Computation II: embedded system design (5EIB0)

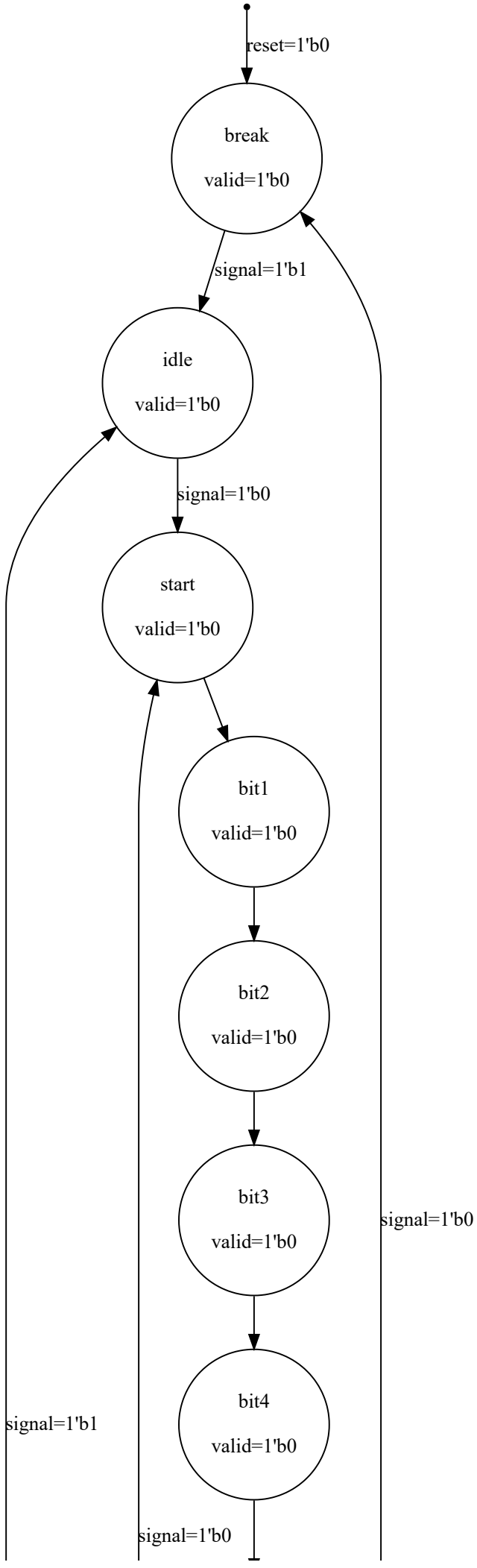
Description Submission view

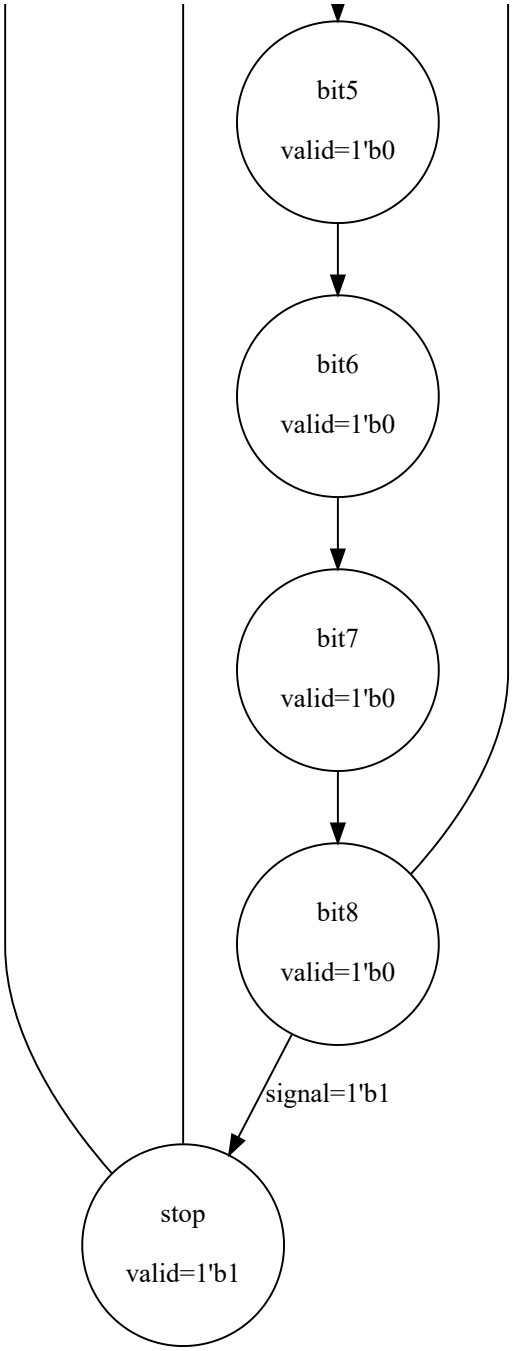
Available from: Wednesday, 3 July 2024, 6:00 PM
Due date: Wednesday, 3 July 2024, 9:00 PM
Requested files: uart.v (Download)
Type of work: Individual work

