Commuter Aided Design
CAD

LECTURE 4

Modern Digital Design Flow Introduction

- The purpose of hardware description languages (HDL) is to describe digital circuitry using a text-based language.
- HDLs provide a means to describe large digital systems without the need for schematics, which can become impractical in very large designs.
- HDLs have evolved to support logic simulation at different levels of abstraction. This provides designers the ability to begin designing and verifying functionality of large systems at a high level of abstraction and postpone the details of the circuit implementation until later in the design cycle.

Modern Digital Design Flow Introduction

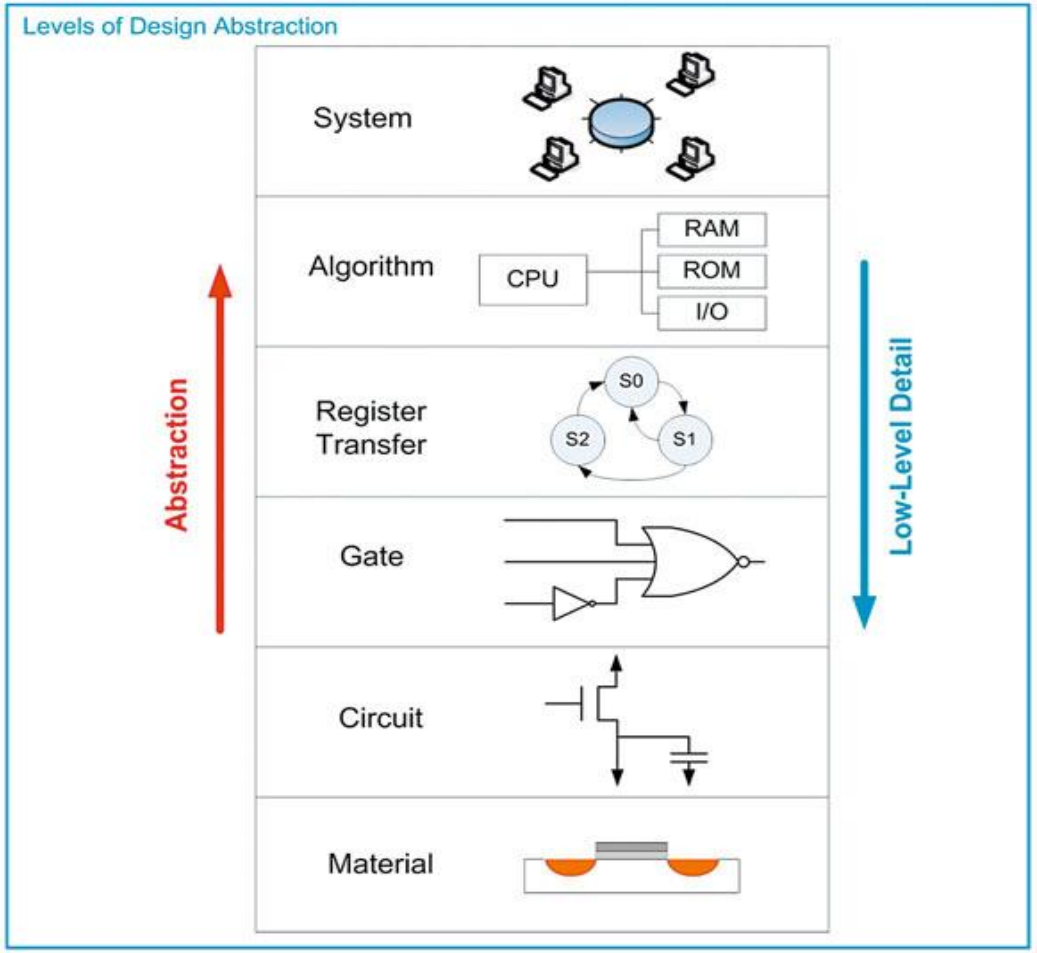
- HDLs have also evolved to support automated synthesis, which allows the CAD tools to take a functional description of a system (e.g., a truth table) and automatically create the gate-level circuitry to be implemented in real hardware.
- This need for schematics, which allows designers to focus their attention on designing the behavior of a system and not spend as much time performing the formal logic synthesis steps as in the classical digital design approach.

Modern Digital Design Flow Introduction

- There are two dominant hardware description languages in use today. They are VHDL and Verilog.
- VHDL stands for **v**ery high speed integrated circuit **h**ardware **d**escription **l**anguage. Verilog is not an acronym but rather a trade name.
- The use of these two HDLs is split nearly equally within the digital design industry. Once one language is learned, it is simple to learn the other language, so the choice of the HDL to learn first is somewhat arbitrary.

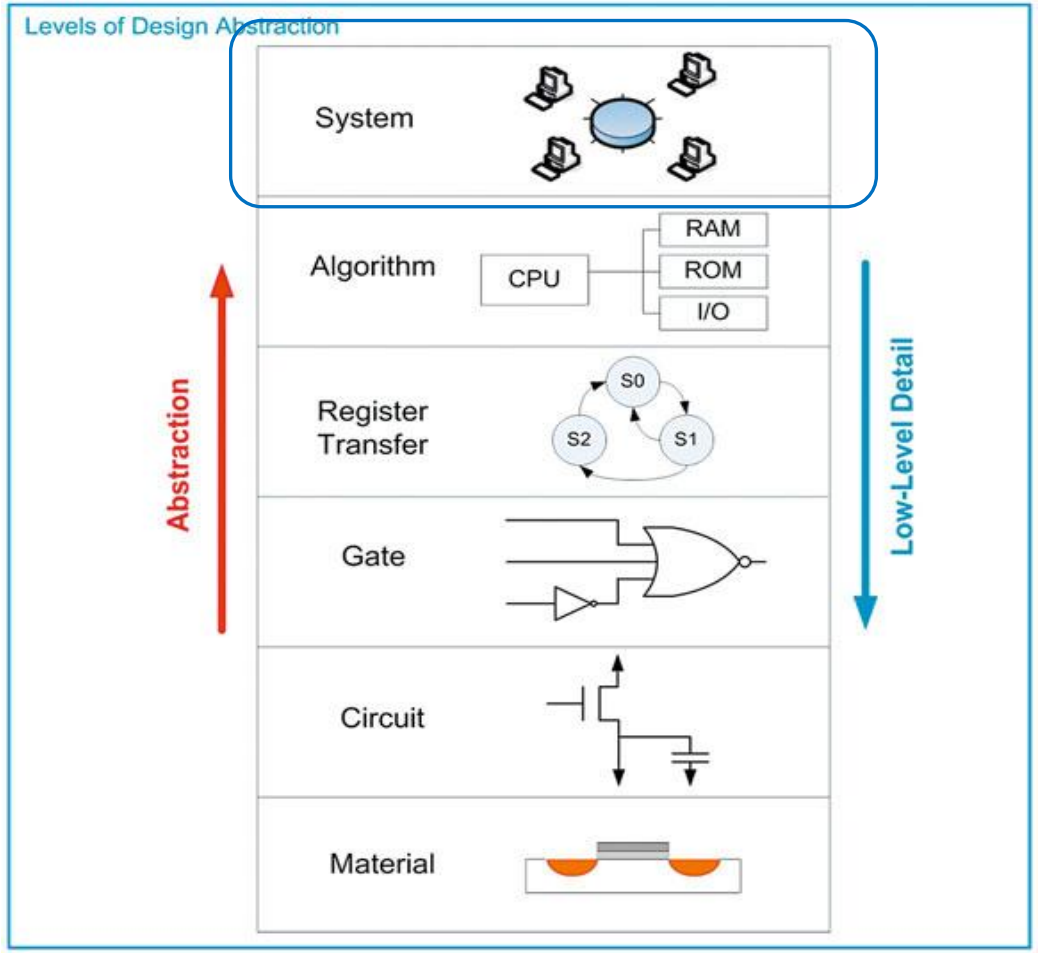
HDL Abstraction

- HDLs were originally defined to be able to model behavior at multiple levels of abstraction.
- Abstraction is an important concept in engineering design because it allows us to specify how systems will operate without getting consumed with implementation details.



