2. Registers

Daniel E. Gisselquist, Ph.D.

Gisselquist Technology, LLC
Lesson Overview

- What is a register (\texttt{reg})?
- How do things change with time?
- Discover the system clock

Objectives

- Learn how to create combinatorial logic with registers
- Learn to create clocked (synchronous) logic
- Understand that registers can “remember” things
- Understand where your System Clock comes from
- Timing Checks, and why they are important
Registers

Why use registers?

- Wires have no memory
- Only registers can hold state (data)

Two basic types, both set with an always

1. Combinatorial: Like wires

   ```
   always @(*)
   A = B;
   ```

   This form can be easier to read when the logic becomes complex

2. Synchronous: Only changes values on a clock

   ```
   always @(posedge i_clk)
   A <= B;
   ```
Registers can only be assigned in `always` blocks.

Always blocks may consist of one statement, or

Many statements between a `begin` and `end` pair

```verilog
always @(*)
  begin
    o_led = A ^ i_sw;
    o_led = o_led + 7;
    if (i_reset)
      o_led = 0;
  end
```
Combinatorial Registers

```
always @(*)
begin
    o_led = A \^ i_sw;
    o_led = o_led + 7;
    if (i_reset)
        o_led = 0;
end
```

This block

- Looks like software
- Acts like you would expect in a simulator
- Takes no time at all in hardware
  - The hardware acts as if all statements were done at once

*Only use “=” in a combinatorial always block*
What happens here?

```verilog
input wire i_S;
input wire [7:0] i_V;
output reg [7:0] o_R;

always @(*)
if (i_S)
  o_R = i_V;
```

This is called a latch

- It requires memory
- May do one thing in simulation, another in hardware
- Most FPGA's don't support latches
- Can have subtle timing problems in hardware

Avoid using latches!
What happens here?

```verilog
always @(*)
begin
    o_R = 0;
    if (i_S)
        o_R = i_V;
end
```

No latch is inferred

- This is a very useful pattern!
- `o_R` now has a default value
  This prevents a latch from being inferred
- No memory is required
- The last assignment gives `o_R` its final value
Any registers set within an `always @(posedge i_clk)` block will transitions to their new values on the next clock edge only

- *Only a bonafide clock edge should be used*
- Do not transition on anything you create in logic

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

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

- Blocking assignment

```vhdl
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
Non-Blocking

What value will be given for \( A \)?

- Assume it starts at zero
- What will it be after one clock tick?

```verilog
always @(posedge i_clk)
begin
    A <= 5;
    A <= A + 1'b1;
end
```

- The assignment only takes place on the clock edge
- Last assignment wins
- \( A \) is set to 1, then 2 on the next clock, 3 on the clock after, etc.
Now what value will be given for \( A \)?

- Assume it starts at zero
- What will it be after one clock tick?

\[
\begin{align*}
\text{always } @\text{posedge } \text{i_clk} \\
\text{begin} \\
A &= 5; \\
A &= A + 1'b1; \\
\text{end}
\end{align*}
\]

- Again, the assignment only takes place on the clock edge
- It appears as though it took several steps
- \( A \) is set to 6
What if something depends upon $A$ in another block?

- Assume $A=0$ before the clock tick

```vhdl
always @(posedge i_clk)
begin
    A = 5;
    A = A + 1'b1;
end

always @(posedge i_clk)
B <= A;
```

- This result is *simulation dependent*!
- $B$ may be set to 0, or it may be set to 6

Don’t do this! Use <= within an `always @(posedge i_clk)`
Non-Blocking

Now what will B be set to?

- Assume \( A = 0 \) before the clock tick

```verbatim
always @(posedge i_clk)
begin
    A <= 5;  // Ignored!
    A <= A + 1'b1;
end

always @(posedge i_clk)
    B <= A;
```

- \( A \) will be set to 1, and \( B \) will be set to 0
- On the next clock, \( A \) will be set to 2 and \( B \) to 1, etc.

Now simulation matches hardware
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 chosen device
  - This is not a simulator task
  - Requires using the synthesizer periodically
- Make sure it maintains an appropriate clock rate
  - We’ll get to timing checks in a moment
Feedback

- Wires in a loop created circular logic
- Clocked registers in a loop creates feedback

```verilog
class assign
  err = i_actual - o_command;
always @(posedge i_clk)
begin
  o_command <= o_command + (err >> 5);
end
```

Feedback is used commonly in control systems
Let’s make an LED blink!

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

    reg [26:0] counter;
    initial counter = 0;
    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
- Rate is determined by the 26 above
Here’s a common beginner mistake

```verilog
reg counter;

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

assign o_led = counter;
```

Don’t make this mistake

- Notice that counter is only 1-bit
- This will blink at half the i_clk frequency
- Result is typically way too fast to see any changes
- LED may glow dimly
- Need to slow it down
Simulating our design (blinky) now requires a clock:

