
CSE 369 Section 2

— Modules and Gates —

Administrivia

- **Lab kit pickups:** If you haven't picked up a kit yet, please make an ED post ASAP and we will confirm which office hours you can come to
- **Lab 1&2:** Report due next Wednesday (10/16) @ 2:30 pm, demo by last OH on Friday (10/18), but expected during your assigned slot.



SystemVerilog Review

FPGA Engineer



Just send the
signal to both ports.

C++ Developer

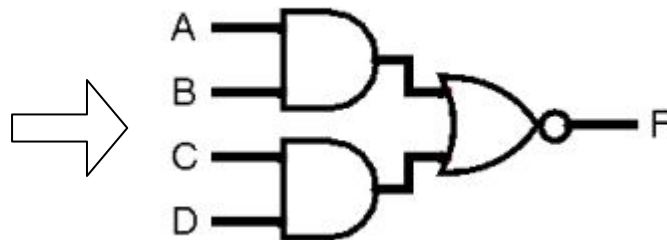


Oh no ! Two of my
threads tried to access
this data at the same time
and my output is random !

What is SystemVerilog?

- SystemVerilog is a Hardware Description Language (HDL).
 - We can describe digital circuits in code!

```
module AOI (F, A, B, C, D);  
    output logic F;  
    input logic A, B, C, D;  
    assign F = ~((A & B) | (C & D));  
endmodule
```



- Different from your normal programming language:
 - The language primitives are fundamentally different (e.g., wires and gates instead of variables).
 - Hardware execution is **concurrent** (i.e., hardware never goes away and is constantly computing), as opposed to **sequential** software execution (i.e., one instruction at a time).

Modules

- The basic building block in SystemVerilog is the **module**, which represents connected “black boxes” in our designs.
 - One *definition*, enclosed between the keywords `module` and `endmodule`.
 - As many *instances* as desired, each identified uniquely by name.

Definition:

```
module AOI (F, A, B, C, D);  
  output logic F;  
  input logic A, B, C, D;  
  ... // implementation  
endmodule
```

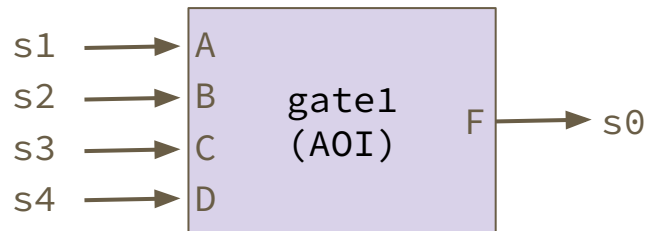
Annotations: `module name` points to `AOI`; `port list (e.g., inputs and outputs)` points to `(F, A, B, C, D)`; `port types` points to `output logic` and `input logic`.

Instantiation:

```
AOI gate1 (.F(s0), .A(s1), .B(s2),  
           .C(s3), .D(s4));
```

Annotations: `instance name` points to `gate1`; `port connections (here, explicit)` points to `(.F(s0), .A(s1), .B(s2), .C(s3), .D(s4))`.

Block:



Logic Gates

- Basic gates can be specified using operators:
 - `~` is a 1-input NOT
 - `&` is a 2-input AND
 - `|` is a 2-input OR
 - All other gates can be built from combinations of these
- Other gate variants can be instantiated as built-in modules:
 - `<gate> <instance_name> (output, input, ...);`
 - e.g., `and g1 (F, A, B, C, D);` *// 4-input AND gate named g1*

Combinational Logic in SystemVerilog

- `assign` – a single continuous assignment statement
 - The specified relationship will hold true for ALL time.
 - e.g., `assign F = ~((A & B) | (C & D));`
 - Can have as many `assign` statements as needed, but each must set a *different* signal (i.e., no contention/conflicts).

