

INTRODUCTION TO VLSI DESIGN

1.1 INTRODUCTION

The word digital has made a dramatic impact on our society. More significant is a continuous trend towards digital solutions in all areas –from electronic instrumentation, control, data manipulation, signals processing, tele communications, etc., to consumer electronics. Development of such solutions has been possible due to good digital system design and modeling techniques.

MICROELECTRONIC EVOLUTION

Year	1947	1950	1961	1966	1971	1980	1990	2000
Technology	Inven ⁿ	Discrete	SSI	MSI	LSI	VLSI	ULSI	GSI
Approximate Comp On Chip	1	1	10	100-1000	1000-20000	20000-1 Million	1 Million-10 Million	> 10 Million
Typical Products	-	Diodes Transistors	Logic Gates F/F	Counter Mux Adders	8-Bit MP ROM RAM	16-32 Bit MP & Sophisticated Peripherals	Special Proces sors M/C	

1.1.1 VLSI DESIGN

The complexity of VLSI being designed and designed and used today makes the manual approach to design impractical. Design automation is the order of the day. With the rapid technological developments in the last two decades, the status of VLSI technology is characterized by the following:

- ❖ A steady increase in the size and hence the functionality of the ICs.
- ❖ A steady reduction in feature size and hence increase in the speed of operation as well as gate or transistor density.
- ❖ A steady improvement in the predictability of circuit behavior.
- ❖ A steady increase in the variety and size of software tools for VLSI design.

The above developments have resulted in a proliferation of approaches to VLSI design.

1.1.2 VLSI DESIGN FLOW

The design process, at various levels, is usually evolutionary in nature. It starts with a given set of requirements. Initial design is to be developed and test impact analyst must be considered. The Y-chart (first introduced by D. Gajski) is shown in below figure1 illustrates a design flow for most logic chips, using design activities on the three different axes (domains). Y chart of three major domains, they are:

- Behavioral domain
- Structural domain
- Geometrical layout domain

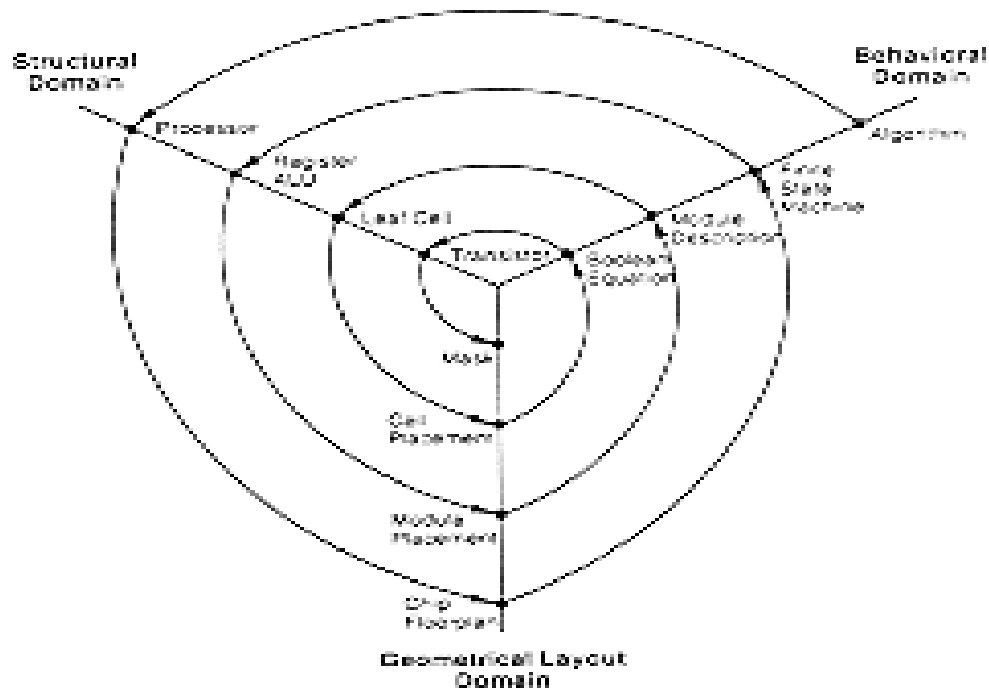


Figure 1 Typical VLSI design flow in three domains (Y-chart representation)

The design flow starts from the algorithm that describes the behavior of the target chip. The corresponding architecture of the processor is first defined. It is mapped onto the chip surface by floor planning. The next design evolution in the behavioral domain defines finite state machines (FSMs) which are structurally implemented with functional modules such as registers and arithmetic logic units (ALUs). These modules are then geometrically placed onto the chip surface using CAD tools for automatic module placement followed by routing, with a goal of minimizing inter- connects area and signal delays.

The third evolution starts with a behavioral module description. Individual modules are then implemented with leaf cells. At this stage the chip is described in terms of logic gates (leaf cells), which can be placed and interconnected by using a cell placement & routing program. The last evolution involves a detailed Boolean description of leaf cells followed by a transistor level implementation of leaf cells and mask generation. In standard-cell based design, leaf cells are already pre-designed and stored in a library for logic design use.

1.1.3 ABSTRACTION MODEL

The model divides the whole design cycle into various domains (see figure 2) with such an abstraction through a division process the design is carried out in different layers. The designer at one layer can function without bothering about the layers above or below. The thick horizontal lines separating the layers in the figure signify the compartmentalization. As an example, let us consider design at the gate level. The circuit to be designed would be described in terms of truth tables and static tables. With these as available inputs, he has to express them as Boolean logic equation and realize them in terms of gates and flip-flops. In turn these form the inputs to the layer immediately below. Compartmentalization of the approach to design in the manner described here is the essence of abstraction; it is the basics for the development and use of CAD tools in the design at various levels.

1.1.4 ASIC DESIGN FLOW

As with any other technical activity, development of an ASIC starts with an idea and takes tangible shape through the stages of development as shown in figure 3 and shown in detail in figure 4. The first step in the process is to expand the idea in terms of behavior of the target circuit. Through stages of programming, the same is fully developed into a design description- in terms of well defined standard constructs and conventions.

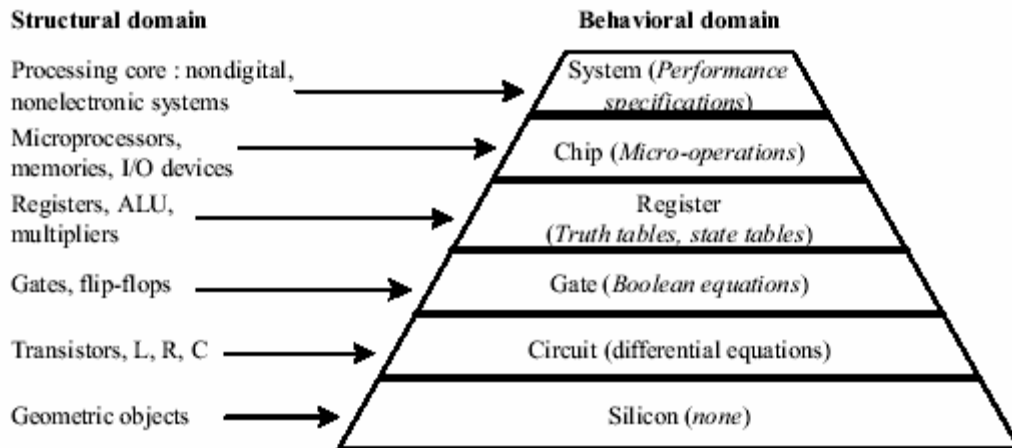


Figure 2 Design domain and levels of abstraction

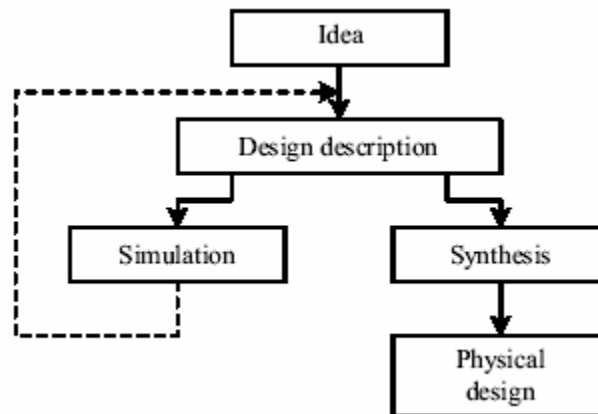


Figure 3 Major activities in ASIC design

The design is tested through a simulation process; it is to check, verify, and ensure that what is wanted is what is described. Simulation is carried out through dedicated tools .with every simulation results are studied to identify errors in the design description. The errors are corrected and another simulation run is carried out. Simulation and changes to design description together form a cyclic iterative process, repeated until an error –free design is evolved.

Design description is an activity independent of the target technology or manufacturer. It results in a description of the digital circuit. To translate it into a tangible circuit, one goes through the physical design process. The same constitutes a set of activities closely linked to the manufacturer and the target technology.

1.1.5 DESIGN DESCRIPTION

The design is carried out in stages. The process of transforming the idea into a detailed circuit description in terms of the elementary circuit components constitutes design description. The final circuit of such an IC can have up to a billion such components; it is arrived in a step-by-step manner.

The first step in evolving the design description is to describe the circuit in terms of its behavior. The description looks like a program in a high level language like C. once the behavior level design description is ready, it is tested extensively with the help of simulation tool; it checks and confirms that all the expected functions are carried out satisfactorily. If necessary, this behavioral level routine is edited, modified, and rerun – all done manually. Finally, one has a design for the expected system- described at the behavioral level. The behavioral constructs not supported by the synthesis tools replaced by data flow and gate level constructs. To surmise, the designer has to develop synthesizable codes for his design.

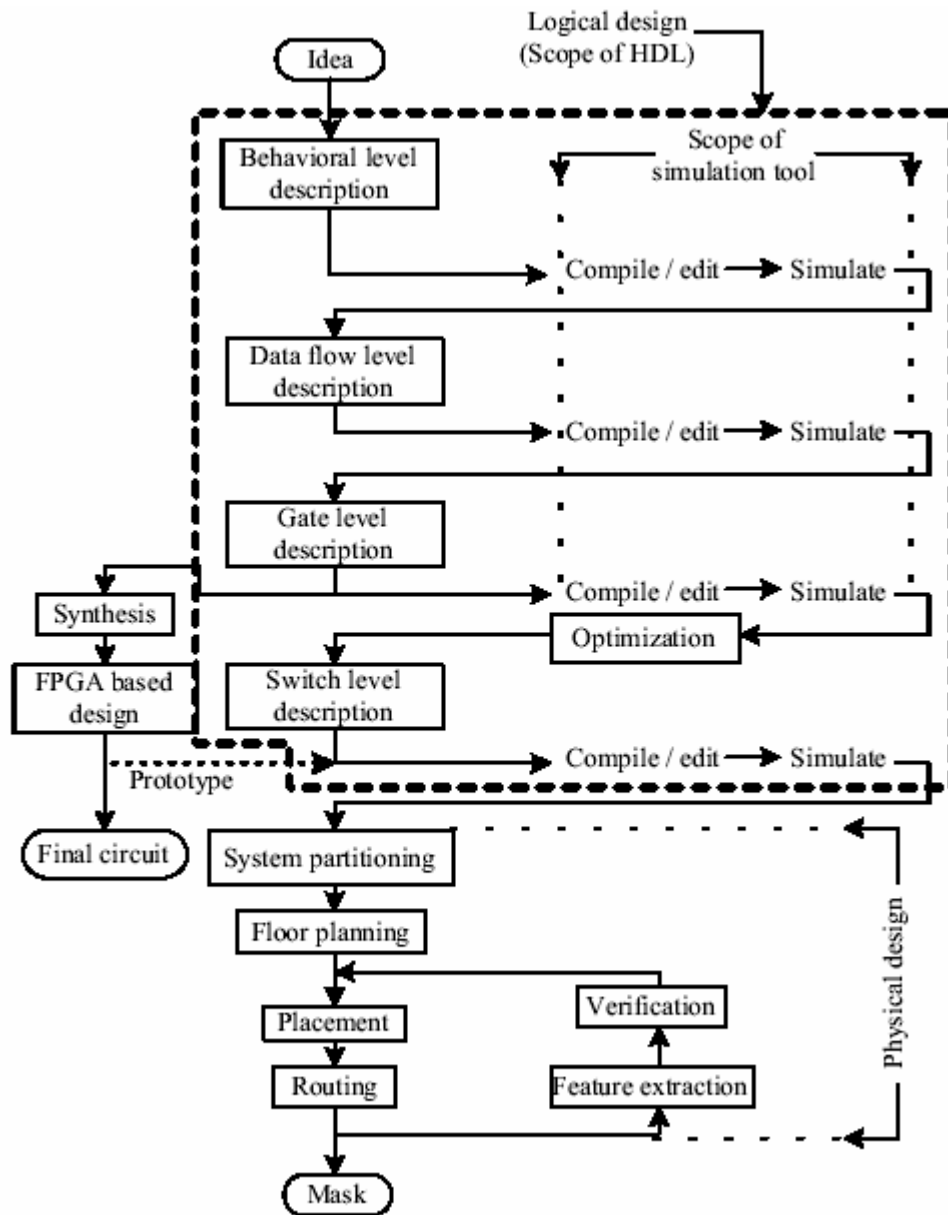


Figure 4 ASIC design and development flow.

The design at the behavioral level is to be elaborated in terms of known and acknowledged functional blocks. It forms the next detailed level of design description. Once again the design is to be tested through simulation and iteratively corrected for errors. The elaboration can be continued one or two steps further. It leads to a detailed design description in terms of logic gates and transistor switches.

1.1.6 OPTIMIZATION

The circuit at the gate level- in terms of the gates and flip-flops- can be redundant in nature. The same can be minimized with the help of minimization tools. The minimized design is converted to a circuit in terms of the switch levels cells from standard libraries provided by the foundries. The cell based design generated by the tool is the last step in the design process; it forms the input to the first level of physical design.

1.1.7 POST LAYOUT SIMULATION

Once the placement and routing are completed the performance specifications like silicon area, power consumed, path delays, can be computed. Equivalent circuit can be extracted at the component level and the performance analysis carried out. This constitutes the final stage called “verification”. One may have to go through the placement and routing activity once again to improve performance.

1.1.8 CRITICAL SUBSYSTEMS

The design may have critical subsystems. Their performance may be crucial to the overall performance; in other words, to improve the system performance substantially, one may have to design such subsystems afresh. The design here may imply redefinition of the basic feature size of the component, component design, and placement of the components, or routing done separately and specifically for the sub systems. A set of masks used in the foundry may have to be done a fresh for the purpose.

1.2 EMERGENCE OF HDLs

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital

circuits. Thus, Hardware Description Languages (HDLs) come into existence. HDLs allowed the designers to model the concurrency of process found in hardware elements.

1.2.1 IMPORTANCE OF HDLs

Designers can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific technology. Logic synthesis tools can automatically convert the design to any fabrication technology.

