Softcore System on a Chip Class Report 3: Blinking-LED Core

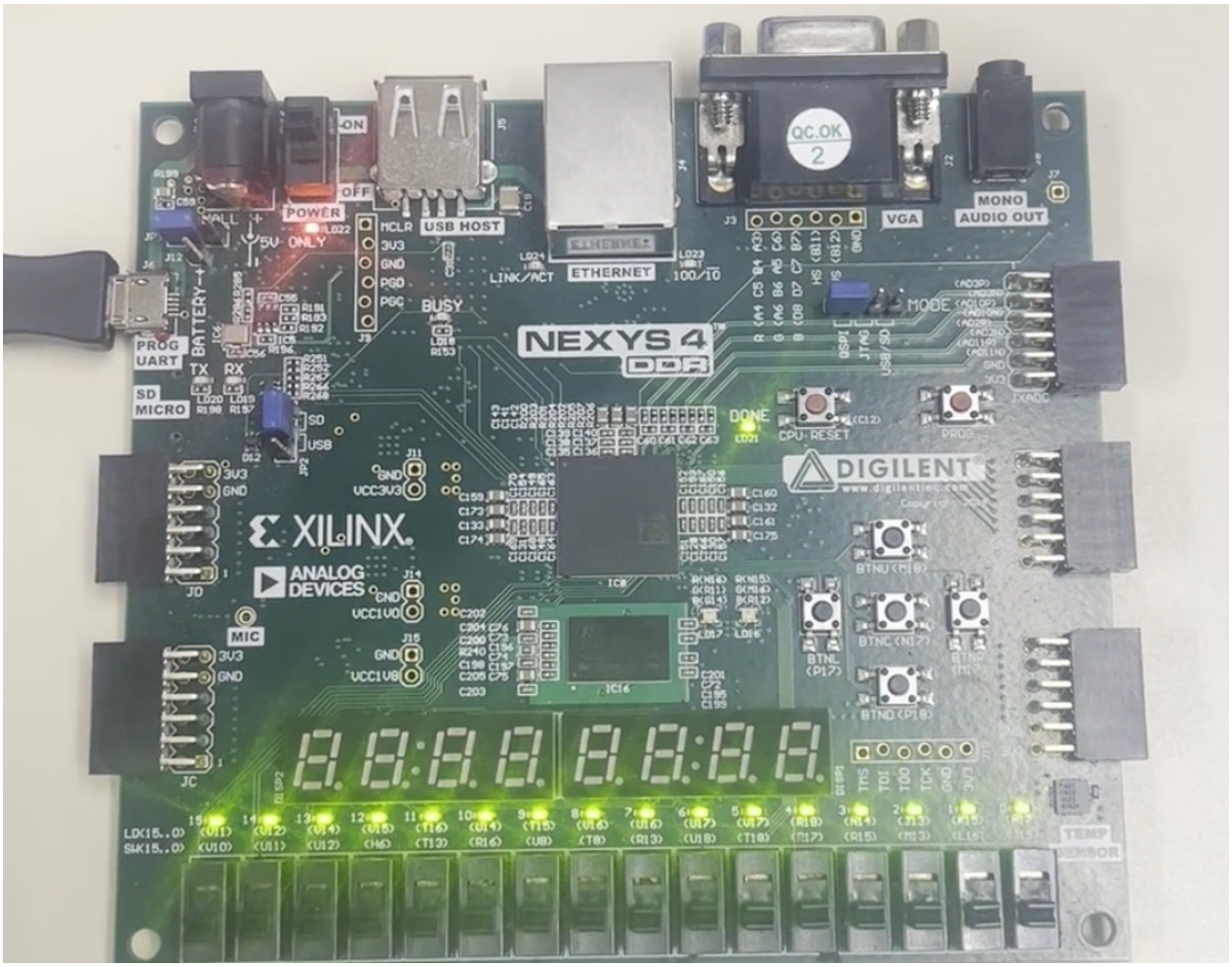
David Edeni

December 13th, 2025

Lab Description:

This project is a Verilog implementation of a blinking-LED core that can turn LEDs on and off at specific rates. The core has a four-bit output signal connected to four discrete LEDs. It has four 16-bit registers that specify the values of the individual blinking intervals in milliseconds. With the blinking-LED core, the processor only needs to write the registers.

