Design of Digital Circuits
Summary

April 20, 2020

Chapter 1

From Zero to One

1.1 Managing Complexity

Abstraction: The levels of abstraction for an electronic computer:

- **physics:** The motion of electrons
- **electronic devices:** Transistors which have connection points (terminals) and can be modeled by the relationship between voltage and current as measured at each terminal.
- **analog circuits:** Devices which are assembled to create components such as amplifiers. They input and output a continuous range of voltages
- **Digital circuits:** e.g logic gates restrict the voltages to descrete ranges which we use to indicate 0 and 1.
- **Microarchitecture:** Links the logic and architecture levels of abstraction. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture-
- **Architecture:** Describes the computer from the programmers perspective. A particular architecture can be implemented by one of many different microarchitectures.
- **Operating system:** Handles low-level details such as accessing a hard drive or managing memory.
- **Application software:** Uses the facilities provided by the operating system to solve a problem for the user.

Discipline: the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction.

The Three- Y's:

- **Hierarchy:** involves dividing a system into modules then further subdividing each of these modules until the pieces are easy to understand
- **Modularity:** states that the modules have well-defines functions and interfaces so that they connect together easily without unanticipated side effects
- **Regularity:** seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed

1.2 Digital Abstraction:

Discrete-valued variables: Variables with a finite number of distinct values. Most electronic computers use a binary representation in which a high voltage indicates a 1 and a low voltage indicates a 0. The amount of information D in a descrete valued variable with N distinct states is measured in units of bits as:

$$D = log_2 N \text{ bits}$$

hence a binary variable conveys $log_2 2 = 1$ bit of information

1.3 Number Systems

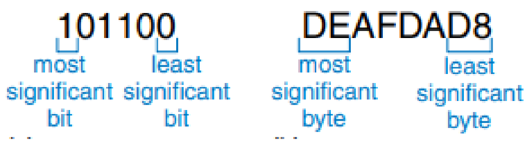
Binaray Numbers: Bits represent one of two values 0 or 1 and are joined together to form binary numbers. Each column of a binary number has twice the weight of the previous column, hence they are base 2 (the base is denoted as subscript e.g 10110_2).

Hexadecimal Numbers: groups of four bits i.e base 16. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Bytes,Nibbles,etc: A group of eight bits is called a byte. The size of objects stored in computer memories is measured in bytes rather than bits. A group of four bits, is called a nibble i.e one hexadecimal digit stores one nibble. Microprocessors handle data in chunks called words which the size of depends on the architecture of the microprocessor (64 bit processors indicate that they operate on 64-bit words)

Most/Least significant bit (lsb/msb): Within a group of bits in the 1's column is called the lsb and the bit at the other end is called the msb



Estimating Powers of Two:

- $2^{10} \approx 10^3$
- $2^{20} \approx 10^6$
- $2^{30} \approx 10^9$

Binary Addition
:

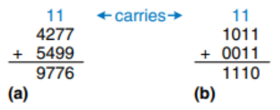


Figure 1.8 **Addition examples showing carries: (a) decimal (b) binary**

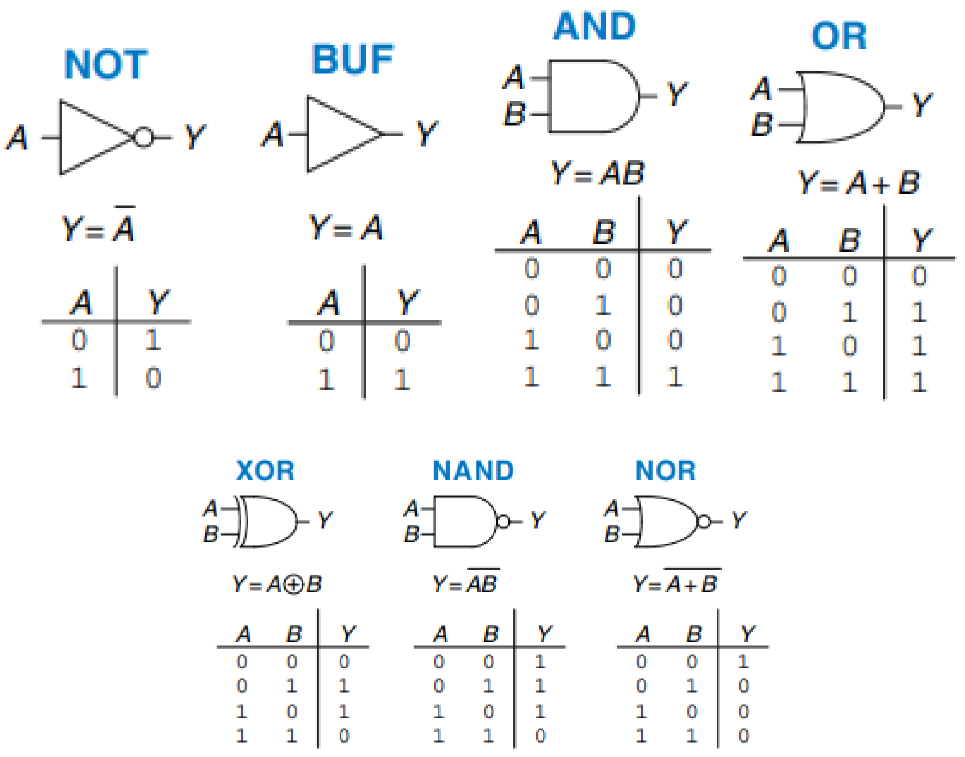
Signed Binary Numbers: The two most widely employed systems to represent signed binary numbers are:

- Sign/magnitude: The msb is used as the sign and the remaining N-1 bits is the absolute value. 0 indicates positive and 1 indicates negative. Ordinary binary addition does not work for sign/magnitude numbers. (0 has two representations)
- Two's complement: Identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. Zero has a single representation and ordinary addition works. The sing of a two's complement number is reversed in a process called taking the two's complement. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. When addint N-bit numbers, the carry out of the Nth bit is discarded. Subtraction is performed by taking the two's complement of the second number, then adding. The range of an N-bit twos complement number spans $[-2^{N-1}, 2^{N-1} - 1]$

1.4 Logic Gates

Logic Gates: are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input and the output. Inputs are drawn on the left or top and recieve a letter near the beginning of the alphabet and the outputs on the right or bottom and recieve the letter Y.

Truth table: Lists the inputs on the left and the corresponding output on the right. One row is designated for each possible combination of inputs.



NOT

A	Y
0	1
1	0

BUF

A	Y
0	0
1	1

AND

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NAND

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 1.1: Common Gates

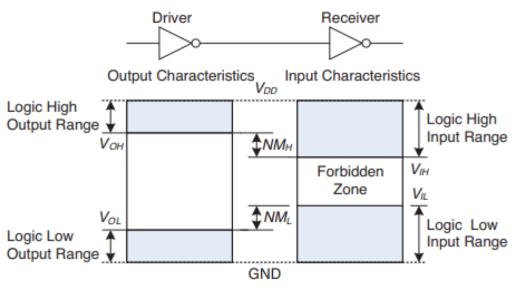
Bubble: circle on the output of a logic gate indicating an inversion.

