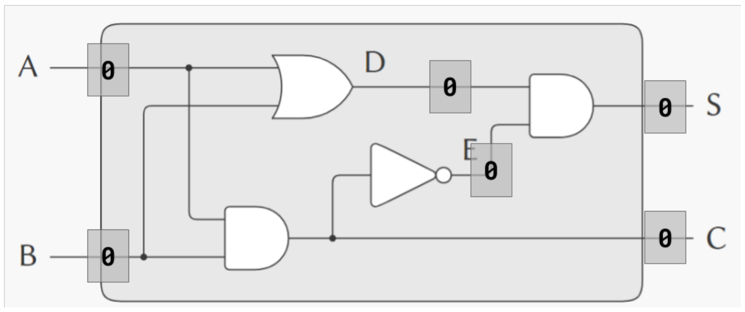

Figure 1: Half-Adder wire signals at the start if using `accept-action-procedure!` suggested in the problem.

Exercise 3.31

Problem statement

The internal procedure accept-action-procedure! defined in make-wire specifies that when a new action procedure is added to a wire, the procedure is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the systems response would differ if we had defined accept-action-procedure! as

```
(define (accept-action-procedure! proc)
  (set! action-procedures
        (cons proc action-procedures)))
```

Solution

The current definition of `accept-action-procedure!` is

```
(define (accept-action-procedure! proc)
      (set! action-procedures
            (cons proc action-procedures))
      (proc))
```

Also remember the `set-my-signal!` procedure

```
(define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each
                  action-procedures))
          'done))
```

Notice that this only calls the downstream procedures when the signal *changes*. Imagine we were to use the `accept-action-procedure!` suggested in the problem text. Figure 1 shows the signals on the wire for the half adder. Notice that the signal on wire E is 0, which is wrong, as it's supposed to invert the signal on wire C. This is because wires get created with `(let ((signal-value 0) ... )))`, and that 0 sign never actually gets modified.

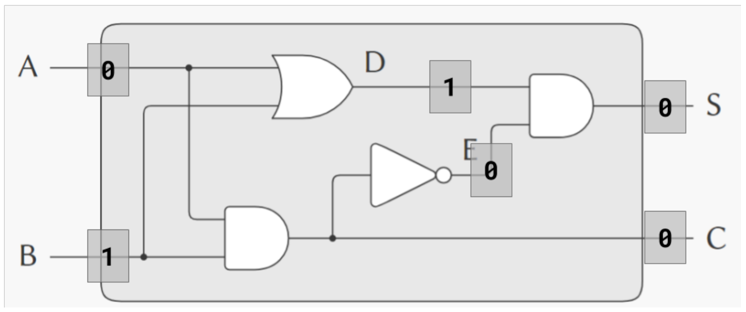

Figure 2: Signals on half adder wires after changing the signal on wire B from 0 to 1 given the initial state was the one shown in 1

But how big of an issue is this? After all, the signals on C and S are both 0, which is correct. Let's say that we modify the signal on B to be 1. This change will trigger changes in the system. The new wire signals are shown in Figure 2.

This time the result is wrong. Since wire E will keep the previous signal, now the AND gate that generates the signal on wire S will compute 0 rather than 1.

If you check my solutions to the previous problems I corrected for this by having my version of wire structures call the downstream procedures every time `set-signal!` was called, regardless of whether the signal actually changed or not. I also made sure to set all initial signals to 0, as to trigger changes down the line.

The SICP approach is better, as it doesn't trigger lots of redundant changes, and it doesn't require priming the circuit structure.