Modules

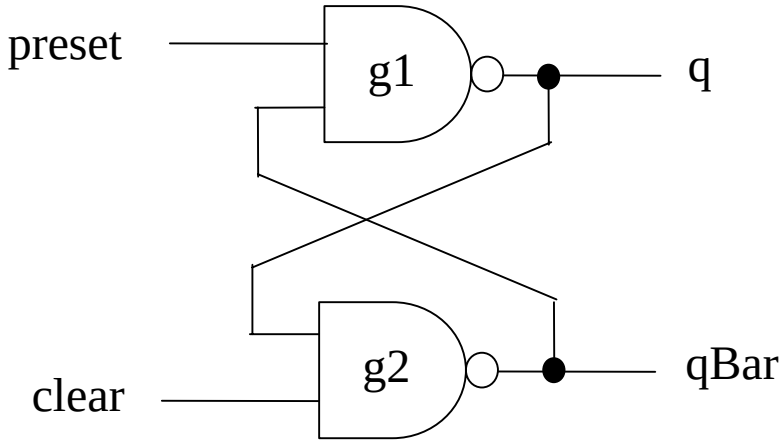
- The Verilog language describes a digital system as a set of ***modules***.
- Each module has an interface to other modules as well as a description of its contents.
- A module represents a logical unit that can be described either
  - by specifying its internal logical structure — for instance, describing the actual logic gates it is comprised of, or
  - by describing its behavior in a program-like manner — in this case, focusing on what the module does rather than on its logical implementation.
- The modules are then interconnected with nets, allowing them to communicate.

A Structural Description

- A simple NAND latch

```
module ffNand;
        wire        q, qBar;
        reg         preset, clear;

        nand #1
                    g1 (q, qBar, preset),
                    g2 (qBar, q, clear);
endmodule
```



A Structural Description

- A simple NAND latch

```
module ffNand; ← Module name
        wire        q, qBar; ← Wire declarations
        reg         preset, clear; ← Register declarations

        nand #1 ← NAND gate having a delay of one time unit
                    g1 (q, qBar, preset), ← Instantiation of a NAND gate
                    g2 (qBar, q, clear); ← Instantiation of a NAND gate
endmodule
```

A Structural Description

- A simple NAND latch
  - Each module definition includes the keyword ***module*** and its name, and is terminated by the ***endmodule*** statement.
  - The wires, declared by the ***wire*** statements, are used to transmit logic values among the submodules of this module.
  - The registers, declared by the ***reg*** statements, represent the storage elements that will hold logic values.
  - NAND gates are one of the predefined logic gate types in the language.
  - The gate instances are connected to the wires and registers.
  - The first label in the parentheses is the gate's output and the others are inputs.

A Structural Description

- Module definition versus module instantiation
  - Using the module statement, we define a module once specifying all of its inner detail.
  - The module may be used (instantiated) in the design many times.
  - Each of these instantiations are called instances of the module, and can be separately named and connected differently.

A Structural Description

- Nets
  - Gates are connected by nets.
  - Nets are one of the two fundamental data types of the language.
  - Registers are the other data type.
  - Nets are used to model an electrical connection between structural entities such as gates.
  - A wire is one type of net.
  - Other net types include wired-AND, wired-OR and trireg connections.
  - The trireg net models a wire as a capacitor that stores electrical charge.
  - Except for the trireg net, nets do not store values but only transmit values that are driven on them.

A Structural Description

- Hierarchical descriptions
  - Several gates or modules are built into larger modules in a hierarchical manner.
  - In a simple NAND latch example, NAND gates were used to build the *ffNand* module.
  - This *ffNand* module could then be used as a piece of a larger module by instantiating it into another module.
  - The use of hierarchical descriptions allows us to control the complexity of a design by breaking the design into smaller and more meaningful submodules.
  - When instantiating the submodules, all we need know about them is their interface.

Simulating a Module

- A simple NAND latch to be simulated

```
module ffNandSim;
        wire        q, qBar;
        reg         preset, clear;

        nand #1
                    g1 (q, qBar, preset),
                    g2 (qBar, q, clear);
```

Simulating a Module

```
        initial
        begin
                $monitor($time,,
                        "preset = %b clear = %b q = %b qBar = %b",
                        preset, clear, q, qBar);
                #10 preset = 0; clear = 1;
                #10 preset = 1;
                #10 clear = 0;
                #10 clear = 1;
                #10 $finish;
        end
endmodule
```

Simulating a Module

- A simple NAND latch to be simulated includes
  - A structural description of a simple NAND latch
  - The statements that will provide stimulus to the NAND gate instances
  - The statements that will monitor the changes in the outputs.

- The ***monitor*** statement
  - A simulation command to monitor and print a set of values when any one of the values changes
  - %b represents a printing control for binary.
  - The ,, put extra spaces between the display of the time and the quoted string.

