Section 6

User Input and Top-Level Modules

Administrivia

- **Lab 6:** Report due next Wednesday (11/13) @ 2:30 pm, demo by last OH on Friday (11/15), but expected during your assigned slot.
  - ⚠️ This lab is a LOT harder than Lab 5 ⚠️
- **Lab 7:** Report due 11/20, demo by last OH on 11/22.
  - Building on top of Lab 6 using building blocks; comparable in difficulty

User Input

User Input Issues

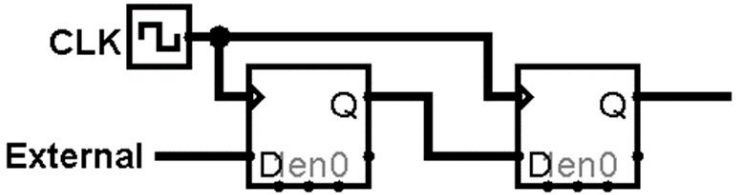
- A major advantage of digital systems is the computational speed (*e.g.*, 50 MHz clock means a computation every 20 ns!)
- This can be difficult to reconcile with user inputs because humans can't move that quickly.
  - We can't control when in a clock cycle our input changes. How do we avoid metastability?
  - What if we only wanted to change an input for a clock cycle?

User Input Issues

- A major advantage of digital systems is the computational speed (*e.g.*, 50 MHz clock means a computation every 20 ns!)
- This can be difficult to reconcile with user inputs because humans can't move that quickly.
  - We can't control when in a clock cycle our input changes. How do we avoid metastability?
  - What if we only wanted to change an input for a clock cycle?
- Solutions:
  - Add a **synchronizer** (typically for all external inputs except `reset`).
  - Add a edge detector / **pulse generator** (typically for `KEY`s).

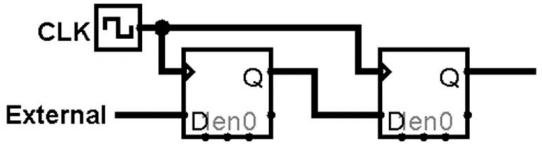
Exercise 1a (Synchronizer)

- We will use a **2 flip-flop synchronizer** to combat metastability:



  - Implement this in a module called **synch**.

Exercise 1a (Solution)



- A **2 flip-flop synchronizer** to combat metastability:

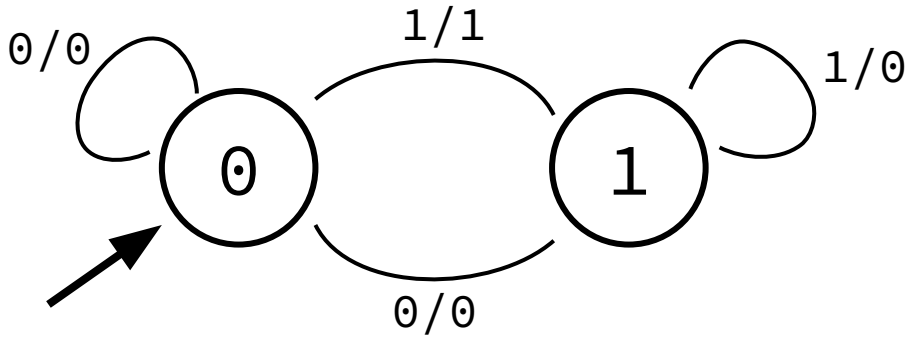
```
module synch (input logic clk, reset, in, output logic out);

  logic mid;  // output of first FF, could be named anything

  always_ff @(posedge clk)
    if (reset)
      {mid, out} <= 2'b00;
    else
      {mid, out} <= {in, mid};

endmodule  // synch
```

Exercise 1b (Pulse Generator)

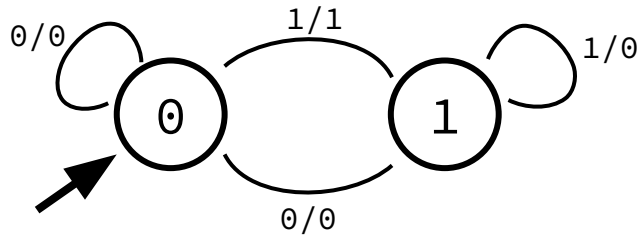
- We will use a *finite state machine* to **generate pulses** for rising edges of an input signal (*i.e.*, outputs 1 for one clock cycle each time the input goes from low to high, no matter how long the input is held high).
  - Create an FSM state diagram.
  - Then create a SystemVerilog implementation in a module called **`pulse`**.

