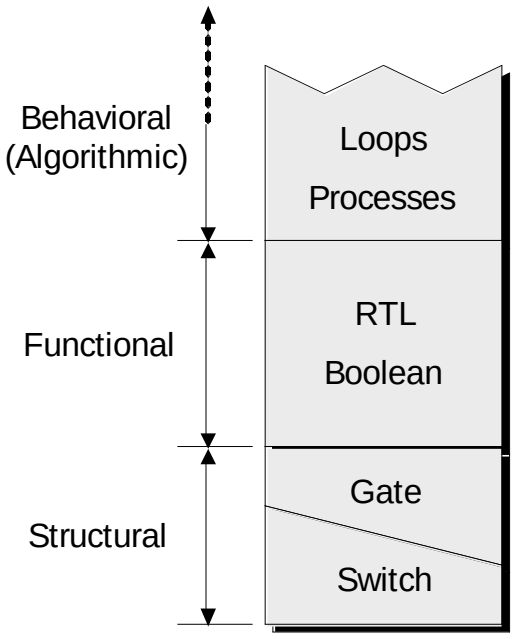
HDL-Based Design

Eduardo Sanchez
EPFL

Introduction

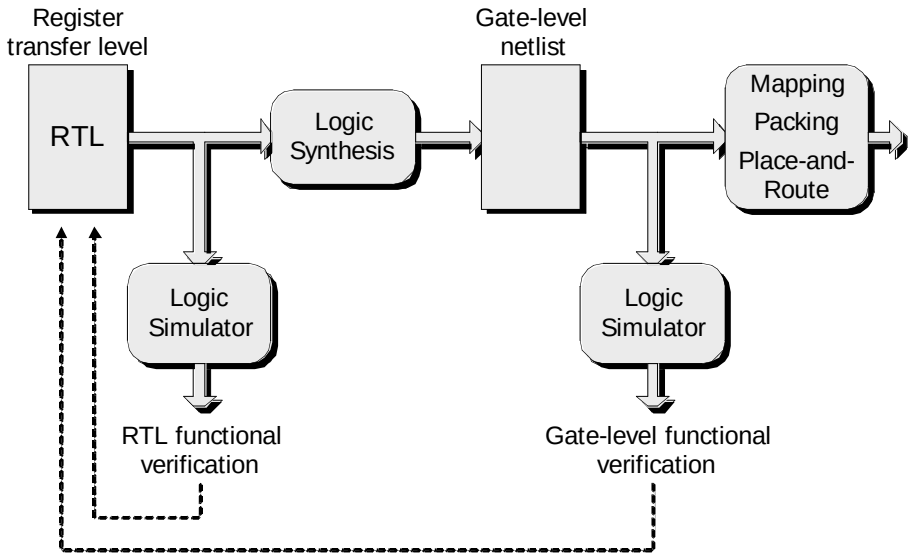
- As designs grew in size and complexity, schematic-based design began to run out of steam
- In addition to the fact that capturing a large design at the gate level of abstraction is prone to error, it is also extremely time-consuming
- One solution is the design based on the use of hardware description languages, or HDLs

- The functionality of a digital circuit can be represented at different levels of abstraction and different HDLs support these levels of abstraction to a greater or lesser extent
- At the register transfer level (RTL), a design is considered as a collection of registers linked by combinational logic
- The highest level of abstraction supported by traditional HDLs is known as behavioral, which refer to the ability of describe the behavior of a circuit using abstract constructs like loops and processes