Simulating a Module

- Simulator
  - Executes the statements in the ***initial*** statement and propagates changed values from the outputs of gates and registers to other gate module inputs.
  - Keeps track of time, causing the changed values appear at some specified time in the future rather than immediately.
  - The future changes are typically stored in a time-ordered event queue.
  - When the simulator has no further statement execution or value propagation to perform at the current time, it finds the next time-ordered event from the event queue, updates time to that of the event, and executes the event.
  - Continues until there are no more events to be simulated or the user halts the simulation.

Simulating a Module

- How does the simulator work in this example ?
  - The first assignment statement

    #10 preset = 0; clear = 1;

    specifies the registers *preset* and *clear* will be loaded zero and one respectively 10 time units from the current time (time 0).
  - At time 10, *preset* and *clear* are set to zero and one respectively, and then the changed values are propagated.
  - At time 11, *q* becomes one, which is the result of *preset* being zero.
  - The value of *q* is propagated to the input of NAND gate *g2* which is then scheduled to propagate its changed output 1 time unit in the future.
  - At time 12, *qBar* becomes zero.

Simulating a Module

- How does the simulator work in this example ?
  - The second assignment statement

    #10 preset = 1;

    specifies the register *preset* will be loaded one 10 time units from the current time (time 10).
  - At time 20, *preset* is set to one, and *q* and *qBar* remain constant.
  - The third assignment statement

    #10 clear = 0;

    specifies the register *clear* will be loaded zero 10 time units from the current time (time 20).
  - At time 30, *clear* is set to zero and the latch changes state after 2 gate delays.

Simulating a Module

- How does the simulator work in this example ?
  - The fourth assignment statement

    #10 clear = 1;

    specifies the register *clear* will be loaded one 10 time units from the current time (time 30).
  - At time 40, *clear* is set to one, and *q* and *qBar* remain constant.
  - The last initial statement

    #10 $finish;

    stops the simulation and returns control to the host operating system at time 50.

