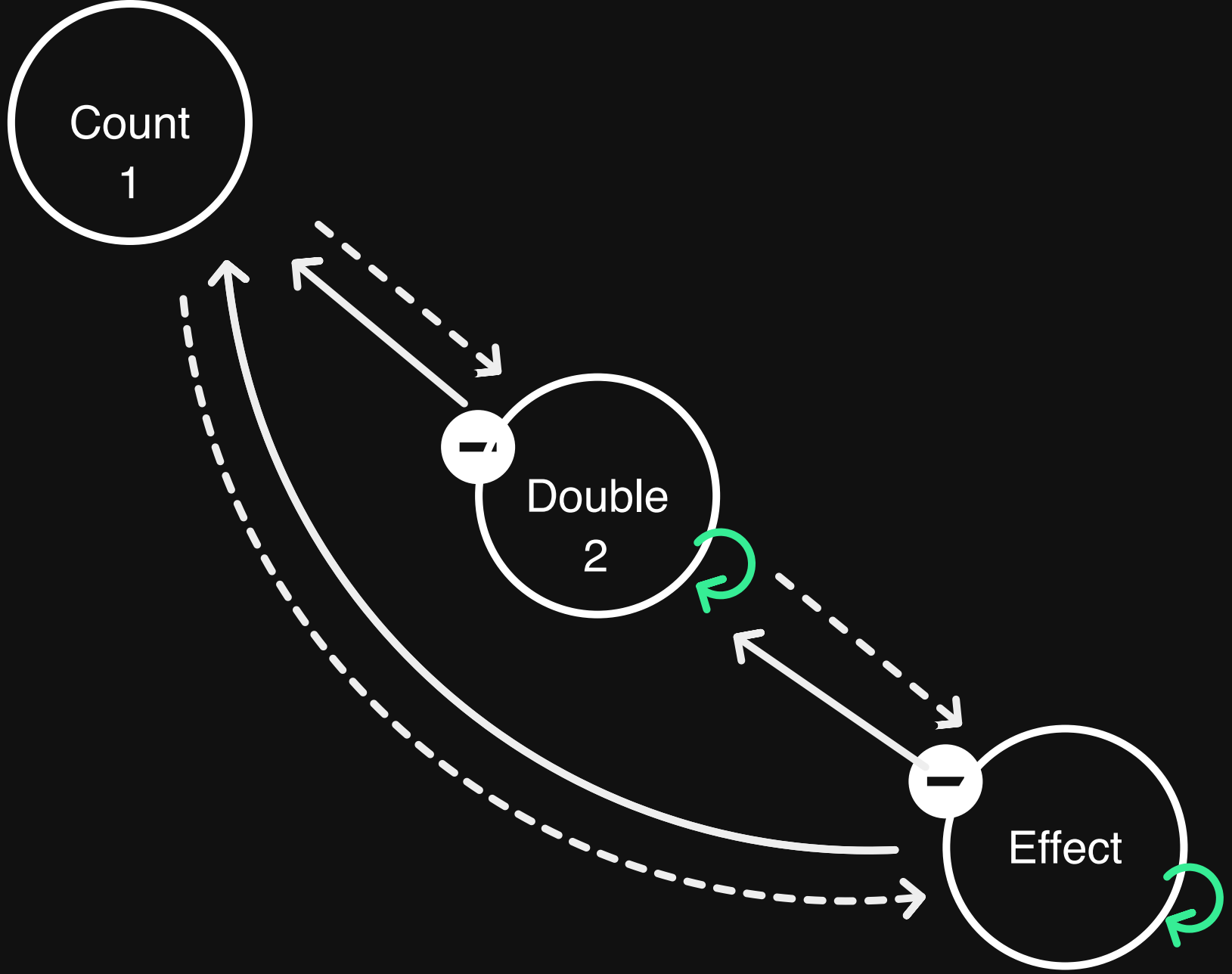

Count
1
Double
Effect

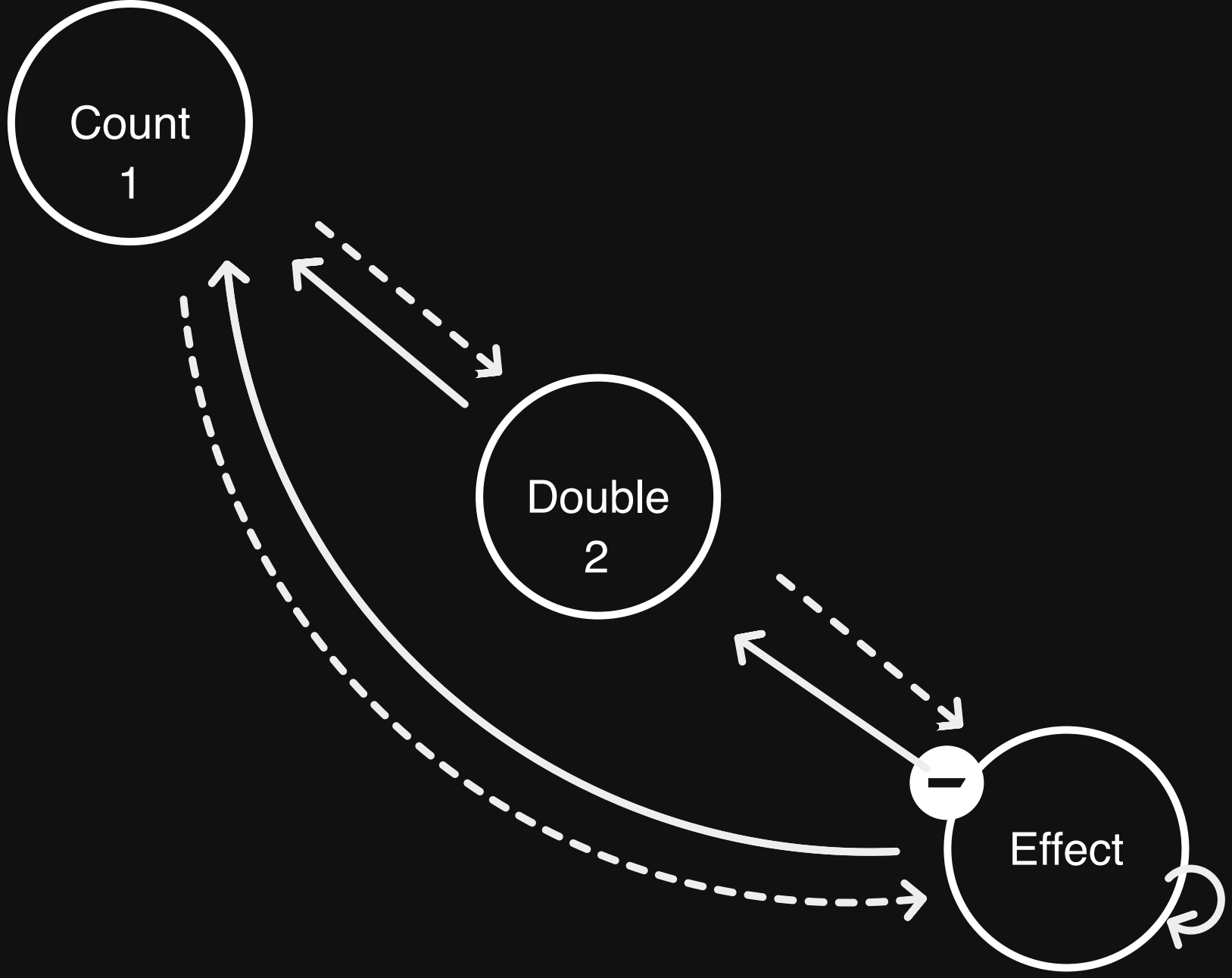

Count
1
Double
Effect

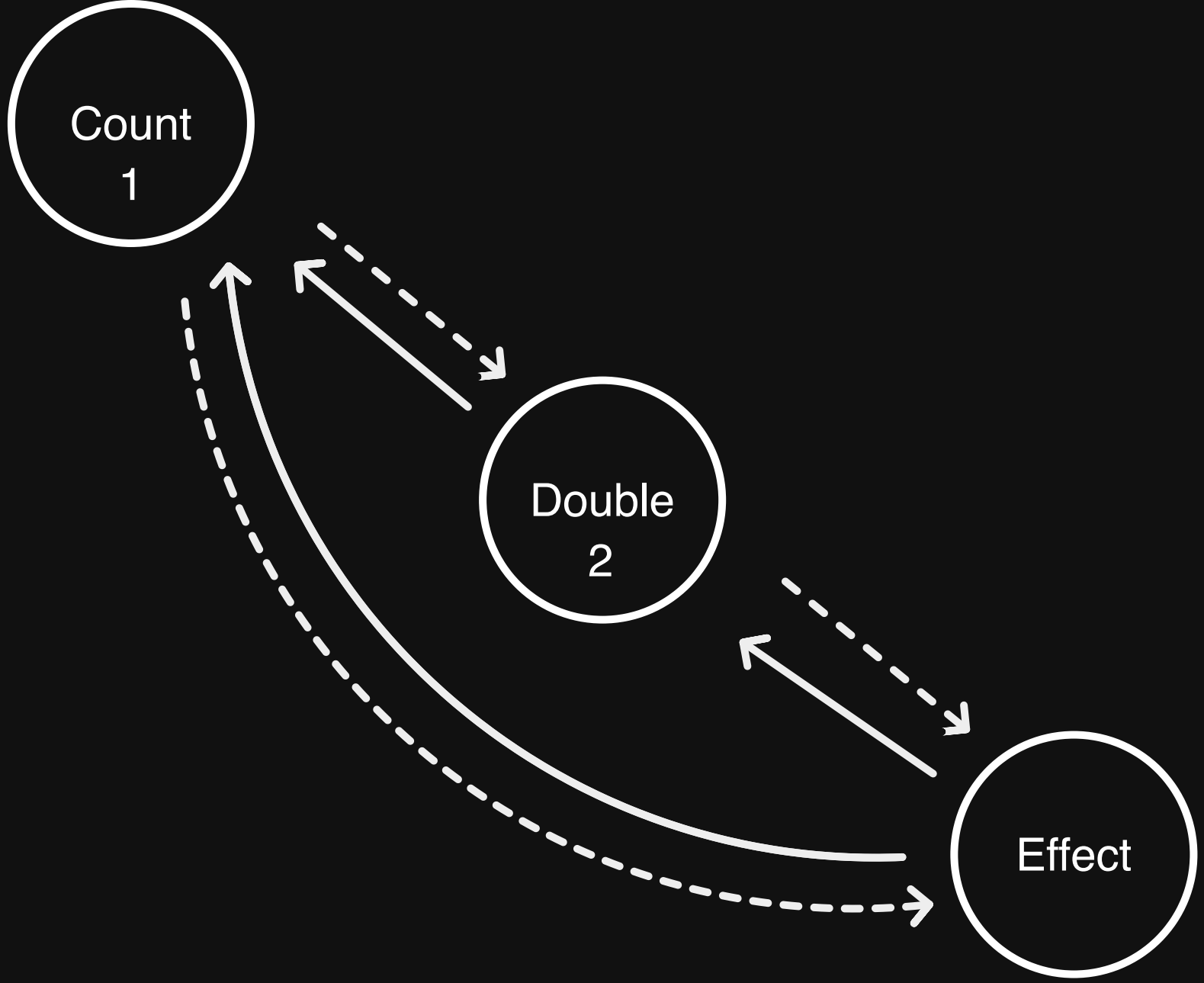

Count
1
Double
Effect


Count
1
Double
Effect


Count
1
Double
Effect

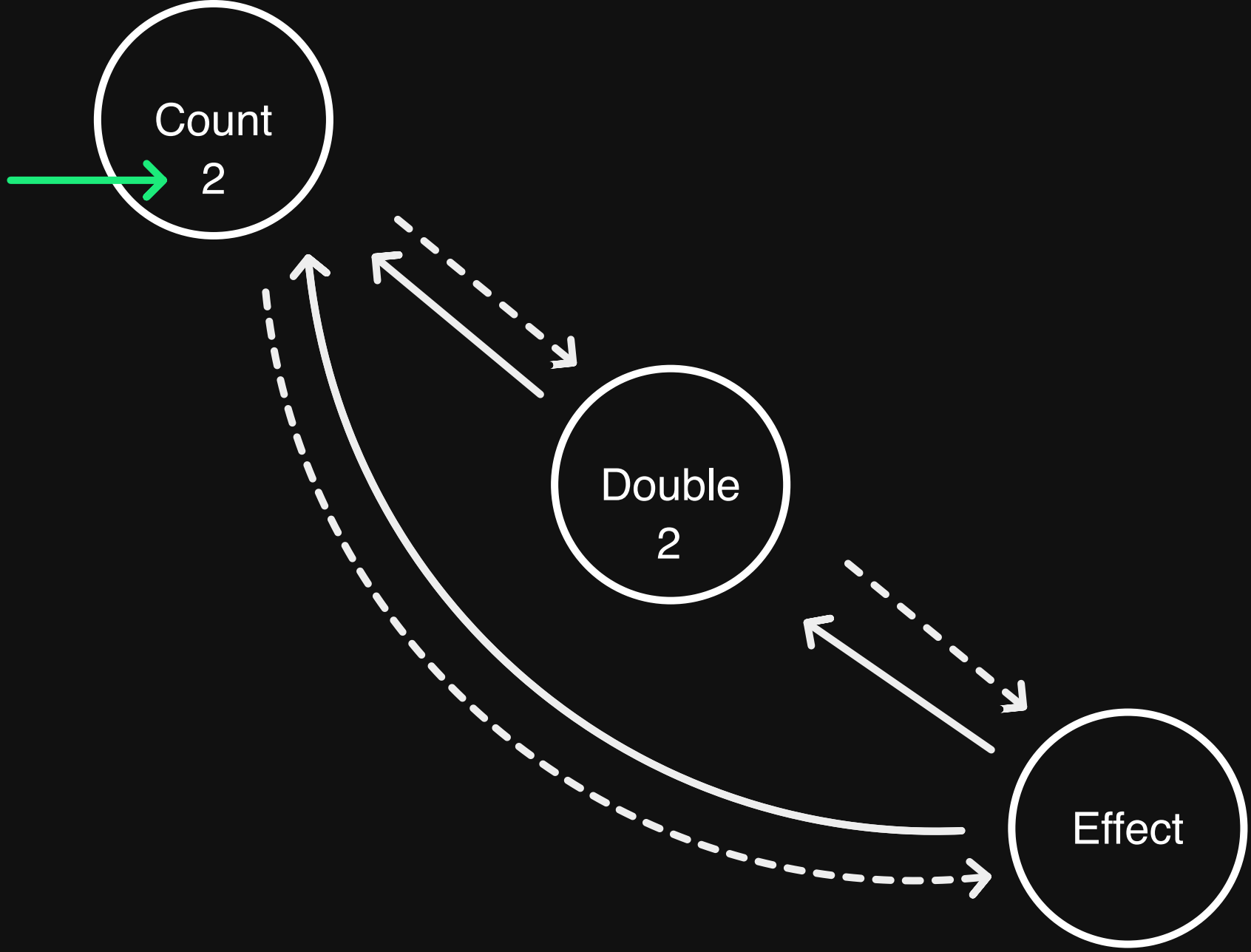

Count
1
Double
2
Effect

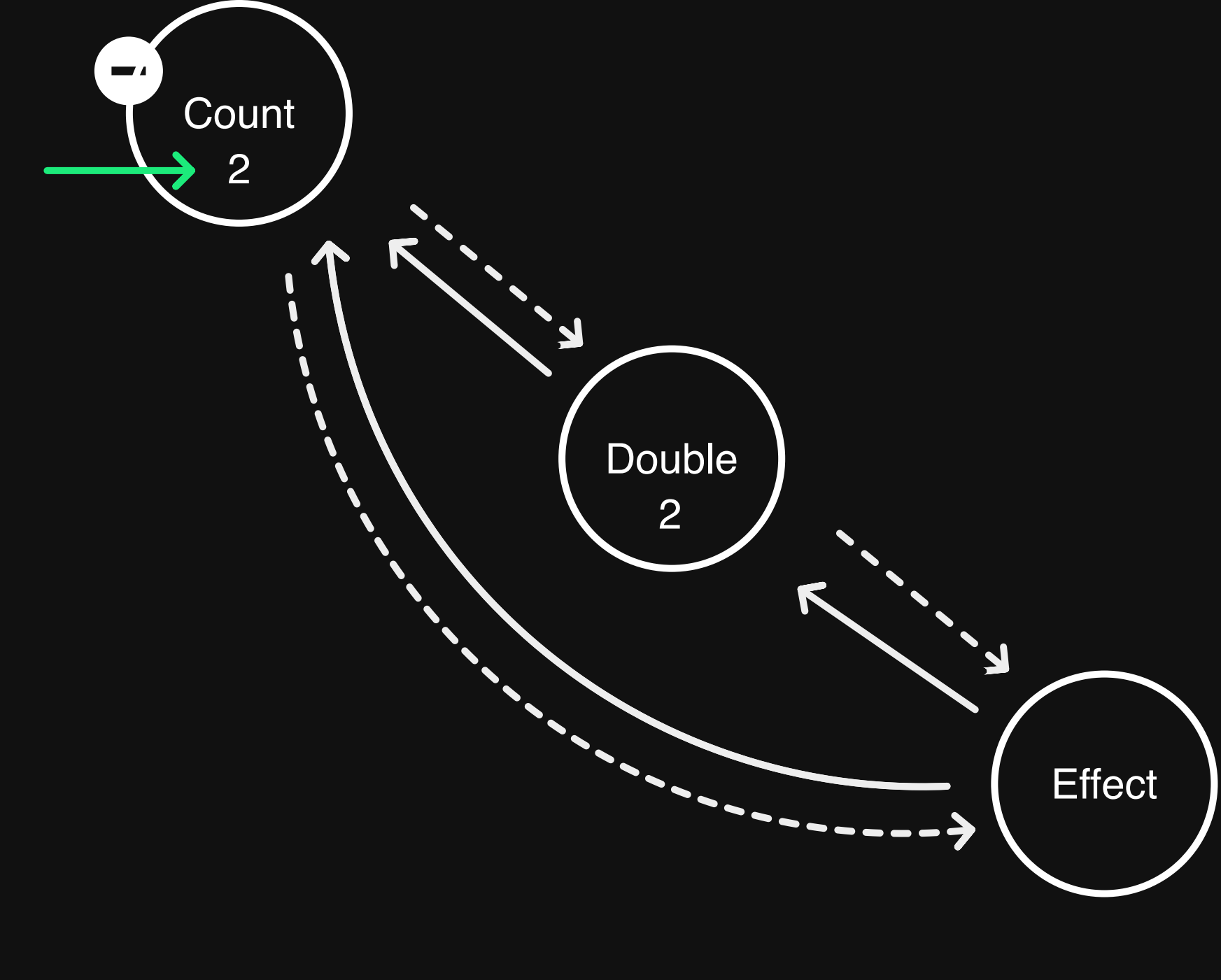

Count
1
Double
2
Effect

