A Simulator for Digital Circuits

Shawn Mathew and Amy Wong

CSc 33500 - Programming Language Paradigms

Fall Semester 2015

Professor Douglas Troeger

15 December 2015

Tables of Contents

Introduction

In the technologically driven world of today, the ability to design complex digital systems, such as computers, is an important skill. Computer simulations of circuit designs for the digital systems are often used to showcase the complex network connections they contain. In this project, a system is designed to perform digital logic simulations. This system represents a program that uses an event-driven simulation. In other words, actions called events will trigger further events that will happen at a later time. In turn, these events can further trigger more events, and so on.

Based on the model given by the textbook, Structure and Interpretation of Computer Programs, written by Harold Abelson and Gerald Jay Sussman with Julie Sussman, a prototype was created to simulate circuits that are implemented in real life with various gates. The computation model for each circuit is composed of objects, such as wires, that correspond to the components from which the circuit is constructed from. The program contains various types of digital function boxes that represent the circuit gates. Using the simple function boxes as building blocks, more complex circuit representations such as a d-latch, d flip-flop and register are made. A clock function was also generated as it is vital to the operation of synchronizing a circuit. Propagation delay that is found in real circuits was also incorporated into the program.

We specifically chose a simulator for digital circuits as our final project because we had rigorously used digital circuits in a prior course at The City College of New York called Fundamental of Computer Systems (CSc 211). We were interested in how the digital circuits we

had implemented through Quartus, a logic design software, were typified and implemented through functional programming, more specifically in the language of RSR5.

Accomplishments

Upon starting the project, we proceeded to read through the textbook to gain a better understanding of the complex simulator. Questions that were posed by the textbook were completed such as creating an or-gate as seen in the code. As we processed and ran the provided source code, it appeared to have a small amount of bugs. Many of these involved typos and other small things were of this nature as well. Unfortunately, there also appeared to be an issue with the textbook's use of queues. In their definition of the queue, a function named front-ptr is used to access the list of elements in the queue. However, the agenda and time segments expected this function to return the first element in the list instead, which is also a function. This issue was fixed simply by running the list through the car function when it was called.

One of the first improvements attempted on the simulator was to create a method to make the program run in a more time related fashion. The obvious place to add this would be in the propagate function where the propagation delay is taken into account. The function compares the the current time to a time in the first segment. If the times do not match, then the current time is set equal to the time of the first segment. There lies a problem as this method completely skips over all the numbers in between. For instance, a time difference with a value of 3 will seem be the same as a time difference of 100. Hence, another approach was selected where a sleep function was added into the propagation function. This, however, brought up syntax errors. Upon

further research, it appears that the sleep function does not work in R5RS version of scheme. The only way to get around this would be to make calls to the operating system, which would be a project of its own.

Another additional item in the program is a clock. The clock appeared to be one of the items that was difficult to implement into the system. The idea of clock started of by being rather simple. The initial idea was to create a specific type of wire that would have its functions varied as it would mimic a clock. This however failed, as multiple problems would arise whenever a connection was made by adding an action. There would be extra calls to the clock for changing its values. This causes a negative reaction on the memory. In conclusion, it was decided to treat the clock like a gate. The only issue with this was that the clock would never terminate. The initial solution to this was to see if there was another item in the agenda. A function called second-agenda-item? was created for that purpose. This, in the end, was removed from the clock. When another gate or circuit relied on clock as an input, there would always be a function call after the clock. This is due to the fact that altering the clock value forced other wires to be reevaluated and more items were created on the agenda. In the end, we decided to embrace this due to the fact that the clock is not meant to end until the circuit itself is turned off. In order to prevent our program from executing an infinite loop, a limiting case was added to the propagate function. This made it so that the clock and other functions would make the program run a finite amount of steps.

In addition to the provided and-gate and inverter, we implemented the rest of the gates known in digital circuits, which are the or-gate, nor-gate, and nand-gate. Since gates are the most basic elements in circuitry. When connected with one another a circuit is formed, primitive

digital logical functions for these gates had to be created. By creating functions for all the gates, an endless amount of circuits can be produced. All the logical functions were conformed to a format the textbook provided for the and-gate function. The given inverter function was renamed to be the not-gate and redone to follow the format of the and-gate.

All the gates, excluding the not-gate, contain a constraint where they can only accept two input wires. In actuality, the logic gates should be able to accept as many inputs as possible and still give an output wire that corresponds the logical computation. However, for simplicity's sake, this wasn't done. A method for computing the infinite amount of input wires would be take a list of input wires as a parameter along with one output wire for an iterative function. The auxiliary function will take two arguments, a list of input wires and a result signal value. The iterative call in the function will compute the new result signal value by doing the logical computation of the current result signal value with the signal value of the wire that is the car of the list. The iterative call will also take the cdr of the list as the list for the next call. Termination of the iterative auxiliary function occurs when the list of wires is null and the value of the output wire signal is set to the result signal value. The gate function will initialize the call to the auxiliary function by setting the result signal value as the wire signal value of the car of the list and the cdr of list as the list argument.