N-input XOR: Produces true when an odd number of inputs are TRUE.

1.5 Beneath the digital Abstraction

Supply Voltage: The lowest voltage in a system is called ground or GND (0V). The highest voltage comes from the power supply and is usually called V_{DD}.

Logic Levels and Noise Margins: used to map continuous variables onto discrete binary variables.



The driver produces a LOW(0) output in the range of 0 to V_{OL} or a HIGH(1) in the range of V_{OH} to V_{DD}. If the reciever gets an input in the range of 0 to V_{IL} it will consider the input to be LOW, if it gets an input in the range of V_{IH} to V_{DD} it will consider the input to be HIGH. If the receivers input falls into the forbidden zone the behaviour of the gate is unpredictable. $V_{OH}, V_{OL}, V_{IH}, V_{IL}$ are called the output and input high and low logic levels. The noise margin is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input.

$$NM_L = V_{IL} - V_{OL}$$
$$NM_H = V_{OH} - V_{IH}$$

DC transfer characteristics: of a gate describes the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up. A reasonable place to choose logic levels is where the slop of the transfer characteristics $\frac{dV(Y)}{dV(A)} = -1$ These two points are called **unity gain points**. This usually maximizes the noise margins.

Static Discipline: Given logically valid inputs, every circuit element will produce logically valid outputs. The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels. Gates are grouped into logic families such that all gates in a logic family obey the static discipline when used with other gates in the family.

1.6 CMOS Transistors

Transistors: Electrically controlled switches that turn on or off when a voltage or current is applied to a control terminal. There are two main type of transistors:

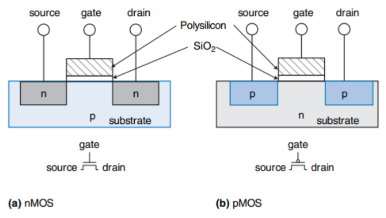
- bipolar junction transistors
- metal-oxide-semiconductor field effect transistors (MOSFETs or MOS transistors)

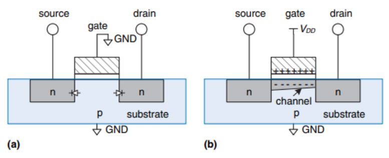
Semiconductors: MOS transistors are built from silicon. The conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants(impurities in the silicon lattice). Adding a dopant with more electrons than the silicon can bind to creates a negative charge which can move around the lattice and is called an n-type dopant. Adding a dopant with less electrons creates a positive charge and hence called a p-type dopant.

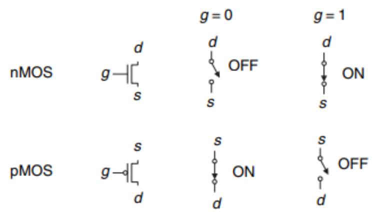
Diodes: Junction between p-type and n-type silicon. The p-type region is called the anode and the n-type region called the cathode. When voltage on the anode rises above the voltage on the cathode, the diode is forward biased, and current flows through the diode from the anode to the cathode otherwise it is reverse biased and no current flows.

Capacitors: Consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric charge and the other conductor accumulates the opposite charge -Q. The capacitance C of the capacitor is the ratio of charge to voltage $C = \frac{Q}{V}$. More capacitance means that a circuit will be slower and require more energy to operate(charging and discharging takes time).

nMOS and pMOS Transistors: n-type transistors called nMOS, have regions of n-type dopants adjacent to the gate and built on a p-type semiconductor substrate. pMOS transistors are the opposite. Chips which have both nMOS and pMOS transistors are called complementary MOS or CMOS.







Chapter 2

Combinational Logic Design

2.1 Basic Definitions:

functional specification: Describes the raltionship between inputs and outputs

timing specification: Describes the delay between inputs changing and outputs responding

combinational circuit: The output depends only on the current values of the inputs. (e.g a logic gate) A combinational circuit is memoryless.

sequential circuit: The output depends on both current and previous values of the inputs. Sequential circuits have memory.

Bus: A bundle of multiple signals. A single line with a slash through it and a number next to it indicates a bus with the number specifying the number of signals

Combinational composition: A circuit is combinational if:

- Every circuit element itself is combinational
- Every node(wire) of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

2.2 Boolean Equations

complement: The complement of a variable/literal A is its inverse $\overline{A}$

product/implicant: The AND of one or more literals

minterm: A product involving all of the inputs to the function

sum: The OR of one or more literals

maxterm: A sum involving all the inputs to the function

order of operations: NOT has the highest precedence followed by AND then OR

Sum of Products Form: Given a truth table we can create a boolean equation by summing each of the minterms for which the outpu Y is TRUE.

Product of Sums Form: We can write a boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE.

2.3 Boolean Algebra

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots)$	De Morgan's Theorem

Rules for bubble pushing:

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs
- Pushing bubbles on all gate inputs forward toward the output puts a bubble on the output
- Begin at the output of the circuit and work toward the inputs
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output instead of the complement of the output
- Working backward,draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceeding gate with an output bubble. If the current gate does not have an input bubble draw the preceding gate without an output bubble

2.4 From Logic to Gates

schematic: A diagram of a digital circuit showing the elements and the wires that connect them together. The guidlines for drawing schematics:

- Inputs are on the left (or top) side of a schematic
- Outputs are on the right (or bottom) side of a schematic
- Whenever possible, gates should flow from left to right
- Straight wires are better to use than wires with multiple corners
- Wires always connect as a T junction
- A dot where wires cross indicates a connection between the wires
- Wires crossing without a dot make no connection

Dont Cares: The symbol X is used to describe inputs that the output doesn't care about. When creating a boolean equation from a truth table with dont cares we can use the sum of products and ignore inputs with X's.

2.5 Multilevel Combinational Logic:

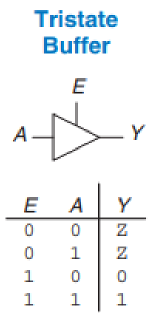
two-level logic: Logic in sum-of-products because it consists of literals connected to a level of AND gates connected to a level of OR gates. Circuits with more than two levels of logic may use less hardware.

2.6 X's and Z's

Illegal Value: X The symbol X indicates that the circuit node has an unkown or illegal value. This happens if it is being driven to both 0 and 1 at the same time. This situation is called **contention** and is considered an error and must be avoided. The voltage at contention is usually in the forbidden zone. In a truth table X is defined as "dont care" i.e it is unimportant if it is a 0 or a 1.

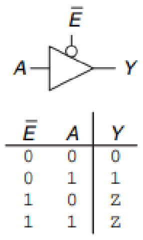
Floating Value: Z This symbol indicates that a node is being drigven neither HIGH nor LOW. A floating node does not always mean there is an error in the circuit, so long as some other circuit element drives the node to a valid logic level when its value is relevant

Tristate Buffer: Commonly used on busses that connect multiple chips. Only one chip at a time is allowed to assert its enable signal to drive a value onto the busses.



