Firmware Workgroup Session 2

Last time…

- Introduction to tools
  - ISE vs. Vivado (sticking to ISE here)
- A bit on VHDL:
  - Process blocks.
  - Libraries: I recommend “IEEE.NUMERIC_STD.ALL” for best cross-vendor compatibility.
  - Only use std_logic and std_logic_vector
- UCF files to associate top-level design with FPGA pins.
- Blinky lights examples.

Blinky Lights Implementations (1)

- Goal was to flash an LED at ~1 Hz.
- Two implementations looked very similar:

```vhdl
entity ToggleCounter is
   Generic (
      CYCLES_PER_HALF_PERIOD_G : integer := 50000000
   );
   Port (
      clk       : in  STD_LOGIC;
      toggleOut : out  STD_LOGIC
   );
end ToggleCounter;

architecture Behavioral of ToggleCounter is

   signal counter  : unsigned(31 downto 0) := (others => '0');
   signal outValue : std_logic := '0';

begin

   process(clk) begin
--   process(counter) begin
      if rising_edge(clk) then
         counter <= counter + 1;
         if counter = CYCLES_PER_HALF_PERIOD_G then
            counter  <= (others => '0');
            outValue <= not(outValue);
         end if;
      end if;
   end process;

   toggleOut <= outValue;

end Behavioral;
```

*Library definitions not shown for space… check out code on github.

```vhdl
entity ToggleCounter2 is
   Generic (
      BIT_SELECT_G : integer := 10
   );
   Port (
      clk       : in  STD_LOGIC;
      toggleOut : out  STD_LOGIC
   );
end ToggleCounter2;

architecture Behavioral of ToggleCounter2 is

   signal counter  : unsigned(BIT_SELECT_G downto 0) := (others => '0');

begin

   process(clk) begin
      if rising_edge(clk) then
         counter <= counter + 1;
      end if;
   end process;

   toggleOut <= counter(BIT_SELECT_G);

end Behavioral;
```

Module Name	Partition	Slices	Slice Reg	LUTs
TopLevel		0/37	0/64	0/108
U_ToggleCounter		21/21	33/33	74/74
U_ToggleCounter2		7/7	25/25	25/25

ToggleCounter compares a 32-bit counter against a specific 32-bit value. Takes significantly more resources, but can hit its target exactly.

Blinky Lights SR Implementation (1)

- Per UG384 – “Spartan-6 FPGA Configurable Logic Block User Guide”

Shift Registers (SRLs) Primitive

One primitive is available for the 32-bit shift register (SRLC32E). Figure 40 shows the 32-bit shift register primitive.

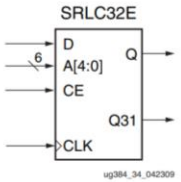


Figure 40: **32-bit Shift Register**

Instantiating several 32-bit shift register with dedicated multiplexers (F7AMUX, F7BMUX, and F8MUX) allows a cascadable shift register chain of up to 128-bit in a slice. Figure 20 through Figure 22 in the *Shift Registers (SLICEM only)* section of this document illustrate the various implementation of cascadable shift registers greater than 32 bits.

Port Signals

Clock – CLK

Either the rising edge or the falling edge of the clock is used for the synchronous shift operation. The data and clock enable input pins have setup times referenced to the chosen edge of CLK.

Data In – D

The data input provides new data (one bit) to be shifted into the shift register.

Clock Enable - CE

The clock enable pin affects shift functionality. An inactive clock enable pin does not shift data into the shift register and does not write new data. Activating the clock enable allows the data in (D) to be written to the first location and all data to be shifted by one location. When available, new data appears on output pins (Q) and the cascadable output pin (Q31).

Address – A[4:0]

The address input selects the bit (range 0 to 31) to be read. The nth bit is available on the output pin (Q). Address inputs have no effect on the cascadable output pin (Q31). It is always the last bit of the shift register (bit 31).

FPGA has shift-register primitives.
If we can map our logic onto them specifically, we can make more efficient use of the FPGA primitives.