After creating the primitive digital logical functions for the gates, different circuits were designed such as a d-latch, d flip-flop, and a 4-bit shift register. We originally designed each circuit with a format similar to the gates with the exception that the gate logical functions were used in the circuit's logical function. This was the wrong implementation because circuits are made up of gates. Therefore, the circuit designs should reflect this by using the actual gate

functions themselves, and not by using their logical functions. In fact, the gate functions already took care of the procedures that should occur when a signal in the network of wires changed, so there was no point in adding the procedure in the circuit function. Hence, the circuit functions were redesigned to use to the gates or simpler circuit functions. The new designs show a better view of how circuits are constructed of simpler elements in a particular order in real life.

One of the major flaws that the textbook's system lacks is a simple way to display the values of the wires. The textbook offered a probe function that displayed the signal value of a wire if it is changed. This, however, clogged up the screen with lots of numbers and names to the point where it would take extra time to decypher the output for the values. To fix this, a display-group function was created. The functionality behind the procedure was that wires are usually associated with one another and users would want to see them side by side. The function takes in any number of names and wires and outputs the display anytime one of the values is altered in the group. There is one small bug that couldn't be worked around when calling the display-group. Due to nature of the add-action! function of the wires, the display repeats itself multiple times when it is called. Apart from that, the function provides a much simpler way to organize the wires and makes it easier to draw conclusions from the simulation.

Technology

Data Structures and Objects

In order to gain a better understanding of the overall complex program of the simulator for digital circuits. The program is broken down into smaller components consisting of data

structures and objects. The data structures mentioned in detail are primitive function box, circuits, queue, segment, and the agenda. The concepts surrounding in the objects of digital circuitry are represented as functions in the program. The objects created in the simulator include wire, gates, and lastly, circuits.

Wire

Circuits are connected through input and output wires, which contain digital signals. These digital signals can only carry one of two possible values, 0 and 1, at any given moment. The construction of wires in the simulator is done in a procedure called make-wire. Each wire in the simulation has two local state variables, a signal-value and a collection of action-procedures contained in a list, which are called every time the wire signal changes value. A newly created wire object will have its signal-value initialized to 0 and an empty list for the action-procedures. A message-passing style is used to implement the wire by tying the local procedures together with a final local procedure named dispatch which is called at the end of make-wire. The dispatch procedure is responsible for selecting and executing the appropriate local operation in the object.

There are three choices of operations that the dispatch function computes, which are get-signal, set-signal!, and add-action!. If the argument passed is not one of these operations, then an error message is displayed for the wire object. If get-signal is passed, then the value of the wire signal is given. The argument set-signal! calls on the local procedure set-my-signal!. This function determines if the new signal value changes the signal on the wire. If it does, then the list of action-procedures associated with the wire will be executed based on their ordering in the list. The add-action! operation will run the newly given procedure after adding it to the

action-procedures collection. All of the local procedures make it possible for a wire object to communicate a change to its neighboring wires. This done by calling the action procedures provided to it when the wire connections were established.

Primitive Function Box

The and-gate, inverter, or-gate, nor-gate, and nand-gate are categorized as primitive digital logic functions and implemented by primitive function boxes. They are considered to be the most basic elements in circuits. When the elements are combined together in a certain order by connecting their wires, they form procedures that represent complex circuits in the simulator. Even though the primitive function boxes are the most basic elements, they have the power to implement a change in the entire circuit by simply changing a signal on one wire. This is the reasoning behind why this program is considered to be an event-driven simulator. The changing of one wire signal can trigger the changing of signals of further wires that are connected to it. The function boxes are built by using the operations of the wire object (get-signal, set-signal!, and add-action!). Also, in each of the primitive logic functions, a procedure called after-delay is used to take in a time delay. After the time delay is executed, the procedure of setting the output wire to its new signal is done.

Gates

Using the primitive function boxes and the procedures associated with them, primitive digital logic functions are created. As mentioned before, the primitive logic functions are: not-gate, and-gate, or-gate, nor-gate, and nand-gate. Each circuit created in the program is represented by a function. The not-gate function is simplest primitive function in the digital logic simulator. It takes an input wire and an output wire as parameters. A connection between the

input wire and output wire is created through the gate by calling add-action!. This will associate the input wire with the procedure of the not-gate and will run whenever the signal of the input wire changes value. The not-gate procedure uses the signal of the input wire and computes the logical-not value. The computed value is then set as the new output signal value after the specified time delay for this gate occurs.

The other primitive gate functions follow a similar format in their definitions. The action procedure in each function is added to the list of action procedures for each input wire. Thus, the logical value of the input wires is computed and set as the signal of the output wire after the gate time delay passes. However, the remaining gates have a slight difference as they take in two input wires as parameters instead of one. Therefore, the add-action procedure must be executed if either of the inputs to the gates changes values.