Exercise 1b (Solution)

- **Generate pulses** for rising edges of an input signal:



Exercise 1b (Solution)



- **Generate pulses** for rising edges of an input signal:

```
module pulse (input logic clk, reset, in, output logic out);

  enum logic {ZERO, ONE} ps, ns;

  assign ns = in ? ONE : ZERO;

  always_ff @(posedge clk)
    ps <= reset ? ZERO : ns;

  assign out = (ps == ZERO) & in;

endmodule  // pulse
```

Top-Level Module Block Diagrams

Block Diagrams (Review)

- **Block diagrams** are the basic design tool for digital logic.
  - The diagram itself is a module → inputs and outputs shown and connected.
  - Major components are represented by blocks (“black boxes”) with their internals abstracted away → each block becomes its own module.
  - All ports for each block should be shown and labeled and connected to the appropriate part(s) of the rest of the system → sets your port connections.
  - Wires and gates can be added/shown as needed.
- From Wikipedia: The goal is to “[end] in block diagrams detailed enough that each individual block can be easily implemented.”
  - For designs that involve multiple modules, should always create your block diagram *before* coding anything!

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).

Top-Level Modules

Top-Level Module Notes

- The top-level module interfaces with the actual device ports (*e.g.*, `CLOCK_50`, `SW`, `KEY`, `LEDR`, `HEX`).
  - We recommend putting as *little* logic (some gates and routing elements okay) in the top-level module to increase the portability of your code across different devices (*i.e.*, moving from DE1-SoC to DE0-Nano should only be reconnecting different I/O ports to modules).
  - For readability, can also “rename” signals using `assign` statements (*e.g.*, `assign clk = CLOCK_50;`, `assign reset = ~KEY[3];`) or use internal signal names in Wave pane in ModelSim.

Top-Level Module Implementation

1) Start with the normal module outline.
2) Generate/declare internal signals (*e.g.*, port connections for modules).
    a) Use copy-and-paste from internal module definitions to avoid typos!
3) Instantiate internal modules and make port connections.
4) Can include some logic but generally want to keep this to a minimum.
    a) Significant logic should be abstracted into internal modules.

Top-Level Module Test Bench

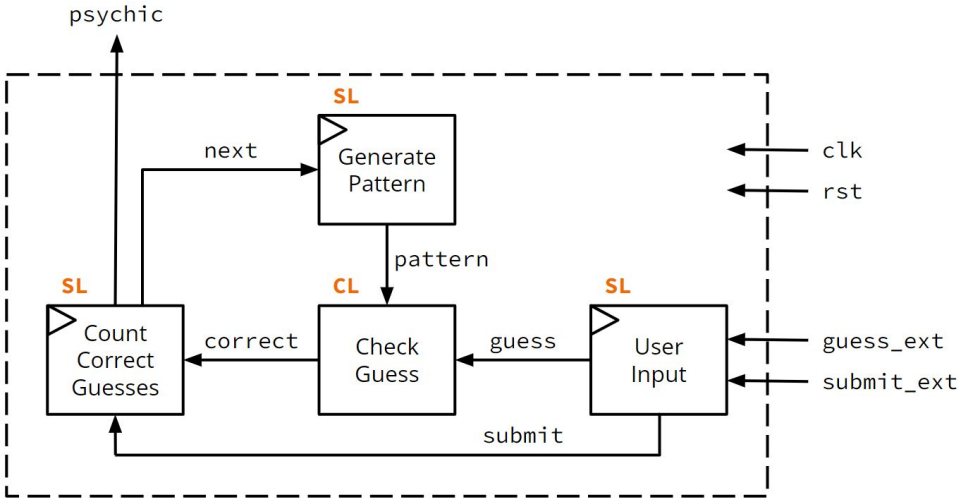
- Structurally like a normal test bench:
  - Module skeleton.
  - Create signals for module you're going to test.
  - Instantiate device under test
  - Generate clock (if needed)
  - Define test vectors in an `initial` block.

Top-Level Module Test Bench

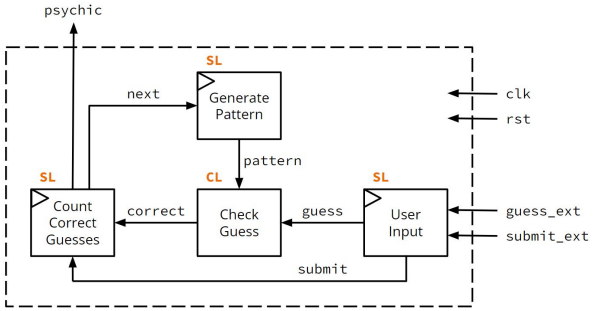
- Structurally like a normal test bench:
  - Module skeleton.
  - Create signals for module you're going to test.
  - Instantiate device under test
  - Generate clock (if needed)
  - Define test vectors in an `initial` block.
- Test vectors should focus on testing module *interconnections*:
  - Test 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 2

- In Lecture 6, we drew out the block diagram for a **psychic tester**, where the user needs to correctly guess 8 consecutive 4-bit patterns to be declared a psychic. Here, we'll look at a slightly modified version of it:
  - The User Input module will use `synch` and `pulse`, making it sequential logic (SL).
  - To abstract away hardware, changed `KEY` and `SW` inputs to `guess_ext`, `submit_ext`.



