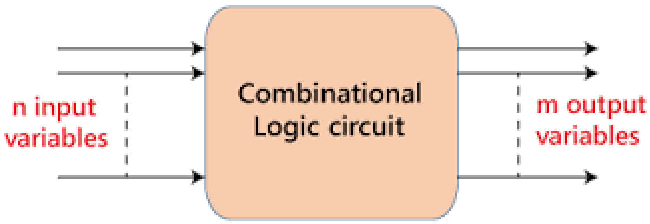
Lab 0 introduction

Index

The difference between Combinational circuit and Sequential circuit

Combinational Circuit



Block diagram of a combinational circuit

- Take some combinational logic circuits for example. Logic gate(e.g. and, or...), MUX, Decoder, Selector...
- Coding example:

```
//assign
wire a,b,c;
assign a = b & c;

//always block
reg a; // this is not a DFF
wire b,c;
always @(*) begin
    a = b & c;
end
```

Sequential Circuit

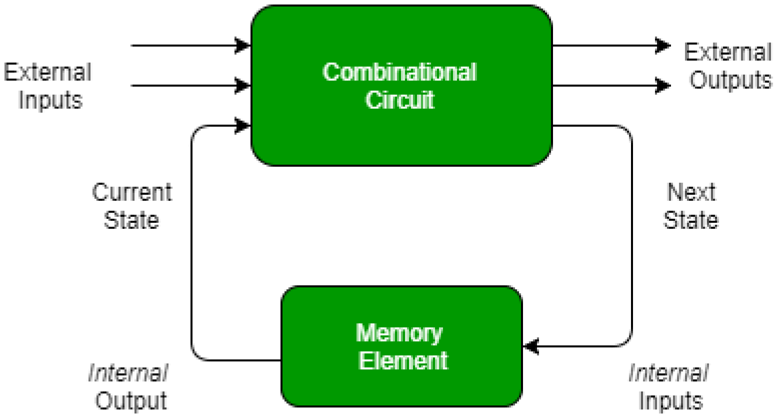


Figure: Sequential Circuit

- Sequential elements are used for storage
- Coding example:

```
//synchronous negedge reset
reg a; // this is a DFF
wire b;
always @(posedge clk) begin
    if(!rst_n) a <= 0;
    else a <= b;
end

//asynchronous posedge reset
reg a; // this is a DFF
wire b;
always @(posedge clk or posedge rst) begin
    if(rst) a <= 0;
    else a <= b;
end
```

Coding Style

Naming conventions

- rst for reset, clk for clock
- _n for active-low
- Using _ rather than - in naming reg/wire
- Naming must be meaningful
- Naming example: I want to define a flag that represents reg A larger than reg B, it can call A_lr_B or A_larger_B.
- uppercase letters and lowercase letters are different in Verilog.
- You can use uppercase letters or _ to separate reg/wire naming variables. For example, current_state or currentState.
- Use uppercase letters for names of constants and user-defined types.
  e.g. `define BUS_LENGTH 32 or localparam BUS_LENGTH = 32
- Use lowercase letters for all signals, variables, and ports.
  e.g. wire clk, rst...
- Other naming conventions
  *_r: register type(DFF)
  *_w: wire type or reg type but represent combinational logic.

Reg/Wire declaration

- Using little-endian for vector initialization, for example, reg [7:0] counter or wire [7:0] adderResult.

- Using big-endian for multi-bit array declaration, for example, reg [31:0] mem [0:31].

