# CS152: Introductory Digital Design Laboratory

TA Xu

Thilan Tran

Spring 2021

## Contents

# CS152A: 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
```

# CS152B: Digital Design Project Laboratory

---

# FPGA Implementation

---

- **programmable logic devices (PLDs)** are the larger family of devices that contain FPGAs:
  - alternative to custom **application specific integrated circuits (ASICs)**
  - *pros*:
    - designs can be rapidly loaded onto PLDs, unlike slow and expensive ASIC development process
  - *cons*:
    - the hidden programmable logic for connecting logic blocks are slower, more expensive, and consume more power
    - when high clock frequencies and very dense logic gates are required, only ASICs can be used
      - on the other hand, when logic is not very dense, the chip may have a minimum area determined by the number of I/O pads ie. pad limited, so PLD may still even be cheaper
- **generic array logic (GAL)** devices:
  - any logical expression can be represented as a sum of products
  - GALs provide a programmable array of AND and OR gates
    - every input and inverted input is pulled into AND gates, and groups of these products are fed into separate OR gates
  - modern GALs use EEPROM technology and CMOS switches to allow for reprogrammability
    - past technology used fuses that could only be "blown" ie. programmed once
  - in the AND array, connections can be programmed so that the input term is disconnected from the AND input
    - then, the OR connections can be hardwired since the AND array is fully programmable
  - a structure called a **macrocell** performs other configurations such as OR / NOR polarities and selecting flip-flops
    - ie. determines how the boolean expression is handled and how the associated pins operate
  - to implement a GAL, HDL is converted into a netlist that is then fitted to a target device to create a **fuse map**
  - *pros*:
    - simplicity

  - *cons*:
    * rigidity, with respect to timing or design size
    * very high scaling in cost vs. logic density
      · AND matrix increases in square function of I/O terms
- **complex PLDs (CPLDs)** are the more common macrocell-based PLDs in the industry today:
  - CPLDs have a linear scaling of connectivity to logic density by using a segmented architecture with multipled fixed-size GAL-style logic blocks
  - more scalable because the logic blocks are fixed in size and small enough:
    * sometimes just 5 product terms per macrocell
      · product term sharing allows a macrocell to borrow terms from neighboring cells
    * switch matrix does become more complicated with more pins
  - best for control paths that only require small number of flip-flops
    * eg. small state machines, small registers, control logic
  - *pros*:
    * simplicity
    * better scaling than GALs
  - *cons*:
    * more complex fitting process than GALs due to hierarchical structure and macrocell sharing
    * still limited to simpler control path applications
- **field programmable gate arrays (FPGAs)** are devices that address data path applications:
  - consists of an array of small logic cells, each with a flip-flop, lookup table (LUT), and supporting logic for multiplexing and arithmetic carries
    * boolean expressions can be evaluated through the LUTs, which are implemented as small SRAM arrays
  - cells are arranged on a grid of routing resources that can make connections between arbitrary cells to build logic paths
    * I/O cells are located around the edges of the chip
  - very high logic densities are achieved by scaling the cell array
  - routing now becomes the limiting factor due to the grid interconnect system:
    * routing can an especially long amount of time
    * large, fast designs require iterative routing and placement algorithms
  - a new issue is with skew across the large system:
    * most FPGAs support several global clocks
    * phase or delay locked loops can be used to intentionally de-skew clock signals
  - other FPGA considerations:

    * RAM blocks that can be used in arbitrary width and depth configurations for different purposes
        · also, using logic cells as RAM
    * third-party logic cores
    * I/O cell architecture

# FPGA Performance

---

- the FPGA is a general-purpose device with digital logic building blocks:
    - basic building block is a **logic cell or element (LC/LE)** which contains a look-up table and flip-flop
    - dedicated logic blocks, internal memory, buses, peripherals, controllers, and even processors can be constructed form this general-purpose FPGA logic
        * a processor built with LCs is called a **soft processor**, and must be synthesized and placed
    - *pros*:
        * high level of customization
            · any custom combination of peripherals and controllers
        * soft processors allow for obsolescence mitigation since the processor HDL code is permanent
        * FPGA component versatility allows larger systems to be reduced into a single FPGA
            · reduced components and cost
        * option for hardware acceleration for any software bottlenecks
            · FPGA design tools allow C code to be easily adapted into hardware
    - *cons*:
        * requires additional design by the embedded programmer compared to a traditional off-the-shelf processor
        * design tools are more complex
        * FPGA is more expensive than equivalent processor
            · but can take advantage of FPGA already in the system at no additional cost

## Performance Enhancing Techniques

---

**Non-FPGA Specific**

- the first set of techniques are code manipulation strategies:
  1. compiler optimization levels:
     - optimizations include jump and pop, loop unrolling, function inlining, etc.
     - level 2 is standard, performs all optimizations that do not increase code size
     - level 3 is the highest level that includes all optimizations that can increase code size
  2. manufacturers provide some optimized instructions for FPGAs, eg. `xil_printf` which is 5% the size of the standard `printf`, with some limitations
  3. assembly and in-line assembly is supported
  4. other optimizations such as locality of reference, small data sections, loop length, minimal recursion
- the remaining techniques deal with different memory usages:
  - the fastest option is to put everything in local memory called **BlockRAM (BRAM)**:
    * essentially L1 or L2 cache of the system
    * usually between 32-64 KB
  - the slowest option is to put everything in external memory only:
    * eg. SRAM, SDRAM, or DDR SDRAM
    * incurs memory access time as well as peripheral bus latency
  - if the entire program cannot fit in local memory, we can consider caching external memory:
    * for some systems, such as MicroBlaze architectures, the cache memory is not dedicated silicon but instead constructed from BRAM and LCs
      · this reduces the system frequency since the cache controller adds additional logic and complexity
    * thus, enabling the MicroBlaze cache may improve performance, even with a slower system clock, but can also detract from performance
      · multiple cache misses make it so cached external memory performances even worse than external memory without cache
  - thus, the best performance solution in terms of memory for MicroBlaze systems usually involves partitioning code:
    * critical data, instructions, and stack are placed in local memory, with everything else in external memory
    * data cache is not used, allowing for a larger local memory
    * instruction cache can be used if instructions cannot fit in local memory

**FPGA Specific**

- one class of techniques deal with increasing the FPGA operating frequency:
  1. only connecting utilized peripherals and buses
     - eg. disabling debug logic modules, using trimmed address buses, simpler GPIO pin version, etc.
  2. specifying area and timing constraints so that the FPGA place and route tools perform better
- the remaining techniques deal with hardware acceleration:
  - hardware acceleration consumes additional FPGA resources, but allows some software limitations to be overcome
  1. modules such as dividers and barrel-shifters can be customized to be done in hardware instead of software
     - this consumes more logic but improves performance
  2. arbitrary software bottlenecks can be converted to hardware:
     - custom hardware logic can be designed to offload an FPGA embedded processor
     - usually facilitated by low-latency access points onto the processor
     - tools exist that generate FPGA hardware directly from C code
     - eg. FFT, DES and AES encryption, matrix manipulation, etc.