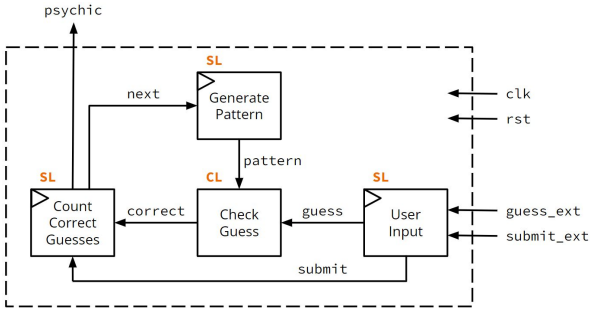
Exercise 2a



- **Implement `psychic_tester`**:
  - Provided modules:

```
module genPatt (clk, rst, pattern, next);
module userIn (clk, rst, guess, submit, guess_ext, submit_ext);
module checkGuess (pattern, guess, correct);
module countRight (clk, rst, correct, submit, next, psychic);
```

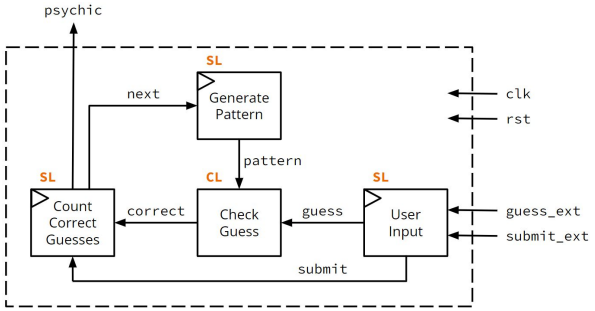
Exercise 2a (Solution)



- Module outline:

```
module psychic_tester (clk, rst, guess_ext, submit_ext, psychic);
  input  logic       clk, rst, submit_ext;
  input  logic [3:0] guess_ext;
  output logic       psychic;









endmodule  // psychic_tester
```

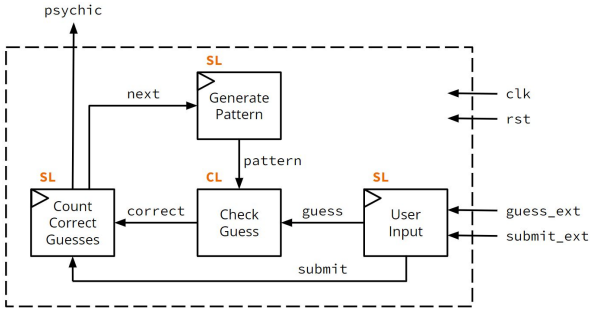
Exercise 2a (Solution)



- Generate/declare internal signals:

```
module psychic_tester (clk, rst, guess_ext, submit_ext, psychic);
  input  logic       clk, rst, submit_ext;
  input  logic [3:0] guess_ext;
  output logic       psychic;

  logic [3:0] pattern, guess;
  logic correct, next, submit;









endmodule  // psychic_tester
```

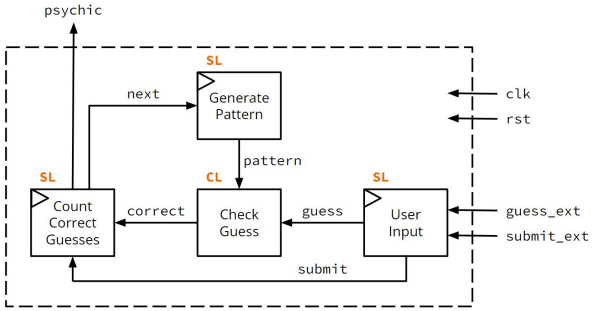
Exercise 2a (Solution)



- Instantiate internal modules:

```
module psychic_tester (clk, rst, guess_ext, submit_ext, psychic);
  input  logic       clk, rst, submit_ext;
  input  logic [3:0] guess_ext;
  output logic       psychic;

  logic [3:0] pattern, guess;
  logic correct, next, submit;

  genPatt    pat (.clk, .rst, .pattern, .next);
  userIn     inp (.clk, .rst,
                  .guess_ext, .submit_ext, .guess, .submit);
  checkGuess chk (.pattern, .guess, .correct);
  countRight cnt (.clk, .rst, .correct, .submit, .next, .psychic);
endmodule  // psychic_tester
```

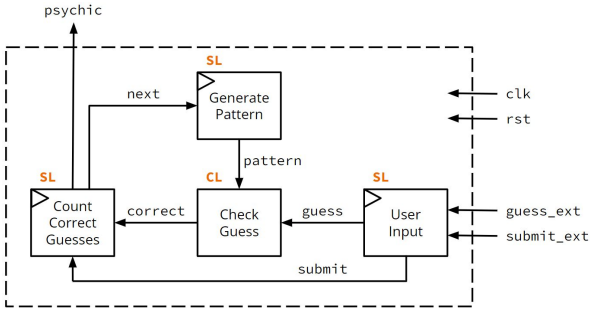
Exercise 2a (Solution)



- There's no logic other than the port connections!