Coding Precautions

- Adding some proper comments or documentation for recording.
- Avoid using both edges of a clock due to the reason that it is difficult for DFT(Design-For Testability) process.
- Avoid tri-state buses
- Codes must be synthesizable. For example
  assign, always block, called sub-modules, if-else if-else, cases, parameter, operators

- *Bitwise operators* — perform operations on each bit of a vector
  `~  &  |  ^  ^~  ~^`
- *Unary reduction operators* — collapse a vector to a 1-bit result
  `~  &  |  ^  ~&  ~|  ^~  ~^`
- *Logical operators* — return a true/false result, based on a logical test
  `&&  ||`
- *Equality, relational operators* — return true/false, based on a comparison
  `==  !=  <  <=  >  >=`
- *Shift operators* – shift the bits of a vector left or right
  `<<  <<<  >>  >>>`
- *Concatenate operators* — join multiple expressions into a vector
  `{ }  {n{ }}`
- *Conditional operator* — selects one expression or another
  `?:`
- *Arithmetic operators* — perform integer and floating point math
  `+  -  *  /  %  **`

- Data has to be described in one always block, for example

```
//multiple source drive is not allow
always @(posedge clk) begin
    if(!rst_n) out_r <= 'd0;
    else out_r <= out_r + 'd1;
end
always @(posedge clk) begin
    if(ready) out_r <= 'd1;
end

//correct
always @(posedge clk) begin
    if(!rst_n) out_r <= 'd0;
    else if(ready) out_r <= 'd1;
    else out_r <= out_r + 'd1;
end
```

- Only use "<=" when you are writing sequential blocks, and do not use "<=" and "=" in one always block.
- Avoid assigning unknown or high impedance values in your code.

- Bit width must be matching when you are using an assigned statement. For example

```
wire [3:0] a;
wire [2:0] b;
assign a = b // this is not allowed
```

- Avoid combination feedback circuits, for example

```
//wire feedback is not allowed
wire [1:0] a = a + 'd1;

//correct
reg [1:0] a_r;
wire [1:0] a = a_r + 'd1;
always @(posedge clk) begin
    if(!rst_n) a_r <= 'd0;
    else a_r <= a;
end
```

- Suggest using only a variable in one always block.

```
reg [1:0] a_r,b_r;
// bad
always @(posedge clk) begin
    if(!rst_n) begin
        a_r <= 'd0;
        b_r <= 'd0;
    end
    else if(trigger_1) b_r <= 'd1;
    else a_r <= 'd2;
end

// good
always @(posedge clk) begin
    if(!rst_n) begin
        a_r <= 'd0;
    end
    else a_r <= 'd2;
end
always @(posedge clk) begin
    if(!rst_n) begin
        b_r <= 'd0;
    end
    else if(trigger_1) b_r <= 'd1;
end
```

- Suggest combinational and sequential logic separating.

```
// bad
always @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    cur_state <= IDLE;
  end
  else begin
    if     (cal_start) cur_state <= CAL;
    else if(cal_done)  cur_state <= OUT
    else if(out_done)  cur_state <= IDLE;
  end
end

//good
always @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    cur_state <= IDLE;
  end
  else begin
    cur_state <= next_state;
  end
end
always @(*) begin
  case(curr_state)
    IDLE : next_state = cal_start ? CAL  : IDLE;
    CAD  : next_state = cal_done  ? OUT  : CAD;
    OUT  : next_state = out_done  ? IDLE : OUT;
    default :  next_state = IDLE;
end
```

Latch

- Avoid using Latch in your code. For example

1. Using case statements without default declaration in combination circuit.
2. Using if-else if-else statement without else in combination circuit.

```
// case 1 : lack of else
always @(*) begin
    if(m==2'd0) out_w = 2'd0;
    else if(m==2'd1) out_w = 2'd1;
end

// case 2 : lack of default
always @(*) begin
    case(m)
        2'd0: out_w = 2'd0;
        2'd1: out_w = 2'd1;
end

// case 1 : correct
always @(*) begin
    if(m==2'd0) out_w = 2'd0;
    else if(m==2'd1) out_w = 2'd1;
    else out_w = 2'd2;
end

// case 2 : correct
always @(*) begin
    case(m)
        2'd0: out_w = 2'd0;
        2'd1: out_w = 2'd1;
        default: out_w = 2'd2;
end
```

