EC3057D Modeling and Testing of Digital Systems

Winter 2021

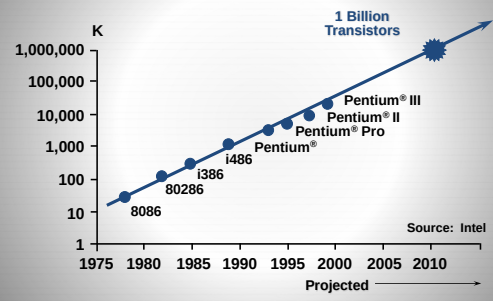
Reference books

- Ciletti M.D., *Advanced digital design with the Verilog HDL*, Second Edition, Prentice Hall, 2010.
- Palnitkar S.,*Verilog HDL: A guide to digital design and synthesis,* Prentice Hall; 2003.
- Charles Roth, Lizy Kurian John, ByeongKil Lee, *Digital systems design using Verilog*, First Edition,Cengage Learning, 2014.
- J. Bhasker, *Verilog HDL Synthesis: A Practical Primer,* B. S. Publications, 2001.
- Bushnell M, Agrawal V., *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits,* Springer Science & Business Media, 2004.

Introduction

- **Moore's Law**
  - In 1965, Gordon Moore noted that the number of transistors on a chip doubled every 18 to 24 months.

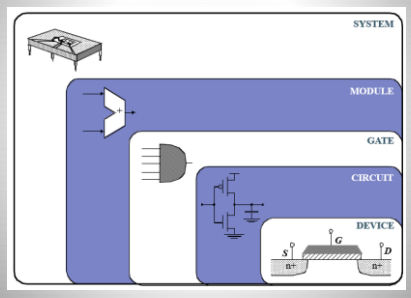
Transistor count



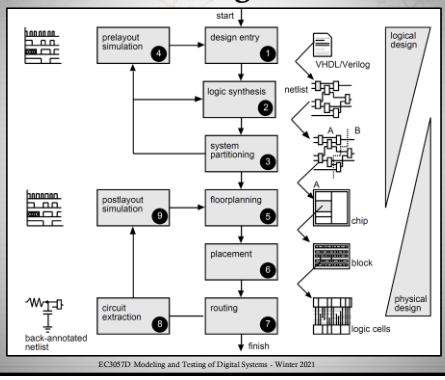
Computer-Aided Design

- Every new generation can integrate 2x more functions per chip
  - Chip price does not increase significantly
  - Cost of a function decreases by 2x
- However,
  - Design engineering population does not double every two years.
  - How to design much more complex chips (with more and more functions)?
- Great need for ultra-fast design methods
  - Design Automation (Computer-Aided Design)

Design Abstraction Enables CAD



VLSI Design Flow



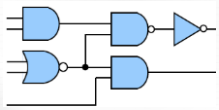
Hardware Description Languages

Evolution of Computer Aided Digital Design

- Digital circuits were designed with vacuum tubes and Transistors
- ICs were then invented
  - SSI, MSI, LSI, VLSI
- Due to complexity of circuits it was not possible to verify circuits on board.
- CAD do verification and design of VLSI circuits, also do automatic placement and routing of circuit layouts.
- Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on Chip

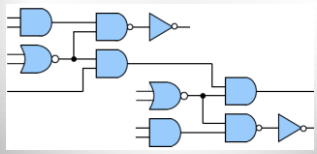
Hardware Description Languages - Introduction

- In the beginning, designs had only a few gates, and thus it was possible to verify these circuits on paper or with breadboards.



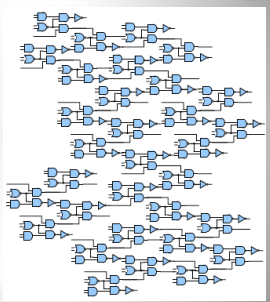
HDL – Introduction (Cont..)

- As designs grew larger and more complex, verification using paper or breadboards became impossible.
- Hence, designers began to use gate-level models described in a Hardware Description Language to help with verification before fabrication



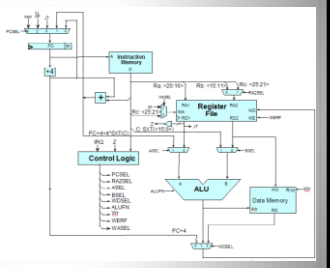
HDL – Introduction (Cont..)

- When the number of gates in the designs are in the ranges of 100,000 gate designs, these gate-level models also became complex for the functional specification.



HDL – Introduction (Cont..)

- Designers again turned to HDLs for help – abstract behavioural models written in an HDL provided both a precise specification and a framework for design exploration.



HDL – Introduction (Cont..)

- Better be **standard** than be proprietary.
- Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates.
- Digital circuits could be described at a register transfer level (RTL) by use of an HDL.
- The designer had to specify how the data flows between registers and how the design processes the data.
- The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.

HDL - Introduction

- HDLs also began to be used for system-level design.
- HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic).
- Can describe a design at some levels of abstraction
- Can be used to document the complete system design tasks
  - Testing, simulation, ..., related activities
- Comprehensive and easy to learn
- Most popular logic synthesis tools support verilog HDL
- All fabrication vendors provide Verilog HDL libraries.

What is Verilog HDL?

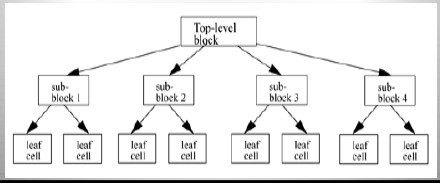
- Invented by Philip Moorby in 1983/84.
- The original standard was IEEE 1364
  - The first version was published in 1995.
  - Revised in 2001 and 2005.
- Allows different levels of abstraction to be mixed in the same design.
- Single language for design and testbench.

Contd..

- Simulated to check functionality.
- Synthesized (netlist generated).
- Built-in primitives, logic function
- User-defined primitives
- Built-in data types
- High-level programming constructs

Hierarchy of design methodologies

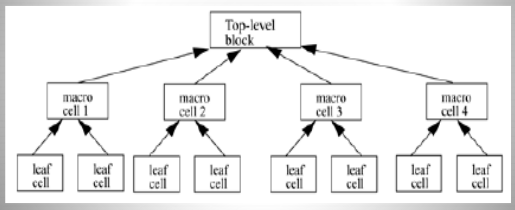
- **Top-Down Design**
  - This style is convenient and efficient.
  - Start with system specification
  - Decompose into subsystems, components, until indivisible
  - Realize the components
  - But it is very difficult to follow a pure top-down design. Hence, most designs are mix of both the methods.



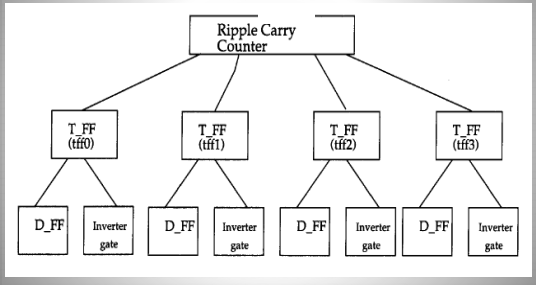
Hierarchy of design methodologies

- **Bottom-Up Design**
  - The traditional method of electronic design is bottom-up (designing from transistors and moving to a higher level of gates and, finally, the system).
  - But with the increase in design complexity traditional bottom-up designs have to give way to new structural, hierarchical design methods.
  - Start with available building blocks
  - Interconnect building blocks into subsystems, then system
  - Achieve a system with desired specification

- Bottom – up Design



Hierrarchy



Levels of Abstraction

- **Behavioral Level :**Module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Very similar to C programming
- **Dataflow Level**: Module designed by specifying dataflow. The designer is aware of how data flows between hardware registers and how the data is processed in the design
- **Gate Level:** Module implemented in terms of logic gates like (and ,or) and interconnection between gates
- **Switch Level:** Module implemented with switches and interconnects. Lowest level of Abstraction

Levels of Abstraction

- **Behavioral Level :**Module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Very similar to C programming
- **Dataflow Level**: Module designed by specifying dataflow. The designer is aware of how data flows between hardware registers and how the data is processed in the design
- **Gate Level:** Module implemented in terms of logic gates like (and ,or) and interconnection between gates
- **Switch Level:** Module implemented with switches and interconnects. Lowest level of Abstraction

Basic Unit (Module)

- A module is the basic building block in Verilog.
- In Verilog a module is declared by the keyword **"module"**.
- Elements are grouped into modules to provide the common functionality that is used at many places in the design.
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs).
- A corresponding keyword **"endmodule"** must appear at the end of the module definition.

Module- Basic building block

```
module <module name> (<port list>);
    <declarations>;      // input, output, inout
                         // wire, register, etc.
    <statements>;        // initial, begin, end, always
                         // dataflow statements
endmodule
```


Structure of a module

- The <**module name**> is an identifier that uniquely names the module.
- The <**port list**> is a list of input, inout and output ports which are used to connect to other modules.
- The <**declarations**> section specifies data objects as registers, memories and wires as wells as procedural constructs such as functions and tasks.
- The <**statements**> may be initial constructs, always constructs, continuous assignments or instances of modules.

Typical structure of a module

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
```
→ Module Name and Ports List

```
output out;
input i0, i1, i2, i3, s1, s0;
reg out;
```
→ Declarations

```
always @ (s1, s2, i0, i1, i2, i3)
begin
    case ({s1, s0})
        2'd0: out = i0;
        2'd1: out = i1;
        2'd2: out = i2;
        2'd3: out = i3;
        default:$display ("invalid control signals");
    endcase
end
```
→ Statements

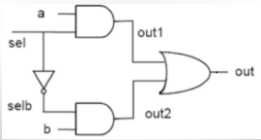
```
endmodule
```
→ End of Module

Modules (Cont…)

- Modules CANNOT be nested.
- The process of creating objects from a module template is called **instantiation** and the objects are called **instances**.
- One module can instantiate another module.
- Module instantiation is like creating actual objects (Instances) from the common template (module definition).
- Each instance of module has all the properties of that module.
- Module instantiations are used for:
  - connecting different parts of the designs, and
  - connecting test bench to the design.

Multiplexer 2x1

sel	out
0	b
1	a




```
1 module mux2x1_str(out,a,b,sel);
2 //declaration of input and output
3 output out;
4 input a,b,sel;
5 //declaration of reg, wire and other data types
6 wire selb,out1,out2;
7
8 //Statements describing the functionality of the design
9 //Structural model includes instantiations
10 //The order in which the modules are instantiated doesn't matter
11 and A1(out1,a,sel);
12 and A2(out2,b,selb);
13 not INV1(selb,sel);
14 or O1(out,out1,out2);
15
16 endmodule
```

Input and output declarations done outside the portlist

```
1 module mux2x1_str(output out,input a,b,sel);
2
3 wire selb,out1,out2;
4
5 and A1(out1,a,sel);
6 and A2(out2,b,selb);
7 not INV1(selb,sel);
8 or O1(out,out1,out2);
9
10 endmodule
```

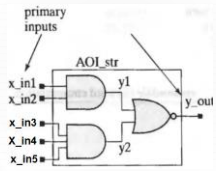
Input and output declarations done inside the portlist

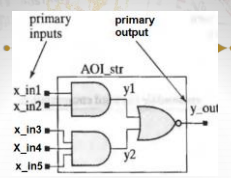
Module instantiations can be done in any order as the statements here will be executed concurrently

Testbench

```
1 module tb_mux2x1_str;
2 //Declare all the inputs as data type reg
3 //Declare all the outputs as data type wire
4 reg a, b, sel;
5 wire out;
6
7 //Instantiate the Design Under Test (DUT)
8 //Signals in the portlist need to be in the same order as in design module
9 mux2x1_str DUT(out,a,b,sel);
10
11 //All the input signal combinations are generated
12 initial
13 begin
14              a=0;b=0;sel=0;
15          #10 a=0;b=0;sel=1;
16          #10 a=0;b=1;sel=0;
17          #10 a=0;b=1;sel=1;
18          #10 a=1;b=0;sel=0;
19          #10 a=1;b=0;sel=1;
20          #10 a=1;b=1;sel=0;
21          #10 a=1;b=1;sel=1;
22 end
23
24 endmodule
```

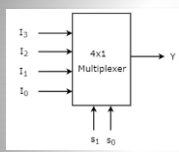

AND-OR-INVERT module





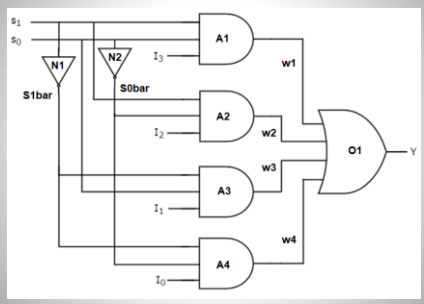

```
1 module aoi_str(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
2 output y_out;
3 input x_in1,x_in2,x_in3,x_in4,x_in5;
4
5 wire y1,y2;
6
7 nor N1(y_out,y1,y2);
8 and A1(y1,x_in1,x_in2);
9 and A2(y2,x_in3,x_in4,x_in5);
10
11 endmodule
```


Multiplexer 4x1



Selection Lines		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Multiplexer 4x1



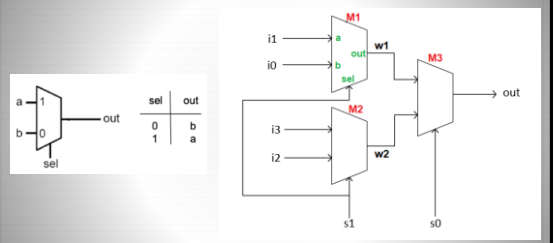
Multiplexer 4x1 - Design

```
1 module mux4x1_str(output y, input s0,s1,i0,i1,i2,i3);
2
3 wire s0bar,s1bar,w1,w2,w3,w4;
4
5 and A1(w1,s1,s0,i3);
6 and A2(w2,s1,s0bar,i2);
7 and A3(w3,s1bar,s0,i1);
8 and A4(w4,s1bar,s0bar,i0);
9
10 not N1(s1bar,s1);
11 not N2(s0bar,s0);
12
13 or O1(y,w1,w2,w3,w4);
14
15 endmodule
```


Multiplexer 4x1 - Testbench

```
1 module tb_mux4x1_str;
2
3 wire ty;
4 reg sel0, sel1, x0,x1,x2,x3;
5
6 mux4x1_str DUT(ty, sel0, sel1, x0,x1,x2,x3);
7
8 initial
9 begin
10   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=0;
11   #10;
12   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=1;
13   #10;
14   sel0=0; sel1=0; x0=0; x1=0; x2=1; x3=0;
15   ......
16   ......
17   ......
18   ......
19   ......
20 end
21
22 endmodule
```


Mux 4x1 using Mux 2x1



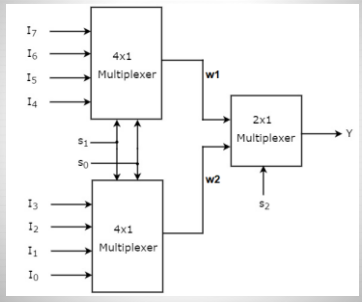
Design