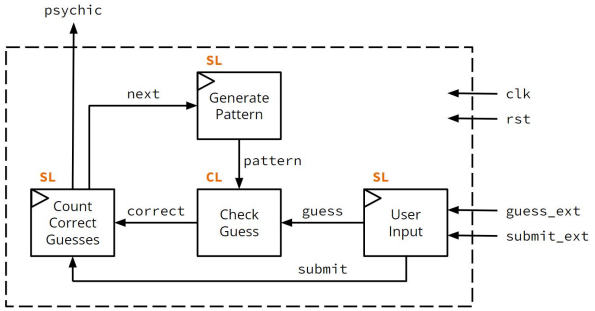
```
module psychic_tester (clk, rst, guess_ext, submit_ext, psychic);
  input  logic       clk, rst, submit_ext;
  input  logic [3:0] guess_ext;
  output logic       psychic;

  logic [3:0] pattern, guess;
  logic correct, next, submit;

  genPatt    pat (.clk, .rst, .pattern, .next);
  userIn     inp (.clk, .rst,
                  .guess_ext, .submit_ext, .guess, .submit);
  checkGuess chk (.pattern, .guess, .correct);
  countRight cnt (.clk, .rst, .correct, .submit, .next, .psychic);
endmodule  // psychic_tester
```

Exercise 2b



- **Create a test bench** for **`psychic_tester`**:
  - The “randomly” generated patterns will be:
    `4'b0000` → `4'b0001` → `4'b0011` → `4'b0111` →
    `4'b1110` → `4'b1101` → `4'b1011` → `4'b0110`.
  - Remember that it will take 2 clock cycles for the `*_ext` signals to go through the `synch` modules in the User Input module to reach the rest of the system.
  - **Group brainstorm:** What behaviors/situations do we want to test?

Exercise 2b



- **Create a test bench** for **`psychic_tester`**:
  - **Group brainstorm:** What behaviors/situations do we want to test?
    - Reset behavior.
    - `submit` should trigger a single update 2-3 clock cycles later, even when held.
    - A correct guess should increment count and generate a new pattern.
    - An incorrect guess should reset count and generate a new pattern.
    - `psychic` going high after 8 guesses was likely tested in `countRight`'s test bench but could test again.

Exercise 2b (Solution)

- Module outline, signal declarations, device under test instantiation:

```
module psychic_tester_tb ();
  logic       clk, rst, submit_ext, psychic;
  logic [3:0] guess_ext;

  psychic_tester dut (.*);









endmodule  // psychic_tester_tb
```

Exercise 2b (Solution)

- Clock generation, skeleton for test vectors:

```
module psychic_tester_tb ();
  ...  // signal declarations
  ...  // dut instantiation

  parameter T = 20;  // clock period
  initial
    clk = 0;
  always
    #(T/2)  clk <= ~clk;

  initial begin
    $stop;
  end
endmodule  // psychic_tester_tb
```

Exercise 2b (Solution)

- Test reset behavior.

```
module psychic_tester_tb ();
  ...  // signal declarations, dut instantiation, clock generation

  initial begin
    // test case 0: verify reset behavior
    {rst, submit_ext, guess_ext} <= 6'b1_0_0000;  @(posedge clk);
    {rst, submit_ext, guess_ext} <= 6'b0_0_0000;  @(posedge clk);
    $stop;
  end
endmodule  // psychic_tester_tb
```

Exercise 2b (Solution)

- Test submit signal unenabled.

```
module psychic_tester_tb ();
  ...  // signal declarations, dut instantiation, clock generation

  initial begin
  ...  // after reset
    // test case 1: user inputs guesses but did not trigger submit button
    // guess is not counted even if the guess matches the correct pattern
    {submit_ext, guess_ext} <= 5'b0_0000;     @(posedge clk);
    {submit_ext, guess_ext} <= 5'b0_0001;     @(posedge clk);
                                              @(posedge clk);

    $stop;
  end
endmodule  // psychic_tester_tb
```

Exercise 2b (Solution)

- Test 8 correct guesses with a wrong guess in between

```
module psychic_tester_tb ();
  ...  // signal declarations, dut instantiation, clock generation
  initial begin
    ...  // after reset
    {submit_ext, guess_ext} <= 5'b0_0000;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (1)
    {submit_ext, guess_ext} <= 5'b0_0001;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (2)
    {submit_ext, guess_ext} <= 5'b0_0011;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (3)
    {submit_ext, guess_ext} <= 5'b0_0111;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (4)
    {submit_ext, guess_ext} <= 5'b0_1111;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // wrong guess
                                            @(posedge clk);

    $stop;
  end
endmodule  // psychic_tester_tb
```

Exercise 2b (Solution)

- Test psychic going high after 8 correct guesses

```
module psychic_tester_tb ();
  ...  // signal declarations, dut instantiation, clock generation
  initial begin
    ...  // after reset
    {submit_ext, guess_ext} <= 5'b0_0000;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (1)
    {submit_ext, guess_ext} <= 5'b0_0001;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (2)
    {submit_ext, guess_ext} <= 5'b0_0011;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (3)
    {submit_ext, guess_ext} <= 5'b0_0111;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (4)
    {submit_ext, guess_ext} <= 5'b0_1110;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (5)
    ...  // continue correct guess
    $stop;
  end
endmodule  // psychic_tester_tb
```