Based on a specified hardware configuration, this is what the Nexys DDR-4 FPGA Board would look like in when LEDs are flashing.



To produce this flashing pattern, Pong Chu's vanilla System Verilog Interface was used. This interface included a microblazeI cpu and utilized dual-HDL programming functionality (Vivado and Vitis).

The mcs top vanilla System Verilog module is a system wrapper for a MicroBlaze MCS-based design. It connects the CPU to memory-mapped peripherals, including switches, LEDs, and UART. This module creates a complete embedded system where software running on MicroBlaze controls hardware peripherals via memory-mapped IO.

Figure 1: The mcs top vanilla (system wrapper) module:

```
module mcs_top_vanilla
#(parameter BRG_BASE = 32'hc000_0000)
(
   input logic clk,
   input logic reset_n,
   // switches and LEDs
   input logic [15:0] sw,
   output logic [15:0] led,
   // uart
   input logic rx,
   output logic tx
);

   // declaration
   logic clk_100M;
   logic reset_sys;
   // MCS IO bus
   logic io_addr_strobe;
   logic io_read_strobe;
   logic io_write_strobe;
   logic [3:0] io_byte_enable;
   logic [31:0] io_address;
   logic [31:0] io_write_data;
   logic [31:0] io_read_data;
   logic io_ready;
   // fpro bus
   logic fp_mmio_cs;
   logic fp_wr;
   logic fp_rd;
   logic [20:0] fp_addr;
   logic [31:0] fp_wr_data;
   logic [31:0] fp_rd_data;

   // body
   assign clk_100M = clk;                  // 100 MHz external clock
   assign reset_sys = !reset_n;

   //instantiate uBlaze MCS
   cpu cpu_unit (
    .Clk(clk_100M),                      // input wire Clk
    .Reset(reset_sys),                   // input wire Reset
    .IO_addr_strobe(io_addr_strobe),     // output wire IO_addr_strobe
    .IO_address(io_address),             // output wire [31 : 0] IO_address
    .IO_byte_enable(io_byte_enable),     // output wire [3 : 0] IO_byte_enable
    .IO_read_data(io_read_data),         // input wire [31 : 0] IO_read_data
    .IO_read_strobe(io_read_strobe),     // output wire IO_read_strobe
    .IO_ready(io_ready),                 // input wire IO_ready
    .IO_write_data(io_write_data),       // output wire [31 : 0] IO_write_data
    .IO_write_strobe(io_write_strobe)    // output wire IO_write_strobe
   );

   // instantiate bridge
   chu_mcs_bridge #(.BRG_BASE(BRG_BASE)) bridge_unit (.*, .fp_video_cs());

   // instantiated i/o subsystem
   mmio_sys_vanilla #(.N_SW(16),.N_LED(16)) mmio_unit (
   .clk(clk),
   .reset(reset_sys),
   .mmio_cs(fp_mmio_cs),
   .mmio_wr(fp_wr),
   .mmio_rd(fp_rd),
   .mmio_addr(fp_addr),
   .mmio_wr_data(fp_wr_data),
   .mmio_rd_data(fp_rd_data),
   .sw(sw),
   .led(led),
   .rx(rx),
   .tx(tx)
  );
endmodule
```

The blink led System Verilog module implements a memory-mapped LED blink controller. Each LED has a programmable blink rate stored in registers that software can read and write. The software controls LED blinking behavior dynamically by writing millisecond values into registers.