Signals in SystemVerilog

- Basics:
 - “Variables” still need to be declared but correspond to either wires (`wire`) or variable voltage sources (`reg`)
 - We will use `logic` for everything in this class (compiler resolves to `wire/reg`)
 - A **bus** (multi-bit variable) can be declared by adding a dimension to the variable type (e.g., `logic [2:0]`)

Signals in SystemVerilog

- Basics:
 - “Variables” still need to be declared but correspond to either wires (`wire`) or variable voltage sources (`reg`)
 - We will use `logic` for everything in this class (compiler resolves to `wire/reg`)
 - A **bus** (multi-bit variable) can be declared by adding a dimension to the variable type (e.g., `logic [2:0]`)
- Signal manipulation:
 - `bus[#]` – Get an individual value from a bus
 - `bus[#:#]` – Get a group/slice of values from a bus
 - `{ sig, sig, ... }` (concatenation) – Create a new bus from an ordered collection of existing signals
 - `{N{sig}}` (replication) – Create a new bus from N copies of a signal

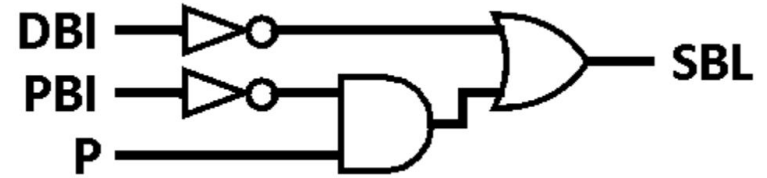
Coding Exercises

Exercise 1

- Write a SystemVerilog module that implements the Seat Belt Light circuit from Lecture 1:
 - SeatBeltLight (DriverBeltIn, PassengerBeltIn, Passenger)
 - Don't mix-and-match – use either all built-in operators or all built-in gates



Exercise 1 (Solution)

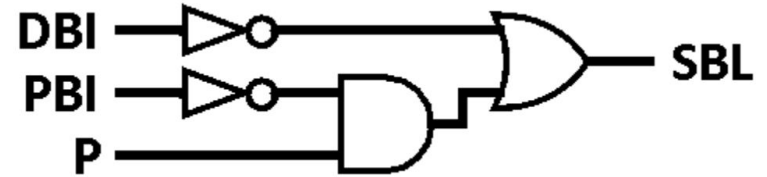


- Module skeleton

```
module seatbelt_light      (input  logic DBI, PBI, P,  
                           output logic SBL);
```

```
endmodule // seatbelt_light
```

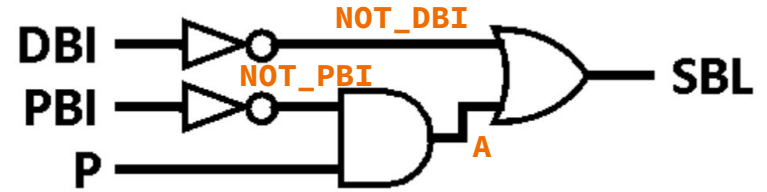
Exercise 1 (Solution)



- **Version 1:** using built-in operators, single assignment

```
module seatbelt_light_ops1(input logic DBI, PBI, P,  
                           output logic SBL);  
  
    assign SBL = (~DBI) | (P & ~PBI);  
  
endmodule // seatbelt_light_ops1
```

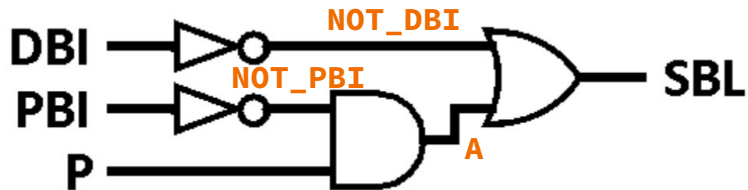
Exercise 1 (Solution)



- **Version 2:** using built-in operators, with intermediate signals

```
module seatbelt_light_ops2(input logic DBI, PBI, P,  
                           output logic SBL);  
  
    // Intermediate signals  
    logic NOT_DBI, NOT_PBI, A;  
  
    NOT_DBI = ~DBI;  
    NOT_PBI = ~PBI;  
    A = NOT_DBI & NOT_PBI;  
    SBL = A | P;  
  
endmodule // seatbelt_light_ops2
```

Exercise 1 (Solution)



- **Version 2:** using built-in operators, with intermediate signals

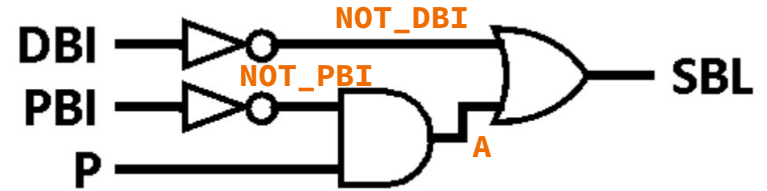
```
module seatbelt_light_ops2(input logic DBI, PBI, P,
                          output logic SBL);

    // Intermediate signals
    logic NOT_DBI, NOT_PBI, A;

    // Individual signal assignments
    assign NOT_DBI = ~DBI;
    assign NOT_PBI = ~PBI;
    assign A       = P & NOT_PBI;
    assign SBL     = A | NOT_DBI;

endmodule // seatbelt_light_ops2
```