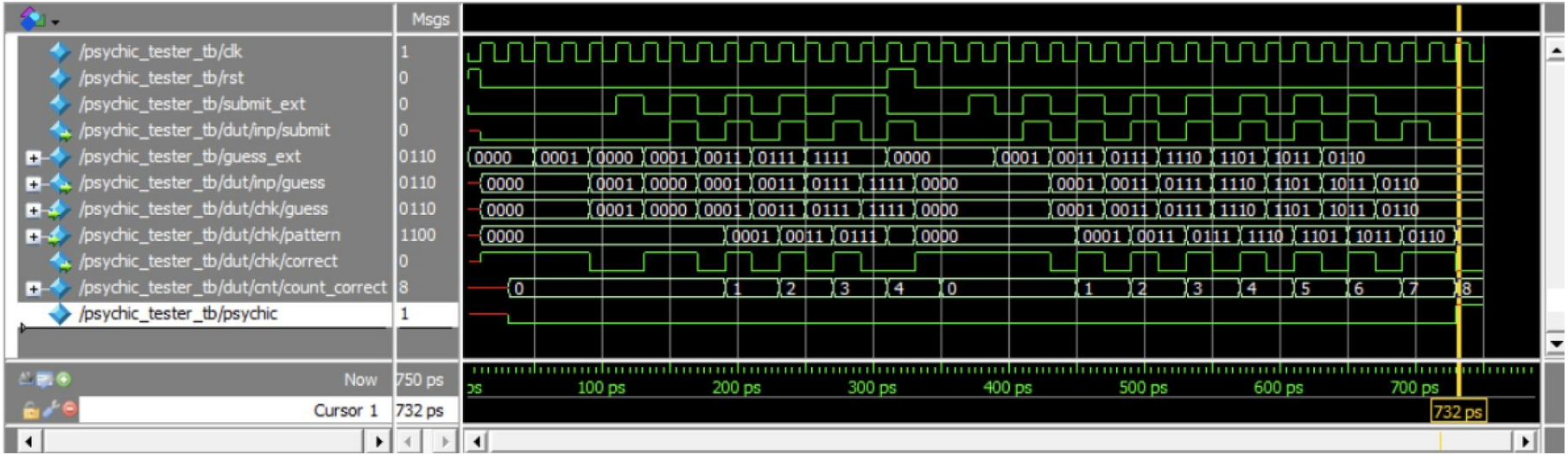
Exercise 2b (Solution)

- Test psychic going high after 8 correct guesses

```
module psychic_tester_tb ();
  ...  // signal declarations, dut instantiation, clock generation

  initial begin
    ...  // user already guessed correctly 5 times
    {submit_ext, guess_ext} <= 5'b0_1101;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (6)
    {submit_ext, guess_ext} <= 5'b0_1011;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (7)
    {submit_ext, guess_ext} <= 5'b0_0110;   @(posedge clk);
    submit_ext <= 1'b1;                     @(posedge clk); // correct (8)
    submit_ext <= 1'b0;                     @(posedge clk);
                                            @(posedge clk);
                                            @(posedge clk);
                                            @(posedge clk); // psychic is HIGH

    $stop;
  end
endmodule  // psychic_tester_tb
```

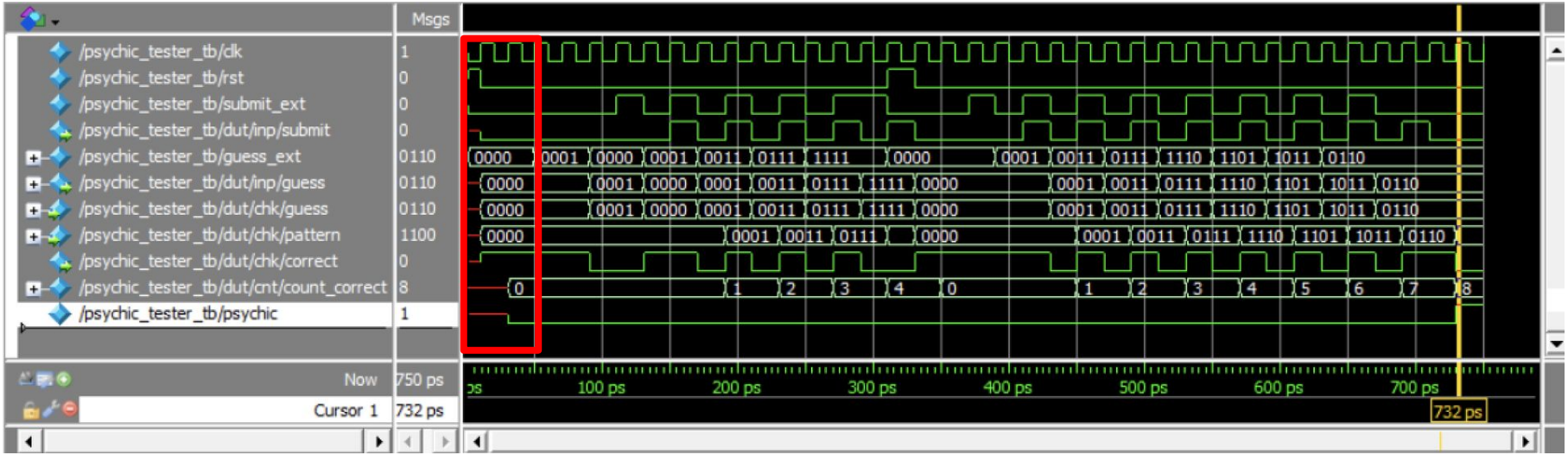
Exercise 2b (solution)

