COUNTERS AND VERILOG ASSIGNMENT TYPES

Counters in Digital Logic

Counters in digital logic are sequential circuits used to count the number of clock cycles or events. They can be designed using various flip-flops, such as D flip-flops or JK flip-flops, and can be either synchronous or asynchronous.

There are different types of counters, including:

1. Binary counters: Binary counters count in binary representation, which means they can count from 0 to 2^n-1, where n is the number of bits in the counter. For example, a 4-bit binary counter can count from 0000 to 1111.
2. Decade counters: Decade counters count in decimal representation, which means they can count from 0 to 9. These counters are commonly used in applications where counting in decimal digits is required, such as digital clocks.
3. Up/down counters: Up/down counters can count both upwards and downwards, depending on the control signals. They have an additional input that determines the direction of the count. These counters are used in applications where the count needs to be incremented or decremented based on certain conditions.

Counters are widely used in digital systems for applications such as frequency division, event counting, and timekeeping.

Synchronous counters are a type of counters in digital logic that operate based on a common clock signal. They are widely used in digital systems for applications such as frequency division, event counting, and timekeeping.

TYPES OF COUNTERS

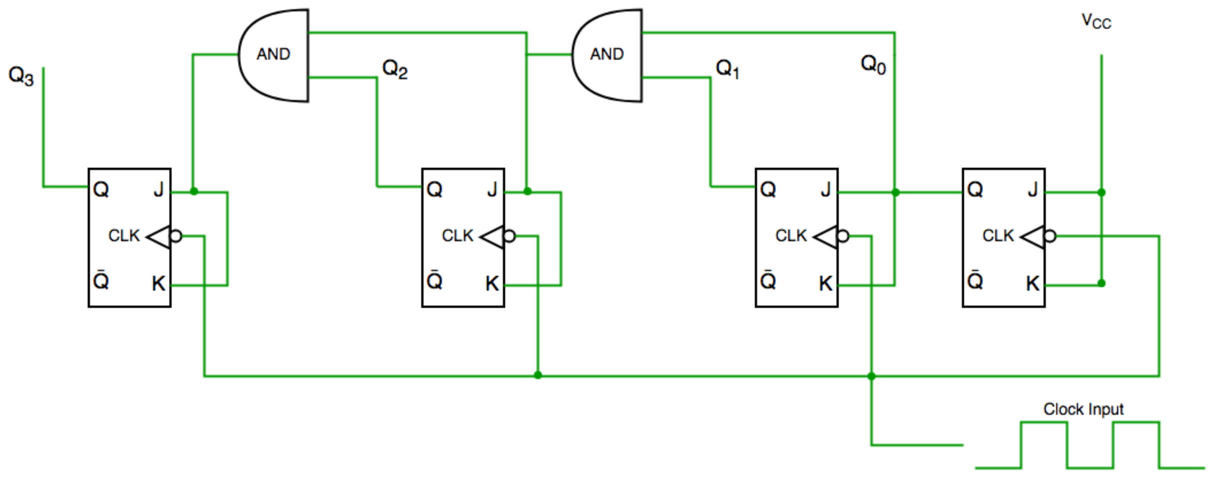
1. SYNCHRONOUS COUNTERS

In a synchronous counter, all flip-flops within the counter change their states simultaneously at the rising or falling edge of the clock signal. This ensures that the counter transitions to the next state in a synchronized manner. The synchronous nature of these counters allows for precise timing and synchronization in the counting process.

One advantage of synchronous counters is that they are more reliable and less prone to errors compared to asynchronous counters. Since all flip-flops change their states at the same time, there is no possibility of conflicting or inconsistent outputs. This makes synchronous counters suitable for applications where accuracy and reliability are crucial.

Synchronous counters can be designed using various flip-flops, such as D flip-flops or JK flip-flops. The choice of flip-flop type depends on the specific requirements of the counter and the desired functionality. These counters can also have additional inputs, such as clear and load inputs, to control the starting state and enable external control.