Simulating a Module

- Results of simulating the simple NAND latch

0 preset = x clear = x q = x qBar = x

10 preset = 0 clear = 1 q = x qBar = x

11 preset = 0 clear = 1 q = 1 qBar = x

12 preset = 0 clear = 1 q = 1 qBar = 0

20 preset = 1 clear = 1 q = 1 qBar = 0

30 preset = 1 clear = 0 q = 1 qBar = 0

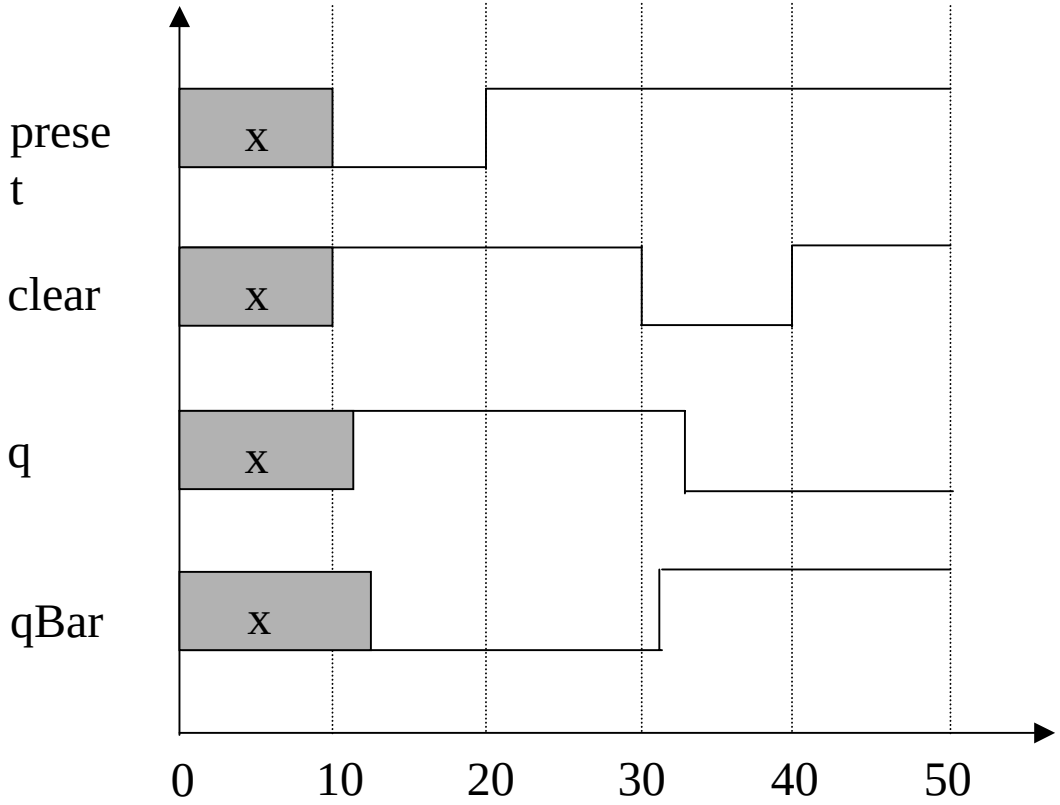
31 preset = 1 clear = 0 q = 1 qBar = 1

32 preset = 1 clear = 0 q = 0 qBar = 1

40 preset = 1 clear = 1 q = 0 qBar = 1

Simulating a Module

- Results of simulating the simple NAND latch



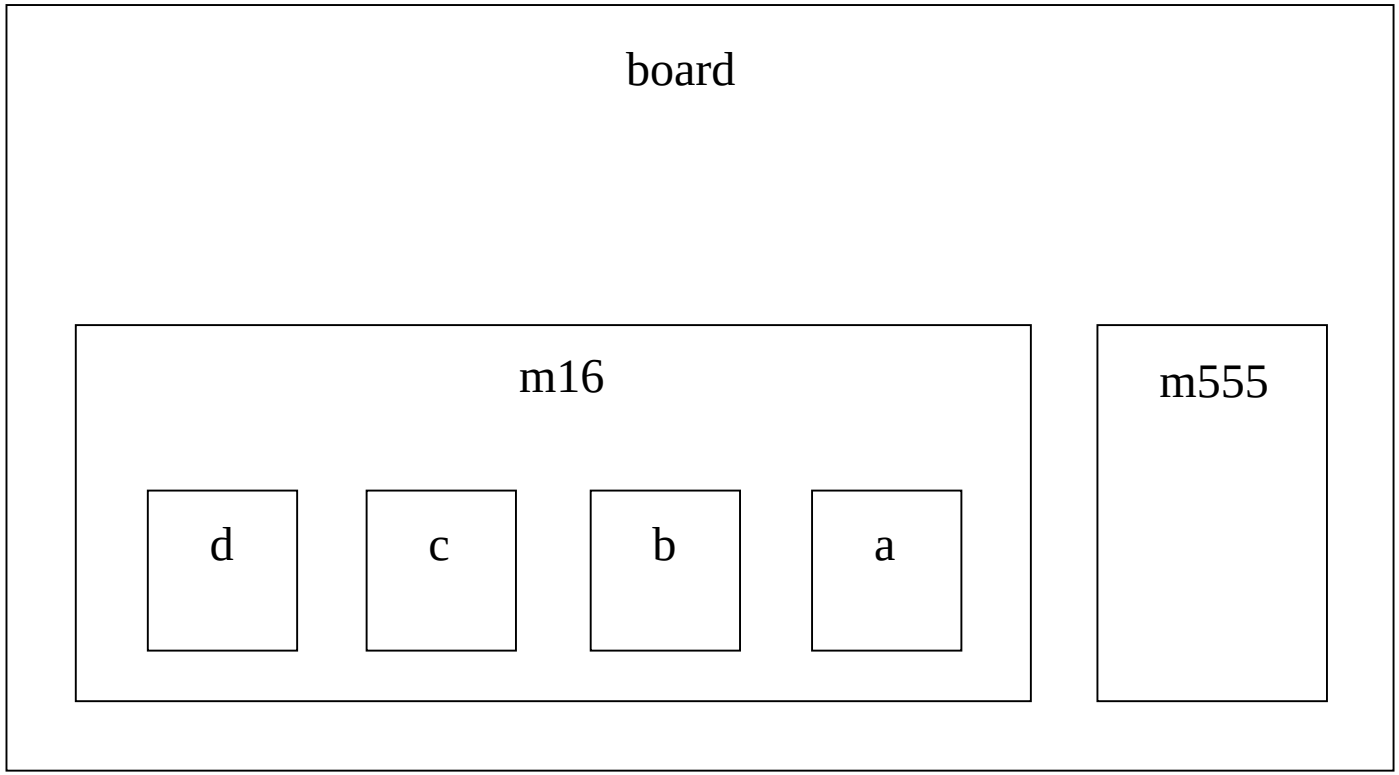
Simulating a Module

- The four values that a bit may have in the simulator
  - 1 : true (one) state
  - 0 : false (zero) state
  - x : unknown state
  - z : high-impedance state

- Why were *preset* and *clear* defined as registers ?
  - A means of setting and holding the input values during simulation
  - Wires do not hold values but merely transmit values.

Module Hierarchy

- Top-level view



Module Hierarchy

- Counter module

```
module m16 (value, clock, fifteen, altFifteen);
        output [3:0]        value;
        output              fifteen,
                            altFifteen;
        input               clock;
```

Module Hierarchy

```
        dEdgeFF a (value[0], clock, ~value[0]),
                b (value[1], clock, value[1] ^ value[0]),
                c (value[2], clock, value[2] ^ &value[1:0]),
                d (value[3], clock, value[3] ^ &value[2:0]);

        assign fifteen = value[0] & value[1] & value[2] & value[3];
        assign altFifteen = &value;
endmodule
```

Module Hierarchy

- Ports
  - Must be declared to be inputs, outputs, or inouts.

