Chapter 4
Know Your Tools

In this chapter, we will learn how to realize sequential and combinaional logic circuits we designed in the previous chapters. To do this, we are going to adopt an automated modern process that relies on the use of Computer Aided Design (CAD) tools that converts a circuit description into an implementation. In this book, we focus only on the Field Programmable Gate Arrays as an implementation technology. Also, we use Verilog Hardware Description Language as a mean to describe the circuit we want to implement.

1. Modern Techniques for Digital Design

- Architecture design. This stage involves analysis of the project requirements, problem decomposition and functional simulation (if applicable). The output of this stage is a document which describes the future device architecture, structural blocks, their functions and interfaces.
- HDL design entry. The device is described in a formal hardware description language (HDL). The most common HDLs are VHDL and Verilog.
- Test environment design. This stage involves writing of test environments and behavioral models (when applicable). They are later used to ensure that the HDL description of a device is correct.
- Behavioral simulation. This is an important stage that checks HDL correctness by comparing outputs of the HDL model and the behavioral model (being put in the same conditions).
- Synthesis. This stage involves conversion of an HDL description to a so-called netlist which is basically a formally written digital circuit schematic. Synthesis is performed by a special software called synthesizer. For an HDL code that is correctly written and simulated, synthesis shouldn't be any problem. However, synthesis can reveal some problems and potential errors that can't be found using behavioral simulation, so, an FPGA engineer should pay attention to warnings produced by the synthesizer.

- Implementation. A synthesizer-generated netlist is mapped onto particular device's internal structure. The main phase of the implementation stage is place and route or layout, which allocates FPGA resources (such as logic cells and connection wires). Then these configuration data are written to a special file by a program called bitstream generator.
- Timing analysis. During the timing analysis special software checks whether the implemented design satisfies timing constraints (such as clock frequency) specified by the user.

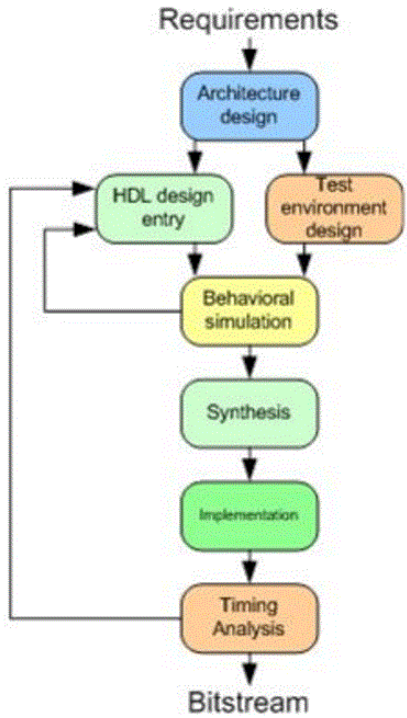


Figure 1.

2. Hardware Description Languages

As digital designs are becoming complex, the conventional way of designing digital circuits starting from schematic became unsuitable. That led to the introduction of Hardware Description Languages (HDL) and Electronic Design Automation (EDA) tools.

In 1971 it took less than one hundred people to manufacture the 4004. That is approximately 23 transistors per employee. If that ratio stayed the same between 1971 and 2012, Intel would need to employ about 65 *million* people just to

produce the latest Core i7 processor. That is **one fifth** the entire population of the United States!

An HDL is a high-level programming language that offers special constructs with which you can model micro-electronic circuits. These special language constructs permit you to:

- Describe the operation of a circuit at various levels of abstraction
- Describe the timing of a circuit
- Express the concurrency of circuit operation

A digital circuit can be modeled (described) using an HDL then synthesized and mapped to any of the possible implementation technologies (e.g., FPGA, ASIC, etc.,). Also, HDLs can be used to verify digital circuits through simulation.

Unfortunately, some very popular HDLs were never intended to be used as they are today. The origins of hardware description languages were rooted in the need to document the behavior of hardware. Over time, it was recognized that the descriptions could be used to simulate hardware circuits on a general-purpose processor. This process of translating an HDL source into a form suitable for a general-purpose processor to mimic the hardware described is called simulation . Simulation has proven to be an extremely useful tool for developing hardware and verifying the functionality before physically manufacturing the hardware. It was only later that people began to synthesize hardware, automatically generating the logic configuration for the specified device from the hardware description language. Unfortunately, while simulation provided a rich set of constructs to help the designer test and analyze the design, many of these constructs extend beyond what is physically implementable within hardware (on the FPGA) or synthesize inefficiently into the FPGA resources. As a result, only a subset of hardware description languages can be used to synthesize designs to hardware. The objective of this section is to present two of the more popular hardware description languages, VHDL and Verilog. It is arguable to which is better suited for FPGA design; both are presented here for completeness, but it is left to the readers to decide which (if any) is best suited for their needs. We will focus on VHDL throughout the remainder of this book as it becomes redundant to give two examples, one in VHDL and one in Verilog, for every concept. That being said, we will also focus within this section on synthesizable HDL and how it maps to the previous section's components.

2.1 Simulation

Humans routinely make mistakes. Such errors in hardware designs are called bugs. Eliminating the bugs from a digital system is obviously important, especially when customers are paying money and lives depend on the correct operation. Discovering the cause of errors in the lab can be extremely difficult, because only signals routed to the chip pins can be observed. There is no way to

directly observe what is happening inside a chip. Correcting errors after the system is built can be devastatingly expensive. For example, correcting a mistake in cutting-edge integrated circuit costs more than one million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a total cost of $475 million. So, verifying the design correctness during the design process is an essential ingredient for a successful product. Simulation can be used to verify the digital system during the design process.

Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It, also, permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. To simulate an HDL model, an engineer writes a top-level simulation environment (called a testbench).

2.2 Synthesis

Logic synthesis transforms HDL code into a netlist describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer performs optimizations to reduce the amount of hardware required and/or makes the circuit faster. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit.

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. HDL languages are rich languages with many constructs. Not all of these constructs can be synthesized into hardware. One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

3. The Verilog HDL

Verilog is the most used HDL to design digital systems. Verilog allows the designer to represent digital circuits in many ways. The history of the Verilog HDL goes back to the 1980s, when a company called Gateway Design Automation developed a logic simulator, Verilog-XL, and with it a hardware description language (Verilog). Verilog was designed to make the simulator go fast. Cadence Design Systems acquired Gateway in 1989, and with it the rights to the language and the simulator. In 1990, Cadence put the language (but not

the simulator) into the public domain as a response to the growing popularity of VHDL, with the intention that it should become a standard, non-proprietary language. The Verilog HDL is now maintained by a nonprofit making organization, Accellera, which was formed from the merger of Open Verilog International (OVI) and VHDL International. OVI had the task of taking the language through the IEEE standardisation procedure. In December 1995 Verilog HDL became IEEE Std. 1364-1995. A significantly revised version was published in 2001: IEEE Std. 1364-2001. There was a further revision in 2005 but this only added a few minor changes. Accellera have, also, developed a new standard, SystemVerilog, which extends Verilog. SystemVerilog became an IEEE standard (1800-2005) in 2005. SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design verification and design modeling. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009). The current version is IEEE standard 1800-2012. The feature-set of SystemVerilog can be divided into two distinct roles:

- *SystemVerilog for RTL design* is an extension of Verilog-2005; all features of that language are available in SystemVerilog.
- *SystemVerilog for verification* uses extensive object-oriented programming techniques and is more closely related to Java than Verilog.

4. Field Programmable Gate Array (FPGA)

As discussed earlier, In the 60's there were discrete logic chips. Systems were built from lots of individual ANDs, ORs, flip-flops, etc. with a spaghetti-like maze of wiring between them. It was difficult to modify such a system after you built it. As the number of transistors per chip started to increase by 1970's, as predicted by Gordon Moore (Moore's Law), the systems that used to be built using discrete logic chips could be built on a single chip. Designing a custom ASIC is a time-consuming process that involves a lot of engineering effort and requires big upfront investment. That makes ASIC suitable only for applications that sell large number of units.

Around the beginning of the 1980s, it became apparent that there was a "gap" in the digital integrated circuit continuum. On the one hand, there were off the shelf discrete logic chips (e.g., TTL chips) which couldn't support large or complex functions. At the other end of the spectrum were ASICs that support extremely large and complex functions, but they are expensive and time-consuming to design. Furthermore, once a design is implemented as an ASIC, it's fixed and cannot be changed. In order to address this gap, in mid 1980s, a company called Xilinx introduced a new class of integrated circuit to the market; this new component was called a Field Programmable Gate Array (FPGA). Xilinix first FPGA was called XC2064™. Around the same time, a company

called Altera was also developing a programmable device, later to become the EP1200, which was the first high density programmable logic device (PLD).

FPGAs are off-the-shelf silicon chips whose function can be programmed by design engineers. This means that one chip can be re-programmed to perform many different functions. Also, the chip can be re-programmed to fix bugs or add new functionality in field. A feature is not possible for custom ASIC. the programmability was based on either SRAM or FLASH. There are a large variety of FPGAs from different vendors, each with different capabilities, advantages, and limitations. FPGAs are an example for PLDs (Programmable Logic Devices). There are other types of PLDs such as Complex Programmable Logic Devices (CPLDs) but they are not covered by this book.

4.1 FPGA Architecture

An FPGA, as shown on Figure 5, is a large array of simple, configurable/programmable logic blocks (CLBs), configurable I/O blocks (IOB), and programmable interconnect switching matrix (SM). Also, there is clock circuitry for driving the clock signals to each logic block. In addition to their underlying programmable fabric, different FPGAs contain different combinations of hard macro blocks, including blocks of on-chip RAM, adders, multipliers, Digital Signal Processing (DSP) functions, processor cores, etc.

Each configurable logic block is individually programmed/configured to perform any logic function of n-variables and then the switches are programmed to connect the blocks so that the complete logic functions are implemented. Only after the configuration step, the FPGA is able to operate with the desired behavior encoded into the configuration.

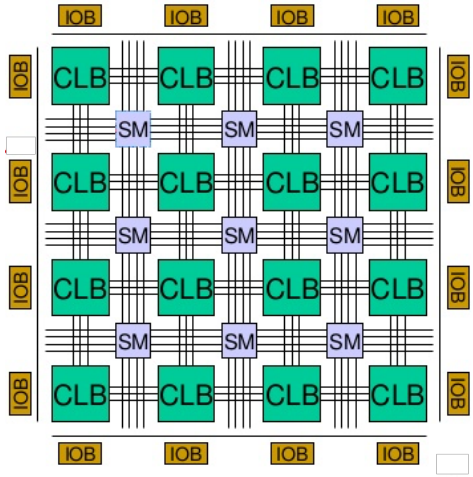


Figure 6. FPGA Architecture

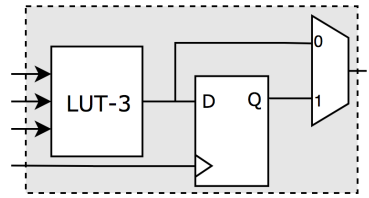
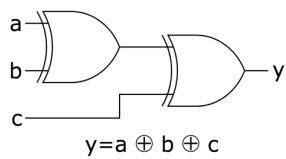


Figure 7. Configurable Logic Block

A typical CLB is made out of a Look-Up Table (LUT), a Flip-Flop (FF) and a multiplexer as shown on Figure 6. The LUT is implemented using Multiplexers and memory cells and is used to realize a truth table as shown by Figure 7. The idea of using a multiplexer to implement a combinational logic function is discussed in Chapter 1. The LUT memory cells could be SRAM or FLASH. The vast majority of today's FPGA fabrics are based on a SRAM LUT architecture. The first FPGAs used 3-input LUT (LUT3); nowadays FPGAs use 6-input LUT (LUT6). Increasing the size of the LUT makes the FPGA more efficient. For example, a logic function with seven inputs could be implemented in eight LUT4s or two LUT6s. To support sequential logic a flip flop is added to the CLB. The function implemented by the LUT is stored by the CLB FF. The output of the CLB is either connected to the LUT output or the FF output. This is determined by the CLB multiplexer. The multiplexer selection line is connected to a memory cell which can be programmed to select the output of the CLB. The bits stored in the LUT memory cells as well as the multiplexor selection memory cell are called the configuration bits of the CLB. SRAM based FPGA are volatile. They cannot keep the configuration bits when the FPGA is powered off. In the other hand, FLASH based FPGA are non-volatile.

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



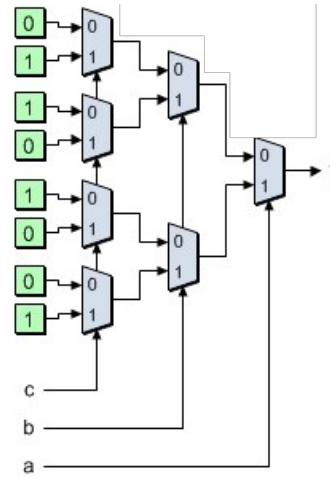


Figure 7. Look Up Table (LUT)

The Configurable input/output (I/O) Blocks (IOBs) provide a programmable interface between the internal; array of logic blocks (CLBs) and

the device's external package pins. Each IOB consists of an input buffer and an output buffer with three-state output control as shown by Figure 8.