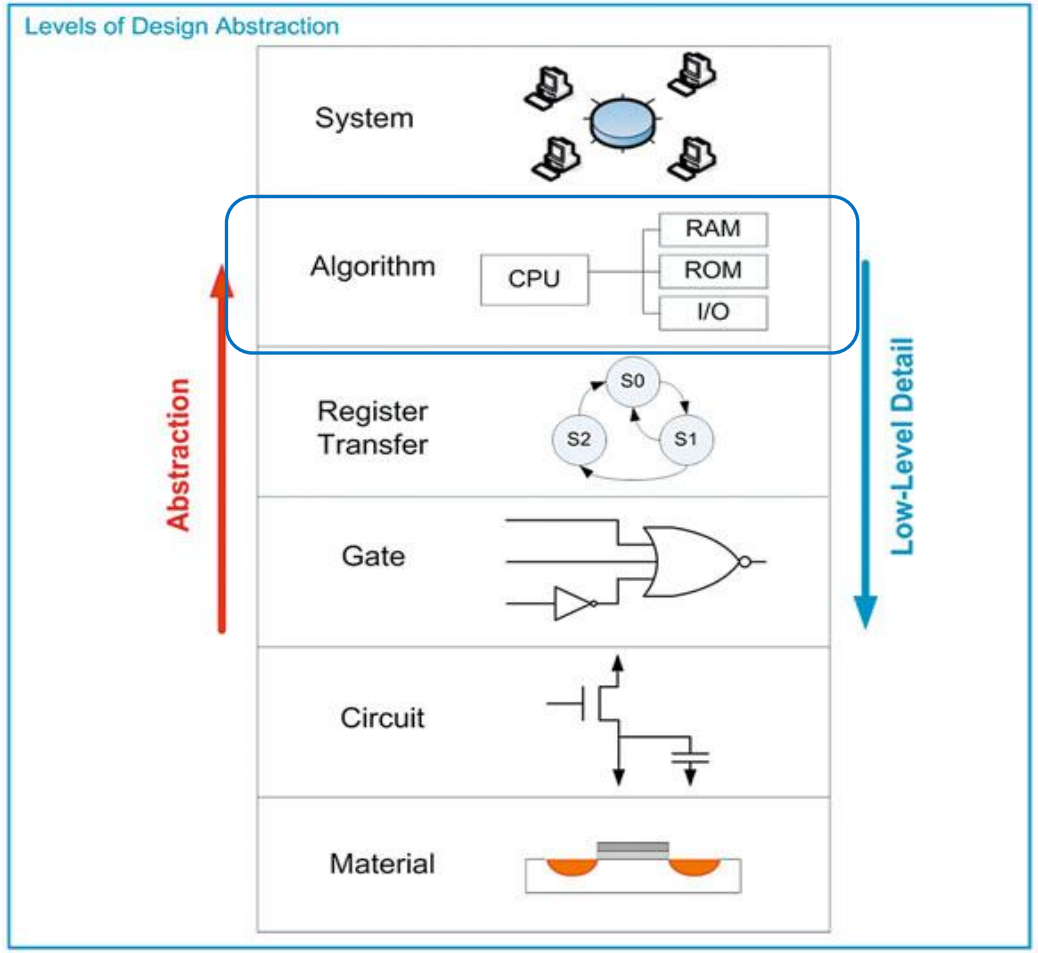
HDL Abstraction (System Level)

- The highest level of abstraction is the system level. At this level, behavior of a system is described by stating a set of broad specifications.
- An example of a design at this level is a specification such as “the computer system will perform 10 Tera Floating Point Operations per Second (10 TFLOPS) on double precision data and consume no more than 100 W of power.”
- Notice that these specifications do not dictate the lower-level details such as the type of logic family or the type of computer architecture to use.



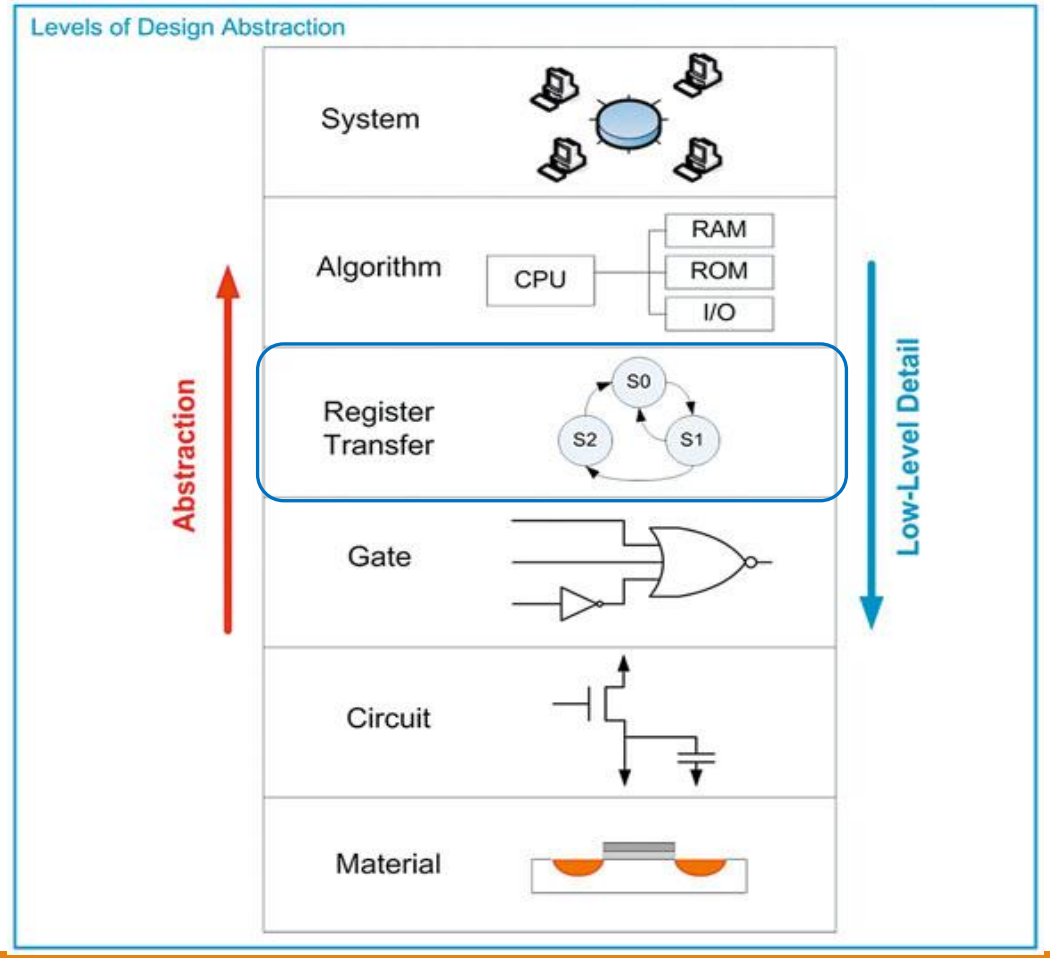
HDL Abstraction (Algorithmic Level)

- At algorithm level, the specifications begin to be broken down into subsystems, each with an associated behavior that will accomplish a part of the primary task.
- The example computer specifications might be broken down into subsystems such as a central processing unit (CPU) to perform the computation and random-access memory (RAM) to hold the inputs and outputs of the computation.