```
1 module mux4x1_2x1_str(output out, input s0,s1,i0,i1,i2,i3);
2
3 wire w1,w2;
4
5 //port connection using ordered list
6 mux2x1_str M1(w1,i1,i0,s1);
7
8 //Port connection using signal names where order does not matter
9 mux2x1_str M2(.out(w2), .a(i3), .b(i2), .sel(s1));
10 mux2x1_str M3(.sel(s0), .a(w2), .out(out), .b(w1));
11
12 endmodule
```


Testbench

```
1 module tb_mux4x1_2x1_str;
2
3 wire ty;
4 reg sel0, sel1, x0,x1,x2,x3;
5
6 mux4x1_2x1_str DUT(ty, sel0, sel1, x0,x1,x2,x3);
7
8 initial
9 begin
10   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=0;
11   #10;
12   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=1;
13   #10;
14   sel0=0; sel1=0; x0=0; x1=0; x2=1; x3=0;
15   ......
16   ......
17   ......
18   ......
19   ......
20 end
21
22 endmodule
```

Mux 8x1 (Homework)

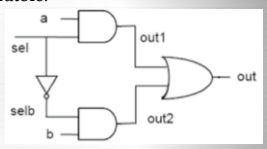


Dataflow Modeling

- Combinational circuits are described by their function rather than by their gate structure.
- Uses a number of operators that act on operands to produce the desired results.
- Uses continuous assignments and the keyword **assign.**
- A continuous assignment is a statement that assigns a value to a net.
- The datatype net is used in Verilog HDL to represent a physical connection between circuit elements.

- The value assigned to the net is specified by an expression that uses operands and operators.

sel	out
0	b
1	a



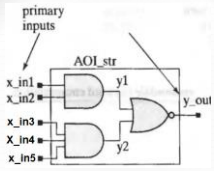
$$out = a.sel + b.\overline{sel}$$


```
1 //Structural modelling of 2x1 Mux
2 module mux2x1_str(output out,input a,b,sel);
3
4 wire selb,out1,out2;
5
6 and A1(out1,a,sel);
7 and A2(out2,b,selb);
8 not INV1(selb,sel);
9 or O1(out,out1,out2);
10
11 endmodule
```

```
1 //Dataflow modelling of 2x1 Mux
2 module mux2x1_df(output out,input a,b,sel);
3
4 assign out = (a & sel) | (b & ~sel);
5 //This can be modelled using ternary operator also
6 //assign out = sel ? a : b;
7
8 endmodule
```

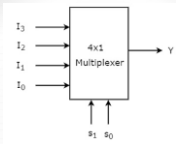
- Left Hand side of the assign statement should be of type wire
- By default output signals are of type wire

AOI – Dataflow modelling




```
assign y_out = ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));
```


4x1 Multiplexer (Dataflow)




```
assign y = (s1 & s0 & i3) | (s1 & ~s0 & i2) | (~s1 & s0 & i1) | (~s1 & ~s0 & i0);
```

OR

Using Conditional operator

```
assign y = so ? (s1 ? i1:i0) : (s1?i3:i2);
```

Behavioral Modelling

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- Behavioral description use the keyword **always** followed by a list of procedural assignment statements.
- The target output of procedural assignment statement must be of the **reg** data type.


```
1 //Behavioral modelling of 2x1 Mux
2 module mux2x1_beh(output reg out,input a,b,sel);
3
4 always @ (a or b or sel)
5 begin
6         if(sel)
7                 out = a;
8         else
9                 out = b;
10 end
11
12
13
14 endmodule
```

Homework

Behavioral Modelling of Mux 4x1 and Mux 8x1

Basic Language Concepts

Lexical Conventions

- Keywords – Special case of identifiers reserved to define the language constructs.
  - In lower case
  - Case sensitive
- String – a sequence of characters that are enclosed by double quotes
  - It cannot be on multiple lines.
  - Strings are treated as a sequence of one-byte ASCII values.
  - Eg: "Hello Verilog World" // is a string
  - Eg: "a / b" // is a string

- Identifier – names given to objects so that they can referenced in the design.
  - A letter or _ can be followed by letters, digits, $ and _
  - Max 1024 characters
  - They cannot start with a digit or a $ sign

- Numbers: can be sized or unsized
  - [<sign>] [<size>] <base> <num>
    - <size> is written only in decimal and specifies the number of bits in the number
    - If <size> is not specified, have a default number of bits that is simulator- and machine-specific
    - base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).
    - If <base format> is not specified, they are decimal numbers by default.
  - **Negative numbers** can be specified by putting a minus sign before the size for a constant number.

- **X or Z values** : An unknown value is denoted by an x. A high impedance value is denoted by z or ?.
- If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z
- If the most significant digit is 1, then it is also zero extended.

- 4'b1111 // This is a 4-bit binary number
- 12'habc // This is a 12-bit hexadecimal number
- 16'd255 // This is a 16-bit decimal number.
- 23456 // This is a 32-bit decimal number by default
- 'hc3 // This is a 32-bit hexadecimal number
- 'o21 // This is a 32-bit octal number
- 12'h13x // This is a 12-bit hex number; 4 least significant bits unknown
- 6'hx // This is a 6-bit hex number
- 32'bz // This is a 32-bit high impedance number
- -6'd3 // 8-bit negative number stored as 2's complement of 3
- 4'd-2 // Illegal specification
- 12'b1111_0000_1010 // Use of **underscore** characters for readability
- 4'b10?? // Equivalent of a 4'b10zz

- **Comments**
  - Verilog supports 2 type of comment syntaxes
  - Single line comment start with //, and end with newline.
  - Block comment, start with /*, and end with */. Block comment cannot be nested.

Data Types

Value Set

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Wire

- **Nets:**
  - Nets represent the connections between hardware elements.
  - Nets are one-bit values by default unless they are declared explicitly as vectors.
  - They are always driven by some source.
  - Default value for any net type variable is 'z' (trireg net, which defaults to x)
  - Usually, declared by the keyword ***wire***.

Registers

- Registers represent data storage elements.
- These correspond to variables in the C language.
- A **reg** type variable is the one that can hold a value.
- DO NOT confuse with hardware registers built with flip-flops.
- Register data types always retain their value until another value is placed on them.
- Unlike nets, registers do not need any drivers.

Registers (Cont..)

Example

```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
    reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
    #100 reset = 1'b0; // after 100 time units reset is deasserted.
End
```

- Registers can also be declared as signed variables

Example

```
reg signed [63:0] m; // 64 bit signed value
integer i; // 32 bit signed value
```

Rules for reg and wire

- The common rule in Verilog:
  - **"A variable on the Left Hand Side (LHS) of a procedural block assignment is always declared as a register data type."**
  - **"All other variables are of net type."**
- So, reg is assigned within ***always*** or ***initial*** blocks.
- A variable is declared of type wire if it appears on the left side of an continuous assignment statement.
- Continuous assignment statements start with the keyword assign.

Vectors

- Vectors have multiple bits and are often used to represent buses.
- The left most number is an MSB (Most Significant Bit).
- There are 2 representations for vectors:
  - A little-endian notation: [high# : low#]
  - A big-endian notation: [low# : high#]
  - The left number in the squared brackets is always the most significant bit of the vector

```
wire [3:0] busA;  // little-endian notation
wire [0:15] busC; // big-endian notation
reg [1:4] busB;
```


Vectors (Cont…)

- **Vector Part Select**
  - Example 1

```
reg [15:0] data;
reg [0:7] carry;
reg [0:2] inter_carry;
initial begin
    data [15:8] = 8'h12;
    inter_carry = carry [1:3];
end
```

  - Example 2

```
reg [63:0] out;
reg [3:0] dest_addr;
initial begin
   dest_addr = out [63:60];
end
```

  - Example 2

```
reg [2:0] out1;
reg [0:2] out2;
initial begin
  out1 = out2;
end
```


Vector Part Select

```
data[15:8] = 8'h_12;          // Accessing only bits 16 to 9 of data
inter_carry = carry[1:3];
```

Slice Management

```
reg [63:0] out;
reg [3:0] dest_addr;
initial begin
      dest_addr = out[63:60];
end
```

=

```
dest_addr[0] = out[60];
dest_addr[1] = out[61];
dest_addr[2] = out[62];
dest_addr[3] = out[63];
```


- Vector assignment (**by position!!**)

```
…
reg [2:0] bus_A;
reg [0:2] bus_B;
initial begin
      bus_A = bus_B;
end
…
```

=

```
bus_A[2] = bus_B[0];
bus_A[1] = bus_B[1];
bus_A[0] = bus_B[2];
```


Vectors (Cont…)

- Variable Vector Part Select
  - [<starting_bit>+ : <width>]
  - [<starting_bit>- : <width>]

```
reg [31:0] data1; reg [0:31] data2;
reg [7:0] byte1; reg [3:0] nibble1;
reg [0:7] byte2; reg [0:3] nibble2;
…
nibble1 = data1[31-:4]; //selects 4 bits from 31 to down, i.e. [31:28]
byte1 = data1[24-:8]; // selects data1[24:17]
byte2 = data2[10+:8]; // selects data2[10:17]
nibble2 = data2[28+:4]; // selects data2[28:31]
```

Integers

- A general purpose register data type with default value having all x bits.
- Declared with keyword integer.
- Usually preferred for arithmetic manipulations over reg.
- Default width: host machine word size (minimum 32 bits).
- Differs from reg type: integers stores signed quantities as opposed to ***reg storing unsigned quantities***.

Real Numbers

- Real number constants and real register data types are declared with the keyword **real**.
- Real numbers cannot have a range declaration.
- Their default value is 0.
- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 106).
- When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
real delta; // Define a real variable called delta
initial
begin
   delta = 4e10; // delta is assigned in scientific notation
   delta = 2.13; // delta is assigned a value 2.13
end
integer i; // Define an integer i
initial
   i = delta; // i gets the value 2 (rounded value of 2.13)
```


Datatype: Time

- A special time register data type is used in Verilog to store simulation time.
- A time variable is declared with the keyword **time**.
- The width for time register data types is implementation-specific but is at least 64 bits.
- The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time
initial
   save_sim_time = $time; // Save the current simulation time
```


Arrays

- Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types.
- Verilog supports multi-dimensional arrays.
- Each element of the array can be used in the same fashion as a scalar or vector net.
- Declaration:<type> <vector_size> <ary_name> <ary_size>;
  - <ary_size> is declared as a range.
  - Elements are accessed by: <ary_name> [<index>].

Arrays (Cont..)

Example 1

```
reg [7:0] mem[1023:0];      //The 'mem' variable is a memory that contains 1024                                    8-bit                                    words.
integer i_mem[8:1];                    //The 'i_mem' variable has 8 words (each word is an integer register).
reg [4:0] port_id[0:7];       // Array of 8 port_ids; each port_id is 5 bits wide
integer matrix[4:0][0:255]; // Two dimensional array of integers
reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array
wire [7:0] w_array2 [5:0];   // Declare an array of 8 bit vector wire
wire w_array1[7:0][5:0];    // Declare an array of single bit wires
```

Example 2

```
reg [3:0] mem[255:0], r;       //This line a declares 4-bit register 'r' and memory 'mem', which contains 256 4-bit words.
```


Example 3

```
reg [7:0] mem[255:0], r;
r = mem[135];
r[3:1] = 3'b100;
mem[135] = r;
```

- shows how to access particular bits of memory.

Operators

Operators

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	Multiply	2
	/	Divide	2
	+	Add	2
	-	Subtract	2
	%	Modulus	2
	**	Power (Exponent)	2
Logical	!	Logical Negation	1
	&&	Logical AND	2
	||	Logical OR	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal	2
	<=	Less than or equal	2

Operators (Cont…)

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Equality	==	Equality	2
	!=	Inequality	2
	===	Case equality	2
	!==	Case inequality	2
Bitwise	~	Bitwise Negation	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	^~ or ~^	Bitwise XNOR	2

Operators (Cont…)

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	^~ or ~^	Reduction XNOR	1
Shift	>>	Right Shift	2
	<<	Left Shift	2
	>>>	Arithmetic Right Shift	2
	<<<	Arithmetic Left Shift	2
Concatenation	{}	Concatenation	Any Number
Replication	{{}}	Replication	Any Number
Conditional	?:	Conditional	3

Arithmetic Operators

- **Binary** and **unary** operators
- Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%).
- Binary operators take two operands.

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers
A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
F = E ** F; //E to the power F, yields 16
```


Arithmetic Operators (Cont…)

- If any operand bit has a value x, then the result of the entire expression is x. (This is because if an operand value is not known precisely, the result should be an unknown.)

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

- Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

```
13 % 3 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign of the first operand
7 % -2 // Evaluates to +1, takes sign of the first operand
```

Arithmetic Operators (Cont…)

- The operators + and - can also work as unary operators.
- They are used to specify the positive or negative sign of the operand.
- Unary + or - operators have higher precedence than the binary + or - operators.

```
-4 // Negative 4
+5 // Positive 5
```


Logical Operators

- Logical operators are logical-and (&&), logical-or (||) and logical-not (!). Operators && and || are binary operators.
- **Conditions for Logical Operators**
  1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
  2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
  3. Logical operators take variables or expressions as operands.

