Elementare Schaltelemente

```
(define (inverter eingabe ausgabe)
  (define (invert-eingabe)
    (let ((neuer-wert
           (logisches-nicht (get-signal eingabe))))
      (verzoegert inverter-verzoegerung
                  (lambda ()
                    (set-signal! ausgabe
                                 neuer-wert)))))
  (add-vorgang! eingabe invert-eingabe))

(define (logisches-nicht s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Ungueltiges Signal"))))

(define (und-gatter a1 a2 ausgabe)
  (define (und-vorgang-prozedur)
    (let ((neuer-wert
           (logisches-und (get-signal a1)
                          (get-signal a2))))
      (verzoegert und-gatter-verzoegerung
                  (lambda ()
                    (set-signal! ausgabe
                                 neuer-wert)))))
  (add-vorgang! a1 und-vorgang-prozedur)
  (add-vorgang! a2 und-vorgang-prozedur))
```

Implementierung der Drähte

```
(define (konstr-draht)

  (let ((signal-wert 0) (vorgang-prozeduren '()))

    (define (set-mein-signal! neuer-wert)
      (if (not (= signal-wert neuer-wert))
          (begin (set! signal-wert neuer-wert)
              (jede-aufrufen vorgang-prozeduren))
          'fertig))

    (define (add-vorgang-prozedur proz)
      (set! vorgang-prozeduren
            (cons proz vorgang-prozeduren))
      (proz))

    (define (zuteilen m)
     (cond
      ((eq? m 'get-signal) signal-wert)
      ((eq? m 'set-signal!) set-mein-signal!)
      ((eq? m 'add-vorgang!) add-vorgang-prozedur)
      (else (error "Unbekannte Operation"))))

    zuteilen))
```

Implementierung der Drähte (Fortsetzung)

```
(define (jede-aufrufen prozeduren)
  (if (null? prozeduren)
      'fertig
      (begin
        ((car prozeduren))
        (jede-aufrufen (cdr prozeduren)))))
```

```
(define (get-signal draht)
  (draht 'get-signal))
```

```
(define (set-signal! draht neuer-wert)
  ((draht 'set-signal!) neuer-wert))
```

```
(define (add-vorgang! draht vorgang-prozedur)
  ((draht 'add-vorgang!) vorgang-prozedur))
```

Benutzung des Zeitplanes

```
;;; Realisierung der Verzoegerung

(define (verzoegert verzoegerung vorgang)
  (hinzufuegen-plan! (+ verzoegerung
                        (aktuelle-zeit der-plan))
                     vorgang
                     der-plan))


;;; Oberste Ebene der Simulation

(define (fortfuehren)
  (if (leerer-plan? der-plan)
      'fertig
      (let ((erster-eintrag
                 (erster-plan-eintrag der-plan)))
        (erster-eintrag)
        (entferne-ersten-plan-eintrag! der-plan)
        (fortfuehren))))
```

Implementierung der Zeitsegmente

```
(define (konstr-zeit-segment zeit warteschlange)
  (cons zeit warteschlange))

(define (segment-zeit s) (car s))

(define (segment-warteschlange s) (cdr s))
```

Implementierung des Zeitplanes

```
(define (konstr-plan) (list 0))

(define (aktuelle-zeit plan) (car plan))

(define (set-aktuelle-zeit! plan zeit)
   (set-car! plan zeit))

(define (segmente plan) (cdr plan))

(define (set-segmente! plan segmente)
  (set-cdr! plan segmente))

(define (erstes-segment plan) (car (segmente plan)))

(define (rest-segmente plan) (cdr (segmente plan)))

(define (leerer-plan? plan) (null? (segmente plan)))
```

Implementierung des Zeitplanes (Fortsetzung)

```
(define (hinzufuegen-plan! zeit vorgang plan)

  (define (gehoert-vor? segmente)
    (or (null? segmente)
        (< zeit (segment-zeit (car segmente)))))

  (define (konstr-neues-zeit-segment zeit vorgang)
    (let ((q (konstr-warteschlange)))
      (hinzufuegen-warteschlange! q vorgang)
      (konstr-zeit-segment zeit q)))

  (define (hinzufuegen-segmente! segmente)
    (if (= (segment-zeit (car segmente)) zeit)
        (hinzufuegen-warteschlange!
            (segment-warteschlange (car segmente))
            vorgang)
        (let ((rest (cdr segmente)))
          (if (gehoert-vor? rest)
              (set-cdr!
                 segmente
                 (cons (konstr-neues-zeit-segment
                                      zeit
                                      vorgang)
                       rest))
              (hinzufuegen-segmente! rest)))))
```

```
  (let ((segmente (segmente plan)))
    (if (gehoert-vor? segmente)
        (set-segmente!
          plan
          (cons (konstr-neues-zeit-segment
                                      zeit
                                      vorgang)
                segmente))
        (hinzufuegen-segmente! segmente))))

(define (entferne-ersten-plan-eintrag! plan)
  (let ((q (segment-warteschlange
                (erstes-segment plan))))
  (entfernen-warteschlange! q)
  (if (leere-warteschlange? q)
    (set-segmente! plan (rest-segmente plan)))))


(define (erster-plan-eintrag plan)
 (if (leerer-plan? plan)
     (error "Plan ist leer in erster-plan-eintrag")
     (let ((seg1 (erstes-segment plan)))
      (set-aktuelle-zeit! plan (segment-zeit seg1))
      (anfang (segment-warteschlange seg1)))))
```

Beobachtung der Simulation

```
;;; Anbringen von Sonden am Draht

(define (sonde name draht)
  (add-vorgang! draht
                (lambda ()
                 (display name)
                 (display (aktuelle-zeit der-plan))
                 (display "  neuer-wert = ")
                 (display (get-signal draht))
                 (newline))))
```