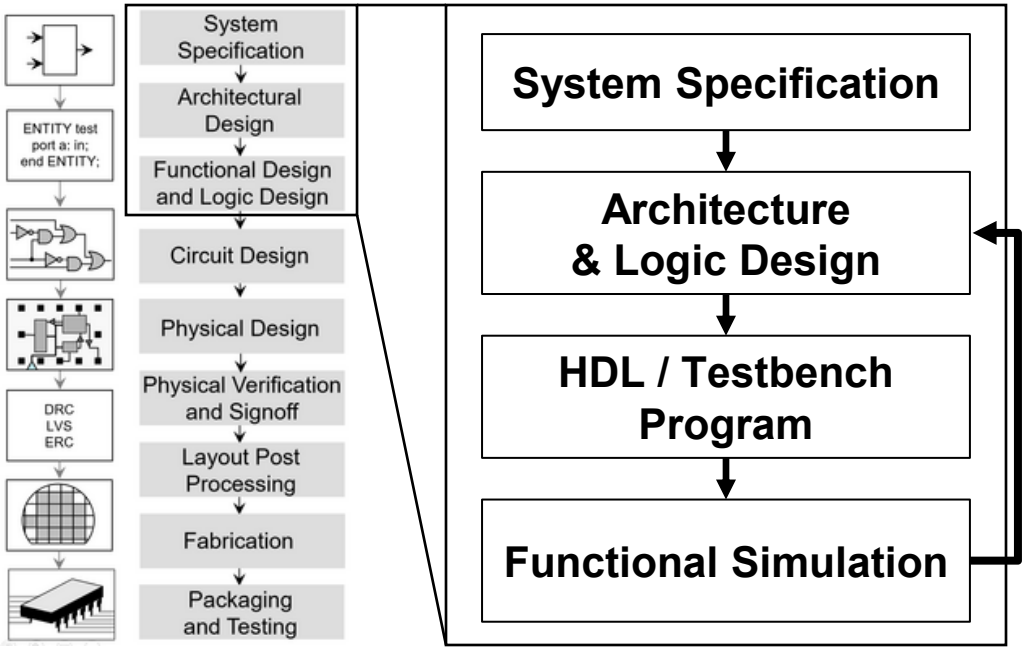
Lecture 2: Verilog

Hunjun Lee
hunjunlee@hanyang.ac.kr

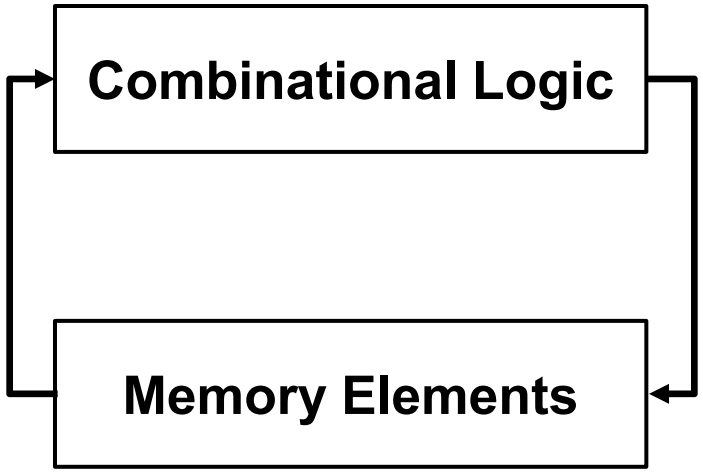
How to describe hardware?

- **RTL (Register Transfer Level) model** is a high-level abstraction describing the hardware
- RTL model should be functionally verified **prior to be physically implemented into electrical circuits**



What is HDL?

- HDL describes what happens at each cycle! (i.e., register transfers)
  - Which data are read from the memory elements
  - How the data are modified (using combinational logic)
  - Where the modified data are stored



Assumes that the described behaviors happen "within a cycle"!

Why do we need HDL (instead of C/C++ …)?

- **HDLs enable easy description of hardware structures**
  - Wires, gates, registers, flip-flops, clock, rising/falling edge, …
  - Combinational and sequential logic elements

- **HDLs enable seamless expression of parallelism inherent in hardware**
  - All hardware logic operates concurrently

There are ways to use C-style languages to describe hardware modules

➔ *You will learn these along with the HDL*

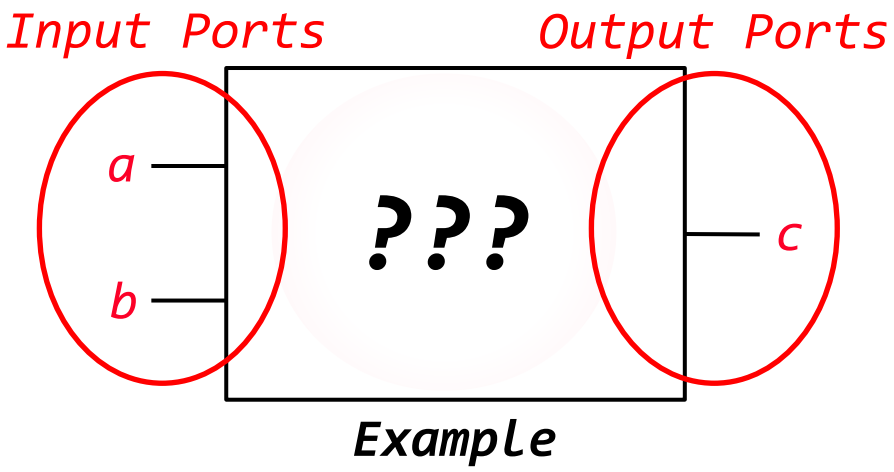
Quick Question! Then why would you use C/C++?

Modular design

What is a module?

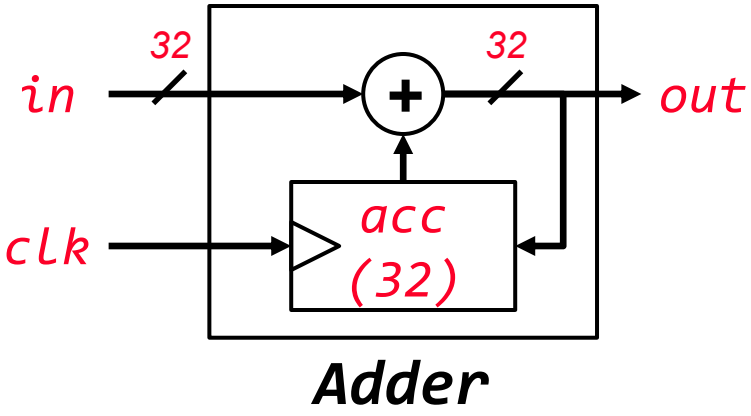
- A module is a main building block in Verilog
  - Think of a class in OOP

- We need to specify the following:
  - Name of the module
  - Input and Output of the module
  - The functionality of the module



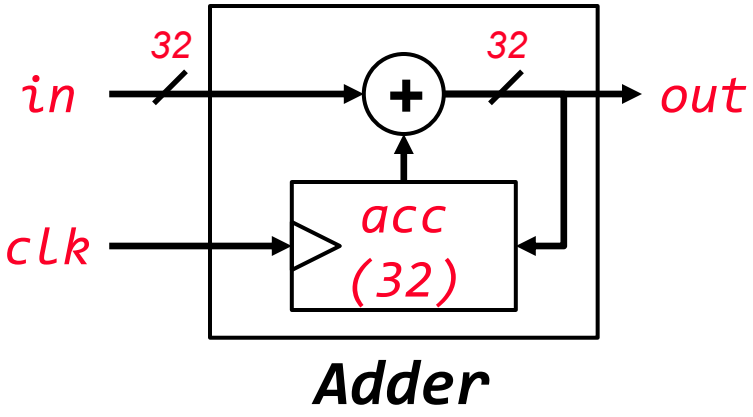
Suppose I want an adder

- The specification I want is an adder that
  - Receives a 32-bit input *in*
  - Stores a 32-bit value *acc*
  - Adds *in* and *acc* to generate 32-bit output result, *out*
  - Result is stored to *acc* upon receiving a positive edge clock (*clk*)



I want an adder: CPP Style

```
class Adder {
    void tick(); // Describes what happens @ each cycle
    unsigned acc;
}

void Adder::tick (
    unsigned in,
    unsigned *out        // output (pass by ref)
) {
    *out = in + acc;
    acc = *out;
}
```



