

---

---

# Section 8

— LED Board and Problem  
Decomposition —

---

---

# Administrivia

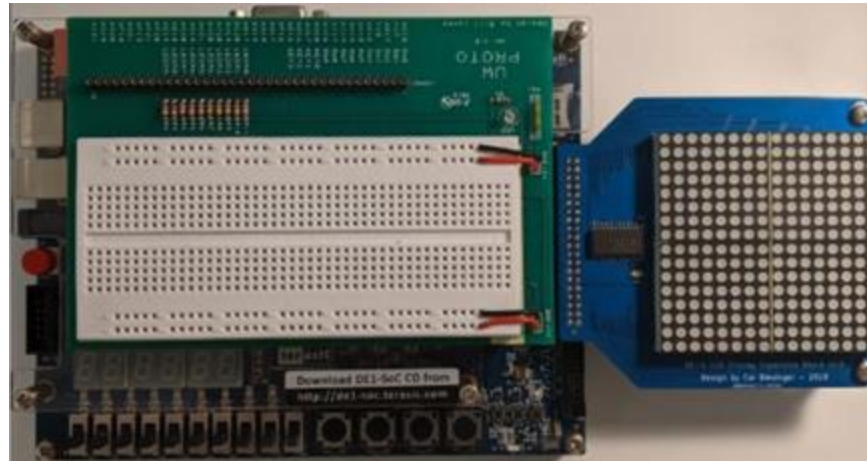
- **Lab 8:** It's final project time!
  - Proposal due next week – submit PDF to Gradescope ahead of time, then talk through with TA during demo slots on (03/05) and (03/06)
  - User's Manual (*i.e.*, Report) due last Friday of class (03/14).
  - Demo done during demo slot the last week of class or scheduled separately with TAs during Finals Week.
- **Quiz 3:** In two Tuesdays (03/11) *in place of* Lecture
  - Less formulaic: routing elements, computational building blocks
  - Study from past quizzes on course website!
- **This is the last section!** No section next week.



# The LED Board

# The LED Board Intro and Demo

- Your lab kit comes with a 16 x 16 Bicolor LED Expansion Board for Lab 8.
  - Actually tri-color: **red**, **green**, **orange** (red + green).
  - Find tutorial PDF and starter code (including test program) from Lab 8 spec.
- Attached to DE1-SoC via GPIO1 pin header/connector:



# Pixel Arrays

- LED Board is controlled by two 16 x 16 arrays called `RedPixels` and `GrnPixels`.
  - Defined as: `logic [15:0][15:0] RedPixels, GrnPixels;`
  - These have two *packed* dimensions (left of variable name) and 0 *unpacked* dimensions (right of variable name).
- Can be used on both the right and left side of assignments:
  - Access *row i* via `RedPixels[i]`.
  - Access *pixel* at row *i* and column *j* via `GrnPixels[i][j]`.

