ECE 385 – Digital Systems Laboratory

Lecture 4 – FPGA Intro

Zuofu Cheng

Fall 2019

Link to Course Website

ECE ILLINOIS

ILLINOIS

TTL Debugging Hints

- If in open-lab, try using oscilloscope to monitor signals in addition to switch box.
  - Go to display -> persistence -> turn on persistence for one second.
  - This will let you see if transient or unusually noisy signals.
- Be aware that TTL ICs are ultimately implemented using real circuits which may behave in an analog fashion.
  - Example, if output of a chip is the wrong logical level while it is connected in your circuit, do not assume that chip is bad. It could be that the output is connected to something which is pulling it to the wrong logic level (e.g. if you have two outputs shorted together).
  - When checking the functionality of an IC, only trust unit testing, not in-circuit testing.

Experiment 4 Goals

- Read tutorials (Introduction to Quartus, Introduction to SystemVerilog) and make a project for the 8-bit logic processor.
  - Full design files are provided for 4-bit version, including testbench
  - Extend to 8-bit (code is well commented), use 8-bit test bench included (1 demo point)
  - Show TA RTL block diagram (generated from Quartus) for 8-bit logic processor
  - Design 3 16-bit adders and test on the FPGA board
    - Carry ripple adder
    - Carry look-ahead adder
    - Carry select adder
  - Will need state machine for testing (provided) and pin assignments (made with pin planner tool)
    - Analyze performance, power, and area for each adder
    - Should only need to write combinational logic (test setup is provided for you)
    - Demo on FPGA board using built in switches and LEDs (don't need switchbox)

Lab 4 – Do Before Lab Session

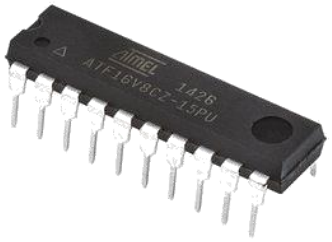
- Install Quartus Prime on home computer or use EWS Windows machines (18.1 recommended– use Lite Edition)
- Read **Introduction to Quartus II** (lab manual)
- Read **Introduction to SystemVerilog** (lab manual)
- Demo points (for next week's lab session)
  - 1.0 point: Functional simulation completed successfully for the 8-bit serial processor (annotations necessary)
  - 1.0 point: RTL block diagram of the 8-bit logic processor extended from 4-bits. This can be automatically generated using Quartus (instructions are in the IQT tutorial)
  - 1.0 point: Correct operation of the Carry-Ripple Adder on the DE2 board
  - 1.0 point: Correct operation of the Carry-Lookahead Adder on the DE2 board using a 4x4 hierarchical design (TA's will look at code)
  - 1.0 point: Correct operation of the Carry-Select Adder on the DE2 board using a 4x4 hierarchical design (TA's will look at code)