Logical Operators (Cont…)

```
//General Case
A = 3; B = 0;
A && B      // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B      // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A          // Evaluates to 0. Equivalent to not (logical-1)
!B          // Evaluates to 1. Equivalent to not (logical-0)
// Unknowns
A = 2'b0x; B = 2'b10;
A && B      // Evaluates to x. Equivalent to (x && logical 1)
// Expressions
(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.
                     // Evaluates to 0 if either is false.
```


Relational operators

- Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=).
- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value x.

```
A = 4, B = 3
X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx
    A <= B          // Evaluates to a logical 0
    A > B           // Evaluates to a logical 1
    Y >= X          // Evaluates to a logical 1
    Y < Z           // Evaluates to an x
```


Equality Operators

- Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==).
- When used in an expression, equality operators return logical value 1 if true, 0 if false.
- These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.
- The logical equality operators (==, !=) will yield an x, if either operand has x or z in its bits.
- The case equality operators (===, !==) compare both operands bit by bit, including x and z and results is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly.
- Case equality operators never result in an x.


```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx
A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
```

Bitwise operators

- Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^).
- Bitwise operators perform a bit-by-bit operation on two operands.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. A z is treated as an x in a bitwise operation.
- The unary negation operator (~), which takes only one operand and operates on the bits of the single operand.


```
// X = 4'b1010, Y = 4'b1101,  Z = 4'b10x1

~X      // Negation. Result is 4'b0101
X & Y   // Bitwise and. Result is 4'b1000
X | Y   // Bitwise or. Result is 4'b1111
X ^ Y   // Bitwise xor. Result is 4'b0111
X ^~ Y // Bitwise xnor. Result is 4'b1000
X & Z   // Result is 4'b10x0
```


Reduction Operators

- Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).
- Reduction operators take only one operand.
- Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
- Reduction operators work bit by bit from right to left.
  - Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively


```
// X = 4'b1010
&X      //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X      //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X      //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
        //A reduction xor or xnor can be used for even or odd
        //parity generation of a vector.
```


Shift Operators

- Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<).
- The operands are the vector and the number of bits to shift.
- When the bits are shifted, the vacant bit positions are filled with zeros.
- Shift operations do not wrap around.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.
  - **Arithmetic Shift Right (>>>)**: Shift right specified number of bits, fill the value of sign bit if the expression is signed, otherwise fill with 0.
  - **Arithmetic Shift Left (>>>)**: Shift left specified number of bits, filling with 0


```
// X = 4'b1100
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB
position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB
position.
Y = X << 2;      //Y is 4'b0000. Shift left 2 bits.

integer a, b, c;     //Signed data types
a = 0;
b = -10;             // 111...10110 binary
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple operands.
- The operands must be sized.
- Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.

```
A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C}                    // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001}   // Result Y is 11'b10010110001
Y = {A , B[0], C[1]}           // Result Y is 3'b101
```


Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 10'b1111000010
```

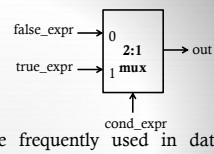

Conditional Operator

- The conditional operator(? :) takes three operands.

  Usage: condition_expr ? true_expr : false_expr ;
- The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated.
- If the result is x (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.
- Similar to if – else expression.

Conditional Operator (Cont..)

- Similar to a multiplexer



- Conditional operators are frequently used in dataflow modeling to model conditional assignments.

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

- Conditional operations can be nested.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```


Operator Precedence

Operator Type	Operator Symbol	Precedence
Unary	+, -, !, ~	Highest
Multiply, Divide, Modulus	*, /, %	
Add, subtract	+, -	
Shift	<<, >>	
Relational	<, <=, >, >=	
Equality	==, !=, ===, !==	
Reduction	&, ~& ^, ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest

Compiler Directives

- `timescale
- `define

Timescale

- Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 s, and delay values in another module need to be defined by using a different time unit, e.g. 100 ns.
- Verilog HDL allows the reference time unit for modules to be specified with the `timescale compiler directive.

  Usage: `timescale <reference_time_unit> / <time_precision>
- The <reference_time_unit> specifies the unit of measurement for times and delays. The <time_precision> specifies the precision to which the delays are rounded off during simulation.
- Only 1, 10, and 100 are valid integers for specifying time unit and time precision.
  - Eg: `timescale 1 us / 10 ns //Reference time unit is 1 microseconds //and precision is 10 ns

- The `timescale compiler directive must be written outside the boundary of the modules.
- <time precision> must be at least as small as the <time unit>
- Each Timescale directive has two numbers:

  `timescale 100 ps / 10 ps

  #10 CLK <= 1'b0;
  - Wait for 10 time units, then assign a value of zero to signal CLK.
  - 10 time units means 1 ns here.
- Verilog compiler over-writes timescale every time a new one is read

`define

- The `define directive is used for text substitution:

  `define <macro_name> <macro_text>
- It will just substitute whatever follows the macro name.

  `define whatever 1234;
  - `whatever will be 1234;
  - Semicolon will be included.
  - 'define WORD_REG reg [31:0] //you can then define a 32-bit register as 'WORD_REG reg32;
- This is similar to the #define construct in C.

`include

- To include entire contents of a Verilog source file in another Verilog file during compilation.
- This works similarly to the #include in the C
- This directive is typically used to include header files
  - eg: 'include header.v

`ifdef

System Tasks for Simulation

- $display
- $write
- $monitor
- $strobe
- $time
- $stop
- $finish

$display

- Displays the value of signals or variables in a design or test fixture.

  Usage: $display(s1, s2, s3,….sn);

  s1, s2, etc. are signals in the uut or variables in a testbench.
- Includes new line character by default.
- **$display string formatting**
  - %d display a variable in decimal
  - %b display a variable in binary
  - %s display a string
  - %c display an ASCII character
  - %h display a variable in hex
  - %g display a real number in scientific notation or decimal, whichever is shorter.

$time

- $time calls out the simulation time.
- It has nothing to do with the time of day.
- Internally, it is represented by a 64-bit number, so it can keep track of a long simulation.

  Example: $display($time);

  - 80
  - 80 time units have passed since the simulation started.

$write

- $write is exactly the same as $display except that it does not implicitly include a new line character.

$monitor

- Monitors a signal when its value changes.
  - Usage: $monitor(p1,p2,p3…pn);

    p1, p2, etc. are signals in the uut or variables in a test bench.
- $monitor displays the values of all objects in its list whenever any one of them changes.
- Same formatting as $display
- Only ONE $monitor can be active at a time.

  $monitor("clock = %b reset = %b", clk, rst);
  $monitor("state = %h", state);
  • will result in only "state" being monitored.
- Difference Between $monitor and $display
  - $monitor is continuous.
  - $display only runs once.

$strobe

- Very similar to $display.
- The difference is that $strobe is always the last task to be executed at any time specification whereas $display may not be.

Scope of variables

- All system task examples so far have not had any scope operator.
- They would all operate on variables in the test bench.
- Internal signals/variables at any level of hierarchy can also be displayed, monitored, etc. by scoping down to where they are.
  - By using hierarchical names.
    - ckt1 UUT(CLK, X, Y, Z);
    - $display ("a = %b, X = %b", UUT.a, X);
      - The value of the variable "a" that is in the uut will be shown, as will the variable X in the test bench.
- To display the level of hierarchy, use the special character %m in the $display task.

$stop and $finish

- $stop halts simulation and puts the simulator into interactive mode.
  - Mostly used for debugging in lab.
- Resume simulation by typing "." at the prompt.
- $finish ends the simulation.

Modules and Ports

- Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.
- Note that all port declarations are implicitly declared as wire in Verilog.
- Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout.
- However, if output ports hold their value, they must be declared as reg.

D Flip-flop

```
// module D_FF with asynchronous reset
module D_FF(q, d, clk, reset);
output q;
input d, clk, reset;
reg q;

always @(posedge reset or negedge clk)
  if (reset)
    q <= 1'b0;
  else
    q <= d;
endmodule
```


- Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

- **Inputs**
  - Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.
- **Outputs**
  - Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.
- **Inouts**
  - Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

- **Unconnected ports**
- Verilog allows ports to remain unconnected.
  - `fulladd4 fa0(SUM, , A, B, C_IN); // Output port c_out is unconnected`

Connecting Ports to External Signals

Connecting by ordered list

- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.

- **Connecting ports by name**
- For large designs where modules have, many ports, remembering the order of the ports in the module definition is impractical and error-prone.
- Verilog provides the capability to connect external signals to ports by the port names, rather than by position.
- The port connections can be specified in any order as long as the port name in the module definition correctly matches the external signal.

  Eg: fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);
- Unconnected ports can be dropped.

  fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);

Modeling in Verilog

Gate Level Modeling

- Verilog has built in primitives like gates, transmission gates, and switches.
- The gates have one scalar output and multiple scalar inputs.
- The first terminal in the list of gate terminals is an output and the other terminals are inputs.
- The output of a gate is evaluated as soon as one of the inputs changes.

Verilog Primitives

- Just like drawing schematics, circuits of any arbitrary size can be made with primitive components.
- Primitive logic operators are
  - AND
  - OR
  - XOR
  - NOT
  - NAND, NOR, XNOR, BUF
    - Any XOR/XNOR will output an X (unknown) if any input is X or Z.
- Primitives are instantiated in a module
  - Eg: nand (out, in1, in2, in3, in4);
    - nand: primitive type. Verilog keyword

and	i1			
	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

(i2)

or	i1			
	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

(i2)

xor	i1			
	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

(i2)

nand	i1			
	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

(i2)

nor	i1			
	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

(i2)

xnor	i1			
	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

(i2)

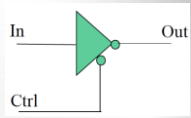
buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

- Primitive Pins Are Expandable
  - The number of pins for a primitive gate is defined by the number of nets connected to it.
- All gates except not and buf can have a variable number of inputs, but only one output.
- The not and buf gates can have a variable number of outputs, but only one input.

Primitives with Control Inputs

- Bufif0
- Bufif1
- Notif1
- Notif0



Output is Z (high impedance) if control is not asserted.
Output is X if control IS asserted and input is X or Z.

```
//Instantiation of bufif gates.
        bufif1 b1 (out, in, ctrl);
        bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
        notif1 n1 (out, in, ctrl);
        notif0 n0 (out, in, ctrl);
```


bufif1		ctrl			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

notif1		i1			
		0	1	x	z
i2	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

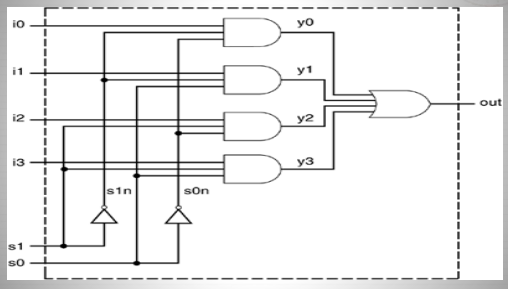
bufif0		i1			
		0	1	x	z
i2	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

notif0		i1			
		0	1	x	z
i2	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

Array of Instances

```
wire [3:0] OUT, IN1, IN2;
// basic gate instantiations.
nand n_gate[3:0](OUT, IN1, IN2);
// This is equivalent to the following 4 instantiations
        nand n_gate0(OUT[0], IN1[0], IN2[0]);
        nand n_gate1(OUT[1], IN1[1], IN2[1]);
        nand n_gate2(OUT[2], IN1[2], IN2[2]);
        nand n_gate3(OUT[3], IN1[3], IN2[3]);
```


- Write a gate level model for a 4: 1 multiplexer



```
/* Module 4-to-1 multiplexer. Port list is taken exactly from the I/O diagram.*/
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;
// Gate instantiations
// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);
// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
// 4-input or gate instantiated
or (out, y0, y1, y2, y3);
endmodule
```


- Write the structural code for a 1 bit full adder
- Write the structural code for a 4 bit ripple carry adder using above adders

Gate Delays

- In real circuits, logic gates have delays associated with them.
- Gate delays allow the Verilog user to specify delays through the logic circuits.
- Pin-to-pin delays can also be specified in Verilog.
  - Rise Delay
  - Fall Delay
  - Turn – off Delay

Gate Delays (Cont…)

- **Rise delay:** Associated with a gate output transition to a 1 from another value.

- **Fall Delay:** Associated with a gate output transition to a 0 from another value.

- **Turn-off delay:** Associated with a gate output transition to the high impedance value (z) from another value.
- If the value changes to x, the minimum of the three delays is considered.

Gate Delays (Cont…)

Types of Delay Specification

```
// Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);

// Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions
and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6
bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5
```


Gate Delays (Cont…)

Min /Typ /Max value

- The min/typ/max value is the minimum/typical/max delay value that the designer expects the gate to have.

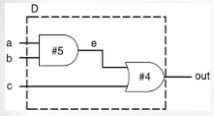
```
// if only One delay is specified
and #(4:5:6) a1(out, i1, i2);
        // mindelay= 4
        // typdelay= 5
        // maxdelay= 6
```

Gate Delays (Cont…)

```
// Two delays are specified
and #(3:4:5, 5:6:7) a2(out, i1, i2);
// mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// maxdelays, rise= 5, fall= 7, turn-off = min(5,7)

// Three delays
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
// mindelays, rise= 2 fall= 3 turn-off = 4
// typdelays, rise= 3 fall= 4 turn-off = 5
// maxdelays, rise= 4 fall= 5 turn-off = 6
```


Gate Delays (Cont…)

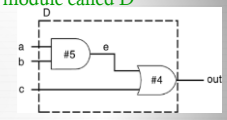
Verilog Definition for Module D with Delay



Gate Delays (Cont…)

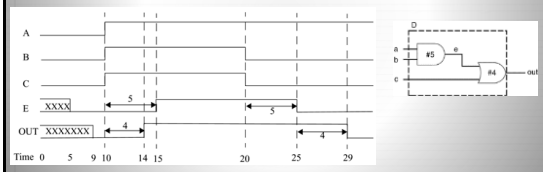
Verilog Definition for Module D with Delay

```
// Define a simple combination module called D

module D (out, a, b, c);
output out;
input a,b,c;
wire e;
and #(5) a1(e, a, b); //Delay of 5 on gate a1
or #(4) o1(out, e,c); //Delay of 4 on gate o1
endmodule
```



Gate Delays (Cont…)

1. The outputs E and OUT are initially unknown.
2. At time 10, after A, B, and C all transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.
3. At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.



Dataflow (RTL) Modeling

- For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually.
- In complex designs the number of gates is very large.
- Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

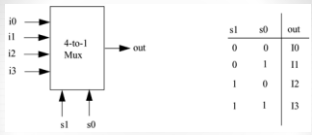
Continuous Assignments

- A continuous assignment is the most basic statement in dataflow modeling, used to ***drive a value onto a net***.
- The assignment statement starts with the keyword ***assign***.

  Syntax: assign [drive_strength][delay] list_of_net_assignments;
  - The default value for drive strength is strong1 and strong0
- Continuous assignments are always active.
- The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
- The operands on the right-hand side can be registers or nets or function calls.
- Delay values are used to control the time when a net is assigned the evaluated value.

