COMPE 470

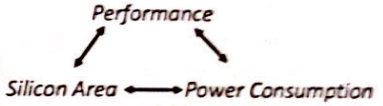
Digital VLSI System Design

Dr. Amir Alimohammad

Chapter 01 – Fundamentals of Digital Logic

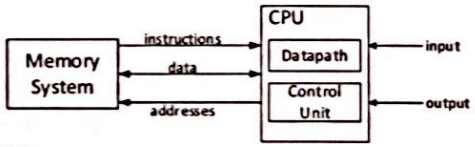
Digital design

- *Digital:* using two voltage levels to represent data
- *Design specifications* is a functional description along with various *design goals*
  - Design goals are specified as constraints
- Some of the most important *design constraints* include:
  - (Maximizing) Performance, (minimizing) power dissipation, (minimizing) silicon area, (increasing) testability, reliability, flexibility, and scalability
  - These constraints can be combined into a single cost function
- What is *design?* Given a functional description and a set of design constraints, come up with an implementation that meets the functional requirements while satisfying constraints
  - Design is optimized to meet all constraints
- As a designer you must make the tradeoffs necessary to implement the function within some predefined constraints
  - You can improve one design metric at the expense of worsening one or more of the others
  - General purpose processors: Designed to maximize the performance "optimized for speed"
  - Brain-implantable chips: Design to minimize energy dissipation and silicon area



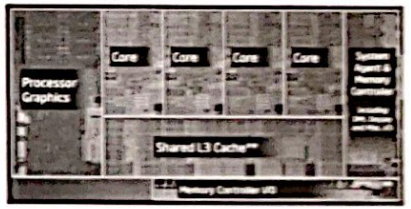
Hierarchical design

- Two main concepts that are helpful to deal with the increasing complexity of integrated circuits (ICs) are *hierarchy* and *abstraction*
- Digital circuits are too complex (may consist of millions of logic gates or billions of transistors) for us to design at once
- To manage design complexity, the design is typically divided into smaller sub-components (following the well-known *divide-and conquer* engineering approach) and each module is further divided, until the VLSI system can be described using building blocks
- *Hierarchical design*
  - Divide the complex system into sub-systems
  - Design and verify each module independently
  - Compose subsystems to form the system
  - Verify the complete system



- Hierarchical design is a critical approach for managing complex designs
- Two approaches for hierarchical design: top-down/bottom-up design

Design hierarchy and modularity



- *Design hierarchy* is an effective approach for managing the design of complex systems
- A complex system can be partitioned hierarchically into multiple *sub-systems*
- Each sub-system is built from various *units*
- Each unit in turn is composed of multiple *functional blocks*, or modules
- Each module is implemented using standard cells consisting of *transistors*
- A design hierarchy can be viewed as a tree structure with the overall chip as the *root* and the primitive cells as *leafs*
- The system can be more easily described at the high level of abstraction with well-defined interfaces and functions rather than describing the system at the transistor-level
- *Modularity* requires that the blocks have well-defined interfaces among sub-systems
- Hierarchically specifying each domain at successively detailed levels of abstraction allows us to design VLSI systems and verify the design more effectively

Digital systems

- *Integrated circuits* (ICs) are microelectronic devices (mainly transistors and wires) manufactured on a semiconductor substrate
  - Integrated circuits are nicknamed *chips*
- Digital systems are highly complex
- From a more modular viewpoint, these elements may be grouped into a number of functional components, such as datapath, control units, and memories
- Translating block diagrams into circuit schematics at the gate or transistor level is time-consuming and prone to error
- *Hardware description languages* (HDLs) have evolved to aid in the design of complex digital systems at various abstraction levels
- *Separating behavioral description from implementation*: A HDL allows design and debugging at a higher level of abstraction without detailed description of VLSI systems at the gate or transistors levels
  - For example, a 32-bit multiplier schematic is a complex structure
  - In contrast, the multiplier can be described with one line of HDL code
  - The designer or CAD tool will choose what type of multiplier architecture to use based on the specified constraints
- HDL is a convenient, device-independent representation of digital systems

Hardware description languages

- A hardware description language is not a *software programming language*
- Software programming language can be translated into machine instructions and then *executed* on a processor
  - When describing a functionality in HDL, we are specifying hardware that operates in *parallel* rather than a software program that executes sequentially on a processor
- HDL allows describing what the hardware should do at the high-level of abstraction without detailed descriptions about the gate level or transistor level implementation of the design
- HDL allows designers to separate *behavioral* description from *implementation* details at various levels of abstraction
  - Allows designing for small area, high performance, and low power consumption at the high level of abstraction earlier in the design process
- HDL is used extensively by industry and academia for modeling, verification, and implementation of digital circuits

Hardware description languages

- Primarily describes the functional behavior of all the *modules* and their *interfaces*
- Translating functional descriptions into circuit schematics is time-consuming and prone to errors
- Some other reasons for using HDLs instead of describing the circuits at the gate or transistor level (schematic capture)
  - Greatly improves designer *productivity*
  - Improves *quality*: more time can be spent on logic verification and optimization rather than on the detailed gate or transistor level design
  - *Design reuse*: A more concise and readable design makes it easier for the design to be reused by others
  - *Earlier design decisions*: Allows making decisions about performance, power consumption, and area earlier in the design process
- Using CAD tools, HDL description is synthesized into a *netlist* (i.e., gate level description)
- Synthesis tools optimize the logic for various constraints

VHDL and Verilog

- HDL provides a mechanism for system description, modeling and documentation, simulation, synthesis (implementation), testing (test generation, fault simulation), and verification
- *Verilog* and *VHDL* HDLs are used extensively by industry and academia
  - They were originally intended for documentation and simulation, but are now used to design complex ICs that meet various design constraints
- VHDL stands for *VHSIC Hardware Description Language*, where *VHSIC* in turn was a Department of Defense project on *Very High Speed Integrated Circuits*
  - VHDL is a strongly-typed language and is a relatively verbose language
  - Verilog is not as strongly-typed as VHDL, is less verbose, and is closer in syntax to C
- Many high-tech companies use Verilog while defense and telecommunications companies often use VHDL
- Verilog became an IEEE standard in 1995
  - Verilog-1995, which is the first generation, represents the standardization of the original Verilog language
  - Verilog-2001 : IEEE released the incremental updated Verilog standard in 2001

Design productivity

How to increase design productivity?

Design productivity

- Four issues impact design productivity:
- (i) Exponential increase in ICs' complexity
  - Complex system-on-a-chip with billions of transistors; Increasing with Moore's law
- (ii) Time-to-market pressure for competitive markets
- (iii) Greater diversity of on-chip elements
  - Digital circuits, memory banks, along with analog components
- (iv) Smaller transistor geometries causes various *deep sub-micron* effects and increases design risks
  - Signal integrity (does the signal reach its destination when it is supposed to? And also, when it gets there, is it in good condition?)
  - *Deep submicron* refers to process technologies with a feature size of 0.25µm or lower
  - *Nanometer* technologies refer to process technologies with feature size below 0.1µ (e.g., 90-nm, 65-nm, 45-nm, and 32-nm)
  - The *non-recurring engineering* (NRE) effort and cost of developing an ASIC are significant, making it suitable only for high-volume markets where there is no requirement for future upgrades
- How to increase design productivity?
  - Using *programmable* devices, such as GPPs and ASIPs, rather than fixed function ASICs
  - Except for very high volume productions, FPGAs can be used efficiently instead of expensive and time-consuming custom ICs (ASICs)

How to increase design productivity?

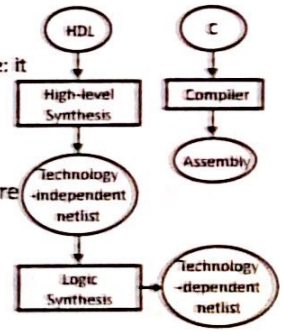
- *Decompose* complex system specifications into a hierarchy of simpler and more manageable modular designs
- Rely on high-level synthesis CAD tools over full-custom methodology
- The synthesis process can be viewed as making *transformations* from one domain to another while maintaining the equivalency of the domains
- During *behavioral synthesis*, the textual behavioral description of a module described in HDL is transformed into a register transfer level (RTL) description
- Synthesized circuits with CAD tools reduces the labor and achieves faster time-to-market
  - Even with the assistance of CAD tools, extreme design complexity and heterogeneity of systems-on-a-chip (SOC) has become a limiting factor in VLSI system design
- Describe the design at the *higher levels of abstraction*; leave the low-level details and optimizations to CAD tools
  - Behavioral descriptions are transformed to structural *register transfer-level* (RTL) descriptions, which in turn are transformed to netlist (gate level) and physical (transistor) descriptions
- Emphasis on abstract design representations: create *parameterizable* and *portable* designs using HDLs
- *Design reuse*: commercially-available and open source IP cores (opencores.org)

Simulation and synthesis

- *Simulation* is the process of verifying the proper functionality of HDL description
  - Similar to executing a software program in C and verifying the results
  - Does the design behave as specified by its behavioral description?
  - Given a module M(x) that is supposed to implement $f(x)$. Does $M(x) = f(x)$ for all x?
- *Functional simulation* is the behavioral verification of the design without timing considerations
  - During functional simulation test, inputs (stimuli) are applied to a module and the outputs are compared against expected output values
  - HDL description allows simulation at an early stage in the design process
  - Functional simulation should be performed before and after synthesizing the design
- *Timing simulation* verifies if the circuit meets the timing requirements after that the design is placed and routed using accurate gate and wire delays
- Synthesis tools (e.g., Xilinx ISE and Altera Quartus for FPGA platforms and Synopsys Design Compiler for ASIC designs) read hardware descriptions in HDL and perform various optimizations to minimize delay, area, power or some combination of them

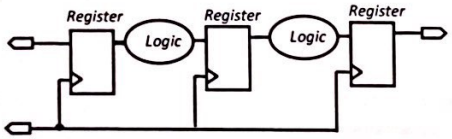
Logic synthesis

- During logic synthesis, *technology-independent netlist* description of the hardware is transformed into *technology-dependent gates* from the technology library and their interconnection (library-based implementation) that describes the structure and behavior of a hardware
- Hence, the behavioral or RTL description of the hardware is transformed into a *netlist* (logic gates) using *standard cells* to meet design constraints
  - The netlist may be a text file, or it may be displayed as a schematic capture to help visualize the circuit
  - A logic synthesis tool is similar to a compiler for hardware: it maps HDL code onto a library of gates called netlist
- The synthesis tool then *maps* the library-based netlist onto hardware resources
- *Placement* is the process of placing synthesized hardware modules such that the routing among modules are minimal
- *Routing* is the process of wiring the placed hardware modules



Design abstraction levels

- *Behavioral Level*: At this level, the behavioral functionality of the design is described in a HDL
  - For example, an algorithm may be described without specifying which operation is performed in which clock cycle
  - A behavioral synthesis tool takes a behavioral level model as input, performs a series of high-level synthesis optimizations, and generates an RTL HDL, to be processed further
- *Register Transfer Level (RTL)*: The HDL description is a cycle-accurate specification of the architecture to be designed
  - RTL description involves the architectural description of a design using the functional description of logic between registers
  - Most digital systems are *synchronous*
  - The data computed from the previous clock cycle is latched in the registers at the active clock edge



Design abstraction levels

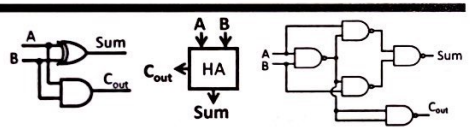
- *Gate Level*: This refers to the representation of the design using logical gates
  - A logic synthesis tool generates an optimized logic netlist from the descriptions at the higher level of abstraction
- *Circuit Level*: This refers to the circuit structure and the choice of appropriate geometry for transistors and wires
- *Device level*: This refers to the choice of the semiconductor materials and processes used to fabricate circuits
- *Digital abstraction* enabled the design of complex digital circuits at the behavioral level
- In the *full-custom design* approach, circuit designers draw schematics at the gate level and/or transistor level
- Circuit layouts can be automatically generated using synthesis tools from gate-level specifications
- Synthesis tools generally produce circuits that are neither as dense nor as fast as those handcrafted by a skilled designer at the transistor level
- Nevertheless, the synthesized circuits are good enough for the great majority of digital circuits built today

Synthesize your first Verilog description

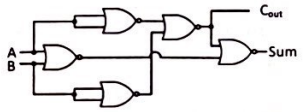
A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$Sum = A \oplus B = AB' + A'B$

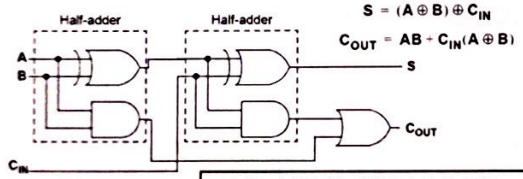
$C_{out} = A.B$




```
module halfAdder_dataflow (input A, B,
                           output Sum, Cout);
  assign Sum  = A ^ B;  //^ denotes logical XOR
  assign Cout = A & B;  //& denotes logical AND

endmodule
```



Inputs		Carry-in	Sum	Carry-out
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$S = (A \oplus B) \oplus C_{IN}$

$C_{OUT} = AB + C_{IN}(A \oplus B)$

- Note that you are describing the hardware in HDL that operates in parallel

```
module fullAdder(input A,B,CIN,
                 output S,COUT);

  assign COUT = A & B | CIN & (A ^ B);
  assign S = CIN ^ A ^ B;
endmodule
```

A testbench for functional verification

```
`timescale 1ns/1ns
module tb_fullAdder;

reg   A,B,CIN; // stimuli are defined as reg
wire  S,COUT; //outputs are defined as wire

 // Instantiate the design under test (DUT)
 fullAdder FA00 (.A(A),.B(B),.CIN(CIN),.COUT(COUT),.S(S));

 initial  begin
       A=1'b0; B=1'b0; CIN=1'b0; // time = 0
  #10;  CIN=1'b1;                // time = 10.  Only CIN is changed at time=10
  #10;  B=1;//the same as b=1'b1; // time = 20
  #10;  CIN=1'b0; A=1'b1;        // time = 30
  #1    $finish;                 // Built-in system task
 end
endmodule
```

```
module fullAdder (input A,B,CIN,
                  output S,COUT);

  assign COUT = A & B | CIN & (A ^ B);
  assign S = CIN ^ A ^ B;
endmodule
```

Design under test (DUT)

- Use `tb_modulename` for naming your testbenches to distinguish between design modules and testbenches
- In the **`initial`** statement, list the signals only when their values change
- **`$finish`** is a system task that exits the simulator and returns the control back to the operating system

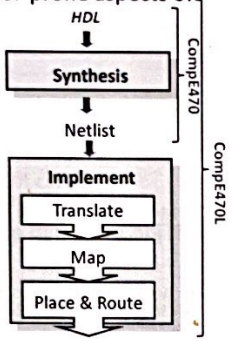
in HDL that operates in parallel

```
  assign COUT = A & B | CIN & (A ^ B);
  assign S = CIN ^ A ^ B;
endmodule
```

the operating system

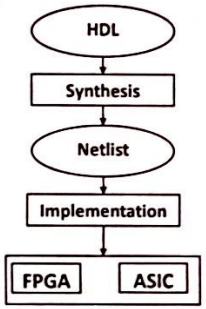
Design implementation

- *Design implementation* is the process of determining the collection of logical components and the interconnections between them to perform a specified function, such as data processing, control, or communication
- *CAD tools* use a HDL and automate the most tedious and error-prone aspects of design
- Implementation consists of three phases:
  - *Translate*: Merge multiple design files into a single netlist
  - *Map*: Maps the library-based netlist (gates) onto hardware resources
  - *Place* is the process of selecting physical components in the target hardware where design gates will reside (e.g., configurable logic blocks, DSP units, memory blocks, etc. in FPGAs)
  - *Route* is the physical routing of the interconnect between the hardware modules

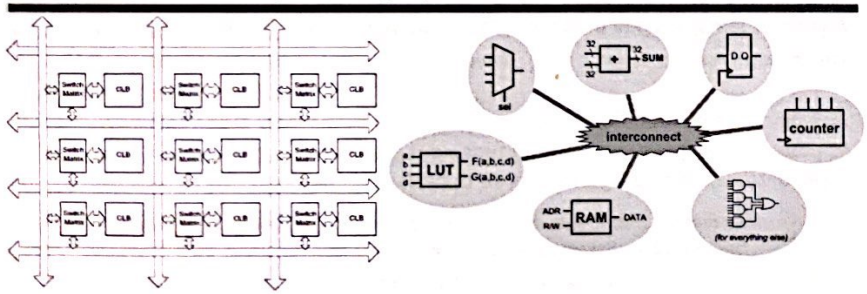


Design and implementation in CompE 470

- CompE 470: Design and verification of digital systems using hardware description languages, computer aided design (CAD) tools, and commercially-available devices
- Implementation results can be programmable or non-programmable
- *Non-programmable*:
  1) ASIC (application-specific integrated circuits): lowest flexibility, but highest energy efficiency and performance (CompE 572, CompE570, EE670)
- *Programmable*
  1) *Field-programmable gate arrays* (FPGAs) are reprogrammable via hardware reconfigurability to implement a wide range of digital functions (CompE470)
  2) *Application-specific processors* (ASIPs), such as DSP and graphics processing units (GPUs), combine the software programmability of GPPs with the performance of ASICs (CompE 475)
  3) *General-purpose processors* (GPPs): lowest energy efficiency, but highest flexibility via software programmability (CompE 475)

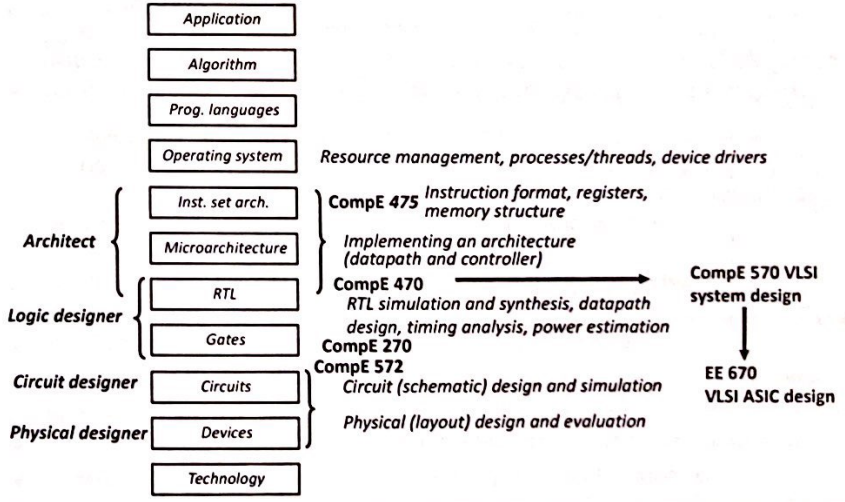


Field-programmable gate arrays (FPGAs)



- The main architectural components of FPGAs are *configurable logic blocks* (CLBs), configurable arithmetic units, embedded distributed and block memories, and *input/output blocks* (IOBs), interconnected by programmable routing resources
- Configurable routing resources are arranged in a horizontal and vertical grid
- FPGA routing resources, including a special interconnect module that serves as a configurable switch matrix, provide programmable connectivity between CLBs, IOBs, embedded memory blocks, and other modules
- Except for very high-volume devices, FPGAs can be used efficiently instead of expensive and time-consuming custom ICs (ASICs)

Digital design abstraction levels and courses

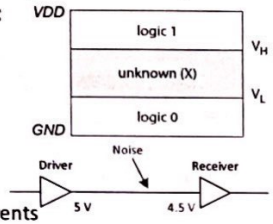


Most VLSI graduates work at Google, Apple, Qualcomm, Broadcomm , Intel, nVidia, AMD, among many others

Logic values

- The transformation of an electrical voltage into *logic values* (discrete values) is accomplished via the definition of nominal voltage levels:

  V_H: High logic level and V_L: Low logic level



- A range of voltages is used to represent 0 and 1 to allow for noise in the system
  - A range of voltages near VDD corresponds to logic 1 and a region close to GND corresponds to logic 0
  - Noise represents unwanted variations of voltages and currents
  - Note that even when an ideal input signal is applied to a logical gate input, the output may deviate from the ideal, due to *noise* and *output loading*
- The voltages between V_L and V_H are said to be in the *undefined* (*indeterminate*) *region* or *forbidden zone* and do not represent legal digital logic levels denoted by (X). Although signals must swing through the X region, no node signal should ever produce X as its final value
  - Given logically valid inputs, every circuit element must produce logically valid outputs
- What would be the upper boundary of the logic 0 region and the lower boundary of the logic 1 region? For a gate to be *robust* and *insensitive to noise* disturbances, it is essential that the "0" and "1" intervals be as large as possible

Boolean algebra

- George Boole (1854) showed that logic is math, not just philosophy!
- A two-valued *Boolean algebra* is a form of algebra that deals with binary values {0, 1} and logical operations and hence (also called mathematics of binary values)
  - *Logical operators* operate on binary values
- An *algebraic structure* is defined by a set $B = \{0, 1\}$, two binary operators (. and +), and a unary operator '
  - **NOT**: unary operator complements a logical value and is represented as A', ~A, or with an overbar (reads not A)
  - **AND**: binary operator which performs logical multiplication: i.e. A ANDed with B would be represented as AB or A.B
  - **OR**: binary operator which performs logical addition: i.e. A ORed with B would be represented as A+B

X	ZNOT
0	1
1	0

X	Y	ZAND
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	ZOR
0	0	0
0	1	1
1	0	1
1	1	1

- The Boolean algebra where the Boolean set *B* is restricted to 0 and 1 is known as *switching algebra* because the algebra can be represented with switching circuits
  - Note that Boolean algebra can be defined in general by a set *B* that can have more than two values

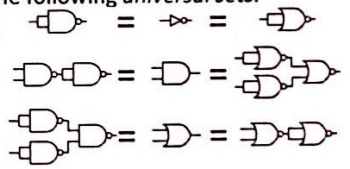
Boolean functions

- In the digital abstraction, digital circuits use only *binary digits* or *bits* (0 and 1) to represent digits
  - The inputs of a digital circuit are zeros and ones and the outputs of a digital circuit are also zeros and ones
- In addition to binary values, Boolean algebra can be applied to binary variables
  - Binary variables take on one of two values: 1 or 0 (or true or false)
  - Logical operators operate on binary values and binary variables
- Let $\{0,1\}^n$ denote the set of *n*-bit strings
- A Boolean function is a mapping $f : \{0,1\}^n \rightarrow \{0,1\}^k$, which defines output value for all possible combinations of input value
- Boolean functions using logical operators operate on binary inputs and binary variables and produce binary outputs
- In general, there are $2^{(2^n)}$ functions of *n* input variables
  - For example, there are 16 possible functions for two input variables A and B (two bits)

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		0	AB		A		B		A ⊕ B	A + B		$\overline{B}$		$\overline{A}$		$\overline{AB}$	1
		Null		Inhibition		A⊕B		$\overline{A+B}$			Implication						Identity

Truth tables

- A Boolean function can be represented uniquely using a *truth table*
- Truth table is a tabular listing of the values of a logical function for all possible combinations of its inputs
- Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using any of the following *universal sets:*
  - AND, OR and NOT gates
  - Only NOR gates
  - Only NAND gates
  - Only NAND and NOR gates

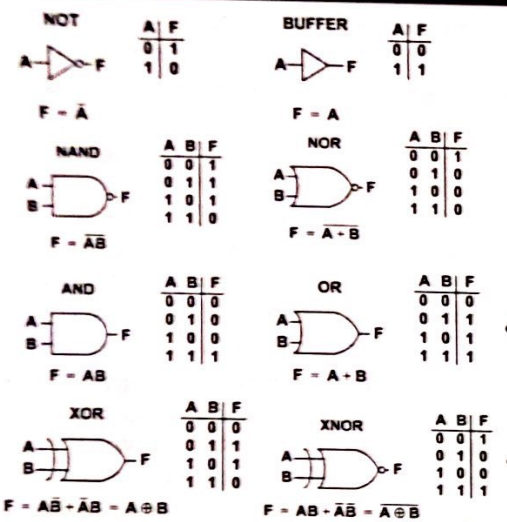


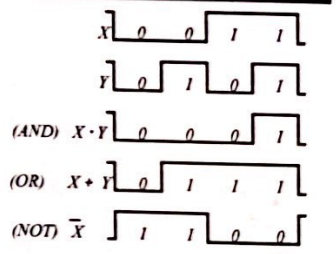
- Each Boolean function can be represented by different Boolean expressions
- Each Boolean expression has alternative implementations in logic gates
  - Hence, truth tables are unique, but expressions and gate-level are not
  - Nevertheless, Boolean equations, truth tables, and logic diagrams can be used to describe the same Boolean function

Logic gates, truth tables, and timing diagrams

Link between Boolean algebra and circuits

Logic gates, truth tables, and timing diagrams





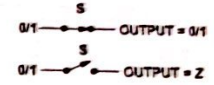
Timing diagrams

- NAND and NOR can be implemented using each other
  - X NAND $Y = (X'$ NOR $Y')'$
  - NOR $Y = (X'$ NAND $Y')'$
- DeMorgan's law can be expressed in two forms:
  - $(X + Y)' + = X'Y'$ $(XY)' = X' + Y'$

Link between Boolean algebra and circuits

Claude Shannon (1916–2001)

- Claude Shannon noticed similarities between Boolean algebra and electronic telephone switches
- Shannon's 1937 MIT Master's thesis introduces the world to binary digital electronics
- In a paper drawn from his thesis, "A Symbolic Analysis of Relay and Switching Circuits", he demonstrated that electrical application of Boolean algebra could be used to construct any logical functions using switches
- Switches provide three states on their outputs: logic-0, logic-1, and high impedance (i.e., infinite resistance, floating, no connection)
  - Path exists for logic-0 and logic-1 when the switch is CLOSED; meaning that the impedance/resistance is small enough to allow ample flow of current



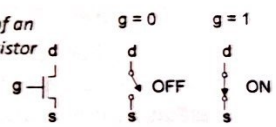
  - High impedance is a state where the switch is OPEN, meaning that the resistance is very large allowing nearly no current flow between two terminals
- An *electrically-controlled switch* has a control terminal and two other terminals that are connected or disconnected depending on the current or voltage applied to the control terminal

7

MOS transistors

- A Metal Oxide Semiconductor (MOS) transistor can be considered as a switchwith three terminals, source s, drain d, and gate g, controlled by its gate input
- In a simple explanation, an nMOS switch is on (i.e., drain and source are connected) only when the controlling gate signal g=1 and is off when the gate g=0
- A pMOS switch is on (i.e., the two terminals are connected) only when the controlling gate signal g=0 and is off when the gate g=1

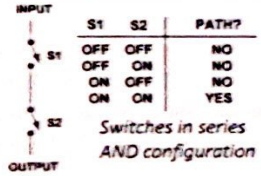
Schematic of an nMOS transistor



Ideal switch

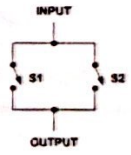
Schematic of a pMOS transistor

- Composing switches in a *network*, such as the series and parallel configurations, allows implementing various Boolean functions



S1	S2	PATH?
OFF	OFF	NO
OFF	ON	NO
ON	OFF	NO
ON	ON	YES

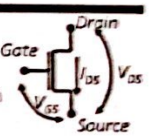
Switches in series AND configuration



S1	S2	PATH?
OFF	OFF	NO
OFF	ON	YES
ON	OFF	YES
ON	ON	YES

Switches in parallel OR configuration

Transistor operation



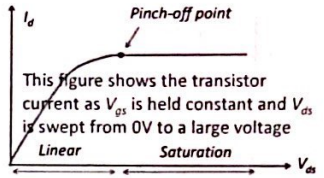
- A transistor is a *voltage-controlled switch*, where source and drain are connected depending on the voltage on the control input gate
  - For the sake of simplicity, assume gate and source are inputs and drain is the output
  - Positive current from drain to source corresponds to electrons flowing from source to drain
  - The voltage applied to the gate (gate-to-source voltage V_{gs}) controls the amount of current from the drain to source (I_{ds}) (electrons from S to D)
- An nMOS transistor operates in three regions:
- **(1)** *Cutoff:* When the gate voltage (V_{gs}) is below a critical voltage, called the *threshold voltage* V_t>0, the nMOS transistor is OFF and almost zero current ($I_{ds} \approx 0$) flows from drain to source
  - Even if V_{ds} is applied, almost no current flows because there is a very large resistance between the source and the drain
  - An ideal transistor operation assumes current only flows from source to drain when $V_{gs} > V_t$
  - Threshold voltage is a function of materials, doping, insulator thickness, substrate voltage (V_{sb}), doping, temperature, etc.

Transistor operation

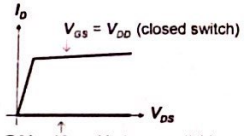
- **(2)** *Linear* (or *ohmic*): When $V_{gs}>V_t$ and a positive voltage is applied to the drain (i.e., $V_{ds}>0$), then a conducting path with a relatively small resistance is established between drain and source terminals
  - V_{ds} is the potential energy available to push current from drain to source
  - If V_{ds} is zero (i.e., $V_{gs} = V_{gd}$), the drain current I_{ds} is zero
- When $V_{gs}-V_t>V_{ds}$, I_{ds} increases with V_{ds} (similar to linear resistor)
- When V_{ds} is small, the transistor behaves like a resistor
  - The current through the channel I_{ds} is directly proportional to the voltage across it V_{ds} (i.e., $I_{ds}=V_{ds}/R_{ds}$ if V_{ds} is small). Most often V_{ds} is not small
  - The resistance is proportional to $L/W \times 1/(V_{DD}-|V_t|)$, where L and W denote the length and the width of the transistor, respectively, and also depends on various process parameters, such as mobility and temperature
- **(3)** *Saturation*: As V_{ds} increases, $V_{gs}-V_t \le V_{ds}$, the drain current I_{ds} will no longer increase with the voltage V_{ds} and approximately remain the same (i.e., independent of V_{ds}) and the transistor acts similar to a current source
- When $V_{ds} > V_{gs}-V_t$, $V_{gs} - V_t$ is called V_{dsat} and we say current saturates
  - Thus increasing V_{ds} above $V_{gs}-V_t$ has little effect (theoretically, no effect) on I_{ds}
  - Note that $V_{gs} = V_g - V_s$, $V_{gd} = V_g - V_d$, and thus $V_{ds} = V_d - V_s = V_{gs} - V_{gd}$

Transistor operation

- When $V_{ds}>V_{gs}-V_t$, I_{ds} is controlled only by the gate voltage and ceases to be influenced by V_{ds}
  - Thus the voltage at the gate (or V_{gs}) of an nMOS transistor controls the amount of current that can flow between the source and drain
- Note that the transition from the ON state to the OFF state is gradual



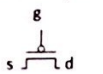
- Note that the behavior of transistors implies that they are *highly non-linear* (recall that a linear function $f(x)$ satisfies $f(a \cdot x) = a \cdot f(x)$)
  - A small change in V_{gs} around V_t has a large effect on the output
  - In particular, transistors only follow Ohm's Law (i.e. $V = I \cdot R$) in the linear region
- A transistors works as a *voltage-controlled switch* and the gate is the *control input*
  - V_{gs} controls the amount of current that can flow between the source and drain



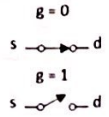
- When $V_{gs} < V_t$ the transistor is OFF and when $V_{gs} = V_{DD}$ it is ON
  - Open switch: $V_{gs} < V_t$ (sub-threshold)
  - Closed switch: $V_{gs} = V_{DD}$ (or $V_{gs}>V_t$)

pMOS transistors

- A pMOS transistor behaves in a dual manner to nMOS
  - When $V_{gs}>V_t$, the transistor is OFF (i.e., the source-to-drain resistance is very large)
  - When $V_{gs}<V_t$, the resistance between drain and the source is low, current flows between drain and source and the transistor is ON
- For pMOS, V_t is a negative value so V_{gs} should be sufficiently more negative
  - pMOS is on when $V_{gs}<V_t$ or $V_{sg}>-V_t$
- I_{DS} is defined to be flowing from the source to the drain, the opposite as the definition for an nMOS
- pMOS transistor operates in three modes:
  - *Cutoff:* $V_{gs} > V_t$, $I_{ds} \approx 0$
  - *Linear:* $V_{gs} < V_t$, $V_{ds} > V_{gs}-V_t = V_{dsat}$
  - *Saturation:* $V_{gs} < V_t$, $V_{ds} < V_{gs}-V_t = V_{dsat}$

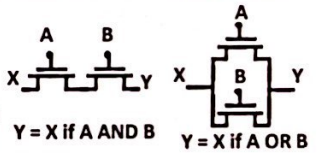


Schematic of a pMOS transistor

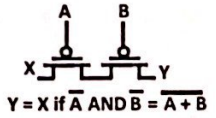


Implementing logical gates and functions using MOS transistors

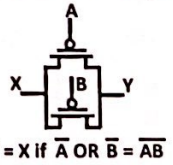
- A logical gate is built from transistors and wires
- The inputs and outputs of a gate are often referred to as terminals, ports, or pins
- The relation between the logical value of the outputs and the logical value of the inputs is specified by a Boolean function
- nMOS transistors connected in series corresponds to an *AND* function
  - With all the inputs high, the series combination conducts and the value at one end of the chain is transferred to the other end



- nMOS transistors connected in parallel corresponds to an *OR* function
  - A conducting path exists between the input and output terminals if at least one of the inputs is high



- A series connection of pMOS conducts if both inputs are low, representing a *NOR* function ($A'.B' = (A+B)'$), while pMOS transistors in parallel implement a *NAND* function ($A'+B' = (A \cdot B)'$)

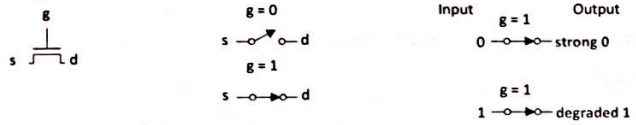


Pass transistors

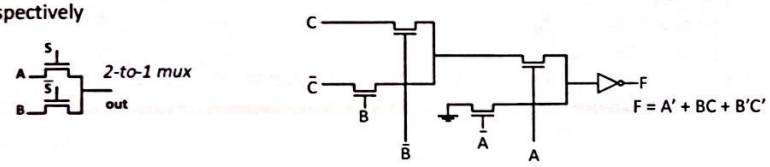
Threshold voltage

Pass transistors

- Unlike ideal switches, MOS transistors are not ideal switches and pass some voltage levels better than others
- An nMOS transistor is an *imperfect* switch, and is called a *pass transistor*
- A closed nMOS transmits logic level 0 well, but transmits logic level 1 weakly
  - The *strength* of a signal is measured by how closely it approximates an ideal voltage source. An nMOS transistor passes 0s well, but weak 1s. Hence, pass transistors produce *degraded* or *weak* "1"

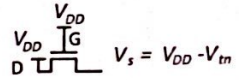


- Similarly, pMOS transistors pass 1s well but 0s poorly
  - The power source V_{DD} and the ground GND are the source of the strongest 1s and 0s, respectively

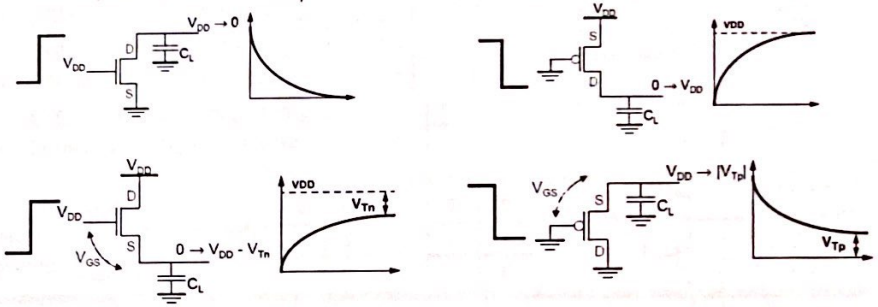


Threshold voltage

- How weak is the degraded voltage?
  - $V_s = VDD - V_{tn}$, where V_{tn} is the *threshold voltage* of an nMOS transistor

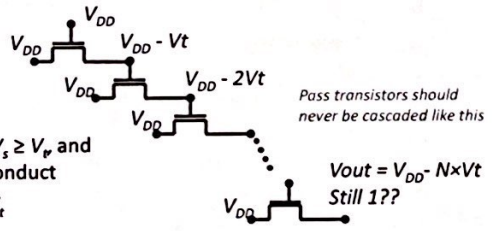


- nMOS transistors pass a strong "0", but a weak "1" (only pulls up to $V_{DD} - V_{tn}$ when passing V_{DD})
- pMOS transistors pass a strong "1", but a weak "0" (only pulls down to $|V_{tp}|$ when passing 0s, where V_{tp} is the threshold voltage of the pMOS transistor)

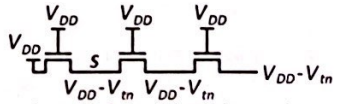


Degradation using pass transistors can be a problem

- Since a pass transistor gate is *not regenerative*, a gradual signal degradation will be observed after passing through a number of subsequent stages
- If a degraded output drives the gate of another transistor, the second transistor can produce an even further degraded output
  - For nMOS operation: $V_{gs} \geq V_t$, $V_g - V_s \geq V_t$, and V_{DS} has to be greater than 0 to conduct
  - In this case: $V_{DD} - V_s \geq V_t$. $V_s \leq V_{DD} - V_t$
  - $(V_{DD} - V_t) - V_s \geq V_t$. So $V_s <= V_{DD} - 2V_t$



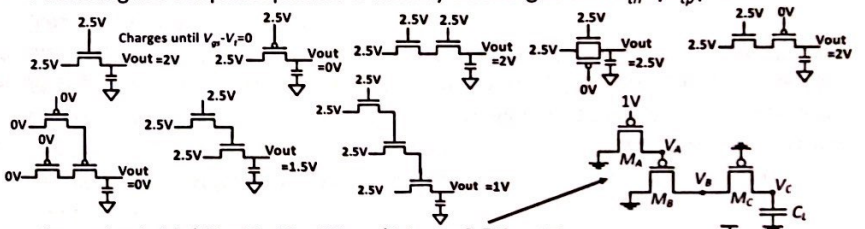
- This logic suffers from reduced noise margins and static power dissipation (reduced voltage level may be insufficient to turn off the subsequent transistor)
- In the second case, as the source can rise to within a threshold voltage of the gate, *Vs* increases to the point where $V_{gs} \geq V_t$
  - At this point transistor turns itself off
  - The output of several transistors in series is no more degraded than that of a single transistor



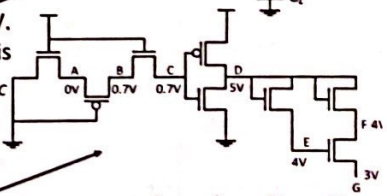
- We usually use a buffer (two inverters) at the output to restore the output, to increase the drive strengths and reject noise

Pass transistor examples

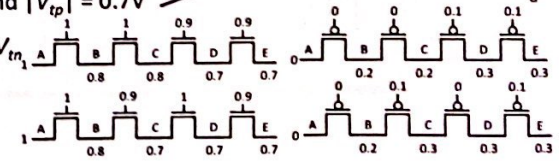
- Assuming the output capacitor is initially discharged and $V_{tn} = |V_{tp}| = 0.5V$



- Assuming initial $V_A = V_B = V_C = 3V$ and $V_{TP} = -0.5V$. $V_A = V_{GA} - V_{TA} = 1.5V$. Since V_A < initial V_B, M_B is also off and thus $V_B = V_A - V_{TB} = 2V$. Finally, M_C passes logic "1" to the output $\Rightarrow V_C = 2V$

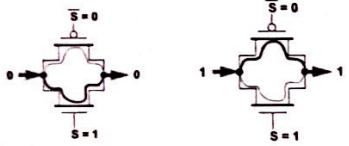


- In this circuit, the initial voltage on each node is 2.5V, $V_{DD} = 5V$, $V_{tn} = 1V$ and $|V_{tp}| = 0.7V$
- In this example, $VDD = 1V$, $V_{tn} = 0.2V$ and $|V_{tp}| = 0.2V$, and all the internal nodes have the initial voltage of 0.5V

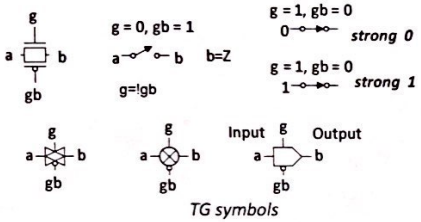


Transmission gates (TGs)

- A TG is a switch that uses two transistors: one nMOS and one pMOS transistors
  - It builds on the complementary properties of nMOS and pMOS transistors: nMOS transistors pass a strong 0 but a weak 1, while pMOS transistors pass a strong 1 but a weak 0
  - A TG thus passes both 0 and 1 well
- Both the normal control input and its complement are required by the TG
  - Transmits signal from input to output when the transistors are on
  - This is called *double rail logic*



Logic-0 at the input *Logic-1 at the input*

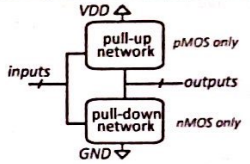


TG symbols

- A TG acts as a *bidirectional switch*
- The TG is a *nonrestoring* gate, i.e., a degraded voltage level on the input will not be restored to a strong 1 or a strong 0

Complementary MOS (CMOS)

- CMOS logic consists of two networks, a *pull-down network* (PDN) implemented using nMOS transistors and a *pull-up network* (PUN) implemented using pMOS transistors
  - The function of the PUN is to provide a connection between the output and *VDD* when the output of the logic is meant to be 1 based on the inputs
  - The function of the PDN is to connect the output to GND when the output of the logic is meant to be 0 based on the inputs
  - pMOS are best for PUN and nMOS are best for PDN

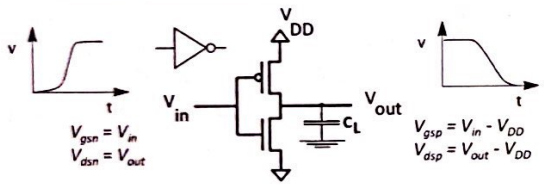


In static CMOS, the source is the terminal closer to the supply rail and the drain is the terminal closer to the output

- The PUN and PDN networks are constructed in a *mutually exclusive* fashion such that one and only one of the networks is conducting in *steady state*
- At any point in time, except during switching the output, the output is strongly driven through a low-resistance path by V_{DD} or connected to GND in steady state (i.e., the output has rail-to-rail swing)
  - The CMOS logic is thus called *static logic* as the output is a logical function of inputs and given stable inputs, the output does not change over time
  - CMOS is a *restoring logic* (produces restored output), which always produces a strong 0 or a strong 1 and the levels are never degraded

CMOS inverter

- The CMOS inverter inverts the binary input value



When $V_{in}=V_{dd}$:
$V_{gsn}=V_{dd}$ (>V_t) nMOS on
$V_{sgp}=0$ (<-V_{tp}) pMOS off

When $V_{in}=0$:
$V_{gsn}=0$ (<V_t) nMOS off
$V_{sgp}=V_{dd}$ (>V_{tp}) pMOS on

- In static CMOS, the source is the terminal closer to the supply rail and the drain is the terminal closer to the output
  - In normal operation, a positive voltage applied between drain and sources (V_{ds})
- If the input voltage is high, then the source-to-drain resistance in the pMOS is very high and the source-to-drain resistance in the nMOS is very low
  - Since the source of the N-transistor is connected to low voltage (i.e. ground), the output of the inverter is low
- If the input voltage is low, then the source-to-drain resistance in the nMOS is very high and the source-to-drain resistance in the pMOS is very low
  - Since the source of the P-transistor is connected to high voltage, the output of the inverter is high

CMOS inverter, NAND, and NOR gates

Pull-up network

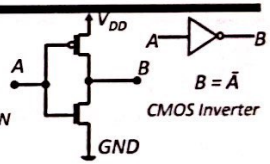
A	B
0	1
1	Z

Pull-down network

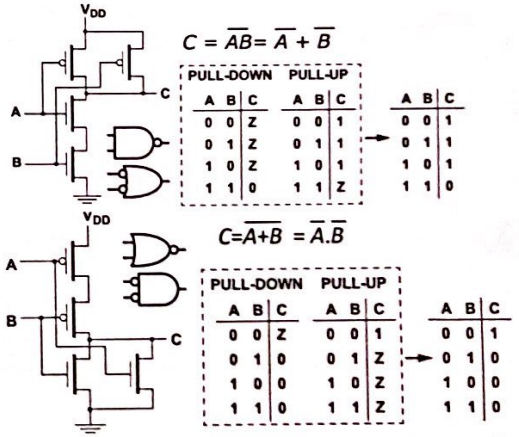
A	B
0	Z
1	0

A	B
0	1
1	0

Combined PUN and PDN



- Series of nMOS transistors: C=0 when both inputs are 1
- Parallel pMOS transistors: C=1 when either input is 0
- PUN and PDN of a CMOS structure are *dual networks*
  - This means that a parallel connection of transistors in the PUN corresponds to a series connection of the corresponding transistors in the PDN, and vice versa
  - Using De Morgan's theorems $(A \cdot B)' = A' + B'$ and $(A + B)' = A' \cdot B'$



CMOS AND and OR gates

CMOS XOR and XNOR gates

very high and the source-to-drain resistance in the pMOS is very low

- Since the source of the P-transistor is connected to high voltage, the output of the inverter is high

- Using De Morgan's theorems $(A \cdot B)' = A' + B'$ and $(A + B)' = A' \cdot B'$

A	B	C
0	0	Z
0	1	0
1	0	0
1	1	0

A	B	C
0	0	1
0	1	Z
1	0	Z
1	1	Z

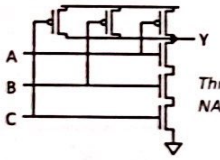
→

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

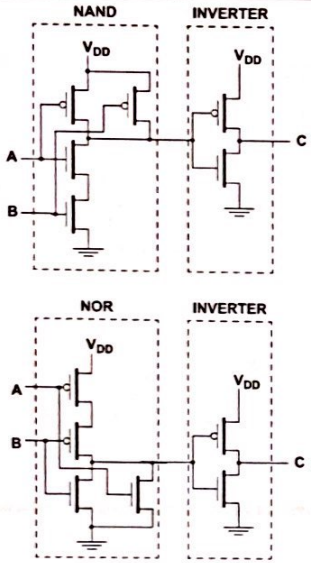
CMOS AND and OR gates

- A CMOS logic is an *inverting* logic, implementing only functions such as NAND, NOR, and XNOR
- An inverter can be used at the output to implement a non-inverting Boolean function
  - For example, a buffer can be built using two cascaded inverters

$Y = A$



Three input NAND gate



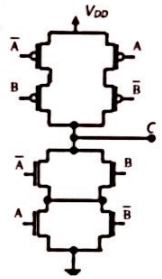
$C = AB$

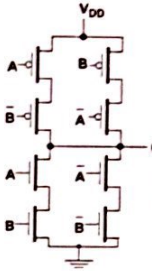
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

$C = A + B$

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

CMOS XOR and XNOR gates



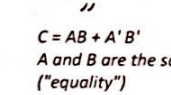


$C = \bar{A}B + A\bar{B}$

$C = \overline{AB + \bar{A}\bar{B}}$

Pull-up network			Pull-down network			CMOS XOR gate		
A	B	C	A	B	C	A	B	C
0	0	Z	0	0	0	0	0	0
0	1	1	0	1	Z	0	1	1
1	0	1	1	0	Z	1	0	1
1	1	Z	1	1	0	1	1	0

A or B but not both ("inequality" or "difference" gate)

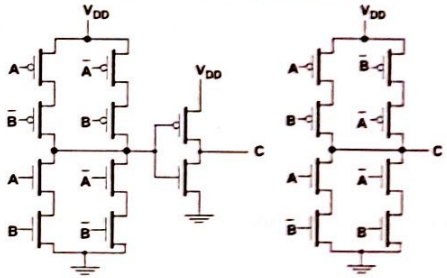


$C = AB + A'B'$

A and B are the same ("equality")

$C = \overline{A\bar{B} + \bar{A}B}$

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1



- Can XNOR be implemented without the extra inverter at the output?

XOR gates

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

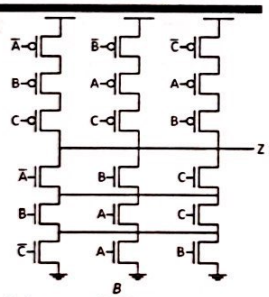
$Z = A \oplus B \oplus C = A\bar{B}\bar{C} + B\bar{A}\bar{C} + C\bar{A}\bar{B}$

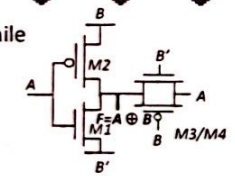
$\bar{Z} = (\bar{A}+B+C)\cdot(\bar{B}+A+C)\cdot(\bar{C}+A+B)$

$Z = \overline{(\bar{A}+B+C)\cdot(\bar{B}+A+C)\cdot(\bar{C}+A+B)}$

$Z = \overline{(\bar{A}+B+C)} + \overline{(\bar{B}+A+C)} + \overline{(\bar{C}+A+B)}$

Three input XOR gate



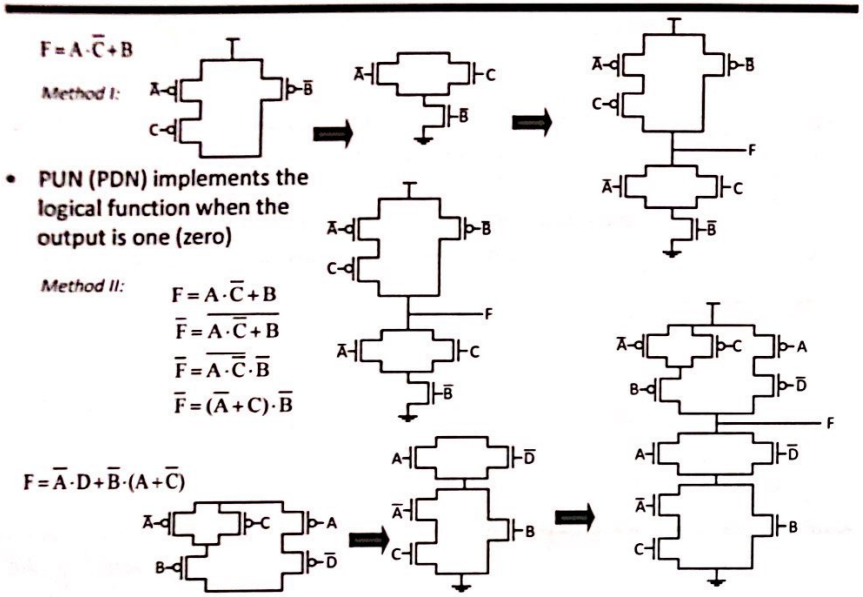


- For this circuit consider two cases: B = 0 and B = 1
- When B = 1, transistors, M1 and M2 act as an inverter while the TG is off; hence F = A'B
- When B = 0, M1 and M2 are disabled and F = AB'
- Regardless of the values of *A* and *B*, node *F* always has a connection to either *VDD* or *GND* and is hence a low-impedance node (no voltage drop and no need to refresh)
  - This XOR gate requires only six transistors, compared to the twelve transistors required for a CMOS implementation
  - Hence, TGs are effectively used in fast adders

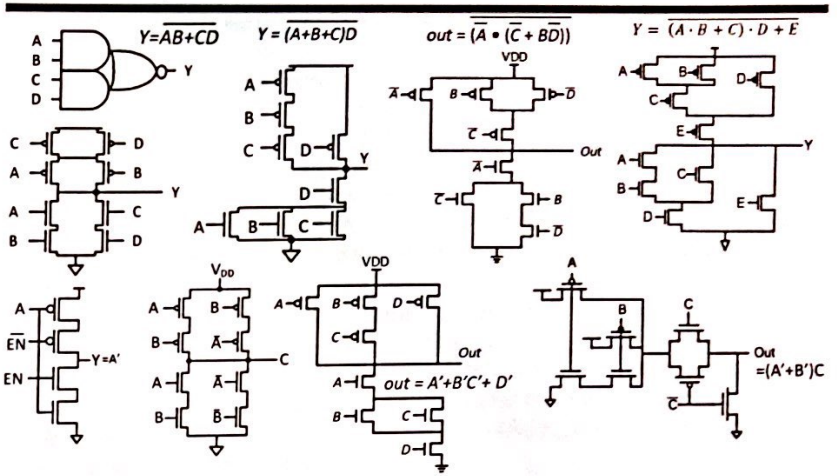
CMOS logic schematic

- Each variable in a given Boolean function corresponds to a pMOS transistor in the PUN and an nMOS transistor in the PDN
- Method I: *Start from the PUN*: Draw the PUN using pMOS based on the Boolean function. AND operation is drawn in series and OR operation is drawn in parallel. Invert each variable of the Boolean function as the gate input for each pMOS in the PUN. Draw PDN using nMOS in dual form (Parallel PUN to series PDN and series PUN to parallel PDN). Label with the same inputs of the PUN. Label the output
- *Start from the PDN*: Invert the Boolean function. With the right-hand side of the newly inverted expression, draw the PDN using nMOS. AND operation drawn in series and OR operation drawn in parallel. Label each variable of the Boolean expression as the gate input for each nMOS in the PDN. Draw the PUN using pMOS in dual form (Parallel PDN to series PUN and series PDN to parallel PUN). Label with the same inputs of PDN. Label the output
  - The reason to invert the Boolean function is because output is discharged to 0 when there is a path formed by nMOS transistors
- Both Methods I and II lead to exactly the same implementation of a CMOS logic

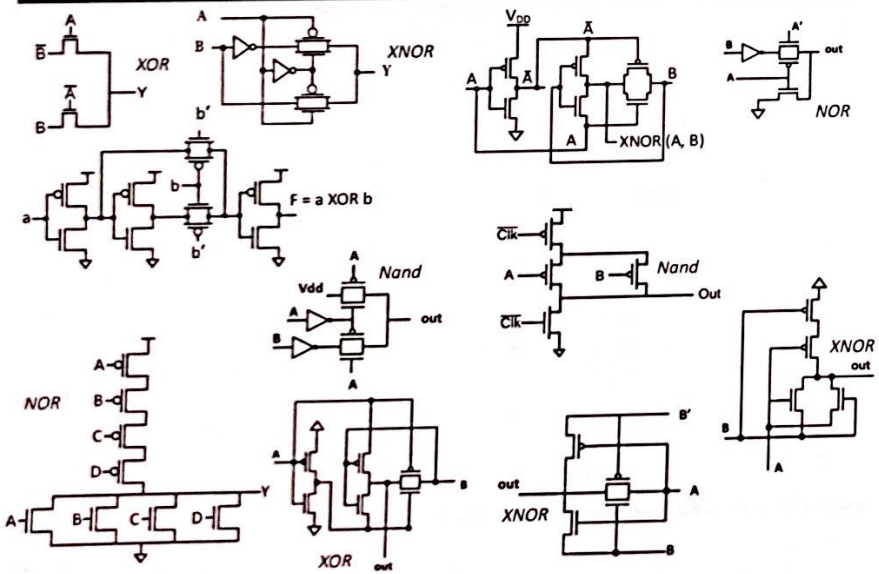
Circuit examples



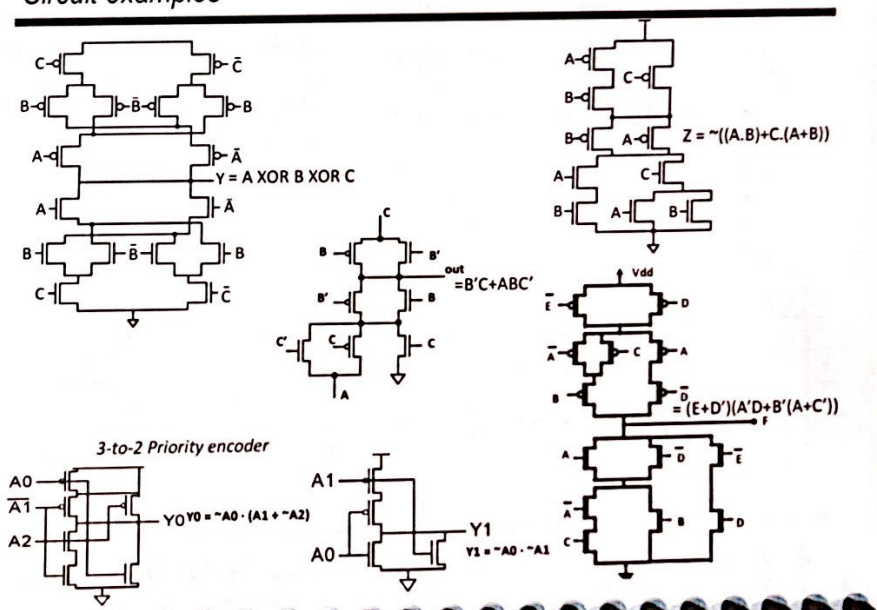
Circuit examples



Basic logic gates



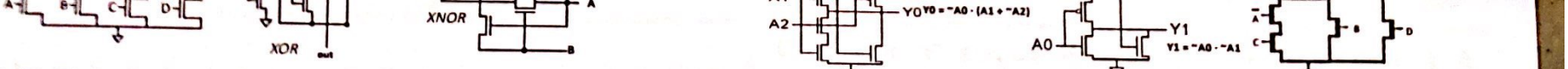
Circuit examples



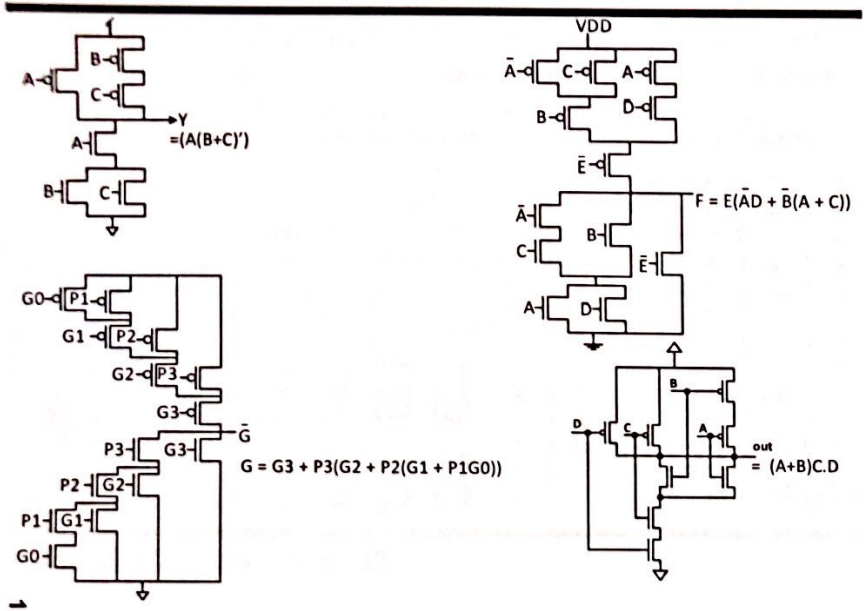
Circuit examples

Implementing the invert function

Scanned with CamScanner



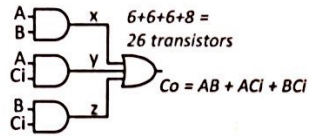
Circuit examples



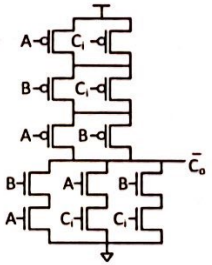
Implementing the invert function

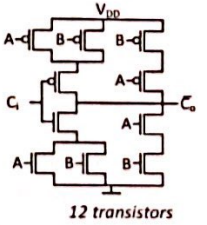
- Sometimes it is more compact (requires fewer transistors) to implement the *inverse* of a function and add an inverter at the output

$$C_o = ab + c_i(a+b) \Rightarrow \overline{C_o} = \overline{AB + C_{in}(A+B)} = (\bar{A}+\bar{B})\cdot\bar{C}_i + \bar{A}\bar{B}$$