By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits.

1.2.2 HARDWARE DESCRIPTIVE LANGUAGE:

There are two main hardware descriptive languages in use in the industry today for Very Large Scale Integration (VLSI) of chips. They are:

- **Verilog HDL**
- **VHDL**

1.2.3 OVERVIEW OF VERILOG HDL:

Verilog HDL is a Hardware Description Language (HDL). A Hardware Description Language is a language used to describe a digital system, for example, a computer or a component of a computer. One may describe a digital system at several levels. For example, an HDL might describe the layout of the wires, resistors and transistors on an Integrated Circuit (IC) chip, i.e., and the switch level. Or, it might describe the logical gates and flip flops in a digital system, i.e., the gate level. An even higher level describes the registers and the transfers of vectors of information between registers. This

is called the Register Transfer Level (RTL). Verilog supports all of these levels. The industry is currently split on which is better. Many feel that Verilog is easier to learn and use than VHDL.

Verilog was introduced in 1985 by Gateway Design System Corporation, now a part of Cadence Design Systems, Inc.'s Systems Division.

Verilog HDL allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i.e. , gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. A primary use of HDLs is the simulation of designs before the designer must commit to fabrication.

1.2.4 POPULARITY OF VERILOG HDL:

- ❖ Verilog HDL is a general purpose HDL that is easy to use and learn. It is similar in syntax to the C programming language.

- ❖ Verilog HDL allows different levels of abstraction to be mixed in the same model.

- ❖ Most popular logic synthesis tool support Verilog HDL. This makes it the language of choice for designers.

All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.

1.2.5 OVERVIEW OF VHDL:

As the size and the complexity of digital system increases, more computer aided design tools are introduced into the hardware design process. The early papered pencil design methods have given way to sophisticated design entry, verification and automatic hardware generation tools. The newest addition to this design methodologies the

introduction of hardware description language (HDL). Actually the use of this language is not new languages such as CDI, ISP and AHPL have been used for last some years. However, their primary application has been the verification of designs architecture. They do not have the capability to model design with a high degree of accuracy that is, their timing model is not precise and/or their language construct implies a certain hardware structure newer languages such as VHDL have more universal timing models and imply no particular hardware structure.

Hardware description languages have two main applications documenting a design and modeling it. Good documentation of a design helps to ensure design accuracy and design portability. Since a simulator supports them inherent in a HDL description can be used to validate a design. Prototyping of complicated system is extremely expensive, and the goal of those concerned with the development of hardware languages is to replace this prototyping process with validation through simulation and silicon compilation.

Once an entity has been modeled, it needs to be validated by the VHDL system. A typical VHDL system consists of an analyzer and a simulator. The analyzer reads in one or more design units contained in a single file and compiles them into a design library after validating the syntax and performing some static semantic checks. The design library is a place in the host environment where compiled design units are stored.

The simulator simulates an entity, represented by an entity-architecture pair or by a configuration, by reading in its compiled description from the design library & then performing the following steps:

1. Elaboration
2. Initialization
3. Simulation

VHDL is an acronym for VHSIC Hardware description language (VHSIC is an acronym for very high speed integrated circuits). It is a hardware description language that can be

used to model a digital system at many levels of abstraction, ranging from the algorithmic level to the gate level.

The complexity of a digital system being modeled could vary from that of simple gate to a complete digital electronic system, or anything in between.

The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description.

The VHDL language can be regarded as an integrated amalgamation of the following languages.

- Sequential language.
- Concurrent language.
- Net list language.
- Timing specifications.
- Waveform generation language.

Therefore, the language has constructs that enable you to express the concurrent or sequential behavior of a digital system as an interconnection of components. All the above constructs may be combined to provide a comprehensive description of the system in a single model.

The language not only defines the syntax but also defines very clear simulation semantics for each language construct. Therefore models written in this language can be verified using a VHDL simulator. It inherits many of its features especially the sequential part, from the Ada programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand, fortunately, it is possible to quickly assimilate a core subset of the language that is both easy and simple to understand without learning the more complex features. The complete language however has sufficient power to capture the descriptions of the most complex chips to a complete electronic system.

1.3 BASIC CONCEPTS OF VERILOG:

1.3.1 LEXICAL CONVENTIONS

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, string, identifiers, and keywords. Verilog HDL is a case-sensitive language. All keywords are in low case.

1.3.2 WHITESPACE:

Blank spaces (\b), tabs (\t) and newlines (\n) comprise the whitespace. Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

1.3.3 COMMENTS:

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with “//”. Verilog skips from that point to the end of line. A multiple-line comment starts with “/*” and ends with “*/”. Multiple-line comments cannot be nested.

```
a = b && c; // This is a one-line comment

/* This is a multiple line
   comment */

/* This is /* an illegal */ comment */
```

1.3.4 OPERATORS:

Operators are of three type's unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

1.3.5 NUMBER SPECIFICATION

There are two types of number specification in verilog: sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format> <number>.

<Size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). the number is specified as consecutive digits from 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f. only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

```
4'b1111 // This is a 4-bit binary number
12'habc // This is a 12-bit hexadecimal number
16'd255 // This is a 16-bit decimal number.
```

UNSIZED NUMBERS:

Numbers that are specified without a <base format > specification are decimal numbers by default. Numbers that are written without <size> specifications have a default number of bits is simulator – and machine –specific (must be at least 32).

```
23456 // This is a 32-bit decimal number by default
'hc3 // This is a 32-bit hexadecimal number
'o21 // This is a 32-bit octal number
```

X OR Z VALUES:

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an **x**. a high impedance value is denoted by **z**.

```
12'h13x // This is a 12-bit hex number; 4 least significant bits unknown
6'hx // This is a 6-bit hex number
32'bz // This is a 32-bit high impedance number
```

An **x** or **z** values sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is **0**, **x**, **z**, the number is automatically extended to fill the most significant bits, respectively, with **0**, **x**, or **z**. This makes it easy to assign **x** or **z** to whole vector. If the most significant digit is 1, then it is also zero extended.

NEGATIVE NUMBERS:

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format > and <number>.

```
-6'd3 // 8-bit negative number stored as 2's complement of 3
4'd-2 // Illegal specification
```

UNDERSCORE CHARACTERS AND QUESTION MARKS:

An underscore character “_” is allowed anywhere in a number except the first character. Underscore character are allowed only to improve readability of number and are ignored by Verilog.

A question mark “?” is the Verilog HDL alternative for **z** in the context of numbers. The ? is used to enhance readability in the **case x** and **case z** statements behavioral modeling, where the high impedance value is a don't care condition.

```
12'b1111_0000_1010 // Use of underline characters for readability
4'b10?? // Equivalent of a 4'b10zz
```

1.3.6 STRINGS:

A string is a sequence of character that is enclosed by double quotes. The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

```
"Hello Verilog World" // is a string
"a / b" // is a string
```

1.3.7 IDENTIFIERS AND KEYWORDS:

Keywords are special identifiers reserved to define the language constructs. Keywords are in lowercase. Identifiers are names given to objects so that they can be referenced in the design. Identifiers are made up of alphanumeric characters, the underscore (_) and the dollar sign (\$) and are low case sensitive. Identifier starts with an alphabetic character or an underscore. They cannot start with a number or a \$ sign.

```
reg value; // reg is a keyword; value is an identifier
input clk; // input is a keyword, clk is an identifier
```

1.3.8 ESCAPED IDENTIFIERS:

Escaped identifiers begin with the backslash (\) character and end with white space (space, tab, or new line).all characters between backslash and whitespace are processed

literally. Any printable ASCII character can be included in escaped identifiers. The backslash or whitespace is not considered a part of the identifier.

```
\a+b-c  
\**my_name**
```

1.4 DATA TYPES:

1.4.1 VALUE SET


Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in table 1.

Table 1 Value levels

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

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value level **0** and **1** can have the strength levels listed in table 2.

Table 2 Strength levels

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

If two signal of unequal strength are driven on a wire, the stronger signal prevails. For example, if two signals of strength **strong 1** and **weak 0** contend, the result is resolved as a **strong1**. If two signals of equal strengths **strong 1** and **strong 0** conflict, the result is an **x**. Strength levels are particularly useful for accurate modeling of signal contention, MOS device, dynamic MOS, and low-level devices. Only **triereg** nets can have storage strengths large, medium, and small.

1.4.2 NETS:

Net represent connection between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to. In figure 5 net a is connected to the output of **and** gate g1.net a will continuously assume the value computed at the out put of gate g1, which is b & c.

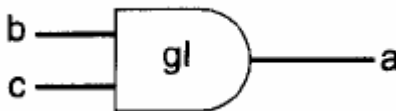


Figure 5 Example of nets

Nets are declared primarily with the keyword **wire**. Nets are one-bit values by default unless they are declared explicitly as vectors. The term **wire** and net are often used interchangeably. The default value of a net is **z**. Nets get output value of their drivers. If a net has no driver, it gets the value **z**.

```
wire a; // Declare net a for the above circuit
wire b,c; // Declare two wires b,c for the above circuit
wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.
```

1.4.3 REGISTERS:

Registers represent data storage elements. Registers retain until another value is placed onto them. Do not confuse the term registers in Verilog with hardware registers from edge-triggered flip-flops in real circuits. In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register can be changed anytime in a simulation by assigning a new value to the register.

Register data types are commonly declared by the keyword **reg**. The default value for a **reg** data type is **x**.

Example 1

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

1.4.4 VECTORS:

Nets or **reg** data types can be declared as vectors. If bit width is not specified, the default is scalar (1-bit).

```

wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register, virtual address 41 bits wide

```

Vectors can be declared at [high#: low #] or [low#: high #]. But the left number in the squared brackets is always the most significant bit of the vector. In the example shown above, bit 1 is the most significant bit of vector virtual_addr.

1.4.5 STRINGS:

Strings can be stored in **reg**. the width of the register variables must be large enough to hold the string. Each character in the string takes up 8 bits (1 byte). If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros. If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string. It is always safe to declare a string that is slightly wider than necessary.

```

reg [8*18:1] string_value; // Declare a variable that is 18 bytes
wide
initial
    string_value = "Hello Verilog World"; // String can be stored
// in variable

```

Special characters serve a special purpose in displaying strings, such as newline, tabs, and displaying argument values. Special characters can be displayed in string only when they are preceded by escape characters, as shown in the table 3.

Table 3 Special characters

Escaped Characters	Character Displayed
<code>\n</code>	newline
<code>\t</code>	tab
<code>%%</code>	%
<code>\\</code>	\
<code>\"</code>	"
<code>\ooo</code>	Character written in 1–3 octal digits

1.5 SYSTEM TASKS & COMPILER DIRECTIVES:

1.5.1 SYSTEM TASKS:

Verilog provides standard system tasks to do certain routine operations. All system tasks appear in the form `$<keyword>`. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks.

Displaying information

`$display` is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: `$display (p1, p2, p3, ..., pn);`

P1, p2, p3, ..., pn can be quoted strings or variables or expressions. The format of `$display` is very similar to print f in "c". A `$display` inserts a newline at the end of the string by default. Strings can be formatted by using the format specifications listed in table 4.

Table 4 String format specifications

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

Monitoring information:

Verilog provides a mechanism to monitor when its value changes. This facility is provided by the **\$monitor** task.

Usage: **\$monitor** (p1, p2, p3..., pn);

The parameters p1, p2... pn can be variables, signal names, or quoted strings. Monitor continuously monitors the values of the variables or signals specified in the parameter list and display all parameters in the list whenever the value of any one variable or signal changes. Unlike **\$display**, **\$monitor** needs to be invoked only once. Only one monitoring list can be active at a time. If there is more than one **\$monitor** statement in your simulation, the last **\$monitor** statement will be the active statement.

Two tasks are used to switch monitoring on and off.

Usage: **\$monitor on;**

\$monitor off;

The **\$monitor on** task enables monitoring and the **\$monitor off** tasks disables monitoring during a simulation. An example of monitor statement is given below.

Example 1 monitor statement

```
//Monitor time and value of the signals clock and reset
//Clock toggles every 5 time units and reset goes down at 10 time units
initial
begin
    $monitor($time,
              " Value of signals clock = %b reset = %b", clock,reset);
end
```

Partial output of the monitor statement:

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

Stopping & finishing simulation:

The task **\$ stop** is provided to stop during a simulation.

Usage: **\$ stop**;

The **\$ stop** task puts the simulation in an interactive model. The designer can then debug the design from the interactive mode. The **\$stop** task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.

The **\$finish** task terminates the simulation.

Usage: **\$finish**;

Examples of **\$stop** and **\$ finish** are shown below in example 2.

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time.
initial // to be explained later. time = 0
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

Example 2 Stop and finish tasks

1.5.2 COMPILER DIRECTIVES:

Compiler directives are provided in Verilog. All compiler directives are defined by using the `<keyword>` construct. We deal with two most useful compiler directives.

- **``define:`**

The ``define` directive is used to define text macros in Verilog. This is similar to `# define` construct in “C”. The defined constants or text macros are used in the Verilog code by preceding them with a ``` (**back tick**). The Verilog compiler substitutes the text of the macro whenever it encounters a `<macro_name>`.

Example 3 ``Define directive`

```
//define a text macro that defines default word size
//Used as `WORD_SIZE in the code
`define WORD_SIZE 32

//define an alias. A $stop will be substituted wherever `S appears
`define S $stop;

//define a frequently used text string
`define WORD_REG reg [31:0]
// you can then define a 32-bit register as `WORD_REG reg32;
```

- **``include:`**

The ``include` directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation. This works similarly to the `#include` in the “C” programming language. This directive is typically used to include header files, which typically contain global or commonly used definitions.

Example 4 ``include directive`

```
// Include the file header.v, which contains declarations in the
// main verilog file design.v.
`include header.v
...
...
<Verilog code in file design.v>
...
...
```


1.5.3 MODULES:

A module in Verilog consists of distinct parts, as shown in figure 6.