- Vectors and scalars
  - The square brackets construct declares the range of bit numbers, the first number being the most significant bit and the second being the least significant bit.
  - The bit numbers used in the specifications must be positive.
  - Registers, nets and ports declared to have a range of bits are said to be vectors.
  - Single-bit registers, nets and ports are said to be scalars.

Module Hierarchy

- Association of ports
  - The order of the names in the instantiation port list must match the order of ports in the definition.
  - Analogous to the software situation where we need to know the order of parameters when calling a subroutine.
  - If the *dEdgeFF* module has the following port definition:

    dEdgeFF (q, clock, data)

    the instance *a* will have the following connections:

    q (output of dEdgeFF) → value[0] (output of m16)

    clock (input of dEdgeFF) → clock (input of m16)

    data (input of dEdgeFF) → ~value[0] (output of m16)

Module Hierarchy

- Operators used in this example
  - ~ : Bitwise negation

    Complements each bit in the operand.
  - ^ : Bitwise XOR

    Produces the bitwise exclusive OR of two operands.
  - & : Unary reduction AND

    Produces the single bit AND of all of the bits of the operand.
  - & : Bitwise AND

    Produces the bitwise AND of two operands.

    Unary reduction and binary bitwise ANDs are distinguished by syntax.

Module Hierarchy

- The ***assign*** statement
  - Another way of describing a combinational logic function
  - Called continuous assignment statement because the result of the logical expression on the right-hand side of the equal sign is evaluated anytime one of its inputs changes and the result drives the output in a simulation.
  - Allows us to describe a combinational logic function without regard to its actual structural implementation, i.e., there are no instantiated modules.
  - In this example, *fifteen* and *altFifteen* have the same logic function.

- Part-select of a vector
  - A selected range of bits from the entire vector is used in the operation.

    value[1:0] ← value[3:0]

Module Hierarchy

- D-type negative edge-triggered flip-flop module

```
module dEdgeFF (q, clock, data);
        output    q;
        reg       q;
        input     clock, data;

        initial
                  #10 q = 0;
        always
                  @(negedge clock) q = #10 data;
endmodule
```

Module Hierarchy

- Registers
  - A register named *q* drives output *q*.
  - Register *q* directly models the flip-flop's storage bit.
  - When simulated, *q* is initialized to be zero as specified in the initial statement.

- The ***always*** statement
  - Basis for modeling sequential behavior
  - Essentially a while (true) statement
  - Includes one or more procedural statements that are repeatedly executed.
  - The procedural statements within the always statement execute much like a software program.

Module Hierarchy

- @(negedge clock) q = #10 data;
  - When there is a negative edge on the *clock* input, then the value of the *data* input will be copied and, after 10 time units, register *q* will be loaded with that copied value.
  - When *q* is loaded, it is loaded with the value that *data* had before the delay started.

- Behavioral descriptions
  - Capture all of the functionality of the module.
  - Leave the actual logical implementation open to the designer.

Module Hierarchy

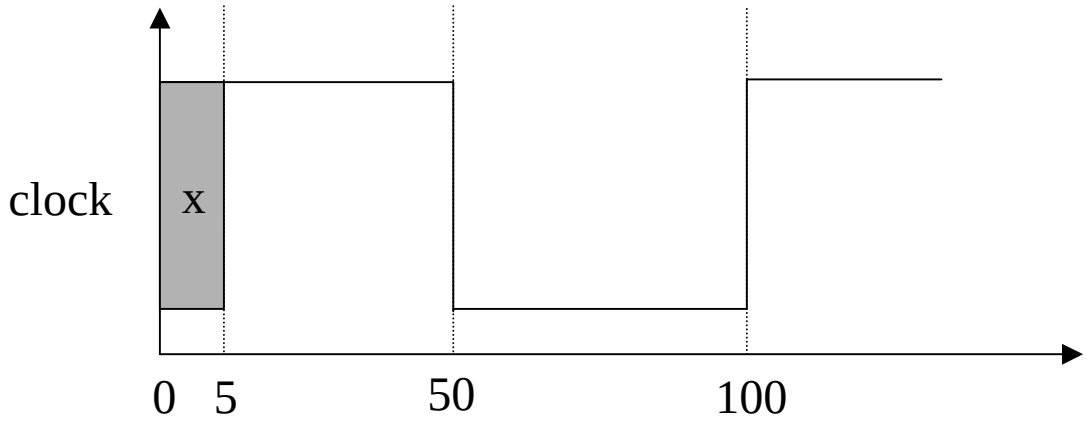
- Clock generator module

```
module m555 (clock);
        output    clock;
        reg       clock;

        initial
                  #5 clock = 1;
        always
                  #50 clock = ~clock;
endmodule
```

Module Hierarchy

- Clock generator module
  - The *clock* is initialized to be one after 5 time units have passed.
  - After the first 50 time units have passed, the always statement will be scheduled to execute and change its value.
  - Because *clock* will change value every 50 time units, we have created a clock with a period of 100 time units.