Co = (AB)|(AC$_{in}$)|(BC$_{in}$);





- Both circuits are equivalent. One uses fewer transistor and having less input capacitance on each input. This is a static CMOS gate even though the pull-down and pull-up networks are not complementary. For any combination of inputs, *Co* is always connected to VDD or GND

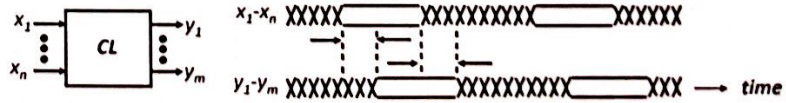
$$\overline{C_o} = \overline{\bar{A}BC_i + A\bar{B}C_i + AB\bar{C}_i + ABC_i} = \overline{AB + AC_i + BC_i}$$

CMOS pros and cons

- The vast majority of digital circuits use static CMOS because it is *robust*: Given the correct inputs, it will produce the correct output given that as there were no errors in logic design or manufacturing
- Since CMOS is a *fully restored* logic (provides full rail-to-rail swing), it simplifies circuit design considerably
- CMOS provides good *noise margin*
- CMOS is insensitive to device variations, relatively easy to design, widely supported by CAD tools, and readily available in standard cell libraries
- CMOS is increasingly applied to *ultra-low power systems*, such as implantable medical devices, that require years of operation using a battery and sensors that scavenge their energy from the environment
- One disadvantage is that CMOS requires both PUN and PDN, which for complex digital systems could results in a relatively large silicon area

Combinational logic

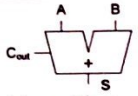
- A combinational circuit implements a Boolean function as a network of primitive logic gates and every Boolean function can be implemented by a comb. circuit
- A digital circuit is combinational if its output values at any time only depend on its input values
  - No *state* (*memoryless*): Combinational logic has no memory elements (e.g., flip-flops or register)
  - No storage of past inputs, past intermediate values, or previous output values
  - Combinational logic circuits have no feedback



- As soon as one or more inputs change, outputs will change after a *propagation delay*, which is the critical path delay of the circuit
- Outputs can have multiple logical transitions before settling to the correct value
  - Before storing the output values, whave to wait long enough for output glitches (transient value changes) to settle down
- After all output signals have stabilized, the present output value of a combinational logic is entirely determined by the present input values

Combinational logic example - One-bit binary adder

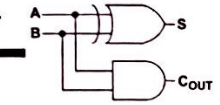
- Half-adder (HA)

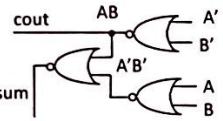


A	B	S	C_{OUT}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

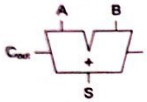
$S = A \oplus B$

$C_{out} = AB$





- A full-adder (FA) is a half-adder with a carry input, which can be implemented using two HAs and an OR gate



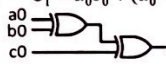
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S = (A \oplus B) \oplus C_{IN}$

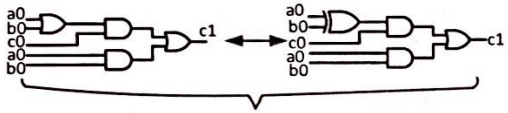
$C_{OUT} = AB + C_{IN}(A \oplus B)$

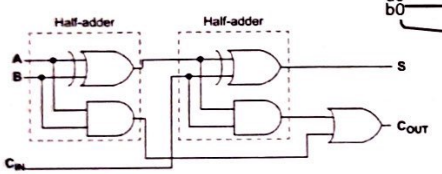
$$S = a_0 \cdot b_0 \cdot c_0 + a_0 \cdot \overline{b_0} \cdot \overline{c0} + \overline{a0} \cdot \overline{b0} \cdot c0 + \overline{a0} \cdot b0 \cdot \overline{c0}$$
$$= (a0 \cdot b0 + \overline{a0} \cdot \overline{b0}) \cdot c0 + (a0 \cdot \overline{b0} + \overline{a0} \cdot b0) \cdot \overline{c0}$$
$$= \overline{a0 \oplus b0} \cdot c0 + a0 \oplus b0 \cdot \overline{c0} = a0 \oplus b0 \oplus c0$$

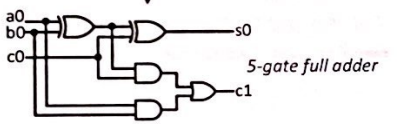
$$C_1 = a_0 b_0 + (a_0 + b_0) c_0$$



6-gate full adder



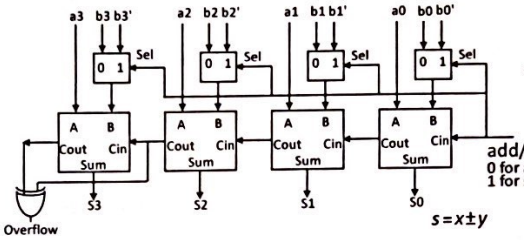


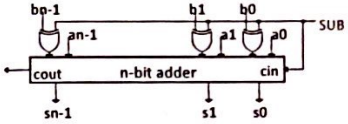


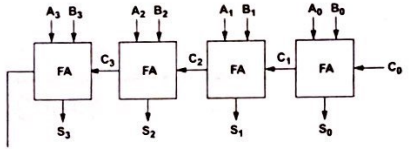
5-gate full adder

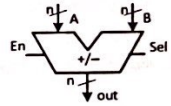
Other combinational logic examples

- An *n-bit binary adder* can be implemented by cascading *n* full-adders
  - For example, a 4-bit binary adder can be formed with four full-adders
- Two's-complement adder/subtractor can be implemented using a single adder
- Addition in two's complement can be done using hardware identical to that used in unsigned addition, because the operations are the same



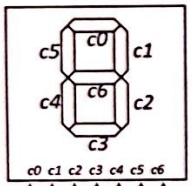


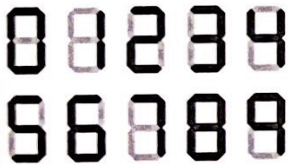




n-bit adder/subtractor block diagram

Combinational logic examples





$C0 = A + B\,D + C + B'\,D'$
$C1 = C'\,D' + C\,D + B'$
$C2 = B + C' + D$
$C3 = B'\,D' + C\,D' + B\,C'\,D + B'\,C$
$C4 = B'\,D' + C\,D'$
$C5 = A + C'\,D' + B\,D' + B\,C'$
$C6 = A + C\,D' + B\,C' + B'\,C$

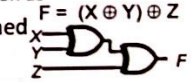
A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	X	X	X	X	X	X	X	X
1	1	X	X	X	X	X	X	X	X	X

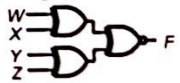
- An n-bit unsigned magnitude comparator can be implemented using an XNOR gate (*equivalence gate*) for each of the *n* bits as: $x_i = A_i B_i + \overline{A_i}\,\overline{B_i} = \overline{A_i \oplus B_i}$
  - For example, the Boolean functions for a 4-bit magnitude comparator can be written as:

$$(A = B) = x_3 x_2 x_1 x_0$$
$$(A > B) = A_3 \overline{B_3} + x_3 A_2 \overline{B_2} + x_3 x_2 A_1 \overline{B_1} + x_3 x_2 x_1 A_0 \overline{B_0}$$
$$(A < B) = \overline{A_3} B_3 + x_3 \overline{A_2} B_2 + x_3 x_2 \overline{A_1} B_1 + x_3 x_2 x_1 \overline{A_0} B_0$$

Odd and even functions

- An *odd function* is 1 when the number of 1's in the input combination is odd
- The complement of an odd function, an *even function*, is 1 when the number of 1's in the input combination is even
- Odd and even functions can be implemented using trees of XOR or XNOR gates
- An XOR of n-variables is an *odd function* which generates a value of 1 if the number of 1's in the input combination is odd, otherwise it generates a value of 0
  - For example, a 3-input odd function with 2-input XOR can be written as $F = (X \oplus Y) \oplus Z$
- The complement of an odd function (an even function) is obtained by replacing the output gate with an exclusive-NOR gate
  - For example, a 4-input even function with 2-input XOR and XNOR gates can be written as: $F = (W \oplus X) \oplus (Y \oplus Z)$





- Use an odd function to generate the even parity bit
- Use an even function to generate the odd parity bit
- To check for even parity use an odd function
- To check for odd parity use an even function
  - For example to design an even parity generator and checker for 3-bit codes, use a 3-bit odd function to generate even parity bit and use a 4-bit odd function to check for errors in even parity codes

Parity generator and checker

Example

Message X Y Z	Even Parity Bit, P	C

To generate an even parity bit
$P = X \oplus Y \oplus Z$

$(A<B) = \overline{A}_3B_3 + x_3\overline{A}_2B_2 + x_3x_2\overline{A}_1B_1 + x_3x_2x_1\overline{A}_0B_0$

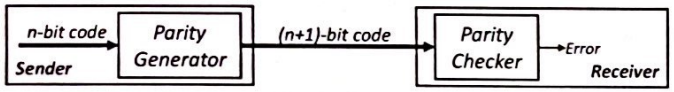
odd function to generate even parity bit and use a 4-bit odd function to check for errors in even parity codes

Parity generator and checker

- A parity bit is an extra bit concatenated to a binary message to make the total number of 1's in the (*n*+1)-bit code (including the parity bit) either odd or even
  - In an *odd-parity code* (*odd-parity code*), the parity bit is specified so that the total number of ones in (*n*+1)-bit code is *odd* (*even*)

Example: `0 1 0 0 0 0 1 1`
`↑`
`Added odd parity bit`

- A parity bit is used for the purpose of detecting a single bit errors during transmission of binary information
- The circuit that generates the parity bit at the transmitter is called *parity generator* and the circuit that checks the parity at the receiver is called *parity checker*



- The message, including the parity bit, is transmitted and then checked at the receiver for errors
  - An error is detected if the checked parity does not match with the one transmitted

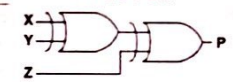
Example

Message X Y Z	Even Parity Bit, P	C
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	0	0
1 0 0	1	0
1 0 1	0	0
1 1 0	0	0
1 1 1	1	0

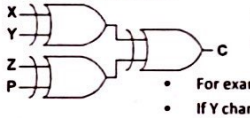
If no errors detected, C = 0

Truth table of the even-parity generator

To generate an even parity bit
$P = X \oplus Y \oplus Z$



To check a even parity bit
$C = X \oplus Y \oplus Z \oplus P$



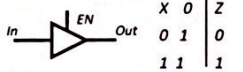
$$P = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ$$
$$= \overline{X}(\overline{Y}Z + Y\overline{Z}) + X(\overline{Y}\overline{Z} + YZ)$$
$$= \overline{X}(Y \oplus Z) + X(\overline{Y \oplus Z})$$
$$= X \oplus (Y \oplus Z)$$
$$= X \oplus Y \oplus Z$$

- For example, for (X,Y,Z) = (0,0,1) P = 1
- If Y changes from 0 to 1 between transmitter and receiver, then C = 1 indicates an error

- What if there are two bit errors?
- XOR is widely used in arithmetic units, such as adders and multipliers. XOR properties: x⊕0=0,x⊕1=x',x⊕x=0,x⊕x'=1,x⊕y'=x'⊕y=(x⊕y)'. XOR can be used as a *programmable* inverter: if A=0, Z=B; if A=1, Z=~B
- XOR and XNOR are associative operations:
  - (x⊕y)⊕z= x⊕(y⊕z)=x⊕y⊕z
  - ((x⊕y)'⊕z)'= (x⊕(y⊕z)')'=x⊕y⊕z
- Unlike AND, OR, XOR, and XNOR, the NAND and NOR operations are not associative
  - (X NAND Y) NAND Z ≠ X NAND (Y NAND Z)
  - (X NOR Y) NOR Z ≠ X NOR (Y NOR Z)

Tristate buffers

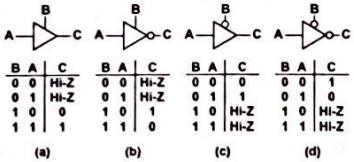
- When the enable control input *EN* is high, the tristate bufferpasses the input *in* to output *out*
- When EN is low, the output is disconnected from the *VDD* or *GND*
  - The output value of a tristate buffer is z when not enabled
  - Floating output z might be 0, 1, or somewhere in between



In	EN	Out
X	0	Z
0	1	0
1	1	1

Non-inverting tri-state buffer

- Four kinds of tristate buffers
- We should only use value on Out when its being driven (using a floating value may cause failures)



B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(a)

B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0

(b)

B	A	C
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z

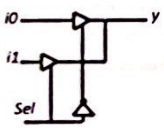
(c)

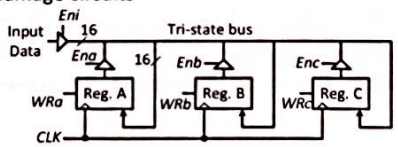
B	A	C
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

(d)

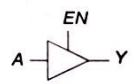
- One application of tri-state buffers is in busses
  - Tristate buffers allow more than one gate to be connected to the same output bus so long as only one of the buffers is enabled at any time
  - Pulling high and low at the same time can damage circuits

A tri-state 2-to-1 mux
when Sel is high i1 is connected to y
when Sel is low i0 is connected to y

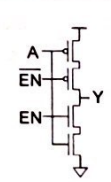




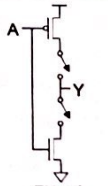
Transistor level schematic of tri-state buffers



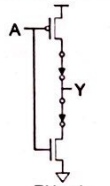
EN	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1



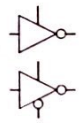
Tristate inverter



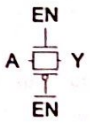
EN = 0
Y = 'Z'



EN = 1
Y = $\overline{A}$

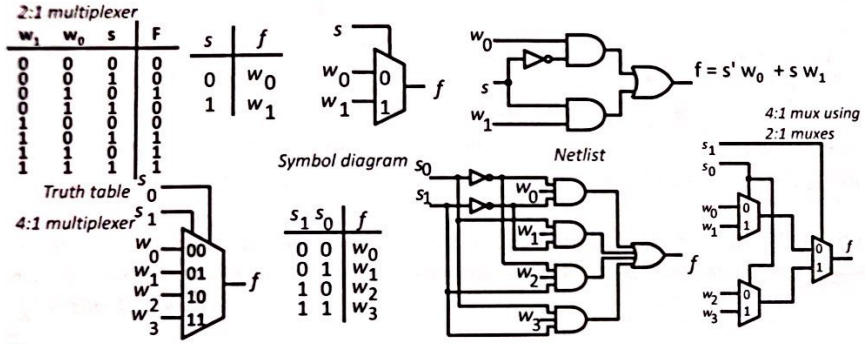


- When both pull-up and pull-down are OFF (i.e., EN = 0), the output is in a *high impedance* or *floating* output state
- The above tristate inverter produces *restored output*, but violates *conduction complement* property of CMOS logic (it does not produce 0 or 1 only)
- A transmission gate can also act as a tristate buffer
  - Only two transistors
  - But *nonrestoring:* degraded value of A is passed on to Y



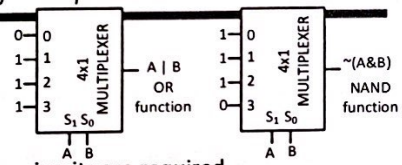
Multiplexers

- A multiplexer has a number of inputs, one or more select inputs, and one output
  - It passes the signal value on one of inputs to the output
  - Control signal value forms binary index of input connected to output
- An N:1 multiplexer selects between one of *N* inputs to connect to the output
- An $\lceil \log_2 N \rceil$-bit select input (control input) is required, where $\lceil . \rceil$ denotes the ceiling operation
  - A multiplexer usually has 2^N data inputs, *N* select inputs, and one output

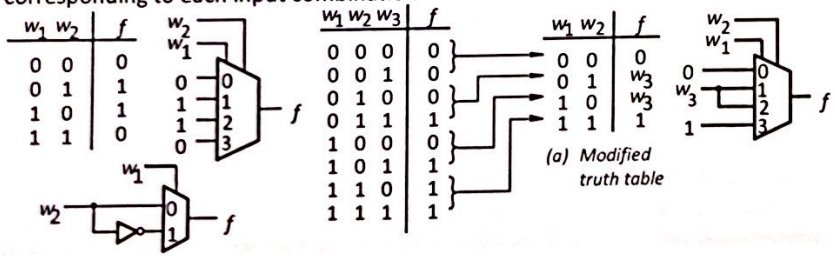


Logical function implementation using multiplexers

- Any Boolean function can be implemented by setting the inputs corresponding to the function values and the variables as the multiplexer's select signals

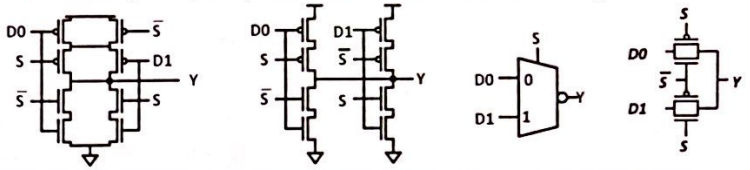


- To implement an *n*-bit logical unit, *n* of these circuits are required
  - Each one-bit unit is called the *bit slice*
- 2^n:1 multiplexer implements any function of *n* variables with the variables used as control inputs and data inputs tied to 0 or 1 based on the function value corresponding to each input combination

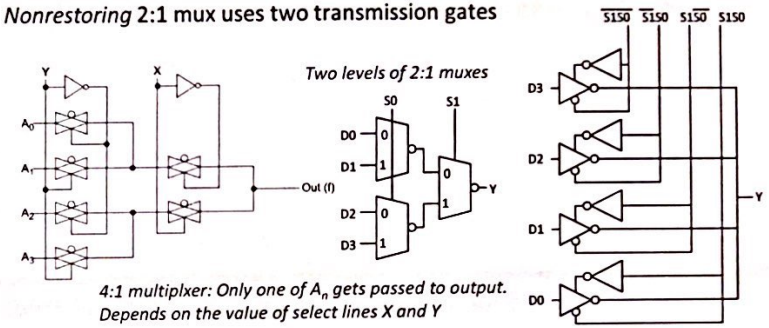


Multiplexers

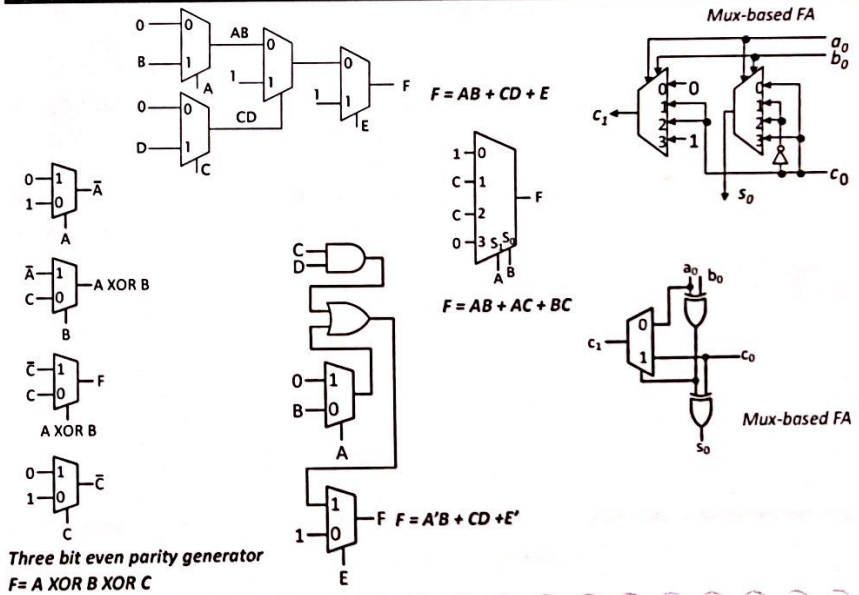
- An *inverting multiplexer* can be implemented using two tristate inverters
  - For a noninverting multiplexer, an inverter should be added at the output



- *Nonrestoring* 2:1 mux uses two transmission gates



Examples



Demultiplexer

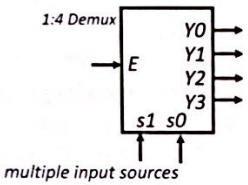
Decoders

Three bit even parity generator
F= A XOR B XOR C

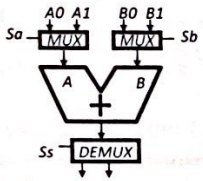
Demultiplexer

- A multiplexer routes one of *N* inputs to the output
- A demultiplexer routes a single input to one of *N* outputs
- Multiplexers can be used for resource sharing for compact implementations
- A demultiplexer is useful for routing an input to any one of N modules
  - Select line represents binary index of output to which the input is connected
- Multiplexers and demultiplexers are used for direct point-to-point connections between modules
  - Multiplexers and demultiplexers can be used together to permit the "time-multiplexed" sharing of wires and busses

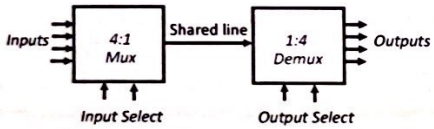
s1	s0	Y0	Y1	Y2	Y3
0	0	E	0	0	0
0	1	0	E	0	0
1	0	0	0	E	0
1	1	0	0	0	E



multiple input sources

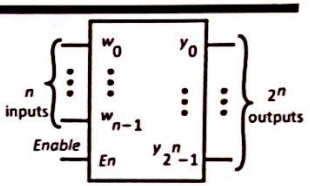


multiple output destinations



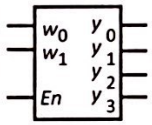
Decoders

- A decoder has *n* inputs and 2^n outputs and is used to decode encoded information. If *Enable* = 1, only one output corresponding to the value of the inputs is asserted at a time. The word *asserted* indicates that a signal that is or should be driven logically high, and *deasserted* to denote logically low
- *One-hot encoded output*: 2^n-bit binary code where exactly one bit is set to 1

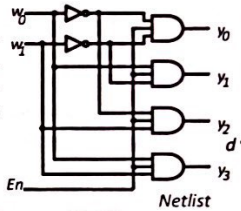


En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

Truth table



Block diagram



Netlist

$y_0 = w_{n-1}' \cdots w_1' w_0' En$

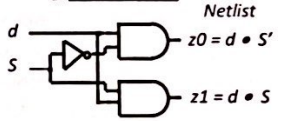
$y_1 = w_{n-1}' \cdots w_1' w_0 En$

$y_2 = w_{n-1}' \cdots w_1 w_0' En$

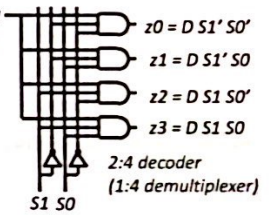
...

$y_{2^n-1} = w_{n-1} \cdots w_1 w_0 En$

- Decoders can be considered as demultiplexers
  - Single data input (enable), *n* control (select) inputs and 2^n outputs

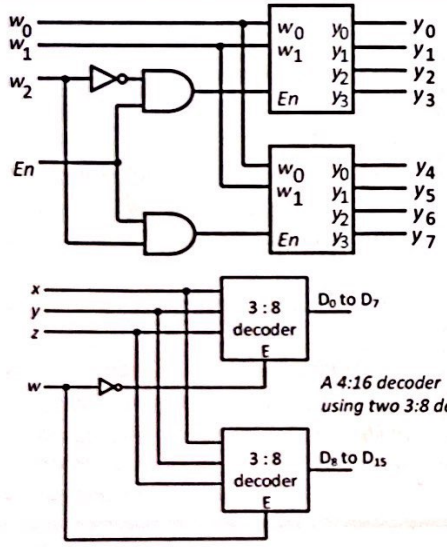


1:2 decoder (1:2 demultiplexer)



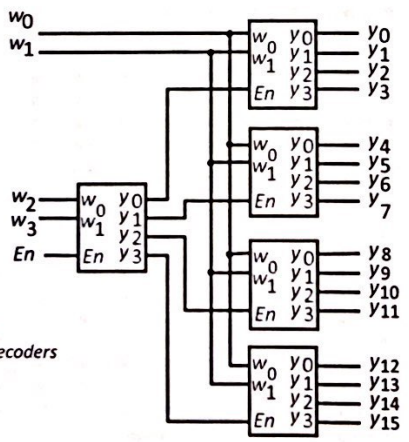
2:4 decoder (1:4 demultiplexer)

Implementing larger decoders



A 4:16 decoder using two 3:8 decoders

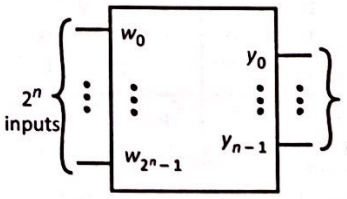
A 3:8 decoder using two 2:4 decoders



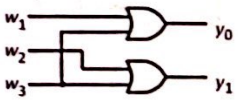
A 4:16 decoder using five 2:4 decoders

Encoders

- A binary encoder encodes 2^n inputs into an *n*-bit code
  - Exactly one of the input signals should have a value of 1, and outputs represent the binary value that identifies which input is equal to 1
- Encodes given information into a more compact form: Opposite of a decoder
  - Thus it generates binary codes at output corresponding to input codes
- An encoder can be used to reduce the number of bits for transmission and storage of information



w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



Priority encoders

- Encoders are useful when the occurrence of one of several *disjoint* events needs to be represented by an integer identifying the event
- Truth table for an 8:3 encoder is shown
- What happens if more than one input is 1?
- A *priority encoder* associates a priority level to each input
- The priority encoder outputs indicate the active input that has the highest priority
  - i.e., a priority encoder takes the input of 1 with the highest priority index and translates that index to the output

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

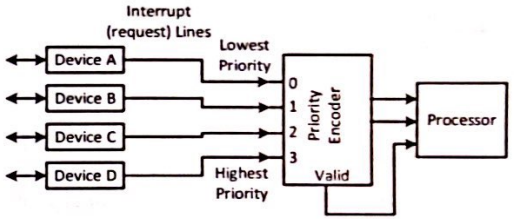
Truth table for an 8-to-3 priority encoder

Priority encoder example

- For example, when resolving interrupt requests, inputs have a predefined priority and the input with the highest priority is always selected first
  - In this example, "d_0" has highest priority

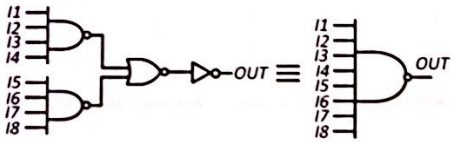
d_3	d_2	d_1	d_0	y_1	y_0	valid
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Truth table for a 4:2 priority encoder



Challenges in implementation of Boolean functions

- The required number of rows in a truth table grows exponentially with the number of inputs
  - For example, a 10-input function has a truth table containing 2^10 = 1024 rows
  - More compact representations are required
- Graphical methods, such as Karnaugh maps, are effective ways of logic simplification, but hard to draw and visualize for more than 4 dimensions
- One can directly apply theorems of Boolean algebra for algebraic simplification, however, how do you know when the minimum realization has been found?
- A particular logic function can be implemented using a variety of logical gates
  - For instance, using NAND gates only or combination of NAND, NOR, and NOT, depending on the design constraints
- For increasing speed, gates with a relatively large number of inputs (or fan-in) may be replaced with multiple gates with fewer inputs
  - For example an 8-input NAND might be faster using three smaller gates
  - However the tradeoff exists between speed and cost: 22 transistors vs. 16 transistors



Choosing an optimal gate combination

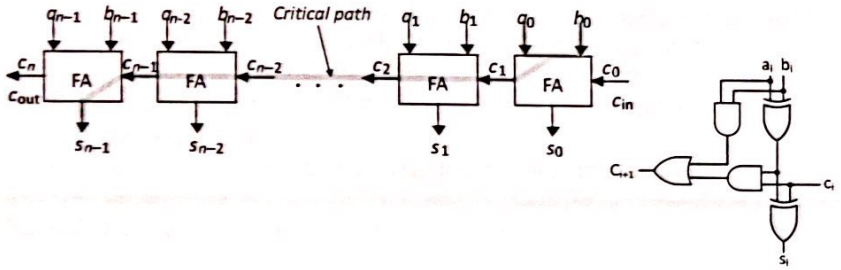
- For *increasing performance*, we should also reduce number of levels of gates
  - Fewer level of gates implies reduced signal propagation delays
  - Minimum delay typically requires more gates, wider, less deep circuits
- For *area minimization*, we should reduce number of input variables and gates
  - Fewer literals (input variable, complemented or not) means fewer transistors and hence, smaller circuits
  - Cost is proportional to the number of transistors or area of integrated circuit
- How to implement a Boolean function using the minimum number of transistors?
  - What combination of logical gates require minimum number of transistors?
  - What should be the fan-in and fan-out of each gate?
- How do we explore tradeoffs between increased circuit delay and size?
- Use CAD tools for logic minimization and optimization
  - CAD tools search among various netlist realizations for the best implementation
  - *Logic minimization*: reduce number of gates and literals
  - *Logic optimization*: area reduction while trading off against delay

Ripple-carry adder (RCA) or carry-propagate adder (CPA)

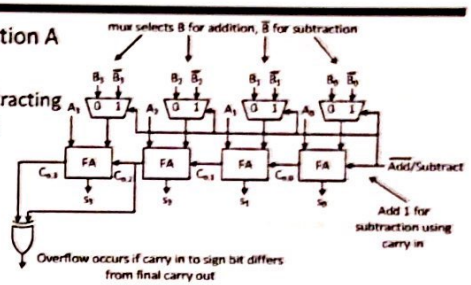
transistors

Ripple-carry adder (RCA) or carry-propagate adder (CPA)

- A *ripple carry adder* for *N*-bit numbers can be implemented by concatenating *N* full adders. At the *i-th* bit position, the *i-th* bits of operands A and B and a carry signal from the preceding adder stage are added to generate the *i-th* bit of the sum, s_i, and a carry, c_{i+1}, to the next adder stage. The carry signal "ripple" from the LSB position to the MSB position and every sum bit s_i depends on c_i, which depends on c_{i-1} and so on. Thus each sum bit depends on all previous carry bits and hence, RCA is inherently slow
  - All 2n+1 input bits of *a*, *b*, and c_{in} are assumed to be available at the same time
  - *Linear worst case propagation delay* with the number of bits. The RCA produces correct sum after this fixed delay. RCA is too slow for wide (32 bits, 64 bits) additions

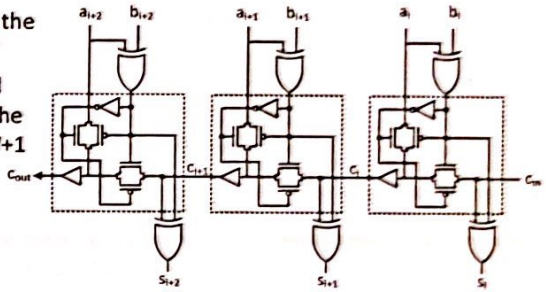


- Adders can also be used for subtraction A − B = A + (− B) = A + B' + 1
  - For two's complement format, subtracting B is the same as adding the bitwise complement of B then adding 1



- Multiplexers can be used in the critical path
- Assuming multiplexer based XOR gate implementation, the critical path will consist of *N*+1 pass transistor delays

$$s_i = a_i \oplus b_i \oplus c_i$$
$$c_{i+1} = a_i b_i + (a_i + b_i)c_i$$
$$= a_i b_i + (a_i \oplus b_i)c_i$$

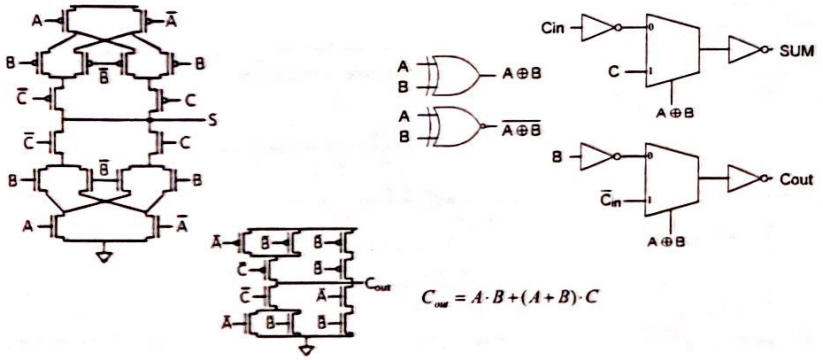


32-transistor CMOS full-adder

$$S = A\cdot B\cdot C + A\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot\overline{C} + \overline{A}\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C}$$
$$= (A\cdot B + \overline{A}\cdot\overline{B})\cdot C + (A\cdot\overline{B} + \overline{A}\cdot B)\cdot\overline{C}$$
$$S' = A'B'C' + A'BC + AB'C + ABC'$$

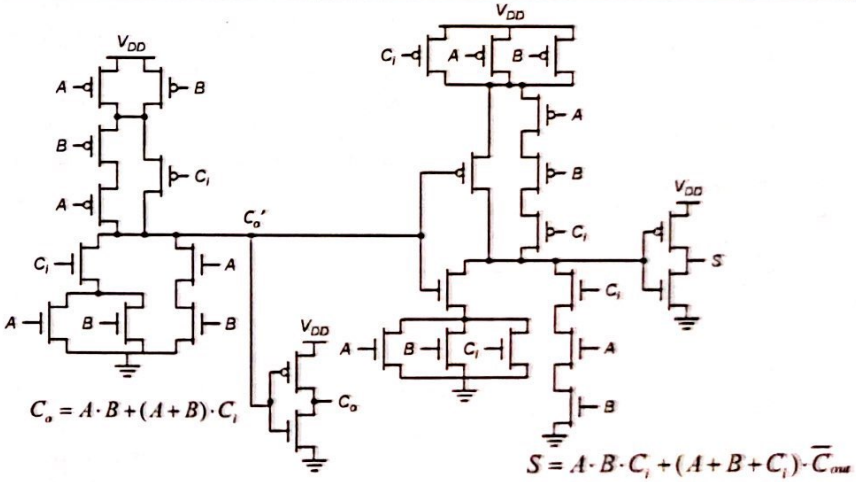
$$Sum = A\oplus B\oplus C_{in}$$
$$A\oplus B = 0 \Rightarrow S = 0\oplus C_{in} = C_{in}$$
$$A\oplus B = 1 \Rightarrow S = 1\oplus C_{in} = \overline{C}_{in}$$

$$C_{out} = A\cdot B + C_{in}\cdot(A+B)$$
$$= A\cdot B + C_{in}\cdot(A\oplus B)$$



$$C_{out} = A\cdot B + (A+B)\cdot C$$

28-transistor CMOS full-adder

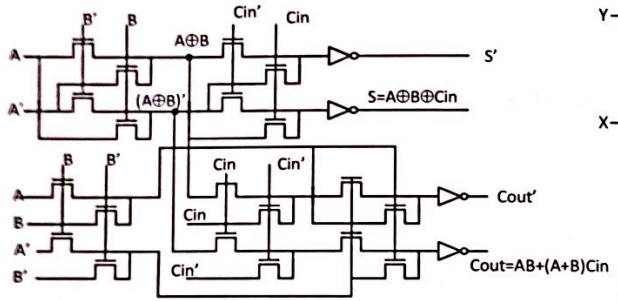


$$C_o = A\cdot B + (A+B)\cdot C_i$$

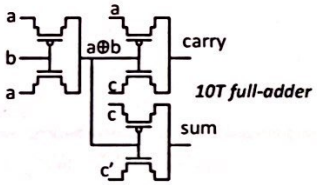
$$S = A\cdot B\cdot C_i + (A+B+C_i)\cdot\overline{C}_{out}$$

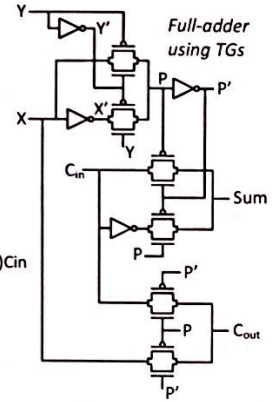
$$S = A\cdot B\cdot C_i + (A+B+C)Cout' = A\cdot B\cdot C_i + (A+B+C)[(A'+B')(A'B'+C')]$$
$$= A\cdot B\cdot C_i + (A+B+C)(A'B' + A'C' + B'C') = A\cdot B\cdot C_i + AB'C' + A'BC' + A'B'C$$

FA using pass transistor and TGs



28 transistors, dual-rail complementary pass transistor
Beware of threshold drops

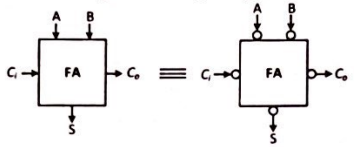




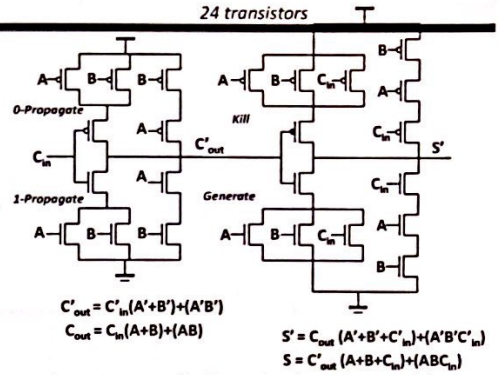
$S = (X \oplus Y) \oplus Cin$
$Cout = (X \oplus Y)\, Cin + XY$
$S = (X \oplus Y)'\, Cin + (X \oplus Y) Cin'$
$Cout = (X \oplus Y)\, Cin + (X \oplus Y)'\, X$
$Cout = (X \oplus Y)\, Cin' + (X \oplus Y)'\, X'$
$p = X \oplus Y$
$Cout = XY + (X \oplus Y)\, Cin$
$= p'X + p\, Cin$

The mirror adder

- Inversion property can be used to minimize critical path by reducing inverting stages

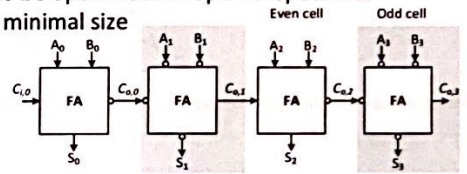


$\bar{S}(A,B,C_i) = S(\bar{A},\bar{B},\bar{C_i})$
$\bar{C_o}(A,B,C_i) = C_o(\bar{A},\bar{B},\bar{C_i})$



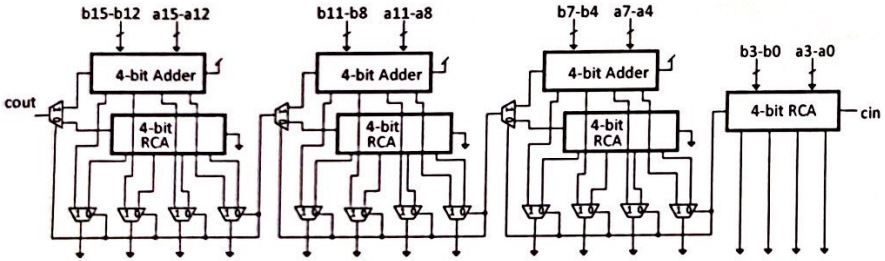
$C'_{out} = C'_{in}(A'+B')+(A'B')$
$C_{out} = C_{in}(A+B)+(AB)$
$S' = C_{out}(A'+B'+C'_{in})+(A'B'C'_{in})$
$S = C'_{out}(A+B+C_{in})+(ABC_{in})$

- The objective is to minimize the capacitance at node Cout. In the mirror adder only two transistors in series are in the carry-generation circuitry. Only the transistors in the carry stage have to be optimized for optimal speed. All transistors in the sum stage can be minimal size
- Utilizing the inversion property, RCA requires two different types of cells



Carry-select adder (CSA)

- The basic idea of a carry-select adder is to pre-compute the sums for both possible input values C_i=0 and C_i=1
- An 16-bit CSA is built from seven 4-bit RCAs and additional multiplexers. The three high order stages perform calculation twice, one for each of the two possible Ci values. When C4 is known, the correct s4-s7 output is chosen. While CSA uses more logic than a RCA, it reduces carry-propagate critical path



- An *N*-bit adder can be divided into two groups of *N*/2 bits and each group can be further divided into two sub-groups of *N*/4, and so on

Carry generation and propagation

- There are three possible cases when adding two bits:
  - If *ab*=00 : no carry propagation
  - If *ab*=10 or ab=01 : incoming carry will be *propagated*
  - If *ab*=11 : carry-out C_{out} will be *generated* regardless of incoming carry C_{in}

a	*b*	c_{in}	C_{out}	*sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- Thus there is exactly two ways a carry out can occur: either it is "generated", or it is "propagated" from the carry in to carry out

Carry generate: G = ab
Carry propagate: P = a ⊕ b

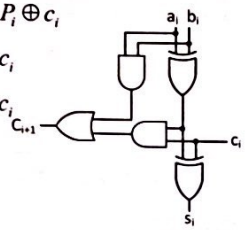
- G is known as the *carry generate* signal since a carry out (C_{out}) is generated whenever G=1, regardless of the input carry C_{in}
- P is known as the *carry propagate* signal since whenever P =1, the input carry is propagated to the output carry, i.e., $C_{out} = C_{in}$
- Note that whenever P_i =1, G_i =0 and vice versa

G	*P*	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

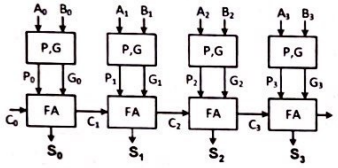
Carry-propagate and carry-generate

- The carry out and sum equations can be written in terms of carry-generate and carry-propagate as follows
  - a_i, b_i, and c_i are the inputs to the i-th full adder stage, and s_i and c_{i+1} are the sum and carry outputs from the i-th stage, respectively

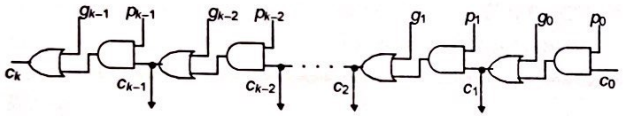
$$s_i = a_i \oplus b_i \oplus c_i = P_i \oplus c_i$$
$$c_{i+1} = a_i b_i + (a_i + b_i)c_i$$
$$= a_i b_i + (a_i \oplus b_i)c_i$$
$$= G_i + P_i c_i$$



- Consider the addition of two n-bit numbers $A=a_{n-1},a_{n-2},\ldots,a_0$ and $B=b_{n-1},b_{n-2},\ldots,b_0$
  - *Carry generate*: $G_i = a_i b_i$
  - *Carry propagate*: $P_i = a_i \oplus b_i$
- P_i and G_i can be computed in parallel for all bits
  - P_i's (G_i's) are valid one XOR-gate (AND-gate) delay after the two operands are made available



- Carry recurrence $C_{i+1} = G_i + P_i C_i$



Carry-lookahead adder (CLA)

- A CLA eliminates carry ripple and generates all incoming carries to all stages in parallel and hence, sum bits can be computed in parallel
- The values of P_i and G_i only depend on a_i and b_i. Thus the carry recurrence $C_{i+1} = G_i + P_i C_i$ can be *recursively unrolled* to obtain each carry signal directly from A and B and C_0, rather than from intermediate carries

$$\boxed{C_{i+1} = G_i + P_i C_i}$$

$C_1 = G_0 + P_0 \cdot C_0$

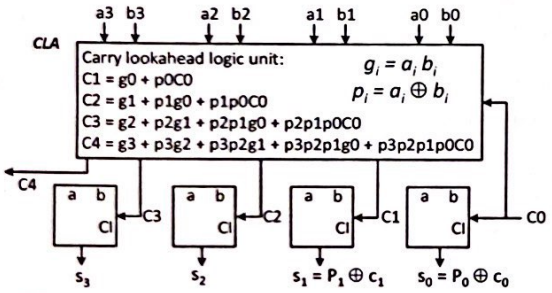
$C_2 = G_1 + P_1 \cdot C_1$
$C_2 = G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0)$
$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$

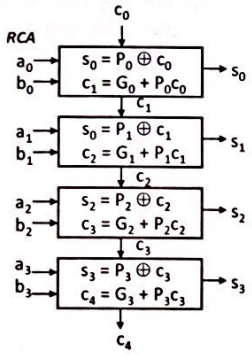
$C_3 = G_2 + P_2 \cdot C_2$
$C_3 = G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$
$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$

$C_4 = G_3 + P_3 \cdot C_3$
$C_4 = G_3 + P_3 \cdot (G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0)$
$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$

4-bit CLA

$$c_1 = G_0 + c_0 P_0,$$
$$c_2 = G_1 + G_0 P_1 + c_0 P_0 P_1,$$
$$c_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2,$$
$$c_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3$$





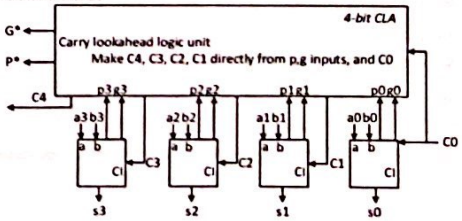
- The carry computations takes longer for every subsequent stage (gates with higher number of inputs). Carry C_k has (k+1) terms OR'ed, and the widest AND has k inputs. The more bits involved in an AND gate (higher fan-in gates), the longer it will take to propagate the signal through the gate as there will be more transistors in series. One solution is to use *multi-level look ahead adders*

Multi-level CLA

- In a multi-level look-ahead adder, N addition stages are divided into equal-sized blocks for modularity. Each block is implemented using a separate CLA
  - *Block Propagate* P* equals 1 if a carry-in to block propagates through the entire block to produce a carry-out (of the block): $P^* = P_i P_{i+1} \ldots P_{i+k}$
  - *Block Generate* G* equals 1 if the block generates a carry independent of the incoming carry : $G^* = G_{i+k} + P_{i+k} G_{i+k-1} + \ldots + (P_{i+1} P_{i+2} \ldots P_{i+k}) G_i$
- For example, the block propagate and block generate for a 4-bit adder can be written as:

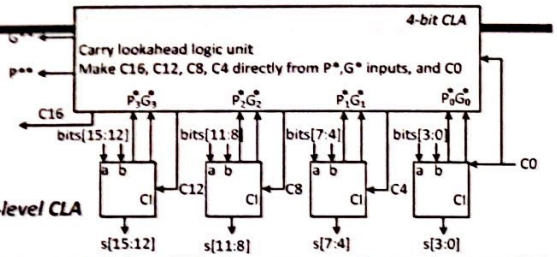
$$P^* = P_0 P_1 P_2 P_3$$
$$G^* = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1$$

- P_i^* and G_i^* represent the carry propagate and the carry generate for the 4-bit block, respectively. The block generate and block propagate signals, can be propagated to the higher-level lookahead adder
- The p's and g's logic depends on A and B and are separated from the CLA logic. The 4-bit CLA blocks can be used to produce propagate and generate signals for a two-level 16-bit CLA using the block generate G* and block propagate P* signals



Two-level CLA

- Multi-level CLAs are fast at the expense of a relatively large area



16-bit two-level CLA

- Each block generates block generate G_i^* and block propagate P_i^* signals in parallel
- Each block calculates individual sum bits with internal carry-look-ahead in parallel
- The block generate G_i^* and block propagate P_i^* signals are propagated to the higher-level carry lookahead logic
- The CL logic unit receives P_i^* and G_i^* of the groups and produces the carry inputs C_4, C_8, C_{12}, and C_{16} of each group

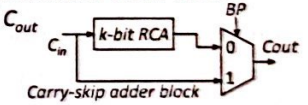
$$C_4 = G^*_0 + C_0 P^*_0$$
$$C_8 = G^*_1 + G^*_0 P^*_1 + C_0 P^*_0 P^*_1$$
$$C_{12} = G^*_2 + G^*_1 P^*_2 + G_0 P^*_1 P^*_2 + C_0 P^*_0 P^*_1 P^*_2$$

- A set of four blocks (16 bits) can be defined as a *section*. The CLA logic unit produces section-carry generate G** and section-carry propagate P** signals

$$P^{**} = P^*_0 P^*_1 P^*_2 P^*_3$$
$$G^{**} = G^*_3 + G^*_2 P^*_3 + G^*_1 P^*_3 P^*_2 + G^*_0 P^*_3 P^*_2 P^*_1$$

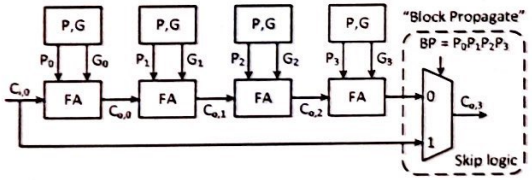
Carry-skip adder

- Since C_{in} to C_{out} represents the longest path in the ripple-carry-adder, the objective is to *accelerate carry propagation* through the adder. We know that for each input bit (a_i, b_i), the propagate-conditions is $P_i = a_i \oplus b_i$. The worst case delay for a RCA is when the propagate-condition P_i is true for each input bit pair (a_i, b_i), i.e., when $P_0P_1P_2P_3...P_n$=1. When all propagate-conditions are true, then the carry-in bit c_0 ripples through the n-bit adder and appears as the carry-out bit
- The idea of carry-skip adder is that when signal would propagate through carry chain, skip the carry propagation through the block. Hence, carry will skip over groups of consecutive FA stages for which P_i=1 (i.e., $a_i \oplus b_i = 1$ or $a_i \neq b_i$)
- Since a single *n*-bit carry-skip adder has no performance benefit compared to an *n*-bit RCA, in a carry-skip adder, the operands are divided into $\lceil n/k \rceil$ array of *k*-bit RCA blocks of equal size of *k*-bits. An *k*-bit carry-skip adder block consists of a *k*-bit RCA, and a *skip logic*. The skip-logic consists of a *k*-input AND-gate for *block propagate signal* BP = $P_0P_1P_2P_3...P_k$, which is the logic AND of all P_is in that block, for a group of *k* bits, and a multiplexer. If in each block all $a_i \# b_i$, the block's carry-in will skip (bypass) the entire block). The multiplexer switches either the last carry-bit c_n or the carry-in C_{in} to the carry-out signal C_{out}
- When BP=1, the carry-in will skip the block and bypass directly to the carry input of the next block

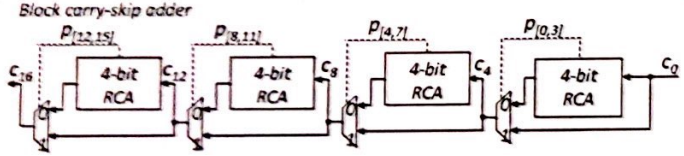


Carry-skip adder example

- Instead of making the critical path *short*, the idea is to make long paths *false*
- In this 4-bit carry-skip adder block, if block propagate BP=$P_0P_1P_2P_3$ = 1, then Co,3 = Ci,0 otherwise the block itself kills or generates the carry internally
- For each 4-bit carry-skip adder, the critical path for generation of the carry out bit is through one P,G unit and four full adders

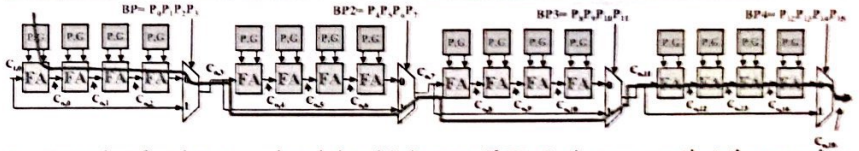


Block carry-skip adder



- All BP signals are generated in parallel and equally affect the critical path. Thus we only need to add the contribution of generating the carry out bit for a 4-bit adder once

16-bit carry-skip adder example



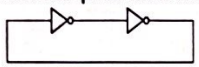
- Note that for the second and the third stage, if BP=0, that means that the carry in that stage will not be propagated. That means for that stage, the carry will be either generated or will be killed. So stage 2 and 3 will calculate their carry out at the same time as the first stage. Therefore, for the critical path computation we consider the path originating from the leftmost 4-bit adder and pass through three 2-to-1 multiplexers. Finally, the critical path is dependent on the computation of the most significant sum bit, which is a function of the propagate and carry-in bit: $S_{15} = P_{15} \oplus C_{0,15}$. $C_{0,15}$ is a function of the final 4-bit adder so the critical path must pass through the last four full-adders. Therefore, for a carry-skip adder, the critical path consists of the ripple path and the skip element of the first block, the skip paths that are enclosed between the first and the last block, and finally the ripple-path of the last block

Storage mechanisms

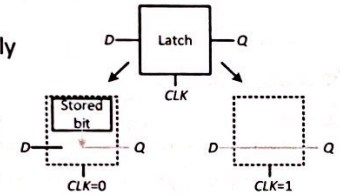
Positive and negative latches

Storage mechanisms

- Storage elements can be either *static* or *dynamic*
- *Static memories* are built using *positive feedback* or *regeneration* to preserve the state as long as the power is turned on

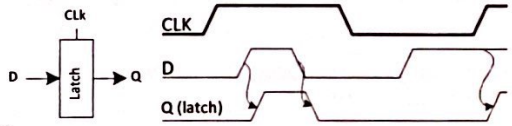


  - They consist of connections between the output and the input of a combinational circuit
- *Dynamic memories* store state for a short period of time—on the order of milliseconds
  - They are based on the principle of temporary *charge stored in parasitic capacitors* associated with MOS transistors
  - The capacitors have to be refreshed periodically to combat charge leakage, which increases with higher temperature
  - Dynamic memories tend to be more compact and smaller, resulting in significantly higher speed and lower power dissipation
- *Latches* and *flip-flops* are the two most commonly used storage or sequencing elements
- Both have at least three terminals: data input *D*, clock *CLK*, and data output *Q*



Positive and negative latches

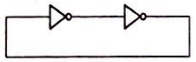
- A latch is *level-sensitive* circuit that passes the input *D* to the output *Q* when the enable control signal, which is usually the clock signal *CLK*, is active and passes the stored bit when *CLK* is inactive
- *Active high*, *transparent high*, or *positive latches* are transparent (i.e., passes input *D* to output *Q*) when *CLK*=1 and store the input on the falling clock edge
  - When CLK=1, the latch is in *transparent* mode, i.e., D flows through to Q as if the latch was just a buffer
  - When CLK goes to 0, the input data sampled on the falling edge of the clock is held stable at the output Q for the rest of the clock cycle, even if D changes. When *CLK* = 0 the latch is in *hold* (*opaque*) mode
- Before the first rising clock edge, the latch is in an unknown state
- *Active low*, *transparent low*, or *negative latches* are transparent when *CLK*=0 and store the input in the latch on the rising clock edge



- Therefore, a latch is transparent for a portion of the clock period, and stores the input on the clock edge that causes the latch to become opaque

How to build a latch?