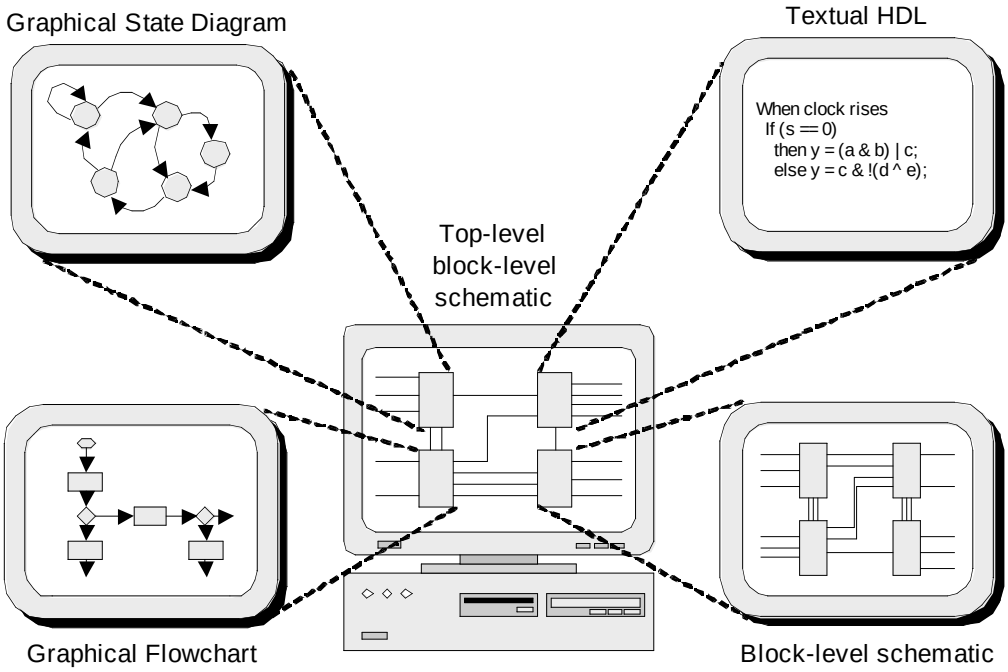


A simple HDL-based FPGA flow

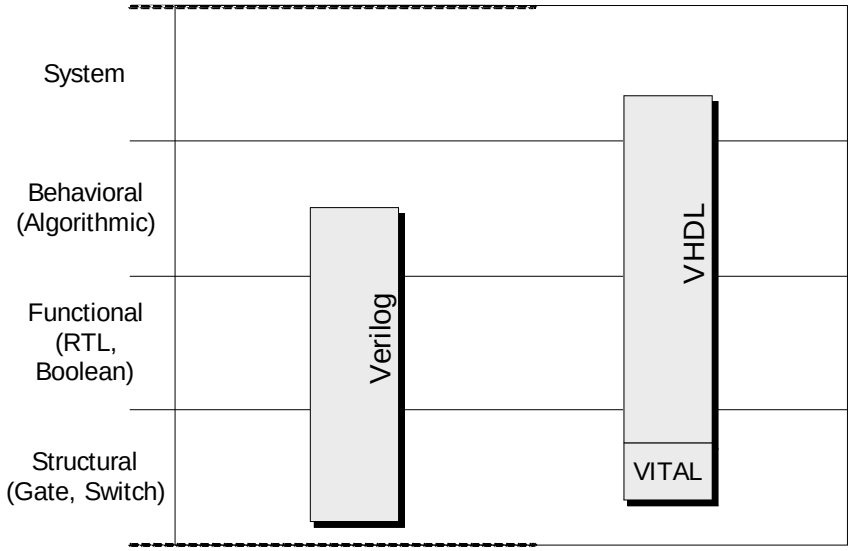
- It wasn't until the very early 1990s that HDL-based flows featuring logic synthesis technology became fully available in the FPGA world



- As a picture tells a thousand words, graphical entry techniques remain popular at a variety of levels



Some examples of HDLs



- Relatively easy to learn
- Fixed data types
- Interpreted constructs
- Good gate-level timing
- Limited design reusability
- Limited design management
- No structure replication

- Relatively difficult to learn
- Abstract data types
- Compiled constructs
- Less good gate-level timing
- Good design reusability
- Good design management
- Supports structure replication

- Verilog was originally designed with simulation in mind
- VHDL was created as a design documentation and specification language that took simulation into account
- As a result, one can use both of these languages to describe constructs that can be simulated but not synthesized

- The Open Verilog International (OVI) and VHDL International organizations linked up to form a new body called Accellera
- The mission of this new organization was to focus on identifying new standards and formats, to develop these standards and formats, and to foster the adaptation of new methodologies based on these standards and formats
- In 2002, Accellera released the specification for a hybrid language called System Verilog 3.0. It include things like the assertion (key to the formal verification strategy known as model checking) and extended synthesis capabilities

C/C++-based design flows

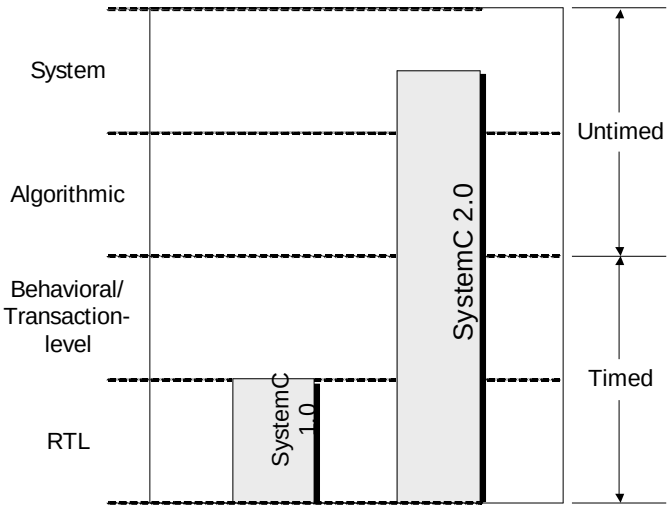
- C/C++-based HDLs can be used to describe designs at the RTL level of abstraction
- It is necessary to augment standard C/C++ with special statement to support such concepts as clocks, pins, concurrency, synchronization, and resource sharing
- These descriptions can subsequently be simulated 5 to 10 times faster than their Verilog or VHDL counterparts, and synthesis tools are available to convert them into gate-level netlists
- It provides a more natural environment for hardware/software codesign and coverification

