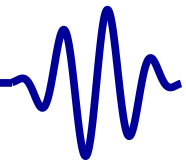
2. Registers

Gisselquist Technology, LLC

Daniel E. Gisselquist, Ph.D.

Lesson Overview

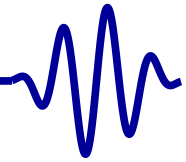


- What is a register (**reg**)?
- How do things change with time?
- Discover the system clock

Objectives

- Something

Registers



- Wires have no memory
- Only registers can hold state

  Well, that and memories, but we haven't gotten to those yet

Memoryless Regs


```
reg      [7:0]    A;

always @(*)
         A = 9'h87;

assign   o_led = A ^ i_sw;
```

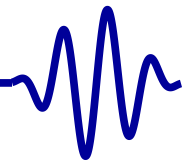
- Registers can be assigned in always blocks.
- Always blocks may consist of one statement, or
- Many statements between a **begin** and **end** pair

Flip Flops


```
        reg     [9:0]   A;

        always @(posedge i_clk)
                A <= A + 1'b1;
```

- Any registers set within an **always** @(**posedge** i_clk) block will transitions to their new values on the next clock edge only
  - *Only put a bonafide clock edge should be used*
- Note that we are using <= for assignment
  - This is a *non-blocking* assignment
  - Most, if not all, clocked register should be set with <=

Blocking



- This is a non-blocking assignment

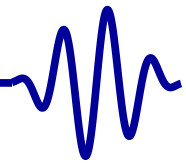
```
always @(posedge i_clk)
        A <= A + 1'b1;
```

- Blocking assignment

```
always @(posedge i_clk)
        A = A + 1'b1;
```

- A blocking assignment's value may be referenced again before the clock edge
  - Creates the appearance of time passing within the block
  - *It may also cause simulation-hardware mismatch*
  - *Use with caution*
- In this case, both generate the same logic

All in Parallel

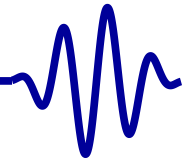


- A design may contain multiple always blocks
- The hardware will execute all at once
- The simulator will execute one at a time

Rules: When using the simulator, . . .

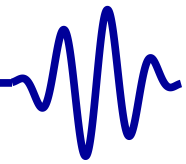
- Make sure your design can be synthesized
- Make sure it fits within your device
- Make sure it maintains an appropriate clock rate

Feedback



- Wires in a loop created circular logic
- Registers in a loop creates feedback

Blinky



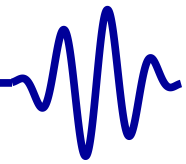
Let's make an LED blink!

```
module blinky(i_clk, o_led);
        input   wire    i_clk;
        output  wire    o_led;

        reg     [26:0]  counter;

        always @(posedge i_clk)
                counter <= counter + 1'b1;

        assign  o_led = counter[26];
endmodule
```

Feel free to synthesize and try this

- The LED should blink at a steady rate

Broken Blinky

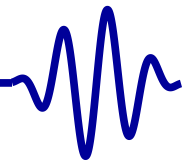


Don't do this: Here's a common beginner mistake

```
reg      counter;
always @(posedge i_clk)
         counter <= counter + 1'b1;
assign   o_led = counter;
```

- Notice that counter is only 1-bit
- This will blink at half the `i_clk` frequency
- Result is too fast to see the changes
- Need to slow it down

Verilator

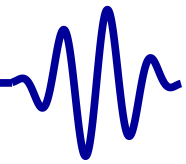


Simulating our design now requires a clock:

```
void    tick(Vblinky *tb) {
        // The following eval() looks
        // redundant ... many of hours
        // of debugging reveal its not
        tb->eval();
        tb->i_clk = 1;
        tb->eval();
        tb->i_clk = 0;
        tb->eval();
}
```

- We'll need to toggle the clock input for anything to happen
- This operation is so common, it deserves its own function, **tick()**

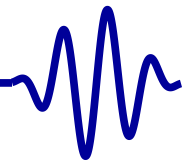
Verilator



We can now simplify our main loop a touch

```
int main(int argc, char **argv) {
    int last_led;
    // .... Setup

    last_led = tb->o_led;
    for(int k=0; k<(1<<20); k++) {
        // Toggle the clock
        tick(tb);

        // Now let's print the LEDs value
        // anytime it changes
        if (last_led != tb->o_led) {
            printf("k = %7d, ", k);
            printf("led = %d\n", tb->o_led);
        } last_led = tb->o_led;
    }
```

Verilator



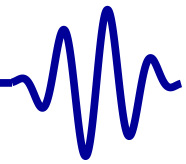
Can we simulate this? Not easily

- Counting to 2^{27} may take seconds in hardware, but ...
- It's extreme slow in simulation. Let's slow blinky down in the Sim.
- We can do this by adjusting the width of the counter

We'll use a parameter to do this

```
parameter       W=27;
reg     [W-1:0] counter;
// ....
assign  o_led = counter[W-1];
```

