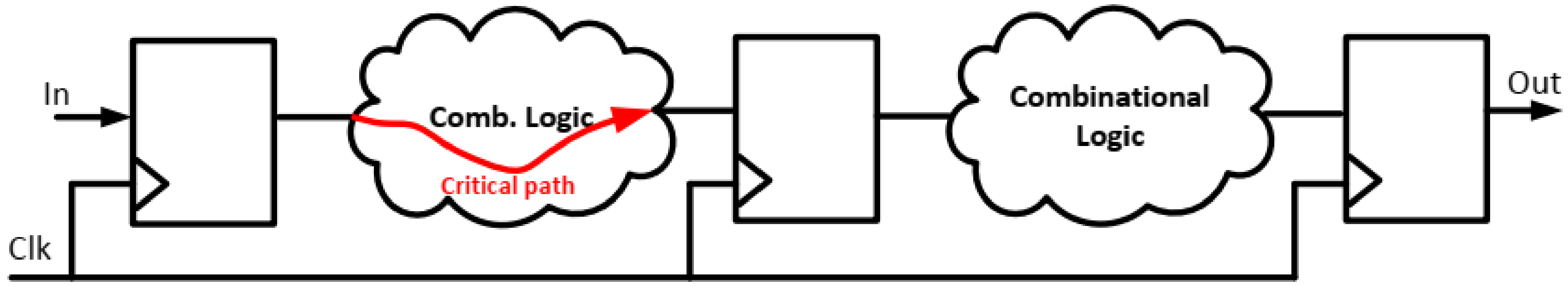
Digital System Design

Hajar Falahati

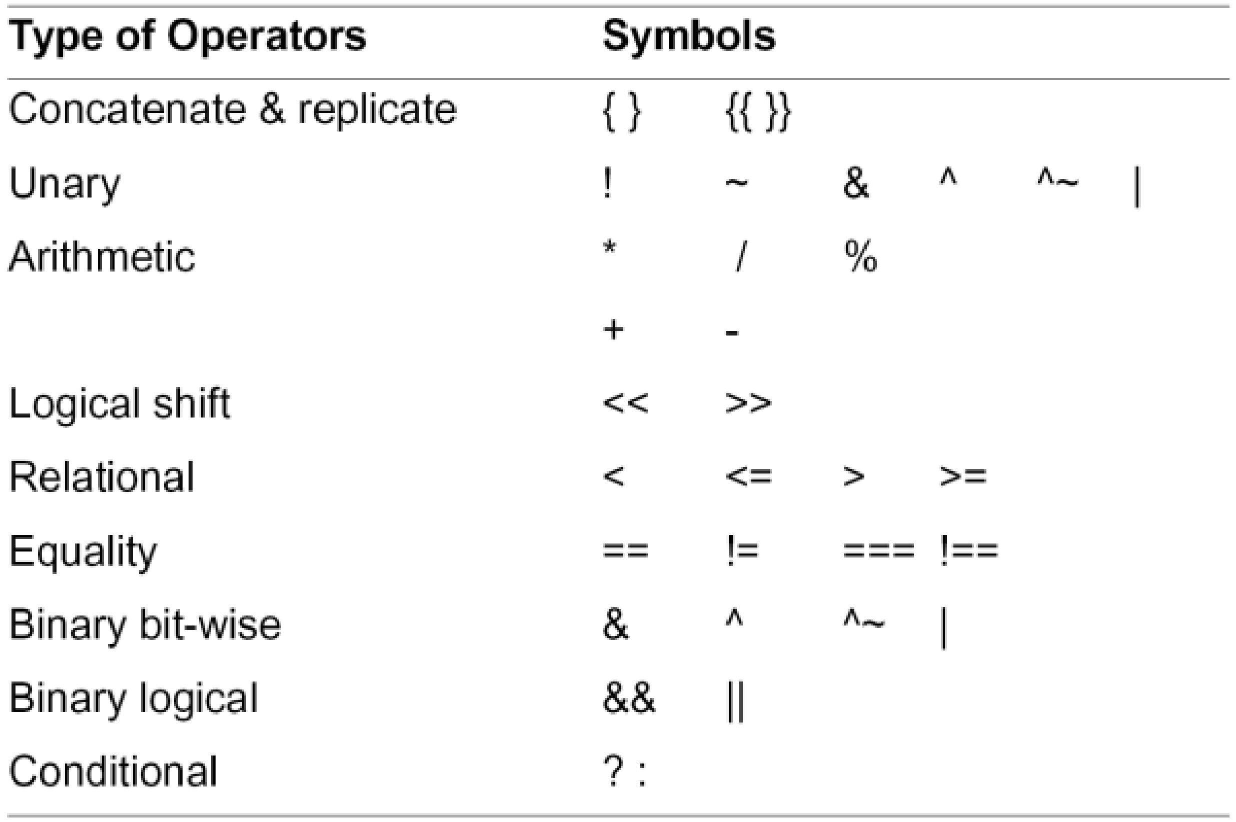
hfalahati@ipm.ir
hfalahati@ce.sharif.edu