Module Hierarchy

- Two different delay mechanisms

```
#50 clock = ~clock;
q = #10 data;
```

  - The first statement waits for 50 time units and loads *clock* with its complement at that time.
  - The second statement makes an internal copy of the current value of *data*, waits 10 time units, and then loads *q* with that internal copy.

- The ***timescale*** compiler directive for real time units

```
`timescale 1ns / 100ps
```

  - Used to attach units (1ns) and a precision (100ps) for rounding to the time numbers.

Module Hierarchy

- Top-level module

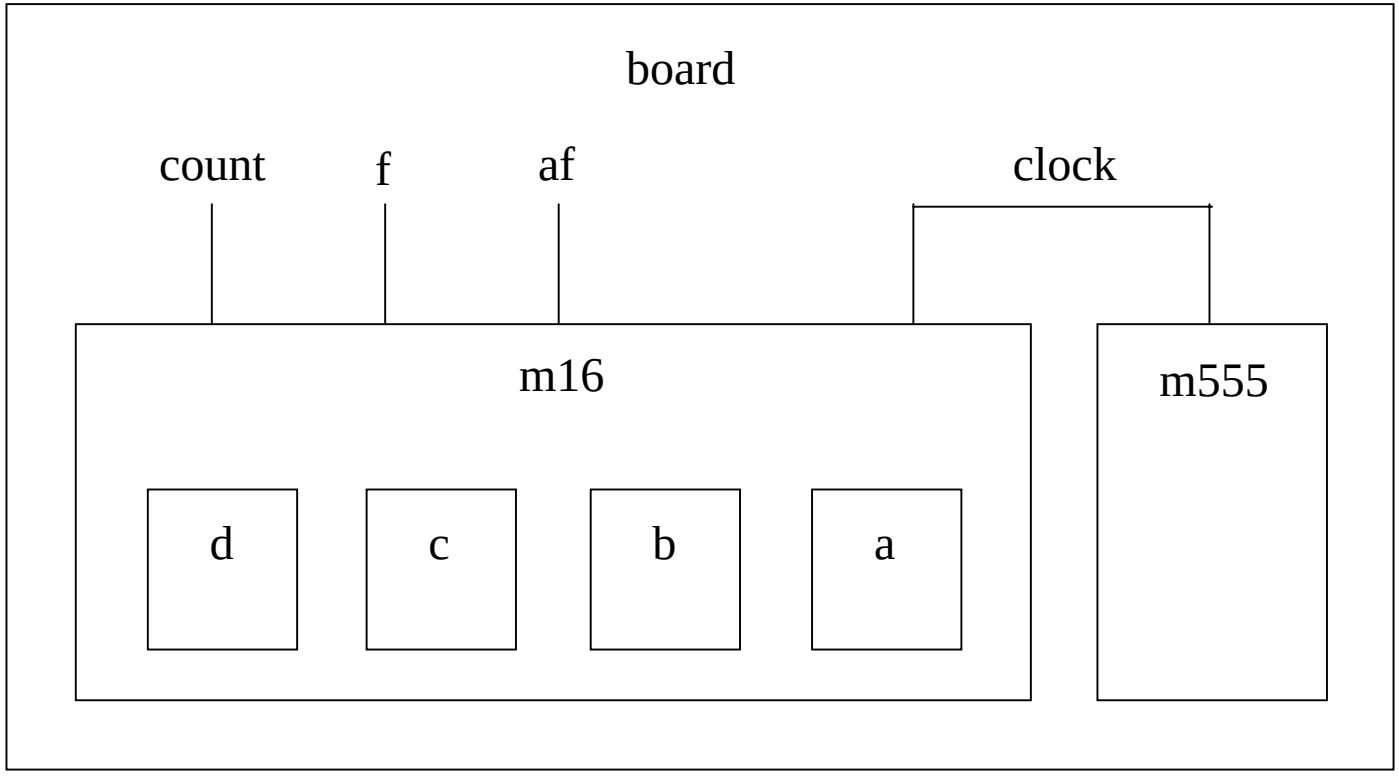
```
module board;
        wire [3:0]          count;
        wire                clock, f, af;

        m16         counter (count, clock, f, af);
        m555        clockGen (clock);

        always @(posedge clock)
                $display ($time,,, “count = %d, f = %d, af = %d”, count, f, af);
endmodule
```

Module Hierarchy

- Top-level view with internal connections



Module Hierarchy

- The ***display*** statement
  - Similar to a print statement in a programming language.
  - %d represents a printing control for an unsigned decimal integer.
  - The ,,, put extra spaces between the display of the time and the quoted string.
  - In combination with the always statement, the printout in this example will display the values at the positive edge of the *clock*.