Reset

- Remember to reset all storage elements

```
This can help you avoid accepting unknown signals.
```

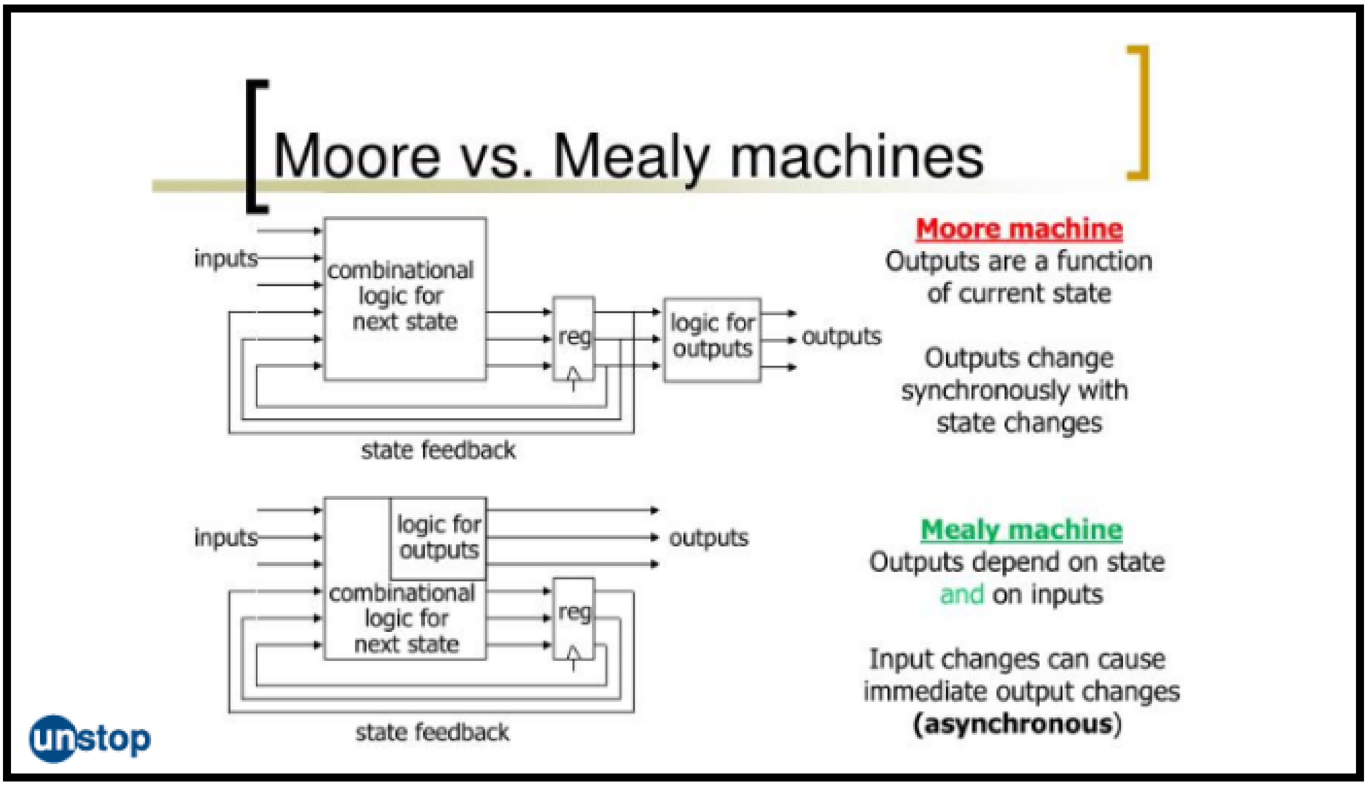
- Some Poor coding style example

```
//poor example
always @(posedge clk) begin
    if(!rst_n || !a) sig_r <= 'd0;
    else sig_r <= a;
end

//suggest example
always @(posedge clk) begin
    if(!rst_n) sig_r <= 'd0;
    else if(!a) sig_r <= 'd0';
    else sig_r <= a;
end
```

