Introduction to Verilog

Suheon Bae (suheon.bae@snu.ac.kr)

Hunjun Lee (hunjunlee7515@snu.ac.kr)

Dongup Kwon (dongup@snu.ac.kr)

High Performance Computer System (HPCS) Lab.

Mar. 13, 2018

Project

- In this project, you will design and implement an **in-order pipelined CPU** using Verilog.
- Tools
  - Hardware description language (HDL): Verilog
  - Design & simulation tool: Xilinx Vivado Design Suite

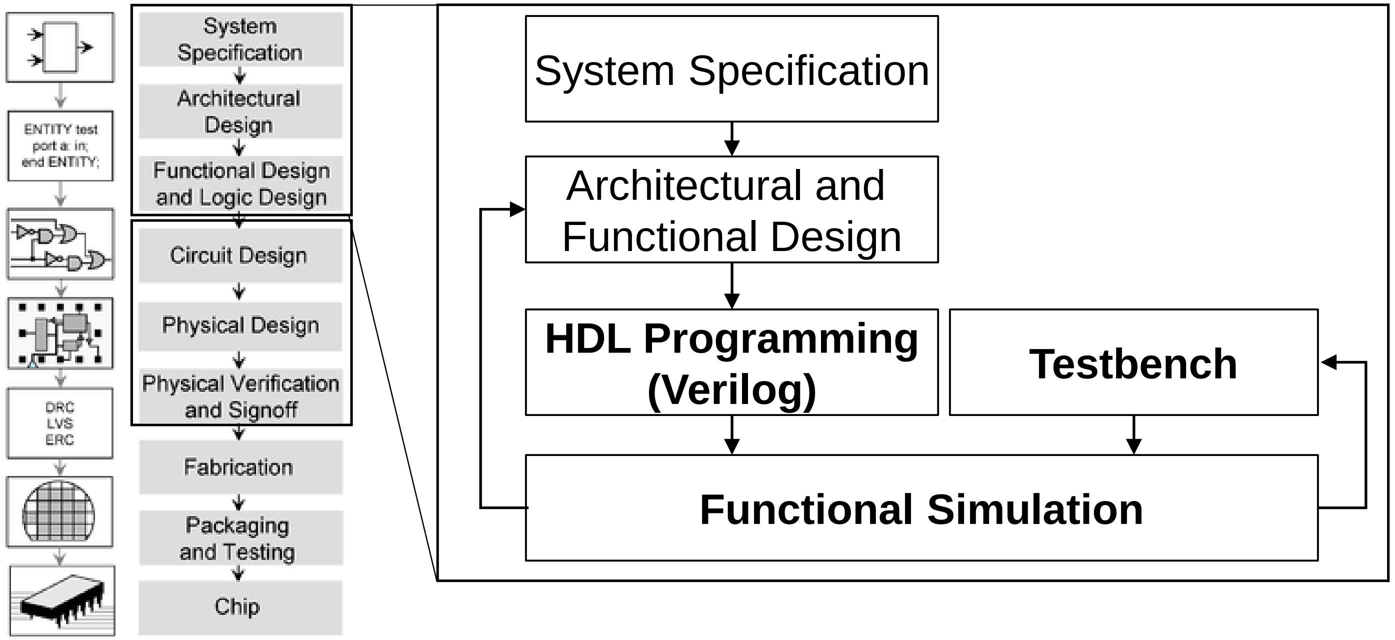
Overview

- In this project, you will learn **how to describe digita l circuits in Verilog** and **how to use design tools** *(e. g., Vivado Design Suite).*
- All the assignments are related; that is, you need to complete all previous steps to do next assignment.
- Your assignment for this week is implementing a si mple 16-bit ALU. This will also be used in the makin g of a CPU.
- **Goals**
    - Understand how to describe digital circuits in Verilog
    - Implement an ALU (i.e., a basic functional unit in CPUs)

Hardware Description Language (HDL)