# LED Board ModelSim Tips

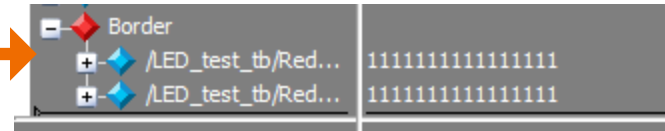
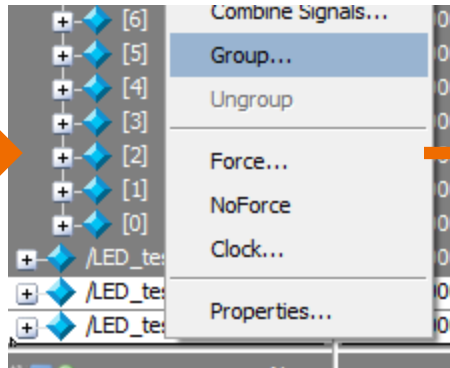
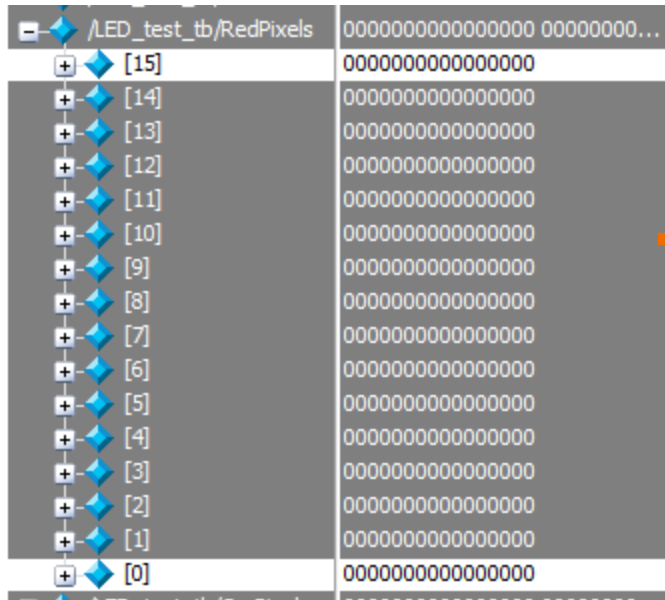
- Analyzing a 16x16 buffer can be difficult in ModelSim...

```
0000000000000000 0000000000000000 000... 1111111111111111 1100000000000011 101...
0000000000000000 } 1111111111111111
0000000000000000 } 1100000000000011
0000000000000000 } 1011111111111101
0000000000000000 } 1011000000001101
0000000000000000 } 1010111111110101
0000000000000000 } 1010110000110101
0000000000000000 } 1010101111010101
0000000000000000 } 1010101101010101
0000000000000000 } 1010101011010101
0000000000000000 } 1010101111010101
0000000000000000 } 1010110000110101
0000000000000000 } 1010111111110101
0000000000000000 } 1011000000001101
0000000000000000 } 1011111111111101
0000000000000000 } 1100000000000011
0000000000000000 } 1111111111111111
0000000000000000 0000000000000000 000... 1000000000000001 0111111111111110 011...
```

- Here are some tips that could help!

# LED Board ModelSim Tips

- Divide your buffer into smaller relevant groupings:
  - Let's say we view Columns 0 and 15 as some type of border.



# LED Board ModelSim Tips

- Create custom radices:
  - It can be tiresome to interpret the lengthy buffer values; one alternative is to create a custom radix.
  - Let's say we that in our design all 0's are classified as OFF and all 1's are classified as ON. **In your runlab.do file, before the run command, add:**

```
radix define States {  
    16'b0000010000000000 "OFF",  
    16'b1111111111111111 "ON" ,  
    -default hex  
    -defaultcolor white  
}
```



# LED Board ModelSim Tips

- Create custom radices:
  - It can be tiresome to interpret the lengthy buffer values; one alternative is to create a custom radix.
  - Let's say we that in our design all 0's are classified as OFF and all 1's are classified as ON. **Then in ModelSim:**



Border		( Border )
+ ◆	/LED_test_tb/R...	OFF
+ ◆	/LED_test_tb/R...	OFF

OFF	} ON
OFF	} ON

# Digital System Design Process

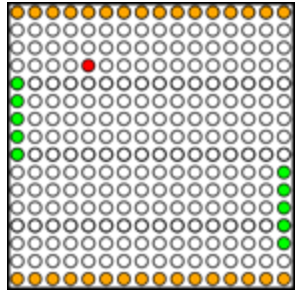
# HDL Organization

- A module is not a *function*, it is closest to a **class**.
  - Something that you **instantiate**, not something that you *call* – hardware cannot appear and disappear spontaneously.
- Treat modules as **resource managers** rather than temporary helpers.
  - Decompose problem into the major resources and computations and build separate modules around those.
- Hardware organization tends to be more **horizontal** (*i.e.*, modules computing things in parallel alongside each other) rather than **vertical** (*i.e.*, a call stack with functions waiting on each other).