Overall, synchronous counters offer a reliable and synchronized way of counting in digital logic systems. Their precise timing and synchronization capabilities make them suitable for a wide range of applications that require accurate counting and event tracking.



Here’s an example of synchronous counter!

```
module synchronous_counter(clk, reset, count);
  input clk, reset;
  output [3:0] count;
  reg [3:0] count;

  always @(posedge clk) begin
    if (reset)
      count <= 4'b0000;
    else begin
            if (count == 4'b1111)
                count <= 4'b0;
            else
      count <= count + 1;
        end
  end
endmodule
```

Here's an example of a 4-bit synchronous up/down counter in Verilog:

```
module synchronous_up_down_counter(clk, reset, up_down, count);
  input clk, reset, up_down;
  output [3:0] count;
  reg [3:0] count;

  always @(posedge clk) begin
    if (reset)
      count <= 4'b0000;
    else if (up_down) begin
            if (count == 4'b1111)
                count <= 4'b0;
            else
      count <= count + 1;
        end
    else begin
            if (count == 4'b0)
```

```
                  count <= 4'b1111;
              else
          count <= count - 1;
            end
      end
    endmodule
```

In this code, the `clk` signal is the clock input, `reset` signal is used to reset the counter, `up_down` signal determines the direction of the count, and `count` is the 4-bit output representing the current count value.

The `always @(posedge clk)` block ensures that the counter updates its value on the rising edge of the clock signal. When the `reset` signal is high, the counter is reset to 0. When `up_down` is high, the counter increments by 1, and when `up_down` is low, the counter decrements by 1.

You can modify the code according to your specific requirements, such as changing the bit width of the counter or adding additional functionality.

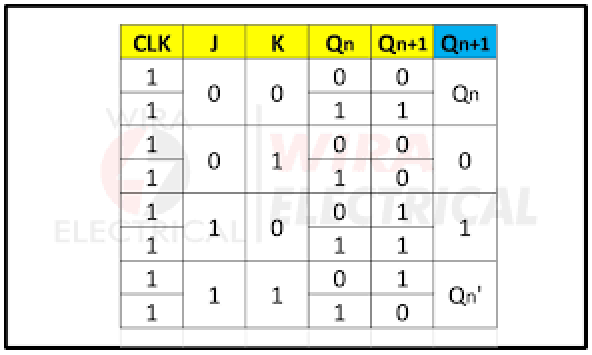
2. ASYNCHRONOUS COUNTER

In contrast to synchronous counters, asynchronous counters do not rely on a common clock signal to change their states. Each flip-flop within the counter changes its state independently, based on the output of the previous flip-flop.

Asynchronous counters are more flexible in terms of their operation and can be easily designed using simple flip-flops, such as D flip-flops or T flip-flops. However, they are more prone to errors and glitches compared to synchronous counters. The lack of synchronization can lead to inconsistent outputs and timing issues.

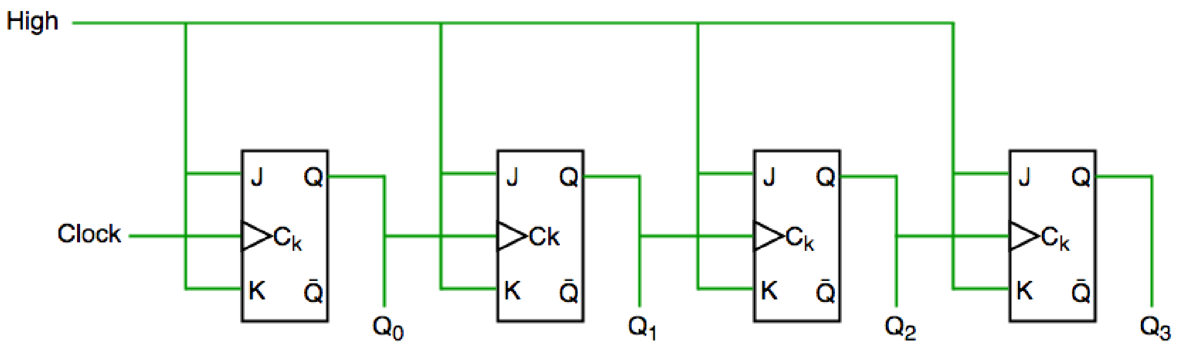
One advantage of asynchronous counters is their simplicity and lower hardware requirement. They are suitable for applications where precise timing and synchronization are not critical. Asynchronous counters are often used in simple counting tasks where accuracy is not the primary concern.