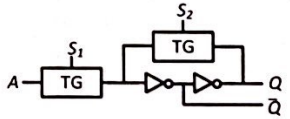
- To store data for an indefinite period of time, a feedback loop can be used to maintain or "store" the bit



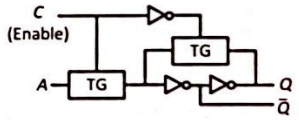
- The cross-coupling of two inverters results in a *bistable* circuit, that is, a circuit with two stable states, corresponding to 0 and 1 logic states
  - This circuit serves as a memory element, storing either a 1 or a 0
- In absence of any triggering, the circuit remains in a single state (assuming that the power supply remains applied to the circuit), and hence remembers a value
  - Since a stored value remains valid as long as the supply voltage is applied to the circuit, hence the name *static*
- A wide variety of *static* implementations exists for the realization of latches
- To store a bit in the loop, we need a way of loading an input bit into the feedback loop structure and making/breaking the feedback loop
- To make or break the connections, *transmission gates* (TGs) can be used
- TGs can be used to pass the input data while the latch is transparent and feedback to hold the data while the latch is opaque
  - Complementary TGs ensure that storage node is always strongly driven

Inverter-based latch

- This data latch (*D latch*) uses two TGs that are controlled using a control signal *C*
  - The control signal *C* is usually derived from the clock signal *CLK*, or a signal from the control unit
  - When C=1, the latch is in transparent mode
  - When C=1, the latch is in opaque mode (D gets held in the feedback loop)

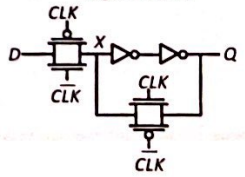


The latch is level-sensitive. If A changes while S1 = 1, then Q will change as well



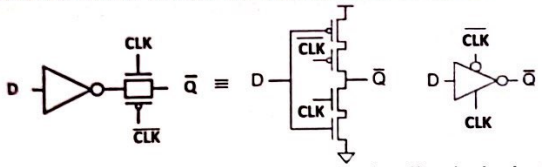
- The schematic of an active-low or negative latch is shown
  - *CLK* = 0: The left TG is on, *Q* follows *D* input (latch is transparent)
  - *CLK* = 1: The left TG gate is off, the latch output is determined by the storage node X
- An inverter and a TG can create a *clocked tri-state inverter*

Active-low or negative latch

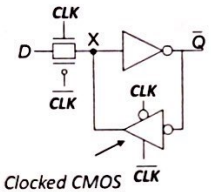


Inverter-based latch

- An inverter and a TG can create a *clocked tri-state inverter*

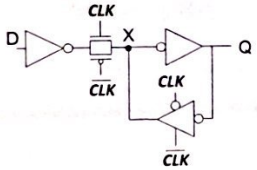


Clocked tri-state inverter



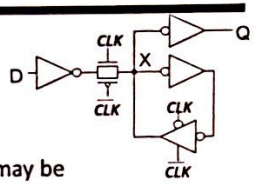
Clocked CMOS (C2MOS) feedback inverter

- In this inverting latch, when *CLK*=1, the feedback clocked tristate inverter is OFF and the latch is transparent
- When *CLK*=0, the input TG turns OFF, while, the feedback tristate turns ON, holding node *X* at the correct level
- Unfortunately, this design is sensitive to output noise: a large noise on the output can propagate backward through the feedback gates and corrupt the state node *X*
  - This is called *back-driving*
- The first noninverting latch adds an additional input inverter to buffer the input when *D* is too weak
  - This latch has also the back-driving risk



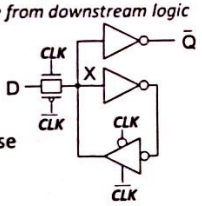
Inverter-based latches

- The second latch is static, all nodes swing rail-to-rail, the state X is isolated from output noise (no back-driving), and the input is buffered
  - This is a robust latch



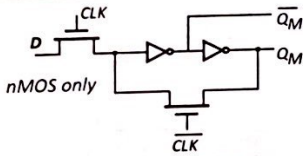
- If input noise can be better controlled, the input inverter may be omitted, as shown in the third latch
  - This inverting latch is faster and more compact
  - However, it has a relatively high *clock load* (the capacitive load presented to the clock signal (i.e., four transistors))
  - The clock load (i.e., the number of transistors that the clock is connected to) per latch should be kept as small as possible because the clock signal has an *activity factor* of 1 and thus accounts for much of the power consumption of the clock network

Output buffer shields storage node from downstream logic

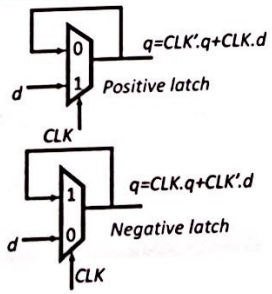


Latches with reduced clock load and multiplexer-based latches

- It is possible to reduce the clock load to two transistors by using nMOS only pass transistors
  - The use of nMOS only pass transistors results in the passing of a degraded high voltage of $VDD-V_{tn}$ to the input of the inverter on the left

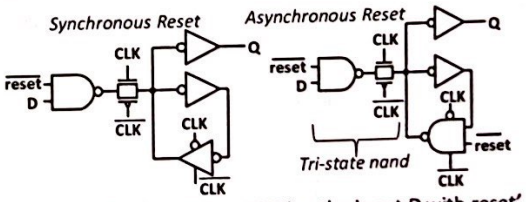


- This impacts both *noise margin* and the switching performance, especially in the case of low values of *VDD* and high values of V_{tn}
- Since the maximum input-voltage to the inverter equals $VDD-V_{tn}$, the pMOS device of the inverter is never completely turned off, resulting is a static current flow and hence, *static power dissipation* in the inverter on the left
- A latch can be implemented using a multiplexer with a feedback
- This *positive latch* is transparent when *CLK*=1
- The feedback holds the output stable while *CLK*=0
- The latch stores the input on the falling clock edge
- This negative latch is transparent when *CLK* = 0 and stores the input on the rising clock edge



Resettable latches

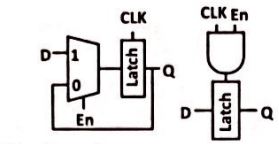
- Sequencing elements typically require a reset signal to enter a known state at initialization and ensure deterministic behavior
- There are two types of reset: *synchronous* and *asynchronous*
- Asynchronous reset forces *Q* low immediately, while synchronous reset forces Q low on the active clock ednge



  - Synchronous reset simply requires ANDing the input *D* with *reset'*
  - Asynchronous reset requires gating both the data and the feedback to force the reset independent of the clock
- Settable latches are similar to resettable elements but replace NAND with NOR and *reset'* with *set*

Enabled latches

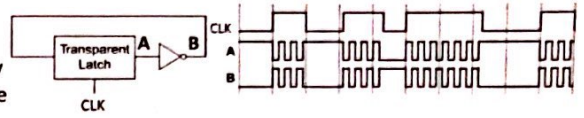
- A latch with enable input *En* can be implemented using an input *multiplexer* or *clock gating*
  - When *En* = 0, the latch retains its state independently of the clock
- *En* must be stable while the clock is high to prevent glitches on the clock input to the latch
- Multiplexer adds area and delay from input data *D*
- Clock gating does not affect delay from the data input and the AND gate can be shared among multiple clocked elements
- Clock gating also reduces power consumption because the clock on the disabled latches does not toggle
- However, the AND gate delays the clock, potentially introducing *clock skew*



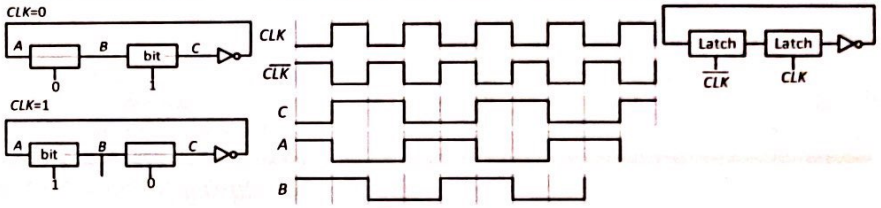
Mux-based enabled latch *Clock gating*

Problem with transparency

- Level-sensitivity of latches allows any momentary changes on the input to be passed to the output. This could be problematic if the input of a latch depends on the output of the same latch. Let's design a system that flips a stored bit "once" whenever *CLK* goes high
  - When the *CLK* signal goes high, the output will follow the input and passes all the changes on A to B
  - In this example, instead of the desired bit toggle when *CLK=1*, the output oscillates

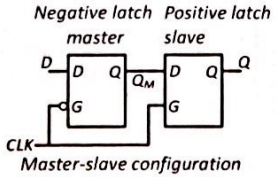


- The problem with transparent, level-sensitive latches can be fixed by cascading two transparent latches that trigger on opposite clock edges to separate the input from the output. This design flips a stored bit whenever *CLK* goes high

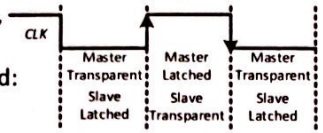


Building an edge-triggered flip-flop (FF) using two latches

- A flip-flop is a single bit storage unit that can be built using two cascaded latches
- These two latches trigger on opposite edges and hence, they are called *master-slave latch pair* or *master-slave flip-flop*
- Two level-sensitive latches, one negative latch at the master stage and one positive latch at the slave stage, build an *edge-triggered flip-flop*
- When CLK=0, the master stage is *transparent* and the D input is passed to the master stage output, Q_M
  - During this period, the slave stage is in the *hold* mode, keeping its previous value
- On the rising edge of the clock, the roles are reversed: the master stage stops sampling the input, and the slave stage starts sampling
  - After rising edge of the clock and during the high phase of the clock, the slave stage samples the output of the master stage Q_M
  - Since the master stage is in a hold mode, Q_M is not changing during the high phase of the clock and hence, <u>the output Q makes only one transition per cycle</u>
  - For the rest of the clock cycle, the FF is opaque and hence, the flip-flop value is held, and the input of the flip-flop has no effect on the output

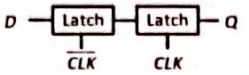


Master-slave configuration

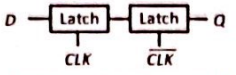


Positive and negative edge-triggered flip-flops

- Contrary to *level-sensitive* latches, a master-slave flip-flop (FF) is an edge-triggered element: data is sampled (captured) on the active (rising or falling) edge of clock
  - In a positive edge-triggered flip-flop, the master latch uses the negative clock phase to latch the input into the first latch and uses the positive clock to change the output with the second latch
  - <u>The value of Q is the value of D right before the rising edge of the clock</u>, achieving the *positive edge-triggered* effect, i.e., it uses the <u>negative clock phase</u> to latch the input into the master and uses the <u>positive clock edge</u> to change the output with the slave latch
- In a negative edge-triggered flip-flop, the master latch uses the positive clock phase to latch the input into the positive master latch and uses the negative clock to change the output with the negative slave latch



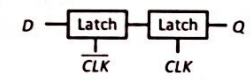
Positive edge-triggered FF

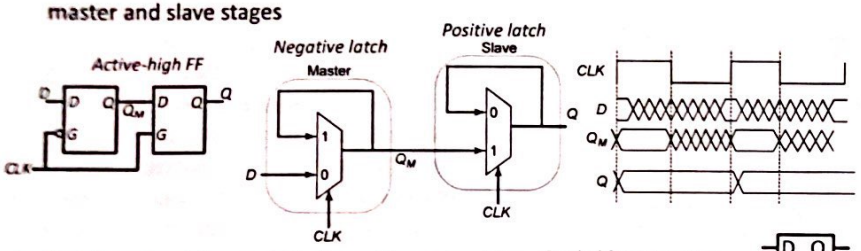


Negative edge-triggered FF

Master-slave flip-flops

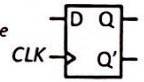
- Master-slave is the most common approach for implementing an edge-triggered FF
- Multiplexer-based latches can be used to realize the master and slave stages





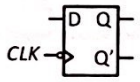
- *Positive edge-triggered flip-flops:* The output Q which is the bit stored in the FF will change on the positive edge of the *CLK*
- *Negative edge-triggered flip-flops:* The output Q, will to change on the negative edge of the *CLK*

Symbol for rising-edge triggered D flip-flop



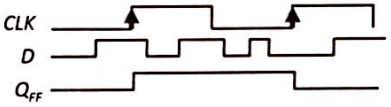
The triangle means edge triggered clock input

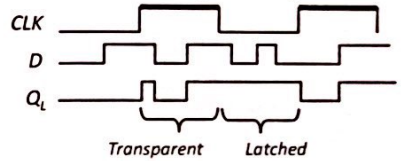
Symbol for falling-edge triggered D flip-flop



Latch versus flip-flop

- Flip-flops are edge sensitive and latches are level sensitive
- An active-high *D*-latch passes input *D* to output *Q* when *CLK* is high (transparent) and is latched when clock is low (i.e., ignores changes on input *D*)
- Flip-flop is edge triggered: Positive edge-triggered FFs store the input at a rising clock edge and negative edge-triggered FF store the input at a falling clock edge
- The output behavior of a latch and a FF are the same unless input changes while the clock is high

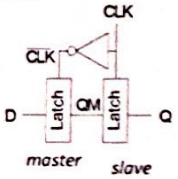


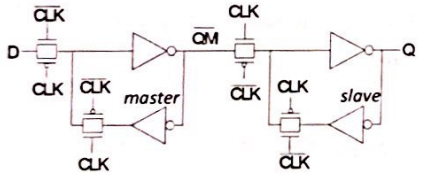


- A latch is transparent for a portion of the clock period, and stores the input on the clock edge that causes the latch to become opaque
- Contrary to *level-sensitive* latches, *edge-triggered* FFs sample the input value at the rising or falling clock transition — 0-to-1 for a *positive edge-triggered* FF and 1-to-0 for a *negative edge-triggered* FF
- For the rest of the clock period, FFs are opaque, and the input of the FF cannot affect the output (i.e., it ignores the input at all other times)

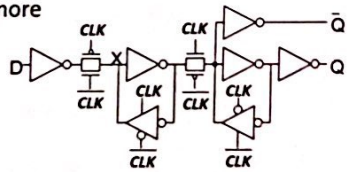
Inverter-based flip-flops

- A positive edge-triggered inverter-based FF is shown
- The FF has a clock load of 8 transistors (ignoring the overhead required to invert the clock signal since the buffer inverter overhead can be amortized over multiple register bits



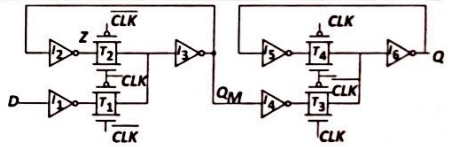


- This FF uses tri-state inverters and adds three more inverters to implement a noninverting static FF
- Most standard cell libraries employ this design because it is robust



Inverter-based flip-flop operation

- Consider this master-slave positive edge-triggered FF



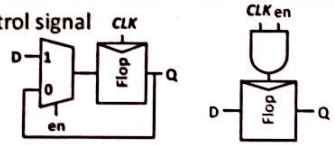
- During *CLK*=0, T1 is on, T2 is off, T3 is off and T4 is on
  - The D input follows the path I1, T1, and I3 and is sampled onto node QM
  - The cross-coupled inverters I5, I6 holds the state of the slave latch
- When *CLK*=1, *T1* is *off* and *T2* is *on*
  - The master stage stops sampling the input and goes into the hold mode
  - The cross coupled inverters I2 and I3 holds the state of QM
  - T3 is on and T4 is off, and QM is transferred to Q
- Since the output arrives at the positive edge of *CLK*, it is a positive edge triggered FF
- Ignoring the overhead required to invert the clock signal each register has a clock load of 8 transistors

Flip-flops with control inputs

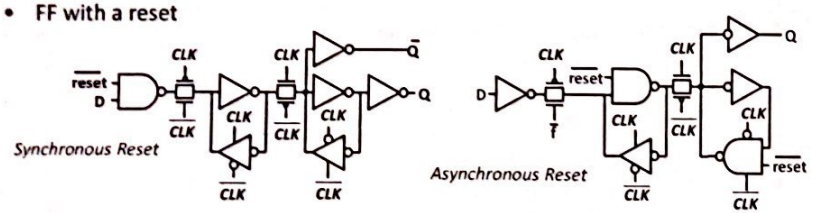
Flip-flop-based vs. latch-based designs

Flip-flops with control inputs

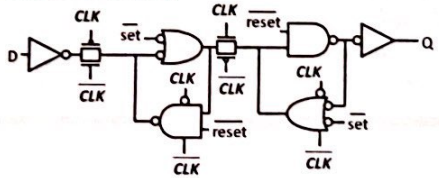
- FF with an enable *en* control signal



- FF with a reset



- FF with asynchronous set and reset



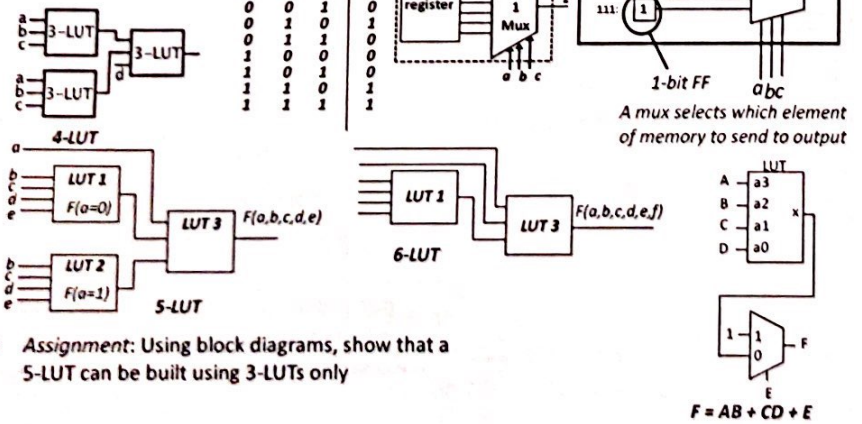
Flip-flop-based vs. latch-based designs

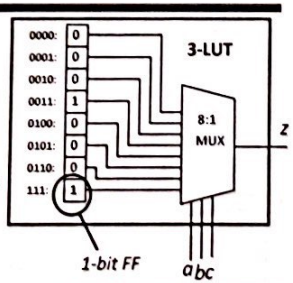
- A master-slave flip-flop uses twice as much logic than a latch. Edge-triggered designs require more transistors (FFs versus latches) Edge-triggered designs can be slower than latch-based designs. Latch-based designs offer greater density and higher performance than FF-based designs. Latch-based designs were popular in the past, especially for some high-performance processor designs. However, a level-sensitive circuit design requires additional care to make it operate correctly because state changes are not instantaneous as in the edge-triggered scheme. Any glitches on the latch's input data are propagated directly through to the output while a register-based design may be larger than its latch-based equivalent, but it will be more robust. The cost of the increased complexity of latch-based design has risen significantly with the increase in design size and the need for design reuse. Today, deep submicron technology has made billions of transistors available to the chip designer and in most designs, the size of on-chip memory is more than the size of the datapath. Moreover, delays are dominated by interconnect delay, so the difference in the effective delay between latches and FF is minimal. Most systems are FF-based

Look-up table (LUT)

- A Boolean function of *n* inputs can be implemented using an *n*-LUT (i.e., a LUT with 2^n inputs)
  - e.g., A 4-LUT can implement any function with four inputs and one output
- Example: $z = ab + bc'$

a	b	c	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

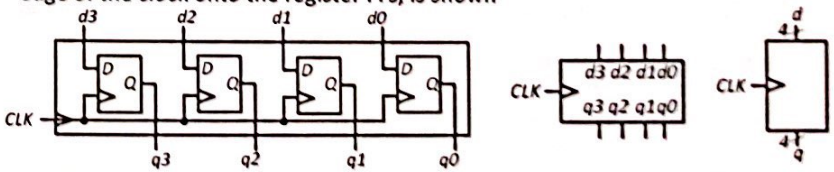




- *Assignment*: Using block diagrams, show that a 5-LUT can be built using 3-LUTs only

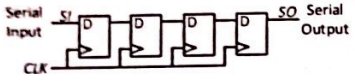
Registers

- Digital systems require storage elements or memories to store data used and generated by computational units
- An *n*-bit *register* can be formed by grouping *n* FFs together sharing a clock signal
  - They store a relatively small amount of data
  - In essence, a FF is a 1-bit register
- A 4-bit register with parallel load, i.e., input data *d* can be loaded on the rising edge of the clock onto the register FFs, is shown

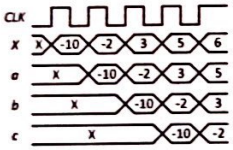


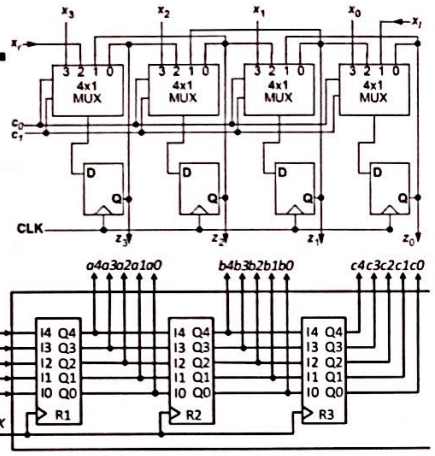
Shift registers

- A *shift register* is built by cascading a number of FFs with a serial input and a serial output

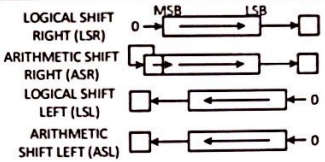


- 3-stage 5-bit shift register. New value goes into the register at stage K while the current value in stage K goes into register at stage K+1

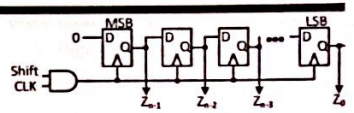




Logical and arithmetic shift units

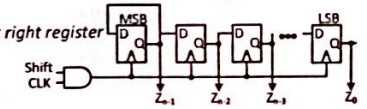


Logical and arithmetic shift lefts are the same

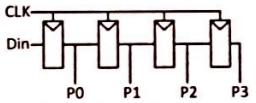


Logical shift right (LSR) register with parallel out

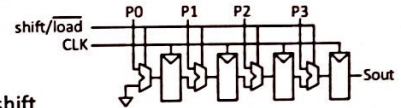
Arithmetic shift right register



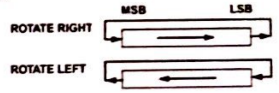
- Serial-in parallel-out shift register reads in serial data and after N clock cycles, an n-bit parallel output is available



- A parallel-in serial-out shift register, loads all *n* bits in parallel when load = 0 and shift one bit out per cycle when shift=1

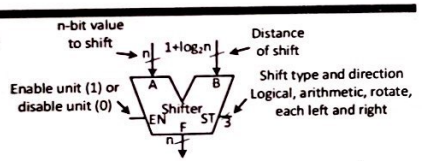


- A *rotate* register is the same as a logical shift register except that instead of dropping the bit off of the least or most significant positions in the shifting, the discarded bit is placed on the other end

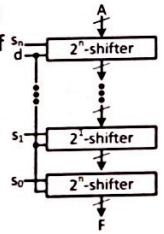


Barrel Shifter

- A *barrel shifter* shifts an n-bit vector by an arbitrary position instead of a fixed number of positions



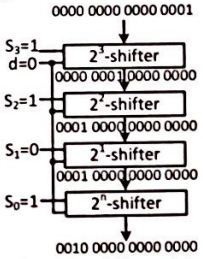
- In this barrel shifter, *d* input defines the direction of the shift (d=0 to the left and d=1 to the right) and $s_n \ldots s_1\ s_0$ define the number of bit positions that input *A* should be shifted
- The first level shifts *A* by 2^n bits if $S_n = 1$. If $S_n = 0$, A is not shifted and will be passed to the output
- Subsequent levels shift their corresponding input by 2^k bits, depending on the value of their input



Examples

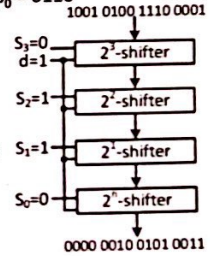
- The following barrel shifter can shift the input by an arbitrary number of positions up to 15 bit positions in either direction left or right
- Consider the input of 0000 0000 0000 0001
- For a logical shift to the left by 13 positions, d = 0 and $s_3s_2s_1s_0 = 1101$

Shift the input to the left by 13 bit positions

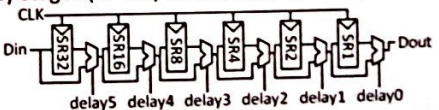


- Consider the input of 1001 0100 1110 0001
- For a logical shift to the right by 6 positions, d = 1 and $s_3s_2s_1s_0 = 0110$

Shift the input to the right by 6 bit positions



- A *tapped delay line* is a shift register with a programmable number of stages. The number of delay stages (0 – 63) can be set with delay controls to mux

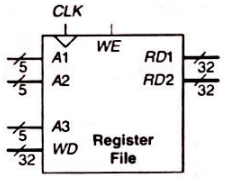


Register files

Register file

Register files

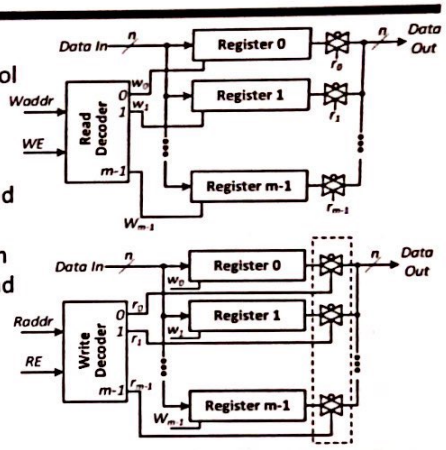
- Digital systems use individual registers to store values of variables. A *register file* (RF) is built as a relatively small, multi-ported static random access memory (SRAM) array, because it is more compact than an array of flip-flops
- An *mxn* register file has *m* registers that are each *n*-bits wide. Any register can be read or written by specifying the address of the register
- Register files have one or more read/write ports. Each port gives read and/or write access to one memory address. Multi-ported register files can access several addresses simultaneously
- A 32 x 32 RF has two read ports (*RD1* and *RD2*) and one write port (*WD*) for reading data from and writing data onto the RF, respectively. *A1* and *A2* are the two address ports for *RD1* and *RD2*, respectively, and *A3* is the address port for the write port *WD*. Two registers can be read and one register written simultaneously on the clock *CLK* edge. Writing the value of the *WD* port occurs only if the write enable *WE* signal is 1



- Intel's Itanium, register file has 128 registers with 8 read ports and 4 write ports

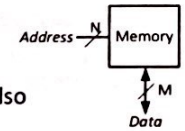
Register file

- A *m*×*n* RF file with *m* registers that are each *n*-bits wide is shown. r_k is the control signal for the output transmission gates (multiplexer) and w_k is the write enable signal of the registers. The r_k and w_j signals indicate which register to read and write, respectively. For reading, a given address `Raddr` (with `RE = 1`) selects which register, 0 through *m* - 1, to read from and output to `DataOut`. For writing to a register, a given address `Waddr` selects which register to store the input from `DataIn` when `WE = 1`

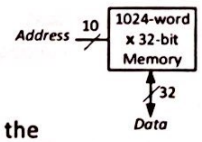


Memory

- Registers are built from flip-flops and store a relatively small amount of data
- *Memory arrays*, or just *memories*, are instead used to efficiently store large amounts of data
- A memory is organized as a two-dimensional array of *memory cells*, each cell storing one bit



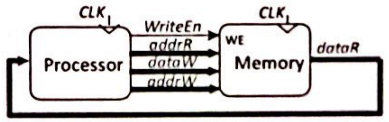
- The memory reads or writes the contents of one of the rows (also called a *word*) of the array
  - This row is specified by an *address* and the value read or written is called *data*
- A memory with an *N*-bit address port and an *M*-bit data port has 2^N rows (words) and *M* columns
  - Each row of data is called a *word* and hence, the array contains 2^N *M*-bit words



- The *depth* of an array is the number of words, and the *width* is the number of columns, also called the *word size* or *wordlength*
  - The *size* of an array is given as *depth* x *width*
  - The total size of this 1024-word x 32-bit array is 32 kilobits (Kb)
- Memory arrays are specified by their size (depth x width) and the number and type of ports

Processor and memory interface

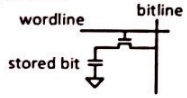
- Programmers can assume the memory is "infinite" and it is called *virtual memory*
- In reality, *physical memory* size is significantly smaller
- The system software and hardware map virtual memory addresses to physical memory space transparently to the programmer
- The processor communicates with the memory over a *memory interface*
  - The processor sends an address over the *address bus* to the memory
  - For a write, *WriteEn* is 1 and the processor sends address on *addrW* and data on the *dataW* bus to memory
  - For a read, *WriteEn* is 0 and the memory returns the data on the *dataR* bus from *addrR* location



- The internal memory that is used in the datapath and control unit is *foreground memory*, and is most often organized as individual registers or register files
- External memory achieves higher area densities through efficient use of array structures and by trading off performance for size

Dynamic random access memory (DRAM)

- In a *random access memory* (RAM) any data word is accessed with the same delay as any other
  - A *sequential access memory*, such as a tape recorder, accesses nearby data more quickly than faraway data (e.g., at the other end of the tape)
  - RAM is *volatile;* it loses its data when the power is turned off
- DRAM stores a bit as the presence or absence of charge on a capacitor
- When the capacitor is charged to VDD, the stored bit is 1 and when it is discharged to GND, the stored bit is 0
- The capacitor is *dynamic* because it is not actively driven high or low by a transistor tied to VDD or GND
- The nMOS transistor behaves as a switch that either connects or disconnects the capacitor from the bitline
  - When the wordline=1, the stored bit value transfers to or from the bitline
  - One transistor per bit: small size; Ideal for large memories
  - Built with a special process optimized for density
  - Low power consumption
  - Relatively slow
  - Relatively inexpensive



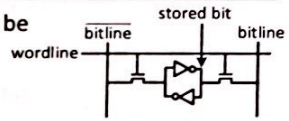
DRAM bit cell

Read and write operations

- To write a bit cell:
  - The bitline is strongly driven to the desired value
  - Then the wordline =1, connecting the bitline to the stored bit
  - Upon a write, data values are transferred from the bitline to the capacitor
- To read a bit cell:
  - The bitline is left floating (Z)
  - Then the wordline=1, allowing the stored value in the capacitor to drive the bitline to 0 or 1
- Charge on the capacitor gradually leaks away through the transistor (resistor)
- Also, reading destroys the bit value stored on the capacitor, so the data word must be rewritten after each read
- Therefore, DRAM cells must be periodically *refreshed* (read and rewritten) every few milliseconds
  - That is why this memory structure is called dynamic
  - DRAMs refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle
  - Cycle time consists of latency (finding the right word and preparing to access it) and transfer time

Static random access memory (SRAM)

- Static RAM is *static* because stored bits do not need to be refreshed
- Static RAM stores data using a pair of cross-coupled inverters
- Each cell has two outputs, bitline and bitline'
- 4 transistors for storage and 2 transistors for access
  - More area and power consumption than a DRAM cell
- When the wordline =1, data values are transferred to or from the bitlines
- Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value



SRAM bit cell

- SRAMs:
  - Low density (six transistors per bit)
  - Large area on die
  - Built with normal high-speed CMOS technology
  - High power consumption
  - Faster than DRAMs: DRAM *latency* is longer than that of SRAM
  - Relatively expensive
  - DRAM also has lower *throughput* than SRAM, because it must refresh data periodically and after every read

SDRAMs, DDRs, and DIMMs

- *Access time, response time, or latency,* is the amount of time it takes for the memory to respond to a read or write request
- *Memory cycle time* is the time between the start of one RAM access to the time when the next access can be started
  - SRAMs have a fixed access time to any data, though the read and write access times may differ
  - SRAMs don't need to refresh and so the access time is very close to the cycle time
- To eliminate the time for the memory and processor to synchronize, *synchronous DRAMs* or *SDRAMs* use a clock signal
- If data is transferred on both the rising *and* falling edge of the clock, the DRAM is called *double data rate* (DDR) SDRAM. It provides twice as much bandwidth
  - A 1600 MHz DDR4-3200 DRAM can transfer 3200 million data per second
- While Personal Mobile Devices, such as iPad, use individual DRAMs, memory for desktops are commonly installed on small boards called *dual inline memory modules* (DIMMs)
  - DIMMs typically contain 4–16 DRAMs, and they are normally organized to be 8 bytes wide
  - A DIMM using DDR4-3200 SDRAMs could transfer at 8 × 3200 = 25,600 megabytes per second

Read only memory (ROM)

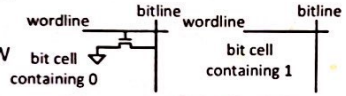
Combinational logic vs. sequential logic

read

- wide
  - A DIMM using DDR4-3200 SDRAMs could transfer at 8 × 3200 = 25,600 megabytes per second

Read only memory (ROM)

- A *read only memory* (ROM) could only be read but not written
  - ROMs are randomly accessed too and most ROMs can be written as well as read
  - RAMs are volatile and ROMs are nonvolatile, i.e., it retains its data indefinitely, even without a power source
  - ROMs take a longer time to write than RAMs
- ROM stores a bit as the presence or absence of a transistor
- To read the cell, the bitline is weakly pulled HIGH
- Then the wordline = 1
  - If the transistor is present, it pulls the bitline LOW
  - If it is absent, the bitline remains HIGH

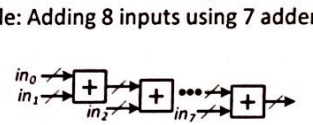


ROM bit cell

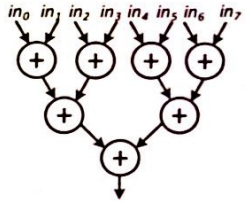
- There are many flavors of ROMs that vary by how they are written and erased
- *Flash memory* is a nonvolatile semiconductor memory
  - It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks
  - Access times are about 5 to 50 microseconds

Combinational logic vs. sequential logic

- Combinational circuits are built using logic gates
- Combinational circuits' outputs at any point in time are a function of its *current* input signals by some Boolean function, assuming that enough time has elapsed so that the transients through the logic gates have settled
- Example: Adding 8 inputs using 7 adders



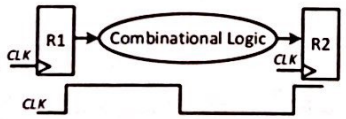
n-1 adders are required for combinational design



- This is in contrast to sequential circuits, for which the output is not only a function of the current input data, but also of previous values of the input signals
  - In other words, a sequential circuit remembers some of the past history of the system
  - Sequential circuits thus include memory or *storage elements*, which represent the *state of the circuit*
  - The purpose of state registers is to enforce sequence, to distinguish the *current state* from the *previous* or *next state*
  - Therefore, they are also called *sequencing elements*

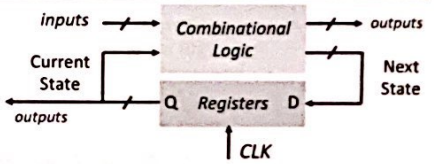
Clocking scheme

- A *clocking scheme* defines when data can be read from and when they can be written into storage elements. Storage elements include latches, flip-flops, registers, register files, and memory
- In an *edge-triggered clocking* values stored in sequential elements are updated only on an active clock edge. The outputs of storage elements can be used in the current clock cycle and the input values to the storage elements will be written into registers in the following clock cycle
- Since only registers can store data values, a combinational logic must receive its inputs from registers and its outputs written into registers
- Synchronous designs eliminate the problems associated with speed variations through different paths of logic (fast and slow paths)
- Interfacing between two blocks of logic is simplified by the synchronous behavior. *Asynchronous interfaces* demand handshaking or token passing to ensure integrity of information while synchronous designs with known timing characteristics can guarantee correct reception of data

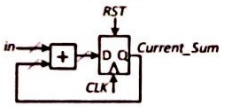


Sequential logic

- In *synchronous sequential systems*, all registers are under control of a single *global clock* signal
- The clock signal is used to update the state of the register simultaneously

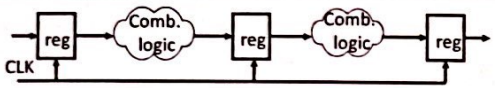


  - State registers hold the current state of the sequential circuit
  - The the next state is computed based on the current state and the current inputs using a combinational logic
  - This allows our system to perform operations in a sequential manner (i.e. one after another)
- The output values depend on current input values and present state (i.e., previous input values)
- A sequential adder requires only one adder to sum the inputs serially over multiple clock cycles



Sequential logic

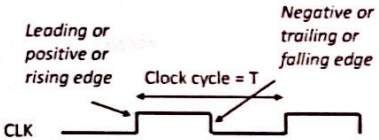
- Most digital systems are synchronous sequential logic



- In a *synchronous sequential system*, the clock signal orchestrates the sequence of operations
  - All state (i.e., content of the registers) changes occur immediately following the active edge of the clock signal
  - Synchronous clocking is used to synchronize computations and transfer of data
  - Synchronizing using a given clock simplifies design and avoids the need for circuits to signal the completion of an operation
  - All *switching transients* occur between the clock pulses and do not propagate among stages
- In a synchronous logic, interfacing between two blocks of logic is simplified
- *Asynchronous interfaces* requires *handshaking* to transfer data between stages
- Well-developed design methodologies, support of CAD tools, relatively easy to design, debug, and test, are the key reasons for utilizing synchronous systems

Clock cycle and clock frequency

- Synchronous systems use a periodic clock signal to keep operations in sequence (i.e., causes state of memory elements to change simultaneously)
- The clock signal has two edges: rising edge and falling edge
  - We use the convention that the clock rise occurs in the beginning of the clock period
- The *clock period*, or *cycle time T*, is the time between the consecutive positive (rising) or negative (falling) edges



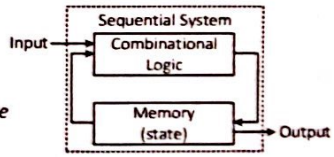
- *Clock frequency* or *clock rate* is the reciprocal of the *clock period* (or *clock cycle*) $f=1/T$, determines the speed at which the circuit operates
  - Frequency is measured in units of *Hertz* (Hz), or *cycles per second*
  - 1 Hz = 1 cycle/sec 1 KHz = 10^3 cycles/sec
  - 1 MHz = 10^6 cycles/sec 1 GHz = 10^9 cycles/sec
  - 2 GHz clock has a cycle time = $1/(2\times10^9)$ = 0.5 nanosecond (ns)

Datapath + Control

- A digital system operates in a finite set of states
- The behavior of the system at a given point in time depends upon the current state and the inputs
- A circuit can be decomposed into two major sub-components: the *datapath* and the *control unit*
- The datapath performs the computation required for the functional description of the design
- The control unit contains two CLs: computing the next state and computing the output control signals for the datapath
  - The inputs to the datapath from the control circuit include multiplexer select lines, enables for registers, and operation selection of datapath components
- The design process starts with the design partitioning to datapath and control unit; Then the functional description of the hierarchical design is described in HDL for the datapath and control unit; The the design is simulated for functional simulation and finally synthesized and implemented.

Finite state machine (FSM)

- A *finite state machine* (FSM) is a sequential circuit that is designed to sequence through specific patterns of finite states based on the inputs and present state of the FSM
- A finite state machine has:



- **1.** A finite set of states $S_1, S_2, ..., S_k$
  - One of the states is distinguished as the *initial state*
- **2.** A set of binary inputs: $I_1, I_2, ..., I_m$
- **3.** A set of binary outputs: $O_1, O_2, ..., O_n$
- **4.** A set of state transitions specifying, for each choice of current state S_i and input values, a next state S_j (i.e., transition function)
  - The transitions depend on the current state and external input
- For a given clock cycle, the FSM has only one current state
  - The current state may change upon the edge of clock signal
- **5.** An output function determines the values of output control signals given just the current state or given the current state and the input values
- On every clock cycle, a FSM samples inputs, computes next state, may change state, and computes outputs

FSM variations

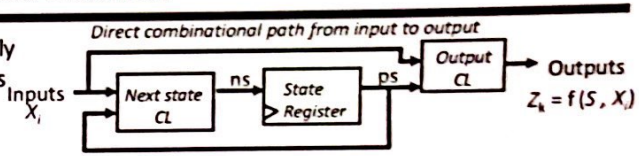
Mealy and Moore state machines

FSM variations

- There are two variations of FSMs, *Mealy* and *Moore*
  - Named after George Mealy and Edward Moore, who studied control unit design in the 1950s
- In both Mealy and Moore FSMs, the FSM consists of three components:
- **1.** *State register*: The register is used to hold the state ("present state" or simply "state") of the FSM
- **2.** *Next state logic*: An FSM can only be in one "state" at any given time, and every clock cycle causes it to change from its "current state" to the "next state", as defined by the next state logic
  - The next state is a function of the FSM's inputs and its present state
- **3.** *Output logic*: Outputs are a function of the current state in Moore FSMs and a function of present state and the FSM's inputs in Mealy FSMs
- Moore and Mealy FSMs are thus distinguished by their output generation

Mealy and Moore state machines

- The outputs of a Mealy machine are functions of both present state (ps) and inputs

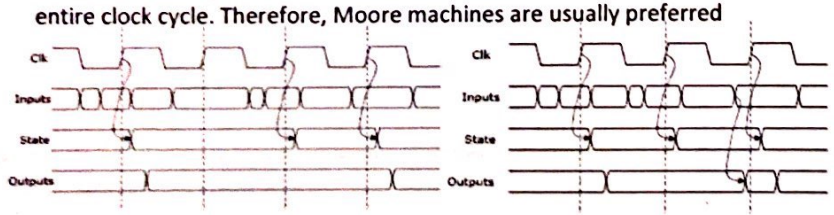


- On every clock cycle, the two CL compute outputs and next state (ns). Both the next state and the output CL are functions of the current state and the inputs
- In a Mealy machine, an input change may cause an output change. So the outputs may be asynchronous
- The outputs of a Moore FSM are functions of only the present state

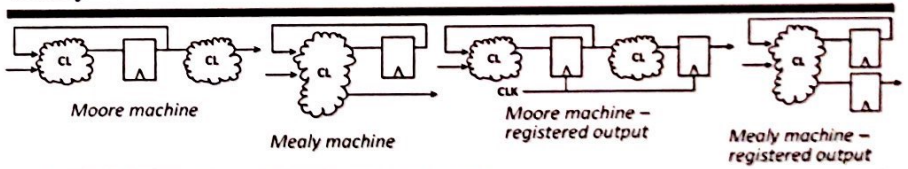
- Next state depends on the present state and the inputs
  - Outputs change *synchronously* with state changes
- The output logic in Moore machines can often be omitted by choosing the state encoding appropriately
- In both Mealy and Moore machines, the present state may change only on the clock edge. A state change doesn't necessarily cause an output change

Mealy vs. Moore

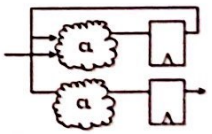
- The input-output response of Moore machines is slower than Mealy machines since the output from a Moore machine can change in response to an input change only after the state transition, which occurs on the active edge of the clock. Thus there is an input-output response delay of up to one clock cycle in Moore machines
- A system using a Mealy machine can be faster because an output may be produced immediately as a function of inputs instead of at the next clock cycle
- Moreover, a Mealy machine might be specified and implemented using fewer states because it is capable of producing different outputs in a given state as a function of the present state and inputs as opposed to a set of output values per state, which are only functions of present state, in Moore machines. However, Moore machines produce glitch free outputs and the outputs are stable for an entire clock cycle. Therefore, Moore machines are usually preferred



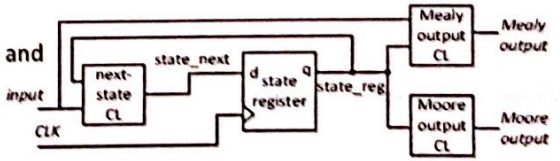
Mealy vs. Moore



- What is the difference between Mealy and Moore machines with registered outputs? Let's draw the Moore machine with registered output differently

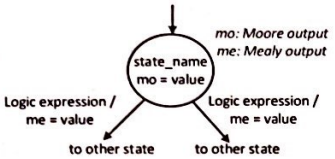


- It is recommended that inputs to FSMs be synchronous
- Make sure your FSM does not have dead states and all state machine outputs are always registered
- Also, separate the state machine logic from all arithmetic functions and datapaths descriptions
- A FSM can have both Moore and Mealy outputs



State transition diagram

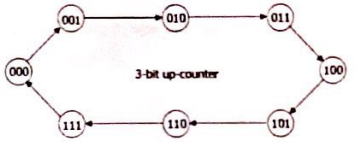
- A *state transition diagram* is composed of *nodes*, which represent states and are drawn as circles, and *transitions* between states, which are represented by arrows among states
- A logic expression expressed in terms of input signals is associated with each transition arc and represents a specific condition



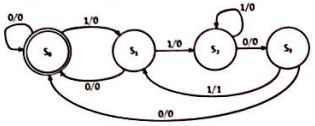
- The transition is taken when the corresponding expression is evaluated true
- Arcs leaving a state are mutually exclusive, i.e., for a set of input values there's at most one applicable arc
- Arcs leaving a state are collectively exhaustive, i.e., for any combination of input values there's at least one applicable arc
- So for each state and for any combination of input values, there is exactly one applicable arc

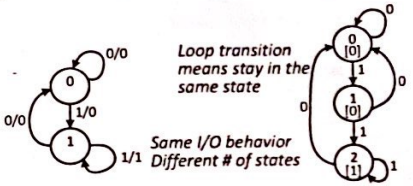
FSM examples

- Output may be either part of state (for Moore machines) or on arcs (for Mealy machines). Moore output values (mo) are placed inside the circles (states) since they depend only on the present state. Mealy output values (me) are associated with the conditions of transition arcs since they depend on the present state and external input
- An n-bit binary counter consists of *n* flip-flops and can count in binary from 0 through 2^n-1. This example shows a 3-bit up-counter FSM that proceeds through a well-defined state sequence



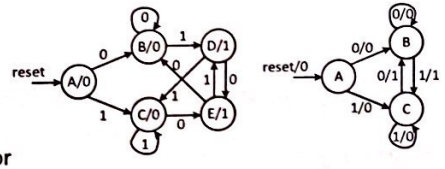
- This example shows the FSM for a 1101 sequence detector



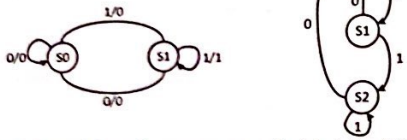


Vending machine FSM

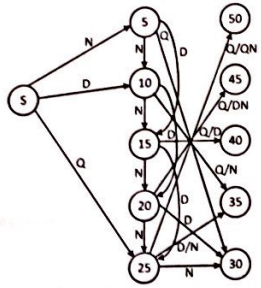
- The FSM procures a "1" when a 0 to 1 or a 1 to 0 transition occurs on the input signal "x" and produces "0" otherwise



- The FSM produces a "1" when two or more consecutive "1"s are entered. Otherwise the output is "0"



- The machine dispenses a soft drink for 30 cents. It accepts nickels N, dimes D, and quarters Q and returns change. The machine has a button for dispensing the drink and the other to return the deposited coins. The return button returns all coins the user has deposited, or a set of coins with the same value



Chapter 02 – Verilog Hardware Description Language

Verilog module

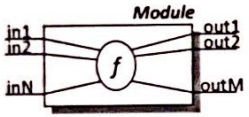
Verilog syntax

- Describe one Verilog module in a single file for an effective revision control

return button returns all coins the user has deposited, or a set of coins with the same value

Verilog module

- Verilog describes a digital system as a set of *modules*. A module is the the basic unit of digital logic in Verilog describing the functionality of logic
- Each module has a *port interface* for interconnecting to other modules




```
module module_name (ports declaration);
   local variables declaration;
   functional description of module
endmodule
```

- A module definition starts with the keyword **module** and ends with the keyword **endmodule**. **module_name** is a user-defined name for the model
- The port declaration defines the interface signals of a module to interconnect with other modules or primitive gates. All the ports in a list of port definitions must be declared
- Modules may contain local signal (variable) declarations. Internal or local variables are only used in the module and are neither inputs nor outputs
- Ports and the local variables of a module are informally called *signals*
- The functional description of a module is stated in the module's body
- A complete command in the module's body is called a *statement*. Each statement in a Verilog module ends with a semicolon, except **endmodule** and **end**

Verilog syntax

- Describe one Verilog module in a single file for an effective revision control
  - Typically, the file name is the same as the module name. This is not a requirement, but it helps to maintain the source code of large designs
- Nested module definitions are not allowed, i.e., a module cannot contain definitions of other modules.
- Identifiers using A ... Z, a ... z, 0 ... 9, and underscore. First character of an identifier must not be a digit and an _ (underscore) is used to enhance readability
- Use lowercase letters for all signal names, variable names, and port names and uppercase letters for control signals
- Note that Verilog is *case-sensitive*. For example, y1 and Y1 are different variables in Verilog. Using separate signals within a module that only differ in their capitalization can be confusing
- Do not use reserved words for names of any elements in your source files
  - Because designs must be portable from VHDL to Verilog and vice versa, it is important not to use VHDL reserved words in Verilog code (e.g., `entity`, `architecture`, `attribute`, `block`, `body`, `buffer`, `bus`, `component`, `configuration`, `constant`, `downto`, `abs`, `all`, `array`, `exit`, `file`, `generic`, `group`, `label`, `library`, `loop`, `next`, `null`, `others`, `port`, `range`, `record`, `register`, `signal`, `then`, `type`, `units`, `when`, `with`)

Comments

- Verilog comments are like comments in C
  - Comments beginning with /* continue, possibly across multiple lines, to the next */
  - Comments beginning with // continue to the end of the line
  - Multi-line comments may not be nested. However, there may be single line comments inside a multi-line comment
- Use comments to explain ports, signals, and variables
  - Comments should be brief, concise, and explanatory; Comments should be placed near the code that they describe; Obvious functionality does not need to be commented; Insert comments before a block of code to describe what the block does
- Include a commented, informational header at the top of every HDL source file
  - Description of function and list of key features of the module
  - Date the file was created
  - Modification history including date, name of modifier, and description of the change

Four-valued system

- Verilog supports four logic values: 0: Logic-0 or FALSE; 1: Logic-1 or TRUE; x: Undefined, unknown, or don't care; and z: High impedance, floating, unconnected (no current flowing)
  - x and z are not case sensitive
- z is useful for describing a tristate buffer, whose output floats when the output is disabled
- We can set bits to be x in situations where we don't care what the value is. This can improve synthesis quality
- Verilog uses x to indicate a 0, 1, z, in *transition, don't care,* an *invalid logic level*
  - When a signal is driven to different values (0, 1, or x) by multiple drivers simultaneously, it is in *contention* and is denoted by x
  - The x may be interpreted by the circuit as 0 or 1, leading to unpredictable behavior
  - You should understand the origin of any x in your simulation results and resolve them if necessary

Signal data type - `reg`

- Verilog supports two data types: *"net data types"* and *"variable data types"*
- Nets represent physical wires and are used to connect modules and primitive gates. The most commonly used net type is **`wire`**. Thus a net will map to a wire during synthesis
- A **`reg`** type declaration starts with the keyword **`reg`**
- The default value of register variables is 'x'
- The **`reg`** data type is used as a general purpose variable for modeling hardware behavior in procedural blocks (e.g., **`always`** and **`initial`**)
  - The type **`reg`** simply means a variable that can hold a value (an abstract storage element) until a new value is assigned to it
  - **`reg`** type can store 4-state logic values 0, 1, Z and X
  - Note that a register data type maps either to a wire or to a storage cell depending on the context under which a value is assigned
- **`reg`** type in Verilog should not be confused with hardware registers
  - The name is used in Verilog to indicate a software register (i.e., a variable). Thus there is no correlation between using a **`reg`** variable and the hardware that will be inferred
  - It is the context in which the **`reg`** variable is used that determines if the hardware represented is combinational logic or sequential logic

Signal data type - `wire`

- A net declaration starts with one of the net data type keywords (e.g., **`wire x,y;`**)
- Initial values may be specified in declarations: **`wire wire_variable = value;`**
- A **`wire`** does not store its value but must be driven continuously by its driver
- The default value of net types is **`z`**, i.e., if a net is not driven by any driver, its value is **`z`**
- Local variable and net data types are *internal* signals of a module that cannot be accessed by other modules (except for testing)
  - Ports are *external* signals to interface with other modules and outside environment
- The value of a net is determined by the values of the net's *drivers*. Thus the value changes when the driver changes value. A *net driver* can be a gate's output, module's output, or continuous signal assignment statements (using the **`assign`** keyword)
- The two most commonly used synthesizable net types are **`wire`** and **`tri`**. The **`tri`** net type might be used where multiple drivers drive a net. Both the **`wire`** and **`tri`** nets are identical in their syntax and functions
- Logical conflicts from multiple sources on a **`wire`** or a **`tri`** net result in unknown values **`x`**

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

`wire` data type declaration

- A local net declaration defines an internal wire variable that is only used within the module. For an undefined scalar signal, by default a net data type is inferred
  - Declare all nets *explicitly* at the beginning of each module before being used, even when an implicit declaration could be used. This improves the readability and maintainability of the Verilog code
- *Compiler directives* are instructions to the Verilog compiler. Compiler directives start with a back tic ` and executed prior to simulation time zero and synthesis. The directive must be stated outside of module definitions
- **`` `default_nettype none ``** compiler directive disables default net declarations
  - In this case, any undeclared signals will be a syntax error, which prevents hard-to-debug wiring errors

Variable data type

- *Variable data types* include **`reg, integer, real, realtime, time`**
  - **`reg`** is a one bit unsigned value
  - **`integer`** is a 32-bit signed integer; **`real`** is a 64-bit floating-point number
  - **`realtime`** is of type real used for storing time as a floating-point value
  - **`time`** is 64-bit unsigned data type used for simulation time
- Synthesizable variable data types are **`reg`** and **`integer`**
  - A **`reg`** can be used to model actual hardware registers
  - '**`integer`**' is used in testbenches for integer variables, e.g. **`for`** loop counters
- All nets and variables can have any of 4 logic values for each bit: 0, 1, Z and X
- For an uninitialized variable, an unknown value X are set to all bits at time zero
- A variable changes its value upon assignment and holds its value until another assignment. This is similar to traditional programming language variables and is used in procedural statements
- Note that Verilog only supports built-in data types. User-defined types are not allowed in Verilog

Vectors (packed arrays)

Arrays (unpacked arrays)

- The dimensions declared after the signal name are referred to as the "array"

Vectors (packed arrays)

- Signals can be scalar (one bit) or vector (multiple bits)
- Vectors are declared by specifying a range of bits in square brackets, which indicates the vector width, followed by the vector name
  - `reg|wire [msb_index : lsb_index] List_of_Variables;`
  - `msb_index` indicates the *most significant bit* (MSB) index and `lsb_index` indicates the *least significant bit* (LSB) index
  - There are no requirements on the absolute values or the relationship of the MSB or LSB of the range specifier. Both must be integers and one is not required to be larger than the other. They can be equal in which case only one instance will be generated. The `msb_index` and `lsb_index` can be any numbers (negative numbers too)

```
wire [15:0] x; //little endian convention; 16-bit signal; x[15] MSB
reg [0:1023] x; //big endian convention; x[0] is the MSB
```

- We normally use the little endian convention, i.e., `msb_index>lsb_index` when describing vectors `[msb_index : lsb_index]`
  - Using a consistent ordering helps improving the readability of the code and reduces the chance of accidently swapping order between connected wires

Arrays (unpacked arrays)

- The dimensions declared after the signal name are referred to as the "array" dimensions. This dimension is typically explained in big endian convention
- Array declaration syntax: `<data_type> <vector_width> <array_name> <array_size>`
  - `reg [15:0] RAM [0:4095]; // memory array`
  - `reg [7:0] LUT [0:255]; // a one-dimensional array of 256 8-bit variables`
    `reg buffer [0:15]; //Array with addresses from 0 to 15`
  - `wire wire_x [0:7]; // a one-dimensional array of 8 1-bit nets`
  - `real r [0:15]; // a one-dimensional array of 16 real variables; not synthesizable`
- Any variable or net type can be declared as an array, and also multi-dimensional arrays can be declared
- *Bit select* is the selection of an individual bit as `variable_name[index]`
- *Part select* is the selection of a group of bits as `variable_name[msb:lsb]`
  - Part selects must address a more significant bit on the left of the colon
- Bit-selects and part-selects can be used as operands in expressions
- Verilog-2001 added *variable part selects* by allowing to use variables to select a group of bits from a vector, which is synthesizable (i.e., the value of the variable must be available during synthesis

Ports definition

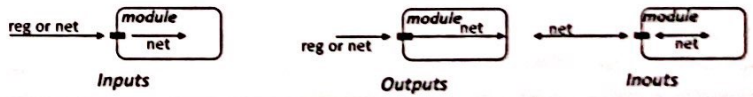
- Ports have a direction, type, and size. The port direction is followed by an optional type declaration, and then an optional size declaration

```
port_direction [port_type] [port_size] port_name1, port_name2,... ;
```

- The port direction must be explicitly declared as an `input`, `output`, or `inout`
  - An `input` port is driven by another module or gate. Input ports are connected to gate inputs, instantiated module inputs, and the right-hand side of continuous and procedural assignments
  - An `output` port is driven by the module and is available to other module
  - An `inout` port is a bidirectional port and can be driven either by the module or an external module
  - Therefore, `input` ports can be read but cannot be written, `output` ports can be written but cannot be read, and `inout` ports can be read and written
- The port type defines the type of the port, such as `wire` or `reg`. If the optional port type is not specified, the default type for port signals is `wire`
- `reg` type cannot be used for `input` or `inout` ports
- Ports can be scalars or vectors. If the optional port size is not specified, the port defaults to the default width of the data type. The default width of type `wire` and `reg` is one bit

Port restrictions

- Following the optional width declaration is a comma-separated list of one or more port names
- Each port in the list will be of the direction, type, and size specified
- The port type cannot be changed for a subsequent port without re-specifying the port direction, even if it is the same direction as the preceding port. The vector size of the port cannot be changed for a subsequent port without re-specifying the port direction and optional type
- `inputs`: internally must always be of type net, externally the inputs can be connected to a `reg` or a net
- `outputs`: internally can be of type net or `reg`, externally the outputs must be connected to a net
- `inouts`: internally or externally must always be type net, can only be connected to a net



