CS152: Introductory Digital Design Laboratory

TA Xu

Thilan Tran

Spring 2021

Contents

CS152: Introductory Digital Design Laboratory

FPGA

- a **field-programmable gate array (FPGA)** is an integrated circuit:
  - designed to be configured after manufacturing
  - AKA programmable circuit
  - components include:
    - logic blocks for implementing combinational and sequential logic
    - interconnects ie. wires for connecting input blocks to logic blocks
    - I/O blocks forexternal connections
  - many applications, useful for prototyping
- FPGA design fundamentals:
  1. design:
     - create modules and define interations between modules
     - control logic and state machine drawings
     - define external I/O
  2. implementation:
     - express each module in HDL source code
     - connect modules in a hierarchical manner
  3. simulation:
     - important debugging tool for FPGA
     - fast to quickly simulate
  4. logic synthesis:
     - a logic synthesis tool analyzes the design and generates a netlist with common cells available to FPGA target
     - converting software to hardware
       - netlist should be functionally equivalent to the original source code
     - done by XST from ISE
  5. technology mapping:
     - synthesized netlist is mapped to device-specific libraries
     - creates another netlist closer to the final target device
     - done by NGDBUILD from ISE
  6. cell placement:
     - the cells instantiated by the final netlist are placed in teh FPGA layout
     - can be time-consuming
     - done by MAP from ISE

7. route:
   - AKA place-and-route in combination with cell placement
   - connecting the cells in the device to match the netlist
   - done by PAR from ISE
8. bitstream generation:
   - produces a programming file AKA a **bitstream** to program the FPGA
   - ie. compiling a FPGA design
   - done by BITGEN from ISE

Verilog

- **Verilog** overview:
  - similar to C eg. case sensitive, same comment style and operators
  - two standards Verilog-1995 and Verilog-2001
  - VHDL is another widely used hardware definitiion language (HDL)
- Verilog has historically served two functions:
  1. create **synthesizable code** ie. describe your digital logic design in a high-level way instead of using schematics
  2. create **behavioral code** to model elements that interact with your design
     - ie. testbenches and models
- data types:
  - main data types are `wire` and `reg`
  - must be declared before used
  - the `wire` type models a basic wire that holds transient values:
    - changing the value on the RHS changes the LHS
    - only wires can be used on the LHS of continuous assign statement
    - default is wire type
  - the `reg` type is anything that stores a value
    - only regs can be used on the LHS of non-continuous assign statements
- data values:
  - values include `1, 0, x/X, z/Z`
    - `x` is an unknown value, `z` represents a high impedance value
  - radices include `d, h, o, b`
  - format is `<size>'<radix><value>` :
    - eg. `8'h7B, 'b111_1011, 'd123`
    - underscores have no value, purely for readability
- operators:
  - bitwise operators eg. `~, &, |, ^, ~^`
  - reduction operators are bitwise operations on a *single* operand and pro-

duce one bit result
  - logical operators eg. `!, &&, ||, ==, !=, ===, !==`
    - triple equals also compare `x, z`
  - arithmetic operators eg. `+, -, *, /, %`
  - shift operators `<<, >>, <<<, >>>`
    - `<<` logical, `<<<` arithmetic
  - conditional operator `sel ? a : b`
  - concatenation operator `{a, b}`
  - replication operator `{n{m}}`
  - assignment operator `= ≤`
    - in a block assignment `=` , assignment is immediate
    - in a non-blocking assignment `≤` , assignment are deferred until all right hand sides has been evaluated, closer to actual hardware behavior
- assignment operator guidelines:
  - for pure sequential logic, use non-blocking
  - for pure combination logic, use blocking
  - do not mix block and non-blocking in the same always block
  - both sequential and combination logic in the same always block, use non-blocking
- a **module** is the basic building block:
  1. define input, output, inout ports
  2. signal declarations eg. `wire [3:0] a;`
  3. concurrent logic blocks:
     - continuous assignments
       - can leave off `assign` keyword for shorthand
     - always block, either sequential or combinatorial
     - initial blocks, used in behavioral modeling
     - forever blocks, used in behavioral modeling
     - continuous assignments are used for assigning to wires
       - all other blocks are used for assigning to registers
  4. instantiations of sub-modules

Example module:

```
module top(a, b, ci, s, co);
    input a, b, ci;
    output s, co;

    wire s;
    reg g, p, co;

    assign s = a ^ b ^ ci;
```

```
    // combinatorial always block using begin/end
    always @* begin // @* is the sensitivity list
        g = a & b;
        p = a | b;
        co = g | (p & ci);
    end
endmodule
```

- the sensitivity list defines when to enter the function in the `begin/end` block
    1. when level sensitive, changes to any signals in the list will invoke the always block, used in combinational circuits
        - eg. `always @ (a or b)`, `always @*`
    2. when edge sensitive, invoke always block on specified signal edges, used in sequential circuits
        - eg. `always @ (posedge clk or posedge reset)`
- always blocks:
    - an `if/else` or case statement may fail to cover all cases
        - add a `default` statement or put statement before case
    - can loop with `while, for, repeat`

Example modulo 64 counter:

```
module counter(clk, rst, out);
    input clk, rst;
    output [5:0] out;

    reg [5:0] out;
    always @(posedge clk or posedge rst)
    begin
        if (rst)
            out ≤ 6'b000000;
        else
            out ≤ out + 1;
    end
endmodule
```

- module instantiation:
    - using other modules within one module
    - restrictions:
        - port order doesn't matter
        - unused output port allowed
        - name must be unique
    - eg. `counter counter1(.clk(test_clk), .rst(test_rst), .out(out));`

Full-adder example using module instantiation:

```
module half_adder(a, b, x, y);
    input a, b;
    output x, y;
    assign x = a & b;
    assign y = a ^ b;
endmodule

module full_adder(a, b, ci, s, co);
    input a, b, ci;
    output s, co;
    wire g, p, pc;
    half_adder(h1(.a(a), .b(b), .x(g), .y(p));
    half_adder(h2(.a(p), .b(ci), .x(pc), .y(s));
    assign co = pc | g;
endmodule
```

- testbench:
    1. instantiate unit under test AKA `uut`
    2. provide inputs
    3. simulate and verify behavior

Testbench example:

```
reg clk, rst;
wire out;
counter uut(.clk(clk), .rst(rst), .out(out));

initial begin
    // assign inputs
end
always begin
    // alternatively, initial with forever block
    #5 clk = ~clk; // every 5 ms, update clk
end
```