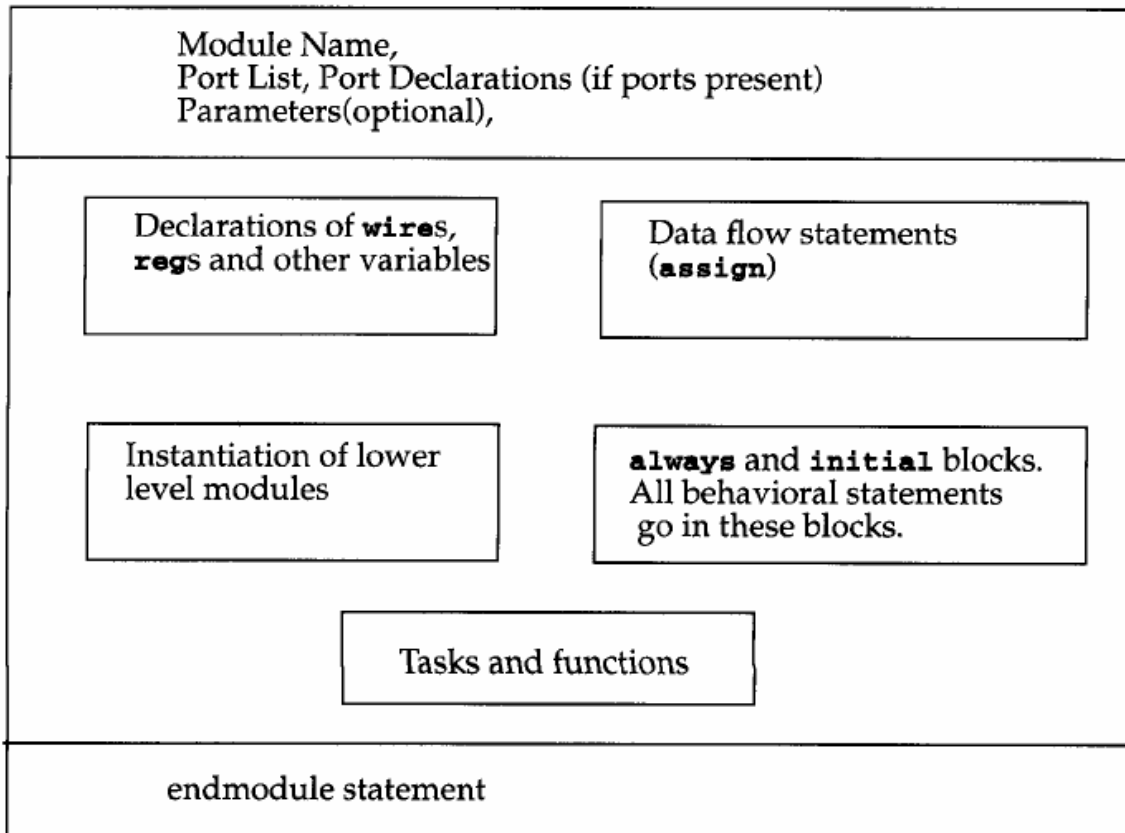


Figure 6 components of Verilog module

A module definition always begins with the keyword **module**. The module name, port list, port declarations, and optional parameters must come first in a module definition. Port list and port declarations are present only if the module has any ports to interact with the external environment. The five components within a module are – variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions. These components can be in any order and at any place in the module definition. The **end module** statement must always come last in the module definition. All components except **module**, **module name**, and **end module** are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

1.5.4 PORTS:

Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. Ports are also referred to as “terminals”.

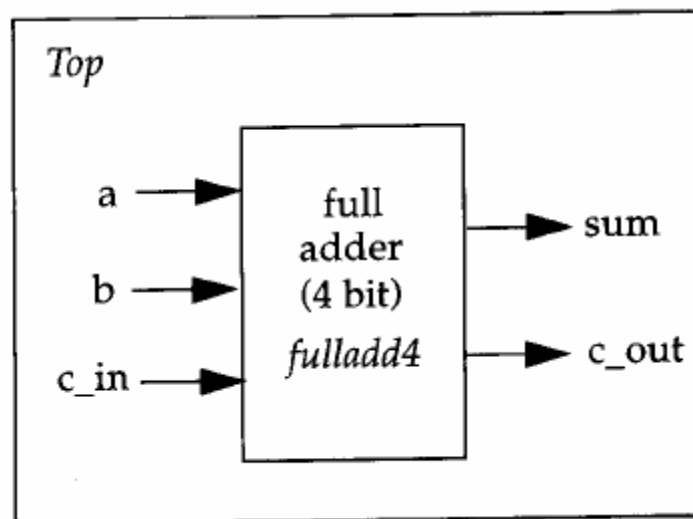
1.5.5 PORT DECLARATION:

All ports in the list of ports must be declared in the module. ports can be declared as follows:

Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

Each port in the list is defined as **input**, **output**, or **inout**, based on the direction of the port signal. Thus, for the example of 4-bit full adder.

Figure 7 full adder (4-bit)



Example 5 port declarations

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

Note that all ports declarations are implicitly declared as **wire** in verilog. Thus, if a port is intended to be a **wire**, it is sufficient to declare it as **output, input, or in out**. **Input or outputs** are normally declared as **wires**. However, if **output** ports hold their value they must be declared as **reg**.

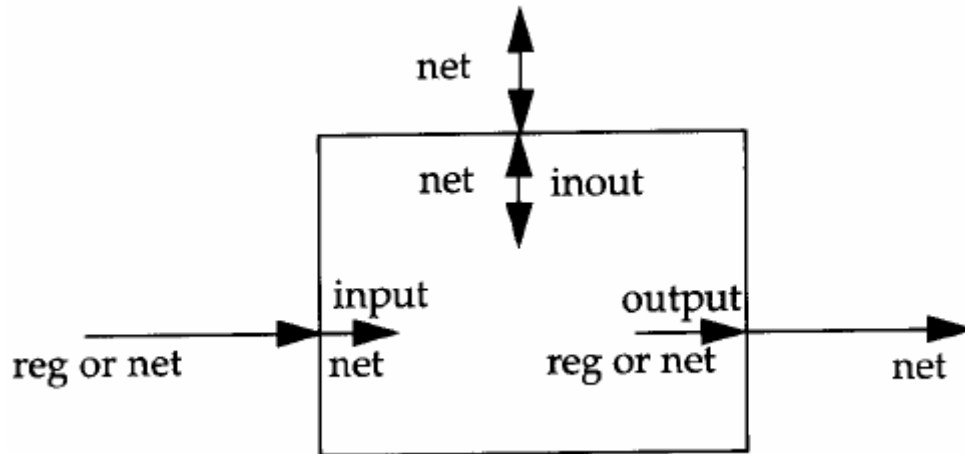
1.5.6 PORT CONNECTION RULES:

One can visualize a port as consisting of two units, one unit that is internal to the module another that is external to the module. The internal and external units are connected. There are rules governing port connections when modules are initiated within other modules. The Verilog simulator complains if any port connection rules are violated. These rules are summarized in figure 8.

- **Inputs**

Internally, input ports must always be of the type **net**. Externally, the inputs can be connected to a variable which is a **reg** or a **net**.

Figure 8 port connection rules



- **Outputs:**

Internally, outputs port can be of the type **reg** or **net**. Externally, outputs must always be connected to a **net**. They cannot be connected to a **reg**.

- **Inouts:**

Internally, inout ports must always be of the type **net**. Externally, inout ports must always be connected to a **net**.

- **Width watching:**

It is legal to connect internal and external items of different sizes when making inter-module port connections. However, a warning is typically issued that the widths do not match.

- **Unconnected ports:**

Verilog allows ports to remain unconnected. For example, certain outputs ports might be simply for debugging and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating module as shown below.

```
fulladd4 fa0(SUM, , A, B, C_IN); // Output port c_out is unconnected
```

1.5.7 CONNECTING PORTS TO EXTERNAL SIGNALS:

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed. These methods are discussed in the following sections.

- **Connecting by ordered list:**

Connecting by ordered list is the most intuitive method for most beginners. The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition. Once again, consider the module `fulladd4` defined in Example 5. To connect signals in module `Top` by ordered list, the Verilog code is shown in Example 6. Notice that the external signals `SUM`, `C_OUT`, `A`, `B`, and `C_IN` appear in exactly the same order as the ports `sum`, `c_out`, `a`, `b`, and `c_in` in module definition of `fulladd4`.

Example 6 Connection by order list

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

//Instantiate fulladd4, call it fa_ordered.
//Signals are connected to ports in order (by position)
fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
...
<stimulus>
```

```

    """
    endmodule

    module fulladd4(sum, c_out, a, b, c_in);
    output[3:0] sum;
    output c_out;
    input [3:0] a, b;
    input c_in;
    """
    <module internals>
    """
    endmodule

```

- **Connecting ports by name:**

For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone. Verilog provides the capability to connect external signals to ports by the port names, rather than by position. We could connect the ports by name in Example 6 above by instantiating the module `fulladd4`, as follows. Note that you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```

// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN),
.a(A),);

```

Note that only those ports that are to be connected to external signals must be specified in port connection by name. Unconnected ports can be dropped. For example, if the port `c_out` were to be kept unconnected, the instantiation of `fulladd4` would look as follows. The port `c_out` is simply dropped from the port list.

```

// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);

```

Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.

MODELING CONCEPTS

GATE-LEVEL MODELING:

2.1 Gate Types:

A logic circuit can be designed by use of logic gates. Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two classes of basic gates: and/or gates and buf/not gates.

2.1.1 And/Or Gates:

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. The and/or gates available in Verilog are shown below.

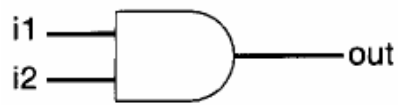
and	or	xor
nand	nor	xnor

The corresponding logic symbols for these gates are shown in Figure 9. We consider gates with two inputs. The output terminal is denoted by out. Input terminals are denoted by i1 and i2.

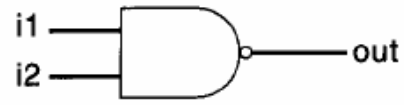
These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In Example , for all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives.

Figure 9

Basic gates



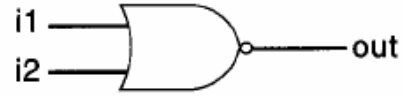
and



nand



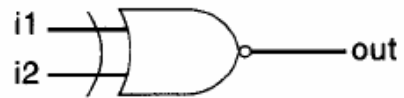
or



nor



xor



xnor

More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation (see Example 7). Verilog automatically instantiates the appropriate gate.

Example 7 Gate instantiation of and/or gates

```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// More than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name
and (OUT, IN1, IN2); // legal gate instantiation
```


The truth tables for these gates define how outputs for the gates are computed from the inputs. Truth tables are defined assuming two inputs. The truth tables for these gates are shown in Table5. Outputs of gates with more than two inputs are computed by applying the truth table iteratively.

Table 5 Truth Table For and/Or gates

<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <thead> <tr> <th colspan="2" rowspan="2"></th> <th colspan="4">i1</th> </tr> <tr> <th>0</th> <th>1</th> <th>x</th> <th>z</th> </tr> </thead> <tbody> <tr> <th rowspan="4" style="writing-mode: vertical-rl; transform: rotate(180deg);">i2</th> <th>and</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <th>1</th> <td>0</td> <td>1</td> <td>x</td> <td>x</td> </tr> <tr> <th>x</th> <td>0</td> <td>x</td> <td>x</td> <td>x</td> </tr> <tr> <th>z</th> <td>0</td> <td>x</td> <td>x</td> <td>x</td> </tr> </tbody> </table>			i1				0	1	x	z	i2	and	0	0	0	0	1	0	1	x	x	x	0	x	x	x	z	0	x	x	x	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <thead> <tr> <th colspan="2" rowspan="2"></th> <th colspan="4">i1</th> </tr> <tr> <th>0</th> <th>1</th> <th>x</th> <th>z</th> </tr> </thead> <tbody> <tr> <th rowspan="4" style="writing-mode: vertical-rl; transform: rotate(180deg);">i2</th> <th>nand</th> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <th>1</th> <td>1</td> <td>0</td> <td>x</td> <td>x</td> </tr> <tr> <th>x</th> <td>1</td> <td>x</td> <td>x</td> <td>x</td> </tr> <tr> <th>z</th> <td>1</td> <td>x</td> <td>x</td> <td>x</td> </tr> </tbody> </table>			i1				0	1	x	z	i2	nand	1	1	1	1	1	1	0	x	x	x	1	x	x	x	z	1	x	x	x
			i1																																																												
		0	1	x	z																																																										
i2	and	0	0	0	0																																																										
	1	0	1	x	x																																																										
	x	0	x	x	x																																																										
	z	0	x	x	x																																																										
		i1																																																													
		0	1	x	z																																																										
i2	nand	1	1	1	1																																																										
	1	1	0	x	x																																																										
	x	1	x	x	x																																																										
	z	1	x	x	x																																																										
<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <thead> <tr> <th colspan="2" rowspan="2"></th> <th colspan="4">i1</th> </tr> <tr> <th>0</th> <th>1</th> <th>x</th> <th>z</th> </tr> </thead> <tbody> <tr> <th rowspan="4" style="writing-mode: vertical-rl; transform: rotate(180deg);">i2</th> <th>or</th> <td>0</td> <td>1</td> <td>x</td> <td>x</td> </tr> <tr> <th>1</th> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <th>x</th> <td>x</td> <td>1</td> <td>x</td> <td>x</td> </tr> <tr> <th>z</th> <td>x</td> <td>1</td> <td>x</td> <td>x</td> </tr> </tbody> </table>			i1				0	1	x	z	i2	or	0	1	x	x	1	1	1	1	1	x	x	1	x	x	z	x	1	x	x	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <thead> <tr> <th colspan="2" rowspan="2"></th> <th colspan="4">i1</th> </tr> <tr> <th>0</th> <th>1</th> <th>x</th> <th>z</th> </tr> </thead> <tbody> <tr> <th rowspan="4" style="writing-mode: vertical-rl; transform: rotate(180deg);">i2</th> <th>nor</th> <td>1</td> <td>0</td> <td>x</td> <td>x</td> </tr> <tr> <th>1</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <th>x</th> <td>x</td> <td>0</td> <td>x</td> <td>x</td> </tr> <tr> <th>z</th> <td>x</td> <td>0</td> <td>x</td> <td>x</td> </tr> </tbody> </table>			i1				0	1	x	z	i2	nor	1	0	x	x	1	0	0	0	0	x	x	0	x	x	z	x	0	x	x
			i1																																																												
		0	1	x	z																																																										
i2	or	0	1	x	x																																																										
	1	1	1	1	1																																																										
	x	x	1	x	x																																																										
	z	x	1	x	x																																																										
		i1																																																													
		0	1	x	z																																																										
i2	nor	1	0	x	x																																																										
	1	0	0	0	0																																																										
	x	x	0	x	x																																																										
	z	x	0	x	x																																																										
<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <thead> <tr> <th colspan="2" rowspan="2"></th> <th colspan="4">i1</th> </tr> <tr> <th>0</th> <th>1</th> <th>x</th> <th>z</th> </tr> </thead> <tbody> <tr> <th rowspan="4" style="writing-mode: vertical-rl; transform: rotate(180deg);">i2</th> <th>xor</th> <td>0</td> <td>1</td> <td>x</td> <td>x</td> </tr> <tr> <th>1</th> <td>1</td> <td>0</td> <td>x</td> <td>x</td> </tr> <tr> <th>x</th> <td>x</td> <td>x</td> <td>x</td> <td>x</td> </tr> <tr> <th>z</th> <td>x</td> <td>x</td> <td>x</td> <td>x</td> </tr> </tbody> </table>			i1				0	1	x	z	i2	xor	0	1	x	x	1	1	0	x	x	x	x	x	x	x	z	x	x	x	x	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <thead> <tr> <th colspan="2" rowspan="2"></th> <th colspan="4">i1</th> </tr> <tr> <th>0</th> <th>1</th> <th>x</th> <th>z</th> </tr> </thead> <tbody> <tr> <th rowspan="4" style="writing-mode: vertical-rl; transform: rotate(180deg);">i2</th> <th>xnor</th> <td>1</td> <td>0</td> <td>x</td> <td>x</td> </tr> <tr> <th>1</th> <td>0</td> <td>1</td> <td>x</td> <td>x</td> </tr> <tr> <th>x</th> <td>x</td> <td>x</td> <td>x</td> <td>x</td> </tr> <tr> <th>z</th> <td>x</td> <td>x</td> <td>x</td> <td>x</td> </tr> </tbody> </table>			i1				0	1	x	z	i2	xnor	1	0	x	x	1	0	1	x	x	x	x	x	x	x	z	x	x	x	x
			i1																																																												
		0	1	x	z																																																										
i2	xor	0	1	x	x																																																										
	1	1	0	x	x																																																										
	x	x	x	x	x																																																										
	z	x	x	x	x																																																										
		i1																																																													
		0	1	x	z																																																										
i2	xnor	1	0	x	x																																																										
	1	0	1	x	x																																																										
	x	x	x	x	x																																																										
	z	x	x	x	x																																																										

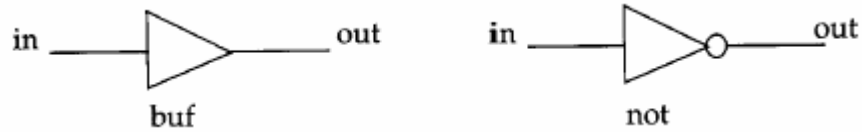
2.1.2 Buf/Not Gates:

Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one output.