To understand modelling we have to first understand the functioning of a flipflop with toggling outputs (example jk).

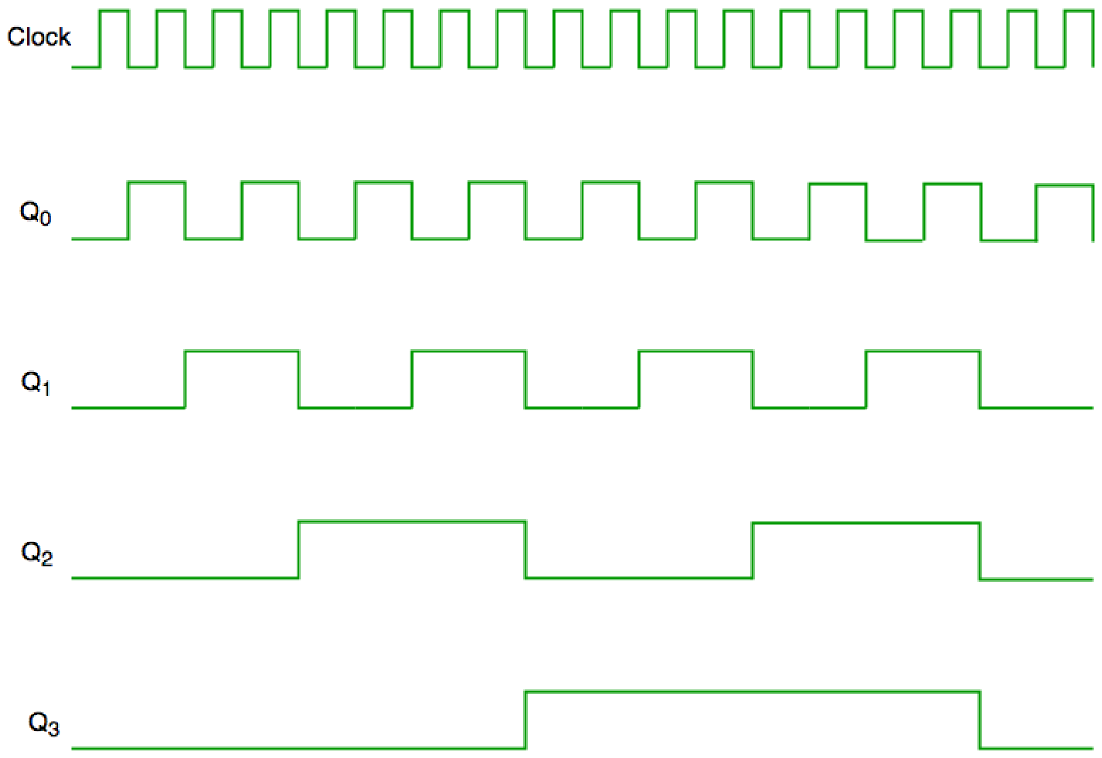


CLK	J	K	Qn	Qn+1	Qn+1
1	0	0	0	0	Qn
1			1	1	
1	0	1	0	0	0
1			1	0	
1	1	0	0	1	1
1			1	1	
1	1	1	0	1	Qn'
1			1	0	

when the input j and k both are high the output keeps toggling. This functionality is made use of to make asynchronous counters.