RTL

- Functional level
- Describe a design architecture in sufficient detail that a synthesis tool can construct the circuit
- RTL is the closet one to the actual hardware implementation



Operators: All in one



Type of Operators	Symbols					
Concatenate & replicate	{ }	{{ }}				
Unary	!	~	&	^	^~	|
Arithmetic	*	/	%			
	+	-				
Logical shift	<<	>>				
Relational	<	<=	>	>=		
Equality	==	!=	===	!==		
Binary bit-wise	&	^	^~	|		
Binary logical	&&	||				
Conditional	? :					

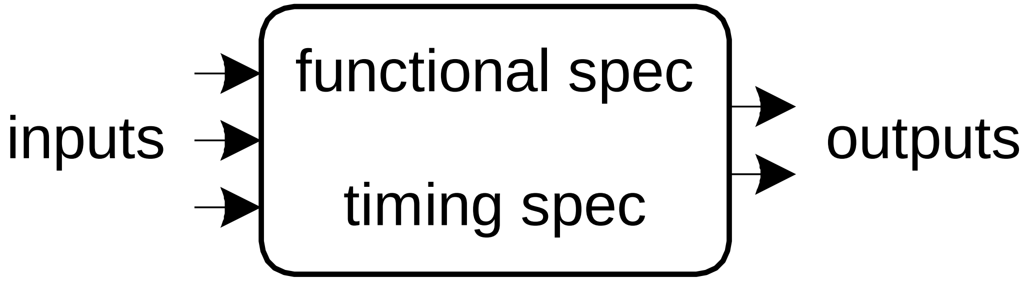
Outline

- How to implement a combinational logic at behavioral-level?
- How to implement a sequential logic at behavioral-level?

Combinational Logic

Logic Circuits

- A logic circuit is composed of:
  - Inputs
  - Outputs
- *Functional specification* (describes relationship between inputs and outputs)
- *Timing specification* (describes the delay between inputs changing and outputs responding)



Combinational Logic & Verilog

- Use **always block** + **"blocking" assignments**
  - Normally for high-complexity Comb. Logic
  - When output depends on several conditions, which requires if-else

Which one is correct?

- Implement an accumulator

```
always @ (*)
      begin
       for (k=0; k<4; k=k+1)
              Count = Count + k;
      end
```

```
always @ (*)
      begin
       for (k=0; k<4; k=k+1)
              Count <= Count + k;
      end
```

Let’s Check Their Behavior?

```
always @ (*)
    begin
     for (k=0; k<4; k=k+1)
           Count = Count + k;
    end
```

✓

```
Count = Count + 0;
Count = 0 + 1;
Count = 0 + 1 + 2;
Count = 0 + 1 + 2 + 3;
```

Result: Count = 6

```
always @ (*)
    begin
     for (k=0; k<4; k=k+1)
           Count <= Count + k;
    end
```

✗

```
Count <= Count + 0;
Count <= Count + 1;
Count <= Count + 2;
Count <= Count + 3;
```

Result: Count =3

Use only blocking assignments for combinational logic

In multiple concurrent non-blocking assignments to a variable the last one executes

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

- This statement describes what happens to the signal result
  - When inv is 1, result is ~data
  - When inv is not 1, result is data
- The circuit is combinational (no memory)

```
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
```