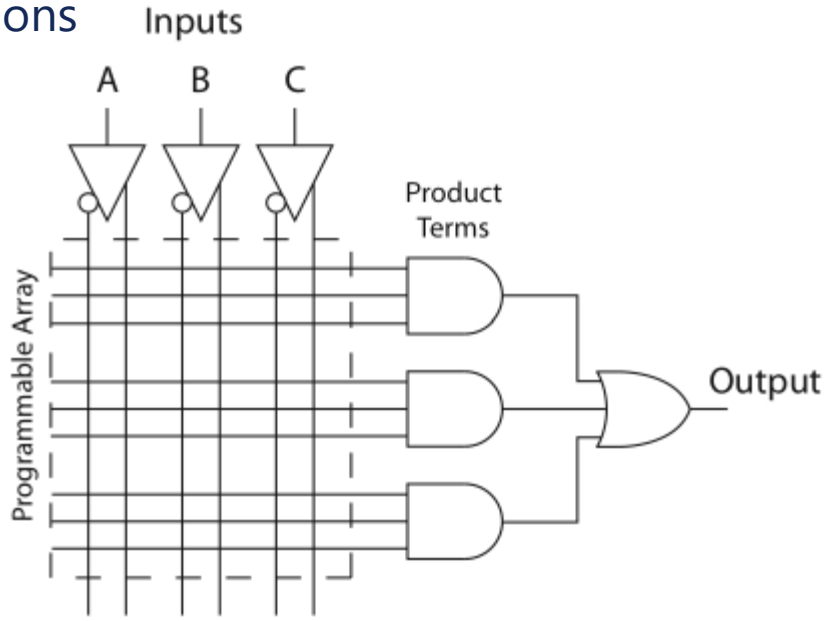
FPGAs and SystemVerilog

- Done with TTL for the semester (after Lab 3)
- Will use SystemVerilog for the rest of the semester
- Understanding SV requires understanding underlying model of hardware
- This is analogous to learning to program in C in 120/220
  - Must understand principles of computers (bits/bytes, logical & arithmetic operations, instructions, memory)
  - Can abstract away details of ISA (instructions, number of registers, memory architecture)
  - SV is the same way, need to understand model & principles behind FPGAs, but do not need to be expert at FPGA architecture

Programmable Logic Devices

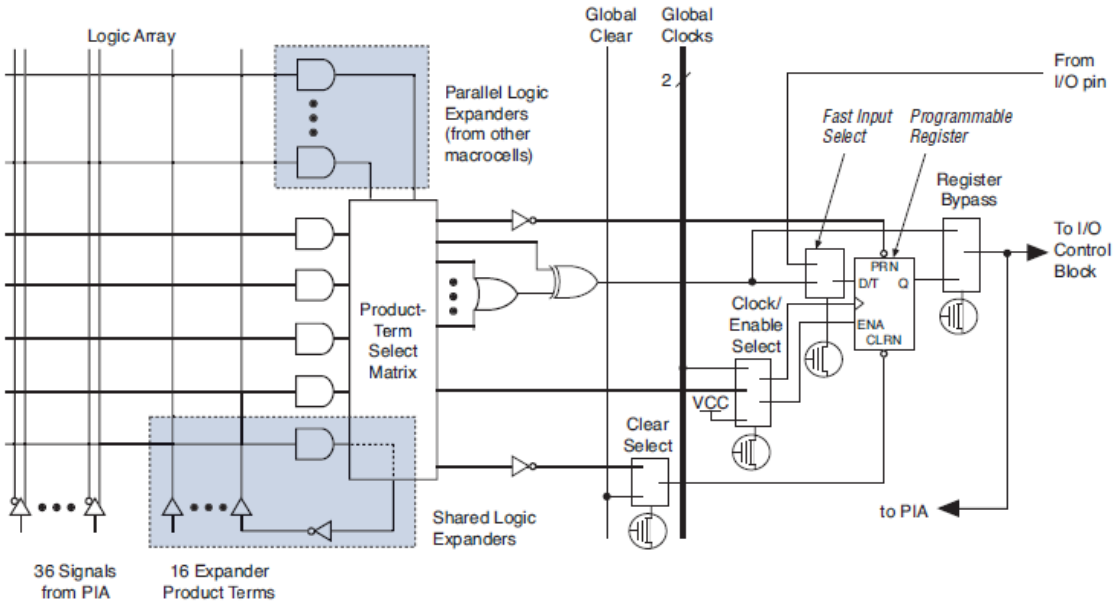
- Evolved from PALs (Programmable Array Logic) and GALs (Generic Array Logic) and CPLD (Complex Programmable Logic Device)
- First generation devices (PALs) are one-time programmable ways to create combinational logical function
- Engineer writes logic expressions and defines I/Os on computer
- Software simplifies expression to SOP form and generates programming file
- Programming cable is used to set fuses / anti-fuses
- No memory / feedback elements limits applications
- Non-volatile (maintains design on power cycle)





