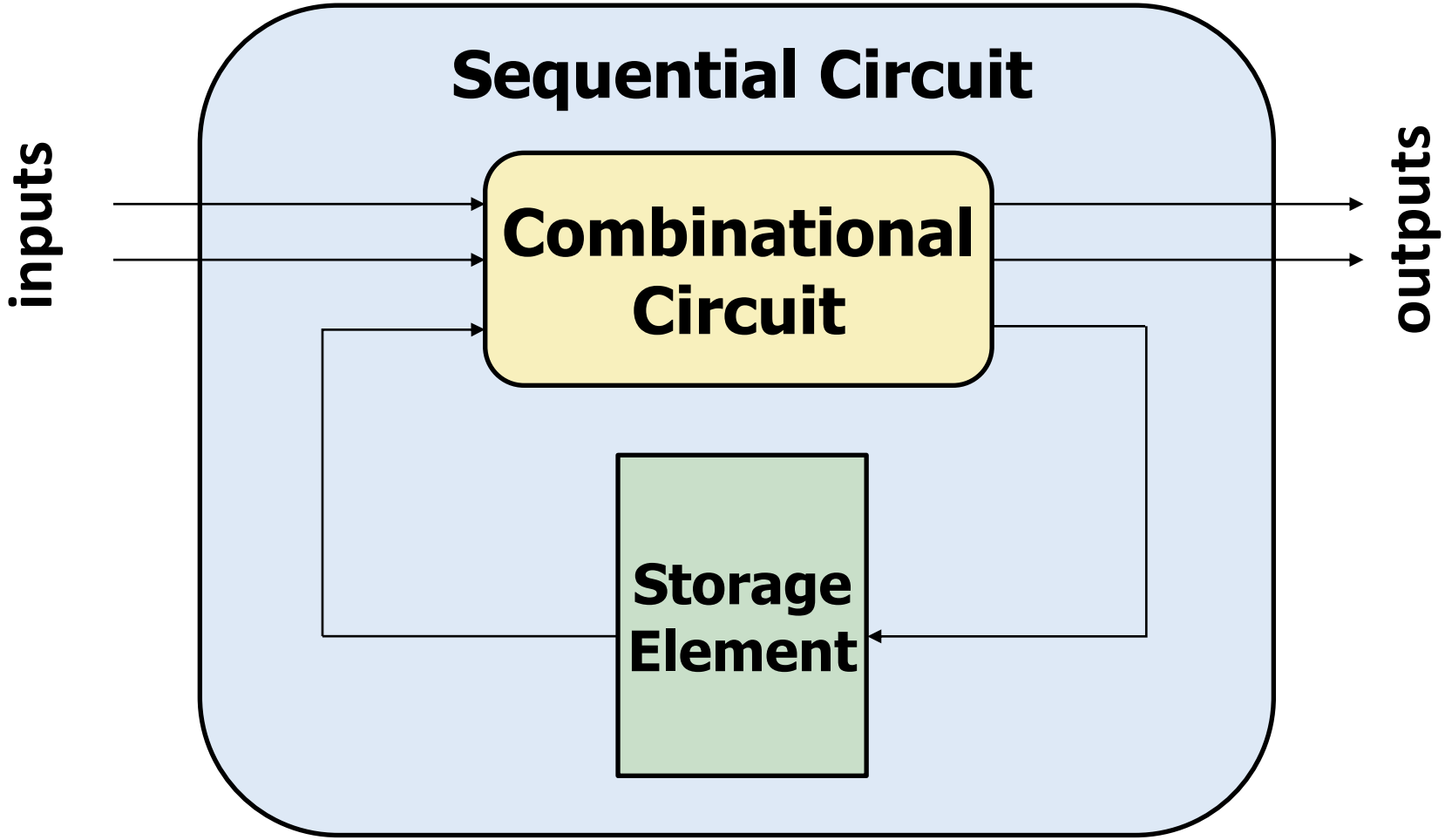
Hardware Description Languages and Verilog