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

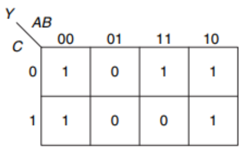
Figure 2.41 Tristate buffer with active low enable

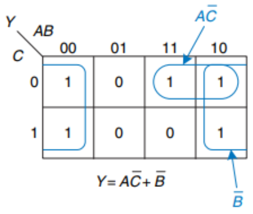
2.7 Karnaugh Maps

Karnaugh maps (K-maps) A graphical method for simplifying Boolean equations. Each square in the K-map corresponds to a row in the truth table and contains the value of the output Y for that row (each square represents a single minterm). The combinations of the row are in gray code (00,01,11,10) which ensures entries differ only in a single variable. The K-map wraps around i.e squares on the far right are effectively adjacent to the squares on the far left.

Logic Minimization with K-Maps: We can minimize logic by circling all the rectangular blocks of 1's in the map, using the fewest number of circles. Each circle should be as large as possible. Then read off the implicants that were circled. Rules for finding a minimized equation from a K-map are as follows:

- use the fewest circles necessary to cover all the 1's
- all the squares in each circle must contain 1's
- Each circle must span a rectangular block that is a power of 2
- each circle should be as large as possible
- a circle may wrap around the edges of the K-map
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used

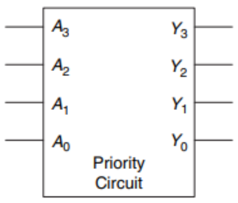


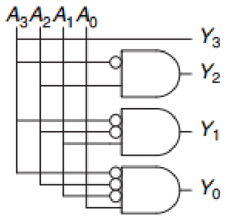


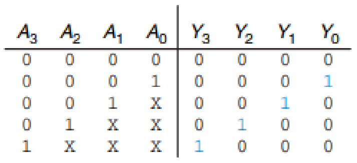
Dont Cares: Dont cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never happen. Such outputs can be treated as either 0's or 1's at the designers discretion. In a K-map they can be circled if they help cover the 1's with fewer or larger circles.

2.8 Combinational Building Blocks:

Priority Circuit: Output with the highest priority signal becomes 1 the rest 0.

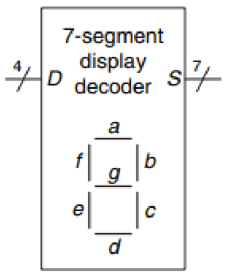


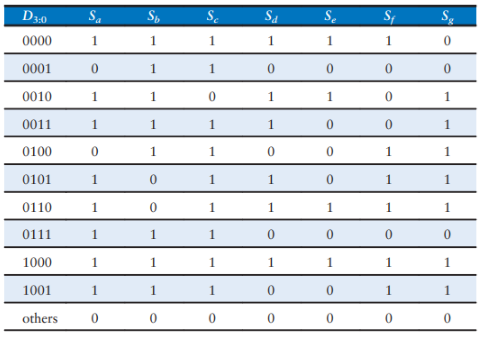




A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

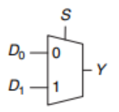
Seven Segment Display Decoder: Takes a 4-bit data input $D_{3:0}$ and produces seven outputs to control light emitting diodes to display a digit from 0 to 9.

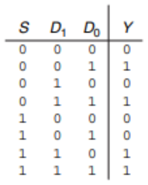




$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

Multiplexers: Chooses an outpu from several possible inputs based on the value of a select signal. (Multiplexer is also called a mux) An N:1 multiplexer needs $log_2 N$ select lines. A 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs.





S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

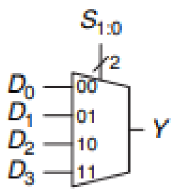
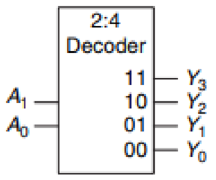
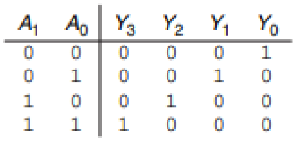


Figure 2.57 **4:1 multiplexer**

Decoder: A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. An N-input function with M 1's in the truth table can be built with an $N : 2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table.





A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

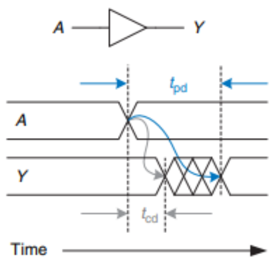
2.9 Timing

delay: The time it takes the output to change in response to an input change. Delay is measured from the 50% point of the input signal to the 50% point of the output signal

rising/falling edge: The transition from LOW to HIGH (rising) and HIGH to LOW (falling).

propagation delay: denoted t_{pd} is the maximum time from when an input changes until the output or outputs reach their final value. It is the sum of the propagation delays through each element on the critical path.

contamination delay: denoted t_{cd} is the minimum time from when an input changes until any output starts to change its value. The sum of contamination delays through each element on the short path.



Critical path: The longest and slowest path in a circuit. The path is critical because it limits the speed at which the circuit operates.

Short path: The shortest and therefore the fastest path through the circuit.

Glitches\Hazards : A single input transition which causes multiple output transitions. In a K-map the transition across the boundry of two prime implicants(bubbles) indicates a possible glitch. To fix this we add another circle that covers that prime implicant boundary

Chapter 3

Sequential Logic Design

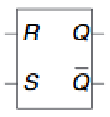
State: The state of a digital sequential circuit is a set of bits called state variables that contain all the information about the past necessary to explain the future behavior of the circuit. An element with N stable states conveys $log_2 N$ bits fo information (a bistable element stores on bit)

3.1 Latches and Flip-Flops

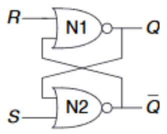
bistable: An element with two stable states

cross-coupled: The input of one element is the output of the other element and vice versa

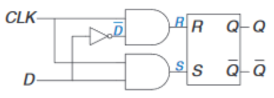
SR-Latch: Consists of two cross-coupled NOR gates. The latch has two inputs, S (set) and R (reset). Setting a bit means to make it TRUE. To reset a bit means to make it FALSE.



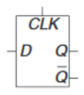
Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0



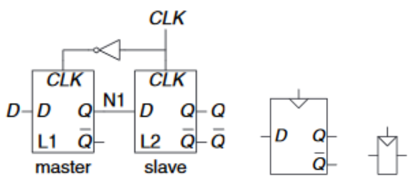
D Latch: Consists of two inputs. The data input D, controls what the next state should be. The clock input CLK controls when the state should change. When CLK = 1 the latch is transparent the data at D flows through to Q as if the latch were just a buffer. When CLK = 0, the latch is opaque, it blocks new data from flowing through to Q, and Q retains the old value



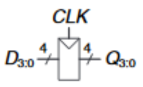
CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0



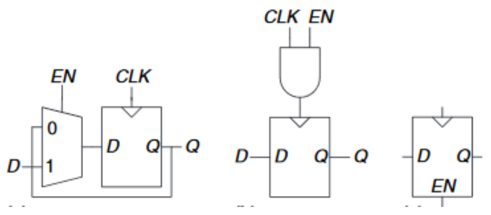
D Flip-Flop: Is built from two back to back D latches controlled by complementary clocks. The first latch L1 is called the master. The second latch L2, is called the slave. The node between them is named N1. When the $\overline{Q}$ is not needed the condensed symbol is used. The D flip flop copies D to Q on the rising edge of the clock, and remembers its state at all other times. Triangle in the symbols denotes an edge-triggered clock input.