Exercise 1 (Solution)



- **Version 3:** using built-in gates

```
module seatbelt_light_gate(input logic DBI, PBI, P,
                          output logic SBL);

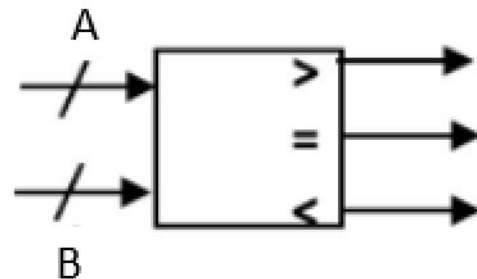
    // Intermediate signals
    logic NOT_DBI, NOT_PBI, A;

    // Individual signal assignments
    not gate1(NOT_DBI, DBI);      // ~DBI
    not gate2(NOT_PBI, PBI);     // ~PBI
    and gate3(A, P, NOT_PBI);    // P & ~PBI
    or gate4(SBL, A, NOT_DBI);   // A | NOT_DBI

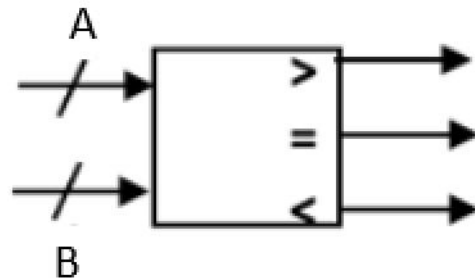
endmodule // seatbelt_light_gate
```


Comparator

- Circuit that compares two numbers.
 - Inputs:
 - A: first number
 - B: second number
 - Inputs assumed *signed*
 - Outputs:
 - `is_gt (>)`: $A > B$
 - `is_eq (=)`: $A == B$
 - `is_lt (<)`: $A < B$



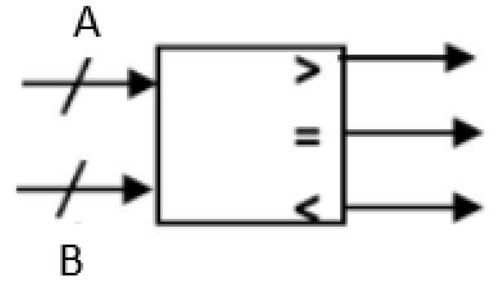
Comparator



- Circuit that compares two numbers.
 - Inputs:
 - A: first number
 - B: second number
 - Inputs assumed *signed*
 - Outputs:
 - `is_gt (>)`: $A > B$
 - `is_eq (=)`: $A == B$
 - `is_lt (<)`: $A < B$
- For simplicity, we will take advantage of the subtraction/minus (-) operator in Verilog.
 - `is_lt`: (Most significant bit of $A-B$) == 1 (negative)
 - `is_eq`: NOR all bits of $A-B$
 - `is_gt`: (MSB of $A-B$) == 0 AND $\sim is_eq$
 - Note: these fail some edge cases but we will ignore those for now.

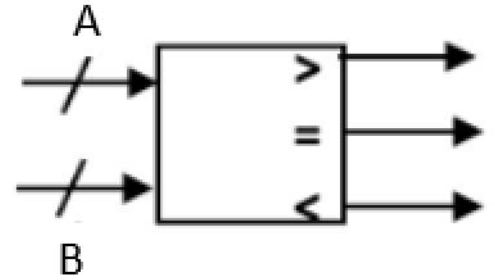
Exercise 2

- Create a comparator module for 3-bit inputs.