# Choosing Blocks

- **Blocks:** List out major components (decompose into reusable components where applicable).
  - The managed resource for a module, if applicable, will be an internally-defined signal.
  - Determine whether the component needs state (sequential logic/FSM) or not (combinational logic).
- **Ports:** Figure out necessary connections (information passing) between modules – make as general as possible.

# Exercise 1

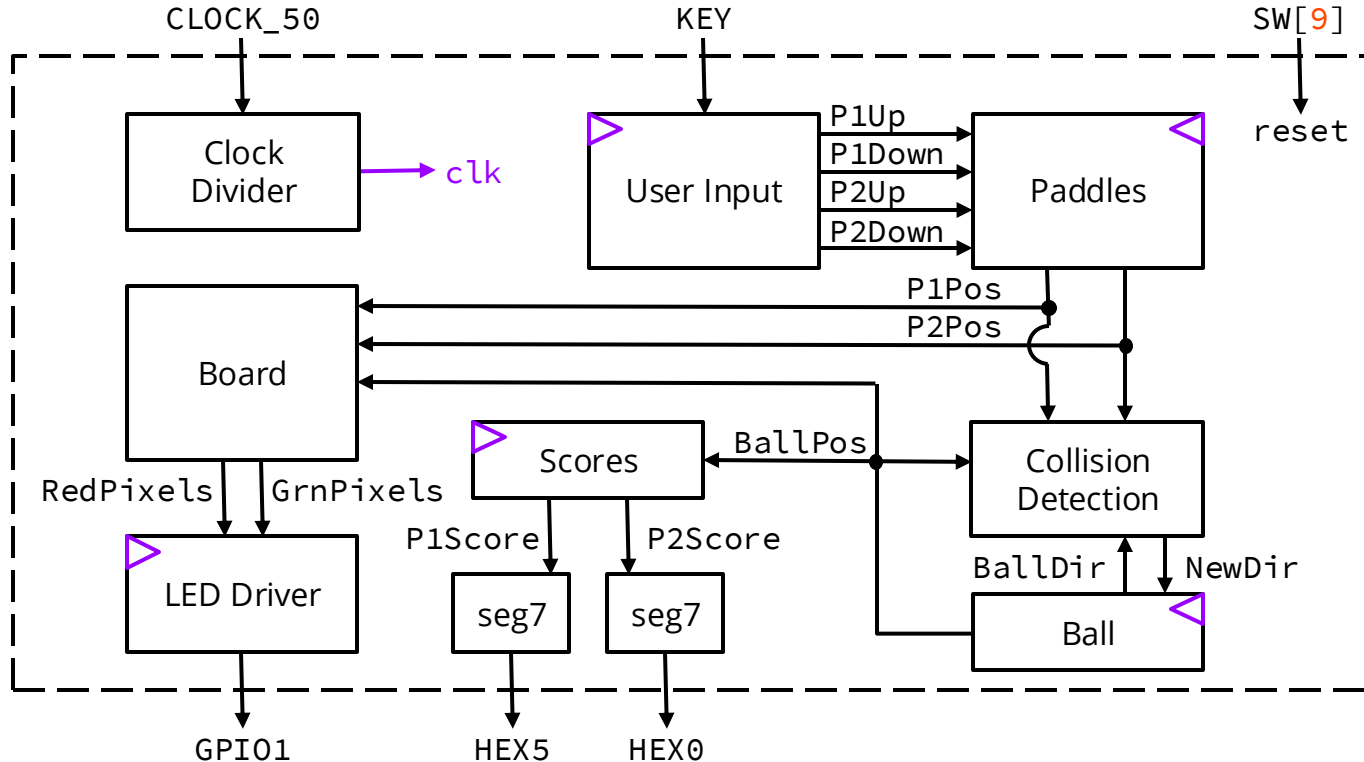


- We'd like to implement the game of [Pong](#) using the LED Board.
  - Player paddles (size 5) should be **green**, ball (size 1) should be **red**, top and bottom row (size 16) should be **orange**.
  - Ball only moves diagonally, bounces against top and bottom rows and paddles, and starts from (7,7) in a random diagonal direction (4 options).
  - A point is scored if the ball reaches the leftmost or rightmost column.
  - The paddles move faster than the ball.
  - Player 1 (left) controlled by **KEY[3]** (up) and **KEY[2]** (down).
  - Player 2 (right) controlled by **KEY[1]** (up) and **KEY[0]** (down).
  - **SW[9]** is reset, player scores shown on **HEX0** (Player 1) and **HEX5** (Player 2).
- Brainstorm the major components/resources and create a block diagram.
  - You will need to include `LEDDriver` (provided), `seg7`, and `clock_divider`.

# Exercise 1 (Sample Solution)

- Major components (sequential logic, combinational logic):
  - LED Driver – provided code to interface with GPIO1 pins
  - Clock Divider – `divided_clocks` to slow down system
  - Input module – synchronizers, pulse generators
  - Paddle module(s) – paddle positions
  - Ball module – ball position, ball direction, LFSR for starting direction
  - Score module(s) – P1 score, P2 score, win detection
  - Collision detector – uses paddles and ball to trigger ball direction change
  - Board module – combines paddles and ball into `RedPixels`, `GrnPixels`
  - 7-segment display driver (x2) – display scores on HEX

