Zadanie nr 2 na pracownię

Na wykładzie zobaczyliśmy prosty symulator obwodów logicznych. Niestety nie nadawał się on do symulacji niektórych obwodów, a w szczególności układów synchronicznych, ponieważ nie uwzględniał opóźnień bramek logicznych. Celem drugiego zadania na pracownie jest rozbudowa symulatora o kontrolę nad czasem w symulacji.

Przewody, bramki i kolejka zdarzeń

Kluczową koncepcją w naszym symulatorze będzie przewód, łączący elementy. Przewód ma swój stan opisany wartością logiczną, która może się zmieniać w czasie. Dodatkowo z przewodem powinna być skojarzona lista akcji, które mają się wykonać za każdym razem, gdy stan przewodu ulegnie zmianie.

Takie podejście pozwala na zaimplementowanie bramek logicznych. Do bramki logicznej podłączone jest kilka przewodów wejściowych i jeden wyjściowy. Gdy stan któregokolwiek z przewodów wejściowych ulegnie zmianie, bramka oblicza wartość wyjściową na podstawie stanów wejść i odpowiednio modyfikuje stan przewodu wyjściowego. Zauważ, że w naszym symulatorze bramka nie jest reprezentowana jako pojedynczy obiekt, tylko jako szereg akcji podpiętych do kilku przewodów.

Istotną zmianą w stosunku do symulatora z wykładu ma być wprowadzenie do symulacji czasu i opóźnień bramek logicznych. Jeśli na jednym z przewodów wejściowych bramki nastąpi zmiana stanu, to stan przewodu wyjściowego powinien zostać uaktualniony z pewnym opóźnieniem. Zamodelujemy to w naszym symulatorze wprowadzając następujące dwie mutowalne zmienne do symulacji.

- Bieżący czas — wartość liczbowa reprezentująca ile czasu minęło od początku symulacji mierzonego w umownych jednostkach. Wartość ta może tylko rosnąć. Użytkownik może wykonywać symulację poprzez zażądanie przesunięcia czasu do przodu o zadaną wartość.

- Kolejka zdarzeń. Akcje, o których wiemy, że mają się wydarzyć w najbliższej przyszłości umieszczamy w specjalnej strukturze danych zwanej *kolejką zdarzeń* wraz czasem, kiedy dane wydarzenie ma nastąpić. Kiedy czas próbujemy przesunąć do przodu, wykonujemy wszystkie akcje, których czas wykonania przestaje być w przyszłości, co może skutkować dodaniem nowych zdarzeń do kolejki. Do implementacji kolejki zdarzeń użyj kopców z biblioteki `data/heap`.

Mając kolejkę zdarzeń, można łatwo zaimplementować opóźnienia bramek logicznych — bramka zamiast od razu modyfikować stan wyjścia, dodaje odpowiednią akcję do kolejki zdarzeń, która wykona się z pewnym opóźnieniem.

Interfejs symulatora

Celem zadania jest dostarczenie modułu eksportującego szereg funkcji pozwalających tworzyć i symulować obwody logiczne. Poniżej znajduje się interfejs modułu wraz z opisem poszczególnych funkcji.

Moduł powinien dostarczać dwie struktury reprezentujące odpowiednio symulację i przewody.

```
(provide sim?
         wire?
```

Struktura reprezentująca symulację (`sim`) powinna zawierać bieżący czas symulacji oraz kolejkę zdarzeń. Czas i kolejka zdarzeń nie powinny być zmiennymi globalnymi, ponieważ użytkownik mógłby chcieć mieć kilka niezależnych od siebie symulacji w jednym programie. Zwróć uwagę, że przewód powinien należeć do tylko jednej symulacji (dlaczego?). Rozważ dodanie pola w strukturze przewodu zawierającego symulację do której dany przewód należy.