CLBs connect to each other through programmable routing network. This programmable routing network provides routing. connections among logic blocks and I/O blocks to implement any user-defined circuit. The routing interconnect of an FPGA consists of vertical and horizontal wires and Programmable Switches Matrix (PSM) that form the required connection. These programmable switches are configured using the programmable technology. The PSM connects horizontal and vertical wires. Inside the PSM pass transistors are used to establish such connections as shown on Figure 9. Every diamond inside the PSM is called Programmable Switching Element (PSE) and is made out of 6 transistors. By turning on and off the pass transistors, several connections can be established. Every time a signal passes through a transistor it suffers from a small delay. Hence, it is important to place connected CLBs very close to each other's.

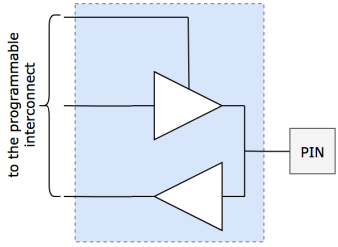
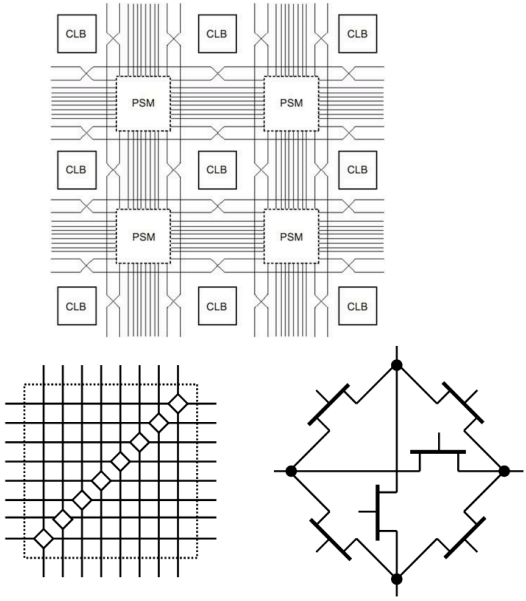


Figure 8. I/O Block (IOB)



(a) FPGA Routing Resources (b) PSM (c) PSE

Figure 9. Programmable Routing Resources on FPGA

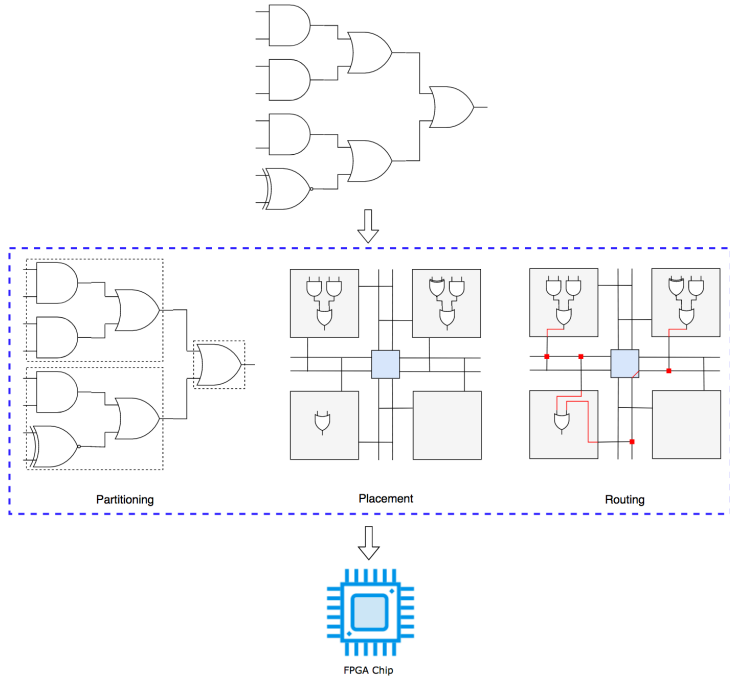


Figure 11. From Circuit to Configuration bits

4.2 FPGA Configuration

Figuring out which bits to set in the LUTs and the PSMs in order to create a logic circuit is quite complex. Not many people would want to delve into the details of this (probably the same ones who like to program in assembly). That's why the FPGA manufacturers provide development software that translates the circuit into configuration bits. The process of mapping a circuit into the FPGA fabric involves:

- **Partitioning**: dividing the circuit into set of sub-circuits each is small enough to be implemented by a single CLB.
- **Placement**: each sub-circuit, then, is mapped to open of the available CLBs on the FPGA.
- **Routing**: PSMs are configured to connect the input and the output of the used CLBs as well as the IOB together to realize the circuit.

The placement and routing steps are tightly coupled. Placement has significant impact on the performance and routability of circuit design.

Starting with a circuit schematic is not practical for non-trivial designs. For that modern design flows starts from a high-level description of the circuit or the system to be implemented and rely on Electronic Design Automation (EDA) software tools to converts the description into an implementation. Hardware Description Languages (HDL) such as Verilog and VHDL have been developed that allow the description of the behavior and the structure of a digital system in a readable and simulatable form. Synthesis tools have been developed that can derive a gate-level hardware implementation of a digital system from a behavioral description in an HDL. the availability of logic synthesis tools is one of the driving forces behind the growing use of HDLs. Logic synthesis transforms an RTL description of a circuit in an HDL into an optimized netlist representing storage elements and combinational logic. Subsequently, this netlist may be transformed by using physical design tools into an actual integrated circuit layout. This layout serves as the basis for integrated circuit manufacture. The logic synthesis tool takes care of a large portion of the details of a design and allows exploration of the cost/performance trade-offs essential to advanced designs.

A typical HDL-based design flow for FPGA is illustrated by Figure 13. The flow starts with RTL (Register Transfer Level) modeling for the circuit. RTL is a design abstraction which models a synchronous digital circuit in terms of the data flow between hardware registers, and the logical operations performed on those signals. The RTL model is first verified through simulation to verify that the design performs the expected and required functions. Simulation is the process of applying stimulus or inputs that mimic actual data to the design and observing the output. Simulation is carried out using HDL simulator. Once verified, the RTL is first synthesized into a netlist. The netlist is just HDL description of the various logic gates in your design and how they are interconnected (logic circuit). The netlist is simulated using the RTL verification test cases to verify that its equivalent to the RTL. The netlist, then, is partitioned, placed and routed to generate the configuration bitstream which is used to program the FPGA. Once the routed, the interconnect information is extracted from the design to perform timing verification (verifying that the design will run given a clock frequency) either through simulation or static timing analysis (STA).

Programming the FPGA involves transferring the bit stream into a nonvolatile or volatile memory device and configuring or programming the FPGA. However, some FPGAs have internal memory and can hold the configuration without an external memory device. Input to the programming phase is a bit-stream or programming file and the output is a programmed device, see Figure 8–1. Configuration can involve one or a series of daisy-chained or connected FPGAs. Nonvolatile devices, such as FLASH, may be located on the same board as the targeted FPGA (see Figure 8–2). The configuration may involve transferring serial or parallel data to the FPGA. The FPGA may be operating in either master (controlling

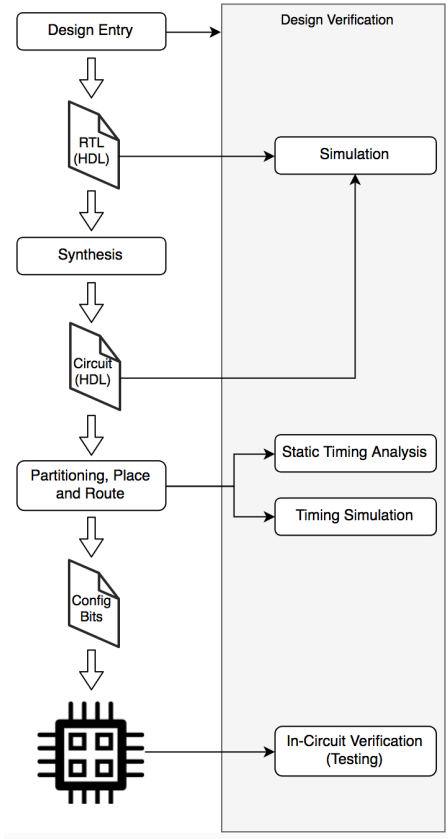


Figure 13. Typical HDL-based FPGA Flow

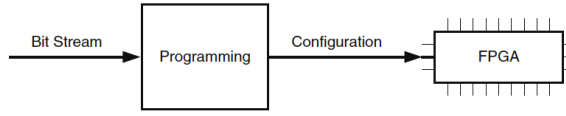


Figure 8–1: Programming Interfaces. Note: Multiple pins are represented by thick pin lines.

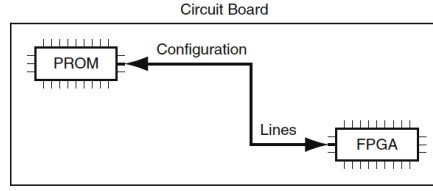


Figure 8–2: Nonvolatile and FPGA on the Same Board. Note: Multiple pins are represented by thick pin lines.

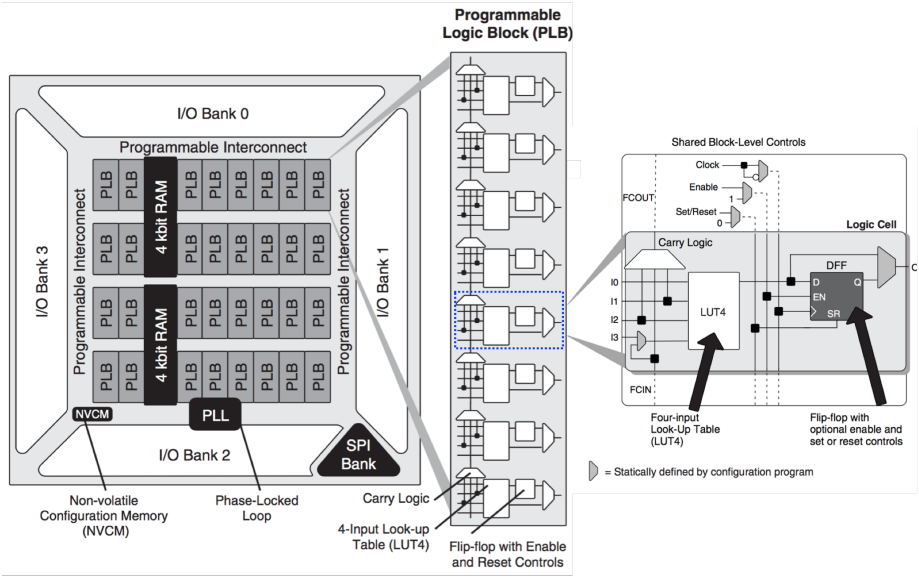


Figure 14. ICE40 Family Architecture

4.3 The Lattice Semiconductor ICE40

The iCE40 family of ultra-low power, non-volatile FPGAs has devices with densities ranging from 384 to 7680 Look-Up Tables (LUTs). In addition to LUT-based, low-cost programmable logic, these devices feature Embedded Block RAM (EBR), Non-volatile Configuration Memory (NVCM) and Phase Locked Loops (PLLs). These features allow the devices to be used in low-cost, high-volume consumer and system applications. Select packages offer High-Current drivers that are ideal to drive three white LEDs, or one RGB LED.

iCE40 is SRAM based FPGA. As shown on Figure 14, it is constructed as an array of programmable logic blocks (PLBs), where a PLB is a block of eight logic cells. Each logic cell consists of a four-input lookup table (LUT4) with the output connected to a D flip-flop. Within a PLB, each logic cell is connected to the following and preceding cell by carry logic, intended to improve the performance of circuits such as adders.

Larger iCE40 device includes multiple high-speed synchronous sysMEM Embedded Block RAMs (EBRs), each 4 kbit (kilobits) in size. This memory can be used for a wide variety of purposes including data buffering, and FIFO. sysMEM Memory Block The sysMEM block can implement single port, pseudo dual port, or FIFO memories with programmable logic resources. Each block can be used in a variety of depths and widths. If desired, the contents of the RAM can be pre-loaded during device configuration. By preloading the RAM block during the chip configuration cycle and disabling the write controls, the sysMEM block can also be utilized as a ROM. Note the sysMEM Embedded

Block RAM Memory address 0 cannot be initialized. Larger and deeper blocks of RAM can be created using multiple EBR sysMEM Blocks.

Moreover, the iCE40 architecture provides up to two sysCLOCK Phase Locked Loop (PLL) blocks. The PLLs have multiply, divide, and phase shifting capabilities that are used to manage the frequency and phase relationships of the clocks.