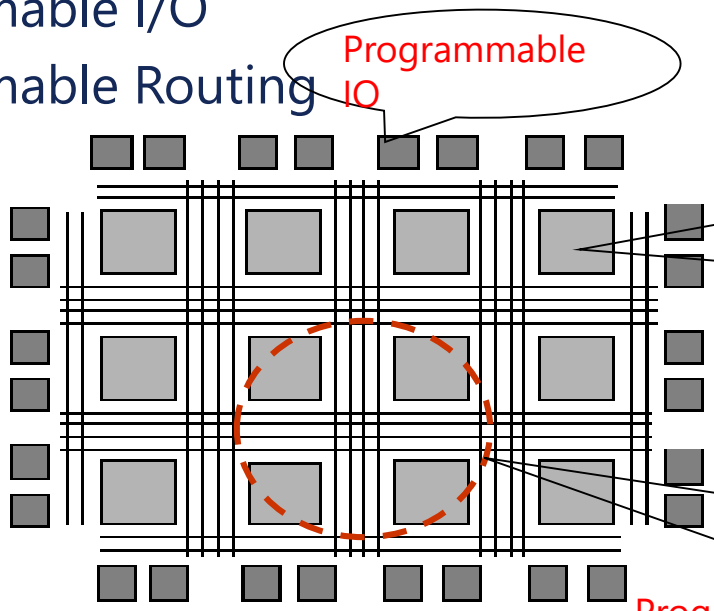
CPLD

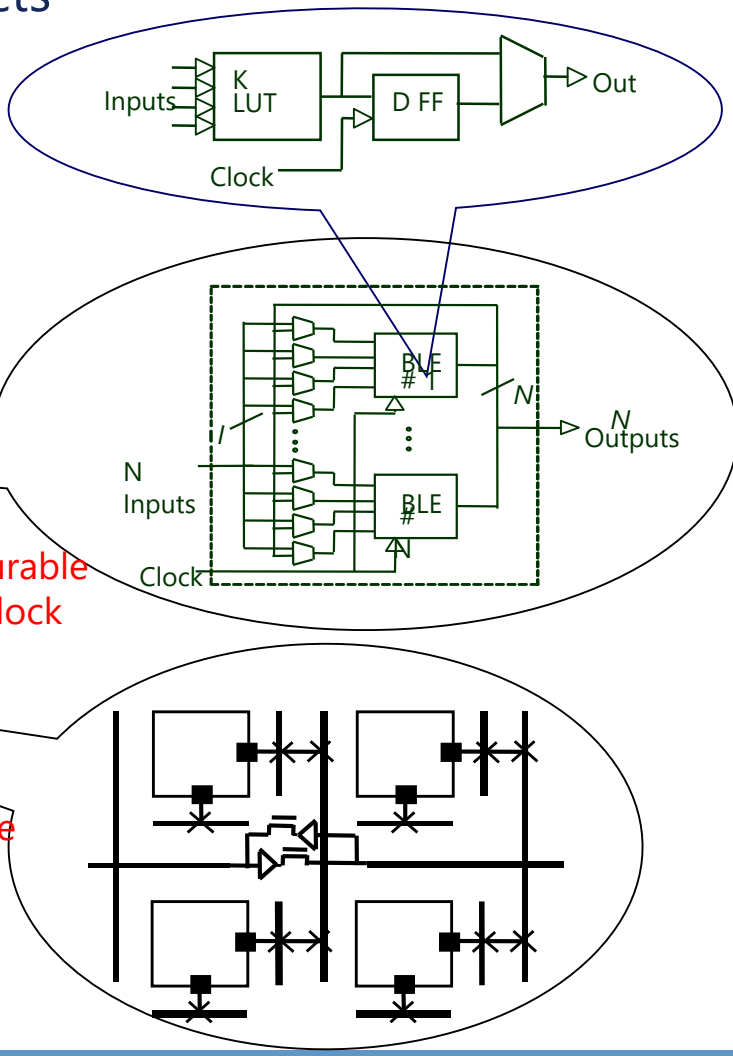
- CPLDs (Complex Programmable Logic Devices) are one step more advanced than PAL/GAL.
- Integrates multiple levels of AND/OR logic with memory (flip flop)
- Typically used for simple designs with small number of logic elements with limited internal routing (feedback)
- Predictable delay due to limited routing options, low density by today standards
- Non-volatile (maintains design on power cycle)



FPGA (Field Programmable Gate Array)

