

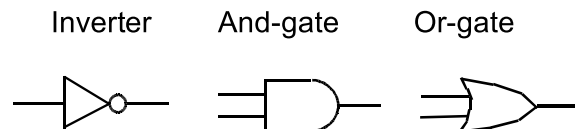
Digital circuit simulator

- Illustrates principle of **event-driven simulation**, which is related to **event-driven programming** (such as in GUIs)
- A good example of combining several concepts
 - builds a language (of circuits)
 - models with objects
 - uses queues to implement an agenda

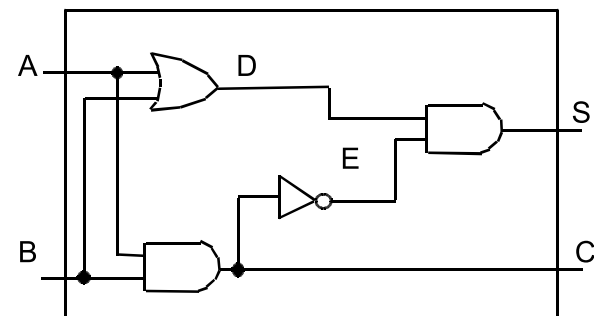
Digital circuits

- Composed of **wires** and **function boxes**
- Digital signals are 0 or 1
- Function boxes have time delays
- Function boxes can be composed of other function boxes

Primitive function boxes



Half-adder



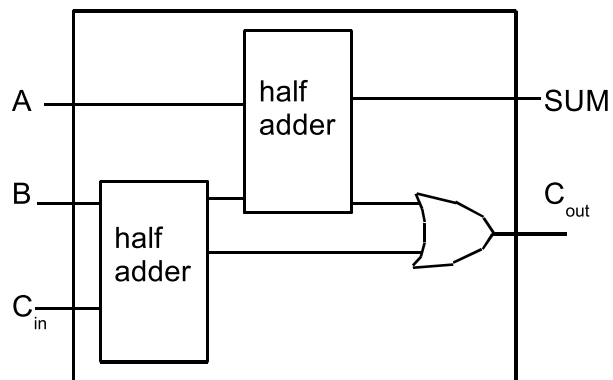
Circuit building language

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define s (make-wire))
(half-adder a b s c)
```

Building half-adder

```
(define (half-adder a b s c)
  (let ((d (make-wire))
        (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

Full adder



Building full adder

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

Wires as objects

- A wire carries a signal
- The signal is determined by the output of a function box that the wire is connected to
- A wire activates the function boxes to which it is connected as input
- Selector: `(get-signal <wire>)`
- Mutators:
`(set-signal! <wire> <value>)`
`(add-action! <wire> <0-arg proc>)`

Building inverters

```
(define (inverter input output)
  (define (invert-input)
    (let ((val (logical-not
                (get-signal input))))
      (after-delay inverter-delay
        (lambda ()
          (set-signal! output
            val))))))
  (add-action! input invert-input)
  'ok)
```

Logical-not

```
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else
         (error "invalid s" s))))
```

Building and-gate

```
(define (and-gate s1 s2 output)
  (define (and-action-proc)
    (let ((val
           (logical-and (get-signal s1)
                        (get-signal s2))))
      (after-delay and-gate-delay
        (lambda ()
          (set-signal! output val))))))
  (add-action! s1 and-action-proc)
  (add-action! s2 and-action-proc)
  'ok)
```

Wire constructor

```
(define (make-wire)
  (let ((sig-val 0)(action-procs ()))
    (define (set-my-sig! new-val)
      (if (not (= sig-val new-val))
          (begin (set! sig-val new-val)
                  (call-each action-procs))
          'done-already))
    (define (accept-action-proc! proc)
      (set! action-procs (cons proc action-procs))
      (proc)))
```

continued

```
(define (dispatch m)
  (cond ((eq? m 'get-signal) sig-val)
        ((eq? m 'set-signal!)
         set-my-sig!)
        ((eq? m 'add-action!)
         accept-action-proc!)
        (else
         (error "unknown op--WIRE" m))))
dispatch))
```

Call-each procedure

```
(define (call-each procedures)
  (if (null? procedures)
      'done
      (begin
         ((car procedures))
         (call-each
          (cdr procedures))))))
```

Conventional functional interface

```
(define (get-signal wire)
  (wire 'get-signal))
(define (set-signal! wire new-val)
  ((wire 'set-signal!) new-val))
(define (add-action! wire action-proc)
  ((wire 'add-action!) action-proc))
```

Time-delayed procedure calls (uses agenda)

```
(make-agenda)
(empty-agenda? <agenda>)
(first-agenda-item <agenda>)
(remove-first-agenda-item! <agenda>)
(add-to-agenda! <time>
               <action>
               <agenda>)
(current-time <agenda>)
```

Posting future procedure calls

```
(define the-agenda (make-agenda))
(define (after-delay delay action)
  (add-to-agenda!
   (+ delay
      (current-time the-agenda))
   action
   the-agenda))
```

The main simulation loop

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item
             (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

A wire probe

```
(define (probe name-of-wire wire)
  (add-action!
   wire
   (lambda ()
     (newline)
     (display name-of-wire)
     (display " ")
     (display (current-time the-agenda))
     (display " New-value = ")
     (display (get-signal wire))))))
```

Simulation of half-adder

```
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))
(probe 'sum sum)
(probe 'carry carry)
(half-adder input-1 input-2 sum carry)
```

Simulation trace

```
(set-signal! input-1 1)
done
(propagate)
sum 8 New-value = 1
done
(set-signal! input-2 1)
done
(propagate)
carry 11 New-value = 1
sum 16 New-value = 0
```

continued

```
(set-signal! input-1 0)
done
(propagate)
carry 19 New-value = 0
sum 24 New-value = 1
```

The agenda

- Has current time and an ordered list of time segments
- Agenda looks like
(<cur-time> <time-seg1> <time-seg2> ...)
- Each time segment has a time and a queue of the actions that are to take place at that time

Time segment implementation

```
(define (make-time-segment time queue)
  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))
```

Agenda implementation

```
(define (make-agenda) (list 0))
(define (current-time agenda) (car agenda))
(define (set-current-time! agenda time)
  (set-car! agenda time))
(define (segments agenda) (cdr agenda))
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

add-to-agenda!

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
```

continued

```
(define (add-to-segments! segments)
  (if (= (segment-time (car segments)) time)
      (insert-queue!
       (segment-queue (car segments))
       action)
```

continued 2

```
(let ((rest (cdr segments)))
  (if (belongs-before? rest)
      (set-cdr!
        segments
        (cons (make-new-time-segment
                time
                action)
              (cdr segments)))
      (add-to-segments! rest))))
```

continued 3

```
(let ((segments (segments agenda)))
  (if (belongs-before? segments)
      (set-segments!
        agenda
        (cons (make-new-time-segment time action)
              segments))
      (add-to-segments! segments))))
```

remove-first-agenda-item!

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue
            (car (segments agenda)))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda
                        (cdr (segments agenda))))))
```

first-agenda-item

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "empty agenda--FIRST-AGENDA-ITEM")
      (let ((first-seg (car (segments agenda))))
        (set-current-time!
          agenda
          (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```