Notes for ECE 27000 - Introduction to Digital System Design

Zeke Ulrich

April 10, 2025

Contents

Course Description

An introduction to digital system design, with an emphasis on practical design techniques and circuit implementation.

Verilog

SystemVerilog is a hardware description and verification language (HDVL) that extends Verilog by adding high-level programming constructs. It is widely used for modeling, simulating, and verifying digital systems.

The listing below is an example of a NAND gate expressed in SystemVerilog.

Listing 1: NAND

```
module nand_gate (
    input  logic a,
    input  logic b,
    output logic y
);
    assign y = ~(a & b);
endmodule
```

In SystemVerilog, numbers are written in format `[size]'[base][number]`, for example:

- 4'b1001 (binary, 9 in decimal, bit width 4 bits)
- 8'hf1 (hex, equals 421, bit width 8 bits)
- 3'o3 (octal, 3, bit width 3 bits)
- 32'b1001_1101_0101_1111 (binary, 40255, bit width 32 bits)

Number Systems

In daily life, we primarily interact with the familiar base-10 numbers. However, when interaction with digital systems, we must also concern ourselves with base-2, base-8, base-16, and other bases which are friendly to binary states. Unless completely unambiguous, the base of a number is written as a right subscript such as 144_{10} for base-10 or 1001_2 for base-2.

For binary numbers, each digit represents a power of two. To convert a binary number to decimal, you sum the products of each binary digit with its corresponding power of two. For example, the binary number 1001 is calculated as

$$2^3 \times 1 + 2^2 \times 0 + 2^1 \times 0 + 2^0 \times 1 = 8 + 0 + 0 + 1 \tag{1}$$

$$= 9_{10} \tag{2}$$

To convert from hexadecimal to decimal, each digit represents a power of sixteen. For instance, the hexadecimal number f1 is calculated as

$$15 \times 161 + 1 \times 160 = 240 + 1 \tag{3}$$

$$= 241_{10} \tag{4}$$

where f represents the decimal value 15. When converting to another base, reverse the process by dividing the decimal number by the target base, recording the remainder, and repeating with the quotient until it reaches zero. The remainders give you the digits of the number in the new base, read in reverse order.

To convert a decimal number into binary, for example, you repeatedly divide the number by 2 and record the remainders. For the decimal number 9, dividing by 2 gives a quotient of 4 and a remainder of 1. Dividing 4 by 2 gives a quotient of 2 and a remainder of 0. Dividing 2 by 2 gives a quotient of 1 and a remainder of 0, and finally, dividing 1 by 2 gives a quotient of 0 and a remainder of 1. Reading the remainders from bottom to top, the binary representation of 9 is 1001.

Boolean Algebra

Computers operate in binary. To represent the state of a computer we require a suitable mathematical framework, provided by boolean algebra. In boolean algebra, variables can only take on two values: 0 and 1.

Rule	**Expression**
Commutativity	$X + Y = Y + X$ $X \cdot Y = Y \cdot X$
Associativity	$(X + Y) + Z = X + (Y + Z)$ $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$
Distributivity	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$ $(X + Y) \cdot (X + Z) = X + Y \cdot Z$
Covering	$X + X \cdot Y = X$ $X \cdot (X + Y) = X$
Combining	$X \cdot Y + X \cdot Y = X$ $(X + Y) \cdot (X + Y) = X$
Consensus	$X \cdot Y + X \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$ $(X + Y) \cdot (X + Z) \cdot (Y + Z) = (X + Y) \cdot (X + Z)$
Generalized Idempotency	$X + X + \cdots + X = X$ $X \cdot X \cdot \cdots \cdot X = X$
DeMorgan's Theorems	$(X_1 \cdot X_2 \cdot \cdots \cdot X_n)' = X_1' + X_2' + \cdots + X_n'$ $(X_1 + X_2 + \cdots + X_n)' = X_1' \cdot X_2' \cdot \cdots \cdot X_n'$
Generalized DeMorgan's	$F(X_1, X_2, \ldots, X_n, +, \cdot) = F(X_1, X_2, \ldots, X_n, \cdot, +)'$
Shannon's Expansion	$F(X_1, X_2, \ldots, X_n) = X_1 \cdot F(1, X_2, \ldots, X_n) + X_1' \cdot F(0, X_2, \ldots, X_n)$ $F(X_1, X_2, \ldots, X_n) = [X_1 + F(0, X_2, \ldots, X_n)] \cdot [X_1' + F(1, X_2, \ldots, X_n)]$