Exercise 2 (Solution)

- Module skeleton

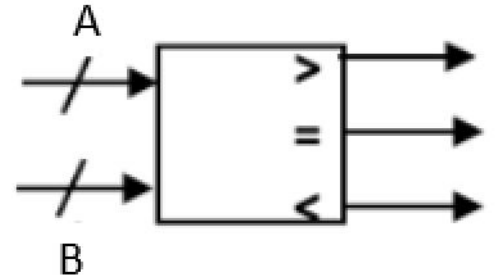


```
module comparator (input logic [2:0] A, B,  
                  output logic is_lt, is_gt, is_eq);
```

```
endmodule // comparator
```

Exercise 2 (Solution)

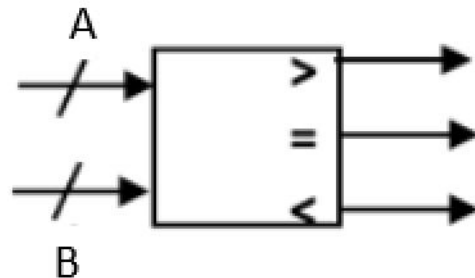
- Compute intermediate result



```
module comparator (input logic [2:0] A, B,  
                  output logic is_lt, is_gt, is_eq);  
  
    // subtraction result (intermediate)  
    logic [2:0] sub;  
    assign sub = A - B;  
  
endmodule // comparator
```

Exercise 2 (Solution)

- Compute outputs



```
module comparator (input logic [2:0] A, B,  
                  output logic is_lt, is_gt, is_eq);  
  
    // subtraction result (intermediate)  
    logic [2:0] sub;  
    assign sub = A - B;  
  
    assign is_eq = ~(sub[0] | sub[1] | sub[2]);  
    assign is_lt = sub[2];  
    assign is_gt = ~is_eq & ~is_lt;  
  
endmodule // comparator
```

Block Diagrams

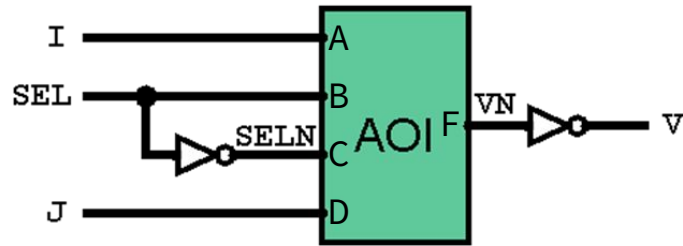
- **Block diagrams** are the basic design tool for digital logic.
 - The diagram itself is a module → inputs and outputs shown and connected.
 - Major components are represented by blocks (“black boxes”) with their internals abstracted away → each block becomes its own module.
 - All ports for each block 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.

Block Diagrams

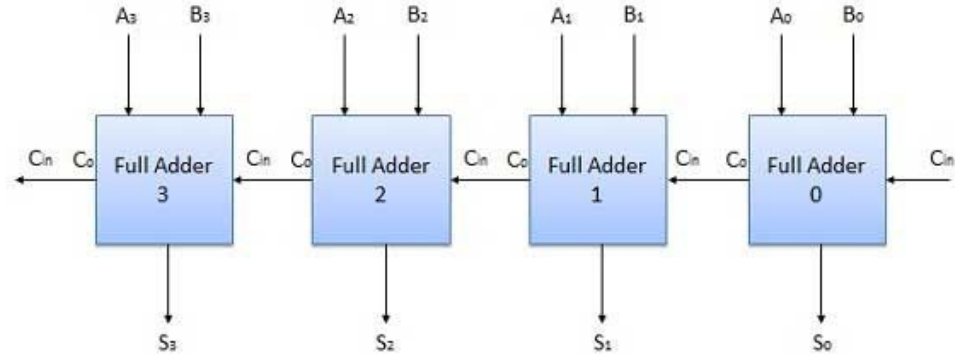
- **Block diagrams** are the basic design tool for digital logic.
 - The diagram itself is a module → **inputs and outputs shown and connected.**
 - Major components are represented by blocks (“black boxes”) with their internals abstracted away → **each block becomes its own module.**
 - All ports for 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!

Block Diagram Examples

- MUX2 from AOI (Lecture 2)



- Ripple Carry Adder (Lecture 6)

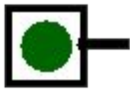


Exercise 3

- Create a magic number guessing game using the comparator module:
 - Your system should have a “secret” hard-coded number (you choose!).
 - Reminder: a constant in SystemVerilog looks like `3'b001`.
 - `SW[2:0]` is the user’s guess.
 - `KEY[0]` is pressed this when the user is ready to check their guess (**check**).
 - KEYS are *active-low* (i.e., `0` is “on”).
 - LEDs should indicate the outcome of the guess if **check** is asserted:
 - `LEDR[0]` should light up if the guess $>$ the secret number (signed comparison).
 - `LEDR[1]` should light up if the guess $==$ the secret number.
 - `LEDR[2]` should light up if the guess $<$ the secret number (signed comparison).
- 1) Draw a block diagram of your proposed system
 - 2) Implement the system in SystemVerilog