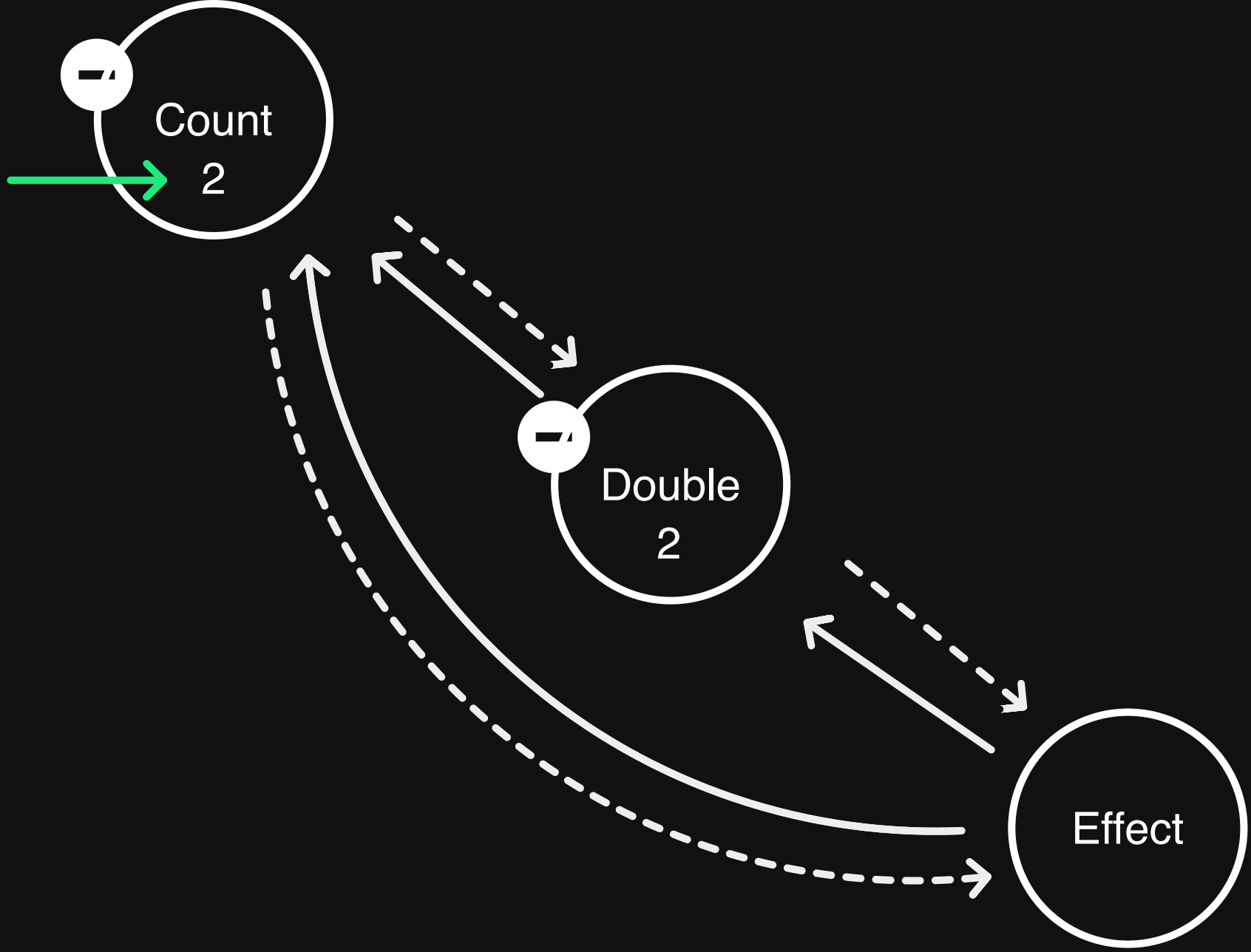

Count
1
Double
2
Effect

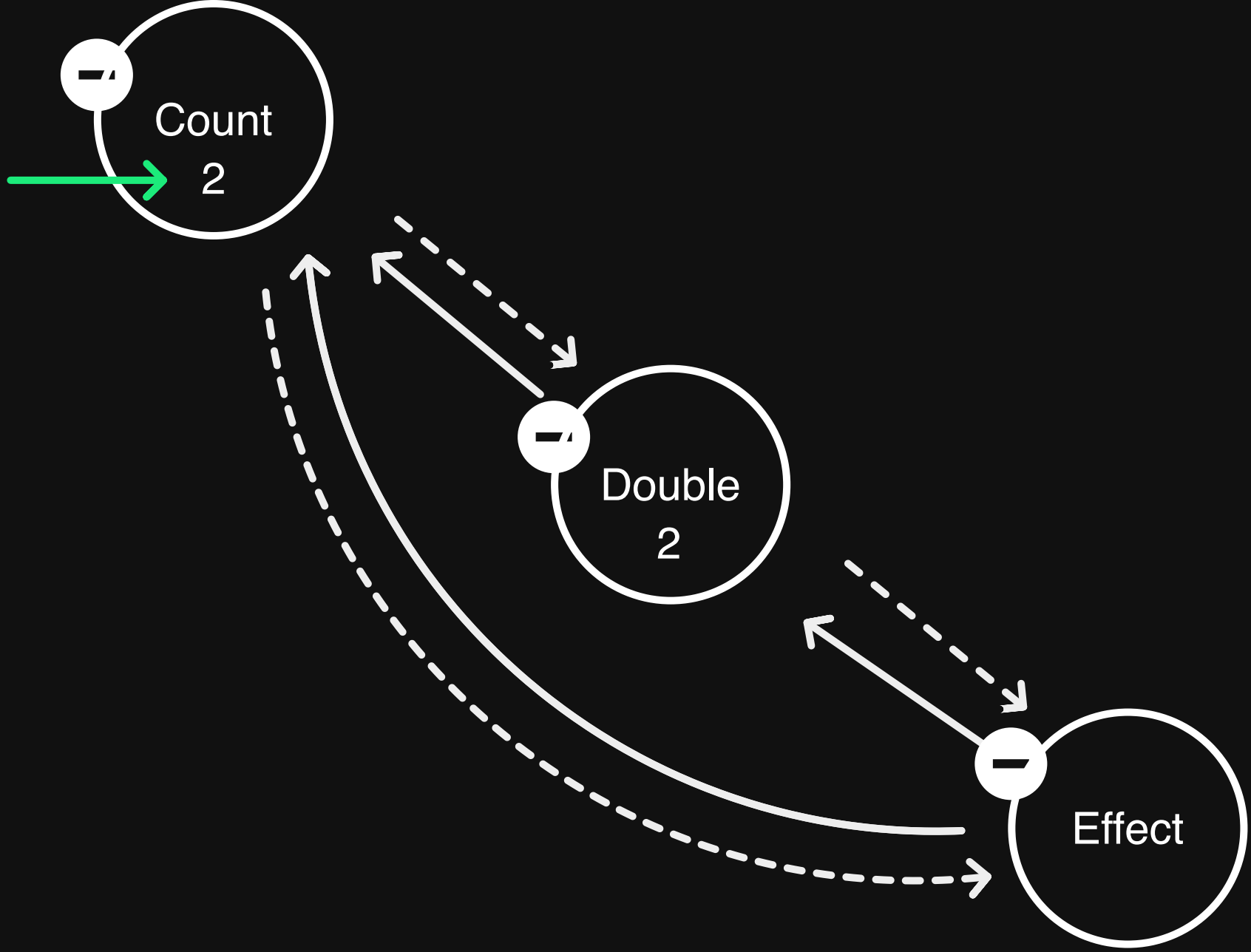
What are JS Signals

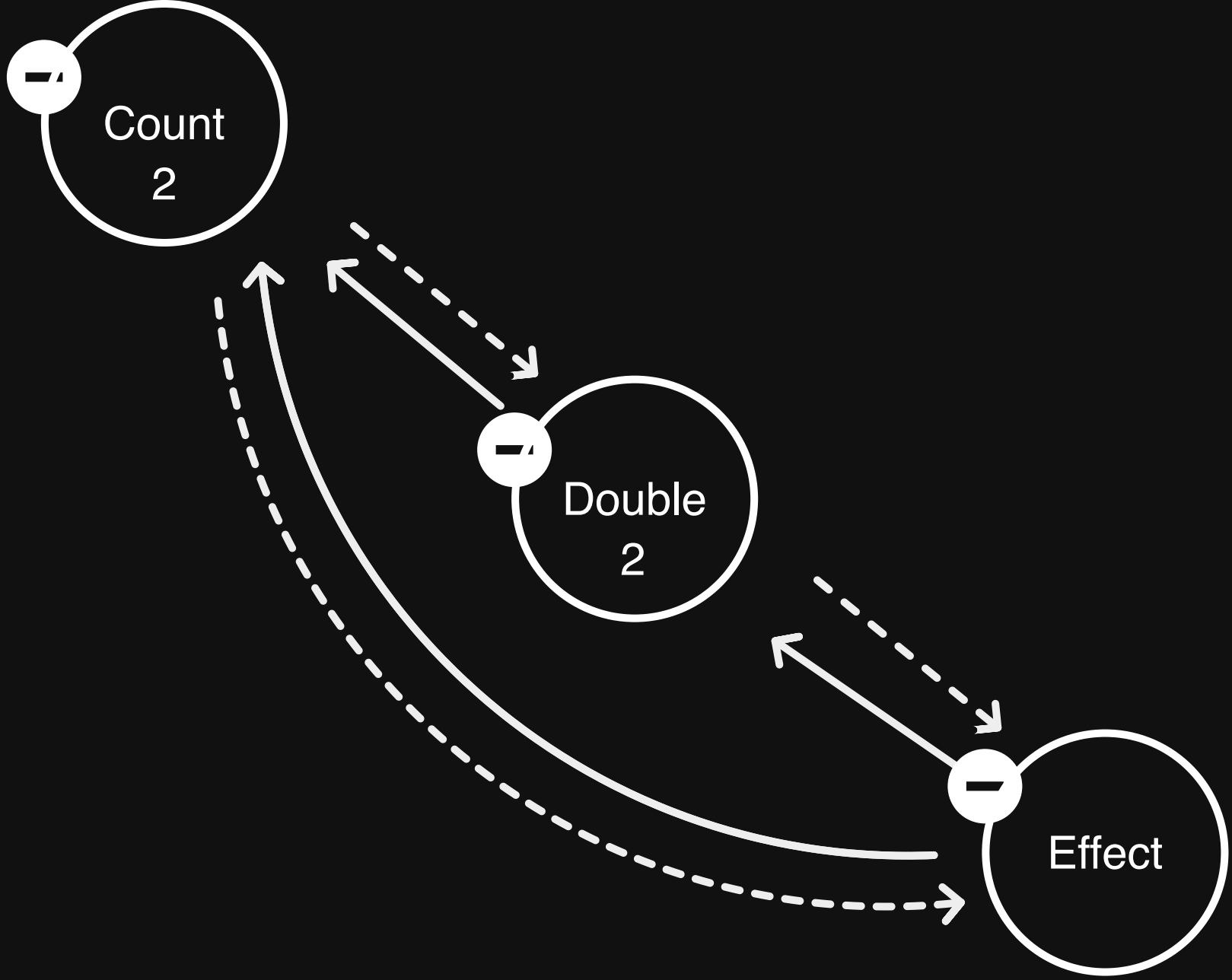
```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => {
  count.value++;
});

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

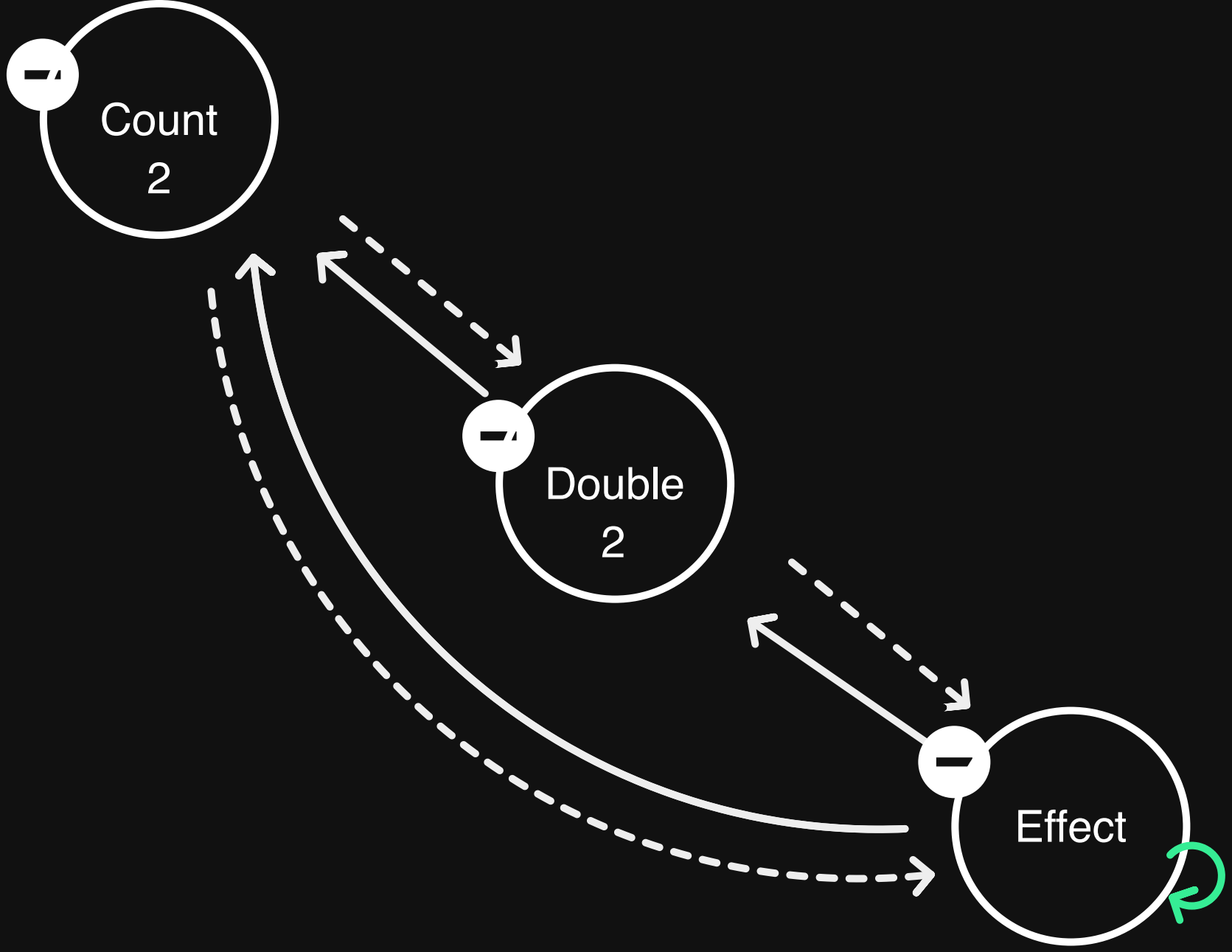

Count
2
Double
2
Effect


Count
2
Double
2
Effect