Table 1: Boolean Algebra

An interesting and useful property in boolean algebra is "duality", where replacing all ANDs with ORs and all 1s with 0s gives a valid and equivalent theorem. For instance,

X AND 0 = 0	X OR 1 = 1
X OR 0 = X	X AND 1 = X

Table 2: Boolean Duality

Any logic can be implemented using just the following:

- AND, OR, and NOT gates
- NAND gates
- NOR gates

Logic Gates

Buffer

A	Output
0	0
1	1

NOT

A	Output
0	1
1	0

OR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

NAND

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

NOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

XOR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

CMOS

Complementary Metal-Oxide-Semiconductor (CMOS) technology is the dominant semiconductor technology for modern integrated circuits. CMOS combines both n-type (NMOS) and p-type (PMOS) metal-oxide-semiconductor field-effect transistors (MOSFETs) to create the familiar logic gates such as AND, NOT, XOR, etc.

PMOS

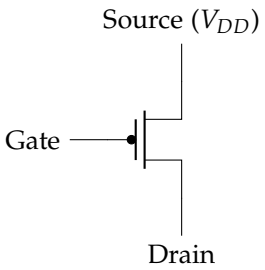

Figure 1: PMOS transistor circuit symbol

PMOS transistors consist of:

- p+ source and drain regions
- n-type substrate (body)
- SiO_2 gate dielectric
- Polysilicon gate electrode

PMOS operates with negative gate-to-source voltage (V_{GS}):

- **Cut-off Region** ($V_{GS} > V_{th,p}$):

$$I_D = 0$$

- **Linear Region** ($V_{GS} \leq V_{th,p}$ and $V_{DS} \geq V_{GS} - V_{th,p}$):

$$I_D = -\mu_p C_{ox} \frac{W}{L} \left[(V_{GS} - V_{th,p}) V_{DS} - \frac{V_{DS}^2}{2} \right]$$

- **Saturation Region** ($V_{GS} \leq V_{th,p}$ and $V_{DS} < V_{GS} - V_{th,p}$):

$$I_D = -\frac{1}{2} \mu_p C_{ox} \frac{W}{L} (V_{GS} - V_{th,p})^2$$

NMOS

NMOS transistors consist of:

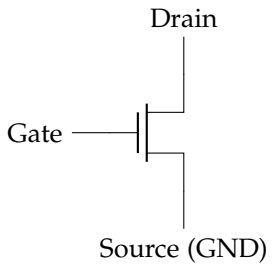

Figure 2: NMOS transistor circuit symbol

- n+ source and drain regions
- p-type substrate (body)
- SiO_2 gate dielectric
- Polysilicon gate electrode

NMOS operates with positive gate-to-source voltage (V_{GS}):

- **Cut-off Region** ($V_{GS} < V_{th,n}$):

$$I_D = 0$$

- **Linear Region** ($V_{GS} \geq V_{th,n}$ and $V_{DS} \leq V_{GS} - V_{th,n}$):

$$I_D = \mu_n C_{ox} \frac{W}{L} \left[(V_{GS} - V_{th,n}) V_{DS} - \frac{V_{DS}^2}{2} \right]$$