FSM

- Mealy_vs_Moore



Coding Example:

```
//current state logic
always @(posedge d_clk or negedge rst_n) begin
    if(!rst_n) begin
        curr_state <= IDLE;
    end
    else begin
        curr_state <= next_state;
```

```
    end
end
//mealy machine next state logic
always @(curr_state or input_value) begin
    case (curr_state)
        IDLE : next_state = input_value[0] ? CAL : IDLE;
        CAL  : next_state = input_value[1] ? ...;
        DONE : next_state = DONE;
        default : ...;
    endcase
end
//Moore machine next state logic
always @(curr_state) begin
    case (curr_state)
        IDLE : next_state = trigger_1 ? CAL : IDLE;
        CAL  : next_state = trigger_2 ? ...;
        DONE : next_state = DONE;
        default : ...;
    endcase
end
```

Introduce 3 level circuit

- Behavioral level
- Dataflow level
- Gate level or Structural level

Behavioral level

1. initial block

- Usually used for writing testbench.
- non synthesizable

2. always block

- Used for reg type variable
- Describing sequential circuit or combinational circuit.

Dataflow level

```
wire [1:0] a;
wire b,c;
//case 1
assign a = {b,c};//concatenate
//case 2
assign a = {b,1'b0};//concatenate
//case 3
assign a = {2{1'b0}};//concatenate
```

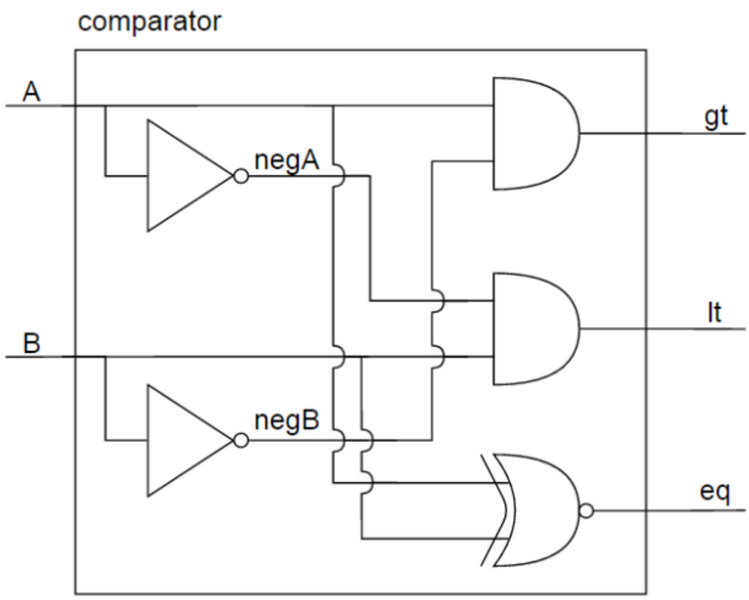
Gate level or Structural level

```
module comparator( A, B, gt, lt, eq );
    input A, B;
    output gt, lt, eq;

    wire negA, negB;

    not not1( negA, A );
    not not2( negB, B );
    and and1( gt, A, negB );
    and and2( lt, negA, B );
    xnor xnor1( eq, A, B );

endmodule
```



REFERENCE


```
1. Combinational circuit
https://www.javatpoint.com/combinational-logic-circuits-in-digital-electronics

2. sequential Circuit
https://www.geeksforgeeks.org/introduction-of-sequential-circuits/

3. Mealy vs Moore
https://unstop.com/blog/difference-between-mealy-and-moore-machine

4. 描述電路的三種層次 - HackMD
https://hackmd.io/@dppa1008/BJWS5_B_G?type=view

5. Verilog Synthesis Methodology
https://people.ece.cornell.edu/land/courses/ece5760/Verilog/coding_and_synthesis_w
ith_verilog.pdf
```