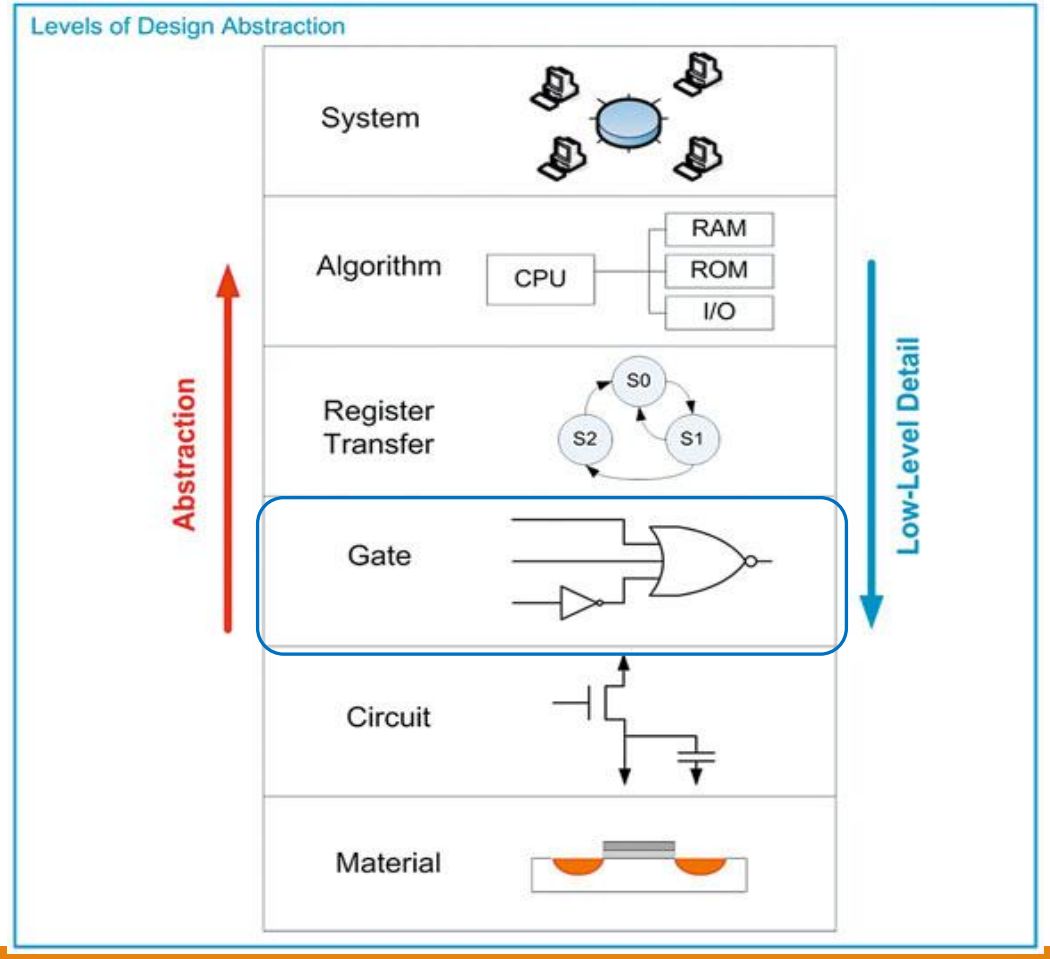
HDL Abstraction (Register Transfer Level)

- At Register Transfer (RTL) level, the details of how data is moved between and within subsystems are described in addition to how the data is manipulated based on system inputs.



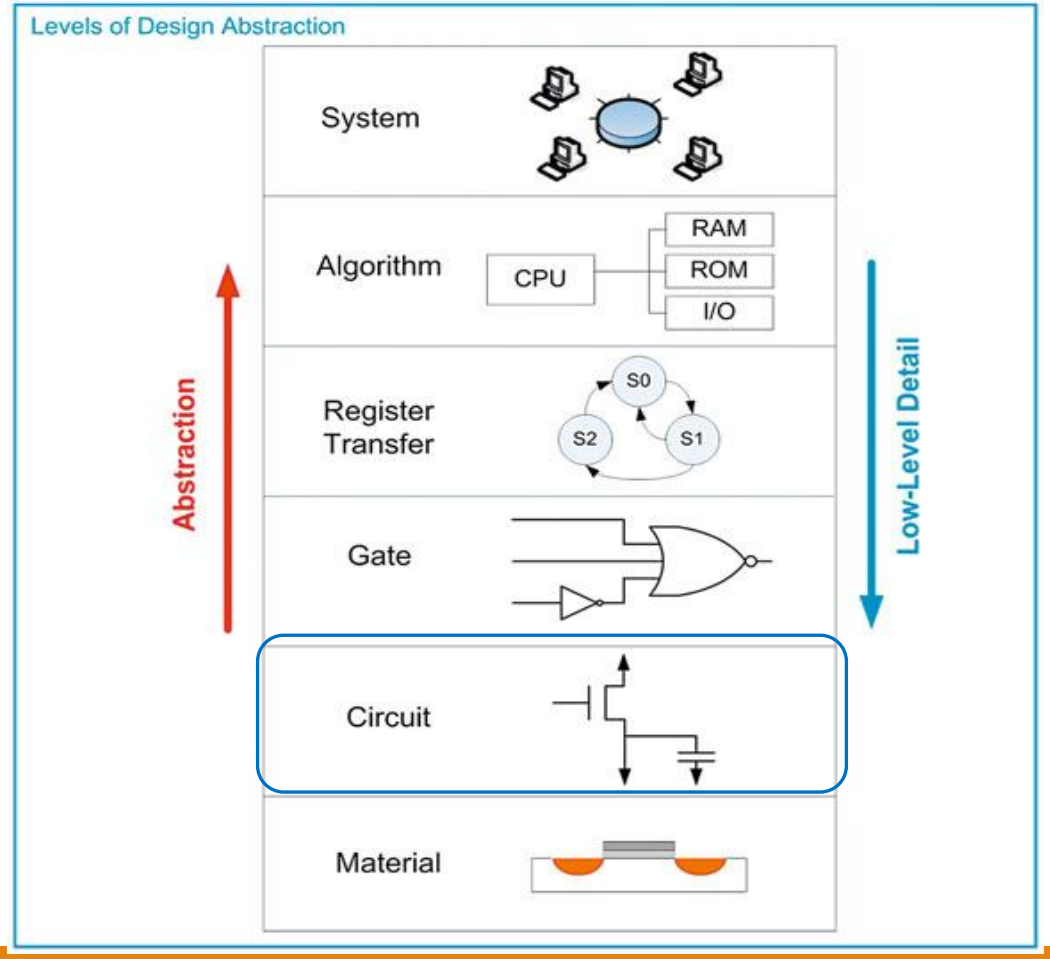
HDL Abstraction (Gate Level)

- At gate level, the design is described using basic gates and registers (or storage elements).
- The gate level is essentially a schematic (either graphically or text based)that contains the components and connections that will implement the functionality from the above levels of abstraction.