Queue

The queue that was presented in the textbook has an interesting data structure. It keeps track of the list of items and the last item. The data structure itself is a list within a list. The innermost list contains all the elements that are contained in the queue. This is the first element in the outer list of the queue. The second element is a reference to the last element in the inner list. Since the last element in the outer list is a reference to the last element in the inner list, changing that value would also alter the cdr of that element in the inner list. This is taken advantage of in the case where additions are made to a nonempty queue. By adding to the reference of the last element, the inner list also gets the addition. From there, all that is left to do is to update the reference to new last element.

Segments

The next data structure that is used is the segment. This data structure is not as complex as the queue, but it is an integral part of the agenda. The segment is made up of a list containing a numerical value and a queue. The numerical value represents a time and the queue is meant to hold a list of actions that is called at the corresponding time. Hence, this data structure is not too complex as it only has selectors.

The Agenda

The most integral data structure to the circuits is the agenda. The agenda ensures that all the wires in the circuit changes value at the correct time. This is how time is interpreted for each circuit. Furthermore, it is also one of the methods for implementing propagation delay into the system. The agenda is comprised of a list. This list holds one numerical value and a number of segments. The numerical value is always seen at the beginning of the list and is the current time the agenda is at. The rest of list is meant to contain segments that will be called upon when the current time reaches the value of the time of that segment.

One of the important functions in the agenda is the add-to-agenda! function. As its name states, it is meant to add a function to an agenda. The inputs for this function are a time, the action that is being added, and the agenda that is being operated on. When the agenda is called, there are a few cases that are checked in order to correctly place the new segments of actions to the procedure. The first case looks at where the new action belongs, either before the first segment or it needs to be added to another or in between other segments. The if statement in the function checks which case the input falls under. If it falls under the first condition the segment is simply added to at the beginning on the agenda. If this is not the case, a recursive function is

called to put the action in its correct position. The recursive function is called add-to-segments. In this function there are three cases that need to be addressed. The first instance determines whether the time matches that of the first segment in the list of segments. If this is true, then the action is simply added to the queue in that time segment. The next case is that the time is before the following segment in the list. In this case, a new segment must be created and it needs to be added to the list using set-cdr!. This is done due to the fact that we are using the following segment and the cdr can just be set to the cons of the action to the cdr of the segments list. The last case is that the time is either equal to or greater than the following segments time. In this case, a recursive call is used on the cdr of segments.

Propagate

One of the main functions that makes the system work properly is the propagate function. The function, along with an agenda, is used to create the propagation delay that is involved in each gate the the wires are inputted. It also makes sure that functions get called in the correct order. The propagate function starts by updating the time to match to the lowest time in the agenda. Afterwards, it calls the first function in the queue for the segment that has the same time as the current time. When function completes its run, it is removed from the queue and propagate gets called again on this new agenda. The original base case was simply when the agenda was empty. This alone, however, did not suffice when the clock was a part of the system. The clock alters the value of the inputted wire with an inputted delay. Since the calling of this function added another item onto the agenda, the agenda would never be empty and the program would never stop. In order to prevent this, an additional base case was added where if the current time passes a certain value, the propagate function stops.

When initially testing the latches, there appeared to be many problems with it, even though the latch definition was correct. The problem found was that the outputs would be constantly varying. The issue occurs because the input values were changing faster than the latch could finish evaluating the wires at the end. This is especially important to those circuits that have output wires that are fed back into a different piece of the circuit as inputs. All the wires should have to be set within a circuit before the input values are changed. This provoked us to alter the clock function to have a delay input, so that it could give the circuits time to finish evaluating the output wires and intermediate wires.

Circuits

Using the primitive digital logic functions, more complex circuit functions can be created. In addition to the textbook provided adder circuits, a d-latch, a d flip-flop, and a 4-bit register were created. A half adder circuit contains two input signals — each representing a binary bit — and the output consists of a sum signal and a carry signal for overflow that would be used in the next digit if a multi-digit addition was done. The half-adder procedure takes in two input wires and two output wires as the external wire attachments for the circuit representation. The circuit is composed of primitive gates, and in order to connect the wire signals being passed through them, new local wire variables had to be created. The half adder circuit can then be used as a building block for creating a more complex full adder circuit. The full-adder function is comprised of two half-adder functions, and an or-gate. One local wire is constructed to connect one of the outputs from the first half-adder to the second half-adder. Also, two other local wires are constructed to hold the carry signals for each half-adder and are used in determining the carry signal for the full-adder.

At first, we had planned on creating one extra circuit to the simulator, a d flip-flop. However, we realized that a d flip-flop is composed of two d-latches. Therefore, it only made sense to construct a d-latch out of primitive gates and use it as a building block for the d flip-flop procedure. Furthermore, the d flip-flop can be used as an element of a more complex circuit, a register. All of these complex circuits require a clock input as all the changes to their wire signals are synchronized by a clock signal.