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {co, s[3:0]} = a[3:0] + b[3:0] + cn;
```


Dataflow model multiplexer



Dataflow model multiplexer

// 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow_ logic equation
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
//Logic equation for out
assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) |
                        (s1 & ~s0 & i2) | (s1 & s0 & i3) ;
endmodule
```


Dataflow model multiplexer

- **4-to-1 Multiplexer, Using Conditional Operators**

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```


Implicit Continuous Assignment

- Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.
- There can be only one implicit declaration assignment per net because a net is declared only once.

```
//Regular continuous assignment
wire out;
assign out = in1 & in2;
//Same effect is achieved by an implicit continuous assignment
wire out = in1 & in2;
```


Implicit Net Declaration

- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name.
- If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
// Continuous assign. out is a net.
wire i1, i2;
assign out = i1 & i2; //Note that out was not declared as a wire
                      //but an implicit wire declaration for out
                      //is done by the simulator
```

Delays

- A delay control expression specifies the time duration between initially encountering the statement and when the statement actually executes.
  e.g. #10 A = A + 1;
- There are different ways to specify delays in continuous assignments.

- **Regular Assignment Delay:** This is the most commonly used method.
  e.g.- assign #10 q = x + y;
- **Implicit Continuous Assignment Delay:**
  e.g. – wire #10 out = a ^ b;
  // which is equivalent to the following:
  wire out;
  assign #10 out = a ^ b;
- **Net Declaration Delay:** The delay can be put on the net in declaration itself.
  e.g. – wire #10 out;
  assign out = a & b;
  // which is equivalent to the following:
  wire out;
  assign #10 out = a & b;

Behavioral Modeling

- Behavioral modeling represents the circuit at a very high level of abstraction.
- Verilog provides designers the ability to describe design functionality in an algorithmic manner.
- The designer can describe the behavior of the circuit.

Structured Procedures

- There are two structured procedure statements in Verilog: *always* and *initial*.
- These statements are the two most basic statements in behavioral modeling.
- All other behavioral statements can appear only inside these structured procedure statements.
- Group of statements coming under always and initial blocks are called ***procedural blocks***.
- Assignment inside procedural blocks are called ***procedural assignment***.

Procedural Blocks

- Procedural blocks are the basic components for behavioral modeling.

```
initial
begin
… procedural statements …
end
```

- Runs when simulation starts
- Terminates when control reaches the end
- Good for providing stimulus

```
always
begin
… procedural statements …
end
```

- Runs when simulation starts
- Restarts when control reaches the end
- Good for modeling / specifying hardware

Procedural Blocks (Cont..)

- Procedural blocks are like concurrent processes.
- All blocks execute in parallel.
- Statements in a block are executed sequentially, but all within one unit of simulated time. (unless delay is specified)
- **initial block**
  - Executes only once.
- **always block**
  - Executes repeatedly.
  - It must have timing control, otherwise it become **INFINITE LOOP**

initial Statement

- All statements inside an *initial* statement constitute an initial block.
- An initial block starts at time 0, executes exactly once during a simulation, and then ***does not execute*** again.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0.
- The initial blocks are typically used for *initialization.*

initial Statement - Example

```
module stimulus;
reg x,y, a,b, m;
initial
    m = 1'b0;
initial
begin
    #5 a = 1'b1;
    #25 b = 1'b0;
end
initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end
Initial
    #50 $finish;
endmodule
```


initial Statement - Example

```
module stimulus;
reg x,y, a,b, m;
initial
    m = 1'b0;
initial
begin
    #5 a = 1'b1;
    #25 b = 1'b0;
end
initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end
Initial
    #50 $finish;
endmodule
```

Time	Statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish

Combined Variable Declaration and Initialization

Variables can be initialized when they are declared.

```
//The clock variable is defined first
reg clock;
//The value of clock is set to 0
initial clock = 0;

//Instead of the above method, clock variable can be initialized
//at the time of declaration
//This is allowed only for variables declared at module level.
reg clock = 0;
```


Combined Port/Data Declaration and Initialization

- The combined port/data declaration can also be combined with an initialization

```
module adder (sum, co, a, b, ci);
output reg [7:0] sum = 0; //Initialize 8 bit output sum
output reg co = 0; //Initialize 1 bit output co
input [7:0] a, b;
input ci;
--
--
endmodule
```



```
module adder (output reg [7:0] sum = 0, //Initialize 8 bit output
              output reg co = 0, //Initialize 1 bit output co
              input [7:0] a, b,
              input ci);
--
--
endmodule
```

always Statement

- All behavioral statements inside an always statement constitute an always block.
- The always statement starts at time 0 and *executes* the statements in the always block *continuously* in a looping fashion.
- This statement is used to model a block of activity that is repeated continuously in a digital circuit.

always Statement - Example

```
module clock_gen (clock);
output reg clock;
//Initialize clock at time zero
initial
    clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
    #10 clock = ~clock;
initial
    #1000 $finish;
endmodule
```


Procedural Assignments

- Procedural assignments update values of reg, integer, real, or time variables.
- The syntax for the simplest form of procedural assignment is shown below.
- assignment ::= variable_lvalue = [delay_or_event_control] expression
- There are two types of assignment statements are there in Verilog:
  - Blocking statements
  - Non-blocking statements.

Blocking Assignments

- It is a way of "blocking" the further statements until the current statement execution is completed.
- The blocking assignment operator is an equal sign (=).
- Evaluated and assigned in a single step.
- A blocking assignment must evaluate the RHS arguments and update the LHS expression of the blocking assignment without interruption from any other Verilog statement.
- The blocking assignment with timing delays on the RHS of the blocking operator, is considered to be a poor coding style.

Blocking Assignments (Cont..)

- A problem with blocking assignments occurs when
  - The RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block, and
  - Both equations are scheduled to execute in the same simulation time step, such as on the same clock edge.
- If blocking assignments are not properly ordered, a race condition can occur.
- When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

Blocking Assignments - Example

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
   x = 0; y = 1; z = 1;   //Scalar assignments
   count = 0;             //Assignment to integer variables
   reg_a = 16'b0; reg_b = reg_a;        //initialize vectors
   #15 reg_a[2] = 1'b1;          //Bit select assignment with delay
  #10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to
                               // part select of a vector
   count = count+ 1;   //Assignment to an integer (increment)
end
```

Blocking Assignments - Example

- All statements x = 0 through reg_b = reg_a are executed at time 0
- Statement reg_a[2] = 0 at time = 15
- Statement reg_b[15:13] = {x, y, z} at time = 25
- Statement count = count + 1 at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

Non-blocking Assignments

- Non-blocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
- A <= operator is used to specify nonblocking assignments.
- Evaluated and assigned in two steps:
  - Evaluate the RHS of non-blocking statements at the beginning of the time step.
  - The assignment to the left-hand side is postponed until other evaluations in the current time step are completed.
- Also, the RHS expression of other Verilog non-blocking assignments can also be evaluated and LHS updates scheduled. The non-blocking assignment does not block other.

Non-blocking Assignments – Eg.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
   x = 0; y = 1; z = 1;  //Scalar assignments
   count = 0;            //Assignment to integer variables
   reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
   reg_a[2] <= #15 1'b1; //Bit select assignment with delay
   reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
                                  //to part select of a vector
   count <= count + 1; //Assignment to an integer (increment)
end
```


Non-blocking Assignments – Eg.

- The statements x = 0 through reg_b = reg_a are executed sequentially at time 0.
- Then the three nonblocking assignments are processed at the same simulation time.
  1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)
  2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)
  3. count = count + 1 is scheduled to be executed without any delay (i.e., time = 0)

- The simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution.
- Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.
- However, it is recommended that blocking and nonblocking assignments not be mixed in the same always block.

Each Verilog simulation time step is divided into different queues

Time 0:

- Q1 — (in any order) :
  - Evaluate RHS of all non-blocking assignments
  - Evaluate RHS and change LHS of all blocking assignments
  - Evaluate RHS and change LHS of all continuous assignments
  - Evaluate inputs and change outputs of all primitives
  - Evaluate and print output from $display and $write
- Q2 — (in any order) :
  - Change LHS of all non-blocking assignments
- Q3 — (in any order) :
  - Evaluate and print output from $monitor and $strobe

- Concurrent blocking assignments have unpredictable results

```
always @(posedge clk)
#5 A = A + 1;
always @(posedge clk)
#5 B = A + 1;
```

Unpredictable Result:
(new value of B could be evaluated before or after A changes)

- Concurrent non-blocking assignments have predictable results

```
always @(posedge clk)
#5 A <= A + 1;
always @(posedge clk)
#5 B <= A + 1;
```

Predictable Result:
(new value of B will always be evaluated before A changes)

Application of nonblocking assignments

- They are used as a method to model several concurrent data transfers that take place after a common event.

```
always @(posedge clock)
begin
   reg1 <= #1 in1;
   reg2 <= @(negedge clock) in2 ^ in3;
   reg3 <= #1 reg1; //The old value of reg1
end
```


Nonblocking Statements to Eliminate Race Conditions

```
//Two concurrent always blocks with blocking statements
always @(posedge clock)
   a = b;
always @(posedge clock)
   b = a;

//Eg 2: Two concurrent always blocks with nonblocking statements
always @(posedge clock)
   a <= b;
always @(posedge clock)
   b <= a;
```


Timing Controls

- **Delay-Based Timing Control**
- **Level-Sensitive Timing Control**
- **Event-Based Timing Control**

Delay-Based Timing Control

- Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed.
- **Regular delay control:** used when a non-zero delay is specified to the left of a procedural assignment.
- A timing control before an assignment statement will postpone when the next assignment is evaluated
  - Evaluation is delayed for the amount of time specified


```
begin
#5 A = 1;          -> delay for 5, then evaluate and assign
#5 A = A + 1;      ->delay 5 more, then evaluate and assign
B = A + 1;         ->no delay; evaluate and assign
end
```

What values do A and B contain after 10 time units?

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;
initial begin
    x = 0; // no delay control
    #10 y = 1; // delay control with a number. y = 1 by 10 units
    #latency z = 0; // Delay control with identifier.
    #(latency + delta) p = 1; // Delay control with expression
    #y x = x + 1; // Delay control with identifier.
    #(4:5:6) q = 0; // Minimum, typical and maximum delay.
end
```


Delay-Based Timing Control (Cont..)

- **Intra-assignment delay control:** Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator.
  - The right-hand side is evaluated before the delay
  - The left-hand side is assigned after the delay

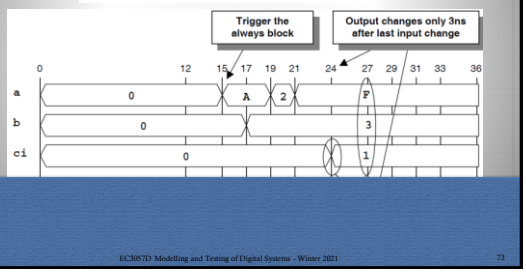
```
always @(A)
B = #5 A;          //A is evaluated at the time it changes, but
                   //is not assigned to B until after 5 time units
```

  - always @(negedge clk)
  - Q <=@(posedge clk) D; //D is evaluated at the negative edge of CLK, Q is changed on the //positive edge of CLK

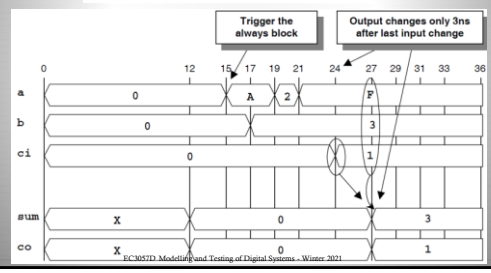

```
//intra assignment delays
reg x, y, z;
initial begin
    x = 0; z = 0;
    y = #5 x + z; //Take value of x and z at the time=0, evaluate
                  //x + z and then wait 5 time units to assign value to y.
End

//Equivalent method with temporary variables and regular delay
    control
initial begin
    x = 0; z = 0;
    temp_xz = x + z;
    #5 y = temp_xz;   //Take value of x + z at the current time and store it
        //in a temporary variable. Even though x and z might change
        //between 0 and 5, the value assigned to y at time 5 is unaffected.
end
```


- Regular delays defer the execution of the entire assignment.
- Intra-assignment delays compute the right hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
- Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.


```
module adder_t1 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    #12 {co, sum} = a + b + ci;
endmodule
```




```
module adder_t1 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    #12 {co, sum} = a + b + ci;
endmodule
```



Delay-Based Timing Control (Cont..)

Zero delay control

- Procedural statements in different always-initial blocks may be evaluated at the same simulation time.
- The order of execution of these statements in different always-initial blocks is nondeterministic.
- Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.


```
initial begin
    x = 0;
    y = 0;
end
initial begin
    #0 x = 1; //zero delay control
    #0 y = 1;
end
```

- In Example, four statements x = 0, y = 0, x = 1, y = 1 are to be executed at simulation time 0. However, since x = 1 and y = 1 have #0, they will be executed last.
- Thus, at the end of time 0, x will have value 1 and y will have value 1. The order in which x = 1 and y = 1 are executed is not deterministic.
- using #0 is not a recommended practice.

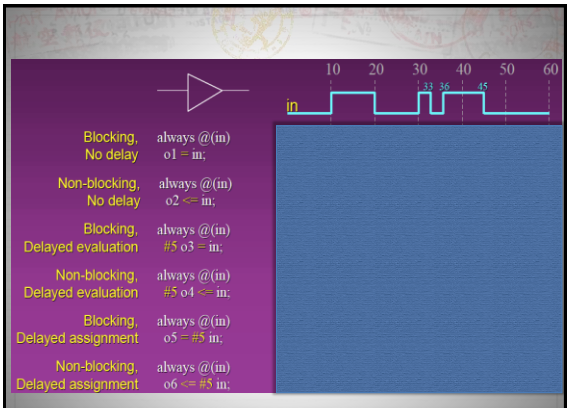
Event-Based Timing Control

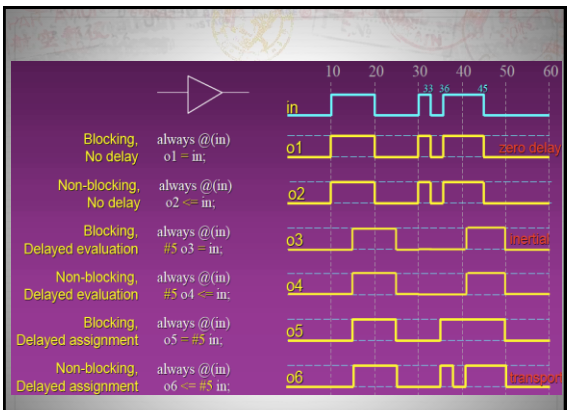
Regular event control (Event Control)

The @ symbol is used to specify an event control.

Statements can be executed on changes in signal value or at a positive or negative transition of the signal value.