Figure 2: The blink led (blinking-LED) module:

```
`timescale 1ns / 1ps // Defines this module's simulation time unit (1 ns) and time precision unit (1 ps)

module blink_led#(parameter N = 4)( // Defines the top-level module that controls multiple LED blink rates via memory-mapped registers on a Nexys4 DDR™ FPGA Board
    input logic clk, // Input clock signal (usually 100 MHz for FPGAs). This signal provides timing and synchronization for digital circuits.
    input logic reset, // Active-high reset input signal. This signal resets a digital circuit to a known starting state. reset = 1 → reset is on (the circuit resets). reset = 0 → reset is off (the circuit runs normally).
    input logic chip_select, // Active-high chip select input control signal that enables this peripheral to respond to read and write transactions on the system memory-mapped bus, all other bus activity is ignored when deasserted.
    input logic read_enable, // Read enable input control signal for register access. This signal indicates whether a valid read transaction has occured, when this signal is asserted, the addressed register's contents are driven onto the read data bus, when deasse
    input logic write_enable, // Write enable input control signal for register access. This signal indicates whether a valid write transaction has occured, when this signal is asserted, the data on the write data bus is written into the addressed register, when t
    input logic [4:0] address_bus, // 5-bit address bus input control signal used to select internal memory-mapped registers within the module. Each unique address corresponds to a specific register or control function, allowing the CPU or external controller to s
    input logic [31:0] write_data, // 32-bit write data bus for writing values to the LED blink rate registers
    output logic [31:0] read_data, // 32-bit read data bus for reading values from the LED blink rate registers
    output logic [N-1:0] LED // 4-bit output control vector that drives the physical LEDs on a Nexys4 DDR™ FPGA Board
    );

    logic [15:0] led_blink_rate_register [3:0]; // Declare four (3:0) 16-bit registers storing LED blink rates
    logic internal_write_enable; // Internal write-enable control signal generated from chip select and write enable input control signal

    always_ff @(posedge clk, posedge reset) begin // Sequential logic block that runs on a rising edge of a clock (when clock goes from 0 → 1) or when reset goes high (when reset goes from 0 → 1)
      if(reset) begin  // If reset is active (reset = 1)
         led_blink_rate_register[0] <= 16'd0; // Clear the LED blink rate register for LED 0
         led_blink_rate_register[1] <= 16'd0; // Clear the LED blink rate register for LED 1
         led_blink_rate_register[2] <= 16'd0; // Clear the LED blink rate register for LED 2
         led_blink_rate_register[3] <= 16'd0; // Clear the LED blink rate register for LED 3
      end
      else if(internal_write_enable) begin // If a valid write transaction is requested
         case(address_bus) // Decode the address bus to select which register to write to
            5'd0: led_blink_rate_register[0] <= write_data; // Write the blink rate for LED 0
            5'd1: led_blink_rate_register[1] <= write_data; // Write the blink rate for LED 1
            5'd2: led_blink_rate_register[2] <= write_data; // Write the blink rate for LED 2
            5'd3: led_blink_rate_register[3] <= write_data; // Write the blink rate for LED 3
         endcase
      end
    end

    assign internal_write_enable = chip_select && write_enable; // Assert the internal write enable signal only when chip/peripheral is selected and write enable is active

    always_comb begin // Combinational logic block for read transactions
      case(address_bus[2:1]) // Use address bus bits 2 and 1 to select which LED blink rate register to read from
         2'd0: read_data <= {16'd0, led_blink_rate_register[0]}; // Read the LED blink rate for LED 0
         2'd1: read_data <= {16'd0, led_blink_rate_register[1]}; // Read the LED blink rate for LED 1
         2'd2: read_data <= {16'd0, led_blink_rate_register[2]}; // Read the LED blink rate for LED 2
         2'd3: read_data <= {16'd0, led_blink_rate_register[3]}; // Read the LED blink rate for LED 3
         default: read_data = 32'd0; // If none of the specified register addresses match, return a zeroed 32-bit value to indicate an invalid or unused address. This prevents accidental propogation of undefined or stale data onto the read data bus.
      endcase
    end

    led led1( // Instantiates the first LED controller module for the Nexys4 DDR™ FPGA Board
       .clk(clk), // Connect the system clock signal to the LED controller
       .reset(reset), // Connect the reset signal to the LED controller
       .led_blink_rate_ms(led_blink_rate_register[0]), // Provide a LED blink rate for LED 0 from the LED blink rate register
       .LED(LED[0]) // Drive physical LED 0 (first LED in our configuration)
    );
     led led2( // Instantiates the second LED controller module for the Nexys4 DDR™ FPGA Board
       .clk(clk), // Connect the system clock to the LED controller
       .reset(reset), // Connect the reset signal to the LED controller
       .led_blink_rate_ms(led_blink_rate_register[1]), // Provide a LED blink rate for LED 1 from the LED blink rate register
       .LED(LED[1]) // Drive physical LED 1 (second LED in our configuration)
    );

    led led3( // Instantiates the third LED controller module for the Nexys4 DDR™ FPGA Board
       .clk(clk), // Connect the system clock signal to the LED controller
       .reset(reset), // Connect the reset signal to the LED controller
       .led_blink_rate_ms(led_blink_rate_register[2]), // Provide a LED blink rate for LED 2 from the LED blink rate register
       .LED(LED[2]) // Drive physical LED 2 (third LED in our configuration)
    );

    led led4( // Instantiates the fourth LED controller module for the Nexys4 DDR™ FPGA Board
       .clk(clk), // Connect the system clock signal to the LED controller
       .reset(reset), // Connect the reset signal to the LED controller
       .led_blink_rate_ms(led_blink_rate_register[3]), // Provide a LED blink rate for LED 3 from the LED blink rate register
       .LED(LED[3]) // Drive physical LED 3 (fourth LED in our configuration)
    );

endmodule // End of module definition
```