(1) Gate-level modeling

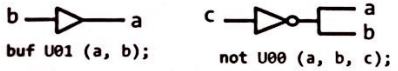
- Verilog supports a set of primitive logic gates, **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf**, that allows a module to be described at the gate level
- A primitive gate may be explicitly instantiated as follows:

```
gate_type [delay] instance_name (<output>, <inputs>)
```

- Gates can have a single output, which always comes first in the port list, but any number of inputs. The input is always the last port in the port list
- The **buf** and **not** primitives can have multiple outputs, but only one input
- Port order for **and**,**or**,**nand**,**nor**,**xor**,**xnor** gates: (one_output, one-or-more_inputs)
- Port order for **not**, **buf** gates: (one-or-more_outputs, one_input)

```
nand U00(out, in1, in2);        //2-input NAND gate
```

- For all primitive gates the output port must be connected to the **wire** type and the input ports may be connected to nets or **reg** type variables



Primitives' truth tables

- Primitive gates respond to **0**, **1**, x, or z in a logical way. Truth tables of several logical gates for all four possible signal values of the inputs are shown
- The output of a gate with Z inputs is **0**,1, or x

1 & z = x
1 & x = x
z & x = x

not	
inputs	outputs
0	1
1	0
x	x
z	x

buf	
inputs	outputs
0	0
1	1
x	x
z	x

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

- An x output may correspond to a floating gate input (z) or uninitialized input (x) when the gate can't determine the correct output value
- 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**

Gate-level modeling examples

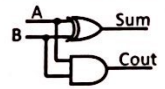
Truth table of a half-adder

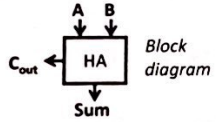
A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Boolean expression

Sum = A ⊕ B=AB'+A'B
Cout = A.B

Gate-level schematic

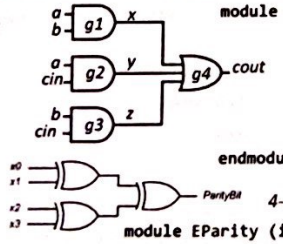




Block diagram

```
module halfAdder_gate (input A, B,
                       output Sum,Cout);
  xor U00 (Sum, A, B);   // instantiating primitive gates
  and U01 (Cout, A, B);  // all statements are executed concurrently
endmodule
```

- The half adder is described at the gate level by instantiating an XOR and an AND gate. U00 and U01 are *instance names* for the XOR and AND gates, respectively
- A module can be described at various levels of abstraction
- Gate-level modeling is not recommended as it creates constraints for the synthesis process. Modeling at the higher level of abstraction is recommended



4-bit even parity generator

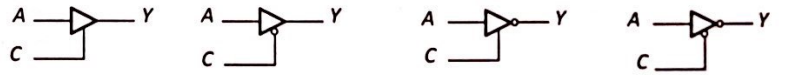
```
module carry(input  a,b,cin,
                    output cout);
  wire   x, y, z;
  and g1(x, a, b);
  and g2(y, a, cin);
  and g3(z, b, cin);
  or  g4(cout, x, y, z);
endmodule
```

```
module EParity (input [3:0] x,
                output parity);
  xor U00 (parity,x[0], x[1], x[2], x[3]);
endmodule
```

Tri-state buffers

- In addition to values of **1** and **0**, tri-state buffers can have a high-impedance output state where the output has no effect on connected circuits. This allows multiple outputs to be connected together (e.g. to a *tri-state bus*)
- Tri-state buffers have *one output*, *one data input*, and *one control input*. The control input is used to enable or disable the buffer

```
<gate> instance_name (<output>, <input>, <control>);
```



bufif1 U(Y,A,C); **bufif0** U(Y,A,C); **notif1** U(Y,A,C); **notif0** U(Y,A,C);

bufif1	CONTROL 0	1	x	z
DATA 0	z	0	L	L
DATA 1	z	1	H	H
DATA x	z	x	x	x
DATA z	z	x	x	x

bufif0	CONTROL 0	1	x	z
DATA 0	0	z	L	L
DATA 1	1	z	H	H
DATA x	x	z	x	x
DATA z	x	z	x	x

notif1	CONTROL 0	1	x	z
DATA 0	z	1	H	H
DATA 1	z	0	L	L
DATA x	z	x	x	x
DATA z	z	x	x	x

notif0	CONTROL 0	1	x	z
DATA 0	1	z	H	H
DATA 1	0	z	L	L
DATA x	x	z	x	x
DATA z	x	z	x	x

- If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value z
  - The symbol H represents a result which has a value of 1 or Z
  - The symbol L represents a result which has a value of 0 or Z

Tri-state bus and array of instances

Bitwise and reduction operators

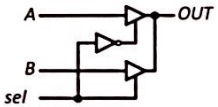
```
xor U00 (parity,x[0], x[1], x[2], x[3]);
endmodule
```

- The symbol L represents a result which has a value of 0 or Z

Tri-state bus and array of instances

- A tri-state bus can be driven by several drivers. One driver on a net should be enabled at a time and the net takes on that value. The remaining drivers should dive Z (high-impedance, float)

```
module muxtri (input A,B,sel,
               output tri OUT);
   bufif1 U00 (OUT,B,sel);
   bufif0 U01 (OUT,A,sel); //instead of using a buf and a not
endmodule
```



- If all the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by z. A *contention* occurs if a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates). This will show up as an X value in simulation, and could result in damage to the drivers in a physical device
- An array of primitive gates can be instantiated individually separated by a comma. This is rather tedious because each XOR instance has to be individually instantiated with the appropriate bit
- An array of instances can be defined using an array name to provide the numbering of the instance names. Given msb and lsb, 1 + abs(msb-lsb) instances will be generated

```
module xor8 (output [1:8] xout,
             input [1:8] xin1, xin2);
  xor a8 (xout[8], xin1 [8], xin2[8]),
      a7 (xout[7], xin1 [7], xin2[7]),
      a6 (xout[6], xin1 [6], xin2[6]),
      a5 (xout[5], xin1 [5], xin2[5]),
      a4 (xout[4], xin1 [4], xin2[4]),
      a3 (xout[3], xin1 [3], xin2[3]),
      a2 (xout[2], xin1 [2], xin2[2]),
      a1 (xout[1], xin1 [1], xin2[1]);
endmodule
module xor8 (output [1:8] xout,
             input [1:8] xin1, xin2);
  xor a[1:8] (xout, xin1, xin2);
endmodule
```

Bitwise and reduction operators

- Verilog has built-in logical operations, which are defined for four-valued signals
- Logic gates are inferred for the corresponding logical operators
- *Bitwise operators*: act on single-bit signals or on multi-bit busses and do a bit-by-bit logical operation between two operands
  - and &, or |, not ~, xor ^, xnor ~^ (or ^~)

A	B	A&B	A|B	~A	A^B	A~^B
1010	0011	0010	1011	0101	1001	0110

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b	XNOR

- *Reduction operators*: operate on all the bits of an operand vector and return a single-bit value as output. These are the unary (one operand) form of the bit-wise operators above
  - and &, or |, xor ^, nand ~&, nor ~|, xnor ~^ (or ^~)

A	&A	~&A	|A	~|A	^A	~^A
1010	0	1	1	0	0	1

Reduction

&	AND
~&	NAND
|	OR
~|	NOR
^	XOR

- Unary reduction operators imply a multiple-input gate acting on a vector

```
module ngate (input [2:0] x, output y);
  assign y = ~|x;
endmodule
```

Logical operators

- Verilog supports three *logical operators*: and &&, or ||, not !
- They return a one-bit result as 1 (true), 0 (false), or x
- Note the distinction between the *unary reduction operators* and the *bitwise logic operators,* which look the same. The meaning depends on the context
- ~a is a bitwise negation and !a is logical not (used in logical expressions)
- Logical operators are typically used in conditional (if ... else) statements
  - e.g., `if ((x==y) && z) a=1; //if x equals y and z is non-zero`
- They can work on expressions, integers, or groups of bits
- The logical operators treat their operands as Boolean quantities
  - A non-zero operand is considered true (1'b1)
  - A zero operand is considered false (1'b0)
  - An ambiguous value (i.e. one that could be true or false, such as 4'bXX00) is unknown (1'bX)

Logical

!a	NOT
a && b	AND
a || b	OR

A	B	A&&B	A||B	!A	!B
1010	00	0	1	0	1
1010	011	1	1	0	0

(2) Dataflow level modeling

- Three general styles for describing a module functionality are *dataflow*, *behavioral* and *structural.* Structural: specifies how a module is composed of other modules or primitive gates. Behavioral: specifies the functionality of a hardware module at the highest level of abstraction. Dataflow: specifies manipulation of data values using primitive operations
- In dataflow modeling, the behavior of a module is described using *continuous signal assignment* statements
- A continuous signal assignment statement starts with the **assign** keyword

```
assign [delay] List_of_assignment_statements
```

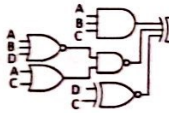
- where the List_of_assignment_statements are in the form

```
net = expression {, net = expression };
```

- An *expression* consists of a set of operands, one or more operators (logical, numerical, relational), literal values, and sub-expressions
- A continuous **assign** statement specifies a value to be driven onto a net
  - The expression on the RHS of an **assign** statement may contain both "register" or "net" type variables and the LHS must be a net-type signal (a port or an internal net)
  - If undeclared, the left-hand side is implicitly declared as a scalar (one bit) net

Combinational logic using dataflow level modeling

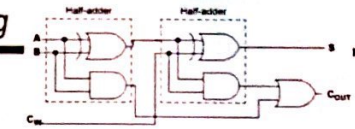
- The continuous signal assignment is always active (driving a **0**, **1**, **x**, or **z**)
  - Any time any of the operands on the RHS expression of an **assign** statement change, the assign statement will be reevaluated and the result will be assigned to the LHS net
  - Thus the RHS expression is continuously evaluated as a function of changing inputs. This is the characteristic of combinational logic
- Continuous signal assignments can be used to describe combinational logic that can relatively easily be described using expressions
- A Verilog module can contain any number of continuous assignment statements
- All continuous signal assignment statements execute *concurrently*. Thus concurrent signal assignment statements can be written in any order. This is different from conventional programming languages in which statements are evaluated in the order they are written
- **assign y<=a+b;** has a syntax error
- **assign out=a<=b;** is a correct statement




```
module gates_DL (input A,B,C,D,
                 output F);
 assign F = (A & B & C) ^ ( ~(~(A|B|D) & (A|C)) ) ^ (D ^~ C);
endmodule
```

```
module gates_GL (input A,B,C,D,
                 output F);
wire w1, w2, w3, w4, w5;
  nand U00 (w1,A,B,C);
  nor  U01 (w2,A,B,D);
  or   U02 (w3,A,C);
  nand U03 (w4,w2,w3);
  xnor U04 (w5,D,C);
  xor  U05 (F,w1,w4,w5);
endmodule
```


Dataflow level modeling



$S = (A \oplus B) \oplus C_{IN}$

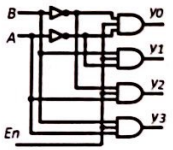
$C_{OUT} = AB + C_{IN}(A \oplus B)$

Inputs		Carry-in	Sum	Carry-out
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
module halfAdder_dataflow (input a, b,
                           output sum,cout);
  assign sum  = A ^ B;  //^ denotes logical XOR
  assign cout = A & B;  //& denotes logical AND
endmodule
```

```
module fullAdder(input a, b, cin,
                 output sum, cout);
wire p, g;
      assign p = a ^ b;
      assign g = a & b;
      assign sum = p ^ cin;
      assign cout = g | (p & cin); //do we need parenthesis here? No, and has higher precedence
endmodule
```

```
module fullAdder(input a,b,cin,
                 output sum,cout);
   assign cout = a & b | cin & (a ^ b);
   assign sum = cin ^ a ^ b;
endmodule
```




```
module decoder2to4 (input A,B,En,
                    output [3:0] y);
wire Anot,Bnot;
   not // multiple gates of the
       // same type are separated by ,
      U0 (Anot,A),
      U1 (Bnot,B);
   and
      U2 (y[0],Anot,Bnot,En),
      U3 (y[1],Anot,B,En),
      U4 (y[2],A,Bnot,En),
      U5 (y[3],A,B,En);
endmodule
```

Array of instances separated by commas

```
module decoder2to4 (input A, B, En,
                    output [3:0] y);
  assign y[0] = ~A & ~B & En,
         y[1] = ~A & B & En,
         y[2] = A & ~B & En,
         y[3] = A & B & En;
endmodule
```

```
module decoder2to4 (input in,
                    input [1:0] sel,
                    output [3:0] out);
  assign out[sel] = in;
endmodule
```

Violates the definition of combinational logic. All the bits of "out" are not assigned

Examples: N-input gates examples

```
module or8(input [3:0] a,
           output y);
assign y = |a;  // using the reduction operator
// |a is much easier to write than
// assign y = a[3]|a[2]|a[1]|a[0];
endmodule
```

```
// Three-input gates
module gates   (input wire [2:0] x ,
                output wire and3,
                output wire nor3 ,
                output wire xor3);
    assign and3 = &x,
           nor3 = ~|x, // reduction NOR
           xor3 = ^x;
endmodule
```

```
module gates2 (input a, b,
               output and_,nand_,nor_, or_, xnor_, xor_) ;
    assign and_ = b & a,
           nand_ = ~(b & a),
           or_ = b | a,
           nor_ = ~(b | a),
           xor_ = b ^ a,
           xnor_ = ~(b ^ a);
endmodule
```

```
assign z = x[1] & x[2] & ... & x[n]; //n-input and gate
assign z = &x; //it is much easier to describe an n-input and gate
and A00 (z,x[1],x[2],...,x[n]); //n-input and gate

assign z = ~(x[1] & x[2] & ... & x[n]); //n-input nand gate
assign z = ~&x; //n-input nand gate
nand N00 (z,x[1],x[2],...,x[n]); //n-input nand gate

assign z = ~(x[1] ^ x[2] ^ ... ^ x[n]); //n-input xnor gate
assign z = ~^x; //n-input xnor gate
xnor X00 (z,x[1],x[2],...,x[n]); //n-input xnor gate
```

In-line initialization

- Value initialization can be specified as part of variable declaration
- Initialization of variables using in-line initialization is not synthesizable
- Initialization occur in a nondeterministic order at simulation time zero
- In this example, the nondeterministic event ordering specified in the Verilog standard does not guarantee that **i** would be initialized first, and so **j** would be initialized to 3
  - –j might be assigned the value of i before i has been initialized to 3, which means j would receive a value of X
- The only deterministic way to model the initialization is to an **initial** statement

```
//version I
integer i=3;
integer j=i;

//Version II
integer i,j;
initial i = 3; //initialize i to 5 at time 0
initial j = i;
  //initialize j to the value of i at time 0

//Version III
integer i,j;
initial begin
  i = 3;
  j = i;
end
```

Delay in assign statements

Functional simulation using testbenches

- The idea of simulating a Verilog module is similar to an engineer's workbench

```
assign z = ~^x; //n-input xnor gate
xnor X00 (z,x[1],x[2],...,x[n]); //n-input xnor gate
```

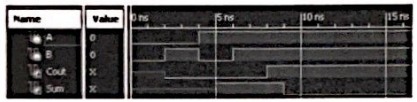
Delay in assign statements

- Delay can be specified in the signal declaration, gate instantiation, signal assignment statement, or in the procedural statements
- When a delay is given in a net declaration, the delay is added to any driver that drives the net

```
wire [7:0] #10 AXorB;
assign #5 AXorB = a ^ b;
```

  - If **a** or **b** changes, **AXorB** receives the result after 15 time units
- A timing (delay) control before a statement causes the execution of the immediately following statement to be delayed `#Δt Procedural_statement;`

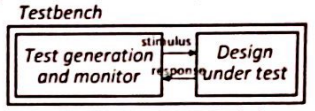
```
module HalfAdder (input A, B,
                  output Sum, Cout);
  assign #2 Cout = A & B; //& denotes logical AND
   //after 2 time units assign A&B to Cout
   assign #3 Sum = A ^ B; //^ denotes logical XOR
  //after 5 time units assign A^B to Sum
endmodule
```

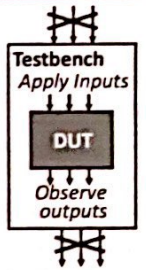


- The synthesis tool ignores the delays specified in a module. Do not use any delay values in your synthesizable code. The delay values will be incorrect for physical implementations and may cause simulation behavior to differ from synthesis results
- A delay control #Δt; is mainly used in testbenches to delay or schedule execution of statements. When no delay is specified, the default delay is zero

Functional simulation using testbenches

- The idea of simulating a Verilog module is similar to an engineer's workbench where the designed system is wired to a test generator that provides inputs to the *design under test* (DUT) and monitors the outputs
- A *testbench* is a Verilog module that generates and passes test vectors (or *stimuli*) to the DUT and can check its responses
- The test vectors are defined as type **reg** and the outputs are defined as type **wire**
- A testbench has no input or output ports
- Simulator can apply a sequence of testvectors to the DUT at some specified times. The expected output values for the test stimuli should be known
- Testbenches are not synthesizable so it can also use all non-synthesizable data types, operations, etc.
- The key limitation of functional verification through simulation is that a comprehensive set of stimuli to validate the entire design may be impractical, forcing the designer to rely on some method of random stimulus generation for full design coverage





`initial` *statement*

- One or more **initial** statements can be used to apply stimuli to a DUT
- An **initial** statement contains one or several procedural statements
- **initial** statement is used to initialize variables and uses procedural statements and delays to apply the test vectors in the appropriate order

```
initial [begin]
... Procedural statements ...
[end]
```

- **initial** statements are not synthesizable and should only be used in testbenches for applying stimuli to the DUT, not in modules intended to be synthesized into actual hardware
- An initial statement is executed only once at the beginning of the simulation (at time zero). The statements are executed in order until it finds a specified delay, it is then suspended and scheduled to wake up at the end of the specified time delay. When it is completed, it does not repeat; it becomes inactive

```
module tb_decoder2to4;
wire  [3:0] out; //outputs are defined as wires
reg   in1, in2, in3; // stimuli are defined as regs

    // Instantiate the decoder (DUT)
    decoder2to4 UUT (.A(in1), .B(in2), .En(in3), .y(out));

    initial  begin
       in1 = 1'b0;  in2 = 1'b0; in3 = 1'b0; // time = 0
      #10; in3 = 1'b1;                      // time = 10
      #10; in1 = 1'b0;  in2 = 1'b1;         // time = 20
      #10; in1 = 1'b1;  in2 = 1'b0;         // time = 30
      #10; in1 = 1'b1;  in2 = 1'b1;         // time = 40
      #5;  in3 = 1'b0;                      // time = 45
      #5 $finish; ////specifies the end of simulation
    end
endmodule
```

`timescale *compiler directive*

- What is the period of CLK? `assign #5 CLK = ~CLK;`

  - Is it 10 picoseconds? 10 nanoseconds?
- Simulation times are described in terms of *time units*. Instead of specifying the units of time with the time value, Verilog specifies time units as a command to the simulation tool, using a `` `timescale `` compiler directive
- `` `timescale `` is used to define time units of any delay operator (#) and the precision to which time calculations will be rounded
- This directive has two components: the time units and the time precision

```
`timescale TimeUnit / PrecisionUnit
```

  where `TimeUnit` and `PrecsionUnit` are in **TU** format: `T = {1, 10, 100}` and `u = {s, ms, us, ns, ps, fs}`
- Precision is the maximum number of decimal places used in time values
  - The precision unit defines how delay values are to be rounded off during simulation. All delays are rounded to the nearest precision unit
  - Precision unit must be less than or equal to the time unit
- Only 1, 10 or 100 are valid integers for specifying time units
- The default is `` `timescale 1ns/100ps ``, which delays will be rounded to the nearest one hundred picoseconds (0.1ns). e.g., 10.512ns is interpreted as 10.5ns

`timescale and shift operators

- `timescale 1ns/10ps instructs the synthesis tool to use time units of 1 nanosecond, and a precision of 10 picoseconds, which is 2 decimal places, relative to 1 nanosecond
- `timescale 10ps/1ps

```
nor #3.57 (z, x1, x2); // nor delay = 3.57 x 10 ps = 35.7 ps => 36 ps
```

- Verilog's time units and time precision have no meaning in synthesis, but are useful in simulation. The `timescale directive can have a significant impact on the simulation time and simulation memory usage
  - Synthesis ignores these keywords, which allows them to be used in codes that will be both simulated and synthesized
- *Logical shift* operators var >> len and var << len shift var by len bit positions to the right and left, respectively. Result is the same size as var, always zero filled from the left or right. There is no sign extension

```
a = 4'b1010;  d = a >> 2; // d = 0010  c = a << 1; // c = 0100
```

- *Arithmetic shift* left operator <<< behaves the same as <<. An arithmetic right-shift operation >>>, however, maintains the sign of a value, by filling with the sign-bit value as it shifts to the right
  - For example, if d=8'b10100111 is an 8-bit signed variable, then d >> 3 is the logical shift yields 8'b00010100 and d >>> 3 is the arithmetic shift and yields 8'b11110100

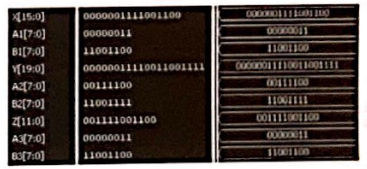
Concatenation operator

- Often, it is necessary to operate on a subset of a bus or to concatenate, i.e., join together, signals to form busses. The *concatenation operator* {op1, op2, ..} combines two or more operands to form a larger vector
  - The operands must be sized

```
reg a;
reg [2:0] b, c;
a = 1'b1;  b = 3'b010;
x = {a, b};    // x = 1010
y = {b, 2'b11, a};  // y = 010_11_1 //underscore is just for readability
z = {b, 1};// incorrect
```

```
reg [15:0] a,b;
{b[7:0],b[15:8]}= {a[15:8],a[7:0]}; // byte swap
wire [7:0] A,B;  wire [3:0] C;  wire [11:0] D;
assign {A,B} = {C,D};
assign {A,B} = X; // split X into A and B. The width of A and B are defined
```

```
wire [7:0] A1,B1,A2,B2,A3,B3;
wire [15:0] X = 16'b0000001111001100;
assign {A1,B1} = X;//split X into A and B
wire [19:0] Y = 20'b00000011110011001111;
assign {A2,B2} = Y;
wire [11:0] Z= 12'b001111001100;
assign {A3,B3} = Z;
```



Replication operator

- The *replication operator* {N{data}} makes N copies of data, where N is a constant value
- This example shows how to *sign-extend* a 16-bit number to 32 bits by using the replication operator
- We can use concatenation operator on the left-hand side of a statement to split an output into sub-parts

```
module mul (input [7:0] a, b,
            output [7:0] upper, lower);
   assign {upper, lower} = a*b;
endmodule
```

```
a = 1'b1; b = 3'b101;
r = {3{a}, b}; // r = 111_101
//{4{4'b1001,1'b0}} // 10010100101001010010
assign x = {2{1'b0}, a}; // 001

module signextend (input [15:0] a,
                   output [31:0] y);
  assign y = {{16{a[15]}}, a[15:0]};
endmodule

module uadd4bit(input [3:0] A,B,
                input Cin,
                output [3:0] SUM,
                output Cout);
 assign {Cout,SUM} = A + B + Cin;
 // the 4 least significant bits of
 // A+B+Cin will be stored in SUM
 // the MSB will be stored in Cout
endmodule
```

Relational and equality operators

- Relational operators compare two operands and indicate whether the comparison is true or false. They evaluate to a Boolean *false*, which is equivalent to one bit 1'b0 and Boolean *true*, which is equivalent to 1'b1
- The relational operators include > (greater than), >= (greater than or equal), == (equal), and != (not equal).
- They are typically used in conditional expressions and are synthesize to comparators. Since reg and wire types are unsigned, the synthesized comparators will be unsigned
- A comparison operator may return 0, 1 or 1'bx (if the comparison is ambiguous)

```
2'b10 > 2'b0X // is true (1'b1)
2'b11 > 2'b1X // is unknown (1'bX)
```

```
module mulTest (output [63:0] y,
                input  [31:0] a, b);
  wire eq;
  assign y = a * b;     //defaults to wire, width of port y
  assign eq = (a == b); // defaults to 1-bit wire in Verilog-2001
                        // ERROR in Verilog 1995: 'eq' is not declared
                        // Synthesis tool will not generate a comparator
endmodule
```

Equality operators with don't care wildcards

- The *logical equality operator* == and the *case equality operator* === (also called

Conditional operator

- The conditional operator (?:) can be used when one of two values is to be selected for assignment as:

Equality operators with don't care wildcards

- The *logical equality operator* `==` and the *case equality operator* `===` (also called the *identity operator*) handle logic X and logic Z values in the operands differently
  - The `==` logical equality operator will consider any comparison where there are bits with X or Z values are in either operand to be unknown, and return a `1'bx`
  - The `===` case equality operator performs a bit-wise comparison of the two operands
- Each of these operators has a not-equal counterpart, `!=` and `!==`
  - `==` and `!=` will return an `X` if any bit of an operand is `X` or `Z`
  - `===` and `!==` operators consider `X` and `Z` values in comparison

Operand A	Operand B	===	!==	==	!=
0110	0110	1	0	1	0
0110	0XX0	0	1	X	X
0XZ0	0XZ0	1	0	X	X

- If the operands are not the same size, then the logical equality operators will expand the vectors to the same size before performing the comparison
- The case equality operators are not generally synthesizable

Comparison

a < b a > b a <= b a >= b	Relational
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

Conditional operator

- The conditional operator (?:) can be used when one of two values is to be selected for assignment as:

```
signal = conditional_expression ? true_expression : false_expression;
```

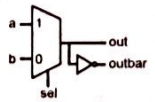
- If the `conditional_expression` is TRUE (or nonzero), then the operator chooses the value of the `true_expression` to be assigned to `signal`. Otherwise the value of `false_expression` will be assigned to `signal`
- *?:* is also called a *ternary operator* because it takes three inputs
- The conditional operators are synthesized to multiplexers or tri-states

```
module mux(input [0:3] a, b,
           input sel,
           output [0:3] f);
  assign f = sel ? a : b;
endmodule
```

```
//dataflow-level description
module mux(output f,
           input a,b,sel)
  assign f=(a&~sel) | (b&sel);
endmodule
```

```
//gate-level description
module mux(output f,
           input a,b,sel);
wire nsel,d0,d1;
  not g1 (nsel, sel);
  and g2 (d0, nsel, a);
  and g3 (d1, sel, b);
  or  g4 (f, do,d1);
endmodule
```

```
module mux2 (input a, b, sel,
             output out, outbar);
wire w;
   assign w = sel ? a : b;
   assign out = w;
   assign outbar = ~w; //~ denotes not
endmodule
```



- Could have we defined out as `inout` instead of using `wire w`?

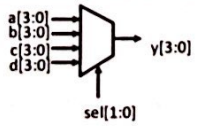
Conditional operator example

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

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

  - If `s[1] = 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] = 1 and d2 if `s[0] = 0`).
  - If `s[1] = 0`, then the multiplexer similarly chooses the second expression, which gives either d1 or d0 based on `s[0]`

```
module mux_4bits(input  [3:0] a, b, c, d,
                 input  [1:0] sel,
                 output [3:0] y);
   assign y =
     (sel == 0) ? a :
     (sel == 1) ? b :
     (sel == 2) ? c :
     (sel == 3) ? d : 4'bx;
endmodule

module Comparator (input [7:0] A, B,
     output CMP);
assign CMP = (A >= B) ? 1'b1 : 1'b0;
endmodule
```



Using a variable index in the RHS expression to generate a MUX

```
module mux (input [0:3] data,
            input [0:1] sel,
            output out);
  assign out = data[sel];
endmodule
```

Arithmetic operators

- Arithmetic operators include addition +, subtraction -, multiplication * (synthesizable), division / and modulo % (not synthesizable)
- The power operator ** will return a real number if either operand is a real value, and an integer value if both operands are integer values

```
// 8-bit adder
module adder ( input  [7:0] op1, op2,
               output [7:0] sum );
   assign sum = op1 + op2;
endmodule
```

```
// unsigned 8-bit multiplier
module umul_8bit (input  [7:0] A,
                  input  [3:0] B;
                  output [11:0] RES);
   assign RES = A * B;
endmodule
```

```
module uadder_8bit(input  [7:0] A, B,
                   output [7:0] SUM,
                   output CO);
wire [8:0] tmp; // internal wires are local variables
   assign tmp = A + B;
   assign SUM = tmp [7:0];
   assign CO  = tmp[8];
endmodule
```

- All the assign statements execute concurrently
  - This is unlike conventional programming languages, in which all statements execute sequentially

- The presence of a '`z`' or '`x`' in a `reg` or `wire` being used in an arithmetic expression will result in an unknown `x`

(3) Structural modeling

- To describe the functionality of a digital system, the design can be partitioned into modules, which can then be further divided until the building blocks are simple enough to be described. This *hierarchical* description allows a designer to control the complexity of a complex design through the *divide-and-conquer* approach
- In a *top-down description*, each module instances can be described *structurally* based on its building blocks (other modules and primitive gates) recursively until the modules are simple enough to be described *behaviorally*
- In a *bottom-up description*, once a module has been designed (mostly at the behavioural level), it can be used by (instantiated in) other modules, which creates a module hierarchy
- A module is *defined* by specifying the functionality of that module. Each unique copy of a module or a primitive gate is called an *instance*. Multiple instances of the same module are distinguished by distinct *instance names*. Instance names can be used for debugging. A module instance may then be *instantiated* in other modules as many times as required. Design hierarchy can be created by instantiating a module within another module
- At the structural level, a design is described by specifying the interconnection of other modules and logical gates it is comprised of. These interconnections are typically modeled using net type **wire**

Named port connection

- For instantiating a module, the name of each port is specified explicitly along with the name of the net that is connected to that port as
  .<port_name>(<net_or_variable_name>)

```
<module_name> <instance_name> (.port1    (wire_connected_to_port1),
                               .port2    (wire_connected_to_port2),
                                  . . .
                               .portn    (wire_connected_to_portn) );
```

- Since both port names and connection names are specified, the port connections may be listed in any order
- When possible, use the same names for ports and signals that are connected
  - Consistent naming makes code easier to read, especially by others, and helps in debugging and maintenance

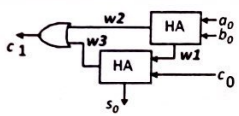
```
module fullAdder (input a0, b0, c0,
                  output s0, c1);
wire [1:3] w; //wire [msb:lsb] wire1, wire2,...
  halfAdder HA0(.in1(a0), .in2(b0), .S(w[1]), .C(w[2]));
  halfAdder HA1(.in1(w[1]), .in2(c0), .S(s0), .C(w[3]));
  or u0(c1, w[2], w[3]);
endmodule
```

```
module halfAdder(input in1, in2,
                 output S,C);
  xor U00 (S, in1, in2);
  and U01 (C, in1, in2);
endmodule
```

- An unconnected port can be left either unspecified, or explicitly named with an empty parentheses set to show that there is no connection

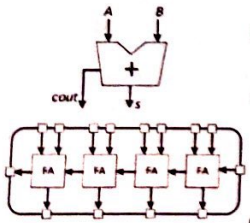
Full-adder description at the structural level

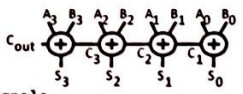
```
module halfAdder(input in1, in2,
                 output S,C);
  xor U00 (S, in1, in2);
  and U01 (C, in1, in2);
endmodule
```




```
module fullAdder (input a0, b0, c0,
                  output S0, c1);
wire [1:3] w; //wire [msb:lsb] wire1, wire2,...
  halfAdder HA0 (.in1(a0),.in2(b0),.S(w[1]),.C(w[2])); //instantiation of HAs
  halfAdder HA1 (.in1(w[1]),.in2(c0),.S(S0),.C(w[3])); //HA is defined in another module
  or u0 (c1, w[2], w[3]);
endmodule
```

- A four-bit adder can be built by instantiating four full-adders. In this example, 4bitAdder is the *top-level module* and FA0–FA3 are four instances of full-adders

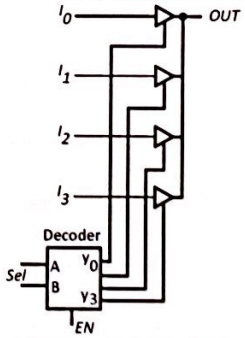





```
module 4bitAdder (input [3:0] A,B,
                  output [3:0] S,
                  output Cout);
wire [3:1] C;  //Intermediate carry signals
   // instantiate an array of instances
 fullAdder FA0(.a0(A[0]),.b0(B[0]),.S0(S[0]),.C(C[1])),
           FA1(.a0(A[1]),.b0(B[1]),.c0(C[1]),.S0(S[1]),.C(C[2])),
           FA2(.a0(A[2]),.b0(B[2]),.c0(C[2]),.S0(S[2]),.C(C[3])),
           FA3(.a0(A[3]),.b0(B[3]),.c0(C[3]),.S0(S[3]),.C(Cout));
endmodule
```

Module instantiation example

- Module definitions cannot be placed inside another module
  - They must be defined externally and then instantiated in other modules



4:1 mux with tri-state output

```
module decoder2to4 (input A, B, En,
                    output [0:3] y);
   assign y[0] = ~A & ~B & En,
          y[1] = ~A & B & En,
          y[2] = A & ~B & En,
          y[3] = A & B & En;
endmodule
```

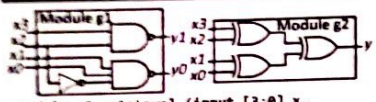
```
module FourtoOneMux (input [0:3] I,
                     input [1:0] Sel,
                     input EN,
                     output tri OUT);
   wire [0:3] d;

decoder2to4 DEC (.A(Sel[1]),.B(Sel[0]),.En(EN),.y(d));

bufif1 U00 (OUT,I[0],d[0]),
       U01 (OUT,I[1],d[1]),
       U02 (OUT,I[2],d[2]),
       U03 (OUT,I[3],d[3]);
// the same as bufif1 U[3:0] (OUT,I,d);
endmodule
```

- Verilog modules can be described at the structural level, behavioral level, dataflow level, or any mix of these descriptions

Example

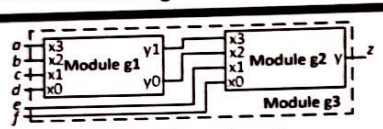
endmodule

Gate, dataflow, and structural level modeling




```
module g1_gateLevel (input [3:0] x,
                     output [1:0] y);

wire w;
  not  N00 (w, x[2]);
  nand N01 (y[1], x[3], x[2]);
  nand N02 (y[0], x[1], x[0], w);
endmodule
module g1_dataflow (input [3:0] x,
                    output [1:0] y);

wire w;
  assign w = ~x[2];
  assign y[1] = ~(x[3] & x[2]);
  assign y[0] = ~(x[1] & x[0] & w);
endmodule
module g2_gateLevel (input [3:0] x,
                     output y);

wire [1:0] w;
  xor N00 (w[1], x[3], x[2]);
  xor N01 (w[0], x[1], x[0]);
  xor N02 (y, w[1], w[0]);
endmodule
```

```
module g2_dataflow (input [3:0] x,
                    output y);

wire [1:0] w;
  assign w[1] = x[3] ^ x[2];
  assign w[0] = x[1] ^ x[0];
  assign y = w[1] ^ w[0];
endmodule
```




```
module g3 (input a, b, c, d, e, f,
           output z);
wire g1_y0, g1_y1;
  g1_dataflow MODULE_G1(.x({a,b,c,d}),.y({g1_y1,g1_y0}));
  g2_dataflow MODULE_G2(.x({g1_y1,g1_y0,e,f}),.y(z));
endmodule
```

```
`timescale 1ns / 1ps
module tb_g3;
reg [5:0] d;
wire z;
  g3 DUT (.a(d[5]), .b(d[4]), .c(d[3]), .d(d[2]),
          .e(d[1]), .f(d[0]), .z(z));
initial begin
 d=6'b000000;
 #10; d=6'b111111;
 #10; d=6'b010101;
 #10; d=6'b101010;
 #10;
 $finish;
end
endmodule
```


Numbers

- Constant values may be specified in either the *sized* or the *un-sized* form
- Syntax for sized constant values: **<size>'<base><value>**
  - **size** *(optional)* is the *number of bits* to represent the number
  - **'base** *(optional)* represents the radix. The default base is decimal
  - **<value>** defines the value
- Numbers can be specified in different bases
- Note that the base letters and hexadecimal digits are not case sensitive

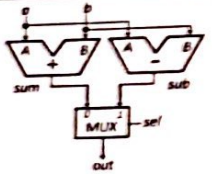
Base	*Symbol*	*Legal Values*
binary	b or B	0,1,x,X,z,Z,?,_
octal	o or O	0-7,x,X,z,Z,?,_
decimal	d or D	0-9,_
hexadecimal	h or H	0-9,a-f,A-F,x,X,z,Z,?,_

- Logic **x** in a number can be represented by the characters **x** or **X**, and logic **z** in a number can be represented by the characters **z**, **Z** or **?**. Thus the characters **z** and **?** are equivalent in numbers
- When specifying a constant value, if the size specified is less than the number of bits, the bits are truncated from the left (most significant bits). If the size is greater, the number is padded on the left with zeros
- If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. It is better to explicitly specify the size

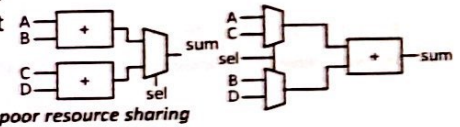
Example

```
module Adder (input signed [3:0] a, b,
              output reg signed [4:0] sum);
  always @ (*) sum = a + b;
endmodule
module Sub (input signed [3:0] a, b,
            output reg signed [3:0] diff);
  always @ (*)  diff = a - b;
endmodule
module Mux (input signed [4:0] in1,
            input signed [3:0] in2,
            input sel,
            output reg signed [4:0] out);
  always @ (*) out = sel ? in2 : in1;
endmodule
module Top (input signed [3:0] a, b,
            input sel,
            output signed [4:0] out);
  wire signed [4:0] sum_out;
  wire signed [3:0] diff_out;
  Adder ADD00 (.a(a), .b(b), .sum(sum_out));
  Sub SUB00 (.a(a), .b(b), .diff(diff_out));
  Mux M00 (.in1(sum_out), .in2(diff_out), .sel(sel), .out(out));
endmodule
```

```
module tb_Top;
  //Inputs as reg
  reg signed [3:0] a, b;
  reg sel;

  //Outputs as wire
  wire signed [4:0] out;

  //DUT Instance
  Top DUT(.a(a), .b(b), .sel(sel), .out(out));
  //Apply stimuli
  initial begin
    a = 4'b0001; b = 4'b1100; sel = 1'b0;
    #10; sel = 1'b1;
    #10; a = 4'b0010; b = 4'b0110; sel = 1'b0;
    #10; sel = 1'b1;
    #10;
    $stop;
  end
endmodule
```



- *Resource sharing* can reduce the number of logic operations needed to implement a module. Numerical operators, such as adders and multipliers, are resource intensive compared to multiplexers



Sized and unsized numbers

- To specify a literal value with all bits set to one, a fixed size must be specified
  - For example: `data=16'hFFFF;` Note that the apostrophe character (') is not the same as the grave accent (`), which is sometimes referred to as a "back tick"
- To make an assignment of all ones scalable, instead of specifying a literal value, a one's complement operator or a two's complement operator can be used

```
data = ~0; // one's complement operation
data = -1; // two's complement operation
```

- A vector can be filled with all zeros, all xs, or all zs

```
data = 'bz; // fills all bits of data with Z
data = 'bx; // fills all bits of data with X
```

- Constants starting with **Z** or **X** are padded with leading **Z**s or **X**s to reach their full length when necessary

10	unsized	decimal	0...01010 (32-bits)
'o7	unsized	octal	0...00111 (32-bits)
1'b1	1 bit	binary	1
8'hC5	8 bits	hex	11000101
6'hF0	6 bits	hex	110000 (truncated)
6'hF	6 bits	hex	001111 (zero filled)
6'hZ	6 bits	hex	ZZZZZZ (Z filled)

- If an expression is assigned to a signal with a smaller vector size, the left-most bits of the value of the expression are truncated. If the expression is assigned to a larger vector size, then the value of the expression is *left-extended*
  - An unsigned expression is *zero-extended* and a signed expression is *sign-extended*

Size casting and data type conversion

```
-253          // A signed decimal number
'haf          // An unsized hex number
'h12    // hex number 12 (18 decimal number)
'o12    // octal number 12 (10 decimal number)
'b1001        // binary number 1001 (9 decimal number)
8'b10010011  // 8-bit binary number
8'h12         // hex number 12 taking 8 bits
6'o67         // A 6 bit octal number
8'd12         // decimal number 12 taking 8 bits
8'b1010_0011 // 10100011
8'b1    // binary number 00000001 (zero filled)
16'hf1       // 0000000011110001 (zero filled)
8'bx          // An 8 bit unknown number (8'bxxxx_xxxx)
4'b100Z       // 100Z
4'bz1         // All but the lsb are Z (4'bzzz1) (Z filled)
8'h1?         // 0001ZZZZ
2'b1?         // 1Z
4'b10XX       // 10XX
```

- Verilog is a loosely typed language that allows a value of one data type to be assigned to a variable or net of a different data type. Verilog does type conversion when assigning the values
  - When a **wire** type is assigned to a **reg** variable, the value on the **wire** (which has 4-state values, and multi-driver resolution) is automatically converted to a **reg** value type (which has 4-state values, but no multi-driver resolution)
  - If a **real** is assigned to a **reg** variable, the 64-bit floating-point value is automatically rounded off to an integer of the size of the **reg**

Operator precedence

- When an expression is evaluated, the operator with higher precedence is evaluated first
- This table shows the precedence of operators from highest to lowest
  - Operators on the same level evaluate from left to right

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X = 3'B101 and Y=3'B110, then X&Y=3'B100, X^Y=3'B011
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3'B101, 3'B110} = 6'B101110;
{{ }}	replication; {3{3'B110}} = 9'B110110110
*, /, %	multiply, divide, modulus. *Z and % not supported for synthesis*
+, -	binary add, subtract
<<, >>	shift left, shift right; X<<2 is multiply by 4
<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers
==, !=	logical equality, logical inequality
===, !==	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; OR together all the bits in a word
&&	logical AND; Treat all variables as False (zero) or True (nonzero); (3&&0) is (T&&F) = 0
||	logical OR; (7||0) is (T||F) = 1, (2||-3) is (T||T) = 1
? :	conditional; x=(cond) ? T : F;

Operator precedence

- For example, in the `a + b >> 1` expression, `a + b` is evaluated first and then `>> 1` is evaluated
- We can use parentheses to alter the precedence, as in `a + (b >> 1 )`
  - It is a good practice to use parentheses to make an expression clearer, even when they are not required (e.g.,`(a + b) >> 1`)
- Integer division truncates any fractional part towards zero
- Modulus operator % gives the remainder of the first operand divided by the second. The result takes the sign of the first operand
- The operators / and % are only synthesizable as shifts, or in constant expressions (e.g. /2 means shift right)

Signed and unsigned signals and numbers

- Nets and **reg** types are *unsigned* by default. In this example, **reg** is assigned a negative value but it is considered as an unsigned

```
reg [15:0] A; // A is sized
A = -16'd12;  // stored as 1111_1111_1111_0100= 2^16-12= 65524. so A/3 = 21861
```

- Verilog uses 2's complement arithmetic for *signed* operands and values
- **integer** is a signed data type that is 32 bits in most synthesis tools
- An integer number with no radix specified (e.g. -12) is considered a signed value, however, an integer with a radix specified (e.g. -8'd12) is considered an unsigned value

```
- -12/3;          // evaluates to -4
- -8'd12 / 3;   // evaluates to 244/3=81 (i.e. 8'b11110100/3)
```

- Sized numbers (constant values) are considered unsigned

```
16'hC501  //an unsigned 16-bit hex value
-8'd12    // stored as 11110100 unsigned
```

- Sized numbers may be signed by combining letter 's' with the radix as

```
<size>'<S><base><value>

16'shC501 //a signed 16-bit hex value
```

Signed and unsigned modifiers

- A negative number is not sign-extended when assigned to a sized register

Signed and unsigned arithmetic

- To perform signed arithmetic, all operands in an expression must be signed. If any

Signed and unsigned modifiers

- A negative number is not sign-extended when assigned to a sized register

```
reg [7:0] byte;
reg [3:0] nibble;
initial begin     //Initial statements are used in testbenches
  nibble = -1;    // i.e. 4'b1111
  byte = nibble;  // byte becomes 8'b0000_1111
end
```

- Signed data types can be defined using the **signed** keyword. This modifier overrides the default definition of unsigned data types in Verilog

```
reg [63:0] u; // unsigned 64-bit variable
reg signed [63:0] s; // signed 64-bit variable
wire signed [7:0] vector;
input signed [31:0] a;
```

- Part-select results are unsigned, regardless of the operands, even if part-select specifies the entire vector

```
input signed [7:0] a, b;
output signed [15:0] z1, z2;
assign z1 = a[7:0];     // a[7:0] is unsigned -> zero-extended
assign z2 = a[6:0] * b; // a[6:0] is unsigned -> unsigned multiply
```

Signed and unsigned arithmetic

- To perform signed arithmetic, all operands in an expression must be signed. If any operand in an expression is unsigned, the operation is considered to be unsigned
- Signed arithmetic operations is supported when all the operands of the expression are **signed**

```
module sadder (input signed [7:0] A, B,
               output signed [8:0] SUM);
  assign SUM = A + B;
  // A and B are implicitly sign extended
endmodule
```

```
//signed multiplier
module mult_signed
    (input signed [2:0] a, b,
     output signed [5:0] prod);
  assign prod = a*b;
endmodule
```

- If you add two *N*-bit numbers the correct output has *N*+1 bits and if you multiply two *N*-bit numbers, the output has 2*N* bits
- Addition and Boolean logic operations are identical for signed or unsigned numbers, however, magnitude comparison, multiplication, and arithmetic right shifts are performed differently for signed numbers

Signed and unsigned system functions

- System functions use $ symbol as their first character. The system functions **$signed** and **$unsigned** can be used to convert an unsigned value to signed or vice-versa
- Casting using **$unsigned(signal_name)** will zero-extend the input and casting using **$signed(signal_name)** will sign-extend the input
- Assigning to a smaller bit width signal will truncate the necessary significant bits
- Casting to the same bit width will have no effect
  - The casting operators, **$unsigned** and **$signed**, only have effect when casting a smaller bit width to a larger bit
- If the sign bit is **X** or **Z**, the value will be sign extended using **X** or **Z**, respectively

```
// signed multiply
input [7:0] a, b;
output [15:0] z;
wire signed [15:0] z_sgn;
assign z_sgn = $signed(a) * $signed(b);
assign z = $unsigned(z_sgn);
```

```
//signed multiply
input signed [7:0] a;
output signed [15:0] z1, z2;
// cast constant into signed
assign z1 = a * $signed(4'b1011);
// mark constant as signed
assign z2 = a * 4'sb1011;
```

```
input signed [7:0] a, b;
assign z1=a; //a is signed. It is sign-extended and is assigned to z1
assign z2=$signed(a[7:0]) * b;//cast a[7:0] to signed and perform signed multiply
```

Errors can be hard to debug

- Adding two values that are n-bits wide will produce a n+1 bit result
  - In general adding an *m*-bit and an *n*-bit numbers require max(*m*,*n*)+1 bit for results
- A and B and cin are unsigned and hence the addition is unsigned
- {A[2], A} results in a one bit sign-extended and hence, 4-b operand

```
module uadd (input [2:0] A,
             input [2:0] B,
             input cin,
             output [3:0] Sum);
  assign Sum = {A[2],A} + {B[2],B} + cin;
endmodule
```

- Note that concatenation results are unsigned, regardless of the operands
  - Thus, **$signed** system function is used
- This code is incorrect
  - If **cin = 1**, then the **$signed** operator sign extends the **cin** to **4'b1111**, which is -1 in 2's complement format. Thus this means subtracting 1 instead of adding 1
- A similar functional error occurs if we declare **cin** to be a **signed** input

```
module sadd (input signed [2:0] A,
             input signed [2:0] B,
             input cin,
             output signed [3:0] Sum);
  assign Sum = A + B + $signed(cin);
endmodule

module sadd (input signed [2:0] A,
             input signed [2:0] B,
             input cin,
             output signed [3:0] Sum);
  assign Sum = A + B + $signed({1'b0,cin});
endmodule
```

Mixing signed and unsigned

- Do not mix **unsigned** and **signed** types in one expression. In an expression if any of the operands is unsigned, the entire expression is evaluated as unsigned
- If we multiply –3 (3'b101) by 2 (3'b010) with the following code we get 10 (6'b001010)

```
module mult (input signed [2:0] a,
             input [2:0] b,
             output signed [5:0] prod);
  assign prod = a*b;
endmodule
```

```
//unsigned multiply
input [7:0] a;
input signed [7:0] b;
output signed [15:0] z;
   assign z = a * b;
```

```
// unsigned multiply
input signed [7:0] a;
output signed [11:0] z;
// constant is unsigned
assign z = a * 4'b1011;
```

```
//signed multiply
input [7:0] a;
input signed [7:0] b;
output signed [15:0] z;
assign z = $signed(a) * b;
```

```
//signed multiply
input signed [7:0] a, b;
output signed [15:0] z;
assign z = a * b;
```

- The synthesizer generates a warning message when unsigned-to-signed/signed-to-unsigned conversions occurs
  - The warning messages in the synthesis report should be carefully reviewed

Parameters and parameterized modules

- The code is easier to understand by using parameters instead of hard-coded numbers. You can use the values 0 and 1, but do not use hard-coded numeric values in your design
- **parameter** defines a constant value. A parameter constant is local to a module
- Module definitions may be described using inline parameters definition, creating a *parameterized module*
- The parameters can be declared right after the **module** keyword and name as:

```
parameter [ signed ] [ range ] list_of_param_assignments;
                    | parameter integer list_of_param_assignments;
                    | parameter real list_of_param_assignments;
                    | parameter realtime list_of_param_assignments;
                    | parameter time list_of_param_assignments;
```

- Parameters can be signed, sized (with a range) and typed. If no size is specified for the constant, 32-bits are assumed
- The list of parameters is declared before the port list
- Port specifications can be parameterized

```
module multiplier #(parameter SizeA = 8,
                              SizeB = 4)
                   (output [SizeA+SizeB-1:0] mult,
                    input [SizeA-1:0] a,
                    input [SizeB-1:0] b);
  assign mult = a * b;
endmodule
```

Parametric module example

- This example presents an 8-bit **XOR** module that instantiates 8 **XOR** gates. Instead of instantiating 8 XOR gates, a single **assign** statement used in this parameterized module

```
module xorx  #(parameter width = 8, delay = 2)
             (output [1:width] xout,
              input  [1:width] xin1, xin2);
  assign #(delay) xout = xin1 ^ xin2;
endmodule
```

- When module xorx is instantiated, the default values specified in the parameter declaration are used. However, an instantiation of this module may override these parameters
  - In the following example, the **#(16,4)** specifies that the value of the first parameter **width** is 16 for this instantiation, and the value of the second parameter **delay** is 4
  - If the **#(16,4)** was omitted, then the values specified in the module definition would be used instead

```
module overriddenParameters(input [15:0] b1,c1,b2,c2,
                            output [15:0] a1, a2);
  xorx #(16,4) a(a1, b1, c1),
               b(a2, b2, c2);
endmodule
```

Parameter values for module instances

- For instantiating parameterized modules, the value of parameters can be redefined for each instance of a module using in-line parameter redefinition
- Since parameters can be overridden, they allow *customization* of a parameterized module during instantiation by specifying the value of the parameters at each instantiation of the module, making the module configurable and reusable
  - Reusable means the same module definition can be used for various designs. For example, the parameterized modules can be used (instantiated) for multipliers with different widths, simply by changing the default value of the parameters
- Verilog constants receive their final value at elaboration time. *Elaboration* is the process of building the hierarchy of the design represented by module instances. Some synthesis tools have separate compile and elaboration phases and some tools combine them into a single process

Overriding parameters

`define compiler directive

Overriding parameters

- The module parameters can be explicitly overridden by naming the parameters at the instantiation
- The syntax for the parameter declaration is the same as the named port declaration of a module
- Parameters can be re-defined and listed in any order. Those not listed at the instantiation will retain their default values

```
module mux2g  #(parameter WL = 4)
              (input [WL-1:0] a, b,
               input sel,
               output [WL-1:0] y);
   assign y = sel ? a : b;
endmodule
```

```
// 8-bit 2-to-1 MUX using a parameter
module mux2_8bits (input [7:0] a,
                   input [7:0] b,
                   input sel,
                   output [7:0] y);
   mux2g #(.WL(8)) M8 (.a(a),
                       .b(b),
                       .sel(sel),
                       .y(y));
endmodule
```

`define compiler directive

- *Macros* are used for global definition of an identifier, which will not be modified elsewhere in the design. It can be defined as `define <macroName> <textString> [(arguments)], and the `macroName (a macro is invoked with the quoted macro name) in the code will be substituted with textString, at the beginning of synthesis and simulation. Macros improve the readability and maintainability of the Verilog code. A macro definition does not end with a semicolon

```
`define BUS reg [31:0]
`BUS data;
```

```
`define add(a,b) a + b
f = `add(1,2); // f = 1 + 2;
```

- Clock signal can be generated using a `forever` statement

```
forever [begin]
... Procedural statements ...
[end]
```

- All compiler directives, including macro definitions, are active from the point of definition and remain active across all files read after the macro definition is made until overridden by a subsequent `define, `undef macro_name (used to undefine a macro definition) or `resetall directive (resets all compiler directives to default values)

```
`define CYCLE 10
module tb_cycle;
initial begin
  clk = 1'b0;
  forever #(`CYCLE/2) clk=~clk;
end
// ...
endmodule
```

```
`define BUS_WIDTH 16
     reg [`BUS_WIDTH-1:0] dataBus;
  _
`undef BUS_WIDTH
```


Macros vs. parameters

- Since macros are defined for all files read after the macro definition, using macro definitions generally makes compiling order-dependent. If the same macro name has been given multiple definitions in a design, only the last definition will be effective
- Since macros are used for global definitions, include all macro definitions in your top-level module so that they are globally available to all modules compiled in the design or in one "macro.vh" file and read the file first using `include compiler directive
- A parameter, after it is declared, is referenced using the parameter name, however, a `define macro definition, after it is defined, is referenced using the macro name with a preceding back-tic character. Parameter declarations can only be made inside of module boundaries, however, macro definitions can exist either inside or outside of a module declaration

(4) Behavioural level modelling

- The dataflow model using continuous assignment statements is used when a combinational logic can be described using a few assign statements. However, more complex combinational functions are typically easier to describe at the *behavioral level*. The behavioral description specifies what a particular module does at a higher level of abstraction than the gate-level and dataflow level
- A behavioral model of a module is an abstraction of how the module works (describing the function of a module behaviorally) without directly specifying how the module is implemented in terms of logic gates, structural interconnection of sub-modules, or dataflow description of signals
- Behavioral models are useful at the early stages of the design cycle where a designer is focusing on modeling the system's intended functionality. Then the logic synthesis tool reads the behavioral description of a module and automatically generates its equivalent gate-level netlist
  - A behavioral model can be synthesized into various netlist implementations of the same behavior, but having different area, performance, and power consumption

Behavioral model

- The behavior of digital systems can be described using a set of independent, but communicating *processes*
- A process can described using a *procedural block*. Two procedural blocks in Verilog include **always** and **initial** statements
  - **always** is a synthesizable compound statement for describing the logic behavior
- An **always** block uses *procedural statements* to model algorithmic behavior, without describing the implementation details of that behavior
- An **always** block containing more than one procedural statement must enclose the statements in a **begin-end** block (i.e., a compound statement)
- Procedural statements execute in the order they are written or simultaneously
- An **always** block models the continuous operation of hardware by continuously executing its statements. The **always** statement, essentially a "while (TRUE)" statement, where its procedural statements are repeatedly executed. When the statements are completed, it returns to the beginning
- To control the simulation time, the loop must contain time control in the form of:
  - a fixed delay, represented with the # symbol; a delay until an expression evaluates as true, represented with the **wait** keyword; a delay until an expression changes value, represented with an event control expression

Event control expression

- An *event control expression* has the format of @(sensitivity list) provides a means of monitoring events
- The *sensitivity list* includes the list of signals that **always** is monitoring
- An event is a change in a signal's (a net or **reg**) value and the change may be a *level change*, or an *edge change*
- If the event control lists the name of one or more signals, **always** is sensitive to level change and if it is described using the *edge specifiers* **posedge** (0→1, 0→X,0→Z, X→1, Z→1) or **negedge** (1→0, 1→X,1→Z, X→0, Z→0) of signals, **always** is sensitive to positive or negative transitions on signals
  - Event identifiers are separated with commas or **ors**

```
always [event_control]
begin [: name_for_block]
[variable declaration]
... Procedural statements ...
end
```

```
always @ (variable, variable, . . .) statement;
always @ (posedge variable or negedge variable) statement;
```