The d-latch procedure is comprised of two inputs, a wire and a clock, and two output wires. The signals on these output wires should always be inverted values of each other. Similar to how the half-adder was constructed, interior wires were constructed to connect the gates in the d-latch procedure. The functionality of this d-latch procedure is that the output signal value will be changed to match the input signal value whenever the clock input signal has a value of 1. Once the d-latch was completed a d flip-flop is rather simple. It is made up of two d-latches and a not gate. The d flip-flop circuit changes the output wire signal value to match the input wire signal value whenever the clock signal changes from 0 to 1.

Lastly, a 4-bit shift register was built using four d flip-flops. Each d flip-flop holds one bit of binary information. Hence, the reason for its name. This circuit is different from the rest because the output is a list of four wires called qlist. The output wires are used to connect the d flip-flops together by linking the output of a d flip flop to the input value of the next d flip flop. Every time the clock signal value changes from 0 to 1, the values of the output wire signals will shift to the right and the rightmost output value will receive its new signal value based on the input wire. Anothering interesting find, is that the qlist is a type of data abstraction. It has a constructor, which builds the list of 4 wires, selectors for getting wires at certain positions in the

list and a function for setting the wires signals in the list. There is also a specialized display function for the qlist in displaying the wires and their signal values.

Conclusion

This project was a very fulfilling one. We implemented the simulator provided by the text and added improvement to it. All kinds of circuits can be simulated in the program that was created. There are still a few things that we would loved to have added to the system. One of the major ones is a more noticeable time difference in the propagate function. A prototype for the function was created, but as previously stated, R5RS doesn't have a sleep function. If there was more time, the project could have been moved to a different version of Scheme that had a sleep function built in. Along with this, the display function could be improved. Another way to add actions to wires would be necessary to fix the problem that the display function currently has.

Appendix

Designed Circuit Outputs

D-Latch Setup

```
(define d (make-wire))
(define clk (make-wire))
(define q (make-wire))
(define nq (make-wire))

(set-signal! d 1)
(after-delay 500 (lambda() (set-signal! d 0)))
(after-delay 700 (lambda() (set-signal! d 1)))
(clock clk 200)
(d-latch d clk q nq)

(display-group "input" d "d" clk "clock" q "q" nq "nq")
(propagate)
```

D-Latch Output

```
input    d    clock    q    nq    time
         1    0        0    1     0
input    d    clock    q    nq    time
         1    0        0    1     0
input    d    clock    q    nq    time
         1    0        0    1     0
input    d    clock    q    nq    time
         1    0        0    1     0""

input    d    clock    q    nq    time
         1    1        0    1     200
input    d    clock    q    nq    time
         1    1        0    0     207
input    d    clock    q    nq    time
         1    1        1    0     211
input    d    clock    q    nq    time
         1    0        1    0     400
input    d    clock    q    nq    time
         0    0        1    0     500
input    d    clock    q    nq    time
         0    1        1    0     600
input    d    clock    q    nq    time
         0    1        0    0     607
input    d    clock    q    nq    time
         0    1        0    1     611
input    d    clock    q    nq    time
         1    1        0    1     700
input    d    clock    q    nq    time
         1    1        0    0     707
input    d    clock    q    nq    time
         1    1        1    0     711
input    d    clock    q    nq    time
         1    0        1    0     800
input    d    clock    q    nq    time
         1    1        1    0     1000done
```

D-Flip Flop Setup

```
(define d (make-wire))
(define clk (make-wire))
(define q (make-wire))
(define nq (make-wire))

(set-signal! d 1)
(after-delay 500 (lambda() (set-signal! d 0)))
(after-delay 700 (lambda() (set-signal! d 1)))
(clock clk 200)
(d-flipflop d clk q nq)

(display-group "input" d "d" clk "clock" q "q" nq "nq")
(propagate)
```

D-Flip Flop Output

```
input    d    clock    q    nq    time
         1    0        0    1     0
input    d    clock    q    nq    time
         1    0        0    1     0
input    d    clock    q    nq    time
         1    0        0    1     0
input    d    clock    q    nq    time
         1    0        0    1     0""

input    d    clock    q    nq    time
         1    1        0    1     200
input    d    clock    q    nq    time
         1    1        0    0     207
input    d    clock    q    nq    time
         1    1        1    0     211
input    d    clock    q    nq    time
         0    1        1    0     300
input    d    clock    q    nq    time
         0    0        1    0     400
input    d    clock    q    nq    time
         0    1        1    0     600
input    d    clock    q    nq    time
         0    1        0    0     607
input    d    clock    q    nq    time
         0    1        0    1     611
input    d    clock    q    nq    time
         1    1        0    1     700
input    d    clock    q    nq    time
         1    0        0    1     800
input    d    clock    q    nq    time
         1    1        0    1     1000done
```