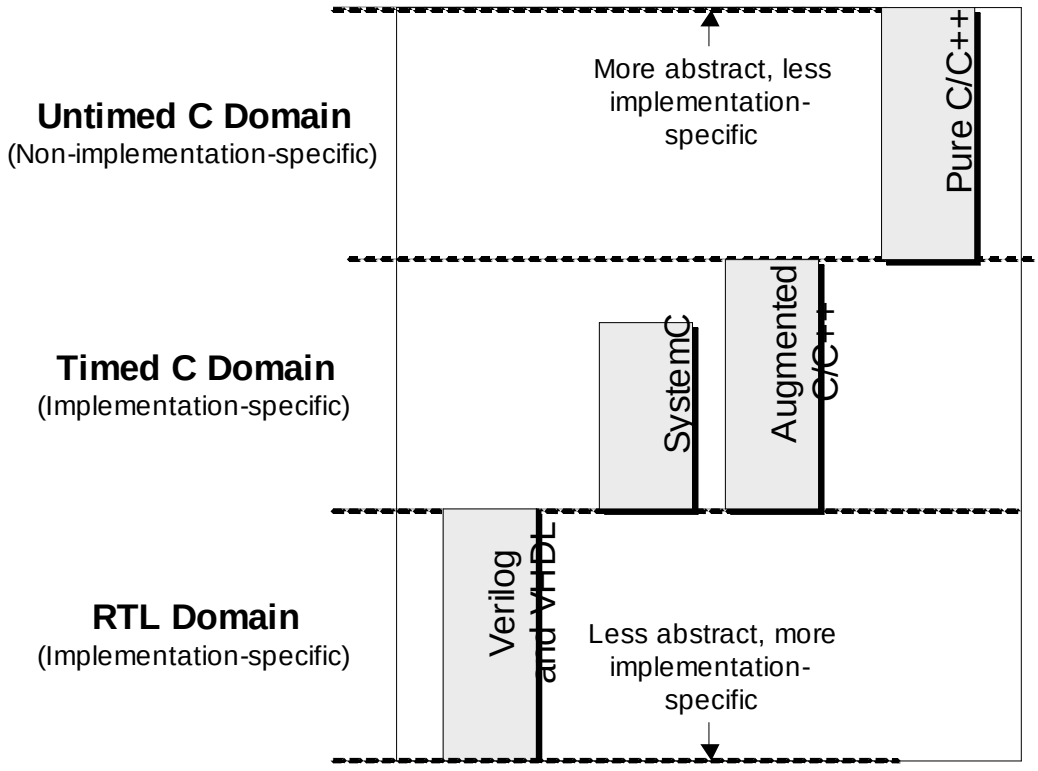
SystemC 1.0

- SystemC 1.0 was a C++ class library, created in 2000, that facilitated the representation of notions such as concurrency, timing, and I/O pins
- By means of this class library, engineers could capture designs at the RTL level of abstraction

SystemC 2.0

- In 2002, SystemC 2.0 augmented the 1.0 release with some high-level modeling constructs such as FIFOs
- This release also included a variety of behavioral, algorithmic, and system-level modeling capabilities, such as the concepts of transactions and channels (which are used to describe the communication of data between blocks at an abstract level)





An Introduction to SystemC

- SystemC is essentially a C++ class library used for modeling concurrent systems in C++
- Along with concurrency, SystemC provides a notion of timing as well as an event driven simulation environment
- SystemC isn't a new language, it's C++. Consequently, existing software IP can be seamlessly linked into a SystemC project
- The SystemC Class Library has been developed by a group of companies forming the Open SystemC Initiative (OSCI)
- The SystemCC reference simulator is freely available at www.systemc.org

The class

- The class in C++ is called an Abstract Data Type (ADT). It defines both data members and access functions (also called methods)
- Data members and access functions are not visible from the outside world. The designer is responsible for making publicly available the essential set of access functions for manipulating an ADT

- Example: the declaration of an ADT called **`counter`** with a data member **`value`** and publicly available access functions: **`do_reset`**, **`do_count`** and **`do_read`**

```
class counter
{
  int value;
public:
  void do_reset() { value = 0; }
  void do_count_up() { value++; }
  int do_read() { return value; }
};
```