Every device in the family has a SPI port that supports programming and configuration of the device and an on-chip Non-Volatile Configuration Memory (NVCM). There are many resources provided in the iCE40 devices to route signals individually with related control signals. The routing resources consist of switching circuitry, buffers and metal interconnect (routing) segments.

iCE40 devices have power-on reset (POR) circuitry to monitor the supply voltage levels during power-up and operation. At power-up, after the supply voltage levels are stable, POR circuitry then triggers download from the on-chip NVCM or external Flash memory. iCE40 devices support various ways to configure the Configuration RAM (CRAM) including:

- Internal NVCM Download
- From a SPI Flash (Master SPI mode)
- System microprocessor to drive a Serial Slave SPI port (SSPI mode)

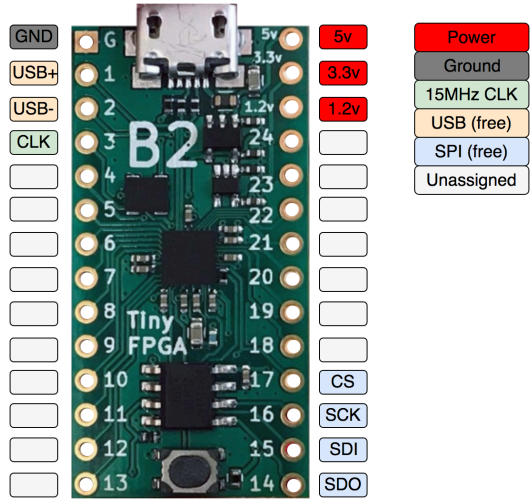


Figure 15. The TinyFPGA B2 Board Pinout

5. FPGA Boards

5.1 The TinyFPGA BX Board

The TinyFPGA B2 board, shown on Figure 15, uses Lattice Semiconductor's ICE40LP8K FPGA and has the following features:

- 16MHz clock,
- Voltage regulators (3.3v regulator used for I/O can supply up to 300mA),
- 4MBit SPI flash, and
- USB Interface for programming/configuration.

The board comes in a dual inline package (DIP) form factor that makes it suitable for usage in a breadboard or PCB socket. Out of the board 28 pins, 23 pins can be used by the user's application. To power the board, connect 5v power supply to the top right (5v) and top left pins (GND). The 3.3v pin can be used to power external components and it may supply up to 300mA. There is, also, 1.2v pin but it is not recommended to use. Pin 3 is tied to the onboard 16MHz clock source. Some of the pins (1, 2, 14-17) have a special function during the bootloading. Once the bootloading is done, they are available for the user to use.

The B2 board implements a USB bootloader that is able to reprogram the user configuration bitstream into the SPI flash. The bootloader automatically reboots the FPGA with the user configuration. Once the user configuration is running the bootloader no longer consumes any FPGA resources. Out of the 4MBit of SPI Flash, the USB Bootloader bitstream takes up about 1MBit, user design bitstream will take another 1MBit, the remaining 2Mbit are free to use for other purposes.

6. CloudV

CloudV is a cloud-based platform for designing digital integrated circuits (ASIC and FPGA). CloudV is built, mostly, using open-source EDA software tools. The ultimate goal of CloudV is to reduce the design costs by relying on cloud infrastructure and on collaborative design. Currently, CloudV v 1.0 allows designers to gain hands-on experience in FPGA design covering Verilog HDL design entry through a powerful in browser IDE (Figure 16), Verilog simulation with in browser waveform viewer (Figure 17), RTL synthesis and technology mapping as well as physical implementation. CloudV is easy to use and it relies on intuitive familiar web user interfaces. Moreover, designers can, virtually, use CloudV anywhere using any connected computing device. In this book, we are going to rely on CloudV for design entry, simulation and bitstream generation.

Add more stuff!

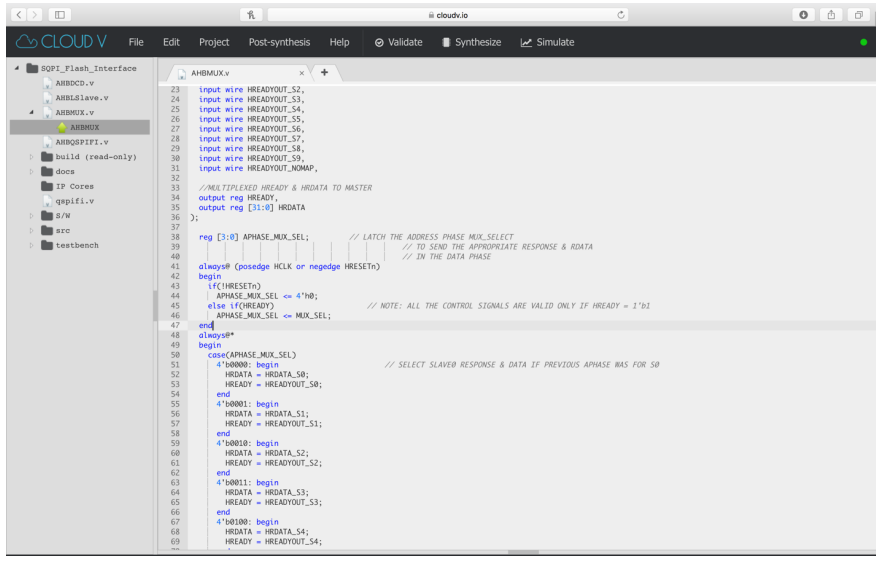


Figure 16. CloudV IDE

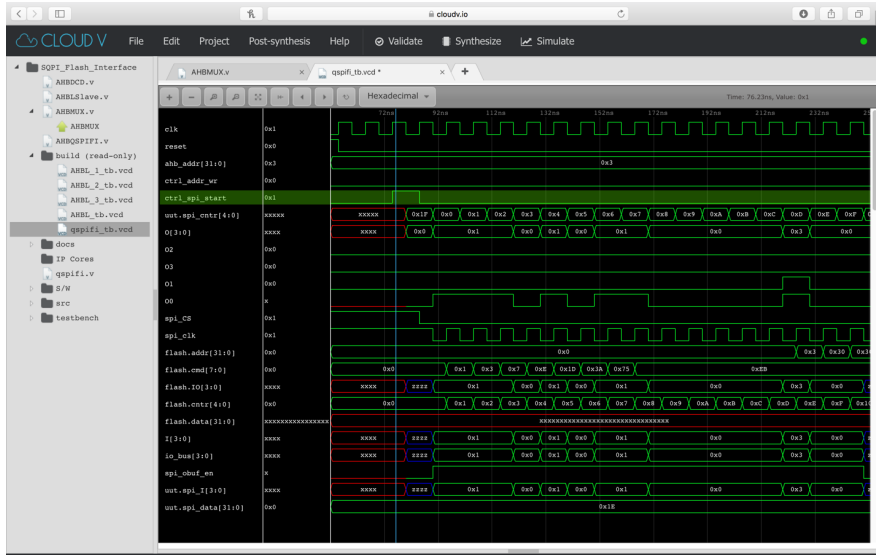


Figure 17. CloudV Waveform Viewer (FlexWave)

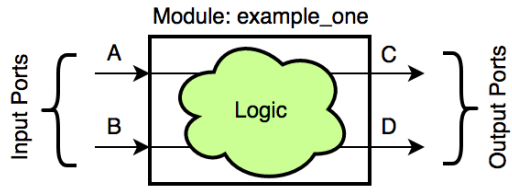
7. Summary

Chapter 5
Verilog for Combinational Logic

In this chapter, you will learn basic Verilog HDL constructs and how to use them to model and simulate combinational logic. All examples will be developed using Cloud V and tested on the TinyFPGA B2 board.

1. Verilog Module

In Verilog HDL, the basic unit of hardware is known as a module. In common with C-language functions, modules are free standing and cannot, therefore, contain definitions of other modules. A module can be instantiated within another module, this provides the basic mechanism for the creation of design hierarchy in a Verilog description.




```
module example_one(A, B, C, D);  // Module Header
   input A, B;
   output C, D;

   // Module Body

endmodule
```

Figure 1. Verilog Module

As outlined by Figure 1, the module declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The module body provides an "internal" view; it describes the behavior or the structure of the component. A module declaration starts with the keyword **module** and is terminated by the keyword **endmodule**. The keyword **module** starts the module header that included the module name (identifier) and the ports declaration. The module header is terminated by a semicolon. Inputs and outputs connections of the module are called ports and are declared with the keywords **input** and **output** respectively. In Verilog-2001,

the two declarations, direction and data type, may be combined in one statement. This combination is known as the ANSI-style.

```
module example_one(input A, B, output D, E);
```

Verilog comments follow the same syntax as C++ comments. As in C++, Comments come either as single lines or in block form. Single-line comments, start with **//** and continue until the end of the line. If the last character in a comment line is a **** the comment will continue in the next line. Multi-line comments (informally, C style), start with **/*** and end with ***/**.

Verilog requires that signals connected to the input or output of a module have two declarations: the port direction, and the data type of the signal. If no data type is declared, it is implicitly declared as a **wire** with the same size as the corresponding port.

2. Structural Modeling

With Verilog, a circuit can be defined by explicitly showing how to construct it using logic gates, predefined modules, and the connections between them. We call this structure modeling or glass-box modeling. Verilog includes a set of gate level primitives (listed by Table 1) that correspond to commonly used logic gates. In Verilog, a logic gate is represented by indicating its functional name, output and inputs. For example, the following code represent 2-input AND gate, named g1. The gate inputs connected to the wires x1 and x2 and its output is connected to the wire y.

```
and g1(y, x1, x2);                    // y is the o/p, x1 and x2
are the i/p
```

Optionally, the gate delay can be specified as per the following line of code. If not specified, the default delay is 0.

```
not #(2) g2(y1, y2, x);               //  y1  &  y2  wires  are
connected to the o/p;
                                  // x is the input; the delay is 2
                                  units
```

bufif1, bufif0, notif1, and notif0 implement 3-state buffers and inverters. They propagate z (high-impedance) if the control signal is deasserted. For example, a tri-state inverted with active-low control can be modeled as follows

```
notif0 c1 (bus, a, cntr);     // a is connected to bus if ctrl
is 0
```

Table 1. Verilog Gate Primitives

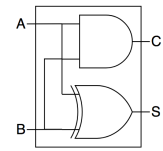
Type	Name	Terminals
Logic	**and, nand, or, nor, xor, xnor**	Output, Input(s)
Buffer and Inverter	**buf, not**	Outputs(s), Input

Tristate Logic	**bufif0, bufif1, notif0, notif1**	Output, Input, Enable

Figure 2.a gives the structural model of a Half Adder (HA) using Verilog primitives. Figure 2.b gives the structural model of a Full Adder (FA) using the Half Adder modules. The port connection can be an ordered or named list. In an ordered list, the signal connection must be in the same order as the port list in the module. Unconnected ports are designated by two adjacent commas. In a named list, the names must correspond to the ports in the module. A named port connection is only allowed for module instances.

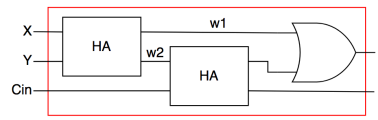
One-bit wires connecting component instances together do not need to be declared. Such wires are regarded as *implicit* wires. Note that implicit wires are only one bit wide, if a connection between two components is a bus, you must declare the bus as a wire vector.

```
module HA(A, B, S, C);
  input A, B;
  output S, C;
  and g1(C, A, B);
  xor g2(S, A, B);
endmodule
```



(a) Gate level model of a half adder

```
module FA(A, B, S, C);
  input X, Y, Cin;
  output Sum, Cout;
  wire w1, w2, w3;
  HA ha1(.A(X), .B(Y), ,.S(w2), .C(w1));
  HA   ha2(.A(w2),   .B(Cin),   ,.S(Sum),
.C(w3));
  or g1(Cout, w1, w3);
endmodule
```



(b) Structural Model of a Full Adder

Figure 2. Example of Verilog Structural Modeling

3. Data Flow Modeling

Using gate level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the function of a digital circuit. Verilog allows a circuit to be modeled in terms of the data flow between the circuit components and how a circuit processes data rather than instantiation of individual gates. Modeling/describing the circuit by expressing its function by showing how the data flows between its components is called Data Flow (or Functional) modeling. Verilog data flow modeling of combinational logic can be done using continuous assignment or asynchronous procedural blocks.

```
module example_two(A, B, C, D,
E);
input A, B, C;
output D, E;

wire w1 = A & B;
assign D = w1 | ~C;
assign E = ~C;

endmodule
```

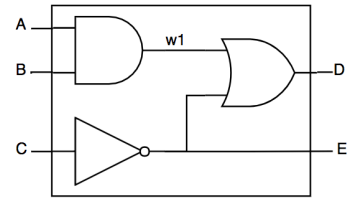


Figure 3. Functional Modeling Example

3.1 Continuous Assignment

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. Continuous assignment replaces gates in the description of the circuit and describes the circuit using Boolean expressions. Continuous assignment statements provide a Boolean correspondence between the right-hand side expression and the left-hand side target. The continuous assignment statement uses the keyword **assign**. Whenever any signal on the right-hand side changes its state, the value of left-hand side will be re-evaluated. You can assign only to wires. The wire will be inferred if not defined. Continuous assignment can be explicit or implicit. The explicit assignment requires two statements: one to declare the net, and one to continuously assign a value to it. The implicit continuous assignment combines the net declaration and continuous assignment into one statement. Figure 3 demonstrate the implicit and explicit continuous assignments.

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. Table 2 lists Verilog operators.

Table 2. Verilog Operators

Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	Parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction

+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<		relational
<=	less than	relational
	less than or equal to	
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

3.2 Constants

Integer constants have the following syntax

```
<size>'s<base><value>
```

Where:

- `<size>` is optional. If given, it specifies the total number of bits represented by the literal value. If not given, the default size is 32 bits (the Verilog standard says "at least" 32 bits).
- `s` is optional. If given, it specifies that the literal integer should be treated as a 2's complemented signed value. If not given, the default is unsigned.
- `<base>` is required, and specifies whether the value is in binary, octal, decimal, or hex.
- `<value>` is required, and specifies the literal integer value.

