# Simulador de Circuitos Digitais

Carla Parreiras e Fernanda Scovino

December 15, 2016

**Abstract**

## 1 Introdução

O projeto consiste na criação de um programa de simulação de circuitos lógicos digitais, um tipo de simulação descrita como event-driven simulation, no qual os eventos (mudanças de estado do sistema) ocorrem de forma encadeada, e não contínua.
Tomamos como base para esse trabalho a subseção 3.3.4 do livro Structure and Interpretation of Computer Programs, a partir da qual montaremos o programa e resolveremos os exercícios propostos e acrescentamos ao projeto a implementação de um Codificador Decimal-BCD(Binary Coded Decimal).

## 2 Elementos de um circuito digital

Nessa seção apresentaremos em detalhes cada elemento que compõe um circuito digital

### 2.1 Wires

Wires são fios por onde a corrente irá passar. Esses fios carregam sinais digitais, que podem assumir os valores 0 (sem corrente) ou 1 (com corrente). Abaixo temos o código que usamos para definir um fio:

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '()))
    (define (set-my-signal! new-value) ;toda vez que o wire muda de sinal, chama as ações.
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures))
          'done))
    (define (accept-action-procedure! proc)
      (set! action-procedures (mcons proc action-procedures))
      (proc))
    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure!)
            (else (error "Unknown operation - WIRE" m))))
    dispatch))
```

### 2.2 Blocos funcionais

Os blocos funcionai são objetos que conectam os fios que carregam o sinal do input para os fios que carregam o sinal do output, sendo esse sinal transformado a partir do teste lógico específico do bloco. O livro apresenta os três primeiro, entretanto implementamos outros modelos de blocos funcionais. Esses blocos são classificados em:

1. **Inverter (NOT)**

   Esse bloco simboliza a operação complementar, que só pode ser realizada sobre uma variável por vez, por isso o inverter possui somente uma entrada e uma saída, invertendo o sinal da entrada.

| Input | Output |
|---|---|
| 0 | **1** |
| 1 | **0** |

Table 1: Teste lógico - NOT-gate

   Código referente ao Inverter:

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input)
  'ok)
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))
```

2. **AND-gate**

   Esse bloco possui o valor lógico **E**, tendo no mínimo 2 entradas e uma única saída. Ele irá gerar como output o sinal 1 se, e somente se, todos os sinais da entrada forem iguais a 1.

| Input1 | Input2 | Output |
|---|---|---|
| 1 | 1 | **1** |
| 0 | 1 | **0** |
| 1 | 0 | **0** |
| 0 | 0 | **0** |

Table 2: Teste lógico - AND-gate

   Código referente ao AND-gate:

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok)
(define (logical-and s1 s2)
  (if (= s1 s2 1)
      1
      0))
```

3. **OR-gate**

   Esse bloco possui o valor lógico **OU** (inclusivo), tendo no mínimo 2 entradas e uma única saída. Ele irá gerar como output o sinal 1 se pelo menos um dos sinais da entrada for 1.

| Input1 | Input2 | Output |
|---|---|---|
| 1 | 1 | **1** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 0 | 0 | **0** |

Table 3: Teste lógico - OR-gate

Código referente ao OR-gate:

```
(define (or-gate a1 a2 output)
  (define (or-action-procedure)
    (let ((new-value
           (logical-or (get-signal a1) (get-signal a2))))
      (after-delay or-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure)
  'ok)
(define (logical-or s1 s2)
  (if (= s1 s2 0)
      0
      1))
```

4. **XOR-gate**

   Esse bloco possui o valor lógico **OU** (exclusivo), tendo no mínimo 2 entradas e uma única saída. Ele irá gerar como output o sinal 1 se pelo menos um dos sinais da entrada for igual a 1, excluíndo o caso em que todas as entradas recebem o valor 1.

| Input1 | Input2 | Output |
|---|---|---|
| 1 | 1 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 0 | 0 | **0** |

Table 4: Teste lógico - XOR-gate

Código referente ao XOR-gate:

```
(define (xor-gate a1 a2 output)
  (define (xor-action-procedure)
    (let ((new-value
           (logical-xor (get-signal a1) (get-signal a2))))
      (after-delay xor-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 xor-action-procedure)
  (add-action! a2 xor-action-procedure)
  'ok)
(define (logical-xor s1 s2)
  (if (and (= s1 s2 0)(= s1 s2 1))
      0
      1))
```

5. **NOR-gate**

   Esse bloco pussui o valor lógico de **negação** do **OU**, tendo no mínimo 2 entradas e uma única saída. Ele irá gerar como output o sinal 1 se, e somente se, nenhum sinal de entrada

for igual a 1.

| Input1 | Input2 | Output |
|---|---|---|
| 1 | 1 | **0** |
| 0 | 1 | **0** |
| 1 | 0 | **0** |
| 0 | 0 | **1** |

Table 5: Teste lógico - NOR-gate

Código referente ao NOR-gate:

```
(define (nor-gate a1 a2 output)
  (define (nor-action-procedure)
    (let ((new-value
           (logical-nor (get-signal a1) (get-signal a2))))
      (after-delay nor-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 nor-action-procedure)
  (add-action! a2 nor-action-procedure)
  'ok)
(define (logical-nor s1 s2)
  (if (= s1 s2 0)
      1
      0))
```

6. **NAND-gate** Esse bloco possui o valor lógico de NEGAÇÃO do E, tendo no mínimo 2 entradas e uma única saída. Ele irá gerar como output o sinal 1 se, pelo menos um dos sinais de entrada for igual a 0.

| Input1 | Input2 | Output |
|---|---|---|
| 1 | 1 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 0 | 0 | **1** |

Table 6: Teste lógico - NAND-gate

Código referente ao NAND-gate:

```
(define (nand-gate a1 a2 output)
  (define (nand-action-procedure)
    (let ((new-value
           (logical-nand (get-signal a1) (get-signal a2))))
      (after-delay nand-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 nand-action-procedure)
  (add-action! a2 nand-action-procedure)
  'ok)
(define (logical-nand s1 s2)
  (if (= s1 s2 1)
      0
      1))
```

## 2.3 Operadores

Nos blocos funcionais existem operadores primitivos habilitados para extrair o sinal do wire e modificar o valor do sinal de acordo com o procedimento chamado. São esses:

1. **get-sinal**

   Retorna o valor corrente do sinal do fio.

2. **set-sinal!**

   Muda o valor do sinal do fio.

3. **add-action!**

   Afirma que o procedimento designado deve ser executado sempre que o sinal no fio muda de valor.