Notes on Elementary Verilog-based Digital Logic Design

(edited / compiled by)
Sachin B. Patkar,
(with contributions from)
Janak Porwal, Vinay B.Y., Jasveer Singh Jethra,
Madhumita P. Date, Mahesh Bhaganagare, Jeebu Thomas
David Dhas, Priyanka Porwal
(Dept. EE, IIT Bombay, Mumbai, India)

Latest update (8 years after last update in 2014)

mainly to replace unnecessary uses of blocking assignments by non-blocking assignments

Caution : Don't look for any advanced Verilog and advanced Digital Design here !

January 26, 2022

Chapter 1

Review of Basic Combinational Logic

Disclaimer : These notes are constituted significantly based on portions of published material of various texts and other sources. The influence of the already published material is retained in order for these notes to focus on the conventional and basic topics in synchronous digital logic design. The aim is to make these notes easy and compact enough to enable quick-start of Verilog based synchronous digital logic design.

1.1 Boolean Algebra, Logic Gates etc.

Boolean Algebra is an algebra formalized by George Boole. An important special Boolean Algebra is so called Switching (Boolean) Algebra. In this booklet, we will concern ourselves only with switching boolean algebra, and therefore we will allow ourselves liberty of referring to this special boolean algebra as boolean algebra itself.

(Switching) Boolean algebra is an algebra with a rich set of operators on 2-valued (ie. binary) variables. These two values are normally taken as 1 or 0, or equivalently true and false. It can indeed be any pair of distinct values.

(Switching or two-valued) Boolean algebra is a natural algebra to process digital information which is mostly encoded in binary form. And in typical computing machines and in computers this is indeed the base, because over last century, the field of digital electronics has made fantastic progress in facilitating massive storage of information in binary form using almost nanoscale electronics components which can store 0/1 valued information, as well as manipulate such information in a variety of ways. Such digital electronics circuits use low and high voltages to represent binary 0 and 1.

Commonly 0 is regarded as 'false' and the binary value 1 is regarded as 'true'.

Logic Gates are typically digital electronics circuit components, which realize the boolean algebra operators such as 'AND', 'OR', 'NOT', 'XOR' etc. By a "logic circuit" one means a network of such logic gates. Certain types of logic circuits can be used to physically realize any given expression in boolean algebra. (Boolean expressions are constructed by syntactically correct combinations of the boolean variables and boolean algebra operators). An expression in (switching) Boolean Algebra may naturally be regarded as a function which takes as inputs the variables which appear in the expression, and the value of the function is given by evaluating the expression any specified values of the input variables. Indeed, if a logic circuit does not have any feedbacks, then it represents some boolean expression, that is a function in switching boolean algebra. We call such boolean function, a "combinational logic function". By the virtue of being a function, the value of the output of the logic circuit to which the combinational logic function corresponds to, depends only on the current values of the inputs. Thus there is no memory involved.

The other types of logic circuits are called "sequential logic circuits". They are characterized by use of memory elements, which store the relevant knowledge about the past inputs, so as to let this past (memorized) knowledge, along with the current values of the inputs, influence the current values of output of such logic circuits.

1.1.1 Explicit Representation Combinational Logic Functions using Truth Tables

Because a combinational logic function can be completely specified by a truth table. For a combinational logic function with k input variables, there would be 2^k rows in the truth table, one each corresponding to a possible combinations of input values. In each row of the truth table, the value of output is specified for the particular input combination.

Truth Tables

To illustrate, we show a truth table of a combinational logic block with three inputs, A, B, and C, and two outputs, The function is defined as follows: D is true if at least one input is Notice $2^3 = 8$ entries in this truth table, corresponding to different possible combinations of binary values of the 3 input variables.

Truth Table for 3-variable odd-parity and majority function

Clearly this explicit, bruth-force representation of combinational logic functions is far too expensive due to exponential growth in the size of the truth-tables. We need cleverer ways of representing combinational logic functions, and these should ideally also be computer-friendly, as these days one relies heavily on Logic Synthesis tools, to perform logic design and analysis.

1.1.2 Brief Recap of Boolean Algebra

The three most commonly and fundamental operators of Boolean Algebra are AND, OR, NOT.

- The OR operator is commonly denoted by + (as it may be regarded as logical-sum), as in (A + B). Application of OR operator on two binary variables results in 1 if and only if either of the input variables takes value 1.

- The AND operator is commonly denoted by . (as it may be regarded as logical-product), as in (A . B). Application of AND operator on two binary variables results in 1 if and only if both the input variables takes value 1
- The unary operator NOT is commonly denoted by $\bar{A}$ or as A', which represents complement of 'A'. Thus $\bar{A}$ represents NOT(A). Application of NOT operator results in complemented value of the input.

Boolean algebra is governed by some laws, which are crucial in effective manipulations of combinational logic expressions.

- Identity law: $x + 0 = x$ and $x.1 = x$.
- Null or Dominance laws: $x + 1 = 1$ and $x.0 = 0$.
- Complementation laws: $x + \bar{x} = 1$ and $x.\bar{x} = 1$.
- Double complement law : $\bar{\bar{x}} = x$.
- Commutative laws: $x + y = y + x$ and $x.y = y.x$.
- Idempotent Laws: $x + x = x$ and $x.x = x$.s
- Absorption Laws: $x.(x + y) = x$, and $x + (x.y) = x$.
- $x.(\bar{x} + y) = x.y$ and $x + (\bar{x}.y) = x + y$.
- Associative laws: $x + (y + z) = (x + y) + z$ and $x.(y.z) = (x.y).z$.
- Distributive laws: $x.(y+z) = (x.y)+(x.z)$ and $x+(y.z) = (x+y).(x+z)$.
- DeMorgan's laws: $\overline{x + y} = \bar{x}.\bar{y}$, and $\overline{x.y} = \bar{x} + \bar{y}$.

1.1.3 Representing Gates Pictorially

The standard schematic symbols for logic gates are as below.

Any combinational logic function can be constructed using gates only from the collection of AND gates, OR gates, and NOT gates. We express this differently by saying that the collection {AND, OR, NOT} of logic gate types forms a "complete set" of logic gates. Mildly surprising, but well-known and easy-to-prove assertion is that the singleton collection { NAND } of logic gate types also forms a complete set of logic gate types. Therefore one also calls NAND gate as an "universal logic gate". Similarly NOR gate is also "universal".

1.1.4 Verilog HDL notation for Logic Circuit Schematic

Verilog HDL can be used to describe combinational and sequential circuits.

In this subsection, we use absolutely elementary notions in Verilog to describe combinational circuits. You may treat these descriptions as just a lazy alternative to drawing schematic for logic circuits (here the "laziness" refers to the reluctance to drawing diagrams using software packages).

Using Verilog HDL we can describe logic circuits as (Verilog) modules. In general a logic circuit module is constituted from combinational logic subcircuits and sequential logic subcircuits. To begin with we will see a few examples only of combinational type.

Within a Verilog module we can describe a logic circuit as a interconnected network of instances of logic circuit components, or as a collection of combinational logic functions, wherein each logic function is expressed using a a so-called continuous assignment statement, in which on the LHS, we have the name of the output of the combinational function; and on the RHS, we provide a boolean logic expression representing the combinational logic of the subcircuit.

Across collection of such combinational subcircuits, we could have outputs of some subcircuit appearing as variables within boolean logic expressions on RHS that describes the combinational logic of some other subcircuit.

```
module xor_AND_OR_network ( a_xor_b, a, b );
  input a, b;
  output a_xor_b;

  wire a_inv, b_inv, a_and_b_inv, b_and_a_inv;
```

```
  not  g1 (a_inv, a);
  not  g2 (b_inv, b);
  and g3 (a_and_b_inv, a, b_inv);
  and g4 (b_and_a_inv, b, a_inv);
  or   g5 (a_xor_b, a_and_b_inv, b_and_a_inv );
endmodule

module xor_SOP_network ( a_xor_b, a, b );
  input a, b;
  output a_xor_b;

  wire a_inv, b_inv, a_and_b_inv, b_and_a_inv;

  assign a_inv = ~a;
  assign b_inv = ~b;
  assign a_and_b_inv = a & b_inv;
  assign b_and_a_inv = b & a_inv;
  assign a_xor_b = a_and_b_inv | b_and_a_inv;

endmodule

module xor_SOP_expression ( a_xor_b, a, b );
  input a, b;
  output a_xor_b;

  assign a_xor_b = ( a & ~b) | ( ~a & b );

endmodule

module xor_primitive_operator ( a_xor_b, a, b );
  input a, b;
  output a_xor_b;

  assign a_xor_b = a ^ b;

endmodule

module mux2_1 ( mux_in0, mux_in1, select, mux_out );
  input  mux_in0, mux_in1, select;
  output mux_out;

  assign mux_out=(~select & mux_in0) | (select & mux_in1);
endmodule
```

```
module decoder2_4 ( in0, in1, out0, out1, out2, out4 );
  input in0, in1;
  output out0, out1, out2, out3;

  assign out0 = ~in0 & ~in1;
  assign out1 = ~in0 &   in1;
  assign out2 =   in0 & ~in1;
  assign out3 =   in0 &   in1;

endmodule
```

```
module half_adder(sum, carry, a, b);
  input a,b;
  output sum, carry;

  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

```
module full_adder(a, b, carry_in, sum, carry_out);
  input a, b, carry_in;
  output sum, carry_out;

  assign sum = a ^ b ^ carry_in ;
  assign carry_out = (a & b) | (a & carry_in) | (b & carry_in);

endmodule
```

1.2 Representing Boolean Functions

A combinational logic function can be represented in various forms. All these different forms of a same boolean logic expression (or combinational logic function) would have the same truth table. Thus we say that the truth-table is a canonical representation, as equivalent boolean expressions have the same truth-table. Good news is that there are a few other canonical forms too, which are useful in theory as well as practice. Other than the canonical forms, there are some more standard forms of representing combinational logic functions.

The two well-known standard forms are the sum-of-products (SOP) form and the product- of-sums (POS) form. These are standard, but not canonical, in the sense that same combinational logic function can admit several

different SOP forms, and same can be said of POS forms. Note that every combinational logic function has a unique representation in any canonical form, say as a truth-table canonical form or the so-called min-term canonical form.

In the sum-of-products (SOP) form the expression is formed by performing logical-OR of subexpressions each of which is a logical-AND of variables or their complements. For instance, the function majority(x,y,z) can be represented in SOP form as xy + yz + xy'z + xyz. However, xy + z(x+y) which also represents the same "majority(x,y,z)" function is not in SOP form. In the products-of-sums (POS) form the expression is formed by performing logical-AND of subexpressions each of which is a logical-OR of variables or their complements.

1.3 Some Standard Combinational Logic Components

We briefly mention a few combinational logic components which are popular building blocks in logic circuit design.

1.3.1 Decoders

There are a variety of decoders naturally required in combinational logic design tasks. By the standard decoder, one refers to the decoder that has k-bit input and 2^k outputs, and in which only one output is asserted for an input combination. Thus this decoder converts the k-bit input into a 2^k-bit output that is "one-hot". The only "hot" (ie. asserted) output bit, is at the index represented by the k-bit input. Let the output bits be enumerated as out[0], out[1], . . . , out[$2^k - 1$]. If the k-bit input represents the number i, then the bit out[i] will be asserted (ie. 'hot') and all other outputs will be false (ie. 'cold').

1.3.2 Multiplexer

A multiplexer is a very important powerrful combinational logic building block. Indeed multiplexer is a universal logic gate, you will be able to analyze this claim easily. A multiplexer may also be viewed as a data-selector. It "routes" the binary value that appears on one of the input lines to a single output line. The selection is specified by a set of control lines.

The most basic one is a two-input multiplexor. The combinational logic function realized by such two-input multiplexor

$$out = (in0.\bar{sel}) + (in1.sel)$$

More generally, we can have a larger multiplexer with 2^n data inputs, and a single data output. The number of selection control lines would need to be $\lceil log_2(2^n) \rceil$, that is n.

Multiplexors are easily represented combinationally in Verilog by using if expressions. For larger multiplexors, case statements are more convenient, but care must be taken to synthesize combinational logic.

1.4 Exercises in CombLogic in Verilog Syntax

1. Three dice are rolled simultaneously. For each die, let a binary signal represent whether the outcome is even or odd. Design a circuit whose output represents whether the sum of the outcomes of the three dice is even or odd.

   Solution

```
module evenodd(a, b, c, out);
    input a, b, c;
    output out;

    assign out = a ^ b ^ c;
endmodule
```

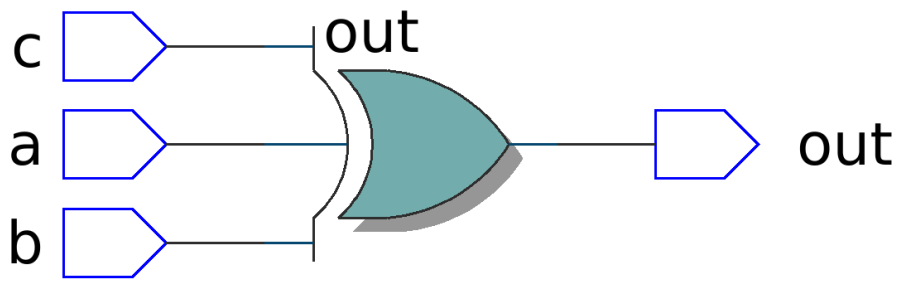


Figure 1.1: evenodd

2. **Draw a circuit to determine whether a 3 bit binary number is an exact power of 2.**

   Solution

```
module power2(num, out);
    input [2:0] num;
    output out;

    wire o1, o2, o3;

    assign o1 =  ~(num[1] | num[0]);
    assign o2 =  ~(num[2] | num[0]);
```

```
    assign o3 =  ~(num[1] | num[2]);

    assign out = o1 | o2 | o3;
endmodule
```

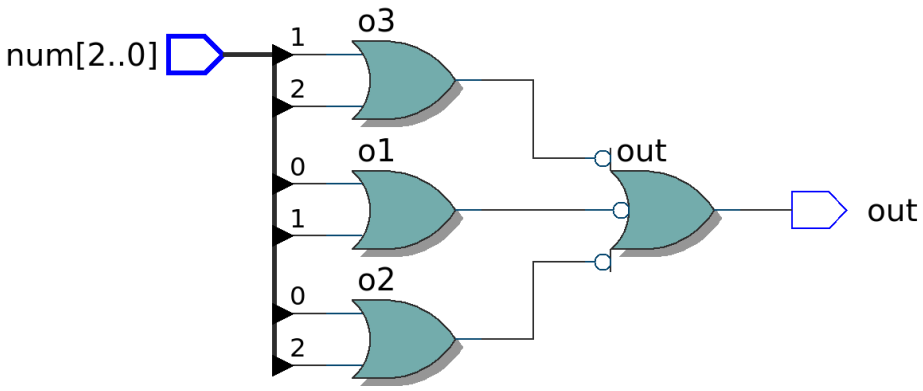


Figure 1.2: power2

3. **A 7 segment LED display is shown in the figure. You are to design a system which takes a 3 bit binary number as input, and display 'H' (for high) if the number is < 4 and 'L' (for low) otherwise. Write boolean equations or draw the circuit to compute the on/off state of each of the 7 segments.**

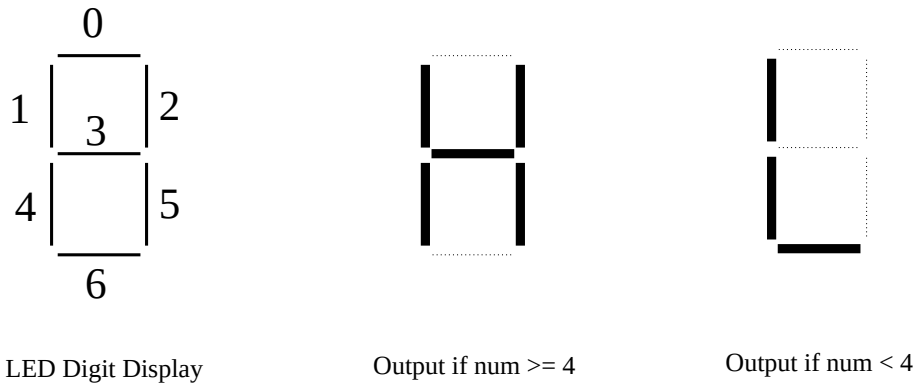


Figure 1.3: Problem 10

Solution: As can be seen, segment S0 is OFF in both cases, while S1 and S4 are ON in both cases. Thus, S0 = 0, S1 = 1, S4 = 1. Furthermore, number >= 4 if and only if the highest bit is 1.

```
module ledHL(number, segments);
    input[2:0] number;
    output[6:0] segments;

    assign segments[0]=0;
    assign segments[1]=1;
    assign segments[4]=1;

    assign segments[2]=number[2];
    assign segments[3]=number[2];
    assign segments[5]=number[2];

    assign segments[6]=~number[2];
endmodule
```

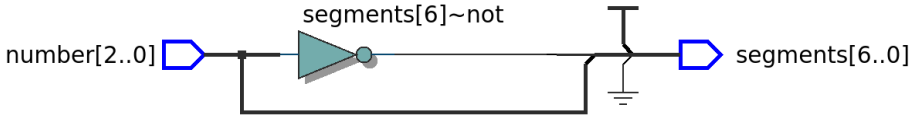


Figure 1.4: ledHL

4. **Design a circuit to compare two 2-bit numbers. The output should be a 2-bit value - 10/01 if A is less/greater than B and 00 if both are equal.**

   Solution

```
module  comparator2(a, b, c);
    input[1:0] a;
    input[1:0] b;
    output[1:0] c;

    wire  a1, b1, agt;

    assign a1 = (a[1] & ~b[1]);
    assign b1 = (b[1] & ~a[1]);


    assign c[1] =   a1 | (~b1 & a[0] & ~b[0]);
    assign c[0] =   b1 | (~a1 & ~a[0] & b[0]);
endmodule
```

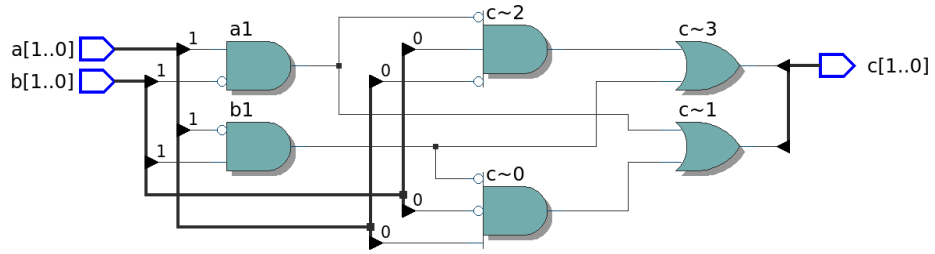


Figure 1.5: comparator2

5. **Design a structural circuit to compare two 4-bit numbers, using the 2-bit comparator.**

   Solution

```
module  comparator4(a, b, c);
    input[3:0] a;
    input[3:0] b;
    output[1:0] c;

    wire[1:0]  first, second;

    comparator2  c1(a[3:2], b[3:2], first);
    comparator2  cr(a[1:0], b[1:0], second);


    assign c[1] =   first[1] | (~first[0] & second[1]);
    assign c[0] =   first[0] | (~first[1] & second[0]);
endmodule
```

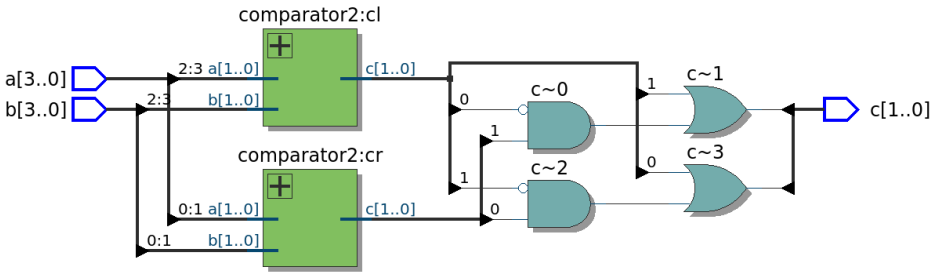


Figure 1.6: comparator4

6. **A room in a smart home has a cooler and a fan, each having its own switch. The fan is controlled by an additional logic.**

If the cooler is ON and the temperature is below 30 degrees, the fan will remain off. Design a circuit to control the state of the fan.

Solution

```
module fanState(coolerSwitch, fanSwitch, tempLow, fanOn);
    input coolerSwitch, fanSwitch, tempLow;
    output fanOn;

    wire coolerEffect;

    assign coolerEffect=coolerSwitch & tempLow;
    assign fanOn=fanSwitch & ~coolerEffect;
endmodule
```

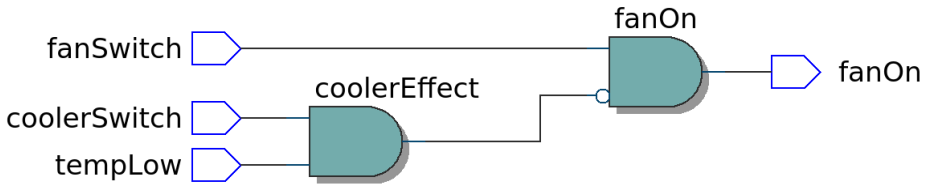


Figure 1.7: fanState

7. **In a mobile phone, a two bit signal represents whether the profile is "General" (00), "Outdoors" (01), "Silent" (10) or "Meeting" (11). Ringtone is ON in the "General" and "Outdoors" profiles and OFF in other profiles. Vibrate is ON in all profiles except in the "General" profile. The mobile has a "Power Save" mode, in which, vibrate is turned OFF when ringtone is ON and battery is LOW, irrespective of the profile settings. Draw a circuit to determine whether the mobile will vibrate on receiving a call.**

   Solution

```
module vibration(profile, powerSave, batteryLow,  out);
    input[1:0] profile;
    input batteryLow;
 input powerSave;
    output out;
```

```
    wire ringtoneOn, vibrateOn,  forceVOff;

    assign ringtoneOn=~profile[1];
    assign vibrateOn=(~profile[1] & ~profile[1]);
    assign forceVOff=ringtoneOn & powerSave & batteryLow;

    assign out=vibrateOn  & ~forceVOff;
endmodule
```

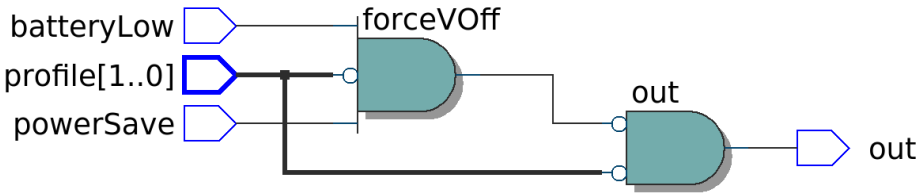


Figure 1.8: vibration

8. **A teacher has to form a team for an inter-school quiz competition. Alice is good at Maths, Economics and History. Bob is good at Physics and Maths. Charles is good at English and Physics. David is good at Economics and Physics. Design a logical circuit, whose output represents whether a subset of these students will be good at all of the subjects – Maths, Economics, History, Physics and English. (Use binary variables to represent whether a student is part of the team or not).**

   Solution

```
module quizTeam(alice, bob, charles, david,  covers);
    input alice, bob, charles, david;
    output covers;

    wire english, maths, economics, history, physics;

    assign english=charles;
    assign maths=alice | bob;
    assign economics=alice | david;
    assign history=alice;
```

```
    assign physics=bob | charles | david;

    assign covers = english&maths&economics&history&physics;
endmodule
```

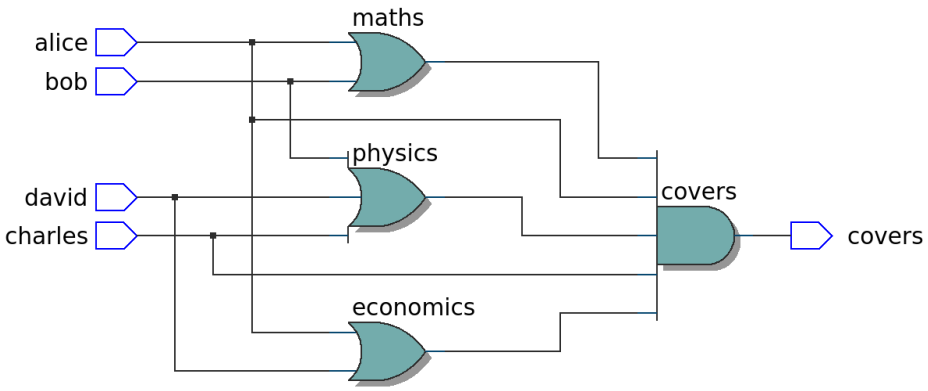


Figure 1.9: quizTeam

9. **Ram likes sandwich, pastry and ice-cream. Mohan likes sweetcorn and ice-cream. Anjali likes sandwich, pastry and sweetcorn. They have enough money to buy two items. Write a module which can determine, whether a choice of food items would contain at-least one food of liking for each child. (Assign 0-1 values to each food item).**

   Solution

```
module  choiceOkay
        (sandwich, icecream, sweetcorn, pastry, okay);
    input sandwich, icecream, sweetcorn, pastry;
    output okay;

    wire ram, mohan, anjali;

    assign ram=sandwich | pastry | icecream;
    assign mohan=sweetcorn | icecream;
    assign anjali=sandwich | pastry | sweetcorn;

    assign okay=ram & mohan & anjali;
endmodule
```

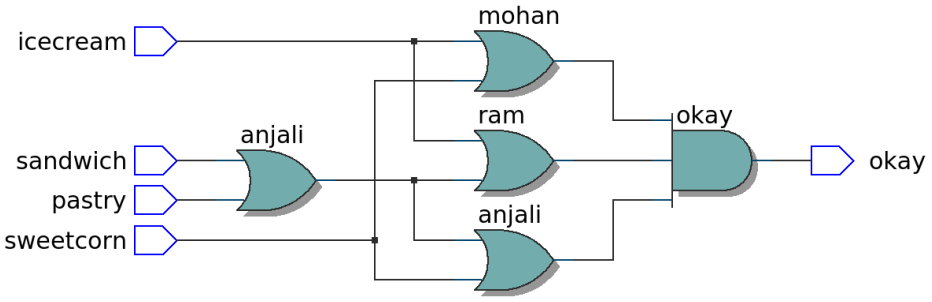


Figure 1.10: choiceOkay

10. Let the board position in an X & 0 (tic-tac-toe) game be denoted by (v_{11}, v_{12}, v_{13}, v_{21}, v_{22}, v_{23}, v_{31}, v_{32}, v_{33}) where $v_{ij}s$ are 2-bit values. Let v_{ij}=11 (00) denote presence of X (0) and any other value denote an empty block. Design a circuit to output whether the player whoes turn it is can win the game in the current move.

Solution

```
module blockvalue(value, turn, isPresent, isBlank);
    input[1:0] value, turn;
    output isPresent, isBlank;

    wire isX, isZero;

    assign isX=(value[1] & value[0]);
    assign isZero=~(value[1] | value[0]);

    assign isPresent= (turn & isX) | (~turn & isZero);
    assign isBlank=~(isX | isZero);
endmodule
```

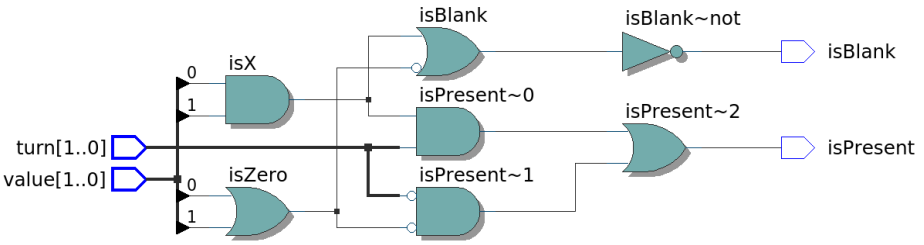


Figure 1.11: blockvalue

```
module check3(v1, v2, v3, turn, out);
    input v1, v2, v3, turn;
    output out;

    wire isP1, isP2, isP3;
    wire isB1, isB2, isB3;
    wire o1, o2, o3;

    blockvalue  bv1(v1, turn, isP1, isB1);
    blockvalue  bv2(v2, turn, isP2, isB2);
    blockvalue  bv3(v3, turn, isP3, isB3);

    assign o1 = (isB1 & isP2 & isP3);
    assign o2 = (isB2 & isP1 & isP3);
    assign o3 = (isB3 & isP1 & isP2);

    assign out = o1 | o2 | o3;
endmodule
```

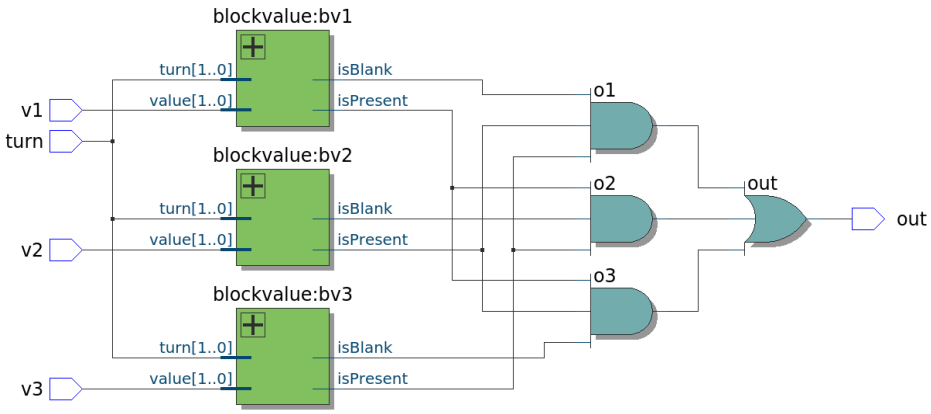


Figure 1.12: check3