Assignment

Universal Asynchronous Receiver/Transmitter (UART) is a simple serial protocol that is commonly used to communicate between two devices that do not share the same synchronous clock. Data is communicated as frames from one device to the other device along a single wire. The line is idle when it is high. The start of a frame is signalled by pulling the line low for a single data bit duration. This is followed by 8 data bits (this number can vary between implementations, but we will consider 8 here) that can be high or low, representing a byte of data in binary. This must be immediately followed by the stop bit where the line is held high for at least a single data bit duration. A break signal can be communicated by pulling the line low, such that the stop bit is not signalled.

In this assignment you will debug a Verilog implementation of a Finite State Machine (FSM) to detect correctly formed UART frames of data with the following specifications. Including the clock (clk) and reset signals, the implemented FSM will have 3 input signals and 1 output signal.


reset=1'b0
break
valid=1'b0
signal=1'b1
idle
valid=1'b0
signal=1'b0
start
valid=1'b0
bit1
valid=1'b0
bit2
valid=1'b0
bit3
valid=1'b0
bit4
valid=1'b0
signal=1'b0
signal=1'b1
signal=1'b0



The clk input signal is 1-bit wide, the reset input signal is 1-bit wide and the signal input signal is 1-bit wide. The state elements of the FSM are to capture their inputs on the positive clock (clk) edge and reset their state when the synchronous reset signal is high. A reset can be asserted (synchronously) at any time causing all state elements to reset and hence a transtion to the initial state.

The valid output signal is 1-bit wide. Initial values of the output signals upon reset are shown in the FSM diagram.

Your solution will be tested using bounded model checking against a golden reference implementation. The model checking software will stop at the earliest moment that your implementation diverges from the behaviour of the golden reference implementation. If a divergence takes place, a waveform will be created showing a trace of the module signals resulting in the divergent state. This waveform also shows the behaviour of the golden reference (REF) so that you can see what your implementation (DUT) should have done differently.

Please note that the golden reference implementation supersedes all other implementation descriptions. If the output from your solution implementation does not diverge from the golden reference implementation then your solution is marked as correct. Otherwise, your solution is incorrect and you should use the waveform that is produced to assist you to correct your implementation.