# Exercise 1 (Sample Solution Block Diagram)



**Note:** This diagram assumes that a point/score can be detected with just `BallPos`, which will trigger a score update (`Scores`) and a ball reset (`Ball`).

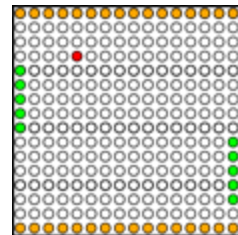
**Note:** It took the staff 5 revisions to reach this diagram – these take time to get right!

# Module Implementation and Testing (CL)

- Create module:
  - Module skeleton (`end/module`, name, port list).
  - Declare intermediate signals.
  - Implement logic (`assign`, `always_comb`).
- Create test bench:
  - Start with the module skeleton (`end/module`).
  - Create signals for all ports of the module you're going to test.
  - Instantiate device under test (`dut` as instance name).
  - Define test vectors in an `initial` block.
    - Run through all possible input combinations in simulation to thoroughly test.
    - Add arbitrary time delays `#<num>;` (e.g., `#10;`) in-between input changes.

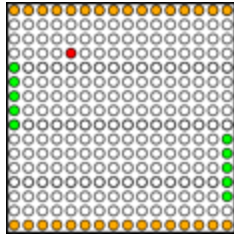


# Exercise 2



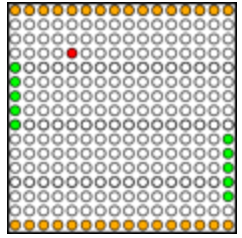
- Brainstorm how you would implement (on computer or pseudocode) the **collision\_detection** module of our game.
  - Inputs:
    - `logic [3:0] P1Pos, P2Pos;` // paddle positions (offset from top at Row 1: 0-10)
    - `logic [3:0][1:0] BallPos;` // 2-D position: (x, y) with both coordinates 0-15
    - `logic [1:0] BallDir;` // 0 = NE, 1 = NW, 2 = SE, 3 = SW
  - Output:
    - `logic [1:0] NewDir;` // matches BallDir if no collision, else new direction

# Exercise 2



- Brainstorm how you would implement (on computer or pseudocode) the **collision\_detection** module of our game.
  - Inputs:
    - `logic [3:0] P1Pos, P2Pos;` // paddle positions (offset from top at Row 1: 0-10)
    - `logic [3:0][1:0] BallPos;` // 2-D position: (x, y) with both coordinates 0-15
    - `logic [1:0] BallDir;` // 0 = NE, 1 = NW, 2 = SE, 3 = SW
  - Output:
    - `logic [1:0] NewDir;` // matches BallDir if no collision, else new direction
- Some hints:
  - `always_comb` blocks can be written a bit like software since the blocking assignments will override the effects of previous assignments.