Sample 1

- Implement a 2-1 MUX

Sample 1

Implement a 2-1 MUX

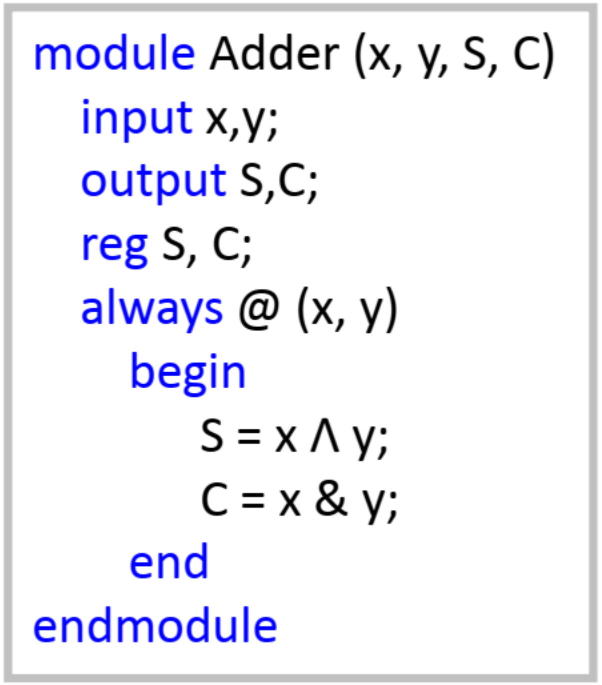
```
module Mux (a,b, sel, result)
   output reg  [31:0] result;
   input       [31:0] a, b;
   input              sel,

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

endmodule
```

Sample 2

- Implement a full adder

Sample 2

- Implement a full adder



```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   reg S, C;
   always @ (x, y)
      begin
          S = x Λ y;
          C = x & y;
      end
endmodule
```

Summary

- An **always** block defines a **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

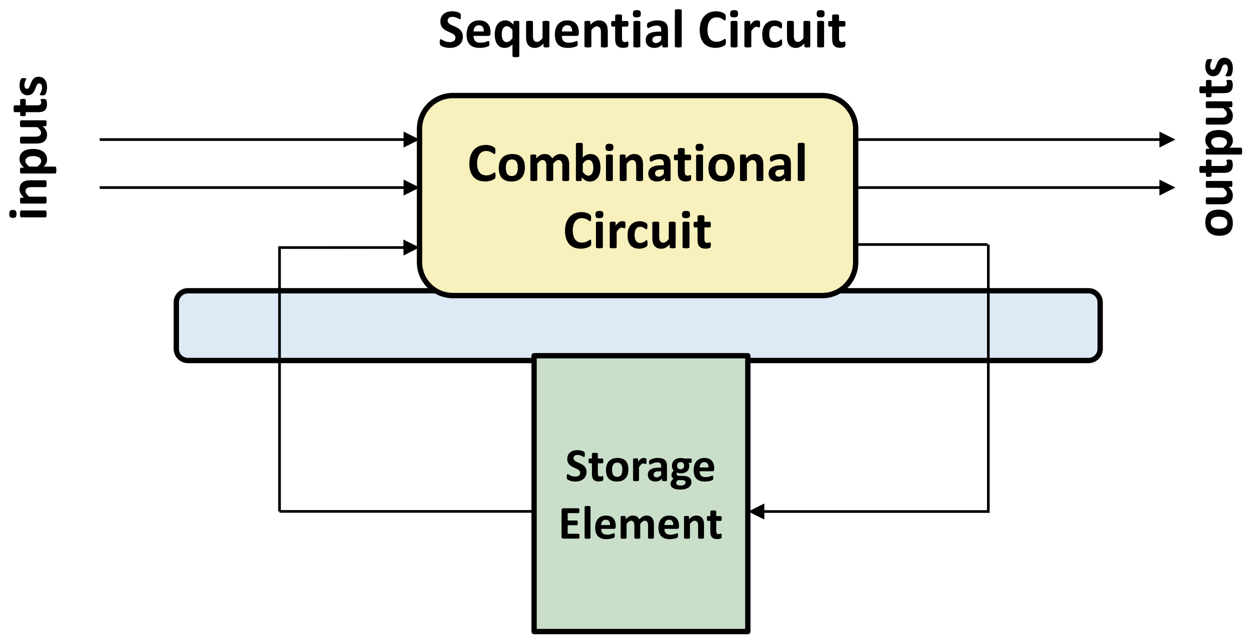
- It is easy to make mistakes and unintentionally describe memorizing elements (latches)
  - Vivado will most likely warn you.
  - Make sure you check the warning messages