- Example of possible simulation



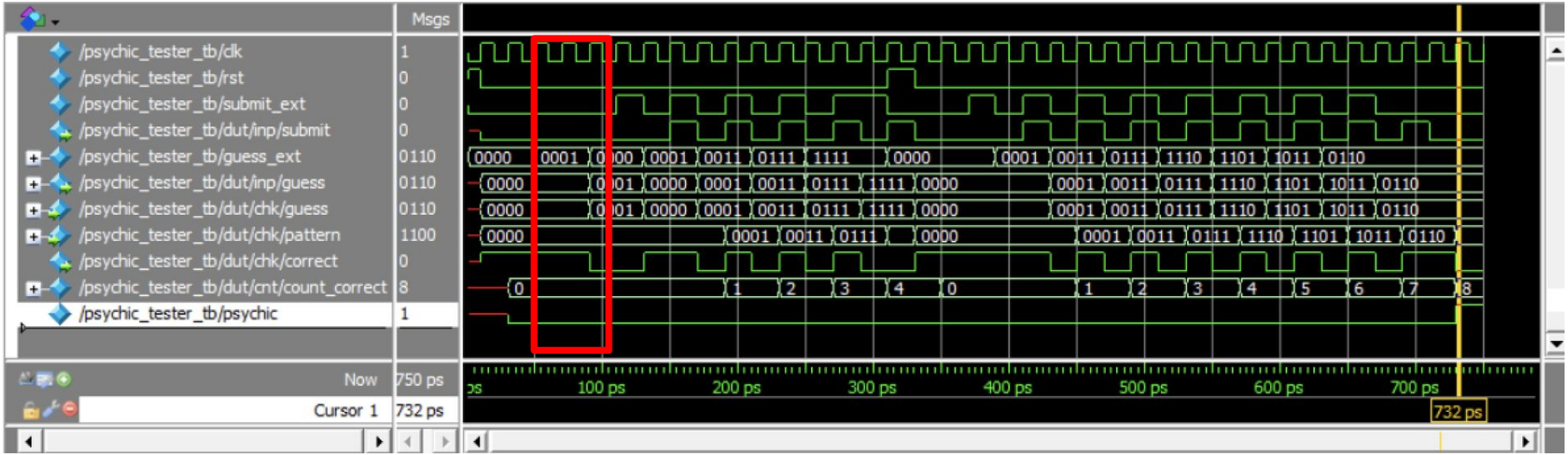
Exercise 2b (solution)

- Verify reset behavior



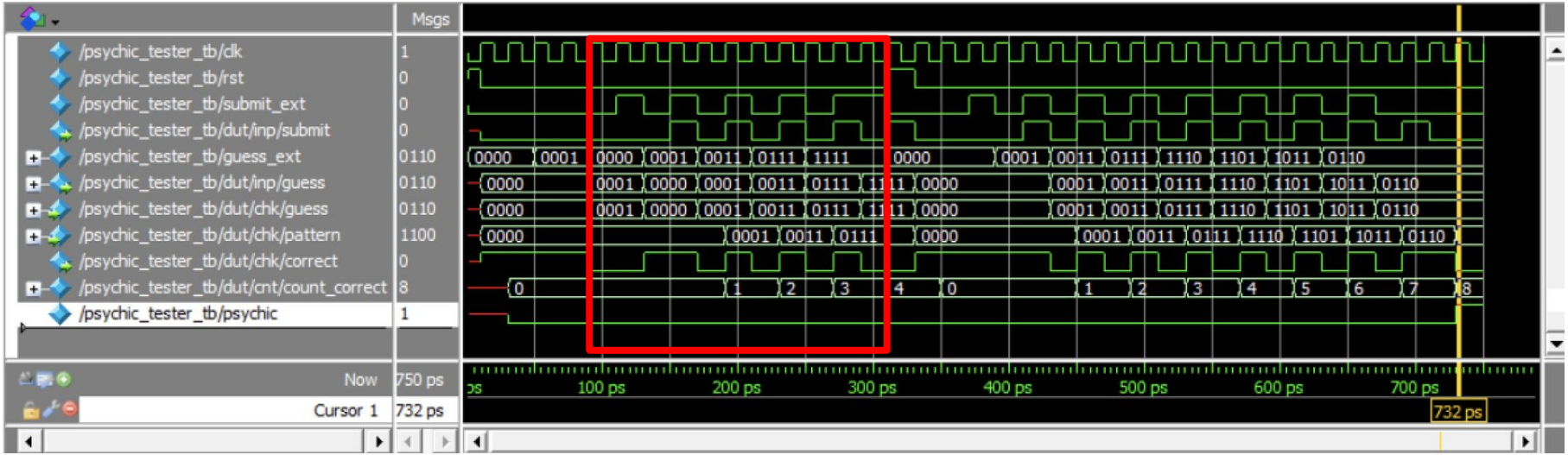
Exercise 2b (solution)