- A class declaration will commonly also contain specialized functions such as constructors and a destructor. When constructors are used, they provide initial values for the ADT's data members. This mechanism is the only allowed means for setting a default value to any data member
- A destructor is used to perform clean-up operations before an instance of the ADT becomes out of scope. Pragmatically, a destructor is used for closing previously opened files or de-allocating dynamically allocated memory

The object

- An object is an instance of an ADT
- Example of object instantiation and message passing:

```
void main() {

counter first_counter;
counter second_counter(55);

// message passing
first_counter.do_reset();
for (int i=0; i<10; i++) { first_counter.do_count_up(); }
second_counter.do_count_up();

first_counter.do_reset();
second_counter.do_reset();
}
```


Inheritance

- C++ provides a sophisticated mechanism for reusing code called inheritance
- With inheritance, designers are able to create new ADTs from existing ones by accessing all public elements of a parent class into a child class

- Example: creation of a **`modulo_counter`** ADT from an existing **`counter`** ADT whilst using inheritance
- A new access function called **`do_count_up`** is created. This overrides the inherited function from the parent class **`counter`**. The remaining public access functions found in the parent class (**`do_reset`**, **`do_read`**) are now available to the child class (**`modulo_counter`**)


```
class modulo_counter : public counter {
  int terminal_count;
public:
  void do_count_up() {
    if ( do_read() < terminal_count ) {
      counter::do_count_up(); }
    else { do_reset(); }
  }
  modulo_counter(int tc): terminal_count(tc), counter(0) {
    cout << "A new modulo_counter instance" << endl; }
};
```

Modules and processes

- SystemC provides processes to support the construction of networks of independent (concurrent/parallel) pieces of code
- To deal with large designs, SystemC uses hierarchy
- Hierarchy is implemented in SystemC by using the module, a class that can be linked to other modules using ports
- Modules may contain processes, and instances of other modules

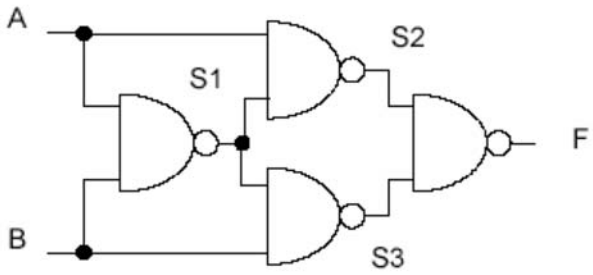
- Anx SystemC module has to be derived from the existing class **`sc_module`**
- SystemC modules are analogous to Verilog modules or VHDL entity/architecture pairs
- By definition, modules communicate with other modules through channels and via ports
- Typically, a module will contain numerous concurrent processes used to implement their required behavior

- Ports are created from existing SystemC template classes: **`sc_in<>`** and **`sc_out<>`**
- A constructor function is used to register any of the access functions as concurrent processes

- Two kinds of process exist in SystemC: **`SC_METHOD`** and **`SC_THREAD`**. To some extent, the 2 processes are similar since they will both contain sequential statements and have their own thread of execution
- By definition, **`SC_METHOD`** cannot be suspended during its execution. Once the execution of the **`SC_METHOD`** has been performed, it halts and waits for new activities on its static or dynamic sensitivity list before executing again
- **`SC_THREAD`** can be suspended during execution and resumed at a later stage. Furthermore, by definition, **`SC_THREAD`** only executes once during simulation and then suspends

Example

- A NAND gate is a combinational circuit: it has no memory, and requires no clock
- The model of the NAND gate can use the simplest kind of SystemC process, an **SC_METHOD**
- The function must be declared as an **SC_METHOD** and made sensitive to its inputs




```
#include "systemc.h"
SC_MODULE(nand2) // declare nand2 sc_module
{
  sc_in<bool> A, B; // input signal ports
  sc_out<bool> F; // output signal ports

  void do_nand2() // a C++ function
  {
    F.write( !(A.read() && B.read()) );
  }

  SC_CTOR(nand2) // constructor for nand2
  {
    SC_METHOD(do_nand2); // register do_nand2 with kernel
    sensitive << A << B; // sensitivity list
  }
};
```

```
#include "systemc.h"
#include "nand2.h"
SC_MODULE(exor2)
{
  sc_in<bool> A, B;
  sc_out<bool> F;
  nand2 n1, n2, n3, n4;
  sc_signal<bool> S1, S2, S3;

  SC_CTOR(exor2) : n1("N1"), n2("N2"), n3("N3"), n4("N4")
  {
    n1.A(A);
    n1.B(B);
    n1.F(S1);
    n2 << A << S1 << S2;
    n3(S1);
    n3(B);
    n3(S3);
    n4.A(S2);
    n4.B(S3);
    n4.F(F);
  }
};
```