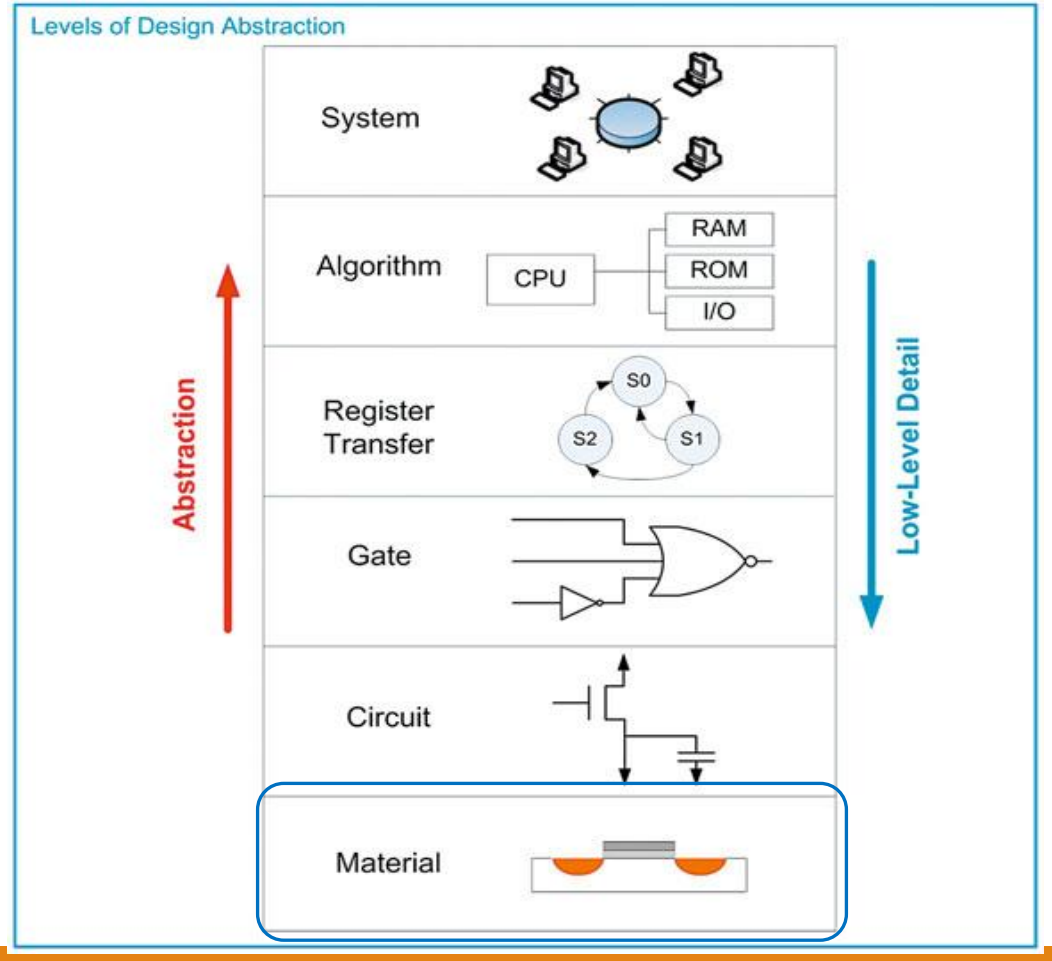
HDL Abstraction (Circuit Level)

- The circuit level describes the operation of the basic gates and registers using transistors, wires, and other electrical components such as resistors and capacitors.

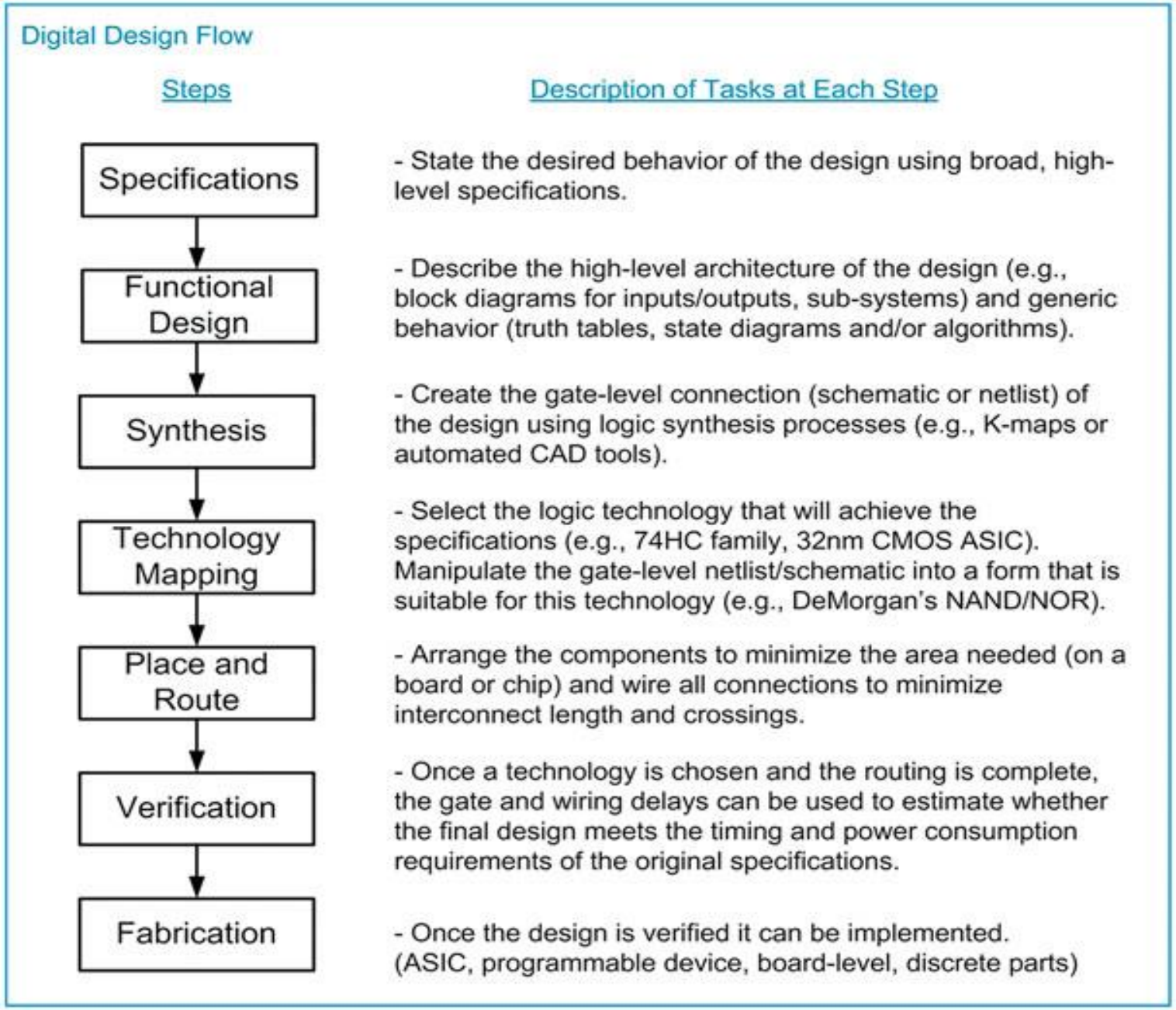


HDL Abstraction (Material Level)