Finally, the led System Verilog module controls one physical LED, toggling it at a programmable rate. This module turns a human-friendly time value (milliseconds) into precise hardware timing using counters.

Figure 3: The led (basic LED) module:

```
`timescale 1ns / 1ps // Defines this module's simulation time unit (1 ns) and time precision unit (1 ps)

module led( // Defines an LED control module for driving a single LED on a Nexys4 DDR™ FPGA Board
    input logic clk, // Input clock signal (usually 100 MHz for FPGAs). This signal provides timing and synchronization for digital circuits.
    input logic reset, // Active-high reset input signal. This signal resets a digital circuit to a known starting state. reset = 1 → reset is on (the circuit resets). reset = 0 → reset is off (the circuit runs normally).
    input logic [15:0] led_blink_rate_ms, // 16-bit LED blink rate input control signal that specifies the LED blink rate period in milliseconds. This value defines how long the LED will stay in one state before toggling. A value of 0 disables blinking and
    output logic LED // Output control signal that drives a physical LED on a Nexys4 DDR™ FPGA Board
    );

    logic [31:0] count; // Declare a 32-bit register for a current counter value (count)
    logic [31:0] clock_cycle_rate; // Declare a 32-bit register that stores the number of clock cycles required for the desired blink interval (clock_cycle_rate)

    assign clock_cycle_rate = led_blink_rate_ms * (CLK100MHZ * 1000); // Converts the LED blink rate from milliseconds into clock cycles based on the 100 MHZ system clock.

    always_ff @(posedge clk, posedge reset) begin // Sequential logic block that runs on a rising edge of a clock (when clock goes from 0 → 1) or when reset goes high (when reset goes from 0 → 1)
       if(reset) begin  // If reset is active (reset = 1)
          count <= 0; // Reset counter to zero (non-blocking [<=] used here because this is sequential logic, not combinational logic)
          LED <= 0;   // Turn/drive the LED off to ensure a known startup state (non-blocking [<=] used here because this is sequential logic, not combinational logic)
       end
       else begin // If reset is not active (reset = 0)
          if(led_blink_rate_ms == 0) begin // If the LED blink rate is zero
             LED <= 0;   // Turn/drive the LED off to ensure a known startup state (non-blocking [<=] used here because this is sequential logic, not combinational logic)
             count <= 0; // Reset counter to zero (non-blocking [<=] used here because this is sequential logic, not combinational logic)
          end
          else if(count >=  clock_cycle_rate - 1) begin // If the counter has reached the total number of clock cycles required for the programmed blink interval. When this condition is true, the specified time delay has fully elapsed, meaning it is time to
             LED <= ~LED; // Toggle the LED state (non-blocking [<=] used here because this is sequential logic, not combinational logic)
             count <= 0;  // Reset counter to zero (non-blocking [<=] used here because this is sequential logic, not combinational logic)
          end
          else begin // If the blink interval has not yet elapsed, the counter continues counting clock cycles until it reaches the required value that corresponds to the selected blink period.
             count <= count + 1; // Increment counter value up by 1 on each clock cycle (non-blocking [<=] used here because this is sequential logic, not combinational logic)
          end
       end
    end

endmodule // End of module definition
```

Overall, the Blinking-LED Core Project taught me how to: design the blinking circuit for one LED and duplicate it four times, determine the register map and derive the wrapping circuit, derive the HDL code, derive the device driver, expand the vanilla MMIO subsystem to include a blinking-LED core slot 4, modify the vanilla FPro system to connect the led signal to the blinking-LED core, synthesize the new system, derive a testing program, and verify its operation.