Parameters



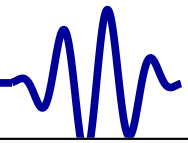
Parameters are very powerful! They allow us to

- Reconfigure a design, after it's been written
- Examples:
  - ZipCPU cache sizes can be adjusted by parameters
  - Internal memory sizes, implement the divide instruction or not, specify the type of multiply
  - Default serial port speed, number of GPIO pins supported by a GPIO controller, and more

Verilator argument `-GW=12` sets the `W` parameter to `12`

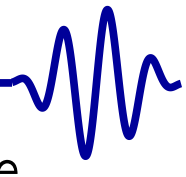
```
% verilator -Wall -GW=12 -cc blinky.v
```

Sim Result




```
% ./blinky
k =     2047, led = 1
k =     4095, led = 0
k =     6143, led = 1
k =     8191, led = 0
k =    10239, led = 1
k =    12287, led = 0
k =    14335, led = 1
k =    16383, led = 0
k =    18431, led = 1
k =    20479, led = 0
# .... (Lines skipped for brevity)
%
```

Trace Generation



This is easy. For more complex designs, we'll need a trace

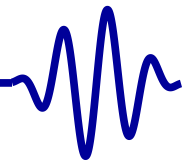
- That means writing to a trace file on every clock

Steps

1. Add the `--trace` option to Verilator
2. Create a trace from your .cpp file

```
#include "verilated_vcd_c.h"
// ...
int main(int argc, char **argv) {
        // ...
        unsigned tickcount = 0;
        // ...
```

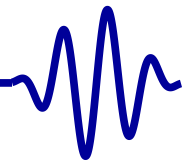
Trace Generation



Create the trace file within C++

```
// ...
int main(int argc, char **argv) {
        // ...
        // Generate a trace
        Verilated::traceEverOn(true);
        VerilatedVcdC* tfp = new VerilatedVcdC;
        tb->trace(tfp, 00);
        tfp->open("blinkytrace.vcd");

        // ...
        for(int k=0; k<(1<<20); k++) {
                tick(++tickcount, tb, tfp);
                // ...
        }
}
```

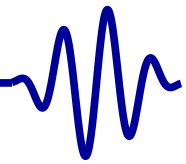
Trace Generation



3. Write trace data on every clock

```
void    tick(int tickcount, Vblinky *tb,
                            VerilatedVcdC* tfp) {
        tb->eval();
        if (tfp)
                tfp->dump(tickcount * 10 - 2);
        tb->i_clk = 1;
        tb->eval();
        if (tfp)
                tfp->dump(tickcount * 10);
        tb->i_clk = 0;
        tb->eval();
        if (tfp) {
                tfp->dump(tickcount * 10 + 5);
                tfp->flush();
        }
```

Trace Generation



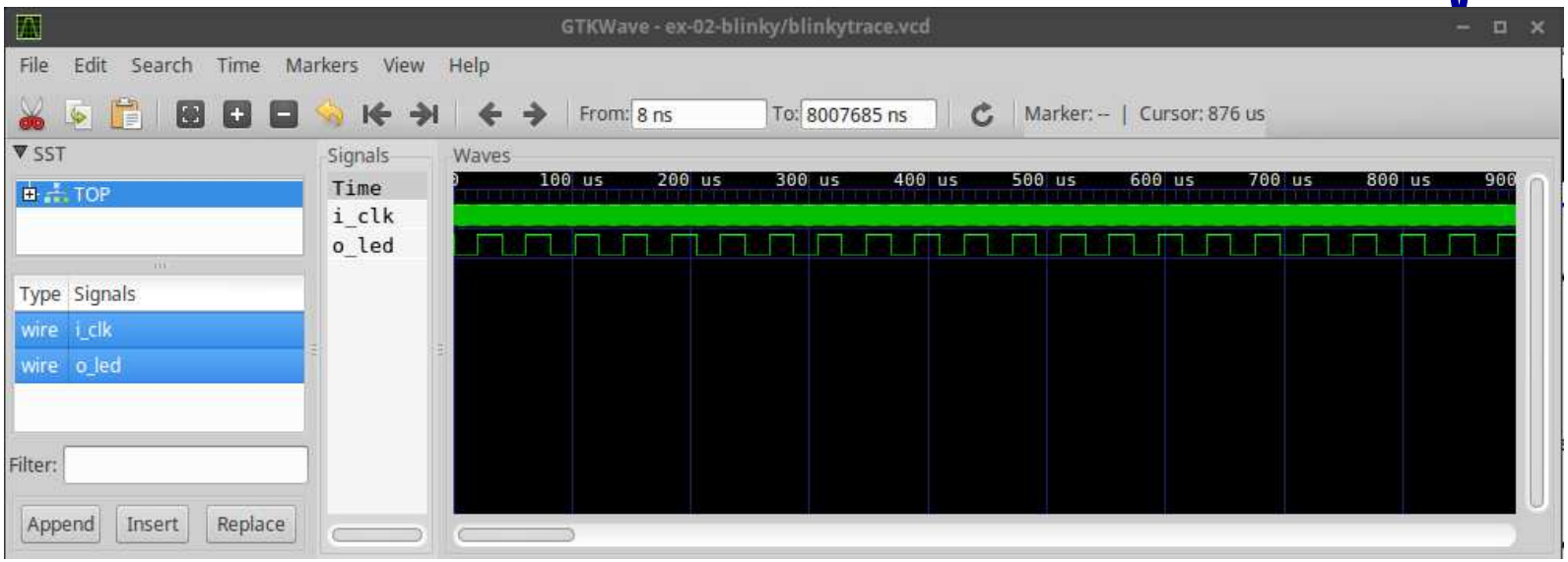
- Now, running blinky will generate a trace

```
% ./blinky
# ...
```