I want an adder: Verilog Style

```
`timescale 1ns / 100ps

module Adder (
    input clk,
    input  [31:0]  in,
    output [31:0]  out
);

    reg [31:0] acc;

    assign out = in + acc;

    always @(posedge clk) begin
        acc <= out;
    end

endmodule
```

#1. Timescale

#2. Module

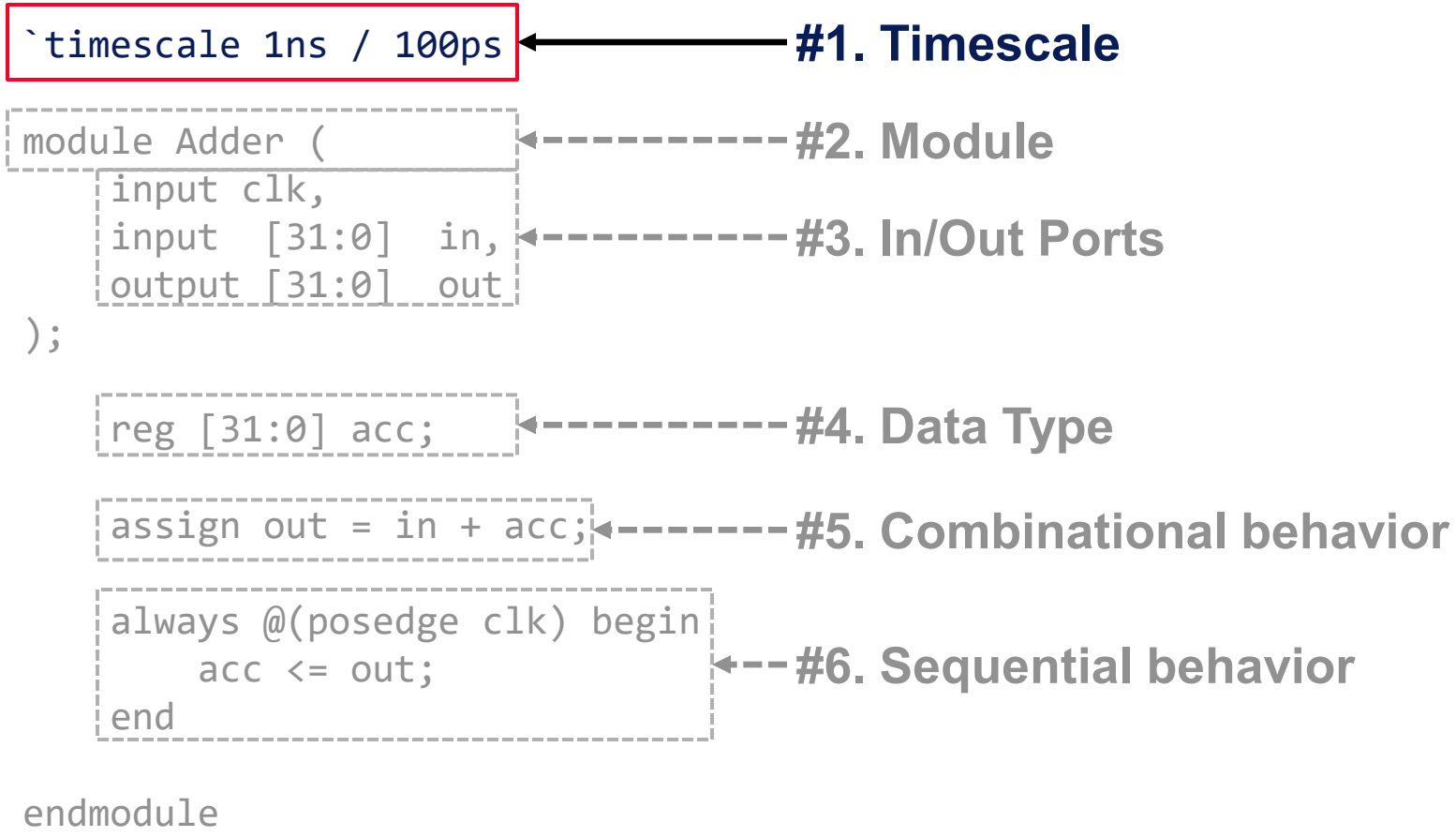
#3. In/Out Ports

#4. Data Type

#5. Combinational behavior

#6. Sequential behavior

I want an adder: Verilog Style



Timescale

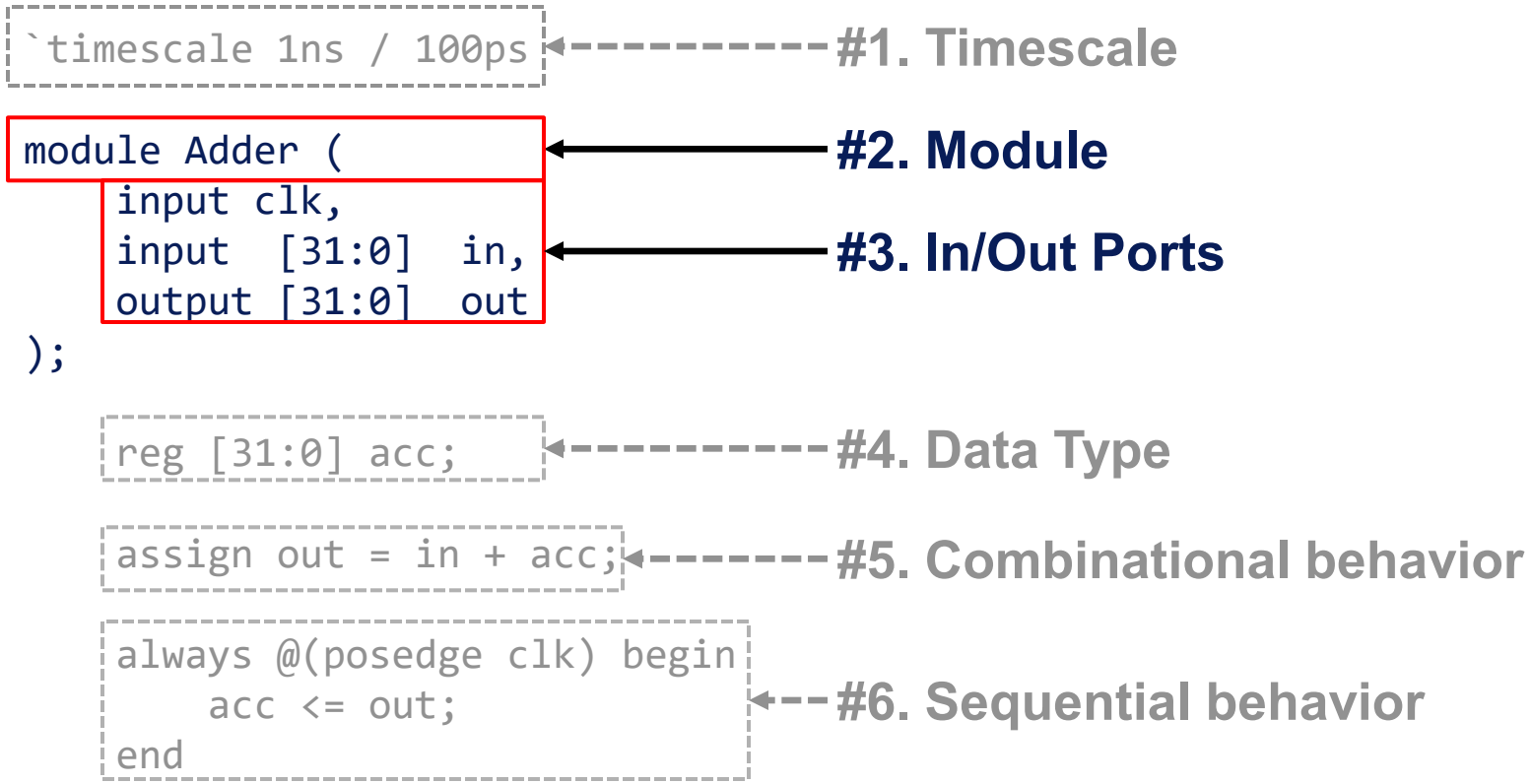
- Syntax:
  - `` `timescale time_unit / time_precision ``
- Of course, the `time_precision` cannot be longer than `time_unit`
  - Ex) `` `timescale 1 ns / 100 ps ``
- When it is used?
  - Indicate the delay of some logic (`#5` → `5 ns`)

```
assign #5 a = b; // assign b to a after 5 ns
```