- **Saturation Region** ($V_{GS} \geq V_{th,n}$ and $V_{DS} > V_{GS} - V_{th,n}$):

$$I_D = \frac{1}{2} \mu_n C_{ox} \frac{W}{L} (V_{GS} - V_{th,n})^2$$

Parameter	**PMOS**	**NMOS**
Majority Carrier	Holes	Electrons
Substrate Type	n-type	p-type
Threshold Voltage	Negative	Positive
Mobility (μ)	Lower ($\approx 150 \frac{cm^2}{V_s}$)	Higher ($\approx 400 \frac{cm^2}{V_s}$)
Speed	Slower	Faster

Table 3: PMOS vs NMOS characteristics

Propagation Delays and Transition Times

Propagation Delay

The propagation delay T_P is defined as the time delay between the 50% crossing of the input and the corresponding 50% crossing of the output. There are two kinds:

t_{pHL}: The time between an input change and the corresponding output change when the output is changing from HIGH to LOW.

t_{pLH}: The time between an input change and the corresponding output change when the output is changing from LOW to HIGH.

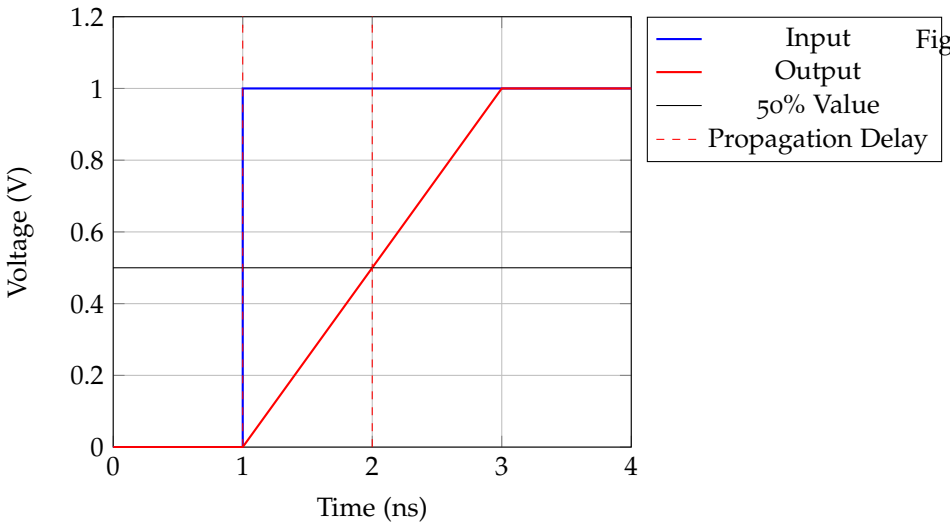

Figure 3: Propagation Delay

Transition Time

The amount of time that the output of a logic circuit takes to change from one state to another is called the transition time. An output takes a certain time, called the rise time (t_r), to change from LOW to HIGH, and a possibly different time, called the fall time (t_f), to change from HIGH to LOW. The rise time and fall time of the output signal are defined as the time required for the voltage to change from its 10% level to its 90% level or vice versa.