```
always @(a or b)       // level-triggered
@(CLK) Q = D;          // assignment is performed whenever signal CLK changes to its value
@(posedge CLK) Q = D; // assignment is performed whenever signal CLK has a rising edge
@(negedge CLK) Q = D; // assignment is performed whenever signal CLK has a falling edge
```

Events and event control

- Sensitivity list can be used to describe both combinational and sequential logic
- The **@** (<signals>) event control expression may synthesizes to a combinational logic or to a sequential logic
- Sensitivity list for the sequential logic has the format of **@(<pos|neg>edge <signal>)**
- An event control (**@posedge** or **@negedge**) can be used before a statement and it causes the execution of the immediately following statement to be delayed (similar to timing control using **#**). It is commonly used in testbenches
- At the start of simulation at time zero, all **always** and **initial** statements enter an *initialization* phase, each signal is given its initial value, and then the simulation cycle enters the *suspended state*. Then an event on any one (or more) of the signals listed in the sensitivity list (i.e., change in the logical value of a signal) activates the **always** block. Then all statements within the **always** statement are executed *sequentially* once. The execution is then *suspended* until the next event occurs on the signals in its sensitivity list
- **always** statements can contain any number of time controls or event controls, and the controls can be specified anywhere within the procedural block
- An **always** with no event control will loop forever

Combinational logic using **always**

- A combinational logic (CL) will be inferred if the sensitivity list includes all of the CL inputs and the always block describes the output value for every possible input combinations. This follows the definition of CL: *any* change of *any* input value may have an effect on the resulting output and for any change of value on any inputs, all the outputs should have a value
- **always** @(event list) synthesizes to a CL if the list includes all the variables (not temporary variables) on the RHS of the procedural statements within the always. If one or more inputs of a CL is missing from the sensitivity list, the synthesized logic would be a sequential logic
- For a CL, the list must specify only level changes. The level-sensitive event control cannot contain **posedge** or **negedge** qualifiers. The **always** block cannot contain any other event controls
- All LHS variables within the **always** block must be updated for all possible input conditions. Any variables written to by the **always** block cannot be written to by any other procedural block

```
module mux(output reg f,
           input a,b,sel)
   always @(a,b,sel)
     f=(a&~sel) | (b&sel);
endmodule
```

```
module halfAdder (input A, B, output reg Sum, Cout);
always @(A,B) begin
     Sum  = A ^ B;
     Cout = A & B;
   end
endmodule
```

always @*

- `@*` or `@(*)` event controls is shorthand for listing all the signals on the right-hand side of the statements in the always block in the sensitivity list
- Thus `always @(*)` or `@*` allow modeling CL sensitivity lists without having to specify all the input signals to the CL

```
module shift #(parameter pos = 2)
              (input [3:0] data,
               output reg [3:0] q1, q2);
always @* begin
  q1 = data << pos; // logical shift left
  q2 = data >> pos; //logical shift right
 end
endmodule
```

- A behavioral description of a hardware can be described using one or more **always** blocks. Several **always** processes are executed concurrently and interact
- Note that a module may contain a combination of *behavioral* modeling statements (**always** statements), *continuous assignment* statements (**assign** statements), or module or gate *instantiations*. When an **always** block is waiting to continue (due to **#**, **wait** , or event controls), other **always** blocks, **assign** statements, and modules are executing
- **initial** and **always** statements cannot be nested. Modules can not be instantiated inside procedural blocks, such as **always** statements. They are only instantiated in the main module's body

Combinational logic examples

```
module arithmetic (input [3:0] A, B,
                   output reg [4:0] q1,
                   output reg [3:0] q2,
                   output reg [7:0] q3);
   always @* begin
        q1 = A + B; //addition
        q2 = A - B; //subtraction
        q3 = A * B; //multiplication
   end
endmodule
```

```
module logical (input [3:0] A, B;
                output reg Q[1:6]);
 always @* begin
   Q[1] = A > B;  //greater than
   Q[2] = A < B;  //less than
   Q[3] = A >= B; //greater than equal to
   Q[4] = A <= B  //less than equal to
   Q[5] = A == B;  //equality
   Q[6] = (A != B) //inequality
end
endmodule
```

- The general rule is that a variable with **reg** data type must be used when modeling using **initial** and **always** procedural blocks, and a net must be used when modeling using continuous assignments, module instances, or primitive instances

LHS must be a reg type

```
reg [8:0] sum;
always @(a or b)
   sum = a + b;
```

LHS must be a net type

```
wire [8:0] sum;
assign sum = a + b;
```

```
wire y;
and g1(y, c, d);
```

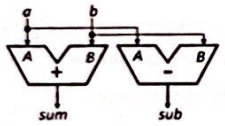
- If you change the module description from dataflow level to the behavioral level or vice versa, make sure the data types are declared correctly

Example

```
module adder (input [7:0] in1, in2,
              output reg [7:0] out);
    always @* out = in1 + in2;
endmodule
```

```
module tb;
reg [7:0] in1, in2;
wire [7:0] out;
parameter timeStep = 5;
    adder DUT (.in1(in1), .in2(in2), .out(out));
    initial begin
      in1 = 8'h01; in2 = 8'hff;
      #(2*timeStep) in1= 8'haa; in2=8hbb;
      #(2*timeStep)
      $finish;
    end
endmodule
```

```
module addSub (input signed [3:0] a, b,
               output reg signed [4:0] sum, sub);
 always @(a,b) begin
    sum = a + b;
    sub = a - b;
  end
endmodule
```




```
module addSub_tb;
reg [3:0] a,b;
wire [4:0] sum, sub;
  addSub UUT (a,b,sum,sub);
  initial begin
       a = 4'b0000; b = 4'b0000;
    #10 a = 4'b1111; b = 4'b1111;
    #10 a = 4'b1110; b = 4'b0111;
    #10 a = 4'b1010; b = 4'b0101;
    #10 a = 4'b0011; b = 4'b1100;
  end
endmodule
```

Restrictions on using **reg** and net types

- Nets are used when modeling using continuous assignments and to model connections between module instances or primitive gates
- Net variables can be read anywhere in a module (i.e., in the procedural blocks as well), but they may not be assigned within procedural blocks
- A **reg** variable can be read anywhere in module, but they can be assigned only with procedural statements
- A **reg** type must be used when the signal is on the LHS of a procedural statement (e.g., within an **always** and **initial** blocks). Why only **reg** type variables can be assigned within an **always** block?
  - The sequential **always** block executes only when the event expression triggers. At other times the block is in the suspended mode (not like nets that are continuously driven). A signal being assigned to must therefore retain the last value assigned
- A **reg** type variable cannot be driven by multiple sources from different **always** blocks. Variables do not have built-in *resolution functionality* to resolve a final value when two or more devices drive the same variable. Only nets can have multiple sources as they have built-in resolution functions to resolve multi-driver logic. For example, a data bus declared as a net data type, can be driven from several drivers

Procedural conditional statements: `if` statement

- An if-then-else statement is used to conditionally execute procedural statements based on the value a Boolean expression
- If statements generally synthesize to multiplexers
- Each condition of the if-then-else statement is checked in order until a true condition is found

```
if (expression) begin
... Procedural Statements ...
end
else if (expression) begin
... Procedural Statements ...
end
...more else if blocks ...
else begin
... Procedural Statements ...
end
```

- If more than one statement is required to be executed in either the `if` or the `else` branch, the statements must be enclosed in a `begin-end` block. If there is
- An `else` is associated with the immediately preceding `if`, unless an appropriate `begin-end` is present
- In this example, the `begin-end` block causes the `else` to be paired with the first `if` rather than the second

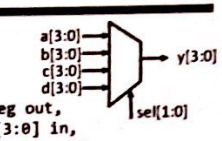
```
if (expressionA)
 if (expressionB)
   a = a + b;
else
   q = r + s;
```

```
if (expressionA) begin
 if (expressionB)
   a = a + b;
end
else
q = r + s;
```

- There can be as many `else if` statements as required, but only one `if` block and one `else` block are allowed. Both the `else if` and `else` statements are optional
- If the `if` statement's expressions involve unknown or high impedance values, the expression will be interpreted as FALSE

`if` examples

```
module mux_2x1(input a, b, sel,
               output reg out);
   always @* begin
         if (sel) //if (sel == 1)
             out = a;
         else out = b;
   end
endmodule
module mux4x1_4bits
(input [3:0] a, b, c, d,
 input [1:0] sel,
 output reg [3:0] y);
  always @*
     if (sel == 0) y = a;
     else if (sel == 1) y = b;
     else if (sel == 2) y = c;
     else if (sel == 3) y = d;
     else y = 4'bx; --sel can be X or Z
endmodule
```




```
module mux4x1_1bit(output reg out,
                   input [3:0] in,
                   input [1:0] sel);
  always @*
       if (sel == 0)      out = in[0];
       else if (sel == 1) out = in[1];
       else if (sel == 2) out = in[2];
       else out = in[3];
endmodule
```

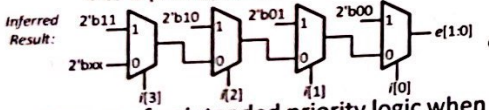
- *Tri-state buffers* are CL circuits that have three output values: one, zero, and high impedance (`z`)

```
module synTriState (output reg bus,
                    input in, Enable);
  always @(*)
    if (Enable)
     bus = in;
    else bus = 1'bz;
endmodule
```

Priority logic

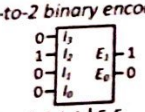
- A set of nested if-else statements can be used to create *priority logic*
- If `i[0]` is 1, the result is 00 regardless of the other inputs. So `i[0]` has the highest priority
  - This code will be synthesized to a 4-to-2 priority encoder

```
module binary_encoder (input [3:0] i,
                       output reg [1:0] e);
 always @* begin
  if (i[0]) e=2'b00;
  else if (i[1]) e=2'b01;
  else if (i[2]) e=2'b10;
  else if (i[3]) e=2'b11;
  else e=2'bxx;
 end
endmodule
```



- Beware of unintended priority logic when using if statements. Priority encoded logic can impact timing due to their cascading structure. If mutually-exclusive conditions are chosen for each branch, then the synthesis tool can generate a circuit that evaluates the branches in parallel

4-to-2 binary encoder



I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

```
module binary_encoder (input [3:0] i, output reg [1:0] e);
 always @* begin
  if (i==4'b0001) e=2'b00;
  else if (i==4'b0010) e=2'b01;
  else if (i ==4'b0100) e=2'b10;
  else if (i ==4'b1000) e=2'b11;
  else e=2'bxx;
 end
endmodule
```

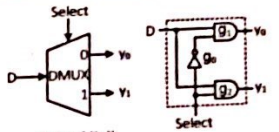
Assuming only one of the bits of i is 1, these are mutually-exclusive conditions

Synthesizing combinational logic with different execution paths

- When using conditional statements (e.g., if), there may be many different execution paths in the `always` block. To produce a CL, the output of the CL must be assigned in each and every one of the different execution paths. If a variable is not assigned to for all the possible branch conditions, a latch will be inferred
- Thus in the behavioral description of a CL, a LHS variable must be assigned a value at least once in every execution of the `always` block

```
module demux (input D, select,
              output [1:0] y);
   assign y[0] = (~select) & D;
   assign y[1] = select & D;
endmodule
module demux (input D, select,
              output [1:0] y)

wire N;
   and g1(y[0], D, N);
   and g2(y[1], D, Select);
   not g0(N, Select);
endmodule
```

```
module demux (input D, select,
              output reg [1:0] y);
   always @( D or select ) begin
      if( select == 1'b0) begin
          y[0] = D;
          y[1] = 1'b0;
      end
      else begin
          y[0] = 1'b0;
          y[1] = D;
      end
   end
endmodule
```




```
module demux #(parameter WL = 8)
              (input [WL-1:0] In,
               input sel,
               output reg [WL-1:0] y0, y1);
  always @* begin
      if(sel) begin
          y1 <= In; y0 <= 0;
      end
      else begin
          y1 <= 0;  y0 <= In;
      end
  end
endmodule
```

Latch inferrence

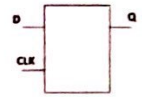
- Synthesis tools can only interpret the HDL code, and "*infer*" the logic based on the module description
- Synthesis tool decides what is the best architectural implementation for a given description, depending on the design constraints, such as area and performance,
  - Typically there are several implementations in the standard cell library for each operator. For example, ripple-carry adders, carry-look-ahead adders, etc.
- If the **always** block is executed and no value is assigned to a signal for some input conditions, the circuit needs to remember the previous value of the signals. In this case, the signal is a function of the current inputs *and* the previous input. This is the fundamental characteristic of a *sequential circuit*, where the state is remembered in a *latch*. Therefore, the synthesized design will have storage elements to implement the sequential nature of the description
- Therefore, if a signal is left unassigned in at least one of the execution paths, then the previous value will be remembered and will cause a storage element implemented by latches to be inferred

D-Latch

- Sequential circuits use the latches and flip-flops as storage elements. Latches are *level-sensitive* storage devices. A *positive* (*negative*) *D latch* is transparent when the clock is high (low), allowing data to flow from input to output. The latch becomes *opaque* when the clock is low (high), retaining its state
- Latches are not explicitly specified. Rather, they are implemented by inference from the way in which a description is written. In the latch description, the sensitivity list must not contain any edge specifiers (**posedge or negedge**)
- This code infers a latch, because the output **Q** is not assigned under all possible conditions

Incompletely specified if statements cause synthesis tool to infer latches

```
module dLatch (input CLK, D, output reg Q);
  always @(D or CLK)
      if (CLK)
        Q = D;
endmodule
```



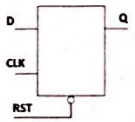
- To prevent the tool from inferring unintentional latches, we need to make a default assignment to **Q** outside of the **if** statement or add an **else** branch to the **if** statement. Make sure that your HDL does not imply any unintended latches
- Synthesis tool may issue warnings for signals that are read in an **always** block but are not listed in the sensitivity list
- Synthesis tools usually infer latches and flip-flops from **always** blocks, but not from continuous assignments

D-latch with control signals

- Set and/or reset inputs may change the storage element state either *synchronously* or *asynchronously* with respect to the clock
- Asynchronous control signals must be listed in the sensitivity list of the **always** block and the tests for the set and reset conditions are done first in the **always** block

```
module dlatchwReset (input RST, CLK, D, output reg Q);
    always @ (RST or CLK or D)
      if (!RST) //using logical operator
          Q = 1'b0; // Q = 0;
      else if (CLK)
          Q = D;
    endmodule
```

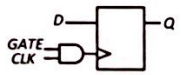
D-latch with active-high clock and asynchronous reset




```
module latchwPreset (input CLK, PRE,
                     input [3:0] D,
                     output reg [3:0] Q);
  always @(CLK,D,PRE) begin
     if (PRE) Q = 4'b1111;
     else if (!CLK) Q = D;
  end
endmodule
```

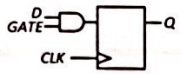
//4-bit latch with active-low clock and asynchronous preset

```
module dlatch (input CLK, GATE, D,
               output reg Q);
  always @ (CLK,D,GATE)
      if (CLK && GATE)
          Q = D;
endmodule
```



D-latch with gated clock

```
module dlatch (input CLK, GATE, D,
               output reg Q);
  always @(CLK,D,GATE)
  //the same as always@(CLK, D, GATE)
      if (CLK)   Q = (D & GATE);
endmodule
```



D-Latch with gated data

D-latch examples

- An adder/subtractor with an output latch

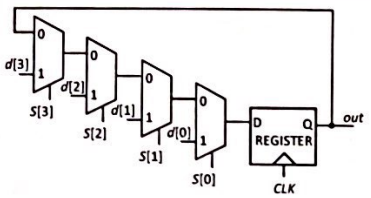
```
module addSub #(parameter Width = 4)
               (output reg [Width-1:0] out
                input [Width-1:0] a, b,
                input EN, addsub);
  always @(*)
    if (EN) begin
      if (addsub) out= a+b;
      else out = a-b;
    end
endmodule
```

- In this example, the signal **EN** is read, but it is not in the sensitivity list. Assuming that **RST** is stable at 0, a change in **EN** from 0 to 1 does not trigger the **always** block, so the value of **D** does not get latched onto **Q**

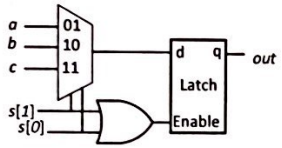
```
always @(D or RST)
  if (RST)
    Q = 1'b0
  else if (EN)
    Q = D;
```

D-latch inferrence

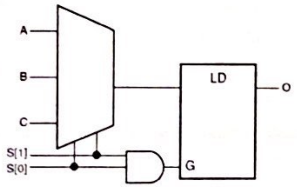
```
module binary_encoder #(parameter N = 16)
                       (output reg out,
                        input CLK,
                        input [N-1:0] d,
                        input [3:0] S);
 always @(posedge CLK)
  if (S[0]) out<= d[0];
  else if (S[1]) out<= d[1];
  else if (S[2]) out<= d[2];
  else if (S[3]) out<= d[3];
endmodule
```




```
module module1 (input a,b,c,
                input [1:0] s,
                output reg out);
   always @* begin
    if (s == 2'b01) out = a;
    else if (s == 2'b10) out = b;
    else if (s == 2'b11) out = c;
  end
endmodule
```




```
module module2 (input a,b,c,
                input [1:0] s,
                output reg out);
   always @* begin
    if (s == 2'b00) out = a;
    else if (s == 2'b01) out = b;
    else if (s == 2'b10) out = c;
  end
endmodule
```



Blocking assignment statements

- HDL supports two procedural assignment statements, *blocking assignment* using an equal operator = , and the *nonblocking assignment* denoted with <=
- The primary purpose of the blocking and nonblocking assignment operators is to accurately emulate the behavior of combinational and sequential logic
- A blocking assignment "blocks" following assignments in an **always** block from occurring until after the current assignment has been completed. The LHS variable of a blocking assignment gets updated before the next statement is executed. A group of blocking assignments are evaluated in the order they appear in the code. Thus the order of blocking assignment statements is important. Assignments made using the blocking assignment take effect immediately and the value written to the LHS variable of a statement is available for use in the next statement
- It is most efficient to use blocking and non-blocking assignments to generate CL and sequential logic, respectively. Ignoring this guidelines can still infer the correct synthesized logic, but the pre-synthesis simulation might not match the behavior of the synthesized circuit
- Note that continuous signal assignment statements using the **assign** keyword are used outside **always** statements

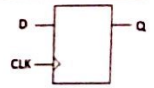
Non-blocking assignment statements

- A nonblocking assignment does not block subsequent non-blocking statements from being evaluated
- Evaluation of nonblocking assignments can be viewed as a two-step process
  I. Evaluating the RHS expression of all nonblocking statements concurrently using the current values of signal at the beginning of the current time step
  II. Update the LHS variables of nonblocking statements at the end of the current time step concurrently
- The order of non-blobking assignments is not important (unless two or more nonblocking statements are assigned to the same LHS variable, in which case the value of the RHS of the last statement will be assigned to the LHS variable)

```
always begin
  a <= x+y;
  a <= z+k;
end
```

- The non-blocking assignment can be used for *synchronized* implementation of digital logic. When the active clock edge occurs, all of the RHS expressions of non-blocking assignment statements will be evaluated using the current values of signals (i.e., values before the active edge of the clock) and all of the assignments occur at the same time to the LHS variables concurrently
- If an **always** block only has one procedural statement, then blocking and nonblocking assignments are equivalent

Flip-flops



- Flip-flops (FFs) are *edge-sensitive* storage devices. Their behavior is controlled by a positive or negative edge of the clock signal
- Edge-triggers are specified by **posedge** and **negedge** keywords
- When an edge event occurs, the input data **D** is passed to the output
- The procedural event control statement **@(posedge CLK)** monitors for the positive transition of **CLK** and then assigns the value of **D** to **Q**

```
module dff (input D, CLK,
            output reg Q);

  always @(posedge CLK)
    Q <= D;
endmodule
```

- The value assigned to **Q** is the value of **D** just before the positive edge of the clock
- After the active clock edge, a change on **D** will not change the FF state. So the **D** input is not included in the sensitivity list
- Only the transition of FF control signals (here **CLK**) can cause state changes
- The sensitivity list of the **always** block includes only the edge specifies for control signals. Level and edge specifiers cannot be combined in the same sensitivity list

D flip-flops with reset and preset

- In this example, four instances of FFs are used to implement an eight-bit register. Note that CLK, being a one-bit scalar, is connected to each FF instance

```
module dff (output q,
            input d,
            input CLK);
  always @(posedge CLK) q <= d;
endmodule
```

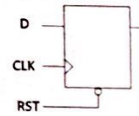
```
module register (output [3:0] q,
                 input [3:0] d,
                 input CLK, RST);
  dff r[3:0] (.q(q), .d(d), .CLK(CLK));
endmodule
```

```
module register (output [3:0] q,
                 input [3:0] d,
                 input CLK);
  dff b3 (.q(q[3]), .d(d[3]), .CLK(CLK)),
      b2 (.q(q[2]), .d(d[2]), .CLK(CLK)),
      b1 (.q(q[1]), .d(d[1]), .CLK(CLK)),
      b0 (.q(q[0]), .d(d[0]), .CLK(CLK));
endmodule
```

- FFs typically include reset signals to initialize their state at system start-up
- If an active-low reset signals, you can write:

`if (!reset)` or `if (reset == 1'b0)` or `if (reset == 0)`

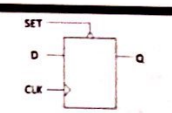
```
module dff (input D, CLK, RST, output reg Q);
  always @(posedge CLK)
    if (!RST)  Q <= 1'b0;
    else       Q <= D;
endmodule
```



DFF with synchronous reset

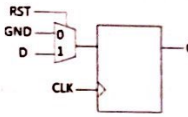
FF examples

```
module dff_sync_pre (input D, CLK, SET,
                     output reg Q);
  always @(posedge CLK)
    if (!SET)  Q <= 1'b1; //Q <= 1
    else       Q <= D;
endmodule
```



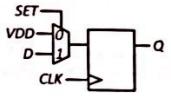
flip-flop with synchronous active-low preset

```
module dff_sync_rst (input D, CLK, RST,
                     output reg Q);
  always @(posedge CLK)
    if (!RST) Q <= 1'b0;
    else      Q <= D;
endmodule
```



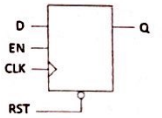
flip-flop with synchronous active-low reset

```
module dff_sync_pre (input D, CLK, SET,
                     output reg Q);
  always @ (posedge CLK)
    if (!SET) Q <= 1'b1; //active-low preset
    else      Q <= D;
endmodule
```



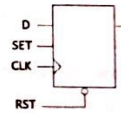
flip-flop with synchronous active-low preset

```
module dff_async_nrst (input D, CLK, RST, EN,
                       output reg q);
  always @ (posedge CLK or negedge RST)
    if (!RST)  Q <= 0;
    else if (EN) Q <= D;
endmodule
```



flip-flop with asynchronous active-low reset and enable

```
module dff_async_nrst (input CLK, RST, SET, D,
                       output reg Q);
  always @ (posedge CLK or negedge RST or posedge SET)
    if (!RST)     Q <= 0;
    else if (SET) Q <= 1;
    else          Q <= D;
endmodule
```

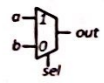


flip-flop with asynchronous active-low reset and preset

Examples

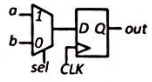
Combinational

```
module combinational(input a, b, sel,
                     output reg out);
always @(a,b,sel) begin
    if(sel) out = a;
    else out = b;
  end
endmodule
```



Sequential

```
module sequential(input a, b, sel, CLK,
                  output reg out);
always @(posedge clk) begin
    if(sel) out <= a;
    else out <= b;
  end
endmodule
```




```
// Wrong code
reg [15:0] word;
always @(posedge CLK) begin
  word[15:8] = word[ 7:0];
  word[ 7:0] = word[15:8];
end
```

```
//swap bytes in word
always @(posedge CLK) begin
  word[15:8] <= word[ 7:0];
  word[ 7:0] <= word[15:8];
end
always @(posedge CLK) begin
  word[7:0] <= word[15:8];
  word[15:8] <= word[7:0];
end
```

The order of nonblocking assignment statements is not important when the assignments are to different LHS registers

```
module reg_maps_to_wire (input A, B, C,
                         output reg f1, f2);
  always @(A or B or C) begin
    f1 = ~(A & B);
    f2 = f1 ^ C;
  end
endmodule
```

The synthesis system will generate a wire for f1

```
module reg_maps_to_wire (input A, B, C,
                         output reg f1, f2);
  always @(A or B or C) begin
    f2 = f1 ^ f2;
    f1 = ~(A & B);
  end
endmodule
```

Will the synthesis generate a storage cell for f1?

Sequential logic testing and clock signal generation

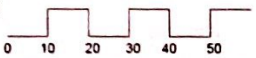
- *Cycle-based simulation* of sequential logic can be developed using testbenches, in which delays are multiple of the clock cycle
- Every clock cycle, a test vector can be applied to the DUT. We need a clock signal
- This code does not generate a periodic clock signal because the initial value of CLK (`wire` data type) is z (`~z = x` and `~x = x`)

```
wire CLK;
assign #10 CLK = ~ CLK;
```

- CLK has to be defined as type `reg` in order to be used in an `initial` statement
- Not including any delay control or event control in an `always` causes an infinite loop in the simulator. This `always` loop initializes the clock to 1 at time 10 and thereafter toggles its value every 10 time units

```
// Declare a constant clock period
parameter ClockPeriod = 20;
reg CLK;
initial CLK=0;//initialize the Clock signal
always #(ClockPeriod / 2) CLK = ~CLK;
```

```
//Clock generation
reg CLK =0;
always begin
  #10 CLK = 1;
  #10 CLK = 0;
end
```



CLK signal with 20ns clock cycle

Clock generation

```
reg Clk=0;
initial begin
   forever
     CLK = #(ClockPeriod / 2) ~ CLK;
end
```

- This **forever** loop is controlled with a **@(posedge CLK)** statement

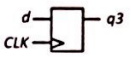
```
forever begin
  @(posedge CLK) a = a + 1;
end
```

- The use of the names, such as clock and reset, have no special meaning for a synthesis tool. These names are used for clarity. Could be any name
- Use consistent names and naming for your signals. For example, for active-low signals, end the signal name with an _n
- Use one-bit signals for clock, reset, preset, and enable control signals

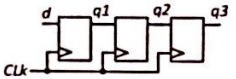
```
reg CLK;
 initial begin
   CLK = 0;
   forever begin
     #10 CLK = 1;
     #10 CLK = 0;
   end
end

 parameter T=10; // clock period
 //reset for the first half cycle
 initial begin
   reset = 1 'b1;
   #(T/2) reset = 1'b0;
 end
 always begin
  CLK = 1'b1;
  #(T/2) CLK = 1'b0;
  #(T/2);
 end
```

Order of blocking assignments

```
module pipe1 (output reg [7:0] q3,
              input [7:0] d, input clk);
reg [7:0] q2, q1;
always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
end
endmodule
```




```
module pipe2 (output reg [7:0] q3,
              input [7:0] d, input clk);
reg [7:0] q2, q1;
  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```



- Blocking assignments do not infer the behavior of multi-stage sequential logic, however, with reversing the orders of the blocking assignments, *a shift register* can be inferred

```
always@ (posedge CLK) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
end
```

```
always@ (posedge CLK) q1<=d ;
always@ (posedge CLK) q2<=q1;
always@ (posedge CLK) q3<=q2;
```

```
always@ (posedge CLK) begin
      q3 <= q2;
      q2 <= q1;
      q1 <= d;
end
```

```
always@ (posedge CLK) q3<=q2;
always@ (posedge CLK) q2<=q1;
always@ (posedge CLK) q1<=d;
```

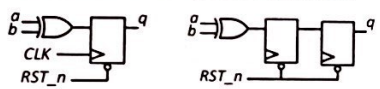
- While a blocking signal assignment statement can be used instead of a non-blocking statement when the always block has only one statement, it is not recommended

```
always@ (posedge CLK) q1=d ;
always@ (posedge CLK) q2=q1;
always@ (posedge CLK) q3=q2;
```


Mixing blocking and nonblocking assignments

- Mixing blocking and nonblocking assignments in the same **always** is not allowed

```
module test (output reg q,
             input CLK, RST, a, b);
reg tmp;
 always @*  tmp = a ^ b;
 always @(posedge CLK or negedge RST)
   if (!RST) q <= 0;
   else q <= tmp;
endmodule
```



Asynchronous active-low reset

- Any **reg** type variable assigned to in the sequential **always** block will be implemented using flip-flops or latches in the resulting synthesized circuit. Thus you cannot describe purely combinational logic in the same **always** block where you describe sequential logic

```
always @(posedge CLK or negedge RST)
    if (!RST) q <= 0;
    else begin
      tmp <= a ^ b;
      q <= tmp;
    end
```

```
always @(posedge CLK or negedge RST)
  if (!RST) q <= 0;
  else q  <= a ^ b;
```

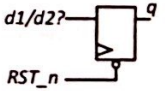
Combining the combinational logic with the sequential logic

- You can partition your design by separating the combinational logic from the sequential logic. You can describe your RTL design using separate **always** statements for sequential logic and combinatorial logic
- When modeling both sequential and combinational logic within the same **always** block, use nonblocking assignment statements

Race condition – Assigning from more than one block

- Verilog may simulate multiple **always** blocks in any order. A *race* condition occurs when two or more **always** that are scheduled to execute in the same simulation time step, would give different results if the order of executions is changed
- In this example, the two **always** blocks are executing at the same active edge of the clock and can be scheduled in any order

```
module badcode1 (output reg q,
                 input d1, d2, CLK, RST);
always @(posedge CLK or negedge RST)
   if (!RST) q <= 1'b0;
   else q <= d1;

always @(posedge CLK or negedge RST)
   if (!RST) q <= 1'b0;
   else q <= d2;
endmodule
```



Synthesis result does not match simulation

  - The pre-synthesis simulation logic may not match the post-synthesis logic due to the race condition
- The same reg type variable should not be assigned from more than one **always**
- *Inter-dependent assignments* occurs when the LHS variable of one assignment in one procedural block is also the RHS variable of another assignment in another procedural block and both statements are scheduled to execute in the same simulation time step

Race condition – Blocking and nonblocking statements

In-line initialization

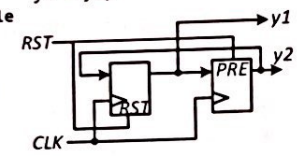
Race condition – Blocking and nonblocking statements

- If the first **always** block executes first after a reset, both y1 and y2 will take on the value of 1. If the second always block executes first after a reset, both y1 and y2 will take on the value 0
- Therefore, if blocking assignments are not properly ordered, a *race condition* can occur

```
module fbosc1 (output reg y1, y2;
               input CLK, RST);

always @(posedge CLK or posedge RST)
  if (RST) y1 = 0; // reset
  else     y1 = y2;

always @(posedge CLK or posedge RST)
  if (RST) y2 = 1; // preset
  else     y2 = y1;
endmodule
```

- In this example, inter-dependent nonblocking assignments execute in the same time step

```
module fbosc2 (output reg y1, y2,
               input CLK, RST);
always @(posedge CLK or posedge RST)
  if (RST) y1 <= 0; // reset
  else y1 <= y2;

always @(posedge CLK or posedge RST)
  if (RST) y2 <= 1; // preset
  else y2 <= y1;
endmodule
```



On RST, y1=0 and y2=1. After RST, on next posedge CLK, y1=1 and y2=0

No race condition!

- Concurrent nonblocking assignments have predictable results

```
always @(posedge CLK)
  A <= A + 1;
always @(posedge CLK)
  B <= A + 1;
```

New value of B will always be evaluated before A changes

Race condition

- Blocking assignment may create a simulation race condition in sequential **always**
- This design creates a race condition, between the procedural block that increments count and the procedural block that reads the value of count

```
always @(posedge clock)
  if (!resetN) count = 0;
  else count=count+1;
always @(posedge clock)
  case (state)
    HOLD: if (count == MAX)
    ...
```

  - Both procedural blocks trigger at the same time, on the positive edge of clock. Will `count` in this example can be read by the second procedural block before or after `count` is incremented?
- <u>Nonblocking assignments would resolve the race condition in the above example</u>
  - The behavior of a nonblocking assignment is that all concurrent processes will read the value of a variable before the assignment updates the value of the variable, which models the behavior of a sequential logic
- To avoid race conditions:
  - Assignments to the same variable from more than one **always** block is not allowed
  - For combinational logic use blocking assignments and for sequential logic use nonblocking assignments
  - For mixed sequential and combinational logic in the same **always** block use nonblocking assignments, but be aware of additional inferred registers/latches. You may use separate always blocks for CL and sequential logic

In-line initialization

- In-line variable initialization will be executed at simulation time zero. A simulation event will occur only if the initial value assigned to the variable is different than its current value. Thus in-line variable initialization may or may not cause simulation events at simulation time zero
- In this example, two different simulation results can occur:

```
reg RST = 0; // declare & initialize reset
always @(posedge CLK, negedge RST)
  if (!RST) count <= 0; // active low reset
  else count <= count + 1;
```

  - Case I: A simulator could activate the **always** procedural block first, prior to initializing the RST variable. In this case, at simulation time zero, when RST is initialized to 0, which results in an X to 0 transition, the **always** block will sense the event, and reset the counter
  - Case II: Simulator execute the initialization of RST before the **always** procedural block is activated. Since the initialization of RST has already occurred, the **always** wait until the next positive edge of CLK or negative edge of RST

Named blocks and identifier's scope

- A *named block* identifies a group of statements with a label by appending **:<label>** after the **begin** keyword

```
always [event_control_statement]
 begin [: name_for_block]
   [variable declaration]
   ... Procedural statements ...
end
```

- In addition to local signals defined between the **module...endmodule** keywords, local signals including include **reg** type variables, **integers**, and parameters, can be declared in the named blocks. They are only accessible within the blocks in which they are declared and hence, another variable with the same name outside of this scope can be declared and used
- An identifier's *scope* is the range of the Verilog description over which the identifier is known. Identifiers can be defined within four entities: modules, named blocks, tasks, and functions. Each of these entities defines the *local scope* of the identifier. Within a local scope, there may only be one identifier with the given name
- Module names are known globally across the whole design
- Identifiers for modules, tasks, functions, and named **begin-end** blocks are allowed to be *forward referenced*, i.e., they may be used before they have been defined

Understanding blocking and non-blocking statements

- The block label creates a hierarchy scope where using *hierarchical referencing* the signals (variables or net data types) declared within a named block can be accessed from anywhere in other modules for verification purposes (typically used in testbenches and is not synthesizable). A hierarchical reference can uniquely specify a signal name using a list of identifiers separated by periods (".")
  - For example, the first identifier can be a module name, then each succeeding identifier specifies the named scope (e.g., named **always** block) within which to continue searching downward. The last identifier specifies the name of the signal. For example, if you instantiated "**UUT**" in your testbench "**tb**", you can access the signal "x" in "**UUT**" as **UUT.x** from your testbench

```
`timescale 1ns / 1ns
module tb_example;
reg CLK,RST;
reg [2:0] c;
reg [3:0] a;
integer i;
// Clock signal
parameter ClockPeriod = 10;
initial CLK = 0;
always #(ClockPeriod / 2)
    CLK = ~CLK;
initial begin
  RST = 1;
  @(posedge CLK) RST = 1;
  @(posedge CLK) RST = 0;
end
```

```
always @(posedge CLK) begin: always_block
reg [2:0] b;
  if (RST) begin
   b = 3'b010;    a <= 4'b0001; c <= 3'b011;
  end
  else begin
     a <= a + 4'b0001;
     for (i=0; i<3; i=i+1) begin
        if (b == 3'b101) a <= a + 4'b0010;
        b = b + a;
        c <= b;
     end
  end
end
endmodule
```

$display *format specifiers*

- Using the *format specifier* **"%0d"** will print a decimal number without leading zeros or spaces. This may be used with **h**, **d**, and **o** also. The formats **%nd**, **%nb** etc. would give exactly n spaces for the number instead of the space needed

```
`timescale 1ns / 1ns
module tb_adder;
    reg [3:0] a, b; // Inputs
    wire [3:0] sum; // Outputs

    // Instantiate the Design Under Test (DUT)
    adder uut (.a(a), .b(b), .sum(sum));

    initial begin: stimuliBlock //stimuliBlock is the initial block label
       reg [7:0] i;
       a = 0;b = 0;
         #10 $display("Starting test ...");
         for (i=0; i<256; i=i+1) begin
           {a,b} = i;
           #10 $display("a b = %h %h, sum = %h", a, b, sum);
           if (sum != a + b) $display("Error, sum should be %h, is %h", a+b, sum);
         end
    end
endmodule
```

```
module adder (input [3 : 0] a, b,
              output [3 : 0] sum);
   assign sum = a + b;
endmodule
```

$display *system task and time system functions*

- The system tasks **$display** can be used in an **initial** or **always** statement for displaying formatted messages on the screen during a simulation
- **$display** always prints a newline character at the end of its execution
- Format specifiers: **%d** (decimal), **%h** (hexadecimal), **%b** (binary), **%c** (character), **%s** (string) and **%t** (time), **%m** (hierarchical name)

```
initial begin
  $display($time, " << Starting the Simulation >>");
  CLK = 1'b0;      // at time 0
  RST = 1;         // reset is active
  #20 RST = 1'b0;  //deactivate reset after 20 time units
end
always @* // This statement is always executing a $display statement
  $display("At %d: a=%b b=%b sum=%b", $time, a, b, sum);
```

- Time system functions **$time**, **$stime**, and **$realtime** return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively
- By appending **b**, **h**, **o** to the task name (**$displayb**), the default format is changed to binary, octal, or hexadecimal, respectively
- Two adjacent commas „ will print a single space. Other special characters may be printed with escape sequences: **\n** is the new line character, **\t** is the tab character, **** is the \ character, and **\"** is the " character. For instance: ("Hello world\n"); will print the quoted string with two newline characters

repeat *procedural statement*

- The **repeat** statement is also synthesizable and executes a statement or block of statements a specified (fixed) number of times specified by a numerical value or the value of an expression

```
repeat (numerical value or an expression)[begin]
    ... Procedural statements ...
[end]
```

- If the control expression evaluates to **0**, **X** or **Z**, no loop iteration is executed
- Unlike **for** loop, **repeat** loop has no index variable. The **for** loop is more flexible and gives access to the loop variable for control of the end-of-loop-condition
- **for**, **repeat**, and **while** are synthesizable loop constructs

```
module testbench;
reg CLK;
  Design UUT (.CLK(CLK));
  always #5 CLK = ~CLK;
  initial begin
    CLK = 0;
    repeat (5) @(posedge CLK);
    $finish;
  end
endmodule
```

for *statement*

for *statement*

`for` statement

- Verilog loop statements include `repeat`, `for`, `while`, and `forever`. `for` loops repeatedly execute a statement or a block of statements a given number of times

```
for (index = init; index </<=/>/>= limit; index = index +/- step)
[begin]
    ... Procedural statements ...
[end]
```

- The loop counter `index`, which is used to control the for loop, must be declared prior to the loop (usually type `integer`, which is synthesizable, or a sized `reg` type variable). `index` is initialized once at the beginning of the loop. The loop counter is updated after every execution of the body of the loop and before the next check. The second expression is executed before each iteration of the loop to determine if the body of the loop's statements should be executed. Execution stays in the loop while the second expression is TRUE. The `step` variable and the value of the loop counter `limit` are determined once at the beginning of the execution of the loop
- If the loop contains only one statement, the `begin ... end` statements may be omitted
- Synthesis tool should be able to statically determine the value of `limit`

`for` statement

- The `for` loop can be used for repetitive of a combinational logic
- Synthesis tool unrolls the for loops and implement repeated hardware structures
- In this example, each iteration of the loop specifies a different `xor` gate indexed by the loop variable `i`. Thus, eight `xor` gates are instantiated

```
module Xor8 (output reg [1:8] xout,
             input [1:8] xin1, xin2);
reg [1:8] i;
always @*
  for (i = 1; i <= 8; i = i + 1)
    xout[i] = xin1[i] ^ xin2[i];
endmodule
```

```
module parityGen (parameter WL=8)
                 (input [WL-1:0] in,
                  output reg par);
integer ctr;
  always @(in) begin
   par = 0;
   //par = ^in;
   for ( ctr=0; ctr < WL; ctr=ctr+1)
     par = par ^ in[ctr];
  end
endmodule
```

```
module parity #(parameter WIDTH = 2)
               (input [WIDTH-1 : 0] in,
                output reg p);
  always @(in) begin: loop
    integer i;
    reg parity = 0;
    for (i = 0; i < WIDTH; i = i + 1)
      parity = parity ^ in[i];
    p = parity;
  end
endmodule
```


Parameterizable modules using `for` *loops*

- A parameterizable decoder is modeled by setting all the output bits to 0 and then changing the appropriate output bit to 1 based on the input value

```
module decoder #(parameter N = 3)
                (input [N-1:0] in,
                 output reg [2**N-1:0] out);
  always @* begin
     out = 0;
     out[in] = 1;
  end
endmodule
```

```
module decoder #(parameter N=8, log2N=3)
         (input [log2N-1:0] in,
          output reg [N-1:0] out);
integer i;
  always @(in) begin
    for(i=0;i<N;i=i+1)
      out[i] = (in == i);
    end
endmodule
```

```
module adder #(parameter WIDTH=8)
              ( input [WIDTH-1:0] a,b,
                output cout,
                output [WIDTH-1] sum );
wire [WIDTH:0] carry;
  assign carry[0] = 1'b0;
  assign cout = carry[WIDTH]
  always @* begin
      carry[0]=1'b[0];
      for (i=0;i<WIDTH; i=i+1)
         {carry[i+1],sum[i]} = a[i] + b[i] + carry[i];
  end
endmodule
```

```
module decoder #(parameter N=8, log2N=3)
                (input [log2N-1:0] in,
                 output reg [N-1:0] out);
always @(in) begin
    out = 0;
    out[in] = 1'b1;
  end
endmodule
```

Using `for` *loops in testbenches and* `disable` *statement*

- Using a `for` loop to apply the testvectors to the design under test makes the testbench more compact
- The `disable` statement terminates any named blocks (using `begin` and `end`), or any loop statements and passes control to the next statement following the block. The syntax is `disable block_name;`
- Note that the loop counter is local to the main block and the same variable can be declared in another `always` block

```
always @* begin: main // named block
integer  i; // local variable
integer firstOne=-1;
 for (i=0;i<63;i=i+1) begin: pass
   if (data[i]) begin
     firstOne=I;
     disable main;
   end
 end
end
```

```
module tb_DUT;
wire [7:0] response;
reg [7:0] stimulus, x;
reg CLK;
integer i;

  DUT u1 (response, stimulus, CLK);
  initial CLK = 0;
  always begin
    #10 CLK = 1;
    #10 CLK = 0;
  end

  initial begin
    for (i = 0; i <= 255; i = i + 1) begin
      @(posedge clk) stimulus = i;
      x = U1.LBL.x;//hierarchical referencing
    end
    #20 $finish;
  end
endmodule
```

- Avoid `disable` statement in `always` blocks

`while` procedural statement

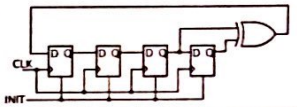
- A `while` loop executes the procedural statements of the loop as long as a loop control expression is true
  - If the loop contains only one statement, the `begin ... end` may be omitted
  - The control expression is tested at the beginning of each pass through the loop
  - It is possible that a while loop might not execute at all
- The `while` expression must be updated as part of the loop statement execution
- The `while` loop is synthesizable only if loop termination can be computed statically at elaboration time

```
while (expression) [begin]
    ... Procedural statements ...
[end]
```

```
//Leading one detector
reg [4:0] loc;
reg [31:0] data;

always @* begin
   loc = 31;
   while (data[loc] != 1) begin
       loc = loc - 1;
   end
end
```

Linear feedback shift register

- Assignment: Describe the following sequential circuit in Verilog. The flip-flops are initialized using an asynchronous INIT signal with a non-zero SEED value of "0101" (i.e., the left most flip-flop is reset to zero, the second flip-flop from the left is preset to 1, the third is reset to zero and the right most flip-flop is preset to 1). In every clock cycle, the two lower bit positions are xored and the result is shifted into the most significant bit position of the register and the contents of the flip-flops are shifted to the right one bit position. The output of this circuit is the contents of the flip-flops.




```
module dff_arst (input d, CLK, RST,
                 output reg q);
  always @(posedge CLK or posedge RST)
    if (RST)  q <= 1'b0;
    else      q <= d;
endmodule
```

```
module dff_aprst (input d, CLK, PRST,
                  output reg q);
  always @(posedge CLK or posedge PRST)
    if (PRST)  q <= 1'b1;
    else       q <= d;
endmodule
```

```
module LFSR_struct_dataflow (input CLK, INIT,
                             output [3:0] q);
wire [4:0] w;
  xor X00 (w[4], w[1], w[0]); // the same as assign w[4]=w[1]^w[0];
  dff_arst  D03 (.d(w[4]), .CLK(CLK), .RST(INIT),  .q(w[3]) );
  dff_aprst D02 (.d(w[3]), .CLK(CLK), .PRST(INIT), .q(w[2]) );
  dff_arst  D01 (.d(w[2]), .CLK(CLK), .RST(INIT),  .q(w[1]) );
  dff_aprst D00 (.d(w[1]), .CLK(CLK), .PRST(INIT), .q(w[0]) );
  assign q = w[3:0];
endmodule
```

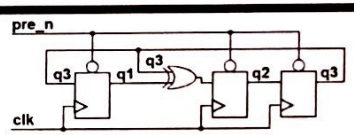

Linear feedback shift register

```
module LFSR_behavioral (input CLK, INIT,
                        output reg [3:0] q);
   always @(posedge CLK  or posedge INIT) begin
      if (INIT) q <= 4'b0101;
      else q <= {q[1]^q[0], q[3], q[2], q[1]};
   end
endmodule
```

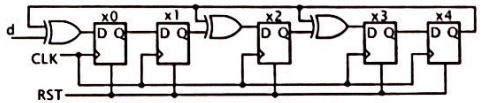
```
module tb_LFSR;
  reg CLK, INIT; //inputs
  wire [3:0] out1, out2; // outputs
        // Clock generation
        parameter HalfClkPeriod = 5;
        integer ClkPeriod = HalfClkPeriod*2;
                initial begin : ClockGenerator
                CLK = 1'b0;
                forever #HalfClkPeriod CLK = ~CLK;
        end
        // Instantiate the Unit Under Test (UUT)
        LFSR_struct_dataflow DUT0 (.CLK  (CLK), .INIT (INIT), .q(out1));
        LFSR_behavioral DUT1 (.CLK  (CLK), .INIT (INIT), .q (out2));
        initial begin
          // Reset for two clock cycles
          INIT = 1'b1; @(posedge CLK);
          @(posedge CLK) INIT = 1'b0;
          repeat (10) @(posedge CLK);
          $finish; // exit simulation
        end
endmodule
```

Linear feedback shift register

```
module lfsrn1 (output reg q3,
               input clk, pre_n);
reg q2, q1;
wire n1;
assign n1 = q1 ^ q3;
always @(posedge clk or negedge pre_n)
   if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
   end
   else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
   end
endmodule
```




```
module lfsrn2 (output reg q3,
               input clk, pre_n);
reg q2, q1;

always @(posedge clk or negedge pre_n)
  if (!pre_n) {q3,q2,q1} <= 3'b111;
  else {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

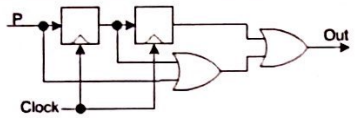



```
module LFSR (input d,
             input CLK, RST,
             output reg [0:4] x);
always @(posedge CLK)
   if (RST) x <= 0;
  else begin
   x <= {{d^x[4]}, x[0], {x[1]^x[4]}, {x[2]^x[4]}, x[3]};
endmodule
```

```
`timescale 1ns / 1ns
module tb_LFSR;
reg CLK, RST, d;
wire [0:4] x;
// Clock signal
parameter ClockPeriod = 10;
initial CLK = 0;
always #(ClockPeriod / 2) CLK = ~CLK;
// Synchronous reset
initial begin
  RST = 1; @(posedge CLK) RST = 0; d = 1;
  @(posedge CLK) d = 1;
  @(posedge CLK) d = 0;
  @(posedge CLK) d = 0;
  @(posedge CLK) d = 1;
  @(posedge CLK) d = 1;
  @(posedge CLK) d = 1;
  @(posedge CLK) d = 0;
  @(posedge CLK);
  @(posedge CLK);
  $finish;
 end
LFSR U00(.d(d),.CLK(CLK),.RST(RST),.x (x));
endmodule
```

Examples

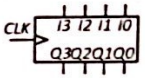
```
module PulseWidener (input P , Clock,
                     output Out);
reg q1 , q2 ;
  always @(posedge Clock) begin
     q1 <= P ;
     q2 <= q1 ;
  end
  assign Out = P | q1 | q2 ;
endmodule
```




```
module dff_async_rst (input CLK, RST,
                      input d,
                      output reg q);
  // DFF with active-low asynchronous reset
  always @(posedge CLK or negedge RST)
      if (~RST)  q <= 1'b0;
      else       q <= d;

endmodule
module reg3Bits (input CLK, RST,
                 input [2:0] d,
                 output [2:0] q);
   dff_async_rst dff00 (.CLK(CLK), .RST(RST), .d(d[2]), .q(q[2]));
   dff_async_rst dff01 (.CLK(CLK), .RST(RST), .d(d[1]), .q(q[1]));
   dff_async_rst dff02 (.CLK(CLK), .RST(RST), .d(d[0]), .q(q[0]));
   //dff_async_rst reg_3bit[2:0] (.CLK(CLK), .RST(RST), .d(d), .q(q));
endmodule
```

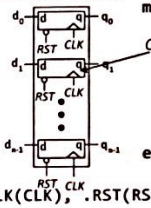
Registers

```
module Reg4bit(input [3:0] I,
               output reg [3:0] Q,
               input CLK, RST);
   always @(posedge CLK) begin
      if (RST == 1)  Q <= 0;//active-high reset
      else           Q <= I;
   end
endmodule
```




```
// 4-bit register with asynchronous preset and clock enable
module reg4 (input CLK, CE, PRE,
             input [3:0] D,
             output reg [3:0] Q);
   always @(posedge CLK or posedge PRE) begin
      if (PRE) Q <= 4'b1111;
      else if (CE) Q <= D;
   end
endmodule
```

```
module dff (output reg q,
            input d, CLK, RST);
  always @(posedge CLK)
     if (!RST)  q <= 1'b0;//active-low reset
     else       q <= d;
endmodule
```




```
//Parameterized register
module register  #(parameter WIDTH=8)
                  (output reg [WIDTH-1:0] q,
                   input [WIDTH-1:0] d,
                   input CLK, RST);
  always @(posedge CLK, negedge RST)
    if (!RST) q <= 0;
    else q <= d;
endmodule
```

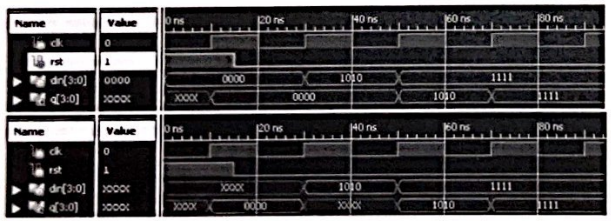
```
module RegN #(parameter N=16)
             (output [N-1:0] q,
              input [N-1:0] d,
              input CLK, RST);
 dff i[N-1:0] (.q(q), .d(d), .CLK(CLK), .RST(RST));
endmodule
```


Testbench example

```
module register #(parameter  WIDTH = 8)
                 (input [WIDTH-1:0] din, input CLK, RST,
                  output reg [WIDTH-1:0] q);
   always @(posedge CLK)
     if (RST)   q <= 0;
     else q <= din;
endmodule
```

```
`timescale 1ns/1ns
module testbench;
 reg  [3:0] din;
 reg  CLK, RST;
 wire [3:0] q;
 // DUT instantiation
 register  #(.WIDTH(4))
           R00 (.din(din),.CLK(CLK),.RST(RST), .q(q));
    //Generating clock signal
    parameter ClockPeriod = 20;
    initial CLK = 0; // initialize the Clock signal
    always #(ClockPeriod / 2) CLK = ~CLK;
 initial begin  //Generating test vectors
  RST = 1; din = 0;
  @(posedge CLK);
  #5 RST = 0;
  din = 0;
  @(posedge CLK); din = 4'b1010;
  @(posedge CLK); din = 4'b1111;
 end
endmodule
```

Simulation results; the first din=0 is not required

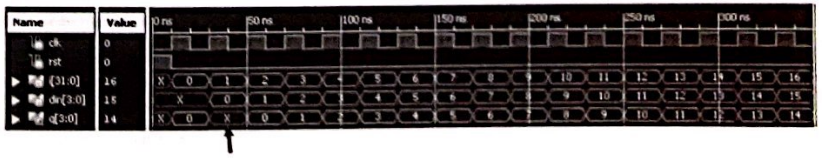


If we remove the first and second din=0

- Note that q is updated only on rising clock edges. Hence q is unknown until first clock edge

Testbench example

```
`timescale 1ns/1ns
module testbench;
 reg  [3:0] din;
 reg  CLK, RST;
 wire [3:0] q;
 integer i;
 // DUT instantiation
 register  #(.WIDTH(4)) R00 (.din(din),.CLK(CLK),.RST(RST), .q(q));
 // generating the clock signal
     parameter ClockPeriod = 20;
     initial CLK = 0; // initialize the Clock signal
     always #(ClockPeriod / 2) CLK = ~CLK;
 // test vectors
  initial begin
    RST = 1; @(posedge CLK) RST = 0;
    for (i = 0; i <= 15; i = i + 1)
      @(posedge CLK) din = i;
    #ClockPeriod $finish;
  end
endmodule
```

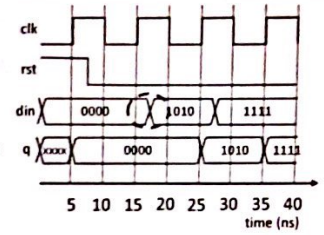


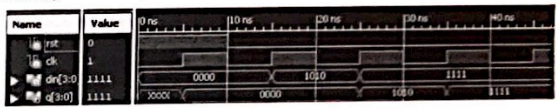
Testbench example

```
`timescale 1ns/1ns
module testbench;
 reg  [3:0] din;
 reg  CLK, RST;
 wire [3:0] q;
 integer i;
 // DUT instantiation
 register  #(.WIDTH(4)) R00 (.din(din),.CLK(CLK),.RST(RST), .q(q));
 //Generating clock signal
     parameter ClockPeriod = 20;
     initial CLK = 0; // initialize the Clock signal
     always #(ClockPeriod / 2) CLK = ~CLK;
 //Generating test vectors
  initial begin
    RST = 1; @(posedge CLK) RST = 0;
    for (i = 0; i <= 15; i = i + 1) begin
     din = i; @(posedge CLK);
   end
   #ClockPeriod $finish;
  end
endmodule
```

Register testbench

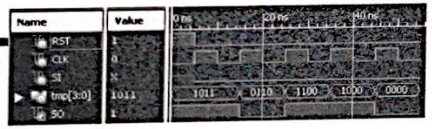
```
`timescale 1ns/1ns
module Testbench;
   reg [3:0] din;
   reg CLK, RST;
   wire [3:0] q;

   // DUT instantiation
   register R00 (.din(din),.CLK(CLK),.RST(RST),.q(q));

   //Generating clock signal
   always begin
      CLK = 0;
      #5; CLK = 1;
      #5;
   end

   //Generating test vectors
   initial begin
      RST = 1; din = 4'b0000;
      @(posedge CLK); #5 RST = 0;
      din = 4'b0000;
      @(posedge CLK); din = 4'b1010;
      @(posedge CLK); din = 4'b1111;
   end
endmodule
```



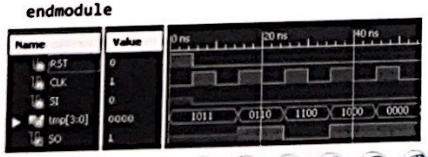


Shift registers

```
//4-bit shift-left register with async.Reset,
//serial-in and serial-out
module shRegister (input CLK, SI, RST,
                   output SO);
reg [3:0] tmp;
   always @(posedge CLK or posedge RST) begin
         if (RST) tmp <= 4'b1011;
         else     tmp <= {tmp[2:0], SI};
   end
   assign SO = tmp[3];
endmodule

module shRegister2 (input CLK, SI, RST,
                    output reg SO);
reg [3:0] tmp;
   always @(posedge CLK or posedge RST) begin
         if (RST) tmp <= 4'b1011;
         else begin
            tmp <= {tmp[2:0], SI};
            SO  <= tmp[3];
           end
   end
endmodule
```



```
`timescale 1ns / 1ns
module testbench;
reg CLK, RST;
reg SI;
wire SO;
   parameter ClockPeriod = 10;
   initial CLK = 0;
   always #(ClockPeriod / 2) CLK = ~CLK;
   initial begin
         RST=1;
         @(posedge CLK) RST=0; SI = 0;
         @(posedge CLK);@(posedge CLK);
         @(posedge CLK);@(posedge CLK);
         @(posedge CLK);$finish;
   end
   shRegister2 U00 (.CLK(CLK), .RST(RST),
   .SI(SI), .SO(SO));
endmodule
```



Shifters and counters

```
// a parametric shift register
module Shifter #(parameter WIDTH = 8)
                (input CLK, In, Load,
                 input [WIDTH-1:0] Data,
                 output Out);
reg [WIDTH-1:0] ShiftReg;
   always @(posedge CLK)
    if (Load) ShiftReg <= Data;
    else ShiftReg <= {ShiftReg[WIDTH-2:0], In};
  assign Out = ShiftReg[WIDTH-1];
endmodule
```

- Counters count the number of occurrences of an event