  - When setting the clock period

```
always
    #5 clk = ~clk; // clk changes every 5 ns (10 ns period)
```

I want an adder: Verilog Style



Module syntax

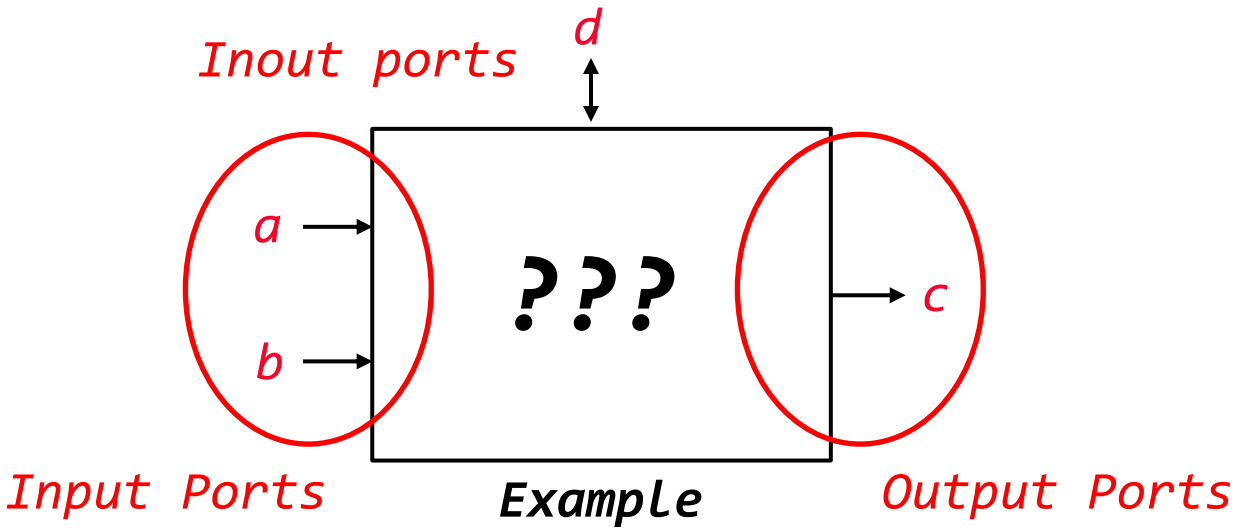
- We can declare modules in two different ways

Method #1

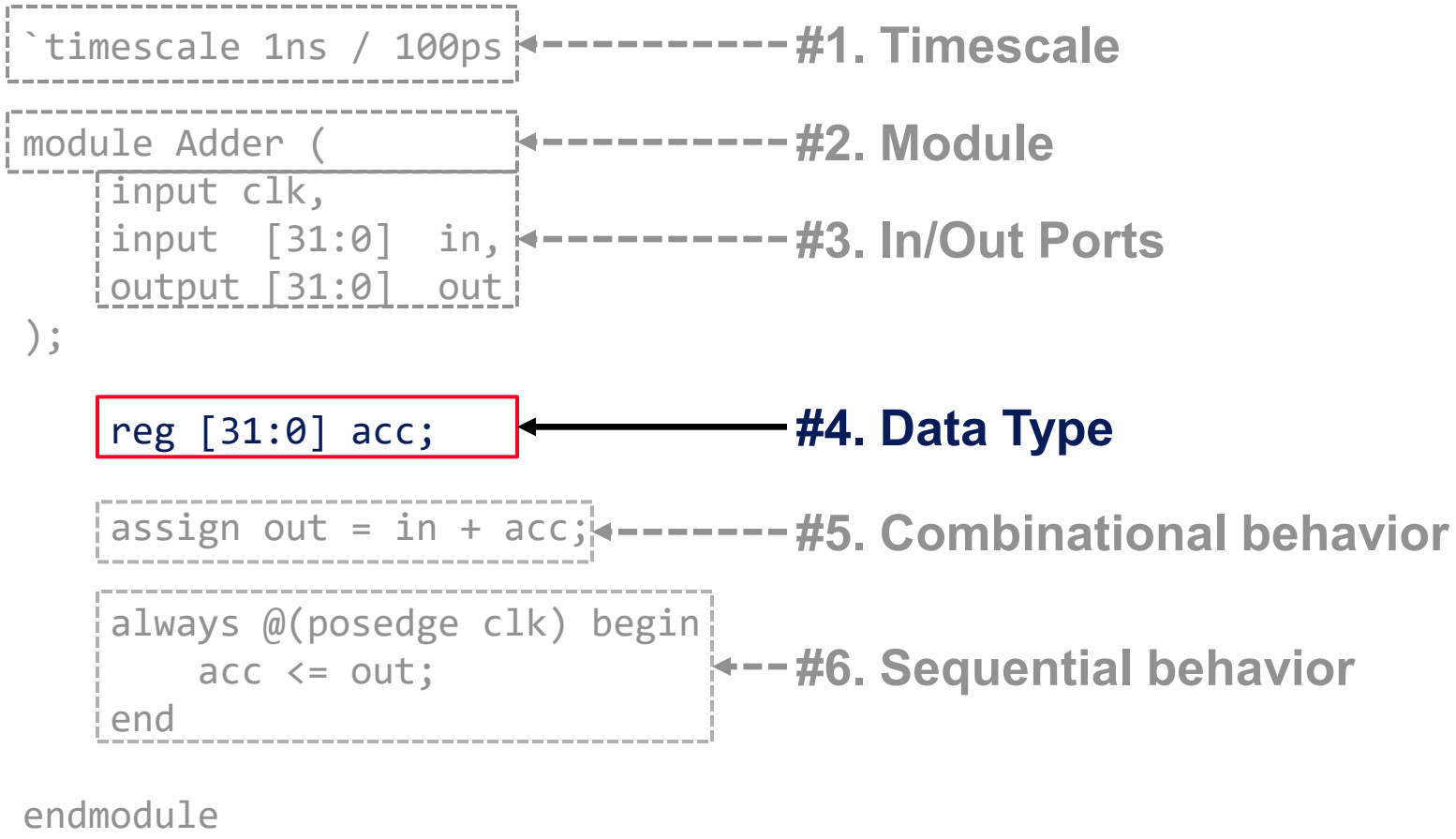
```
module Example (
    input  [31:0]  a,
    input  [7:0]   b,
    output [7:0]   c,
    inout [7:0]    d
);
endmodule
```

Method #2

```
module Example (a, b, c, d);
input  [31:0]  a;
input  [7:0]   b;
output [7:0]   c;
inout [7:0]    d;

endmodule
```



I want an adder: Verilog Style



Data Types

- There are two types of data: (1) stored data and (2) transient transmission data
  - **net:** a representation of physical connections (i.e., wire)
  - **variable:** an abstraction of data storage elements (i.e., reg)

  same as input & output ports, each value can be ***multiple bits***

- There can be two different values (per bit)
  - **0:** a logic zero
  - **1:** a logic one
  - **x:** an unknown logic value (can be both 0 and 1)
  - **z:** a high-impedence state (not connected)

Data type - wire

- Wire defines a physical connection with a continuous assignment
- Example

```
input [31:0] op1;
input [31:0] op2;
wire [31:0] out;
assign out = op1 + op2; // we'll get to this later on
```

- You can use assign statement of indicate where the wire is connected to!
  - In the example above, `out` is connected to the result of the adder (that receives `op1` and `op2` as inputs)

Data type – reg (1)

- Reg can also be used to define a physical connection
- But, also defines a storage element which can be read or written
- Example (physical connection)

```
input [31:0] op1;
input [31:0] op2;
reg [31:0] out;
// same as wire [31:0] out (but with different
// assignment syntax)
always @(op1, op2)     // We'll get to this later on
   out = op1 + op2;
```

Data type – reg (2)

- Example (storage element)

```
input [31:0] op1;
input [31:0] op2;
output [31:0] out;
input clk;
reg [31:0] acc;
// completely different meaning with wire [31:0] acc
assign out = acc; // reading the value
always @(posedge clk)  // We'll get to this later on
   acc <= op1 + op2;  // writing the value
```