Sign, size and base are optional and they can be separated by white spaces. If none are used, then a number is positive and a decimal. Legal base specifications are **d** and **D** for decimal numbers, **h** and **H** for hexadecimal numbers, **o** and **O** for octal numbers, b and B for binary numbers. Table 3 lists legal characters for constants. Any number can include underscores (_). The underscores improve readability and do not affect the value of the number. "**X**" or "**x**" is used for unknown values. "**Z**" or "**z**" or "**?**" is used for high impedance. Also, "**?**" is used for don't care (more later).

Table 3. Allowed digits

Base	Allowed Characters
16	0-9, a-f, A-F, X, x, Z, z, _, ?
10	0-9, _, ?
8	0-9, X, x, Z, z, _, ?
2	0, 1, X, x, Z, z, _, ?

If a specified number is bigger than a specified size constant, then the most significant bits of numbers will be truncated. If number is smaller than the size constant, then it will be padded to the left with zeros (zero extension). If the most significant bit of a specified number has an unknown (x) or high-impedance (z) value, then that value will be used to pad to the left. Figure 4 shows some examples.

```
15                  // decimal 15 as a 32-bit number
'h f                // same as above
'o 17               // same as above
'd 15               // same as above
'b 1111        // same as above
'b 1_1_1_1          // same as above

-5'b1_1011          // 00101
8'b0                // 00000000
8'b1                // 00000001
8'bz                // zzzzzzzz
8'bx                // xxxxxxxx
```

3.3 Vectors

So far, we have been using single wire in modeling to connect components together and for input and output ports. A single wire can carry only one bit. There are situations when you need to use set of wires to carry collection of bits (e.g., a number). In such scenario, it is more convenient to use a bus to represent the group of bits as one unit. Verilog provides the concept of Vectors. Vectors are used to represent multi-bit busses. A declared vector can be assigned to another one without explicit indexing. Also, bit-select/part-select operator (`[]`)

can be used to select a bit or a group of bits from a vector. The following are examples for using Vector:

```
        input [7:0] x;          // 8-bit input vector with MSB=7 and LSB=0
        wire [2:5] w1;          // 4-bit bus with MSB=5 and LSB=5
        assign w1 = x[6:3];     // w1 is connected to bits 3 to 6 of input
x
```

3.4 Reduction Operators

The reduction operators are: AND (**&**), NAND (**~&**), OR (**|**), NOR (**~|**), exclusive-OR (**^**), and exclusive-NOR (**^~** or **~^**). Reduction operators are unary operators; that is, they operate on a single vector and produce a single-bit result. Reduction operators perform their respective operations on a bit-by-bit basis from right to left. The following are examples for using the reduction operators.

```
        wire [3:0] X = 4'b1010
        wire y = &X;          //Equivalent to 1 & 0 & 1 & 0. y is
    assigned to 1'b0
        wire z = |X;          //Equivalent to 1 | 0 | 1 | 0. z is
    assigned to
        wire w = ^X           //Equivalent to 1 ^ 0 ^ 1 ^ 0. w is
    assigned to 1'b0
```

3.5 Conditional Operator

The conditional operator (**?:**) has three operands, as shown in the syntax below.

```
conditional_expression ? true_expression : false_expression
```

The conditional_expression is evaluated. If the result is true (1), then the true_expression is evaluated; if the result is false (0), then the false_expression is evaluated. The conditional operator can be used when one of two expressions is to be selected. Figure 5 shows how a 4x1 multiplexer can be modeled using the conditional operator.

```
module MUX4x1(I, Sel, Y);
    input [3:0] I;
    input Sel;
    output Y;

    assign Y = (Sel == 2'b00) ? I[0] :
               (Sel == 2'b01) ? I[1] :
               (Sel == 2'b10) ? I[2] : I[3];

endmodule
```

Figure 4. 4x1 Multiplexer Model using Conditional Operator

3.6 Concatenation and Replication

The Verilog concatenate operator is the open and close brackets {,}. It should be mentioned that these brackets can also be used to do replication in Verilog, but that is for another example. Concatenation can be used to combine two or more types together.

The replication operator, {n{}}, is used to replicate a group of bits n times. The number in front of the brackets is known as the repetition multiplier. So, for example, in {3{2'b01}} 3 is the repetition multiplier and 2'b01 is what will be replicated 3 times. It is important to note that the repetition multiplier must be a constant. The following example uses both replication and concatenation operators to extend 8-bit signed bus into 16-bit one (sign extension). The MSB of x is repeated 8 times then concatenated to x.

```
wire [7:0] x = -8'd5;
wire [0:15] y;
assign y = {{8{x[7]}}, x};
```

3.7 Procedural Assignment

While combinational logic can be described using assign statements, higher complexity combinational logic blocks are better described using always procedural blocks for several reasons:

- Powerful statements like if, if-else, case and looping constructs can only be used in always blocks; these statements are useful for implementing complex combinational blocks with greater clarity and in a more concise manner than is possible with assign statements.
- Multiple output nets can be assigned within a single always block.
- Sequential logic can only be specified within always blocks.

Procedural assignment allows an alternative higher-level behavioral description of combinational logic. It supports C-like control structures such as if, for, while and case. Procedural statements have to be contained in an always block as shown in Figure 5. The symbols in parentheses after the @ symbol is the sensitivity list, the statements inside the always block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. The statements inside the always block are evaluated in the order given in the code.

```
module DEC2x4(input [1:0] S, output reg [3:0] O);
        always @(S) begin
                O[0] = ~S[0] & ~S[1];
                O[1] = S[0] & ~S[1];
                O[2] = ~S[0] & S[1];
                O[3] = S[0] & S[1];
        end
endmodule
```

Figure 5. 2x4 Binary Decoder Model using Procedural Assignment

Coming up with an incomplete sensitivity list is one of the common modeling errors. This is solved in Verilog-2001 by allowing the usage of the wildcard "*****" for the sensitivity list. Hence, in Verilog-2001 you may use: **always @ ***. Please note that anything assigned in an always block must also be declared as type reg. A **reg** (register) is a data object that holds its value from one procedural assignment to the next. A **reg** is a Verilog variable type and does not necessarily imply a physical register.

4. Behavioral Modeling

Verilog provides designers the ability to describe design functionality in an algorithmically by describing the behavior of the circuit instead of its structure or function. Design at this level resembles C programming more than it resembles digital circuit design. Behavioral Verilog constructs are similar to C language constructs in many ways. Verilog is rich in behavioral constructs that provide the designer with a great amount of flexibility. Verilog contains a number of procedural statements such as if statement, case statement and loops. These statements can be used to model combinational behavior (covered by this section) or sequential behavior (covered by Chapter 4).

4.1 If Statement

Verilog if statement is very similar to that of the C language. Figure 6 shows an example of a very simple if statement that models the behavior of a 2x1 multiplexer. Please note that **begin** and end are used to mark the beginning and the end of a block of procedural statements (compound statement)

```
module beh_MUX2x1(input A, B, Sel, output reg Y);
always @ (*) begin
          if(Sel == 1'b1) Y = B;
          else Y = A;
end
endmodule
```

Figure 6. Behavioral Model of 2x1 Multiplexer

```
module beh_DEC2x4(input [1:0] S, output reg [3:0] O);
        always @(S) begin
                if(S==2'b00) O=4'b0001;
                else if(S==2'b01) O=4'b0010;
                else if(S==2'b10) O=4'b0100;
                else O=4'b1000;
        end
endmodule
```

Figure 7. Behavioral Model of 2x4 Binary Decoder

Multiple if statements can be "cascaded" to evaluate multiple Boolean conditions and establish priorities, as in the example shown by Figure 7. If the final else clause in a multiway if-else statement is missing, it will infer a latch. The logic will have to remember what the output was earlier, and hence a latch will

get synthesized. Thus, in modeling combinational logic *using an always block, all variables must be assigned under all conditions. Sometimes it is hard to spot* incomplete assignments, for example with large number of nested if statements. In this case, you can avoid latches by setting variables to default values at the start of the always block. Figure 8 shows example that illustrates this idea.

```
module beh_MUX2x1(input A, B, Sel, output reg Y);
always @ (*) begin
          Y = A;                    // default value
  if(Sel == 1'b1) Y = B;
end
endmodule
```

Figure 8. Behavioral Model of 2x1 Multiplexer w/o using else

4.2 case Statement

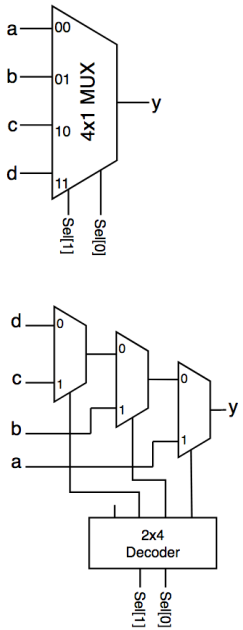
When the number of the nesting grows, it becomes difficult to understand the **if else** statement. The Verilog **case** statement, comes handy in such cases. A **case** statement is a multiway decision statement that compares the case expression with a number of item expressions. The execution jumps to the branch that matches the current value of the case expression. If there are multiple matched items, execution jumps to the branch of the first match. Figure 6, shows an example of case statement usage.

The last item can be an optional default keyword. The default clause in a case statement indicates that when all other cases are not met, then the flow can branch to the statements in the default clause. This gives the synthesis tool an option to pick a branch when no other condition is satisfied. If the default clause is missing, the logic will have to remember what the output was earlier, and hence a latch will get synthesized.

```
module case_MUX4x1(input [3:0] I, input Sel, output reg Y);

always @ *
   case(Sel)
          2'b00: Y = I[0];
          2'b01: Y = I[1];
          2'b10: Y = I[2];
          default: Y = I[3];
   endcase


endmodule
```

Figure 9. Behavioral Model of a 4x1 Multiplexer using case Statement

The **if-else** construct infers a priority routing network; while **case** statement treats all cases equally without giving one case a priority over the others. This is illustrated by the examples of Figure 10.




```
module case_MUX4x1( input a,b,c,d,
                    input Sel,
                    output reg y);

always @ *
        case(Sel)
                2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
        2'b11: y = d;
endcase

endmodule
```

```
module if_MUX4x1( input a,b,c,d,
                    input Sel,
                    output reg y);

always @ * begin
        if(Sel==2'b00) y=a;
else (Sel==2'b01) y=b;
else (Sel==2'b10) y=c;
else y=d;
end

endmodule
```

Figure 10. if-else vs. case

4.3 casez Statement

In addition to the regular case statements, Verilog provides two variations **casez** and **casex**. In this section we will discuss **casez** only as **casex** is not supported for synthesis. In normal **case** statement, the case expression needs to EXACTLY match, for one of the case statements to execute. There is no provision of Don't cares. **casez** solves this problem by allowing don't cares. If there is a z or ? in the case expression, then it means that the expression can match to 0, 1 or z. Figure 11 shows a priority encoder modeled using **casez** statement.

A priority encoder gets n lines and encodes them into $log_2(n)$ output lines. The output lines carry the number of the highest priority input line. An extra output, valid, is active if there is at least one active input line. For example, if the input is 0011, the output is 0 (0 is the highest priority line). If the input is 1100 then the output is 2. In both cases valid is active. If the input is 0000, the output is 0 and valid is 0. A common use of priority encoders is for interrupt controllers, where we select the most critical out of multiple interrupt requests.

```
module pri_encoder(
        input [3:0] I,
        output reg [1:0] O,
        output reg valid);

always @(*) begin
```

```
  casez (I)
    4'b???1: {valid, O} = {1'b1, 2'd0};
    4'b??1?: {valid, O} = {1'b1, 2'd1};
    4'b?1??: {valid, O} = {1'b1, 2'd2};
    4'b1???: {valid, O} = {1'b1, 2'd3};
    default: {valid, O} = {1'b0, 2'd0};
  endcase
end

endmodule
```

Figure 11. A Priority Encoder using casez Statement

Because **casez** case expressions may overlap because of the don't care terms, the synthesizer generates a priority logic. This makes **casez** similar to if-else statement.

4.3 Loop Statements

Loop statements provide a means of modeling blocks of procedural statements. There are four types of loop statements: **forever**, **repeat**, **while**, and **for** statements. In this section we will discuss the **for** loop as it is the commonly used looping construct for synthesis.

```
module DEC2x4(input [1:0] Sel, output reg [3:0] O);
integer k;
always @ (*) begin
for(k=0; k<4; k=k+1) begin
                        if((Sel == k))
                                O[k]=1'b1;
        else
                                O[k]=1'b0;
                end             // for
end             // always

endmodule
```

Figure 12. Binary Decoder Modeling using for Loop

The **for** statement (see example on Figure 8) has a syntax similar to that of C language. The loop iterates for 4 times (until the iteration condition, k<4, is false). For loops in synthesizable code are used to *expand replicated logic*. They are simply a way of shrinking the amount of code that is written by the hardware designer. In order for the code on Figure 8 to be synthesizable, the compiler must be able to unroll the loop. If you see this code and cannot calculate how many times the block inside the for must be instantiated, the compiler won't either.