- User inputs guesses but did not trigger submit button
- guess doesn't submit even though the guess matches the correct pattern



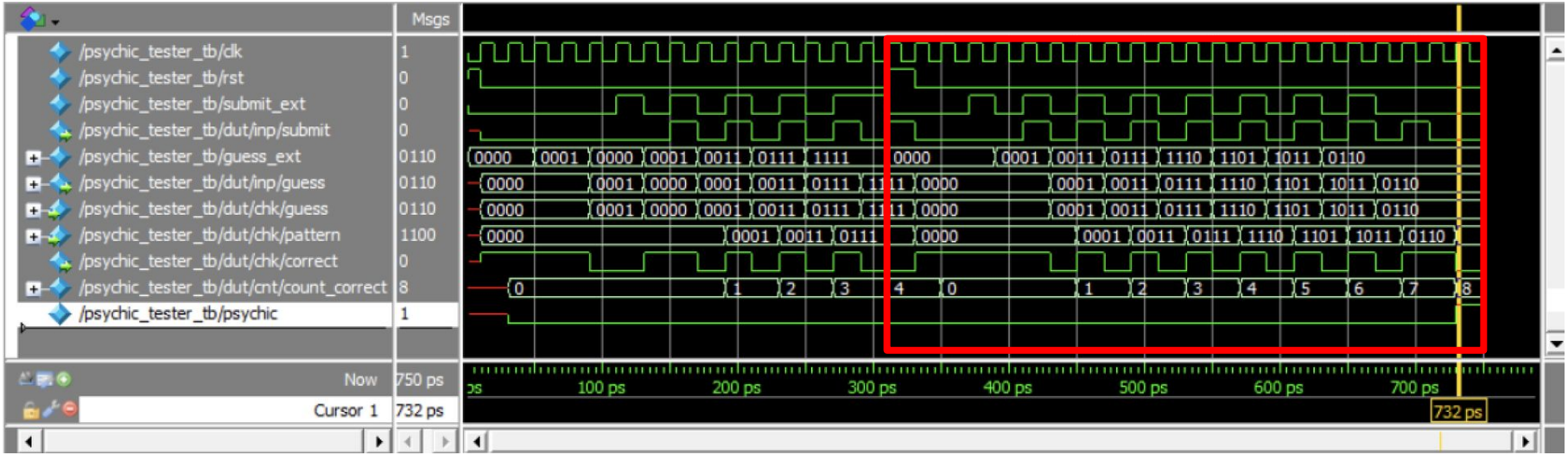
Exercise 2b (solution)

- User guesses wrong in their 5th try
- count_correct is cleared and psychic signal should is false



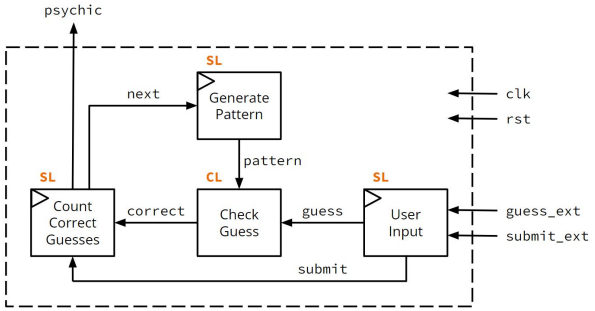
Exercise 2b (solution)

- Reset and user guesses correctly 8 times
- psychic signal is true after 8 guesses

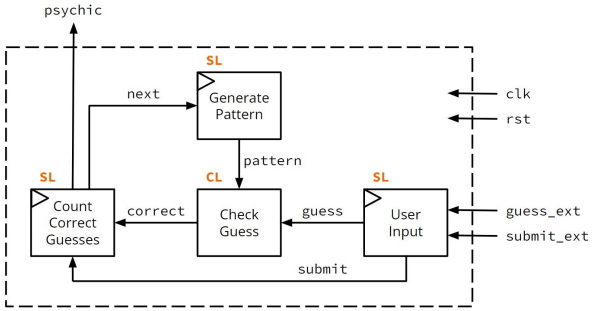


Exercise 2c

- **Implement `DE1_SoC` to connect to hardware:**
  - Use `SW[3:0]` as `guess`,
    `~KEY[3]` as `rst` (reset),
    `~KEY[0]` as `submit`, and
    `LEDR[0]` as `psychic`.