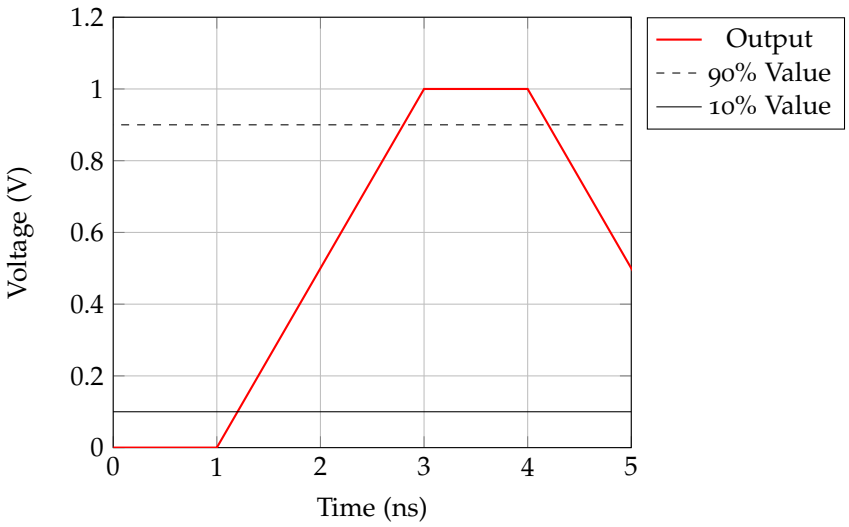

Figure 4: fig:risefalltime

Power Consumption

The power consumption of a CMOS circuit whose output is not changing is called *static power dissipation*. The power a CMOS circuit consumes during signal transitions is called *dynamic power dissipation*.

In general dynamic power dissipation is much larger than static. One significant source of dissipation is output transitions. The power consumed as the voltage transitions is given by

$$P_T = C_{PD} V_{CC}^2 f \tag{5}$$

where

- P_T is circuit's internal power dissipation due to output transitions
- C_{PD} is power-dissipation capacitance, normally specified by the device manufacturer
- V_{CC} is power supply voltage
- f is transition frequency of the output signal

A second (and often more significant) source of CMOS power consumption is the capacitive load (C_L) on the output.

Noise

$$N_{MH} = V_{OH} - V_{IH} \tag{6}$$

$$N_{ML} = V_{IL} - V_{OL} \tag{7}$$

$$N_{MH}(A \rightarrow B) = V_{OHA} - V_{IHB} \tag{8}$$

$$N_{ML}(A \rightarrow B) = V_{ILB} - V_{OLA} \tag{9}$$

K-Maps

A Karnaugh Map (K-Map) is a visual representation used to simplify Boolean expressions and minimize logic functions. It is a method to perform Boolean algebra simplifications by organizing truth table values into a grid format, allowing easy identification of common terms. K-Maps are structured as grids where each cell corresponds to a specific combination of input variables. The number of cells in a K-Map depends on the number of variables:

- 2-variable K-Map: $2^2 = 4$ cells
- 3-variable K-Map: $2^3 = 8$ cells
- 4-variable K-Map: $2^4 = 16$ cells

For a function with two variables (A and B), the K-Map is a 2×2 grid:

AB	0	1
0	F(0,0)	F(0,1)
1	F(1,0)	F(1,1)

For three variables (A, B, C), the K-Map has $2^3 = 8$ cells:

AB \C	0	1
00	F(0,0,0)	F(0,0,1)
01	F(0,1,0)	F(0,1,1)
11	F(1,1,0)	F(1,1,1)
10	F(1,0,0)	F(1,0,1)

The goal of a K-Map is to group adjacent cells containing 1s into power-of-two groups (1, 2, 4, etc.), forming simplified expressions. Groups can wrap around the edges of the K-Map.

An example would be illustrative. Consider the K-Map in figure 5.

A \BC	00	01	11	10
0	0	1	1	0
1	0	0	1	1

Figure 5: K-Map Example

There are two groups, the two 1s in the first row and the two 1s in the second. From the top group we get the term $A'B'C + A'BC = A'C$. From the bottom group we get the term $ABC + ABC' = AB$, and summing these product terms we get the final expression $A'C + AB$.

Timing Hazards

Timing hazards occur in digital circuits when changes in input signals cause changes in the output to unexpected values before settling to the expected value. They occur because of propagation delays.

Static Hazards

Static hazards occur when an output temporarily glitches to an incorrect value before settling. They are classified as:

- **Static-1 Hazard**: The output should remain at logic '1', but briefly drops to 'o'.
- **Static-0 Hazard**: The output should remain at logic 'o', but briefly rises to '1'.

These hazards arise due to differing propagation delays through alternative logic paths.