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

Sequential Logic

Combinational + Memory = Sequential

- Sequential logic can **only** be realized using an **always block**
- When using the **always block** for the sequential Logic, **“Non-blocking” assignments** are used



Sequential Circuits in Verilog

- Define blocks that have memory
  - *Flip-Flops*
    - *Positive edge of the clock (posedge)*
    - *Negative edge of the clock (negedge)*
  - *Latches*
    - Sensitive to level of the signal
  - *Finite State Machines*

```
always @ (posedge Clk)
always @ (negedge Clk)
```

- Sequential Logic state transition is triggered by a ‘CLOCK’ event

Sample: D Flip-Flop

- 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

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

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

endmodule
```

Sample: D Flip-Flop

- Only procedural statements are used within an always block
- “Non-blocking” assignments

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

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

endmodule
```

Sample: D Flip-Flop

- Assigned variables need to be declared as reg

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

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

endmodule
```

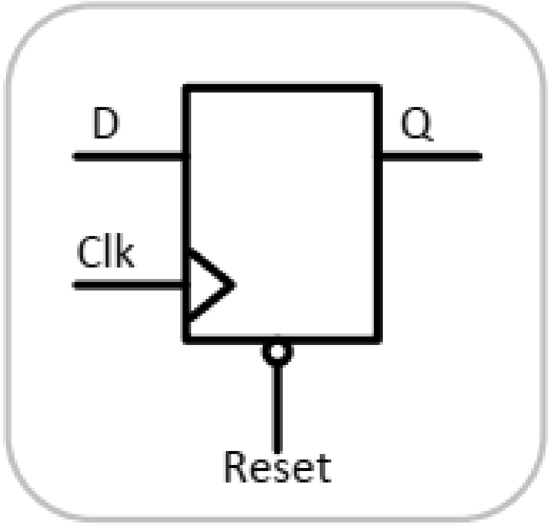
Sample 1: D Flip-Flop

- 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

- **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



D Flip-Flop with Asynchronous Reset

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

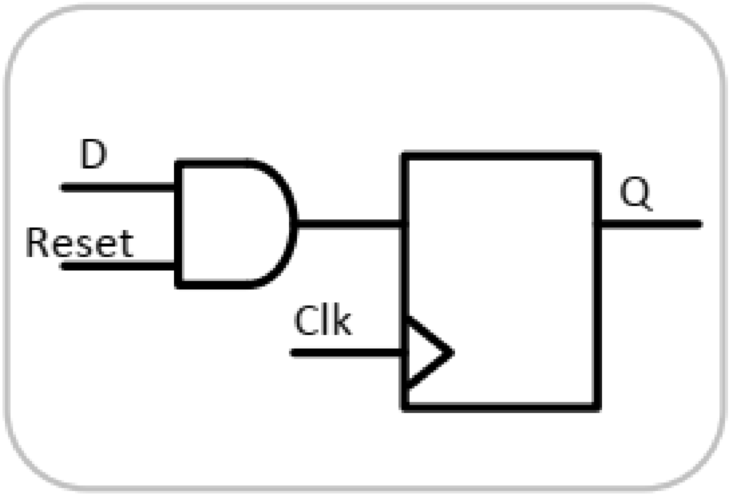
```
module flop_ar (input              clk,
                input              reset,
                input        [3:0] d,
                output reg   [3:0] q);

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

D Flip-Flop with Asynchronous Reset

- 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

```
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
```