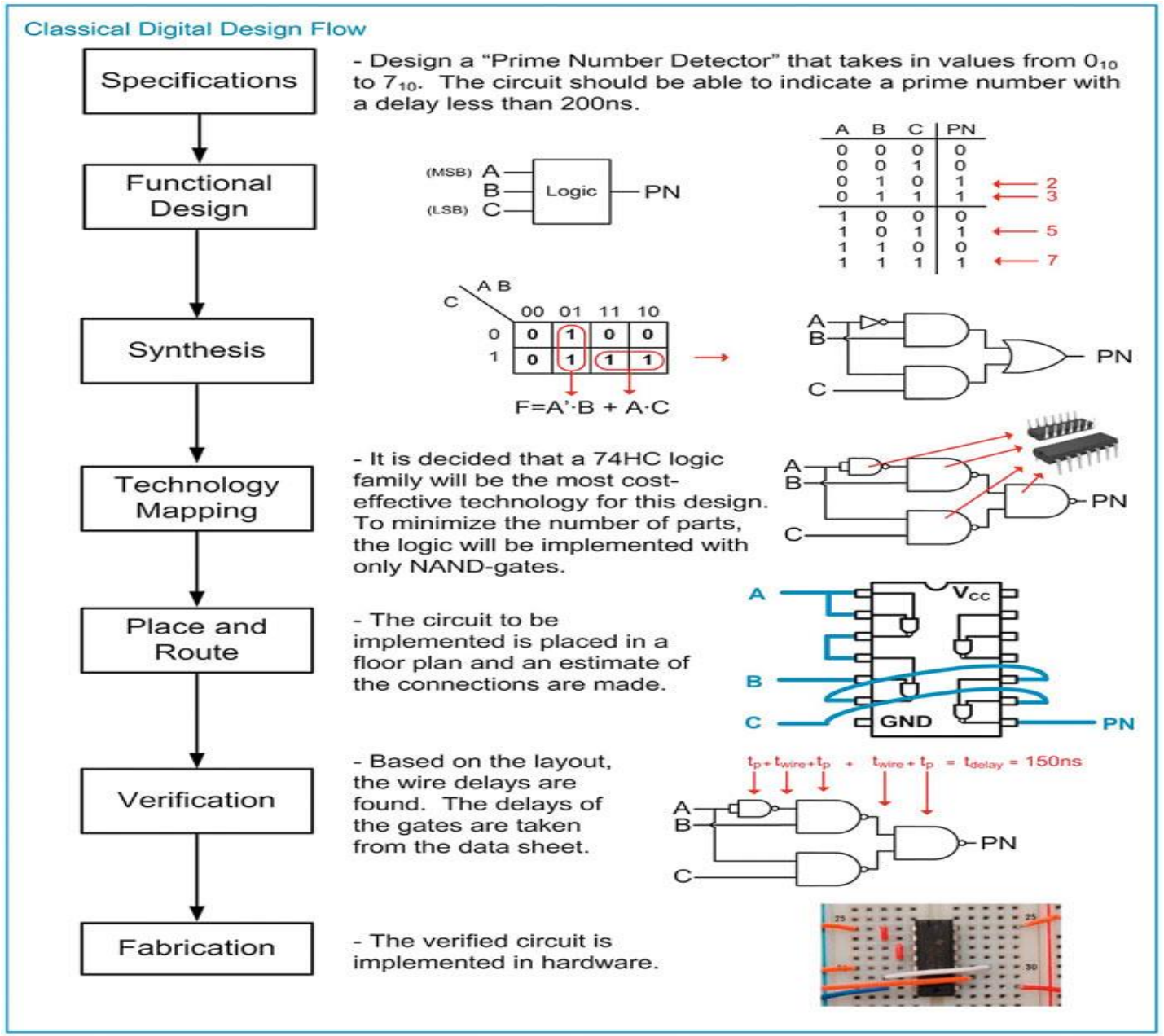
- This level describes how different materials are combined and shaped in order to implement the transistors, devices, and wires from the circuit level.
- HDLs are designed to model behavior at all of abstraction levels except for the material level.