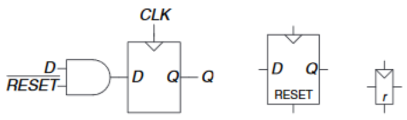
Register: An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.



Enabled Flip-Flop: An enabled flip-flop adds another input EN to determine whether data is loaded on the clock edge. When EN is TRUE the enabled flip-flop behabes like an ordinary D flip-flop. When EN is FALSE the enabled flip-flop ignores the clock and retains its state. Enabled flip flops are useful when we wish to load a new value into a flip flop only some of the time, rather than on every clock edge.



Resettable Flip-flop: A resettable flip flop adds another input called RESET. When RESET is FALSE it behaves like an ordinary D flip-flop. When RESET is TRUE the resettable flip-flop ignores D and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e 0) into all the flip flops in a system when we first turn it on. They can be synchronously (reset themselves only on the rising edge of CLK) or asynchronously resettable (reset as soon as RESET becomes TRUE, independent of CLK). Active low signal means that the reset signal performs its function when it is 0, not 1.



3.2 Synchronous Logic Design

cyclic paths: Outputs are fed directly back to inputs. These circuits are sequential (Combinational logic has no cyclic paths and no races)

synchronized: The state of a system only changes at clock edge

sequential circuit: A circuit with a finite set of discrete states $\{S_0, S_1, \ldots, S_{k-1}\}$

synchronous sequential circuit: A sequential circuit which has a clock input whose rising edge indicate a sequenc of times at which state transitions occur. The timing specification consists of an upper bound t_{pcq} and a lower bound t_{ccq} on the time from the rising edge of the clock until the output changes, aswell as setup and hold times t_{setup}, t_{hold} which indicate when the inputs must be stable relative to the rising edge of the clock. A circuit is a synchronous sequential circuit if:

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

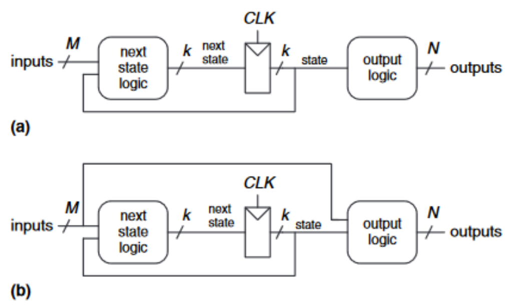
Sequential circuits that are not synchronous are called **asynchronous**. Current state variable is denoted S and the next state variable S'.

3.3 Finite State Machines(FMS)

FSM's: A circuit with k registers can be in one of 2^k unique states. An FSM has M inputs, N outputs and k bits of state. It also recieves a clock and optionally a reset signal. An FSM consists of two blocks of combinational logic, next state logic and output logic and a register that stores the state. On each clock edge the FSM advances to the next state, which was computed based on the current state and inputs.

Moore Machines: FSM where the outputs depend only on the current state of the machine. (a)

Mealy Machines: FSM where the output depends on both the current state and the current inputs (b)



State Transition diagram: Indicates all the possible states of a system and the transitions between these states. Circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock and is not shown in the diagram because a clock is always present in a synchronous sequential circuit. The arcs are labeled with input indicating how it reaches the next state. The value that the outputs have while in a particular state are indicated in the state (for Moore machines). For Mealy machines, the outputs are labeled on the arcs instead of in the circles.

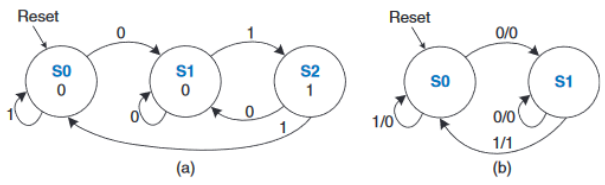
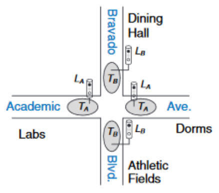


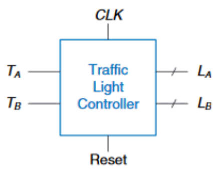
Figure 3.30 **FSM state transition diagrams: (a) Moore machine, (b) Mealy machine**

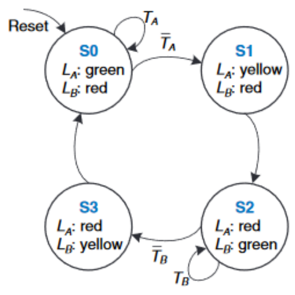
State Transition Table: Indicates for each state tand input what the next state should be. The table uses dont care symbols (X) whenever the next state does not depend on a particular input. Reset is omitted from the table. The states and outputs must be assigned binary encodings

Output Table: Indicates for each state what the output should be in that state

Example: Traffic Light: Given inputs T_A, T_B (sensors detecting when people are present) and outputs L_A, L_B (Traffic lights) create the State Machine circuit.







(a) State Transition diagram

Table 3.1 **State transition table**

Current State S	Inputs T_A	T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 **State encoding**

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 **Output encoding**

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Table 3.4 **State transition table with binary encodings**

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 **Output table**

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

From the state transition table and output table we can get the boolean equations needed for the next states and outputs.

- $S'_1 = S_1 \oplus S_0$
- $S'_0 = \overline{S}_1\overline{S}_0\overline{T}_A + S_1\overline{S}_0\overline{T}_B$
- $L_{A1} = S_1$
- $L_{A0} = \overline{S}_1 S_0$
- $L_{B1} = \overline{S}_1$
- $L_{B0} = S_1 S_0$

Using these formulas we can build the next state logic and the output logic and hence the complete circuit.

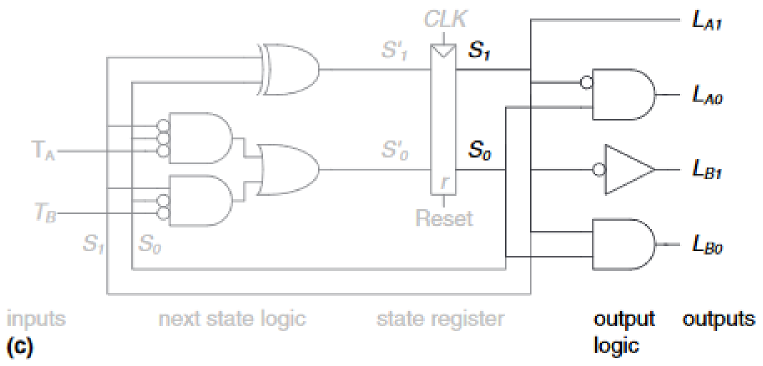


Figure 3.26 **State machine circuit for traffic light controller**

Binary encoding: Each state is represented as a binary number. A system with K states only need $log_2 K$ bits of state

One-hot encoding: A seperate bit of state is used for each state. Only one bit is "hot" (TRUE) at any time. This encoding requires more flip flops than binary encoding, however the next-state and output logic is often simpler so fewer gates are required. The best encoding choice depends on the specific FSM.