- You can view it with GTKwave

```
% gtkwave blinkytrace.vcd
```

GTKWave



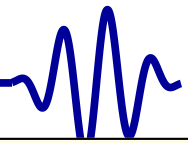
This is how debugging is done

- The simulator trace shows you every registers value
- . . . at every clock tick
- You can zoom in to find any bugs

Strobe


```
module strobe(i_clk, o_led);
        input   wire    i_clk;
        output  wire    o_led;

        reg     [26:0]  counter;

        always @(posedge i_clk)
                counter <= counter + 1'b1;

        assign  o_led = &counter[26:24];
endmodule
```

Toggled




```
module blinky(i_clk, i_sw, o_led);
        input   wire    i_clk, i_sw;
        output  wire    o_led;

        reg     [26:0]  counter;

        always @(posedge i_clk)
        if (i_sw)
                counter <= counter + 1'b1;

        assign  o_led = counter[26];
endmodule
```

PPS


```
module pps(i_clk, o_led);
        parameter CLOCK_RATE_HZ = 100_000_000;
        parameter [31:0] INCREMENT
                        = (1<<30)/(CLOCK_RATE_HZ/4);
        input    wire     i_clk;
        output   wire     o_led;

        reg      [31:0]   counter;

        always @(posedge i_clk)
                counter <= counter + INCREMENT;

        assign   o_led = counter[31];
endmodule
```

- After `CLOCK_RATE_HZ` clock edges, the counter will roll over

PPS


```
parameter CLOCK_RATE_HZ = 100_000_000;
parameter [31:0] INCREMENT
                 = (1<<30)/(CLOCK_RATE_HZ/4);
always @(posedge i_clk)
          counter <= counter + INCREMENT;
```

- After `CLOCK_RATE_HZ` clock edges, the counter will roll over
- The divide by four above, on both numerator and denominator, is just to keep this within 32-bit arithmetic

$$\texttt{INCREMENT} = \frac{2^{32}}{\texttt{CLOCK\_RATE\_HZ}}$$

- This is called a *fractional clock divider*
  - The division isn't exact
  - It's often good enough

Stretch


```
module stretch(i_clk, o_led);
        input   wire    i_clk;
        output  wire    o_led;

        reg     [26:0]  counter;

        always @(posedge i_clk)
        if (i_sw)
                counter <= 0;
        else if (! (&counter))
                counter <= counter + 1;

        assign  o_led = !counter[26];
endmodule
```

FPGA signals are often too fast to see

Stretch


```
module stretch(i_clk, o_led);
        // ...
        reg      [26:0]  counter;
        always @(posedge i_clk)
        if (i_sw)
                counter <= 0;
        else if (! (&counter))
                counter <= counter + 1;
        assign  o_led = !counter[26];
endmodule
```

FPGA signals are often too fast to see

- This slows them down to eye speed
- Only works for a single event though
- Multiple events would overlap, and be no longer distinct

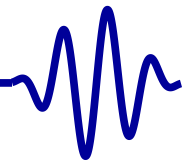
Too Slow


```
module tooslow(i_clk, o_led);
        input   wire    i_clk;
        output  wire    o_led;

        parameter                       NBITS = 1024;
        reg     [NBITS-1:0]             counter;

        always @(posedge i_clk)
                counter <= counter + 1;

        assign  o_led = counter[NBITS-1];
endmodule
```

This is guaranteed to fail a timing check

- It's now time to learn how to check timing
- This design should fail, for reasonable clock speeds

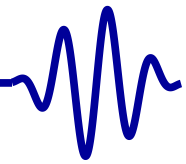
Dimmer



Can you tell me what this will do?

```
module dimmer(i_clk, o_led);
        input   wire    i_clk;
        output  wire    o_led;

        reg     [27:0]  counter;

        always @(posedge i_clk)
                counter <= counter + 1;

        assign  o_led = (counter[7:0]
                        < counter[27:20]);
endmodule
```

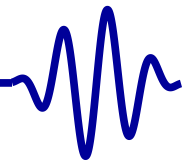
Exercise #1



The walking LED

- Required hardware: 4+ LED's
- Implement blinky, where:
  - Only one LED is ever on at a time
  - The LED that is on moves back and forth

Exercise #2



Knight Rider's LEDs

- Cause the walking LED to fade rather than turn off
- The LED should be (roughly) off by the time it is turned on again