Exercise 3 (Solution) – Block Diagram

KEY[0] 

SW[2] 

SW[1] 


SW[0] 


 LEDR[2]

 LEDR[1]

 LEDR[0]

Exercise 3 (Solution) – Block Diagram

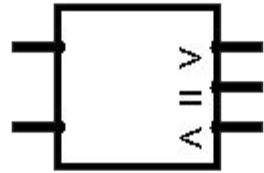
KEY[0] 

SW[2] 

SW[1] 

SW[0] 

Secret
Number 1 

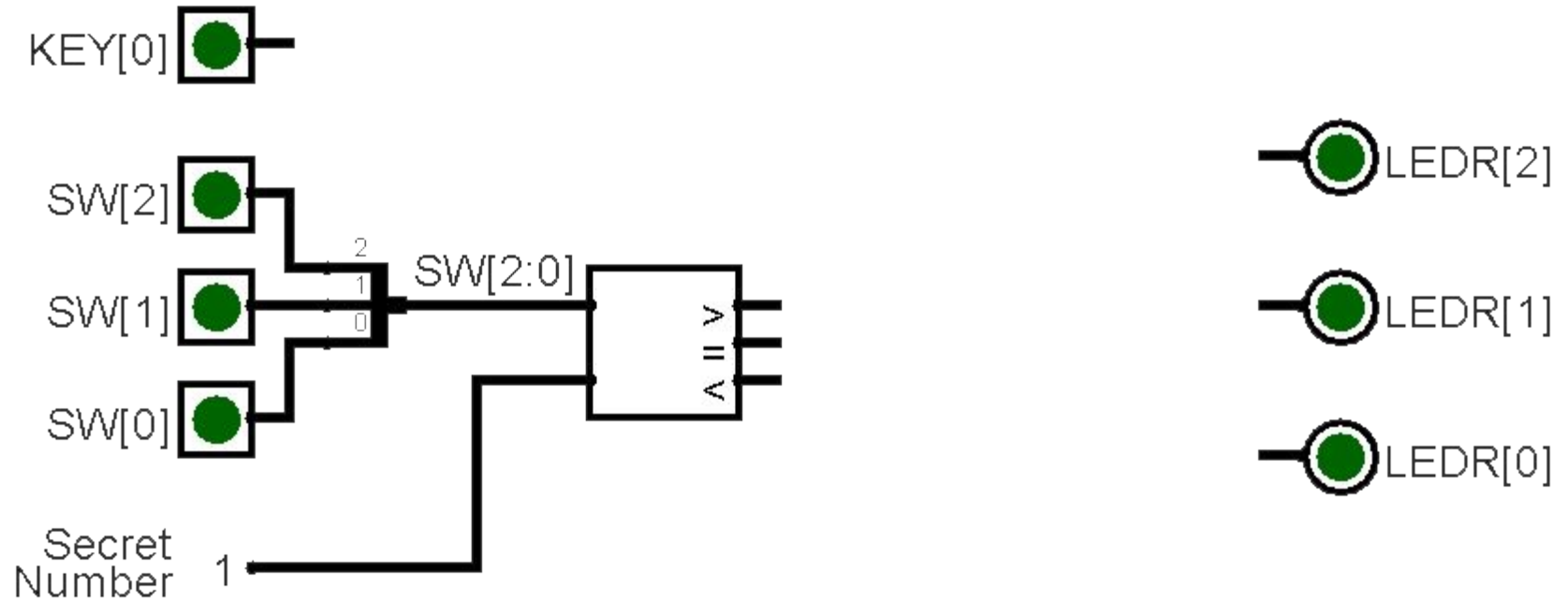


 LEDR[2]