```
module canWin(v11, v12, v13, v21, v22, v23,
        v31, v32, v33, turn,  out);
    input[1:0] v11,v12,v13,v21,v22,v23,v31,v32,v33;
    input turn;
    output out;

    wire  r1, r2, r3;
```

```
    wire  c1, c2, c3;
    wire  d1, d2;

    check3 cr1(v11, v12, v13, turn, r1);
    check3 cr2(v21, v22, v23, turn, r2);
    check3 cr3(v31, v32, v33, turn, r3);

    check3 cc1(v11, v21, v31, turn, c1);
    check3 cc2(v12, v22, v32, turn, c2);
    check3 cc3(v13, v23, v33, turn, c3);

    check3  cd1(v11, v22, v33, turn, d1);
    check3  cd2(v13, v22, v31, turn, d2);

    assign out = r1 | r2 | r3 | c1 | c2 | c3 | d1 | d2;
endmodule
```

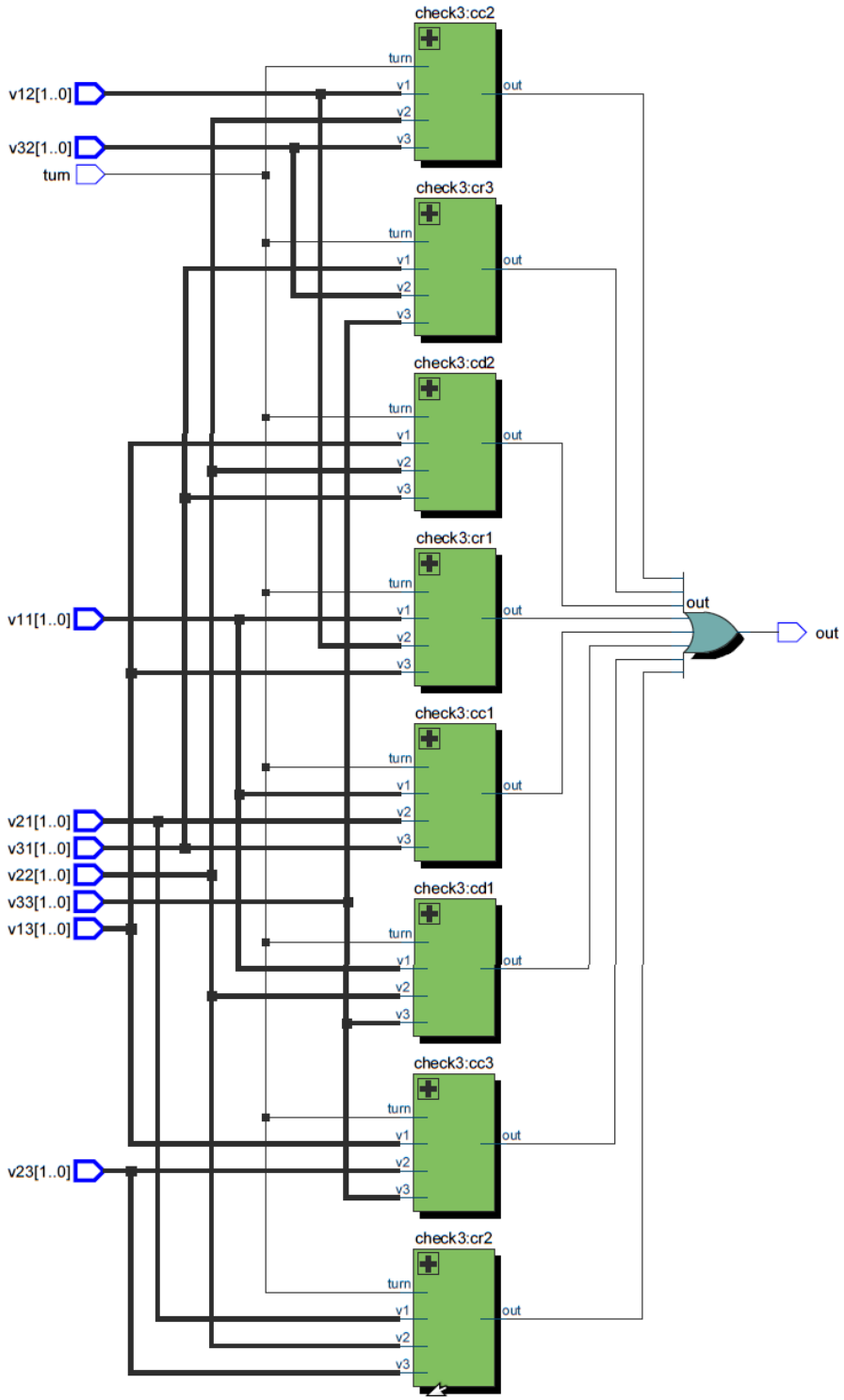


Figure 1.13: canWin

11. Extend the above circuit, to output the winning move (i, j coordinates) if a winning move exists, and i,j=0 if no winning move exists.

Chapter 2

Hardware Description Language

Knowledge of hardware description languages (HDLs) is a must for any digital designer today, as these days most digital design is done using a hardware description language. An HDL serves two main purposes, facilitates simulation and (automatic) synthesis. An abstract description of the hardware can be specified in using an HDL, so as to perform simulation and debugging of the design. Furthermore the logic synthesis tools enable automatic passage from HDL based specification of a digital logic design into the hardware implementation.

Verilog is one of the most popular hardware description languages. More popular in industry and based on C. Although many experts may find it lacking in discipline and features, the main reason for its adoption for this booklet is due to easier learning curve.

Architecture of digital design can be specified in both so-called behavioral and structural styles. In behavioral style specification one describes how the digital logic circuit functionally operates. Whereas in structural style of specification the detailed organization is given, quite typically in a hierarchical manner to conquer the complexity of the structure of architecture. The hierarchical organization of structure of digital logic hardware is constituted from basic primitives, such as gates and cmos switches, to arbitrarily complex user-defined digital logic components.

Mostly the structural style is used to describe the so-called datapath, which constitutes from a suitable interconnection of well designed and characterized logic macros such as multiplexors, decoders, encoders, adders, arithmatic units. memories, register files etc Such logic macros themselves datapath components are often modeled using behavioural facilities of Verilog HDL, however with some critical portions hand-crafted in structural manner.

The algorithms for logic synthesis are at the heart of CAD tools which are employed to generate the complex controller logic that is best specified in a behavioral fashion. The vast collection of libraries of powerful and generic logic macros mentioned above, are available with the CAD tools to help the designer get jumpstart as well as ensure use of efficient components, and focus more on the overall system architecture design.

For synthesizability and furthermore efficient use of hardware resources, the specification should be written carefully, following certain stirct rules to guide the synthesis tool towards good understanding of the specifications and efficient use of standard logic macros. At the simplest level, to ensure synthesizability, one must specify combinational and sequential logic in an unambiguous manner, so that the synthesis tool, understands clealy which kind of flipflops or latches are to be used, and what is the combinational logic for computing the updates of the memory elements and the outputs.

2.0.1 Datatypes and Operators in Verilog

A signal in a digital logic hardware is for the purpose of carrying information (data or control) from one place to another. We will use this term "signal" rather loosely in this document about Verilog. In particular, it should not be confused with "signal" as used in VHDL. Similarly the notion of "variables" in Verilog too should not be confused with that of those in VHDL on in general procedural programming language, or the general notion of "variables". We will try to take care to refer to "variables" in Verilog as "verilog-variables".

A "signal" is modeled in a Verilog based design using either a net or a verilog-variable. These are further classified as you will soon discover. You might have guessed that the term 'net' is borrowed from electrical jargon, because we would be using signals to interconnect different nodes in a circuit.

Another Caution : Unfortunately more often than not, a verilog-variable might not represent a "signal" in proper sense, as something that represents a signal carrying interconnect in a digital logic circuit. But in a broad sense, let us continue to use these terminologies.

A (verilog-) **net** is continuously driven (even though sometimes tri-stated, i.e. driven by Hi-Z). A (verilog-) **variable** is not necessarily continuously driven, and thus is crucial for behaviourally-modeling a signal that is backed by memory (e.g. a level-sensitive latch or an edge-triggered flip-flop). It should however be kept in mind that a verilog-variable may often appear in a code in which it is continuously driven (i.e. driven under all possible circumstances). Such use is effective for modeling output of a combinational logic.

One declares a net or a verilog-variable using the following syntax:

```
type [range] signal_name{, signal_name};
```

As is usual in the conventions of specifying syntax of programming languages, the square brackets mean an optional field. The curly braces mean repetition. The single-bit signals (that is those without the range specifiers) are called 'scalar signals'. Multi-bit signals (a.k.a. bit-vectors or "bus"ses) are described using their range specifiers.

In this booklet, we will restrict ourselves mainly to the following two primary types of signals in Verilog : "wire" and "reg".

1. A **wire**-type signal must be driven by purely combinational logic (that is, without memory). This is a crude characterization of **wire**-type signals in Verilog. As we go along, we would develop clearer understanding of which signals should be declared as of type **wire**.

2. A **reg**-type signal/object need not be continuously driven. So it might need to hold on to a value for a while. This value can change later when the reg object is driven again. However, reg need not necessarily correspond to an actual hardware memory element in an implementation, although it often will. In other words, a **reg**-type signal can be backed by memory (but need not always be the case , and in such cases it is backed by combinational logic). Reader is alerted to note that some particular "reg" data object may even be used repeatedly to refer to different expressions during the synthesis or simulation of a Verilog-based design.

A *bus*, named say "dBus", of 32 **reg** or **wire** signals, each carrying single bit, is declared as a **bit-vector** in the following manner. In Verilog parlance it is called a **bit-vector** quite obviously to indicate its nature as a vector of bits.

```
.......
reg [31:0] dBus;
.......

.......
wire [31:0] dBus;
.......
```

The index-range specifier, ("[31:0]" in the above example) is to be read off as "31 downto 0", and thereby designates the index of 0 as the least

significant bit of the bus of signals. Required contiguous subfield of such a **reg** or **wire**-bus, can be referred to using the range-specifier [**starting index : ending index**].

Subtly different is the notion of **array** of signal-objects (each signal object could be single-bit or multi-bit bus). Such an array of signal objects is used for a structure like an array of addressible memory locations in a ROM or RAM etc. Thus, the declaration

```
reg [31:0] mem_arr [0:15];
```

specifies a an array-of-signal object, where each signal in this array is a 32-bit wide **reg**-signal, with 0 index as the least significant index. The above illustrates that the index-range specifier for the array itself, "[0:15]" in the above example, indicates that the first signal of this array of fifteen signals, each 32-bit wide, is at the index 0, and the last one is at the index 15.

When accessing an array, we can refer to a single element, as in C, using the notation

```
.......................
... mem_arr [ index ] ..
.......................
```

A (single-bit) reg/integer or wire object in Verilog can take the following values

- 1'b0 or 1'b1, representing logical false or true
- 1'bx, representing "unknown" value, which is clearly useful in simulation to indicate that this signal has not been initialized yet. This specifically relates to "reg/integer"-type signal objects.
- 1'bz, representing the high-impedance or floating state which occur quite naturally due to electronics of digital logic implementation technologies such as Open Collector TTL. Quite simply, if a signal is not being driven, ie. disconnected from power-supply and also from the ground, we say that the signal is in High-impedence condition, ie. at value 1'bz.

Verilog facilitates specification of constant values in various formats, including decimal, binary, octal, or hexadecimal representations. The exactl how number of bit-level signals which will carry such constant field is also specified by prefixing the value with the size information in decimal format. For example:

- 4'b0100 specifies a 4-bit binary constant with the value 4, as does 4'd4.
- -8'h4 specifies an 8-bit constant with the value -4 (in two's complement representation)

Similar to constants, Verilog facilitates use of 'parameter's. A parameter is associated with a constant.

```
parameter bus_width = 32;
parameter stReset = 2'b00, stGot1 = 2'b01,
      stGot10 = 2'b10, stGot101 = 2'b11;
```

Use of the parameter identifiers (instead of the constants they are associated with) help improve readability and is an important coding practice in Verilog (as what holds for similar features in other HDLs).

Concatenation is a very useful operator in specification of logic design. One specifies a concatenation of signals, by listing them in comma-separated manner within { }. A special useful case of concatenation is replication. Replication is specified using notation **{k { sig_name }}** which indicates that the signal object **sig_name** is replicated **k** times. For example:

- **{16{2'b01}}** creates a 32-bit value with the pattern 0101 . . . 01.
- **{ 16{B[15]}, B[15:0] }** creates a value whose upper 16 bits are 16 copies of the MSB **B[15]** of **B[15:0]**.

Verilog HDL supports standard unary and binary operators for arithmetic, logical operators, comparison, shift, ternary operations. The arithmetic operators include the following $(+, -, *./)$. The operators (&, |, $\sim$) perform logical operations. For comparison operations we use $(==, != , >, <, <=, >=)$. The shift operators $(<<, >>)$ are used for shifting contents of bit-vectors. The ternary conditional operator '?', can elegantly express mux-based combinational logic. The format of ternary operator is as follows: **condition ? expr1 :expr2** and has usual meaning as in C language. The unary logic operators include the **reduction** operators (&, |, ^). Reduction operators perform the specified operation associatively on the bits of the bit-vector that is operand of the reduction operator. For instance,

```
^ A
```

returns via associatively XORing of the bits of signal 'A'. This yields 1 if and only if the number of 1-valued bits of the signal A is odd. The following operators too have their meaning as reduction operators, namely,

```
|, ^, ~&, ~|
```

reduction operators OR, XOR, NAND, and NOR respectively.

2.0.2 Structure of a Verilog Program

Modules are the basic units in Verilog models. A module may represent an arbitrarily complex (including simplest) digital logic system. Modules are thus blueprints, using which reuired number of instances of desired digital logic subsystems can be generated and used to compose together a larger system. They contain structural / functional (behavioural) descriptions of the architecture and have input, output ports in order to interfaces with other hardware objects. Hierarchical designs are built using modules which contain instantiations of other modules. A Verilog program is constituted from a set of modules.

```
module <module_id> [ ( <port_id> {, <port_id> } )];
  [declaration of parameters]
  [redeclaration of input ports]
  [redeclaration of output ports]

  [declaration of 'wire'-type signals]
  [declaration of 'reg/integer' type signals]
  [declaration of functions]
  [concurrent statements of the kind 'continuous assignment']
  [concurrent statements of using 'always ...' constructs]
  [concurrent statements instantiating primtive gates]
  [concurrent statements instantiating non-primitive modules]
  [initial blocks]
endmodule
```

As seen in the syntax above, a module could also declare additional signal objects.

The architecture of the logic circuit modeled in a module in Verilog HDL is described is captured essentially using the following:

- Continuous assignments, which use **assign LHS = RHS ;** format. The RHS describes combinational logic which drives the signals specified on the LHS. Thus **continuous assignments** are really intended for describing combinational logic using logic expressions on RHS.

- **always** constructs, which can define either sequential or combinational logic.

- Instances of other modules or of primitive gates, which are used to implement the module being defined.

The above two kinds of concurrent statements create digital logic subcircuits that interact with each other using wire/reg/integer signals.

These are called concurrent statements because the ordering among them does not matter. They can be jumbled up without any change in meaning.

Other than these, you would have also noticed, **initial** blocks and **function** declarations in the body of a Verilog module. To mention briefly, a **initial** constructs is used mainly for simulation to specify the events on signals from time 0. For the use with such **initial** blocks, we need to, use signals of the type **'reg/integer'** which can be assigned to within an **initial** block at different simulation time steps. **initial** block is also useful for initializing values in memory-array (RAM/ROM) etc.

Verilog, naturally provides constructs for defining **function** which models a reusable piece of combinational logic block, via use of blocking-assignments and sequential constructs (much like regular assignments and procedural constructs in a procedural programming language such as C). The calls to these functions can be used inside expressions on the right-hand-sides (RHS) or in conditions, in order to represent the logic of that combinational function.

We will mainly focus on the three kinds of constructs that are used to model logic, namely, structural instantiations (structural style), continuous assignment (dataflow style) , and **always** construct (behavioral style) containing block of so-called sequential or procedural statements (because such a bunch of statements need to be executed / interpreted in the order they are written).

Continuous assignment statements which have the form

```
assign LHS = RHS;
```

where LHS is a bunch of signals of the type **wire** and RHS is a combinational expression constituted from logical and arithmetic operators etc. Clearly such statement models combinational logic circuit corresponding to the expression on RHS, whose outputs are driving the **wire** signals on the LHS.

Always blocks can be used either for combinational logic or sequential logic. And it is highly recommended to use this construct of Verilog in an unambiguous way so that not only the designer herself, but also the synthesis tool (or simulator) gets a clear idea about whether the logic being modeled is combinational or sequential.

Other than the above two types of specifications of constituent logic blocks, namely, using continuous assignments and always-construct, module can contain instantiations of other modules, including those of primitive gates. An instantiation of a module specifies the name of the module whose instance is to be created, and the port connections, which describes how the

instantiated (child) module is connected with the rest of the logic blocks of the (parent) module.

Internal Signals

In addition to the input and output ports, a designer typically uses several more signals (that are internal to the module), for the sake of clarity of the design and to manage the complexity by interconnecting subcircuits that are defined by different concurrent statements, by the use of these internal signals.

Technically, it is not necessary to declare single-bit wires. However, it is necessary to declare multi-bit busses. It is also good practice to declare all signals. Some Verilog simulation and synthesis tools give errors that are difficult to decipher when a **wire**-tyoe object is not declared.

2.0.3 A Little More about Verilog Operators etc.

We will briefly elaborate on a few of the operators available in Verilog, especially the ones which a newcomer to HDL might need some introduction to.

Bitwise Operators

Bitwise operators act on bit-vectors. For example, the following module descrbes logic to compute 1's complement of a 4-bit number.

```
module ones_complement (in, out);
  input [3:0] in;
  output [3:0] out;
  assign out = ~in;
endmodule
```

Note that for single-bit signals, these bitwise operators are merely the usual operators on single bit logic values.

Other basic logic operators too extend to bitwise operators.

```
module bitwise_demo (a, b, a_and_b, a_or_b, a_xor_b,
      x_nand_b, a_nor_b);
  input [3:0] a, b;
  output [3:0] a_and_b, a_or_b, a_xor_b,
      x_nand_b, a_nor_b;
  assign a_and_b = a & b; // AND
```

```
  assign a_or_b = a | b; // OR
  assign a_xor_b = a ^ b; // XOR
  assign a_nand_b = ~(a & b); // NAND
  assign a_nor_b = ~(a | b); // NOR
endmodule
```

Reduction Operators

The following is module checks whether the input bit-vector is a non-zero or not.

```
module is_nonzero(A, A_is_nonzero);
  input [3:0] A;
  output A_is_nonzero;
  assign A_is_nonzero = !A; // reduction-OR
endmodule
```

(Ternary) Conditional Operator

The (ternary) conditional operator "?" takes three operands, just as in C or Java. If the first operand evaluates to true value, the conditional operator returns the value of the second operand, otherwise the third operand is evaluated and its value is returned as that of this conditional operator based expression.

```
module mux4_ternary(d0, d1,d2, d3,  sel, f);
  input [3:0] d0, d1, d2, d3;
  input [1:0] sel;  output  f;
  assign f = sel[1] ? (sel[0] ? d3:d2) : (sel[0] ? d1:d0);
endmodule
```

Arithmetic Operators

Arithmetic operators (including arithmetic comparison) available in Verilog HDL nclude $+, -, *, <, >, <=, >=, ==, !=, <<, >>, /$. Such operators are describe important datapath components. Their realization require significant amount of hardware, especially when used on bit-vectors of large width. How many 2-input XOR gates would be needed (with an additional inverter, possibly) for realizing n-bit equality operator "==" and inequality operator "! =" ? Addition, subtraction require an adder. (Arithmetic) Comparison can be done with the help of adder, but can also be done more compactly using priority encoders. Adders, and consequently multipliers are

resource-intensive as far as number of simple gates required to realize them is concerned. A lot of research has been done and continues to be done in design of efficient adders, multipliers etc. Barrel shifters can be specified by left and right shift operators << and >>.

Tristates

It is possible to leave a signal floating rather than drive it to 0 or 1. This floating value is called **1'bz** in Verilog. For example, the following code models a bunch of 4 tri-state buffers produces a floating output when the enable is false.

```
module tristate(a, en, y);
  input [3:0] a;
  input en;
  output [3:0] y;
  assign y = en ? a : 4'bz;
endmodule
```

EXERCISE : Explain tristate concept. Describe a possible implementation of tristate buffer (single bit input, control and output), using TTL. Describe a simple CMOS transistor based circuit of a tristate buffer, too.
A digression : Floating inputs to gates can cause undefined outputs, which are displayed as **1'bx** in Verilog. At start-up, state nodes such as the internal node of flipflops are also usually initialized to the value **x**. Other than this a properly designed Verilog models should not have to use 1'bx as signal values.

Coming back to tristates, let us look at an interesting simple application of tristates. We could define a multiplexer using two tristates so that the output is always driven by exactly one tristate. This guarantees there are no floating nodes.

```
module mux2_tristate(d0, d1, s, y);
  input [3:0] d0, d1;
  input s;
  output [3:0] y;
  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

2.0.4 Instantiating Other Modules

Verilog permits primitive gates to be instantiated in a module and wired with other parts of the module. The same applies to instantiating a module within another. For regular structures, designer can use the repetition constructs provided by Verilog for creating multiple instances of user defined module, primitives etc.

Consider the following illustration of a module instantiation.

```
module full_adder_dataflow ( a, b, cin, sum, cout );
  input a, b, cin;
  output sum, cout;

  assign {cout, sum} = a + b + cin;

endmodule
```

A multi-bit binary adder can be constructed using a cascade of full adders, as captured in the following structural-style Verilog description of the architecture of 4-bit adder. We connect the output carry port 'cout' of each full-adder to the input port 'cin', of the next full adder.

```
module ripple_carry_adder_4_bit ( A, B, cin, sum, cout );
  input [3:0] A, B;
  input cin;
  output [3:0] sum;
  output cout;
  wire [4:1] c;

  full_adder_dataflow
      i0(.a(A[0]),.b(B[0]),.cin(cin), .sum(sum[0]),.cout(c[1]));
  full_adder_dataflow
     i1(.a(A[1]),.b(B[1]),.cin(c[1]),.sum(sum[1]),.cout (c[2]));
  full_adder_dataflow
     i2(.a(A[2]),.b(B[3]),.cin(c[2]),.sum(sum[2]),.cout (c[3]));
  full_adder_dataflow
      i3(.a(A[3]),.b(B[3]),.cin(c[3]),.sum(sum[3]),.cout (c[4]));

  assign cout = c[4];

endmodule
```

In the above Verilog model of a 4-bit ripple carry adder, employing the structural style, to specifically make use of a cascade of 4 full-adders with carry rippling through the cascade. You may recall that structural style is used when the specific structure is to be ensured during the synthesis for the sake of efficiency or some other important considerations.

Let us study this illustration of instantiation of components (here instantiations of full_adder components). The syntax of instantiation has the name of the module which is being instantiated, in this case "full_adder". We also provide an name to thse instances, e.g. "i0, i1, i2, i3" as in the above example. More importantly, the port connections must be specified. The above example uses the scheme of explicit port connections (using the dot notation), in which the signals within the parent module are connected to the ports of the instantiated module using the explicit pairing of ports and signals, as in the above example). Alternatively, port connections can be made by listing the signals to be connected to ports in the order corresponding to the order of ports themselves. For instance, the above example could have been rewritten equivalently as follows (employing a mix of explicit and ordered port connections).

```
  full_adder_dataflow
      i0(.a(A[0]),.b(B[0]),.cin(cin),.sum(sum[0]),.cout(c[1]));
  full_adder_dataflow
      i1(A[1] ,  B[1] ,  c[1] ,  sum[1] ,  c[2] );
  full_adder_dataflow
      i2(.b(B[2]),.a(A[3]),.cin(c[2]),.cout(c[3]),.sum(sum[2]));
  full_adder_dataflow
      i3(A[3] ,  B[3] ,  c[3] , sum[3] , c[4] );
```

You may also note that in the explicit way of port connections, we need not list the port-signal pairs in the order of the list of ports. Furthermore, this manner of explicit port connections, also permit unconnected ports to be omitted from the instantiation statement.

For instance, supposing we were not to bother about the final output carry, from the 4-bit ripple carry adder, then the Verilog model would look as below.

```
module ripple_carry_adder_4_bit_ignore_cout ( A, B, cin, sum );
  input [3:0] A, B;
  input cin;
  output [3:0] sum;
  wire [3:1] c;
```

```
  full_adder_dataflow
      i0 (.a(A[0]),.b(B[0]),.cin(cin),.sum(sum[0]),.cout (c[1]));
  full_adder_dataflow
      i1 ( A[1] ,  B[1] ,  c[1] ,  sum[1] ,  c[2] );
  full_adder_dataflow
      i2 (.b(B[2]),.a(A[3]),.cin(c[2]),.cout(c[3]),.sum(sum[2]));
  full_adder_dataflow
      i3 (.a(A[3]), .b(B[3]), .cin(c[3]), .sum(sum[3] ) );

endmodule
```

Carry Lookahead Adder : a digression

Let us digress a little to extend the above discussion about multi-bit adder to consider another architecture for multi-bit adder that is more efficient in terms of timing performance.

In a 4-bit ripple-carry adder, there are nine inputs, namely A[3:0], B[3:0] and 'cin'. One needs to worry about the delay incurred in propagation of signals through such cascade of combinational logic blocks. In this circuit, the output signal 'cout', depends combinationally on c[3], which in turn depends (combinationally) on c[2], and so on. Finally c[1] combinationally depends on 'cin'. Thus in the worst case, after the delays through these 4 full-adders, the output signal 'cout' will attain its stable value. The output of each full adder will not settle to its final value until the input carry is available from the previous stage.

Recall the following typically used architecture of a full adder.

```
module FullAdder_assign (a,b,cin,sum,cout );
  input a, b, cin; output sum, cout;

  wire p, g;  // p==propagate, g==generate
  assign g = a ^ b;  // XOR gate
  assign p = a & b;  // AND gate
  assign sum = p ^ cin; // XOR gate
  assign cout = g | (p & cin);
      // AND followed by  OR gate

endmodule
```

As you may note from the above Verilog model of a full adder, the effect of signal 'cin' propagates through 2 gate levels to influence the signal 'cout'.

Therefore in case of a multi-bit full-adder designed as in Verilog module RippleCarryAdder, the influence of an event on the input signal 'cin' would propagate through 2*n gate levels, (where n is the number of full-adder stages), to the output 'cout' in the worst case. How do we reduce this delay (which is proportional to the number of cascaded stages, that is the width of the binary numbers this RippleCarryAdder can add) ? Other than the obvious use of faster gates (which has its technological limit), one can reduce the propagation delay by use of cleverer combinational circuit for such multi-bit adder. Use of carry lookahead is a popular method, resulting is a combinational logic circuit called Carry Lookahead Adder (CLA for short !).

We have for i=0,1,2,3,

$$p[i] = A[i] \oplus B[i]$$

$$g[i] = A[i].B[i]$$

$$sum[i] = p[i] \oplus c[i]$$

$$c[i+1] = g[i] + p[i].c[i]$$

where $c[0] = cin$ and $cout = c[4]$

Let us perform substitutions to obtain the carry signals, including the carry signals generated at intermediate stages, as combinational expressions in terms of A[3:0], B[3:0] and cin.

$$c[1] = g[0] + p[0]cin$$

$$c[2] = g[1] + p[1]c[1] = g[1] + p[1]g[0] + p[1]p[0]cin$$

$$c[3] = g[2] + p[2]c[2] = g[2] + p[2]g[1] + p[2]p[1]g[0] + p[2]p[1]p[0]cin$$

$$c[4] = g[3] + p[3]g[2] + p[3]p[2]g[1] + p[3]p[2]p[1]g[0] + p[3]p[2]p[1]p[0]cin$$

Clearly, one can always use an AND-OR (two-level) logic network to implement all the above combinational logic functions. And the depth of this logic network, which is 2, is clearly independent of the number of bits in the pair of numbers being added. This small depth logic circuit means that the worst case delay is under better control.

Binary Subtraction using Full Adders

You are likely to be aware of 2's complement representation. Use of 2's complement facilitates a natural reuse of Full Adders for performing multi-bit subtraction, using minimal extension of ripple-carry-adder's hardware. Specifically, to perform subtraction A[3:0]-B[3:0], we perform the addition of A[3:0] and 1's complement of B[3:0], along with constant 1 as carry-in. The following Verilog module captures this scheme.

```
module RippleCarryBorrowAddSubtract_4bit_structural
      ( A,B, add_sub, sum_diff, cout_borrow );
  input [3:0] A,B; input add_sub ;
  output [3:0] sum_diff;
  output cout_borrow;

  wire cin;
  wire [4:0] c_b;  // carry or borrow
  wire [3:0] BB_bar; // b or b_bar

  assign BB_bar = B ^ {4{add_sub}};
  assign cin = add_sub;
  assign c_b[0] = cin;

  FullAdder_assign fa0(A[0],BB_bar[0],c_b[0],sum_diff[0],c_b[1]);
  FullAdder_assign fa1(A[1],BB_bar[1],c_b[1],sum_diff[1],c_b[2]);
  FullAdder_assign fa2(A[2],BB_bar[2],c_b[2],sum_diff[2],c_b[3]);
  FullAdder_assign fa3(A[3],BB_bar[3],c_b[3],sum_diff[3],c_b[4]);

  assign cout_borrow = c_b[4];

endmodule

module FullAdder_assign (a,b,cin,sum,cout );
  input a, b, cin; output sum, cout;

  ......
  ......

endmodule
```

Rules for 'wire' and 'reg' vis-a-vis I/O Ports on Instances

In view of the semantics of "wire" and "reg" signal objects, the following rules must be observed while making port connections in module instantiation.