Deriving an FSM from a schematic: Deriving the state transition diagram from a schematic follows nearly the reverse process of FSM design:

- Examine circuit, stating inputs,outputs, and state bits
- Write next state and output equations
- Create next state and output tables
- Reduce the next state table to eliminate unreachable states
- Assign each valid state bit combination a name
- Rewrite next state and output tables with state names
- Draw state transition diagram
- State in words, what the FSM does

3.4 Timing of sequential Logic

aperture time: The total time for which the input must remain stable, consisting of the time t_setup before the rising edge of the clock and t_hold after the rising edge. Dynamic discipline states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge, hence we guarantee that the flip flops sample signals while they are not changing

clock to Q contamination delay: denoted $t_c cq$ time after clock rises for which the outputs may start to change

clock to Q propagation delay: denoted $t_p cq$ time after clock rises for which the outputs must settle to final value

clock period/cycle time: denoted T_c is the time between rising edges of a repetitive clock signal

clock frequency: $f_c = \frac{1}{T_c}$. Increasing the clock frequency increases the work that a digital system can accomplish per unit of time. Frequency is measured in units of Hertz (Hz)

setup time constraint/max-delay constraint: $t_{pd} \leq T_c - (t_{pcq} + t_{setup})$. Ideally the entire cycle time T_c would be available for useful computation in the combinational logic t_{pd} however the **sequencing overhead** $(t_{pcq} + t_{setup})$ of the flip flop cuts into this time.

hold time constraint/min-delay constraint: $t_{cd} \geq t_{hold} - t_{ccq}$. This constraint limits the minimum delay through combinational logic. (i.e if two flip flops are connected back to back then $t_{cd} = 0$) In general adding buffers can usually (not always) solve hold time problems without slowing the critical path.

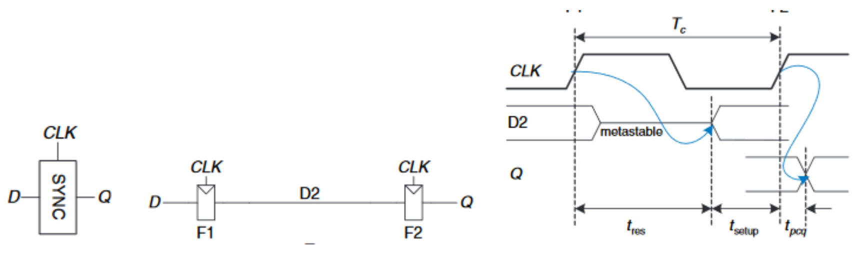
Clock skew: The variation at which the clock signal reaches all the registers. The setup and hold time constraints become:

- $T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$
- $t_{cd} \geq t_{hold} + t_{skew} - t_{ccq}$

Clock skew effectively increases both the setup time and the hold time. It adds to the sequencing overhead, reducing the time available for useful work in the combinational logic. It also increases the required minimum delay through the combinational logic.

Metastable State: When a flip-flop samples an input that is changing during its aperture the output Q may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. Eventually it will resolve the output to a stable state(either 0 or 1). The resolution time (time required to reach the stable state) is unbounded. Every bistable device has a metastable state between the two stable states.

Synchronizer: A device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time. The output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.



A synchronizer fails if Q becomes metastable. This may happen if D2 has not resolved to a valid level by the time it must setup at F2.

3.5 Parallelism

Token: A group of inputs that are processed to produce a group of outputs.

Latency: Time required for one token to pass through the system from start to end.

Throughput: The number of tokens that can be produced per unit of time. Throughput can be improved by processing several tokens at the same time.

Spatial parallelism: Multiple copies of the hardware are provided so that multiple tasks can be done at the same time.

Temporal parallelism/Pipelining: A task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a different task will be in each stage at any given time so multiple tasks can overlap.

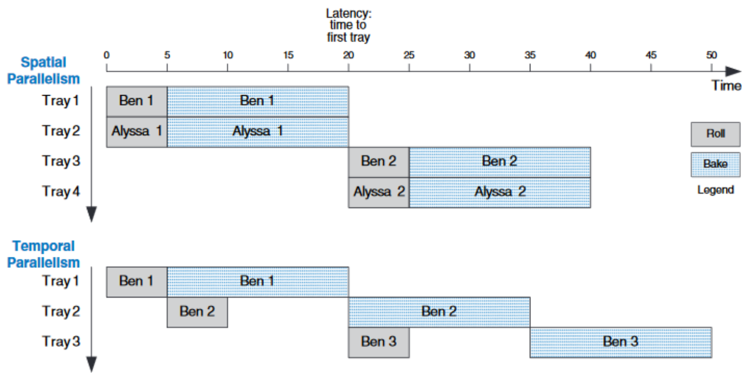


Figure 3.57 Spatial and temporal parallelism in the cookie kitchen

Chapter 4

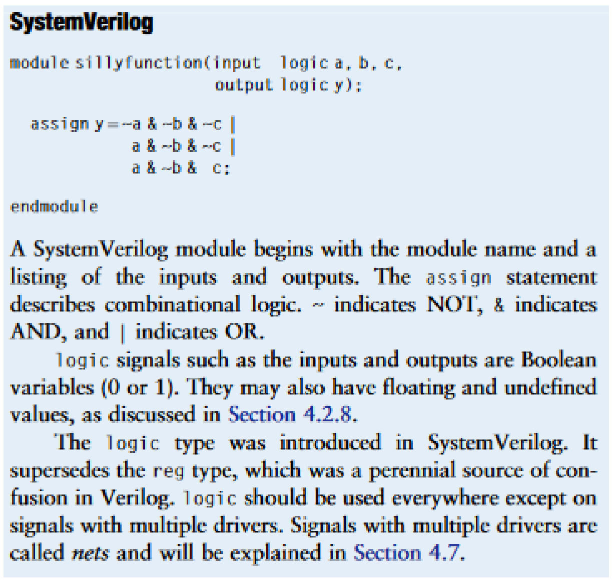
Hardware Description Languages

4.1 Basics

Module: A block of hardware with inputs and outputs (e.g AND gate, multiplexer, priority circuit etc.)

Behavioral models: Describe what a module does.

Structural models: Describe how a module is built from simpler pieces.



SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

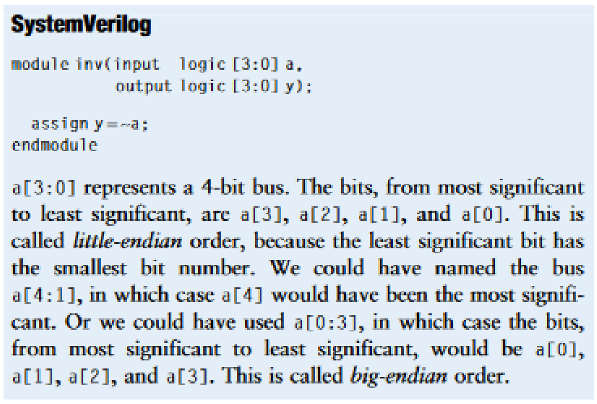
A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. ~ indicates NOT, & indicates AND, and | indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in Section 4.7.