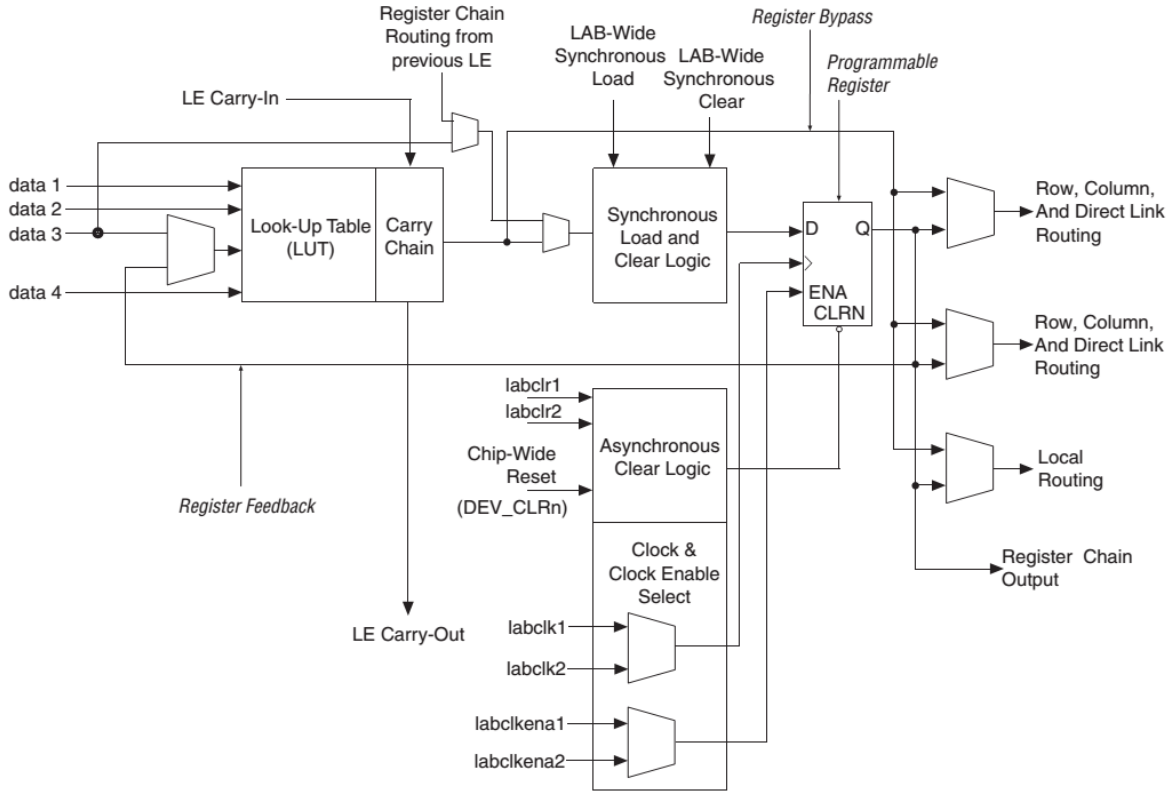
- We will be using FPGAs in this class (Cyclone IV)
- High density and designed for internal interconnects
- Fully programmable through bit-stream file
  - Programmable Logic
  - Programmable I/O
  - Programmable Routing





FPGA (Field Programmable Gate Array)

- FPGA logic element block diagram
- The number on the FPGA (e.g. 115) typically refers to number of LEs (e.g. 115,000 LEs)
- Combination of LEs and routing hardware is referred to as “fabric”
- How does FPGA LE differ from CPLD macrocell?



