Simulador de Circuitos Digitais

Carla Parreiras e Fernanda Scovino

December 16, 2016

1 Introdução

O projeto foi dividido em duas partes, a primeira 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.O que caracteriza um circuito digital é a utilização de sinais elétricos em somente dois níveis de tensão (ou corrente) para a representação do valor de um dígito binário: "0" ou "1"; originando daí a denominação "digital". Tal fato auxilia a representação do estado de dispositivos que trabalham em dois níveis ou estados, tais como nível–alto/nível-baixo, ligado/desligado (on/off), verdadeiro/falso, fechado/aberto, presença/ausência, etc..., como também em virtude da vantagem que estes dispositivos levam em termos de confiabilidade, além da representação de valores numéricos no sistema binário. A segunda parte do projeto foi a implementação em Racket do circuito do Codificador BCD (Binary Coded Decimal), que recebe um número em decimal e codifica em binário. Ele que é composto por quatro bits, tendo cada bit um alor equivalente ao do sistema numérico binário. Assim, o código representa cada um dos números decimais de 0 a 9 no sistema binário.

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 (or (= 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))
```

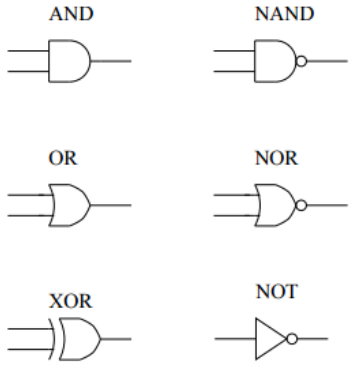

Figure 1: Blocos Funcionais

2.3 Adicionadores Binários

Os Adicionadores Binários (Binary Adder) são circuitos lógicos combinacionais, construídos por alguns blocos funcionais, que permitem a soma de dois ou mais sinais do circuito.

1. **Half-Adder**

   O half-adder recebe dois sinais binários por wires (A e B) e produz duas saídas, uma soma (sum) e um carry-out (carry).

   Código referente ao 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))
```

A	B	Sum	Carry
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

Table 7: Tabela verdade - Half-Adder

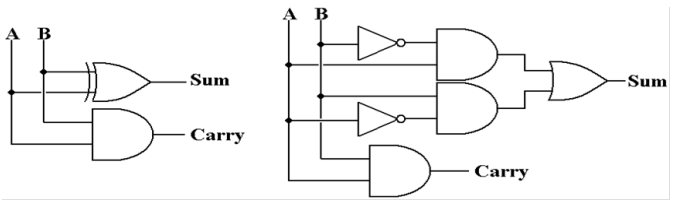

Figure 2: Half-Adder

2. **Full-Adder**

O full-adder é composto por uma combinação de Half-Adders. Ele recebe três sinais binários por wires (A, B e C) e produz duas saídas, uma soma (sum) e um carry-out.

Código referente ao 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))
```

A	B	C	Sum	Carry
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

Table 8: Tabela verdade - Full-Adder

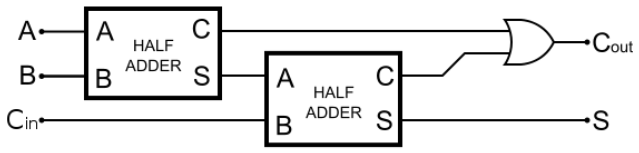

Figure 3: Full-Adder

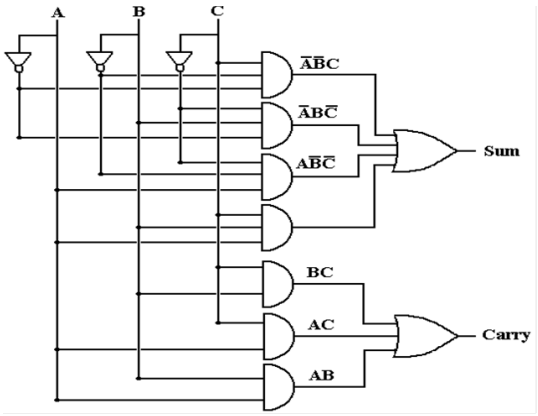

Figure 4: Full-Adder

3. **Ripple-Carry-Adder**

   O exercício 3.30 pede para que se crie um novo Binary Adder como uma função que verifica se o cdr das listas A e B (representadas por list-a e list-b) está vazio e, caso verdadeiro, muda o sinal de c (carry) para 0; caso falso, executa a função full-adder com o primeiro valor das listas A, B e S, o wire c e um novo wire c-out, que vai receber o valor da soma que "sobe" uma casa, e faz uma recursão, chamando a função ripple-carry-adder com o cdr das listas A, B e S, e c-out.Garantindo, assim, que estamos somando o valor Ak com Bk e que passamos o valor Ck para a próxima "casa", ou seja, para ser adicionado a Ak+1 e Bk+1.

   Código referente ao Ripple-Carry-Adder:

```
(define (ripple-carry-adder list-a list-b list-s c)
  (let ((c-out (make-wire)))
    (if (null? (cdr list-a))
        (set-signal! c 0)
        (ripple-carry-adder (cdr list-a) (cdr list-b) (cdr list-s) c-out))
    (full-adder (car list-a) (car list-b) c-out (car list-s) c)))
```

2.4 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.

3 Representando fios

Os fios serão tratados como objetos computacionais que possuirão duas variáveis de estado: o sinal, que inicialmente é 0 para todos, e uma lista de procedimentos de aç eos que serão executadas quando houver mudança de valor do sinal. A implementação dos códigos de manipulação dos fios é feita usando o estilo de passagem de mensagens juntamente com o procedimento dispach. Abaixo temos implementação dos fios:

```
(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))
          (display "")))
    (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))
```

Os fios foram modelados de forma a serem modificados por atribuições. Quando um novo fio é criado, um novo conjunto de variáveis de estado é alocado (pela expressão let em make-wire) e um novo procedimento despach é construído e retornado, capturando o ambiente com as novas variáveis de estado. Como os fios são todos conectados, cada alteração feita em um, acaba carregando essa modificação para seus fios vizinhos chamando os procedimentos de ação fornecidos a ele quando as conexões forem estabelecidas.

4 Construção da Agenda

A agenda mantém uma estrutura de dados que contém o cronograma das ações, denominada pelo livro por the-agenda. As operações da agenda são:

```
(define the-agenda (make-agenda))
```

1. **make-agenda**

   Retorna uma agenda vazia.

   ```
   (define (make-agenda) (mcons 0 '()))
   ```

2. **empty-agenda?**

   É verdade se a agenda for vazia.

   ```
   (define (empty-agenda? agenda)
     (null? (segments agenda)))
   ```

3. **first-agenda-item**

   Retorna o primeiro valor da agenda.

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

4. **remove-first-agenda-item!**

   Remove o primeiro item da agenda.

   ```
   (define (remove-first-agenda-item! agenda)
     (let ((q (segment-queue (first-segment agenda))))
       (begin (delete-queue! q)
              (if (empty-queue? q)
                  (set-segments! agenda (rest-segments agenda))
                  'pass))))
   ```

5. **add-to-agenda!**

   Modifica a agenda adicionando a ação a ser executada em uma hora específica.

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