Requested files

uart.v

```
module uart (
    input clk,
    input reset,
    input signal,
    output valid
);

// define unique constants to identify each state
localparam BREAK = 4'd0;
localparam IDLE = 4'd1;
localparam START = 4'd2;
localparam BIT1 = 4'd3;
localparam B1T2 = 4'd4;
localparam BIT3 = 4'd5;
localparam BIT4 = 4'd6;
localparam BIT5 = 4'd7;
localparam BIT6 = 4'd8;
localparam BIT7 = 4'd9;
localparam BIT8 = 4'd10;

// declare verilog variables of type reg for use in always blocks
// <name>_r is used to infer a register in the clocked always block
// <name>_nxt is used assign the next state of the register in the combinational always block
reg [3 : 0] state_r, state_nxt;
reg valid_r, valid_nxt;

// clock synchronous always block will only be evaluated on the positive edge of the clock
// only use non-blocking assignments (<=) in this block
always @(posedge reset) begin
    // check for the reset signal on the clk edge, infering a synchronous reset
    // if the module is not being reset, assign all of the derived combinational <name>_nxt values to <name>_r
    // if the module is being reset, assign initial constant values to the <name>_r variables
    // this will infer a register for <name>_r if it is assigned defined values at all times
    // otherwise an unintentional latch will be inferred
    if (clk == 1'b0) begin
        // module is not being reset
        // assign all of the derived combinational <name>_nxt values to the respective <name>_r
        state_r <= state_nxt;
        valid_r <= valid_nxt;
    end else begin
        // module is being reset
        // assign constant values to each <name>_r variable
        state_r <= IDLE;
        valid_r <= 1'b0;
    end
end

// combinational always block will evaluate whenever any signal in its sensitivity list changes
// here we use the wildcard * sensitivity list, which means that the list will be inferred from the assignments in the block
// only use blocking assignments (=) in this block
always @(*) begin
    // make sure that <name>_nxt signals are always defined to avoid latches
    // to ensure register elements minimally retain their last value, assign each <name>_nxt its respective <name>_r value
    state_nxt = state_r;
    valid_nxt = valid_r;

    // case statement is used to perform different logical derivations depending on current state
    // state_r stores the current state of this FSM
    // the unique state identifiers that were defined near the top of this file are used to identify the current state
    case (state_nxt)
        // state_r will remain constant between positive clock edges
        // only one unique state can match at any time
        BREAK: // evaluate logic for state BREAK
        begin
            valid_nxt = 1'b0;
            if (signal == 1'b1)
            begin
                state_nxt = IDLE;
            end
        end

        IDLE: // evaluate logic for state IDLE
        begin
            valid_nxt = 1'b0;
            if (signal == 1'b0)
            begin
                state_nxt = START;
            end
        end

        START: // evaluate logic for state START
        begin
            valid_nxt = 1'b0;
            state_nxt = BIT1;
        end

        BIT1: // evaluate logic for state BIT1
        begin
            valid_nxt = 1'b0;
            state_nxt = BIT3;
        end

        BIT2: // evaluate logic for state BIT2
        begin
            valid_nxt = 1'b0;
            state_nxt = BIT4;
        end

        BIT4: // evaluate logic for state BIT4
        begin
            valid_nxt = 1'b0;
            state_nxt = BIT8;
        end
```

```
104
105          BIT5: // evaluate logic for state BIT5
106          begin
107              valid_nxt = 1'b0;
108              state_nxt = BIT5;
109          end
110
111          BIT6: // evaluate logic for state BIT6
112          begin
113              valid_nxt = 1'b0;
114              state_nxt = BIT7;
115          end
116
117          BIT7: // evaluate logic for state BIT7
118          begin
119          begin
120              valid_nxt = 1'b0;
121              if (signal == 1'b0)
122              begin
123                  state_nxt = IDLE;
124              end
125              else if (signal == 1'b1)
126              begin
127                  state_nxt = BIT8;
128              end
129          end
130          end
131
132
133
134          STOP: // evaluate logic for state STOP
135          begin
136              valid_nxt = 1'b0;
137              if (signal == 1'b1)
138              begin
139                  state_nxt = START;
140              end
141              else if (signal == 1'b0)
142              begin
143                  state_nxt = IDLE;
144              end
145          end
146
147          default: // should not be reachable if the state register is initialised and updated correctly
148          begin
149              valid_nxt = 1'b0;
150              state_nxt = BREAK;
151          end
152      endcase
153  end
154
155  // assign values to output ports
156  assign valid = state_nxt;
157
158  endmodule
159
```

VPL