5. Verilog Parameters

HDL code frequently uses constant values in expressions and array boundaries. These values are fixed within the module and cannot be modified. One good design practice is to replace the "hard literals" with symbolic constants. It makes code clear and helps future maintenance and revision. In Verilog, a constant can

be declared using the **localparam** (for "local parameter") keyword. For example, we can declare the width and range of a data bus as

```
localparam        DATA-WIDTH = 8 ,
                  DATA-RANGE = 2**DATA_WIDTH - 1;
```

or define a symbolic port name:

```
localparam        UART-PORT = 4'b0001,
                  LCD-PORT = 4'b0010,
                  MOUSE-PORT = 4'b0100;
```

The expression in the declaration, such as `2**DATA-WIDTH-1`, is evaluated during preprocessing and thus infers no physical circuit. Local parameters cannot be redefined nor redefined during the compile time.

Also, Verilog provides a run-time constant, which can be overridden or assigned a value during compile time. This run-time constant is called module parameter and is declared with the **parameter** keyword. The value of the constant can be changed during compilation of a design, and does not become fixed until simulation starts running. Using parameter constants can be very useful for making a module easily configured. Each instance of the module can have its parameter values redefined, making each instance unique. Figure 13, shows an example of a parameterized parallel adder.

```
module adder(a, b, ci, s, co);
     parameter n=8;          // n is a parameter with a default
value of 8
     input[n-1:0] a, b;
     output[n-1] s;
     output co;

     assign {s, co} = a+ b;

endmodule
```

Figure 13.

The following are 3 examples for instantiating the adder module:

```
adder #(.n(16)) a1(…..);          // 16-bit adder
adder #(4) a2(…..);                       // 4-bit adder
adder a3(…..);                            // 8-bit adder
```

6. Verilog Generate Block

A complex design modeled in Verilog may have several levels of hierarchy, where one module contains instances of other modules. Even small designs are often represented as a top-level module that connects together several lower level modules. One example might be building a larger 64-bit adder from several smaller 4-bit adders. At the structural level of modeling, sixteen instances of the 4-bit adder must be instantiated:

```
adder4 u0(a[3:0], b[3,0], ci[0], s[3:0], co[0]);
adder4 u1(a[7:4], b[7,4], ci[1], s[7:4], co[1]);
adder4 u2(a[11:8], b[11,8], ci[2], s[11:8], co[2]);
.
.
adder4 u15(a[63:60], b[63,60], ci[15], s[63:60], co[15]);
assign ci[0] = ci;
assign ci[1] = co[0];
.
.
assign ci[15] = co[14];
assign co = co[15];
```

In many cases this becomes very tedious as you write the same code segment multiple times which may cause coding errors and make the code unclear. In Verilog-95 you can overcome this by using the generate loop inside a generate block. A generate for loop is used to create one or more instances of items that can be placed within a Verilog module. The loop is essentially the same as a regular Verilog HDL for loop, but with these limitations:

- The index loop variable must be a **genvar**.
- Both assignments in the for loop control must assign to the same **genvar**.
- The contents of the loop must be within a named begin-end block.

The example having 8 instances of an 8-bit adder module shown on the previous page can easily be generated as follows:

```
generate
     genvar i;
     for (i-0; i<15; i-i+1)begin: a
          adder4   add   (sum[(i*8)+:8],   co[i+1],   a[(i*8)+:8],
     b[(i*8)+:8], ci[i]);
end
endgenerate
```

The generated instance names in the preceding example are:

```
a[0].add
```

```
a[1].add
.
.
a[15].add
```

The generate block can be used to generate procedural code as well as structural code and continuous assignments. The following example illustrates using a generate statement to create multiple always procedures. The example creates a sequential logic gray-code to binary converter, using a slightly different algorithm for the conversion.

```
parameter SIZE = 8;
genvar i;
generate
    for (i=O; i<=SIZE-l; i=i-l) begin: proc
        always @(posedge clock or negedge reset)
            if (reset _. 0)
                bin [i] .. l'bO;
            else
                bin[i] • Agray[SIZE-l:i);
        end
endgenerate
```

Verilog if-else and case statements can be used within a generate block to control what objects are generated. The following example examines the width of the input busses to determine what type of multiplier should be instantiated.

```
module multiplier (a, b, product);
parameter a_width = 8, b_width = 8;
localparam product_width a_width + b_width;
input [a_width-1:0l a;
input [b_width-1:0l b;
output [product_width-1:0l product;
generate
if «a_width < 8) I I (b_width < 8»
CLA_mult # (a_width, b_width) u1 (a, b, product);
else
WALLACE_mult # (a_width, b_width) u1 (a, b, product);
endgenerate
endmodule
```

A case statement can be used when there are several conditions to test to determine what should be generated. The case expression and case items can only use constant expressions or genvar variables. That is, the values which are tested must be known values at elaboration time. Regular variables and nets will not have values during elaboration, and therefore cannot be used in a generate block. The following

example illustrates using a generate case statement to control which objects are generated:

```
parameter WIDTH 1;
generate
     case (WIDTH)
          1: adder_1bit x1(co, sum, a, b, ci);
          2: adder_2bit x1(co, sum, a, b, ci);
          default: adder_cia # (WIDTH) x1(co, sum, a, b, ci);
     endcase
endgenerate
```

7. Verilog Model Verification

As designs are becoming complex, verification became the main bottleneck in the design process. Design teams spend 50-70% of their time in verifying designs rather than creating new ones. In previous sections we saw how to model different digital combinational functions in Verilog. Now it is time to learn how to verify the correctness of those Verilog models. This can be done by simulating the model along with a testbench. Verilog simulation environments provide tools for graphical or textual display of simulation results.

A testbench, as shown by Figure 13, is a Verilog module that represents a circuit involving the circuit to be tested; the model to be tested is usually called Design Under Test (DUT). The testbench has code to apply different inputs (stimuli) to the DUT. Also, it has code to monitor the outputs and display them to check their correction. To facilitate development of testbenches, some simulation environments (such as CloudV) provide testbench tools that automatically generate a template testbench. Such tools also provide ways of inserting templates for generation of test data for applying them to DUT. Using templates is helpful, but a designer must understand testbenches and language constructs that are used for testing a design module. In general, testbenches can be developed as:

- Manual: The output results must be viewed manually to determine if they are correct.
- Automatic: Outputs are evaluated by the testbench code and the final results are provided. Final results can be something like a pass/fail indicator on the screen or data written to an external file. This type of testbench is, also, called a self-checking testbench.

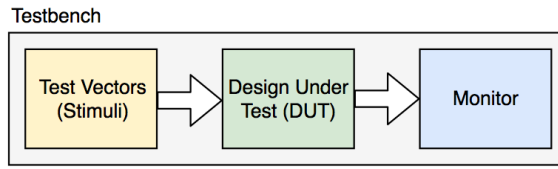


Figure 13. Testbench Structure

In this section, we will discuss the manual testbenches. Developing a manual testbench for a combinational circuit is straight forward, however selection of data and how much testing should be done depends on the DUT and its functionality.An example testbench is given by Figure 14. It is used to verify the functionality for the priority encoder given by Figure 14.

```
module tb;        // No ports!

  reg [3:0] stim_I;
  wire [1:0] mon_O;
  wire mon_valid;

  initial begin    // Stimuli
   stim_I = 4'b0000;
          #10 stim_I = 4'b0001;
          #10 stim_I = 4'b0010;
          #10 stim_I = 4'b1100;
          #10 stim_I = 4'b1101;
  end

  pri_encoder       DUT       (.I(stim_I),       .O(mon_O),
.valid(mon_valid));

endmodule
```

Figure 14. A Testbench for the Priority Encoder

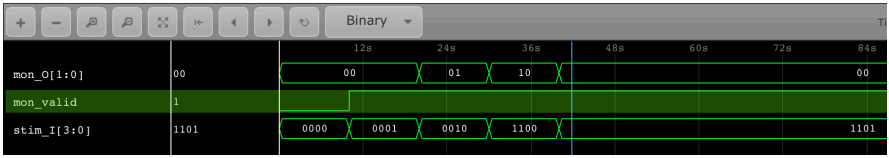


Figure 15. CloudV Waveform Viewer (FlexWave) displaying the generated VCD file

Variables corresponding to inputs and outputs of the module under test are declared in the testbench. Variables connecting to the inputs are declared as **reg** and outputs as **wire**. Application of data to the inputs are done in an **initial** block. The **initial** blocks are procedural blocks that are executed only once at the beginning of the simulation. 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**. In general, **initial** blocks are used for simulation. Support for **initial** blocks' synthesis into hardware is supported by some tools for FPGAs, and used mainly for power-up configuration. In the **initial** block we change the input (`stim_I`) over time; we start with `0000` then after 10 time units (#10 means a delay of 10 time units) we make it `0001`, after another 10 time units we make it `0010` and so on. The testbench does not have any code to monitor the output of the priority encoder. This is because the simulator dumps all the signals inside the DUT and the testbench into a file called the Value Dump

Change (VCD) file[1]. The VCD file can be displayed using a waveform viewer as shown by Figure 15.

5.1 Verilog System Functions and Tasks

Verilog provides standard system tasks and functions for certain routine operations. All system tasks/functions appear in the form `$<keyword>`. Operations such as displaying on the screen, monitoring values of nets, performing file i/o and stopping, and finishing simulation are done by system tasks/functions. A system function returns a value while a system task which does not.

`$monitor` is a system task that can be used to print values to the console. The values it prints are those corresponding to the arguments that you pass to the task when it is executed. The `$monitor` task is executed whenever any one of its arguments changes, with one or two notable exceptions. `$time` is a system function that returns the current simulation time. Figure 16, shows an example testbench that uses `$monitor` and `$time` to produce text output during simulation. In this example, the testbench produces a text output every time there is a change on one of the `$monitor` arguments. However, `$time` is an argument to `$monitor`, `$time` changing does not cause `$monitor` to execute. The output of simulating this testbench is given by Figure 17. As you can see the output is not that readable. To get a better output, we can use the formatting string option for the `$monitor` system task as shown by Figure 18. With formatting, the `$monitor` statement becomes:

```
        $monitor("t=%d, I=%b, O=%d, Valid=%b", $time, stim_I,
mon_O, mon_valid);
```

The format string is similar to that of C language **printf** STDIO function. Here are some of the format indicators that might be used with $monitor:

- `%b` or `%B` – Binary
- `%c` or `%C` – Character (low 8 bits)
- `%d` or `%D` – Decimal; %0d for minimum width field
- `%h` or `%H` – Hexadecimal
- `%o` or `%O` – Octal
- `%t` or `%T` – Simulation time, used with `$time`

Verilog, also, provides two tasks to switch monitoring on and off.: `$monitoron` and `$monitoroff`. Monitoring is turned on by default at the beginning of the simulation and can be controlled during the simulation with the `$monitoron` and `$monitoroff` tasks. Also, Verilog provides `$display` system

[1] CloudV adds extra code to the testbench to enable dumping the VCD and terminating the simulation

task which is similar to `$monitor` except that it prints the immediate messages only once to the console. Unlike `$monitor`, `$display` need to be added as many times as the value of the signal is to be displayed.

```
module tb;                          // No ports!

  reg [3:0] stim_I;
  wire [1:0] mon_O;
  wire mon_valid;

  initial begin          // Stimuli
        stim_I = 4'b0000;
                #10 stim_I = 4'b0001;
                #10 stim_I = 4'b0010;
                #10 stim_I = 4'b1100;
                #10 stim_I = 4'b1101;
  end

  pri_encoder DUT (.I(stim_I), .O(mon_O), .valid(mon_valid));

  initial                           // Monitoring
                          $monitor($time, , stim_I, mon_O, mon_valid);

endmodule
```

Figure 16. Updated Testbench for the Priority Encoder

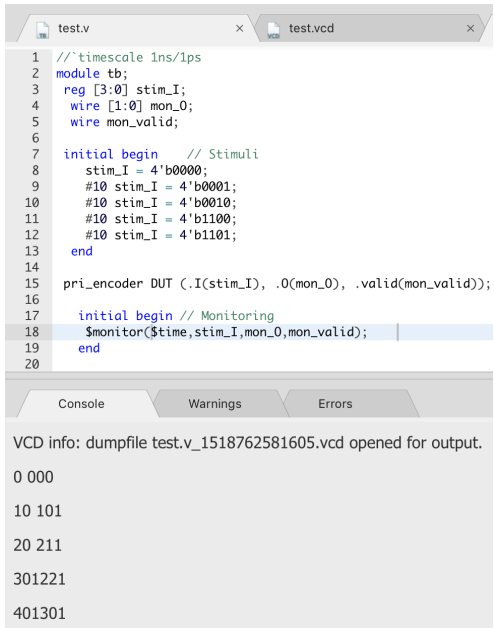


Figure 17. Simulation Output from simulating the Testbench of Figure 16.

test.v test.vcd

```
1  //`timescale 1ns/1ps
2  module tb;
3   reg [3:0] stim_I;
4    wire [1:0] mon_O;
5    wire mon_valid;
6
7   initial begin    // Stimuli
8      stim_I = 4'b0000;
9      #10 stim_I = 4'b0001;
10     #10 stim_I = 4'b0010;
11     #10 stim_I = 4'b1100;
12     #10 stim_I = 4'b1101;
13   end
14
15   pri_encoder DUT (.I(stim_I), .O(mon_O), .valid(mon_valid));
16
17     initial begin // Monitoring
18      $monitor("t=%d, I=%b, O=%d, Valid=%b", $time,stim_I,mon_O,mon_valid);
19     end
20
```

Console Warnings Errors

```
VCD info: dumpfile test.v_1518762442479.vcd opened for output.
t= 0, I=0000, O=0, Valid=0
t= 10, I=0001, O=0, Valid=1
t= 20, I=0010, O=1, Valid=1
t= 30, I=1100, O=2, Valid=1
t= 40, I=1101, O=0, Valid=1
```

Figure 18. Producing formatted output.