The circuit in figure 6 may exhibit a hazard when inputs change due to different propagation delays.

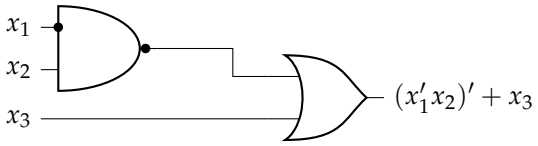

Figure 6: Hazardous Circuit

Dynamic Hazards

Dynamic hazards occur when an output makes multiple transitions before settling to the correct value. This happens when there are multiple paths with differing delays, leading to oscillations in the output before stabilization.

Mitigation Techniques

To minimize timing hazards, designers employ several techniques:

- **Redundant Logic**: Adding extra logic gates to ensure that different paths remain synchronized.
- **Synchronizing Inputs**: Using flip-flops and registers to control when signals change.
- **Glitch Filtering**: Employing filters or careful circuit layout to suppress transient glitches.
- **Delay Balancing**: Ensuring all paths experience similar propagation delays to prevent unintended transitions.

Multiplexers

A multiplexer (MUX) is a combinational logic circuit that selects one of many input signals and forwards the selected input to a single output.

A multiplexer has multiple input lines, a set of selection lines, and a single output line. The selection lines determine which input is forwarded to the output. For an n-to-1 multiplexer, there are n input lines and $\log_2 n$ selection lines.

2-to-1 Multiplexer

The simplest multiplexer is a 2-to-1 multiplexer, which has two inputs (I_0, I_1), one selection line (S), and one output (Y). The function of a 2-to-1 MUX is defined as:

$$Y = S \cdot I_1 + \bar{S} \cdot I_0$$

This means:

- If $S = 0$, then $Y = I_0$.
- If $S = 1$, then $Y = I_1$.

4-to-1 Multiplexer

A 4-to-1 multiplexer has four inputs (I_0, I_1, I_2, I_3), two selection lines (S_0, S_1), and one output (Y). The output equation is:

$$Y = (\bar{S_1} \cdot \bar{S_0} \cdot I_0) + (\bar{S_1} \cdot S_0 \cdot I_1) + (S_1 \cdot \bar{S_0} \cdot I_2) + (S_1 \cdot S_0 \cdot I_3)$$

Multiplexers in Digital Systems

Multiplexers are used in:

- Data selection and routing.
- Logic function implementation.
- Memory addressing.
- Communication systems.

Implementation Using Logic Gates

A multiplexer can be implemented using basic logic gates such as AND, OR, and NOT gates. For example, a 2-to-1 multiplexer can be constructed using two AND gates, one OR gate, and one NOT gate.

Decoders and Encoders

Decoders

Decoders are combinational circuits that convert binary information from n input lines to a maximum of 2^n unique output lines. They are used in various applications such as memory address decoding, data demultiplexing, and seven-segment displays. A common type of decoder is the 2-to-4 line decoder, which has 2 input lines and 4 output lines. The output lines are mutually exclusive, meaning only one output line is active (high) at any given time based on the binary value of the inputs.

Encoders

Encoders are combinational circuits that perform the inverse operation of decoders. They convert 2^n input lines into an n-bit binary code. Encoders are used in applications such as priority encoding, data compression, and binary coding of input signals. A common type of encoder is the 4-to-2 line encoder, which has 4 input lines and 2 output lines. The encoder generates a binary code corresponding to the active input line.

In practice, encoders often include additional features such as priority encoding, where the encoder outputs the binary code of the highest-priority active input.

State Machines

State machines consist of states, inputs, outputs, and transition functions that determine how the machine moves between states based on inputs.