Count
2
Double
2
Effect

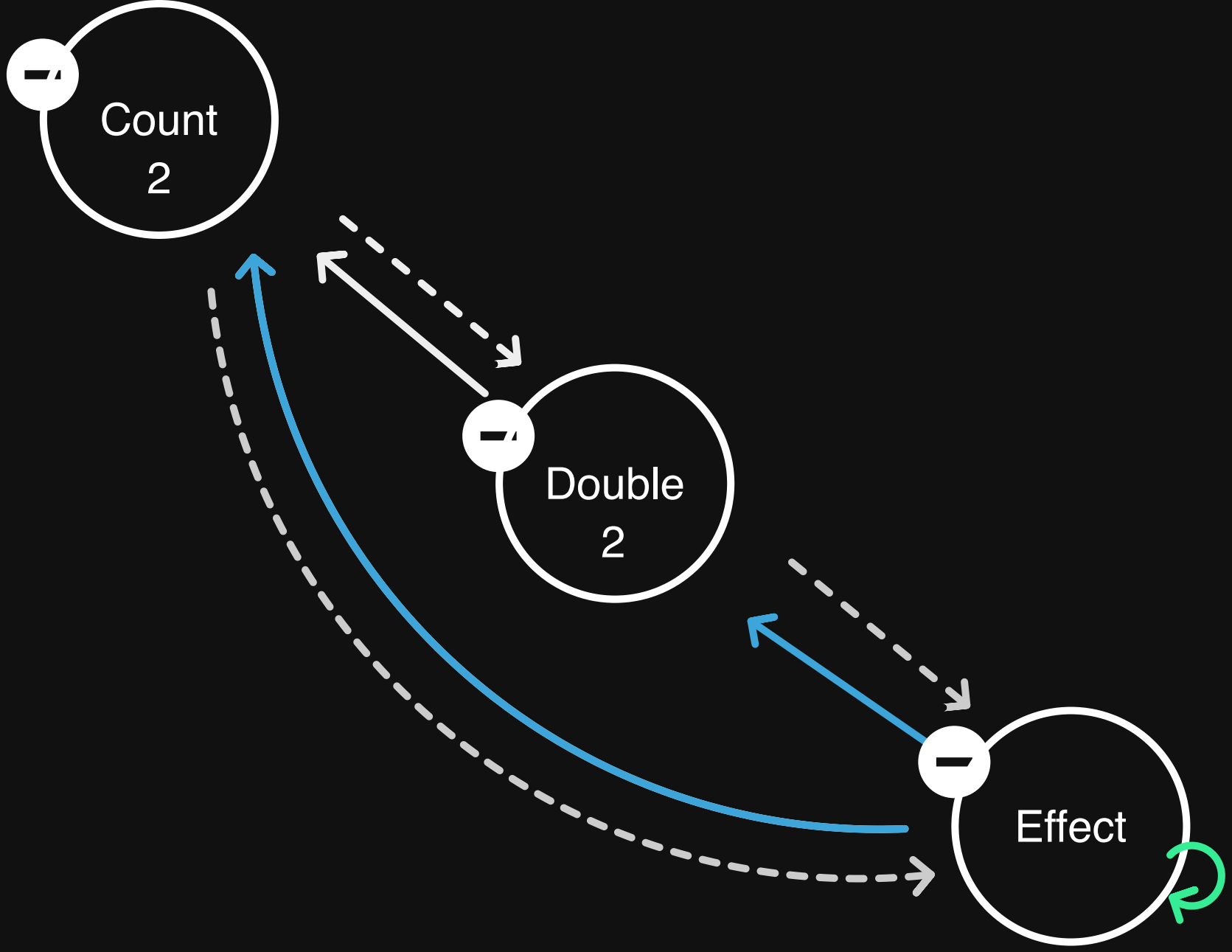

Count
2
Double
2
Effect

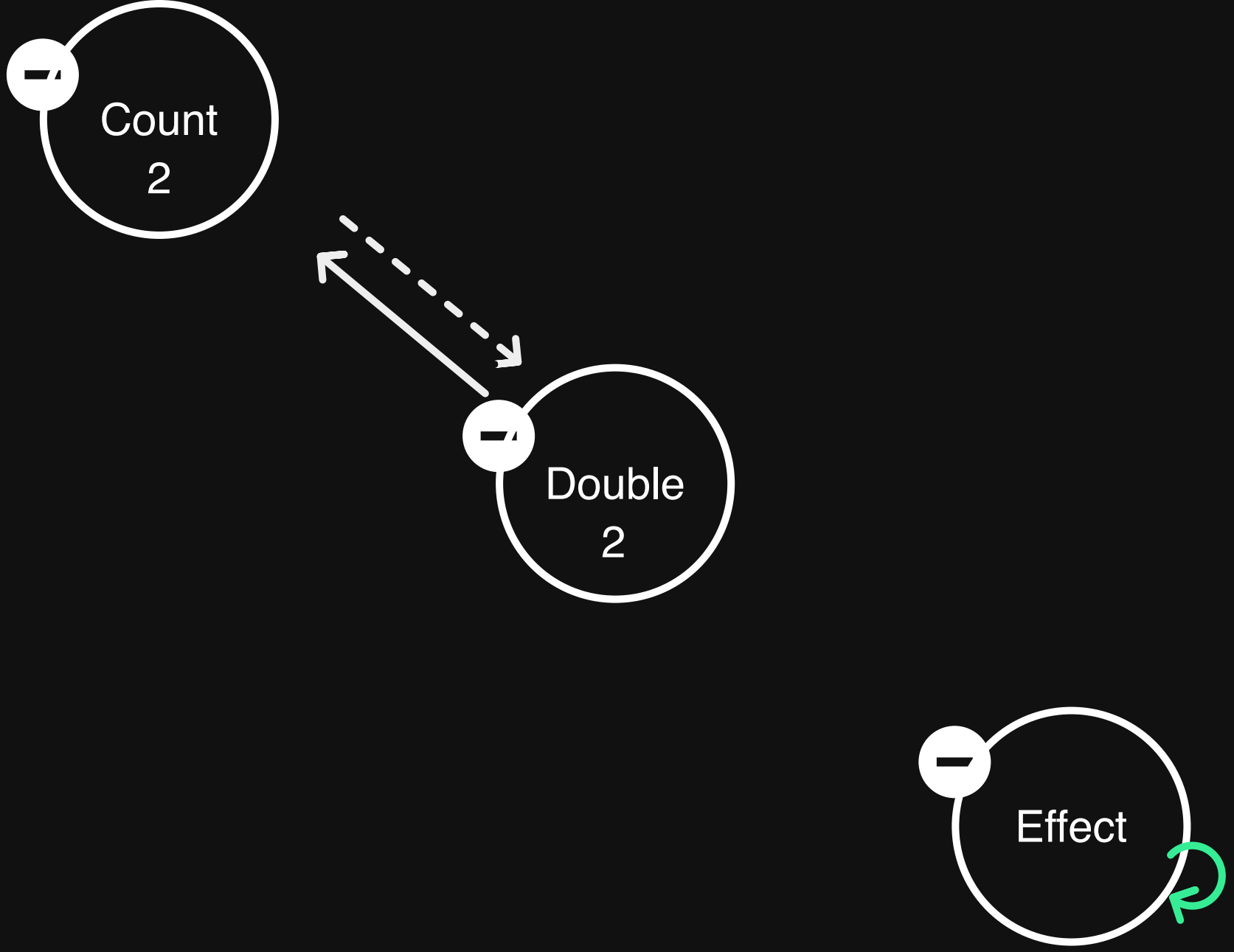

Count
2
Double
2
Effect

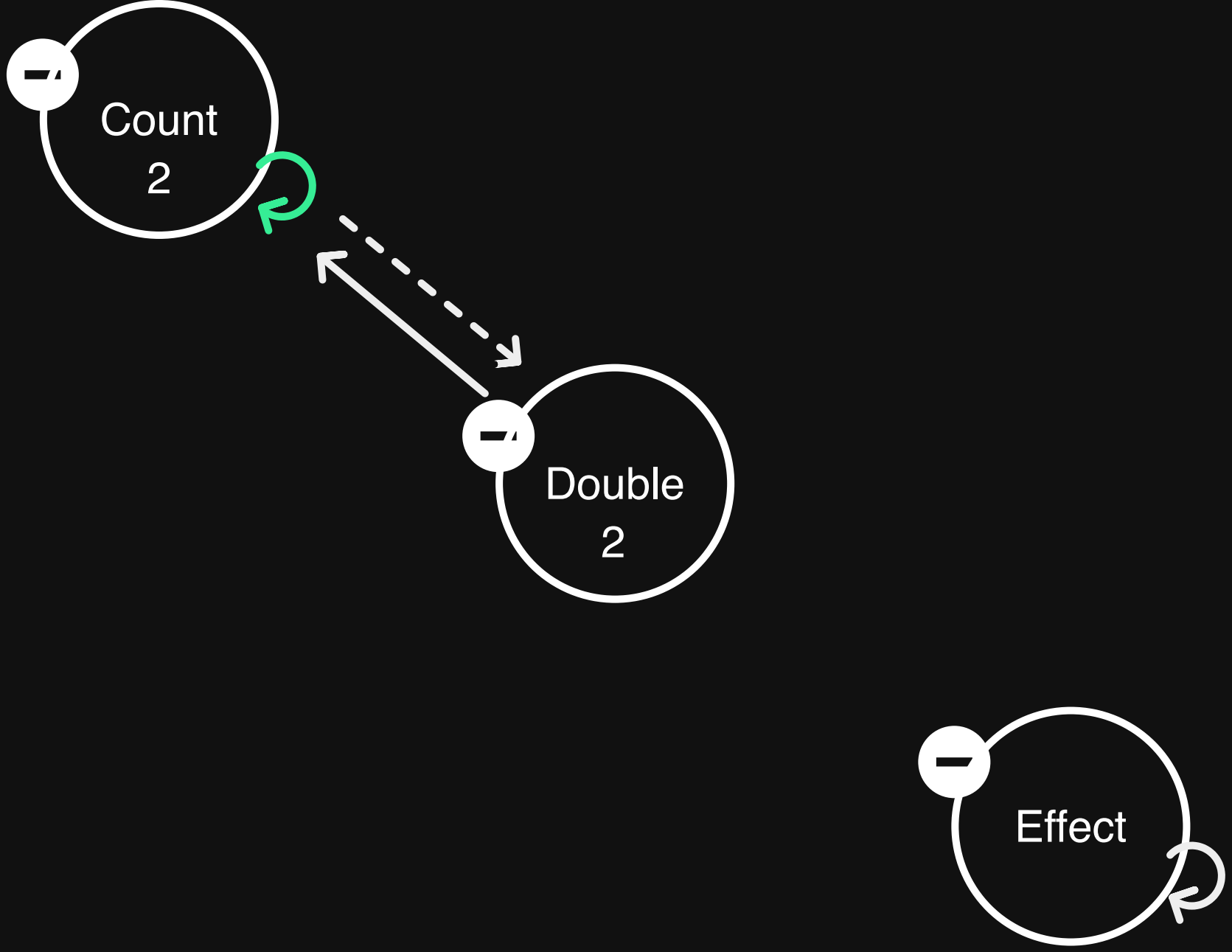

Count
2
Double
2
Effect

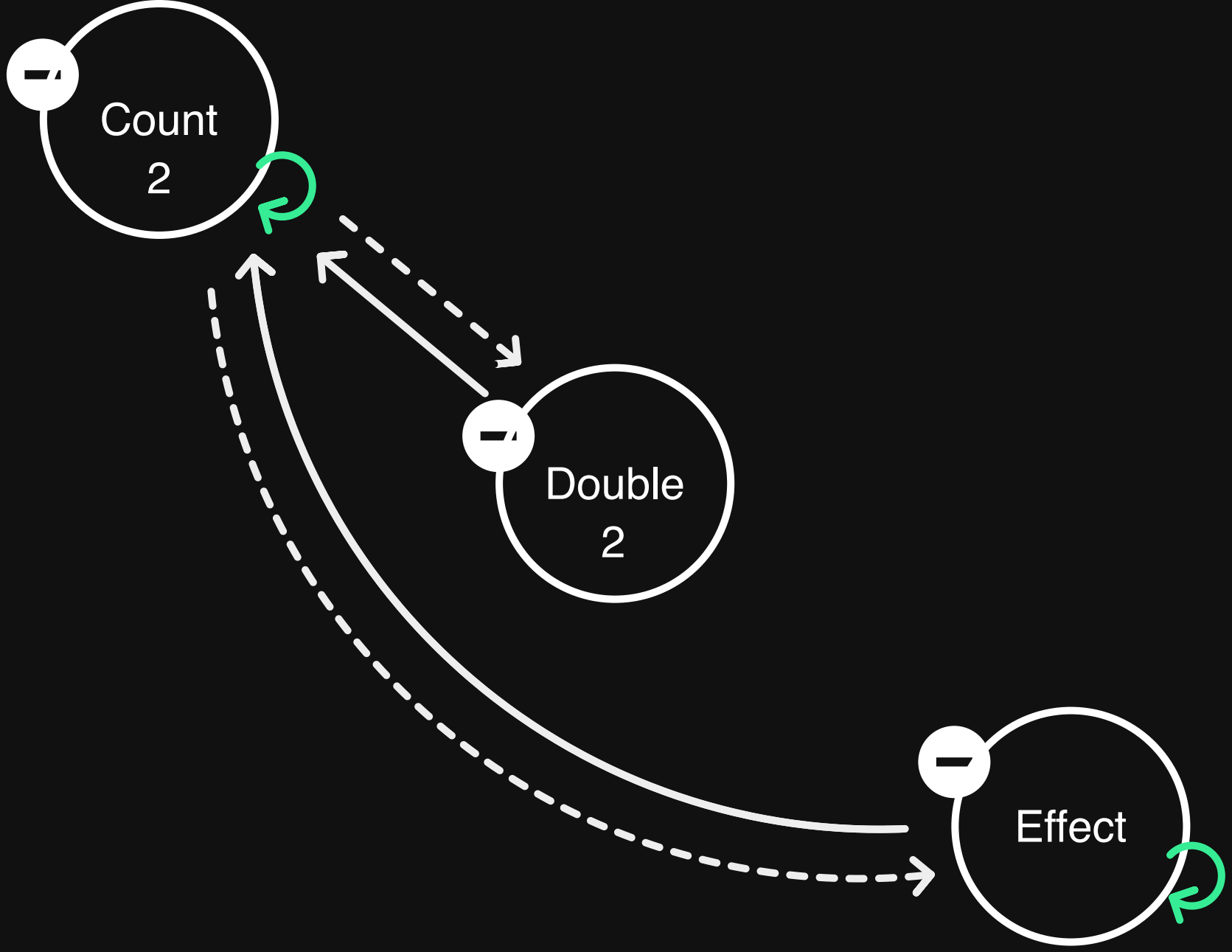
What are JS Signals

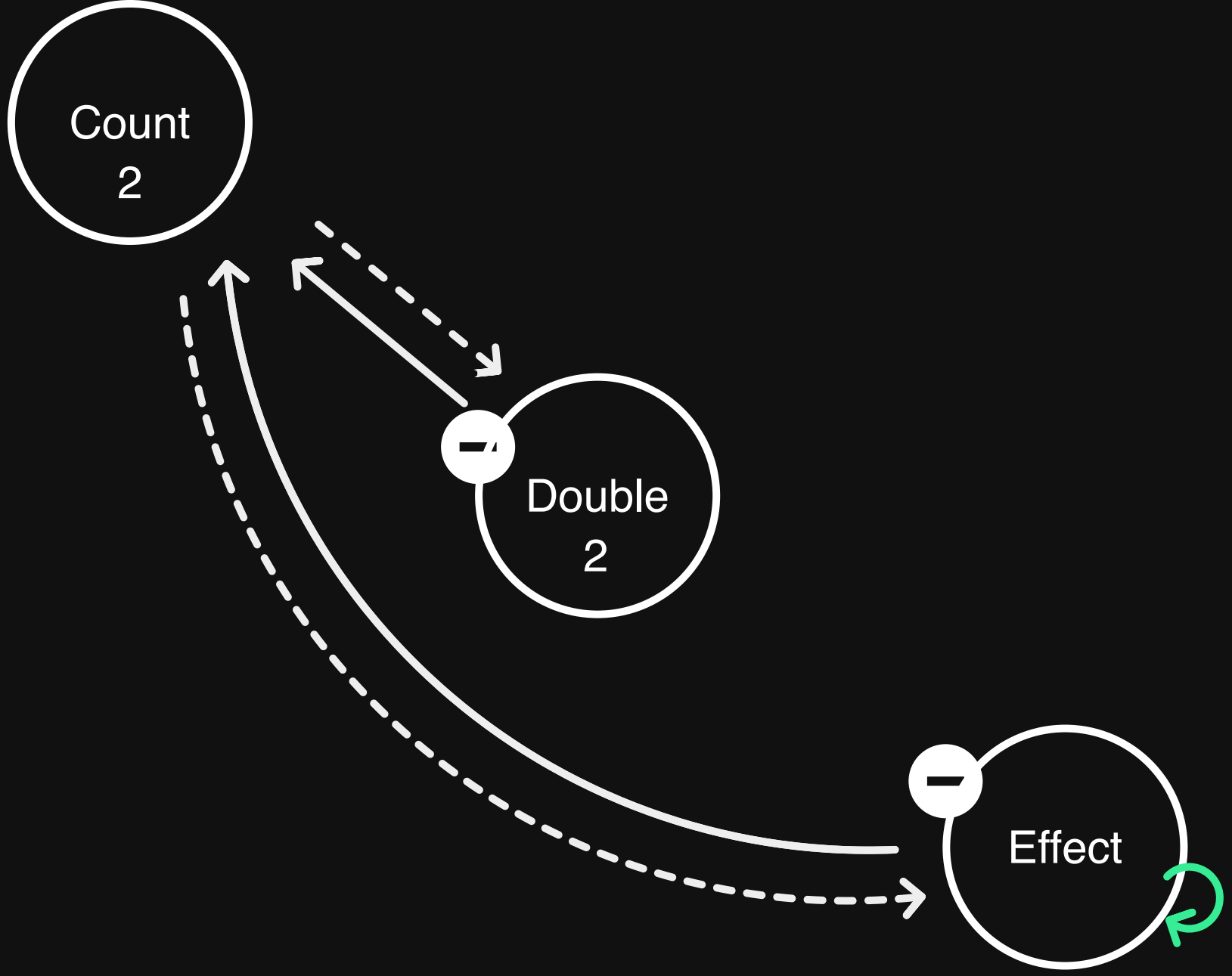
```
import { signal, computed, effect } from './signals';