- You can define the sequential logic (synchronous)
  - In the example above, `out` is a storage element (latch) which latches the result of the adder (at the positive edge of a clock)

Data type – reg (3)

- You can generate a memory array using reg syntax

```
wire [7:0] data1;
wire [7:0] data2;

// an array of 8 bit data (10 bit address)
reg [7:0] mem [0:1023];

assign data1 = mem[0]; // data at address 0
assign data2 = mem[1]; // data at address 1
…
```

Data type – reg (4)

- Why not change the order?

```
// data.txt
0 1 2 3

// you can also make an array using the following
reg [7:0] mem_new [3:0];

initial begin
   $readmemh(“data.txt”, mem_new);
end

// This is awkward …
// mem_new[0] = 3, mem_new[1] = 2, …
```

Signed / Unsigned

- You can explicitly determine whether the data is signed or unsigned (default: unsigned)
  - This is influence how the operators behave

```
reg signed [31:0] a; // indicates that a is signed
wire signed [31:0] b;
```

  - You can explicitly set signed or unsigned

```
wire [31:0] result;
assign result = $unsigned(a) * $unsigned(b);
```

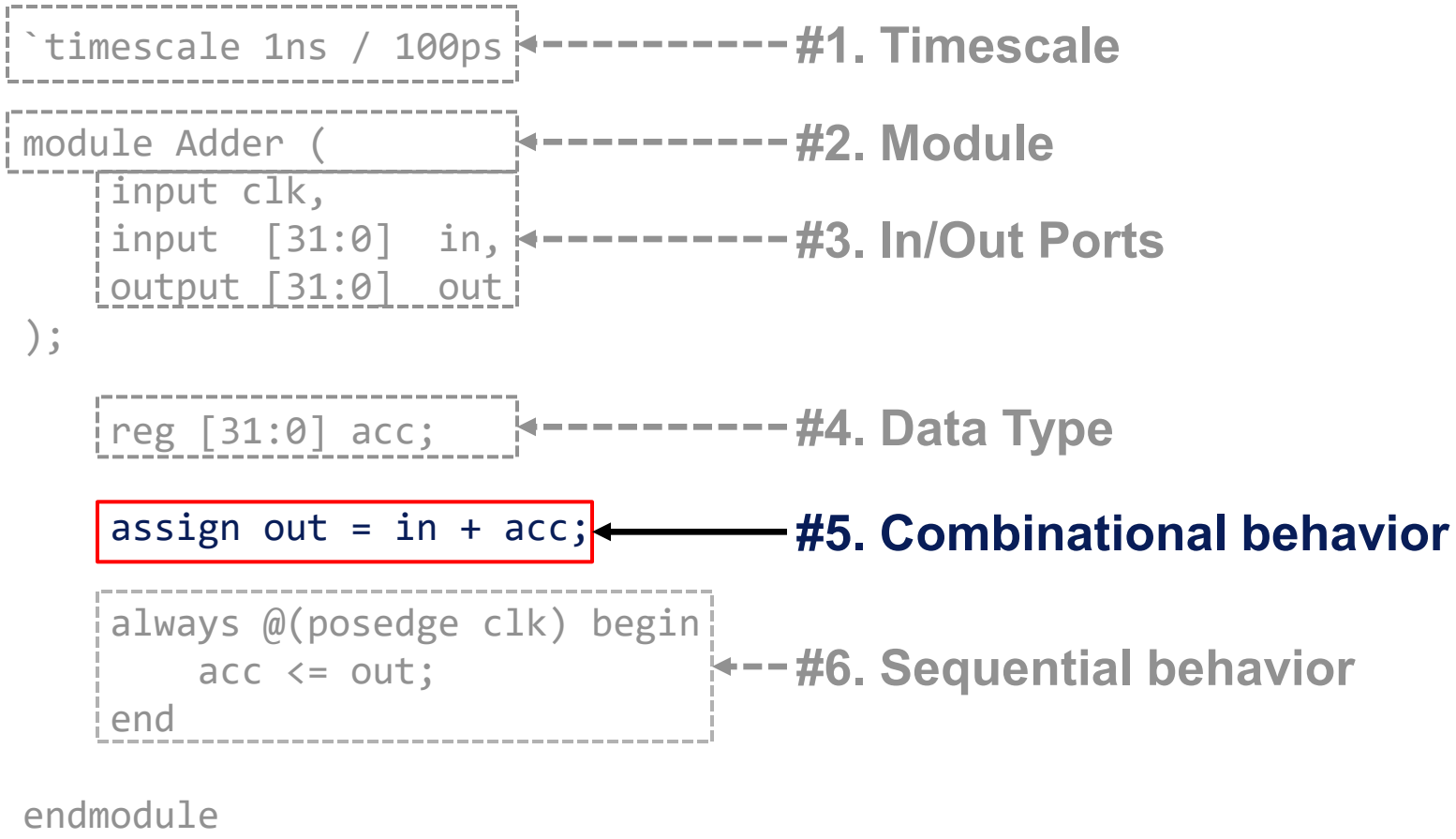
Representing a value

- Represent number using: `<size>'<radix><value>`
  - <size> : # of bits
  - <radix> : h(hex), d(decimal), b(binary)
  - <value> : Depend on the radix
    - X : "Unknown", Z : "High Impedance", _ : (Ignored)
  - Example: `8'b1111_0000, 16'd7, 32'hAAAA_BBBB`
    - What about this? `4'd200`

- Example

```
wire [7:0] example1;
assign example1 = 8'd10;
```

I want an adder: Verilog Style



Operators in Verilog

- Arithmetic
  - Binary : +, -, *, /, %
  - Unary : +, -
- Relational
  - Binary : <, >, <=, >=
- Equality
  - Binary : ===, !==
    - (comparing include x & z)
  - Binary : ==, !=
    - (returns x if an operand has x or z)
- Logical
  - Unary : !
  - Binary : &&, ||
- Bit-wise
  - Unary : ~
  - Binary : &, |, ^, ~^
- Reduction
  - Unary : &, ~&, |, ~|, ^, ~^
- Shift
  - Binary : <<, >>, <<<, >>>
- Concatenation
  - {a, b, c, …}
- Replication
  - {n, {m}}
- Conditional
  - a?b:c

Operators in Verilog

- Examples

```
input [31:0] op1;
input [31:0] op2;
input cond;
wire [31:0] result1, result2, …

assign result1 = op1 + op2; // add
assign result2 = op1 / op2; // div
assign result3 = cond ? op1 : op2 // conditional assign
assign result4 = op1 < op2;
assign result5 = &op1;
assign result6 = ^op1;
…
```

Manipulating bits

- Partial assignment

```
wire [15:0] short;
wire [7:0] char1, char2;
assign char1 = short[7:0]; // lower 8 bits
assign char2 = short[15:8]; // upper 8 bits
…
```

- Concatenation

```
assign short1 = {char, char};
assign short2 = {char, {8'b0}};
…
```

- Multiple copies

```
assign short = {2{char}};
assign new_char = {8{char[0]}};
…
```