Channels

- Channels are SystemC's communication medium. They can be seen as a more generalized form of signals
- SystemC provides an exhaustive range of predefined channels for generic uses such as: **`sc_signal`**, **`sc_fifo`**, **`sc_semaphore`**, etc
- By definition, modules are interconnected via channels and ports. In turn, ports and channels communicate via a common interface

Simulation

- Simulation instructions are usually located inside a function called **`sc_main`**
- This function will execute simulation specific commands such as setting the simulator's resolution, channels to be traced, top level instance, simulation running time and more

An Introduction to Verilog

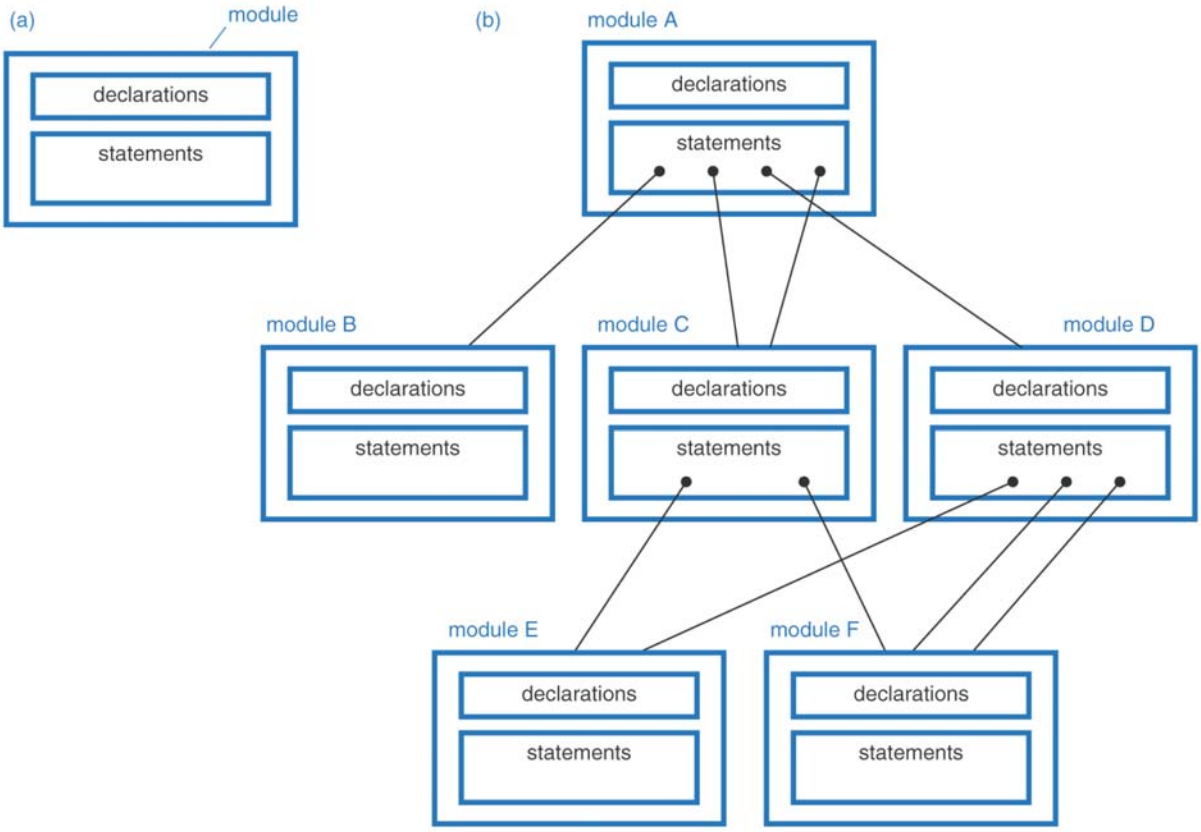
- Verilog was introduced by Gateway Design Automation in 1984, as a proprietary hardware description and simulation language
- Synopsys introduced Verilog-based synthesis tools in 1988
- Gateway was bought by Cadence in 1989
- IEEE has published two Verilog standards, in 1991 and 2001

Main features

- Designs may be decomposed hierarchically
- Each design element has both a well-defined interface and a precise functional specification
- Functional specifications can use either a behavioral algorithm or an actual hardware structure to define an element's operation
- Concurrency, timing, ad clocking can all be modeled
- The logical operation and timing behavior of a design can be simulated

Program structure

- The basic unit of design and programming in Verilog is a module
- A Verilog module has declarations (descriptions of its I/Os, as well as local signals, variables, constants and functions) and statements (specification of the module's operation)
- Module's specification can be made behaviorally or structurally
- The scope of signal, constant, and other definitions remains local to each module; values can be passed between modules only by using declared input and output signals
- Verilog is sensitive to case