HDL and Verilog

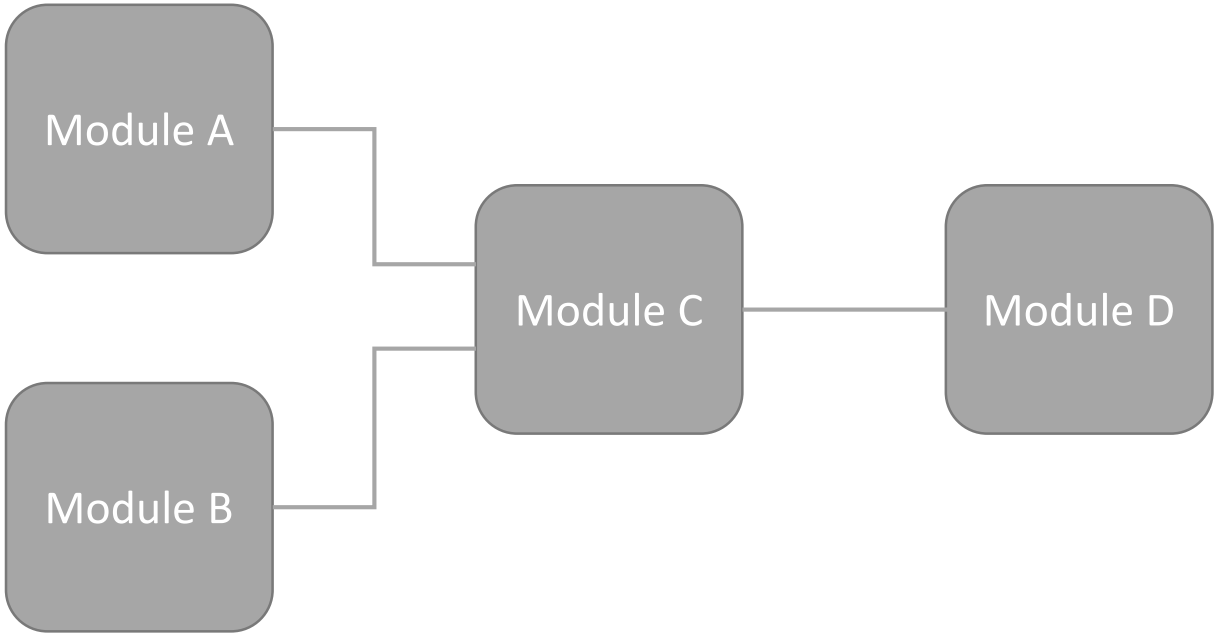
- **HDL**: A formal notation designed to describe *electronic circuits*
- **Verilog**: Most widely used and well-supported HDL



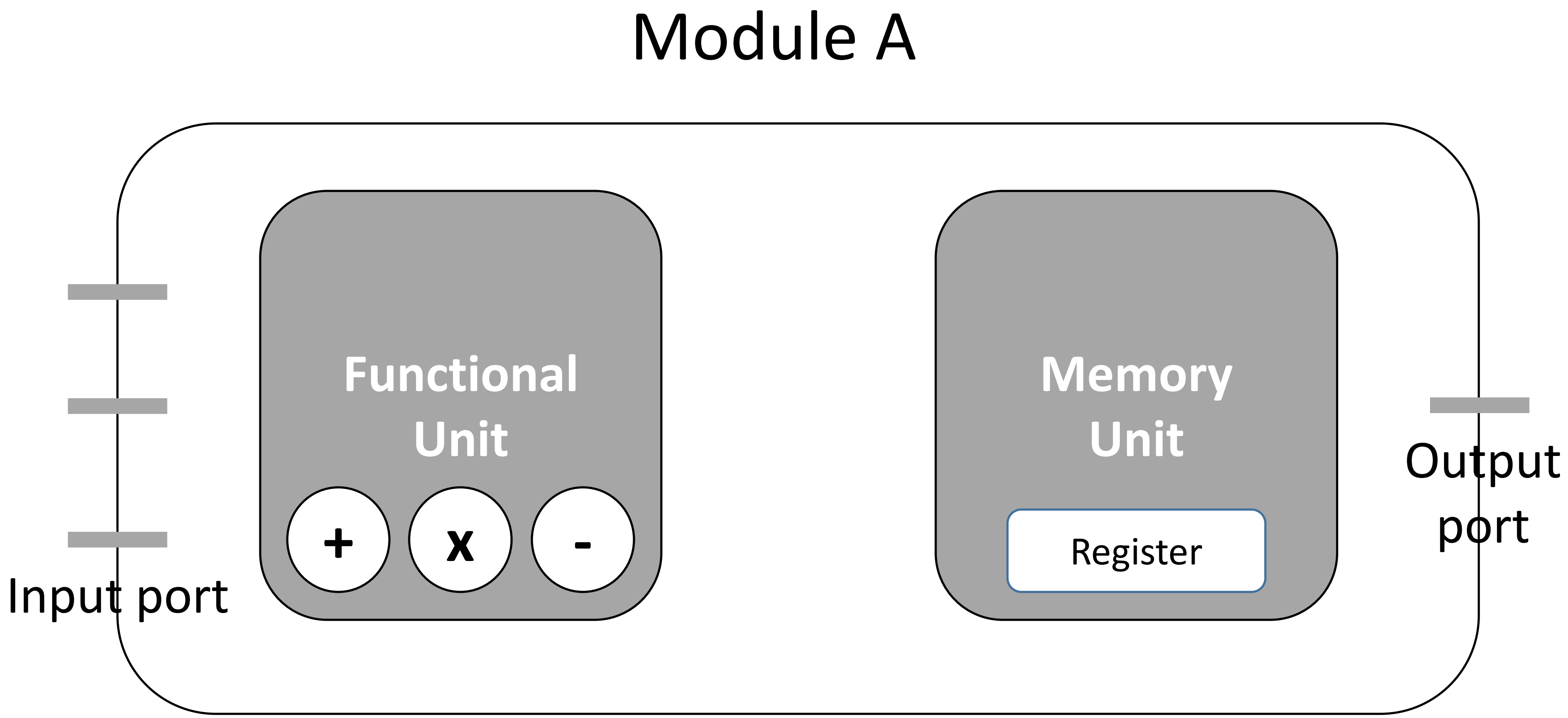
https://commons.wikimedia.org/w/index.php?curid=18276753