陈刚
副教授
人工智能与无人系统研究所
计算机学院
中山大学

Implementing Sequential Logic Using Verilog

Combinational + Memory = Sequential



Sequential Logic in Verilog

- Define blocks that have memory
  - *Flip-Flops*, *Latches*, *Finite State Machines*

- Sequential Logic state transition is triggered by a "CLOCK" signal
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of signal

- Combinational HDL constructs are **not** sufficient to express sequential logic
  - We need **new constructs**:
    - `always`
    - `posedge/negedge`

The “always” Block

```
always @ (sensitivity list)
      statement;
```

Whenever the event in the sensitivity list occurs,
the statement is executed

Example: D Flip-Flop

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                    // pronounced “q gets d”

endmodule
```

- posedge defines a rising edge (transition from 0 to 1).
- Statement executed when the clk signal rises (posedge of clk)
- Once the clk signal rises: the value of d is copied to q

Example: D Flip-Flop

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

- assign statement is **not** used within an always block
- <= describes a non-blocking assignment
  - We will see the difference between blocking assignment and non-blocking assignment soon

Example: D Flip-Flop

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

- Assigned variables need to be declared as reg
- The name reg does not necessarily mean that the value is a register (It could be, but it does not have to be)
- We will see examples later

Asynchronous and Synchronous Reset

- Reset signals are used to initialize the hardware to a known state
  - Usually activated at system start (on power up)

- **Asynchronous Reset**
  - The reset signal is sampled independent of the clock
  - Reset gets the highest priority
  - Sensitive to glitches, may have metastability issues
    - Will be discussed in Lecture 8

- **Synchronous Reset**
  - The reset signal is sampled with respect to the clock
  - The reset should be active long enough to get sampled at the clock edge
  - Results in completely synchronous circuit

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input              clk,
                input              reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
       if (reset == 0) q <= 0;   // when reset
       else            q <= d;   // when clk
    end
endmodule
```

- In this example: two events can trigger the process:
  - A ***rising edge*** on clk
  - A ***falling edge*** on reset

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input              clk,
                input              reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == 0) q <= 0;   // when reset
      else            q <= d;   // when clk
    end
endmodule
```

- For longer statements, a begin-end pair can be used
  - To improve readability
  - In this example, it was not necessary, but it is a good idea

D Flip-Flop with **Asynchronous** Reset

```
module flop_ar (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == 0) q <= 0;    // when reset
      else            q <= d;    // when clk
    end
endmodule
```

- First reset is checked: if reset is 0, q is set to 0.
  - This is an asynchronous reset as the reset can happen independently of the clock (on the negative edge of reset signal)
- If there is no reset, then regular assignment takes effect

D Flip-Flop with **Synchronous** Reset

```
module flop_sr (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk)
    begin
       if (reset == '0') q <= 0;   // when reset
       else              q <= d;   // when clk
    end
endmodule
```

- The process is sensitive to only clock
  - Reset ***happens only*** when the ***clock rises***. This is a synchronous reset

D Flip-Flop with Enable and Reset

```
module flop_en_ar (input             clk,
                   input             reset,
                   input             en,
                   input      [3:0] d,
                   output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == '0') q <= 0;   // when reset
      else if (en)      q <= d;   // when en AND clk
    end
endmodule
```

- A flip-flop with **enable** and **reset**
  - Note that the en signal is ***not*** in the *sensitivity list*
- q gets d only when clk is rising **and** en is 1

Example: D Latch

```
module latch (input             clk,
              input       [3:0] d,
              output reg  [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;       // latch is transparent when
                           // clock is 1
endmodule
```

Summary: Sequential Statements So Far

- Sequential statements are within an `always` block
- The sequential block is triggered with a change in the sensitivity list
- Signals assigned within an **always** must be declared as `reg`
- We use `<=` for (non-blocking) assignments and do not use `assign` within the always block.

Basics of **always** Blocks

```
module example (input             clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;          // standard wire
  reg  [3:0] special;         // assigned in always

  always @ (posedge clk)
    special <= d;             // first FF array

  assign normal = ~ special; // simple assignment

  always @ (posedge clk)
    q <= normal;              // second FF array
endmodule
```

You can have as many always blocks as needed

Assignment to the same signal in different always blocks is not allowed!

Why Does an **always** Block Remember?