```c
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()`
We can now simplify our main loop a touch

```c
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;
}
```
Can we simulate this? Not easily

- Counting to $2^{27}$ may take seconds in hardware, but . . .
- It’s extreme slow in simulation.
- Let’s speed blinky up–just for simulation
- We can do this by adjusting the width of the counter

We’ll use a parameter to do this

```verilog
parameter WIDTH = 27;
reg [WIDTH-1:0] counter;
// ....
assign o_led = counter[WIDTH-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 `-GWIDTH=12` sets the `WIDTH` parameter to 12

```bash
% verilator -Wall -GWIDTH=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)
%
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 the Verilator command line

   ```
   % verilator -Wall --trace -GWIDTH=12 \
   -cc blinky.v
   ```

2. Create a trace from your .cpp file

   ```
   #include "verilated_vcd_c.h"
   // ...
   int main(int argc, char **argv) {
   // ...
   unsigned tickcount = 0;
   // ...
   ```
Create the trace file within C++

```cpp
// ...
int main(int argc, char **argv) {
    // ...
    // Generate a trace
    Verilated::traceEverOn(true);
    VerilatedVcdC* tfp = new VerilatedVcdC;
    tb->trace(tfp, 99);
    tfp->open("blinkytrace.vcd");

    // ...
    for(int k=0; k<(1<<20); k++) {
        tick(++tickcount, tb, tfp);
        // ...
    }
}
```
3. Write trace data on every clock

```c
void tick(int tickcount, Vblinky *tb, VerilatedVcdC* tfp) {
    tb->eval();
    if (tfp) // dump 2ns before the tick
        tfp->dump(tickcount * 10 - 2);
    tb->i_clk = 1;
    tb->eval();
    if (tfp) // Tick every 10ns
        tfp->dump(tickcount * 10);
    tb->i_clk = 0;
    tb->eval();
    if (tfp) { // Trailing edge dump
        tfp->dump(tickcount * 10 + 5);
        tfp->flush();
    }
}
You’ll need to add `verilated_vcd_c.cpp` to your `g++` build command in order to support generating a trace as well.

```
% export VINC=/usr/share/verilator/include
% g++ -I${VINC} -I obj_dir
  ${VINC}/verilated.cpp
  ${VINC}/verilated_vcd_c.cpp blinky.cpp
  obj_dir/Vblinky__ALL.a -o blinky
```

Now, running `blinky` will generate a trace.

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

You can view it with GTKwave.

```
% gtkwave blinkytrace.vcd
```
This is how logic debugging is done

- The simulator trace shows you every register’s value
- ... at every clock tick
- You can zoom in to find any bugs
How is this design different from blinky?

```verilog
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
```
Can we get an LED to blink once per second?

```vhdl
always @(posedge i_clk)
  if (counter >= CLOCK_RATE_HZ/2 - 1)
  begin
    counter <= 0;
    o_led <= !o_led;
  end else
  counter <= counter + 1;
```

When \( \text{CLOCK\_RATE\_HZ}/2 \) ticks have passed, the LED will toggle

- This structure is known as an integer clock divider
- It offers an exact division
Can we get an LED to blink once per second?

```verilog
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;

initial counter = 0;
always @(posedge i_clk)
    counter <= counter + INCREMENT;
assign o_led = counter[31];
```
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

\[
\text{INCREMENT} = \frac{2^{32}}{\text{CLOCK\_RATE\_HZ}}
\]

- This is called a \textit{fractional clock divider}
  - The division isn’t exact
  - It’s often good enough
module stretch (i_clk, i_event, o_led);
  input wire i_clk, i_event;
  output wire o_led;
  reg [26:0] counter;

  always @(posedge i_clk)
  if (i_event)
    counter <= 0;
  else if (!counter[26])
    counter <= counter + 1;
  assign o_led = !counter[26];
endmodule

FPGA signals are often too fast to see
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
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
Follow your chip vendor’s instructions to do a timing check

- Use your system clock frequency
  - For now, that’s the clock frequency coming into your board
  - We’ll adjust it later

- Make sure this design fails
  - The carry chain takes time to propagate
  - Extra long carry chains take extra long
  - If the propagation doesn’t complete before the next clock . . . your design will fail (like this one)

- From now on, always check timing for a design
  - Before loading it onto a board
  - Every now and then while simulating
Can you tell me what this will do?

```verilog
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
```
Exercises

- Implement blinky on your hardware
- Implement one of the two PPS designs
  - Using a stopwatch, verify the blink rate of 1Hz
  - Make the blinks shorter, but at the same frequency
- Verify that the 1024 bit tooSlow counter will fail timing
- Implement the dimmer