Symulacja

Nową symulację można tworzyć za funkcji `make-sim`. Początkowo czas symulacji ustawiony jest na zero.

```
   (contract-out
    [make-sim        (-> sim?)]
```

Aby uruchomić symulację należy zawołać procedurę `sim-wait!`, podając przez ile jednostek czasu symulacja ma działać. W wyniku działania tej funkcji czas symulacji powinien przesunąć się do przodu o podaną wartość, a z kolejki zdarzeń powinny zostać usunięte i wywołane wszystkie akcje, których czas już upłynął (jest niewiększy niż bieżący czas symulacji). Pamiętaj, że a) akcje powinny być wykonane chronologicznie, b) w momencie wywołania akcji czas symulacji powinien być ustawiony na czas przypisany do akcji c) wywołanie akcji może spowodować pojawienie się nowych akcji w kolejce zdarzeń, które być może również będzie trzeba wykonać.

```
    [sim-wait!       (-> sim? positive? void?)]
```

O bieżący czas symulacji można odpytać się funkcją `sim-time`. Dodatkowo można dodawać własne zdarzenia do kolejki zdarzeń za pomocą funkcji `sim-add-action!`, podając za ile jednostek czasu mają się wykonać.

```
[sim-time        (-> sim? real?)]
[sim-add-action! (-> sim? positive? (-> any/c) void?)]
```

Przewody

Nowy przewód konstruujemy funkcją `make-wire` podając do jakiej symulacji on należy.

```
[make-wire       (-> sim? wire?)]
```

Do przewodu można dodawać akcje, które wykonają się przy każdej zmianie wartości przewodu. Aby uniknąć dziwnych stanów początkowych symulacji (jakich?), należy dodawaną akcję od razu wywołać. Można też odpytać się o wartość przewodu i ją ustawić. Pamiętaj, że zmiana wartości przewodu powinna skutkować wykonaniem wszystkich akcji do niego podpiętych.

```
[wire-on-change! (-> wire? (-> any/c) void?)]
[wire-value      (-> wire? boolean?)]
[wire-set!       (-> wire? boolean? void?)]
```

Listy przewodów można łączyć w magistrale przechowujące wartości liczbowe w zapisie binarnym. Implementacja poniższych dwóch funkcji znajduje się już w szablonie rozwiązania.

```
[bus-value (-> (listof wire?) natural?)]
[bus-set!  (-> (listof wire?) natural? void?)]
```

Bramki logiczne

Moduł powinien udostępniać szereg funkcji tworzących bramki logiczne, które reprezentują kolejno negację, koniunkcję, zaprzeczenie koniunkcji, alternatywę, zaprzeczenie alternatywy i alternatywę wykluczającą. Pierwszym argumentem do tych funkcji jest zawsze przewód do którego zostanie podpięte wyjście z bramki. Czas po jakim po zmianie wejścia bramki zmieni się jej wyjście powinien wynosić 1 dla wszystkich bramek, oprócz alternatywy wykluczającej (XOR), dla której ten czas wynosi 2. Aby uniknąć dziwnych sytuacji wyścigu w symulacji (zastanów się jakich?) nowy stan przewodu wyjściowego powinien być obliczany na podstawie stanu przewodów wejściowych w momencie ustawiania wartości wyjścia, a nie zmiany wejścia (choć to drugie było by bardziej realistyczne).

```
[gate-not  (-> wire? wire? void?)]
[gate-and  (-> wire? wire? wire? void?)]
[gate-nand (-> wire? wire? wire? void?)]
[gate-or   (-> wire? wire? wire? void?)]
[gate-nor  (-> wire? wire? wire? void?)]
[gate-xor  (-> wire? wire? wire? void?)]
```

Aby ułatwić tworzenie bardziej skomplikowanych układów, można połączyć tworzenie przewodu z tworzeniem bramki, która tym przewodem steruje. Nowy przewód wyjściowy tworzonej bramki jest wartością zwracaną przez poniższe funkcje.

```
[wire-not  (-> wire? wire?)]
[wire-and  (-> wire? wire? wire?)]
[wire-nand (-> wire? wire? wire?)]
[wire-or   (-> wire? wire? wire?)]
[wire-nor  (-> wire? wire? wire?)]
[wire-xor  (-> wire? wire? wire?)]
```

Z bramek logicznych, które opóźniają sygnały można budować układy mające wewnętrzny stan. Przydatną abstrakcją podczas budowy takich układów jest *przerzutnik* wyzwalany zboczem narastającym sygnału zegarowego.

```
[flip-flop (-> wire? wire? wire? void?)]))
```

Implementacja przerzutnika już znajduje się w szablonie rozwiązania. Składa się on wyłącznie z bramek logicznych.

```
(define (flip-flop out clk data)
  (define sim (wire-sim data))
  (define w1  (make-wire sim))
  (define w2  (make-wire sim))
  (define w3  (wire-nand (wire-and w1 clk) w2))
  (gate-nand w1 clk (wire-nand w2 w1))
  (gate-nand w2 w3 data)
  (gate-nand out w1 (wire-nand out w3)))
```