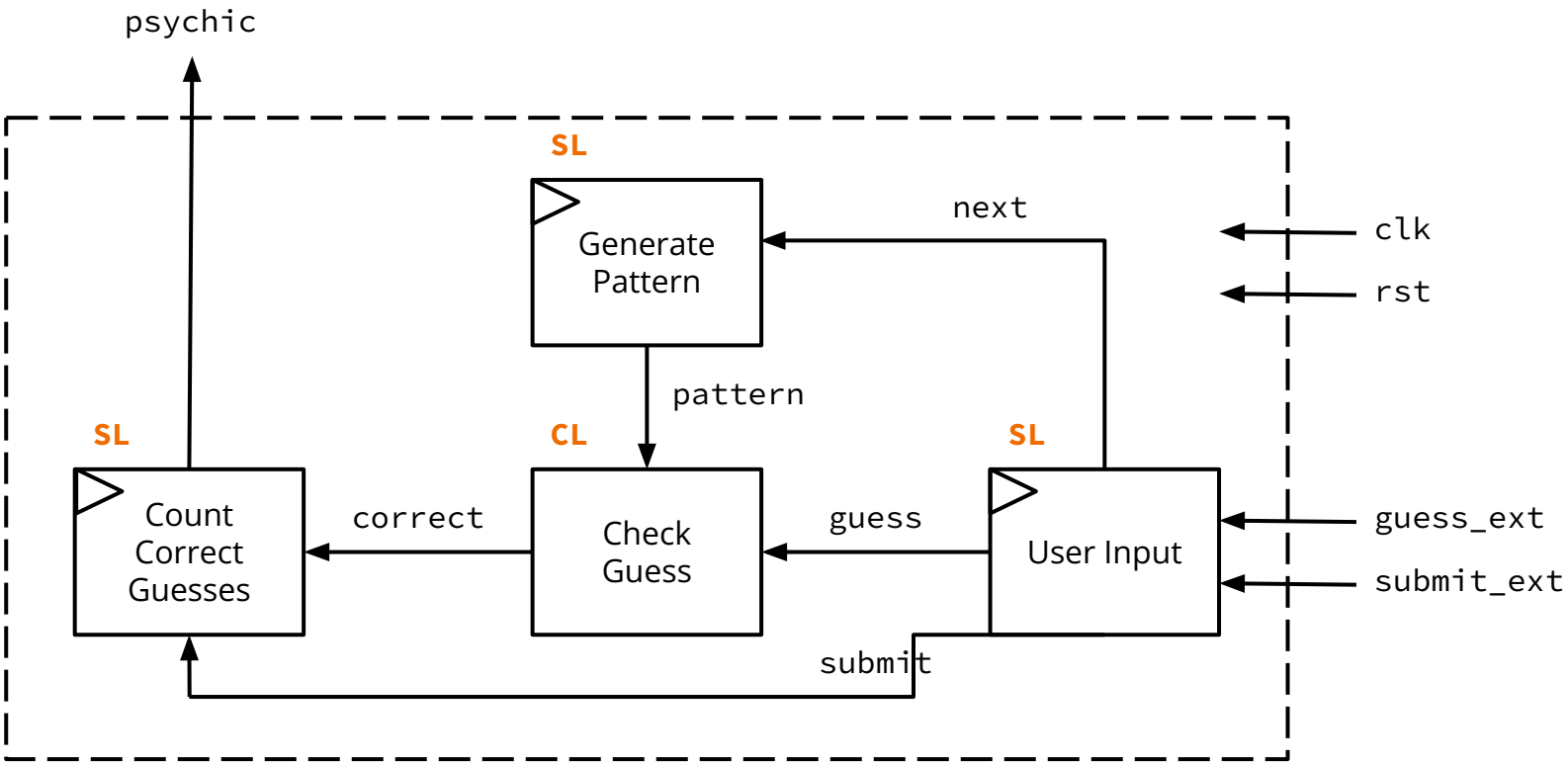


Exercise 2c (Solution)



- Wrapper module for hardware (Version 1):

```
module DE1_SoC (CLOCK_50, SW, KEY, LEDR);
  input  logic       CLOCK_50;
  input  logic [9:0] SW;
  input  logic [3:0] KEY;
  output logic [9:0] LEDR;

  logic clk, rst, guess_ext, submit_ext, psychic;
  assign clk = CLOCK_50;
  assign rst = ~KEY[3];
  assign psychic = LEDR[0];
  assign guess_ext = SW[3:0];
  assign submit_ext = ~KEY[0];

  psychic_tester psych (.*);
endmodule  // DE1_SoC
```


psychic
SL
Generate
Pattern
next
clk
rst
pattern
SL
CL
SL
Count
Correct
Guesses
correct
Check
Guess
guess
User Input
guess_ext
submit_ext
submit