Syntax of a module declaration

```
module module-name (port-name, port-name, ..., port-name);
  input declarations
  output declarations
  inout declarations
  net declarations
  variable declarations
  parameter declarations
  function declarations
  task declarations

  concurrent statements
endmodule
```

- Example:

```
module VrInhibit( X, Y, Z ); // also known as 'BUT-NOT'
  input X, Y;                // as in 'X but not Y'
  output Z;                  // (see [Klir, 1972])

  assign Z = X & ~Y;
endmodule
```


- A 1-bit signal can take on one of only four possible values:
  - **0**: logical 0, or false
  - **1**: logical 1, or true
  - **x**: an unknown logical value
  - **z**: high impedance
- Bitwise boolean operators:
  - **&**: AND
  - **|**: OR
  - **^**: XOR
  - **~^**: XNOR
  - **~**: NOT

- Verilog has two classes of signals:
  - net: a wire in a physical circuit, connecting modules and other elements
  - variable: a value used during a program's execution, without any physical significance in a circuit
- There are several kinds of nets. The default net type is **`wire`**. Another commonly used type is **`tri`**, for modeling of 3-state connections
- There several kinds of variables. The most commonly used variable types are **`reg`** and **`integer`**
- A variable's value can be changed only within procedural code within a module; it cannot be changed from outside the module. Thus, input and inout ports cannot have a variable type
- Procedural code can assign values only to variables

- Constants can be declared as parameters
- Example:

```
parameter BUS_SIZE = 32,
          MSB = BUS_SIZE-1, LSB = 0;
parameter ESC = 7'b0011011;
```

- Nets, variables and constants can all be vectors. A vector is declared including a range specification [msb:lsb]. The range can be ascending or descending

- The Verilog arithmetic and shift operators are:
  - **+**: addition
  - **-**: substraction
  - ***:** multiplication
  - **/**: division
  - **%**: modulus (remainder)
  - **<<**: shift left
  - **>>:** shift right
- The vectors are treated as unsigned integers. For signed arithmetic, the signal declaration has to include the **signed** keyword

- The Verilog logical operators are:
  - **&&**: logical AND
  - **||**: logical OR
  - **!**: logical NOT
  - **==**: logical equality
  - **!=**: logical inequality
  - **>**: greater than
  - **>=**: greater than or equal
  - **<**: less than
  - **<=**: less than or equal

- Syntax and examples of the Verilog conditional operator:

logical-expression ? *true-expression* : *false-expression*

```
X ? Y : Z

(A>B) ? A : B;

(sel==1) ? op1 : (
  (sel==2) ? op2 : (
    (sel==3) ? op3 : (
      (sel==4) ? op4 : 8'bx )))
```


Structural design elements

- Each concurrent statement in a Verilog module "executes" simultaneously with the other statements in the same module declaration
- In the structural style of description, individual gates and other components are instantiated and connected to each other using nets
- The Verilog built-in gates are:

and	xor	bufif0
nand	xnor	bufif1
or	buf	notif0
nor	not	notif1

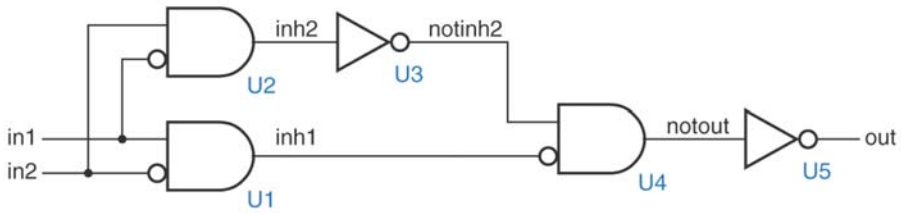
- The syntax of Verilog instance statements is:

```
component-name instance-identifier ( expr, expr, ..., expr );

component-name instance-identifier ( .port-name(expr),
                                     .port-name(expr),
                                     ...
                                     .port-name(expr) );
```

- Example:

```
module VrInh( in, invin, out );
  input in, invin;
  output out;
  wire notinvin;

  not U1 (notinvin, invin);
  and U2 (out, in, notinvin);
endmodule
```


- Example:




```
module VrSillyXOR(in1, in2, out);
  input in1, in2;
  output out;
  wire inh1, inh2, notinh2, notout;

  VrInh U1 ( .out(inh1), .in(in1), .invin(in2) );
  VrInh U2 ( .out(inh2), .in(in2), .invin(in1) );
  not U3 ( notinh2, inh2 );
  VrInh U4 ( .out(notout), .in(notinh2), .invin(inh1) );
  not U5 ( out, notout );
endmodule
```