Verilog Programming

How to describe a digital circuit?



What’s inside a module?



Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- Timescale
- Module
  - Ports
  - Data types
  - Operators
  - Behavior models
    - Assignments
    - If/case/loop statements
    - Timing / event control
    - Structured procedures

Structure

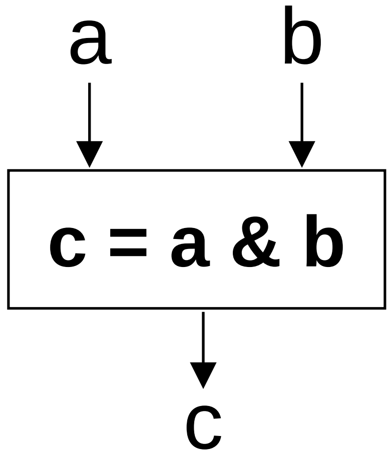
Timescale

- **`timescale** time_unit / time_precision
  - Example: **`timescale** 1 ns / 100 ps
- **time_unit**: the unit of measurement
  - Example: #100 = 100 * 1ns delay
- **time_precision**: the unit of simulation
- time_precision cannot specify a longer unit of time than time_unit.

Module

- A unit of hierarchical hardware structure
  - High-level modules create the instances of low-level modules and communicate with them through ports.
  - A **top-level** *module* is the highest-level module.
- Interfaces + behavior models

```
module AND (a, b, c);
        input a;
        input b;            } Interfaces (ports)
        output c;
        assign c = a & b;   } Behavioral models
endmodule
```



Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- ~~Timescale~~
- Module
  - Ports
  - Data types
  - Operators
  - Behavior models
    - Assignments
    - If/case/loop statements
    - Timing / event control
    - Structured procedures

Structure

Port

- A means of interconnecting modules
  - Optional, but usually necessary
- Port declaration: **input**, **output**, **inout** (bidirectional)

```
module AND (
    input a,
    input b,
    output c
    );
    assign c = a & b;
endmodule
```

Data Type

- Data types is designed to represent the data storage and transmission elements.
- Values
  - **0**: a logic zero
  - **1**: a logic one
  - **x**: an unknown logic value
  - **z**: a high-impedance state
- **net** and **variable** data-type groups
  - **net**: a representation of physical connections (e.g., **wire**)
  - **variable**: an abstraction of data storage elements (e.g., **reg**)
- We will use only **wire** and **reg** data types

Data Type - Wire

- It defines a physical connection with a continuo us assignment.
  - **wire** data type is the most representative.
- Example

```
wire [7:0] c;
assign c = a & b;
```

  - Then **c** is “physically connected” to the result of the “AND gate”.
  - **wire** data type usually describes a combinational cir cuit.

Data Type - Reg

- It is an abstraction of “storage elements” which can be read and written.
  - **reg** data type is the most representative. We use only this.
- Example

```
reg [7:0] c;
always c = a & b;
```

  - It is mainly used in a sequential circuit.