```
(define (add-to-segments! segments)
  (if (= (segment-time (mcar segments)) time)
      (insert-queue! (segment-queue (mcar segments))
                     action)
      (let ((rest (mcdr segments)))
        (if (belongs-before? rest)
            (set-mcdr!
             segments
             (mcons (make-new-time-segment time action)
                    (mcdr segments)))
            (add-to-segments! rest)))))
(let ((segments (segments agenda)))
  (if (belongs-before? segments)
      (set-segments!
       agenda
       (mcons (make-new-time-segment time action)
              segments))
      (add-to-segments! segments))))
```

6. **current-time**

   Retorna o tempo de simulação atual.

   ```
   (define (current-time agenda) (mcar agenda))
   ```

5 Implementação da Simulação e da Agenda

Nessa seção implementaremos a simulação e a agenda.O procedimento a seguir, que coloca um probe no fio, mostra o simulador em ação. O probe diz ao fio que, sempre que seu sinal muda de valor, ele deve imprimir o novo valor de sinal, juntamente com a hora atual e um nome que identifica o fio:

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

A função propagate instancia a agenda de ações, executando as mesmas em sequência. As ações só ocorrem caso aja alguma mudança de sinal dos wires em questão.

```
(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))))
```

Com todo o código necessário, a criação de um simulador pode ser feita. Primeiro deve-se inicializar a agende e especificar os delays para cada bloco funcional, assim cada função saberá a hora em que será executada. Depois define-se os wires e coloca-se o prob nos que precisarem. Por fim conectamos os fios em um circuito e fazemos as operaçãoe para esse circuito.

A implementação da agenda se dá a partir dos times-segments, que é um segmento que consistem de um número, tempo em que o procedimento será executado, e uma queue, que contém os procedimentos que estão programados para serem realizados no tempo.

```
(define (make-time-segment time queue)
  (mcons time queue))
(define (segment-time s) (mcar s))
(define (segment-queue s) (mcdr s))
```

Cada conjunto de ações de um segment da agenda é implementado como um queue pois é importante que as ações de cada tempo ocorram na ordem em que foram adicionadas, ou seja, seguindo a ordem FIFO (fisrt in, first out).

6 Codificador BCD

Os codificadores são circuitos lógicos dedicados que convertem informações alfanuméricas ou de controle para um código determinado. A maior aplicação dos codificadores está na conversão de dados de um sistema de interface com o usuário, como o teclado, para o código com o qual o respectivo sistema digital trabalha. Como já foi dito na introdução, escolhemos para o projeto o Codificador BCDque recebe um número em decimal e codifica-o em binário. Esse circuito possui 9 entradas e 4 saídas e está representado gráficamente abaixo:

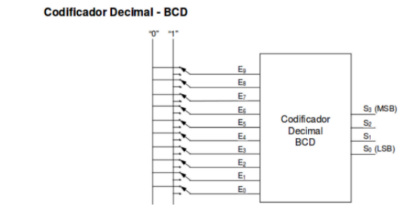

Figure 5: Circuito BCD

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figure 6: Sistema de codificação

O circuito Binary Coded Decimal possui 10 inputs (que representam os números de 0 a 9) e 4 outputs. Dessa forma, criasse o circuito abaixo:

```
(require "manipulacao-wire.rkt"
         "blocos-logicos.rkt")
(provide cod-bcd-8421)

(define (cod-bcd-8421 in out3 out2 out1 out0) ;codificador BCD-8421
  (let ((in0 (list-ref in 0))
        (in1 (list-ref in 1))
        (in2 (list-ref in 2))
        (in3 (list-ref in 3))
        (in4 (list-ref in 4))
        (in5 (list-ref in 5))
        (in6 (list-ref in 6))
        (in7 (list-ref in 7))
        (in8 (list-ref in 8))
        (in9 (list-ref in 9))
        (c1 (make-wire))
        (c2 (make-wire))
        (c3 (make-wire))
        (c4 (make-wire))
        (c5 (make-wire))
        (c6 (make-wire)))
    (or-gate in4 in5 c1)
    (or-gate in6 in7 c2)
    (or-gate in2 in3 c3)
    (or-gate in7 in9 c4)
    (or-gate in5 in3 c5)
    (or-gate in1 c4 c6)
    (or-gate c6 c5 out0)
    (or-gate c1 c2 out2)
    (or-gate c2 c3 out1)
    (or-gate in8 in9 out3)
    'ok))
```