A state machine can be defined mathematically as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states
- Σ is a finite set of input symbols
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states

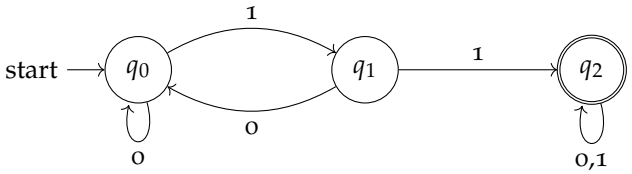

Figure 7: A simple finite state machine with three states

Moore and Mealy Machines

In digital design, state machines are typically categorized into two primary types: Moore machines and Mealy machines.

Moore Machines

In a Moore machine, outputs depend solely on the current state. The output function can be defined as $\lambda : Q \to \Delta$, where Δ is the output alphabet. This makes Moore machines synchronous and deterministic, as outputs change only when the state changes at clock edges.

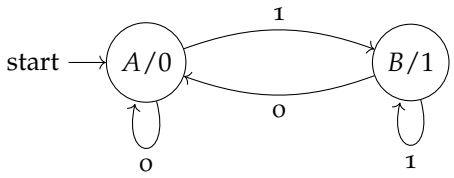

Figure 8: Moore machine with outputs indicated in each state

Mealy Machines

In a Mealy machine, outputs depend on both the current state and the current input. The output function is defined as $\lambda : Q \times \Sigma \to \Delta$. As a result, Mealy machines can change their outputs without changing states, providing faster response to inputs.

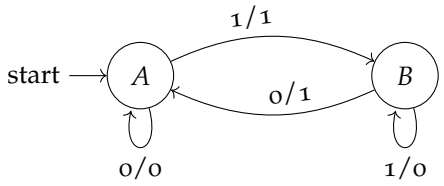

Figure 9: Mealy machine with input/output pairs on transitions

State Encoding and Implementation

State machines are implemented in hardware using memory elements (flip-flops) to store the current state and combinational logic to determine the next state and outputs. The process involves:

1. Drawing the state diagram
2. Creating a state transition table
3. Encoding the states using flip-flops
4. Deriving the next-state and output functions
5. Implementing the design using logic gates and flip-flops

The number of flip-flops required equals $\lceil \log_2 n \rceil$, where n is the number of states.

Sequential Elements: Flip-Flops and Latches

The core building blocks of state machines are sequential elements that can store state information. The two primary types are latches and flip-flops.

Latches

Latches are level-sensitive devices that change their output as long as a control signal (enable) is active. The basic latch types are:

- SR Latch (Set-Reset)
- D Latch (Data)