(a) Asynchronous counter



(b) Timing Diagram

Here's an example of a 4-bit asynchronous counter in Verilog:

```
module asynchronous_counter(clk, reset, count);
  input clk, reset;
  output [3:0] count;
  reg [3:0] count;
  always @(posedge clk or posedge reset) begin
    if (reset)
      count <= 4'b0000;
    else if (count == 4'b1111)
      count <= 4'b0000;
    else
      count <= count + 1;
  end
endmodule
```

In this code, the `clk` signal is the clock input, `reset` signal is used to reset the counter, and `count` is the 4-bit output representing the current count value.

The `always @(posedge clk or posedge reset)` block ensures that the counter updates its value on the rising edge of the clock signal or when the reset signal is high. When the reset signal is high, the counter is reset to 0. When the count reaches its maximum value (15 in this case, represented by `4'b1111`), it wraps around to 0 and starts counting again.

You can modify the code according to your specific requirements, such as changing the bit width of the counter or adding additional functionality.

Code for asynchronous counter using T- flip flop

```
module t_flipflop(input clk, input reset, output reg q);
  always @(posedge clk or posedge reset) begin
    if (reset)
      q <= 1'b0;
    else
      q <= ~q;
  end
endmodule

module asynchronous_counter_using_t_flipflop(clk, reset, count)
  input clk, reset;
  output reg [3:0] count;
  reg [3:0] q;
  t_flipflop flipflop0 (.clk(clk), .reset(reset), .q(q[0]));
  t_flipflop flipflop1 (.clk(q[0]), .reset(reset), .q(q[1]));
  t_flipflop flipflop2 (.clk(q[1]), .reset(reset), .q(q[2]));
  t_flipflop flipflop3 (.clk(q[2]), .reset(reset), .q(q[3]));

  always @(posedge clk or posedge reset) begin
    if (reset)
      count <= 4'b0000;
    else
      count <= q;
  end
endmodule
```

In this code, the `t_flipflop` module represents a T flip-flop. The `asynchronous_counter_using_t_flipflop` module uses four instances of the T flip-flop to create a 4-bit asynchronous counter. Each T flip-flop is connected to the clock signal `clk` and the reset signal `reset`. The outputs of the T flip-flops are used as the bits of the counter.

The counter updates its value on the rising edge of the clock signal or when the reset signal is high. When the reset signal is high, the counter is reset to 0000. Otherwise, the counter increments based on the toggling behavior of the T flip-flops.

You can modify the code according to your specific requirements, such as changing the bit width of the counter or adding additional functionality to the T flip-flop module.

VERILOG - assignments

1. Procedural assignments

Procedural blocks in Verilog are sections of code that define the behavior of a digital circuit based on specific conditions or events. Procedural blocks allow for the sequential execution of code and the manipulation of signal values.

There are two main types of procedural blocks in Verilog:

1. `always` Blocks:

- `always` blocks are used to describe the behavior of combinational or sequential logic.
- They are executed whenever there is a change in the signals specified in the sensitivity list.
- The sensitivity list specifies the signals that trigger the execution of the block.
- The code within an `always` block executes sequentially, following the order of the statements.
- Common uses of `always` blocks include modeling combinational logic, sequential state machines, and register behavior.
- **Purpose:** The `always` block is used to describe concurrent (parallel) behavior in Verilog. It is sensitive to certain events, such as changes in signal values or edges of a clock signal.
- **Execution:** The statements inside an `always` block are continuously executed whenever the specified event occurs. For example, an `always @(posedge clk)` block executes its statements on every positive edge of the `clk` signal.