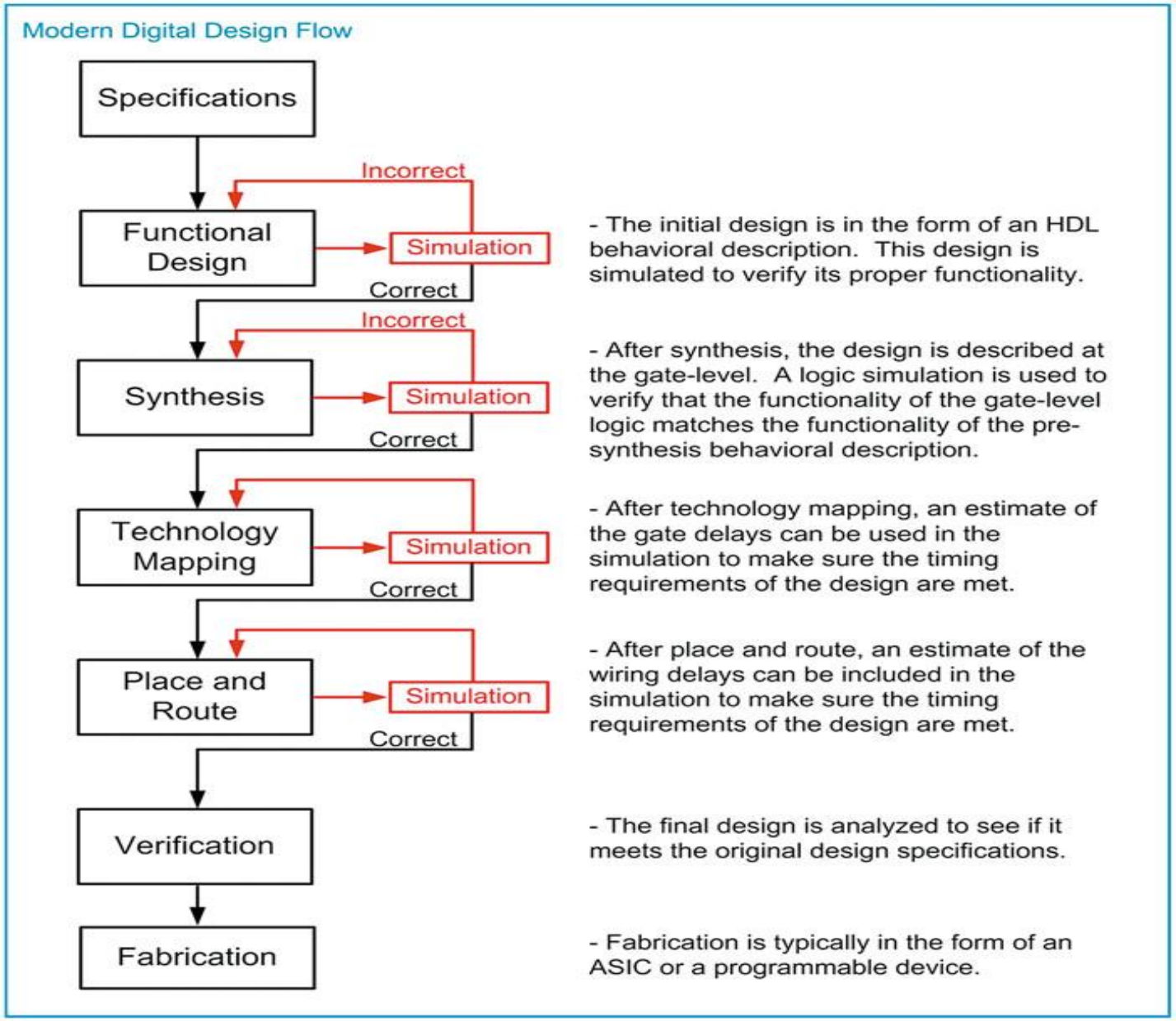
Digital Design Flow



Classical Digital Design Flow



Modern Design Flow



Digital Designs Move from Schematic Entry to HDLs

Why did digital designs move from schematic entry to text-based HDLs?

- HDL models could be much larger by describing functionality in text similar to traditional programming language.
- Schematics required sophisticated graphics hardware to display correctly.
- Schematics symbols became too small as designs became larger.
- Text was easier to understand by a broader range of engineers.

Introduction to Verilog

- The original Verilog standard (IEEE 1364) has been updated numerous times since its creation in 1995.
- The most significant update occurred in 2001, which was titled IEEE 1394-2001. In 2005, minor
- improvements were added to the standard, which resulted in IEEE 1394-2005.
- Verilog is case sensitive. Also, each Verilog assignment, definition, or declaration is terminated with a semicolon (;). As such, line wraps are allowed and do not signify the end of an assignment, definition, or declaration.

Introduction to Verilog

- Comments in Verilog are supported in two ways. The first way is called a line comment and is preceded with two slashes (i.e., //). The second comment approach is called a block comment and begins with /* and ends with a */
- All user-defined names in Verilog must start with an alphabetic letter, not a number.
- User-defined names are not allowed to be the same as any Verilog keyword.

Verilog Data Types

- In Verilog, every signal, constant, variable, and function must be assigned a data type.
- The IEEE 1394-2005 standard provides a variety of predefined data types.
- Some data types are synthesizable, while others are only for modeling abstract behavior.

Value Set

- Verilog supports four basic values that a signal can take on: 0, 1, X, and Z.
- Most of the predefined data types in Verilog store these values.

Value	Description
0	A logic zero, or false condition.
1	A logic one, or true condition.
x or **X**	Unknown or uninitialized.
z or **Z**	High impedance, tri-stated, or floating.

Value Set

- In Verilog, these values also have an associated strength.
- The strengths are used to resolve the value of a signal when it is driven by multiple sources.

If the strength is not specified, it will default to strong drive, or level 6.

Strength	Description	Strength level
supply1	Supply drive for V_{CC}	7
supply0	Supply drive for V_{SS}, or GND	7
strong1	Strong drive to logic one	6
strong0	Strong drive to logic zero	6
pull1	Medium drive to logic one	5
pull0	Medium drive to logic zero	5
large	Large capacitive	4
weak1	Weak drive to logic one	3
weak0	Weak drive to logic zero	3
medium	Medium capacitive	2
small	Small capacitive	1
highz1	High impedance with weak pull-up to logic one	0
highz0	High impedance with weak pull-down to logic zero	0

Net Data Types

- Every signal within Verilog must be associated with a data type.
- A net data type is one that models an interconnection (aka, a net) between components and can take on the values 0, 1, X, and Z.
- A signal with a net data type must be driven at all times and updates its value when the driver value changes.
- The most common synthesizable net data type in Verilog is the wire.

Type	Description
wire	A simple connection between components.
wor	Wired-OR. If multiple drivers, their values are OR'd together.
wand	Wired-AND'd. If multiple drivers, their values are AND'd together.
supply0	Used to model the V_{SS}, (GND), power supply (supply strength inherent).
supply1	Used to model the V_{CC} power supply (supply strength inherent).
tri	Identical to **wire**. Used for readability for a net driven by multiple sources.
trior	Identical to **wor**. Used for readability for nets driven by multiple sources.
triand	Identical to **wand**. Used for readability for nets driven by multiple sources.
tri1	Pulls up to logic one when tri-stated.
tri0	Pulls down to logic zero when tri-stated.
trireg	Holds last value when tri-stated (capacitance strength inherent).

Variable Data Types

- Verilog also contains data types that model storage. These are called variable data types.
- A variable data type can take on the values 0, 1, X, and Z, but does not have an associated strength.
- Variable data Types will hold the value assigned to them until their next assignment.

Type	Description
reg	A variable that models logic storage. Can take on values 0, 1, X, and Z.
integer	A 32-bit, 2's complement variable representing whole numbers between $-2{,}147{,}483{,}648_{10}$ and +2,147,483,647.
real	A 64-bit, floating point variable representing real numbers between $-(2.2 \times 10^{-308})_{10}$ and $+(2.2 \times 10^{308})_{10}$.
time	An unsigned, 64-bit variable taking on values from 0_{10} to $+(9.2 \times 10^{18})$.
realtime	Same as **time**. Just used for readability.

Vectors

- In Verilog, a vector is a one-dimensional array of elements. All of the net data types, in addition to the variable type reg, can be used to form vectors. The syntax for defining a vector is as follows:

  <type> [<MSB_index>:<LSB_index>] vector_name

- While any range of indices can be used, it is common practice to have the LSB index start at zero.Example:

  wire [7:0] Sum; // This defines an 8-bit vector called “Sum” of type wire. The
  // MSB is given the index 7 while the LSB is given the index 0.

  reg [15:0] Q; // This defines a 16-bit vector called “Q” of type reg.

- Individual bits within the vector can be addressed using their index. Groups of bits can be accessed using an index range.

  Sum [0]; // This is the least significant bit of the vector “Sum” defined above.

  Q [15:8]; // This is the upper 8-bits of the 16-bit vector “Q” defined above.

Arrays

- An array is a multidimensional array of elements. This can also be thought of as a “vector of vectors.” Vectors within the array all have the same dimensions.
- To declare an array, the element type and dimensions are defined first followed by the array name and its dimensions. It is common practice to place the start index of the array on the left side of the “:” when defining its dimensions. The syntax for the creation of an array is shown below.

Example:

```
reg[7:0] Mem[0:4095];      // Defines an array of 4096, 8-bit vectors of type reg.
integer A[1:100];          // Defines an array of 100 integers.
```

Arrays

- When accessing an array, the name of the array is given first, followed by the index of the element. It is also possible to access an individual bit within an array by adding appending the index of element.

Example:

```
Mem[2];        // This is the 3rd element within the array named “Mem”.
               // This syntax represents an 8-bit vector of type reg.

Mem[2][7];     // This is the MSB of the 3rd element within the array named “Mem”.
               // This syntax represents a single bit of type reg.

A[2];          // This is the 2nd element within the array named “A”. Recall
               // that A was declared with a starting index of 1.
               // This syntax represents a 32-bit, signed integer.
```

Expressing Numbers Using Different Bases

- If a number is simply entered into Verilog without identifying syntax, it is treated as an integer.
- Verilog also supports an optional bit size and sign of a number.
- When defining the value of arrays, the "_" can be inserted between numerals to improve readability. The "_" is ignored by the Verilog compiler.
- Values of numbers can be entered in either upper or lower case (i.e., b or B, f or F). The syntax for specifying the base of a number is as follows:

  <size_in_bits>'<base><value>
- Note that specifying the size is optional. If it is omitted, the number will default to a 32-bit vector with leading zeros added as necessary.