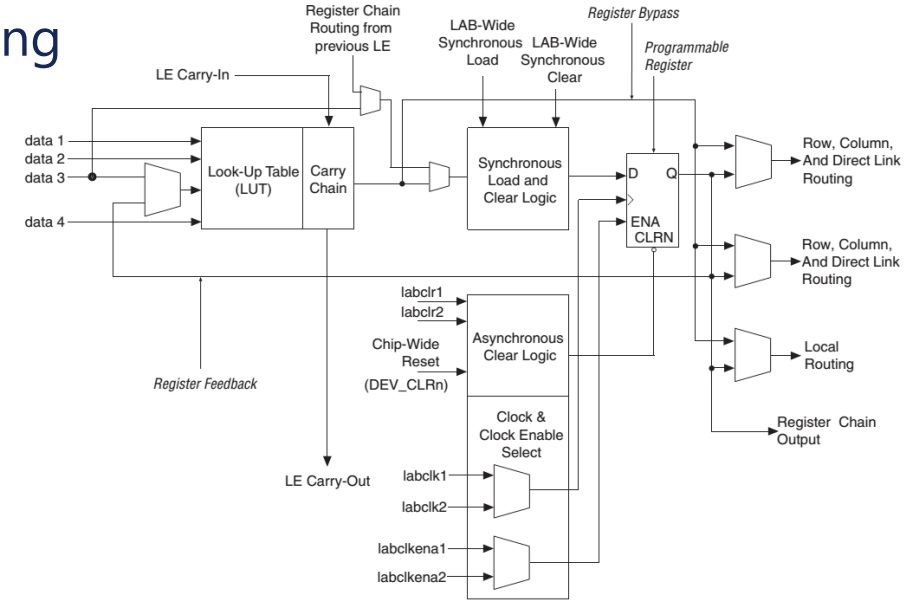
LUT Based Combinational Logic

- FPGAs use LUTs (look-up tables) to resolve combinational logic
- Can be implemented with SRAM for high performance (this means FPGA is volatile, must be programmed on boot-up)
- E.g. 2:1 MUX can be directly implemented in single 4-input LUT (1 unused input)
- 4 input LUT is simply a 16x1 RAM

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

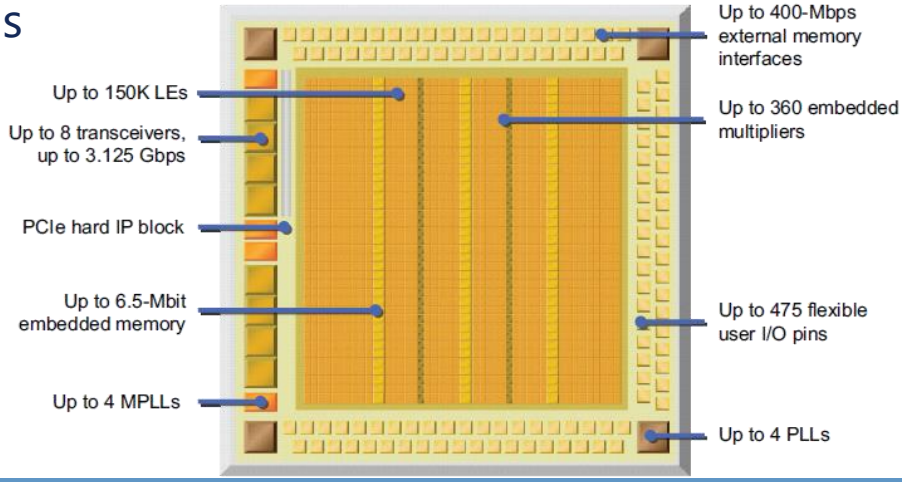
LE Flip-Flop and Internal Routing

- Each LE has a "programmable register" 1-bit memory.
  - May be configured as D, T, J/K, with clock enable
  - Designed for **fully synchronous** design, avoid latches even though FPGA is capable of instantiating them (causes problems for timing tools, prevents use of global clocks)
  - LE registers are also capable of being pre-loaded (at program-time) through reset
  - Routing is handled in a row-column fashion, LEs closer physically have faster routing



Additional FPGA Components

- FPGA has additional components which are not part of the fabric
  - Programmable I/O and Transceivers (FPGA's interface to the outside world)
    - We will use FPGA I/O in single-ended LVTTL mode (similar to TTL but 3.3V)
    - Higher speed devices (cameras, SDR, etc…) may require some type of differential mode (we will not use these in this class)
  - Dedicated Memory (synchronous RAM ~4Mbit)
    - In-fabric memory is extremely limited (1-bit per LE)
    - Dedicated memory blocks when more storage needed
  - Dedicated Multipliers (DSP block)
    - Multipliers are expensive to create with in-fabric logic
    - Used for highly parallel DSP applications