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                                        //a positive transition (
    0 to 1,x or z, x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                                        //a negative transition
    ( 1 to 0,x or z,x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
                                        //to q at the positive
    edge of clock
```

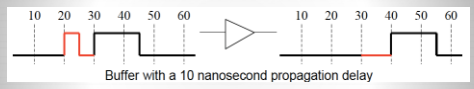





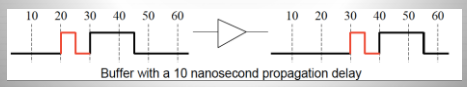
Propagation delay methods

Hardware has two primary propagation delay methods:

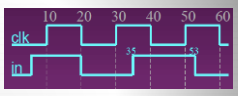
- Inertial delay models devices with finite switching speeds; input glitches do not propagate to the output



- Transport delay models devices with near infinite switching speeds; input glitches propagate to the output

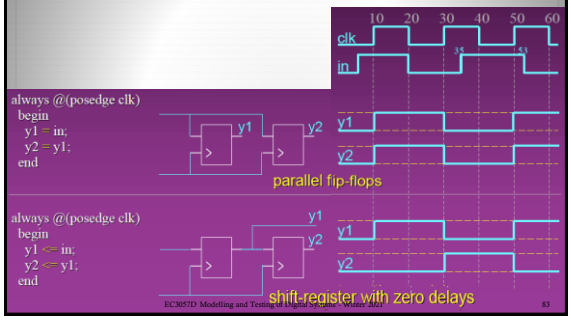


Sequential Logic
Procedural Assignments




```
always @(posedge clk)
 begin
  y1 = in;
  y2 = y1;
 end
```

```
always @(posedge clk)
 begin
  y1 <= in;
  y2 <= y1;
 end
```

Conditional Statements

- Conditional statements are used for making decisions based upon certain conditions.
- These conditions are used to decide whether or not a statement should be executed.
- Keywords ***if*** and ***else*** are used for conditional statements.

```
if (<expression>) true_statement ;
```
← Case 1

```
if (<expression>) true_statement ;
else false_statement ;
```
← Case 2

```
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```
← Case 2

Case 1

```
if(!lock) buffer = data;
if(enable) out = in;
```

Case 2

```
if (number_queued< MAX_Q_DEPTH)
begin
   data_queue = data;
   number_queued = number_queued + 1;
end
else
   $display("Queue Full. Try again");
```

Case 3

```
if (alu_control == 0) y = x + z;
else if(alu_control == 1) y = x - z;
else if(alu_control == 2) y = x * z;
else
$display("Invalid control signal");
```


Multiway Branching

case statement

- The keywords case, endcase, and default are used in the case statement.

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
    ...
    ...
    default:
    default_statement;
endcase
```

```
//Execute statements based on the ALU
control
reg [1:0] alu_control;
...
case (alu_control)
2'd0 : y = x + z;
2'd1 : y = x - z;
2'd2 : y = x * z;
default : $display("Invalid signal");
endcase
```



```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
    reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ({s1, s0}) //Switch based on concatenation of control signals
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display("Invalid control signals");
    endcase
endmodule
```

Case Statement with x and z

- Considers unknown signals on select.
- If any select signal is x then outputs are x.
- If any select signal is z, outputs are z.
- If one is x and the other is z, x gets higher priority.

```
2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
begin
    out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;
end
2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
begin
    out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;
end
```


casex, casez Keywords

- There are two variations of the case statement - casex and casez.
- casez treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also represented by ? in that position.
- casex treats all x and z values in the case item or the case expression as don't cares.

```
reg [3:0] encoding;
integer state;
casex (encoding) //logic value x represents a don't care bit.
        4'b1xxx : next_state = 3;
        4'bx1xx : next_state = 2;
        4'bxx1x : next_state = 1;
        default : next_state = 0;
endcase
```

"an input encoding = 4'b10xz would cause next_state = 3 to be executed"

While Loop

- The keyword while is used to specify this loop.
- The while loop executes until the while expression is not true.
- If the loop is entered when the while-expression is not true, the loop is not executed at all.

```
integer count;
initial
   begin
   count = 0;
     while (count < 128) //Execute loop till count is 127, exit at
                         //count 128
    begin
            $display("Count = %d", count);
            count = count + 1;
     end
   end
```


Example

- Write a code using "while loop" to find the first bit with a value 1 in flag vector variable

- //Initialize array elements
- 'define MAX_STATES 32
- integer state [0: 'MAX_STATES-1]; //Integer array state with elements
- 0:31
- integer i;
- initial
- begin
- for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
- state[i] = 0;


```
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;
initial begin
    flag = 16'b 0010_0000_0000_0000;
    i = 0;
    continue = 'TRUE;
    while((i < 16) && continue ) //Multiple conditions using operators.
    begin
        if (flag[i]) begin
          $display("Encountered a TRUE bit at element number %d", i);
          continue = 'FALSE;
        end
        i = i + 1;
   end
end
```

For Loop

- The keyword for is used to specify this loop.
- The for loop contains three parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable

```
integer count;
initial
    for ( count=0; count < 128; count = count + 1)
        $display("Count = %d", count);
```

- The for loop provides a more compact loop structure than the while loop.
- However, the while loop is more general-purpose than the for loop.

Example

- Write a behavioral code using "for loop" to initialize all the even and odd locations of an array with 0 and 1 respectively.


```
//Initialize array elements
'define MAX_STATES 32
integer state [0: 'MAX_STATES-1];
integer i;
initial begin
    for(i = 0; i < 32; i = i + 2) state[i] = 0;
    for(i = 1; i < 32; i = i + 2) state[i] = 1;
end
```


Repeat Loop

- The keyword repeat is used for this loop.
- The repeat construct executes the loop a fixed number of times. A repeat construct cannot be used to loop on a general logical expression.
- A repeat construct must contain a number, which can be a constant, a variable or a signal value.

```
integer count;
initial
  begin
     count = 0;
     repeat(128)
       begin
          $display("Count = %d", count);
          count = count + 1;
       end
  end
```


Example

- Write the code for a data buffer, which after receiving a data_start signal, reads data for next 8 cycles.


```
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;
always @(posedge clock) begin
   if(data_start) //data start signal is true
   begin
       i = 0;
       repeat(cycles) //Store data at the posedge of next 8 clock cycles
       begin
           @(posedge clock) buffer[i] = data; //waits till next posedge to latch data
           i = i + 1;
       end
   end
end
endmodule
```

Forever Loop

- The keyword ***forever*** is used to express this loop.
- This construct is not synthesizable
- The loop does not contain any expression and executes forever until the $finish task is encountered.
- The loop is equivalent to a while loop with an expression that always evaluates to true.
- A forever loop can be exited by use of the disable statement

Eg: Synchronize two register values at every positive edge of clock

```
reg clock;
reg x, y;
initial
    forever @(posedge clock) x = y;
```


- Example: Clock generation - Use forever loop instead of always block


```
reg clock;
initial
begin
    clock = 1'b0;
    forever #10 clock = ~clock;
end
```


Sequential Statements in Verilog

1. begin
       sequential_statements
   end
2. if (expression)
       sequential_statement
   else
       sequential_statement
3. case (expression)
       expr: sequential_statement
       …….
       default: sequential_statement
   endcase

Sequential Statements in Verilog

4. forever
       sequential_statement
5. repeat (expression)
       sequential_statement
6. while (expression)
       sequential_statement
7. for (expr1; expr2; expr3)
       sequential_statement
8. # (time_value)
       Makes a block suspend for "time_value" time units.
9. @ (event_expression)
       Makes a block suspend until event_expression triggers.

Parameters and Generate Blocks

Objectives of this topic

- Parameter declaration and use
- How to *dynamically* generate Verilog code

Parameters

- Similar to `const` in C
  - But can be overridden for each module at compile-time
- Syntax:

```
parameter <const_id> = <value>;
```

- Gives flexibility
  - Allows to customize the module
- Example:

```
parameter port_id = 5;
parameter cache_line_width = 256;
parameter bus_width = 8;
parameter signed [15:0] WIDTH;

wire [bus_width-1:0] bus;
```


Parameter Example

```
module hello_world;
  parameter id_num = 0;

  initial
       $display("Displaying hello_world id number = %d", id_num);
endmodule

//define top-level module
module top;
  defparam w1.id_num = 1, w2.id_num = 2;
  hello_world w1();
  hello_world w2();
endmodule
```


Better Coding Style

```
module hello_world;
  parameter id_num = 0;

  initial
       $display("Displaying
   hello_world id num = %d",
   id_num);
endmodule

module top;
  defparam w1.id_num = 1,
     w2.id_num = 2;
  hello_world w1();
  hello_world w2();
endmodule
```

```
module hello_world
  #(parameter id_num = 0);

   initial
      $display("Displaying
   hello_world id num = %d",
   id_num);
endmodule


module top;
  hello_world #(1) w1();
  hello_world #(.id_num(2)) w2();
endmodule
```


Parameters (cont'd)

- `localparam` keyword

```
localparam state1 = 4'b0001,
           state2 = 4'b0010,
           state3 = 4'b0100,
           state4 = 4'b1000;
```

Operations for HDL simulation

- Compilation/Parsing
- Elaboration
  - Binding modules to instances
  - Build hierarchy
  - Compute parameter values
  - Resolve hierarchical names
  - Establish net connectivity
- Simulation

Generate Block

- Dynamically generate Verilog code at elaboration time
  - Usage:
    - Parameterized modules when the parameter value determines the module contents
  - Can generate
    - Modules
    - User defined primitives
    - Verilog gate primitives
    - Continuous assignments
    - `initial` and `always` blocks

Generate Loop

```
module bitwise_xor #(parameter N = 32)(output [N-1:0] out, input
   [N-1:0] i0, i1);
  genvar j; // This variable does not  exist during simulation

  generate for (j=0; j<N; j=j+1) begin: xor_loop
       //Generate the bit-wise Xor with a single loop
     xor g1 (out[j], i0[j], i1[j]);
    end
  endgenerate //end of the generate block

/* An alternate style using always blocks:
 reg [N-1:0] out;
 generate for (j=0; j<N; j=j+1) begin: bit
     always @(i0[j] or i1[j]) out[j] = i0[j] ^ i1[j];
   end
 endgenerate
endmodule  */
```

xor_loop is the name given to this begin-end block

If N=3, the instance names will be
xor_loop[0].g1
xor_loop[1].g1
xor_loop[2].g1

Example 2: Ripple Carry Adder

```
module ripple_adder #(parameter N=4) (output co, output [N-1:0] sum, input
   [N-1:0] a0, a1, input ci);
   wire [N:0] carry;
   assign carry[0] = ci;

   genvar i;
   generate for (i=0; i<N; i=i+1) begin: r_loop
        wire t1, t2, t3;
        xor g1 (t1, a0[i], a1[i]);
        xor g2 (sum[i], t1, carry[i]);
        and g3 (t2, a0[i], a1[i]);
        and g4 (t3, t1, carry[i]);
        or  g5 (carry[i+1], t2, t3);
    end
   endgenerate //end of the generate block
   assign co = carry[N];
endmodule
```

Note: hierarchical instance names

Generate Conditional

```
module multiplier (output [product_width -1:0] product, input [a0_width-1:0] a0,
          input [a1_width-1:0] a1);
    parameter       a0_width = 8;
    parameter       a1_width = 8;

    localparam      product_width = a0_width + a1_width;

    generate
          if (a0_width <8) || (a1_width < 8)
                cla_multiplier  #(a0_width, a1_width)   m0 (product, a0, a1);
          else
                tree_multiplier  #(a0_width, a1_width)   m0 (product, a0, a1);
    endgenerate

endmodule
```


Generate Case

```
module adder (output co, output [N-1:0] sum,
       input [N-1:0] a0, a1, input ci);

    parameter N = 4;

    // Parameter N that can be redefined at instantiation time.
    generate
          case (N)
                1:        adder_1bit        adder1(c0, sum, a0, a1, ci);
                2:        adder_2bit        adder2(c0, sum, a0, a1, ci);
                default: adder_cla   #(N)   adder3(c0, sum, a0, a1, ci);
          endcase
    endgenerate

endmodule
```

Nesting

- Generate blocks can be nested
  - Nested loops cannot use the same `genvar` variable

Some System Tasks and Compiler Directives

Compiler Directives

- Instructions to the *Compiler* (not *simulator*)
- General syntax:
  `` `<keyword> ``
- `` `define ``
  - similar to `#define` in C
  - `` `<macro_name> `` to use the macro defined by `` `define ``
- Examples:

```
`define WORD_SIZE 32
`define S $stop

`define WORD_REG reg [31:0]
`WORD_REG a_32_bit_reg;
```


Compiler Directives (cont'd)

- `` `undef ``
  - `Undefine a macro`
- Example:

```
`undef BUS_WIDTH
```

- `` `include ``
  - Similar to `#include` in C
- Example:

```
`include header.v
...
<Verilog code in file design.v>
...
```


Compiler Directives (cont'd)

- `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `elsif ``, and `` `endif ``.

```
`define TEST
`ifdef TEST //compile module test only if macro TEST is defined
   module test;
        ...
        ...
   endmodule
`else //compile the module stimulus as default
   module stimulus;
        ...
        ...
   endmodule
`endif //completion of 'ifdef directive
```


System Tasks

- System Tasks: standard routine operations provided by Verilog
  - Displaying on screen, monitoring values, stopping and finishing simulation, etc.
- All start with `$`
- Instructions for the *simulator*

System Tasks (cont'd)

- `$display`: displays values of variables, strings, expressions.