Register Setup

```
(define d (make-wire))
(define clk (make-wire))

(set-signal! d 1)

(after-delay 1100 (lambda() (set-signal! d 0)))
(clock clk 200)
(define qlist (make-qlist))
(4bit-register d clk qlist)


(display-group "input" d "d")
(display-qlist qlist "register")
(propagate)
```

Register Output

```
input    d    time
         1    0""

register    bit0    bit1    bit2    bit3    time
            0       0       0       0       0
register    bit0    bit1    bit2    bit3    time
            0       0       0       0       0
register    bit0    bit1    bit2    bit3    time
            0       0       0       0       0
register    bit0    bit1    bit2    bit3    time
            0       0       0       0       0""

register    bit0    bit1    bit2    bit3    time
            1       0       0       0       211
register    bit0    bit1    bit2    bit3    time
            1       1       0       0       611
register    bit0    bit1    bit2    bit3    time
            1       1       1       0       1011
input    d    time
         0    1100
register    bit0    bit1    bit2    bit3    time
            0       1       1       0       1407
register    bit0    bit1    bit2    bit3    time
            0       1       1       1       1411
register    bit0    bit1    bit2    bit3    time
            0       0       1       1       1807
register    bit0    bit1    bit2    bit3    time
            0       0       0       1       2207
register    bit0    bit1    bit2    bit3    time
            0       0       0       0       2607done
```

Source Code

```
;;;;Wire

;Create a wire that will be used as connections for the gates and circuits building
(define (make-wire)
  ;Initialize variables within the wire
  ;Signal-value represents the value of the wire
  ;Action-procedure is the list of procedures that the wire must do when its signal gets changed
  (let ((signal-value 0)
        (action-procedure '()))

    ;Alters the value of the wire signal
    (define (set-my-signal! new-value)
      ;Only alter the value of the signal if the value changed otherwise left alone
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 ;Calls all the functions that get called onto the wire to update the rest of the circuit
                 (call-each action-procedure))
          'done))

    ;Add a procedure and then call it
    (define (accept-action-procedure proc)
      (set! action-procedure (cons proc action-procedure))
      (proc))

    ;Interprets input and translates it to one of the functions
    (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))))

    ;Call the dispatch function
    dispatch))
;Function to call each procedure for the given wire
(define (call-each procedure)
  (if (null? procedure)
      'done
      (begin
        ((car procedure))
        (call-each (cdr procedure)))))

;Make the function calls for the wire a little easier to call
;Makes it seem more like a function rather than its inputs
(define (get-signal wire)
  (wire 'get-signal))
(define (wait wire)
  (wire 'wait))
(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))
(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))


;;;;Segments

;Create segment data type for agenda datatype
(define (make-time-segment time queue)
  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))


;;;;Queue

;Create queue datatype (provided by the textbook)
;Issue lies in front-ptr as it returns a list containing the front-ptr
```

```
;Create a queue data structure
(define (make-queue) (cons '() '()))

;Get access to the the first and last item of the queue
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))

;Functions to change values in the queue
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
(define (empty-queue? queue) (null? (front-ptr queue)))

;;Queue Operations

;Insert new item into our queue
(define(insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           ;Queue is empty so we need to set new-pair to the first and last item in the queue
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           ;Queue is not empty so we add the item to the back of the queue
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))

;Removes the item at the front of the queue and returns the modified queue
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))


;;;;The Agenda

;Create agenda data structure
(define (make-agenda) (list 0))

;Selectors for the agenda
(define (current-time agenda) (car agenda))
(define (segments agenda) (cdr agenda))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))

;Alter values for the agenda
(define (set-current-time! agenda time) (set-car! agenda time))
(define (set-segments! agenda segments) (set-cdr! agenda segments))

;Check if the agenda is empty
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

```
;Function to add things to the agenda
(define (add-to-agenda! time action agenda)

  ;Checks if the location is correct
  (define (belongs-before? segments)
    (or (null? segments) (< time (segment-time (car segments)))))

  ;Makes a new segement
  (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 (car segments)) time)
          ;Time exists so we simply need to add to the queue
          (insert-queue! (segment-queue (car segments))
                         action)
          ;Need if the time is greater than the next time so that we know we need to create a new segment.
          (let ((rest (cdr segments)))
            (if (belongs-before? rest)
                ;In this condition it means that there is no time in the-agenda that matches time
                (set-cdr! segments
                          (cons (make-new-time-segment time action) rest))
                ;Recursive call
                (add-to-segments! rest)))))
    ;Call that is actually done
    (let ((segment (segments agenda)))
      (if (belongs-before? segment)
          (set-segments! agenda (cons (make-new-time-segment time action) segment))
          (add-to-segments! segment))))