4.2 Combinational Logic

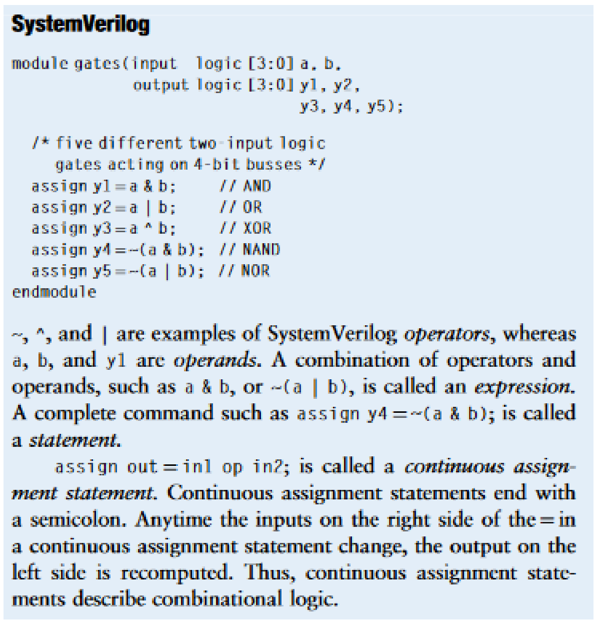
Inverters and Logic Gates: Bitwise operators act on single bit signals or on multi-bit busses.



SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

a[3:0] represents a 4-bit bus. The bits, from most significant to least significant, are a[3], a[2], a[1], and a[0]. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus a[4:1], in which case a[4] would have been the most significant. Or we could have used a[0:3], in which case the bits, from most significant to least significant, would be a[0], a[1], a[2], and a[3]. This is called *big-endian* order.



SystemVerilog

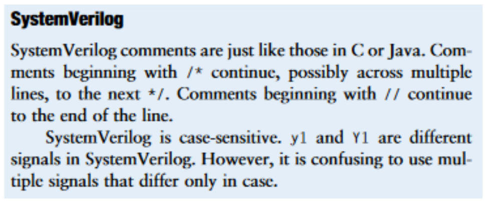
```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR
endmodule
```

~, ^, and | are examples of SystemVerilog *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a & b, or ~(a | b), is called an *expression*. A complete command such as assign y4 = ~(a & b); is called a *statement*.

assign out = in1 op in2; is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

Comments and Name giving: Be consistent in the use of capitalization and underscores in signal and module names. Module and signal names must not begin with a digit.

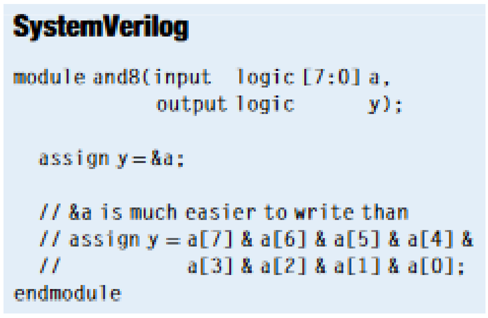
Reduction Operators: Reduction operators imply a multiple input gate acting on a single bus.



SystemVerilog

SystemVerilog comments are just like those in C or Java. Comments beginning with /* continue, possibly across multiple lines, to the next */. Comments beginning with // continue to the end of the line.

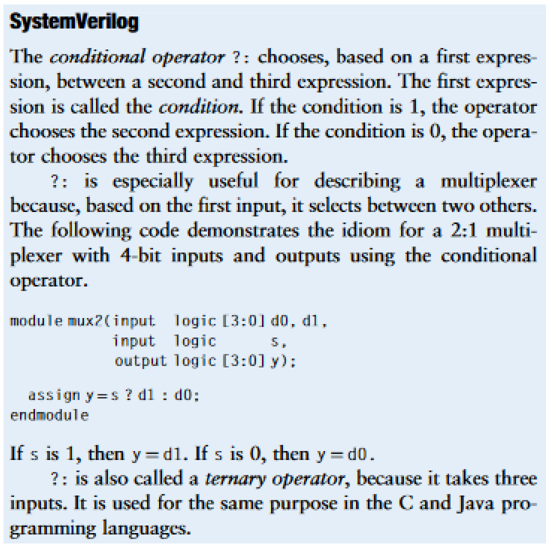
SystemVerilog is case-sensitive. y1 and Y1 are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.



SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

Conditional Assignment: Selects the output from among alternatives based on an input called the condition.



SystemVerilog

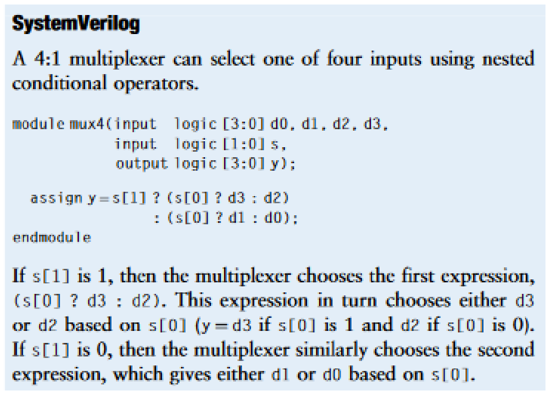
The *conditional operator* ?: chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

?: is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If s is 1, then y = d1. If s is 0, then y = d0.

?: is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.



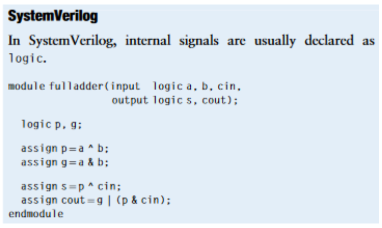
SystemVerilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If s[1] is 1, then the multiplexer chooses the first expression, (s[0] ? d3 : d2). This expression in turn chooses either d3 or d2 based on s[0] (y = d3 if s[0] is 1 and d2 if s[0] is 0). If s[1] is 0, then the multiplexer similarly chooses the second expression, which gives either d1 or d0 based on s[0].

internal variables: They are neither inputs nor outputs but are used only internal to the module. (local variables in programming languages). The order in which statements are written does not matter HDL assignment statements are evaluated any time the inputs, signals on the right hand side change their value.

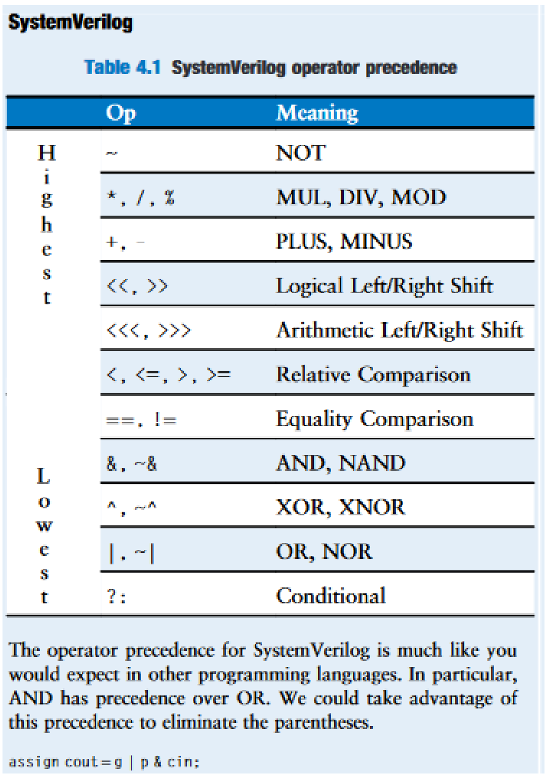


SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Precedence and Numbers: Underscores in numbers are ignored and can be helpful in breaking long numbers into mre readable chunks