Complex addressing methods

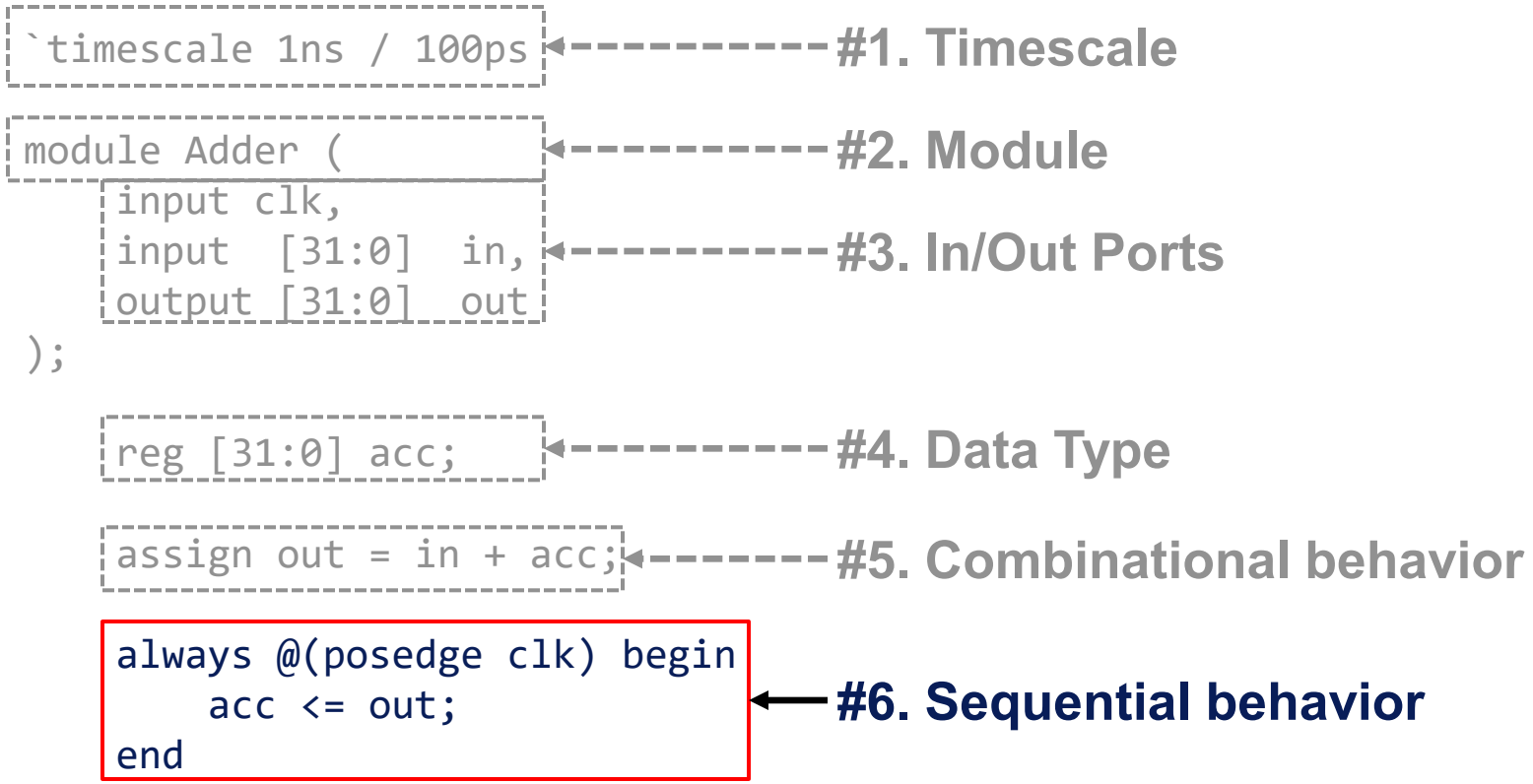
- We can address values in various ways
  - **`reg [63:0] word;`**
  - `word[0]`
  - `word[7:0]`
  - `word[0+:8] // == word[7:0]`
  - `word[15-:8] // == word[15:8]`
  - `word[x] // == x (unknown value)`
  - `word[64] // == x (unknown value)`

  - **`reg [7:0] mem[1023:0];`**
  - `mem[0] // access the first element`
  - `mem[0][3:0] // access lower 4 bits of word`
  - `mem[3:0] // Illegal`

What about FP?

- All the operators are for signed and unsigned data
- We need to define a custom module to support floating point operations (or fixed-point operations)
  - You can search for floating point operators in Verilog (in google or whatever)

I want an adder: Verilog Style



Procedural assignments

- To assign values in reg, use an always block! (or initial)

```
input [31:0] in;
reg [31:0] data;

// at the beginning of the simulation
initial begin
    data = 32'd10;
end

// always change data to in
// actually an infinite loop (not recommended)
always begin
    data = in;
end
```

There are two ways to assign values: blocking vs. non-blocking

- Blocking assignment (=)
  - executes immediately in sequence (similar to C)
  - variables are updated immediately
  - used in combinational logic (when the reg is used as a connection)

```
reg a;
reg b;
initial begin
    a = 0;
    b = 0;
end
always begin
    a = !a;       // a: 1 -> 0 -> 1 -> 0 …
    b = !a;       // b: 0 -> 1 -> 0 -> 1 …
end
```

There are two ways to assign values: blocking vs. non-blocking

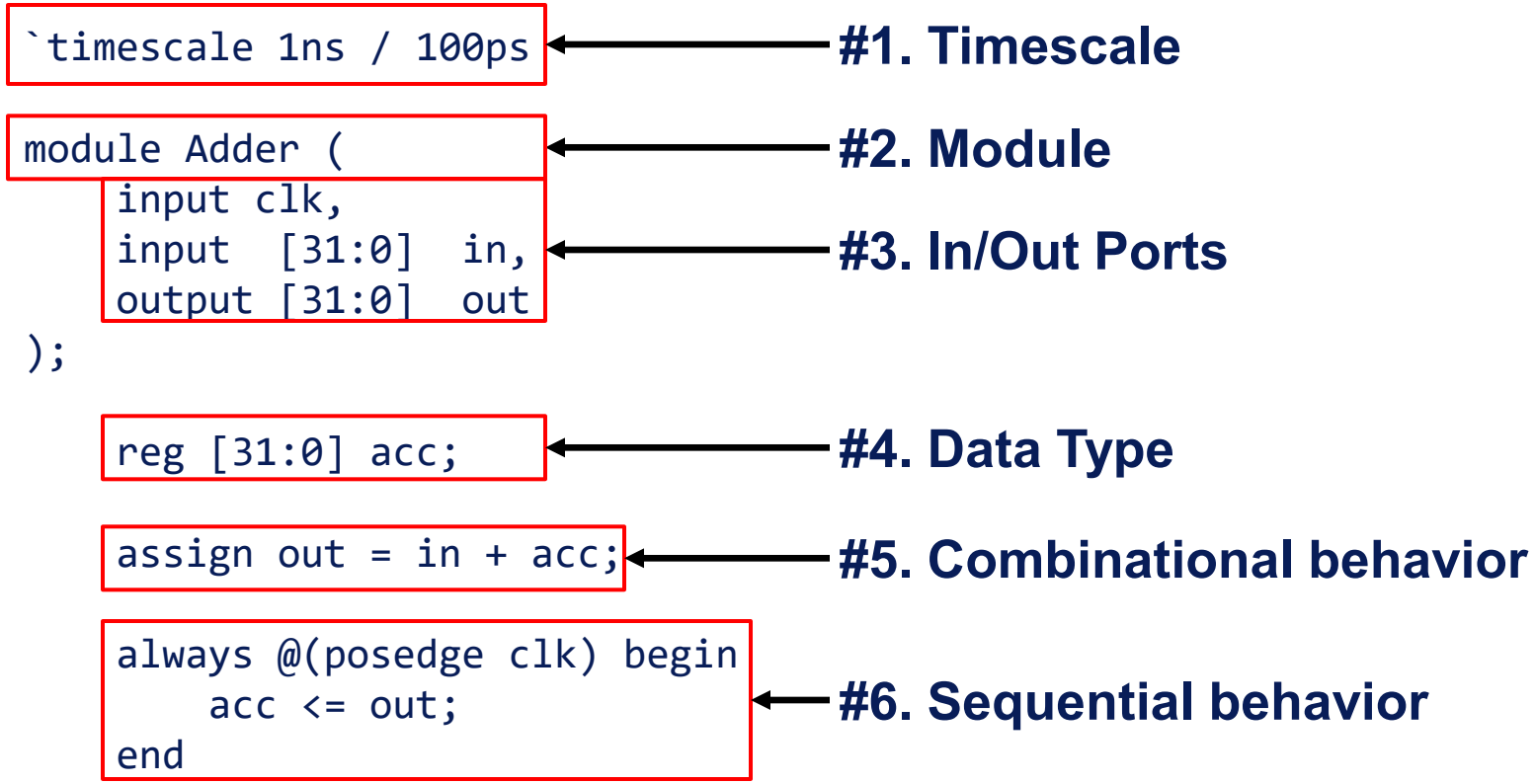
- Non-blocking assignment (<=)
  - variables are updated after the block finishes
  - used in sequential logic (when the reg is used as a storage)

```
reg a;
reg b;
initial begin
    a = 0;
    b = 0;
end

always begin
    a <= !a;      // a: 1 -> 0 -> 1 -> 0 …
    b <= !a;      // b: 1 -> 0 -> 1 -> 0 …
end
```