Syntax

- Comments : // or /* */ (Similar to C language)
- Identifier : [a-zA-Z_]([a-zA-Z0-9_$])+ (Up to 10 24 chars)
- Integer number : <size>'<radix><value>
  - <size> : # of bits
  - <radix> : h(hex), d(decimal), o(octal), b(binary)
  - <value> : Depend on the radix
    - X : “Unknown”, Z : “High Impedance”, _ : (Ignored)
  - Example: 8’b0, 16’d7, 32’hFF

Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- ~~Timescale~~
- Module
    - ~~Ports~~
    - ~~Data types~~
    - Operators
    - Behavior models
        - Assignments
        - If/case/loop statements
        - Timing / event control
        - Structured procedures

Structure

Operators

- Arithmetic
  - Binary : +, -, *, /, %
  - Unary : +, -
- Relational
  - Binary : <, >, <=, >=
- Equality
  - Binary : ===, !==
    - (comparing include x & z)
  - Binary : ==, !=
    - (returns x if a 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

Addressing

- Vector bit-select and part-select addressing
  - **Examples**
    - `**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)`

Addressing

- Vector bit-select and part-select addressing
  - **Examples**
    - `**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`

Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- ~~Timescale~~
- Module
  - ~~Ports~~
  - ~~Data types~~
  - ~~Operators~~
  - Behavior models
    - Assignments
    - If/case/loop statements
    - Timing / event control
    - Structured procedures

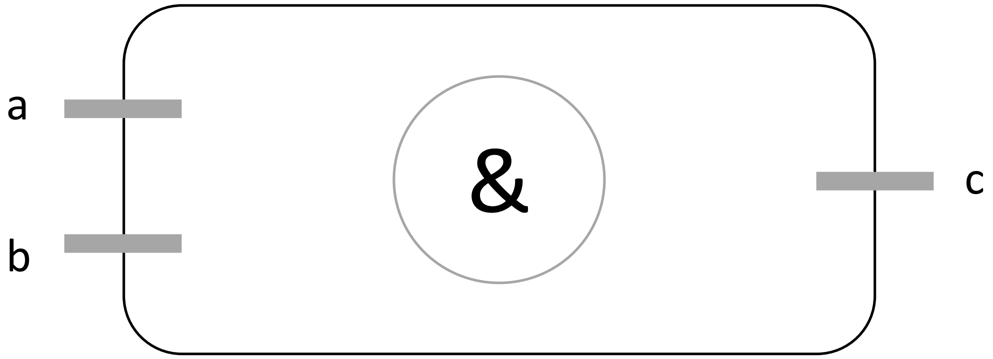
Structure

Behavioral Model

- Continuous assignment
- Procedural assignment
- if statement
- Case statement
- Loop statement
- Timing controls
- Structured procedures

Continuous Assignment

- It drives values onto **wire** or defines a connection for **wire**.
    - **wire** c**;**
    - **assign** c = a & b; // the type of c should be the net.
- With a declaration of wire
    - **wire** c = a & b;



Procedural Assignments

- It puts values in **reg**, and **reg** holds the value until the next procedural assignment to that variable.
- It occurs within procedures such as **always** and **initial**.
  - **reg** x;
  - **always** x = 1; // the data type of x should be **reg**.

  - **wire** x;
  - **always** x = 1; //this kind of assignment to wire type will incur error!! Do not do this!!!!

Non-blocking & Blocking Assignment

- The two mechanism of *procedural assignments*
- It is really important but confusing, so you have to understand it very well.

Blocking assignment (=)

- It defines the execution order of statements.
- **Examples**

```
• reg a, b;
• always begin
•     a = 0;
•     b = 0;
• end
```

Non-blocking assignment (<=)

- It doesn’t specify the execution order of statements.
- Assignments within the same time step can be made without regard to order or dependence upon each other.
- **Examples**
  - **`reg`** `a, b;`
  - **`always begin`**
  - `    a <= 0;`
  - `    b <= 0;`
  - **`end`**

Blocking vs. Non-blocking

```
module example ();
   reg a, b, c;
   initial begin
      b ≪ a;
      c ≪ b;
   end
endmodule
```

≪ is blocking (=)

	a	b	c
Before	0	1	x
After	0	0	0

≪ is non-blocking (<=)

Cycle	a	b	c
Before	0	1	x
After	0	0	1

Mixing Blocking & Non-blocking

- Don’t even think about mixing them insdie a module
- There is no syntactic error, but it is hard to predict the consequence.
- **`reg`** – procedural assignment
  - Edge-sensitive storage elements – non-blocking assignment (ex D flip flops ..)
  - Combinational logics – blocking assignment (ex ALU ..)

Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- ~~Timescale~~
- Module
  - ~~Ports~~
  - ~~Data types~~
  - ~~Operators~~
  - Behavior models
    - ~~Assignments~~
    - If/case/loop statements
    - Timing / event control
    - Structured procedures

Structure

If statement

- It executes a statement conditionally.

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

Case statement

- It executes a statement conditionally.

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

Loop statement - repeat

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

$\equiv$

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

Loop statement - forever

- Same as repeat (∞)

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

≡

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

Loop statement – while, for

- It describes a repetition task, but sometime it cannot be implemented as a real circuit.
- The typical usage is an array initialization.

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

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

Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- ~~Timescale~~
- Module
  - ~~Ports~~
  - ~~Data types~~
  - ~~Operators~~
  - Behavior models
    - ~~Assignments~~
    - ~~If/case/loop statements~~
    - Timing / event control
    - Structured procedures

Structure

Timing Controls - Delay & Event

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

Delay control

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

Delay control

- 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.

Event control

- Event control
  - Synchronize a procedural statement with a value change on a `wire` or `reg` or the occurrence of a declared event
  - **Examples**
    - `@r rega = reab;`
    - `@(posedge clock) rega = regb;`
    - `always @(negedge clock) rega = regb;`

    - `always @(a, b, c, d, e)`
    - `always @(posedge clk, negedge rstn)`
    - `always @(a or b, c, d or e)`

Event control

- 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`**

Event control

- We can represent a complex combinational circuit with an event.

```
module example ();
    wire a, b, c;
    assign c = ~(a & b);
endmodule
```

$\equiv$

```
module example ();
    wire a, b;
    reg c;
    always @(a or b) begin
        c = ~(a & b);
    end
endmodule
```

Event control

- How to describe edge-storage elements
  - Non-blocking assignments and `posedge` events
  - **Examples**
    - **`always @(posedge clk) begin`**
    - `    ff1 <= ff1_next;`
    - `    ff2 <= ff2_next;`
    - `    …`
    - **`end`**
- Can be used to design sequential circuit

Two-input AND gate

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Source code

- ~~Timescale~~
- Module
  - ~~Ports~~
  - ~~Data types~~
  - ~~Operators~~
  - Behavior models
    - ~~Assignments~~
    - ~~If/case/loop statements~~
    - ~~Timing / event control~~
    - Structured procedures

Structure

Structured Procedures

- **always**, **initial**
- They are necessary for designing sequential circuit.

Structured Procedures - initial

- It is executed once at the beginning.

```
module example ();
        reg clk, x;
        initial clk <= 0;
        initial x <= 1;
endmodule
```

$\equiv$

```
module example ();
        reg clk, x;
        initial begin
                clk <= 0;
                x <= 1
        end
endmodule
```

- It initializes reg clk and x.
- It can involve multiple statements with begin … end.

Structured Procedures - always

- **always** is equal to the infinite repetition of a initial block.

```
module example ();
	reg clk;
	initial clk <= 0;
	initial begin
		#50 clk <= ~clk;
		#50 clk <= ~clk;
		#50 clk <= ~clk;
		// infinitely repeated
	end
endmodule
```

$\equiv$

```
module example ();
	reg clk;
	initial clk <= 0;
	always begin
		#50 clk <= ~clk;
	end
endmodule
```

Some more advanced / additional stuffs from now on

Overview

- **Compiler directives**
  - `~~timescale,~~ `include, `define
- **Data types**
  - ~~wire, reg~~, arrays, parameters
- **Expressions**
  - ~~Operators, operands~~
- ~~**Assignments**~~
  - ~~Continuous assignments, procedural assignments, blocking/non-blocking assignment (procedural)~~
- ~~**Behavioral modeling**~~
  - ~~Structured procedures, conditional statement (if-lese), case statement, looping statements~~
  - ~~Timing control, event control, event-expression list~~
- **Hierarchical structures**
  - ~~Modules, ports~~, port connection

Compiler directives (1)

- Control the compilation of a Verilog description
  - All Verilog compiler directives are preceded by the (`) grave accent mark.
  - It is recommended that they appear outside a module declaration.
- **`` `timescale ``**
  - Specify the time unit and time precision
  - **`` `timescale ``** `time_unit / time_precision`

Compiler directives (2)

- **`` `include ``**
  - Insert the entire contents of a source file in another file during compilation
  - **`` `include ``** `filename`
- **`` `define ``**
  - Represent commonly used pieces of text
  - **`` `define ``** `macro_name macro_text`
  - **Examples**
    - **`` `define ``** `wordsize 8`
    - **`reg`** ``[`wordsize-1:0] data;``