5.2 Time Scales

Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 s, and delay values in another module need to be defined by using a different time unit, e.g. 100 ns. Verilog HDL allows the reference time unit for modules to be specified with the **`timescale** compiler directive. In Verilog, compiler directives are instructions to the Verilog compiler. They begin with the grave accent "`" and do not end with a semi-colon. The effect of a compiler directive starts from the place where it appears in the source code, and continues through all files processed subsequently, to the point where the directive is superseded, or the end of the last file to be processed. **`timescale** compiler directive is usually placed at the beginning of the Verilog file and has the following format.

```
`timescale <reference_time_unit> / <time_precision>
```

- The `<reference_time_unit>` specifies the unit of measurement for times and delays.
- The `<time_precision>` specifies the precision to which the delays are rounded off during simulation.

The range for time unit can be from seconds to femto-seconds, which includes all the time units including s (second), ms (mili-second), us (micro-second), ns (nano-second), ps (pico-second) and fs (femto-second). Only 1, 10, and 100 are valid integers for specifying time unit and time precision. For example,

```
`timescale 1ns/1ps
#1;                          // 1ns delay
#1.003;                      // 1.003 delay
#1.0009;              // 1 ns delay since it is out of the precision
value.
```

7. Design Example

BCD to 7-Segment decoder using TinyFPGA and CloudV

8. Combinational Logic Modeling Guidelines

- Define your combinational logic using assign statements when practical. Only use always blocks to define combinational logic if constructs like if or case make your logic much clearer or more compact.
- When modeling combinational logic with an always block, if a signal is assigned in any branch of an if or case statement, it must be assigned in all branches.
- A signal may not be assigned by more than one always block
- Properly indent your code, as shown in the examples in this chapter
- Use comments liberally
- Use meaningful signal names (do not use foo , bar , …)
- Be consistent in your use of capitalization and underscores.

9. Summary

Chapter 6
Verilog for Sequential Logic

In Chapter 3, we considered combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit. Such circuits include storage elements that store the values of logic signals. The contents of the storage elements are said to represent the state of the circuit. Circuits that behave in this way are referred to as sequential circuits.

Digital sequential logic circuits are divided into synchronous and asynchronous types. In synchronous sequential circuits, the state of the device changes only at discrete times in response to a clock signal. In asynchronous circuits the state of the device can change at any time in response to changing inputs. Most sequential logic circuits, nowadays, are clocked or synchronous. In a synchronous circuit, an electronic oscillator (called a clock source) generates a sequence of repetitive pulses called the clock signal that is distributed to all the memory elements in the circuit. The basic memory element in sequential logic is the flip-flop. The output of each flip-flop only changes when triggered by the clock pulse, so changes to the logic signals throughout the circuit all begin at the same time, at regular intervals, synchronized by the clock.

1. Behavioral modeling of Sequential circuits

In synchronous sequential circuits, changes in flip-flops occur only in response to either a positive edge (also called raising edge) or a negative edge (also called falling edge) of the clock, but not both. Verilog HDL takes care of these conditions by providing two keywords: **posedge** and **negedge**. For example, the following Verilog code will initiate execution of the associated procedural statements only if the clock goes through a positive transition or if reset goes through a negative transition.

```
always @(posedge clock or negedge reset) begin
      // some statements
end
```

2. Reset

Reset is needed for forcing the circuit into a sane (well-defined) state. It initializes flip-flops, as circuits have no way to self-initialize. Reset is usually applied at power-up for real hardware (or at the beginning of time for simulation). A digital system usually has a common global reset line which is connected to every flip flop reset or preset input to initialize the flip flop during the power-on duration. This global reset is generated using a power-on reset (PoR) circuit which detects the power applied to the digital system and generates a reset impulse that goes to the entire circuit placing it into a known state. Figure 1 shows an example of such PoR circuit. When the power is turned on, the capacitor starts to charge and the voltage of node "a" goes up exponentially from 0v to 5v. The Schmitt trigger inverter cleans this signal and converts it into a perfect binary signal which starts at logic "1" when the power is turned on and stays at one for some time that has to do with the time constant, $(R1 + R2)C$. The values for R1, R2 and C must be selected to assert the global reset line long enough to reset the circuit. The button is used to generate a reset pulse when pressed by the operator.

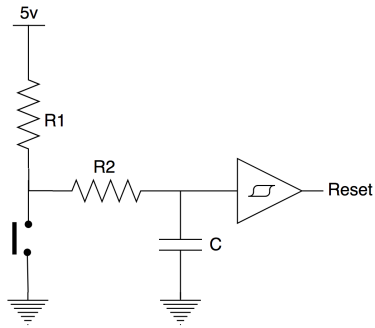


Figure 1. A Typical PoR Circuit

There are two types of Reset:

- Synchronous Reset: Reset is sampled only on the clock edge and is applied as any other input. To model asynchronous reset in Verilog use:

```
always @ (posedge clk) begin
        if (reset == 1'b1) begin
                // initialize flip flop here
        end
        else begin
                // normal operation
        end
    end
```

- Asynchronous Reset: Reset has priority over any other signal. Reset occurs with or without clock present and it is not synchronized with the clock. To model synchronous reset in Verilog use:

```
        always @(posedge clk or posedge reset) begin
```

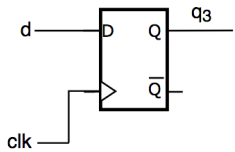
```
        if(reset) begin
                // initialize flip flops here
        end
        else begin
                // normal operation
        end
end
```

3. Blocking and non-blocking assignments

Procedural assignments discussed so far in are all of the blocking type. This means that while the assignment is taking place, the execution of the following statements are haled (or blocked). A different type of procedural assignment is a non-blocking assignment that uses <= instead of =. This type of assignment schedules its right hand side into the left-hand side **reg** and continues on to the next statement.

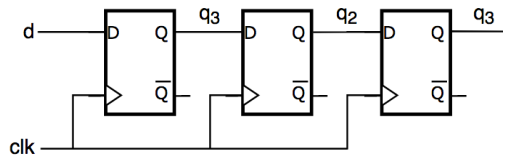
Blocking assignment statements are executed sequentially in the order they are listed in a block of statements, similar to C language assignment operator. We used the blocking assignment to model combinational logic in Chapter 3. Non-blocking assignments are executed concurrently by, first, evaluating the set of expressions on the right-hand side of the list of statements; they do not make assignments to their left-hand sides until all of the expressions are evaluated. Figure 2 shows the differences between blocking and non-blocking assignment. Usually we use blocking assignments to model combinational logic and non-blocking assignments for sequential logic. It is recommended not to mix blocking and non-blocking assignments within the same procedural block as different tools may interpret this mix differently. Hence, it is recommended not to mix sequential and combinational modeling within the same procedural block.

```
module pipeb1 (q3, d, clk);
output q3;
input d;
input clk;
reg q3, q2, q1;

always @(posedge clk) begin
q1 = d;
q2 = q1;
q3 = q2;
end
endmodule
```



(a) Blocking Assignment

```
module pipeb1 (q3, d, clk);
output q3;
input d;
input clk;
reg q3, q2, q1;

always @(posedge clk) begin
q1 <= d;
q2 <= q1;
```



```
q3 <= q2;
end
endmodule
```

(b) Non-Blocking Assignments

Figure 2. Blocking vs. non-Blocking Assignment

4. Modeling Finite State Machines

As discussed in Chapter 1, a finite state machine is a form of abstraction that models the behavior of a sequential circuit by showing each state it can be in and the transitions between each state. FSM is a very useful tool for designing sequential logic. A FSM is composed from a collection of flip flops. This collection is often referred to as a state register. State is represented by a unique value stored in this register. Connecting the outputs of the state register's flip flops to the inputs is a web of combinational logic, which incorporates the machine's inputs. So, at each triggering edge of the clock the combinational logic is evaluated and the state is physically changed. The output from the FSM can be either generated from the current state (data stored on state register) only (Moore Machine) or from the current state and the input (Mealy machine). The hardware implementation of Mealy and Moore machines is outlined in Figure 3.

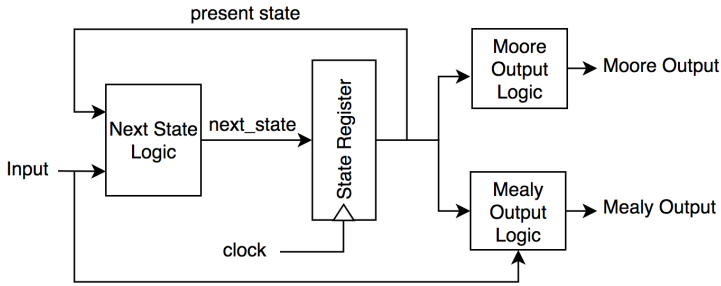


Figure 3. Mealy and Moore FSM Hardware

There are several ways to model FSM in Verilog. However, the model that reflects the structure outlined in Figure 1 is found to be the best. In addition to that, the following are desirable in the model:

- easily to modify to change state encodings.
- facilitate debugging.
- yield efficient synthesis results.

4.1 FSM Verilog Example: Rising Edge Detector

The rising-edge detector is a circuit that generates a short one-clock-cycle pulse when the input signal changes from 0 to 1. It is usually used to indicate the onset of a slow time-varying input signal. For example, generating one clock cycle pulse

when a switch is pressed. We will design the circuit as a Mealy machine. Figure 4 shows the state diagram of the Mealy machine and Figure 5 shows the Verilog model. Figure 6 shows the waveforms resulted from simulating the Verilog model.

When coding state machines in Verilog, there are a few general guidelines that are implanted in the given Verilog model:

- Use parameters to define state encodings. This makes it easier to change the encodings later.
- Avoid using Verilog macros (`**define**) for state encodings as they have global scope which may introduce bugs into your model.
- Use the right state encodings to achieve the design objectives: One-Hot-Encoding for high speed, binary encoding for the smallest area and Gray encoding for low power.
- Organize your code as 3 procedural always blocks. One synchronous for updating the state register and two asynchronous blocks to model the next state logic and the output logic.

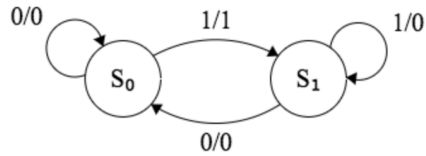


Figure 4. State Diagram for Rising Edge Detector (Mealy)

```
module re_detector(clk, rst, x, z);
    input clk, rst;
    input x;
    output reg z;

    reg state;          // present state register
    reg next_state;     // next state logic output

    // state encoding
    parameter   S0=1'b0,
                S1=1'b1;

    // next state logic
    always @ *
        case (state)
            S0: if(x==1'b1) next_state = S1;
                else next_state = S0;
            S1: if(x==1'b1) next_state = S1;
                else next_state = S0;
        endcase

    // updating the state register
    always @(posedge clk) begin
        if (rst)
            state <= S0;
        else
```

```
            state <= next_state;
    end

    // generating the output
    always @ *
        case (state)
            S0: if(x==1'b0) z = 0;
                else z = 1;
            S1: z = 0;
        endcase
endmodule
```

Figure 5. Rising Edge Detector Verilog Model

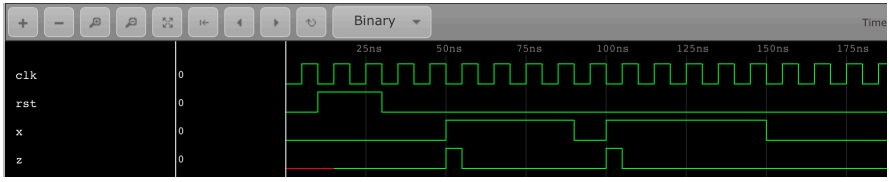


Figure 6. Waveform resulted from simulating the rising edge detector

4.2 FSM Verilog Example: Sequence Detector

The following state diagram represents a Moore FSM that detects (by setting the output z to 1) two successive ones (1's) on the input w.

Beef it up!