There are two ways to assign values: blocking vs. non-blocking

- Do not mix blocking and non-blocking assignments inside a single module (except initial block)
  - This is syntactically correct, but do not do this!!
- Blocking is used for combinational logic (ex ALU)
- Non-blocking is used for sequential elements (storage elements)
  - Except,,, you can use blocking at the initial block (but non-blocking at always)

I want an adder: Verilog Style



Advanced syntax: if

- You can use advanced syntax such as
  - if, case, repeat, forever, while, for inside the initial or always block

- If statement executes a statement conditionally (same as C)

```
module example ();
…
    initial begin
        if (a == 5) begin
            b = 15;
        end
        else begin
            b = 25;
        end
    end
endmodule
```

Advanced syntax: case

- If statement executes a statement conditionally (same as C)

```
module example ();
    …
    initial begin
        case (a)
            5: b = 15;
            default: b = 25;
        endcase
    end
endmodule
```

Advanced syntax: repeat

- Do the same statement multiple times
  - I don't recommend using this aside from initial block (or testbench) ➔ I'll get to this later on …

```
module example ();
// …
initial begin
    repeat (4) c = c + 1;
end
endmodule
```

≡

```
module example ();
// …
initial begin
    c = c + 1;
    c = c + 1;
    c = c + 1;
    c = c + 1;
end
endmodule
```

Advanced syntax: forever

- Repeat for infinite number of times
  - I don't recommend using this aside from initial block (or testbench)

```
module example ();
// …
initial begin
    forever c = c + 1;
end
endmodule
```

≡

```
module example ();
// …
initial begin
    c = c + 1;
    c = c + 1;
    c = c + 1;
    c = c + 1;
    …
end
endmodule
```

Advanced syntax: while & for

- It describes a repetition task, **but sometime it cannot be implemented as a real circuit.**
  - for: can be implemented as a real circuit (synthesized) for iterative combinational logic (e.g., ripple carry adder)
  - while: cannot be synthesized as they require unpredictable number of iterations **(think why …)**

```
module example ();
initial begin
    c = 0;
    while (c < 10) begin
        c = c + 1;
        …
    end
end
endmodule
```

```
module example ();
initial begin
    c = 0;
    for (c = 0; c < 10; c++) begin
        …
    end
end
endmodule
```

Timing controls – delay & event

- *Delay* and *event* are two methods for specifying when the procedural assignments occurs.

- Delay control
  - Delay a procedural statement in its execution
  - **Examples**
    - `#10 rega = reab;`
    - `#10 rega = rega + 1;`

  - It is used for describing the timing of real hardware, but not for controlling behaviors of Verilog code.
  - So, you must not use “delay”, except for a clock signal and a testbench.

Timing controls – delay & event

- Using delay to generate a clock

```
`timescale 1ns / 100ps

reg clk;
initial begin
    clk = 0;
end

always begin
    #5 clk = ~clk;
end
```

What is the clock period?

Timing controls – delay & event

- Event control
  - Synchronize a procedural statement with a value change on a ***wire*** or ***reg*** or the occurrence of a declared event
  - **Examples**

```
// assign when r changes
@r rega = reab;
// assign at positive edge of a clock
@(posedge clock) rega = regb;
// continuously check the negative edge and assign
always @(negedge clock) rega = regb;
// continuously check multiple signals
always @(a, b, c, d, e)
always @(posedge clk, negedge rstn)
always @(a or b, c, d or e)
```

Timing controls – delay & event

- Implicit event-expression list @(*)
  - Add all *wire* and *reg* that are read by the statements
  - **Examples**

```
always @(*) begin // equivalent to @(a or b or c or d)
    x = a ^ b;
    @(*) // equivalent to @(c or d)
        x = c ^ d;
end
```

Sequential logic using reg

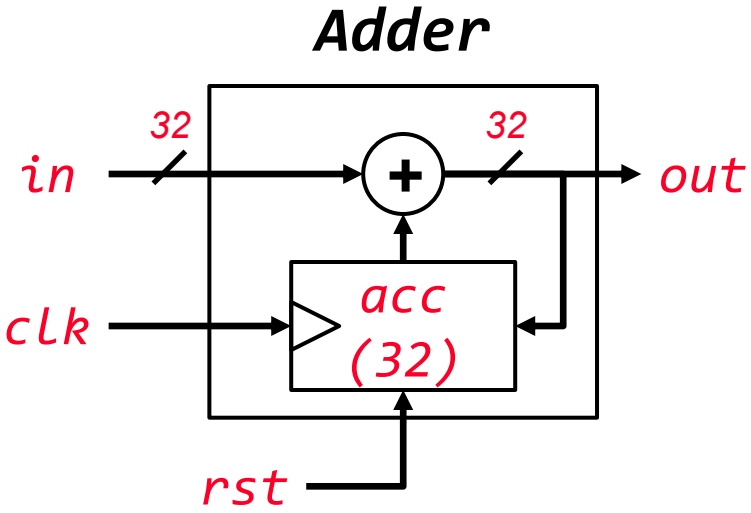
- We can describe a sequential logic using edge-triggered event control + non-blocking assignment
- **Examples**

```
// I want to use data1, data2 as a storage element
input [31:0] in1, in2;
reg [31:0] data1;
reg [31:0] data2;

// store in1 to data1 and in2 to data2
// at positive edge
always @(posedge clk) begin
    data1 <= in1;
    data2 <= in2;
end
```

I missed this one …
Reset Values

Reset!

- We also need a reset signal for correct simulation
  - `acc` is fixed to `0` is `rst` is `1`



Reset implementation

```
`timescale 1ns / 100ps

module Adder (
    input clk,
    input rst,
    input  [31:0]  in,
    output [31:0]  out
);

    reg [31:0] acc;

    assign out = in + acc;

    always @(posedge clk) begin
        if (rst)
            acc <= 0;
        else
            acc <= out;
    end

endmodule
```

synchronous reset

```
`timescale 1ns / 100ps

module Adder (
    input clk,
    input rst_n,
    input  [31:0]  in,
    output [31:0]  out
);

    reg [31:0] acc;

    assign out = in + acc;

    always @(posedge clk, negedge rst) begin
        if (!rst_n)
            acc <= 0;
        else
            acc <= out;
    end

endmodule
```

asynchronous reset

More about input and output

Input and output have datatype

- The input and output are wire by default!
- But you can set output as reg datatype

```
`timescale 1ns / 100ps
module Adder (
    input clk,
    input rst_n,
    input  [31:0]  in,
    output reg [31:0]  out // Set output as reg datatype
);

    reg [31:0] acc;
    always @(*) begin
        out = in + acc;
    end
    always @(posedge clk, negedge rst) begin
        if (!rst_n)
            acc <= 0;
        else
            acc <= out;
    end
endmodule
```