```
// 8-bit up counter with load and
// asynchronous reset
module count_load #(parameter WIDTH = 8)
                   (input LOAD, CLK, RST,
                    input [Width-1:0] data,
                    output [Width-1:0] out);
   reg [Width-1:0] tmp;
      always @(posedge CLK or negedge RST)
         if(!RST)    tmp <= 0;
         else if(LOAD) tmp <= data;
              else tmp <= tmp + 1;
      assign out = tmp;
   endmodule
```

```
//8-bit up counter with count enable and asynchronous
   reset
module counter #(parameter WIDTH = 8)
                (input EN, CLK, RST,
                 output [Width-1:0] out);
reg [Width-1:0] tmp;
   always @(posedge CLK or negedge RST)
         if(!RST)   tmp <= 0;
         else if(EN) tmp <= tmp + 1;
    assign out = tmp;
endmodule
```

defparam

Localparm and `include compiler directive

`localparm name=value;` can be used for defining local

defparam

- **Defparam** overrides the default parameter values at compile time. The constant expression can contain a previously declared parameter

```
defparam [ hierarchy_path. ] parameter_name = constant_expression;
```

- **defparam** parameter values can be changed at compile time when a parameterized module containing parameters is instanced. Modifying parameter values during simulation is not allowed
- **defparam** uses *hierarchical naming* to apply the change. The **defparam** statement can be placed before or after the module instance or anywhere else in the file. Using the parameters makes it clear at the instantiation site that new values are overriding defaults
- If multiple **defparams** are used for a parameter, the parameter takes the value of the last **defparam** statement
- The **defparams** can be declared in one place in a separate file. This allows the system to be parameterized by various **defparam** files rather than by reediting the description of several modules

```
module TestShifter;
  ...
  defparam U1.WIDTH = 10;
  Shifter U1 (...) // the same as Shifter #(10) U1 (…)
endmodule
```

Localparm *and* `include *compiler directive*

- **localparam [signed]** [range] name=value; can be used for defining local constant values. The constant value can be defined based on other constants
- Unlike a **parameter**, a **localparam** cannot be modified by parameter redefinition nor can a **localparam** be redefined by a **defparam** statement
- **localparam** is mainly used to generate local parameter values based on other **parameters** or **defparam** statements while protecting the **localparams** from accidental or incorrect redefinition. Note that since a local parameter assignment expression can contain a parameter (which can be overridden), it can be indirectly overridden
- Verilog does not have a shared (global) declaration space. A declaration that is required in multiple modules must be declared in each module
  - Redundant declarations can lead to errors if a declaration is changed in one design but not in another module that is supposed to have the same declaration
- **`include** "filename"; includes the content of a text file at the point in the module where the compiler directive is. It can be written anywhere in the code
- Since the definitions in macros.vh are placed into a global namespace, it makes sense to never include or redefine those definitions again

```
`include "macros.vh"
module top
  ...
endmodule
```


Counters

```
//8-bit counter with load, count enable, and asynchronous reset
module count_load (out, cout, data, LOAD, CLK, EN, RST);
parameter Width = 8;
input LOAD, CLK, EN, RST;
input [Width-1:0] data;
output cout; // cout=1 when all out bits equal 1
output [Width-1:0] out;
reg [Width-1:0] tmp;
    always @(posedge CLK or negedge RST)
       if(!RST)    tmp <= 0;
       else if(LOAD) tmp <= data;
            else if(EN) tmp <= tmp + 1;
     assign cout = &tmp;
     assign out  = tmp;
endmodule

module tb_counter;
reg CLK, RESET, EN; //inputs
wire [15:0] count;  //outputs
// Clock generation
  initial   CLK = 1'b0;
  always #5 CLK <= ~CLK;
// Instantiate the Unit Under Test (UUT)
  counter  UUT (.clk (CLK), .reset (RESET), .en (EN), .out (count));
// applying test inputs
  initial begin
    RESET  = 0;
    @(posedge CLK) RESET  = 1; EN = 1;
  end
initial
    #50 $finish;
endmodule
```

```
//A ring counter
module ring_counter (input CLK, init,
                     output reg [7:0] count);
always @ (posedge CLK) begin
    if (init)
      count <= 8'b10000000;
    else
      count <= {count[6:0], count[7]};
endmodule

// A ring counter
module ring_counter (input CLK, init,
                     output reg [7:0] count);
  always @ (posedge CLK) begin
    if (init)
      count = 8'b10000000;
    else begin
      count <= count << 1;
      count[0] <= count[7];
    end
  end
endmodule
```

Mod-m counters

- A *mod-m counter* counts from 0 to *m*-1 and wraps around

```
module mod_M_counter  #(parameter M=9, // Limit
                        N=4) // counter width. N should be equal to ⌈log2 M⌉
                      (input CLK, RST,
                       output reg [N-1:0] ctr) ;
    always @(posedge CLK)
     if (RST) ctr <= 0;
     else begin
       if (ctr ==M-1) ctr <=0;
       else ctr <= ctr +1;
     end
endmodule
```

```
module tb_mod_M_counter;
parameter M=9, N=4;
reg CLK, RST;
wire [N-1:0] ctr;
integer i;
 mod_M_counter  #(.M(9), .N(4))
                U00 (.CLK(CLK), .RST(RST), .ctr(ctr)) ;
 parameter ClockPeriod = 10;
 initial CLK = 0;
 always #(ClockPeriod / 2) CLK = ~CLK;
 initial begin
   RST = 1; @(posedge CLK) RST = 0;
   for (i = 0; i <= 19; i = i + 1)
      @(posedge CLK);
   $finish;
 end
endmodule
```

Procedural statements: `case` statement

- The **case** statement supports conditional execution of procedural statements. It allows one of multiple options selected by comparing the *case expression* with a list of *case selection items* (or labels)
- A case selection item is an expression or a comma-separated list of expressions on the left of a colon, or the reserved word **default**
  - Where a label is a comma-separated list of two or more expressions, the label is matched if the case expression matches any one of the expressions in a label
- If more than one statement is to be executed for a particular case selection item, the statements must be enclosed in a **begin-end** block
- The case expression width should match the width of the case selection items
- The **default** case item defines the remaining unspecified case, i.e., if all comparisons fail and the **default** section is given, then its statements are executed. Otherwise none of the statements of the case choices will be executed. At most one **default** statement may be included

```
case (expression)
label1: begin
        ... Procedural statements ...
        end
label2: begin
        ... Procedural statements ...
        end
        ... more case labels blocks ...
[default: begin
        ... Procedural statements ...
          end]
endcase
```

```
always
case (opcode) //case expression
  2'b00: y = a + b;
  2'b01: y = a - b;
  2'b10: y = a * b;
  2'b11: y = a / b;
endcase
```


`case` examples

```
//one-hot 3-to-8 decoder
module decoder3-to-8 (input En,
                      input [2:0] Ain,
                      output reg [7:0] Yout);
 always @* begin
   if (~En) Yout = 8'b0;
   else
    case (Ain)
     3'b000 : Yout = 8'b00000001;
     3'b001 : Yout = 8'b00000010;
     3'b010 : Yout = 8'b00000100;
     3'b011 : Yout = 8'b00001000;
     3'b100 : Yout = 8'b00010000;
     3'b101 : Yout = 8'b00100000;
     3'b110 : Yout = 8'b01000000;
     3'b111 : Yout = 8'b10000000;
     default : Yout = 8'b00000000;
    endcase
end
endmodule
```

```
//one-cold 2-to-4 decoder
module decoder2to4(input  A, B,
                   input  EN,
                   output reg Y3, Y2, Y1, Y0);
    always @* begin
      if (EN == 1'b1) // the same as if (EN)
        case ( {A,B} )
          2'b00:   {Y3,Y2,Y1,Y0} = 4'b1110;
          2'b01:   {Y3,Y2,Y1,Y0} = 4'b1101;
          2'b10:   {Y3,Y2,Y1,Y0} = 4'b1011;
          2'b11:   {Y3,Y2,Y1,Y0} = 4'b0111;
          default: {Y3,Y2,Y1,Y0} = 4'bxxxx;
                                // the same as 4'bx;
        endcase
      else         {Y3,Y2,Y1,Y0} = 4'b1111;
    end

endmodule
```

- When specifying don't care bits, it is easy to inadvertently specify multiple case selection items that could be true at the same time
- In case of *overlapping case items*, where a case expression value could match more than one case selection item, then only the first matching branch is executed

```
case (select)
  2'b1?: aluout = a + b;
  2'b0?: aluout = a - b;
  2'b?0: aluout = a * b;
  2'b?1: aluout = a / b;
endcase
```

Multiplexers using `case` statements

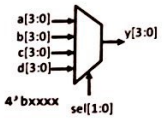
- That **case** labels need not be mutually exclusive. The case labels are evaluated in the order in which they are listed (the process is the same as in a series of **if...else...if** decisions, but the **case** statement is easier to read than a long sequence of **if … else** statements). Therefore, **case** statements are priority-encoded. If the selection items are mutually exclusive, the Synthesis tool optimizes out the additional logic required for priority encoding the selection decisions

```
module mux2to1(input a, b, sel,
               output reg out,
               output outbar);
always @(a,b,sel)
    if(sel) out = a;
    else out = b;
assign outbar = ~out;
endmodule
```

```
module mux2to1(input a, b, sel,
               output reg out,
               output outbar);
always @(a,b,sel)
    case(sel)
    1'b1: out = a;
    1'b0: out = b;
    endcase
assign outbar = ~out;
endmodule
```

```
module mux4x1_1bit (input [3:0]in,
                    input [1:0] sel,
                    output reg out);
  always @*
    case (sel)
      2'b00: out = in[0];
      2'b01: out = in[1];
      2'b10: out = in[2];
      2'b11: out = in[3];
    endcase
endmodule
```

```
module mux4x1_4bits (input [3:0] a, b, c, d,
                     input [1:0] sel,
                     output reg [3:0] y);
  always @*
    case (sel)
      0: y = a;
      1: y = b;
      2: y = c;
      3: y = d;
      default: y = 4'bx; // the same as 4'bxxxx
    endcase
endmodule
```

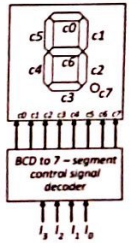



```
module mux4x1_1bit(output reg out,
                   input [3:0] in,
                   input [1:0] sel);
  always @*
    case (sel)
      0: out = in[0];
      1: out = in[1];
      2: out = in[2];
      3: out = in[3];
    endcase
endmodule
```

Seven-segment LED display

```
module bcdLED (input [3:0] I,
               output reg [7:0] c);
 always @*
    case(I)
       0: c  = {1'b1,7'b1000000};
       1: c  = {1'b1,7'b1111001};
       2: c  = {1'b1,7'b0100100};
       3: c  = {1'b1,7'b0110000};
       4: c  = {1'b1,7'b0011001};
       5: c  = {1'b1,7'b0010010};
       6: c  = {1'b1,7'b0000010};
       7: c  = {1'b1,7'b1111000};
       8: c  = {1'b1,7'b0000000};
       9: c  = {1'b1,7'b0011000};
       default: c  = {1'b0, 7'b111111};
    endcase
endmodule
```

```
`timescale 1ns / 1ns
module tb_bcdLED;
 reg [3:0] I;
 wire [7:0] c;
 integer num;

 bcdLED U00 (.I (I), .c(c));
 initial  begin
   I = 0;
   for (num = 1; num <= 15; num = i + 1)
     #5 I = i;
   #5 $finish;
 end
endmodule
```



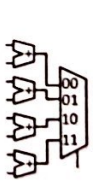
- Resource sharing example:

```
if (a==2'b00)
    c = a + b;
else if(a==2'b01)
    c = d + e;
else if(a==2'b10)
    c = w + y;
else
   c = x + z;
```

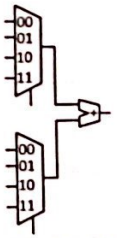
OR

```
case(a)
     2'b00: c = a + b;
     2'b01: c = d + e;
     2'b10: c = w + y;
     2'b11: c = x + z;
endcase
```

Implies

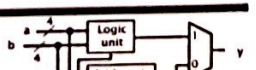


With Resource Sharing



Arithmetic logic unit

```
module Arithmetic(input signed [3:0] a, b, [2:0] sel,
                  output reg signed [7:0] y0);
```



Universal shift register

```
module UniversalShiftReg #(parameter N = 8)
      (input CLK, RST, input [3 : 0] op, input sinR, sinL,
       input [N - 1 : 0] din,
```

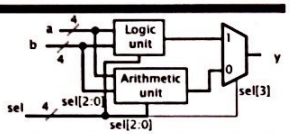
executed

Arithmetic logic unit

```
module Arithmetic(input signed [3:0] a, b, [2:0] sel,
                  output reg signed [7:0] y0);
always @* begin
 case (sel)
 3'b000: y0 = (a + b);
 3'b001: y0 = (a - b);
 ...
 3'b111: y0 = (a * b);
 default: y0 = 8'bX;
 endcase
 end
endmodule
```

```
module Multiplexer(input signed [7:0] y1,y0,
                   input sel,
                   output reg signed [7:0] y);
  always @*
   if (sel) y = y1;
   else y = y0;
endmodule
```

```
module tb_ALU;
    reg [3:0] a,b,sel;
    wire [7:0] y;
    ALU ALU00 (.a(a), .b(b), .sel(sel), .y(y));
    initial begin : Selector
      sel = 0;
      forever #10 sel = sel + 1;
    end
    initial begin
       a = 6; b = -5;
       #100;
       disable Selector;
       #10; $finish;
    end
endmodule
```




```
module Logic(input signed [3:0] a, b, [2:0] sel,
             output signed [7:0] y1);
    reg signed [3:0] i;
    always @* begin
                case (sel)
                3'b000: i = ~a;
                3'b001: i = ~b;
                3'b010: i = (a & b);
                ...
                default: i = 4'bX;
                endcase
    end
    assign y1 = {{4{(1'b0)}}, i[3:0]};
endmodule
```

```
module ALU(input signed [3:0] a, b, [3:0] sel,
           output signed [7:0] y);
wire [7:0] y0;
wire [7:0] y1;
Arithmetic A00 (.a(a), .b(b), .sel(sel[2:0]), .y0(y0));
Logic L00 (.a(a), .b(b), .sel(sel[2:0]), .y1(y1));
Multiplexer M00 (.y1(y1), .y0(y0), .sel(sel[3]), .y(y));
endmodule
```

Universal shift register

```
module UniversalShiftReg #(parameter N = 8)
       (input CLK, RST, input [3 : 0] op, input sinR, sinL,
        input [N - 1 : 0] din,
        output reg soutL, soutR);
reg [N - 1 : 0] shiftReg;
localparam HOLD=4'd0, LOAD=4'd1, SerR=4'd2, SerL=4'd3,
           SLL=4'd4, SRL=4'd5, SRA=4'd6, RotR=4'd7, RotL=4'd8;
  always @(posedge CLK) begin
    if(RST) begin
      shiftReg <= 0; soutL <= 0; soutR <= 0;
    end
    else begin
      case(op)
      HOLD: shiftReg <= shiftReg;
      LOAD: shiftReg <= din;
      SerR: begin
            soutL <= shiftReg[N-1];
            shiftReg <= {shiftReg[N - 2 : 0], sinR};
            end
      SerL: begin
            soutR <= shiftReg[0];
            shiftReg <= {sinL, shiftReg[N-1 : 1]};
            end
      SLL:  shiftReg <= {shiftReg[N - 2 : 0], 1'b0};
      SRL:  shiftReg <= {1'b0, shiftReg[N-1 : 1]};
      SRA:  shiftReg <= {shiftReg[N-1], shiftReg[N-1 : 1]};
      RotR: shiftReg <= {shiftReg[0], shiftReg[N - 1 : 1]};
      RotL: shiftReg <= {shiftReg[N - 2 : 0], shiftReg[N-1]};
      endcase
    end
  end
endmodule
```


Reversed case statements and overlapping case items

- In a typical case statement, a variable or expression is specified as the case expression, and case labels are values of the case expression
- In a *reversed case statement*, the case expression is specified as the literal value to be matched and the case selection items are the variables

```
module priority_encoder (input [0:3] in,
                         output reg [0:1] code);
always @(in)
    case (1'b1)
      in[0] : code = 2'b00;
      in[1] : code = 2'b01;
      in[2] : code = 2'd10;
      in[3] : code = 2'b11;
      default: code = 2'b00;
    endcase
endmodule
```

- Using high-level behavioral descriptions allows synthesis tool to apply various logic optimizations

```
module ALU_4bit (input [1:0] op,
                 input [3:0] a, b,
                 output reg [3:0] f);
parameter ADD=2'b00, SUB=2'b01, MUL=2'b10;
parameter DIV=2'b11;
   always @ (a or b or op)
     case (op)
           ADD : f = a + b;
           SUB : f = a - b;
           MUL : f = a * b;
           DIV : f = a / b;
           default : f=0;
     endcase
   endmodule
```

Incomplete `case` statements and latch inference

- If all of the possible values for the case expression are not specified in the case labels, or the output is not assigned in case labels, or no case choices are true, a latch will be inferred. To prevent the inference of unintentional latch use a `default` choice and assign the output
- You can (i) precede the case selection items with a default assignment to the output signal or (ii) specify all case selection items and or a default selection item with an assignment to the output signal
- A *full case statement* specifies all of the case items either explicitly or by using a `default`

```
module incomp_state_spec (input [0:1] state,
                          output reg [0:1] flag);
  always @ (state)
    case (state)
      0, 1 : flag = 2;
      3 : flag = 0;
    endcase
endmodule
```

The variable 'flag' is not assigned a value in all the branches of case. Latch is inferred

```
module comp_state_spec (input [0:1] state,
                        output reg [0:1] flag);
  always @ (state)
   flag = 0;
   case (state)
      0, 1 : flag = 2;
      3 : flag = 0;
   endcase
endmodule
```

'flag' defined for all values of curr_state. Latch is avoided

```
always @ (a,b,c,sel) begin
    out = 1'bx;
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
    endcase
end
endmodule
```

```
always @ (a,b,c,sel) begin
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        default: out = 1'bx;
    endcase
end
endmodule
```

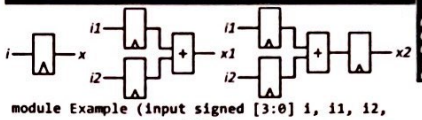
Logic X

- The case expressions may include unknown (x) and high impedance (z) bits. The comparison is done using 4-valued logic and will succeed only when each bit matches exactly with respect to the values **0**,**1**, **x**, and **z**
- Logic X is often used to model an unexpected or don't care condition
- Synthesis tool uses don't care values to optimize the logic. Also, the default assignment of X can help trap design errors in the hardware description

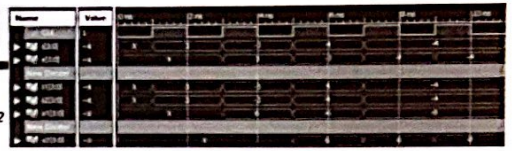
```
case (alu_ctr)
  2'b00: aluout = a + b;
  2'b01: aluout = a - b;
  2'b10: aluout = a & b;
  default: aluout = 1'bx;
endcase
```

- However, this advantage is lost after synthesis, as the post-synthesis design will only outputs 0 or 1 and not logic X values for unexpected case expression values
- If the conditions are *mutually exclusive* (i.e., parallel conditions) then a **case** statement is preferred, because it is easier to organize the parallel states of the description than a long sequence of **if** … **else** statements. If multiple conditions can occur at the same time and priority of conditions is important, use the "**if**" statement and prioritize the conditions (from highest to lowest or most often to least often) using "**else if**" for each subsequent condition. A nested **if … else** will generate a priority circuit (e.g., cascading multiplexer structure), which may have a very long delay (a lengthy mux-chain)

Synchronous design examples

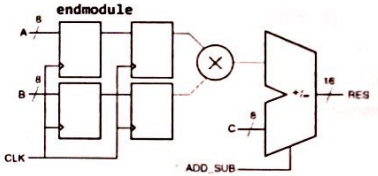



```
module Example (input signed [3:0] i, i1, i2,
                input CLK,
                output reg signed [3:0] x, x2,
                output signed [3:0] x1);
reg signed [3:0] iq1, iq2;
  always @(posedge CLK) x<=i;
  assign x1 = iq1+iq2;
  always @(posedge CLK) begin
    iq1<=i1;
    iq2<=i2;
  end
  always @(posedge CLK) begin
    x2 <= iq1+iq2;
  end
endmodule
```



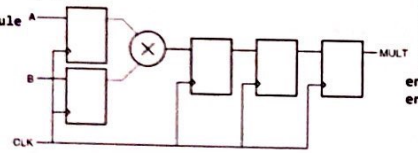
```
module tb_dataflow;
reg signed [3:0] i, i1, i2;
reg CLK;
wire [3:0] x, x1, x2;
  Example UUT (.i (i), .i1 (i1), .i2(i2),
               .CLK (CLK), .x (x), .x1(x1), .x2(x2));
  initial CLK = 0;
  always begin
    #1 CLK = 1; #1 CLK = 0;
  end
initial begin
  @(posedge CLK) begin i = 4'b0011; i1 = i; i2= i; end
  @(posedge CLK) begin i = 4'b1101; i1 = i; i2= i; end
  @(posedge CLK) begin i = 4'b0100; i1 = i; i2= i; end
  @(posedge CLK) begin i = 4'b1100; i1 = i; i2= i; end
  @(posedge CLK)
  @(posedge CLK)
  $finish;
end
endmodule
```

```
module MULADDSUB (input CLK,add_sub,
                  input [7:0] A,B,C,
                  output [15:0] RES);
reg [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
wire [15:0] mult, multaddsub;
 always @(posedge CLK) begin
    A_reg1 <= A; A_reg2 <= A_reg1;
    B_reg1 <= B; B_reg2 <= B_reg1;
 end
 assign mult = A_reg2 * B_reg2;
 assign RES = add_sub ? C + mult : C - mult;
endmodule
```

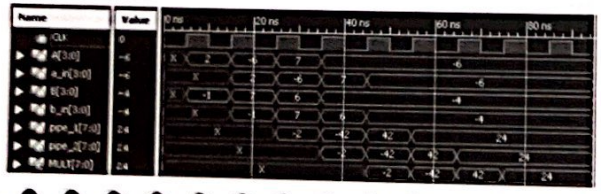


Pipelined multiplier

```
module Mult_P3 #(parameter N=16)
                (input CLK,
                 input signed [N-1:0] A, B,
                 output reg [2*N-1:0] MULT);
reg signed [N-1:0] a_in, b_in;
wire signed [2*N-1:0] mult_res;
reg signed [2*N-1:0] pipe_1, pipe_2;

 assign mult_res = a_in * b_in;
 always @(posedge CLK) begin
   a_in <= A; b_in <= B;
   pipe_1 <= mult_res;
   pipe_2 <= pipe_1;
   MULT <= pipe_2;
 end
endmodule
```

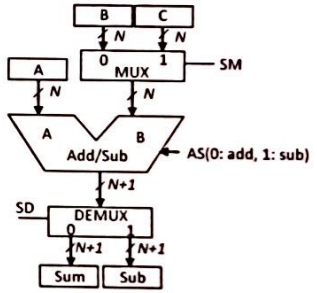



```
module tb_Mult_P3;
parameter N= 4;
reg CLK;
reg signed [N-1:0] A,B;
wire signed [2*N-1:0] MULT;

parameter ClockPeriod = 10;
initial CLK = 0; // initialize the Clock signal
always #(ClockPeriod / 2) CLK = ~CLK;

multipliers_P4  #(.N(4))U00 (.CLK(CLK),.A(A),
                             .B(B),.MULT(MULT));
initial begin
  @(posedge CLK) A = 4'b0010; B=4'b1111;
  @(posedge CLK) A = 4'b1010; B=4'b0111;
  @(posedge CLK) A = 4'b0111; B=4'b0110;
  @(posedge CLK) A = 4'b1010; B=4'b1100;
  #(ClockPeriod*6) $finish;
end
endmodule
```



Datapath example

```
module add_sub #(parameter N=4)
   (input signed [N-1:0] A,
    input signed [N-1:0] B,
    input signed [N-1:0] C,
    input SM, AS, SD,CLK,
    output reg signed [N:0] Sum,
    output reg signed [N:0] Sub);
reg signed [N-1:0] O1, A_reg, B_reg, C_reg;
reg signed [N:0] O2;
 always @* if (SM) O1=B_reg; else O1=C_reg;
 always @* if (AS) O2=A_reg+O1; else O2=A_reg-O1;
 always @(posedge CLK) begin
   if (SD) begin
     Sum <= O2;
     Sub <= 0;
   end
   else begin
     Sum <= 0;
     Sub <= O2;
   end
 end
 always @(posedge CLK) begin
   A_reg <= A;
   B_reg <= B;
   C_reg <= C;
 end
endmodule
```




```
module tb_add_sum;
parameter N = 4;
reg [N-1:0] A, B, C;
reg SM, AS, SD;
wire [N:0] Sum, Sub;
reg CLK = 0;
always begin
      #5; CLK = ~CLK;
end
add_sub #(N) UUT1 (.A(A), .B(B), .C(C), .SM(SM), .AS(AS),
                   .SD(SD), .CLK(CLK), .Sum(Sum), .Sub(Sub));
initial begin
 A = 0; B = 0; C = 0; SM = 0; AS = 0; SD = 0;
 @(posedge CLK); A = 7; B = 1; C = 0; SM = 0; AS = 0; SD = 0;
 @(posedge CLK); A = 0; B = 2; C = -1; SM = 1; AS = 1; SD = 1;
 @(posedge CLK);@(posedge CLK);$finish;
end
endmodule
```

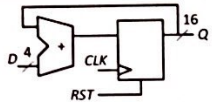
Accumulator

Multiply-accumulate unit – Behavioral level

Accumulator

```
module acc #(parameter WLin=4,
                       WLout=16)
     (input CLK, RST,
      input signed [WLin-1:0] D,
      output signed [WLout-1:0] Q);
reg signed [WLout-1:0] tmp;
always @(posedge CLK) begin
  if (RST) tmp <= 0;
  else     tmp <= tmp + D;
end
assign out = tmp;
endmodule
```

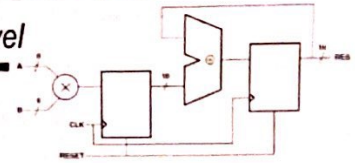
```
module acc (input CLK, RST,
            input signed [3:0] D,
            output reg signed [7:0] Q);

  always @(posedge CLK or posedge RST) begin
          if (RST) Q <= 0;
          else     Q <= Q + D;
  end
endmodule
```




```
module tb_acc;
// Inputs
reg signed [3:0] D;
reg CLK, RST;
// Outputs
wire [15:0] Q;

parameter HalfClkPeriod = 5;
integer ClkPeriod = HalfClkPeriod*2;
// Instantiate the Unit Under Test (UUT)
acc #(.WLin(4), .WLout(16))
    UUT (.CLK(CLK),.RST(RST),.D(D),.Q(Q));
initial begin : ClockGenerator
  CLK = 1'b0;
  forever #HalfClkPeriod CLK = ~CLK;
end
```

```
// Initialize Inputs
initial begin
  RST = 1;
  @(posedge CLK) RST = 0;
  D = 4'b0001;
  @(posedge CLK) in = 4'b1101;
  @(posedge CLK) in = 4'b0011;
  @(posedge CLK) in = 4'b1001;
  repeat(2) // to flush the pipeline
    @(posedge CLK) ;
  $finish; // exit simulation
end
endmodule
```

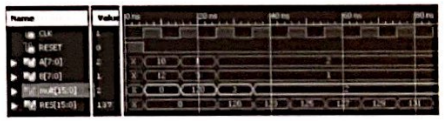
Multiply-accumulate unit – Behavioral level




```
module MAC (input CLK, RST,
            input [7:0] A,
            input [7:0] B,
            output [15:0] RES);
reg [15:0] mult, accum;
   // multiply unit
    always @(posedge CLK) begin
          if (RST) mult <= 0;
          else     mult <= A * B;
    end
    //accumulate unit
    always @(posedge CLK) begin
          if (RST) accum <= 0;
          else     accum <= accum + mult;
    end
    assign RES = accum;
endmodule
```

```
module tb_MAC;
reg CLK;
reg RESET;
reg [7:0] A, B;
wire [15:0] RES;
multiplyAccumulate UUT
  (.CLK(CLK), .RST(RESET),.A(A),.B(B),.RES(RES));
initial begin
   CLK   = 1'b0;  //initialize the clock signal
   RESET = 1'b1;
end
always #5 CLK<=~CLK;
```

```
// multiply accumulate unit using one always
always @(posedge CLK) begin
      if (RST) begin
        mult <= 0;
        accum <= 0;
      end
      else begin
        mult <= A * B;
        accum <= accum + mult;
      end
end
```

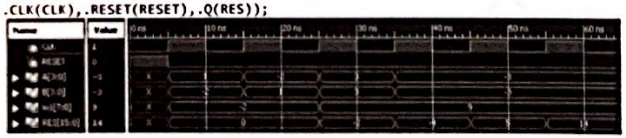



```
// applying test inputs
initial begin
  @(posedge CLK) RESET <= 0;
                 A <= 8'b0001010; B<= 8'b00001100;
  @(posedge CLK) A <= 8'b0000001; B<= 8'b00000011;
  @(posedge CLK) A <= 8'b0000010; B<= 8'b00000001;
end
endmodule
```


Multiply-accumulate unit – Structural level

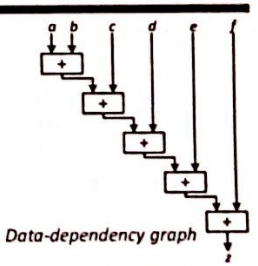
```
module sadd #(parameter NA = 8, NB = 16, NSUM = 16)
             (input signed [NA-1:0] A,
              input signed [NB-1:0] B,
              output reg signed [NSUM-1:0] Sum);
  always @* Sum = A + B;
endmodule
module smul #(parameter N=8)
             (input signed [N-1:0] A, B,
              output signed [2*N-1:0] Mul);
  assign Mul = A * B;
endmodule
module regist #(parameter N=8)
             (input signed [N-1:0] D,
              input CLK, RESET,
              output reg signed [N-1:0] Q);
  always  @(posedge CLK) if (RESET) Q<=0; else Q <= D;
endmodule
module mac #(parameter N=4)
            (input signed [N-1:0] A, B,
             input  CLK, RESET,
             output signed [4*N-1:0] RES);
wire [2*N-1:0] w1, w2; //Internal wires
wire [4*N-1:0] w3;
smul   #(.N(N))   U00 (.A(A),.B(B),.Mul(w1));
regist #(.N(2*N)) U01 (.D(w1).CLK(CLK),.RESET(RESET),.Q(w2));
sadd   #(.NA(2*N),.NB(4*N),.NSUM(4*N)) U02 (.A(w2),.B(RES),.Sum(w3));
regist #(.N(4*N)) U03 (.D(w3),.CLK(CLK),.RESET(RESET),.Q(RES));
endmodule
```

```
module tb_mac;
parameter N=4;
reg CLK, RESET;
reg signed [N-1:0] A, B;
wire signed [4*N-1:0] RES;
// Clock signal
parameter ClockPeriod = 10;
initial CLK = 0;
always #(ClockPeriod / 2) CLK = ~CLK;
// Instantiate the design under test
mac #(.N(4)) U00 (.A(A),.B(B),.CLK(CLK),
.RESET(RESET),.RES(RES));

initial begin
  RESET = 1; @(posedge CLK) RESET = 0;
  A = 1;  B = -2; @(posedge CLK);
  A = -2; B = 1; @(posedge CLK);
  A = 3;  B = 3; @(posedge CLK);
  A = -3; B = -3; @(posedge CLK);
  @(posedge CLK);
  @(posedge CLK);
  $finish;
end
endmodule
```

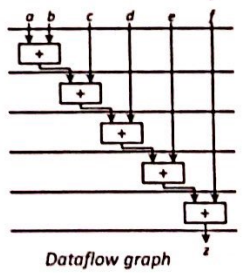


Data dependency graphs and data flow diagram

- *Data dependency graphs* show the dependency and flow of data through computational units. The computation is usually divided over a number of clock cycles. *Data flow graphs* (DFGs) are data-dependency graphs that operations are scheduled into clock cycles. DFG provides an estimate of area and performance For example the following DFD models for $z = a + b + c + d + e + f$ using two-input adders. Horizontal lines mark clock cycle boundaries. Registers are required at the intersection of signals crossing the clock boundaries. Blocks in clock cycles are hardware modules
- Partitioning a data dependency graph involves three tasks: *scheduling*: decide the clock cycle during which an operation must be executed; *allocation*: deciding how many operators are required to implement all of the operations; and *mapping*: deciding which operation will be executed on which hardware computational resources



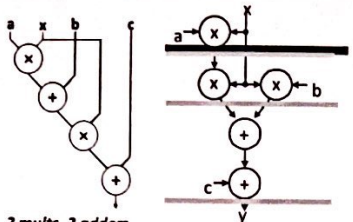
Data-dependency graph



Dataflow graph

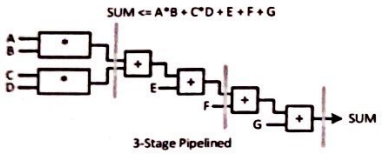
Examples

- Assume an input stream x is passed into a datapath to calculate $y(x_i) = ax_i^2 + bx_i + c$, where a, b, and c are constant values. The pipeline registers are placed such that the DFG is divided into (nearly) equal stages. Throughput can be increased further by pipelining the computational units

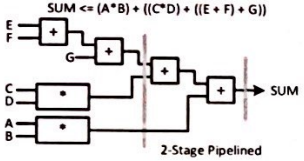


2 mults, 2 adders
Latency = 2, tpd = tm+ta

3 mults, 2 adders
Always have register o
the output(s)

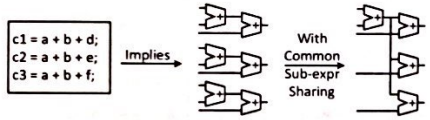


3-Stage Pipelined



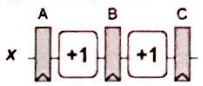
2-Stage Pipelined

- *Common sub-expression* can be shared to reduce the number of logic resources and hence, improving area utilization and timing of the design.
  Parentheses can be used to help the tool recognize common sub-expressions



Pipelined design example

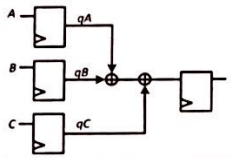
```
module pipelined #(parameter WLx=10, WLout = WLx+1)
                  (input signed [WLx-1:0] x,
                   input CLK,
                   output signed [Wlout-1:0] out);
reg signed  [WLx-1:0] A_out;
reg signed  [WLx  :0] B_out;
wire signed [WLx:0]   B_in;
wire signed [WLx+1:0] C_in;

  always @(posedge CLK) A_out <= x;
  assign B_in = A_out + 1;
  always @(posedge CLK) B_out <= B_in;
  assign C_in = B_out + 1;
  always @(posedge CLK) out <= C_in;
endmodule
```

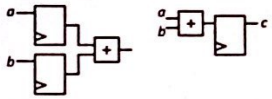



```
// Combining combinationl and sequential descriptions
module pipelined #(parameter WLx=10, WLout = WLx+2)
                  (input signed [WLx-1:0] x,
                   input CLK,
                   output reg signed [Wlout-1:0] out);

  reg signed [WLx-1:0] A;
  reg signed [WLx:0] B;

  always @(posedge CLK) begin
     A   <= x;
     B   <= A + 1;
     out <= B + 1;
  end
endmodule
```

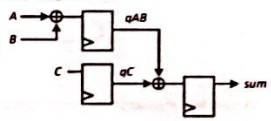
- Why is `Wlout = WLx+1` bits?
- Registering module inputs vs. registering the outputs

```
always @(posedge CLK) begin
   qA <= A;
   qB <= B;
   qC <= C;
   sum <= qA+qB+qC;
end
```






```
always @(posedge CLK) begin
   qAB <= A+B;
   qC <= C;
   sum <= qAB+qC;
end
```



Pipelined signed multiplication

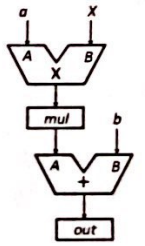
- Multiplying two n-bits values will produce a $2n$-bit result

```
// signed multiplier with input and output registered // 2-stage pipelining
module signed_mult (output reg signed [15:0] out,
                    input CLK,
                    input signed [7:0] a, b);
reg signed [7:0] a_reg;
reg signed [7:0] b_reg;
wire signed [15:0] mult_out;

   assign mult_out = a_reg * b_reg;
   always@(posedge CLK) begin
          a_reg <= a;
          b_reg <= b;
          out <= mult_out;
   end
endmodule
```


```
`timescale 1ns / 1ns
module tb_fall2013;
reg CLK;
reg [7:0] a,b;
wire [15:0] out;
          parameter ClockPeriod = 10;
          initial CLK = 0; // initialize the Clock signal
          always #(ClockPeriod / 2) CLK = ~CLK;
          signed_mult U00 (.out(out),.CLK(CLK),      .a(a), .b(b));
          initial begin
            @(posedge CLK)      a = 2; b = 3;
            @(posedge CLK)      a = -2; b = -2;
            @(posedge CLK)      a = 0; b = 3;
            @(posedge CLK)      a = 4; b = -1;
            @(posedge CLK)      a = -4; b = 4;
          end
endmodule
```

y=aX+b datapath – Behavioral

```
module MultAdd #(parameter WLX = 8, WLA = 3, WLB = 3,
                 parameter signed [WLA-1:0] a = -4,
                 parameter signed [WLB-1:0] b = 3,
                 parameter WLMult = WLX + WLA,
                 parameter WLOut = (WLMult > WLB) ? WLMult + 1 : WLB + 1)
                (input signed [WLX - 1 : 0] x,
                 input CLK,
                 output reg signed [WLOut - 1 : 0] out);
reg signed [WLMult - 1 : 0] multOut;
always @ (posedge CLK)
    multOut <= x * a;
    out <= multOut + b;
  end
endmodule
```




```
module tb_MultAdd;
 parameter WLx = 4, WLA = 3, WLB = 3;
 parameter WLOut = ((WLx + WLA) > WLB) ? WLx + WLA + 1 : WLB + 1;

  reg CLK;
  reg signed [WLx - 1 : 0] x;
  wire signed [WLOut - 1 : 0] out;

parameter HalfClkPeriod = 5;
integer ClkPeriod = HalfClkPeriod*2;
 initial begin : ClockGenerator
    CLK = 1'b0;
    forever #HalfClkPeriod CLK = ~CLK;
 end

  MultAdd #(.WL_x(WL_x), .WL_A(WL_A), .WL_B(WL_B)) DUT (.CLK(CLK), .x(x), .out(out));

 initial begin
     @(posedge CLK); x <= 4'b1001;
     @(posedge CLK); x <= 4'b0001;
     @(posedge CLK); x <= 4'b1111;
     @(posedge CLK);
     @(posedge CLK); $stop;
  end

 endmodule
```

y=aX+b datapath – Structural

Pipelined design

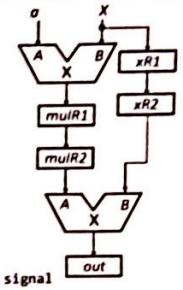
y=aX+b datapath – Structural

```
module smul #(parameter WIDTH = 8)
   (input signed [WIDTH-1:0] A,
    input signed [WIDTH-1:0] B,
    output signed [2*WIDTH-1:0] RESULT);
  assign RESULT = A * B;
endmodule

module axpb #(parameter WIDTH = 4)
   (input signed [WIDTH-1:0] x,
    input CLK,
    output signed [2*WIDTH:0] y);
localparam signed [3:0] a = 3;
localparam signed [3:0] b = -2;
wire signed [2*WIDTH-1 : 0] ax, axq;
wire signed [2*WIDTH : 0] axqpb;

Smul #(.WIDTH(WIDTH)) MUL00 (.A (a),.B(x),.RESULT(ax));
register  #(.WIDTH(2*WIDTH)) REG00 (.D(ax),.Q(axq),.CLK(CLK));
sadd #(.WIDTH(2*WIDTH)) ADD00
(.A(axq),.B({{4{b[3]}},b}),.RESULT(axqpb));
register #(.WIDTH(2*WIDTH+1)) REG01 (.D (axqpb),.Q(y),.CLK(CLK));
endmodule
```

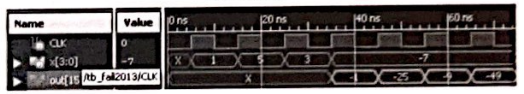
```
module register #(parameter WIDTH = 8)
   (input       [WIDTH-1:0] D,
    output reg   [WIDTH-1:0] Q,
    input CLK);
  always @(posedge CLK) Q <= D;
endmodule

module sadd #(parameter WIDTH = 8)
   (input signed [WIDTH-1:0] A,
    input  signed [WIDTH-1:0] B,
    output signed [WIDTH:0] RESULT);
  assign RESULT = A + B;
endmodule
```

Pipelined design

```
module ax2 #(parameter N=4, a = 4'sb1111)
            (input CLK,
             input signed [N-1:0] x,
             output reg signed [4*N-1:0]
out);
reg signed [2*N-1:0] mulR1, mulR2;
reg signed [N-1:0] xR1, xR2;
  always @(posedge CLK) begin
     xR1 <= x;
     xR2 <= xR1;
     mulR1 <= a*x;
     mulR2 <= mulR1;
     out <= mulR2*xR2;
  end
endmodule
```

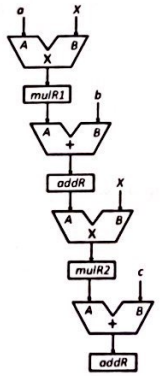



```
`timescale 1ns / 1ns
module tb_fall2013;
localparam WIDTH=4, FACTOR=4'sb1111;
reg CLK;
reg signed [WIDTH-1:0] x;
wire signed [4*WIDTH-1:0] out;
// Clock signal
parameter ClockPeriod = 10;
initial CLK = 0; // initialize the Clock signal
always #(ClockPeriod / 2) CLK = ~CLK;

//Instantiating the DUT
ax2 #(.N(WIDTH), .a(FACTOR)) U00 (.CLK(CLK),.x(x),.out(out));
   initial begin
      @(posedge CLK) x = 4'b0001;
      @(posedge CLK) x = 4'b0101;
      @(posedge CLK) x = 4'b0011;
      @(posedge CLK) x = 4'b1001;
      #(ClockPeriod*4) $finish;
   end
endmodule
```



Pipelined design example

```
module quad #(parameter WLx = 8, WLparam = 4,
              WL_mul1 = WLx + WLparam,
              WL_Add = WL_mul1 + 1,
              WL_mul2 = WL_Add + WLparam,
              WLy = WL_mul2 + 1,
              parameter signed [WLparam - 1 : 0] a = -4,
              parameter signed [WLparam - 1 : 0] b = 3,
              parameter signed [WLparam - 1 : 0] c = -3)
             (input CLK,
              input signed [WLx - 1 : 0] X,
              output reg signed [WLy - 1 : 0] Y);

    reg signed [WLx - 1 : 0] Xq1, Xq2;
    reg signed [WL_mul1 - 1 : 0] mulR1;
    reg signed [WL_Add - 1 : 0] addR;
    reg signed [WL_mul2 - 1 : 0] mulR2;

    always @(posedge CLK) begin
        Xq1 <= X;
        Xq2 <= Xq1;
        mulR1 <= X * a;
        addR <= mulR1 + b;
        mulR2 <= addR * X;
        Y <= mulR2 + c;
    end
endmodule
```



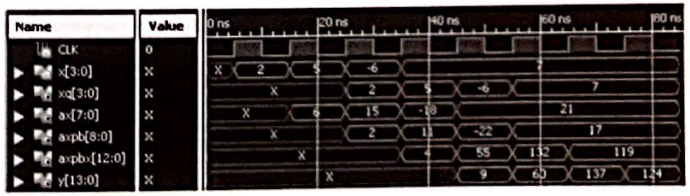

```
module tb_quad();
 parameter WLx = 4;
 parameter WLy = 14;
 reg CLK;
 reg signed [WLx - 1 : 0] X;
 wire signed [WLy - 1 : 0] Y;
 initial CLK <= 0;
 always #5 CLK <= ~CLK;
 quad #(.WLx(WLx)) DUT (.CLK(CLK), .X(X), .Y(Y));
 initial begin
   @(posedge CLK); X <= 4'b1001;
   @(posedge CLK); X <= 4'b0001;
   @(posedge CLK); X <= 4'b1111;
   @(posedge CLK);
   $finish;
 end
endmodule
//if x is not pipelined, you need to apply a new
// input every four clock cycles:  repeat(4) @(posedge CLK);
```

y=(aX+b)x+c datapath

```
module sMul #(parameter WL1 = 4, WL2 = 4)
             (input signed [WL1-1:0] in1,
              input signed [WL2-1:0] in2,
              input  CLK,
              output reg signed [WL1+WL2-1:0] out);
 always @(posedge CLK) out <= in1*in2;
endmodule

`define WIDTH (WL1 > WL2 ? (WL1+1): (WL2+1))
module sAdd #(parameter WL1 = 4, WL2 = 4)
             (input signed [WL1-1:0] in1,
              input signed [WL2-1:0] in2,
              input  CLK,
              output reg signed [`WIDTH-1 : 0] out);
  always @(posedge CLK)
    out <= {{(WL2>WL1 ? (WL2-WL1)+1:1){in1[WL1-1]}},in1} +
           {{(WL1>WL2 ? (WL1-WL2)+1:1){in2[WL2-1]}},in2};
endmodule
```

```
// delaying the input DEPTH clock cycles
module delayLine #(parameter WL = 8, DEPTH = 5)
      (input [WL-1:0] in,
       input CLK,
       output [WL-1:0] out);
reg [WL-1:0] delay [1: DEPTH];
integer i;
  always @(posedge CLK) begin
     delay[1] <= in;
     for (i=2; i<=DEPTH; i=i+1)
      delay[i] <= delay[i-1];
  end
  assign out = delay[DEPTH];
endmodule
```

- The top-level module and the testbench is on the next page



y=(aX+b)x+c datapath

```
module TopLevel #(parameter WLcte = 4, WL = 4)
                 (input CLK,
                  input signed [WL-1:0] x,  // input variable
                  output signed [(WL+WLcte+1+WL+1)-1 : 0] y); // output
// Polynomial coefficients
parameter [WLcte-1:0] a = 3;
parameter [WLcte-1:0] b = -4;
parameter [WLcte-1:0] c = 5;
// Intermediate values
wire [WL+WLcte-1  : 0] ax;    // intermediate value 1: ax
wire [WL+WLcte    : 0] axpb;  // intermediate value 2: ax+b
wire [WL+WLcte+WL : 0] axpbx; // intermediate value 3: x(ax+b)
wire [WL-1        : 0] xq;    // delayed x
  sMul #(WL,WLcte) MUL00 (.in1(x),.in2(a),.CLK(CLK),.out(ax)); // calculating ax
  sAdd #(WL+WLcte,WLcte) ADD00 (.in1(ax),.in2(b),.CLK(CLK),.out(axpb)); // adding ax to b
  delayLine #(.WL (WL), .DEPTH(2)) DELAY00 (.in(x),.CLK(CLK),.out(xq)); // Delay line for x
  sMul #(WL+WLcte+1,WL) MUL01 (.in1(axpb),.in2(xq),.CLK(CLK),.out(axpbx)); // multiplying (ax+b) by x
  sAdd #(WL+WLcte+1+WL,WLcte) ADD01 (.in1(axpbx),.in2(c),.CLK(CLK),.out(y)); // adding (ax+b)x by c
endmodule
```

```
module tb_TopLevel;
parameter WLcte = 4, WL = 4;
reg [WL-1:0] x;
reg CLK;
wire [(WL+WLcte+1+WL+1)-1:0] y;
parameter HalfClkPeriod = 5;
integer ClockPeriod = HalfClkPeriod*2;
TopLevel uut (.x(x),.y(y),.CLK(CLK));
initial begin : ClockGenerator
  CLK = 1'b0;
  forever #HalfClkPeriod CLK = ~CLK;
end
```

```
initial begin
  @(posedge CLK) x = 2;
  @(posedge CLK) x = 5;
  @(posedge CLK) x = -6;
  @(posedge CLK) x = 7;
  #(ClockPeriod*5) $finish;
end
endmodule
```

y=(aX+b)x+c datapath

- Assume the multiplier is 3-stage pipelined and the adder is two-stage pipelined

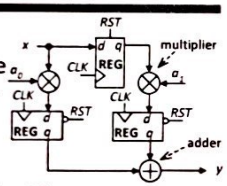
```
module sMul #(parameter WL1=4,WL2=4)
   (input signed [WL1-1:0] in1, // multiplier inputs
    input signed [WL2-1:0] in2,
    input  CLK,
    output reg signed [WL1+WL2-1 : 0] out);
reg signed [WL1+WL2-1:0] stage1;
reg signed [WL1+WL2-1:0] stage2;
  always @(posedge CLK) begin
    stage1 <= in1*in2;
    stage2 <= stage1;
    out <= stage2;
  end
endmodule
```

```
module quadratic #(parameter WLcte = 4, WL = 4)
 (input CLK,
  input signed [WL-1:0] x,
  output signed [WL+WLcte+WL+1 : 0] y);
parameter [WLcte-1:0] a = 3, b = -4, c = 5;
wire [WL+WLcte-1  : 0] ax;       // intermediate value 1: ax
wire [WL+WLcte    : 0] axpb;     // intermediate value 2: ax+b
wire [WL+WLcte+WL : 0] axpbx;    // intermediate value 3: x(ax+b)
wire [WL-1        : 0] xq;     // delayed x
  sMul #(WL,WLcte) MUL00 (.in1(x), .in2(a), .CLK(CLK), .out(ax));
  sAdd #(WL+WLcte,WLcte) ADD00 (.in1(ax), .in2(b), .CLK(CLK), .out(axpb));
  delayLine #(.WL (WL), .DEPTH(5)) DELAY00 (.in (x), .CLK(CLK), .out (xq));
  sMul #(WL+WLcte+1,WL) MUL01 (.in1(axpb), .in2(xq), .CLK(CLK), .out(axpbx));
  sAdd #(WL+WLcte+WL+1,WLcte)    ADD01 (.in1(axpbx), .in2(c), .CLK(CLK), .out(y));
endmodule
```

```
`define WIDTH (WL1 > WL2 ? (WL1+1): (WL2+1))
module sAdd #(parameter WL1 = 4,WL2=4)
  (input signed  [WL1-1:0] in1, // adder inputs
   input signed [WL2-1:0] in2,
   input  CLK,
   output reg signed [`WIDTH-1 : 0] out);
reg [`WIDTH-1: 0] stage1;
  always @(posedge CLK) begin
    // sign extend and add
    stage1 <= { { (WL2>WL1 ? (WL2-WL1)+1 : 1){in1[WL1-1]} },
              in1} + { { (WL1>WL2 ? (WL1-WL2)+1 : 1)
              {in2[WL2-1]} }, in2};
    out <= stage1;
  end
endmodule
```


Filter example – Datapath

- This FIR filter receives an N-bit signed "*x*" as input every clock cycle and produces "*y*" at the output. "*a0*"and "*a1*" are two four-bit signed constant values of 3 and -5, respectively

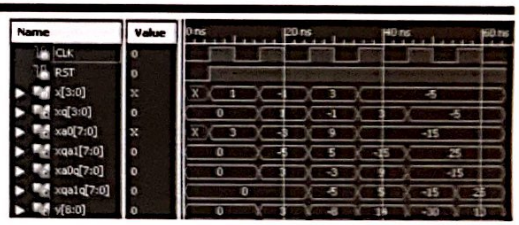



```
module sadd #(parameter NAB = 8, NSUM = 16)
             (input signed [NAB-1:0] A,
              input signed [NAB-1:0] B,
              output reg signed [NSUM-1:0] Sum);
  always @* Sum = A + B;
endmodule
module smul #(parameter NA=8,NB=4)
             (input signed [NA-1:0] A,
              input signed [NB-1:0] B,
              output reg signed [NA+NB-1:0] Mul);
  always @* Mul = A * B;
endmodule
```

```
module Reg #(parameter N = 16)
            (input CLK, RST,
             input [N-1:0] d,
             output reg [N-1:0] q);
    always @(posedge CLK or negedge RST)
          if (!RST) q <= 0;
          else      q <= d;
endmodule
```

```
module FIRfilter #(parameter N = 4)
                  (input signed [N-1 :0] x ,
                   input CLK, RST,
                   output signed [N+4:0] y);
 localparam a0 = 4'sb0011, a1 = 4'sb1011;
 wire [N-1:0] xq;
 wire [N+3:0] xa0, xqa1;
 wire [N+3:0] xa0q, xqa1q;
 Reg  #(.N(N))          U00 (.d (x), .CLK(CLK), .RST (RST), .q(xq));
 smul #(.NA(4), .NB(4)) U01 (.A (a0), .B(x), .Mul (xa0));
 smul #(.NA(4), .NB(4)) U02 (.A (a1), .B(xq), .Mul (xqa1));
 Reg  #(.N(N+4))        U03 (.d (xa0), .CLK(CLK), .RST (RST), .q(xa0q));
 Reg  #(.N(N+4))        U04 (.d (xqa1), .CLK(CLK), .RST (RST), .q(xqa1q));
 sadd #(.NA(N+4), .NB(N+4), .NSUM(N+5)) U05 (.A (xa0q), .B(xqa1q), .Sum (y));
endmodule
```

Filter example – testbench




```
`timescale 1ns / 1ns
module tb_FIRfilter;

parameter N=4;
reg CLK, RST;
reg signed [N-1:0] x;
wire signed [N+4:0] y;

// Clock signal
parameter ClockPeriod = 10;
initial CLK = 0; // initialize the Clock signal
always #(ClockPeriod / 2) CLK = ~CLK;

// Instantiate the design under test
FIRfilter #(.N(4)) U00 (.x(x), .CLK (CLK), .RST (RST), .y(y));

// Synchronous reset
initial begin
  RST = 0; @(posedge CLK) RST = 1;
  x = 1;   @(posedge CLK);
  x = -1;  @(posedge CLK);
  x = 3;   @(posedge CLK);
  x = -5;  @(posedge CLK);
  @(posedge CLK);
  @(posedge CLK);
  $finish;
  end
endmodule
```

3-tap FIR example

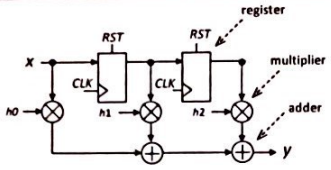
Three-tap FIR filter

```
Reg  #(.N(N+4))          U03 (.d (xa0), .CLK(CLK), .RST (RST), .q(xa0q));
Reg  #(.N(N+4))          U04 (.d (xqa1), .CLK(CLK), .RST (RST), .q(xqa1q));
xadd #(.NA(N+4), .NB(N+4), .NSUM(N+5)) U05 (.A (xa0q), .B(xqa1q), .Sum (y));
endmodule
```

```
  $finish;
 end
endmodule
```

3-tap FIR example

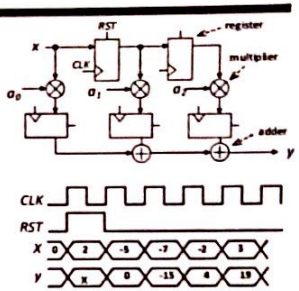
```
module FIR #(parameter WL1 = 4, WL2 = 4)
            (input signed [WL1-1:0] X,
             input RST,CLK,
             input signed [WL2-1:0] h0,h1,h2,
             output reg signed [WL1+WL2:0]Y);
reg signed [WL1 - 1 : 0 ] X1,X2;
always @* Y = (X*h0) + (X1*h1) + (X2*h2);
always @(posedge CLK) begin
  if (RST) begin
     X1 <= 0; X2 <= 0;
  end
  else begin
     X1 <= X; X2 <= X1;
  end
end
endmodule
```




```
`timescale 1ns / 1ps
module tb_FIR;
parameter WL1 = 4, WL2 = 4;
reg signed [WL1 - 1 : 0 ] X;
reg signed [WL2 - 1 : 0]h0,h1,h2;
wire signed [WL1+WL2 : 0]Y;
reg CLK = 0 ;
reg RST;
FIR #(.WL1(WL1),.WL2(WL2)) F1
     (.X(X),.h0(h0),.h1(h1),.h2(h2),
      .RST(RST),.CLK(CLK), .Y(Y));
Always CLK = ~CLK;
initial begin
  RST = 1'b1; h0 = 4'b0011; h1 = 4'b1100; h2 = 4'b1110;
  @(posedge CLK)X = 4'b1111; RST = 1'b0;
  @(posedge CLK)X = 4'b0011;
  @(posedge CLK)X = 4'b1011;
  @(posedge CLK)X = 4'b1000;
end
endmodule
```

Three-tap FIR filter

- The filter receives a WL-bit signed "x" as input every clock cycle and produces "y" at the output




```
module fir_Beh #(parameter WL = 4, N = 4,
                 parameter signed [N-1:0] a0 = 3, a1 = -5, a2 = 2,
                 parameter WL_Mult = WL + N,
                 parameter WL_Out = WL_Mult + 2)
                (input CLK, RST,
                 input signed [WL-1:0] x,
                 output signed [WL_Out-1:0] y);
    reg signed [WL-1:0] xReg1, xReg2;
    reg signed [WL_Mult-1:0] xMult1, xMult2, xMult3;
    always @ (posedge CLK) begin
        if(RST) begin
            xReg1 <= 0;
            xReg2 <= 0;
            xMult1 <= 0;
            xMult2 <= 0;
            xMult3 <= 0;
        end
        else begin
            xReg1 <= x;
            xReg2 <= xReg1;
            xMult1 <= x * a0;
            xMult2 <= xReg1 * a1;
            xMult3 <= xReg2 * a2;
        end
    end
    assign y = xMult1 + xMult2 + xMult3;
endmodule
```