Dataflow design elements

- "Continuous-assignment statements" allow Verilog to describe a combinational circuit in terms of the flow of data and operations on the circuit
- The basic syntax of a continuous-assignment statement is:

```
assign net-name = expression;
assign net-name[bit-index] = expression;
assign net-name[msb:lsb] = expression;
assign net-concatenation = expression;
```

- As with instance statements, the order of continuous-assignment statement in a module doesn't matter

- Example:

```
module Vrprimed (N, F);
input [3:0] N;
output F;
wire N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0;

  assign N3L_N0     = ~N[3]                  & N[0];
  assign N3L_N2L_N1 = ~N[3] & ~N[2] &  N[1]       ;
  assign N2L_N1_N0  =         ~N[2] &  N[1] & N[0];
  assign N2_N1L_N0  =          N[2] & ~N[1] & N[0];
  assign F = N3L_N0 | N3L_N2L_N1 | N2L_N1_N0 | N2_N1L_N0;
endmodule
```

Behavioral design elements

- The key element of Verilog behavioral design ("procedural code") is the **`always`** block
- The syntax of the **`always`** block is:

```
always @ (signal-name or signal-name or ... or signal-name)
  procedural-statement

always procedural-statement
```

- Procedural statements execute sequentially. However, the **`always`** block itself executes concurrently with other concurrent statements in the same module

- A Verilog concurrent statement such as an **`always`** block is always either executing or suspended. A concurrent statement initially is suspended; when any signal in its sensitivity list changes value, it resumes execution, starting with its first procedural statement and continuing until the end. This continues until the statement executes without any of these signals changing value at the current time. In simulation, all of this happens in 0 simulated time
- An instance or continuous-assignment statement also has a sensitivity list, an implicit one. All of the input signals in an instantiated component or module are on the instance statement's implicit sensitivity list. Likewise, all of the signals on the righthand side of a continuous-assignment statement are on its implicit sensitivity list

- If multiple values are assigned to X during a given pas through an **`always`** block, the last value assigned dominates
- If no value is assigned to X, the simulator infers a latch in order to retain the previous value of X
- Procedural statements are written in a style similar to C
- The types of procedural statements are: blocking assignment, nonblocking assignment, **`begin-end`**, **`if`**, **`case`**, **`while`** and repeat

- The syntax of the procedural assignment statements is:

```
variable-name = expression ;    // blocking assignment

variable-name <= expression ;   // nonblocking assignment
```

- A blocking assignment is similar to the assignment statement of any other procedural language. A nonblocking assignment is different: it evaluates its righthand side immediately, but it does not assign the resulting value to the lefthand side until an infinitesimal delay after the entire **`always`** block has been executed. Thus, the "old" value of the lefthand side continues to be available for the rest of the **`always`** block

- Always use blocking assignments (=) in **`always`** blocks intended to create combinational logic
- Always use nonblocking assignments (<=) in **`always`** blocks intended to create sequential logic
- Do not mix blocking and nonblocking assignments in the same **`always`** block
- Do not make assignments to the same variable in two different **`always`** blocks

- A list of one or more procedural statements can be enclosed by the keywords **`begin`** and **`end`**
- The procedural statements within a **`begin-end`** block execute sequentially
- The syntax of the **`begin-end`** block is:

```
begin
   procedural-statement
   ...
   procedural-statement
end

begin : block-name
   variable declarations
   parameter declarations

   procedural-statement
   ...
   procedural-statement
end
```

- Example:

```
module Vrprimeb (N, F);
input [3:0] N;
output F;
reg F;

  always @ (N)
    begin : Fcomp
    reg N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0;
      N3L_N0     = ~N[3]                  & N[0] ;
      N3L_N2L_N1 = ~N[3] & ~N[2] &  N[1]        ;
      N2L_N1_N0  =         ~N[2] &  N[1] & N[0] ;
      N2_N1L_N0  =          N[2] & ~N[1] & N[0] ;
      F = N3L_N0 | N3L_N2L_N1 | N2L_N1_N0 | N2_N1L_N0;
    end
endmodule
```


- The syntax of the Verilog **`if`** statements is:

```
if ( condition ) procedural-statement

if ( condition ) procedural-statement
else procedural-statement
```

- Example:

```
module Vrprimei (N, F);
input [3:0] N;
output F;
reg F;
parameter OneIsPrime = 1; // Change this to 0 if you
                          // don't consider 1 to be prime.
  always @ (N)
    if (N == 1) F = OneIsPrime;
    else if ( (N % 2) == 0 )
      begin if (N == 2) F = 1; else F = 0; end
    else if (N <= 7) F = 1;
    else if ( (N==11) || (N==13) ) F = 1;
    else F = 0;
endmodule
```

- The syntax of the Verilog **case** statements is:

```
case ( selection-expression )
  choice , ... , choice : procedural-statement
  ...
  choice , ... , choice : procedural-statement
  default : procedural-statement
endcase
```