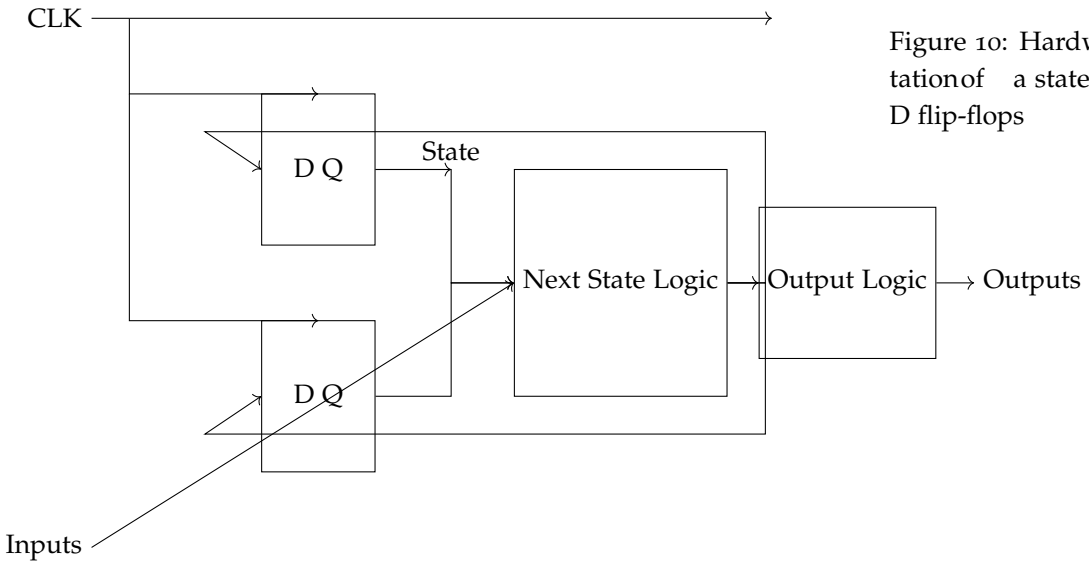

Figure 10: Hardware implementation of a state machine using D flip-flops

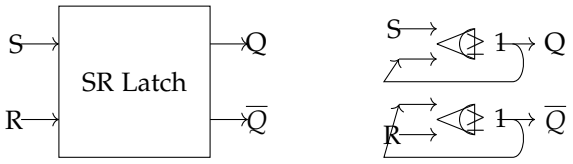

Figure 11: SR Latch symbol and implementation using NOR gates

Flip-Flops

Flip-flops are edge-triggered devices that change state only at the active edge (rising or falling) of a clock signal. Common types include:

- D Flip-Flop
- JK Flip-Flop
- T Flip-Flop

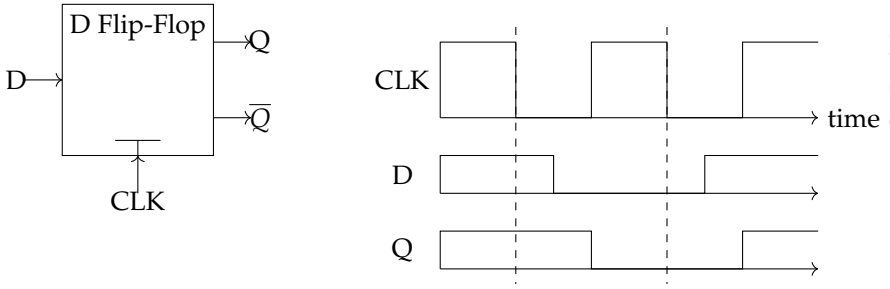

Figure 12: D Flip-Flop symbol and timing diagram showing edge-triggered behavior

State Machine Design Methodology

The design of state machines follows a systematic approach:

1. Problem analysis and specification
2. State diagram creation
3. State minimization (if necessary)
4. State encoding
5. Next-state and output logic derivation
6. Hardware implementation
7. Verification and testing

Example: Sequence Detector

Consider a sequence detector that outputs 1 when it detects the sequence "101" in a serial input stream.

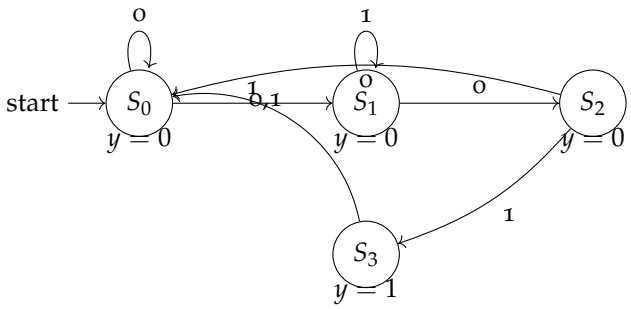

Figure 3: State diagram for a "101" sequence detector implemented as a Moore machine

The corresponding state table would be:

Current State	Input	Next State	Output
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_2	0
S_1	1	S_1	0
S_2	0	S_0	0
S_2	1	S_3	0
S_3	0	S_0	1
S_3	1	S_0	1

Table 4: State transition table for the "101" sequence detector

Special Types of State Machines

Registers and Counters

Registers and counters are specialized state machines with specific purposes. A register stores binary information, while a counter progresses through a sequence of states.