```
module Reg #(parameter WL = 4)
            (input CLK, RST,
             input [WL-1:0] d,
             output reg [WL-1:0] q);
 always @ (posedge CLK)
  if(RST) q <= 0; else q <= d;
endmodule
module Mult #(parameter WL1=4,WL2=4,WLO=WL1+WL2)
             (input signed [WL1-1:0] in1,
              input signed [WL2-1:0] in2,
              output reg signed [WLO-1:0] out);
 always @ (*) out = in1 * in2;
endmodule
```


Three-tap FIR filter

```
module Adder #(parameter WL1 = 4, WL2 = 4,
               parameter WLO = WL1 > WL2 ? WL1 + 1 : WL2 + 1)
              (input signed [WL1-1:0] in1,
               input signed [WL2-1:0] in2,
               output reg signed [WLO-1:0] out);
  always @ (*) out = in1 + in2;
endmodule
module fir_top_struct #(parameter WL = 4, N = 4,
                        parameter signed [N-1:0] a0 = 3, a1 = -5, a2 = 2,
                        parameter WL_Mult = WL + N,
                        parameter WL_Out1 = WL_Mult + 1,
                        parameter WL_Out2 = WL_Mult + 2)
                       (input CLK, RST,
                        input signed [WL-1:0] x,
                        output signed [WL_Out2-1:0] y);
   wire signed [WL-1:0] xRegOut1, xRegOut2;
   wire signed [WL_Mult-1:0] xMult1, xMult2, xMult3;
   wire signed [WL_Mult-1:0] xMult_Reg1, xMult_Reg2, xMult_Reg3;
   wire signed [WL_Out1-1:0] adderOut;
   Reg #(.WL(WL)) UUT0 (.CLK(CLK), .RST(RST), .d(x), .q(xRegOut1));
   Reg #(.WL(WL)) UUT1 (.CLK(CLK), .RST(RST), .d(xRegOut1), .q(xRegOut2));
   Mult #(.WL1(WL), .WL2(N)) UUT2 (.in1(x), .in2(a0), .out(xMult1));
   Mult #(.WL1(WL), .WL2(N)) UUT3 (.in1(xRegOut1), .in2(a1), .out(xMult2));
   Mult #(.WL1(WL), .WL2(N)) UUT4 (.in1(xRegOut2), .in2(a2), .out(xMult3));
   Reg #(.WL(WL_Mult)) UUT5 (.CLK(CLK), .RST(RST), .d(xMult1), .q(xMult_Reg1));
   Reg #(.WL(WL_Mult)) UUT6 (.CLK(CLK), .RST(RST), .d(xMult2), .q(xMult_Reg2));
   Reg #(.WL(WL_Mult)) UUT7 (.CLK(CLK), .RST(RST), .d(xMult3), .q(xMult_Reg3));
   Adder #(.WL1(WL_Mult), .WL2(WL_Mult)) UUT8 (.in1(xMult_Reg1), .in2(xMult_Reg2), .out(adderOut));
   Adder #(.WL1(WL_Out1), .WL2(WL_Mult)) UUT9 (.in1(adderOut), .in2(xMult_Reg3), .out(y));
endmodule
```

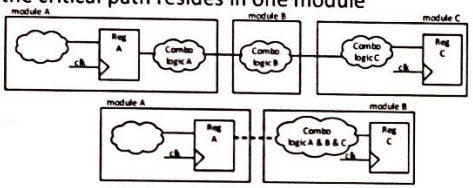
```
module Reg #(parameter WL = 4)
            (input CLK, RST,
             input [WL-1:0] d,
             output reg [WL-1:0] q);
 always @ (posedge CLK)
  if(RST) q <= 0; else q <= d;
endmodule
module Mult #(parameter WL1=4,WL2=4,WLO=WL1+WL2)
             (input signed [WL1-1:0] in1,
              input signed [WL2-1:0] in2,
              output reg signed [WLO-1:0] out);
 always @ (*) out = in1 * in2;
endmodule
```

Three-tap FIR filter testbench

```
module tb_fir;
    parameter WL = 4, N = 4;
    parameter signed [N-1:0] a0 = 3, a1 = -5, a2 = 2;
    parameter WL_Mult = WL + N;
    parameter WL_Out1 = WL_Mult + 1;
    parameter WL_Out2 = WL_Mult + 2;

    reg CLK, RST;
    reg signed [WL-1:0] x;
    wire signed [WL_Out2-1:0] y;

    fir_top_struct #(.WL(WL), .N(N), .a0(a0), .a1(a1), .a2(a2), .WL_Mult(WL_Mult),
                     .WL_Out1(WL_Out1),.WL_Out2(WL_Out2)) UUT (.CLK(CLK), .RST(RST),
                     .x(x), .y(y));
    initial CLK = 0;
    always #5 CLK = ~CLK;
    initial begin
        x <= 4'b0000; RST <= 0;
        @ (posedge CLK); RST <= 1; x <= 4'b0010;
        @ (posedge CLK); RST <= 0; x <= 4'b1011;
        @ (posedge CLK); x <= 4'b1001;
        @ (posedge CLK); x <= 4'b1110;
        @ (posedge CLK); x <= 4'b0011;
        @ (posedge CLK); $finish;
    end
endmodule
```

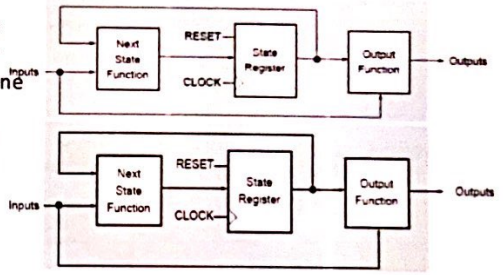
Design partitioning

- For relatively large designs, it is often desirable to manage the complexity by partitioning the design into smaller blocks. Block sizes are typically chosen so that the logical data flow between blocks and hence routing, is reduced. Also the best optimization results are achieved when the relevant logic reside in one module
- The best optimization results are achieved when the entire cone of logic for a path is contained within the module undergoing optimization
- When partitioning a design into various blocks, keep related combinational logic together in the same module. Synthesis tool has more flexibility in optimizing a design when related combinational logic is located in the same module than have to move logic across hierarchical boundaries. By placing registers at the output signals of each sub-block, no *snake paths* (combinational paths that "weave" in and out of several modules) are created. Also, no combinational logic will be created between sub-blocks and the critical path resides in one module
- This simplifies logic optimization and also timing analysis by minimizing the critical path delay without having to iterate among various modules descriptions



Finite state machines (FSMs) in Verilog

- **always** blocks are most suitable for describing FSMs. You may describe a FSM using one, two, or three **always** blocks. Each **always** block implies a separate block of logic
- A FSM description using three **always** blocks include one for updating the state machine, one combinational **always** to determine the next state, and one combinational **always** to generate the state machine output values



- Note that a given signal can only be assigned in one **always** block. Assigning values to a signal from two or more **always** blocks may create a race condition and synthesis tools often generate an error
- The next state is a function of the present state and external input
- The state register can be reset, which is usually used as an initialization state

Mealy FSM

```
/* Mealy Machine coded with 3 always blocks */
// state register with sync. reset
always @(posedge CLK)
  begin
    if (RESET)
      current_state <= state_0;
    else current_state <= next_state;
  end
// next state combinational logic
// use blocking assignments
always @(inputs or current_state)
  next_state = _
// output combinational logic
// use blocking assignments
always @(inputs or current_state)
  outputs = _
```

```
/* Mealy Machine coded with 2 always blocks */
// next state logic and state register
always @(posedge clock)
  if (RESET) current_state <= state_0;
  else case (current_state)
       state1:
         if (input1) current_state <= state2;
         else current_state <= state3;
           _
    endcase
// output combinational logic
// use blocking assignments
always @(inputs or current_state)
  outputs = _
```

- State transitions are described using the next-state logic, which is typically modeled using a **case** statement with the state register signal (representing present state) as the case expression
- The next state CL can be described in a separate combinational **always** block or in the sequential state register **always** block. If using a separate combinational **always** block, its sensitivity list should contain the present state signals and all FSM inputs
- Non-registered outputs can be described in a combinational **always** block (or using concurrent signal assignment statements). Registered outputs can be assigned within the sequential **always** block

Moore FSM in Verilog

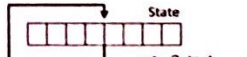
```
/* Moore Machine coded with 3 always blocks*/
// state register with sync. reset
always @(posedge CLK) // sync. Reset
 begin
   if (RESET)
     current_state <= state_0;
   else current_state <= next_state;
 end

// next state combinational logic
// use blocking assignments
always @(inputs or current_state)
   next_state = _

// output combinational logic
// use blocking assignments
always @(current_state)
  outputs = _
```

```
/*Moore Machine coded with 2 always blocks*/
// next state logic and state register
always @(posedge CLK)
  if (RESET) current_state <= state_0;
  else
     case (current_state)
     state1:
        if (input1)
         current_state <= state2;
        else
          current_state <= state3;
          _
     endcase

// output combinational logic
// use blocking assignments
always @(current_state)
  outputs = _
```

- Synthesis tools can automatically recognize FSMs from HDL code
- *State encoding* is the process of assigning binary values to states of a FSM. An n-bit state has 2^n unique possible binary patterns. Often, not all 2^n patterns are required, so the *unused states* should not occur during FSM operation
- The code is more readable and easier to modify by using symbolic constants to represent the FSM's states, rather than referring to them by binary values

8-bit counter example

- A counter perhaps is the simplest FSM. The counter counts through a number of



Verilog FSM description using one always block

- This Moore machine has:
  - four states: s1, s2, s3, s4
  - an asynchronous reset

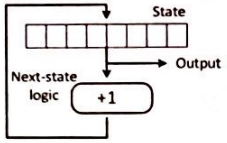
```
// State Machine with a single always block
module fsm_1 (input CLK, RST, x1,
              output reg outp);
localparam  s1 = 2'b00, s2 = 2'b01,
```

using concurrent signal assignment statements). Registered outputs can be assigned within the sequential **always** block

- The code is more readable and easier to modify by using symbolic constants to represent the FSM's states, rather than referring to them by binary values

8-bit counter example

- A counter perhaps is the simplest FSM. The counter counts through a number of finite states. The inputs are clock, reset. The counter value denotes the current state and the also the output of the FSM. Next state is just the register value+1

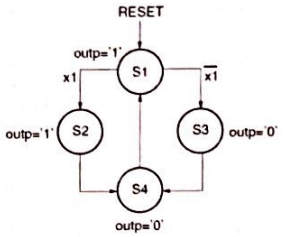



```
module counter_tb;
reg RST, CLK; // inputs
wire [7:0] value; // output

// Instantiate the unit under test (UUT)
counter8 uut(.RST(RST),.CLK(CLK),.value(value));

// Initialize simulation
initial begin
  CLK = 0; RST = 0;
  #5;
  RST = 1;
end

// model a test clock
always begin
  #1; CLK = ~CLK;
end
endmodule
```

```
module counter8 (input RST,
                 input CLK,
                 output [7:0] value);
reg [7:0] state;
wire [7:0] next_state;

  // modeling the state (active low reset)
  always @(posedge CLK or negedge RST) begin
    if (!RST) state <= 8'h00;
    else state <= next_state;
  end

  // modeling the next-state logic
  assign next_state = state + 1;

  // modeling output logic
  assign value = state;
endmodule
```

Verilog FSM description using one always block

- This Moore machine has:
  - four states: s1, s2, s3, s4
  - an asynchronous reset
  - one input: "x1"
  - one output: "outp"




```
// State Machine with a single always block
module fsm_1 (input CLK, RST, x1,
              output reg outp);
localparam  s1 = 2'b00, s2 = 2'b01,
            s3 = 2'b10, s4 = 2'b11;
reg [1:0] state=2'b00; //in-line initialization

  always @(posedge CLK or posedge RST) begin
    if (RST) begin
       state <= s1;
       outp <= 1'b1;
    end
    else begin
      case (state)
      s1: begin
          if (x1) begin
            state <= s2;
            outp <= 1'b1;
          end
          else begin
            state <= s3;
            outp <= 1'b0;
          end
          end
      s2: state <= s4; outp <= 1'b0;
      s3: state <= s4; outp <= 1'b0;
      s4: state <= s1; outp <= 1'b1;
      endcase
    end
  end
endmodule
```

- A four state FSM, requires two flip-flops to implement binary encoding scheme

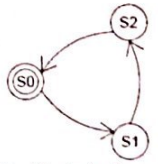
Verilog FSM description using two and three always blocks

```
module fsm (input CLK, RST, x1;
            output reg outp);
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
reg [1:0] state;

  always @(posedge CLK or posedge RST) begin
    if (RST) state <= s1;
    else begin
      case (state)
          s1: if (x1 == 1'b1)
              state <= s2;
             else state <= s3;
          s2: state <= s4;
          s3: state <= s4;
          s4: state <= s1;
      endcase
    end
  end

  //output always statement
  always @(state) begin
   case (state)
     s1: outp = 1'b1;
     s2: outp = 1'b1;
     s3: outp = 1'b0;
     s4: outp = 1'b0;
   endcase
  end
endmodule
```

```
module fsm (input CLK, RST, x1;
            output reg outp);
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
reg [1:0] state, next_state;

  always @(posedge CLK or posedge RST)
    if (RST) state <= s1;
    else state <= next_state;

  always @(state or x1) begin
    case (state)
      s1: if (x1 == 1'b1)
            next_state = s2;
          else next_state = s3;
      s2: next_state = s4;
      s3: next_state = s4;
      s4: next_state = s1;
    endcase
  end

  always @(state) begin
    case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
    endcase
  end
endmodule
```

Divide-by-3 clock FSM example

```
module divideby3FSM (input CLK,
                     input RST,
                     output y);
reg [1:0] state, nextstate;
parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;

// state register
always @ (posedge CLK)
  if (RST) state <= S0;
  else state <= nextstate;

// next state logic
always @ (*)
  case (state)
    S0: nextstate = S1;
    S1: nextstate = S2;
    S2: nextstate = S0;
    default: nextstate = S0;
endcase

// output logic
assign y = (state == S0);
endmodule
```



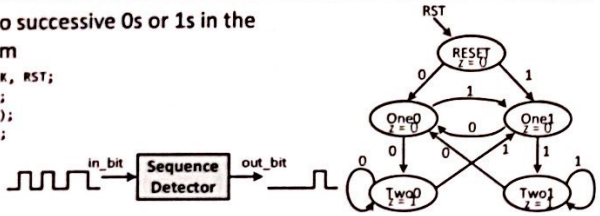
Double circle shows the reset state

```
module tb_divByThree;
// Inputs
reg CLK;
reg RST;
// Outputs
wire y;
   // Instantiate the UUT
   divByThree uut (.CLK(CLK),
                   .RST(RST),
                   .y(y));

   parameter HalfClkPeriod = 5;
   integer ClkPeriod = HalfClkPeriod*2;

   initial begin : ClockGenerator
     CLK = 1'b0;
     forever #HalfClkPeriod CLK = ~CLK;
   end
   initial begin
     RESET = 1; #(3*ClkPeriod) RESET=0;
   end
endmodule
```

- Because the next state logic should be combinational, a **default** is necessary even though the state "11" should never arise

Sequence detector FSM

- This FSM detects two successive 0s or 1s in the serial input bit stream

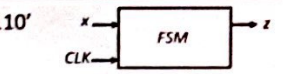


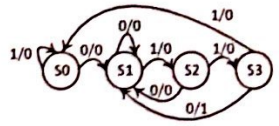

```
module seq_detect (input CLK, RST;
                   input in;
                   output z);
reg [2:0] state, next_state;
// State declarations
parameter  RESET = 3'b000;
parameter  One0  = 3'b001;
parameter  One1  = 3'b010;
parameter  Two0  = 3'b011;
parameter  Two1  = 3'b100;
 // state register sequential logic
  always @ (posedge CLK)
     if (RST == 1) state <= RESET;
     else state <= next_state;
// next state combinational logic
always @ (state or in)
  case (state)
  RESET:
   if (in == 0) next_state = One0;
   else if (in == 1) next_state = One1;
   else next_state = reset_state;
  One0:
   if (in == 0) next_state = Two0;
   else if (in == 1) next_state = One1;
   else next_state = reset_state;
```

```
Two0: if (in == 0) next_state = Two0;
      else if (in == 1) next_state = One1;
      else next_state = reset_state;
One1: if (in == 0) next_state = One0;
      else if (in == 1) next_state = Two1;
      else  next_state = reset_state;
Two1: if (in == 0) next_state = One0;
      else if (in == 1) next_state = Two1;
      else next_state = reset_state;
default: next_state = reset_state;
endcase

  //Output combinationnal logic
assign z = ((state==Two0)|| (state==Two1)) ? 1 : 0;
endmodule
```

Sequence detector and serial parity detector

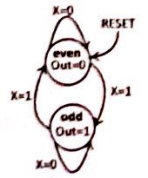
- Sequence detector for the pattern '0110'





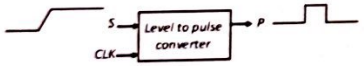

```
// Sequence detector for the pattern '0110'
module seq_detector (input x, CLK,
                     output reg z);
localparam S0=0, S1=1, S2=2, S3=3;
reg [0:1] state, nextState;
   always @ (posedge CLK)
        state <= nextState;

   always @ (state,x)
      case (state)
         S0: begin
           z = x ? 0 : 0;
            nextState = x ? S0 : S1;
         end;
         S1: begin
           z = x ? 0 : 0;
           nextState = x ? S2 : S1;
         end;
         S2: begin
           z = x ? 0 : 0;
            nextState = x ? S3 : S1;
         end;
         S3: begin
           z = x ? 0 : 1;
            nextState = x ? S0 : S1;
         end;
      endcase
endmodule
```

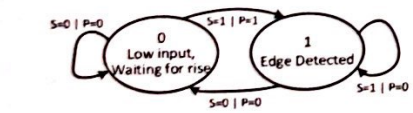
```
module parity_gen (input x, CLK,
                   output reg z);
reg state; // The machine state
parameter EVEN=0, ODD=1;

   always @ (posedge CLK)
     case (state)
       EVEN: begin
         z <= x ? 1 : 0;
         state <= x ? ODD : EVEN;
       end
       ODD: begin
         z <= x ? 0 : 1;
         state <= x ? EVEN : ODD;
       end
     endcase

endmodule
```



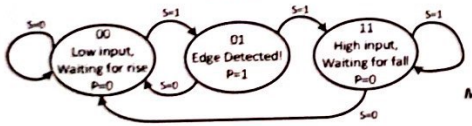
Edge detector FSM

- A *rising-edge detector* is a circuit that generates a one clock-cycle pulse *P* every time the input signal *S* changes from 0 to 1 , which is one clock period wide. Thus it is a synchronous rising-edge detector. The detector also receives the clock signal CLK and a reset signal RESET and generates the output signal *P*





Mealy FSM

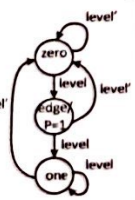


Moore FSM

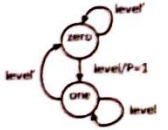
- In Moore machines, the output is a function only of the current state of the machine and in Mealy machines the outputs are a function of the current state of the machine AND the inputs. Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations

Edge detector HDL

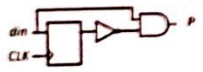
```
module edgeMoore #(parameter zero=2'b00,
                   edge=2'b01, one=2'b10)
     (input CLK, RESET, din,
      output reg P) ;
reg [1:0] state_reg , state_next;
always @(posedge CLK, posedge RESET)
  if (RESET) state_reg <= zero;
  else       state_reg <= state_next ;
//next state and output combinational logic
always @* begin
  state_next = state_reg;
  P = 1'b0; //default output
  case ( state_reg )
   zero: if (din)  state_next = edge;
   edge: begin
           P = 1'b1;
           if (din) state_next = one;
           else     state_next = zero;
         end
   one : if (~din) state_next = zero;
   default: state_next = zero;
  endcase
end
endmodule
```




```
module edgeMealy #(parameter  zero = 1'b0,
                              one  = 1'b1)
          (input CLK, RESET, din,
           output reg P) ;
reg [1:0] state_reg , state_next;
always @ (posedge CLK, posedge RESET)
  if (RESET) state_reg <= zero;
  else state_reg <= state_next;
//next state and output combinational logic
  always @* begin
    state_next = state_reg;
    P = 1'b0; //default output
    case ( state_reg )
     zero: if (din)  begin
             P = 1'b1;
             state_next = one;
           end
     one : if (~din) state_next = zero;
     default: state_next =zero;
    endcase
  end
endmodule
```



- The output can be interpreted as it is asserted only when the current input is 1 and the previous input, which is stored in a FF, was 0, and hence the logic can be designed directly



Edge detector testbench

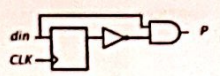
- Mealy outputs generally occur one cycle earlier than a Moore:

Level-to-pulse converter

- In various cases, such as for push buttons pressed by humans for arbitrary periods of time, an asynchronous signal must be converted to a synchronous pulse. We

machine and in Mealy machines the outputs are a function of the current state of the machine AND the inputs. Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations

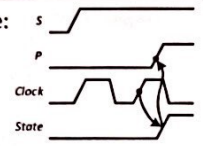
- The output can be interpreted as it is asserted only when the current input is 1 and the previous input, which is stored in a FF, was 0, and hence the logic can be designed directly



Edge detector testbench

- Mealy outputs generally occur one cycle earlier than a Moore:

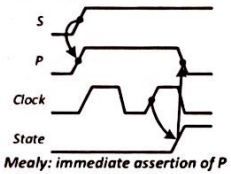
```
module edgeDetector   (input CLK, RESET, input din,
                       output mooreOut, mealyOut) ;
 edgeMoore MOORE (.CLK (CLK), .RESET(RESET), .din(din), .pe(mooreOut));
 edgeMealy MEALY (.CLK (CLK), .RESET(RESET), .din(din), .pe(mealyOut));

endmodule
```



Moore: delayed assertion of P

```
//Testbench for the FSMs
`timescale 1ns / 1ps
module tb_edgeDetector;
 // Inputs
reg CLK, RESET, din;
// Outputs
wire mooreOut, mealyOut, logicOut;
// Instantiate the Unit Under Test (UUT)
edgeDetector uut (.CLK(CLK),.RESET(RESET),
                  .din(din),
                  .mooreOut(mooreOut),
                  .mealyOut(mealyOut),
                  .logicOut(logicOut));
// Clock signal
parameter HalfClkPeriod = 5;
integer ClkPeriod = HalfClkPeriod*2;
initial begin : ClockGenerator
        CLK = 1'b0;
        forever #HalfClkPeriod CLK = ~CLK;
end
```

```
initial begin
 // Initialize Inputs
  RESET = 1; din = 0;
 @(posedge CLK);
 @(posedge CLK) RESET = 0;
 #HalfClkPeriod din = 1;
 @(posedge CLK);
 @(posedge CLK) din=0;
 @(posedge CLK) din=1;
end
endmodule
```



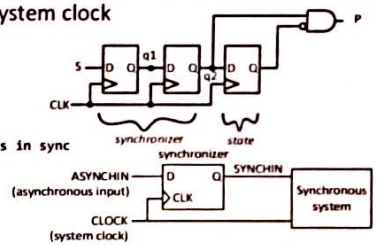
Mealy: immediate assertion of P

- Note that the input is an asynchronous signal. Asynchronous signals have no known timing relationship with the clock of the synchronous module

Level-to-pulse converter

- In various cases, such as for push buttons pressed by humans for arbitrary periods of time, an asynchronous signal must be converted to a synchronous pulse. We can use one or more flip-flops as a *synchronizer* to bring an external asynchronous input signal into the timing domain of the system clock

```
//The output goes high for one clock cycle
//whenever it makes a low-to-high transition
module button (input CLK, input S, output P);
reg q1,q2,q3;
  always @ (posedge CLK) begin
     q1 <= S;  // first FF in synchronizer
     q2 <= q1; // second FF in synchronizer, output is in sync
     q3 <= q2; // remembers previous state of button
  end
  // rising edge = old value is 0, new value is 1
  assign P = ~q3 & q2;
endmodule
```

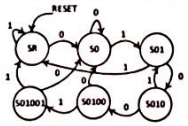


- State machine FFs (state register) may power up in an unknown state. When unused states exist, the FSM should move from any unused state to a used state to avoid lock-up states. Use a reset signal to ensure that the FSM is always initialized to a known valid state before normal operations begin. By default, a reset state is selected as the recovery state. In the absence of a reset, there is no way of predicting the initial value of the state register flip-flops during the "power up" operation. It may initialize in an un-encoded state (lock-up). Use a **default** in the **case** selection items of the next state logic to recover from an unused state

Combination lock example

- A combination lock would unlock if the input sequence is "01001". The lock has a reset input "RESET", two inputs "b0" and "b1", one for 0 and one for 1, respectively, and an unlock output "unlock". In reality, inputs are asynchronous signals, but assume that they are synchronous

```
module lock(input CLK,RESET,
            input b0, b1,
            output unlock);
// State assignments, SR is the reset state
localparam SR=0,S0=1,S01=2,S010=3,
           S0100=4,S01001=5;
reg [2:0] state,next_state;
  always @(posedge CLK)
    if (RESET) state <= SR;
    else state <= next_state;

  always @(state,b0,b1)
  case (state)
    SR: next_state = b0 ? S0 : b1 ? SR : state;
    S0: next_state = b0 ? S0 : b1 ? S01 : state;
    S01: next_state = b0 ? S010 : b1 ? SR : state;
    S010: next_state = b0 ? S0100 : b1 ? S01 : state;
    S0100: next_state = b0 ? S0 : b1 ? S01001 : state;
    S01001: next_state = b0 ? S0 : b1 ? SR : state;
    default: next_state = SR; // unused states
  endcase
  assign unlock = (state == S01001);
endmodule
```




```
module tb_lock;
reg CLK, RESET;
reg b0, b1;
wire unlock;

  lock L00 (.CLK(CLK),.RESET(RESET),
            .b0(b0),.b1(b1),.unlock(unlock));
  always #5 CLK = ~CLK;
  initial begin
     CLK=0;
     RESET = 1;
     @(posedge CLK) RESET <= 0; b0<=1; b1<= 0;
     @(posedge CLK) b0 <= 1; b1 <= 0;
     @(posedge CLK) b0 <= 1; b1 <= 0;
     @(posedge CLK) b0 <= 0; b1 <= 1;
     @(posedge CLK) b0 <= 1; b1 <= 0;
     @(posedge CLK) b0 <= 1; b1 <= 0;
     @(posedge CLK) b0 <= 0; b1 <= 1;
     @(posedge CLK) b0 <= 1; b1 <= 0;
     @(posedge CLK) b0 <= 0; b1 <= 1;
     @(posedge CLK) b0 <= 0; b1 <= 1;
     @(posedge CLK);
     $finish;
  end
endmodule
```

State encoding

- The state encoding of a FSM can influence the number of FFs and also the complexity of the next-state and output logic
- In *binary encoding*, the number of FFs is the smallest number m such that $2^m \geq n$, where n is the number of states in the FSM. Thus an FSM with m-state requires at least $\lceil \log_2 m \rceil$ state FFs
  - Binary encoding uses fewer FFs compared to other encodings, but requires more complicated combinational logic to define the next state
- In *one-hot encoding* one bit (one FF) is associated with each state in the FSM. For a FSM with N states, N flip-flops are used to encode the states and hence, no unused states. Although it requires more state FFs than that of binary encoding, but often much smaller combinational logic for the next state logic. Suitable for high performance systems. Only two bits of the state register will toggle during a transition between two states. It is used in low-power systems since it minimizes the number of power consuming transitions (i.e., 0 to 1)
- In *gray encoding* or *reflected binary encoding* two consecutive values (states) differ in only one bit. For example, 3-bit gray codes are 000, 001, 011, 010, 110, 111, 101, 100. Gray encoding can have lower power consumption compared to binary encoding
- State encoding can be specified for the synthesis process

Vectors, arrays, and memories

- Verilog supports three similar data structures: *vectors*, *arrays* and *memories*
- Vectors are used to represent multi-bit variables and busses

```
reg  [7:0] MultiBitWord1;// 8-bit reg vector with MSB=7 LSB=0; little endian
wire [0:7] MultiBitWord2;// 8-bit wire vector with MSB=0 LSB=7; big endian
```

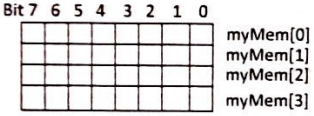
- *Arrays* are used to hold several variables of the same data type

```
reg r[0:4];// An array of 5 1-bit registers
reg membits [0:1023];// 1024 1-bit registers
integer i[0:3];  //integer array with a length of 4; four integer values
```

- *Memories* are arrays of vectors defined using **reg** data type

```
                reg [ msb : lsb ] memory1 [ lower: upper ];
reg [7:0]  myMem [0:3]; // It defines a memory with 4 locations and each
                        //  location contains an 8-bit data
```

```
reg [7:0] mem [0:4095], data;
// 4096 memory words that are 8 bits wide
assign data = select ? mem[address]: 'z;
reg [7:0] memdata[0:255]; // 256 8-bit registers
```



Memories

- A memory row may contain one or more words
- One and multi-dimensional arrays of both **reg**, net, and **real** data types are supported, but only one and two-dimensional arrays are synthesizable in most synthesis tools
- A row of memory can be accessed by a memory index as **mem[index]** similar to a bit select

```
reg [7:0] Mem [0:255], out;
assign out = Mem[address];
```

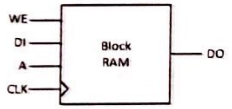
- Arrays can be accessed using bit and part selects

```
reg [31:0] Mem [0:255];
wire [7:0] out; assign out = Mem[100][31:24];
```


Single-port synchronous RAM

- It is important to specify the timing of reads and writes, because if a value is written at the same time it is read, the value of the read could correspond to the old value or the newly written value

```
//Synchronous RAM, always read, read-first mode
module ramRF #(parameter AWL = 6,
                         DWL = 16)
              (input CLK, WE, EN,
               input [AWL-1:0] addr,
               input [DWL-1:0] di,
               output reg [DWL-1:0] do);
// The RAM variable with depth 2**(AWL)
reg [DWL-1:0] RAM [2**AWL-1:0];

   always @(posedge CLK) begin
      if (EN) begin
         if (WE) RAM[addr]<=di;
         do <= RAM[addr];
         // do doesn't get new din
         // in this clock cycle
      end
   end
endmodule
```




```
//Synchronous RAM, always read, write-first mode
// Single-port RAM with read through
module ramWF #(parameter AWL = 6, DWL = 16)
              (input CLK, WE, EN,
               input [AWL-1:0] addr,
               input [DWL-1:0] di,
               output reg [DWL-1:0] do);
reg [DWL-1:0] RAM [2**AWL-1:0];
reg [AWL-1:0] addrq;
   always @(posedge CLK) begin
      if (EN) begin
         if (WE) RAM[addr]<=di;
         addrq <= addr; //registered address
      end
   end
   assign do = RAM[addrq]; //synchronous read
endmodule
```

Single-port synchronous RAM

- If the write-first mode is not supported by memory blocks, the new written data is not available until the next edge of the clock. Therefore, the result of functional simulation would not match with post synthesis

```
//Synchronous RAM, always read, read-first mode with reset
module ramRF #(parameter AWL = 6,
                         DWL = 16,
                         resetValue = 0)
              (input CLK, WE, EN, RST,
               input [AWL-1:0] addr,
               input [DWL-1:0] di,
               output reg [DWL-1:0] do);
reg [DWL-1:0] RAM [2**AWL-1:0];
   always @(posedge CLK) begin
      if (EN) begin
         if (WE) RAM[addr]<=di;
         if (RST) do <= resetValue;
         else     do <= RAM[addr];
      end
   end
endmodule
```

```
//Synchronous RAM, either read or write
module ram #(parameter AWL = 6,
                       DWL = 16)
            (input CLK, WE, EN,
             input [AWL-1:0] addr,
             input [DWL-1:0] di,
             output reg [DWL-1:0] do);
reg [DWL-1:0] RAM [2**AWL-1:0];
   always @(posedge CLK) begin
      if (EN) begin
         if (WE) RAM[addr]<=di;
         else do <= RAM[addr];
      end
   end
endmodule
```

Single and dual-port RAMs

- It is recommended to use RAMs with synchronous read and write operations

$readmemb and $readmemh system functions

- **$readmemb** and **$readmemh** system functions read and load information stored in an ASCII file in binary and hexadecimal formats, respectively, into a memory

```
    end
  end
  assign do = RAM[addrq]; //synchronous read
endmodule
```

```
  end
endmodule
```

Single and dual-port RAMs

- It is recommended to use RAMs with synchronous read and write operations

```
//RAM with asynchronous read, synchronous write
module ram #(parameter AWL = 6, DWL = 16)
            (input CLK, WE,
             input [AWL-1:0] addr,
             input [DWL-1:0] di,
             output [DWL-1:0] do);
reg [DWL-1:0] RAM [2**AWL-1:0];
   always @(posedge CLK) if (WE) RAM[addr]<=di;
  assign do = RAM[addr];
endmodule
```

```
//dual-port RAM with asynchronous read, synchronous write
module ram #(parameter AWL = 6, DWL = 16)
            (input CLK, WE,
             input [AWL-1:0] addr1, addr2,
             input [DWL-1:0] di,
             output [DWL-1:0] do1, do2);
reg [DWL-1:0] RAM [2**AWL-1:0];
   always @(posedge CLK)
      if (WE) RAM[addr1]<=di; //synchronous write
  assign do1 = RAM[addr1]; // two read ports. asynchronous
  assign do2 = RAM[addr2];
endmodule
```

$readmemb and $readmemh system functions

- $readmemb and $readmemh system functions read and load information stored in an ASCII file in binary and hexadecimal formats, respectively, into a memory

```
$readmemh("fileName.ext", memName[, startAddress [, endAddress]]);
```

- memName specifies the memory name to be loaded. startAddress optionally specifies the starting address of the data. If startAddress is given, the memory array will be filled starting at that address and continue until endAddress (or the end of the array) is reached. If none is specified, the left-hand address given in the memory declaration is used. endAddress is the last address to be written into
- The numbers in the file are separated by white space and new lines. The address of data can also be specified within the file using the construct **"@hhh…h"** , which specifies the hexadecimal address to use as the starting address. If no initial address is given, @0 is assumed for the first data word. Data not included in the file will be given xxx... values

```
// data file
// Since start_addr =10 memry[0:9] will all be stored as xxxxxxxx
@00A //address 10. Address in hexadecimal is not case sensitive
10101100 11110000 1x000x11 11110101 01011010 01001100
XxxxZzzz 00000000
@01E // address 30
1100_1010 00011_0001 // underscore is used for more readability
```


Initializing memories

- You can use an **initial** block in the testbench to initialize the contents of a memory from a data file

```
reg [7:0] mem [0:31]; // 32 byte memory.
initial begin
  // Initialize memory contents from file.
  $readmemb("init.dat", mem, 8);
  // words 8 and 9 are not in the file
  // and will default to x
end
```

- The initialization work identically in synthesis and simulation

```
module example;
reg [31:0] Mem [0:255];
initial $readmemh("data.txt",Mem);
integer k;

  initial begin
   $display("Contents of Mem after
reading data file:");
   for (k=0; k<12; k=k+1)
    $display("%d:%h",k,Mem[k]);
  end
endmodule
```

```
module array_tb;
wire [7:0] response;
reg  [7:0] stimulus, stim_array[0:15];
integer i;
parameter clkPeriod = 10;
reg CLK;
always #(clkPeriod/2) CLK <= ~CLK;
   DUT u1 (response, stimulus);
   initial begin
      $readmemb("datafile",stim_array);
      for (i = 0; i <= 15; i = i + 1)
         @(posedge CLK) stimulus = stim_array[i];
      @(posedge CLK) $finish;
   end
endmodule
```

Synchronous ROMs using BlockRAM resources

- A *read-only memory* (ROM) is usually modeled using a case statement with one constant value is being assigned to every word of the ROM
- To treat the ROM as a black box, model the ROM in a separate module, which contain only the ROM logic

```
module rom (input CLK, EN,
            input [9:0] addr,
            output reg [15:0] dout);
reg [15:0] rom [0:1023];
  always @(posedge CLK)
    if (EN)     dout <= rom[addr];
endmodule
```

```
module ROM #(parameter WL=8, AWL=16, DEPTH=2 ** AWL)
              (input CLK, RST,
               input [AWL - 1 : 0] address,
               output reg [WL - 1 : 0] memOut);

       reg [WL - 1 : 0] ROM [0 : DEPTH - 1];

       initial $readmemb("init.dat", ROM);
       always @(posedge CLK) begin
           if(RST) memOut <= 0;
           else memOut <= ROM[address];
       end

endmodule
```

```
//rom with registered output
module romSync #(parameter AWL=3, DWL=4)
                (input clk, en,
                 input [AWL-1:0] addr,
                 output reg [DWL-1:0] data);

  always @(posedge clk) begin
    if (en)
     case(addr)
     3'b000: data <= 4'b0010;
     3'b001: data <= 4'b0010;
     3'b010: data <= 4'b1110;
     3'b011: data <= 4'b0010;
     3'b100: data <= 4'b0100;
     3'b101: data <= 4'b1010;
     3'b110: data <= 4'b1100;
     3'b111: data <= 4'b0000;
     default: data <= 4'bXXXX;
     endcase
  end
endmodule
```

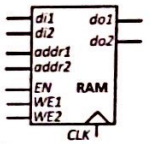
Examples

- To reduce critical path delay, address or the output can be registered

```
//rom with the registered address
module rom #(parameter AWL = 3, DWL = 4)
            (input CLK, EN,
             input [AWL-1:0] addr,
             output reg [DWL-1:0] data);
reg [AWL-1:0] addrq;
  always @(posedge CLK) if (EN) addq <= addr;
  always @(posedge CLK) begin
    if (EN)
     case(addrq)
     3'b000: data <= 4'b0010;
     3'b001: data <= 4'b0010;
     3'b010: data <= 4'b1110;
     3'b011: data <= 4'b0010;
     3'b100: data <= 4'b0100;
     3'b101: data <= 4'b1010;
     3'b110: data <= 4'b1100;
     3'b111: data <= 4'b0000;
     default: data <= 4'bXXXX;
     endcase
  end
endmodule
```

```
//Dual-port synchronous ROM using $readmemb
module dual_port_rom #(parameter AWL = 8, DWL = 8)
                      (input [(AWL):0] addr_a, addr_b,
                      input CLK,
                      output reg [(DWL-1):0] q_a, q_b);
reg [DWL-1:0] rom[2**AWL-1:0];

   initial
    // Read the memory contents from dual_port_rom_init.txt
     begin
       $readmemb("dual_port_rom_init.txt", rom);
     end
   always @ (posedge CLK) begin
     q_a <= rom[addr_a];
     q_b <= rom[addr_b];
   end
endmodule
```

Dual-port RAM in the read-first mode




```
//Synchronous RAM, dual-port, read-first mode
module ramRF #(parameter addWidth = 6,
                         dataWidth = 16)

localparam MEM_DEPTH = 1<<addWidth);
//localparam(s) cannot be redefined when the model is instantiated
// Note 2**N-1 ={N{1'b1}))
          (input CLK, EN, //EN" is the enable control signal for the RAM
           input WE1, WE2, //Write enable control signals for the two input ports
           input [addWidth-1:0] addr1, addr2,
           input [dataWidth-1:0] di1, di2,
           output reg [dataWidth-1:0] do1, do2);
reg [dataWidth-1:0] RAM [MEM_DEPTH-1:0];
  always @(posedge CLK) begin
     if (EN) begin
       if (WE1) RAM[addr1]<=di1;
       if (WE2) RAM[addr2]<=di2;
       RAM[addr2]<=di2;
     end
     do1 <= RAM[addr1];
    do2 <= RAM[addr2];
    end
  end
endmodule
```


Dual-port RAM

- A memory block can have several read and/or write ports

```
module dual_port_ram #(parameter AWL = 6, DWL = 8)
                      (input [(DWL-1):0] data_a, data_b,
                       input [(AWL-1):0] addr_a, addr_b,
                       input WEa, WEb, CLK,
                       output reg [(DWL-1):0] qa, qb);
reg [DWL-1:0] ram[2**AWL-1:0];
  always @ (posedge CLK) begin // Port A
    if (WEa) begin
      ram[addr_a] <= data_a;
      qa <= data_a;
    end
    else qa <= ram[addr_a];
  end
  always @ (posedge CLK) begin //port B
    if (WEb) begin
      ram[addr_b] <= data_b;
      qb <= data_b;
    end
    else qb <= ram[addr_b];
  end
endmodule
```

```
module ram_dual (output [7:0] q,
                 input [7:0] d,
                 input [6:0] addr_in,
                 input [6:0] addr_out,
                 input WE, CLK1, CLK2);
reg [6:0] addr_out_reg;
reg [7:0] q;
reg [7:0] mem [127:0];
  always @ (posedge CLK1) begin
    if (WE) mem[addr_in] <= d;
  end
  always @ (posedge CLK2) begin
    addr_out_reg <= addr_out;
    q <= mem[addr_out_reg];
  end
endmodule
```

RAM with bi-directional data bus

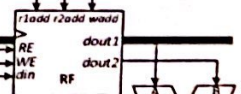
- Tristate drivers can be used to connect a memory module to the processor bus
- The data port is declared as an **inout** because it can be used both as an input and output. This reduces the number of wires needed, but tri-state drivers are required
- A new value is read from the memory when the read enable REn is asserted or when there is a change on the address lines addr
- The value read is stored in temporary register out which drives the data when REn is asserted. If REn is unasserted, data is tristated
- The data port is defined as an **inout**, allowing it be driven by the **assign** statement and also be used when writing into memory
- 'z is a shorthand for filling a bus of arbitrary length with Zs

```
module Memory (inout tri [7:0] data,
               input [15:0] addr,
               input WEn, REn, CLK);
reg [7:0] out;
reg [7:0] Mem [65535:0]; //64K byte memory
 always @(posedge CLK)
    if (WE== 0) Mem[addr] <= data;
 always @(~RE,addr)
    out = Mem[addr];
 assign data = (~RE)? out: 'z;
endmodule
```

RAM with bi-directional data bus

```
module RAM #(parameter AWL = 6, DWL = 32)
```

Register file example




```
module regfile #(parameter AW=8,DW=16)
  (input CLK, RE, WE,
```

Why does it only take 1 clock cycle to add/subtract

RAM with bi-directional data bus

```
module RAM #(parameter AWL = 6, DWL = 32)
            (input CLK, WE,
             input [AWL-1:0] addr,
             inout tri [DWL-1:0] data);
  reg [DWL-1:0] mem[2**AWL-1:0];
     always @(posedge CLK)
         if (WE) mem[addr] <= data;
      assign data = WE ? 'z : mem[addr];
endmodule
```

```
module bidirectionalRAM
  #(parameter AWL=10, DWL=8)
   (inout [DWL-1:0] data,
    input [AWL-1:0] addr,
    input CLK,
    input EN, nRW);
localparam MEM_DEPTH = 1<<AWL;
reg [DWL-1:0] mem [0:MEM_DEPTH-1];
  assign data = (nRW && EN) ? mem[addr] : {DWL{1'bz}};
  always @(posedge CLK)
    if (~nRW && EN) mem[addr] = data;
endmodule
```

- The memory depth-size MEM_DEPTH is protected from incorrect settings by declaring MEM_DEPTH as a **localparam**
  - The MEM_DEPTH parameter will only change if the AWL parameter is modified

- While you can instantiate RAM primitives in your HDL code. To keep your HDL code *technology independent*, write your Verilog codes for memories

Register file example

```
module regfile #(parameter AW=8,DW=16)
  (input CLK, RE, WE,
   input [AW-1:0] r1add, r2add, wadd,
   input [DW-1:0] din,
   output reg [DW-1:0] dout1, dout2);
reg [DW-1:0] reg_file[(2**AW)-1:0];
 //Register file in read-first mode
  always @ (posedge CLK) begin
    if(RE) begin
      if(WE) begin
        reg_file[wadd] <= din;
      end
      dout1 <= reg_file[r1add];
      dout2 <= reg_file[r2add];
    end
    else if (WE) reg_file[wadd] <= din;
  end
endmodule
```

Why does it only take 1 clock cycle to add/subtract and store the result?

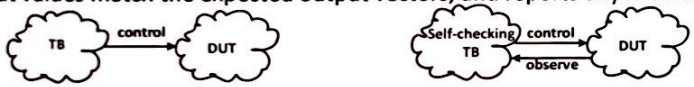
```
always @(*) begin
  if(EN) begin
    if(LENOUT>=LEN) begin
     OVF = 0;
     out = {((LENOUT - LEN) {res[LEN-1]}}, res};
    end
    else begin
     // OvF=if the bits truncated are not all the
     // same or do not match the sign bit
     OVF = ~(&res[LEN-1 : trunc_idx+ 1])
         || (~|res[LEN-1 : trunc_idx+1]));
     if(LENOUT == 1) out = res[LEN-1];
     else out = {res[LEN-1], res[trunc_idx:0]};
    end
  end
  else begin out = 0; OVF = 0; end
 end
endmodule
```

```
module adder_subtractor
  #(parameter LEN1=8,LEN2,LENOUT=9)
   (input signed [LEN1-1:0] in1,
    input signed [LEN2-1:0] in2,
    input EN, AS,
    output reg signed [LENOUT-1:0] out,
    output reg OVF);
  localparam LEN = ((LEN1 > LEN2) ? LEN1+1 : LEN2+1);
  localparam x_trunc = ((LENOUT > 2) ? LENOUT-2 : 0);
  localparam trunc_idx= ((LENOUT < LEN) ? x_trunc: 0);
  reg signed [LEN-1:0] res;
 always @ (*)
    if(AS) res = in1 - in2;  else   res = in1 + in2;
```

```
module datapath #(parameter DW=32, AW=5)
  (input [addWidth-1:0] Xr1add, Yr2add, Zwadd,
   input CLK, WE, RE, en , as);
wire [DW-1:0] din_wire, dout1_wire, dout2_wire;
  regfile #(.AW(AW),.DW(DW)) UUT_regfile
  (.CLK(CLK),.RE(RE),.WE(WE),.r1add(Xr1add),
   .r2add(Yr2add), .wadd(Zwadd), .din(din_wire),
   .dout1(dout1_wire), .dout2(dout2_wire));
  adder_subtractor #(.LEN1(DW), .LEN2(DW),
    .LENOUT(DW)) UUT_adder_subtractor
   (.in1(dout1_wire), .in2(dout2_wire), .EN(EN),
    .AS(AS), .out(din_wire));
endmodule
```


Self-checking testbenches

- Checking the output values visually is tedious. A *self-checking testbench* can be used to monitor output values and confirm that the correct outputs are produced during simulation. The testbench applies a set of test vectors, checks that the output values match the expected output vectors, and reports any mismatach



- Instead of writing a relatively long sequence of test vectors in a testbench, place the test vectors in a separate file and read the file into a memory. During testing, apply stimulus to the DUT from the memory
- **$fopen** system task opens an output file for writing and returns a 32-bit *file descriptor* handle that points to the file. If the simulation tool is unable to open the file for writing it will return a 0
- **$fmonitor** writes to a file whenever any one of its arguments changes and **$fdisplay** writes formatted data to a file whenever it is executed
- **$fclose** will close a file

```
integer file_descriptor;
File_descriptor = $fopen("fileName.ext");
```

```
integer fd;
initial begin
  fd = $fopen("my_chip_outputs");
  $fmonitor(fd, "time=%t out_bus=%h", $realtime, out_bus);
  $fclose(fd);
end
```

Testbench example

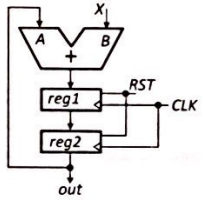
```
module tb_adder;
reg [3:0] a, b;
wire [3:0] sum;
reg  [7:0] tvArray[0:31];
  //Instantiate DUT
  adder uut (.a(a), .b(b), .sum(sum));
  initial begin
     //reading the test vectors from an ASCI file
     $readmemb("tb_adder.tv",tvArray);
     for (i = 0; i < 32; i = i + 1)
       #10 {a,b} = tvArray[i];
     if (sum != a + b) //if statement; similar to C if statement
        $fdisplay("Error, sum should be %h, is %h", a+b, sum);
  end
  #10 $finish;
  $fclose(handle);
  end

  //write the IO transactions onto a file
  initial begin: filIO
  integer handle;
    handle = $fopen("tb_adder.io");
    $fmonitor(handle, "time=%t a=%h , b=%h, sum=%h", $realtime, a, b, sum);
  end
endmodule
```

Example

```
module datapath #(parameter WL1=4,
                            WL2 = 16)
                 (input  CLK, RST,
                  input signed  [WL1-1:0] x,
                  output signed [WL2-1:0] out);
reg [WL2-1:0] reg1, reg2;
wire [WL2-1:0] addOut;
// sign extending x
assign addOut =  reg2 + { {(WL2-WL1){x[WL1-1]}}, x };
always @(posedge CLK) begin
  if (RST) begin
    reg1 <= 0;
    reg2 <= 0;
  end
  else begin
    reg1 <= addOut;
    reg2 <= reg1;
  end
end
assign out = reg2;
endmodule
```

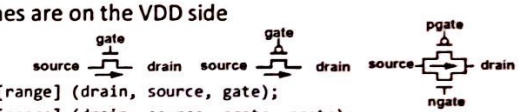
```
module tb_datapath;
parameter WL1=4, WL2 = 16;
reg CLK, RST;      // input
reg [WL1-1:0] x;  //input: test velues
wire [WL2-1:0] out;
// Clock generation
initial   CLK = 1'b0;
always #5 CLK <= ~CLK;
datapath #(.WL1 (WL1), .WL2 (WL2))
  UUT  (.CLK(CLK),.RST(RST),.x(x),.out(out));
initial begin
  @(posedge CLK) RST = 1;
  @(posedge CLK) RST = 0; x=4'b0001;
  @(posedge CLK) x=4'b0101;
  @(posedge CLK) x=4'b0011;
  @(posedge CLK) x=4'b1001;
end
initial  #100 $finish;
endmodule
```

```
reg  [20:0] inout_array[0:8];
reg [WL2-1:0] expOut; // expected output
reg [3:0] i; // for the loop index
initial begin
  $readmemb("D:/FPGAdev/Source/inp_out.txt",inout_array);
   for (i = 0; i <= 8; i = i + 1)
      @(posedge CLK) begin
        x = inout_array[i][20:17];
        RST = inout_array[i][16];
        expOut = inout_array[i][15:0];
        $display("out=%b expOut=%b", out, expOut);
        if (out != expOut)
           $display("Error in the computation!");
      end
end
```



Switch primitives

- Verilog supports primitive switches, **nmos**, **pmos**, **cmos**. A unidirectional transistor passes its input value to its output when it is switched on. Switching "on" means that logic values flow from input of a transistor to its output. Switching "off" means that the output of a transistor is at Z level regardless of its input value
- **nmos** and **pmos** are unidirectional devices representing nMOS and pMOS transistors, respectively. **cmos** is a unidirectional transmission gate with a true and complemented control lines. **nmos**, **pmos** and **cmos** gates conduct from source (input) to drain (output). The source of the **nmos** switches are on the GND side and the source of the **pmos** switches are on the VDD side
- The Verilog syntax is:

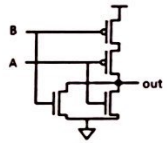



```
nmos #(<delay>) instance_name [range] (drain, source, gate);
cmos #(<delay>) instance_name [range] (drain, source, ngate, pgate);
```

- The **cmos** switch should be treated as combination of a **pmos** switch and an **nmos** switch, which have common data input and data output. ngate and pgate are normally complements of each other

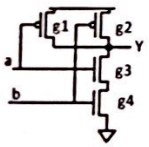
Verilog examples

```
module nor2 (input A, B,
             output out);
wire c;
    supply1 VDD; //logic 1: declare VDD net
    supply0 GND; //logic 0: declare ground net
    pmos P1 (c, VDD, B);
    pmos P2 (out, c, A);
    nmos N1 (out, GND, A);
    nmos N2 (out, GND, B);
endmodule
```



- Net declarations **supply0** and **supply1** declare GND and VDD nets

```
`timescale 1ns/10ps
module nand2 (input a, b, output y);
  supply0 GND;
  supply1 VDD;
  wire im1;
  pmos #(4) g1 (y, VDD, a);
  pmos #(4) g2 (y, VDD, b);
  nmos #(3) g3 (y, im1, a);
  nmos #(3) g4 (im1, GND, b);
endmodule
```

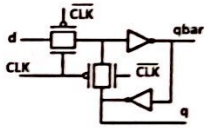



```
`timescale 1ns/10ps
module nand2_tb;
reg a, b;
wire y;
  nand2 U00(.a(a), .b(b), .y(y));
  initial begin
       a = 1'b0; b = 1'b0;
   #10 b = 1'b1;
   #10 a = 1'b1;
   #10 b = 1'b0;
   #10 b = 1'b1;
   #10 a = 1'b0; b = 1'b0;
  end
endmodule
```

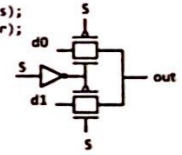
Verilog examples

```
module inv(output out,
           input in)
supply1 pwr;
supply0 gnd;
  pmos p1 (out, pwr, in);
  nmos n1(out, gnd, in);
endmodule

module tb_dlatch;
reg d, clk;
wire q, qbar;
 dff c1(.q(q), .qbar(qbar), .d(d), .clk(CLK));
 initial CLK = 1'b0;
 always #5 CLK = ~ CLK;
 initial begin
   @(posedge CLK) d = 1'b1;
   @(posedge CLK) d = 1'b0;
   @(posedge CLK) d = 1'b1;
 end
 initial $monitor($time," CLK = %b, d = %b, q = %b,qbar = %b ", CLK, d, q, qbar);
end
endmodule
```

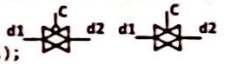
```
module dlatch (output q, qbar;
               input d, CLK)
wire e,nclk;
  inv  nt1(.out(nclk), .in(CLK));
  cmos c1 (e, d, CLK, nCLK);
  cmos c2 (e, q, nCLK, CLK);
  inv  nt2(.out(qbar), .in(e));
  inv  nt3(.out(q),    .in(qbar));
endmodule
```




```
module two_to_one_mux (output out, input s,d0,d1);
wire sbar;
  not U00 (sbar, s);
  cmos cmos_a (out, d0, sbar, s);
  cmos cmos_b (out, d1, s, sbar);
endmodule
```



- Verilog supports primitive bidirectional switches, **tranif0**, **tranif1**. They have three terminals. The first two are the bidirectional data pins (type of **inout**), the third is the control input. **tranif0** switch connects the two signals d1 and d2 only if the control signal C is logical zero