So we used as a base a Shift Register:

```
entity ShiftReg32 is
   generic (
      BIT_CHOICE_G : integer range 0 to 31 := 31
   );
   port (
      clk       : in  STD_LOGIC;
      clkEn     : in  STD_LOGIC;
      toggleOut : out STD_LOGIC
   );
end ShiftReg32;

architecture Behavioral of ShiftReg32 is

   signal shiftReg : std_logic_vector(31 downto 0) := (0 => '1', others => '0');

begin

   process(clk) begin
      if rising_edge(clk) then
         if clkEn = '1' then
            shiftReg <= shiftReg(30 downto 0) & shiftReg(31);
         end if;
      end if;
   end process;

   toggleOut <= shiftReg(BIT_CHOICE_G);

end Behavioral;
```

Blinky Lights SR Implementation (2)

- We've written a very specific kind of shift register:
  - Bits rotate around from top to bottom (rather than losing the bit that's shifted away).
    - Sometimes called a "barrel shifter."
  - Our shift register has 1-bit high, all others low.
  - One bit is selected as the output to the port.

```
entity ShiftReg32 is
   generic (
      BIT_CHOICE_G : integer range 0 to 31 := 31
   );
   port (
      clk       : in  STD_LOGIC;
      clkEn     : in  STD_LOGIC;
      toggleOut : out STD_LOGIC
   );
end ShiftReg32;

architecture Behavioral of ShiftReg32 is

   signal shiftReg : std_logic_vector(31 downto 0) := (0 => '1', others => '0');

begin

   process(clk) begin
      if rising_edge(clk) then
         if clkEn = '1' then
            shiftReg <= shiftReg(30 downto 0) & shiftReg(31);
         end if;
      end if;
   end process;

   toggleOut <= shiftReg(BIT_CHOICE_G);

end Behavioral;
```

This has limitations. Since the shift register is 32-bits long, it can at most give you a pulse every 32-cycles.

So if we want to divide down by a large number (i.e., to turn 40 MHz into ~1 Hz) then we need to chain many of these together…

Blinky Lights SR Implementation (3)

- Example of chaining these together…

```
entity ToggleCounterShiftReg is
   port (
      clk       : in  STD_LOGIC;
      toggleOut : out STD_LOGIC
   );
end ToggleCounterShiftReg;

architecture Behavioral of ToggleCounterShiftReg is

   signal stage0   : std_logic;
   signal stage1   : std_logic;
   signal stage2   : std_logic;
   signal stage3   : std_logic;
   signal stage4   : std_logic;
   signal doToggle : std_logic;
   signal outValue : std_logic := '0';

begin

   doToggle <= stage0 and stage1 and stage2 and stage3 and stage4;

   process(clk) begin
      if rising_edge(clk) then
         if doToggle = '1' then
            outValue <= not(outValue);
         end if;
      end if;
   end process;

   toggleOut <= outValue;

   U_ShiftReg0 : entity work.ShiftReg32
      generic map (
         BIT_CHOICE_G => 31
      )
      port map (
         clk       => clk,
         clkEn     => '1',
         toggleOut => stage0
      );
```

```
   U_ShiftReg1 : entity work.ShiftReg32
      generic map (
         BIT_CHOICE_G => 31
      )
      port map (
         clk       => clk,
         clkEn     => stage0,
         toggleOut => stage1
      );

   U_ShiftReg2 : entity work.ShiftReg32
      generic map (
         BIT_CHOICE_G => 31
      )
      port map (
         clk       => clk,
         clkEn     => stage0 and stage1,
         toggleOut => stage2
      );

   U_ShiftReg3 : entity work.ShiftReg32
      generic map (
         BIT_CHOICE_G => 31
      )
      port map (
         clk       => clk,
         clkEn     => stage0 and stage1 and stage2,
         toggleOut => stage3
      );

   U_ShiftReg4 : entity work.ShiftReg32
      generic map (
         BIT_CHOICE_G => 7
      )
      port map (
         clk       => clk,
         clkEn     => stage0 and stage1 and stage2 and stage3,
         toggleOut => stage4
      );


end Behavioral;
```

Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM
TopLevel		0/37	0/64	0/108	0/5
U_ToggleCounter		21/21	33/33	74/74	0/0
U_ToggleCounter2		7/7	25/25	25/25	0/0
U_ToggleCounterShiftRegister		4/9	1/6	4/9	0/5

Even more efficient, as expected.
…but frequency is a bit off.