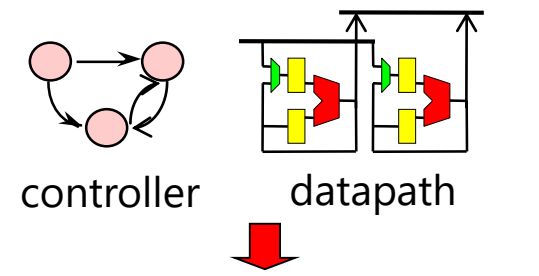
DE2-115 Development Board

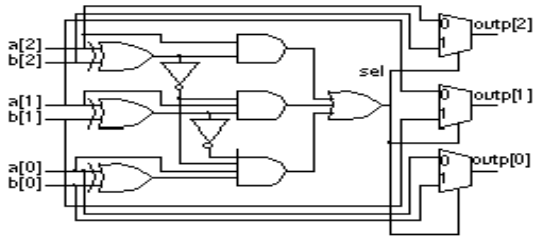
- For clarification: FPGA refers to only the main IC (Cyclone IV EP4CE115)
- For ease of development, we will use DE2-115 development board
  - Includes EP4CE115 and many other peripherals (external to FPGA)
    - LEDs & Switches
    - Clock generation
    - External Memory
    - I/O hardware (VGA, Ethernet, USB, Audio, etc…)
  - Has programming hardware built in (Altera USB Blaster)
- In normal usage scenario, only initial prototyping is done on development board
- Final manufactured designs include only necessary parts to reduce cost

SystemVerilog HDL

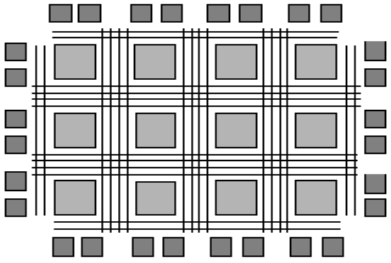
- Two basic flavors of HDLs in industry (Hardware Description Languages)
  - VHDL - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
    - Originally designed to **document** behavior of ICs (describe behavior of complex digital circuits in a textual and unambiguous manner)
    - Meant to be self-documenting and human readable (not initially designed for computer based synthesis
    - Highly verbose and strongly typed (meant to be unambiguous description of IC)
  - Verilog (and SystemVerilog)
    - Designed for simulation and synthesis
    - Borrows some syntax from C programming to aid hardware/software co-design
    - Weakly typed (influenced by C) and less verbose (requires good comments)
    - For this class, we'll use SystemVerilog which has additional improvements over Verilog for simulation and design (we will mostly stick to the subset of SV which is compatible with Verilog)
    - We will use SystemVerilog both for design and verification

FPGA Workflow using HDL





net list



1001110010...

Can simulate at this level (RTL, typical)

RTL design

RTL elaboration and optimization

RTL
(Register Transfer Level)
Synthesis

Architecture-independent optimization

Technology mapping & Architecture-specific optimization

Logic Synthesis

Or this level (post-mapping) atypical

Clustering & placement

Placement-driven optimization & incremental placement

Routing

Physical Design

Bitstream generation

Modules

- A **module** is the primary design unit in SystemVerilog. Multiple modules may be combined to form a larger design. Multiple instances of the same module may exist in design as well.
- **Module declaration** provides the external interface to the design entity. It contains pin-out description, interface description, input-output port definitions etc.
- Good designs are made of individually defined and unit tested modules

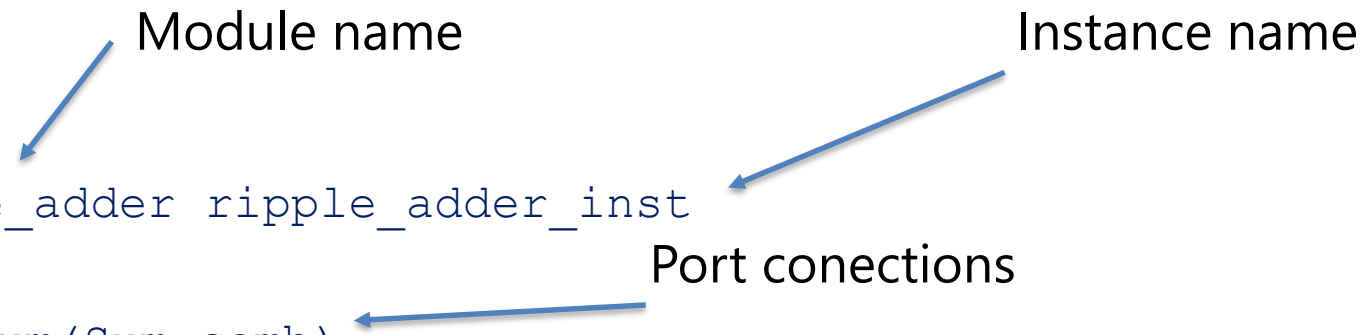
```
// Logic is contained within modules
module Example (input logic clk, enable,  // Single-bit inputs, default logic type
                input logic [2:0] data,   // Three-bit input, default logic type
                output logic out );  // Single-bit output, default logic type

logic [1:0] reg_a; //local variables, in this case, we want register
initial            // "initial" procedure block initializes the variables
begin              // Used for testbench initialization. Not synthesizable.
     out = 0;
end
always_comb         // "always_comb" procedure block for combinational logic
begin
     out = reg_a[0] & data[0];  // Combinational logic goes here
end
always_ff @ (posedge clk)     // "always_ff" procedure block triggered by the
begin                         // rising edge of clk
     reg_a <= data1[1:0];      // Sequential logic goes here
end
endmodule
```