Syntax	Description
'b	Unsigned binary.
'o	Unsigned octal.
'd	Unsigned decimal.
'h	Unsigned hexadecimal.
'sb	Signed binary.
'so	Signed octal.
'sd	Signed decimal.
'sh	Signed hexadecimal.

Expressing Numbers Using Different Bases

Example:

```
10              // This is treated as decimal 10, which is a 32-bit signed vector.
4'b1111         // A 4-bit number with the value 1111_2.
8'b1011_0000    // An 8-bit number with the value 10110000_2.
8'hFF           // An 8-bit number with the value 11111111_2.
8'hff           // An 8-bit number with the value 11111111_2.
6'hA            // A 6-bit number with the value 001010_2. Note that leading zeros
                // were added to make the value 6-bits.
8'd7            // An 8-bit number with the value 00000111_2.
32'd0           // A 32-bit number with the value 0000_0000_16.
'b1111          // A 32-bit number with the value 0000_000F_16.
8'bZ            // An 8-bit number with the value ZZZZ_ZZZZ.
```

Assigning Between Different Types

- Verilog is said to be a weakly typed (or loosely typed) language, meaning that it permits assignments
- between different data types.
- The reason Verilog permits assignment between different types is because it treats all of its types as just groups of bits.
- When assigning between different types, Verilog will automatically truncate or add leading bits as necessary to make the assignment work.

Example:

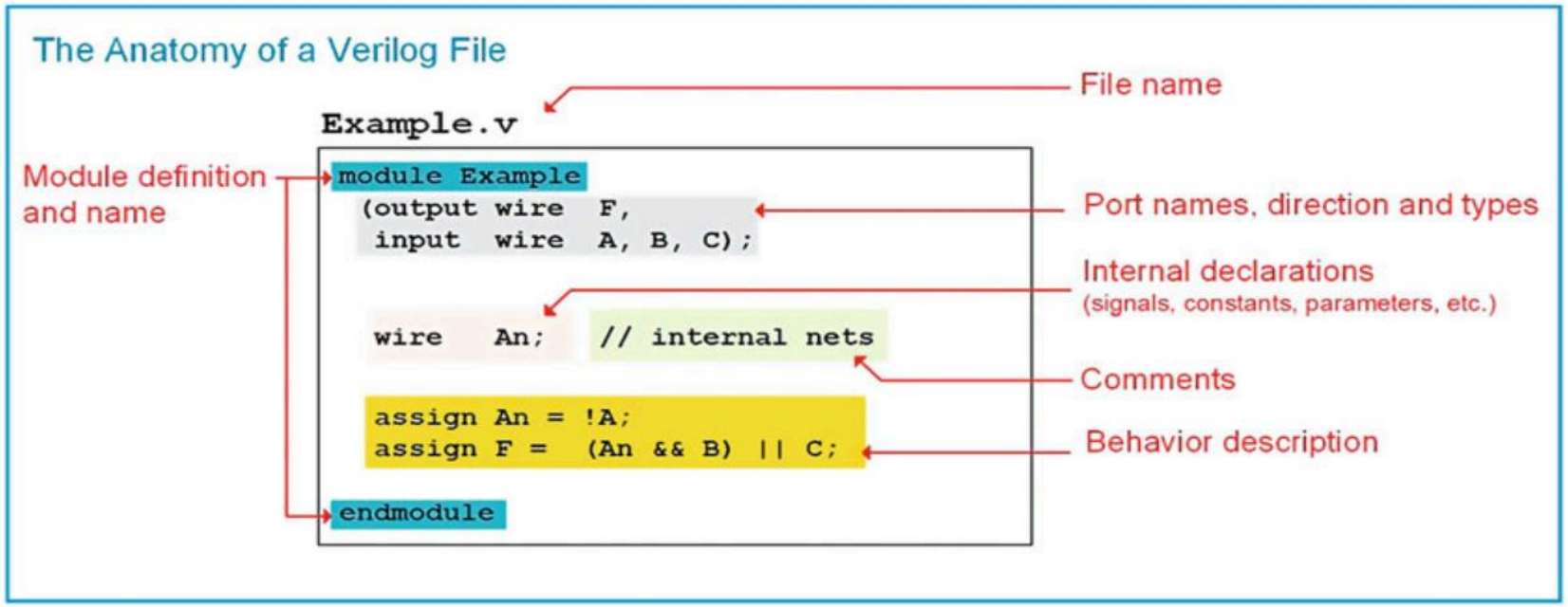
Assume that a variable called ABC_TB has been declared as type reg[2:0].

```
ABC_TB  = 2'b00;      // ABC_TB will be assigned 3'b000. A leading bit is automatically added.
ABC_TB = 5;           // ABC_TB will be assigned 3'b101. The integer is truncated to 3-bits.
ABC_TB = 8;           // ABC_TB will be assigned 3'b000. The integer is truncated to 3-bits.
```

Verilog Module Construction

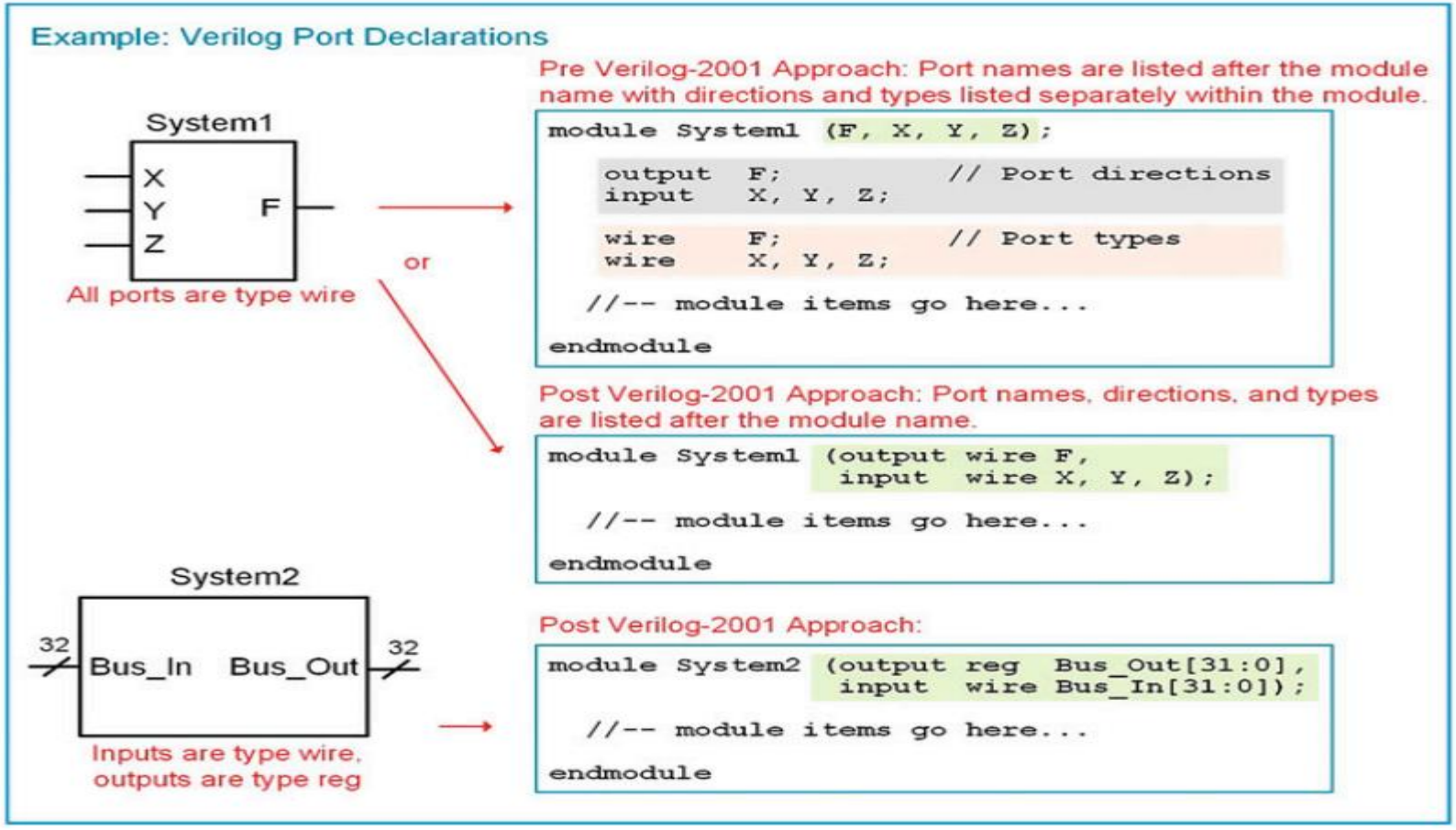
The module includes the interface to the system (i.e., the inputs and outputs) and the description of the behavior.