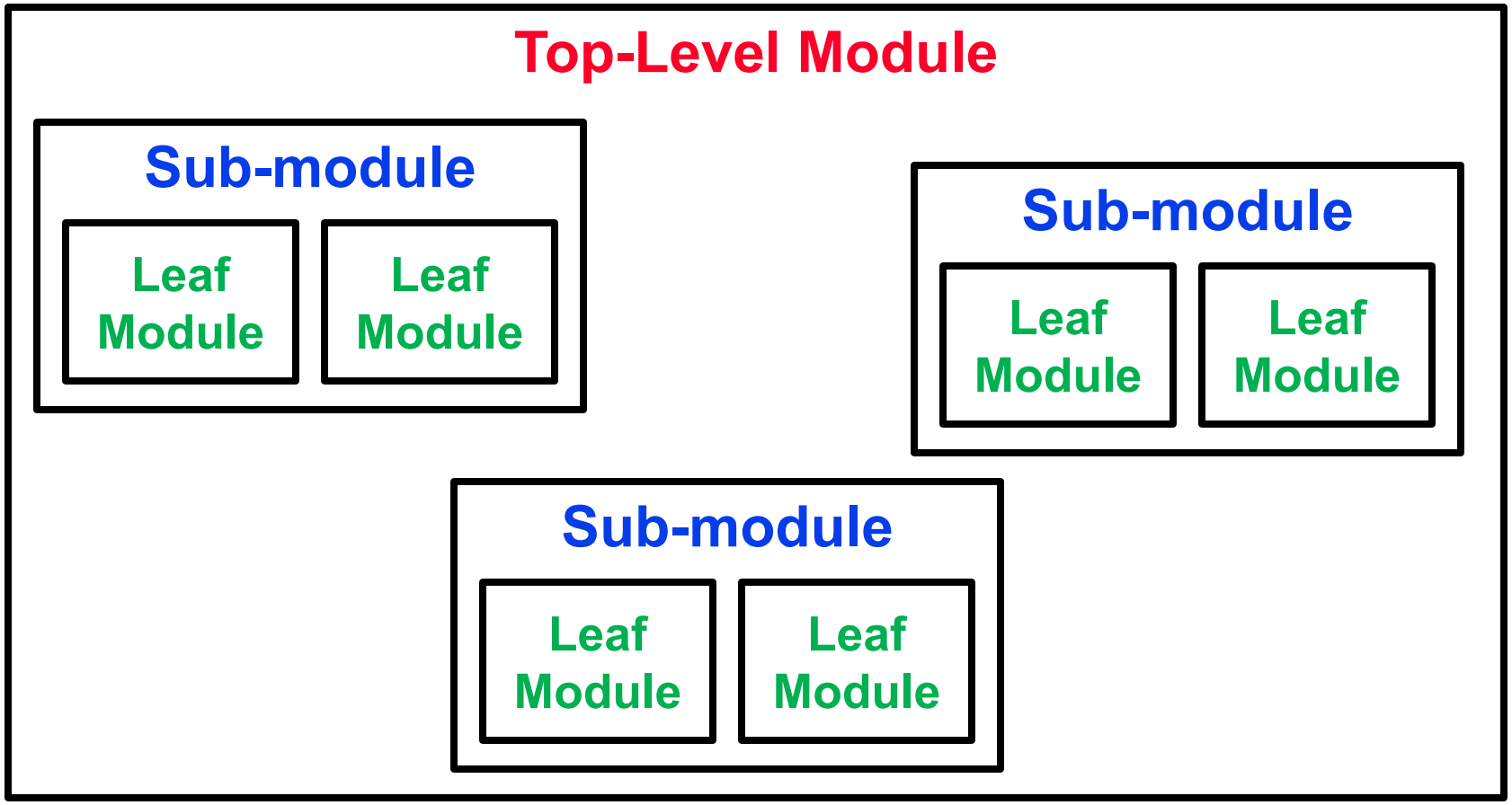
input/output data type : default : wire

→ output이 reg, reg로 선언해 주어야 한다.

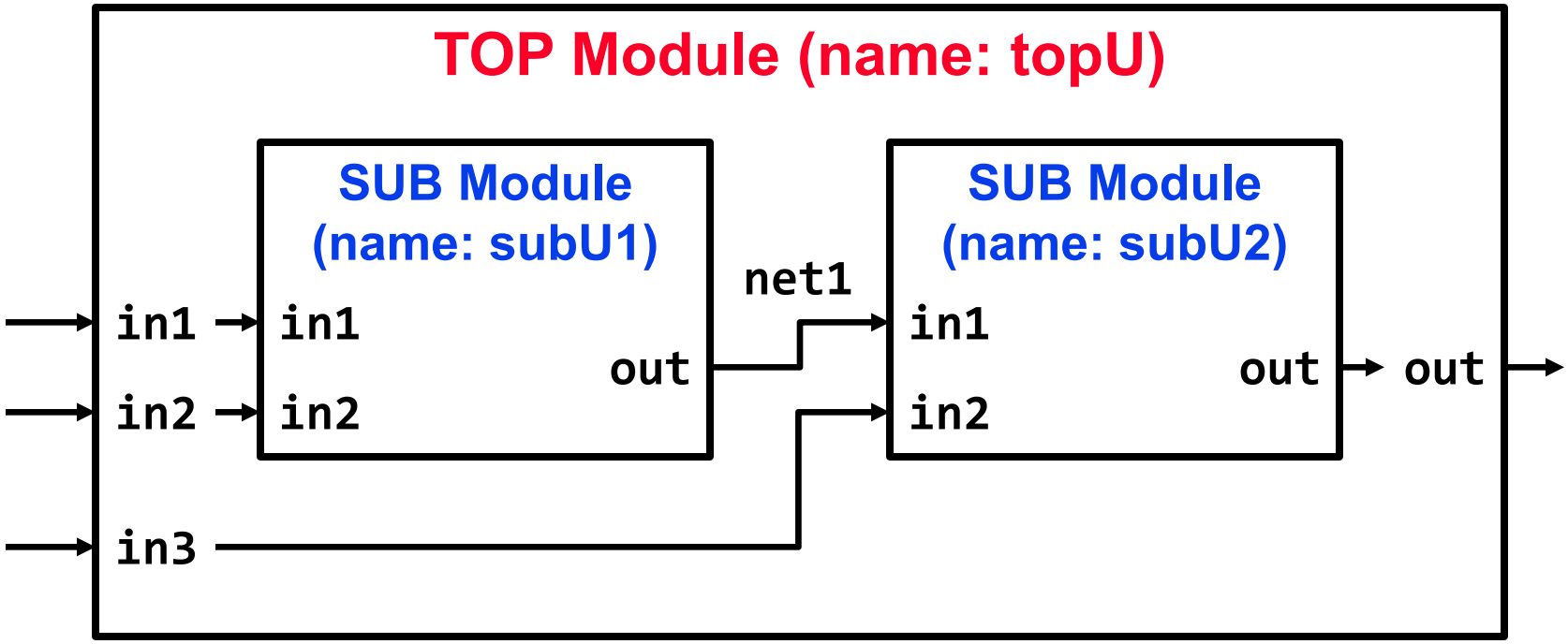
Structural HDL

Modular design methodology

- We define the **top-level module** and identify the **sub-modules** necessary to build the top-level module



Instantiate a module!



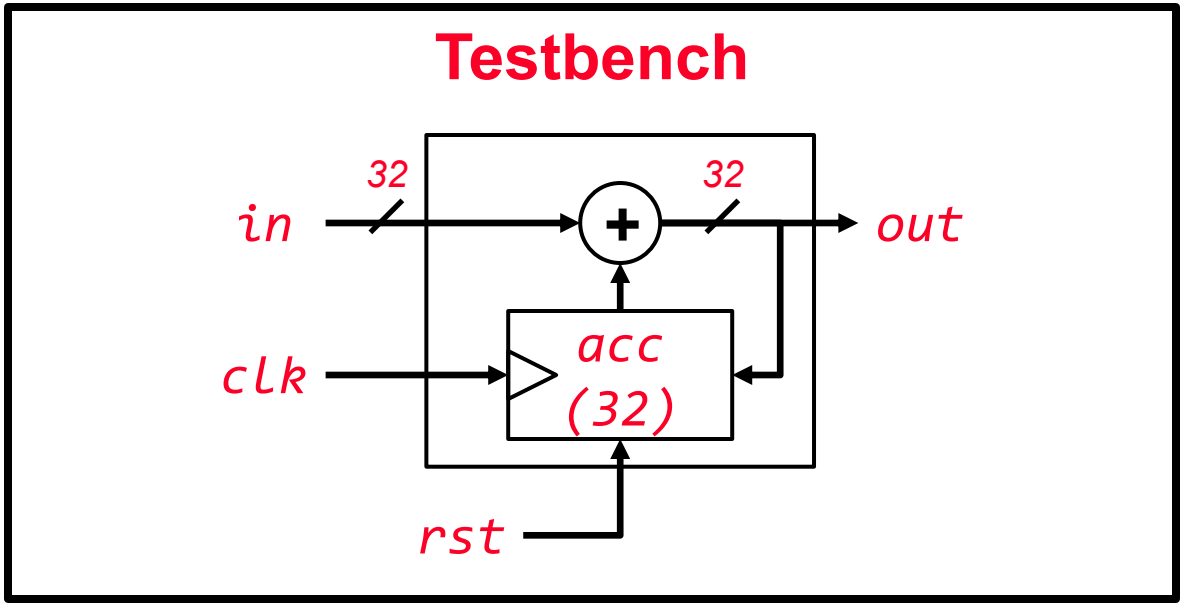
Instantiate a module!

```
module TOP (in1, in2, in3, out);
    input in1, in2, in3;
    output out;
    // you are not allowed to use reg net1;
    wire net1;

    // This is an option
    SUB subU1 (.in1(in1), .in2(in2), .out(net1));
    // You can also do this
    SUB subU2 (net1, in3, out);

endmodule

module SUB (in1, in2, out);
    input in1, in2;
    output out;

    // Description of the SUB body
endmodule
```