- An input port must be declared (if required to be declared) as "wire" (i.e. may not be declared as a "reg").
- A signal that is connected to an output port of an instantiated module must be declared as a "wire" (means never as a "reg").

2.0.5 Representing Complex Combinational Logic in Verilog

Although **always ....** blocks can be used for describing either combinational logic or sequential logic, it is important practice to organize your design into combinational and sequential subcircuits in an umabiguous manner.

Within an "always ... " block, Verilog permits, what is termed as, **procedural or sequential assignments** . These type of statements need to be contrasted with "continuous assignments" statements, each of which by itself is a concurrent statement. The assignment statements inside an "always ... " block are called sequential (or procedural) assignment statements, because they are understood by simulator or synthesis tool in a sequential context. They are not concurrent of each other. The order among such statements is crucial for inferring the logic circuit that is modeled by these group of statements inside an "always .... " block.

There are two types of sequential assignments, namely, "blocking" and "non-blocking" sequential assignments. The "blocking" assignment uses the operator =. And its meaning is as in tranditional procedural programming languages such as C. First, expression on the right-hand side of such a "blocking" assignment is evaluated, and the resulting value is assigned to the signal on the left-hand side. The next statement does not execute until the current "blocking" assignment is completed.

On the other hand, the non-blocking sequential assignment uses the operator <=. In contrast to "blocking" assignment, in non-blocking assignment, the RHSs of the non-blocking assignment are evaluated (immediately of course). However the value obtained by the evaluation is not "immediately" assigned to the signal on LHS, rather the LHS signal assumes the new value at a future (infinitesimally later in future or after specific time in future). "Non-blocking"-ness refers to the fact that the evaluation of the expressions inside next statement is **not** "blocked" by this NBA (non-blocking assignment).

The above-alluded "infinitesimally later in future or ... in future" refers to the electrical fact that the signals have positive propagation delays. Even in ideal situation, we will regard these delays as infinitesimally small, but still greater than 0.

We will later illustrate the subtle difference between blocking and non-blocking assignment.

Let us now see the features of Verilog coding within an "always .... " block. First we shall consider a simple and intuitively clear use of the facility if "if ... else ... " conditional that can be only be used inside an "always ... " block in a Verilog code.

The following code can be easily interpreted to be describing the behaviour of a 2-input, 1-output multiplexer with 1-bit control signal. The data input signals are "in0" and "in1", and the control input is the signal "sel". The output of the multiplexer is named "out" in the following code.

```
module mux_2_1 ( mux_in0, mux_in1, select, mux_out ) ;
  input mux_in0, mux_in1, select;
  output reg mux_out;

  always @( * ) begin
    if ( select ) mux_out <= mux_in1;
    else mux_out <= mux_in0;
  end

endmodule
```

Indeed, the reader should convince herself / himself that the above code is equivalent to the logic circuit captured in the following non-behavioural description.

```
module mux_2_1 ( mux_in0, mux_in1, select, mux_out );
  input  mux_in0, mux_in1, select;
  output mux_out;

  assign mux_out=(~select & mux_in0) | (select & mux_in1);
endmodule
```

To test a Verilog module, we need to instantiate the module to be tested, inside a Verilog module (which has no ports), which is usually called "test-bench" module. Certain internal signals of this testbench module are connected to input ports of the module instance under test. And these signals

are driven with certain values at appropriately chosen time instants so as to "stimulate" the module instance under test to react to these stimuli and produce some values at its output ports. The events at such internal signals connected to the output ports of the instance (under test), can be monitored, displayed etc. to facilitate testing and debugging of the module under test.

Consider the following testbench module to test our multiplexer defined above.

```
module mux_2_1_testbench;
  reg in0, in1, sel;
  wire out;
  mux_2_1  mux_2_1_instance0
      ( .mux_in0(in0),  .mux_in1( in1 ),
        .select(sel), .mux_out( out ));
  initial begin
    $monitor("time=%d in0=%b in1=%b select=%b out=%b",
                   $time, in0, in1, sel, out);
    #0   in0=0; in1=0; sel=0;
    #10  in0=0; in1=1; sel=0;
    #10  in0=1; in1=0; sel=0;
    #10  in0=1; in1=1; sel=0;
    #10  in0=0; in1=0; sel=1;
    #10  in0=0; in1=1; sel=1;
    #10  in0=1; in1=0; sel=1;
    #10  in0=1; in1=1; sel=1;

    #10 $finish;
  end

endmodule
```

In the above, you would note the use of "initial" construct within which we describe a sequence of events on certain signals (which constitute "stimuli" to the instance under test). The events are just value changes on these signals, and the delay annotations, using # operator are used to describe precise timing information about the occurrences of these events. The result of the simulation would be as below.

```
time=0 in0=0 in1=0 sel=0 out=0
time=10 in0=0 in1=1 sel=0 out=0
```

```
time=20 in0=1 in1=0 sel=0 out=1
time=30 in0=1 in1=1 sel=0 out=1
time=40 in0=0 in1=0 sel=1 out=0
time=50 in0=0 in1=1 sel=1 out=1
time=60 in0=1 in1=0 sel=1 out=0
time=70 in0=1 in1=1 sel=1 out=1
```

One more thing to note is that, within an initial block, one typically uses conventional sequential/procedural semantics of a programming language. Thus use of blocking assignments is typical, although not necessary, inside this Verilog-construct "initial".

Use of "if ... else .... " conditionals inside "always .... " blocks

One can make more general use of "if .... else .... " construct as in typical programming languages. However to realize combinational logic using such facility, one must ensure that the target signal is assigned to (in each execution of "always ...." block, irrespective of the execution path determined by the evaluation of conditionals based on the current values of the signals (which appear in the conditions of such conditional statement). This principle holds for other conditionals such as "case ... endcase" too.

```
module mux_4_1 ( in0, in1, in2, in3, sel0, sel1, out );
  input in0, in1, in2, in3;
  input sel0, sel1;
  output reg out;

  always @( * ) begin
    if ( sel1 ) begin
      if ( sel0 ) begin out <= in3; end
      else begin out <= in2; end
    end else begin
      if ( sel0 ) begin out <= in1; end
      else begin out <= in0; end
    end
  end // always
endmodule
```

Later we will see more general use of "if ... else ... " conditionals inside Verilog modeling of finite state machines.

Modeling Binary Decoder using Procedural Block

Next, in order to illustrate facilities that "always ..." construct offers to describe more complex combinational logic, we consider the following example of an "always .... " block to define a decoder. It takes advantage of the case-construct which may only be inside an always block.

```
module decoder_using_always(a, y);
  input [2:0] a;
  output [7:0] y;
  reg [7:0] y;
  // a 3:8 decoder
  always @(a)
    case (a)
      3'b000: y <= 8'b00000001;
      3'b001: y <= 8'b00000010;
      3'b010: y <= 8'b00000100;
      3'b011: y <= 8'b00001000;
      3'b100: y <= 8'b00010000;
      3'b101: y <= 8'b00100000;
      3'b110: y <= 8'b01000000;
      3'b111: y <= 8'b10000000;
        // Note that the above 8 alternatives
        // do NOT exhaust all cases
      default : y <= 8'bx;
        // This specification for the rest
        // of the case alternatives, we use
        // "default" case-alternative to cover
        // all the non-specified alternatives.
        // For instance, 3'b0x1, 3'b1z1  etc.
        // would now be treated in  "default" alternative.
    endcase
endmodule
```

Using the case statement is probably clearer than a description of the same decoder using boolean equations in an (continuous) **assign ...** statement:

```
module decoder_assign(a, y);
  input [2:0] a;
  output [7:0] y;
  wire [7:0] y;
```

```
  assign y[0] = ~a[2] & ~a[1] & ~a[0];
  assign y[1] = ~a[2] & ~a[1] & a[0];
  assign y[2] = ~a[2] & a[1] & ~a[0];
  assign y[3] = ~a[2] & a[1] & a[0];
  assign y[4] = a[2] & ~a[1] & ~a[0];
  assign y[5] = a[2] & ~a[1] & a[0];
  assign y[6] = a[2] & a[1] & ~a[0];
  assign y[7] = a[2] & a[1] & a[0];
endmodule
```

The following is a very compact Verilog description of the above decoder. It uses the semantics of blocking assignment in a very effective manner.

```
module decoder_compact_behavioural (a, y);
  input [2:0] a;
  output [7:0] y;

  reg [7:0] y;
  always @( * ) begin
    y[7:0] = 8'b0;  // blocking assignment
    y[ a[2:0] ] <= 1'b1; // non-blocking finalizing assignment
  end // always
endmodule
```

Modeling a Priority Encoder

Next consider a priority encoder which encodes, using a 2-bit output "y[1:0]", the highest among the indices of the asserted input signal bits "w3:0]". In case none of the input bits is asserted, there is no valid encoding of this situation, therefore an output signal "valid" would be deasserted to indicate such input that is invalid for priority encoding.

The following behavioural description of such a priority encoder makes use of a variation of **case .... endcase** , namely **casex .... endcase** construct. Note that in the usual **case .... endcase** construct, if a **case alternative** uses **'x'**, in a bit position, then it indeed specifies the alternative, in which that particular bit has Verilog value **1'bx**.

However to effectively describe a priority encoder logic, we would like to specify **case alternatives** using *don't care's* in certain bit positions. The **casex ... endcase** allows you to use **'x'** in a bit position in a "case alternative" to reflect that we '*don't care* about the value in that bit position. The following code describes a priority encoder in this fashion.

```
module priority_4_2_always(w, y, valid);
  input [3:0] w;
  output [1:0] y;
  output valid;
  reg [1:0] y;
  reg valid;
  always @( w )
    casex ( w )
      4'b1xxx : begin y <= 2'b11; valid <= 1; end
      4'b01xx : begin y <= 2'b10; valid <= 1; end
      4'b001x : begin y <= 2'b01; valid <= 1; end
      4'b0001 : begin y <= 2'b00; valid <= 1; end
      default : valid <= 0;
    endcase
endmodule
```

EXERCISE : Show the need of "casex" in the above.

The expressive power of **always ...** construct is mainly due to the powerful constructs such as **if ... else ...** , **case ... endcase** and their variants.

To convince yourself about the expressive power of **always ...** construct, try to express the logic architecture of such priority encoder using only **continuous assignments**.

```
module 4_2_priority_assign(w, y, valid);
  input [3:0] w;
  output [1:0] y;
  output valid;
  wire [1:0] y;
  wire valid;

  // To be completed by the reader
  // Express the logic architecture
  // using only "continuous assignments"
  ..............
  ..............

endmodule
```

Here is how one would model a priority encoder in a parameterized manner. This entails use of "for loop" (naturally within always block).

```
module prio_enc_generic (w, y, valid);
  // parameter n=16, logn=4 ;
  parameter n=4, logn=2 ;
  input [n-1:0] w;
  output [logn-1:0] y; output valid;
  reg [logn-1:0] y; reg valid;

  reg v_valid ; reg [logn-1 : 0] v_y ;
    // To be treated as proxies for signals valid and y.
    // In the following parameterized code for
    //     priority encoder, the reg-objects v_valid and v_y
    //     are repeatedly assigned to
    //     ( as is typical in procedural sequential programming )
    //     and therefore blocking assignment makes sense here too.
    // At the exit points of this always block, we deposit these proxies
    //     onto the corresponding signals named valid and y.
  integer i;
  always @( w ) begin
    v_valid = 1'b0 ; v_y = 0 ;
    for ( i = 0 ; i < n; i=i+1 ) begin
      if ( w[i] ) begin v_y = i ; v_valid = 1'b1 ; end
    end
    valid <= v_valid ; y <= v_y ;
  end

  /*
  always @( w )
    casex ( w )
      4'b1xxx : begin y_gold <= 2'b11; valid_gold <= 1; end
      4'b01xx : begin y_gold <= 2'b10; valid_gold <= 1; end
      4'b001x : begin y_gold <= 2'b01; valid_gold <= 1; end
      4'b0001 : begin y_gold <= 2'b00; valid_gold <= 1; end
      default : valid_gold <= 0;
    endcase
  */
endmodule
```

A priority encoder is most naturally coded using either nested "if else" constructs or repeated "if else" construct inside an always block

```
module prio_enc_nested_if_else_style (w, y, valid);
```

```
  input [4-1:0] w;
  output [2-1:0] y; output valid;
  reg [2-1:0] y; reg valid;
  always @( w ) begin
    valid <= 0 ;
    if ( w[3] ) begin valid <= 1 ; y <= 3 ; end
    else if ( w[2] ) begin valid <= 1 ; y <= 2 ; end
    else if ( w[1] ) begin valid <= 1 ; y <= 1 ; end
    else if ( w[0] ) begin valid <= 1 ; y <= 0 ; end
    else begin valid <= 0 ; end
  end
endmodule

module prio_enc_repeated_if_else_style (w, y, valid);
  input [4-1:0] w;
  output [2-1:0] y; output valid;
  reg [2-1:0] y; reg valid;
  always @( w ) begin
    valid <= 0 ;
    if ( w[0] ) begin valid <= 1 ; y <= 0 ; end
    if ( w[1] ) begin valid <= 1 ; y <= 1 ; end
    if ( w[2] ) begin valid <= 1 ; y <= 2 ; end
    if ( w[3] ) begin valid <= 1 ; y <= 3 ; end
  end
endmodule
```

A circuit in which priority encoding is implicit is a comparator.

A 4-bit comparator (not merely just an equality checker) takes a pair of 4-bit signals 'a[3:0]', 'b[3:0]' as inputs, and produces the comparison at the 3 single-bit outputs 'a_lt_b', 'a_eq_b', 'a_gt_b', indicating whether 'a[3:0]' is less than, or equal to, or greater than 'b[3:0]'. The gate level schematic f (behavioral style) or this is nicely structured, but still reasonably complex combinational logic, if one were to describe it using either continuous assignments or structural instantiations. Of course, with the features of Verilog such as "for" construct, one can describe the structure elegantly and compactly. However, the following illustration of behavioural description of a comparator gives the evidence of natural suitability of **always ...** construct to express complex combinational logic.

```
module comparator(a, b, a_lt_b, a_eq_b, a_gt_b);
  input [3:0] a,b;
```

```
  output a_lt_b, a_eq_b, a_gt_b;
  reg a_lt_b, a_eq_b, a_gt_b;
  always @(a,b) begin
    if (a[3] ^ b[3]) begin
      a_eq_b <= 1'b0; a_lt_b <= b[3]; a_gt_b <= a[3];
    end else if (a[2] ^ b[2]) begin
      a_eq_b <= 1'b0; a_lt_b <= b[2]; a_gt_b <= a[2];
    end else if (a[1] ^ b[1]) begin
      a_eq_b <= 1'b0; a_lt_b <= b[1]; a_gt_b <= a[1];
    end else if (a[0] ^ b[0]) begin
      a_eq_b <= 1'b0; a_lt_b <= b[0]; a_gt_b <= a[0];
    end else begin
      a_eq_b <= 1'b1; a_lt_b <= 1'b0; a_gt_b <= 1'b0;
    end
  end
endmodule
```

Caution about Use of "always" Construct

You would have noticed from the above examples of use of **always ...** constructs that the signals on the LHS of any (procedural assignment statement) inside an "always ... " block, is necessarily declared to be of 'reg' type. If, inadvertently, you fail to declare such a signal as of type 'reg', then the Verilog compiler / simulator would indicate an error to bring it to your attention. It is an easy rule to get used to. Recall, on the other hand, the signals appearing on the LHS of an "continuos assignment" statement may never be declared to be of type 'reg'. Basically, a 'reg'-type signal might need to have memory so that it can use its value from the past, as might be specified by the behavioural description inside an "always ... " block.

Thus, the 'reg' type signals may only be driven inside an "always ... ' block, and therefore may not appear on LHS of "assign ... " statements (that is "continuous assignments") and also may not be connected to output ports of an instantiated module.

A Verilog designer should always be alert about the following pitfall. The 'reg' type signals can use memory, and therefore, if a designer misses specifying new value of the signal under some condition, in an "always .... " block, then the resulting synthesized signal would have to be driven by a memory element, and thus the intention of describing combinational logic to drive that signal would have failed. Neverthless the expressive power due to availability of "if ... else ...", "case ... endcase ", "for ... " constructs inside an "always ... " block, is a significant benefit that a Verilog designer cannot

afford to ignore, and mix caution with effective use of "always ... " block, especially while designing complex combinational logic subcircuits.

```
module mux_2_1 ( in0, in1, sel, out ) ;
  input in0, in1, sel; output reg out;

  always @( * ) begin
    if ( sel ) out = in1;
    else out = in0;
  end

endmodule

module d_latch ( in0, in1, sel, out ) ;
  input in0, in1, sel; output reg out;

   // This code is ‘‘identical’’ to that of mux_1_1,
   // with the sole exception of the missing ‘‘else’’ clause
   // in the following.

   // Therefore instead of a mux, the following code would be
   // synthesized into a ‘‘d_latch’’ with ‘‘in1’’ as the d-input
   // and ‘‘sel’’ as control input ‘‘enable’’ to the d-latch

  always @( * ) begin
    if ( sel ) out = in1;
  end

endmodule
```

Expressive Power of "always .... " construct

Here we see a small example of a 4-bit ripple-carry-adder coded in a compact manner using the convenience of for loop and blocking assignments in Verilog.

```
module rca( a,b,sum,cout ) ;
  input [3:0] a ; input [3:0] b ;
  output [3:0] sum ; output cout ;
  reg [3:0] sum ; reg cout ;
  reg tmp_carry ; integer i ;
```

```
  always @( a,b ) begin
    tmp_carry = 1'b0 ;
    for ( i=0 ; i < 4 ; i=i+1 ) begin
      sum[i] <= a[i] ^ b[i] ^ tmp_carry ;
      tmp_carry = (( a[i] | b[i] ) & tmp_carry) | ( a[i] & b[i] ) ;
    end
    cout <= tmp_carry ;
  end
endmodule

module rca_tb ;
  reg [3:0] a ; reg [3:0] b ; wire [3:0] sum ; wire cout ;
  rca dut( a,b,sum,cout ) ;
  initial begin
    $dumpvars() ;
    $monitor( a," ", b, " ", sum, " " ,cout ) ;
    #100 $finish ;
  end
  initial begin
    a = 5 ; b = 10 ;
    #10 ;
    a = 7 ; b = 13 ;
    #10 ;
  end
endmodule
```

In the above example, a for loop is being used in sequential / procedural zone along with a blocking assignment to tmp_carry and a non-blocking assignment to sum[i]. After the for loop, the resulting tmp_carry is being assigned in non-blocking manner to reg-signal cout. Note that tmp_carry is suitably assigned to 0 before the for loop. This sequentially executed code models that the 4 bits of reg-signal sum are calculated using xor of corresponding bits of wire-signals a[3:0] and b[3:0] and the present value of tmp_carry , which at i^{th} iteration of the for loop, represents carry incoming to the i^{th} full-adder. Thus you see in this example, effective use of blocking assignments and a for loop construct to compactly express a relatively complicated combinational logic.

Chapter 3

Sequential Logic

3.1 Clocks

In sequential logic, we need clocks to synchronize updates of the memory elements. A clock is a periodic signal. In this booklet, we consider only **synchronous** sequential logic circuits, a subclass of sequential logic circuits, in which memory updates are triggered either only at rising edge of the clock or only at falling edge of the clock. This we refer to as synchronous (edge-triggered) sequential circuit discipline. Depending on whether the synchronizing events are the rising transitions on the clock signal or the falling edges, the edge-triggering is classified as posedge-triggered or negedge triggered synchronous discipline, respectively. Such edge-triggered methodology is simpler to explain and to design and analyze logic circuits with.

In a posedge-triggered clock discipline, a clock cycle, is the time interval between successive rising transitions on the clock signal. Similarly for negedge-triggering. We call the posedge as the active-edge of the clock for posedge-triggered clocking. Similarly negedge is the active edge for negedge-triggered clocking. Frequency of the clock is the inverse of the duration of a single cycle of the clock.

You need to reinforce the simple and important characteristic of synchronous edge-triggering is thar state elements (a.k.a. memory elements) only change on the active clock edge. It is important to note that certain sequential circuit primitives are designed with this in mind. We refer to these as (edge-triggered) flipflops. The triggering edge of the clock samples the data input which in turn influences the updates in the memory/state elements,

In a synchronous sequential system, one needs to obsever certain important timing constraints. Two of the most important are the “setup-time”

and "hold-time" constraints. Setup-time and hiold time constraints insists that that the signals that are data input to state elements (edge-triggered flipflops) must be stable over an interval around the active edge of the clock. Since these data-inputs to edge-triggered flipflops are driven by combinational circuit that is driven by the stable outputs of the synchronous sequential elements, and combinational circuits do not need feedback, it can be ensured that after a predictable time duration, the data-inputs to flipflops will eventually become valid.

The general structure of a synchronous, sequential logic design is given by the relationship among the state elements and the combinational logic blocks. The state elements, whose outputs change only after the clock edge, provide valid inputs to the combinational logic block. To ensure that the values written into the state elements on the active clock edge are valid, the clock must have a long enough period so that all the signals in the combinational logic block stabilize, then the clock edge samples those values for storage in the state elements. This constraint sets a lower bound on the length of the clock period, which must be long enough for all state element inputs to be valid.

We would be assuming (unless otherwise mentioned) that all state elements are updated on the same clock edge. Some state elements will be written on every clock edge, while others will be written only under certain conditions (such as a enable signal being asserted).

One other advantage of an edge-triggered methodology is that it is possible to have a state element that is used as both an input and output to the same combinational logic block. In practice, care must be taken to prevent races in such situations and to ensure that the clock period is long enough. This will come up later for discussion.

Now that we have discussed how clocking is used to update state elements, we can discuss how to construct the state elements.

3.2 Memory Elements : FlipFlops, Latches, and Registers

We discuss the basic principles behind memory elements. All memory elements store state: the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element. Thus all logic blocks containing a memory element contain state and are sequential.

The simplest type of memory elements are un-clocked; that is, they do

not have any clock input. Although we only use clocked memory elements in this text, an (un-clocked) latch is the simplest memory element, so let's consider the Set-Reset latch first. It can be built from a pair of NOR gates (OR gates with inverted outputs).

```
module SetReseLatch_nor ( Set, Reset, Q, Qb );
  input Set, Reset; output Q, Qb;
  // cross-coupled-nor-gates

  nor g1 ( Q, Reset, Qb );
    // output : 'Q', inputs : 'Reset', 'Qb'
  nor g2 ( Qb, Set, Q );
    // output : 'Qb', inputs : 'Set', 'Q'

endmodule
```

The outputs Q and Qb represent the value of the stored state and its complement.

When neither Set nor Reset are asserted, the cross-coupled NOR gates act as inverters and store the previous values of Q and Qb. For example, if the output, Q, is at logic level 1, then the **nor** gate (acting as just an inverter, due to Set being at logic 0) driving Qb, produces a logic 0 value. This logic 0 value at Q is input to the **nor** gate driving Q (just working as inverter, as Reset is deasserted LOW), This locks Q,Qb to be at logic 1 and logic 0 respectively, due to the nor-gates behaving as inverters for the assumed case. If S is asserted, then the output Q will be asserted and Qb will be deasserted, while if R is asserted, then the output Qb will be asserted and Q will be deasserted. When S and R are both deasserted, the last values of Q and Qb will continue to be stored in the cross-coupled structure. Asserting S and R simultaneously can lead to incorrect operation: depending on how S and R are deasserted, the latch may oscillate or become metastable.

This cross-coupled structure is the basis for more complex memory elements that allow us to store data signals. These elements contain additional gates used to store signal values and to cause the state to be updated only in conjunction with a clock.

3.2.1 FlipFlops and the Constituent Latches

Flipflops and latches are the simplest memory elements. Often flipflops are constructed from more basic latches. Once we have a flipflop, we have a synchronous memory element ready for effective use. Latches, by themselves,

are trickier to use in larger sequential logic circuits. In both flipflops and latches, the output is equal to the value of the stored state inside the element. Furthermore, unlike the S-R latch described above, all the latches and flipflops we will use from this point on are clocked, which means that they have a clock input and the change of state is triggered by that clock. The difference between a flipflop and a latch is the point at which the clock causes the state to actually change. In a clocked latch, the state is changed whenever the appropriate inputs change and the clock is asserted, whereas in a flipflop, the state is changed only on a clock edge. Since throughout this booklet, we use an edge-triggered timing methodology where state is only updated on clock edges, the use of flipflops suffices.

For computer applications, the function of both flipflops and latches is to store a signal. A D latch as well as a D flipflop stores the value of its data input signal in the internal memory. Although there are many other types of latches and flipflops, the ones of D type is the only basic building block that we will need. A D latch has two inputs and two outputs. The inputs are the data value to be stored (called D) and a signal named "enable" that indicates when the latch should read the value on the D input and store it. The outputs are simply the value of the internal state (Q) and its complement (Qb). When the "enable" signal is asserted, the latch is said to be *transparent*, and the value of the output (Q) becomes the value of the input D. When the "enable" is deasserted, the latch is said to be in *hold* mode, and the value of the output (Q) is whatever value was stored the last time the latch was transparent.

One can easily see how a D latch can be implemented with two additional gates added to the cross-coupled NOR gates. Since when the latch is transparent the value of Q changes as D changes, this structure is sometimes called a transparent latch.

```
module SetReseLatch_nor ( Set, Reset, Q, Qb );
  .......
  .......
endmodule

module Dlatch ( enable, d, q );
  input enable, d;
  output q;

  SetResetLatch_nor
      inst0(.Set( d & enable ), .Reset(~d & enable), .Q(q) );

endmodule
```

As mentioned earlier, we use flipflops as the basic building block, rather than latches. Flipflops are not transparent: their outputs change only on the clock edge. A flipflop can be built so that it triggers on either the rising (positive) or falling (negative) clock edge; for our designs we can use either type. A D flipflop can be constructed from a pair of D latches.

```
module MasterSlaveDFF ( clk, d, q );
  // falling edge triggered master-slave DFF

  input clk, d;
  output q;

  wire qM; // output of master d-latch

  Dlatch
      instanceM ( .d( d ), .enable( clk ), .q( qM ) );
        // master d-latch
  Dlatch
      instanceS ( .d( qM ), .enable( ~clk ), .q( q ) );
        // slave d-latch

endmodule
```

In a falling-edge D flipflop, the output is stored when the clock makes falling transition.

3.3 Modeling Sequential Logic with "always" Blocks

Although latches and flipflops can be described by primitive gates and **assign ...** statements, such descriptions are hard to generalize and describing more complex register structures cannot be done this way. This section uses always statements to describe latches and flipflops. We will show that the same coding styles used for these simple memory elements can be generalized to describe memories with complex control as well as functional register structures like counters and shift-registers.

An **always ....** block is re-evaluated only when signals in the header change. Depending on the form, always blocks may imply sequential or combinational circuits.

3.3.1 Latches

Always blocks can also be used to model transparent latches, also known as D latches. When the enable is high, the latch is transparent and the data input flows to the output. When the enable is low, the latch goes opaque and the output remains constant.

```
module my_latch(enable,d,q);
  input enable;
  input [3:0] d;
  output [3:0] q;
  reg [3:0] q;
  always @(enable or d) // May use @(*) instead
    if (enable) q <= d;
endmodule
```

In the above code the behaviour of 4 D-latch, indexed from 3 down to 0, is described by an **always ...** construct. The 4 outputs of these 4 latches are declared as reg because they are being driven inside the always procedural block. Latch clock and data inputs appear in the sensitivity list of the always block making this procedural statement sensitive to any change/event on them. When an event happens at either enable or any of d inputs, then the **always ...** block statement wakes up and it executes all its statements in the sequential order from begin to end. The if-statement enclosed in the always block puts d[3:0] into q[3:0] when enable is asserted HIGH. This behaviour is referred to as *transparency* which is how latches work. While enable is active, a latch structure is transparent and input changes affect its output.