```
module flop (input             clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

- This statement describes what happens to signal q
- … but what happens when the clock is not rising?
- The value of q is preserved (remembered)

An **always** Block Does **NOT** Always Remember

```
module comb (input             inv,
             input      [3:0] data,
             output reg [3:0] result);

  always @ (inv, data)       // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

- This statement describes what happens to signal result
  - When `inv` is 1, result is `~data`
  - When `inv` is not 1, result is `data`
- The circuit is combinational (no memory)
  - result is assigned a value in all cases of the if .. else block, always

always Blocks for Combinational Circuits

- An always block defines combinational logic if:
  - All outputs are always (**continuously**) updated

  1. All right-hand side signals are in the sensitivity list
     - You can use `always @*` for short

  2. All left-hand side signals get assigned in every possible condition of if .. else and case blocks

- It is easy to make mistakes and unintentionally describe memorizing elements (latches)
  - Vivado will most likely warn you. Make sure you check the warning messages

- **Always** blocks allow powerful combinational logic statements
  - `if .. else`
  - `case`

The **always** Block is **NOT** Always Practical/Nice

```
reg  [31:0] result;
wire [31:0] a, b, comb;
wire        sel,

always @ (a, b, sel)    // trigger with a, b, sel
   if (sel) result <= a; // result is a
   else     result <= b; // result is b

assign comb = sel ? a : b;
```

- Both statements describe the **same** multiplexer
- In this case, the always block is more work

always Block for Case Statements (Handy!)

```
module sevensegment (input      [3:0] data,
                     output reg [6:0] segments);

  always @ ( * )                    // * is short for all signals
    case (data)                     // case statement
      4'd0: segments = 7'b111_1110;  // when data is 0
      4'd1: segments = 7'b011_0000;  // when data is 1
      4'd2: segments = 7'b110_1101;
      4'd3: segments = 7'b111_1001;
      4'd4: segments = 7'b011_0011;
      4'd5: segments = 7'b101_1011;
      // etc etc
      default: segments = 7'b000_0000; // required
    endcase

endmodule
```

Summary: **always** Block

- `if .. else` can only be used in always blocks

- The always block is combinational only if all regs within the block are always assigned to a signal
  - Use the `default` case to make sure you do not forget an unimplemented case, which may otherwise result in a latch

- Use `casex` statement to be able to check for don't cares

Non-Blocking and Blocking Assignments

Non-blocking (<=)

```
always @ (a)
begin
   a <= 2'b01;
   b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- All assignments are made at the end of the block
- All assignments are made in parallel, process flow is not-blocked

Blocking (=)

```
always @ (a)
begin
   a = 2'b01;
// a is 2'b01
   b = a;
// b is now 2'b01 as well
end
```

- Each assignment is made immediately
- Process waits until the first assignment is complete, it blocks progress

Rules for Signal Assignment

- Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model synchronous sequential logic

```
always @ (posedge clk)
      q <= d; // non-blocking
```

- Use continuous assignments (`assign`) to model simple combinational logic

```
assign y = a & b;
```

Rules for Signal Assignment (Cont.)

- Use `always @ (*)` and blocking assignments (=) to model more complicated combinational logic.

- You cannot make assignments to the same signal in more than one always block or in a *continuous assignment*

```
always @ (*)
    a = b;

always @ (*)
    a = c;
```

```
always @ (*)
    a = b;

assign a = c;
```