D Flip-Flop with Synchronous Reset



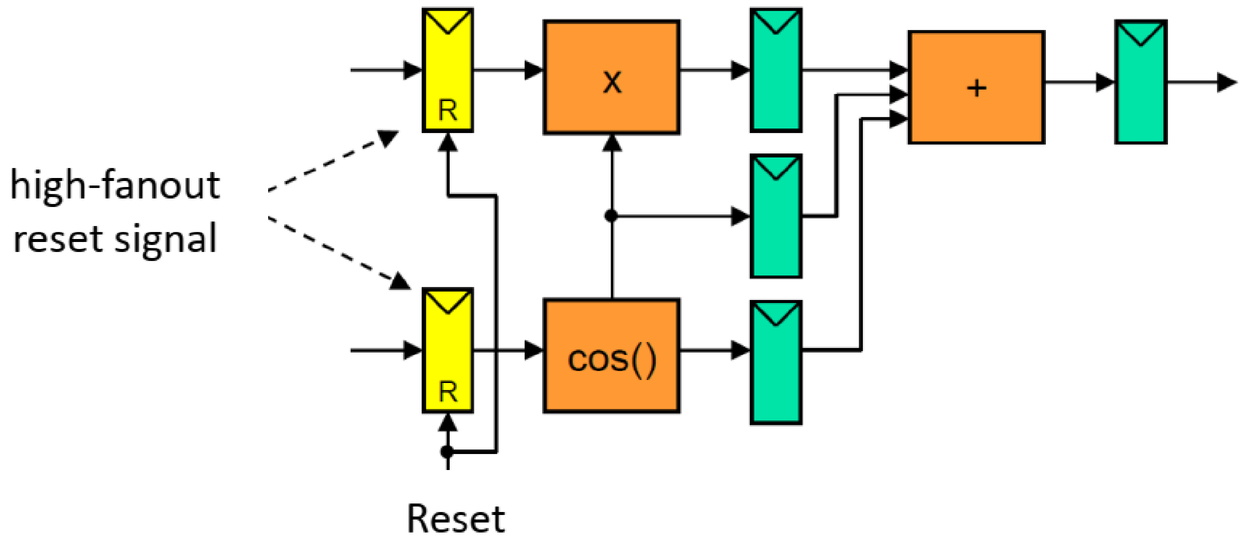
D Flip-Flop with Synchronous Reset

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

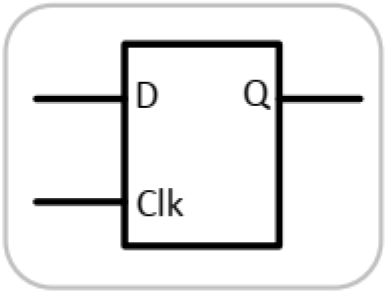
```
module flop_ar (input              clk,
                input              reset,
                input        [3:0] d,
                output reg   [3:0] q);

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

NOTE !

- Use reset-able FFs only where needed
  - FFs are a little larger and higher power
  - Requires the global routing of the high-fanout reset signal



Sample2: D- Latch



Sample2: D- Latch

```
module flop_ar (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
```

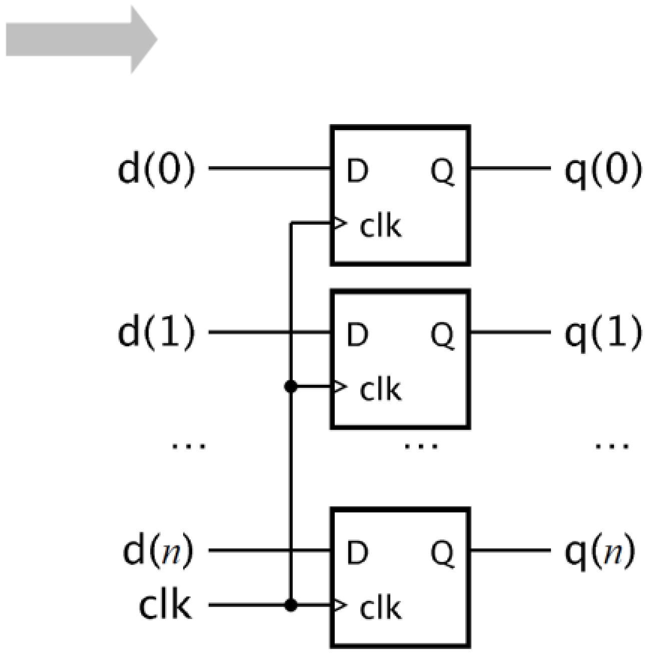
An **always** Block Remember

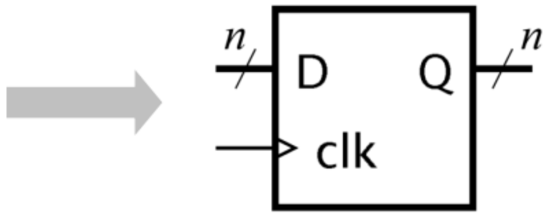
- This statement describes what happens to signal q
- … but what happens when the clock is not rising?
- The value of q is preserved (remembered)