3.3.2 Flipflops

While a latch is transparent a change on the D-input of a D flipflops does not directly pass on to its output. The code below describes a positive-edge trigger D-type flipflop. The sensitivity list of the procedural statement shown includes posedge of clk. This **always ...** statement only wakes up when clk makes a 0 to 1 transition. When it does wake up, the value of d is put into q. Obviously this behavior implements a rising-edge D flipflop.

```
module my_dflop(clk, d, q);
  input clk;
  input d;
  output q;
```

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

The body of the **always ....** statement is only evaluated on the rising (positive) edge of the clock. At this time, the output q is copied from the input d. The sequential assignment statement that uses <= is called a non-blocking assignment. think of it as usual (sequential) "assignment" inside an **always ...** construct for now; we return to the subtle points later. All the signals on the left hand side of assignments in **always ...** blocks must be declared as **reg**. This is a confusing point for new Verilog users. In this circuit, q is also the output. Declaring a signal as **reg** does not mean the signal is actually a register! All it means is it appears on the left side in an **always ...** block. There are examples of combinational signals that are declared reg but have no flipflops. At start-up, the q output is initialized to **x**. Generally, it is good practice to use flipflops with reset inputs so that on power-up you can put your system in a known state. The reset may be either asynchronous or synchronous. Asynchronous resets occur immediately. Synchronous resets only change the output on the triggering edge of the clock.

```
module flop_asynch_reset(clk, reset, d, q);
      // asynchronous reset
  input clk;
  input reset;
  input  d;
  output  q;
  reg  q;
  always @(posedge clk or posedge reset)
    if (reset) q <= 1'b0;
    else q <= d;
endmodule

module flop_synch_reset (clk, reset, d, q);
      // synchronous reset
  input clk, reset;
  input d;
  output q;
  reg q;
  always @(posedge clk)
    if (reset) q <= 1'b0;
```

```
    else q <= d;
endmodule
```

A parallel register is merely a collection of above such flipflops, triggered by the same event on the same clock input. Let us model such a 4-bit register, also with a synchronous reset.

```
module four_bit_register (clk, reset, d, q);
    // synchronous reset
  input clk, reset;
  input [3:0] d;
  output [3:0] q;
  reg [3:0] q;
  always @(posedge clk)
    if (reset) q <= 4'b0;
    else q <= d;
endmodule
```

Note that the asynchronously resettable flop evaluates the always block when either clk or reset rise so that it immediately responds to reset. The synchronously reset flop is not sensitized to reset in the @ list, so it waits for the next clock edge before clearing the output.

One could also consider flipflops with enables that only load the input when the enable is true. The following flipflop retains its old value if both "reset" and "en" are false.

Let us model a 4-bit parallel register consisting of such D flipflops with asynchronous reset and (synchronous) enable signal input.

```
module four_bit_register_with_enable_and_asynch_reset
    (clk, reset, en, d, q);
  input clk;
    // to be used for synchronizing on its posedges
  input reset;
    // to be used as  asynchronous reset
  input en;  //( synchronous ) enable
  input [3:0] d;  // the data inputs
  output [3:0] q; // the outputs
  reg [3:0] q;
  always @(posedge clk or posedge reset)
    if (reset) q <= 4'b0;
    else if (en) q <= d;
endmodule
```

Instead of posedge, use of negedge would implement a falling-edge D flipflop. After the specified edge, the flow into the **always ...** block begins.

The coding style presented for the above simple D flipflop is a general one and can be expanded to cover many features found in flipflops and even memory structures. The following description is of a register whose different D-flipflops can be synchronously initialized to different values (not necessarily all cleared or all preset).

```
module dff_N_bit_register ( clk, load, enable, init_d, d, q );
  parameter N = 4;
  input clk, load, enable;
  input [N-1:0] init_d, d;
  output [N-1:0] q;
  reg [N-1:0] q;

  always @( posedge clk ) begin
    if ( load ) q <= init_d;
    else if ( enable ) q <= d;
  end // always
endmodule
```

The above is equivalent to the following separation into a combinational block and a collection of appropriately driven synchronous memory elements, the DFFs.

```
  reg [N-1:0] q_next; reg [N-1:0] q;
  integer i;

  always @( * ) begin
    for  (i=0; i < N; i=i+1) begin
      q_next[i] <= ( load & init_d[i] ) |
                   ( ~load & enable & d[i] ) |
                   ( ~load & ~enable & q[i] );
    end // for
  end // always

  always @( posedge clk )
    for  (i=0; i < N; i=i+1) q[i] <= q_next[i];

  // or simply
  // always @( posedge clk ) q[N-1:0] <= q_next[N-1:0];
```

EXERCISE : Draw the schematic of a equivalently behaving parallel register with load control, enable control, and and appropriate data inputs.

A N-bit shift-register with right and left-shift capabilities, a serial-input, and parallel loading capability is shown below. Only the positive-edge of clk is included in the sensitivity list of the always block of this code, which makes all activities of the shift-register synchronous with the clock input.

```
module N_bit_shift_register ( clk, load, enable, left_right_b,
          init_d, din, q, dout );
  parameter N = 4;
  input clk, load, enable, left_right_b ;
  input [N-1:0] init_d, d; input din;
  output [N-1:0] q; output dout;
  reg [N-1:0] q; reg dout;

  always @( posedge clk ) begin
    if ( load ) q <= init_d;
    else if ( enable ) begin
      if ( left_right_b )  begin
        dout <= q[N-1]; q[N-1:0] <= { q[N-2:0], din };
      end else begin
        dout <= q[0]; q[N-1:0] <= { din, q[N-1:1] };
      end
    end
  end // always
endmodule
```

EXERCISE : Draw the schematic of a equivalently behaving shift register with left-shift / right-shift control, load control, enable control, and and appropriate data inputs.

3.3.3 Counters

Consider two ways of describing a four-bit counter with asynchronous reset. The first scheme implies a sequential circuit containing both the 4-bit flipflop and an adder. The second scheme explicitly declares modules for the flipflop and adder. Either scheme is good for a simple circuit such as a counter. As you develop more complex finite state machines, it is a good idea to separate the next state logic from the flipflops in your Verilog code. Verilog does not protect you from yourself here and there are many simple errors that lead to circuits very different than you intended.

```
module counter4bit(clk, reset, q);
  input clk; input reset;
  output [3:0] q;
  reg [3:0] q;
  // counter using always block
  always @(posedge clk)
    if (reset) q <= 4'b0; else q <= q+1;
endmodule

module adder4bit ( a,b,result ) ;
  input [3:0] a; input [3:0] b; output [3:0] result ;
  assign result = a + b ;
endmodule

module counter4bit_struct(clk, reset, q);
  input clk; input reset;
  output [3:0] q;
  wire [3:0] nextq;
  // counter using module calls
  four_bit_register register_inst1 (clk, reset, nextq, q);
  adder4bit adder_inst1 (q, 4'b0001, nextq);
      // assumes a 4-bit adder
endmodule
```

3.3.4 RAM

Verilog has an array construct used to describe memories. The following module describes a 64 word x 16 bit RAM that is, synchronously (w.r.t. posedge clk), written when "wrb" is low. Read is asynchronous (i.e. spontaneous), therefore whenever addr changes dout will change.

```
module my_ram64x16(clk, addr, wrb, din, dout);
  input clk ; input [5:0] addr; input wrb; input [15:0] din;
  output [15:0] dout;
  reg [15:0] mem[0:63]; // the memory
  always @(posedge clk)
    if (~wrb) mem[addr] <= din;
  assign dout = mem[addr];
endmodule
```

FPGAs have a limited number of bits of RAM on board. Large memories are extremely expensive. In the Xilinx Foundation tools, it is more efficient to specify a RAM or ROM using the LogiBLOX tool.

3.3.5 Blocking and Non-blocking Assignment

Verilog supports two types of assignments inside an always block. Blocking assignments use the = statement. Non-blocking assignments use the <= statement. Do not confuse either type with the (continuous) assign statement, which cannot appear inside always blocks at all.

Both these kind are indeed "sequential-zone statements" (a.k.a. "procedural-zone statements"), which means to say that these statements appear inside "sequential/procedural zones" of Verilog-based designs. So these statements are executed / interpreted in the sequence they appear in such a zone.

The "blocking" assignment uses the operator =. And its meaning is as in tranditional procedural programming languages such as C. First, expression on the right-hand side of such a "blocking" assignment is evaluated, and the resulting value is assigned to the signal on the left-hand side. The next statement does not execute until the current "blocking" assignment is completed.

On the other hand, the non-blocking sequential assignment uses the operator <=. In contrast to "blocking" assignment, in non-blocking assignment, the RHSs of the non-blocking assignment are evaluated (immediately of course). However the value obtained by the evaluation is not "immediately" assigned to the signal on LHS, rather the LHS signal assumes the new value at a future (infinitesimally later in future or after specific time in future). "Non-blocking"-ness refers to the fact that the evaluation of the expressions inside next statement is **not** "blocked" by this NBA (non-blocking assignment). We will later illustrate the subtle difference between blocking and non-blocking assignment.

A group of blocking assignments inside a begin/end block are evaluated sequentially, just as one would expect in a standard programming language. The objects on the LHS of a blocking assignment, if used in expression in later statement, would use the assigned value / expression. This help creates a substitution effect during synthesis, which is crucially effective for expressing a complicated combinational logic in finer / easier-fathomable steps. As a quick example of this substitution/build-up effect of blocking assignments, consider the following example, in which we compute xor of 1024 bits of a signal named "inp" and assign the result to reg-signal named "outp". Let us pretend for the time being that we are not aware of the "reduction-xor" operator in Verilog.

```
always @( inp[1023:0] ) begin
  tmp = 1'b0 ;
  for (i=0; i<1024; i=i+1 ) begin tmp = tmp ^ inp[i] ; end
  outp <= tmp ;
```

```
  end
```

This small example, also shows a safe kind of mix of blocking and non-blocking assignments. In general, it could be confusing to mix blocking and non-blocking assignments.

A group of non-blocking assignments (assuming no blocking assignments in the same zone) may be regarded as being evaluated in parallel (i.e. concurrently). All the expressions in the statements are evaluated before any of the left hand sides are updated. This is what one would expect in hardware because real logic gates all operate independently rather than waiting for the completion of other gates. For example, consider two attempts to describe a shift register. On each clock edge, the data at "sin" should be shifted into the first flop. The first flop shifts to the second flop. The data in the second flop shifts to the third flop, and so on until the last element drops off the end.

```
module shiftreg(clk, sin, q);
  input clk;
  input sin;
  output [3:0] q;
  // This is a correct implementation
  // using non-blocking assignment
  reg [3:0] q;

  always @(posedge clk) begin
    q[0] <= sin; // <= indicates non-blocking assignment
    q[1] <= q[0];
    q[2] <= q[1];
    q[3] <= q[2];
    // it would be even better to write q <= {q[2:0], sin};
  end
endmodule
```

The non-blocking assignments mean that all of the values on the right hand sides are assigned simultaneously. Therefore, q[1] will get the original value of q[0] and not the value of sin that gets loaded into q[0]. This is what we would expect from real hardware. Of course all of this could be written on one line for brevity.

Blocking assignments are more familiar from traditional programming languages, and if used improperly can inaccurately model hardware. Consider the same module using blocking assignments. When clk rises, the Verilog says that q[0] should be copied from sin. Then q[1] should be copied

from the new value of q[0] and so forth. All four registers immediately get the sin value.

```
module shiftreg(clk, sin, q[3:0]);
  input clk;
  input sin;
  output [3:0] q;
  // An INCORRECT implementation using blocking assignment
  reg [3:0] q;
  always @(posedge clk) begin
    q[0] = sin; // "=" indicates blocking assignment
    q[1] = q[0];
    q[2] = q[1];
    q[3] = q[2];
  end
endmodule
```

One popular recommendation is that use blocking assignments for preparing combinational logic and use non-blocking assignment for driving the finalized combinational logic on to a LHS reg-signal (in general, a verilog-variable on LHS) so that it can be communicated to rest of the circuit. In pure clock-edge-triggered procedural blocks, one should use non-blocking assignments to indicate simultaneous updates to edge-triggered register outputs by their new values.

Always keep at the back of your mind that each concurrent statement, whether a continuous assign statement of an always block models separate subcircuits of logic, which communicate among themselves via wire/reg/integer signal objects.

Therefore, a given reg may be assigned in only one always block. Otherwise, two pieces of hardware with shorted outputs would be implied (however one may indicate use of wired-and / wired-or etc for resolution of such multiple-driven reg-signals).

Chapter 4

Finite-State Machines

As we saw earlier, digital logic systems can be classified as combinational or sequential. Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system. Thus, a sequential system cannot be described with a truth table. Instead, a sequential system is described as a finite-state machine (or often just state machine). A finite-state machine has a set of states and two functions, called the next-state function and the output function. The set of states corresponds to all the possible values of the internal storage. Thus, if there are n bits of storage, there are 2^n states. The next-state function is a combinational function that, given the inputs and the current state, determines the next state of the system. The output function produces a set of outputs from the current state and the inputs.

The state machines we discuss here are synchronous. This means that the state changes together with the clock cycle, and a new state is computed once every clock. Thus, the state elements are updated only on the clock edge. We use this methodology (unless otherwise mentioned).

Some Terminology for Finite State Machines (FSMs)

Logic circuits are of two kinds: combinational and sequential. Combinational circuits are characterized by the property that their current outputs are completely determined by the current inputs. On the other hand a digital logic circuit is said to be sequential if the current outputs are dependent on not only the current input but also on the input values in the past. Thus a sequential circuit must have some memory elements to store the relevant knowledge / information from the past / history. For instance, in case of a sequence detector, as for detecting "101", we can easily build a sequential circuit, with the help of 3 DFFs constituting a 3-bit shift-register, which will store the values of the input that were sampled at the past 3 triggering edges

of the clock. That is an example of what I mean by "memorizing relevant information about the past". Sequential circuits have a finite amount of memory, and hence can capture only finitely many relevant scenarios about the past history.

Finite State Machines (abbreviated as FSMs) simply mean the familiar Sequential (Logic) Circuits that we have all along been talking about so often. Howeve, when we talk about FSMs, we are expected to specify, analyze, synthesize a sequential logic circuit by describing the essence of that circuit using a state diagram (or its equivalent. such as state transition table).

There are two kinds of sequential logic circuits: "asynchronous (spontaneous / clockless) sequential circuits" and the other kind being of "synchronous (edge-triggered) sequential circuits".

"**To synchronize (a synchronized event with a synchronizing event)**" means to ensure that the synchroniz**ed** event (for instance, update of DFF output) is engineered to occur at the synchroniz**ing** event (for instance, at the posedge of the clock). It makes natural sense to call the clock input to DFF, as a **synchronizing** signal, and the D-input to a DFF as **synchronized** signal.

In view of this nomenclature, an "**asynchronous** sequential circuit" is called so, because the update in the memory elements of such circuits are spontaneous (e.g. in an Set-Reset latch, the output of the latch is influenced spontaneously after a pulse at Set input or Reset input arrives; that is, there is no synchronizing signal to make the update wait until the synchroning event occurs). D-latch is another example of such asynchronous sequential circuit, as its output could be updated spontaneously in response to the changes at D-input, while the 'enable' control input is asserted.

Whereas in a **synchronous** sequential circuit element, say a (edge-triggered) DFF, the output of such (**synchronous**) FF memory element changes at the **synchronizing** event on the clock signal (say posedge or negedge).

Also note the usage of the adjective 'asynchronous', as in "**asynchronous active-low clear_b**". In this case, the **asychronous** characteristics manifests in the spontaneous response of the circuit to such active-low pulse at clear_b, in which the output(s) is/are cleared to 0. While in case of "**synchronous clear**", the response (of clearing the outputs to 0), is delayed until the **synchronizing** event (ie. posedge or negedge, whatever is specified) occurs at the **synchronizing** signal (which we typically call as "clock").

Let us get back to the introduction to FSM (ie. Finite State Machine). As remarked earlier, an FSM is simply an abstract picture of a sequential logic circuit (with finite amount of memory). The word "**state**" FSM refers to the contents of the **memory elements** of the sequential logic circuit (which

reflect the relevant knowledge about the past / history; "relevant" in the sense that it still matters in the current value of output). So for instance, in the "shift-register" based sequential circuit for detecting the sequence "101", the 3 memory elements, namely the 3 DFFs, are used for storing different possible "**states**" of this sequence detector. Such different states embody on the relevant amount of knowledge about the past, namely, the values at the input that were sampled at the past 3 synchronizing events on the clock signal. Note that this shift-register based 101-sequence detector also illustrates what we remarked earlier, that the output of a sequential circuit needs relevant information from the past (if the outputs were determined using only the current values of the input, then we would need NO memory; purely combinational logic circuit would have sufficed).

4.1 Review of systematic synthesis for FSMs

You are likely to be familiar with the systematic procedure of design of synchronous counters. The methodology can be outlines as below: formation of state diagrams and its equivalent state table, the states of which are then then encoded using different binary strings; use of a number of (synchronous) flipflops to store these binary string representation of the states; and the design of the combinational logic which will compute the next-state or excitation logic that would be input to the flipflops, so as to cause the flipflops to move to next state in the next clock cycle.

Let us now investigate how same approach can be used for the design of synchronous state machines. We will need some definitions first. We will be restricting our attenption to synchronous FSMs. The synchronous adjective would be assumed.

Synchronous Finite State Machine (FSM) is a synchronous sequential circuit. It produces outputs which depend on its internal state and external inputs. Its internal state stores the relevant knowledge about the past sequence of external inputs. More specifically, the set of (internal) states is the set of internal memorised values. In a diagramatic depiction of an FSM, the states are usually shown as circles with some annotations. FSM receives at its inputs the external stimuli. In state diagrams, the input values appear as labels on arcs; these arcs themselves describe the transition from current state to next state, caused by the current values of the input depicted on the (state-transition) arcs. Finally for an FSM to be a useful digital logic machine, it must produce some output for the external world.

There are two types of (synchronous) FSMs.

Two types of state machines are in general use, namely Moore machines

and Mealy machines. Mostly we will deal with Moore machines. Indeed, Moore and Mealy should be regarded as a classification of outputs. An output is of Moore-type, if its value is completely determined by the current state the FSM is in, whereas a Mealy-type output is one which depends both on the current state of the FSM and the current input. An FSM is said to be of Moore-type if all the outputs are of Moore type, and is said to be of Mealy type, if all the outputs are of Mealy type only.

Moore-type outputs will be combinationally generated only using the outputs of the (synchronous) FFs. As the outputs of these (synchronous) FFs are stable throughout a clock cycle (after a possible update right at the beginning of a clock cycle), the Moore-type outputs have better timing characteristics, in the sense that these outputs are guaranteed to be become stable early enough during a clock cycle (and certainly not depend on the arbitrarily timed event on the external inputs).

In spite of these differences, in terms of functional capabilities (which means we ignoring differences in timing, and focussing only on the values of signals at synchronising events on the clock only), both Moore-type and Mealy type FSMs are equivalent. Let us consider a design of a Moore-type FSM that functions as a traffic light controller.

4.1.1 A Traffic Light Controller (TLC) (without any inputs)

We assume a very small, single road with very little traffic. The pedestrians too are very infrequent. Let us suppose that the visibility at the pedestrian crossing on this narrow road is rather poor, and the traffic lights controller logic will be cautious enoiugh to allow equal time for pedestrian crossing, the vehicles on the road, and also same time interval during which neither vehicles not the pedestrians cross. Tolerate this contrived specifications for the time being. Soon you would be capable of designing an arbitrarily complex traffic lights controller for a very complex traffic junction with multiple roads and pedestrian crossings. So we begin very simple so as to get some feel at the nuts and bolts level.

So the three traffic lights at the pedestrian crossing are Red, Yellow and Green, and that these lights cycle through the following 4 states (staying in each state for the same amount of time, say 1 (slow) clock cycle). The states correspond to the following different combinations of Moore-outputs R, Y, G, namely a state in which RYG=3'b100, another in which RYG=3'b110, third state for which RYG=3'b001 and the fourth one in which RYG=3'b010.

We could indeed name these states, in order to make this correspon-

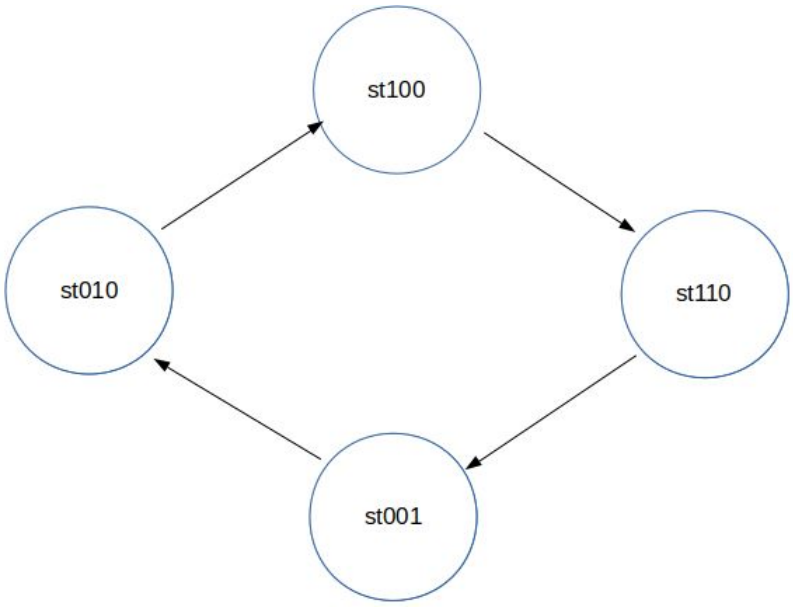


Figure 4.1:

dence with these Moore-outputs explicit, as follows, $st100$, $st110$, $st001$ and $st010$, naturally corresponding to the RYG values, 3'b100, 3'b110, 3'b001 and 3'b010 respectively.

As there are only 4 states 2 FFs would have sufficed. However to make you aware of a trade-off between optimization (of number of FFs) and simplicity of the combinational part of the FSM, we will use 3 FFs in a obviously natural way to store these 4 possible different states. This would make resulting combinational logic simpler. In larger examples such choices could show more significant trade-offs. That you will gain insights into as you practice this art and engineering of FSM design and synthesis. In this booklet, our aims are rather limited to help you get jumpstarted rather quickly (at the unfortunate expense of studying various perfomance, optimization and engineering issues in design and synthesis of FSMs).

Clearly by using 3 FFs to drive the three traffic lights directly, we do not need any output combinational logic. Although it might appear wasteful, as 4 of the $2^3 = 8$ different bit-vectors of these 3 FF outputs are used and 4 are wasted / unused, as remarked above, its benefit is obvious; trivial combinational logic for the output signals R,Y and G which will be used to drive the respective traffic lights.

One can use any kind of synchronous FFs, DFFs, JKFFs or TFFs. All FFs are equivalent in their capability as state elements. Simplest to use are the DFFs.

Let us write down the state transition table.

R	Y	G			nextR	nextY	nextG
1	0	0			1	1	0
1	1	0			0	0	1
0	0	1			0	1	0
0	1	0			1	0	0

If we were to use JKFFs, or TFFs, then we would need to figure out what should be the excitation inputs to these FFs, so that the appropriate next values would be loaded into these FFs at the triggering event of the synchronizing clock. As we have decided to use DFFs, we do not need columns for FF excitation in this tabular representation which would be used for construction of required combinational logic.

Some of the combinations of the bit-values stored in FFs, and some input combinations will never occur, we don't care what output the next state combinational logic gives for these inputs and these useless artificial states. These don't care conditions can be used to simplify the required next state combinational logic.

Let us construct the combinational logic for next-state.

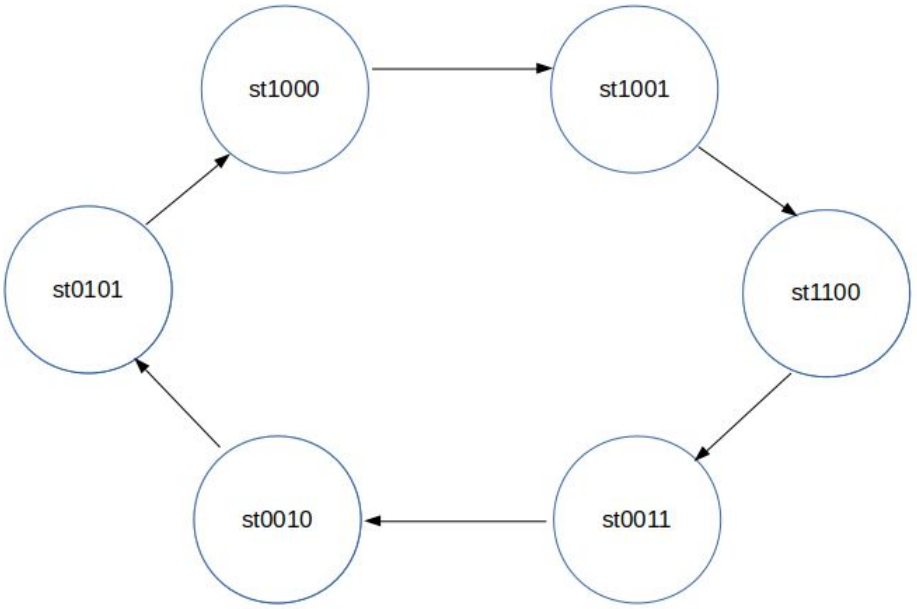
For each of the three FFs, one needs to find combinational logic that would generate nextR, nextY, nextG that would be a combinational function of the current states signals R,Y and G. We can do it using K-maps etc., or in ad-hoc manner. One can show that we can take nextR to be XOR(R,Y). And similarly we can construct suitable combinational logic for nextY and nextG.

```
assign nextR = R ^ Y;
assign nextY = ~Y;
assign nextG = R & Y;
```

How do we appropriately initialize such FSM circuit ? In this example, let us assume that we want the FSM to be in state RYG=3'b100 at the power-up (reset) time. We leave it as an exercise for the reader to see how to drive the asynchronous, (say, active-low) clear and (active-low) preset control inputs on standard FFs, so as to ensure the desired initialization of these FFs, when say an active-low power-up pulse arrives.

Let us modify our specification to enhance this toy version of TLC so that we consolidate our understanding of FSMs. To make it less-contrived, let us now provide more time for the Red and Green lights, say double the time than when the Yellow light is on. As we can see, we can achieve this objective with 2 additional states, making the number of states 6 in total.

The 3 FFs that we used in the above version would have sufficed (as $6 \leq 2^3$). However for the same reasons as above, let us make use of one more DFF whose output is labelled S. The 6 states now represent
{R,Y,G,S}==4'b1000 , {R,Y,G,S}==4'b1001, {R,Y,G,S}==4'b1100,
{R,Y,G,S}==4'b0011, {R,Y,G,S}==4'b0010, {R,Y,G,S}==4'b1001.



Using the state assignment of 4 DFFs, namely R,Y,G and S, we get the following simple combinational logic expressions for their next state values.

```
assign nextR = R ^ Y;
assign nextY = ~(R ^ S);
assign nextG = R & Y | G & S;
assign nextS = ~S;
```

The above examples illustrate how one does this clever manual labour of doing intelligent state encoding and then deriving neat combinational logic expressions for the D-inputs of the state-carrying DFFs.

Having done such effort, it is straightforward to model such finite state machine in Verilog or any HDL, by simply modeling in Verilog, the subcircuits involved, namely, the state-carrying DFFs and the next-state combinational logic expressions. The combinational logic expressions for the fsm-outputs too are constructed using combinational logic manipulations.

This will be an example of structural / dataflow style of HDL based design.

EXERCISE : Write Verilog Models for the FSM examples shown above, using the manually prepared state encoding and corresponding combinational logic expressions for next-state of the DFFs and of the FSM-outputs.

Of course, such logic synthesis can also be done using CAD tools such as Berkeley SIS, ABC etc by feeding the specifications in a user friendly manner. However the HDL CAD tools from fpga vendors will indeed use logic synthesis algorithms similar to those in SIS, ABC etc. to provide further convenience of converting even friendlier HDL-based description directly into a circuit of primitive logic cells.

This involves, as you would have guessed, a behavioral style HDL-based design. We will see that in next section.

4.2 Verilog Modeling of Finite State Machines

We illustrate Verilog-based modeling of an FSM using an example of a traffic light at an intersection of main road and a less busy farm road . This is a toy example, and can be easily extended as an exercise. Sensors are used to detect vehicle on the main road as well on the side road. The main road traffic signal cycles through Green and Red lights. Similarly the side road traffic light cycles through Green and Red lights.

The output signals correspond to the traffic lights being green or red. Specifically, the output signal MRLight is used to indicate Green signal on the main road, when asserted HIGH and Red light signal when deasserted. Similarly, the output signal SRLight is used to indicate Green signal on the side road, when asserted HIGH and Red light signal when deasserted. This system is controlled by inputs from the car sensors on the main road and the side road.

Let us denote the main road sensor signal by MRsensor, and the side road signal by SRsensor. The traffic signal light should let the traffic momentum by sustaining to show green in the same direction as the last car crossing the junction, as long as no car is sensed on the other road. Clearly, this sequential system would be in one of the two states, each of which indicates the road along which the traffic has been allowed to flow. Let us name these states MRgreen and SRgreen to indicate the obvious.

For synthesizing this sequential machine (with these two states, the above mentioned inputs and outputs), we need to work out next state function and the output function. Constructing these functions would become an exercise in building combinational logic circuits (for generating next state and output).

In the graphical representation, nodes are used to represent states. An output is of Moore type, if it is a function only of the current state. Values of such (Moore) outputs are rendered inside the node representing the state. State transitions are rendered using directed arcs between nodes. The input conditions causing the transitions are described as labels on these directed arcs.

A finite state machine would have at its core, sufficient number of (synchronous) memory elements (we usually call them flipflops) to depict the current state. The values stored in these state-holding flipflops and the current input signal values are fed to suitable combinational logic that generates the values of the next state and the outputs.

We need to perform state-assignment. In our example we have two states, namely MRgreen and SRgreen. One D flipflop would suffice. State MRgreen can be encoded using value 0 and SRgreen by value 1.