Module Hierarchy

- Simulation trace

5 count = x, f = x, af = x

100 count = 1, f = 0, af = 0

200 count = 2, f = 0, af = 0

300 count = 3, f = 0, af = 0

400 count = 4, f = 0, af = 0

500 count = 5, f = 0, af = 0

600 count = 6, f = 0, af = 0

700 count = 7, f = 0, af = 0

800 count = 8, f = 0, af = 0

900 count = 9, f = 0, af = 0

Module Hierarchy

1000 count = 10, f = 0, af = 0

1100 count = 11, f = 0, af = 0

1200 count = 12, f = 0, af = 0

1300 count = 13, f = 0, af = 0

1400 count = 14, f = 0, af = 0

1500 count = 15, f = 1, af = 1

1600 count = 0, f = 0, af = 0

1700 count = 1, f = 0, af = 0

1800 count = 2, f = 0, af = 0

1900 count = 3, f = 0, af = 0

Behavioral Modeling

- Behavioral model
  - An abstraction of how the module works
  - The outputs of the module are described in terms of its inputs.
  - No effort is made to describe how the module is implemented in terms of logic gates.
  - Useful early in the design process, when a designer is more concerned with simulating the system's intended behavior to understand its gross performance characteristics with little regard to its final implementation.
  - Later, structural models with accurate detail of the final implementation are substituted and resimulated to demonstrate functional and timing correctness.

Behavioral Modeling

- Counter module described with behavioral statements

```
module m16Behav (value, clock, fifteen, altFifteen);
        output [3:0]        value;
        reg [3:0]           value;
        output              fifteen, altFifteen;
        reg                 fifteen, altFifteen;
        input               clock;

        initial
                value = 0;
        always
```

Behavioral Modeling

```
        begin
                @(negedge clock) #10 value = value + 1;
                if (value = = 15)
                begin
                        altFifteen = 1; fifteen = 1;
                end
                else
                begin
                        altFifteen = 0; fifteen = 0;
                end
        end
endmodule
```

Behavioral Modeling

- Registers
  - The declaration of the registers and outputs of the same name implicitly connect the outputs of the registers to the ports.
  - In behavioral descriptions, the procedural statements require the use of registers (or variables in software programming) to store the values.
  - The structural version of *m16* did not use registers, rather it implicitly connected the output drivers in the instantiated flip-flops to the output port with a wire.

Behavioral Modeling

- Constant numbers
  - Can be specified in decimal, hexadecimal, octal, or binary.
  - May optionally start with a + or -.
  - Can be given in one of two forms.
  - The first form
    - An unsized decimal number specified using the digits from the sequence 0 to 9
    - Verilog calculates a size for use in an expression.

Behavioral Modeling

- Constant numbers
  - The second form specifies the size of the constant and takes the form:

    ss…s'fnn…n

    - ss…s : the size in bits (Specified as a decimal number.)
    - f : the base format (d, h, o, b or D, H, O, B)
    - nn…n : the value (For hexadecimal, a ~ f or A ~ F used.)
    - x and z values may be given in all but the decimal base.
    - In hexadecimal, an x or z would represent 4 bits, in octal, 3 bits.
    - Normally, zeros are padded on the left if nn…n < ss…s.
    - The first digit of nn…n is x or z, then x or z is padded on the left.
    - An underline character may be inserted to improve readability, which must not be the first character (e.g., 12'b0x0x_1101_0zx1).

Behavioral Modeling

- Constant numbers
  - The statement in the example

    altFifteen = 0;

    could be written more exactly as

    altFifteen = 1'b0;