2. `initial` Blocks:

- `initial` blocks are used to initialize variables or perform one-time operations at the beginning of simulation.
- They are executed only once at the start of simulation, before any other procedural or continuous assignments.
- The code within an `initial` block executes sequentially, following the order of the statements.
- `initial` blocks are typically used for testbench code, such as initializing inputs or generating stimulus.
- **Purpose:** The `initial` block is used for describing the initial state or setup of the design. It executes only once at the beginning of simulation.
- **Execution:** The statements inside an `initial` block are executed sequentially at the start of simulation and then the block is not executed again during the simulation.

Both `always` and `initial` blocks can contain procedural assignments, conditional statements (such as `if` and `case`), loops (such as `for` and `while`), and other procedural constructs. These blocks are essential for describing the behavior of digital circuits and allow for the implementation of complex logic and sequential operations.

It's important to understand the differences between `always` and `initial` blocks and use them appropriately based on the specific requirements of your Verilog design.

1.1 BLOCKING ASSIGNMENT

Blocking assignment in Verilog is a mechanism used within procedural blocks (such as `always` or `initial`) to sequentially assign values to variables or registers. The term "blocking" indicates that the next statement in the block waits for the completion of the current assignment before starting its execution. This creates a sequential execution flow.

```
module Example(input wire a, input wire b, output reg c, output

  always @(posedge clk) begin
    c = a & b; // Blocking assignment
    d = c;     // Subsequent statement, waits for completion of
  end

endmodule
```

`c <= a & b;` is a non-blocking assignment. The next statement (`d <= c;`) can start execution immediately after encountering this assignment, without waiting for its completion.

Things to remember

- Statements with blocking assignments are executed sequentially in the order they appear within the procedural block
- The next statement in the block waits for the completion of the current assignment before it starts execution.
- Blocking assignment uses the `=` operator.

1.2 NON BLOCKING ASSIGNMENT

Non-blocking assignments in Verilog are used within procedural blocks to specify how variables or registers are updated over time. Non-blocking assignments allow concurrent execution of statements within the procedural block, providing a more concurrent or simultaneous execution model compared to blocking assignments. Non-blocking assignments are commonly used in the context of describing synchronous digital circuits.

```
module Example(input wire a, input wire b, output reg c, output

  always @(posedge clk) begin
    // Non-blocking assignment
    c <= a & b;

    // Another concurrent statement
    d <= c;
  end

endmodule
```

Things to remember

- Statements with non-blocking assignments can execute concurrently within the procedural block.
- The next statement in the block can start execution immediately after encountering a non-blocking assignment; it doesn't wait for the assignment to complete.
- Non-blocking assignments use the `<=` operator.

2.Continuous Assignment

Continuous assignments in Verilog are used to describe the connections between wires and registers, specifying how signals are driven by combinational logic. Unlike procedural assignments found within `always` or `initial` blocks, continuous assignments are executed continuously throughout the simulation, reflecting the real-time behavior of combinational logic.

- FOR EXAMPLE: `assign c = a & b;`

In this example, the `assign` statement expresses that the value of `c` is the result of the bitwise AND operation between signals `a` and `b`. The assignment is continuous, meaning it reflects the real-time relationship between inputs and output.

Things to remember

- The general syntax of a continuous assignment is `assign <output_signal> = <expression>;`.
- Continuous assignments are typically used for describing combinational logic. They represent the instantaneous relationship between input and output signals.
- Continuous assignments follow a data flow modeling style, where signals are interconnected by expressing logical relationships.
- Continuous assignments are always active, meaning they continuously reflect changes in the input signals without being triggered by specific events.
- Continuous assignments can be used to assign values to both wire and reg data types. However, when assigning to a reg, the left-hand side (LHS) signal is considered a net.
- Continuous assignments follow the "always last" rule, meaning if there are multiple drivers for a signal, the last one encountered takes precedence.

💡 Continuous assignments are not used within procedural blocks like `always` or `initial` Non blocking assignments are frequently used in sequential circuits. Blocking assignments are frequently used in combinational circuits.