Testing the module

Back to the adder example

- We need to provide proper inputs to the adder and check if the output is correct



Test your module

```
`timescale 1ns / 100ps
module Adder_tb;
    // declare in/out ports for your module
    reg [31:0] in;
    reg clk, rst;
    wire out;
    // declare the target module
    Adder adderU (.clk(clk), .rst(rst), .in(in), .out(out));
    integer PASSED, FAILED;
    // generate the clock
    initial begin
        clk = 0;
        forever
            #5 clk = ~clk;
    end
    // set the values
    initial begin

        … (module test code)

    end
endmodule
```

We need a timing margin

Wait for a single period

```
PASSED = 0;
FAILED = 0;
rst = 1;
# 16 // wait for 16 ns
rst = 0;
in = 32'd1;
#1
if (out == 32'd1)
    PASSED = PASSED + 1;
else
    FAILED = FAILED + 1;
#9
in = 32'd2;
#1
if (out == 32'd3)
    PASSED = PASSED + 1;
else
    FAILED = FAILED + 1;
#9
in = 32'd5;
#1
…
$finish();
```

Compiler directives for simpler programming

Compiler directive

- `include
  - Insert the entire contents of a source file in another file during compilation
    - `include filename : filename에 존재하는 모든 내용과 똑같이 된다
- `define macro_name macro_text
  - Represent commonly used pieces of text
  - **Examples**

```
`define wordsize 8
reg [`wordsize-1:0] data;
```

= reg[7:0]

```
`define max(a,b) ((a) > (b) ? (a) : (b))
n = `max(p+q, r+s);
```

Using parameter

- Module parameters (`parameter`)
  - Constant values – cannot be modified at run time
  - Can be modified at compilation time to have values that are different from those specified in the declaration assignment → module customization
  - **Examples**

```
parameter msb = 7;
parameter [3:0] mux_selector = 0;
```

- Local parameters (**`localparam`**)
  - Cannot directly be modified by **`defparam`** or module instance parameter value assignments

Using parameter

- **Parameter & localparam**
  - **Examples**

```
module my_mem (addr, data);
    parameter addr_width = 16;
    parameter data_width = 8;
    localparam mem_size = 1 << addr_width;
...
endmodule

module top;
    ...
    my_mem #(12, 16) m(addr,data);
endmodule
```

Using task

function 과 비슷

- We can use task syntax to simplify the testbench

```
// We can use this @ adder testbench
task Test;
    input [31:0] in_ref;
    input [31:0] out_ref;
begin
    in = in_ref;
    #1
    if (out == out_ref)
        PASSED = PASSED + 1;
    else
        FAILED = FAILED + 1;
    #9
end
```

Test your module using task

```
`timescale 1ns / 100ps
module Adder_tb;
    // declare in/out ports for your module
    …

    // Task declaration
    …

    // set the values
    initial begin
        # 16 // wait for 16 ns

        Test(32'd1, 32'd1);
        Test(32'd2, 32'd3);
        Test(32'd5, 32'd8);
        …

    end
endmodule
```

There are some advanced stuffs

- genvar, include, …
- Search for these if you are interested!

Let’s dive into a complete example

101 Detector - example

- I want to implement a detector which outputs **'1'** if the input sequence is **'1'**, **'0'**, and **'1'**
  - Input: a sequence of binary numbers (1 bit per 1 clock)
  - output: 1 if 101 is detected, 0 otherwise
  - Example:
    - Input: 00100101010001110100
    - Output: 00000001010000000100

- Things you should consider:
  - Draw an FSM (it can be both Moore and Mealy machine)
  - Implement the FSM using Verilog

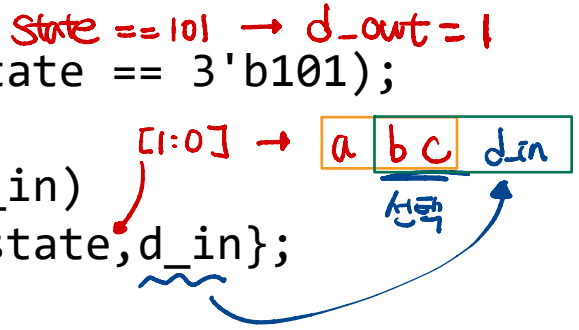
Specification

four Ports

- There are four input and output ports
  - `clk` → clock signal
  - `d_in` → input bit signal
  - `rst` → reset signal (the history is synchronously reset if the signal is one)
    - Synchronous: the history is reset if it the reset signal is one at the positive edge of the clock)
  - `d_out` → output bit signal if the history is 101
    - You can implement it in both Moore and synchronous Mealy machine

Moore-style implementation

```
`timescale 1ns / 100ps
module SequenceDetector (
    input clk,
    input rst,
    input d_in,
    output d_out);

    reg [2:0] state;
    reg [2:0] state_nxt;

    assign d_out = (state == 3'b101);

    always @(state, d_in)
        state_nxt = {state,d_in};

    always @(posedge clk) begin
        if (rst)    state <= 3'b000;
        else        state <= state_nxt;
    end
endmodule
```



(Synchronous) Mealy

```
`timescale 1ns / 100ps
module SequenceDetector (input clk, input rst, input d_in,
output reg d_out);
    reg [1:0] state;
    reg [1:0] state_nxt;
    reg [1:0] d_out_nxt;

    always @(state, d_in) begin
        state_nxt = {state[0],d_in};
        d_out_nxt = ((state == 2'b10) && d_in);
    end

    always @(posedge clk) begin
        if (rst)
            state <= 2'b00;
        else begin
            state <= state_nxt;
            d_out <= d_out_nxt;
        end
    end
endmodule
```

← 이전 state의 맨 끝자리만 가지고 온다

Testbench

```
`timescale 1ns / 100ps
module SequenceDetector_tb();

reg  clk, rst, d_in;
wire d_out;
SequenceDetector top (.clk(clk),
    .rst(rst),
    .d_in(d_in),
    .d_out(d_out));

// Clock signal generation
initial begin : CLOCK_GENERATOR
    clk = 1'b0;
    forever #5 clk = ~clk;
end
reg [19:0] in_seq   =
   20'b0010_1110_0010_1010_0100;
reg [19:0] ref_res  =
   20'b0010_0000_0010_1000_0000;
reg [19:0] test_res = 20'd0;
```

```
integer i;
initial begin
    rst  = 1'b1; d_in = 1'b0;
    #10; rst  = 1'b0;

    for (i = 0; i < 20; i = i + 1)
begin
        d_in = in_seq[i];
        #10; test_res[i] = d_out;
    end
    $display("test %s", (test_res
== ref_res) ? "passed" :
"failed");
    $finish;
end
endmodule
```

Testbench

```
`timescale 1ns / 100ps
module SequenceDetector_tb();

reg  clk, rst, d_in;
wire d_out;
SequenceDetector top (.clk(clk),
    .rst(rst),
    .d_in(d_in),
    .d_out(d_out));

// Clock signal generation
initial begin
    clk = 1'b0;
    forever #5 clk = ~clk;
end
…
```

Testbench

```
// Test
reg [19:0] in_seq   = 20'b0010_1110_0010_1010_0100;
reg [19:0] ref_res  = 20'b0010_0000_0010_1000_0000;
reg [19:0] test_res = 20'd0;

integer i;
initial begin
    rst  = 1'b1; d_in = 1'b0;
    #10; rst  = 1'b0;

    for (i = 0; i < 20; i = i + 1) begin
        d_in = in_seq[i];
        #10; test_res[i] = d_out;
    end
    $display("test %s", (test_res == ref_res) ? "passed" : "failed");
    $finish;
end
endmodule
```

Widely used syntax for debugging in Verilog!

- display & monitor (equal to the printf …)
  - https://www.chipverify.com/verilog/verilog-display-tasks
- task (equal to function when debugging input and output!)
  - https://www.chipverify.com/verilog/verilog-task