The following Verilog code snippet captures the combinational dependence of next state on current state and the current value of the inputs MRsensor, SRsensor, (i.e. stabilized values of these signals at the end of the current clock cycle). This code snippet also describes the combinational dependence of the outout signals, MRlight and SRlight, on the state in the current clock cycle.

```
parameter MRgreen = 1'b1;
parameter SRgreen = 1'b0;

always @(*)
  case ({ current_state, MRsensor, SRsensor } )
    { MRgreen, 2'b00 } : next_state <= MRgreen;
    { MRgreen, 2'b01 } : next_state <= SRgreen;
    { MRgreen, 2'b10 } : next_state <= MRgreen;
    { MRgreen, 2'b11 } : next_state <= SRgreen;
    { SRgreen, 2'b00 } : next_state <= SRgreen;
    { SRgreen, 2'b01 } : next_state <= SRgreen;
    { SRgreen, 2'b10 } : next_state <= MRgreen;
    { SRgreen, 2'b11 } : next_state <= MRgreen;
    default : next_state <= MRgreen;
  endcase

always @(*)
  case ( current_state )
    MRgreen : {MRlight, SRlight} <= 2'b10;
```

```
    SRgreen : {MRlight, SRlight} <= 2'b01;
    default : {MRlight, SRlight} <= 2'b10;
  endcase
```

Exercise : Based on the above Verilog modeling of the combinational logic for next state and the output signals, construct truth-tables, Karnaugh maps to validate the following simple combinational logic expressions.

```
  assign next_state  =
    (~current_state & SRsensor) | (current_state & ~MRsensor);
  assign MRlight = ~ current_state;
  assign SRlight = current_state;
```

Just for the sake of illustrating diverse ways of expressing the same thing, we present the above a bit differently, but equivalently, in the following Verilog model of this FSM.

```
module TLC (clk, rst_b, MRsensor, SRsensor, MRlight, SRlight);

  input clk, rst_b, MRsensor, SRsensor;
  output MRlight, SRlight ;

  reg current_state, next_state;

  parameter MRgreen = 1'b1;
  parameter SRgreen = 1'b0;

  assign MRlight = ~ current_state;
  assign SRlight = current_state;

  always @( * )
    case (current_state)
      SRgreen : next_state <= SRsensor ? SRgreen : MRgreen ;
      MRgreen : next_state <= MRsensor ? MRgreen : SRgreen ;
    endcase

  always @(negedge clk, negedge rst_b)
    if ( rst_b == 0 )
```

```
      current_state <= 0;  // initial (  "on-reset" ) state
    else
      current_state <= next_state;

endmodule
```

4.3 FSM : Some Examples

In this section, we further present some illustrative examples of FSMs.

4.3.1 FSM for a Sequence Detector

Suppose we need to build an FSM that would detect the pattern 101 that has arrived over past 3 clock cycles, one bit a time, over a single bit input port. The recognition should be indicated by asserting a single bit output signal HIGH over the clock cycle immediately after noticing the above 3-bit sequence.

The circuit has one input, w, and one output, y. We assume that the input 'w' changes at most once during a clock cycle (assuming posedge triggered clock, the clock cycle is duration between successive posedges of the clock). By the value of 'w' in a clock cycle we mean the value of 'w' that has stabilized prior to the posedge of the clock, at the end of the clock signal. Let's assume that setup/hold time requirements are met by the input 'w' with respect to the clock signal.

Let us also assume that all the updates of the synchronous memory elements in the logic circuit occur on the positive edge of a clock signal.

The output y is asserted-HIGH if over 3 immediately preceding clock cycles the single bit input 'w' was equal to 101 (i.e. w=1 at the end of third previous clock cycle, 0 at the end of second previous clock cycle, 1 at the end of the just concluded clock cycle) . Otherwise, the value of output y is equal to 0 (i.e. y. is deasserted LOW).

Clearly the circuit needs to have memory as the output y cannot depend solely on the present value of w.

State Diagram

Let us fix a starting state that the circuit should enter on power-up reset. Let the starting state be called 'stRESET'. In this state the output 'y' would be deasserted-low, that is at value 0. While the circuit is in this state, if the input 'w' has a stable value of 0 at the end of the current clock cycle, the circuit would continue to be state 'stRESET' for the duration of the next clock cycle too. If, on the other hand, 'w' is found to be stable at value 1

at the end of the current clock cycle, while the circuit is in state 'stRESET', then the FSM circuit should move to a different state. Let us call this state as 'stGOT1', clearly to indicate that the FSM has by this time seen the prefix '1' of the sequence '101' that is to be recognized. In this case, the FSM circuit would make a transition to state 'stGOT1' at the end of the current clock cycle. In state 'stGOT1' too the circuit would have output y at 0, because it has not yet witnessed the required sequence 101 at input 'w' for past 3 clock cycles. While in state stGOT1 during the current clock cycle, if the input 'w' is (stable at) 1 at the next active clock edge (that is at the end of the current clock cycle), the circuit should stay in state stGOT1. However, if w is at 0, at the end of the current clock cycle, while in state stGOT1, the circuit should change to a state, which we can appropriately call 'stGOT10'. In this state too the output would still be at y = 0. From state stGOT10 the circuit would go to a state, which we could appropriately designate as 'stGOT101', if at the end of the current clock cycle the the value of input w is 1. And when the FSM circuit enters this state 'stGOT101' in the next clock cycle, then the output 'y' would be justifiably asserted-HIGH (ie. y = 1). On the other hand, if 'w' is 0 at the end of current clock cycle, when state is stGOT10, the FSM would stay in the same state in the next clock cycle too.

The above informal verbose specification of the behaviour of the FSM circuit, can be depicted in a pictorial representation, called **state transition diagram**, In such *state transition diagram*, the states of the circuit are shown as annotated nodes (circles) and the transitions between states are shown as annotated directed arcs. States stRESET, stGOT1, stGOT10, and stGOT101 should appear as nodes in this state diagram. In stReset the output 'y' should be 0, and this is indicated by the annotation

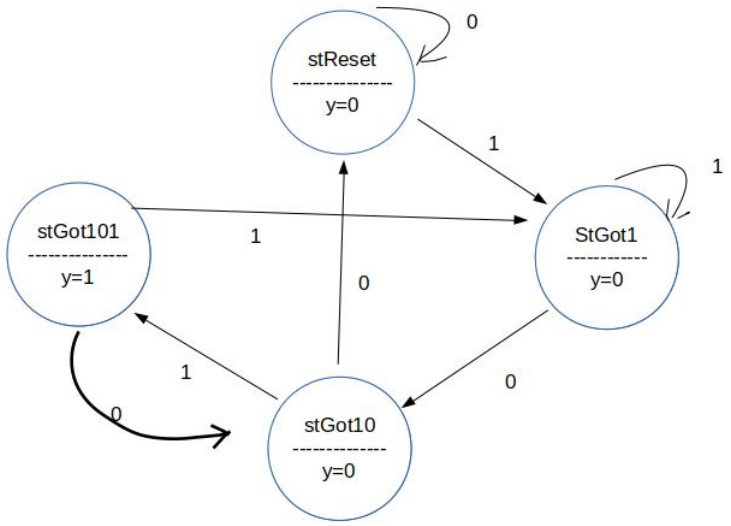
$$\frac{\text{stRESET}}{\text{y=0}}$$

inside the node. The circuit would remain in state stReset, over clock cycles as long as the input 'w' is sampled as of value 0, which is indicated by an arc that originates and terminates at this node. This arc is annotated with label w = 0. The transition caused in the case, when the value of w = 1 at the end of current clock cycle, while FSM is in state stReset, is represented in the state diagram by an annotated directed arc from state stRESET to state stGOT1. This directed arc describing the transition is annotated with the label w = 0. In general the transition arcs would carry the annotation which describe the the condition of input values at the end of current clock cycle that would causes the state transition. In state stGOT1 too the output

is at 0, and this is indicated by the annotation

$$\frac{\text{stGOT1}}{\text{y=0}}$$

in the node.



State Table

State diagram is easy to understand. But to process this abstract information with the help of a computer program (or a systematic mechanical procedure), the information contained in the state diagram is translated into a tabular form, called **state-transition table**.

State table for a Moore sequence detector detecting 101 on its single-bit input 'w' is shown below. The machine has four states that are labeled, stRESET, stGOT1, stGOT10, stGOT101. Starting in stRESET, if the 101 sequence is detected, the machine goes into the stGOT101 state in which the output becomes 1. In addition to the w input, the machine has a 'rst' input that forces the machine into its stRESET state. The resetting of the machine is synchronized with the clock.

CurrentState	NextState		Output
	w=0	w=1	y
stRESET	stRESET	stGOT1	0

```
stGOT1          stGOT10     stGOT1     0
stGOT10         stRESET     stGOT101   0
stGOT101        stGOT10     stGOT1     1
-------------------------------------------------
```

State Assignment (a.k.a. state-encoding)

For the purpose of synthesizing the state-diagram/table based description of an FSM, one needs to represent each state by a distinct binary word. Such binary word is to be regarded as a particular valuation of of 0/1 valued (single-bit) signals carrying state information. Each state-carrying bit may be implemented in the form of a flip-flop. Since 4 states have to be realized, it is sufficient to use 2 state-carrying bits. Let these be modeled using verilog-variables be q[1] and q[0]. The output 'y' is combinationally dependent only on the current state of the circuit. that is a function only of q[1:0]. Such output 'y' is said to be of Moore-type. We need to construct a combinational circuit with q[1:0] as input signals, and which generates corresponding value of 'y'. The state signals q[1:0] are also fed back to another combinational circuit that determines the next state of the FSM. However this combinational circuit would also need to use the primary input signal w. Its outputs are 2-bit signals, next_q[1:0], which represent the new values to be loaded into the (state-holding) flip-flops. Each triggering edge of the clock will cause the flip-flops to change their state so that the values of next_q[1:0] would appear at flipflop outputs during the next clock cycle. Justifiably, we call the signals next_q[1:0] are called the next-state signals, and q[2:0] are called the (current-) state carrying signals.

So one can build a truth-table that will capture the required combinational logic circuits. Therefore, we assign a distinct binary word to q[1:0] for each state. One possible assignment / encoding is as follows where the states stRESET, stGOT1, stGOT10 and stGOT101 are represented by

```
q[1:0] == 2'b00, 2'b01, 2'b10, 2'b11
```

respectively. Next we construct the state-assigned table (or state-encoded transition table). And this state-encoded table would include all of the necessary information to define the combinational logic for next-state and output signals, in terms of valuations of inputs w, q[1:0].

Derivation of Combinational Logic for Next-State and Output Signals

We will use 2 DFFs to provide q[1:0] at their outputs. Let is call the inputs to the flipflops as D[1:0]. Then these inputs D[1:0] are indeed the same as next_q[2:0]. Using Karnaugh maps, one can construct combinational logic expressions for each of bit in next_q[1:0] and also for output y .

```
module moore_101_detector (w, rst, clk, y );
  input w, rst, clk;
  output y; reg y;
  parameter [1:0] stRESET = 0, stGOT1 = 1,
                  stGOT10 = 2, stGOT101 = 3;
  reg [1:0] state;
  always @ ( posedge clk ) begin
    if (rst) state <= stRESET;
    else begin
```

```
      case ( state )
        stRESET: begin
          if ( w==1'b1 ) state <= stGOT1 ;
          else state <= stRESET;
        end
        stGOT1: begin
          if ( w==1'b0 ) state <= stGOT10;
          else state <= stGOT1;
        end
        stGOT10: begin
          if ( w==1'b1 ) state <= stGOT101;
          else state <= stRESET;
        end
        stGOT101: begin
          if ( w==1'b1 ) state <= stGOT1;
          else state <= stGOT10;
        end
        default: state <= stRESET;
          // Only for illegal input ( ?? )
          // Legal inputs would not bring FSM here.
      endcase
    end
  end // always
  // Moore output logic
  // ( combinational function of just "state" )

  always @( * ) begin
    case ( state )
      stRESET : y <= 0;
      stGOT1 : y <= 0;
      stGOT10 : y <= 0;
      stGOT101 : y <= 1;
      default : y <= 0;
    endcase
  end
endmodule
```

After the declaration of inputs and outputs of this module, parameter declaration declares four states of the machine as two-bit parameters. The square-brackets following the parameter keyword specify the size of parameters being declared. Following parameter declarations, the two-bit signal

"state" of type **reg** is declared. This signal holds the current state of the state machine. The clock triggered **always ...** block used in the module describes state transitions. And the combinational **always ...** block describes the output assignments of the state machine as given in the state table. The main task of procedural block for modeling state transitions is to inspect input conditions (values on 'rst' and 'w') during the current state of the FSM defined by "state" and set values into "state" (which will appear in the next clock cycle, ie. after the triggering edge of the clock).

4.3.2 Toy Verilog Model of a Washing Machine

There are counter-like sequential circuits which do not count in the usual sense, but they keep track of current step in a repeatable sequence of steps, by making transitions through a set of states. They also produce a repeated sequence at the outputs. There are numerous applications of such sequence generating circuits. They too have a counter-like structure, and may be regarded loosely as variants of counters.

Consider the following oversimplified example [**?**] of an FSM with four states modeling an oversimplified behaviour of a typical washing machine.

- Until the start button is pressed, the washing machine remains idle.
- Then it is filled with water.
- Next it runs the agitator until a timer triggers it to stop agitating.
- It is followed with the spinning of the tub to get the water out.
- Finally it goes back to idle.

Verilog description of the toy-model of Washing Machine

The Verilog code below models this toy-version of washing machine. You would notice use of 'parameter' declarations to create useful 'aliases' for the binary encodings of the states.

```
.......
parameter stIdle=2'b00, stFill=2'b01,
          stAgitate=2'b10, , stSpin=2'b11;
.......
```

We also declare a 2-bit signal 'state' of type 'reg', for the purpose of holding the value of current state of the FSM. You would also notice a related 2-bit signal, named 'next_state', which, as the name suggests, would hold the

values that should appear in the 'state' holding signal in the next clock cycle. The posedge-triggered always-construct is used to describe the loading of the values of signal 'next_state[1:0]' into the flipflops driving 'state[1:0]', at the posedge of the clock. There are two "always @(*) ..." blocks of statement in the Verilog code. One of them specifies the combinational logic required to generate the 'next_state[1:0]' signal values, from the values of 'state[1:0]', and also the values of signals 'start', 'full', 'timesup' and 'dry'. The other "always @(*) .... " block of statements models the combinational logic that generates values of output signals, namely, values of signals 'water_valve', 'ag_mode' and 'sp_mode'.

```
module washing_machine_toy_model
  ( clk, start, full, timesup, dry,
    water_valve, ag_mode, sp_mode );

  input clk, start, full, timesup, dry;
  output reg water_valve, ag_mode, sp_mode;

  parameter stIdle=2'b00, stFill=2'b01,
            stAgitate=2'b10, stSpin=2'b11;

  reg [1:0] state, next_state;

  always @( posedge clk ) begin
     state <= next_state;
  end // always

  always @( * ) begin
   case ( state )
      stIdle : begin
          if ( start ) next_state <= stFill;
          else next_state <= stIdle;
        end
      stFill : begin
          if ( full ) next_state <= stAgitate;
          else next_state <= stFill;
    end
      stAgitate : begin
          if ( timesup ) next_state <= stSpin;
          else next_state <= stAgitate ;
        end
```

```
      stSpin : begin
         if ( dry ) next_state <= stIdle;
         else next_state <= stSpin;
        end
      default : next_state <= stIdle;
    endcase
  end // always

  always @( * ) begin
    case ( state )
      stIdle : begin
          water_valve <= 0; ag_mode <= 0; sp_mode <= 0;
        end
      stFill : begin
          water_valve <= 1; ag_mode <= 0; sp_mode <= 0;
        end
      stAgitate : begin
          water_valve <= 0; ag_mode <= 1; sp_mode <= 0;
        end
      stSpin : begin
          water_valve <= 0; ag_mode <= 0; sp_mode <= 1;
        end
      default : begin
          water_valve <= 0; ag_mode <= 0; sp_mode <= 0;
        end
    endcase
  end // always
endmodule
Washing Machine ( toy-model ) FSM in Verilog HDL
```

4.3.3 Traffic Lights Controller, Once Again !

Exercise : Study the following Verilog model of a traffic light controller, which is obtained by routine translation (into Verilog) of a VHDL model of a TLC from the text of Tocci-Widmer-Moss [**?**]. Your task is to reconstruct the state diagrams of various FSMs interacting with one another.

```
module mTLC_tocci ( clk, car, reset,
      tmaingrn, tsidegrn, lite, change,
      mainred, mainyelo, maingrn,
      sidered, sideyelo, sidegrn );
```

```
  input clk, reset, car;
  input [4:0] tmaingrn, tsidegrn;
  output [1:0] lite;
  output change;
  output mainred, mainyelo, maingrn;
  output sidered, sideyelo, sidegrn;

  mDelay mDelay_instance ( .clk(clk), .car( car ),
    .reset(reset), .lite(lite), .tmaingrn(tmaingrn),
    .tsidegrn(tsidegrn), .change(change));

  mControl mControl_instance ( .clk(clk),
    .enable(change), .reset(reset), .lite(lite));

  mLiteCtrl mLiteCtrl_instance (
    .lite(lite), .mainred(mainred), .mainyelo(mainyelo),
    .maingrn(maingrn), .sidered(sidered),
    .sideyelo(sideyelo), .sidegrn(sidegrn));

endmodule

module mDelay ( clock, car, reset,
    lite, tmaingrn, tsidegrn, change );

  input clk, car, reset;
  input [1:0] lite;
  input [4:0] tmaingrn, tsidegrn;
  output change ;

  reg [4:0] mach;
  reg change;

  // 1 Hz clock
  always @( posedge clk, negedge reset ) begin
    if ( reset == 0 ) mach <= 4'b0;
    else if ( mach == 0 ) begin
      case ( lite )
        2'b00 : begin
            if ( car == 0 ) mach <= 0;
               // wait for car on side road
```

```
            else mach <= tmaingrn - 1;
               // set time for main's green
          end
        2'b01 : begin
            mach <= 5 - 1;
               // set time for main's yello
          end
        2'b10 : begin
            mach <= tsidegrn - 1;
               // set time for side's green
          end
        2'b11 : begin
            mach <= 5 - 1;
               // set time for side's yellow
          end
        default : mach <= 0;
                 // would NEVER be here
      endcase
    end
    else mach <= mach - 1;
  end // always

  always @( * ) begin
    if ( mach == 1 ) change <= 1;
    else change <= 0;
  end  // always

endmodule

module mControl ( clk, enable, reset, lite );
  input clk, enable, reset;
  output [1:0] lite;

  parameter mgrn=2'b00, myel=2'b01,
        sgrn=2'b10, syel=2'b11;

  reg [1:0] lights;
  reg [1:0] lite;

  always @( posedge clk, negedge reset ) begin
    if ( reset == 0 ) lights <= mgrn;
```

```
    else if ( enable == 1 ) begin
      case ( lights )
        mgrn : lights <= myel;
        myel : lights <= sgrn;
        sgrn : lights <= syel;
        syel : lights <= mgrn;
        default : lights <= mgrn;
      endcase
    end
  end // always

  always @( * ) begin
    case ( lights )
      mgrn : lite <= 2'b00;
      myel : lite <= 2'b01;
      sgrn : lite <= 2'b10;
      syel : lite <= 2'b11;
      default : lite <= 2'b00;
    endcase
  end

endmodule

module mLiteCtrl ( lite,
      mainred, mainyelo, maingrn,
      sidered, sideyelo, sidegrn );

  input [1:0] lite;
  output mainred, mainyelo, maingrn;
  output sidered, sideyelo, sidegrn;

  reg mainred, mainyelo, maingrn;
  reg sidered, sideyelo, sidegrn;

  always @( * ) begin
    case ( lite )
      2'b00 : begin
          maingrn <= 1; mainyelo <= 0; mainred <= 0;
          sidegrn <= 0; sideyelo <= 0; sidered <= 1;
        end
      2'b01 : begin
```

```
          maingrn <= 0; mainyelo <= 1; mainred <= 0;
          sidegrn <= 0; sideyelo <= 0; sidered <= 1;
        end
      2'b10 : begin
          maingrn <= 0; mainyelo <= 0; mainred <= 1;
          sidegrn <= 1; sideyelo <= 0; sidered <= 0;
        end
      2'b11 : begin
          maingrn <= 0; mainyelo <= 0; mainred <= 1;
          sidegrn <= 0; sideyelo <= 1; sidered <= 1;
        end
    endcase
  end
endmodule
```

4.3.4 Combinational Lock (from Peckol's FSM tutorial [?])

Exercise : Study the following Verilog module (courtesy Peckol's tutorial on FSM [?]) and reconstruct the state diagram. Synthesize the state diagram into a DFF based logic circuit and validate the equivalence of its behaviour with that of the following through simulation and fpga-based testing.

This code that is rephrased version of the corresponding one from Peckol's tutorial on FSM [?], is deliberately a little obfuscated, so as to let the reader get a bit of exercise in reading and fathoming the FSM underneath it. The reader is recommended to read the self-contained tutorial document on FSM by Peckol [?] which is the original source of this example code. Reader would also be able to cross-compare her understanding of this code with the completely worked out details of this FSM which can be found in [?].

```
module CombinationLock(clk, reset_b, digit_in, lock_open );
  input clk, reset_b;
    // posedge  triggering clock and synchronous active-low reset
  input [3:0] digit_in; // bcd digit input
  output lock_open; // whether lock opened or not
  reg lock_open; reg [1:0] state;
  always@ ( posedge clk)
    if ( reset_b == 0)  begin  state <= 2'b00;  lock_open <= 0; end
    else
      case (state)
```

```
        2'b00: begin  lock_open <= 0;
            if (digit_in == 4'd3)  state <= 2'b01;
          end
        2'b01: begin lock_open <= 0;
            if ( digit_in == 4'd2 ) state <= 2'b11;
            else  state <= 2'b00;
          end
        2'b11: begin lock_open <= 0;
            if( digit_in == 4'd2 )  state <= 2'b10;
            else  state <= 2'b00;
          end
        2'b10: begin  state <= 2'b00;
            if( digit_in == 4'd1 ) lock_open <= 1;
            else  lock_open <= 0;
          end
        default : begin state <= 2'b00; lock_open <= 0; end
      endcase
endmodule
```

4.3.5 Exercise for the Reader : HDLC fsm

High-Level Data Link Control (HDLC) is a synchronous data link layer protocol developed by ISO.

The communication links on which HDLC frames can be transmitted do not have mechanism to mark the beginning or end of a frame. Therefore, as a frame-delimiter sequence, it uses a unique sequence of bits "01111110". The sender in this protocol needs to encode the data to ensure that the frame-delimiter sequence does not occur inside the frame data. This is ensured by bit-stuffing idea, by inserting a zero after every 5 consecutive 1s. The receiver must detect this and discard the artificially injected 0. An error is to be signalled if there are 7 or more consecutive 1s.

Interpretation of the following code is left as an exercise for the reader. The reader is also encouraged to try this out on the popular HDLbits website (https://hdlbits.01xz.net/wiki/Fs_hdlc).

```
module hdlc_fsm ( clk, reset, inp, discard_flag,
      frame_delimiter_flag, error_flag ) ;
  input clk, reset, inp ;
  output discard_flag, frame_delimiter_flag, error_flag ;
  localparam [3:0] st0=0, st01=1, st02=2, st03=3, st04=4, st05=5,
                   st06=6, st050=7, st060=8, st07=9 ;
```

```
  reg [3:0] state, next_state ;
  always @( posedge clk )
    if (reset) state <= st0 ; else state <= next_state ;
  always @( state , inp ) begin
    next_state <= state ;
    case ( state )
      st0 : begin
            if (inp==1'b1) next_state<=st01 ; else next_state<=st0 ;
          end
      st01 : next_state <= inp ? st02 : st0 ;
      st02 : if (inp) next_state <= st03 ; else next_state <= st0 ;
      st03 : if (inp==1) next_state<=st04 ; else next_state <= st0 ;
      st04 : begin next_state <= (inp==1'b1) ? st05 : st0 ; end
      st05 : begin next_state <= inp ? st06 : st050 ; end
      st06 : begin next_state <= inp ? st07 : st060 ; end
      st050 : next_state <= inp ? st01 : st0 ;
      st060 : begin next_state <= inp ? st01 : st0 ; end
      st07 : begin next_state <= inp ? st07 : st0 ; end
      default : ;
    endcase
  end
  assign discard_flag = ( state==st050 ) ? 1'b1 : 1'b0 ;
  assign frame_delimiter_flag = ( state==st060 ) ? 1'b1 : 1'b0 ;
  assign error_flag = ( state==st07 ) ? 1'b1 : 1'b0 ;
endmodule
```

4.4 Verilog Modeling of a Datapath and Controller-FSM

4.4.1 Exercise : Draw the Datapath, FSM corresponding to the given code

This is an example design for Booth Multiplier. It is complete with an exhaustive testbench. Studying this code is an instructive exercise.

The reader is asked to daw the Datapath, FSM corresponding to this given code.

```
module booth_controller (clk, start, counter_zero, mplr0, mplr0_old,
    load_acc, load_mplr, load_mcnd ,
    clear_acc, clear_mplr , clear_mcnd , clear_mplr0_old ,
    rshft_signed_acc, rshft_mplr,
    add0_sub1, decr_counter, load_counter, done ) ;
```

```
input clk, mplr0, mplr0_old, start , counter_zero ;
output reg  load_acc, load_mplr, load_mcnd ,
  clear_acc, clear_mplr , clear_mcnd , clear_mplr0_old ,
  rshft_signed_acc, rshft_mplr , add0_sub1, decr_counter,
  load_counter, done ;
localparam [2:0] st_wait_start=3'd0, st_inp_data=3'd1, st_add=3'd2 ;
localparam [2:0] st_sub=3, st_shift=4, st_done=5, st_branch=6 ;
reg [2:0] state = st_wait_start;
always @( posedge clk ) begin
  case ( state )
    st_wait_start : if ( start ) state <= st_inp_data ;
    st_inp_data : state <= st_branch ;
    st_branch :
        if ( {mplr0,mplr0_old}==2'b01  &&  (!counter_zero) )
  state <= st_add ;
        else if ({mplr0,mplr0_old}==2'b10 && (!counter_zero))
  state <= st_sub ;
        else if ( counter_zero ) state <= st_done ;
else state <= st_shift ;
    st_shift : state <= st_branch ;
    st_add : state <= st_shift ;
    st_sub : state <= st_shift ;
    st_done : state <= st_wait_start ;
    default : state <= st_wait_start ;
  endcase
end
always @( * ) begin
  load_acc=0 ; load_mplr=0 ; load_mcnd=0 ; clear_acc=0 ;
  clear_mplr=0 ; clear_mplr0_old=0 ;
  rshft_signed_acc=0 ; rshft_mplr=0 ; add0_sub1=0 ;
  decr_counter=0 ; load_counter=0 ; done=0 ;
  case ( state )
    st_wait_start : ;
    st_inp_data :
        begin clear_acc=1 ; clear_mplr0_old=1 ;
          load_counter=1 ; load_mcnd=1 ; load_mplr=1 ;
end
    st_branch : ;  // nothing to be done on datapath
    st_add : begin load_acc=1 ; add0_sub1=0 ; end
    st_sub : begin load_acc=1 ; add0_sub1=1 ; end
    st_shift : begin rshft_signed_acc=1 ;
          rshft_mplr=1 ; decr_counter=1 ; end
    st_done : done = 1 ; default : ;
```

```
    endcase
  end
endmodule

module booth ( clk, inp_data , load_acc, load_mplr, load_mcnd,
    clear_acc, clear_mplr, clear_mplr0_old, rshft_signed_acc,
    rshft_mplr, add0_sub1, decr_counter, load_counter,
    mplr0, mplr0_old, counter_zero , result_h, result_l ) ;
  localparam p_n=8 ; localparam p_logn=3 ;
  input clk, load_acc, load_mplr, load_mcnd, clear_acc,
    clear_mplr, clear_mplr0_old, rshft_signed_acc, rshft_mplr,
    add0_sub1, decr_counter, load_counter ;
  input [ 2*p_n-1:0 ] inp_data ;
  output reg mplr0_old = 0 ; output mplr0, counter_zero ;
  output [ p_n : 0 ]  result_h; output [ p_n-1 : 0 ] result_l ;
  assign mplr0 = mplr[0];
  reg [ p_n : 0 ] acc=0 ; reg [ p_n-1 : 0 ] mcnd=0,mplr=0 ;
  wire [ p_n : 0 ] new_acc ; reg [ p_logn : 0] count = 0 ;
  assign result_h = acc ; assign result_l = mplr ;
  assign counter_zero = ~ ( | count) ;
  always @( posedge clk  )
      if ( clear_acc ) acc <= 0 ;
      else if ( load_acc ) acc <= new_acc ;
      else if ( rshft_signed_acc )
        acc <= { acc[ p_n ] , acc[ p_n : 1 ] } ;
  always @( posedge clk  )
      if ( clear_mplr ) mplr <= 0 ;
      else  if ( load_mplr ) mplr <= inp_data[7:0] ;
      else if ( rshft_mplr )
        mplr <= { acc[0] , mplr[ p_n-1 : 1 ] } ;
  always @( posedge clk  ) if ( load_mcnd ) mcnd <= inp_data[15:8] ;
  always @( posedge clk  )
      if ( clear_mplr0_old ) mplr0_old <= 1'b0 ;
      else if ( rshft_mplr ) mplr0_old <= mplr0 ;
  assign new_acc = add0_sub1 ?
        acc - {mcnd[p_n-1],mcnd} : acc + {mcnd[p_n-1],mcnd} ;
  always @(posedge clk) begin if (load_counter) count <= p_n ;
      else if (decr_counter) count<=count-1; end
endmodule

module booth_mult_top ( clk, start, inp_data, done, result_h, result_l );
  localparam p_n=8 ; localparam p_logn=3 ;
  input clk, start ; input [ 2*p_n-1 : 0] inp_data ;
```

```
  output done ; output [ p_n : 0 ] result_h ;
  output [ p_n-1 : 0 ] result_l ;
  wire load_acc, load_mplr, load_mcnd, clear_acc,
      clear_mplr, clear_mplr0_old, rshft_signed_acc, rshft_mplr,
      add0_sub1, decr_counter, load_counter,
      mplr0, mplr0_old, counter_zero  ;
  booth dp( clk, inp_data , load_acc, load_mplr, load_mcnd,
      clear_acc, clear_mplr, clear_mplr0_old,
      rshft_signed_acc, rshft_mplr,
      add0_sub1, decr_counter, load_counter,
      mplr0, mplr0_old, counter_zero , result_h, result_l ) ;
  booth_controller fsm (clk, start, counter_zero,
      mplr0, mplr0_old, load_acc, load_mplr, load_mcnd ,
      clear_acc, clear_mplr , clear_mcnd , clear_mplr0_old ,
      rshft_signed_acc, rshft_mplr, add0_sub1,
      decr_counter, load_counter, done ) ;
endmodule

//synthesis translate_off
module test ;

  // for p_n=4
  // parameter p_n=4, p_logn=2 ;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=-8 , mplr_inp=7;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=7 , mplr_inp=-8;

  //  for p_n=8
  localparam p_n=8, p_logn=3 ;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=-128 , mplr_inp=127;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=-128, mplr_inp=8'b10101010;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=127 , mplr_inp=-128;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=-128 , mplr_inp=-124;
  // wire signed [ p_n-1 : 0 ] mcnd_inp=-64 , mplr_inp=127;
  //  wire signed [ p_n-1 : 0 ] mcnd_inp=127 , mplr_inp=124;

  wire signed [ p_n-1 : 0 ] mcnd_inp , mplr_inp;

  reg clk=0, start=0 ;
  reg [ 2*p_n-1 : 0 ] inp_data={2*p_n{1'b0}} ;
  wire done ;
  wire [ p_n : 0 ] result_h ; wire [ p_n-1 : 0 ] result_l ;

  wire signed [2*p_n : 0 ] result ;
```

```
  assign mcnd_inp=inp_data[15:8] ;
  assign mplr_inp=inp_data[7:0] ;
  wire signed [2*p_n : 0] gold_result = mcnd_inp * mplr_inp ;

  booth_mult_top dut ( clk, start, inp_data,
                       done, result_h, result_l );
  assign result = {result_h[p_n : 0], result_l} ;
  always #10 clk = ~clk ;

  integer i ; reg [2*p_n-1:0] i16 ; reg error_flag = 0 ;
  initial begin
    // $dumpvars( ) ;
    error_flag = 0 ;
    // $monitor(  $time, " inp_data is=", inp_data,
    //       " result_h=",result_h, " result_l=", result_l,
    //       " mcnd=", dut.dp.mcnd ,
    //       " new_acc=" , dut.dp.new_acc ,
    //       " state=" , dut.fsm.state ) ;
    #1 ;
    for (i=0; i < 2**(2*p_n) ; i=i+1 ) begin
    // for (i=0; i < 2**(p_n) ; i=i+1 ) begin
      i16=i ; // i16=$random ;  // i16 = i ;
      inp_data = i16 ;
      @(posedge clk) ; #2 start = 1 ;
      @(posedge clk ); #2 start = 0 ;
      @(done) ; @(posedge clk ) ;
      // $display("result is %d", result ) ;
      // $display("gold result is %d", gold_result  ) ;
      if ( gold_result != result ) error_flag = 1 ;
    end
    if ( error_flag == 1 ) $display( "Failed !" ) ;
    else $display( "Success !" ) ;
    #100 $finish ;
  end
endmodule
//synthesis translate_on
```