;Removes the first segment from the agenda
;Remove the first item in the queue of the first segement, otherwise if it is empty, remove the segment
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    ;Remove first item in the queue
    (delete-queue! q)
    ;If queue is empty then the empty queue is returned, otherwise the queue is set as the
    ;rest of the queue excluding the first item
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))

;Get first item from the agenda and progress the timer forward
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty -- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        ;Change the time to the time of the first segment
        (set-current-time! agenda (segment-time first-seg))
        ;Return the first item that needs to be done in the agenda
        (car (front-ptr (segment-queue first-seg))))))

;Shows whether is more than one item to be done in the agenda
(define (second-agenda-item? agenda)
  (let ((segment (segments agenda)))
    (not (and (= (length segment)1)
              (= (length (front-ptr (segment-queue (first-segment agenda)))) 1)))))


;;Propagate that with the sleep included if sleep working in R5RS
;;(define (propagate)
  ;;(cond ((or (empty-agenda? the-agenda) (= (current-time the-agenda) 50)) 'done)
  ;;      ((= (current-time the-agenda) (segment-time (first-segment the-agenda)))
   ;;
    ;;     (let ((first-item (first-agenda-item the-agenda)))
     ;;      ((car first-item))
      ;;     (remove-first-agenda-item! the-agenda)
       ;;   (propagate)))))
        ;;(else (sleep 1))))
```

```
;Function that will take care of calling everything that is on the agenda
(define (propagate)
  ;Once the agenda is empty we are done
  (if (or (empty-agenda? the-agenda) (> (current-time the-agenda) 100000))
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        ;Call the first function that is in the agenda then remove it
        (first-item)
        (remove-first-agenda-item! the-agenda)
        ;Recurse until the-agenda is empty
        (propagate))))


;Function that takes care of delay
;Adds to the-agenda by using the add-agenda function and makes the time relative to the
;current time in the-agenda
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda))


;;;;Gates

;Not-gate (inverter)
;Input and output must be wires
;The value of the output wire is the inverted value of the input wire
;(not-gate input ouput): output = (not input)
(define (not-gate input output)

  ;Computes the logical-not of the input signal after the propagation delay, the output signal
  ;is set to the computated value
  (define (not-action-procedure)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay not-gate-delay
                   (lambda () (set-signal! output new-value)
                     (get-signal input)
                     (get-signal output)))))

  ;To connect an input with the ouput through the not-gate, add-action! is used to associate the
  ;input wire with the procedure that will be run whenever the input wire signal changes value
  (add-action! input not-action-procedure)
  'ok)

;Define the logical-not that will input a wire value s and output value is set to (NOT s)
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))


;Or-gate
;Inputs and output must be wires
;The value of the output wire is 0 if both a1 and a2 have the value of 0, otherwise
;the output wire has the value of 1.
;(or-gate a1 a2 ouput): output = (or a1 a2)
(define (or-gate a1 a2 output)

  ;Computes the logical-or of the two input signals and
  ;assigns the computed value to the output wire signal after the propagation delay occurs
  (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)))))

  ;Adds the action procedure to both input wires because if either of the wire signals is changed
  ;the output wire value needs to be re-evaluated
  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure)
  'ok)
```

```
;Create the logical-or that will return value of (OR a1 a2) in terms of 0s and 1s
(define (logical-or a1 a2)
  (cond ((and (= a1 1) (= a2 1)) 1)
        ((and (= a1 1) (= a2 0)) 1)
        ((and (= a1 0) (= a2 1)) 1)
        ((and (= a1 0) (= a2 0)) 0)
        (else (error "Invalid signals" a1 a2))))


;And-gate
;Inputs and outputs must be wires
;The value of the output wire is 1 if both a1 and a2 have the value of 1, otherwise
;the output wire has the value of 0.
;(and-gate a1 a2 ouput): output = (and a1 a2)
(define (and-gate a1 a2 output)

  ;Computes the logical-and of the input wire signals and the output wire signal is
  ;is set the computed value after taking into account the propagation delay for the and-gate
  (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)))))

  ;Adds the action procedure of and to both of the input wires
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok)

;The logical-and that will return value of (AND a1 a2) in terms of 0s and 1s
(define (logical-and a1 a2)
  (cond ((and (= a1 1) (= a2 1)) 1)
        ((and (= a1 1) (= a2 0)) 0)
        ((and (= a1 0) (= a2 1)) 0)
        ((and (= a1 0) (= a2 0)) 0)
        (else (error "Invalid signals" a1 a2))))

;Nand-gate
;Inputs and outputs must be wires
;The value of the output wire is 0 if both a1 and a2 have the value of 1, otherwise
;the output wire has the value of 1.
;(nand-gate a1 a2 ouput): output = (not (and a1 a2))
(define (nand-gate a1 a2 output)

  ;Adds the nand-gate function to the agenda, which assigns the computation value of
  ;of logical-nand for the input wires to the output wire.
  (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)))))