Debugging in Simulation

- Even for this extremely simple set of firmware, compilation takes a couple minutes.
- This quickly grows to order of an hour for larger devices, more complicated designs.
  - Doing quick change-compile-check is not feasible.

➔ Whenever possible, debug in simulation first…

Simulation Code

- Our project is small enough that we can actually simulate the top level.
  - Usually, simulate only smaller blocks!
- To simulate, you need a “testbench”:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY TopLevelTb IS
END TopLevelTb;

ARCHITECTURE behavior OF TopLevelTb IS

    -- Component Declaration for the Unit Under Test (U

    COMPONENT TopLevel
    PORT(
         CLK_33MHZ_SYSACE : IN  std_logic;
         GPIO_LED_0 : OUT  std_logic;
         GPIO_LED_1 : OUT  std_logic;
         GPIO_LED_2 : OUT  std_logic;
         GPIO_LED_3 : OUT  std_logic;
         FPGA_AWAKE : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal CLK_33MHZ_SYSACE : std_logic := '0';

  --Outputs
   signal GPIO_LED_0 : std_logic;
   signal GPIO_LED_1 : std_logic;
   signal GPIO_LED_2 : std_logic;
   signal GPIO_LED_3 : std_logic;
   signal FPGA_AWAKE : std_logic;

   -- Clock period definitions
   constant CLK_33MHZ_SYSACE_period : time := 3.03 ns;

BEGIN

  -- Instantiate the Unit Under Test (UUT)
   uut: TopLevel PORT MAP (
          CLK_33MHZ_SYSACE => CLK_33MHZ_SYSACE
```

```
  -- Instantiate the Unit Under Test (UUT)
   uut: TopLevel PORT MAP (
          CLK_33MHZ_SYSACE => CLK_33MHZ_SYSACE,
          GPIO_LED_0 => GPIO_LED_0,
          GPIO_LED_1 => GPIO_LED_1,
          GPIO_LED_2 => GPIO_LED_2,
          GPIO_LED_3 => GPIO_LED_3,
          FPGA_AWAKE => FPGA_AWAKE
        );

   -- Clock process definitions
   CLK_33MHZ_SYSACE_process :process
   begin
    CLK_33MHZ_SYSACE <= '0';
    wait for CLK_33MHZ_SYSACE_period/2;
    CLK_33MHZ_SYSACE <= '1';
    wait for CLK_33MHZ_SYSACE_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      wait for CLK_33MHZ_SYSACE_period*10;

      -- insert stimulus here

      wait;
   end process;