4.4.2 Discussion on the Controller-FSM of Booth Multiplier

We study the datapath briefly here. The reader should draw the schematic neatly and explain in detail.

```
module booth ( clk, inp_data , load_acc, load_mplr, load_mcnd,
    clear_acc, clear_mplr, clear_mplr0_old, rshft_signed_acc,
    rshft_mplr, add0_sub1, decr_counter, load_counter,
    mplr0, mplr0_old, counter_zero , result_h, result_l ) ;
  localparam p_n=8 ; localparam p_logn=3 ;
  input clk, load_acc, load_mplr, load_mcnd, clear_acc,
    clear_mplr, clear_mplr0_old, rshft_signed_acc, rshft_mplr,
    add0_sub1, decr_counter, load_counter ;
  input [ 2*p_n-1:0 ] inp_data ;
  output reg mplr0_old = 0 ; output mplr0, counter_zero ;
  output [ p_n : 0 ]  result_h; output [ p_n-1 : 0 ] result_l ;
  assign mplr0 = mplr[0];
  reg [ p_n : 0 ] acc=0 ; reg [ p_n-1 : 0 ] mcnd=0,mplr=0 ;
  wire [ p_n : 0 ] new_acc ; reg [ p_logn : 0] count = 0 ;
  assign result_h = acc ; assign result_l = mplr ;
  assign counter_zero = ~ ( | count) ;
  always @( posedge clk  )
      if ( clear_acc ) acc <= 0 ;
      else if ( load_acc ) acc <= new_acc ;
      else if ( rshft_signed_acc )
        acc <= { acc[ p_n ] , acc[ p_n : 1 ] } ;
  always @( posedge clk  )
      if ( clear_mplr ) mplr <= 0 ;
      else  if ( load_mplr ) mplr <= inp_data[7:0] ;
      else if ( rshft_mplr )
        mplr <= { acc[0] , mplr[ p_n-1 : 1 ] } ;
  always @( posedge clk  ) if ( load_mcnd ) mcnd <= inp_data[15:8] ;
  always @( posedge clk  )
      if ( clear_mplr0_old ) mplr0_old <= 1'b0 ;
      else if ( rshft_mplr ) mplr0_old <= mplr0 ;
  assign new_acc = add0_sub1 ?
        acc - {mcnd[p_n-1],mcnd} : acc + {mcnd[p_n-1],mcnd} ;
  always @(posedge clk) begin if (load_counter) count <= p_n ;
      else if (decr_counter) count<=count-1; end
endmodule
```

What are the registers being used here ?

We have (positive-edge-triggered) register acc for purpose of accumulation of addition and subtractions (along with shifts) as needed in Booth Multiplier.

The upper half of the final result (we call it result_h) would be found there.

We use a register storage named mplr for initializing it with multiplier

that is provided as input, and then successively it is going to be shifted rightwards to make space for the lower portion of accumulator that too is being shifted right (but by propagating sign , i.e. an arithmetic right shift).

The lsb (i.e. bit 0) of register mplr is shifted into a DFF called mplr0_old. The names are indicative enough as needed for good programming practices.

The multiplicand would be loaded into register called mcnd. It does not need any shifting etc.

The accumulator register, i.e. acc, is getting loaded from the signal called new_acc, which is output of the AddOrSubtract ALU.

The ALU computes either addition or subtraction of acc and mcnd (with appropriate 9 bit arithmetic).

There is a counter to keep track of 8 steps. It is a down-counter initialized to 8 and which will decrement towards 0.

When this down-counter reaches 0, the signal counter_zero would be used for informing the controller-fsm of this situation.

The controlling inputs to this datapath (generated by the controlling FSM) are load_acc, load_mplr, load_mcnd, clear_acc , clear_mplr, clear_mplr0_old, rshift_signed_acc, rshift_mplr, add0_sub1, load_counter, decr_counter.

The status signals generated by this datapath for use as input to the controller-fsm are counter_zero (which indicates that 8 steps are over), mplr0 , mplr0_old.

The input port named inp_data of this datapath carries the values of multiplicand (in upper half) and multiplier (in the lower half).

When multiplication steps are finished, the datapath outputs result_h and result_l (which are connected to registers acc and mplr) would together represent the result.

4.4.3 Discussion on the Controller-FSM of Booth Multiplier

Now we make brief remarks on the controller-fsm here. The reader should draw the state transition diagram of this controller-fsm neatly and explain in detail.

```
always @( posedge clk ) begin
  case ( state )
    st_wait_start : if ( start ) state <= st_inp_data ;
    st_inp_data : state <= st_branch ;
    st_branch :
        if ( {mplr0,mplr0_old}==2'b01  &&  (!counter_zero) )
  state <= st_add ;
```

```
        else if ({mplr0,mplr0_old}==2'b10 && (!counter_zero))
  state <= st_sub ;
        else if ( counter_zero ) state <= st_done ;
else state <= st_shift ;
    st_shift : state <= st_branch ;
    st_add : state <= st_shift ;
    st_sub : state <= st_shift ;
    st_done : state <= st_wait_start ;
    default : state <= st_wait_start ;
  endcase
end
always @( * ) begin
  load_acc=0 ; load_mplr=0 ; load_mcnd=0 ; clear_acc=0 ;
  clear_mplr=0 ; clear_mplr0_old=0 ;
  rshft_signed_acc=0 ; rshft_mplr=0 ; add0_sub1=0 ;
  decr_counter=0 ; load_counter=0 ; done=0 ;
  case ( state )
    st_wait_start : ;
    st_inp_data :
        begin clear_acc=1 ; clear_mplr0_old=1 ;
          load_counter=1 ; load_mcnd=1 ; load_mplr=1 ;
end
    st_branch : ;  // nothing to be done on datapath
    st_add : begin load_acc=1 ; add0_sub1=0 ; end
    st_sub : begin load_acc=1 ; add0_sub1=1 ; end
    st_shift : begin rshft_signed_acc=1 ;
          rshft_mplr=1 ; decr_counter=1 ; end
    st_done : done = 1 ; default : ;
  endcase
end
```

We have adopted Moore FSM style here, and that means for every different value-combinations of the controller-outputs, we need different state.

The obviously needed inputs to this (positive-edge-clocked) controller-fsm is the clock signal clk, the signal named start to indicate the start (so that the datapath would be initialized and the data inputs would be registered).

Further, this controller-fsm would need some status information from the datapath. That is mplr0, mplr0_old, counter_zero. The reader should make sure to complete any missing information (if any).

The states are st_wait_start, st_inp_data , st_branch , st_add, st_subtract, st_shift, st_done.

Instruction	Instruction Code	Operation
ADD	$00 < addr >$	$AC \leftarrow AC + M[< addr >]$
AND	$01 < addr >$	$AC \leftarrow AC \wedge M[< addr >]$
JMP	$10 < addr >$	$goto < addr >$
INC	$11 < addr >$	$AC \leftarrow AC + 1$

Table 4.1: Instruction set for the Very Simple CPU

It is left to the reader to figure out purposes of these states and transitions among them.

The control signals are generated as Moore outputs of this FSM.

Signal named done is used for indicating that the computation is done.

4.5 Verilog Modeling of a Very Simple CPU [?]

This section is contributed by Vinay B.Y. (PhD candidate, EE, IIT Bombay)

Carpinelli describes a Very Simple CPU (vscpu) in [**?**], where the first 20 pages of Chapter 6 discusses the specifications followed by structural implementation and testing. In this section, we'll briefly state the specifications, and go on to behaviorally describe the design in Verilog.

A CPU runs a task by executing a sequence of instructions, stored in an instruction memory, that make up the task. The execution of these sequence of instructions is done, in general, as sequence of following operations: Fetch, Decode, Execute. The **Fetch** stage fetches an instruction from the instruction memory and moves on to the Decode stage. The Decode stage determines (decodes) what is to be done and moves on to the Execute stage, where the execution eventually happens, followed by going back to the Fetch stage.

4.5.1 Specification

1. The VSCPU must address 64 bytes of memory, which can be done with a 6 bit address port `addr`[5..0], and the data port `data`[7..0]

2. Has only four instructions (as below), and only one 'programmer-accessible' register, an 8-bit accumulator (AC).

3. In addition to AC, this CPU uses the following internal registers: a 6-bit Address regiter (AR); a 6-bit Program counter (PC), which contains the address to the next instruction to be fetched; An 8-bit Data register (DR) which receives instructions/data from our 64 byte memory via `data`[7..0]; and, a 2-bit Instruction register (IR) to encode the 4 instruction op-codes.

4.5.2 Behavior

Note that all 5 registers listed above must be state-elements (stored in flipflops). Our description involves a state-machine in which to run the fetch decode and execute stages so we shall need another state-element (stvar) to hold the various states.

First we reset all our 5 registers to zero. PC, which points to the instruction to fetch, hence starts at instruction number 0 at address 0, and PC is incremented during each fetch cycle to enable fetching the next instruction (or is over written with an address in case of a JMP instruction). The Fetch stage involves writing PC to AR in one cycle, enabling 'read' signal during the next clock cycle and pointing the data to be written into DR, as well as incrementing the PC register. At the same time, we copy the opcode (higher 2 bits) address (lower 6) components of this data into IR and AR respectively. The Fetch cycle, therefore, is executed spreading over three states: Fetch1, Fetch2, Fetch3. As seen from the code [citeCODElisting], in the last state, Fetch3, depending on the opcode in IR, we branch into substates corresponding to the execution stage of the 4 instructions: ADD, AND, JMP, INC. On concluding the execution of a given instruction, the state machine transitions back to Fetch1, where it fetches the next instruction as pointed to by the PC and everything follows.

4.5.3 Verilog: FSM, a coding style note

Note the suffixes _ff, identifying all state-elements that are marked for flipflop storage, and the suffix _ns marked on signals that carry next state values into the corresponding flipflops. Also note that there are two kinds of always blocks—one that nonblocking-writes to left-hand verilog-variables on a clock edge, and the other that uses blocking-writes, essentially describing a combinational logic. It is a good practice to deal with just 1 signal per always-block, or a bunch of related signals (and no others) per always-block.

4.5.4 Code Listing

The following is a code listing of three Verilog modules: vscpu_top_tb, vscpu_top, vscpu.

```
‘timescale  1ns / 1ns

// Not yet parameterized properly though
module vscpu #(
    parameter VSCPU_A_WIDTH = 6,
    parameter VSCPU_D_WIDTH = 8
)
(
    input                               clk,
    input                               reset,
    input        [VSCPU_D_WIDTH-1:0]    data,
    output   reg [VSCPU_A_WIDTH-1:0]    addr,
    output   reg                        read
);

//‘include "vscpu_localparams.vh"

    localparam [5:0] ST_FETCH1  = 6’d1;
    localparam [5:0] ST_FETCH2  = 6’d2;
    localparam [5:0] ST_FETCH3  = 6’d3;
    localparam [5:0] ST_ADD1    = 6’d4;
    localparam [5:0] ST_ADD2    = 6’d5;
    localparam [5:0] ST_AND1    = 6’d6;
    localparam [5:0] ST_AND2    = 6’d7;
    localparam [5:0] ST_JMP1    = 6’d8;
    localparam [5:0] ST_INC1    = 6’d9;

    localparam [1:0] INSTR_add = 2’d0;
    localparam [1:0] INSTR_and = 2’d1;
    localparam [1:0] INSTR_jmp = 2’d2;
    localparam [1:0] INSTR_inc = 2’d3;



//-------------------------------------------------------------
//  STYLE NOTE: The designer should mark which verilog ‘reg‘’s
//          in the design will be flipflops, latches, or just
```

```
//          plain combinational signals
//          Use suffixes:
//           _ff (indicating the signal is a flipflop's output),
//           _l,
//          (none) respectively
//          Nextstate 'plain signals' have special meaning
//          so mark them as such, say with a _ns sufix
//-------------------------------------------------------------

reg [7:0]                AC_ff;  // Accumulator
reg [VSCPU_A_WIDTH-1:0]  AR_ff;  // Address register
reg [VSCPU_A_WIDTH-1:0]  PC_ff;  // Program counter
reg [VSCPU_D_WIDTH-1:0]  DR_ff;  // Data register
reg [1:0]                IR_ff;  // Instruction register

reg [7:0]                AC_ns;  // Accumulator
reg [VSCPU_A_WIDTH-1:0]  AR_ns;  // Address register
reg [VSCPU_A_WIDTH-1:0]  PC_ns;  // Program counter
reg [VSCPU_D_WIDTH-1:0]  DR_ns;  // Data register
reg [1:0]                IR_ns;  // Instruction register


reg [3:0]                stvar_ff;
reg [3:0]                stvar_ns;


initial
begin
    $monitor(
    "%g pc=%x cstate=%x ac=%x ir=%x dr=%x ar=%x data=%x",
        $time, PC_ff, stvar_ff, AC_ff, IR_ff, DR_ff, AR_ff, data);
end

always@(*) addr <= AR_ff;


//-------------------------------------------------------------
//  State updates
//-------------------------------------------------------------
//  STYLE NOTE: It is a recommended practice to deal with
//          1-signal-1-always-block
```

```
//          OR a bunch of signals handled neatly without
//          clutter
//-------------------------------------------------------------
always@(posedge clk)
begin
    if(reset == 1'b1)
        stvar_ff <= ST_FETCH1;
    else
        stvar_ff <= stvar_ns;
end

always@(posedge clk)
begin
    if(reset == 1'b1)
    begin
        AC_ff <= 0;
        PC_ff <= 0;
        AR_ff <= 0;
        IR_ff <= 0;
        DR_ff <= 0;
    end
    else
    begin
        AC_ff <= AC_ns;
        PC_ff <= PC_ns;
        AR_ff <= AR_ns;
        IR_ff <= IR_ns;
        DR_ff <= DR_ns;
    end
end

// Next state logic
always@(*)
begin
    stvar_ns = stvar_ff;

    case(stvar_ff)
        ST_FETCH1:
            stvar_ns = ST_FETCH2;
        ST_FETCH2:
            stvar_ns = ST_FETCH3;
```

```
        ST_FETCH3:
        begin
            case(IR_ff)
                INSTR_add:
                    stvar_ns = ST_ADD1;
                INSTR_and:
                    stvar_ns = ST_AND1;
                INSTR_jmp:
                    stvar_ns = ST_JMP1;
                INSTR_inc:
                    stvar_ns = ST_INC1;
            endcase
        end
        ST_ADD1:
            stvar_ns = ST_ADD2;
        ST_AND1:
            stvar_ns = ST_AND2;
        ST_JMP1:stvar_ns = ST_FETCH1;
        ST_INC1:stvar_ns = ST_FETCH1;
        ST_ADD2:stvar_ns = ST_FETCH1;
        ST_AND2:stvar_ns = ST_FETCH1;

    endcase
end


always@(*)
begin
    if((stvar_ff==ST_FETCH2) ||
        (stvar_ff==ST_ADD1)  ||
        (stvar_ff==ST_AND1))
    begin
        read <= 1;
    end else begin
        read <= 0;
    end

end

always@(*)
begin
```

```
    AR_ns = AR_ff;
    PC_ns = PC_ff;
    DR_ns = DR_ff;
    IR_ns = IR_ff;
    AC_ns = AC_ff;
    case (stvar_ff)
        ST_FETCH1:
            AR_ns   = PC_ff;
        ST_FETCH2:
        begin
            PC_ns   = PC_ff + 1;
            DR_ns   = data;
            // NOTE: there is no fall-through in Verilog,
            // like in C
            // Copy the two higher order bits of DR to IR
            // Copy the 6 remaining lower order bits of AR
            IR_ns   = DR_ns[VSCPU_D_WIDTH-1:VSCPU_D_WIDTH-2];
            AR_ns   = DR_ns[VSCPU_D_WIDTH-3:0];
        end
        ST_FETCH3:;
        ST_ADD1:
        begin
            DR_ns   = data;
        end
        ST_ADD2:
        begin
            AC_ns   = AC_ff + DR_ff;
        end
        ST_AND1:
        begin
            DR_ns   = data;
        end
        ST_AND2:
            AC_ns   = AC_ff & DR_ff;
        ST_JMP1:
            PC_ns   = DR_ff[VSCPU_D_WIDTH-3:0];
        ST_INC1:
            AC_ns   = AC_ff + 1;

    endcase
end
```

```
//For convenience examining the 'monitor' output
always@(posedge clk)
begin
    if(stvar_ns==ST_FETCH1)
        $display(" ----o---o---o----");
end

endmodule



`timescale  1ns / 1ns

`define A_WIDTH 6
`define D_WIDTH 8

module vscpu_top #(
    parameter N = 64
)
(
    input                           clk,
    input                           reset,
    input                           start,
    input       [7:0]               prog
);

`include "vscpu_localparams.vh"

reg [7:0] mem [0:N-1];

integer i;

initial
begin
    mem[0] = {INSTR_add, 2'b0, 4'b0100};
    mem[1] = {INSTR_and, 2'b0, 4'b0101};
    mem[2] = {INSTR_inc, 2'b0, 4'b0000};
    mem[3] = {INSTR_jmp, 2'b0, 4'b0000};
```

```
    mem[4] = 8'h27;
    mem[5] = 8'h39;
    for(i = 6; i<N; i=i+1)
        mem[i] = 0;
end

wire  [7:0] memout;
wire  [5:0] memaddr;
wire read;

assign memout = read?mem[memaddr]:0;
vscpu vscpu(.clk(clk), .reset(reset), .data(memout),
          .addr(memaddr), .read(read));


endmodule


`timescale  1ns / 1ns

`define A_WIDTH 6
`define D_WIDTH 8

module vscpu_top_tb();
reg clk;
reg reset;

wire start;

wire [7:0] prog;

always begin
    #1 clk <= ~clk;
end

initial
begin
    clk = 0;
    reset = 0;
    #2
    reset = 1;
```

```
    #2;
    reset = 0;
    #4;
end
initial
begin
    #100 $finish;
end


vscpu_top vscpu_top(.clk(clk), .reset(reset),
        .prog(prog), .start(start));

endmodule
```

Chapter 5

References

1 Patterson, Hennessy, “Computer Organization and ..... ” , Morgan Kaufman

2 Navabi Z., “Digital PLD ..... ”. *****

3 Harris D. “ Structural Verilog .... ” URL .....

4 Peckol , “ Tutorial FSM .... ”,

5 Tocci, Widmer, Moss, “ Digital Systems : Principles and Practices ”, Pearson

6 “Very Simple CPU”, Chapter 6 of John Carpinelli’s book:: Computer Systems: Organization & Architecture

7 Wassel, “Digital Logic .... ”, slides available from URL *****

5.1 Simulation Examples of FSMs

Traffic Light Controller

```
 module TLC (clk, rst_b, MRsensor, SRsensor, MRlight, SRlight);
    input clk, rst_b, MRsensor, SRsensor;
    output MRlight, SRlight ;
    reg current_state, next_state;
    parameter MRgreen = 1'b1;
    parameter SRgreen = 1'b0;
    assign MRlight = ~ current_state;
    assign SRlight = current_state;
    always @( * )
      case (current_state)
        SRgreen : next_state <= SRsensor ? SRgreen : MRgreen ;
        MRgreen : next_state <= MRsensor ? MRgreen : SRgreen ;
      endcase
    always @(negedge clk, negedge rst_b)
      if ( rst_b == 0 )
        current_state <= 0;
        // initial ( that is, "on-reset" ) state
      else
        current_state <= next_state;
endmodule
```

Test Bench for Traffic Light Controll

```
‘timescale 1ns/1ps

module tb_tlc;
  reg     clk, rst_b, MRsensor, SRsensor;
  wire    MRlight, SRlight;

  TLC m1(clk, rst_b, MRsensor, SRsensor, MRlight, SRlight);

  initial begin
    $dumpfile("tlc.vcd");
    $dumpvars;
```

```
    rst_b = 0;  clk = 1; MRsensor = 0; SRsensor = 0;
    #10    rst_b = 1;
    #10    MRsensor = 0; SRsensor = 1;
    #20    MRsensor = 1; SRsensor = 0;
    #20 MRsensor = 1; SRsensor = 1;
    #10    MRsensor = 0; SRsensor = 1;
    #30    MRsensor = 0; SRsensor = 0;
    #5    MRsensor = 1; SRsensor = 0;
    #10 $finish;
  end

  always
    #5    clk = ~clk;

endmodule
```

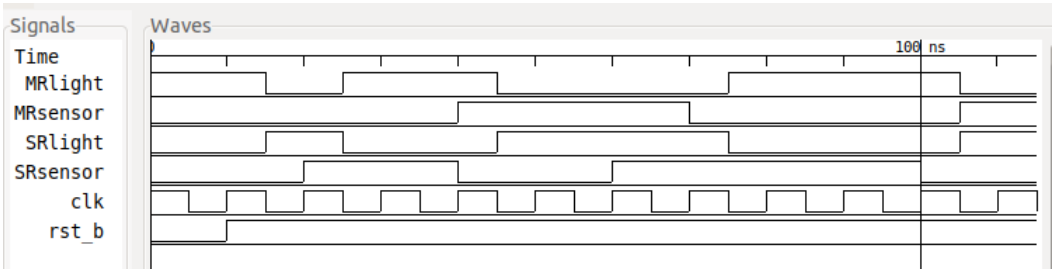


Figure 5.1: Output Waveform for Traffic Light Controller

Sequence Detector

```
module moore_101_detector (w, rst, clk, y );
    input w, rst, clk;
    output y; reg y;
    parameter [1:0] stRESET = 0, stGOT1 = 1,
                    stGOT10 = 2, stGOT101 = 3;
    reg [1:0] state;
    always @ ( posedge clk ) begin
        if (rst) state <= stRESET;
        else begin
```

```
            case ( state )
                stRESET: begin
                    if ( w==1'b1 ) state = stGOT1 ;
                    else state = stRESET;
                end
                stGOT1: begin
                    if ( w==1'b0 ) state = stGOT10;
                    else state = stGOT1;
                end
                stGOT10: begin
                    if ( w==1'b1 ) state = stGOT101;
                    else state = stRESET;
                end
                stGOT101: begin
                    if ( w==1'b1 ) state = stGOT1;
                    else state = stGOT10;
                end
                default: state = stRESET;
                // Only for illegal input ( ?? )
                // Legal inputs would not bring FSM here.
            endcase
        end
    end

    // Moore output logic
    // ( combinational function of just "state" )

    always @( * ) begin
        case ( state )
            stRESET : y <= 0;
            stGOT1 : y <= 0;
            stGOT10 : y <= 0;
            stGOT101 : y <= 1;
            default : y <= 0;
        endcase
    end
endmodule
```

Test Bench for Sequence Detector

```
‘timescale 1ns/1ps
module tb_moore_101_detector;

  reg     clk, rst, w;
  wire    y;

  moore_101_detector m1(w, rst, clk, y );

  initial begin
    $dumpfile("seq_dut.vcd");
    $dumpvars;
    rst = 1;    clk = 1;    w = 0;
    #10    rst = 0;
    #10    w = 1;
    #10    w = 1;
    #10    w = 0;
    #10    w = 1;
    #10    w = 0;
    #10    w = 1;
    #10    w = 1;
    #10    w = 0;
    #10    w = 1;
    #10    w = 0;
    #10    w = 1;
    #10    w = 1;
    #10    w = 0;
//$display("Value = %b",y);
#10 $finish;

end


always
    #5    clk = ~clk;


endmodule
```

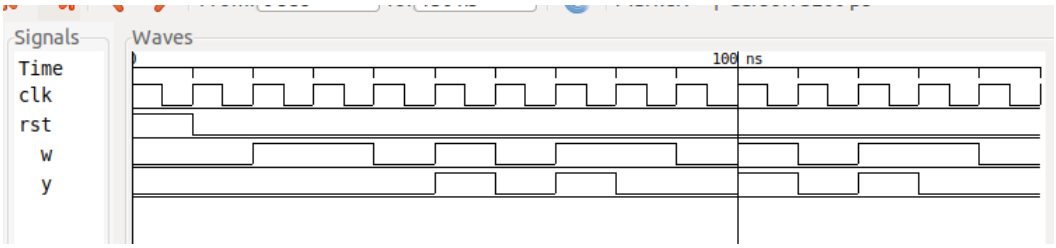


Figure 5.2: Output Waveform for Sequence Detector

Washing Machine Controller

```
module washing_machine_toy_model
    ( clk, start, full, timesup, dry,
    water_valve, ag_mode, sp_mode );

    input clk, start, full, timesup, dry;
    output reg water_valve, ag_mode, sp_mode;

    parameter     stIdle=2'b00, stFill=2'b01,
                stAgitate=2'b10, stSpin=2'b11;

    reg [1:0] state, next_state;

    always @( posedge clk ) begin
        state <= next_state;
    end // always

    always @( * ) begin
        case ( state )
            stIdle : begin
                if ( start ) next_state <= stFill;
                else next_state <= stIdle;
            end
            stFill : begin
                if ( full ) next_state <= stAgitate;
```

```
                else next_state <= stFill;
            end
            stAgitate : begin
                if ( timesup ) next_state <= stSpin;
                else next_state <= stAgitate;
            end
            stSpin : begin
                if ( dry ) next_state <= stIdle;
                else next_state <= stSpin;
            end
            default : next_state <= stIdle;
        endcase
    end // always

    always @( * ) begin
        case ( state )
            stIdle : begin
                water_valve <= 0; ag_mode <= 0; sp_mode <= 0;
            end
            stFill : begin
                water_valve <= 1; ag_mode <= 0; sp_mode <= 0;
            end
            stAgitate : begin
                water_valve <= 0; ag_mode <= 1; sp_mode <= 0;
            end
            stSpin : begin
                water_valve <= 0; ag_mode <= 0; sp_mode <= 1;
            end
            default : begin
                water_valve <= 0; ag_mode <= 0; sp_mode <= 0;
            end
        endcase
    end // always
endmodule
```

Test Bench for Washing Machine Controller

```
‘timescale 1ns/1ps
```

```
module tb_washing_machine_toy_model;

    reg clk, start, full, timesup, dry;
    wire water_valve, ag_mode, sp_mode;

washing_machine_toy_model m1( clk, start, full, timesup, dry,
    water_valve, ag_mode, sp_mode );

initial
begin
$dumpfile("washing_machine_toy_model.vcd");
$dumpvars;
    start = 0; clk = 0;  full = 0;    timesup = 0;    dry = 0;
    #15    start = 1;    full = 0;    timesup = 0;    dry = 0;
    #10    start = 0;    full = 0;    timesup = 0;    dry = 0;
    #10    start = 0;    full = 1;    timesup = 0;    dry = 0;
    #20    start = 0;    full = 0;    timesup = 1;    dry = 0;
    #20    start = 0;    full = 0;    timesup = 0;    dry = 1;
    #20    start = 1;    full = 0;    timesup = 0;    dry = 0;
#10 $finish;
end

always
    #5    clk = ~clk;

endmodule
```

Combinational Lock

```
module ComboLock0(open, state, comboIn, clk, por);
    // declare the inputs and outputs

    input clk, por;
    input [1:0] comboIn;
    output open;
```

Figure 5.3: Output Waveform for Washing Machine Controller

```
output [1:0] state;

reg open;
reg [1:0] state;

//define parameter or name of each state
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b11, s3 = 2'b10;
parameter true = 1'b1, false = 1'b0;
parameter zero = 2'b00, one = 2'b01;
parameter two = 2'b10, three = 2'b11;

// build the combination lock
always @(negedge por or posedge clk) begin
    // reset the lock
    if (por==0) begin
        state <= s0; open <= false;
    end
    // implement the combination lock
    else begin
        case (state)
            // initial state 0 correct combination xxxx
            s0:
                begin
                    open <= false;
                    if (comboIn == three)
                        state <= s1;
                    else
                        state <= s0;
                end
```

```
                // one correct combination xxx3
                s1:
                    begin
                        if (comboIn == two)
                            state <= s2;
                        else
                            state <= s0;
                            open <= false;
                        end
                // two correct combination xx23
                s2:
                    begin
                        if(comboIn == two)
                            state <= s3;
                        else
                            state <= s0;
                            open <= false;
                        end
                endcase
        end
    end // always
endmodule
```

Test Bench for Combinational Lock

```
 ‘timescale 1ns/1ps

module tb_ComboLock0;

    reg clk, por;
    reg [1:0] comboIn;
    wire open;
    wire [1:0] state;

    parameter two = 2’b10, three = 2’b11;

ComboLock0    m1(open, state, comboIn, clk, por);
```

```
initial
begin
$dumpfile("ComboLock0.vcd");
$dumpvars;
    por = 0;    clk = 1;
    comboIn = 2'b0;
    #10    por = 1;
    #10    comboIn = two;
    #10    comboIn = three;
    #10    comboIn = two;
    #10    comboIn = two;
    #10    comboIn = two;
    #10    comboIn = three;
    #10    comboIn = two;
    #10    comboIn = two;
    #10    comboIn = three;
    #10    comboIn = three;
#10 $finish;
end

always
    #5    clk = ~clk;

endmodule
```

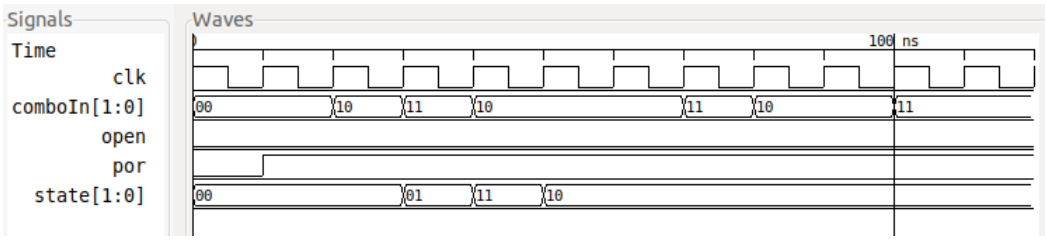


Figure 5.4: Output Waveform for Combinational Lock

5.2 Exercise Set

1. Three dice are rolled simultaneously. For each die, let a binary signal represent whether the outcome is even or odd. Design a circuit whose output represents whether the sum of the outcomes of the three dice is even or odd.