  ;The action procedure of nand gate is added to the input wires.
  (add-action! a1 nand-action-procedure)
  (add-action! a2 nand-action-procedure)
  'ok)

;The logical-nand that will return value of (NOT (AND a1 a2)) in terms of 0s and 1s
(define (logical-nand a1 a2)
  (cond ((and (= a1 1) (= a2 1)) 0)
        ((and (= a1 1) (= a2 0)) 1)
        ((and (= a1 0) (= a2 1)) 1)
        ((and (= a1 0) (= a2 0)) 1)
        (else (error "Invalid signals" a1 a2))))
```

```
;Nor-gate
;Inputs and outputs must be wires
;The value of the output wire is 1 if both a1 and a2 have the value of 0, otherwise
;the output wire has the value of 0.
;(nor-gate a1 a2 ouput): output = (not (or a1 a2))
(define (nor-gate a1 a2 output)

  ;Adds the nor-gate function to the agenda, which connects the input wires to the output wire
  ;through a nor-gate.
  (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)))))

  ;The action procedure is added to both of the input wires.
  (add-action! a1 nor-action-procedure)
  (add-action! a2 nor-action-procedure)
  'ok)

;The logical-nand that will return value of NOT(a1 OR a2) in terms of 0s and 1s
(define (logical-nor a1 a2)
  (cond ((and (= a1 1) (= a2 1)) 0)
        ((and (= a1 1) (= a2 0)) 0)
        ((and (= a1 0) (= a2 1)) 0)
        ((and (= a1 0) (= a2 0)) 1)
        (else (error "Invalid signals" a1 a2))))


;;;;Circuits


;Half-Adder
;Adds the signal values of the two input wires, a and b, and the sum is set as the signal values of
;the output wires, s and c. s is the one digit sum value. c represents overflow, the value for the
;second digit if it was multi-digit addition.
(define (half-adder a b s c)
  ;Create interior wires
  (let ((d (make-wire)) (e (make-wire)))
    ;Half-adder functionality
    (or-gate a b d)
    (and-gate a b c)
    (not-gate c e)
    (and-gate d e s)
    'ok))