const button = document.querySelector('button');
const output = document.querySelector('output');

let count = signal(1);
let double = computed(() => count.value * 2);

button.addEventListener('click', () => {
  count.value++;
});

effect(() => {
  output.innerHTML = `
    <h1>${count.value} * 2 = ${double.value}</h1>
  `;
});
```

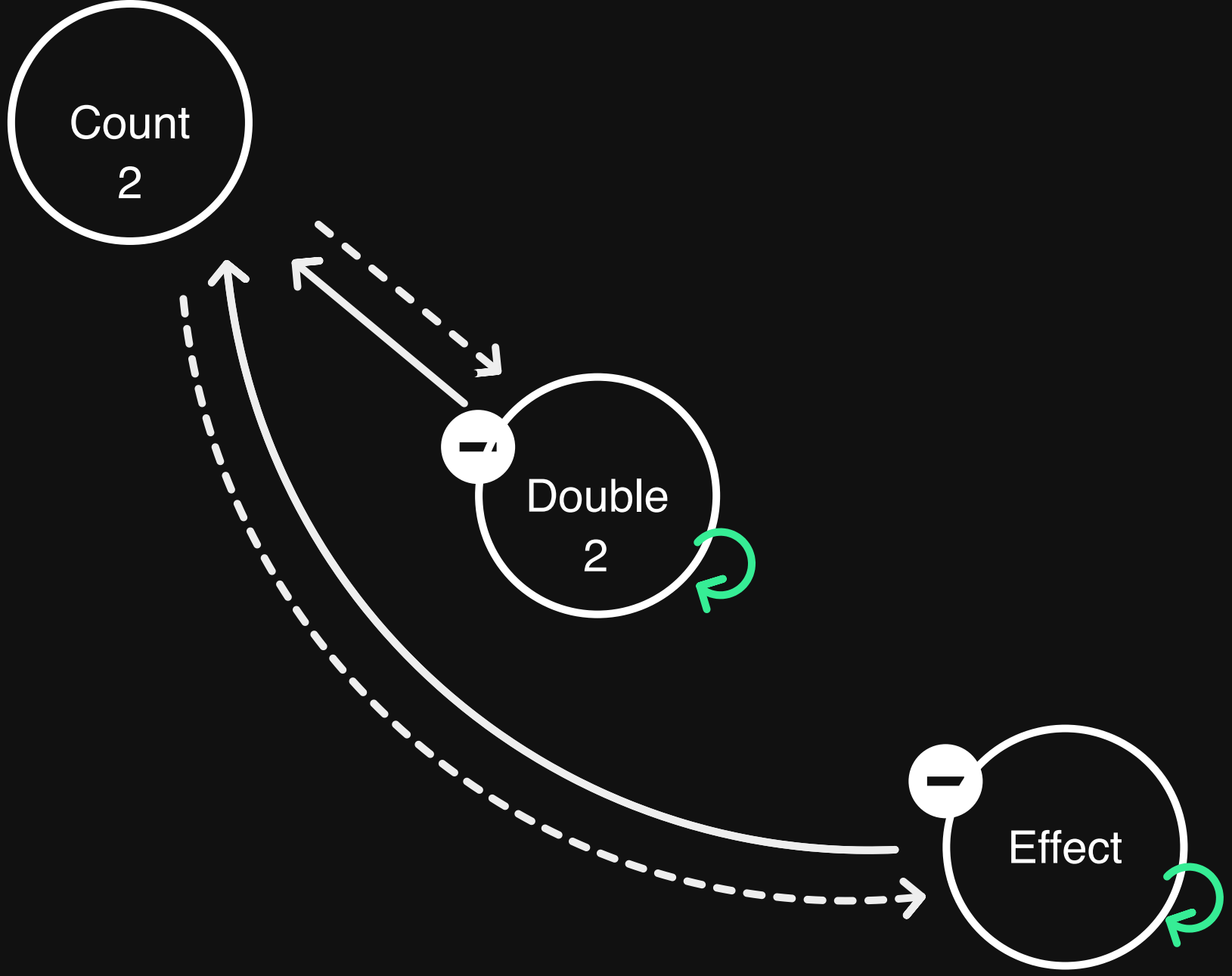

Count
2
Double
2
Effect

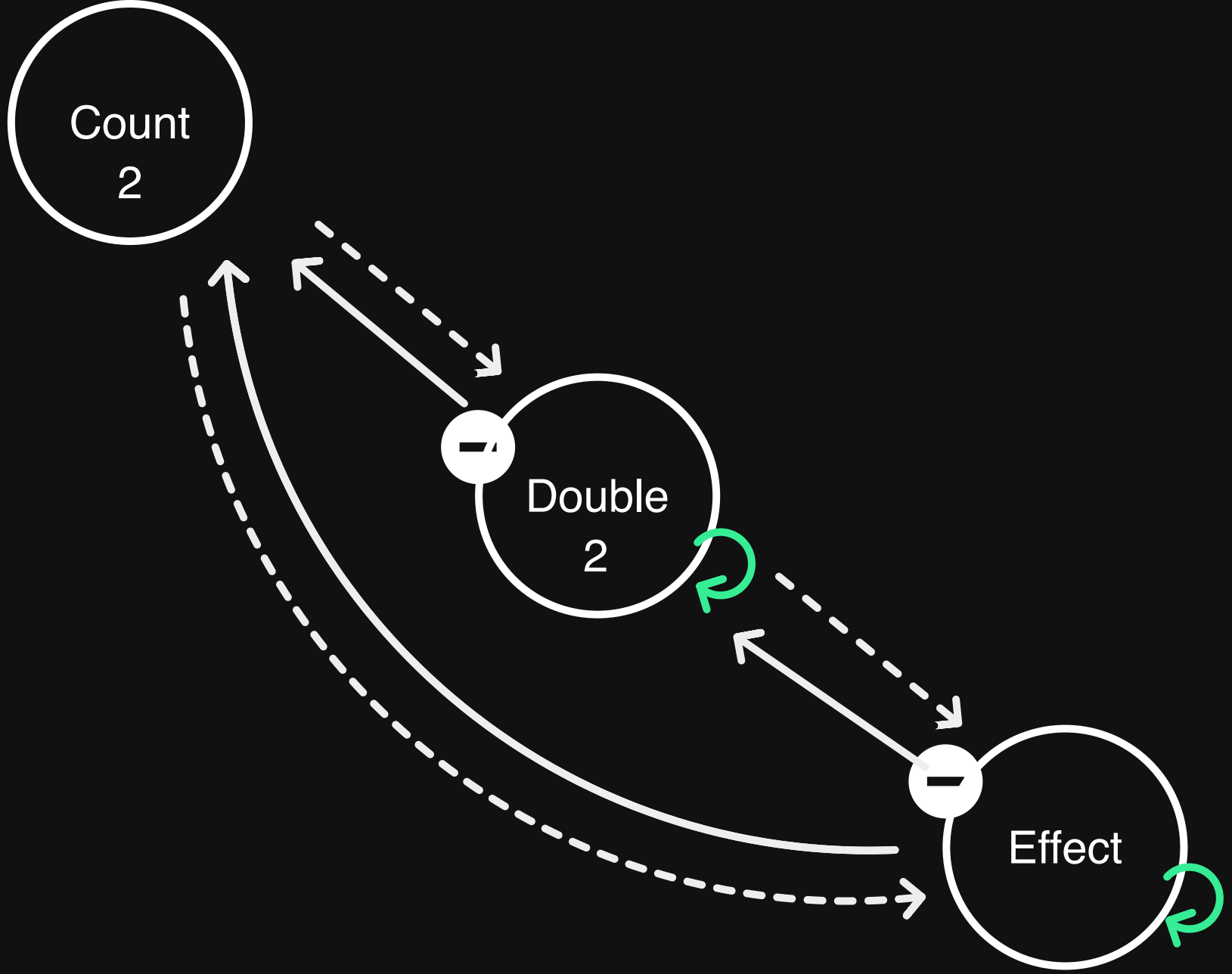

Count
2
Double
2
Effect

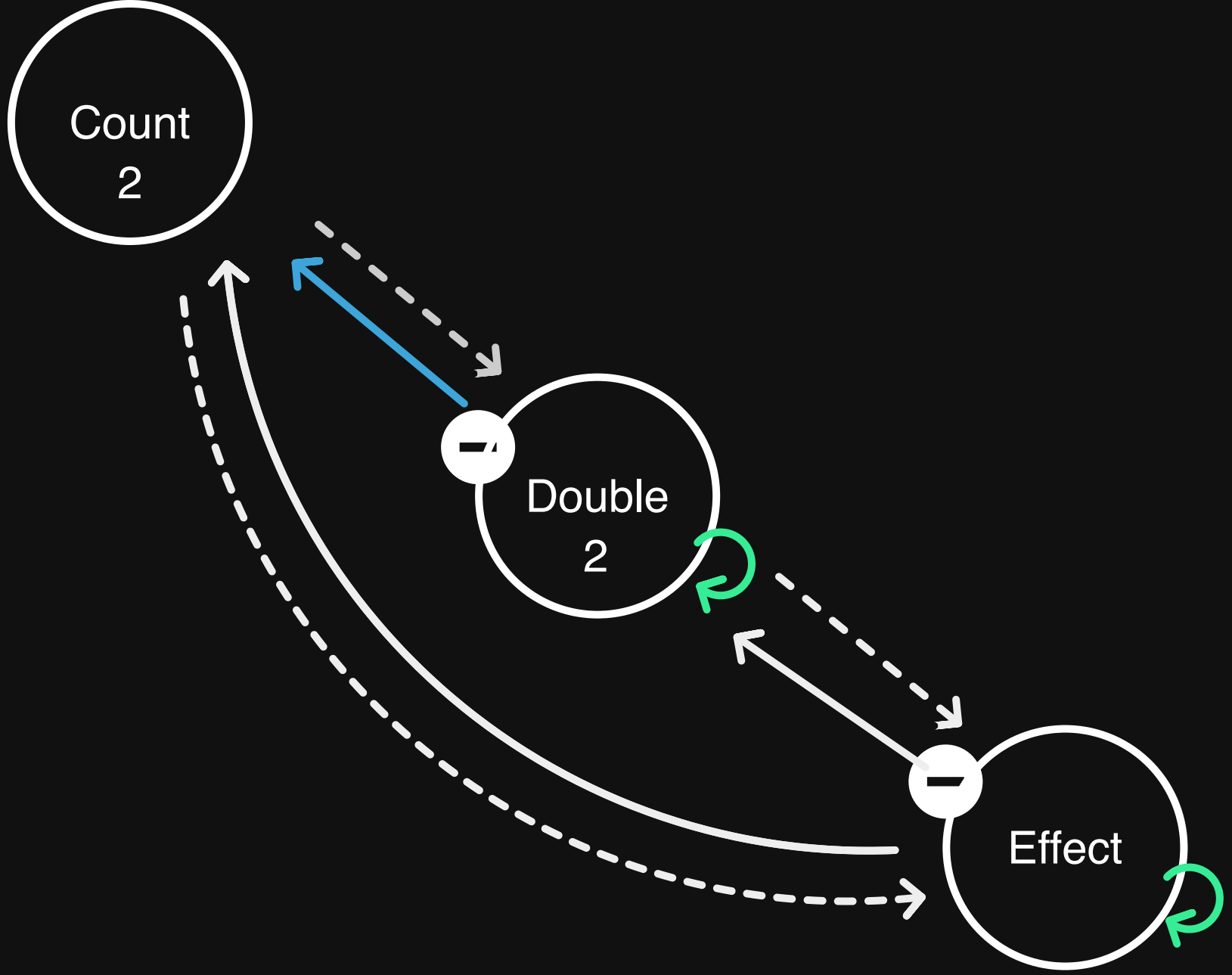

Count
2
Double
2
Effect

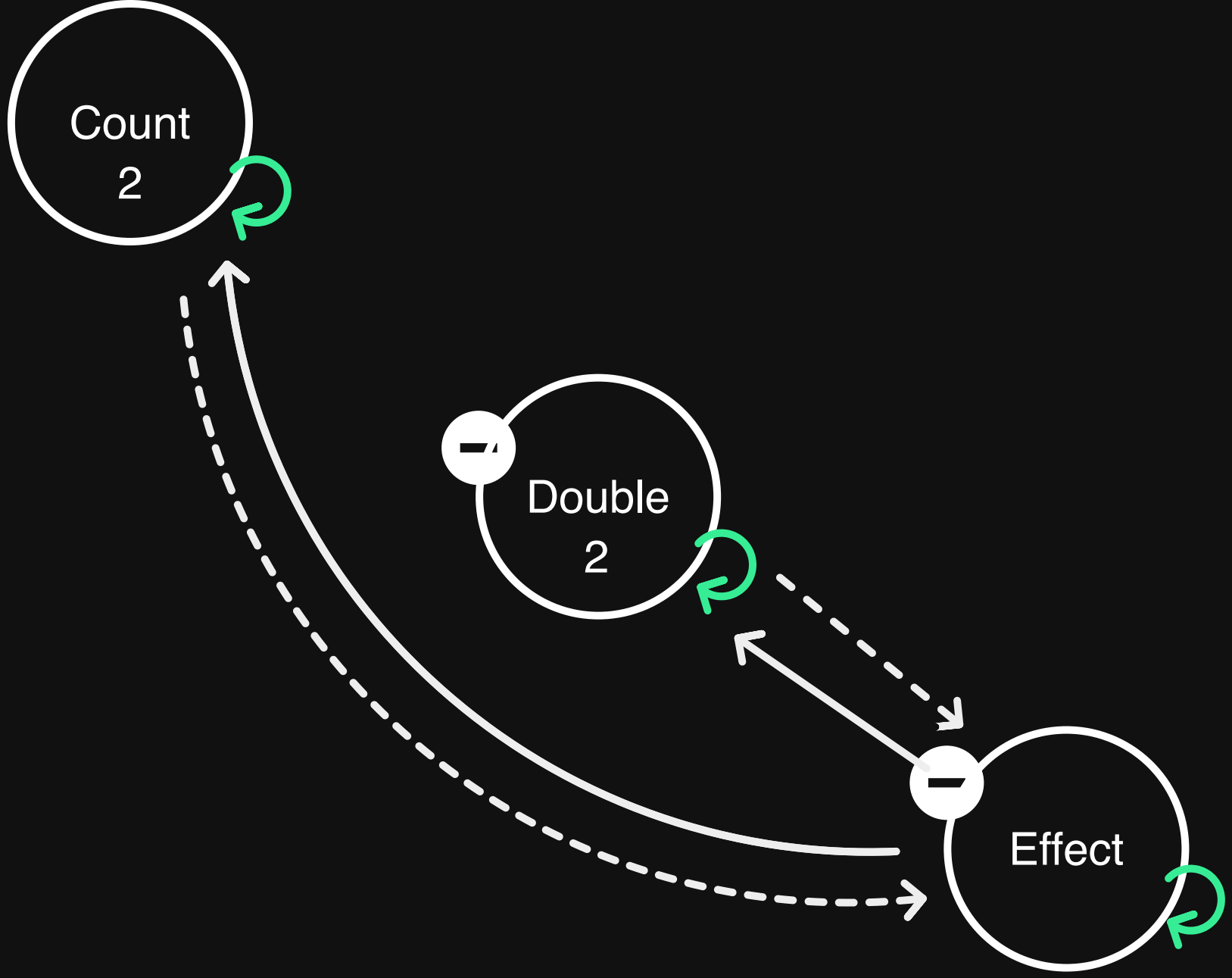

Count
2
Double
2
Effect