END;
```

Verilog testbenches also possible.

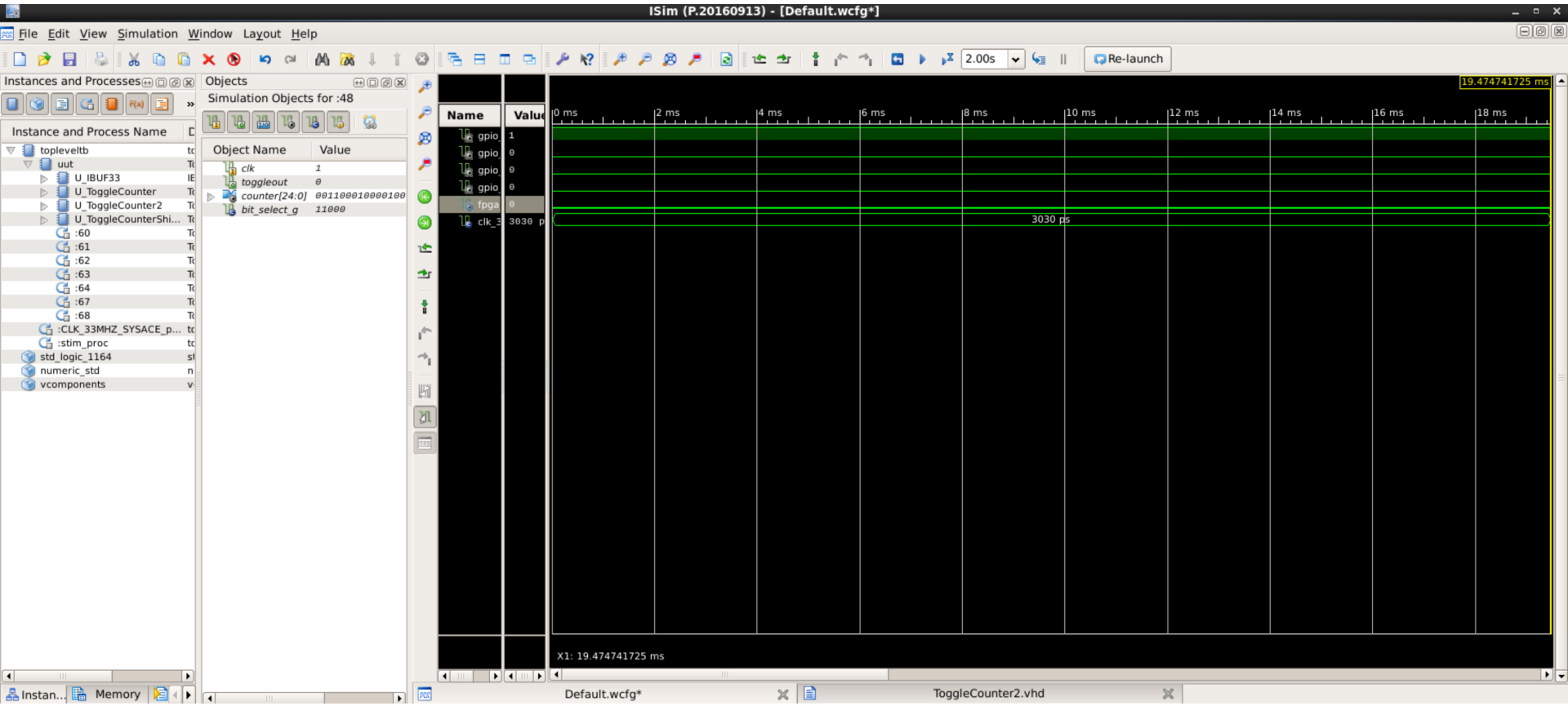
Note no ports in the entity description.

Note <u>non-synthesizable statements</u> like “wait 100 ns”
➔ Synthesized logic has no concept of time.

➔ Note even here, though we *can* simulate top level, it may not be a good idea…

Simulation Output

- This simulation has gotten about 1/50th through one desired cycle after a few minutes…



Optimizing for Simulation

- Can be useful to have hooks (usually as generics) in your code that you can use to speed up simulation of processes that are usually slow.
- For example… if you simulate only ToggleCounter…

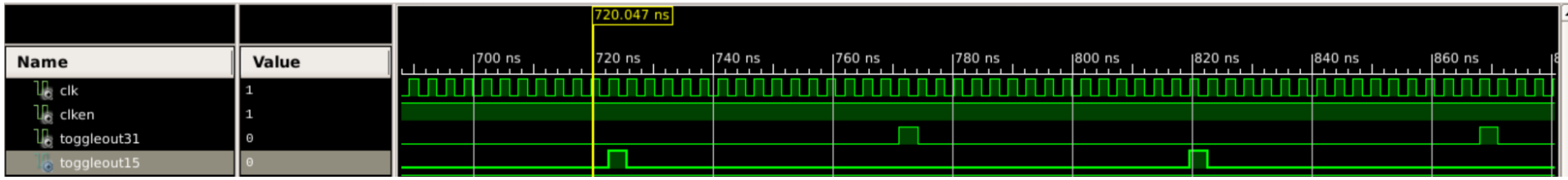
```
entity ToggleCounter is
    Generic (
        CYCLES_PER_HALF_PERIOD_G : integer := 50000000
    );
    Port (
        clk       : in  STD_LOGIC;
        toggleOut : out  STD_LOGIC
    );
end ToggleCounter;
```

  - You can use a smaller value for the generic CYCLES_PER_HALF_PERIOD_G when you instantiate in the simulation file.

Debugging the SR Implementation

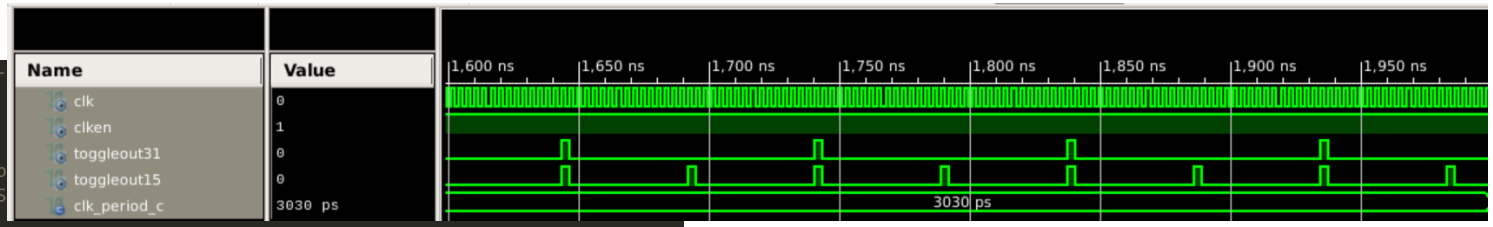
- Let’s instantiate our SR / barrel shifter.
- Actually, let’s do two of them with different parameters and see what we get…
  - ShiftReg32Tb.vhd
    - (with some intentional errors)

What's wrong?