Two basic buf/not gate primitives are provided in Verilog.

The symbols for these logic gates are shown in Figure 10.

Figure 10 **Buf and Not Gates**



These gates are instantiated in Verilog as shown Example 8. That these gates can have multiple outputs but exactly one input, which the last terminal in the port list is.

Example 8 **Gate Instant ion Of And/ Or gates**

```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than two outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation
```

The truth tables for these gates are very simple. Truth tables for gates with one input and one output are shown in Table 6.

Table 6

Truth Tables for Buf /Not gates

buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

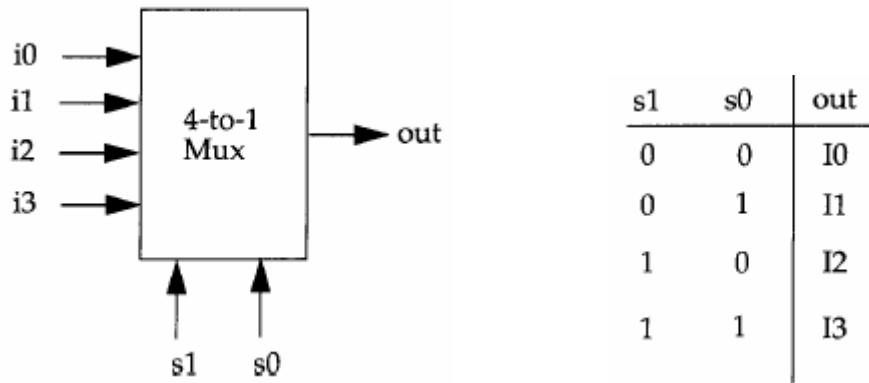
2.1.3 Gate Level Modeling Examples:

4:1 Multiplexer:

We will design a 4-to-1 multiplexer with 2 select signals. Multiplexers serve a useful purpose in logic design. They can connect two or more sources to a single destination.

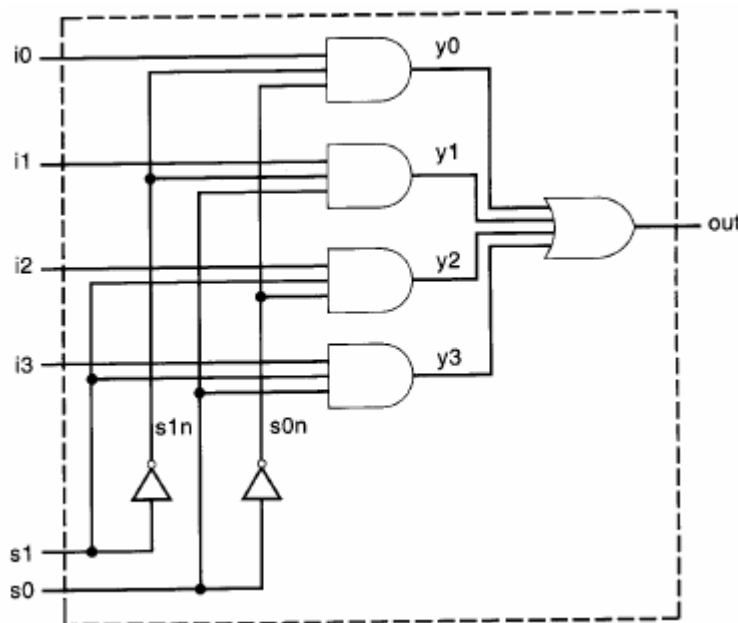
They can also be used to implement Boolean functions. We will assume for this example that signals s_1 and s_0 do not get the value x or z . The I/O diagram and the truth table for the multiplexer are shown in Figure 11. The I/O diagram will be useful in setting up the port list for the multiplexer.

Figure 11 4 to 1 multiplexer



We will implement the logic for the multiplexer using basic logic gates. The logic diagram for the multiplexer is shown in Figure 12.

Figure 12 Logic Diagram for Multiplexer



Two intermediate nets, s0n and s1n, are created; they are complements of input signals s1 and s0. Internal nets y0, y1, y2, y3 are also required. Note that instance names are not specified for primitive gates, not, and, and or.

Example 9 Verilog description of multiplexer

```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```

This multiplexer can be tested with the stimulus shown in Example10. The stimulus checks that each combination of select signals connects the appropriate input to the output. The signal OUTPUT is displayed one time unit after it changes. System task **\$monitor** could also be used to display the signals when they change values.

Example 10 Stimulus for multiplexer

```
// Define the stimulus module (no ports)
module stimulus;

// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;
reg S1, S0;

// Declare output wire
wire OUTPUT;

// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);

// Stimulate the inputs
// Define the stimulus module (no ports)
initial
begin
    // set input lines
    IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
    #1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n", IN0, IN1, IN2, IN3);

    // choose IN0
    S1 = 0; S0 = 0;
    #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

    // choose IN1
    S1 = 0; S0 = 1;
    #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

    // choose IN2
    S1 = 1; S0 = 0;
    #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

    // choose IN3
    S1 = 1; S0 = 1;
    #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end

endmodule
```

The output of the simulation is shown below.

```
IN0= 1, IN1= 0, IN2= 1, IN3= 0
```

```
S1 = 0, S0 = 0, OUTPUT = 1
```

```
S1 = 0, S0 = 1, OUTPUT = 0
```

```
S1 = 1, S0 = 0, OUTPUT = 1
```

```
S1 = 1, S0 = 1, OUTPUT = 0
```

2.1.4 Gate Delays:

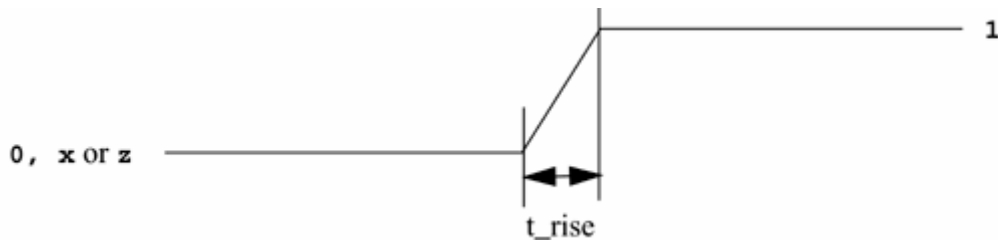
In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, fall, and Turn-off Delays:

There are three types of delays from the inputs to the output of a primitive gate.

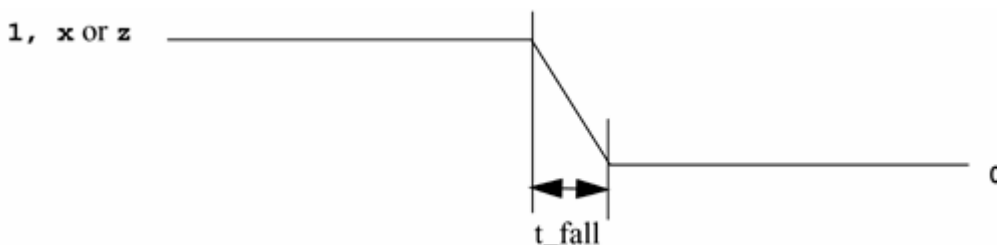
Rise Delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall Delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-Off Delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x , the minimum of the three delays is considered. Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The

turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero.

2.1.5 Min/Typ/Max Values:

Verilog provides an additional level of control for each type of delay mentioned above. For each type of delay—rise, fall, and turn-off—three values, min, typ, and max, can be specified. Any one value can be chosen at the start of the simulation. Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range because of the IC fabrication process variations.

Min value

The min value is the minimum delay value that the designer expects the gate to have.

Typ value

The typ value is the typical delay value that the designer expects the gate to have.

Max value

The max value is the maximum delay value that the designer expects the gate to have.

2.1.6 DELAY EXAMPLE:

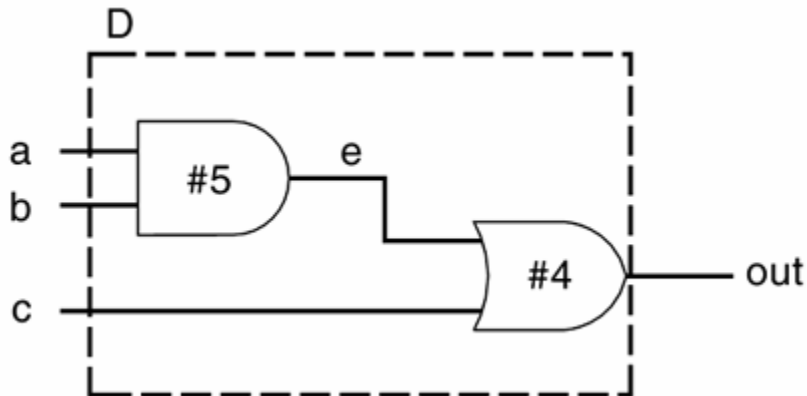
Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits. A simple module called D implements the following logic equations:

$$\text{Out} = (a \cdot b) + c$$

The gate-level implementation is shown in Module D (Figure 13). The module contains two gates with delays of 5 and 4 time units.

The module D is defined in Verilog as shown in Example 11.

Figure 13 module D



Example 11 Verilog Definition for Module D Delay

```
// Define a simple combination module called D
module D (out, a, b, c);

// I/O port declarations
output out;
input a,b,c;

// Internal nets
wire e;

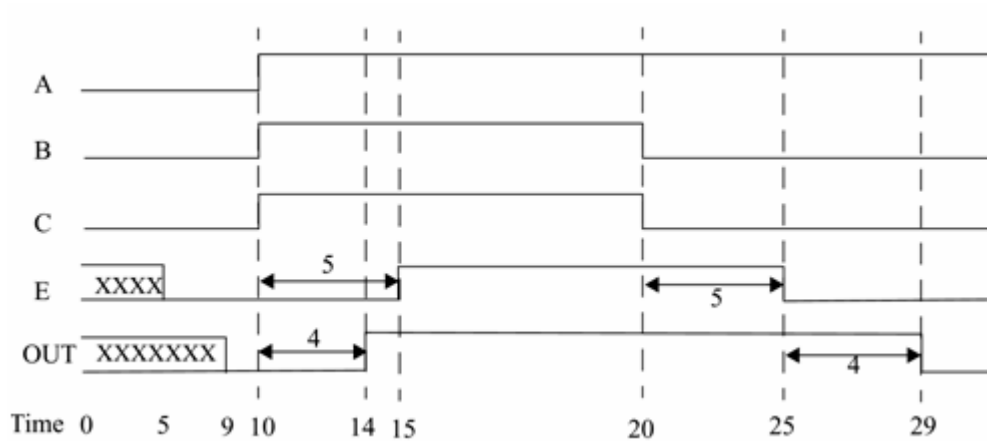
// Instantiate primitive gates to build the circuit
and #5 a1(e, a, b); //Delay of 5 on gate a1
or #4 o1(out, e,c); //Delay of 4 on gate o1

endmodule
```

The waveforms from the simulation are shown in Figure 14 to illustrate the effect of specifying delays on gates. The waveforms are not drawn to scale. However, simulation time at each transition is specified below the transition.

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

Figure 14 Waveforms for Delay Simulation



2.2 DATAFLOW MODELING:

2.2.1 Continuous Assignment:

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword `assign`. The syntax of an `assign` statement is as follows.

```
continuous_assign ::= assign [ drive_strength ] [ delay3 ]
                    list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```

2.2.2 Implicit Continuous Assignment:

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```

//Regular continuous assignment
wire out;
assign out = in1 & in2;

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

```

2.2.3 Implicit Net Declaration:

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

```

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

```

2.2.4 Expressions, Operators & Operands:

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

Expressions:

Expressions are constructs that combine operators and operands to produce a result.

```

// Examples of expressions. Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2

```

Operands:

Operands can be any one of the data types. Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls.

```

integer count, final_count;
final_count = count + 1; //count is an integer operand

real a, b, c;
c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0]; //reg1[3:0] and reg2[3:0] are
                                //part-select register operands

reg ret_value;
ret_value = calculate_parity(A, B); //calculate_parity is a
                                //function type operand

```

Operators:

Operators act on the operands to produce desired results. Verilog provides various types of operators.

```

d1 && d2 // && is an operator on operands d1 and d2
!a[0] // ! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1

```

2.2.5 Operator Types:

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. Table 7 shows the complete listing of operator symbols classified by category.

Table 7 Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
		bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
		reduction or	one
	~	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Shift	>>	Right shift	two
	<<	Left shift	two
Concatenation	{ }	Concatenation	any number
Replication	{ { } }	Replication	any number
Conditional	?:	Conditional	three

Concatenation Operator:

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

Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

```
// X = 4'b1010

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

Replication Operator:

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

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

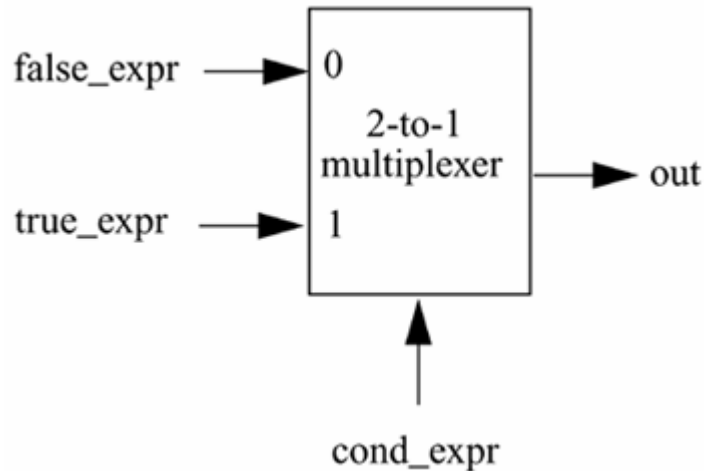
Conditional Operator:

The conditional operator(?) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;

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

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.



Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

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

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

2.3 Behavioral Modeling:

2.3.1 Structured Procedures:

There are two structured procedure statements in Verilog: `always` and `initial`. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

Initial Statement:

All statements inside an `initial` statement constitute an `initial` block. An `initial` block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple `initial` blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks. Multiple behavioral statements must be grouped, typically using the keywords `begin` and `end`. If there is only one behavioral statement, grouping is not necessary. This is similar

to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language. Example 12 illustrates the use of the `initial` statement.

Example 12 Initial Statement

```
module stimulus;

reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
    #50 $finish;

endmodule
```

In the above example, the three initial statements start to execute in parallel at time 0. If delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the `initial` blocks will be as follows.

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

The `initial` blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

Always Statement:

All behavioral statements inside an `always` statement constitute an `always` block. The `always` statement starts at time 0 and executes the statements in the `always` block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. An example is a clock generator module that toggles the clock signal every half cycle. In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on. Example 13 illustrates one method to model a clock generator in Verilog.

Example 13 Always Statement

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

In Example 13, the `always` statement starts at time 0 and executes the statement `clock = ~clock` every 10 time units. Notice that the initialization of `clock` has to be done inside a separate `initial` statement. If we put the initialization of `clock` inside the `always` block, `clock` will be initialized every time the `always` is entered. Also, the simulation must be halted inside an `initial` statement. If there is no `$stop` or `$finish` statement to halt the simulation, the clock generator will run forever.

2.3.2 Procedural Assignments:

Procedural assignments update values of `reg`, `integer`, `real`, or `time` variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. The syntax is shown below.


```
assignment ::= variable_lvalue = [ delay_or_event_control ]  
           expression
```

There are two types of procedural assignment statements: blocking and nonblocking.

Blocking Assignment:

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block.

The = operator is used to specify blocking assignments.

Nonblocking Assignment:

Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments. Note that this operator has the same symbol as a relational operator, less_than_equal_to. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment.

2.3.3 Multiway Branching:

Case Statement:

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

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

Each of `statement1`, `statement2` ..., `default_statement` can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords `begin` and `end`.

Case x , Case z Keywords:

There are two variations of the `case` statement. They are denoted by keywords, `casex` and `casez`.

- ❖ `Casez` treats all `z` values in the case alternatives or the case expression as don't cares. All bit positions with `z` can also be represented by `?` in that position.
- ❖ `Casex` treats all `x` and `z` values in the case item or the case expression as don't cares.

The use of `casex` and `casez` allows comparison of only non-`x` or -`z` positions in the case expression and the case alternatives.

2.3.4 Loops:

There are four types of looping statements in Verilog: `while`, `for`, `repeat`, and `forever`. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an `initial` or `always` block. Loops may contain delay expressions.

While Loop:

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

For Loop:

The keyword `for` is used to specify this loop. The `for` loop contains three parts:

- ❖ An initial condition
- ❖ A check to see if the terminating condition is true
- ❖ A procedural assignment to change value of the control variable

The initialization condition and the incrementing procedural assignment are included in the `for` loop and do not need to be specified separately. Thus, the `for` loop provides a more compact loop structure than the `while` loop.

Repeat Loop:

The keyword `repeat` is used for this loop. The `repeat` construct executes the loop a fixed number of times. A `repeat` construct cannot be used to loop on a general logical expression. A `while` loop is used for that purpose. A `repeat` construct must contain a number, which can be a constant, a variable or a signal value. However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

Forever Loop:

The keyword `forever` is used to express this loop. The loop does not contain any expression and executes forever until the `$finish` task is encountered. The loop is equivalent to a `while` loop with an expression that always evaluates to true, e.g., `while (1)`. A forever loop can be exited by use of the `disable` statement.

A `forever` loop is typically used in conjunction with timing control constructs. If timing control constructs are not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed.

8 Bit ALU

Arithmetic Logic Unit performs the arithmetic and logic operations during execution of an instruction. Contains accumulator CPU registers and related logic such as arithmetic and logic unit. ALU communicates with the internal registers and the external data bus by using internal data bus. Functions performed by the ALU include:

- ❖ Addition
- ❖ Subtraction
- ❖ Logical AND
- ❖ Logical OR
- ❖ Logical Exclusive OR
- ❖ Compare
- ❖ Left or Right Shifts or Rotate
- ❖ Increment
- ❖ Decrement
- ❖ Set/Reset and Test Bit

3.1 Overview:

The arithmetic logic unit (ALU) is the brain of the computer, the device that performs the arithmetic operations like addition, subtraction or logical operations like AND & OR.

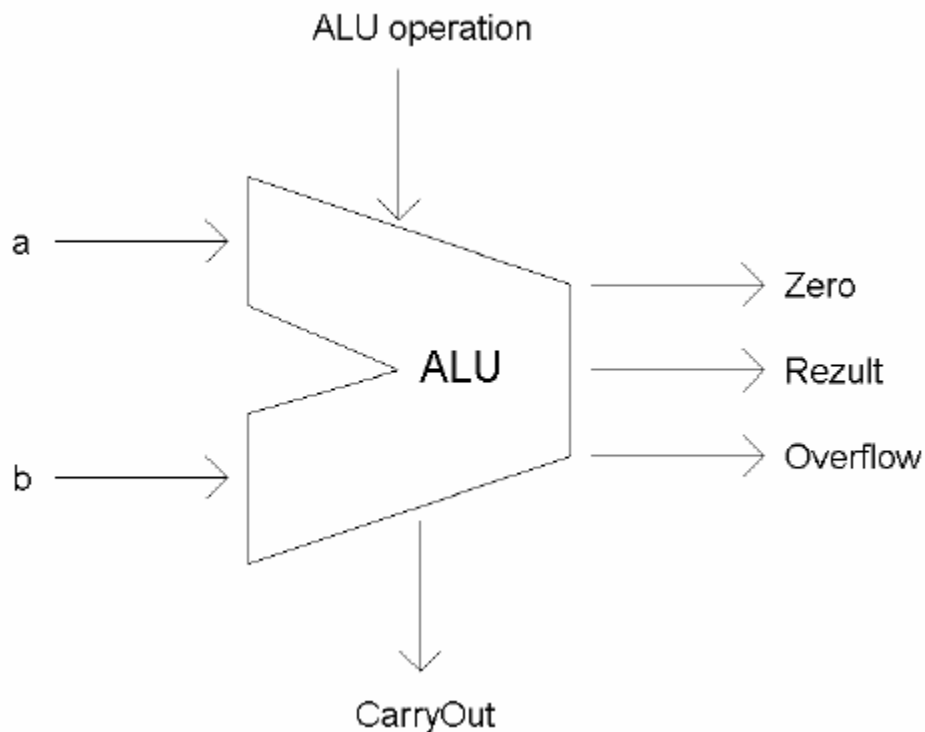


Figure 15 ALU Symbol

For this project I need to design 8-bit ALU to perform 16 operations:

ADD, SUB, Increment, Decrement, AND, CLEAR, NOT, Immediate OR, MOV, MOV Word, Rotate Left, Rotate Right, SWAP, EX-OR, BCF, BSF. The ALU should have two 8-bit inputs s1 and s2, 4-bit ALU operation, output should be 8-bit result, 1-bit Zero signal, 1-bit Carry Out and 1-bit overflow output when an overflow is detected.

3.2 Project Requirements:

Design 8-bit ALU that has inputs s1 [7:0], s2 [7:0], m ALU operation [3:0] and outputs Zero, Result [7:0], Overflow, Carry Out. The ALU should perform the above sixteen (16) operations.

3.3 Design Description:

3.3.1 1-bit ALU for the MSB:

For the most significant bit (MSB) of my ALU I designed a special 1-bit ALU, which has overflow detection logic and a special output Set.

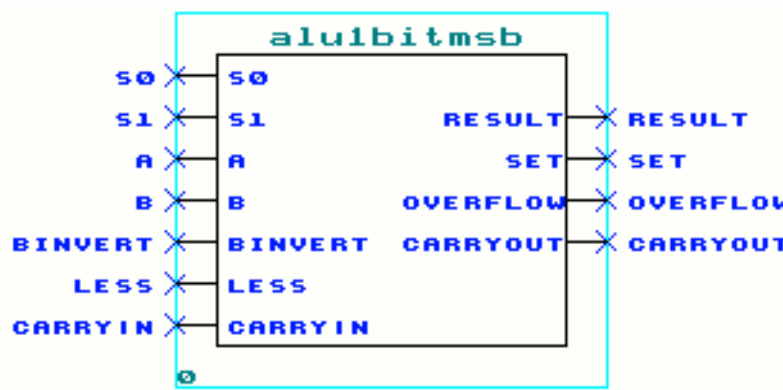


Figure 16 1-bit ALU for the Most Significant Bit (MSB)

3.3.2 Overflow Detection:

The overflow can be found when we compare the CarryIn and the CarryOut signal. If they have different values, than overflow occurs; if both of them are 0s or 1s there is no overflow. This logic can easily be represented by XOR on the two signals.

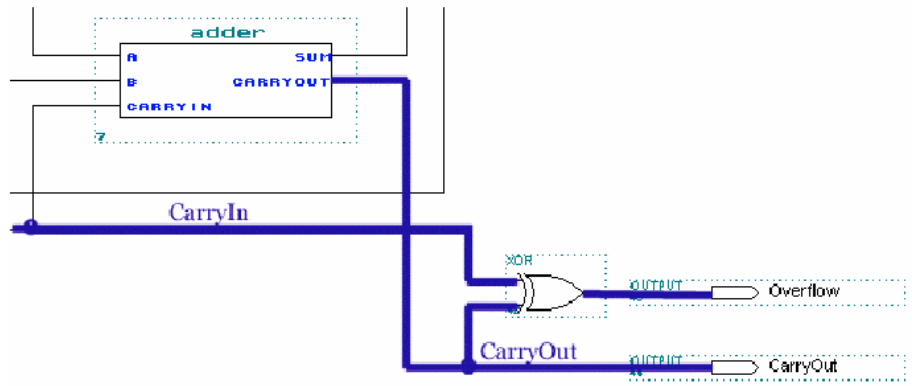


Table 8 Truth Table for XOR

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

3.3.3 Set Output:

This 1-bit ALU differs from the regular one not only by the overflow detection. It has an extra output called Set used for the SLT operation. Set is the Sum output of the adder and is connected to the Less input of the first 1-bit ALU.

3.4 Regular 1-bit ALU:

My regular 1-bit ALU consists of 1-bit full adder, components for logical operations and operation selector. This 1-bit ALU is used for the first 7 bits [6:0] of the 8-bit ALU.

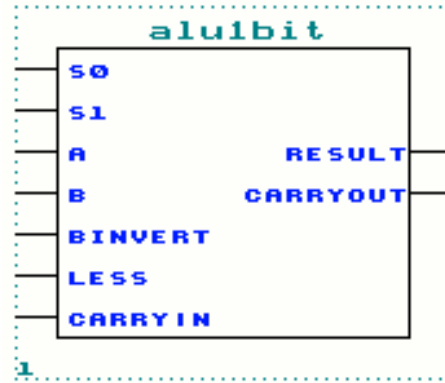


Figure 17 1-bit ALU

3.5 Full Adder:

It can perform the following operations: AND, OR, addition of a & b. The adder has 3

Inputs - a, b, CarryIn; and 2 outputs - Sum and CarryOut and is also called (3,2). Adder with 2 inputs and 2 outputs is called half adder.

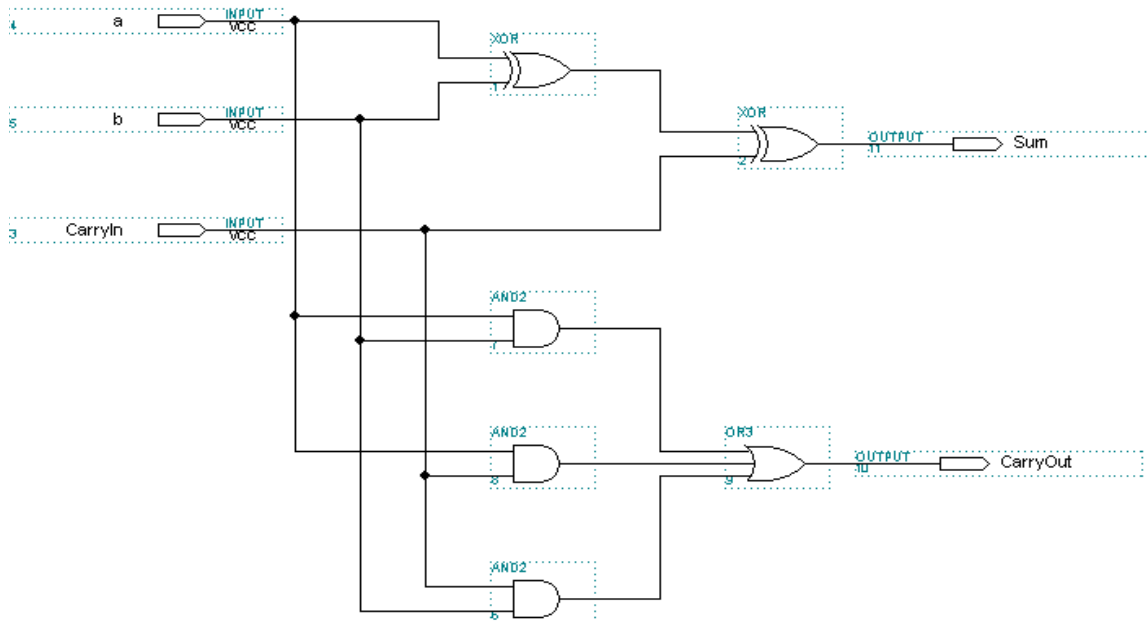


Figure 18 1-bit Full Adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0+0+0=00_{\text{two}}$
0	0	1	0	1	$0+0+1=01_{\text{two}}$
0	1	0	0	1	$0+1+0=01_{\text{two}}$
0	1	1	1	0	$0+1+1=10_{\text{two}}$
1	0	0	0	1	$1+0+0=01_{\text{two}}$
1	0	1	1	0	$1+0+1=10_{\text{two}}$
1	1	0	1	0	$1+1+0=10_{\text{two}}$
1	1	1	1	1	$1+1+1=11_{\text{two}}$

Table 9 Input & Output Specification of 1-bit Full Adder

3.6 Controls Of 1-bit ALU:

Selecting the ALU operation to be performed is implemented with 4x1 multiplexer.