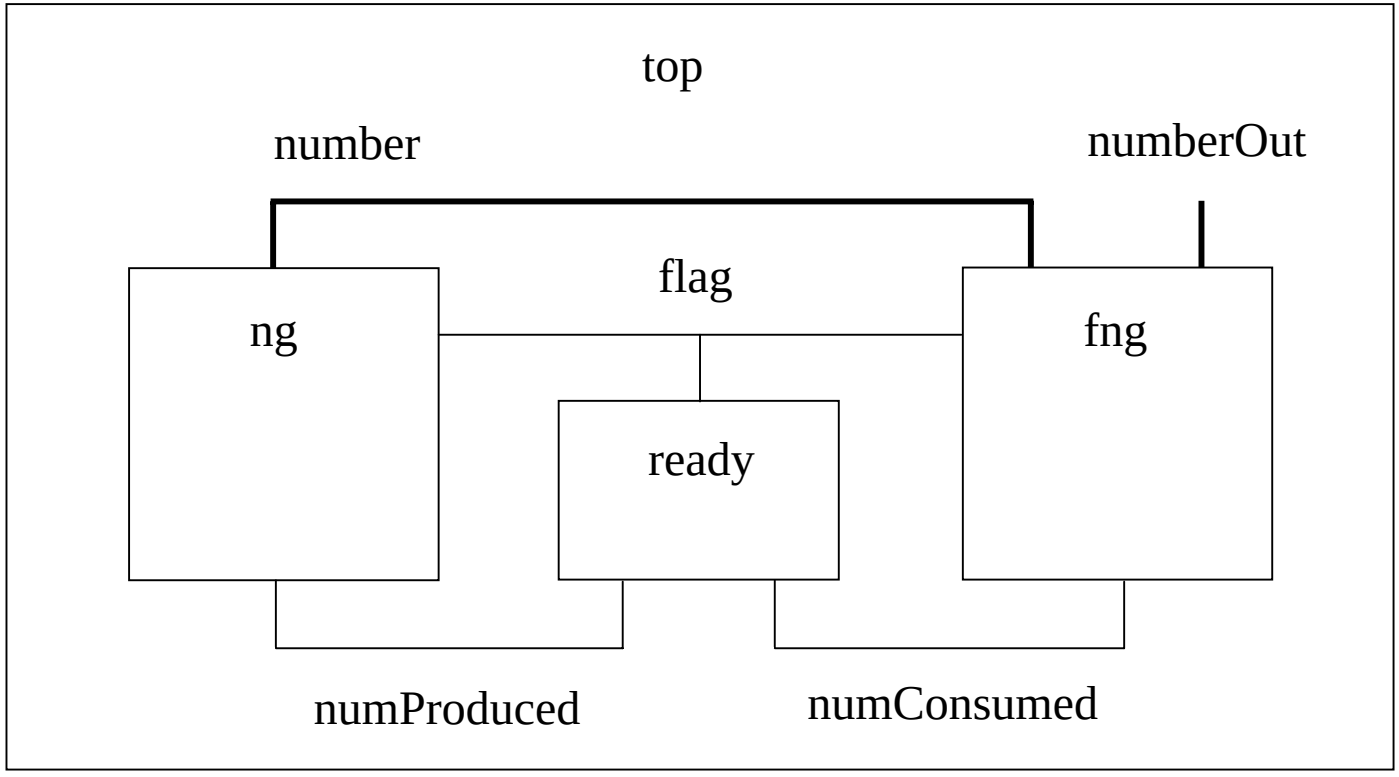
    meaning that one-bit operand with value zero specified in binary is loaded into register *altFifteen*.
  - The issue of whether to use one form versus another is a matter of readability and exactness.

Behavioral Modeling

- Mixing structure and behavior
  - The natural evolution of a digital system is from abstract behavior to detailed, implementable structure.
  - Along the design path, the design will be represented at times by a mixture of behavioral and structural models.
    - Not all modules of a design will be designed down to the detailed structural level at the same time.
    - Part of a design may make use of off-the-shelf (pre-designed) hardware.

Fibonacci Number Generator Example

- Top-level view



Fibonacci Number Generator Example

- Top-level module

```
module top ();
        wire            flag, numProduced, numConsumed;
        wire [15:0]     number, numberOut;

        nandLatch       ready (flag, , numConsumed, numProduced);
        numberGen       ng (number, numProduced, flag);
        fibNumberGen    fng (number, flag, numConsumed, numberOut);
endmodule
```

Fibonacci Number Generator Example

- Operations
  - The seed-number generator module (*ng*) produces a number *n* and passes it onto the Fibonacci number generator module (*fng*).
  - The Fibonacci number generator module (*fng*) calculates the *n*-th Fibonacci number.
  - The latch module (*ready*) is provided for the two modules to signal when to pass another number.
  - The behavioral descriptions in *ng* and *fng* execute concurrently and pass information when the latch output indicates that previous data has been consumed and new valid data is ready.
  - The latch acts as a data-valid signal for a single element queue between producer and consumer modules.

Fibonacci Number Generator Example

- The NAND latch module

```
module nandLatch (q, qBar, set, reset);
        output      q, qBar;
        input       set, reset;

        nand #2
                    (q, qBar, set),
                    (qBar, q, reset);
endmodule
```

Fibonacci Number Generator Example

- Instance names
  - Instances of primitive gates, such as NAND and NOR, need not be individually named.
  - Instances of modules must be individually named.

- Unconnected ports

```
nandLatch ready (flag, , numConsumed, numProduced);
                          ↓
module nandLatch (q, qBar, set, reset);
```

Fibonacci Number Generator Example

- The seed-number generator module

```
module numberGen (number, numProduced, flag);
        output [15:0]       number;
        output              numProduced;
        input               flag;
        reg [15:0]          number;
        reg                 numProduced;
```

Fibonacci Number Generator Example

```
        initial
        begin
                number = 0;
                numProduced = 1;
        end
        always
        begin
                wait (flag = = 1) number = number + 1;
                #100 numProduced = 0;
                #10 numProduced = 1;
        end
endmodule
```

Fibonacci Number Generator Example

- Operations of the seed-number generator module
  - Waits for the input *flag* to be one.
  - As soon as the input *flag* becomes one, the always statement can then continue executing.
  - The *flag* is the *q* output of the *nandLatch* which, when set, signifies that the previous value has been received and another can be generated.
  - A new seed number is generated when the input *flag* becomes one, and a delay of 100 time units will occur.
  - The *numProduced* is set to zero for 10 time units, causing the *q* output of the *nandLatch* to be cleared to signify that a new number is available.