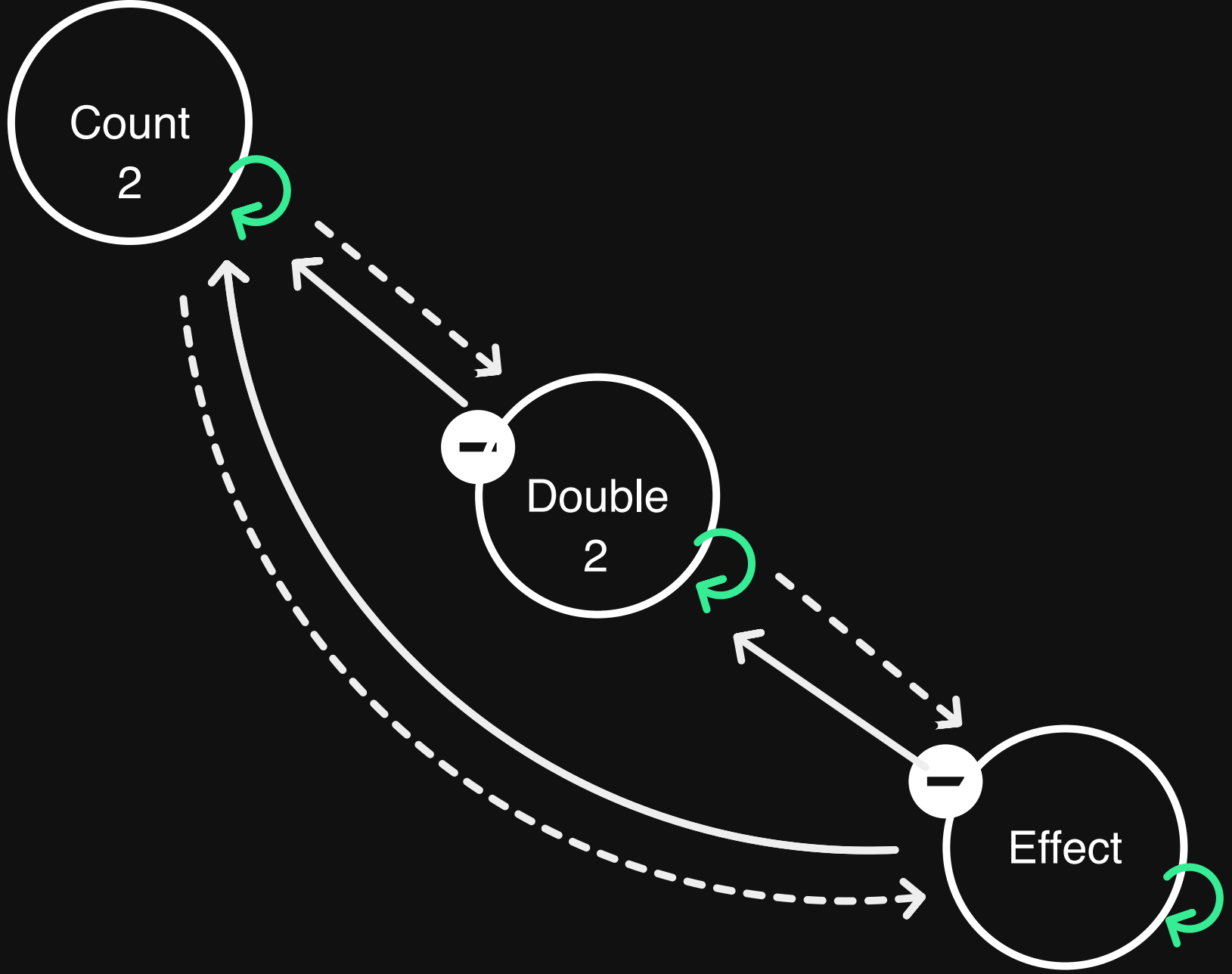

Count
2
Double
2
Effect

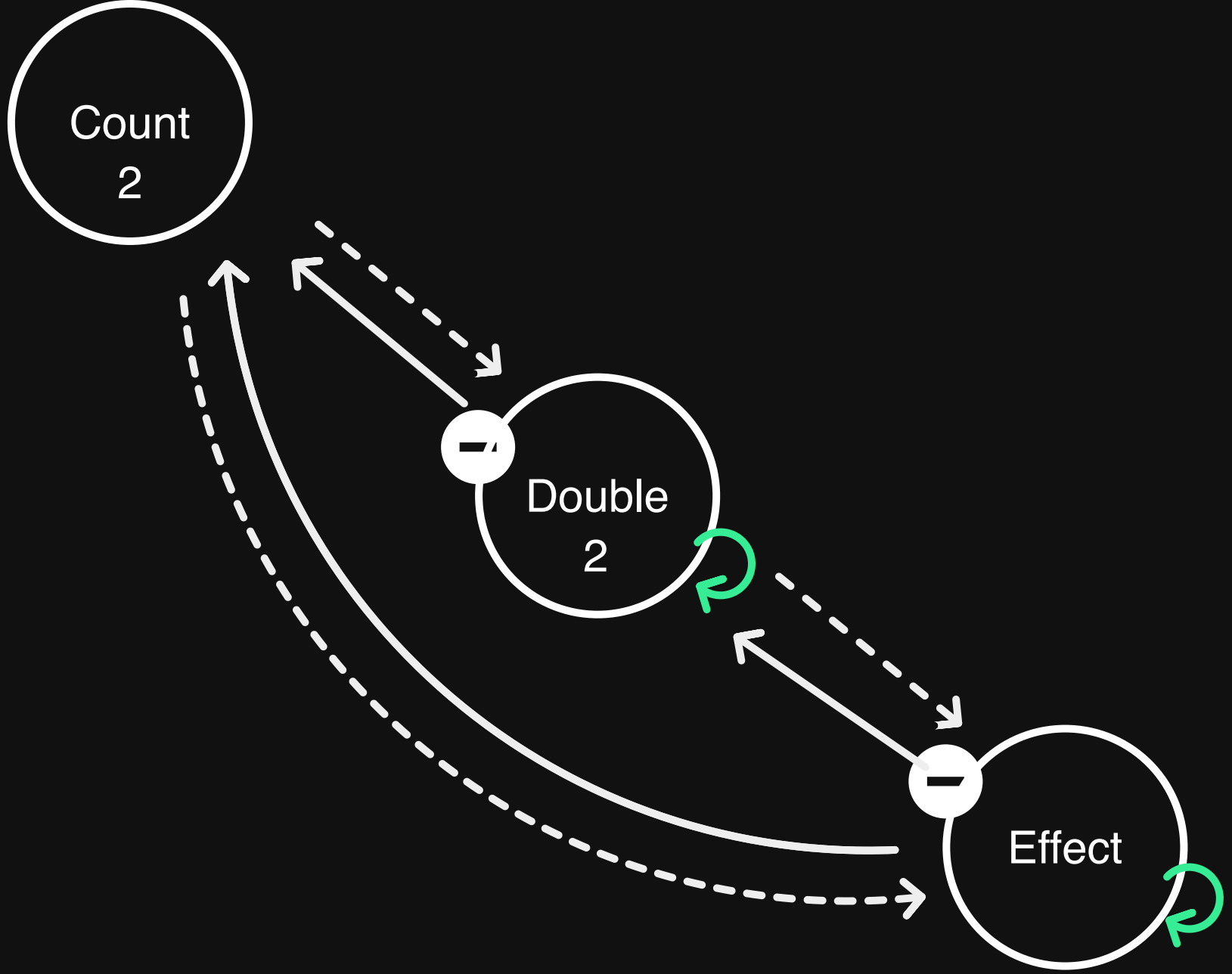

Count
2
Double
2
Effect

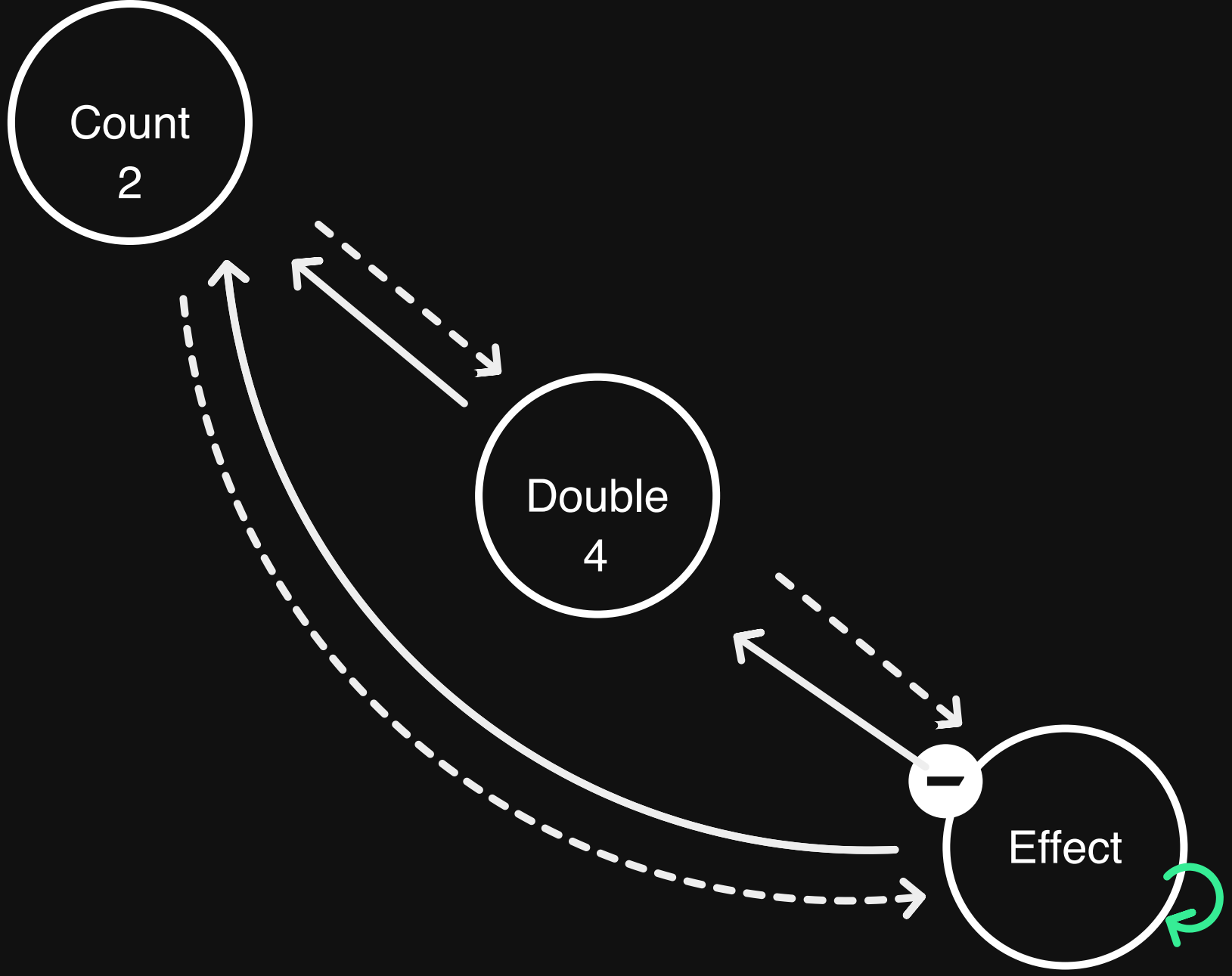

Count
2
Double
2
Effect

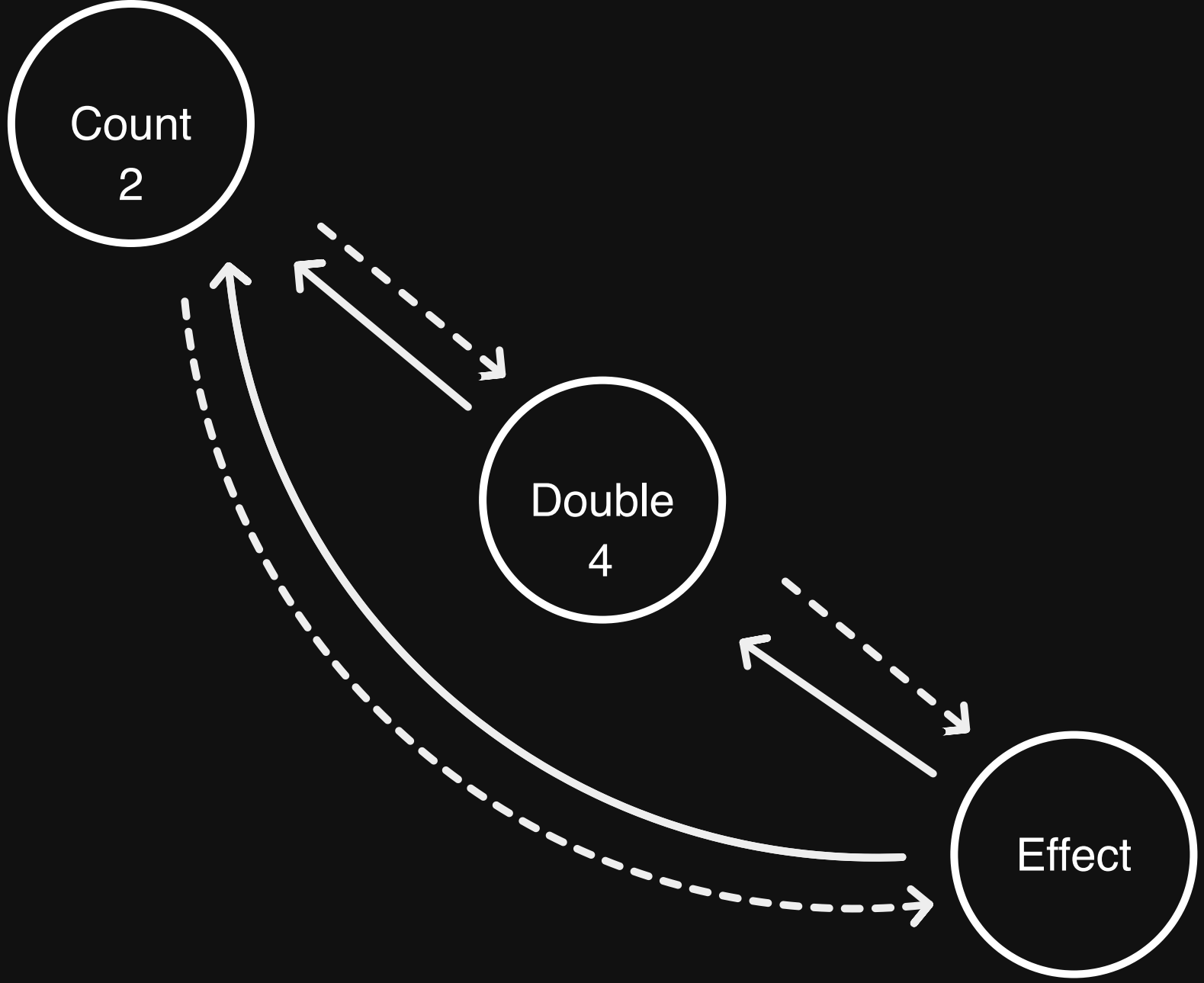

Count
2
Double
2
Effect


Count
2
Double
2
Effect


Count
2
Double
2
Effect


Count
2
Double
2
Effect


Count
2
Double
4
Effect

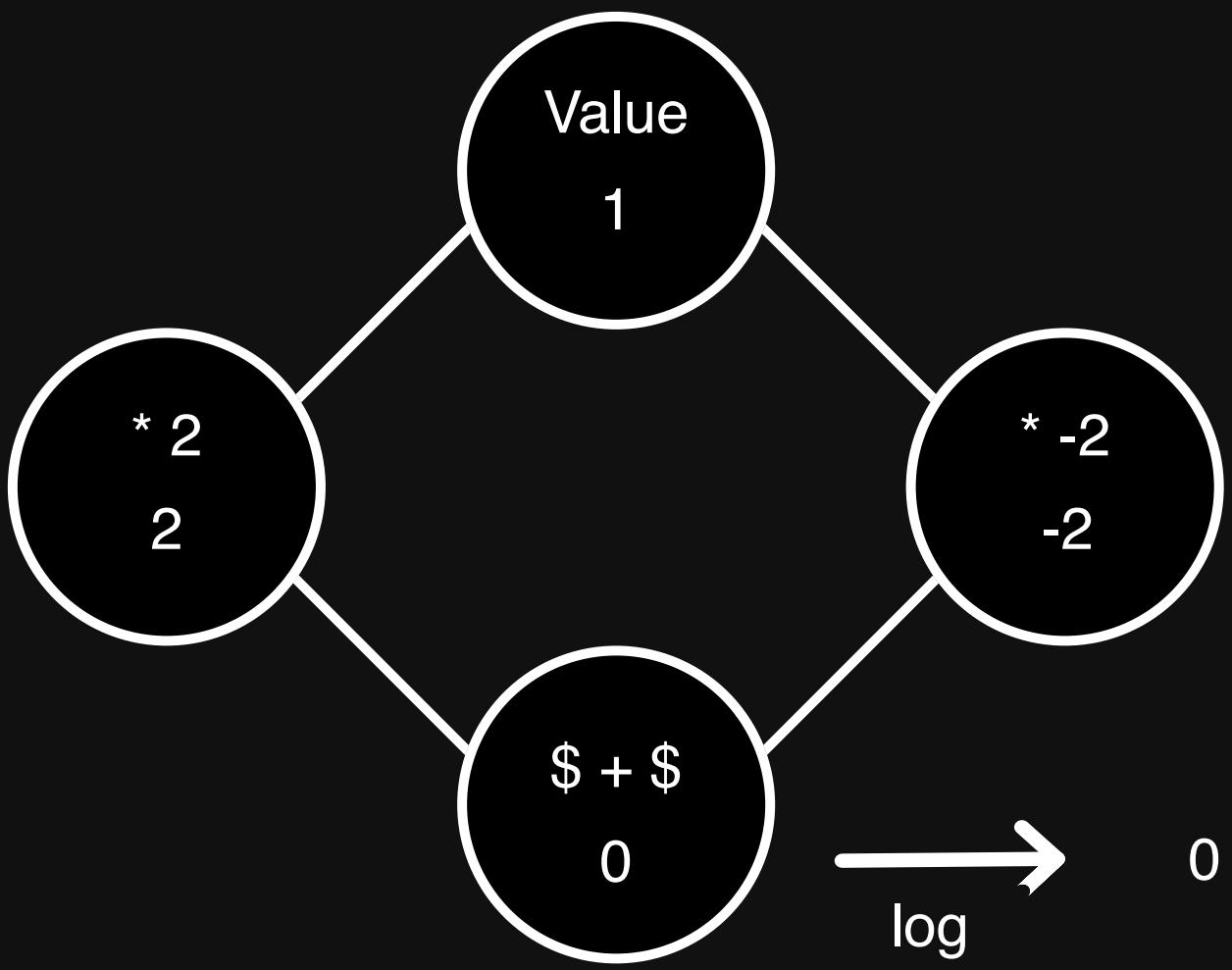

Count
2
Double
4
Effect

The Key Takeaways

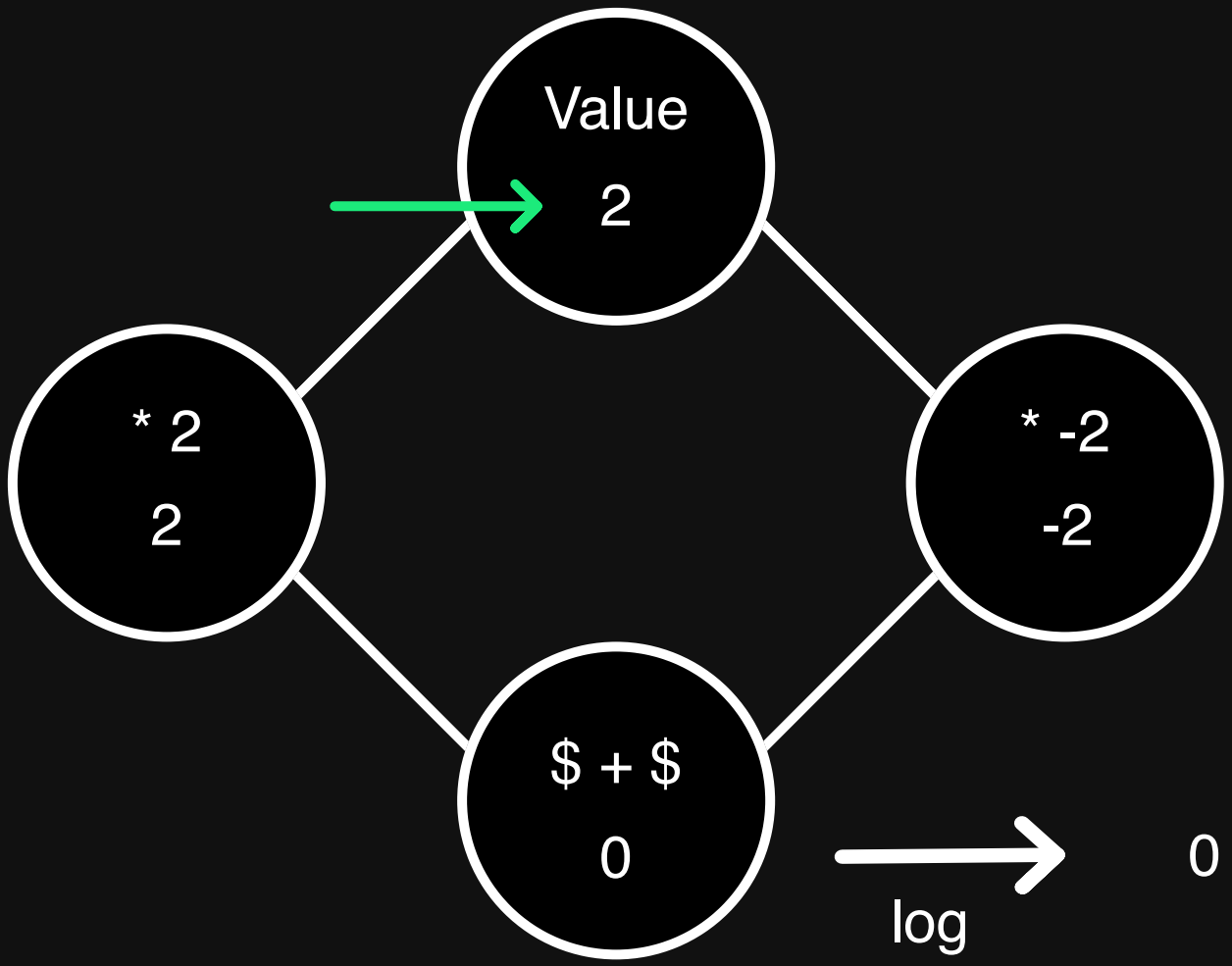
- the "reads" set up the singal trees sink/source relationships
- there are 2 stages, the notify stage and the read