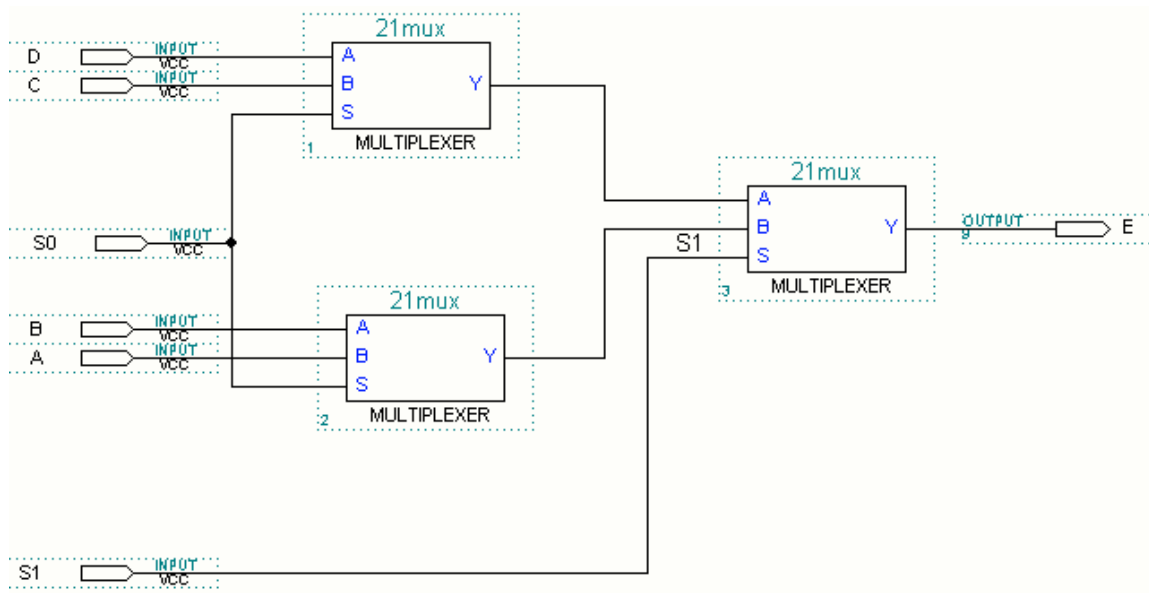


Figure 19 4x1 Multiplexer

The 4x1 multiplexer is delivered with the logic of three 2x1 multiplexers as is shown on Fig. 20. The data inputs are A, B, C and D control; signals are S0 and S1; output result is E. The truth table on table 10 describes the logical functionality of 4x1 multiplexer. The code combination of the control signals S0, S1 will determinate which one of the data signals A, B, C or D will appear at the output E. For example with combination S0=0 and S1=0 the output signal E is A.

Table 10 Truth Table for 4x1 Multiplexer

Control signals		Result
S1	S0	E
0	0	A
0	1	B
1	0	C
1	1	D

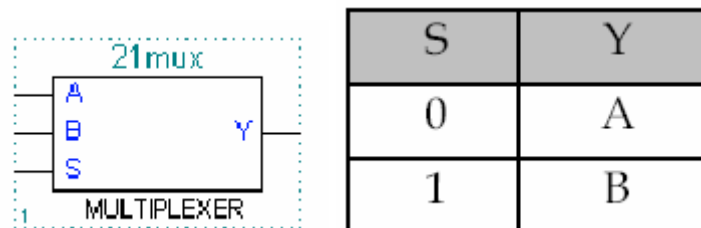


Figure 20 2x1 Multiplexer & Truth Table

3.7 Zero Detector:

It is needed to check if two registers are equal or not. We could say that they are equal if $a - b = 0$.

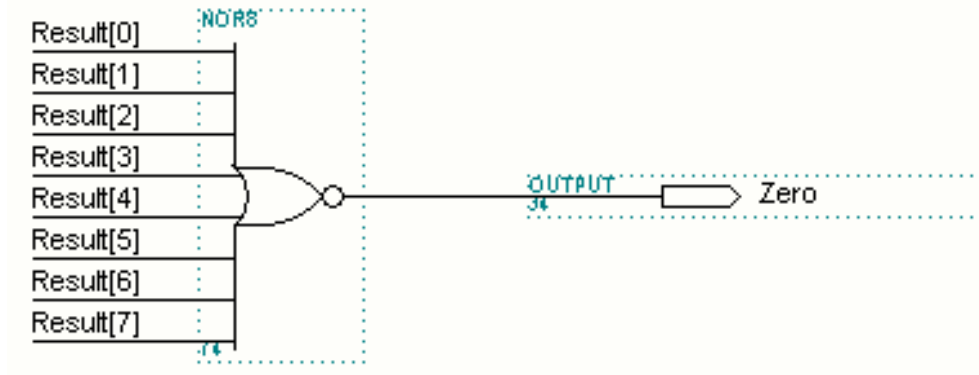


Figure 21 Zero Detectors

The expression for the Zero signal is:

$$\text{Zero} = \overline{(\text{Result7} + \text{Result6} + \dots + \text{Result0})}$$

Hard Ware Kit (Spartan -3A)

Description:

The Spartan-3A Evaluation Kit provides a platform for engineers designing with the Xilinx Spartan-3A FPGA and/or Cypress PSoC® Mixed Signal Array. The board provides the necessary hardware to not only evaluate the advanced features of these devices but also to implement user applications using peripherals and expansion connectors on the Spartan-3A evaluation board. Figure 1 is a picture of the Spartan-3A evaluation board; the block diagram in Figure 2 provides a high-level view of the components and interconnects.

Features:

Xilinx 3S400A-4FTG256C FPGA

• Clocks

- ❖ 16 MHz Oscillator (Maxim)
- ❖ 12 MHz Clock from PSoC device
- ❖ 32 kHz Clock from PSoC device

• Memory

- ❖ 32 Mb Page-Mode Flash Memory (Spansion)
- ❖ 128 Mb SPI Flash Memory (Spansion)

• Interfaces

- ❖ USB 2.0 (PSoC)
- ❖ JTAG Programming/Configuration Port
- ❖ Temperature Sensor (Texas Instruments)

• Buttons and switches

- ❖ Four User LEDs

- ❖ Four PSoC Cap Sense capacitive switches
- ❖ Four FPGA user “pushbuttons” (forwarded from PSoC Cap Sense switches)
- ❖ Reset Push Button Switch

- **User I/O and expansion**

- ❖ Digilent 6-pin header (2)
- ❖ 2x20 0.1” Expansion Connector

- **Configuration and Debug**

- ❖ JTAG

Hardware Flow Diagram:

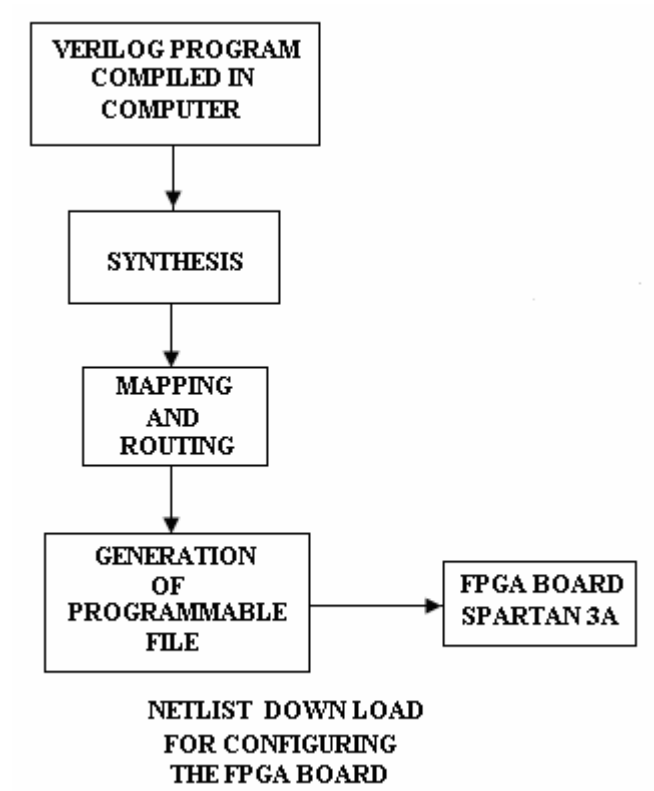


Figure 22 Flow Diagram

Spartan- 3A Evaluation Board Picture

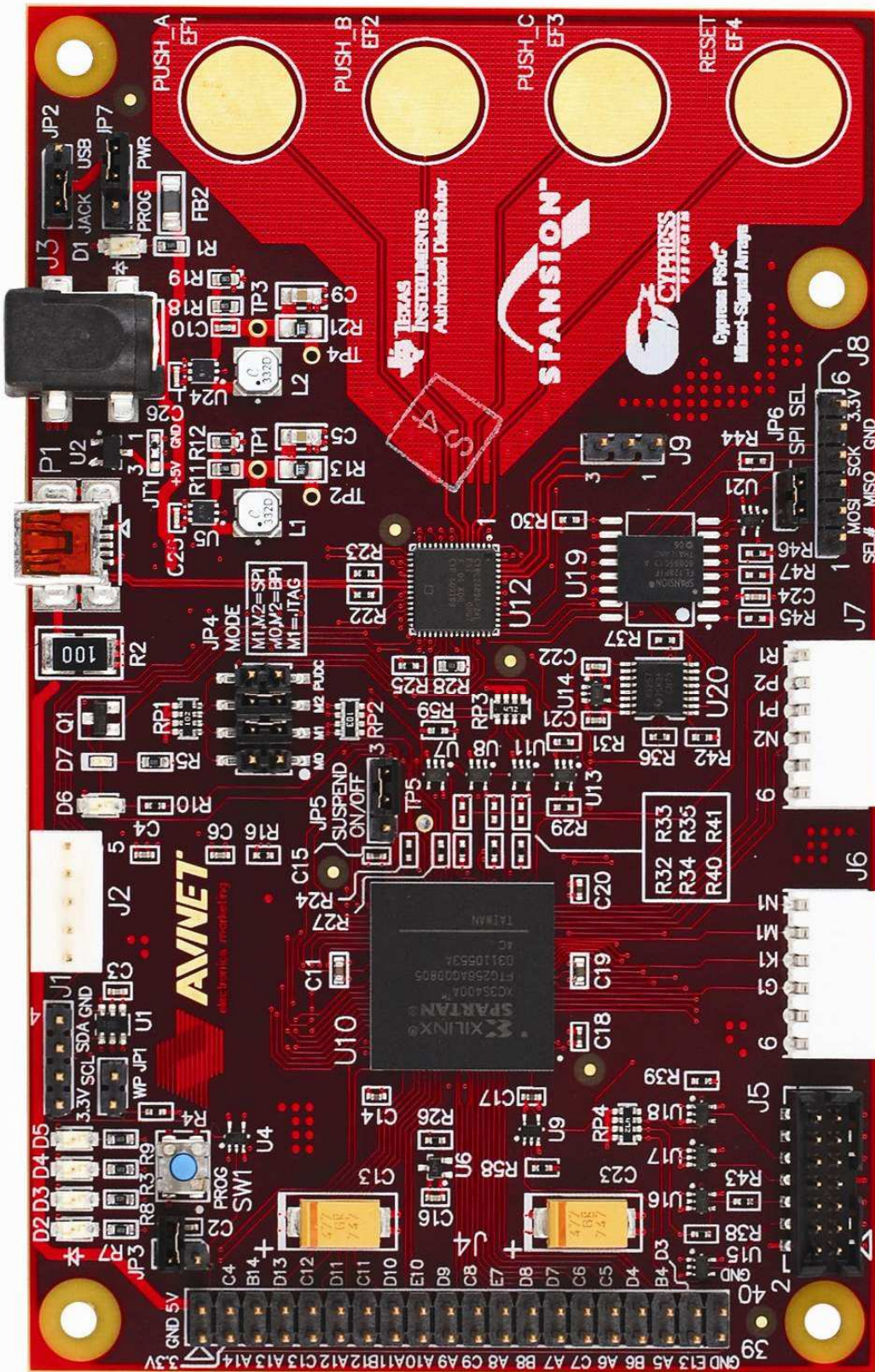


Figure 23 Spartan-3A Evaluation Board

Functional Description:

A Xilinx Spartan-3A (XC3S400A-4FTG256C) 400 K gate FPGA and a Cypress Cy8C24894 PSoC Mixed-Signal Array are the primary components of the Avnet Spartan-3A evaluation board. In addition to on-board processing functions, the PSoC device provides off-board communication via a USB 2.0 full-speed interface. Communication between the PSoC and FPGA is facilitated by a 3.3 V level RS-232 interface between the two devices. This, along with several GPIO lines interconnecting the PSoC and FPGA, provide control and data-transfer mechanisms. A high-level block diagram of the Spartan-3A evaluation board is shown in Figure 23. As can be seen in 2 Figure 23, the USB controller (PSoC), an SPI port, and an I²C port provide off-board communication mechanisms. On-board memory consists of a 128 Mbit SPI memory that may be used by either the PSoC or the FPGA, with FPGA access controlled by the PSoC; and 32 Mbit parallel Flash memory interfaced to the FPGA. Subsequent sections provide details of the board design.

Xilinx Spartan-3A FPGA:

The Xilinx XC3S400A-4FTG256C device designed onto the Spartan-3A evaluation board provides four I/O banks with V_{CCAUX} and I/O voltage of all banks fixed at +3.3 V. The ability to power V_{CCO} and V_{CCAUX} from a common rail is a feature of the Spartan-3A that allows a lower-cost board design. Note that because V_{CCAUX} is set at +3.3 V, each design's UCF must contain the statement:

CONFIG VCCAUX = "3.3";

The four I/O banks are described in Table 11 and detailed I/O pin usage is provided throughout this document. Note that all pins utilized are bidirectional (regardless of usage), the XC3S400A input-only pins are not utilized in this implementation.