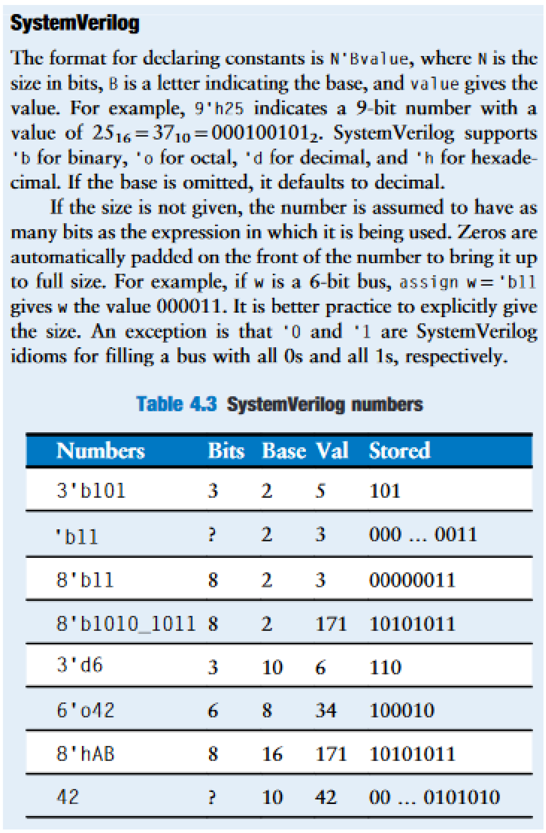
SystemVerilog

Table 4.1 SystemVerilog operator precedence

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
Lowest	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
	?:	Conditional

The operator precedence for SystemVerilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses.

```
assign cout = g | p & cin;
```



SystemVerilog

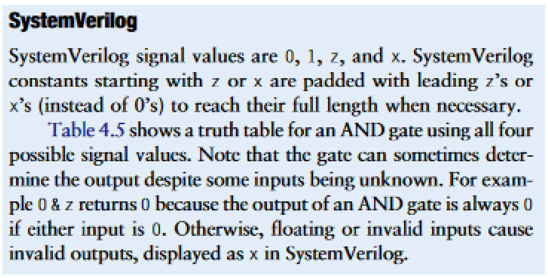
The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary, `'o` for octal, `'d` for decimal, and `'h` for hexadecimal. If the base is omitted, it defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog idioms for filling a bus with all 0s and all 1s, respectively.

Table 4.3 SystemVerilog numbers

Numbers	Bits	Base	Val	Stored
`3'b101`	3	2	5	101
`'b11`	?	2	3	000 ... 0011
`8'b11`	8	2	3	00000011
`8'b1010_1011`	8	2	171	10101011
`3'd6`	3	10	6	110
`6'o42`	6	8	34	100010
`8'hAB`	8	16	171	10101011
`42`	?	10	42	00 ... 0101010

Z's and X's: HDL's use z to indicate a floating value and x to indicate an invalid logic level. If a gate receives a floating input, it may produce an x output when it cant determine the correct output value. Similarly if it recieves an illegal or uninitialized input it may produce an x output.



SystemVerilog

SystemVerilog signal values are 0, 1, z, and x. SystemVerilog constants starting with z or x are padded with leading z's or x's (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example 0 & z returns 0 because the output of an AND gate is always 0 if either input is 0. Otherwise, floating or invalid inputs cause invalid outputs, displayed as x in SystemVerilog.

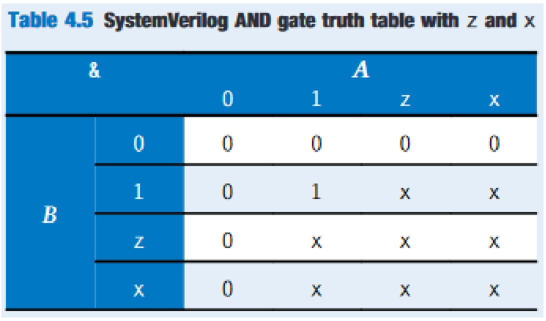
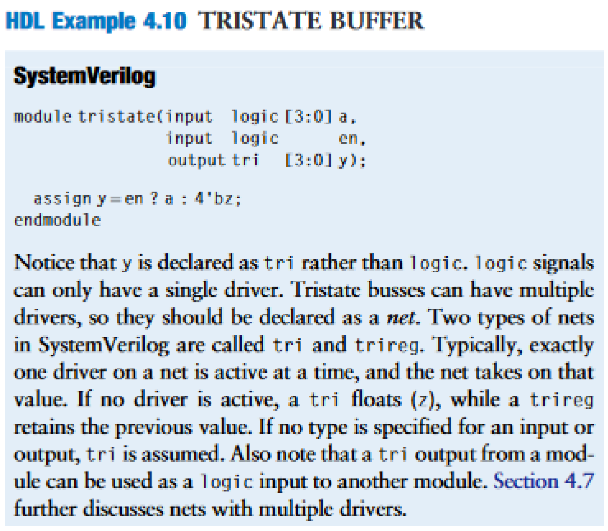


Table 4.5 SystemVerilog AND gate truth table with z and x

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x



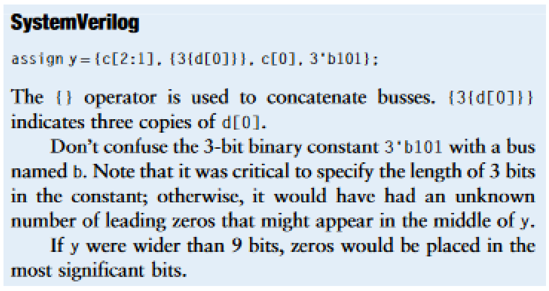
HDL Example 4.10 TRISTATE BUFFER

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that y is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `trireg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (*z*), while a `trireg` retains the previous value. If no type is specified for an input or output, `tri` is assumed. Also note that a `tri` output from a module can be used as a `logic` input to another module. Section 4.7 further discusses nets with multiple drivers.

Bit Swizzling: When operating on a subset of a bus or concatenating signals to form busses. In the example below y is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$



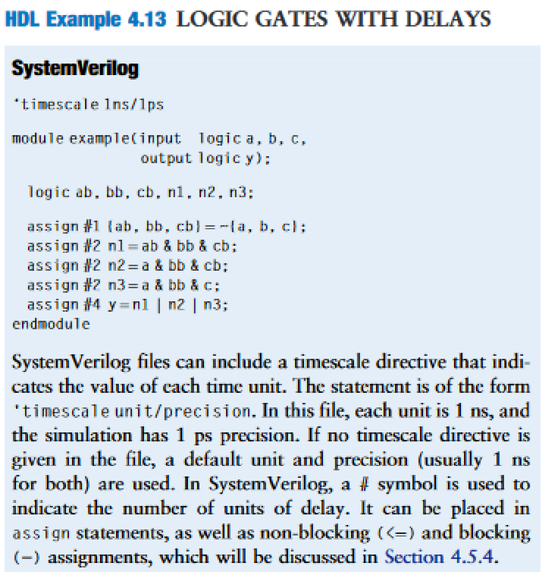
SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The `{}` operator is used to concatenate busses. `{3{d[0]}}` indicates three copies of `d[0]`.