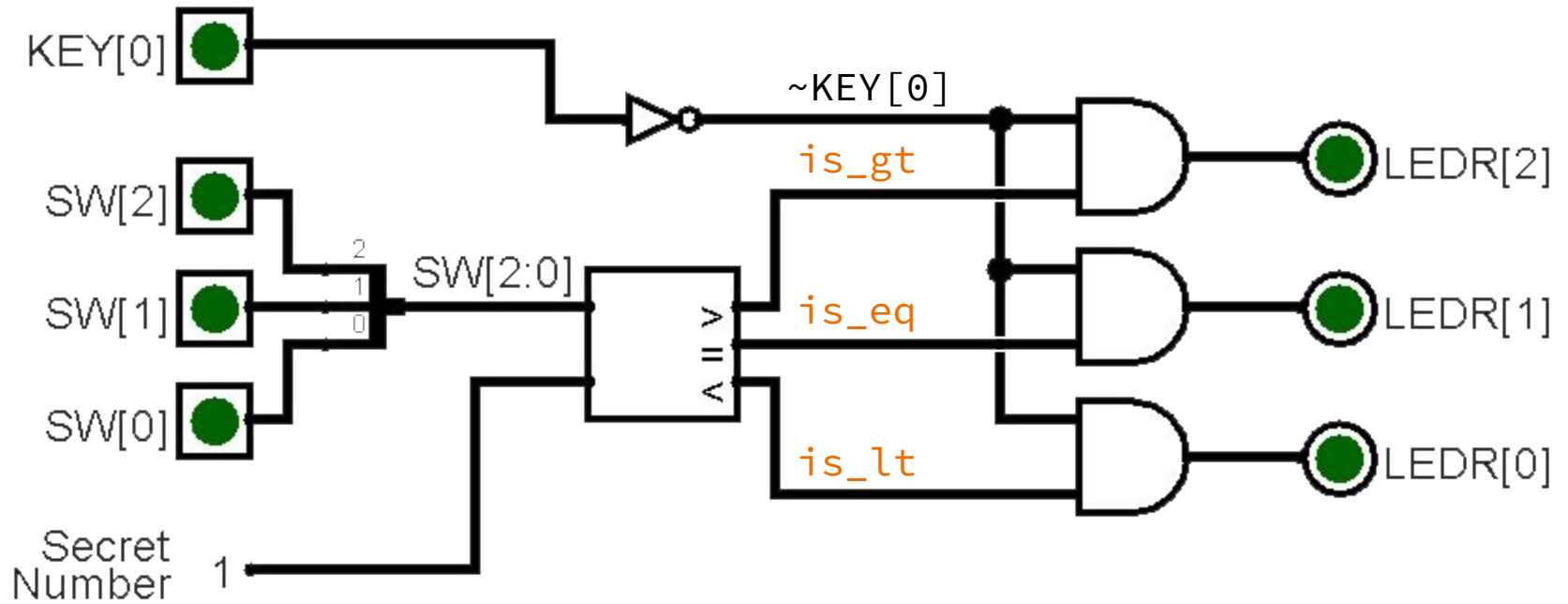
 LEDR[1]

 LEDR[0]

Exercise 3 (Solution) – Block Diagram



Exercise 3 (Solution) – Block Diagram



Exercise 3 (Solution) – Code

- **Define intermediate signals**
 - Needed for module port connections and output computations.

```
module guessing_game (  
    output logic [9:0] LEDR,  
    input  logic [3:0] KEY,  
    input  logic [9:0] SW  
);  
  
    logic is_lt, is_eq, is_gt;  
  
endmodule // guessing_game
```


Exercise 3 (Solution) – Code

- **Module instantiation**

- Hard-coding the secret number directly into a port.
- Ordering of A and B connections matters (subtraction is not commutative).
- Ordering of ports when using explicit connections doesn't matter.

```
module guessing_game (  
    output logic [9:0] LEDR,  
    input  logic [3:0] KEY,  
    input  logic [9:0] SW  
);  
  
    logic is_lt, is_eq, is_gt;  
  
    comparator number_comparator (  
        .A(SW[2:0]),  
        .B(3'b001),      // secret number  
        .is_lt(is_lt),  
        .is_eq(is_eq),  
        .is_gt(is_gt)  
    );  
  
endmodule // guessing_game
```

Exercise 3 (Solution) – Code

- **Compute outputs**
 - Ordering of assignments doesn't matter because we're describing hardware – could have been above comparator instantiation!

```
module guessing_game (  
    output logic [9:0] LEDR,  
    input  logic [3:0] KEY,  
    input  logic [9:0] SW  
);  
  
    logic is_lt, is_eq, is_gt;  
  
    comparator number_comparator (  
        .A(SW[2:0]),  
        .B(3'b001),      // secret number  
        .is_lt(is_lt),  
        .is_eq(is_eq),  
        .is_gt(is_gt)  
    );  
  
    assign LEDR[0] = is_lt & ~KEY[0];  
    assign LEDR[1] = is_eq & ~KEY[0];  
    assign LEDR[2] = is_gt & ~KEY[0];  
  
endmodule // guessing_game
```