Block Diagram

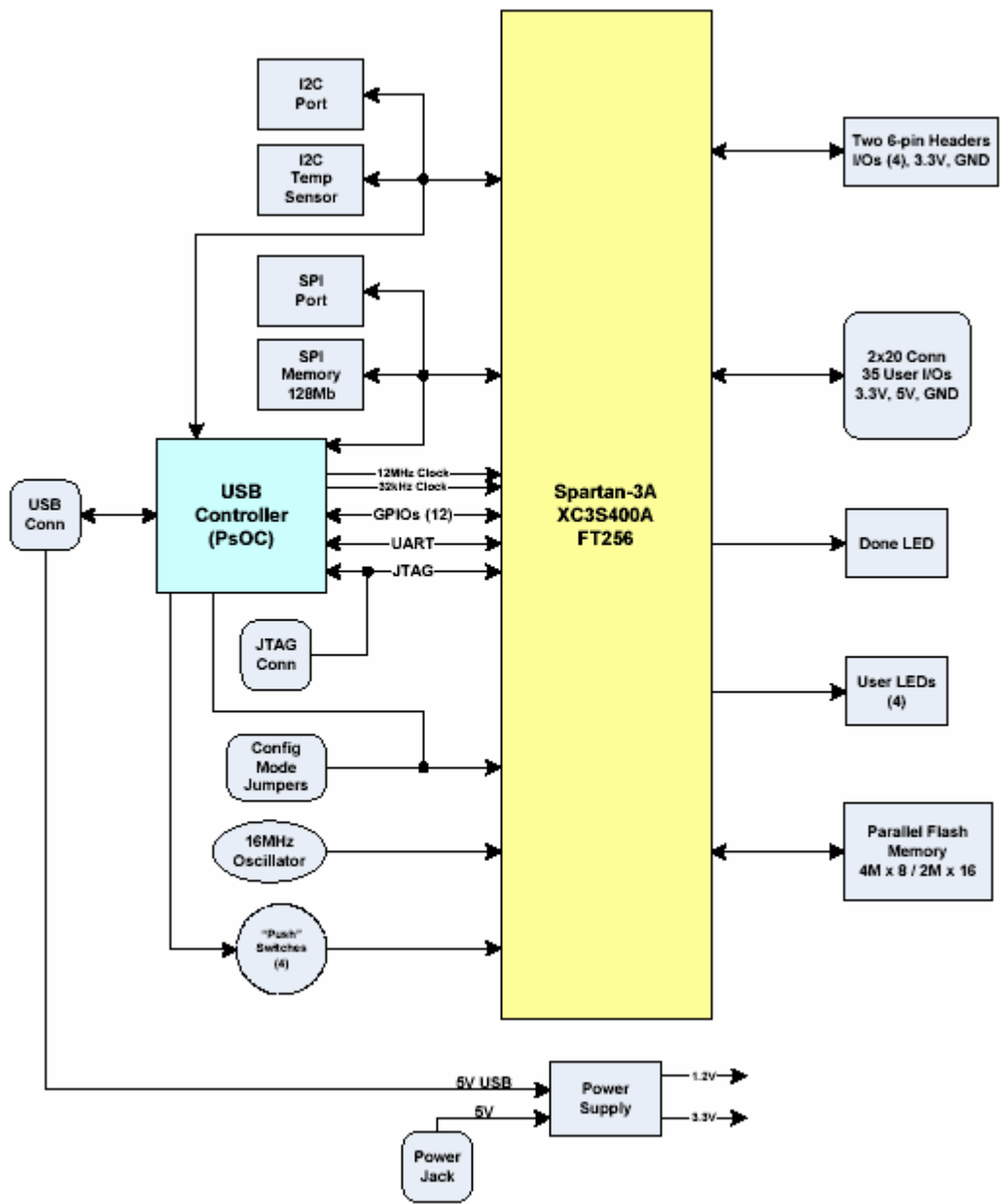


Figure 24 Spartan-3A Block Diagram

I/O Bank #	I/O Function	Number of I/O pins
0	2 x 20 Expansion Connector (J4)	32
0	16 MHz Clock	1 (GCLK4)
0	LED	1
0	UART (FPGA-PSoC Communication)	2
0	Parallel Flash	1
0	FPGA Configuration	1
1	2 x 20 Expansion Connector (J4)	1
1	LED	3
1	I ² C Interface	2
1	PSoC I/O	1
1	Parallel Flash	26 *
2	SPI Interface	6 *
2	Parallel Flash	16
2	12 MHz Clock	1 (GCLK0)
2	32 kHz Clock	1 (GCLK13)
2	LED (AWAKE)	1
2	FPGA Configuration (M[0:2])	3
3	Digilent Headers J6, J7)	8
3	FPGA Reset (from PSoC)	1
3	"Pushbuttons" (CapSense via PSoC)	4
3	PSoC I/O	11
3	2 x 20 Expansion Connector (J4)	2

Table 11 I/O Allocation

Cypress PSoC Mixed-Signal Array:

The Cypress Cy8C24894 is a configurable device containing analog and digital blocks and peripheral devices that allow the user to create customized configurations to support different applications. As configured on the Spartan-3A evaluation board, the PSoC provides a full-speed (12 Mbps) USB interface, RS-232, SPI and I²C interfaces, four capacitive touch-pads (the condition of which is sent to the FPGA), and 15 general-purpose I/O lines (12 connected to the FPGA and three connected to header J9). A 6-pin header that is compatible with the Cypress Mini-Programmer allows configuration of the PSoC's Flash program store. Additionally, the PSoC's JTAG interface may be utilized to program the FPGA; e.g., the FPGA bit file transferred to the PSoC via USB and the PSoC JTAG interface transfers the file into the FPGA.

Memory:

The Spartan-3A evaluation board is populated with both parallel Flash memory (4 Mbytes) and 128 Mbit SPI Serial to support various types of applications. Both Parallel Flash and SPI Serial Flash may be used for FPGA configuration. Figure 25 shows a High-level block diagram of the memory interfaces on this board.

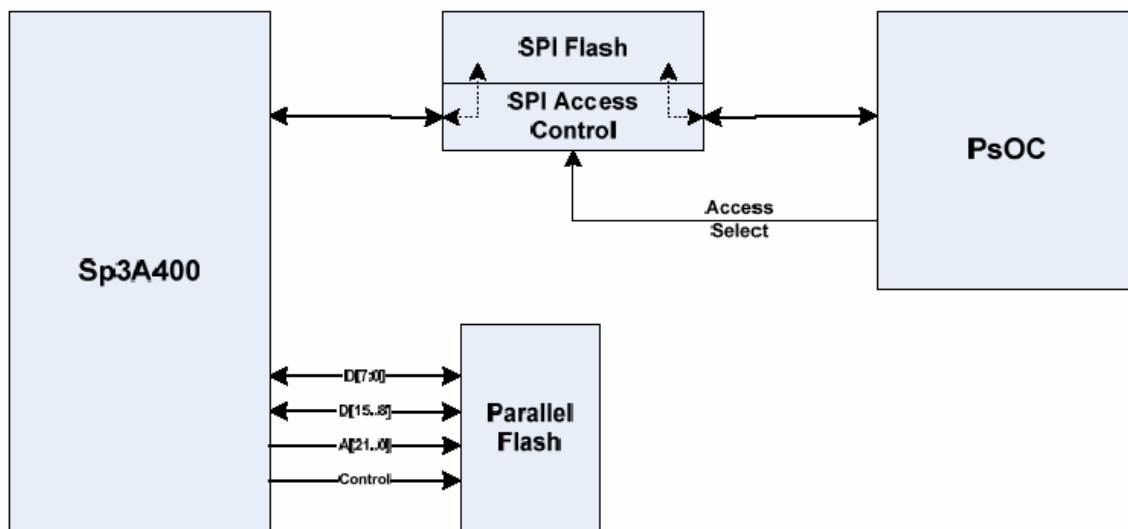


Figure 25 Memory Interfaces

Parallel Flash:

Parallel Flash memory consists of a single 32 Mbit Spansion S29GL032N in a TSOP-56 package interfaced to the FPGA. This device (U22) may be utilized in a 4 M x 8 or a 2 M x 16 configuration by control of the Flash_BYTE# signal. Flash_BYTE# is pulled low through a 10 K resistor to establish the 4 M x 8 default configuration that may be used for FPGA configuration. Following configuration, the FPGA may drive Flash_BYTE# high to establish the 2 M x 16 configuration. Jumper JP1 may be used to write-protect the Flash memory by placing a shunt across pins 1 and 2; default setting is JP1 open. Note the PCB layout also supports the same Spansion Flash device in a TSSOP-48 package. Table 12 provides the FPGA/FLASH pinout.

Parallel Flash Signal	FPGA Pin#	Parallel Flash Signal	FPGA Pin#
Flash CE#	P15	Flash D0	T14
Flash OE#	R15	Flash D1	R13
Flash WE#	N13	Flash D2	T13
Flash RY/BY#	A4	Flash D3	P12
Flash BYTE#	N14	Flash D4	N8
FLASH RESET#	T10	Flash D5	P7
Flash A0	P16	Flash D6	T6
Flash A1	N16	Flash D7	T5
Flash A2	L13	Flash D8	P11
Flash A3	K13	Flash D9	R3
Flash A4	M15	Flash D10	N11
Flash A5	M16	Flash D11	N7
Flash A6	L14	Flash D12	R5
Flash A7	L16	Flash D13	T4
Flash A8	J12	Flash D14	P6
Flash A9	J13	Flash D15	N14 (Flash A0)
Flash A10	G16		
Flash A11	F16		
Flash A12	H13		
Flash A13	G14		
Flash A14	E16		
Flash A15	F15		
Flash A16	G13		
Flash A17	F14		
Flash A18	E14		
Flash A19	F13		
Flash A20	D16		
Flash A21	D15		

Table 12 Parallel Flash Interface Pinout

Serial SPI Flash:

128 Mbits of serial Flash memory is provided by a Spansion S25FL128P device (U19) interfaced to the Spartan- 3A FPGA via its dedicated SPI interface, and to the Cypress PSoC device via a 2:1 multiplexer (U20). This multiplexer is controlled by the PSoC, which is master of this SPI interface. There are two SPI modes as depicted in Figure 25; PSoC/FPGA \leftrightarrow SPI Flash (PSoC_SPI_MODE=0) and PSoC \leftrightarrow FPGA (PSoC_SPI_MODE=1).

In the PSoC/FPGA \leftrightarrow SPI Flash mode, either the PSoC or the FPGA may access the SPI Flash by driving its select line low. This is done through an AND gate (U21) that will drive the SPI Flash's chip select line (SF_SEL#) low in response to a low select signal from the FPGA (FPGA_SPI_SEL#) or the PSoC (PSoC_SPI_SEL). Note that the SPI clock line (SPI_CLK) may be driven by either the PSOC or the FPGA. Since this

configuration has the potential for conflict, if the PSoC intends to access the SPI Flash, it must drive the FPGA's PROG_B pin low to place the FPGA in a reset state to prevent it accessing the SPI Flash.

In the PSoC \leftrightarrow FPGA mode (PSoC_SPI_MODE=1) the multiplexer is configured to interconnect the PSoC and FPGA SPI interfaces; the purpose of this is to enable slave serial configuration from PSoC to FPGA. In this mode the PSoC is master and the FPGA will act as slave. Since the FPGA's SPI interface is only active during SPI boot mode, implementation of the FPGA's slave interface must be accomplished via firmware. While this configuration happens to share common pins with the FPGA SPI port, they will function as slave serial in this mode. Table 13 provides the FPGA's SPI interface pinout.

6-pin header J8 may be used to allow the PSoC to expand the SPI interface to an external environment. Since the PSoC's SPI select signal (PSoC_SPI_SEL#) is common to the SPI Flash as well as the SPI expansion interface, the default jumper on JP6 must be removed prior to using the SPI expansion. Table 14 provides the J8 pinout.

Note that J8 pin 6 provides +3.3 V to another board; if that board is already powered then J8 pin 6 must not be connected.

Signal	FPGA Pin#
FPGA MOSI	P10
FPGA MISO	T14
SPI CLK	R14
FPGA SPI SEL	T2

Table 13 SPI Interface Pinout

Signal	J8 Pin#
PSoC SPI SEL#	1
SPI FLASH SI	2
SPI FLASH SO	3
SPI CLK	4
GND	5
+3.3V	6

Table 14 SPI Header J8 Pinout

Interfaces:

Interfaces on the Spartan-3A evaluation board consist of USB 2.0 via the PSoC, two 0.1” 6-pin right-angle headers designed to T M interface to Digilent modules, a 0.1” 2 x 20 header providing connectivity to available FPGA general-purpose I/O pins, a 0.1” 1 x 6 header for SPI interface expansion, and a sensor providing temperature information via an I²C interface.

USB 2.0:

USB Mini-AB connector P1 connects the PSoC device to a full-speed (12 Mbps) USB host. Power supplied by the USB host via connector P1 (+5V_USB) may be used to power the Spartan-3A evaluation board by jumpering JP2 1:2.

USB-UART:

The USB-UART interface is used for communication between the PSoC and the FPGA but is not utilized externally (e.g., there is no RS-232 connector). This interface operates at 3.3 V and is the mechanism by which the FPGA communicates via USB; e.g., the PSoC device provides UART/USB translation. Note that the net names UART_RXD and UART_TXD on the schematic are named in terms of the PSoC connection. Net UART_RXD is an output from the FPGA and an input to the PSoC, as shown by the direction of the off-page connectors on the schematic. The FPGA Tx signal is connected to the PSoC Rx signal and then the PSoC re-broadcasts the data to the USB. For incoming data from USB, the PSoC transmits on the UART_TXD net which is actually an Rx for the FPGA.

Net Name	Description	FPGA Pin #
FPGA_RS232_Rx	Received Data, RD (Transmitted by PSoC)	A3
FPGA_RS232_Tx	Transmit Data, TD (Received by PSoC)	B3

Table 15 USB-UART Signals

Digilent Headers:

Two right-angle, 6-pin (1 x 6 female) Digilent headers (J6, J7) are interfaced to the FPGA, with each header providing 3.3 V power, ground, and four I/O's. These headers may be utilized as general-purpose I/Os or may be used to interface to Digilent modules. J6 and J7 are placed in close proximity (0.9"-centers) on the PCB in order to support dual Digilent modules. Figure 26 shows the pinout of the Digilent headers; Table 16 provides the FPGA pinout.

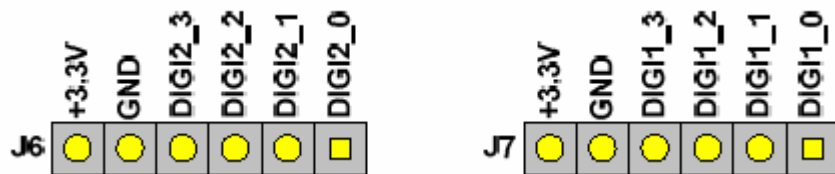


Figure 26 Digilent Header Pinout

J6 Signal	FPGA Pin#
DIGI2_0	N1
DIGI2_1	M1
DIGI2_2	K1
DIGI2_3	G1
J7 Signal	FPGA Pin#
DIGI1_0	R1
DIGI1_1	P2
DIGI1_2	P1
DIGI1_3	N2

Table 16 Connections

Miscellaneous I/O:

Four user push button switches are provided via capacitive touch-pads connected to the Cypress PSoC device. A “touch” at any of these four pads is sensed by the PSoC and forwarded to the FPGA; these “push buttons” and their relationship to the FPGA are depicted in Table 17. Note that FPGA_RESET is a “soft” reset intended for FPGA code usage and does not perform any type of FPGA hardware reset.