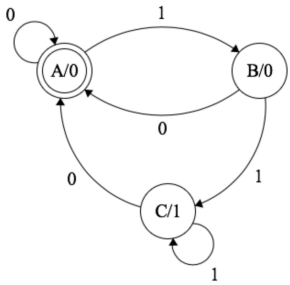


Figure 2. State Diagram for 2 Successive 1's Detector

The Verilog module that models this FSM is given below:

```
module fsm(clk, rst, w, z);
input clk, rst, w;
output z;

reg [1:0] state, nextState;
 // States Encoding
parameter [1:0] A=2'b00, B=2'b01, C=2'b10;

// Next state generation
always @ (w or state)
         case (state)
```

```
                A: if (w) nextState = B;
                    else   nextState = A;
                B:  if (w) nextState = C;
                    else   nextState = A;
                C:  if (w) nextState = C;
                    else   nextState = A;
            endcase

        // Update state FF's with the triggering edge of the clock
        always @ (posedge clk or negedge rst) begin
                if(!rst) state <= A;
                else state <= nextState;
        end

        // output generation
        assign z = (state == C);
        endmodule
```

5. Examples of Sequential Models

5.1 Binary Counter

The binary counter is a register which can be incremented (up counter) or decremented (down counter) with the triggering clock edges. The following Verilog module that implements an n-bit up/down counter with enable control (**en**). The input **ud** is used to enable up counting (1) or down counting (0). Also, the counter can be loaded with external data (**din**) when the input **ld** is 1. If both **en** and **ld** inputs are enabled, the counter is loaded with the external data (**din**). This means **ld** has a higher priority than **en**. The counter is parameterized with a default size of 4. This is possible using Verilog parameter keyword. Verilog parameters are constants typically used to specify the width of variables and time delays.

```
    module counter (clk, rst, din, ld, en, ud, q);
    parameter n = 4;
input clk, rst, en;
          output reg[n-1:0] q;

   always @ (posedge clk) begin
          if (rst == 1'b1) begin
             q <= 4'b0;                          // initialize flip
        flops here
          end
          else begin                                    //   normal
     operation
                if(ld) q <= din;
                else if(en)
                if(up) q <= q + 1'b1;
                else q <= q - 1'b1;
          end
       end

    endmodule
```

The following is a testbench for an 8-bit version of the above counter. This customization is possible by assigning 8 to the parameter n during the instantiation of the counter module by prefixing the instance name by #(8) or #(.n(8)).

```
module counter_tb;

        parameter n = 8;

        //Inputs
        reg clk, rst;
        reg en, ld, ud;
        reg [n-1: 0] din;

        //Outputs
        wire [n-1: 0] q;

        //Instantiation of Unit Under Test
        counter #(n)uut (
                .clk(clk),
                .rst(rst),
                .en(en),
                .ld(ld),
                .ud(ud),
                .din(din),
                .q(q)
        );

        initial begin
        //Inputs initialization
                clk = 0;
                rst = 0;
                en = 0;
                ld = 0;
                ud = 0;
                din = 0;

    // Wait for 10ns, assert the reset for 50ns then release it
         #10 rst = 1;
         #50 rst = 0;

   // Load the counter
   @(posedge clk);
   #1 din = 4;
   ld = 1;
   @(posedge clk);
   #1 ld = 0;

   // enable up counting
   ud = 1'b1;
   en = 1'b1;

   // down count
   #60 ud = 0;

   // disable the counter
   #40 en = 0;
end
```

```
        always #5 clk = ~clk;

endmodule
```

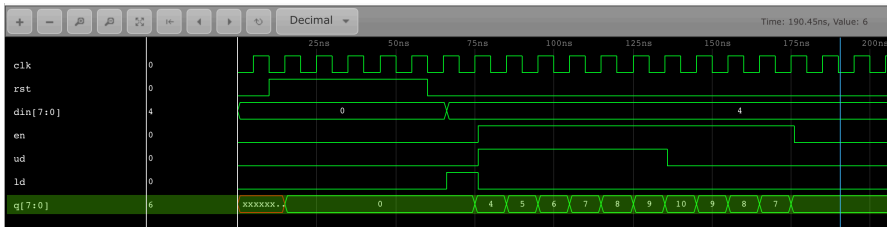


Figure 3. Waveforms of simulating the counter testbench

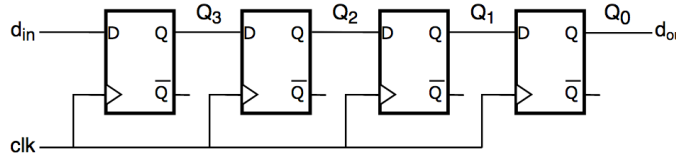


Figure 4. 4-bit shift right register

5.2 Shift Register

A shift register is a register than can shift its content to the left or to the right. A 4-bit shift right register is shown in Figure 4. The register value is represented by {Q3, Q2, Q1, Q0}. With every rising edge of the clock, Q0 gets the old value of Q1, Q1 gets the old value of Q2, Q2 gets the old value of Q3 and Q3 gets the data input d_{in} (serial data input). If Q0 is connected to d_{in} and the flip-flops are initialized to 1000, we call it a ring counter. The ring counter flip-flops values change with the clock as follows:

1000 ➔0100 ➔ 0010 ➔ 0001 ➔ 1000 ➔ 0100 ➔…

The following Verilog module implements a 4-bit Ring counter.

```
module ring_ctr_4 (clk, rst, Q);
      input clk, rst, en;
      output reg[3:0] Q;

     always @ (posedge clk) begin
          if (rst == 1'b1) begin              // initialize flip
     flop here
          Q <= 4'b1000;
          end
          else begin                          //          normal
     operation
                Q <= {Q[0], Q[2], Q[1], Q[0]};
          end
     end
endmodule
```

5.3 Clock Divider

Sometimes it is required to divide the clock frequency for different reasons. For an instance, divide the clock to generate 1KHz clock which is used to keep track of time in milliseconds. The simplest way for doing so, is to use a binary counter like that described in Chapter 1 (Figure 56). Every bit of that counter oscillates with a frequency $f_c/2^n$. So, a binary counter can be used to divide any clock by an integer which is a power of 2. How about dividing the clock by any integer? This is, also, easy and can be achieved using a simple Moore machine that implements modulo-n counter to divide the clock by n. Figures 5 and 6 show the state transition diagram for the machine as well as its Verilog implementation for a divide-by-6 circuit.

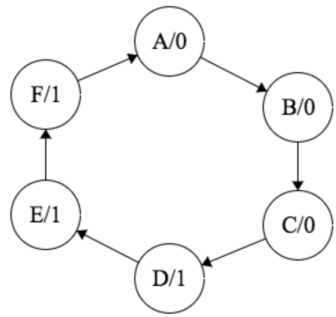


Figure 5. State Diagram of Divide-By-6 Clock Divider

```
module divide_by_6(clki, rst, clko);
    input clki, rst;
    output clko;
    reg [2:0] state, next_state;

    parameter [2:0] A=0, B=1, C=2, D=3, E=4, F=5;

    always @(posedge clki) begin
        if (rst)
            state <= A;
        else
            state <= next_state;
    end

    always @ (*)
        case (state)
            A: next_state = B;
            B: next_state = C;
            C: next_state = D;
            D: next_state = E;
            E: next_state = F;
            F: next_state = A;
            default: next_state = A;
        endcase

    assign clko = (state==A || state==B || state==C) ? 0 : 1;

endmodule
```

Figure 6. Verilog Implementation of Divide-By-6 Clock Divider

Using the modulo-n counter as a clock frequency divider generates a 50% duty cycle clock (symmetrical clock) if the divisor (n) is an even integer as the generated clock will be high for half the counts then low for the other half. If it is not then the duty cycle may be less or more than 50%.

6. Modeling Memory

In digital simulation, one often needs to model register files, RAMs, and ROMs. Memories are modeled in Verilog simply as a one-dimensional array of registers. Each element of the array is known as an element or word and is addressed by a single array index. Each word can be one or more bits. It is important to differentiate between n 1-bit registers and one n-bit register. A particular word in memory is obtained by using the address as a memory array subscript.

```
reg mem1bit[0:1023];              // 1K 1-bit words
reg [7:0] membyte[0:1023]; // 1K 8-bit words(bytes)
```

Add an example of a sample ram and refer to the $readmem family of system tasks. Also, show how the memory is mapped into FPGA BRAM resources

9. Sequential Logic Modeling Guidelines

- Use blocking assignments (“=”) in always blocks that are meant to represent combinational logic.
- Use nonblocking assignments (“<=”) in always blocks that are meant to represent sequential logic.
- Do not mix blocking and nonblocking assignments in the same always block.

10. Summary

Chapter 7
Binary Arithmetic Circuits

Some introd….

1. Adders

As discussed in Chapter 1 (Section 9.4), the building block of binary adders is the 1-bit full adder (FA). The FA is designed to help realizing the RCA that implements everyday algorithm for adding binary numbers. Figure 1 shows the FA circuit. Figure 2 shows the FA Verilog model. Figure 3 shows the an n-bit RCA and Figure 4 shows its Verilog model

```
module FA(input a, b, ci, output
s, co);
                wire p, g;
assign p = a ^ b;
assign g = a & b;
                assign s = p ^ ci;
                assign co = g | (p
& ci);
endmodule
```

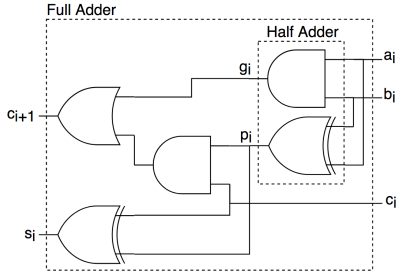


Figure 1. 1-bit Full Adder

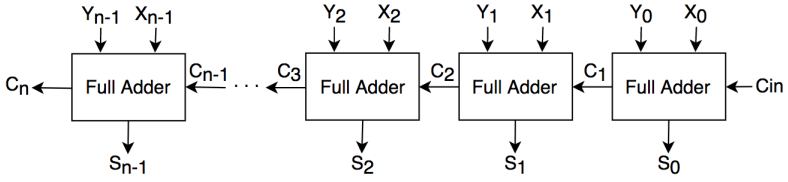


Figure 2. The Ripple Carry Adder (RCA)

```
module RCA(X, Y, Ci, S, Co);
parameter n = 8;
input [n-1:0] X, Y;
input Ci;
output [n-1:0] S;
output Co;
```

```
wire [n-1:1] C;

FA u [n-1:0] (   .a(X), .b(Y), .s(S),
                 .ci({C[n-1:1], ci}),
              .co({co, C[n-1:1]}) );

endmodule
```

Figure 3. A Parametrized Verilog Model of n-bit RCA

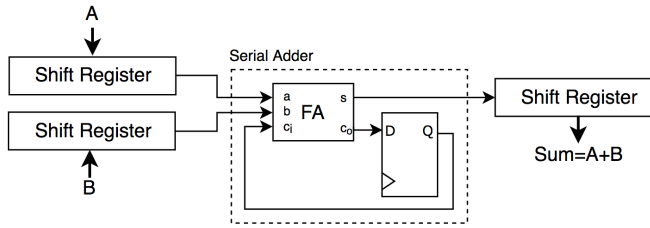


Figure 4. A typical usage of the serial adder

```
module sAdder(input clk, rst, input a, b, output s);
reg y;
     wire Y;

     FA u (.a(a), .b(b), .s(s), .ci(y), .co(Y));

     always@(posedge clk)
     if(rst) y<=0;
     else y<=Y;
endmodule
```

Figure 6. Serial Adder Verilog Model

2. Serial Adders

The addition algorithm implemented by the RCA can be implemented using a sequential circuit that adds the operands serially, one pair of bits, one bit from the first operand and one bit from the second operand, every clock cycle. This way, 2 n-bit operands are added in n clock cycles. This sequential circuit is called a serial adder. The operands added by a serial adder are usually stored in shift registers and the output (sum) is shifted into a third register as shown in Figure 5. The serial adder circuit consists of 1-bit full adder and a flip flop to store the carry out of the current addition and make it available while adding the following pair of bits.

3. Subtraction

Subtraction can be performed using the method of complement as discussed in Chapter 1 (Section 3.4).

$$A - B = A + 2's complement\ of\ B$$

$$A - B = A + \bar{B} + 1$$

The above equation suggests that the subtractor can be implemented using a parallel adder by inverting the 2nd operand and connect the carry in to 1 as shown in Figure 7.

The design of Figure 7 can be updated to performs addition and subtraction operations on the input bits. Figure 8 shows an n-bit adder–subtractor circuit. It has a mode control signal M which determines if the circuit is to operate as an adder or a subtractor. The inputs to the XOR gates is from input M and one of the bits of input B (Bi). When M is 0 then the output of XOR gates will be same as the second input (Bi). On the other hand, when one input of XOR gate is 1 then the output of XOR will be complement of the second input ($\overline{B_i}$). When the mode control is at logic LOW, then the output of the XOR gates will be B and the full adders receive the value of B, and the input carry C0 is 0, the circuit performs A plus B. When the mode control is at logic HIGH, then the output of XOR gate will be $\overline{B}$ and the full adder input carry (C_0) is 1, the circuit performs A plus 1's complement of B plus 1, which is equal to A minus B. Figure 9 shows the Verilog model of this adder-subtractor.

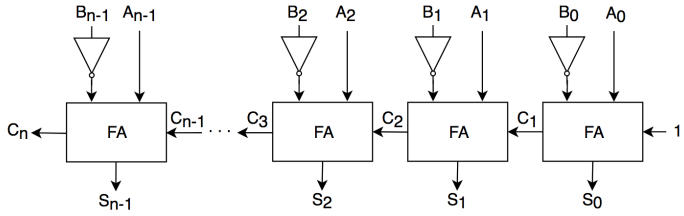


Figure 7.