```
tranif0 #(<delay>) instance_name [range] (inout1,inout2,control);
```

Field-programmable gate arrays (FPGAs)

tranif0 #(<delay>) instance_name [range] (inout1,inout2,control);

Chapter 03 – Field-programmable gate arrays (FPGAs)

FPGAs

- The cost of an FPGA design is much lower than that of an ASIC and time to market for FPGAs is much faster. Applying design changes is possible in FPGAs
- FPGA applications:
  - *Custom silicon*: FPGAs are increasingly being used to implement designs that previously were realized by using only ASICs and custom silicon.
  - *Digital signal processing*: A large number of embedded DSP units and large amounts of on-chip RAM when coupled with the massive parallelism provided by FPGAs, would facilitate and outperform the fastest DSPs
  - *Embedded microcontrollers*: Low cost FPGAs with embedded soft processors combined with a selection of custom I/O functions, make them increasingly attractive for embedded control applications
  - *Physical layer communications*: FPGAs have been used as glue logic that interfaces between physical layer communication chips and high-level networking protocol layers. High-end FPGAs contain multiple highspeed transceivers, which means that communications and networking functions can be consolidated into a single device
  - *Reconfigurable computing:* Exploiting the inherent parallelism and reconfigurability of FPGAs make them suitable for "hardware acceleration" of computationally intensive applications

Field-programmable gate arrays (FPGAs)

- *FPGAs* are digital integrated circuits (ICs) that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks
- "Field-programmable" refers to the fact that its programming takes place "in the field" (as opposed to devices whose internal functionality is hardwired by the manufacturer
- *Programmable logic devices* (PLDs) are devices whose internal architecture is predetermined by the manufacturer, but are created in such a way that they can be configured in the field to perform a variety of different functions
- In comparison to an FPGA, PLDs contain a relatively limited number of logic gates, and the functions they can be used to implement are much smaller and simpler
- *ASICs* offer the ultimate in size (number of transistors), complexity, and performance, however, designing and building one is a time consuming and expensive process, while the final design is "frozen in silicon" and cannot be modified without creating a new version of the device
- FPGAs occupy a middle ground between PLDs and ASICs: their functionality can be customized in the field like PLDs, but they can contain millions of logic gates and be used to implement extremely large and complex functions

Antifuse-based FPGAs

- Three different major technologies are in use today for programming FPGAs: *antifuse*, *SRAM*, and *FLASH EPROM*
- In antifuse FPGAs, each configurable path has an associated link called an antifuse
  - In its unprogrammed state, an antifuse has a high resistance that can be considered as an open circuit; designed to permanently create an electrically conductive path
  - They can be selectively programmed by applying pulses of relatively high voltage and current to the device's inputs
  - Programming "grows" (not blown) a connection, known as a *via*, by converting the insulating silicon into conducting polysilicon to make permanent connections
  - A *fuse* starts with a low resistance and is designed to permanently break an electrically conductive path (typically when the current through the path exceeds a specified limit)
- Antifuse-based FPGAs are *one-time programmable* (OTP)
  - They are programmed off-line using a special programmer
  - Once an antifuse has been grown, it cannot be removed
- Antifuse-based devices are *nonvolatile* (their configuration data remains when the system is powered down)
  - They are immediately available as soon as power is applied to the system
  - They don't require an external memory to store their configuration data, which saves the cost and the board area

Pros and cons of antifuse-based FPGAs and its security

- Advantages of a OTP FPGA: It doesn't require external memory and configuration controller, lower power consumption, lower cost, and faster
- The biggest disadvantage is that their configuration is fixed: once the OTP FPGA is programmed, its implemented logic cannot be changed
- It is almost impossible to reverse-engineer the designs implemented on an anti-fuse FPGA
  - Even if the device is decapped (its top is removed), programmed and unprogrammed antifuses appear to be identical, and the fact that all of the antifuses are buried in the internal metallization layers makes reverse-engineering impossible
- Once an anti-fuse FPGA has been programmed, it is possible to set (grow) a special security antifuse that subsequently prevents any programming data (in the form of the presence or absence of antifuses) from being read out of the device
- Antifuse-based FPGAs are relatively immune to the effects of radiation
  - Of particular interest in military and aerospace applications
  - Once an antifuse FPGA has been programmed, it cannot be altered in this way

SRAM-based FPGAs

- In SRAM-based FPGAs, which is the dominant technology, once a value has been loaded into an SRAM cell, it will remain unchanged unless it is specifically altered or until power is removed from the system
- Relatively large silicon: Each SRAM cell requires six transistors
  - The SRAM cells are created using the same CMOS technologies used for logic, so no special processing steps are required in order to create these components
- FPGA's configuration data will be lost when power is removed from the system
  - SRAM-based FPGAs always need to be reprogrammed when power is first applied to the system using an external non-volatile memory (which has an associated cost and consumes board area) or using a USB cable
- FPGAs with a *fine-grained architecture* allow each logic block to be used to implement a simple function, such as logical gates
- In the case of a *coarse-grained architecture*, each logic block contains a relatively large amount of logic
  - Fine-grained implementations require a relatively large number of connections into and out of each block, while the amount of connections into the blocks decreases in coarse-grained architectures
  - This is especially important since the programmable interconnect resources account for most of the delays

Security of SRAM-based FPGAs

- It can be difficult to protect the *intellectual property* as the configuration file used to program the SRAM-based FPGA is stored in some form of external memory and the design can be reverse-engineered
- SRAM-based FPGAs supports *bitstream encryption*
  - The configuration data is encrypted before being stored in the external memory
  - The encryption key is loaded into a special register in the FPGA via its JTAG port
  - The key will be used by some associated logic to decrypt the incoming encrypted configuration bitstream as it's being loaded into the device
  - A battery on the circuit board is required to maintain the contents of the encryption key register in the FPGA when power is removed from the system
- The state of a configuration cell in a SRAM-based FPGA can be "flipped" by radiation (of which there is a lot in space)
  - For radiation-intensive environments, FFs are protected by triple redundancy, i.e., having three copies of each register and taking a majority vote
- Note that an FPGA can be configured multiple times (re-configured or re-programmed) using different bit files to perform a variety of tasks

FLASH-based FPGAs

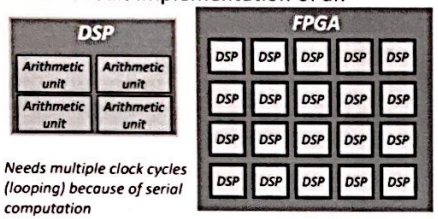
- FLASH-based FPGAs are based on *erasable programmable read-only memory* (EPROM) technology that allows devices to be programmed, erased, and reprogrammed with new data
- FLASH FPGAs are nonvolatile like antifuse FPGAs, but they are also reprogrammable like SRAM FPGAs
  - Once programmed, they would be "instant on" when power is first applied to the system
- FLASH FPGAs use a standard fabrication process like SRAM FPGAs and use lower power like antifuse FPGAs
  - Two-transistor E^2PROM and FLASH cells are smaller than SRAM cells
- FLASH FPGAs are relatively fast
- E^2PROM- or FLASH-based FPGAs can be configured off-line using a programmer
  - Some are *in-system programmable* (ISP), but their programming time is about three times that of an SRAM-based FPGA

FPGA applications

Logic blocks

FPGA applications

- The programming model of a FPGA is mainly based on *register-transfer level* (RTL) descriptions instead of software programming models used in microprocessors
  - This model of design capture is completely compatible with ASIC design
- FPGAs allow the designer to create a custom circuit implementation of an algorithm
- FPGAs can accelerate many computing tasks with abundant data and spatial parallelism by several orders of magnitude over a high-performance microprocessor
  - e.g., vector and matrix computations

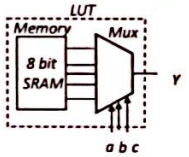


Needs multiple clock cycles (looping) because of serial computation

- FPGAs can be used for prototyping to explore promising architectures and evaluate hardware trade-offs
- FPGAs are widely used in final products for high performance computing, server farms, hardware accelerators, wireless base-stations, signal processing applications, military, automotive, and aerospace

Logic blocks

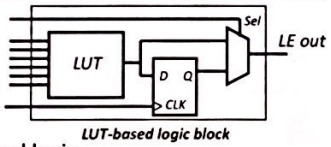
- An FPGA contains a relatively large number of *logic blocks*
  - A logic block is a generic term for a circuit that implements various logic functions
  - Each logic block has a fixed number of inputs and outputs and can be configured to implement a certain set of functions
- An FPGA logic block is called a *configurable logic block* (CLB) in Xilinx's devices
  - Configurable means memory controlled
- Logic blocks use *look-up tables* (LUTs)
  - Any combinational function of up to *n* variables can be implemented by an *n*-LUT, which is an *n*-input LUT with 2^n entries
  - Essentially, LUT is a truth table in which different combinations of the inputs implement different functions to yield output values



  - All logic functions with *n* or fewer inputs take the same number of logical resources and delay
  - LUTs are also called *function generators*
  - For a combinational circuit implemented by a LUT, its delay depends on neither the function type nor the input transition patterns
- Some FPGAs have a variety of different LUTs with different numbers of inputs
- LUTs support fine-grained programmability at the bit level

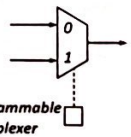
Logic blocks, slices, CLBs

- The flip-flops in the *logic blocks* can be used to implement sequential circuits
- The LUT and flip-flop may be used either separately or together



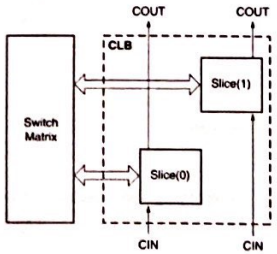
LUT-based logic block

- The multiplexer selects between the combinational logic output (LUT) and the sequential logic output (FF)
  - SRAM cells are used to store the configuration bits
  - This decision is static and is made at the time that the FPGA is configured (when the design is downloaded to the FPGA)
  - The design cannot change the multiplexer while it is running



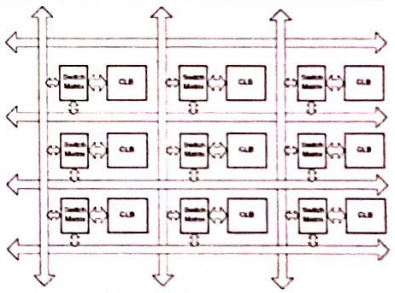
Programmable Multiplexer

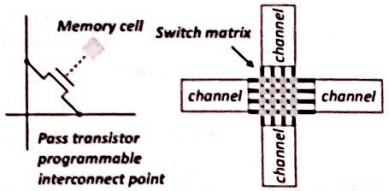
- A *slice* in Xilinx FPGAs contains four LUTs, eight registers, a carry chain, and multiplexers
  - Different devices have different slice architectures
- A *configurable logic block* contains two or more slices
- The dedicated carry signal is used to implement addition efficiently



Configurable routing resources

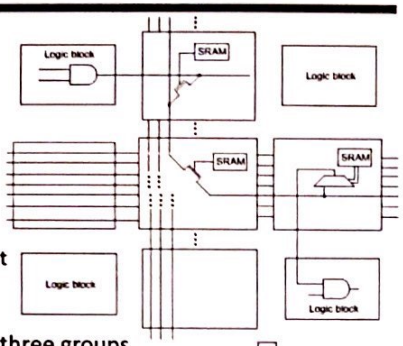
- Routing resources provide *programmable connectivity* between CLBs, *input-output blocks* (IOBs), *embedded memory blocks*, *DSP units*, and other modules
- Routing resources are organized into horizontal and vertical channels (several wires per channel)
- Connections between wires in the wiring channels and also from wiring channels to CLBs are made at programmable interconnection points
- *Configurable switch matrices* connect FPGA resources to routing channels
- The combination of FPGA logic blocks and routing resources is called *FPGA fabric*



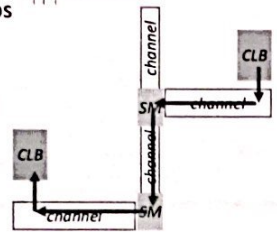


Example: SRAM-controlled programmable switch

- Switch memories must choose channels from source to destination and wires within the channels
- Interconnect fabric characteristics:
  - Switches add delay
  - FPGA interconnect has extra length
  - Some wires will not be utilized
  - Congestion will not be the same throughout the chip



- FPGA interconnects is broadly divided into three groups
  - Local or short wires for local CLB connections
  - Intermediate wires run for *L* logic blocks
  - Global wires run across the FPGA and is used for control signals



Configuration bit file

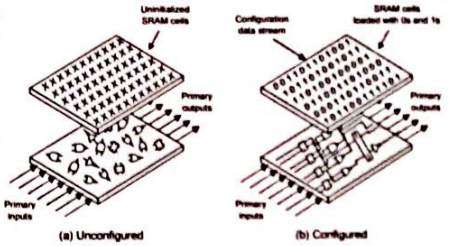
- *Logic synthesis* generates a netlist from HDL description of a logic function
- The *physical synthesis* tool transforms a netlist of generic logic gates into FPGA's cells. During physical synthesis:
  - *Technology map*: Map groups of generic combinational gates into LUTs
  - *Placement*: Assign LUTs and flip-flops to FPGA slices
  - *Routing*: connecting the signals between FPGA slices
- Once a design is successfully implemented (synthesized, mapped, placed and routed), a *configuration file* (also called *bit files* with .bit extension) is created
  - The bit file is loaded into the configuration memory cells that determines the behavior of the logic blocks, interconnects, and other configurable resources so the FPGA performs a specific function
  - After configuration, custom logic circuits described in HDL are mapped onto the reconfigurable fabric
  - In the case of SRAM-based FPGAs, the configuration file contains a mixture of *configuration data* (bits that are used to define the state of programmable logic elements, including logic resources, routing, and IO blocks, directly) and *configuration commands* (instructions that tell the device what to do with the configuration data)
- The bit file can be compressed for reducing the bitstream size and FPGA configuration speed, and can be encrypted to ensure data integrity

Programming (configuring) an FPGA

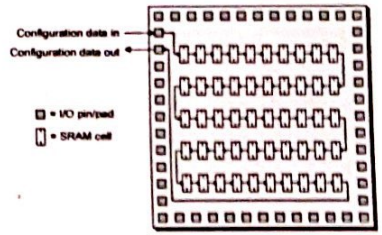
- When the configuration file is in the process of being loaded into the device, the information being transferred is referred to as the *configuration bitstream*
- The configuration data can be read from an external memory and an external microprocessor can control this operation
  - The microprocessor informs the FPGA when it wishes to commence the configuration process. It then reads *N*-bit (e.g., a byte) of data from an external memory
- A small number of *configuration mode pins* are used to inform the device which configuration mode is going to be used
  - Five *configuration modes*: 000 Serial load with FPGA as master; 001 Serial load with FPGA as slave; 010 Parallel load with FPGA as master; 011 Parallel load with FPGA as slave; 1xx Use only the JTAG port
- In the *serial load mode*, the microprocessor reads the data from the memory N-bit at a time and it then converts this data into a series of bits to be written to the FPGA
  - When there are multiple FPGAs on the board. Each could have its own dedicated external memory and be configured in isolation, or they can be cascaded (daisy-chained) together and share a single external memory. In this case, the first FPGA in the chain that is connected to the external memory would be configured in the serial master mode and the others would be in serial slave mode

Programming the FPGA

- Perceive all of the SRAM configuration cells as comprising a single (long) shift register
  - The beginning and end of this register chain are directly accessible from pins
  - FPGA has a configuration data-in pin and also a configuration data-out pin
  - This scheme uses fewer I/O pins on the FPGA



- FPGA configuration is performed by the *configuration controller* inside the FPGA
- On each FPGA power-up, or during a subsequent FPGA reconfiguration, a bitstream is read from the external non-volatile memory, processed by the configuration controller, and loaded to the internal configuration SRAM cells



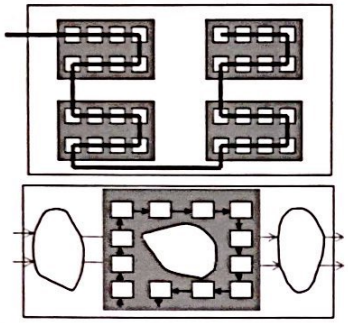
- A bitstream can be loaded into the configuration controller via a *Joint Test Action Group* (JTAG) port

master mode and the others would be in serial slave mode

Group (JTAG) port

JTAG boundary scan

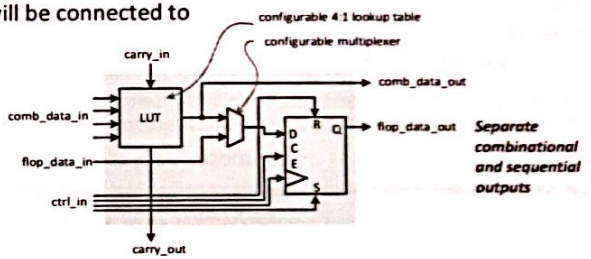
- JTAG was originally designed to implement the *boundary scan* technique for testing ICs and circuit boards
  - Shifting testing vectors into JTAG data in port and shift responses out from JTAG data out port
  - Each of the FPGA's remaining I/O pins has an associated JTAG flip-flop, where these FFs are daisy-chained together
  - In boundary scan using the JTAG port, data can be cocked into the JTAG FFs associated with the input pin serially, FPGA will operate on that data, and the result from this processing in the JTAG FFs will be clocked out of the JTAG port serially



- JTAG interface includes four pins: TDI (test data in), TDO (test data out), TCK (test clock), TMS (test mode select)
  - TCK is the JTAG clock signal and the other JTAG signals are synchronous to TCK
- A special command can be loaded into a special JTAG command register using the JTAG port's data-in pin
- The command will instruct the FPGA to connect its internal SRAM configuration shift register to the JTAG scan chain to program the FPGA

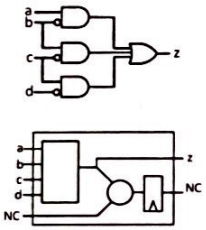
Mapping gates onto FPGA cells

- LUT is a small memory in which the contents of a truth table are written during device configuration
- To allocate gates and FFs to FPGA logic blocks, synthesis use various algorithms
  - The intuition behind the algorithm is to start with the FFs and outputs
  - Then, for each flip-flop and output it traverses backward through the fan-in gathering as much combinational circuitry as possible into the FPGA cell
  - Usually, this means that we continue as long as we have four or fewer inputs to the cell
  - Once we can no longer include more circuitry into an FPGA cell, we start with a new FPGA LB and continue to traverse backward through the fan-in
- Some signals have more than one target and hence, FPGA LBs will be connected to multiple destinations

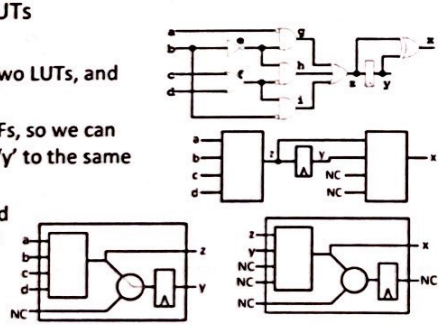


Examples

- (eq. 1) Without performing any algebraic optimizations, let's map this combinational circuit onto CLBs. Use NC (no connection) for any unused pins on the cells
  - The circuit has one output and four inputs and hence, it can fit into a single FPGA LB
  - We start at 'z' and work our way through the fanin until we have a set of gates with a total of four inputs

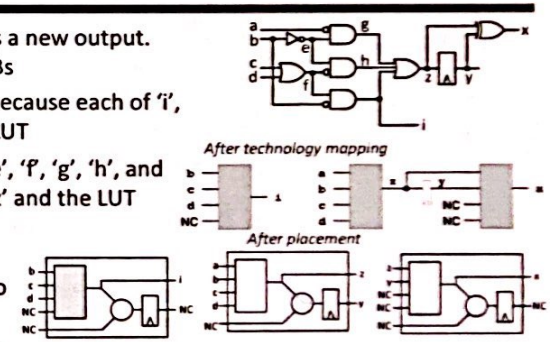


- (eq. 2) To map this circuit onto generic FPGA cells, we start with 'x' and 'z' as the outputs of LUTs and 'y' as the output of the FF
  - Thus, we will have one FF and at least two LUTs, and hence will need at least two FPGA cells
  - Four primary inputs driving 'z' and no FFs, so we can implement it using a single LUT (assign 'y' to the same FPGA cell as z)
  - For the LUT driving 'x', we stop at 'z' and 'y', because they are already outputs of other cells



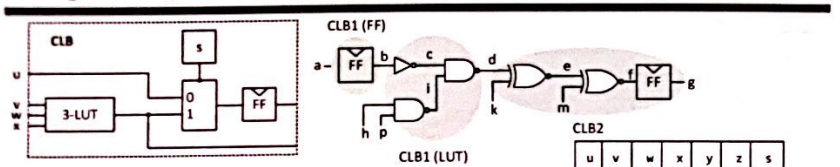
Example

- What if the signal 'l' becomes a new output. Map this circuit onto FPGA LBs
- We need at least three LBs, because each of 'i', 'x', and 'z' is the output of a LUT
- The best solution is to map 'e', 'f', 'g', 'h', and 'i' into both the LUT driving 'z' and the LUT driving 'w'
- Because each LUT can implement any function up to four inputs, this does not consume any additional area



- It might be tempting to use 'i' as an input the LUT driving 'z', similar to the way that we used 'z' as an input to the LUT driving 'x'
  - This mapping uses 'i' as an input to the LUT driving 'z' requires four cells and a slower circuit, because the path from 'c' and 'd' to 'z' would go through three LUTs, rather than just one as it does now

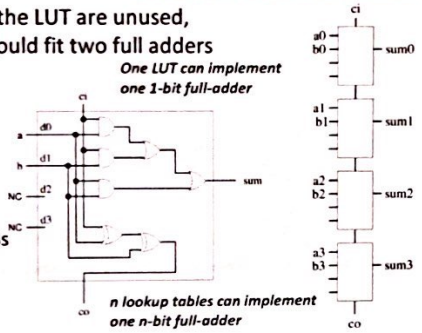
Design examples



CLB2						
u	v	w	x	y	z	s
a	b	h	p	b	d	0
x	d	k	m	g	nc	1

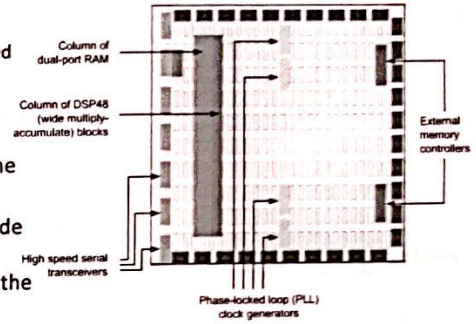
'x' denotes undefined and 'nc' denotes not connected

- How many LUTs are needed for a two-bit adder?
- With a full-adder, two of the inputs to the LUT are unused, which may tempt us to think that we could fit two full adders into a single lookup table
- However, two adders would require two outputs, one for each sum
  - Hence, a two-bit adder requires two LUTs, or two LBs
  - In general, an *n*-bit adder requires *n* LBs



FPGA microarchitectural components

- The main architectural components of FPGAs are logic and IO blocks, interconnect matrices, clocking resources, embedded memories, routing, and configuration logic
- *Memory*: Almost all FPGAs include dedicated memory blocks
  - This is because many FPGAs are fabricated on the same processes as SRAM chips
- *Arithmetic Circuitry*: FPGAs have dedicated circuits for multipliers and adders
  - Using these resources can improve significantly both the area and performance of a design
  - Utilizing dedicated arithmetic resources allows CLB to be saved for other computational logic
- *Microprocessors*: Some modern FPGAs include one or more dedicated microprocessors on the chip
- *Input/Output*: Some FPGAs include dedicated circuits for high bandwidth communication with the external devices

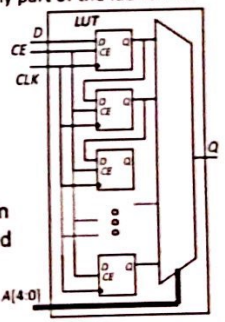


Intellectual properties (IPs)

- *Hard IP* comes in the form of pre-implemented blocks, such as microprocessor cores, gigabit interfaces, multipliers, adders, MAC functions
  - These blocks are designed to be as efficient as possible in terms of silicon area, performance, and power consumption (They are in *placed-and-routed netlist format*)
- *Soft IP* refers to a source-level library of high-level functions that can be instantiated in users' designs
  - For example, it is possible to configure a set of programmable logic blocks to act as a microprocessor, which is called a *soft core*
  - These functions are typically described at the RTL level of abstraction using a HDL
  - Soft cores are simpler and slower than their *hard core* counterparts as they are implemented using general CLBs
  - A soft core can be instantiated several times (limited by FPGA resources)
  - Typically a cycle-accurate C/C++ model for functional is also provided because such a model will simulate much faster than the netlist
- *Firm IP* also comes in the form of a library of high-level functions
  - Unlike their soft IP equivalents, however, these functions have already been optimally mapped, placed, and routed into a group of programmable logic blocks (possibly combined with some hard IP blocks, such as multipliers, etc.)
  - An IP might be tied to a particular FPGA vendor and device family

Embedded memories - Distributed memory

- LUT can be used as general-purpose logic resources
- Some FPGAs allow the SRAM cells forming the LUT to be used as a small block of RAM and is referred as *distributed RAM*
  - e.g., the 16 SRAM cells forming a 4-LUT, could be cast in the role of a 16×1 RAM
- Distributed RAMs are typically used for small RAMs and ROMs realized using LUTs configured in *shift register LUT* (SRL) mode (serial-in and serial-out) with bitwidth programmable from 1 to 16
  - Fastest memory on the FPGA as it can be instantiated in any part of the fabric that improves the performance of the implemented circuit
- SRL can significantly save resources (e.g., using a 4-LUT instead of using 16 FFs)
- Utilizing SRLs would eliminate extra routing resources that would have been lowered the performance otherwise
- There are three ways to incorporate a distributed RAM in the design: directly instantiation of one of the distributed RAM primitives, using COREGEN tool to generate a customizable core, and inferring from the code



Embedded memories – Block RAM

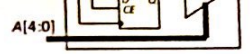
- Embedded memories in FPGAs include: dedicated *block memories* (BRAMs) and *distributed memories* (LUTs)

BRAM performance

- A BRAM has a minimum read latency of one clock cycle
  - Both BRAM and distributed RAM require synchronous data write, however, unlike

combined with some hard IP blocks, such as multipliers, etc.)
- An IP might be tied to a particular FPGA vendor and device family

RAM primitives, using COREGEN tool to generate a customizable core, and inferring from the code



Embedded memories – Block RAM

- Embedded memories in FPGAs include: dedicated *block memories* (BRAMs) and *distributed memories* (LUTs)
- *BlockRAMs* allow a relatively large amount of data to be stored in a small silicon area on the FPGA, within a dedicated and optimized memory element
  - The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations
- Both memories can be configured as a single- or dual-port RAM or ROM
  - In a RAM configuration, the data can be read and written at any time during the runtime of the circuit
  - In a ROM configuration, data can only be read during the runtime of the circuit
  - ROM's data is written as part of the FPGA configuration and cannot be modified
- Xilinx BRAM memories can hold either 18 kbits or 36 kbits
  - The number of these memories available is device specific
- BlockRAMs can be instantiated directly using one of its primitives and also can be inferred from the HDL description
- CORE Generator provides custom-sized BRAMs
- A BlockRAM can be initialized using a COE file
- Multiple BRAMs can be cascaded for building larger memories

BRAM performance

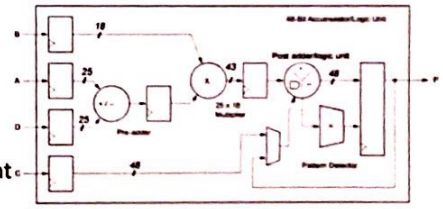
- A BRAM has a minimum read latency of one clock cycle
  - Both BRAM and distributed RAM require synchronous data write, however, unlike BRAM, distributed RAM has an asynchronous read mode with no read latency
- Using BRAM primitives in high speed designs requires careful floorplanning, because routing delays to and from a BRAM might consume significant part of the timing budget
- Also, BRAM has relatively slow clock-to-q time
  - Thus the clock rate when connecting a complex combinatorial logic to the data output of a BRAM is relatively low
  - Enabling BRAM output register will reduce clock-to-q time
  - The disadvantage of adding an output register is increased latency
  - Also, note that when BRAM reset signal is asserted, it resets the output registers and has no impact on the internal content of BRAM
- Distributed RAM, which is implemented using configurable LUTs, doesn't have this problem
  - Minimal impact on logic routing
  - They are very fast as they are tightly coupled to logic

First-in first-out (FIFO) RAMs

- Distributed RAM is constructed from the LUTs within the logic fabric
  - A significant number of LUTs are required to form a memory of comparable size to a BlockRAM
  - It is advantageous to implement small memories using distributed RAMs, both for resource efficiency and because their placement is more flexible (can be placed close to the logic that interact with them, which results in greater performance)
- Distributed RAMs are ideal for small coefficient storage and small *first-in first-out (FIFO) buffers*
- BRAMs also support optional FIFO behavior
- Each BRAM is 36-kbit, which can be configured into:
  - One 36-kb block RAM or FIFO or
  - Two independent 18-kb RAMs or
  - One 18-kb RAM and one 18-kb FIFO
- Synchronous or asynchronous read and write clocks
- Ideal for mid-sized FIFOs and buffers
- Two times performance increase over soft implementations using CLBs
- A FIFO has four flags: full, empty, programmable almost full, and programmable almost empty

DSP48 slices

- LUTs can be used to implement arithmetic operators of arbitrary lengths, but are most efficient for arithmetic operations with short wordlengths
  - Arithmetic circuits for long wordlengths can use a relatively large number of slices, with placement and routing factors resulting in sub-optimal clock frequencies
- *DSP48E1s* are dedicated resources for implementing high-speed arithmetic on signals with medium to long wordlengths
  - Comprise a pre-adder/subtractor, multiplier, and post-adder/subtractor with logic unit
  - Dedicated logic and routing connections translate to higher performance, better utilization of configurable resources, and lower power for implementing DSP functions
- The standard arithmetic wordlengths are adequate for most requirements, but they can also be extended by combining multiple DSP48E1s, if needed

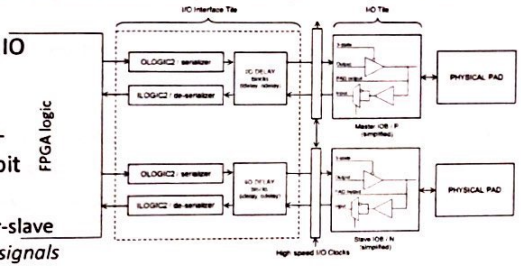


- Cascade pins are included to implement larger arithmetic operations by linking multiple slices together
- There are optional pipeline registers at several points within the DSP slice to maximize performance

Configurable I/O blocks

I/O standard	Application
TTL / CMOS	general purpose
SSTL/HSTL	SRAM / SDRAM / DDR interfaces
LVDS	high speed differential communication
TMDS	DVI/HDMI
PCI, I2C, SDIO	

- An FPGA's general-purpose I/O can be configured to accept and generate signals conforming to various IO standards
- These general purpose I/O signals will be split into a number of *I/O banks*
- Each bank of general-purpose I/Os (*I/O tiles*) can have separate power supply, thus supporting different sets of I/O standards
  - This allows the FPGA to be used to interface between different I/O standards
  - An I/O standard defines the signaling standard and voltage levels
  - Different I/O standards may use signals with voltage levels significantly different from the core voltage
- The *I/O interface tile* contains IO delay blocks for signal phase shaping (needed by DRAM interfaces), and both serial-to-parallel and parallel-to-serial bit stream conversion
  - IOB pairs may work in master-slave configuration for *differential signals*



Gigabit transceivers

- Early microprocessor-based systems use *buses* for interfacing data
- To transfer more data faster, wider buses (32 bits, 64 bits, etc.) were utilized
  - Wider buses requires more pins and a lot of tracks connecting the devices together
  - Routing tracks so that they all have the same length and impedance becomes increasingly challenging as boards grow in complexity
  - Managing *signal integrity* issues (such as susceptibility to noise) in bus-based tracks become increasingly difficult
  - Parallel bus reaches physical limitations when data rates begin to exceed 1 Gb/s and can no longer provide a reliable, cost-effective means for keeping signals synchronized
- *Serial I/O-based designs* offer various advantages over traditional parallel implementations including fewer device pins, reduced board space requirements, fewer *printed circuit board* (PCB) layers, easier layout of the PCB, smaller connectors, lower electromagnetic interference, and better noise immunity
- High-end FPGAs include dedicated high-speed *gigabit transceiver* blocks
- A *serializer/deserializer* (SERDES) manages incoming and outgoing data
  - Two separate SERDES are included on a FPGA: ISERDES is used for dividing incoming high-speed serial data to words and OSERDES performs the reverse operation
  - SERDES use one pair of *differential signals* (a pair of signals that always carry opposite logical values) to transmit data and another pair to receive data

Clock uncertainty

- Most digital systems use clocks to synchronize data transactions
- The maximum clock frequency determines the rate at which the data can be processed
- There are a number of issues with clocking a high performance digital system
- *Clock uncertainty* is the uncertainty in time in which a clock edge will appear
  - Sources of clock uncertainty includes fabrication skew, within die variations, and layout quality
- Clock uncertainty is determined by *clock skew*, *clock jitter* and *clock overhead*
- *Clock skew* represents the difference in delay of two identical clock signals arriving at two different locations on the chip (spatial separation)
  - Typically with reference to clock signals that should in theory switch simultaneously
  - Not only an a skew impact the achievable clock speed but in the worst case it can cause incorrect operation of the circuit
- *Clock jitter* is the clock edge inaccuracy. Jitter is a "dynamic" time difference of a signal with respect to an ideal signal
  - Jitter can result from the clock signal generation circuitry, various time-dependent noise events, or supply voltage variations
- Clock skew and jitter are growing as we move to nanometer processes

Clock buffers

- In synthesizable code, clock nets are normally considered ideal nets, with no delays
- Clock signal can be regenerated by *global clock buffers*
  - Global clock buffers allows glitchless clock signals with high drive strength
- FPGAs have various clock buffer types

Buffer type	*Primitive*
Single-ended input buffer	IBUFG, BUFGP*
Differential input buffer	IBUFDS, IBUFGDS, IBUFGDS_DIFF_OUT
Global buffer	BUFG, BUFGP*, BUFGCE, BUFGMUX, BUFGCTRL
Regional buffer	BUFR, BUFH, BUFHCE
Local clock buffer	BUFIO

*BUFGP includes both IBUFG and BUFG

- *Global buffers* are connected to the *dedicated routing network* to minimize skew
- In this examples a BUFG is used for an internal multiplexed clock circuit
  - Clock signals must come from a top-level port
  - Internally generated clocks are not automatically placed on a BUFG

```
module clock_mux (input data_in, sel,
                  input slow_clock, fast_clock,
                  output reg data_out);

reg clock;
wire clock_gbuff;
reg data_out;

buffg gbuff_for_mux (.in(clock),
                     .out(clock_gbuff));

always @(sel or fast_clk or slow_clk) begin
   if (sel) clock = fast_clk;
   else     clock = slow_clk;
end

always @(posedge clock_gbuff)
   data_out<=data_in;
endmodule
```

Clock buffers

PLL and DLL

- Main components of clocking generation: *crystal oscillators*, *phase-locked loops*

- Jitter can result from the clock signal generation circuitry, various time-dependent noise events, or supply voltage variations
- Clock skew and jitter are growing as we move to nanometer processes

- Internally generated clocks are not automatically placed on a BUFG

```
end

always @(posedge clock_gbuff)
   data_out<=data_in;
endmodule
```

Clock buffers

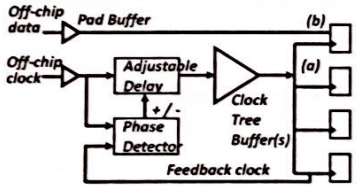
- Poor clock distribution can result in excessive clock skews and can reduce the maximum operating frequency
- To reduce the effect of clock skew, one approach is to balancing the delays of all the paths using clock buffers
- Clock buffers are automatically inserted during *physical synthesis*
  - During place and route, the *clock tree insertion tool* inserts the appropriate structure for creating as close to an ideal, balanced clock distribution network as possible
  - Synthesis tool automatically inserts a clock buffer whenever an internal clock signal reaches a certain fanout or an input signal drives a clock signal
  - Some synthesis tools require you to instantiate a global buffer in your code to use the dedicated routing resource if a clock is driven from a non-dedicated I/O pin
- *Differential signaling* is a method for transmitting information using two complementary signals
  - The technique sends the same signal as a differential pair of signals, each in its own conductor
  - The pair of conductors can be wires (usually twisted) or traces on a circuit board
  - The receiving circuit responds to the electrical difference between the two signals, rather than the difference between a single wire and ground
  - The opposite technique is called single-ended signaling

PLL and DLL

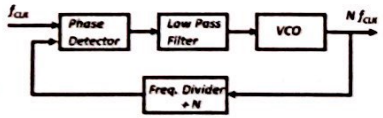
- Main components of clocking generation: *crystal oscillators*, *phase-locked loops* (PLLs) and *delay-locked loops* (DLLs)
- The main functions of the PLL are:
  - *Phase synchronizing* the system clock w.r.t. a reference clock (internal or external)
  - *Clock deskewing* (skew reduction): Reference clock is "aligned" to the feedback clock
  - *Frequency synthesis (multiplication)*: Clock multiplication can simplify board design because the clock path on the board no longer distributes a high frequency signal. By using a crystal-stabilized oscillator, clock frequencies can be multiplied internally to reduce board electromagnetic interference while maintaining stable control over the precise timing of clock edges
  - Jitter filtering
- The main functions of the PLL are:
  - *Delay synchronize* the system clock to a reference clock. Delay inserted between the input clock and feedback clock until the two rising edges align. When the edges from the input clock aligns with the edges from the feedback clock, the DLL "locks"
  - *Clock mirroring:* Cleaning and reconditioning incoming clock (duplicates incoming clock) for clock distribution
  - Multiplies and divides the clock signal
  - Input noise rejection and 50/50 duty cycle correction

DLLs and PLLs

- The delay (a) from the off-chip clock pad to the clock inputs of FFs is likely to be quite larger than the delay (b) from each data input pad to the *D* input of the first flip-flop in the data signal path
  - This difference in delays (a) and (b) can be equalized (modulo the clock period) using a DLL



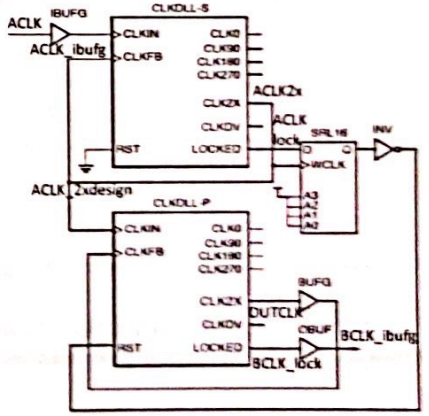
- A *phase detector* is an analog circuit that generates a voltage signal which represents the difference in phase between two signal inputs
- A lower-frequency clock generated at f_{CLK} using an off-chip crystal can be multiplied by a *PLL* to generate a new clock that has the desired higher frequency $N\ f_{CLK}$, where *N* is a suitable positive integer factor



- A *voltage-controlled oscillator* (VCO) is an analog oscillator circuit whose oscillation frequency is controlled by a voltage input

DLLs vs. PLLs

- Some FPGA clock managers are based on PLLs while others are based on DLLs
- DLLs offer advantages in terms of precision, stability, noise insensitivity, and jitter performance
  - DLL's are inherently stable by "only" aligning delay (not delay and frequency as in a PLL)
- PLLs can be implemented using either analog or digital techniques, while DLLs are by definition digital in nature
- DLLs can be cascaded to multiply and divide to get desired speed
  - In this example, two DLLs can be connected to generate a 4x clock
  - Input CLKIN can be driven by IBUFG, IBUF, or a BUFGMUX
  - Global clock input buffers (IBUFG) bring external clocks into CLKDLL



CLKDLL example

- Synthesis tools do not infer CLKDLLs. They must be instantiated

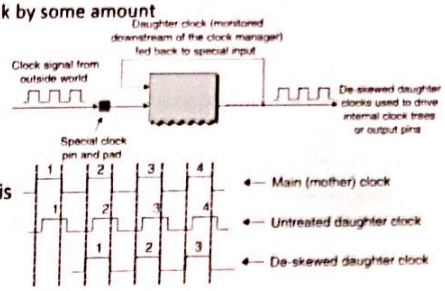
```
//////////////////////////////////////////////////////
// In this example ACLK's frequency is doubled,
// used inside and outside the chip.
// BCLK and OUTBCLK are connected in the board
// outside the chip.
//////////////////////////////////////////////////////
module clock_test(ACLK, DIN, QOUT, BCLK, OUTBCLK, BCLK_LOCK, RESET);
  input  ACLK, BCLK;
  input  RESET;
  input  [1:0] DIN;
  output [1:0] QOUT;
  output OUTBCLK, BCLK_LOCK;
  reg [1:0] QOUT;

IBUFG CLK_ibufg_A
   (.I (ACLK),
    .O(ACLK_ibufg)
    );
BUFG ACLK_bufg
   (.I (ACLK_2x),
    .O (ACLK_2x_design)
    );
IBUFG CLK_ibufg_B
   (.I (BCLK),      // connected to OUTBCLK outside
    .O(BCLK_ibufg)
    );
CLKDLL ACLK_dll_2x   // 2x clock
   (.CLKIN(ACLK_ibufg),
    .CLKFB(ACLK_2x_design),
    .RST(1'b0),
    .CLK2X(ACLK_2x),
    .CLK0(),
    .CLK90(),
    .CLK180(),
    .CLK270(),
    .CLKDV(),
    .LOCKED(ACLK_lock)
    );
CLKDLL BCLK_dll_OUT // off-chip synchronization
   (.CLKIN(ACLK_ibufg),
    .CLKFB(BCLK_ibufg), // BCLK and OUTBCLK is
                        // connected outside the
                        // chip.
    .RST(1'b0),
    .CLK2X(OUTBCLK),    //connected to BCLK outside
    .CLK0(),
    .CLK90(),
    .CLK180(),
    .CLK270(),
    .CLKDV(),
    .LOCKED(BCLK_LOCK)
    );

always @(posedge ACLK_2x_design or posedge RESET)
  begin
    if (RESET)
      QOUT[1:0] <= 2'b00;
    else if (ACLK_lock)
      QOUT[1:0] <= DIN[1:0];
  end
endmodule
```

IBUFG needs to be used for CLK signal

Global buffer needs to be used for CLKFB in CLKDLL

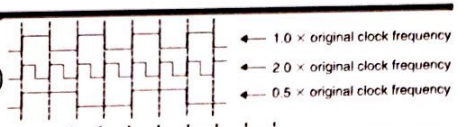
Digital clock manager (DCM)

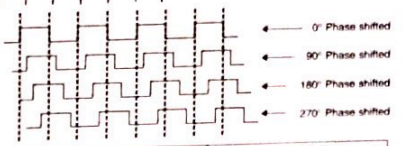
- The *DCM primitive* is used to implement DLL, digital frequency synthesizer, or a digital phase shifter. It performs:
  - *Clock deskewing* via a contained DLL: Eliminate input skew between PAD and internal clock input pins; Eliminates clock distribution delay
  - *Frequency synthesis* by integer multiplication and division
  - *Clock phase shifting*: Provides four quadrature phases of source clock
- Assume a *daughter clock* that has been configured to have the same frequency and phase as the *mother clock* signal coming into the FPGA
  - The clock manager will add some element of delay to the signal thus the daughter clock will lag behind the input clock by some amount
  - The difference between the two signals is known as *skew*
- Example of deskewing with reference to the mother clock
- The *prime* (zero phase-shifted) *daughter clock* will be treated in this way, and all of the other daughter clocks will be phase aligned to this prime daughter clock



DCM operation examples

- Example of frequency synthesis
- Example of phase shifted (delayed) clocks with respect to each other can be generated
  - Clock managers allow selecting from fixed phase shifts of 120° and 240° (for a three-phase clocking scheme) or 90°, 180°, and 270° (if a four-phase clocking scheme is required), or to configure the exact amount of phase shift





Phase (degrees)	% Period Shift
0	0%
90	25%
180	50%
270	75%

  - CLK0, CLK90, CLK180, CLK270 are each phase shifted by 1/4 of the input clock period relative to each other (i.e., phase shifts of 0, 90, 180 and 270 degrees)
- How does phase shifting relate to period of clock?

References

- *Digital Design and Computer Architecture*, D. Harris and S, Harris.
- *Computer Organization and Design: The Hardware/Software Interface*, D. A. Patterson and J. L. Hennessy.
- *CMOS VLSI Design: A Circuits and Systems Perspective*, N. Weste and D. Harris.
- *Verilog HDL: A Guide to Digital Design and Synthesis*, S. Palnitkar.
- *Verilog and SystemVerilog Gotchas*, D. Mills and S. Sutherland.
- *SystemVerilog for Design*, S. Sutherland.
- *FPGA Prototyping by Verilog Examples*, P. P. Chu.
- *Digital VLSI Design with Verilog: A Textbook from Silicon Valley Technical Institute*, D. Thomas and J. Williams.
- *Digital Integrated Circuits: A Design Perspective*, J. Rabaey, A. Chandrakasan, and B. Nikolic.
- *Computer Arithmetic: Algorithms and Hardware Designs*, B. Parhami.
- *VLSI Digital Signal Processing Systems: Design and Implementation*, K. K. Parhi.
- *Low Power Digital CMOS Design*, A. P. Chandrakasan and R. W. Brodersen.
- *Static Timing Analysis for Nanometer Designs: A Practical Approach*, J. Bhasker and R. Chadha.
- *DSP Architecture Design Essentials*, D. Marković and R. W. Brodersen.
- *Digital Design og Signal Processing Systems: A Practical Approach*, S. A. Khan.
- *An ASIC Low Power Primer: Analysis, Techniques and Specification*, R. Chadha and J. Bhasker.
- *Principles of Verifiable RTL Design*, L. Bening and H. Foster.
- *Principles of VLSI RTL Design: A Practical Approach*, by S. Churiwala and S. Garg.
- *Energy Efficient Microprocessor Design*, T. D. Burd and R. W. Brodersen.
- *Digital System Clocking: High-Performance and Low-Power Aspects*, Oklobdzija, V. M. Stojanovic, D. M. Markovic, N. M. Nedovic.

- Special acknowledgements to the many faculty members from highly-ranked universities for providing outstanding notes.

This is a partial reference list. More references will be added to this list.

- *Energy Efficient Microprocessor Design*, T. D. Burd and R. W. Brodersen.
- *Digital System Clocking: High-Performance and Low-Power Aspects*, Oklobdzija, V. M. Stojanovic, D. M. Markovic, N. M. Nedovic.

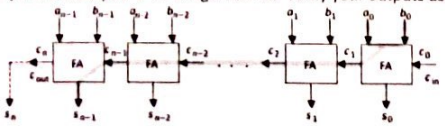
- **Special acknowledgements to the many faculty members from highly ranked universities for providing outstanding notes.**

This is a partial reference list. More references will be added to this list.

CompE-470: Digital Circuits Assignments

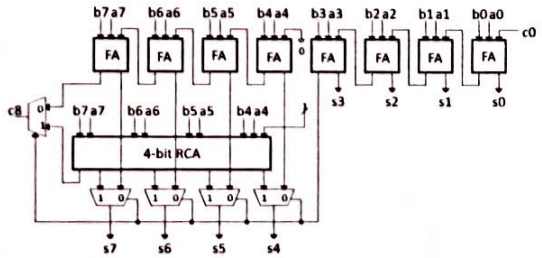
Assignment 1: Gate-level, Dataflow, and Structural Level HDL Modeling

1. The following block diagram shows an *n*-bit ripple carry adder (RCA) implemented using *n* full-adders (FAs). Describe a four-bit RCA that receives two four-bit values "a" and "b" and generates a five-bit output in Verilog HDL. Describe a FA at the gate-level, dataflow level, and behavioral level. Describe your RCA at the structural level by instantiating two FAs at the gate-level and two FAs at the dataflow level. Write a testbench for your RCA and on the simulation output waveform, show the value of "a", "b", and the output "s" in integer format. Verify your outputs using the following test vectors.



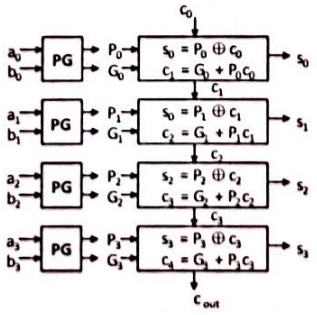

```
a = 4'b0000; b = 4'b0000;
a = 4'b1111; b = 4'b1111;
a = 4'b1110; b = 4'b0111;
a = 4'b1010; b = 4'b0101;
a = 4'b1000; b = 4'b1000;
```

2. Describe the following eight-bit carry-select adder (CSA) at the structural level by instantiating three RCA that you have developed in question 1. Using the same set of testvectors in question 1, compare the outputs of your CSA design with the outputs of your RCA in one testbench.



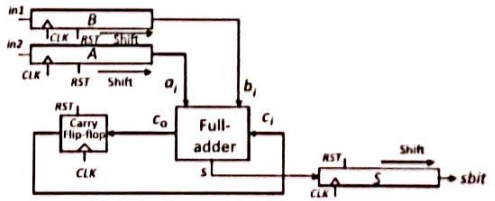
Assignment 2: Gate-level, Dataflow, and Structural Level HDL Modeling

1. Write a Verilog description for a four-bit carry-lookahead adder (CLA). The PG units produces Gi = aibi and Pi = ai ⊕ bi. The outputs are si = Pi ⊕ ci and ci+1 = Gi + Pici. You can choose any levels of abstraction for describing your design. Using the same set of testvectors in question 1, compare the outputs of your CLA with the outputs of your RCA and CSA in one testbench.



Assignment 3: Behavioral-Level Modeling

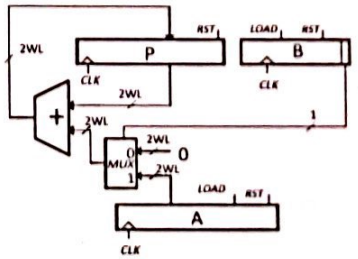
1. The following bit-serial adder uses a full adder (FA), a carry flip-flop (FF), and three WL-bit shift registers A, B, and S. Describe a FA, a carry FF, and a shift register at the behavioral level.



Write a top-level Verilog module at the structural level for this bit-serial adder. Write a testbench for your bit-serial adder. Using the same set of testvectors in question 1 in Assignment 1, show the value of "a", "b", and the output "s" in integer format and compare the outputs of your adder with the outputs of your RCA, CSA, and CLA in one testbench.

2. Describe an unsigned shift and add serial multiplier in Verilog HDL at any abstraction levels. "A" is a shift-left register (2WL bits), "B" is a shift right register (WL bits), and "P" register has no shifting function. The partial products are generated every clock cycle and are stored in "P". Each newly generated partial product is added to previously accumulated partial product in register "P". The reset signal RST and the load signal LOAD are active-high and synchronous. When LOAD is one, the registers load the input values into the two registers. Otherwise, "B" is shifted right one-bit position and "A" is shifted left one-bit position every clock cycle. Note that the data input to registers "A" and "B" are not shown. The algorithm for the multiplier is as follows:

1. 1. A ← multiplicand, B ← multiplier, P ← 0;
2. 2. If LSB of B==1 then add A to P else add 0;
3. 3. Shift B right one-bit position and shift A left one bit position;
4. 4. Repeat steps 2 and 3 WL-1 times;
5. 5. P has product of A and B.

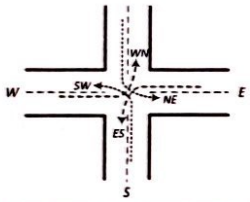


As an example, consider the multiplication of two unsigned 4-bit numbers, multiplicand 5 (0101) and multiplier 11 (1011):

```
  0101 ×              0101 ×
  1011                1011
  ----                ----
  0101                0000  Initial partial product
 0101                 0101  add first multiple
0000                  ----
0101                  0101
-------               00101 shift right
00110111 = 55         0101  add second multiple
                      ----
                      01111
                      001111 shift right
                      0000  add third multiple
                      ------
                      001111
                      0001111 shift right
                      0101  add fourth multiple
                      ------
                      0110111
```


Assignment 4: Finite State Machines in Verilog HDL

1. Consider a traffic intersection with the following default sequence of traffic lights operations:
First: West-to-north (WN) and East-to-South (ES) turning traffic.
Second: North-to-East (NE) and South-to-West (SW) turning traffic.
Third: East and West (EW) through traffic.
Fourth: North and south (NS) through traffic.



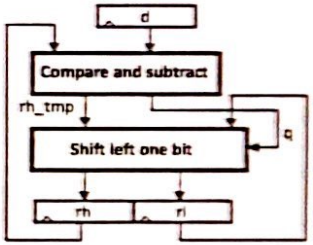
Here are the specifications of the traffic light controller: (1) All traffic lights go through the green, yellow, and red sequence. The through traffic has regular round traffic lights and the turning traffic has arrows. (2) The NS through traffic light remains green unless a car is in a turning lane or on the EW route. If there is no car in the turning lanes or EW route, the traffic light should remain red for those routes and the sequence is skipped for those routes. As an example, assume that there is traffic going NS and a car appears at the intersection that is going EW. The sequence would skip the NE/SW turning light and go straight to EW.

Write a Verilog HDL description for this traffic light controller and test all possible traffic scenarios in your testbench.

Assignment 5: Datapath and Control Unit in Verilog HDL

1. In an unsigned division, the dividend is divided by the divisor and the answer is the quotient. The algorithm can be described in four steps: (i) Zero- extend the dividend and align the divisor to the leftmost bit of the extended dividend; (ii) If the corresponding dividend bits are greater than or equal to the divisor, subtract divisor from that portion of the dividend bits and make the corresponding quotient bit 1. Otherwise, keep the original dividend bits and make the quotient bit 0; (iii). Append one additional dividend bit to the previous result and shift the divisor to the right one position; and (iv) Repeat steps 2 and 3 until all dividend bits are used.
An example of the division of two 4-bit unsigned integers is shown below. In the given datapath, the divisor is initially stored in the register "**d**" and the extended dividend is stored in the registers "**rh**" and "**rl**" registers. In each iteration, the **rh** and **rl** registers are shifted to the left one position. This corresponds to shifting the divisor to the right. Compare **rh** and **d** and perform subtraction if **rh≥d**. After iteration through all dividend bits, the result of the last subtraction is stored in **rh** and becomes the remainder of the division, and all quotients are shifted into **rl**. Note that the remainder should not be shifted in the last iteration.
Describe a WL-bit unsigned division unit in Verilog using a four-state finite state machine. The "idle" state is the reset state. In the "operation" state, the **rh** and **rl** registers are shifted to the left. The "last" state denotes the state for the last step of the algorithm. Finally, the "done" state is for asserting an "avail" signal indicating that the division is complete. Write a testbench for your design and show the quotients and remainders for 14/3 and 17/2.

```
                00110    quotient
divisor 0010 |00001101    dividend
             0000
             ----------------
             0001
             0000
             ----------------
              0011
              0010
             ----------------
              0010
              0010
             ----------------
              0001   remainder
```



Assignment 6: Memory Design in Verilog HDL

1. In a last-in-first-out (LIFO) buffer or a stack, the last value written into a stack is the first one to be read out. In hardware, a stack is an array of registers in which data is written into the stack using the push operation and is read from the stack from the pop operation. The number of data words or stack depth **DEPTH** and the length of data word **WL** are two parameters of the stack Verilog module. The synchronous stack has a single clock port **CLK** for both data-read and data-write operations. No data should be written into a stack when it is full, and no data should be read from the stack when it is empty. When the stack is full, the output signal **full** is asserted and when the stack is empty, the output signal **empty** is asserted. Otherwise they stay at zero. If further pushes are attempted when the stack is full or further pops are attempted when the stack is empty, the **error** output signal goes to high. Otherwise it stays at zero.

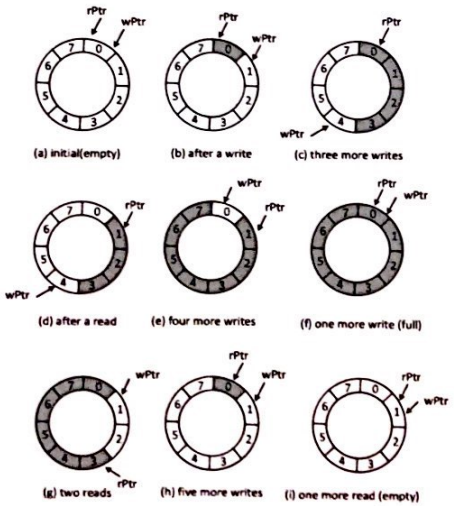
If **wReq** signal is high and the stack is not full, the data present on **din** is written (pushed) onto the top of the stack on the next rising edge of the clock. If **rReq** signal is high and the stack is not empty, the data is read (popped) from the top of the stack and sent onto the **dout** on the next rising edge of the clock. If the read and write requests are both active in a clock cycle, then first read (pop) operation will be done then write (push).

Only the item on the top of the stack (TOS) is accessible. Pushing onto the stack starts from address zero and the top of the stack is always addressed by a stack pointer register **sp**. Hence, the stack pointer **sp** starts at the zero address and increments by one as needed.

Design your parametric synchronous stack in Verilog. Perform the following operations in the order listed below in your testbench and demonstrate the content of the stack, control and data signals in every clock cycle: push 1; push 2; push 4; push 5; push 7; push 4; pop; push 3; push 1; pop; pop; pop; pop; pop; push 2;

Name	I/O	Width	Description
CLK	I	1	Clock input to the stack.
RST	I	1	Active-low synchronous reset input to the stack read and write logics.
din	I	WL	WL-bit data input to FIFO.
dout	O	WL	WL-bit data output from the stack.
rReq	I	1	Read request.
wReq	I	1	Write request.
Full	O	1	Stack is full. New data can't be written.
Empty	O	1	Stack is empty. Data can't be read.

2. In a first-in-first-out (FIFO) buffer, the first data item written onto the FIFO is the first one that is read. One approach to implement a FIFO is to utilize a circular buffer with two pointers. The write-pointer **wPtr** points to the tail of the queue and the read-pointer **rPtr** points to the head of the queue. The pointer advances one position for each write or read operation. A FIFO buffer contains two status signals, **full** and **empty**, to indicate that the FIFO is full and empty, respectively, which occur when **rPtr=wPtr**. The operation of an eight-word FIFO is shown below.



Design and verify a parametric circular FIFO in Verilog HDL. The FIFO has **DEPTH** words and each word is **WL** bits wide. Upon reset, control signals are adjusted. The FIFO has a data input port **din** and a data output **dout** port. When the **wReq** input signal is asserted, the input on the **din** data port is written into the FIFO. When the **rReq** input signal is asserted, the item pointed by **rPtr** is read onto the **dout** port. When the FIFO is full, the output signal **full** is asserted and when the FIFO is empty, the output signal **empty** is asserted. Otherwise they stay at zero. If further writes are attempted when the FIFO is full or further reads are attempted when the FIFO is empty, the **error** output signal is asserted. Otherwise it stays at zero. Read and write operations can occur in the same clock cycle, which means that a new value can be written to a location and read from the top of the FIFO. If the buffer is full and **wReq= rReq=1**, then the buffer acts in first-read mode, which means it reads first and then writes, without asserting the **error** signal. If buffer is empty and **rReq= wReq=1**, the **error** signal will be asserted.

Describe your parametric synchronous FIFO in Verilog. Perform the following operations in the order listed below in your testbench and demonstrate the content of the FIFO, control and data signals in every clock cycle: write 1; write 2; write 4; write 5; write 7; write 4; read; write 3; write 1; read; read; read; read; read; write 2;

Assignment 7: CMOS Circuit Design and Simulation

1. Design a two-bit adder and a two-bit subtractor and perform mixed-signal simulation. Demonstrate your simulation outputs for the following input values.

   - `A = 2'b00, B=2'b11, Cin=1'b0;`
   - `A = 2'b10, B=3'b01, Cin=1'b1;`
   - `A = 2'b00, B=3'b10, Cin=1'b0;`
   - `A = 2'b11, B=3'b10, Cin=1'b0;`