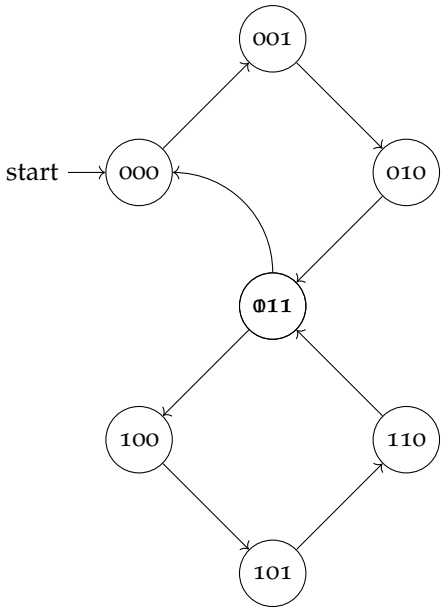

Figure 14: State diagram of 3-bit binary counter

Arithmetic

Signed Numbers

In SystemVerilog to this point, we have been working with *unsigned* (positive) numbers. We now introduce the concept of *signed* binary numbers, which will enable us to perform subtraction and other useful operations.

Signed numbers are typically represented using the *two's complement* method. In this representation:

- The most significant bit (MSB) is the *sign bit*, where 0 indicates a positive number and 1 indicates a negative number.
- The value of a signed number can be calculated as:

$$\text{Value} = -2^{n-1} \cdot b_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot b_i$$

  where b_i represents the bits of the binary number.

  For example, in a 4-bit two's complement system:

- 0110 represents 6 (positive).
- 1010 represents -6 (negative).

Adders and Subtractors

Adders and subtractors are fundamental components in digital arithmetic. They are used to perform addition and subtraction operations on binary numbers.

A *half adder* is a combinational circuit that adds two binary digits and produces a sum and a carry. The truth table for a half adder is:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 5: Truth table for a half adder

The logic equations for the sum and carry are:

$$\text{Sum} = A \oplus B, \quad \text{Carry} = A \cdot B$$

A *full adder* extends the half adder by including a carry-in bit. It adds three binary digits (A, B, and Carry-in) and produces a sum and

a carry-out. The logic equations for the full adder are:

$$\text{Sum} = A \oplus B \oplus \text{Carry-in}$$

$$\text{Carry-out} = (A \cdot B) + (\text{Carry-in} \cdot (A \oplus B))$$

Subtraction can be performed using an adder by taking the two's complement of the number to be subtracted. For example, to compute $A - B$, we calculate $A + (-B)$, where $-B$ is the two's complement of B.

Comparators

Comparators are used to compare two binary numbers and determine their relationship. The outputs of a comparator typically indicate whether one number is less than, equal to, or greater than the other.

For two numbers A and B:

- $A < B$: Output is 1 if A is less than B.
- $A = B$: Output is 1 if A is equal to B.
- $A > B$: Output is 1 if A is greater than B.

A simple 1-bit comparator can be implemented using logic gates:

$$A < B = \overline{A} \cdot B, \quad A = B = A \oplus B, \quad A > B = A \cdot \overline{B}$$

Shifters and Multipliers

Shifters and multipliers are used for more complex arithmetic operations.

Shifters move the bits of a binary number to the left or right. There are three main types of shifters:

- **Logical Shift**: Shifts the bits left or right, filling the vacated positions with 0s.
- **Arithmetic Shift**: Preserves the sign bit when shifting right, filling the vacated positions with the sign bit.
- **Circular Shift (Rotate)**: Rotates the bits, wrapping around the bits that are shifted out.

For example, a logical left shift of 1010 by 1 position results in 0100.

Multipliers perform binary multiplication. The simplest method is repeated addition, but more efficient algorithms like the Booth multiplier or Wallace tree are used in hardware implementations.

For example, multiplying 101 (5 in decimal) by 11 (3 in decimal) yields 1111 (15 in decimal).