```
$display(p1, p2, p3, …, pn);
```

  - `p1,…, pn` can be quoted string, variable, or expression
  - Adds a new-line after displaying `pn` by default
  - Format specifiers:
    - `%d, %b, %h, %o`: display variable respectively in decimal, binary, hex, octal
    - `%c, %s`: display character, string
    - `%e, %f, %g`: display real variable in scientific, decimal, or whichever smaller notation
    - `%v`: display strength
    - `%t`: display in current time format
    - `%m`: display hierarchical name of this module

`$display` examples

```
$display("Hello Verilog World!");
```
Output: `Hello Verilog World!`

```
$display($time);
```
Output: `230`

```
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h",
  $time, virtual_addr);
```
Output: `At time 200 virtual address is 1fe000001c`

`$display` examples (cont'd)

```
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
```
Output: `ID of the port is 00101`

```
reg [3:0] bus;
$display("Bus value is %b", bus);
```
Output: `Bus value is 10xx`

```
$display("Hierarchical name of this module is %m");
```
Output: `Hierarchical name of this module is top.p1`

```
$display("A \n multiline string with a %% sign.");
```
Output:
```
A
      multiline string with a % sign.
```


`$monitor` System Task

- `$monitor`: monitors signal(s) and displays them when their value changes

```
$monitor(p1, p2, p3, …, pn);
```

  - `p1,…, pn` can be quoted string, variable, or signal names
  - Format specifiers similar to `$display`
  - Continuously monitors the values of the specified variables or signals, and displays the entire list whenever any of them changes.
  - `$monitor` needs to be invoked only once (unlike `$display`)
    - Only one `$monitor` (the latest one) can be active at any time
    - **`$monitoroff`** to temporarily turn off monitoring
    - **`$monitoron`** to turn monitoring on again

`$monitor` Examples

```
initial
  $monitor($time, "Value of signals clock=%b,
     reset=%b", clock, reset);
initial
  begin
   clock=0;
   reset=1;
   #5 clock=1;
   #10 clock=0; reset=0;
  end
```

- Output:

```
0 value of signals clock=0, reset=1
5 value of signals clock=1, reset=1
15 value of signals clock=0, reset=0
```


`$stop` System Task

- `$stop`: stops simulation
  - Simulation enters interactive mode
  - Most useful for debugging
- `$finish`: terminates simulation

- Examples:

```
initial
begin
  clock=0;
  reset=1;
  #100 $stop;
  #900 $finish;
end
```

Useful System Tasks: File I/O

Useful Modeling Techniques

Opening a File

- Opening a file

```
<file_handle> = $fopen( "<file_name>" );
```

  - `<file_handle>` is a 32 bit value, called *multi-channel descriptor*
  - Only 1 bit is set in each descriptor
  - Standard output has a descriptor of 1 (Channel 0)

```
//Multichannel descriptor
integer handle1, handle2, handle3; //integers are 32-bit values

//standard output is open; descriptor = 32'h0000_0001 (bit 0 set)
initial
begin
   handle1 = $fopen("file1.out"); //handle1=32'h0000_0002 (bit 1 set)
   handle2 = $fopen("file2.out"); //handle2=32'h0000_0004 (bit 2 set)
   handle3 = $fopen("file3.out"); //handle3=32'h0000_0008 (bit 3 set)
end
```


File Output and Closing

- Writing to files
  - `$fdisplay, $fmonitor, $fstrobe`
  - `$strobe, $fstrobe`
    - The same as `$display, $fdisplay`, but executed after all other statements schedule in the same simulation time
  - Syntax:

```
$fdisplay(<handle>, p1, p2,…, pn);
```

- Closing files

```
$fclose(<handle>);
```


Example: Simultaneously writing to multiple files

```
//All handles defined in Example 9-7
//Writing to files
integer desc1, desc2, desc3; //three file descriptors
initial
begin
   desc1 = handle1 | 1; //bitwise or; desc1 = 32'h0000_0003
   $fdisplay(desc1, "Display 1");//write to files file1.out & stdout

   desc2 = handle2 | handle1; //desc2 = 32'h0000_0006
   $fdisplay(desc2, "Display 2");//write to files file1.out & file2.out

   desc3 = handle3 ; //desc3 = 32'h0000_0008
   $fdisplay(desc3, "Display 3");//write to file file3.out only
end
```


Random Number Generation

- Syntax:

```
$random;
$random(<seed>);
```

- Returns a 32 bit random value

```
//Generate random numbers and apply them to a simple ROM
module test;
integer r_seed;
reg [31:0] addr;//input to ROM
wire [31:0] data;//output from ROM
…
…
ROM rom1(data, addr);

initial
   r_seed = 2; //arbitrarily define the seed as 2.

always @(posedge clock)
   addr = $random(r_seed); //generates random numbers

…
<check output of ROM against expected results>
…
…
endmodule
```


Useful System Tasks Initializing Memory from File

- Keywords:
  - `$readmemb, $readmemh`
- Used to initialize memory (`reg [3:0] mem[0:1023]`)
- Syntax:

```
$readmemb("<file_name>", <memory_name>);
$readmemb("<file_name>", <memory_name>, <start_addr>);
$readmemb("<file_name>", <memory_name>, <start_addr>,
   <finish_addr>);
```

- The same syntax for **`$readmemh`**

```
module test;

reg [7:0] memory[0:7]; //declare an 8-byte memory
integer i;

initial
begin
  //read memory file init.dat. address locations given in memory
  $readmemb("init.dat", memory);
  //display contents of initialized memory
  for(i=0; i < 8; i = i + 1)
    $display("Memory [%0d] = %b", i, memory[i]);
end

endmodule
```

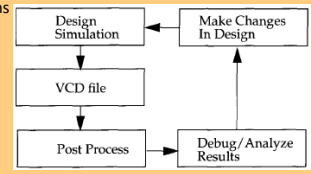
```
@02
11111111 01010101
00000000 10101010

@06
1111zzzz 00001111
```

```
Memory [0] = xxxxxxxx
Memory [1] = xxxxxxxx
Memory [2] = 11111111
Memory [3] = 01010101
Memory [4] = 00000000
Memory [5] = 10101010
Memory [6] = 1111zzzz
Memory [7] = 00001111
```

31

Useful System Tasks
Value Change Dump (VCD) File

- ASCII file containing information on
  - Simulation time
  - Scope and signal definitions
  - Signal value changes
- Keywords
  - `$dumpvars`
  - `$dumpfile`
  - `$dumpon`
  - `$dumpoff`
  - `$dumpall`



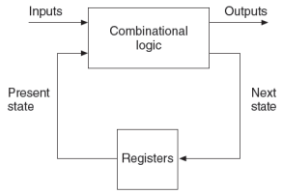

```
//specify name of VCD file. Otherwise,default name is
//assigned by the simulator.
initial
      $dumpfile("myfile.dmp"); //Simulation info dumped to myfile.dmp

//Dump signals in a module
initial
   $dumpvars; //no arguments, dump all signals in the design
initial
   $dumpvars(1, top); //dump variables in module instance top.
```

```
              //Number 1 indicates levels of hierarchy. Dump one
             //hierarchy level below top,i.e.,dump variables in top,
            //but not signals in modules instantiated by top.
initial
   $dumpvars(2, top.m1);//dump up to 2 levels of hierarchy below top.m1
initial
   $dumpvars(0, top.m1);//Number 0 means dump the entire hierarchy
                        // below top.m1

//Start and stop dump process
initial
begin
   $dumpon;            //start the dump process.
   #100000 $dumpoff;  //stop the dump process after 100,000 time units
end