```vhdl
entity ShiftReg32 is
   generic (
      BIT_CHOICE_G : integer range 0 to 31 := 31
   );
   port (
      clk       : in  STD_LOGIC;
      clkEn     : in  STD_LOGIC;
      toggleOut : out STD_LOGIC
   );
end ShiftReg32;

architecture Behavioral of ShiftReg32 is

   signal shiftReg : std_logic_vector(31 downto 0) := (0 => '1', others => '0');

begin

   process(clk) begin
      if rising_edge(clk) then
         if clkEn = '1' then
            shiftReg <= shiftReg(30 downto 0) & shiftReg(31);
         end if;
      end if;
   end process;

   toggleOut <= shiftReg(BIT_CHOICE_G);

end Behavioral;
```

Fixed Version…




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following lib
-- arithmetic functions with S
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ShiftRegMax32 is
   generic (
      BIT_CHOICE_G : integer range 1 to 31 := 31
   );
   port (
      clk       : in  STD_LOGIC;
      clkEn     : in  STD_LOGIC;
      toggleOut : out STD_LOGIC
   );
end ShiftRegMax32;

architecture Behavioral of ShiftRegMax32 is

   signal shiftReg : std_logic_vector(BIT_CHOICE_G downto 0) := (0 => '1', others => '0');

begin

   process(clk) begin
      if rising_edge(clk) then
         if clkEn = '1' then
            shiftReg <= shiftReg(shiftReg'left-1 downto 0) & shiftReg(shiftReg'left);
         end if;
      end if;
   end process;

   toggleOut <= shiftReg(BIT_CHOICE_G);

end Behavioral;
```

Hurray!

Exercise for next time…

Look through the module that instantiates this

ToggleCounterShiftRegMax32.vhd

Modify it to get as close to 1 Hz toggling as you can.

Other notes… next time?

- Synthesis – max clock rate estimates.
- Timing constraints.
- IO standards.
- Records.
- Packages (e.g., StdRtlPkg.vhd).