2. A cricket tournament is played with the following rule. If a match ends in a tie, the team batting second will be declared the winner iff it has a higher run rate. If it doesn't, then the team batting first is declared the winner iff it has lost fewer wickets. Otherwise, the match ends in a draw. Design a circuit whose output represents whether the team batting first has won, whether the team batting second has won, or the match has ended in a draw.

3. A window must remain open when the fire alarm is ON. In absence of a fire alarm, it must remain closed if the AC is on. Otherwise, the window must follow the open/closed switch, except if the theft alarm is ON, in which case the window must remain closed. Draw a circuit to control the state of the window.

4. Ram likes sandwich and ice-cream. Mohan likes sweet corn and ice-cream. Anjali likes sandwich, pastry and sweet corn. They have enough money to buy two items. Draw the logical circuit which can determine, whether a choice of two items would contain at-least one food of liking for each child. (Assign 0-1 values to each food item).

5. Draw a circuit to determine whether a 3 bit binary number is an exact power of 2.

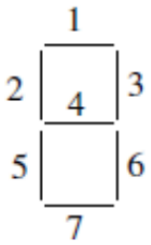
6. Gopal can go to the stadium if it does not rain and his brother comes home before 5:00 pm. He can play a video game at home if his father will not return by 7:00 pm. If it does not rain, his father will attend a meeting and reach home after 8:00pm. Draw a boolean circuit whose output represent the actions which Gopal can take.

7. A die is rolled multiple times. Design a circuit to keep track of whether the sum of the outcomes so far is even or odd.

8. In a mobile phone, a two bit signal represents whether the profile is "General", "Outdoors", "Silent" or "Meeting". Ringtone is ON in the "General" and "Outdoors" profiles and OFF in other profiles. Vibrate is ON in all profiles except in the "General" profile. The mobile has a

"Power Save" mode, in which, vibrate is turned OFF when ringtone is ON and battery is LOW, irrespective of the profile settings. Draw a circuit to determine whether the mobile will vibrate on receiving a call. Extend the circuit, so that every profile change by the user, overrides the power save setting and every power save setting change overrides the profile setting wrt vibration.

9. A 7 segment LED display is shown in the figure. You are to design a system which takes a 3 bit binary number as input, and display 'H' (for high) if the number is < 4 and 'L' (for low) otherwise. Write boolean equations or draw the circuit for to compute the on/off state of each of the 7 segments.



LED Digit Display

Output if num >= 4

Output if num < 4

10. A room in a smart home has a cooler and a fan, each having its own switch. The fan is controlled by an additional logic. If the cooler is ON and the temperature is below 30 degrees, the fan will remain off. Design a circuit to control the state of the fan. Next task is to extend the functionality, so that the switch provided to the user must behave as as toggle switch. That is, the user can toggle the state of the FAN using the switch, irrespective of other settings in effect. After a user has toggled the state of the FAN (switched it ON or OFF), next change in the state should happen only on change in one of the controlling parameters.

11. Note that in above circuit, there can be rapid fluctuations in the fan's state if the temperature is very close to 30 degrees and fluctuates often. Extend the above circuit so that the state of the fan is affected by temperature only if there is atleast 5 degree change in temperature from the time the last state was set. For examples, if the user turns the fan OFF at 32 degrees, fan should not turn ON automatically before temperature reaches 37 degrees.

12. A teacher has to form a team for an inter-school quiz competition. Alice is good at Maths, Economics and History. Bob is good at Physics and Maths. Charles is good at English and Physics. David is good at Economics and Physics. Design a logical circuit, whose output represents whether a subset of these students will be good at all of the subjects – Maths, Economics, History, Physics and English. (Use binary variables to represent whether a student is part of the team or not). Next we wish to design a RAM to represent a general set cover problem. Using an MxN RAM, let the j'th bit in the i'th row of the RAM denote whether student j is good in subject i. Design a circuit which can nd a minimal combination of students which will know all the subjects. Your circuit can output only combination in case of multiple optimal solutions

13. Implement

$$f(v,w,x,y,z) = \sum_{v,w,x,y,x} m(0,4,7,15,18,19,20,21,22,23,24,25,26,27,28,29,30,31) \tag{5.1}$$

by using a 4-to-1 multiplexer and as few 2-input gates (any type of gate allowed) as possible. Use single-rail inputs (that is only uncomplemented inputs) alone. The output of this circuit should be at the output of the multiplexer.

14. Design a combinational circuit that converts a 3-bit Gray code to Binary code.

15. Design a combinational circuit that converts a 3-bit Binary code to Gray code.

16. Design a Half and a Full–subtractors in a manner analogous to half and full adders.

17. Design a 4-bit ripple-borrow subtractor in a manner analogous to a 4-bit ripple- carry-adder.

18. Johnson counters are also called twisted-tail counter because they feed complemented version of the last flip-flop as input to the first flipflop. What are the merits of such a counter built with 4 D-flipflops, in comparison to a ring counter and also as compared to a synchronous binary 3-bit counter.

19. Design a two-level AND-OR implementation, with minimum number of AND gates, for "majority" circuit with five inputs and a single output. Assume that inverters are available for complementing some input signals before feeding as inputs to the AND gates.

20. We want to have a parameterizable approach for design of a logic circuit that has n inputs and n+1 outputs. The n+1 outputs should be one-hot and indicate how many of the n input signals are asserted. Describe an inductive approach of designing a bigger such circuit in terms of smaller such circuit. You are permitted to use simple and small extra logic.

21. Show that the following mirror property holds for the outputs, SUM and COUT of a full-adder, as a function of its inputs A, B, CIN. By inverting the inputs, the outputs are inverted.

22. Design a four-bit combinational circuit 2's complementer. (The output generates the 2's complement of the input binary number.) Show that the circuit can be constructed with exclusive-OR gates. Can you predict what the output functions are for a five-bit 2's complementer?

23. Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to-4-line decoder. Use block diagrams for the components.

24. Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable.

25. Implement a full adder using a pair of 4 x 1 multiplexers.

26. Construct a 16-line-to-1-line multiplexer with three smaller (possibly diffeent sized) multiplexers.

27. Construct a 4-to-16 decoder with smaller decoders.

28. A variation of a positive-edge-triggered flip-flop circuit has two inputs S and R. It is specified that the output Q is updated to 1, if the value of S input at the positive-edge of the triggering clock is seen to be at level 1, otherwise the output Q is updated to 0. Model this behaviour in Verilog HDL.

29. Design using Verilog HDL, a 4-bit synchronous Gray counter.

30. Show that a D-flipflop can be implemented using a single JK-flipflop, with possibly additional combinational logic gates.

31. Show that a JK-flipflop can be implemented using a single D-flipflop, with possibly additional combinational logic gates.

32. Show that a T-flipflop can be implemented using a single D-flipflop, with possibly additional combinational logic gates.

33. Show that a D-flipflop can be implemented using a single T-flipflop, with possibly additional combinational logic gates.

34. Model a 74-series IC for a 4-bit decoder using Verilog HDL.

35. Model some more useful 74-series ICs using Verilog HDL.

36. Design a circuit to detect an overflow when a pair of 2's complement format binary numbers are added.

37. When an n-bit 2's complement signed integer is promoted to an m-bit 2's complement representation (where m ¿ n), then you may observe a simple sign- extension phenomenon. Explain with rigorous justification.

38. Design a sequential circuit that compares pairs of bits of the two input sequences on its two inputs, w1 and w2. If, for 3 consecutive cycles, these pairs are found to be equal, then the output z of this circuit should be asserted HIGH, otherwise it is asserted LOW.

39. Design a digital logic circuit that can be used as a frequency counter

40. Design a clock divider circuit which produces a reduced frequency clock signal of specified duty cycle.

41. ROM based frequency generator.

42. Design using Verilog HDL a reversible (up/down) synchronous 4-bit binary counter.

43. Compare a ripple counter and a synchronous binary counter in terms of respective power consumption due to 0-1 and 1-0 switching at the gate inputs and outputs.

44. Design using Verilog HDL a reversible (up/down) synchronous 4-bit binary counter.

45. Compare a ripple counter and a synchronous binary counter in terms of respective power consumption due to 0-1 and 1-0 switching at the gate inputs and outputs.

46. Design a serial adder with three shift registers and a single flipflop. The pair of binary numbers are shifted into the two shift registers and the result is shifted into a third shift register. Additional (minimal) combinational logic is permitted.

47. Illustrate through examples how to systematically map a 2-level or a multilevel AND-OR networks to a NAND-only or a NOR-only logic networks.

48. Illustrate cause of static hazards, and an approach to mitigate these. Assume single-input-change (SIC), that is, at any given instant only one of the input bits make a transition and there is sufficient duration between successive changes at any input bits.

49. Explain the notion of inertial delay, and describe how such delays can be modeled in Verilog HDL.

50. Explain the notions of setup-time and hold-time. Describe the Verilog HDL support for modeling setup and hold times.

51. Illustrate how cmos-switch-level logic circuits can be modeled using Verilog HDL.

52. Prepare a few tutorial examples for explaining how to write test-benches in Verilog HDL.

53. Implement a 8-bit Manchester Carry Chain Adder

54. Implement a 8-bit Conditional Sum Adder

55. Implement a two player Pong game on a CPLD board using the Verilog HDL. Also, Interface the game with the user using:

    - Push buttons as input to move the paddles left and right
    - A VGA monitor as output, showing the ball, paddles, background and number of points scored on the screen.

56. The Serial Peripheral Interface or SPI-bus is a simple 4-wire serial communications interface used by many microprocessor and microcontroller peripheral chips that enables the controllers and peripheral devices to communicate each other.
    Interface a ADC MCP 3008 to the Krypton CPLD through the Serial to Peripheral Interface (SPI).

57. A graphical LCD JHD12864E is a 128x64(8192) pixel dot matrix LCD. Interface the GLCD JHD12864E to the Krypton CPLD

58. Generate the seven musical notes of Indian Classical Music Saptak (Sa Re Ga Ma....), using Krypton. To listen to this melodious tune, we need to interface a speaker with Krypton. Each note needs to last for 0.5 seconds.

59. Implement a KoggeStone adder

60. Implement a Linear feedback shift register

61. Implement a Ripple Carry adder

62. Implement a Unsigned multiplier

63. Implement a Baugh-Wooley multiplier

64. Implement a Wallace tree

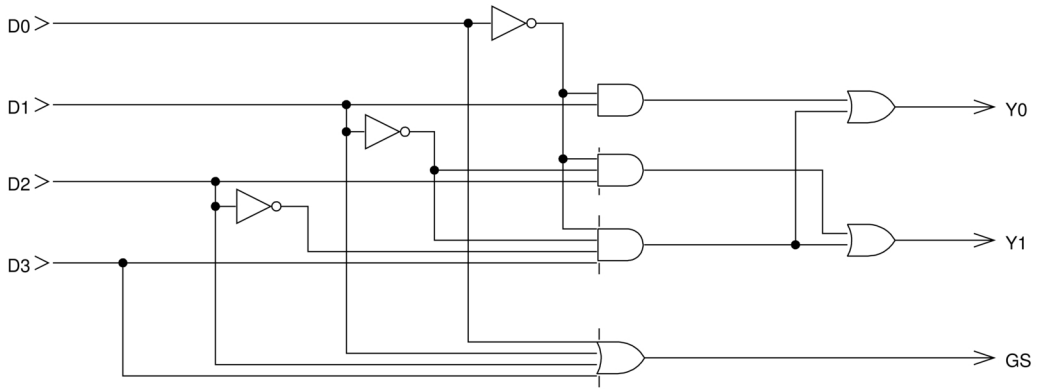
65. Implement a digital logic circuit in which the data is considered at every clock cycle. If the data is not changed from previous clock cycle, it means that the previous data itself has arrived again and the counter continues to count

66. Interface a ADC 0804 with the Krypton board

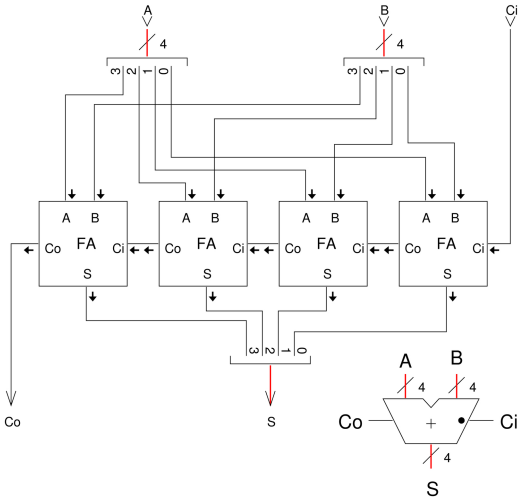
67. Interface a Stepper motor to the Krypton CPLD board with the following design conditions

    - Actuate the stepper motor,so that the wheel attached to the rotor rotates in steps of x degrees.
    - Control the direction of rotation using a switch.
    - Control the speed of rotation using a switch.

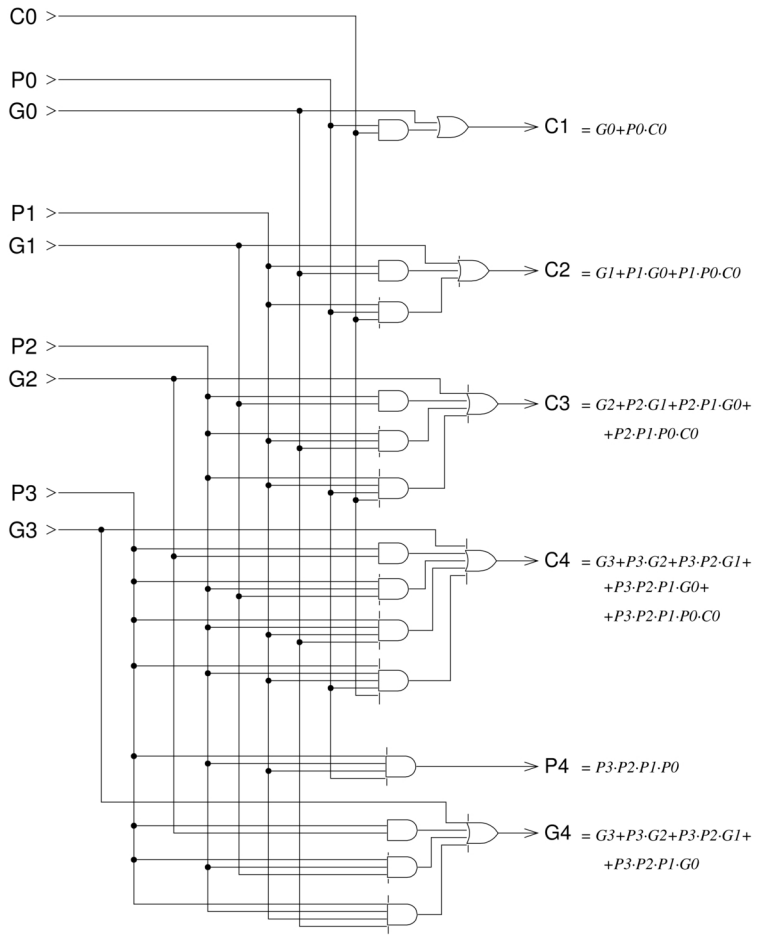
68. Write the Verilog description of the following design (Daniele Giacomini)



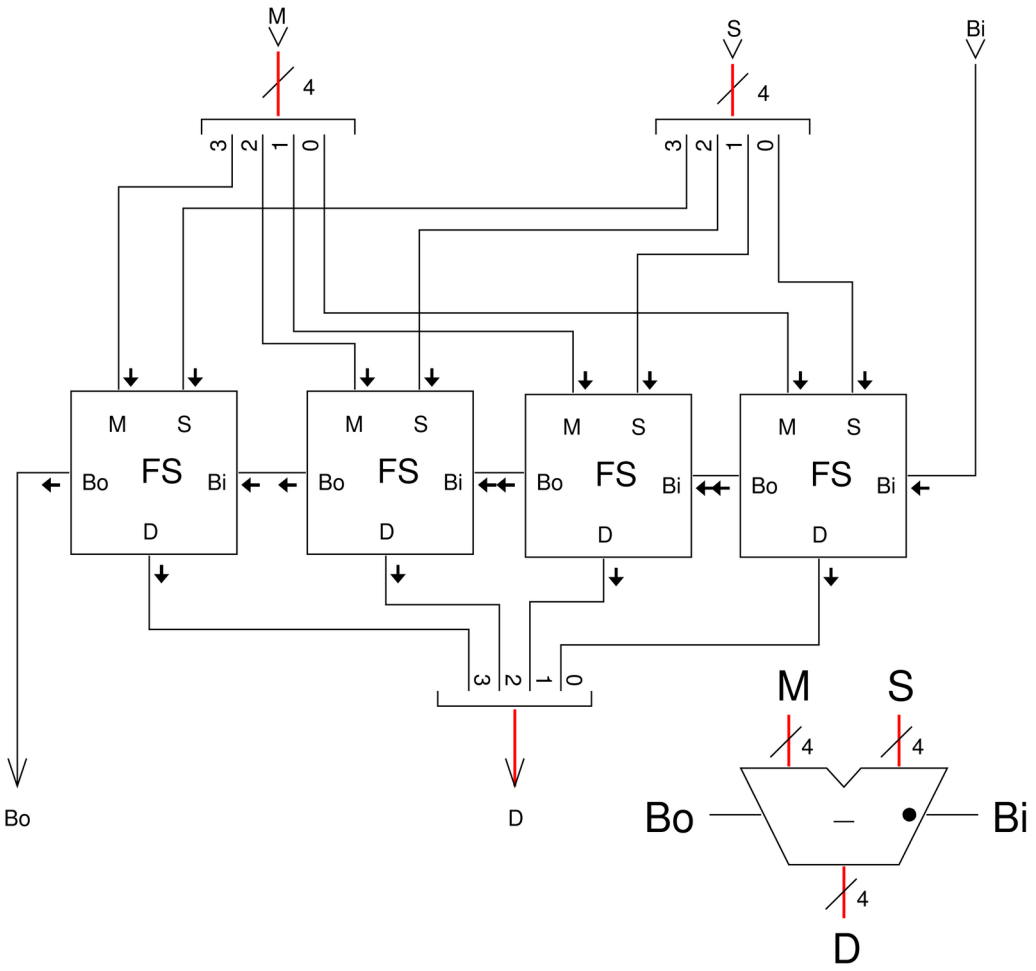
69. Write the Verilog description of the following design (Daniele Giacomini)



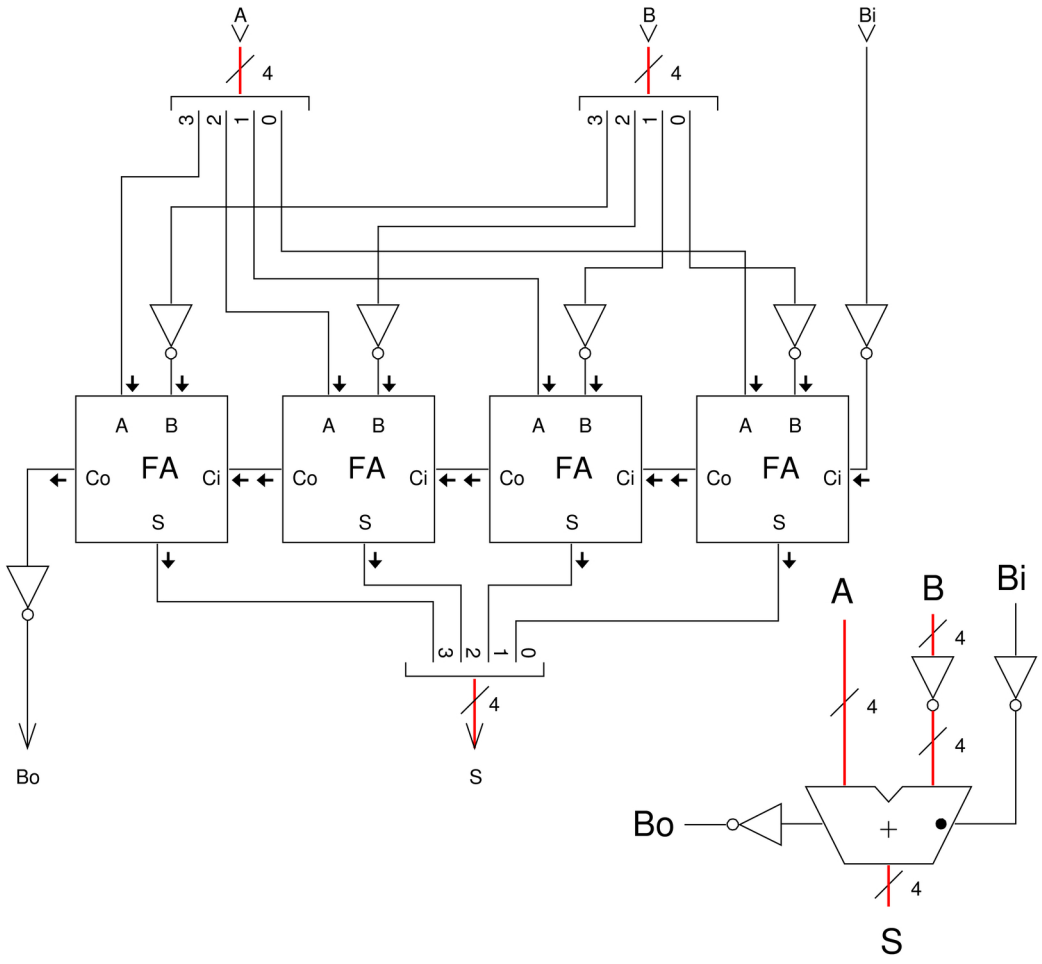
70. Write the Verilog description of the following design (Daniele Giacomini)



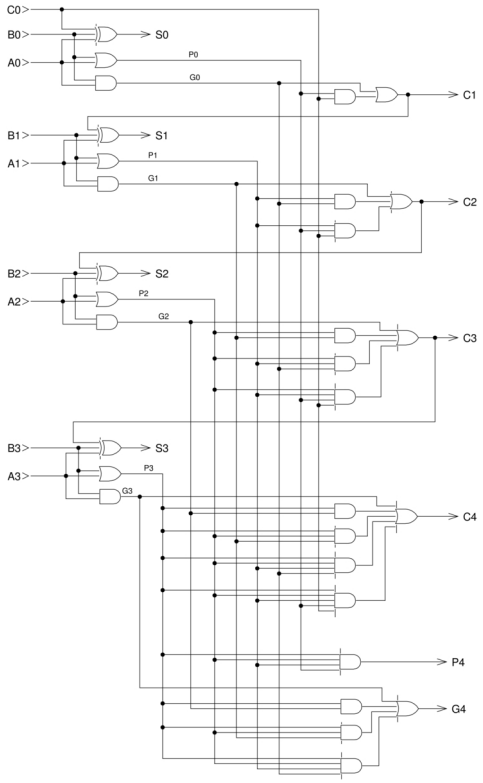
71. Write the Verilog description of the following design (Daniele Giacomini)



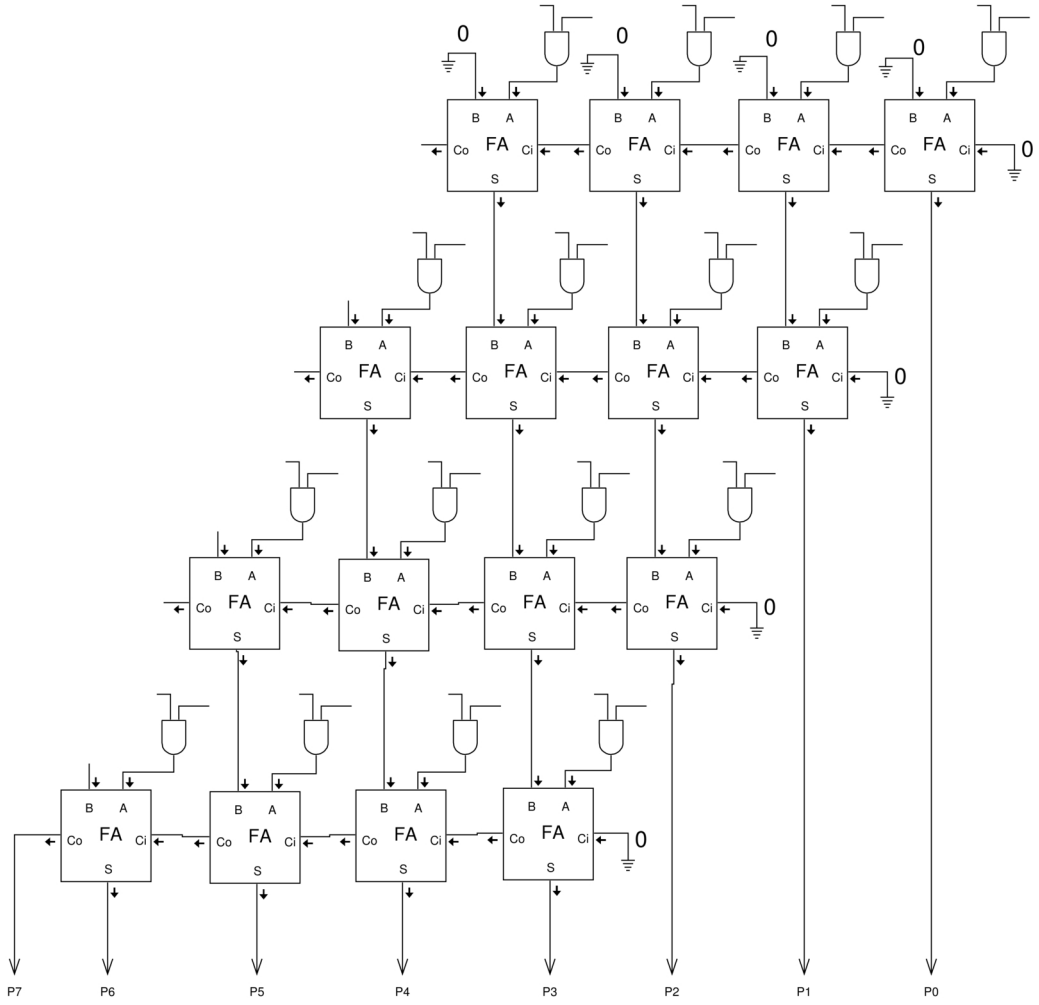
72. Write the Verilog description of the following design (Daniele Giacomini)