PSoC Cap Sense	FPGA "Pushbutton"	FPGA Pin#
EF1	FPGA PUSH A	K3
EF2	FPGA PUSH B	H5
EF3	FPGA PUSH C	L3
EF4	FPGA RESET	H4

Table 17 Connections

LEDs:

Four LEDs are provided for signaling purposes and connected to the FPGA as shown in Table 18. The corresponding FPGA pin must be driven high to light an LED.

LEDs	FPGA Pin#
LED1 (D5)	D14
LED2 (D4)	C16
LED3 (D3)	C15
LED4 (D2)	B15

Table 18 LED Assignment

GPIO Header (2 x 20):

Some unused FPGA pins are connected to 0.1" 2 x 20-pin header J4. Signal names and connector pin/FPGA pin connections are identified in Table 19. All I/O's are +3.3 V CMOS.

FPGA pin #	I/O Signal	Connector Pin #	Connector Pin #	I/O Signal	FPGA pin #
n/a	GND	1	2	+5V	n/a
n/a	+3.3V	3	4	BANK0 IO2	C4
A14	BANK0 IO1	5	6	BANK0 IO4	B14
A13	BANK0 IO3	7	8	BANK0 IO6	D13
C13	BANK0 IO5	9	10	BANK0 IO8	C12
A12	BANK0 IO7	11	12	BANK0 IO10	D11
B12	BANK0 IO9	13	14	BANK0 IO12	C11
A11	BANK0 IO11	15	16	BANK0 IO14	D10
A10	BANK0 IO13	17	18	BANK0 IO16	E10
A9	BANK0 IO15	19	20	BANK0 IO18	D9
C9	BANK0 IO17	21	22	BANK0 IO20	C8
A8	BANK0 IO19	23	24	BANK0 IO22	E7
B8	BANK0 IO21	25	26	BANK0 IO24	D8
A7	BANK0 IO23	27	28	BANK0 IO26	D7
C7	BANK0 IO25	29	30	BANK0 IO28	C6
A6	BANK0 IO27	31	32	BANK0 IO30	C5
B6	BANK0 IO29	33	34	BANK3 IO2	D4
A5	BANK0 IO31	35	36	BANK0 IO32	B4
E13	BANK1 IO1	37	38	BANK3 IO1	D3
	GND	39	40	GND	

Table 19 GPIO Pin Assignment

I²C Temperature Sensor:

A Texas Instruments TMP100 digital temperature sensor is interfaced to the PSoC via an I²C interface. The TMP100 has two address pins to set the low-order I²C slave address bits; both pins are pulled low in this application providing an address of 0x90 (W) and 0x91 (R). The TMP100 will provide temperature readings over its specified operating temperature, -55 °C to +125 °C; well beyond the ability of the Spartan-3A evaluation board to operate.

Module Clocks:

Three clocks are provided to the FPGA; 16.0 MHz from a Maxim MAX7381 CMOS oscillator (U6), and 12.0 MHz and 32.0 kHz from the PSoC. Table 20 provides FPGA connection details.

Clocks	FPGA Pin#
16.0MHz	C10 (GCLK4)
12.0MHz	N9 (GCLK0)
32.0kHz	T7 (GCLK13)

Table 20 Module Clocks

Configuration:

The Spartan-3A evaluation board provides four mechanisms to program and configure the FPGA; these are JTAG, Parallel Flash, Serial Flash, and the Cypress PSoC. The storage devices (Flash and SPI) cannot be programmed via the JTAG connector. The FPGA is the only thing in the JTAG chain on the Spartan-3A evaluation board; however, depending on the setting of configuration jumpers M [2:0], any of these can be the configuration source. The serial Flash, Parallel Flash, and PSoC are described earlier in this document.

Programming the Spartan-3A evaluation board via Boundary Scan requires that a JTAG download cable be attached to the 14-pin 2 mm spaced header J5 (Figure 27) with a ribbon cable or with flying leads. If the Xilinx Parallel Cable IV is used, the ribbon cable connector mates with the keyed J5 connector.

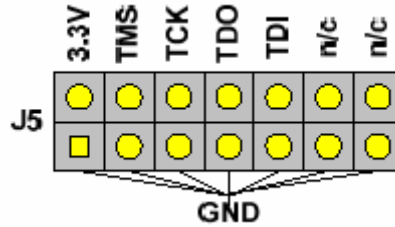


Figure 27 Parallel IV Connector

Configuration Modes:

The following table shows the Spartan-3A evaluation board configuration modes set by Jumper JP4. All mode jumpers (including the PUDC_B pin) are pulled high, with jumper installation grounding the connection. Adding a jumper to the MODE pins ties them to a pull-down that is stronger than the default pull-up. This is necessary in order for the PSoC to overdrive the MODE jumpers regardless of whether or not the jumpers are installed. Figure 6 depicts configuration jumper JP4; Table 21 provides the various configuration settings at JP4, with recommended settings highlighted. A push button labeled “PROG” (SW1) is pulled high and connected to the FPGA PROG via AND gate U9; also connected to U9 is the PSoC (PSOC_FPGA_PROG). Pushing SW1 (or driving PSOC_FPGA_PROG low) activates the FPGA programming mechanism. Upon releasing SW1 (or PSOC_FPGA_PROG going high), a re-configuration is initiated based upon the setting of JP4. A blue LED (D7) should light when FPGA “DONE” is asserted.

Mode	PC Pull-up	Configuration Mode Jumpers			
		1-2 (M2)	3-4 (M1)	5-6 (M0)	7-8 (PUDC_B)
Master Serial	Yes	Closed	Closed	Closed	Closed
Master Serial	No	Closed	Closed	Closed	Open
Slave Serial	Yes	Open	Open	Open	Closed
Slave Serial	No	Open	Open	Open	Open
Master SPI	Yes	Closed	Closed	Open	Closed
Master SPI	No	Closed	Closed	Open	Open
BPI Up	Yes	Closed	Open	Closed	Closed
BPI Up	No	Closed	Open	Closed	Open
Slave Parallel	Yes	Open	Open	Closed	Closed
Slave Parallel	No	Open	Open	Closed	Open
JTAG	Yes	Open	Closed	Open	Closed
JTAG	No	Open	Closed	Open	Open

Table 21 JP4 Settings

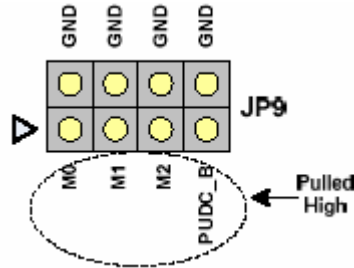


Figure 28 Configuration Jumper (JP4)

Module Power:

The Spartan-3A evaluation board requires a +5 V input at barrel jack J3 or +5 V via USB cable. Jumper JP2 is used to select between the barrel jack (JP2 = 2:3) or USB power (JP2 = 1:2). LED D1 should be illuminated when power is applied. Jumper JP7 1:2 selects the barrel jack/USB input power; JP7 2:3 is not applicable. Note that the barrel jack requires a 2.1 mm plug.

Application of 5 V power is sensed by a Texas Instruments TPS3809K33 Voltage Supervisor. When power is above the TPS3809's threshold, its active-low reset output is driven high supplying the enable for a Texas Instruments TPS62290 1A step-down converter (U5) to supply the +3.3 V rail. The 3.3 V rail provides the enable (a Texas Instruments TPS3106K33 Voltage Supervisor) to a second TPS62290 (U24) which

supplies the +1.2 V rail. When the +1.2 V rail is above the TPS3106's threshold, its active-low reset output is released allowing the power-on reset signal (PO_RESET#) to go high. As mentioned above, pushbutton switch SW1 may be used to momentarily force (via AND gate U23) PO_RESET# low.

Note that 0-ohm jumper JT1 may be utilized to set the operating mode of the TPS62290 converter; JT1 = 1:2 (default) sets fixed-frequency PWM mode, JT1 2:3 sets power-save mode (automatic PFM/PWM switching).

Figures 29, 30 and 31, below, show details of the +3.3 V and +1.2 V power supplies. Figure 29 shows that +1.2 V power (bottom trace) is delayed 114 ms from +3.3 V power. Using a finer scale, Figures 30 and 31 shows the rise of +3.3 V and +1.2 V power (respectively) is monotonic and glitch-free.

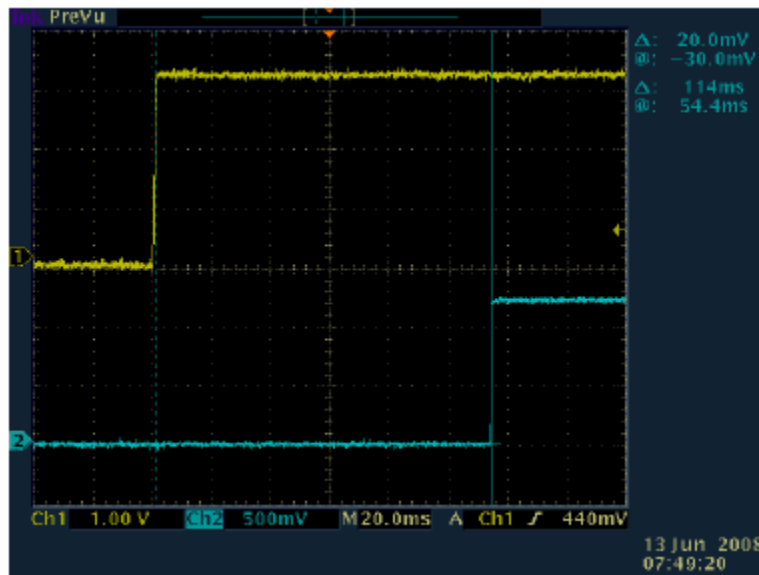


Figure 29 Power Supply Sequencing

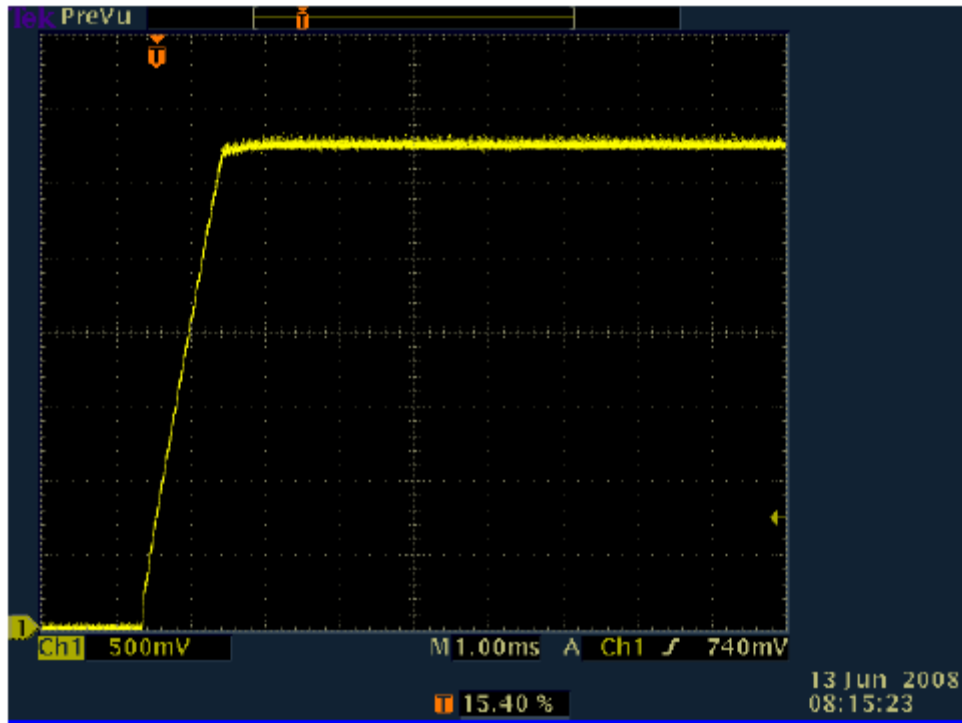


Figure 30 +3.3v Power Supply Startup

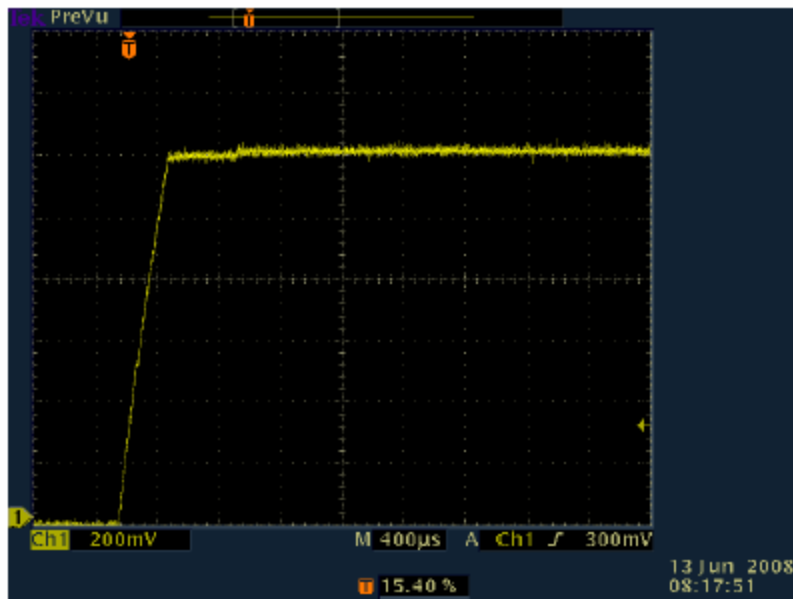


Figure 31 +1.2v Power Supply Startup

PCB Stackup:

Figure 32 shows 4-layer stacks up of the Spartan-3A Evaluation Kit Printed Circuit Board (PCB). The PCB substrate is FR4- class epoxy glass with 1/2oz copper used for all layers.

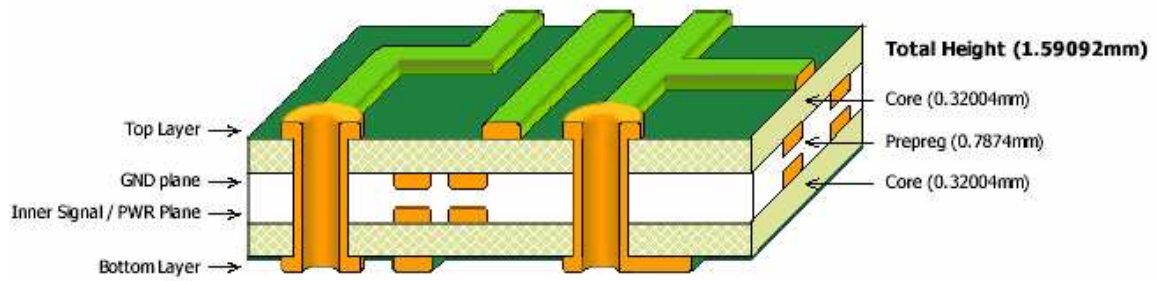


Figure 32 PCB Layer Stack