//Create a checkpoint. Dump current value of all VCD variables
initial
   $dumpall;
```

33

Finite State Machines

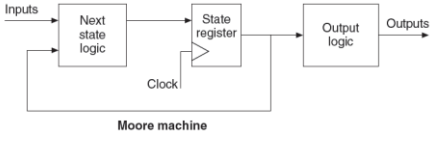
General Sequential System



Models of synchronous sequential systems

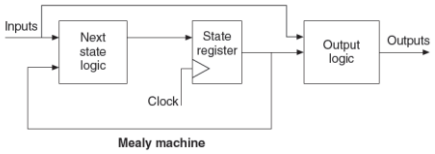
- Two common models of synchronous sequential systems
  - Moore machines
  - Mealy machines

Moore Machine



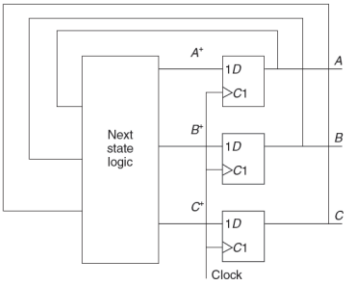
Moore machine

Mealy Machine



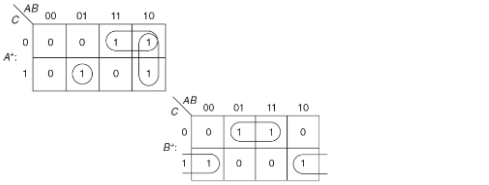
Mealy machine

Design of a three-bit counter



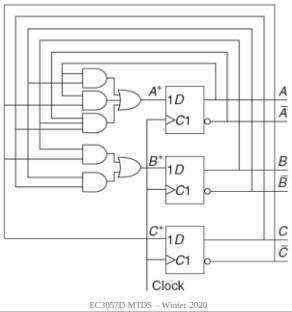
ABC	$A^+B^+C^+$
000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

K-maps for 3-bit counter



$$A^+ = A.\overline{C} + A.\overline{B} + \overline{A}.B.C$$
$$B^+ = B.\overline{C} + \overline{B}.C$$
$$C^+ = \overline{C}$$

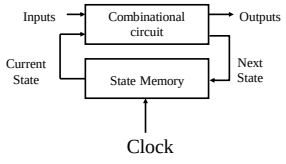
3-bit counter circuit



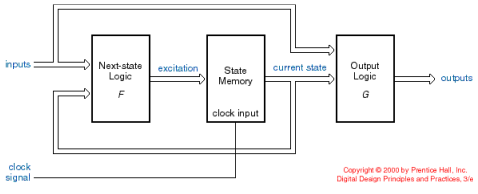
Synchronous Sequential Circuit Analysis

Synchronous Sequential Circuit

- State Memory – A set of *n* edge-triggered flip-flops that store the current state of the machine
  - All flip-flops are triggered from the same master clock signal
  - All flip-flops change state together
- Combinational circuit
  - Next state logic
  - Output logic – Mealy and Moore



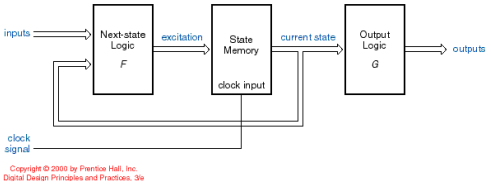
Mealy Model



next state = F (current state, inputs)

outputs = G (current state, inputs)

Moore Model



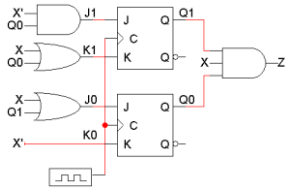
next state = F (current state, inputs)

outputs = G (current state)

Analysis - Goals

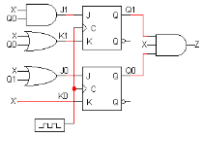
- Characterize as Mealy or Moore machine
- Determine next state equations, i.e., find the function F
  - next state = F (current state, inputs)
- Determine output equations
  - Mealy: outputs = G (current state, inputs), or
  - Moore: outputs = G (current state)
- Express as machine behavior
  - State table, or
  - State diagram
- Formulate English description of machine behavior

An example sequential circuit



- A sequential circuit with two JK flip-flops
- State or memory: Q1Q0
- One input: X; One output: Z

State table of example circuit



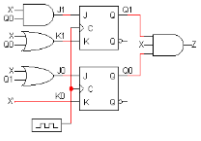
Present State Q_1	Q_0	Inputs X	Next State Q_1	Q_0	Outputs Z
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Output Equations

- From the diagram, you can see that

$Z = Q_1 Q_0 X$

Mealy model circuit !!!



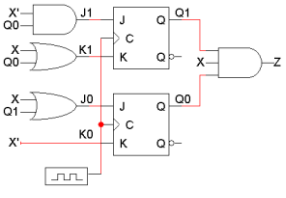
Present State Q_1	Q_0	Inputs X	Next State Q_1	Q_0	Outputs Z
0	0	0			0
0	0	1			0
0	1	0			0
0	1	1			0
1	0	0			0
1	0	1			0
1	1	0			0
1	1	1			1

Next State Equations – Q(t+1)

- Find the flip-flop input equations/excitation equations
- Substitute excitation equations in the flip-flop's characteristic equation

J1 = X' Q0
K1 = X + Q0

J0 = X + Q1
K0 = X'



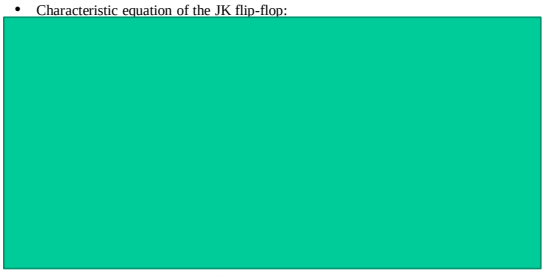
Next State Equations – Q(t+1)

- Excitation equations:

  $J_1 = X' Q_0$ and $K_1 = X + Q_0$

  $J_0 = X + Q_1$ and $K_0 = X'$

- Characteristic equation of the JK flip-flop:



Next State Equations – Q(t+1)

- Excitation equations:

  $J_1 = X' Q_0$ and $K_1 = X + Q_0$

  $J_0 = X + Q_1$ and $K_0 = X'$

- Characteristic equation of the JK flip-flop:

  $Q(t+1) = K'Q(t) + JQ'(t)$

- Next state equations:

Next State Equations – Q(t+1)

- Excitation equations:

  $J_1 = X' Q_0$ and $K_1 = X + Q_0$

  $J_0 = X + Q_1$ and $K_0 = X'$

- Characteristic equation of the JK flip-flop:

  $Q(t+1) = K'Q(t) + JQ'(t)$

- Next state equations:

$Q_1(t+1) = K_1'Q_1(t) + J_1Q_1'(t)$

$= (X + Q_0(t))' Q_1(t) + X' Q_0(t) Q_1'(t)$

$= X' (Q_0(t)' Q_1(t) + Q_0(t) Q_1(t)')$

$= X' (Q_0(t) \oplus Q_1(t))$

$Q_0(t+1) = K_0'Q_0(t) + J_0Q_0'(t)$

$= X Q_0(t) + (X + Q_1(t)) Q_0'(t)$

$= X + Q_0(t)' Q_1(t)$

State Table & Next State Equations

- $Q_1(t+1) = X' (Q_0(t) \oplus Q_1(t))$
  - $Q_1=0, Q_0=0, X= 0 \Rightarrow Q_1(t+1)= 0$
- $Q_0(t+1) = X + Q_0(t)' Q_1(t)$
  - $Q_1=0, Q_0=0, X= 0 \Rightarrow Q_0(t+1)= 0$

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1			0
0	1	0			0
0	1	1			0
1	0	0			0
1	0	1			0
1	1	0			0
1	1	1			1

State Table & Next State Equations

- $Q_1(t+1) = X' (Q_0(t) \oplus Q_1(t))$
  - $Q_1=0, Q_0=1, X= 1 \Rightarrow Q_1(t+1)= 0$
- $Q_0(t+1) = X + Q_0(t)' Q_1(t)$
  - $Q_1=0, Q_0=1, X= 1 \Rightarrow Q_0(t+1)= 1$

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1			0
0	1	0			0
0	1	1	0	1	0
1	0	0			0
1	0	1			0
1	1	0			0
1	1	1			1

State Table & Next State Equations

- $Q_1(t+1) = X' (Q_0(t) \oplus Q_1(t))$
- $Q_0(t+1) = X + Q_0(t)' Q_1(t)$

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

State Table & Characteristic Table

- The general JK flip-flop characteristic equation is:

$$Q(t+1) = K'Q(t) + JQ'(t)$$

- We can also determine the next state for each input/current state combination directly from the characteristic table

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

State Table & Characteristic Table

- With these equations, we can make a table showing J_1, K_1, J_0 and K_0 for the different combinations of present state Q_1Q_0 and input X

$J_1 = X' Q_0$ $\quad$ $J_0 = X + Q_1$

$K_1 = X + Q_0$ $\quad$ $K_0 = X'$

Present State Q_1	Present State Q_0	Inputs X	Flip-flop Inputs J_1	K_1	J_0	K_0
0	0	0	0	0	0	1
0	0	1	0	1	1	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
1	0	0	0	0	1	1
1	0	1	0	1	1	0
1	1	0	1	1	1	1
1	1	1	0	1	1	0

State Table & Characteristic Table

J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

Present State Q_1	Present State Q_0	Inputs X	FF Inputs J_1	K_1	J_0	K_0	Next State Q_1	Next State Q_0
0	0	0	0	0	0	1		
0	0	1	0	1	1	0		
0	1	0	1	1	0	1	1	
0	1	1	0	1	1	0		
1	0	0	0	0	1	1		
1	0	1	0	1	1	0		
1	1	0	1	1	1	1		
1	1	1	0	1	1	0		

State Table & Characteristic Table

J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

Present State Q_1	Present State Q_0	Inputs X	FF Inputs J_1	K_1	J_0	K_0	Next State Q_1	Next State Q_0
0	0	0	0	0	0	1		
0	0	1	0	1	1	0		
0	1	0	1	1	0	1	1	0
0	1	1	0	1	1	0		
1	0	0	0	0	1	1		
1	0	1	0	1	1	0		
1	1	0	1	1	1	1		
1	1	1	0	1	1	0		

A different look

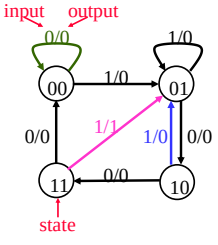
Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

Present State Q1	Present State Q0	Next State Input X= 0		Next State Input X= 1		Output Z X= 0	Output Z X= 1
0	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	1	0	1

State diagrams (Mealy model)

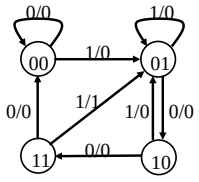
- We can also represent the state table graphically with a state diagram
- A diagram corresponding to our example state table is shown below

Present State Q_1	Present State Q_0	Inputs X	Next State Q_1	Next State Q_0	Outputs Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

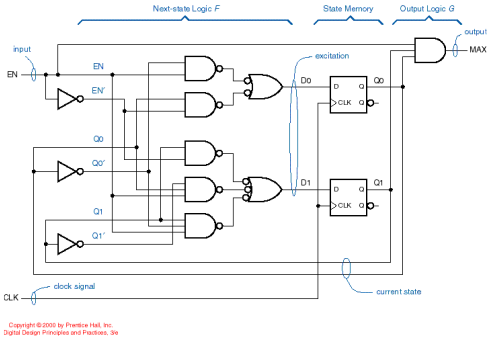


Sizes of state diagrams

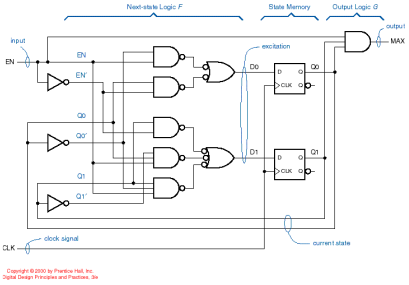
- Always check the size of your state diagrams
  - If there are *n* flip-flops, there should be 2^n nodes in the diagram
  - If there are *m* inputs, then each node will have 2^m outgoing arrows
- In our example,
  - We have two flip-flops, and thus four states or nodes.
  - There is one input, so each node has two outgoing arrows.



Another Mealy Circuit

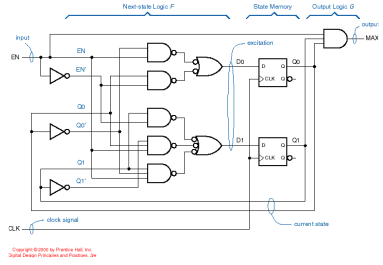


Excitation Equations



- $D_0 = EN' Q_0 + EN Q_0'$
- $D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$

Next State/Output Equations



- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAX = EN Q_1 Q_0$

Mealy State Table

- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAX = EN Q_1 Q_0$

Present State Q1	Present State Q0	Next State Input EN= 0		Next State Input EN= 1		Output MAX EN= 0	Output MAX EN= 1
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	0	1

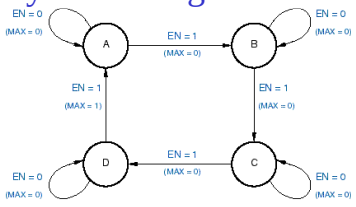
Mealy State Table

Present State Q1	Present State Q0	Next State Input EN= 0		Next State Input EN= 1		Output MAX EN= 0	Output MAX EN= 1
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	0	1

State Q1	State Q0	State Name
0	0	A
0	1	B
1	0	C
1	1	D

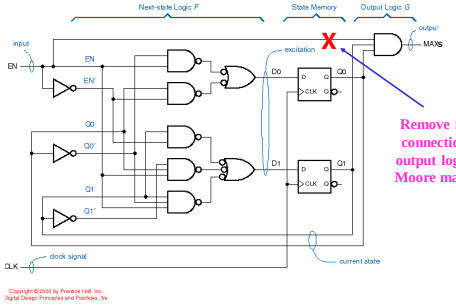
Present State Q1 Q0	Next State Input EN= 0	Next State Input EN= 1	Output MAX EN= 0	Output MAX EN= 1
A	A	B	0	0
B	B	C	0	0
C	C	D	0	0
D	D	A	0	1

Mealy State Diagram



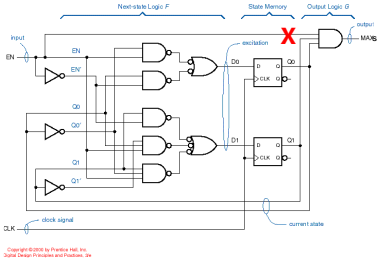
Present State Q1 Q0	Next State Input EN= 0	Next State Input EN= 1	Output MAX EN= 0	Output MAX EN= 1
A	A	B	0	0
B	B	C	0	0
C	C	D	0	0
D	D	A	0	1

Moore Circuit



Remove input connection to output logic => Moore machine

Next State/Output Equations



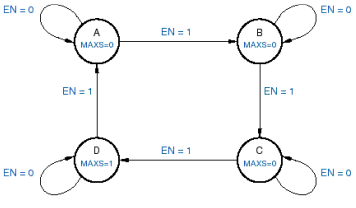
- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAXS = Q_1 Q_0$

Moore State Table

- $Q_0(t+1) = D_0 = EN' Q_0 + EN Q_0'$
- $Q_1(t+1) = D_1 = EN' Q_1 + EN Q_1' Q_0 + EN Q_1 Q_0'$
- $MAXS = Q_1 Q_0$

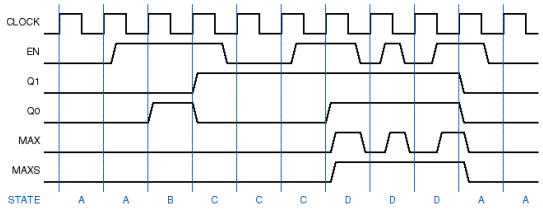
Present State Q1	Present State Q0	Next State Input EN= 0		Next State Input EN= 1		Output MAXS
0	0	0	0	0	1	0
0	1	0	1	1	0	0
1	0	1	0	1	1	0
1	1	1	1	0	0	1

Moore State Diagram



Present State Q1	Present State Q0	Next State Input EN= 0		Next State Input EN= 1		Output MAXS
0	0	0	0	0	1	0
0	1	0	1	1	0	0
1	0	1	0	1	1	0
1	1	1	1	0	0	1

State Transitions



- MAX : Output of the Mealy circuit
- MAXS : Output of the Moore circuit

Sequential circuit analysis summary

- To analyze sequential circuits, you have to:
  - Find Boolean expressions for the outputs of the circuit and the flip-flop inputs
  - Use these expressions to fill in the output and flip-flop input columns in the state table
  - Finally, use the characteristic equation or characteristic table of the flip-flop to fill in the next state columns.
- The result of sequential circuit analysis is a state table or a state diagram describing the circuit

Design of Sequence Detector

Sequence Detector

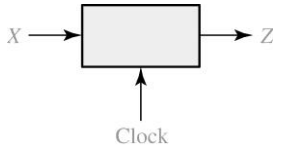
- A circuit that detects the occurrence of a particular pattern on its input is referred to as a sequence detector.

Design a circuit that examine a string of 0's and 1's applied to the input X and for any input sequence ending in 101 will produce an output Z=1 coincident with the last 1.

The circuit does not reset when a 1 output occur.

We assume that the input X can only change between clock pulses

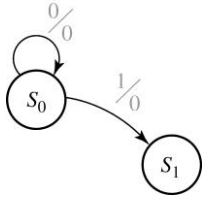
101 Sequence Detector



X =	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**
Z =	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**
(time:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15)**

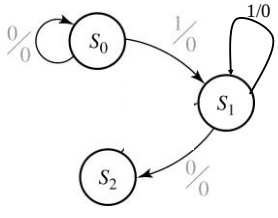
Design of 101 Sequence Detector

- State Diagram:



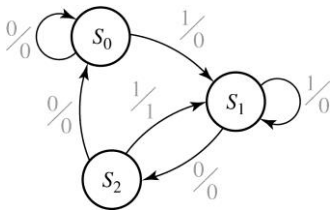
Design of 101 Sequence Detector

- State Diagram:



Design of 101 Sequence Detector

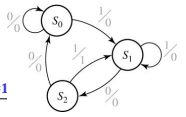
- State Diagram (final):



Design of 101 Sequence Detector

- State Table:

Present state	Next State X = 0	Next State X = 1	Present Output X = 0	Present Output X =1
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1



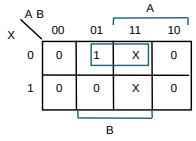
Assign a unique binary code to each state name (State Assignment)

- State Table with State Assignment:

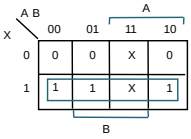
AB	$D_A D_B$ = A^+B^+ X = 0	$D_A D_B$ = A^+B^+ X = 1	Z X = 0	Z X =1
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

Design of Sequence Detector

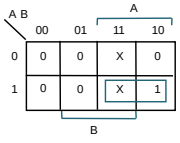
- Derive Boolean Equations:



$D_A = X'.B$

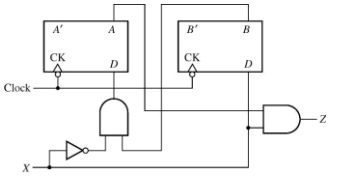


$D_B = X$

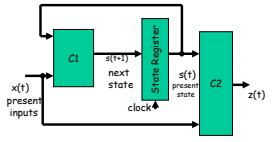


$Z = X.A$

Design of Sequence Detector



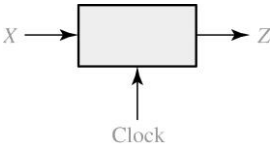
Compare with Typical Mealy Machine



Design of Sequence Detector

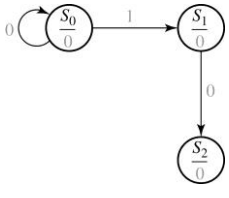
- A Moore Sequence Detector:

101 sequence Detector

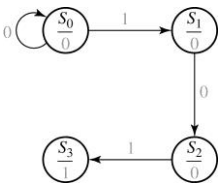


X =	0	0	1	1	0	1	1	0	0	1	0	1	0	1	0	0
Z =	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0
(time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15)

Design of a Sequence Detector



Design of a Sequence Detector



S_0: start

S_1: got 1

S_2: got 10

S_3: got 101

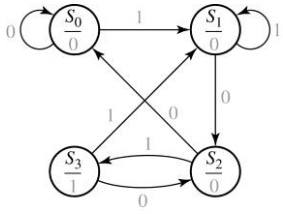
Design of a Sequence Detector

S_0: start

S_1: got 1

S_2: got 10

S_3: got 101



Design of a Sequence Detector

State Table

Present state	Next State X = 0	Next State X = 1	Present Output (Z)
S_0	S_0	S_1	0
S_1	S_2	S_1	0
S_2	S_0	S_3	0
S_3	S_2	S_1	1

Transition Table with State assignment

AB	A^+B^+ X = 0	A^+B^+ X = 1	Z
00	00	01	0
01	11	01	0
11	00	10	0
10	11	01	1

State Diagram Development

- To develop a sequence detector state diagram:
  1. Construct some sample input and output sequences to make sure that you understand the problem statement.
  2. Begin in an initial state in which NONE of the initial portion of the sequence has occurred (typically "reset" state).
  3. Add a state that recognizes that the first symbol has occurred.
  4. Add states that recognize each successive symbol occurring.
  5. Each time you add an arrow to the state graph, determine it can go to one of the previously defined states or whether a new state must be added
  6. The final state represents the input sequence occurrence.
  7. Add state transition arcs which specify what happens when a symbol *not* in the proper sequence has occurred.
  8. Check your state graph for completeness and non-redundant arcs.
  9. When your state graph is complete, test it by applying the input sequences formulated in part1 and making sure the output sequences are correct

Sequential circuit design procedure

Step 1:
Make a state table based on the problem statement. The table should show the present states, inputs, next states and outputs. (It may be easier to find a state diagram first, and then convert that to a table)

Step 2:
Assign binary codes to the states in the state table, if you haven't already. If you have n states, your binary codes will have at least $\lceil \log_2 n \rceil$ digits, and your circuit will have at least $\lceil \log_2 n \rceil$ flip-flops

Step 3:
For each flip-flop and each row of your state table, find the flip-flop input values that are needed to generate the next state from the present state. You can use flip-flop excitation tables here.

Step 4:
Find simplified equations for the flip-flop inputs and the outputs.

Step 5:
Build the circuit!

Another Example

Sequence detector (Mealy)

- A sequence detector is a special kind of sequential circuit that looks for a special bit pattern in some input
- The detector circuit has only one input, X
  - One bit of input is supplied on every clock cycle
  - This is an easy way to permit arbitrarily long input sequences
- There is one output, Z, which is 1 when the desired pattern is found
- Our example will detect the bit pattern "1001":

  Inputs: 1 1 1 0 0 1 1 0 1 0 0 1 0 0 1 1 0 …
  Outputs: 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 …

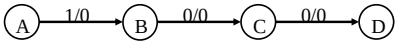
- A sequential circuit is required because the circuit has to "remember" the inputs from previous clock cycles, in order to determine whether or not a match was found

Step 1: Making a state table

- The first thing you have to figure out is precisely how the use of state will help you solve the given problem
  - Make a state table based on the problem statement. The table should show the present states, inputs, next states and outputs
  - Sometimes it is easier to first find a state diagram and then convert that to a table
- This is usually the most difficult step. Once you have the state table, the rest of the design procedure is the same for all sequential circuits

A basic Mealy state diagram

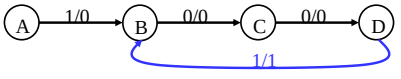
- What state do we need for the sequence detector?
  - We have to "remember" inputs from previous clock cycles
  - For example, if the previous three inputs were 100 and the current input is 1, then the output should be 1
  - In general, we will have to remember occurrences of parts of the desired pattern—in this case, 1, 10, and 100
- We'll start with a basic state diagram:



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

Overlapping occurrences of the pattern

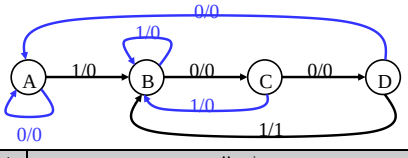
- What happens if we're in state D (the last three inputs were 100), and the current input is 1?
  - The output should be a 1, because we've found the desired pattern
  - But this last 1 could also be the start of another occurrence of the pattern! For example, 1001001 contains *two* occurrences of 1001
  - To detect overlapping occurrences of the pattern, the next state should be B.



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

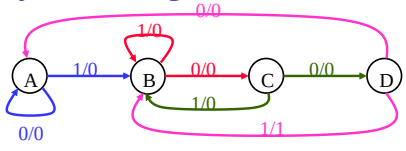
Filling in the other arrows

- *Two* outgoing arrows for each node, to account for the possibilities of X=0 and X=1
- The remaining arrows we need are shown in blue. They also allow for the correct detection of overlapping occurrences of 1001.



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

Mealy state diagram & table



Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	B	1

Step 2: Assigning binary codes to states

- We have four states ABCD, so we need at least two flip-flops Q_1Q_0
- The easiest thing to do is represent state A with Q_1Q_0 = 00, B with 01, C with 10, and D with 11
- The state assignment can have a big impact on circuit complexity, but we won't worry about that too much in this class

Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	B	1

Present State Q_1	Q_0	Input X	Next State Q_1	Q_0	Output Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

Step 3: Finding flip-flop input values

- Next we have to figure out how to actually make the flip-flops change from their present state into the desired next state
- This depends on what kind of flip-flops you use!
- We'll use two JKs. For each flip-flip Q_i, look at its present and next states, and determine what the inputs J_i and K_i should be in order to make that state change.

Present State Q_1	Q_0	Input X	Next State Q_1	Q_0	Flip flop inputs J_1	K_1	J_0	K_0	Output Z
0	0	0	0	0					0
0	0	1	0	1					0
0	1	0	1	0					0
0	1	1	0	1					0
1	0	0	1	1					0
1	0	1	0	1					0
1	1	0	0	0					0
1	1	1	0	1					1

JK excitation table

- An excitation table shows what flip-flop inputs are required in order to make a desired state change

Q(t)	Q(t+1)	J	K	Operation
0	0	0	x	No change/reset
0	1	1	x	Set/complement
1	0	x	1	Reset/complement
1	1	x	0	No change/set

- This is the same information that's given in the characteristic table, but presented "backwards"

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

- Use the JK excitation table on the right to find the correct values for *each* flip-flop's inputs, based on its present and next states

Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Present State Q_1	Q_0	Input X	Next State Q_1	Q_0	Flip flop inputs J_1	K_1	J_0	K_0	Output Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1

Step 4: Find equations for the FF inputs and output

- Now you can make K-maps and find equations for each of the four flip-flop inputs, as well as for the output Z
- These equations are in terms of the present state and the inputs
- The advantage of using JK flip-flops is that there are many don't care conditions, which can result in simpler MSP equations

Present State Q_1	Q_0	Input X	Next State Q_1	Q_0	Flip flop inputs J_1	K_1	J_0	K_0	Output Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1

FF input equations

Present State Q_1	Q_0	Input X	Next State Q_1	Q_0	Flip flop inputs J_1	K_1	J_0	K_0	Output Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1

J_1 K-map (Q_1Q_0 columns 00, 01, 11, 10; X rows 0, 1):

X	00	01	11	10
0	0	1	x	x
1	0	0	x	x

$J_1 = X'Q_0$

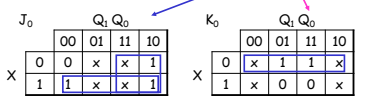
K_1 K-map (Q_1Q_0 columns 00, 01, 11, 10; X rows 0, 1):

X	00	01	11	10
0	x	x	1	0
1	x	x	1	1

$K_1 = X + Q_0$

FF input equations

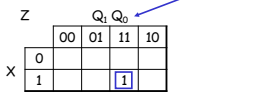
Present State Q_1	Present State Q_0	Input X	Next State Q_1	Next State Q_0	J_1	K_1	J_0	K_0	Output Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1



$J_0 = X + Q_1$ $K_0 = X'$

Output equation

Present State Q_1	Present State Q_0	Input X	Next State Q_1	Next State Q_0	J_1	K_1	J_0	K_0	Output Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1



$Z = X\,Q_1\,Q_0$

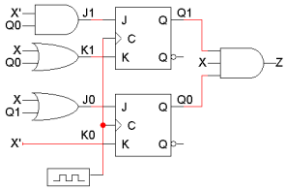
Step 5: Build the circuit

- Lastly, we use these simplified equations to build the completed circuit

$J_1 = X'\,Q_0$
$K_1 = X + Q_0$

$J_0 = X + Q_1$
$K_0 = X'$

$Z = Q_1 Q_0 X$



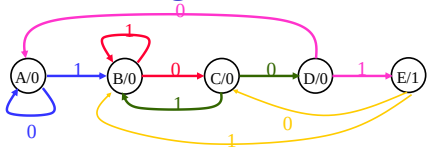
Sequence detector (Moore)

- A sequence detector is a special kind of sequential circuit that looks for a special bit pattern in some input
- The detector circuit has only one input, X
  - One bit of input is supplied on every clock cycle
  - This is an easy way to permit arbitrarily long input sequences
- There is one output, Z, which is 1 when the desired pattern is found
- Our example will detect the bit pattern "1001":

  Inputs: 1 1 1 0 0 1 1 0 1 0 0 1 0 0 1 1 0 …
  Outputs: 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 …

- A sequential circuit is required because the circuit has to "remember" the inputs from previous clock cycles, in order to determine whether or not a match was found

Moore state diagram & table



Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	E	0
E	0	C	1
E	1	B	1

Circuit design is left as an exercise !

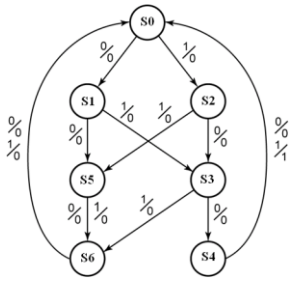
Comparison of Mealy and Moore FSM

- **Mealy** machines have **less states**
  - outputs are on transitions (n^2) rather than states (n)
- **Moore** machines are **safer** to use
  - outputs change at clock edge (always one cycle later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback may occur if one isn't careful
- **Mealy** machines **react faster** to inputs
  - react in same cycle – don't need to wait for clock
  - outputs may be considerably shorter than the clock cycle
  - in Moore machines, more logic may be necessary to decode state into outputs – there may be more gate delays after clock edge

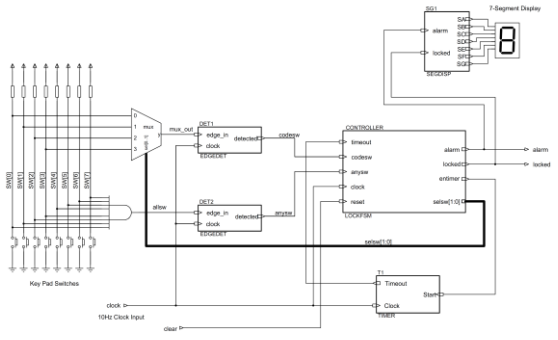
Example: Sequence Detector

A sequential circuit has one input (X) and one output (Z). The circuit examines groups of four consecutive inputs and produces an output $Z = 1$ if the input sequence 0101 or 1001 occurs. The circuit resets after every four inputs. Find a Mealy state graph. A typical input and output sequence is:

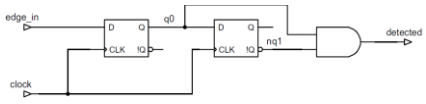
X =	0101	0010	1001	0100
Z =	0001	0000	0001	0000



Combination Lock FSM with Automatic Lock Feature



Logic diagram of edge detector `edgedet`