Ports and Data Types

- Modules are defined to the outside world by ports, of which there are 3 types:
  - **`input`** (logical input into the module) (can define a bus by using array notation)
  - **`output`** (logical output from the module) (can define a bus by using array notation)
  - **`inout`** (bidirectional port) **note: the FPGA does not have tri-state buffers as part of internal interconnect network, should only use inout for top-level FPGA module (I/O pins)**
    - Using `inouts` ports in non-top level modules causes problems (may not work, or may have to route out to IOB and route back with lots of delay)
- Data types in SV are vastly simplified over Verilog (`logic` type in almost all cases)
  - `logic select;`
  - `logic [3:0] latch_4bit;`
  - `logic [7:0] my_reg;`
- What "logic" types get synthesized as (connection or flip flop/register) depend on context
  - Sometimes you will also see "wire" type, which can only be used for connection (does not hold value), but in almost all cases "logic" may be used instead (exception for tristates)

Instantiating Modules




```
ripple_adder ripple_adder_inst
(
    .Sum(Sum_comb),
    .CO(CO_comb),
    .A, //This is shorthand for .A(A) when variable and port have the same name
    .* //Automatically connects everything else which fits above criteria - e.g. .clk(clk), .B(B)
);
```

Procedures

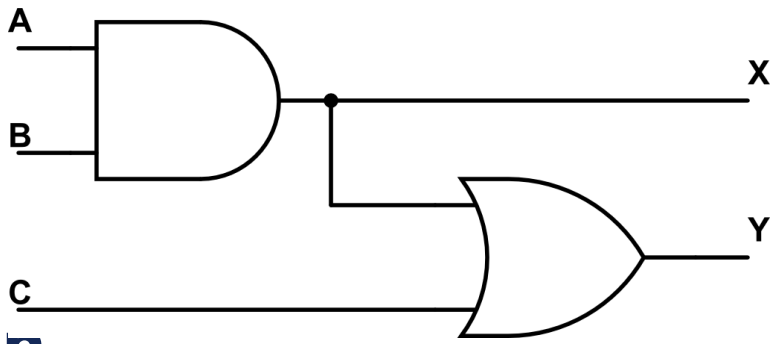
- **Procedure blocks** describe the behavior of an entity or its structure (gates, wires, etc.) using SystemVerilog constructs. There are three types of procedure blocks in SystemVerilog, but we will generally only use the two from Verilog (initial and always, optionally look at fork-join)
- **`initial`:** an initial block is used to initialize testbench assignments and is carried out only once at the beginning of the execution at time zero. It executes all the statements within the "begin" and the "end" statement without waiting. **Non-synthesizable (simulation ONLY with some exceptions)**
- **`always`:** an always block described hardware which operates **under certain conditions (e.g. a clock edge)**. Typically we will use two specialized always blocks:
  - **`always_comb`** : forces synthesizable combinational logic behavior
  - **`always_ff`** : forces synthesizable sequential logic behavior
  - **`always_latch`** : forces synthesizable latch behavior (avoid this!)
- Always think of and partition your modules in terms of combinational and sequential logic (note that this is the underlying model for the FPGA)
- You might encounter "sensitivity list" in Verilog code, this is necessary in Verilog to limit scope of simulator (to reduce CPU power required for simulation). This is unnecessary in SV if you use `always_comb` and `always_ff` blocks (introduced in SV)