Don't confuse the 3-bit binary constant `3'b101` with a bus named `b`. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of `y`.

If `y` were wider than 9 bits, zeros would be placed in the most significant bits.



HDL Example 4.13 LOGIC GATES WITH DELAYS

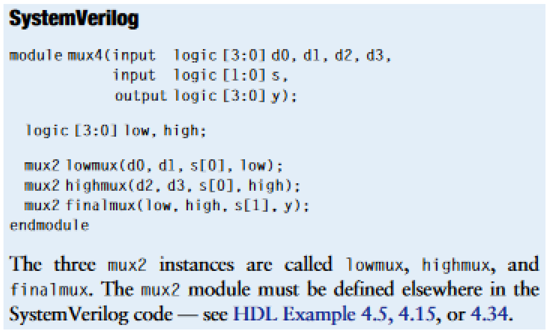
SystemVerilog

```
'timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `'timescale unit/precision`. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) are used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as non-blocking (<=) and blocking (=) assignments, which will be discussed in Section 4.5.4.

4.3 Structural Modeling

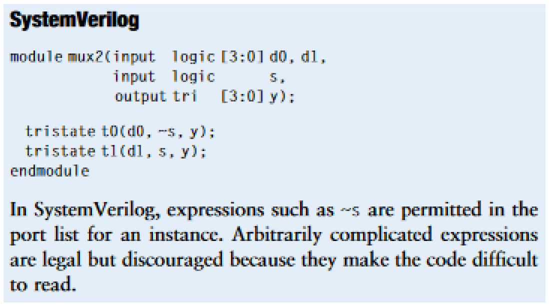
structural modeling: Describes a module in terms of how it is composed of simpler modules. Multiple instances of the same module are distinguished by distinct names.



SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the SystemVerilog code — see HDL Example 4.5, 4.15, or 4.34.



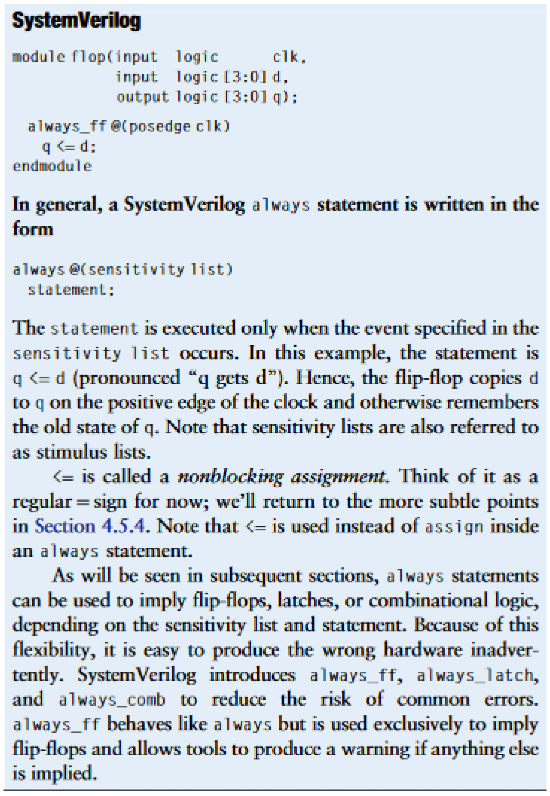
SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

4.4 Sequential Logic:

Registers and Resettable Registers: Generally it is good practice to use resttable registers so that on powerup you can put your system in a known state



SystemVerilog

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

  always_ff @(posedge clk)
    q <= d;
endmodule
```

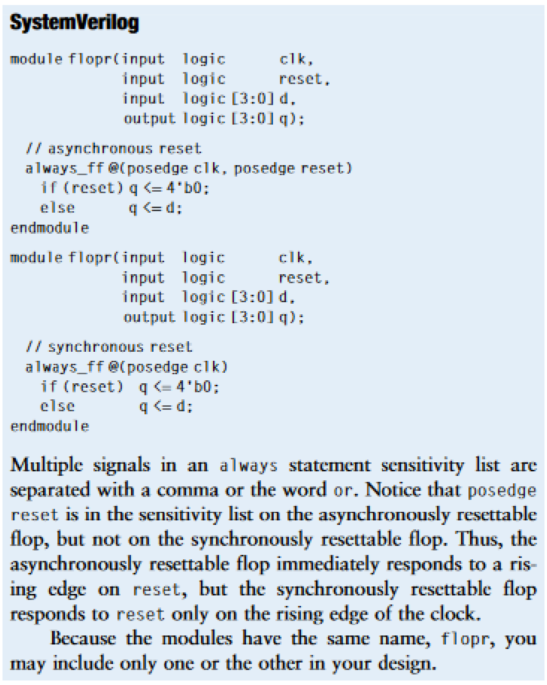
In general, a SystemVerilog `always` statement is written in the form

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

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`. Note that sensitivity lists are also referred to as stimulus lists.

<= is called a ***nonblocking assignment***. Think of it as a regular = sign for now; we'll return to the more subtle points in Section 4.5.4. Note that <= is used instead of `assign` inside an `always` statement.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.



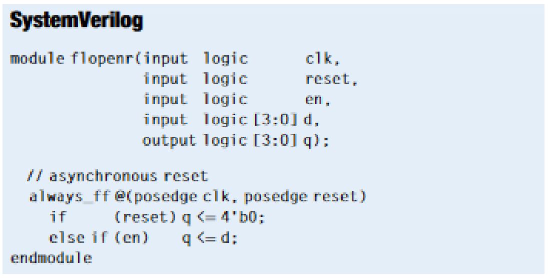
SystemVerilog

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

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset)  q <= 4'b0;
    else        q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

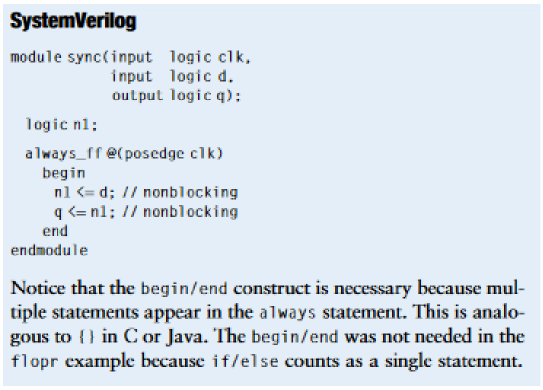
Because the modules have the same name, `flopr`, you may include only one or the other in your design.

enabled registers & multiple registers: Enable registers respont to the clock only when the enable is asserted. Always statement can be used to describe multiple pieces of hardware.



SystemVerilog

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

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```



SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

Notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to { } in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.