```
1  module edgedet(input edge_in,
2                 output detected,
3                 input clock);

4  wire q0, q1;

5  dff dff0(.q(q0), .d(edge_in), .clk(clock));
6  dff dff1(.q(q1), .d(q0), .clk(clock));

7  assign detected = q0 & ~q1;

8  endmodule
```

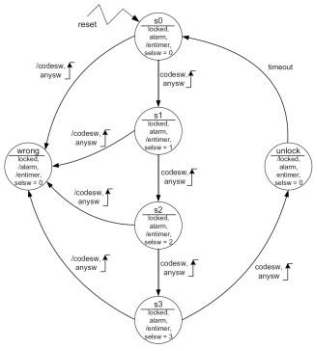

Timer

```
1  module Timer(input Clock, Start, output Timeout);
2  //time delay value in clk pulses
3  localparam NUMCLKS = 300;

4  reg[ 8:0] q;
5  always @(posedge Clock)
6  begin
7    if (!Start||(q == NUMCLKS))
8      q <= 9'b0;
9    else
10     q <= q + 1;
11 end
12 //decode counter output
13 assign Timeout = (q == NUMCLKS);

14 endmodule
```


Seven Segment Display

```
1  module segdisp(input locked,alarm,
2                 output SA,SB,SC,SD,SE,SF,SG);

3  reg[ 6:0] seg;

4  always @(locked or alarm)
5  begin
6    if (alarm == 0)
7      seg = 7'b0001000; //display 'A'
8    else if (locked == 0)
9      seg = 7'b1000001; //display 'U'
10   else
11     seg = 7'b1110001; //display 'L'
12 end

13 assign{SA, SB, SC, SD, SE, SF, SG} = seg;

14 endmodule
```

FSM




```
1  module lockfsm(input clock, reset,
2              codesw, anysw,
3              output reg [1:0] selsw,
4              output locked, alarm, entimer,
5              input timeout);

6  localparam s0=3'b000, s1=3'b001, s2=3'b010,
7              s3=3'b011,
8              wrong=3'b100, unlock=3'b101;

9  reg [2:0] lockstate;

10 always @(posedge clock or posedge reset)
11 begin
12 if (reset == 1'b1)
13   lockstate <= s0;
14 else
15   case (lockstate)
16     s0 : if (anysw & codesw)
17        lockstate <= s1;
18      else if (anysw)
19        lockstate <= wrong;
20      else
21        lockstate <= s0;
22     s1 : if (anysw & codesw)
23        lockstate <= s2;
24      else if (anysw)
25        lockstate <= wrong;
26      else
27        lockstate <= s1;
28     s2: if (anysw & codesw)
29        lockstate <= s3;
30      else if (anysw)
31        lockstate <= wrong;
```



```
32      else
33        lockstate <= s2;
34     s3: if (anysw & codesw)
35        lockstate <= unlock;
36      else if (anysw)
37        lockstate <= wrong;
38      else
39        lockstate <= s3;
40     wrong: lockstate <= wrong;
41     unlock: if (timeout)
42        lockstate <= s0;
43      else
44        lockstate <= unlock;
45    default: lockstate <= 3'bx;
46   endcase
47 end
```



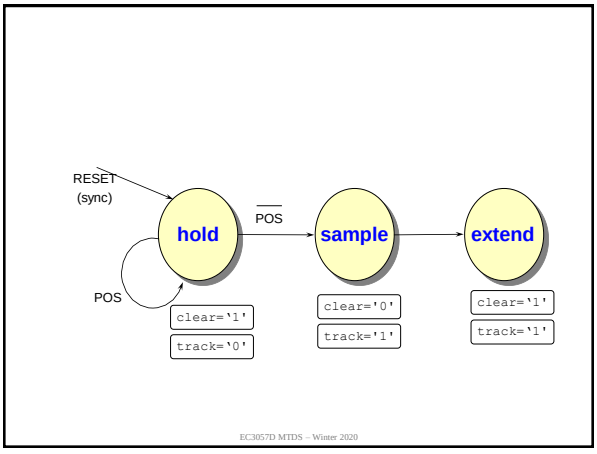
```
48 always @(lockstate)
49 begin
50   case(lockstate)
51     s0: selsw = 0;
52     s1: selsw = 1;
53     s2: selsw = 2;
54     s3: selsw = 3;
55     wrong: selsw = 0;
56     unlock: selsw = 0;
57     default: selsw = 2'bx;
58   endcase
59 end

60 assign locked = (lockstate == unlock) ? 0: 1;
61 assign alarm = (lockstate == wrong) ? 0: 1;

62 assign entimer = (lockstate == unlock) ? 1: 0;

63 endmodule
```



```
1  module comblock(input clock, clear,
2              input [7:0] switches,
3              output alarm, locked,
4              output SA, SB, SC, SD, SE, SF, SG);

5  wire mux_out, anysw, codesw,
6       allsw, entimer, timeout;

7  wire [1:0] selsw;

8  //4-to-1 multiplexor
9  assign mux_out = selsw == 0 ? switches[0] :
10             (selsw == 1 ? switches[1] :
11             (selsw == 2 ? switches[2] :
12             (selsw == 3 ? switches[3] : 1'b0)));

13 //AND gate for all switches
14 assign allsw = &switches;

15 edgedet det1(.edge_in(mux_out),
16              .detected(codesw),
17              .clock(clock));

18 edgedet det2(.edge_in(allsw),
19              .detected(anysw),
20              .clock(clock));

21 Timer t1(.Clock(clock),
22             .Start(entimer),
23             .Timeout(timeout));

24 lockfsm controller(.clock(clock),
25             .reset(clear),
26             .codesw(codesw),
27             .anysw(anysw),
28             .selsw(selsw),
29             .locked(locked),
30             .alarm(alarm),
31             .entimer(entimer),
32             .timeout(timeout));
```



```
33 segdisp sg1(.locked(locked),
34              .alarm(alarm),
35              .SA(SA),

36              .SB(SB),
37              .SC(SC),
38              .SD(SD),
39              .SE(SE),
40              .SF(SF),
41              .SG(SG));
42 endmodule
```


RESET
(sync)
hold
POS
sample
extend
POS
clear='1'
track='0'
clear='0'
track='1'
clear='1'
track='1'
EC3057D MTDS – Winter 2020