Assignment Inside `Always` Procedure

- Two types of assign function inside `always` procedure
  - = blocking assignment (a = b) (sequential assignments)
  - <= non-blocking assignment (a <= b) (simultaneous assignments)
- General rules
  - Always use blocking for combinational logic
  - Always use non-blocking for sequential logic
- Using wrong assignment sometimes creates same result in synthesis but different results in simulation (different behavior between synthesis and simulation = bad)
- Why? Let's take a look at a couple of examples
- For detailed explanation, see: Cummings, C. E. (2000). Nonblocking assignments in Verilog synthesis, coding styles that kill!. *SNUG (Synopsys Users Group) 2000 User Papers*.

Combinational `Always` Procedure

- Consider function to right
  - X = A•B
  - Y = X + C



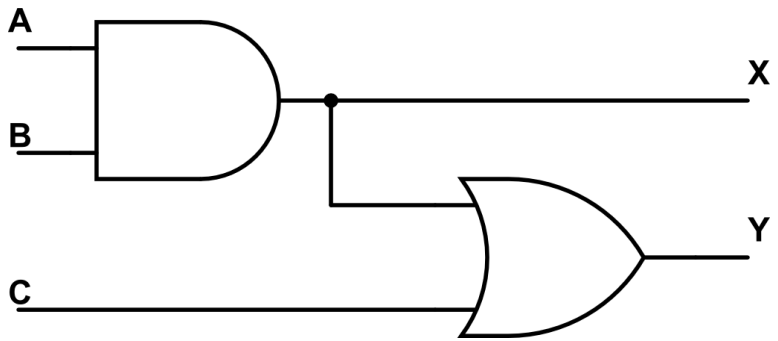
- Note: Combinational function
- Correct way to write (inside `always_comb` procedure)

```
X = A & B;
Y = X | C;
```

- Suppose A = 1, B = 1, C = 0 (initially) and A changes to 0
  - What is expected value for X and Y

Combinational `Always` Procedure

A	B	C	X	Y
1	1	0	1	1
0	1	0	1	1



- **Incorrect way to write**

```
X <= A & B;
Y <= X | C;
```

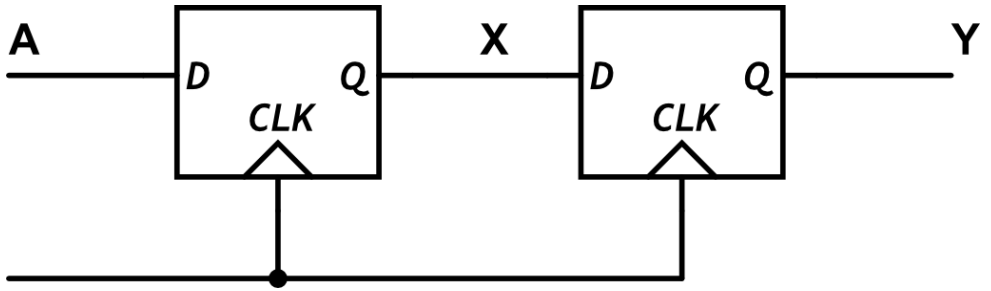
- **Suppose A = 1, B = 1, C = 0 (initially) and A changes to 0**
  - **What is value for X and Y if we use non-blocking assignments?**
- **Simulation will have different result than synthesis**

Assign Primitive

- For the most part all circuit behavior is defined by code inside `always` procedure
- Exception: Assign primitive can be used to describe simple combinational logic **outside of `always` procedure**
- `logic x; assign x = a & b;`
- Useful for simple modules or debugging (for example, fix a given signal to logic '0' to test

Sequential `Always` Procedure

- **Sequential Circuit Example**
  - **Chained flip-flops**
  - **Shift Register or digital filter**



- **Correct way to write (with non-blocking assignment)**

```
always_ff @ (posedge clk) begin
   X <= A;
   Y <= X;
end
```

- **What happens (in simulation) if we try to use blocking assignment?**