Compiler directives (3)

- **`` `define ``**
  - **`` `define ``** `macro_name macro_text`
  - **More examples**
    - **`` `define ``** `first_half “start of string`
    - **`$display`**`` (`first_half end of string”); ``

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

Overview

- **Compiler directives**
  - `~~timescale, `include, `define~~
- **Data types**
  - ~~wire, reg~~, arrays, parameters
- **Expressions**
  - ~~Operators, operands~~
- ~~**Assignments**~~
  - ~~Continuous assignments, procedural assignments, blocking/non-blocking assignment (procedural)~~
- ~~**Behavioral modeling**~~
  - ~~Structured procedures, conditional statement (if-lese), case statement, looping statements~~
  - ~~Timing control, event control, event-expression list~~
- **Hierarchical structures**
  - ~~Modules, ports~~, port connection

Data types (1)

- Arrays
  - Group elements of the declared element type into multidimensional objects
- Memories – a one-dimensional array with elements of type **`reg`**
- **Examples**
  - **`reg`** `[7:0] mem[255:0];`
  - **`reg`** `arrayA[7:0][255:0];`
  - **`reg`** `[7:0] regA ≠` **`reg`** `mem[7:0]`

Data types (2)

- Assignment to array elements
- **More examples**
  - `**reg** [7:0] mem[255:0];`
  - `mem = 0; // Illegal syntax – attempt to write entire array`
  - `mem[1] = 0; // Assigns 0 to the second element of mem`

Data types (3)

- Module parameters (**`parameter`**)
  - Constant values – cannot be modified <u>at run time</u>
  - Can be modified <u>at compilation time</u> to have values that are different from those specified in the declaration assignment $\rightarrow$ 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

Data types (4)

- **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
```

Overview

- **Compiler directives**
  - ~~`timescale, `include, `define~~
- **Data types**
  - ~~wire, reg, arrays, parameters~~
- **Expressions**
  - ~~Operators, operands~~
- ~~**Assignments**~~
  - ~~Continuous assignments, procedural assignments, blocking/non-blocking assignment (procedural)~~
- ~~**Behavioral modeling**~~
  - ~~Structured procedures, conditional statement (if-lese), case statement, looping statements~~
  - ~~Timing control, event control, event-expression list~~
- **Hierarchical structures**
  - ~~Modules, ports~~, port connection

Hierarchical structures (1)

- Module: a unit of hierarchical hardware structure
  - A top-level module is the highest-level module.
  - Interfaces + behavior models
- Module instantiation
  - **Examples**

    ```
    module ffnand (q, qbar, preset, clear);
        output q, qbar;
        input preset, clear;
    endmodule

    // Higher level module
    ffnand ff (out1, out2, in1, in2);
    ```

Hierarchical structures (2)

- Parameter value assignment by ordered list
  - **Examples**

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

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

Hierarchical structures (3)

- Connecting module instance ports by ordered list
  - **Examples**

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;
    modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
    inout wa, wb;
    input c, d;
    …
endmodule
```

Hierarchical structures (4)

- Connecting module instance ports by name
  - **Examples**

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;
    modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule

module modB(wa, wb, c, d);
    inout wa, wb;
    input c, d;
    …
endmodule
```

Now some more practical tips

How to design digital circuit?

- Combinational circuit
    - Computation without memory elements
  - Continuous assignment
  - Blocking assignment

- Sequential circuit
    - Think of it as combinational circuit + memory
  - Non-blocking assignment

Combinational circuit -1

- Continuous assignment

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    assign Q = A & B

endmodule
```

Combinational circuit -2

- Blocking assignment

```
`timescale 1ns / 100ps

module AND (
    input [3:0] A,
    input [3:0] B,
    output [3:0] Q
);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Sequential circuit

- Non blocking assignment

```
`timescale 1ns / 100ps

module Dflipflop (
    input [3:0] A_in,
    input clk,
    output [3:0] A_out
);

    reg [3:0] A_out;

    always @(posedge clk)begin
        A_out <= A_in;
    end
endmodule
```

Assignment

- There will be assignment due next week
    - March, 20th
- You'll make 16bit ALU for your CPU
- You'll learn how to implement combinational logic in Verilog
- Detailed description of the project is uploaded in etl

Reference

- IEEE Standard for Verilog Hardware Description Language, IEEE Computer Society