- everything is read lazily and only if it might have been changed in the notify stage

The Diamond Problem

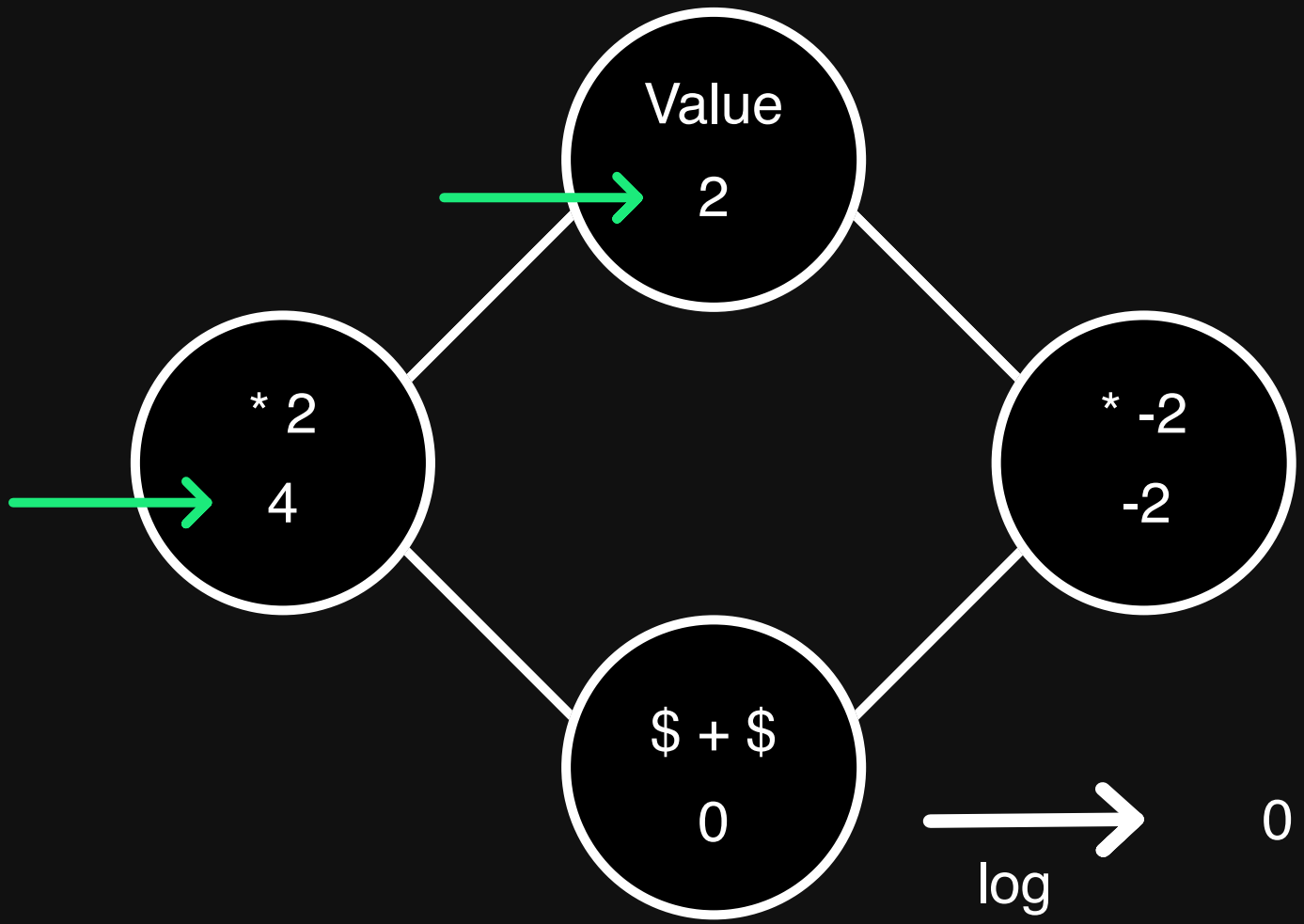
The Diamond Problem



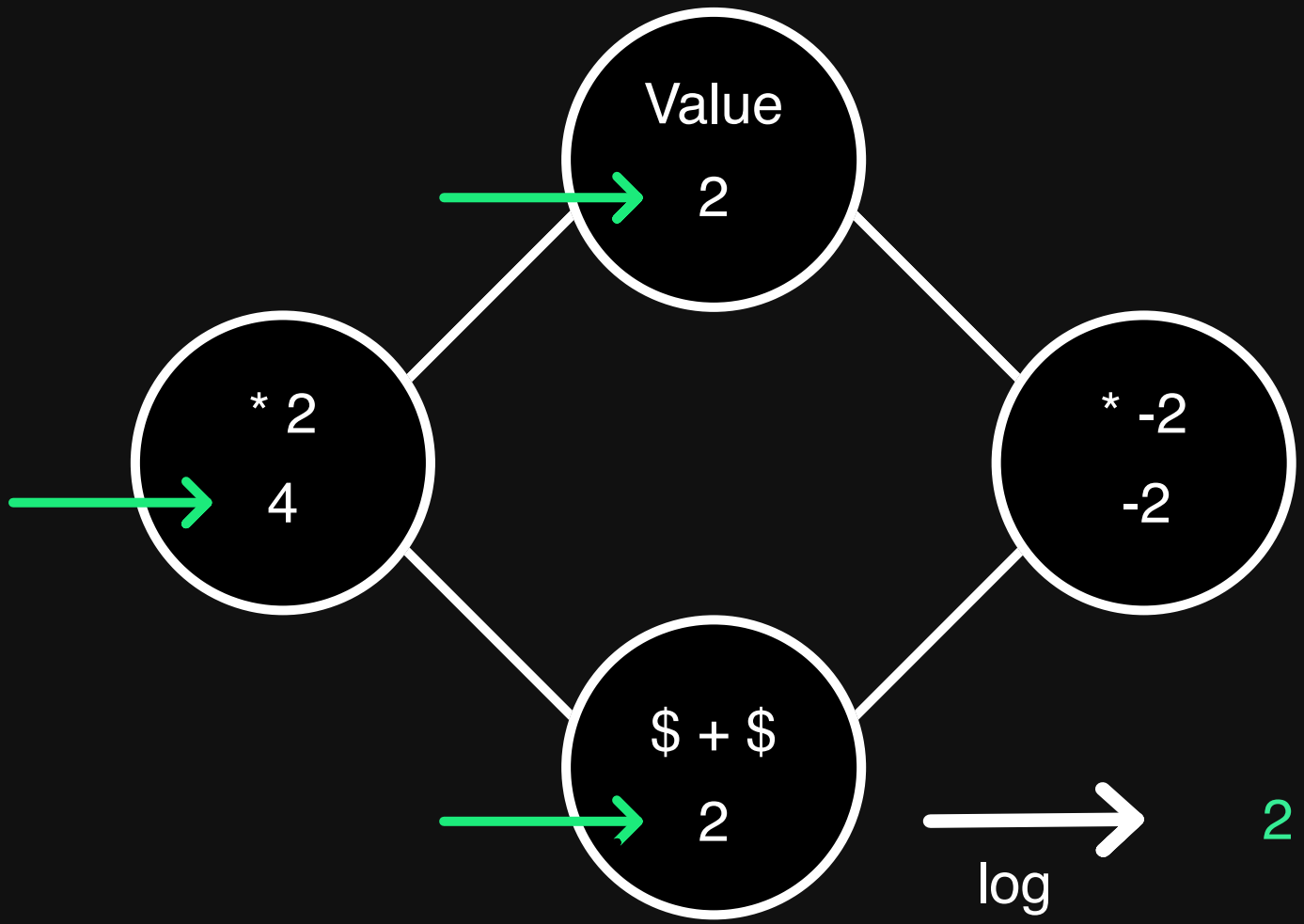
The Diamond Problem



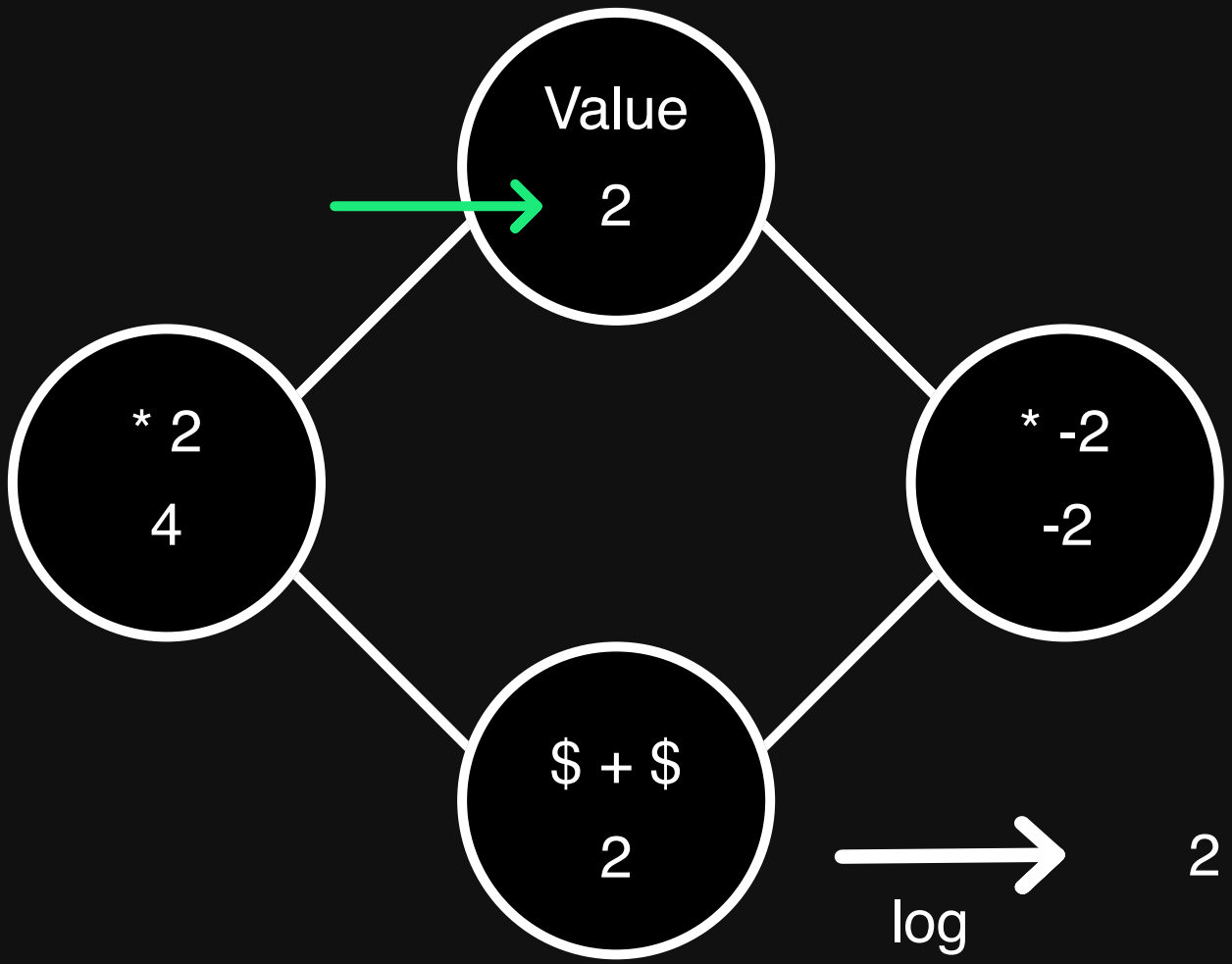
The Diamond Problem



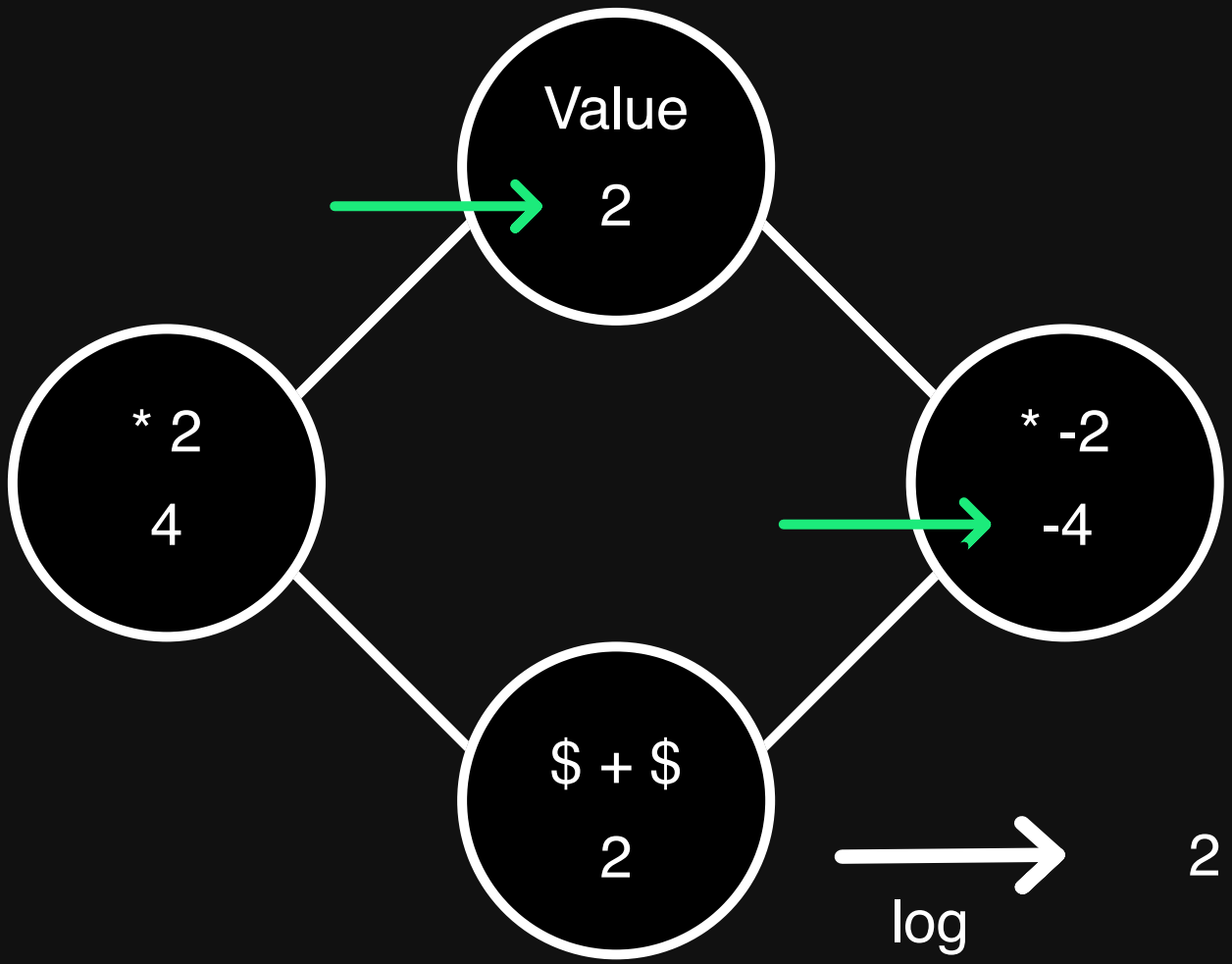
The Diamond Problem



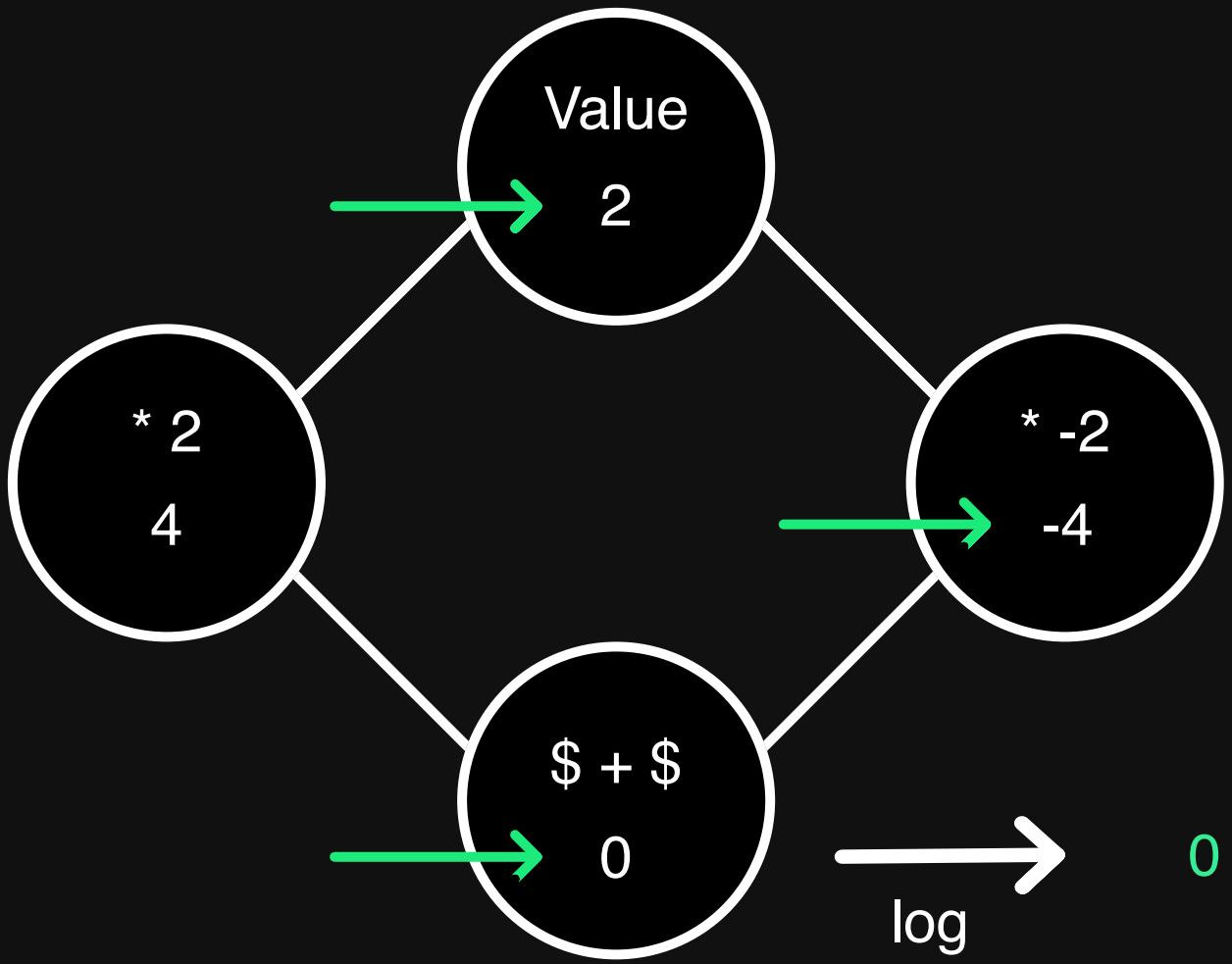
The Diamond Problem



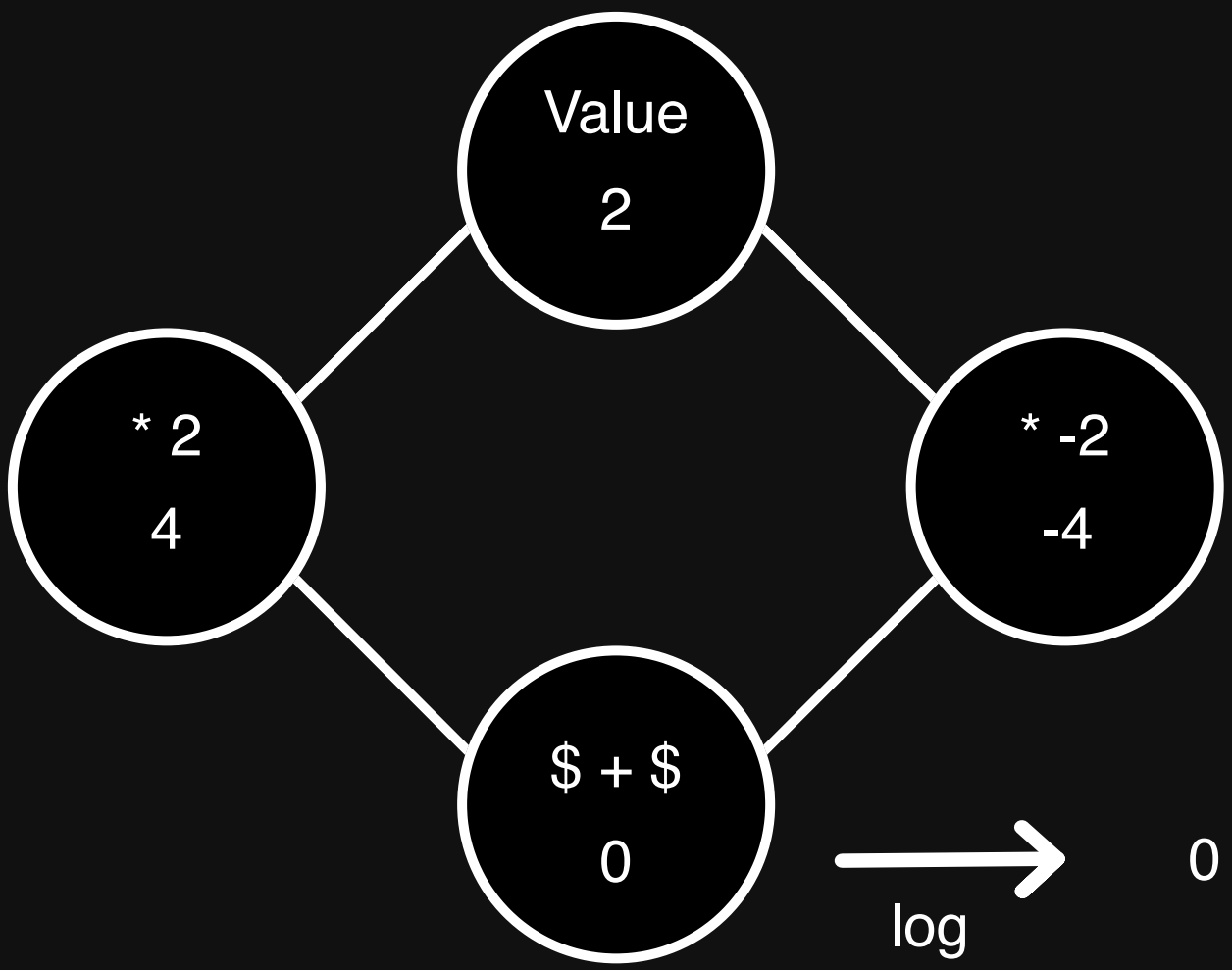
The Diamond Problem



The Diamond Problem



The Diamond Problem




A not-so-n poorly named,
named, reactive state
primitive

Filter

In this article

Constructor

`AbortController()`

Creates a new `AbortController` object instance.

Instance properties

`AbortController.signal` Read only

Returns an `AbortSignal` object instance, which can be used to communicate with, or to abort, an asynchronous operation.

Instance methods

`AbortController.abort()`

Aborts an asynchronous operation before it has completed. This is able to abort fetch requests, consumption of any response bodies, and streams.

Examples

Note: There are additional examples in the `AbortSignal` reference.


Developer Hapiness

Live Coding Result