- Example:

```
module Vrprimecs (N, F);
input [3:0] N;
output F;
reg F;

  always @ (N)
    case (N)
      1, 2, 3, 5, 7, 11, 13 : F = 1;
      default : F = 0;
    endcase
endmodule
```


- Example:

```
module Vrbytecase (A, B, C, sel, Z);
input [7:0] A, B, C;
input [1:0] sel;
output [7:0] Z;
reg [7:0] Z;

  always @  (A or B or C or sel)
    case (sel)
      2'd0 : Z = A;
      2'd1 : Z = B;
      2'd2 : Z = C;
      2'd3 : Z = 8'bz;
      default : Z = 8'bx;
    endcase
endmodule
```

- In order to avoid inferred latches, a good solution is to assign default values to variables at the beginning of the **`always`** block
- Example:

```
module Vrprimef (N, F, special);
input [3:0] N;
output F, special;
reg F, special;

  always @ (N) begin
    F = 1'bx; special = 1'b0; // defaults
    if (N == 1) begin F = 1; special = 1; end
    else if ( (N % 2) == 0 )
      begin if (N == 2) F = 1; else F = 0; end
    else if (N <= 7) F = 1;
    else if ( (N==11) || (N==13) ) F = 1;
    else F = 0;
  end
endmodule
```


- The syntax of the Verilog **`for`** statement is:

```
for ( loop-index = first-expr ; logical-expression ; loop-index = next-expr )
  procedural-statement

for ( loop-index = first; loop-index <= last; loop-index = loop-index + 1; )
  procedural-statement
```

- After initializing the *loop-index*, a **`for`** loop executes *procedural-statement* for a certain number of iterations. At the beginning of each iteration, it evaluates *logical-expression*. If the value is false, the **`for`** loop stops execution. If the value is true, it executes *procedural-statement* and at the end of the iteration it assigns *next-expr* to *loop-index*. Iterations continue until *logic-expression* is false

- Example:

```
module Vrprimebv (N, F);
input [15:0] N;
output F;
reg F, prime;
integer i;

  always @ (N) begin
    prime = 1;   // initial values
    if ( (N==1) || (N==2) ) prime = 1; // Special cases
    else if ((N % 2) ==0) prime = 0;   // Even, not prime
    else for ( i = 3 ; i <= 255 ; i = i+2 )
      if ( ((N % i) == 0) && (N != i) )
        prime = 0;   // Set to 0 if N is divisible by any i
    if (prime==1) F = 1; else F = 0;
  end
endmodule
```

- This design is not synthesizable...

- The other Verilog looping statements are **`repeat`**, **`while`**, and **`forever`**:

```
repeat ( integer-expression )
  procedural-statement

while ( logical-expression )
  procedural-statement

forever
  procedural-statement
```

- These looping statements cannot be used to synthesize combinational logic, only sequential logic, and then only if the procedural statement is a **`begin-end`** block that includes timing control that waits for a signal edge

- Like a function in a high-level programming language, a Verilog *function* accepts a number of inputs and returns a single result
- The syntax of a Verilog *function definition* is:

```
function result-type function-name ;
   input declarations
   variable declarations
   parameter declarations

   procedural-statement
endfunction
```

- A function executes in zero simulated time. Also, the values of any local variables are lost from one function call to the next

- Example:

```
module VrSillierXOR(in1, in2, out);
  input in1, in2;
  output out;
  reg out;

  function Inhibit ;
    input In, invIn;
    Inhibit = In & ~invIn;
  endfunction

always @ (in1 or in2) begin : IB
    reg inh1, inh2;
    inh1 = Inhibit(in1,in2);
    inh2 = Inhibit(in2,in1);
    out = ~Inhibit(~inh2,inh1);
  end
endmodule
```

- A Verilog *task* is similar to a function, except it does not return a result
- Unlike functions, tasks can have inout and output arguments
- While a function call can be used in the place of an expression, a *task call* (or *task enable*) can be used in the place of a statement
- Verilog synthesizers can't handle tasks at all
- The syntax of a Verilog *task definition* is:

```
task function-name ;
   input declarations
   inout declarations
   output declarations
   variable declarations
   parameter declarations

   procedural-statement
endtask
```


- Like combinational behavior, edge-triggered behavior in Verilog is specified using **`always`** blocks. The difference is in the sensitivity list of the **`always`** block: the keyword **`posedge`** or **`negedge`** is placed in front of a signal name to indicate that the block should be executed only at the positive or negative edge of the named signal
- Verilog was originally designed as a logic circuit description and simulation language and was only later adapted to synthesis. Thus, the language has several features and constructs that cannot be synthesized
- Digital designers who use synthesis tools will need to pay reasonably close attention to their coding style in order to obtain good results