73. Write the Verilog description of the following design (Daniele Giacomini)



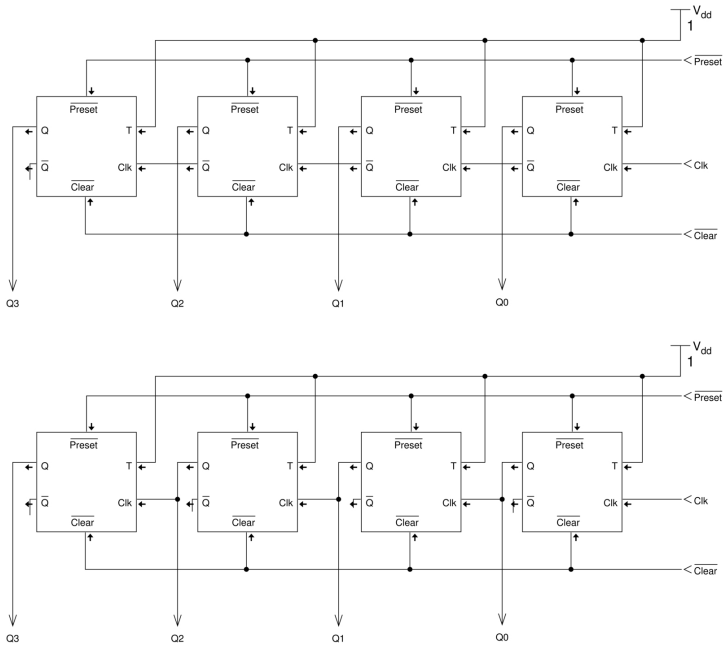
74. Write the Verilog description of the following design (Daniele Giacomini)



75. Write the Verilog description of the following design (Daniele Giacomini)

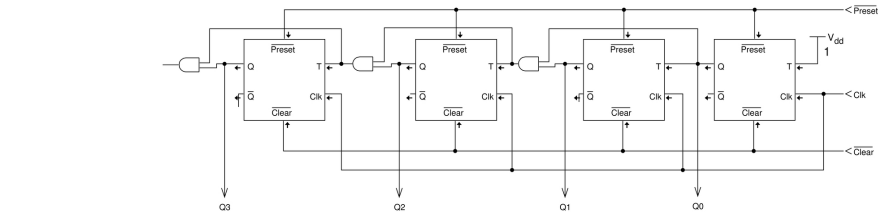
76. Write the Verilog description of the following design (Daniele Giacomini)

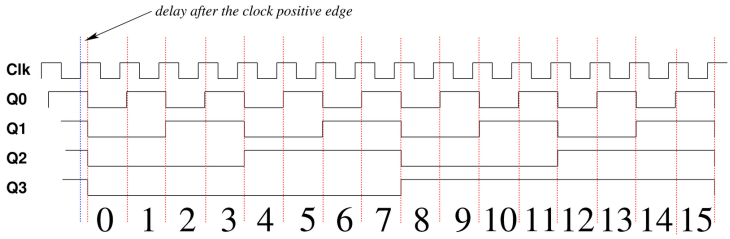
77. Write the Verilog description of the following design (Daniele Giacomini)

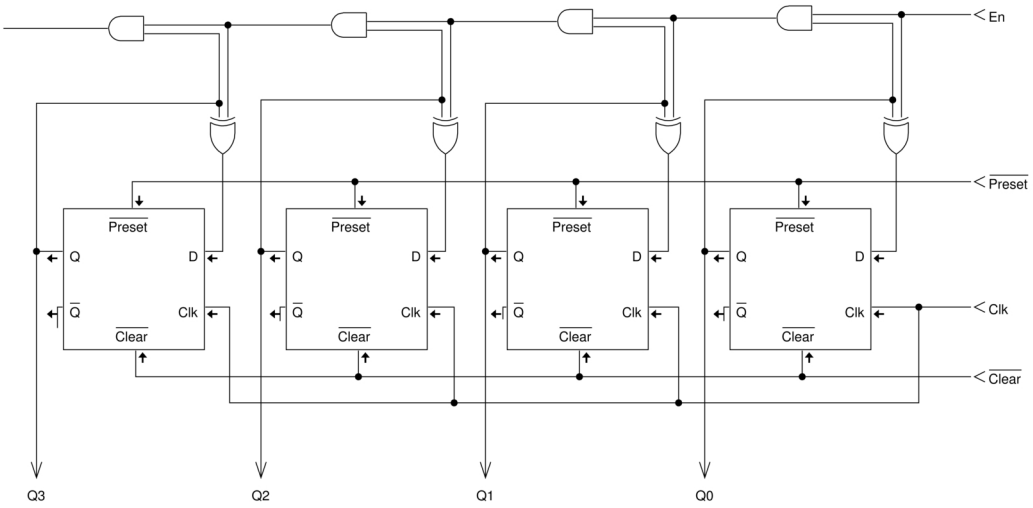


78. Write the Verilog description of the following design (Daniele Giacomini)

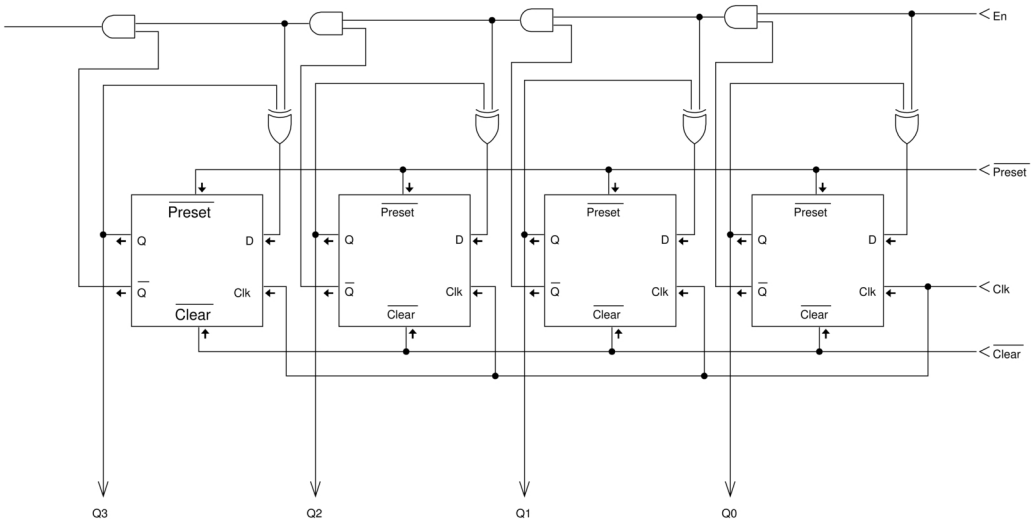
79. Write the Verilog description of the following design (Daniele Giacomini)



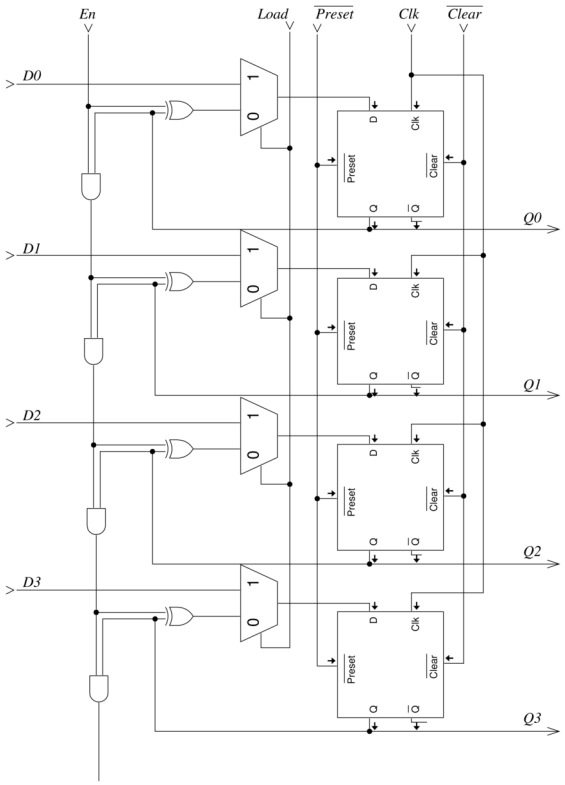




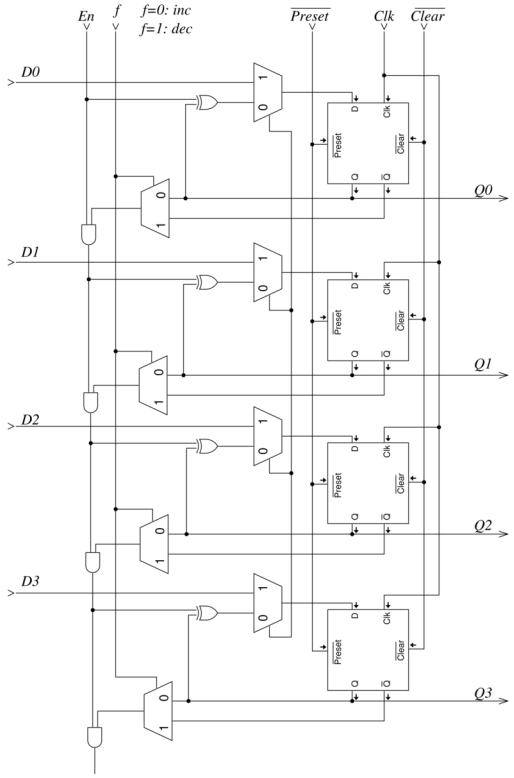
80. Write the Verilog description of the following design (Daniele Giacomini)



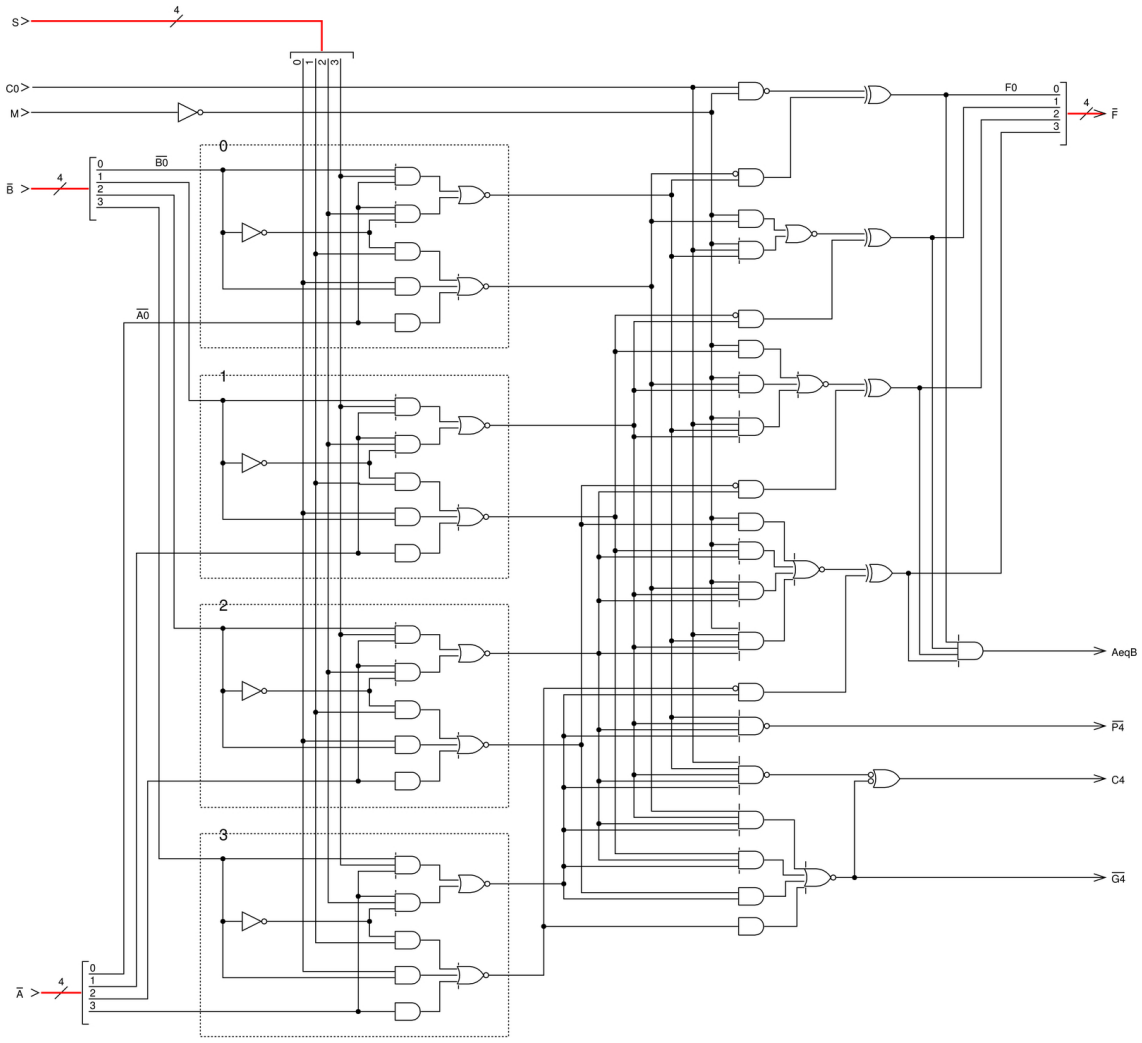
81. Write the Verilog description of the following design (Daniele Giacomini)



82. Write the Verilog description of the following design (Daniele Giacomini)



83. Write the Verilog description of the following design (Daniele Giacomini)



84. Write the Verilog description of the following design (Daniele Giacomini)

85. Design the tkGate implementation for the following Verilog code

```
 module priority_encoder_4_to_2 (Y, GS, D);
  input [3:0] D;
  output [1:0] Y;
  output GS;
  //
  function [1:0] f4to2 (input [3:0] D);
    if (D & 1)
      f4to2 = 0;
    else if (D & 2)
      f4to2 = 1;
    else if (D & 4)
      f4to2 = 2;
    else if (D & 8)
      f4to2 = 3;
    else
      f4to2 = 0;
  endfunction
  //
  assign #8  Y = f4to2 (D);
  assign #8 GS = D[0] | D[1] | D[2] | D[3];
endmodule
```

86. Design the tkGate implementation for the following Verilog code

```
module priority_encoder_4_to_2 (Y, GS, D);
  input [3:0] D;
  output [1:0] Y;
  output GS;
  wire w0, w1, w2;
  assign #4 w0 = ~D[0] & D[1];
  assign #4 w1 = ~D[0] & ~D[1] & D[2];
  assign #4 w2 = ~D[0] & ~D[1] & ~D[2] & D[3];
  assign #4 Y[0]  = w0 | w2;
  assign #4 Y[1]  = w1 | w2;
  assign #8 GS = D[0] | D[1] | D[2] | D[3];
endmodule
```

87. Design the tkGate implementation for the following Verilog code

```
module priority_encoder_4_to_2 (Y, GS, D);
  input [3:0] D;
```

```
  output [1:0] Y;
  output GS;
  wire [2:0] _D;
  wire w0, w1, w2;
  assign _D = ~D;
  assign #4 w0 = _D[0] & D[1];
  assign #4 w1 = _D[0] & _D[1] & D[2];
  assign #4 w2 = _D[0] & _D[1] & _D[2] & D[3];
  assign #4 Y[0]  = w0 | w2;
  assign #4 Y[1]  = w1 | w2;
  assign #8 GS = D[0] | D[1] | D[2] | D[3];
endmodule
```

88. Design the tkGate implementation for the following Verilog code

```
module h74148 (_GS, _Eo, _Y, _Ei, _D);
  input _Ei;
  input [7:0] _D;
  output _GS, _Eo;
  output [2:0] _Y;
  //
  function [2:0] f74148y (input _Ei,
                          input [7:0] _D);
    if (_Ei == 0)
      begin
        if (~_D & 8'b10000000)
          f74148y = 3'b000;
        else if (~_D & 8'b01000000)
          f74148y = 3'b001;
        else if (~_D & 8'b00100000)
          f74148y = 3'b010;
        else if (~_D & 8'b00010000)
          f74148y = 3'b011;
        else if (~_D & 8'b00001000)
          f74148y = 3'b100;
        else if (~_D & 8'b00000100)
          f74148y = 3'b101;
        else if (~_D & 8'b00000010)
          f74148y = 3'b110;
        else if (~_D & 8'b00000001)
          f74148y = 3'b111;
        else
```

```
          f74148y = 3'b111;
      end
    else
      f74148y = 3'b111;
  endfunction
  function f74148gs (input _Ei, input [7:0] _D);
    if (_Ei == 1)
      f74148gs = 1;
    else if (_D == 8'b11111111)
      f74148gs = 1;
    else
      f74148gs = 0;
  endfunction
  function f74148eo (input _Ei, input [7:0] _D);
    if (_Ei == 1)
      f74148eo = 1;
    else if (_Ei == 0 && _D == 8'b11111111)
      f74148eo = 0;
    else
      f74148eo = 1;
  endfunction
  //
  assign #8 _Y  = f74148y (_Ei, _D);
  assign #8 _GS = f74148gs (_Ei, _D);
  assign #8 _Eo = f74148eo (_Ei, _D);
endmodule
```

89. Design the tkGate implementation for the following Verilog code

```
module adder_4 (Co, S, A, B, Ci);
  input [3:0] A, B;
  input Ci;
  output Co;
  output [3:0] S;
  wire [4:0] C;
  //
  function add (input A, input B, input Ci);
    add = A^B^Ci;
  endfunction
  //
  function carry (input A, input B, input Ci);
    carry = A&B|A&Ci|B&Ci;
  endfunction
```

```
  //
  assign #0 C[0] = Ci;
  assign #6 C[1] = carry (A[0], B[0], C[0]);
  assign #6 C[2] = carry (A[1], B[1], C[1]);
  assign #6 C[3] = carry (A[2], B[2], C[2]);
  assign #6 C[4] = carry (A[3], B[3], C[3]);
  //
  assign #5 S[0] = add (A[0], B[0], C[0]);
  assign #5 S[1] = add (A[1], B[1], C[1]);
  assign #5 S[2] = add (A[2], B[2], C[2]);
  assign #5 S[3] = add (A[3], B[3], C[3]);
  //
  assign #0 Co = C[4];
endmodule
```

90. Design the tkGate implementation for the following Verilog code

```
module addsub_4 (CoBo, S, f, A, B, CiBi);
  input [3:0] A, B;
  input f, CiBi;
  output CoBo;
  output [3:0] S;
  wire [4:0] C;
  //
  function add (input A, input B, input C);
    add = A^B^C;
  endfunction
  //
  function carry (input A, input B, input C);
    carry = A&B|A&C|B&C;
  endfunction
  //
  assign #0 C[0] = CiBi^f;
  assign #6 C[1] = carry (A[0], B[0]^f, C[0]);
  assign #6 C[2] = carry (A[1], B[1]^f, C[1]);
  assign #6 C[3] = carry (A[2], B[2]^f, C[2]);
  assign #6 C[4] = carry (A[3], B[3]^f, C[3]);
  //
  assign #5 S[0] = add (A[0], B[0]^f, C[0]);
  assign #5 S[1] = add (A[1], B[1]^f, C[1]);
  assign #5 S[2] = add (A[2], B[2]^f, C[2]);
  assign #5 S[3] = add (A[3], B[3]^f, C[3]);
  //
```

```
  assign #0 CoBo = C[4]^f;
endmodule
```

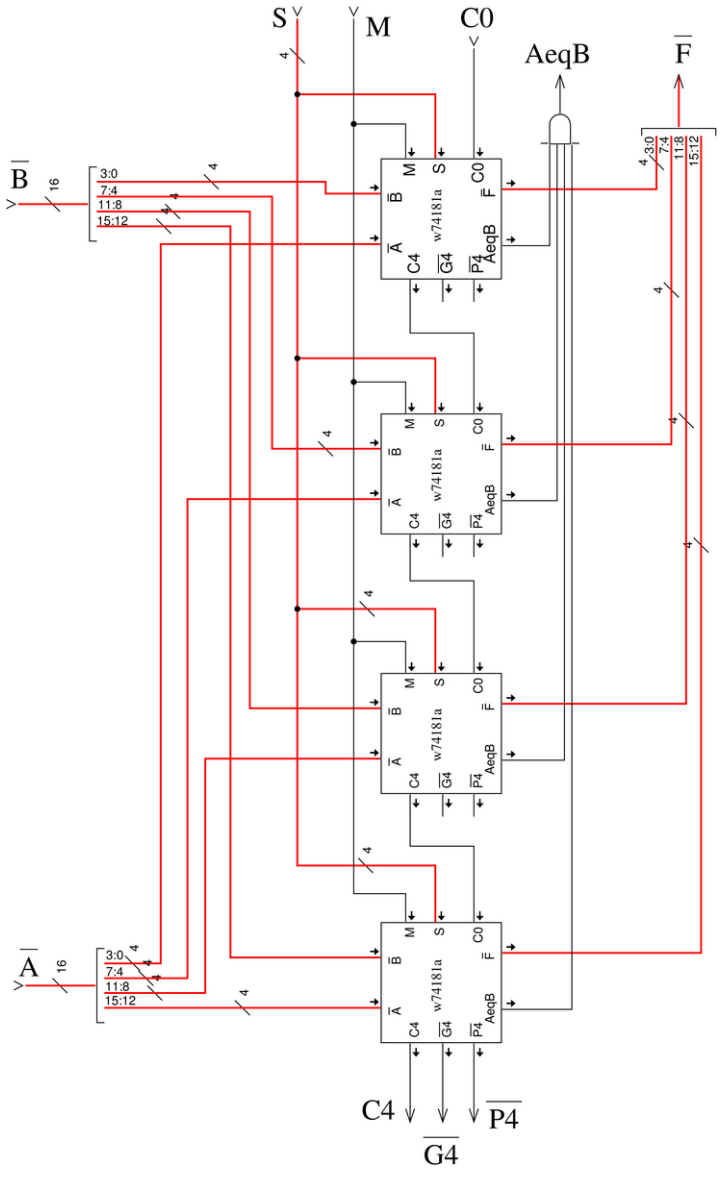
91. Design the tkGate implementation for the following Verilog code

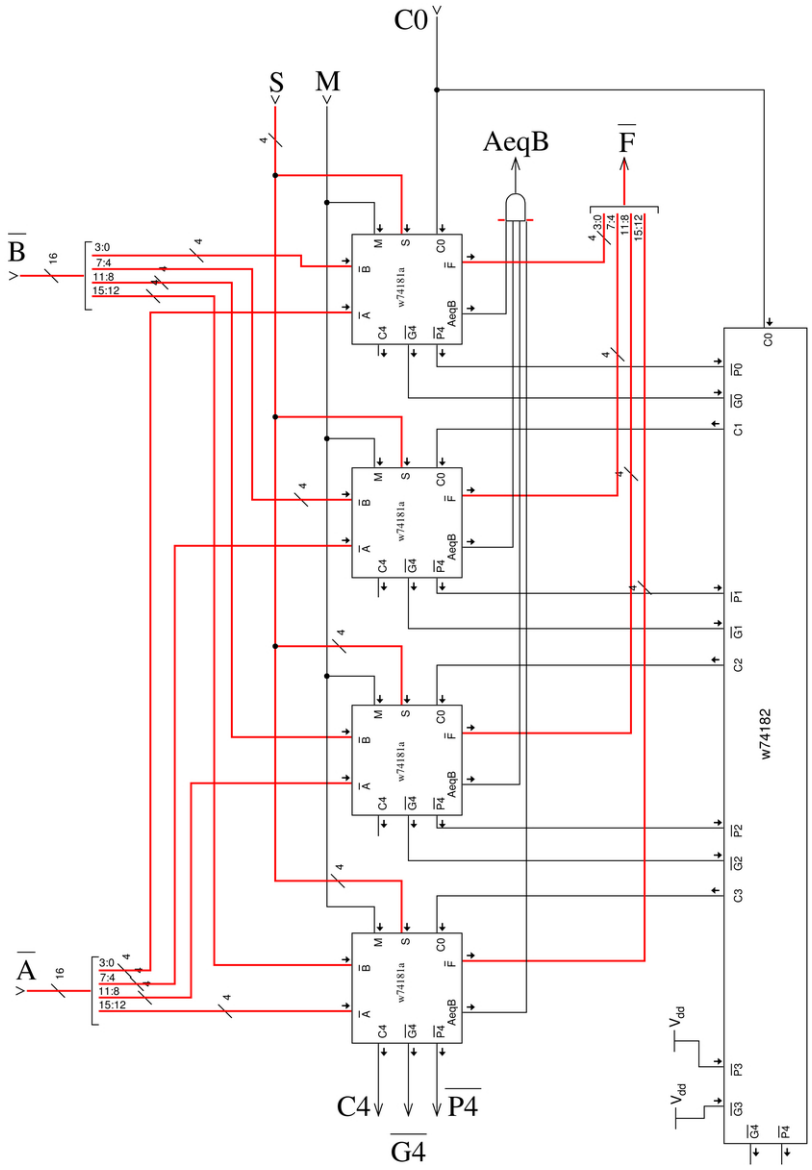
```
module SUM4 (C4, C3, P4, G4, S, A, B, C0);
  input [3:0] A, B;
  input C0;
  output C3, C4, P4, G4;
  output [3:0] S;
  wire [4:0] C, P, G;
  //
  assign #0 C[0] = C0;
  assign #5 S[0] = A[0]^B[0]^C[0];
  assign #5 S[1] = A[1]^B[1]^C[1];
  assign #5 S[2] = A[2]^B[2]^C[2];
  assign #5 S[3] = A[3]^B[3]^C[3];
  //
  assign #5 P[0] = A[0]|B[0];
  assign #5 P[1] = A[1]|B[1];
  assign #5 P[2] = A[2]|B[2];
  assign #5 P[3] = A[3]|B[3];
  assign #6 P[4] = P[3]&P[2]&P[1]&P[0];
  //
  assign #5 G[0] = A[0]&B[0];
  assign #5 G[1] = A[1]&B[1];
  assign #5 G[2] = A[2]&B[2];
  assign #5 G[3] = A[3]&B[3];
  assign #7 G[4] = G[3]|P[3]&G[2]|P[3]&P[2]&G[1]
                  |P[3]&P[2]&P[1]&G[0];
  //
  assign #7 C[1] = G[0]|P[0]&C[0];
  assign #7 C[2] = G[1]|P[1]&G[0]|P[1]&P[0]&C[0];
  assign #7 C[3] = G[2]|P[2]&G[1]|P[2]&P[1]&G[0]
                  |P[2]&P[1]&P[0]&C[0];
  assign #7 C[4] = G[3]|P[3]&G[2]|P[3]&P[2]&G[1]
                  |P[3]&P[2]&P[1]&G[0]
                  |P[3]&P[2]&P[1]&P[0]&C[0];
  //
  assign #0 C4 = C[4];
  assign #0 C3 = C[3];
  assign #0 P4 = P[4];
```

```
  assign #0 G4 = G[4];
endmodule
```

92. Design the tkGate implementation for the following Verilog code

```
module magnitude_4 (BA, EQ, AB, A, B);
  input [3:0] A, B;
  output BA, EQ, AB;
  wire [3:0] ab;
  wire [3:0] eq;
  wire [3:0] ba;
  //
  assign #3 ab[0] = A[0]&~B[0];
  assign #3 ab[1] = A[1]&~B[1];
  assign #3 ab[2] = A[2]&~B[2];
  assign #3 ab[3] = A[3]&~B[3];
  //
  assign #3 ba[0] = ~A[0]&B[0];
  assign #3 ba[1] = ~A[1]&B[1];
  assign #3 ba[2] = ~A[2]&B[2];
  assign #3 ba[3] = ~A[3]&B[3];
  //
  assign #3 eq[0] = ~(ab[0]|ba[0]);
  assign #3 eq[1] = ~(ab[1]|ba[1]);
  assign #3 eq[2] = ~(ab[2]|ba[2]);
  assign #3 eq[3] = ~(ab[3]|ba[3]);
  //
  assign #4 EQ = eq[0]&eq[1]&eq[2]&eq[3];
  //
  assign #3 AB = (ab[0]&eq[1]&eq[2]&eq[3])
               |(ab[1]&eq[2]&eq[3])
               |(ab[2]&eq[3])
               |ab[3];
  //
  assign #3 BA = (ba[0]&eq[1]&eq[2]&eq[3])
               |(ba[1]&eq[2]&eq[3])
               |(ba[2]&eq[3])
               |ba[3];
endmodule
```


S
M
C0
AeqB
F
B
A
3:0
7:4
11:8
15:12
w74181a
C4
G4
P4
C4
G4
P4


C0
S
M
AeqB
F
B
A
w74181a
w74182
C4
G4
P4
Vdd