## Exercise 2 (Code Review and Outline)



- Brainstorm how you would implement (on computer or pseudocode) the **collision\_detection** module of our game.
- Some general ideas:
  - By default, `NewDir` should be the same as `BallDir`. The bits of `BallDir` can be interpreted as `BallDir[1]` is north/south and `BallDir[0]` is west/east.
  - **Paddle collisions** can only occur if (1) `BallPos[1]/x` is `1` and the ball is traveling *west* or (2) `BallPos[1]/x` is `14` and the ball is traveling *east*.
    - Future ball position must be checked against y-positions `P#Pos` to `P#Pos+4`.
  - **Wall collisions** can only occur if (1) `BallPos[0]/y` is `1` and the ball is traveling *north* or (2) `BallPos[0]/y` is `14` and the ball is traveling *south*.

# Module Implementation and Testing (FSM)

- Module Internals:
  - State Encodings and Variables – `enum`
  - Next State Logic (ns) – `always_comb` and `case`
  - Output Logic – `assign/always_comb`
  - State Update Logic (ps) and Reset – `always_ff`
- Create test bench:
  - Same as CL: (1) module skeleton, (2) create signals for `dut` ports, (3) instantiate device under test.
  - Generate your simulated clock.
  - Define test vectors in an `initial` block.
    - Add edge-sensitive time delays `@(posedge clk);` in-between input changes.
    - To thoroughly test your FSM, need to take every transition that we care about.

# Module Implementation and Testing (Other SL)

- **Not all sequential logic will be FSMs!**
- Module Internals:
  - State Update Logic and Reset – `always_ff` (depends on desired behavior)
    - Register: `Q <= D;`
    - Up-counter: `count <= count + 1;`
    - Shift register: `state <= {state[N-2:0], new_bit};`
- Create test bench:
  - Generate your simulated clock.
  - Define test vectors in an `initial` block.
    - Add edge-sensitive time delays `@(posedge clk);` in-between input changes.
    - **Try to thoroughly test all relevant/important behaviors.**

# Exercise 3

- Brainstorm how you would implement (on computer or pseudocode) the **ball** module of our game.
  - Inputs:
    - `logic` clk, reset;
    - `logic [1:0] NewDir;` *// new direction of travel, accounting for collisions*
  - Output:
    - `logic [3:0][1:0] BallPos;` *// 2-D position: (x, y) with both coordinates 0-15*
    - `logic [1:0] BallDir;` *// 0 = NE, 1 = NW, 2 = SE, 3 = SW*

# Exercise 3

- Brainstorm how you would implement (on computer or pseudocode) the **ball** module of our game.
  - Inputs:
    - `logic` clk, reset;
    - `logic [1:0] NewDir;` *// new direction of travel, accounting for collisions*
  - Output:
    - `logic [3:0][1:0] BallPos;` *// 2-D position: (x, y) with both coordinates 0-15*
    - `logic [1:0] BallDir;` *// 0 = NE, 1 = NW, 2 = SE, 3 = SW*
- Some hints:
  - Should we update `BallPos` based on `BallDir` or `NewDir`?
  - Randomized initial direction – can use any 2 bits of LFSR output.

# Exercise 3 (Code Review and Outline)

- Brainstorm how you would implement (on computer or pseudocode) the **ball** module of our game.
- Some general ideas:
  - Use **NewDir** to update **BallPos** to account for collisions. Use **NewDir[1]** to increment/decrement *y-position* and **NewDir[0]** to increment/decrement *x-position*.
  - Use up-counter output with comparator (against constant, *e.g.*, 3) as the **Enable** signal to **BallPos** and **BallDir** registers to slow down the speed.
  - **Reset behavior** should set **BallPos** to any of (7,7), (7, 8), (8,7), or (8,8) – all valid choices – and then use LFSR output to set **BallDir**.