;Full-Adder
;Computes the sum of three input wires, a, b, and c-in, and outputs the sum value to the output
;wires, sum and c-out. Functionality is similar to half-adder except this can compute 3 bit addition.
(define (full-adder a b c-in sum c-out)
  ;Create interior wires
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    ;Full-adder functionality
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

```
;D-Latch
;Accepts a wire and clock object as inputs and gives two output wires.
;If clk = 1, q = d. If clk = 0, q remains the same. qnot is always the inverted value of q.
(define (d-latch d clk q nq)
  ;Create interior wires
  (let ((not-d (make-wire)) (wire-1 (make-wire)) (wire-2 (make-wire)))
    ;D-Latch functionality
    (set-signal! nq 1)
    (not-gate d not-d)
    (and-gate d clk wire-1)
    (and-gate not-d clk wire-2)
    (nor-gate wire-1 q nq)
    (nor-gate wire-2 nq q)
    'ok))

;D Flip-Flop
;Accepts a wire and clock object as inputs and gives two output wires.
;If clk goes from 0 to 1, q = d. Otherwise q remains the same. qnot is always the inverted value of q.
(define (d-flipflop d clk q nq)
  ;Create interior wires
  (let ((midq (make-wire)) (midNotq (make-wire)) (notClk (make-wire)))
    ;D Flip-Flop functionality
    (not-gate clk notClk)
    (d-latch d notCLK midq midNotq)
    (d-latch midq clk q nq)
    'ok))


;qlist constructor
;Creates a list of four wires
(define (make-qlist)
  (let ((a (make-wire))
        (b (make-wire))
        (c (make-wire))
        (d (make-wire)))
    (list a b c d)))

;qlist selectors
;Gets the first wire in qlist
(define (bit0 qlist)
  (car qlist))

;Gets the second wire in qlist
(define (bit1 qlist)
  (cadr qlist))

;Gets the third wire in qlist
(define (bit2 qlist)
  (caddr qlist))

;Gets the fourth wire in qlist
(define (bit3 qlist)
  (cadddr qlist))

;Function that alters the values in qlist
(define (set-qlist qlist a b c d)
  (set-signal! (bit0 qlist) a)
  (set-signal! (bit1 qlist) b)
  (set-signal! (bit2 qlist) c)
  (set-signal! (bit3 qlist) d))

;A display group function to provide a display of all the wires in qlist
(define (display-qlist qlist name)
  (display-group name (bit0 qlist) "bit0" (bit1 qlist) "bit1" (bit2 qlist) "bit2" (bit3 qlist) "bit3"))

;4bit-Register
;Accepts a wire and clock object as inputs and gives a list of four output wires.
;If clk goes from 0 to 1, (bit0 qlist) = d, (bit1 qlist) = (bit0 qlist), (bit2 qlist) = (bit1 qlist),
;and (bit3 qlist) = (bit4 qlist). The values of the output wires in qlist shifts over to the right.
;Otherwise values in qlist remain the same.
```

```
(define (4bit-register d clk qlist)
  ;Create interior wires
  (let ((nq0 (make-wire)) (nq1 (make-wire))(nq2 (make-wire))(nq3 (make-wire)) )

    ;4bit-Register functionality
    (d-flipflop d clk (bit0 qlist) nq0)
    (d-flipflop (bit0 qlist) clk (bit1 qlist) nq1)
    (d-flipflop (bit1 qlist) clk (bit2 qlist) nq2)
    (d-flipflop (bit2 qlist) clk (bit3 qlist) nq3)
    'ok))


;Clock
;S function that alters from 1 to 0 after a delay of delay
(define (clock wire delay)
  ;Get new value for clock
   (let ((new-value (logical-not (get-signal wire))))
     ;Change the value after the delay
     (after-delay delay (lambda ()
                          (set-signal! wire new-value)
                          (clock wire delay)))))


;;;;Displays


;Adding this to a wire will make it so it is called each time that a function value
;is changed the value wil be displayed
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (newline)
                 (display name)
                 (display " ")
                 (display (current-time the-agenda))
                 (display " New-value = ")
                 (display (get-signal wire)))))


;Input must a name and then wires and names to correspond with the wires
;All names must be strings and it must alternate wire and name
;Input : group wire name...
(define (display-group name . args)
  ;Function that represents the display that is created
  (define (create-display)
    ;Creates the display for the first line of wires
    (define (first-line str lst)
      ;Use iteration to create the output string
      ;Need to update lst to (cddr lst) because you have to pass a wire and a string
      (if (null? lst)
          str
          (first-line (string-append str (cadr lst) (create-space 4)) (cddr lst))))
    ;Iterative function that creates the second line's display
    (define (second-line str lst)
      (if (null? lst)
          str
          (second-line (string-append str (number->string (get-signal (car lst)))
                                      (create-space (- (string-length (cadr lst)) 1)) (create-space 4))
                       (cddr lst))))
```

```
    ;Function to create val number of spaces
    (define (create-space val)
      (if (= val 0)
          ""
          (string-append " " (create-space (- val 1)))))
    ;Display everything
    (display "\n")
    (display (string-append name (create-space 4)))
    (display (first-line "" args))
    (display "time")
    (display "\n")
    (display (create-space (+ 4 (string-length name))))
    (display (second-line "" args))
    (display (current-time the-agenda)))

  ;Add the create-display to the wires
  ;Since we have a number arguements we have to iterate through them
  (define (addActions lst)
    (if (null? lst)
        ""
        (begin
          (add-action! (car lst) (lambda() (create-display)))
          (addActions (cddr lst)))))
  ;Call the function to add the procedure to all the wires
  (addActions args))


;Global variable that all circuits refer to for the propagate funciton
(define the-agenda (make-agenda))

;Defining the time delays for the gate functions
(define not-gate-delay 1)
(define and-gate-delay 3)
(define or-gate-delay 5)
(define nand-gate-delay 4)
(define nor-gate-delay 4)


;Testing of D-Latch
;Defining the inputs and outputs
(define d (make-wire))
(define clk (make-wire))
(define q (make-wire))
(define nq (make-wire))

(set-signal! d 1)
(clock clk 200)

(after-delay 500 (lambda() (set-signal! d 0)))
(after-delay 700 (lambda() (set-signal! d 1)))

;Using the circuit
(d-latch d clk q nq)

;Displaying the results
(display-group "Circuit" d "d" clk "clock" q "q" nq "nq")
(propagate)
```

```
;Testing of D-FlipFlop
;Defining inputs and outputs
(define d (make-wire))
(define clk (make-wire))
(define q (make-wire))
(define nq (make-wire))

(set-signal! d 1)
(clock clk 200)

(after-delay 500 (lambda() (set-signal! d 0)))
(after-delay 700 (lambda() (set-signal! d 1)))

;Using the circuit
(d-flipflop d clk q nq)

;Displaying the result
(display-group "Circuit" d "d" clk "clock" q "q" nq "nq")
(propagate)


;Testing of 4bit-Register

;Defining inputs and outputs
(define d (make-wire))
(define clk (make-wire))
(set-signal! d 1)

(after-delay 1100 (lambda() (set-signal! d 0)))
(clock clk 200)
(define qlist (make-qlist))

;Using the circuit
(4bit-register d clk qlist)

;Displaying the circuit output
(display-group "input" d "d")
(display-qlist qlist "register")
(propagate)
```

References

Abelson, Harold, and Gerald Jay Sussman. "Modularity, Objects, and State." *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass.: MIT ;, 1996. Print.