```
let currentEffect = null;

export const signal = (val) => {
  return {
    sinks: new Set(),
    get value() {
      if (currentEffect) {
        this.sinks.add(currentEffect);
        currentEffect.sources.add(this);
      }
      return val;
    },
    set value(newValue) {
      val = newValue;
      [...this.sinks].forEach((sink) => sink.run());
    },
  };
};

export const effect = (fn) => {
  const _effect = {
    sources: new Set(),
    run() {
      this.sources.forEach((source) => {
        source.sinks.delete(this);
      });
      this.sources.clear();

      let prev = currentEffect;
      currentEffect = this;
      fn();
      currentEffect = prev;
    },
  };
  _effect.run();
};

export const computed = (fn) => {
  const s = signal();
  effect(() => (s.value = fn()));
  return s;
};
```

https://github.com/tc39/proposal-signals

README | Code of conduct | MIT license | Security

🚦 JavaScript Signals stand proposal 🚦

Stage 1 (explanation)

TC39 proposal champions: Daniel Ehrenberg, Yehuda Katz, Jatin anathan, Shay Lewis, Kristen Hewell Garrett, Dominic Gannaway, Preston Sego, Milo M, Rob berg

Original authors: Rob Eisenberg and Daniel Ehrenberg

This document describes an early common direction fo nals in JavaScript, similar to the Promises/A+ effort which preceded the Promises standardized by TC39 in ES2015. Try it for yourself, using a polyfill.

Similarly to Promises/A+, this effort focuses on aligning avaScript ecosystem. If this alignment is successful, then a standard could emerge, based on that experience. Several framework authors are collaborating here on a common model which could back their reactivity core. The current draft is based on design input from the authors/maintainers of Angular, Bubble, Ember, FAST, MobX, Preact, Qwik, RxJS, Solid, Starbeam, Svelte, Vue, Wiz, and more...

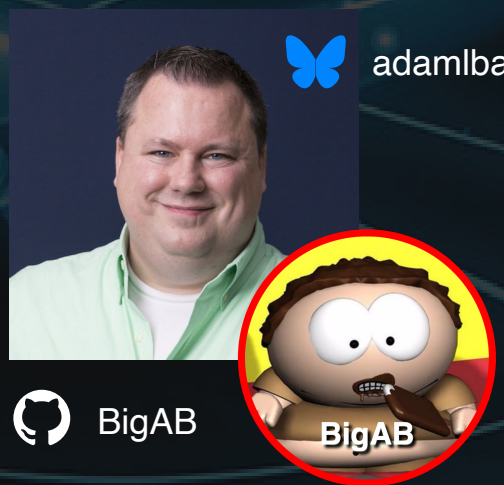
Differently from Promises/A+, we're not trying to solve for a common developer-facing surface API, but rather the precise core semantics of the underlying signal graph. This proposal does include a fully concrete API, but the API is not targeted to most application developers. Instead, the signal API here is a better fit for frameworks to build on top of, providing interoperability through common signal graph and auto-tracking mechanism.

The plan for this proposal is to do significant early prototyping, including integration into several frameworks, before advancing beyond Stage 1. We are only interested in standardizing Signals if they are suitable for use in

Questions?

JS Signals

Use this QR Code to let me know what you thought of this talk!



https://slides.com/bigab