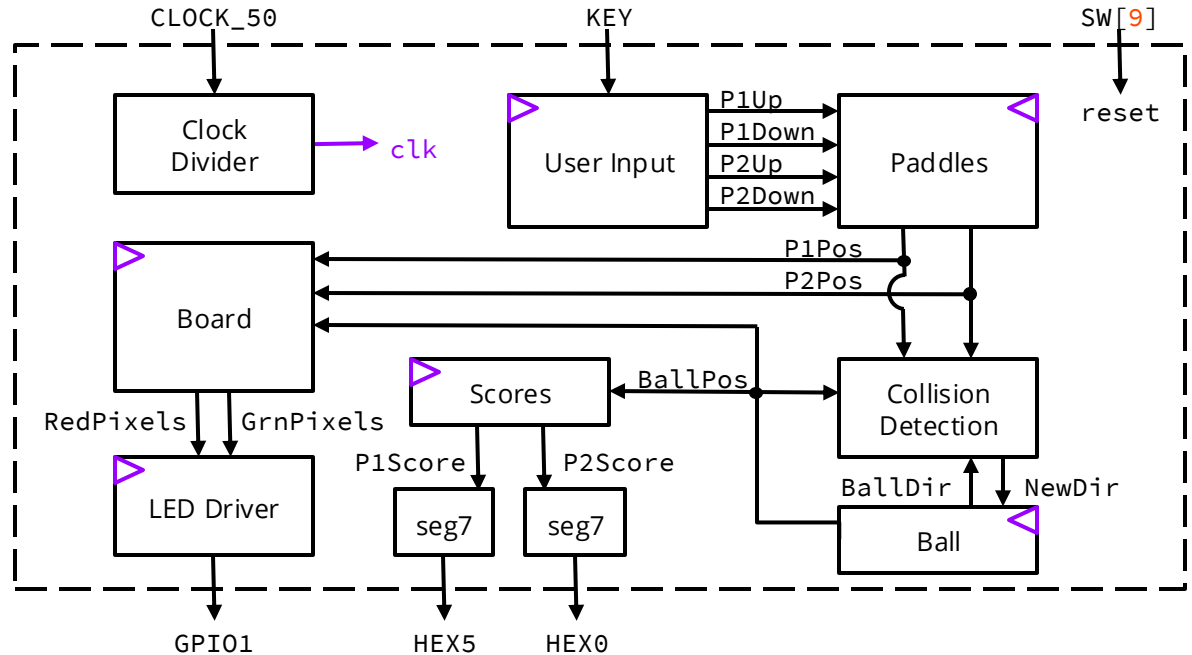


# Module Implementation and Testing (Top-Level)

- Create module:
  - Start with the normal module outline.
  - Generate/declare internal signals (*e.g.*, port connections for modules).
  - Instantiate internal modules and make port connections.
  - Can include some logic but generally want to keep this to a minimum.
- Create test bench:
  - Structured like a normal test bench.
  - Test vectors should focus on testing module interconnections:
    - Test that the reset affects all of the internal modules that it should.
    - Don't thoroughly test internal modules again (rely on individual test benches).
    - Test connections between modules – data passing timing and reactions.

# Exercise 4

- Brainstorm how you would implement (on computer or pseudocode) the top-level **pong** module of our game.



# Exercise 4 (Code Review and Outline)

- Brainstorm how you would implement (on computer or pseudocode) the top-level **pong** module of our game.
- Some general ideas:
  - **Port list** should match the external inputs and outputs from the block diagram.
  - Declare any signals shown between modules as **internal signals**.
  - **Instantiate all of the blocks** shown in the block diagram.
    - Can likely use (.\*) automatic port connections!