```
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
```

Shift Register

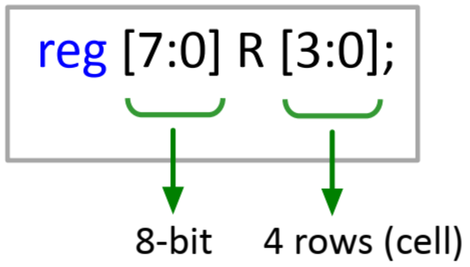
```
wire [n:0] d;
reg [n:0] q;
...
always @ (posedge Clk)
         q<=d;
```



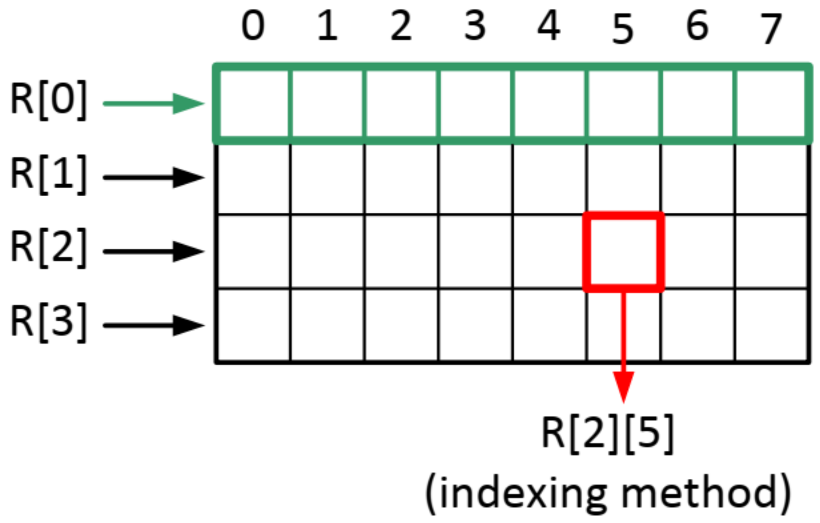


Memory

- A two-dimensional array of bits
- Declared in Verilog as a two-dimensional variable (reg)



```
reg [7:0] R [3:0];
```



- 3-D memory

```
reg [7:0] M [3:0][1:0];
```

```
reg [7:0] A;
```

```
A = M[3][0];
```

Which one is correct?

- Implement a shift register

```
always @ (A)
     begin
        for (k=0; k<3 ;k=k+1)
             A[k]=A[k+1];
        A[3] = A[0];
     end
```

```
always @ (A)
     begin
        for (k=0; k<3;k=k+1)
             A[k]<=A[k+1];
        A[3] <= A[0];
     end
```

Which one is correct?

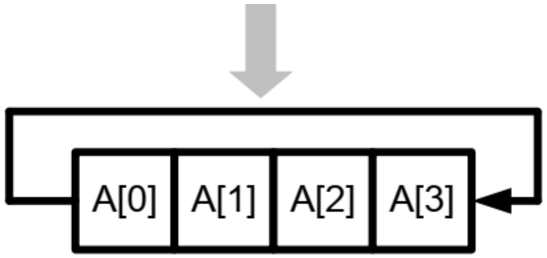
- Implement a shift register

✗

```
always @ (A)
    begin
      for (k=0; k<3 ;k=k+1)
          A[k]=A[k+1];
      A[3] = A[0];
    end
```

```
always @ (A)
    begin
      for (k=0; k<3;k=k+1)
          A[k]<=A[k+1];
      A[3] <= A[0];
    end
```

✓



Thank You