Fibonacci Number Generator Example

- The Fibonacci number generator module

```
module fibNumberGen (startingValue, flag, numConsumed, fibNum);
        input [15:0]        startingValue;
        input               flag;
        output              numConsumed;
        output [15:0]       fibNum;
        reg                 numConsumed;
        reg [15:0]          fibNum;
        reg [15:0]          count, oldNum, temp;
```

Fibonacci Number Generator Example

```
initial
begin
        numConsumed = 0;
        #10 numConsumed = 1;
        $monitor ($time,, “fibNum = %d, startingValue = %d”,
                                fibNum, startingValue);
end

always
begin
        wait (flag = = 0) count = startingValue;
        oldNum = 1;
```

Fibonacci Number Generator Example

```
        numConsumed = 0;
        #10 numConsumed = 1; // Signal ready for new input data

        for (fibNum = 0; count != 0; count = count - 1)
        begin
                temp = fibNum;
                fibNum = fibNum + oldNum;
                oldNum = temp;
        end
        $display ("%d fibNum = %d", $time, fibNum);
    end
endmodule
```

Fibonacci Number Generator Example

- Operations of the Fibonacci number generator module
  - Waits for the input *flag* to be zero.
  - The *fibNumberGen* module waits on a signal complementary to what the *numberGen* module waits for.
  - When the input *flag* becomes zero, a 10 time unit pulse is sent out to set the *nandLatch* module and signal the *numberGen* module to begin producing another new seed number.
  - The *n*-th Fibonacci number is calculated in register *fibNum* by the ***for*** loop statement.
  - The *fibNumberGen* module needs a copy of *startingValue* because it is an input and cannot be changed with a procedural statement.

Fibonacci Number Generator Example

- Edge-triggering versus level-sensitive mechanism
  - The ***event control*** (@) statement models an edge-triggering mechanism.

    @(posedge clock) number = number + 1;
  - The ***wait*** statement is level sensitive.

    wait (clock = = 1) number = number + 1;

Fibonacci Number Generator Example

- Assignment statements
  - The continuous assignment statement

```
wire        a;
assign      a = b | (c & d);
            or
wire        a = b | (c & d);
```

  - ➢ The right-hand side of the equal sign can be thought of as a logic gate whose output is connected to a wire *a*.
  - ➢ For any change at any time to *b*, *c*, or *d*, the output of the expression will be evaluated and made to drive wire *a*.
  - ➢ May only drive nets such as a wire.

Fibonacci Number Generator Example

- Assignment statements
  - The procedural assignment statement

    oldNum = 1;

    - Found in initial and always statements.
    - Can only load values into registers or memory elements.
    - The loading of the value is done only when control is transferred to the procedural statement.
    - Control is transferred to a procedural assignment statement in a sequential manner, like a normal software program.
    - The flow of control is interrupted either by an event (@) or wait statement.

Fibonacci Number Generator Example

- Wires attached to a register

```
reg [15:0]      newValue;
wire [15:0]     y = newValue;
```

  - Wire *y* is connected to the output of register *newValue*.
  - Anytime *newValue* is loaded procedurally with a new value, it will drive wire *y* and the inputs it is connected to with that new value.

Testbench

- Testbench approach
  - Better to separate the design's description and the means of testing it.
  - The design may be simulated and monitored through its ports, and later synthesized using other CAD tools such as a logic synthesizer.
  - If the behavior to test a design is included with the design's module, then it should be removed when synthesizing — an error prone process.
  - Testbench approach is to have a top module within which there are two other modules, one representing the system being designed, and the other representing the test generator and monitor.

Testbench

- Top module

```
module testBench;
        wire        q, qBar, preset, clear;

        design_ffNand       d (q, qBar, preset, clear);
        test_ffNand         t (q, qBar, preset, clear);
endmodule
```

Testbench

- Design module

```
module design_ffNand (q, qBar, preset, clear);
        output      q, qBar;
        input       preset, clear;

        nand #1
                    g1 (q, qBar, preset),
                    g2 (qBar, q, clear);
endmodule
```

Testbench

- Test module

```
module test_ffNand (q, qBar, preset, clear);
        input       q, qBar;
        output      preset, clear;
        reg         preset, clear;
```

Simulating a Module

```
        initial
        begin
                $monitor($time,,
                        "preset = %b clear = %b q = %b qBar = %b",
                        preset, clear, q, qBar);
                #10 preset = 0; clear = 1;
                #10 preset = 1;
                #10 clear = 0;
                #10 clear = 1;
                #10 $finish;
        end
endmodule
```