When working on large designs, it is common practice to place each module in its own file with the same name



Port Definitions

The port directions are declared to be one of the three types: **input**, **output**, and **inout**.



Signal Declarations

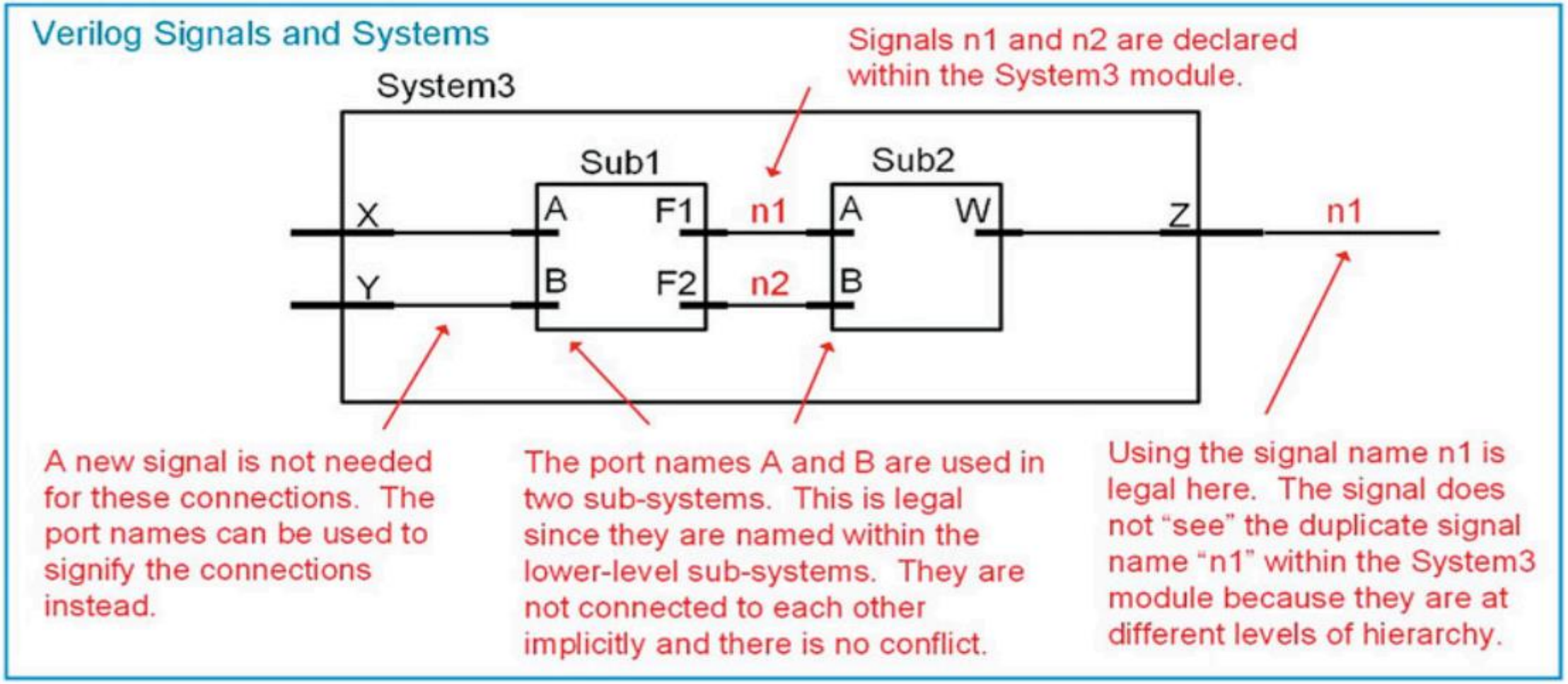
The syntax for a signal declaration is as follows:

```
<type> name;
```

Example:

```
wire    node1;          // declare a signal named "node1" of type wire
reg     Q2, Q1, Q0;     // declare three signals named "Q2", "Q1", and "Q0", all
                           of type reg
wire    [63:0] bus1;    // declare a 64-bit vector named "bus1" with all bits of type
                           wire
integer i,j;            // declare two integers called "i" and "j"
```

Signal Declarations Cont.



Parameter Declarations

The syntax for declaring a parameter is as follows:

```
parameter <type> constant_name = <value>;
```

Example:

Note that the type is optional and can only be integer, time, real, or realtime

```
parameter BUS_WIDTH = 64;
parameter NICKEL    = 8'b0000_0101;
```

Once declared, the constant name can be used throughout the module. The following example illustrates how we can use a constant to define the size of a vector. Notice that since we defined the constant to be the actual width of the vector (i.e., 32-bits), we need to subtract one from its value when defining the indices (i.e., [31:0]).

Example:

```
wire [BUS_WIDTH-1:0] BUS_A;     // It is acceptable to add a "space" for readability
```

Compiler Directives

A compiler directive provides additional information to the simulation tool on how to interpret the Verilog model.

Syntax	Description
`timescale <unit>, <precision>	Defines the timescale of the delay unit and its smallest precision.
`include <filename>	Includes additional files in the compilation.
`define <macroname> <value>	Declares a global constant.

Example:

```
`timescale 1ns/1ps   // Declares the unit of time is 1 ns with a precision of 1ps.
                     // The precision is the smallest amount that the time can
                     // take on. For example, with this directive the number
                     // 0.001 would be interpreted as 0.001 ns, or 1 ps.
                     // However, the number 0.0001 would be interpreted as 0 since
                     // it is smaller than the minimum precision value.
```