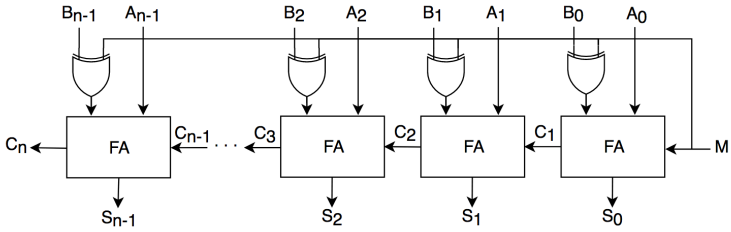


Figure 8.

```
module add_sub(A, B, M, R, Co);
parameter n = 4;
input[n-1:0] A, B;
input M;
output[n-1:0] R;

wire op2 = M ? ~B : B;

RCA #(.n(4)) u(.X(A), .Y(op2), .Ci(M), .Co(Co), .S(R));

endmodule
```

Figure 9.

4. Faster Parallel Adders

In the ripple-carry adder, the carries in different bit positions are generated sequentially. That is, c_{i+1} is dependent on c_i, and c_i cannot be determined unless c_{i-1} is known. We introduce in this section an adder which can generate all the carries in parallel. No carry propagation is in the cause of delay.

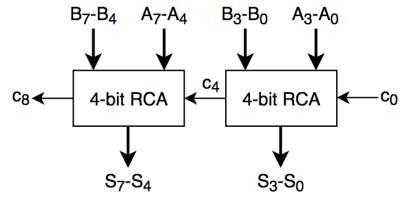


Figure 10. 8-bit RCA out of 2 4-bit RCA

4.1 Carry select adder (CSeA)

Consider an 8 bit RCA adder constructed from 2 4-bit RCA adders as shown in Figure 10. The delay of the 8-bit RCA is twice the delay of 4-bit RCA as the 4-bit RCA stage that adds the upper 4 bits of the operands waits for the carry out (c_4) from the first stage that adds the lower 4 bits of the operands. As the value of c_4 is either 0 or 1, let's disconnect the 2 adders and connect the carry in of the 2nd adder to logic 0. Also, let's add another 4-bit RCA to add the upper 4 bits but with the carry in set to logic 1. Now, all the adders can start calculating their outputs starting at time 0. But, as we have 2 versions of the upper 4 bits of the sum, we need to select one of them based on the value of c_4. Same goes for c_8, its value depends on c_4. Selection can be easily implemented using 2x1 multiplexors. This adder structure, as shown in Figure 121 is called Carry-Select-Adder (CSeA). The delay of this 8-bit CSeA is the delay of 4-bit RCA plus the delay of a 2x1 multiplexor. As the delay of the 2x1 multiplexor is the same (or less) that of 1-bit FA, the delay of the 8-bit CSeA is approximately 5/8 that of the 8-bit RCA but at the cost of adding extra hardware. To construct a 16-bit CSeA out of 4-bit RCA adders, we need to add 2 more stages similar to the 2nd stage of the 8-bit CSeA. That makes the total delay of the 16-bit CSeA:

$$Delay_{4-bit\ RCA} + 3 \times Delay_{2x1\ mux}$$

That makes the delay of the 16-bit CSeA, approximately, 7/16 that of the 16-bit RCA. Figure 12 shows the Verilog model of the 16-bit CSeA.

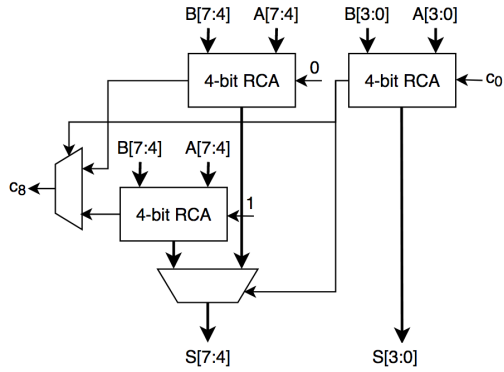


Figure 11. 8-bit CSeA using 4-bit RCA

```
module CSeA8_4(A, B, CI, M, S, Co);

wire C0, C1_0, C1_1;
wire[3:0] sum1_0, sum1_1;

RCA #(.n(4)) u0(       .X(A[3:0]), .Y(B[3:0]),
.Ci(Ci), .Co(C0), .S(S[3:0]) );

RCA #(.n(4)) u1_0(.X(A[7:4]), .Y(B[7:4]),
.Ci(1'b0), .Co(C1_0), .S(sum1_0) );
RCA #(.n(4)) u1_1(.X(A[7:4]), .Y(B[7:4]),
.Ci(1'b1), .Co(C1_1), .S(sum1_1) );

               assign S[7:4] = C0 ? sum1_1 : sum1_0;
               assign Co = C0 ? C1_1 : C1_0;

endmodule
```

Figure 12. Verilog model for 8-bit CSeA using 4-bit RCA

We may construct the 16-bit CSeA using RCA stages with different widths to get a smaller delay. If we use the sizes 2, 2, 3, 4, and 5 for the stages, the total delay will be

$$Delay_{2-bit\ RCA} + 4 \times Delay_{2x1\ mux}$$

which makes it, approximately, 6/16 (or 3/8) that of the 16-bit RCA.

4.2 Carry look-ahead adder (CLA)

The main idea behind carry look-ahead addition is an attempt to generate all incoming carries in parallel using 3-level logic. The carry out, C_{i+1}, produced at the i^{th} stage is given as follows:

$$c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$$

Now let

$$g_i = a_i \cdot b_i$$

$$p_i = a_i \oplus b_i$$

Then

$$c_{i+1} = g_i + p_i \cdot c_i$$

Where gi is a carry generate which produces the carry when both Ai, Bi are one regardless of the input carry. pi is a carry propagate and it is associate with the propagation of carry from Ci to Ci +1. Hence:

$c_1 = g_0 + p_0 \cdot c_0$
$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$
$c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$
$c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$

From the above four expressions, it is clear that all carries don't depend on previous carries and can be calculated starting from time 0. Every carry is generated using 3-level logic and the complexity of the circuit that generate the carry for the i^{th} stage increases as i increases. The above 4 expressions can be used to realize a 4-bit CLA unit to generate the carries. The outputs from such unit can be used to generate the sum as shown in Figure 10. Hence the time needed to generate the sum is the delay of 4 gates. This delay is constant and does not depend on the CLA size. The disadvantage of CLA is that the carry logic block gets very complicated for more than 4-bits. For that reason, CLAs are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4-bits. For example 2 4-bit CLA's can be chained to construct 8-bit adder.

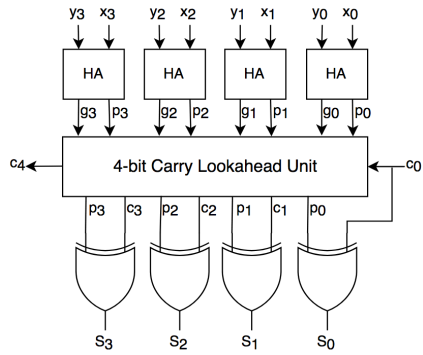


Figure 13. 4-bit CLA Adder

5. Adding Signed Numbers

As discussed in Chapter 1, there are 3 ways for representing a signed number in binary: signed magnitude, 1's complement and 2's complement. In all of them the most significant bit is used as a signed bit (0: positive, 1: negative). The 2's complement is the one usually used in digital systems. One reason for that is that the adders hardware discussed earlier are used as they are for signed numbers.

In the two's complement arithmetic if adding two positive (or negative) numbers results in a sum that has a sign which is different than the operands, an overflow has occurred. This means that the result requires more room than is given to it. Adding 2 numbers of different signs will not result in an overflow condition. For example, the following additions cause overflow condition and the results are not valid.

0	1	0		
	0	1	1	+ve
	0	1	0	+ve
	1	0	1	-ve

1	0	0		
	1	1	1	-ve
	1	0	0	-ve
	0	1	1	+ve

The overflow condition can be detected be examining the carries. If the last 2 carries (C_n and C_{n-1}) are different then there is overflow; otherwise, there is no overflow. In hardware this can be achieved using an XOR gate.

$$Overflow = C_{n-1} \oplus C_n$$

In many cases it is not possible to access Cn-1 in hardware, In this case Cn-1 can be calculated using the sum and the operands last bit.

$$Overflow = (A_{n-1} \oplus B_{n-1} \oplus S_{n-1}) \oplus C_n$$

Figure 14 shows a data flow model for an adder/subtractor with overflow detection.

```
module add_sub_ov(A, B, Ci, M, S, Co, OV);
parameter n;
input[n-1:0] A, B;
input Ci;
input M;
output[n-1:0] S;
output Co, OV;

wire[n-1:0] op2;

assign op2 = M ? ~B : B;
assign {Co, S} = A + op2 + M;
assign OV = S[n-1] ^ A[n-1] ^ B[n-1] ^ Co;

endmodule
```

Figure 14. Adder-Subtractor with Overflow Detection Verilog Model

6. Comparing Binary Numbers

To compare two binary numbers we can subtract the two numbers and check whether the result is zero (equal) or not (not equal). This can be checked in hardware by NORing every bit of the difference. Usually we call the output of this NOR gate the zero (Z) flag (see Figure 12). As the last bit of the difference is the sign, we call this bit the sign (S) flag. Also, the output XOR gate that checks for the overflow is called the overflow (OV) flag. Comparing numbers, either signed or unsigned, can be performed using the flags according to Table 1.

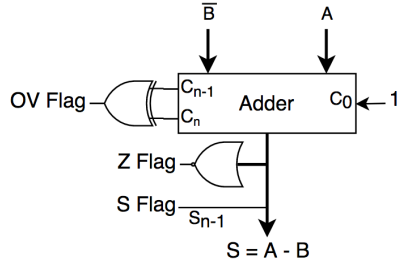


Figure 15. Subtraction and Flags

Table 1. Using Flags to Compare Numbers

Comparison	Flags Testes
Equal	Z=1
Not Equal	Z=0
Negative (<0)	S=1
Positive or Zero (>=0)	S=0
Overflow	OV=1
Unsigned >=	C=1
Unsigned <	C=0
Unsigned >	C=1 & Z=0
Unsigned <=	C=0 or Z=1
Signed >=	S=OV
Signed <	S!=OV
Signed >	Z=0 & S=OV
Signed <=	Z=1 or S!=V

```
module sgte(input[7:0] a, b, output cmp);

        wire[7:0] sum;

        wire ov, co;

        assign {co,sum} = A + ~B + 1'b1;

        assign ov = sum[7] ^ a[7] ^ b[7];
```

```
    assign cmp = (ov == sum[7]);          // ov == s

endmodule
```

Figure 16. 8-bit Signed Greater than or Equal Verilog Model

7. Multiplication

In the pencil-and-paper unsigned multiplication, the multiplicand is multiplied by every bit of the multiplier to generate partial products. As the multiplier bit is either 0 or 1. The partial products are either the multiplicand or 0. Every partial product is shift number of bits equivalent to the position of the multiplier bit. The shifted products are, then, added to find the product. This is illustrated in Figure 17. The partial product can be generated by AND gates, since anything ANDed with 1 will be itself, and that ANDed with 0 will be zero. This method is called shift and add multiplication. The shift and add multiplication works only with unsigned values and it does not generate the correct product if applied, as it is, to signed 2's complement values (the most significant bit is used for the sign). There are many methods to multiply 2's complement numbers. The easiest is to simply find the magnitude of the two numbers, multiply these together, and then use the original sign bits to determine the sign of the product. If the numbers had the same sign, the result must be positive, if the they had different signs, the result is negative. Multiplication by zero is a special case (the result is always zero, with no sign bit).

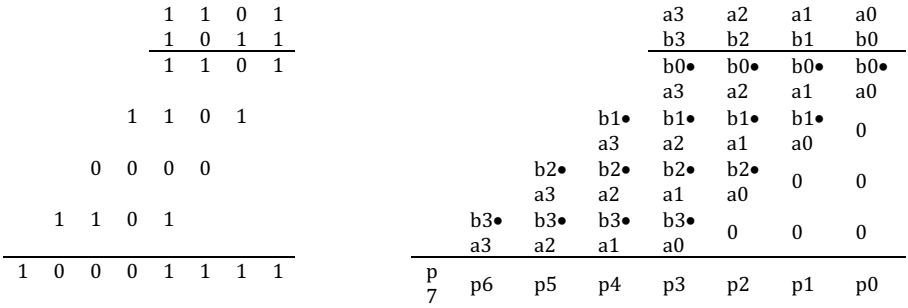


Figure 17. Paper and Pencil Binary Multiplication

7.1 Unsigned Array Multiplier

A combinational circuit out of AND gates and FA's can be constructed to realize such method as illustrated in Figure 18. The multiplier in Figure 18 is called 4x4 array multiplier. The 4x4 array multiplier is constructed out of 16 FA's and 16 AND gates and the longest path delay is the delay of 8 FA's plus the delay of 1 AND gate. To generalize, an $n \times m$ (n: multiplicand size; m: multiplier size) array multiplier needs $n \times (m-1)$ FA's and $n \times m$ AND gates. The delay of $n \times m$ array multiplier is the delay of $n + 2 \times (m-2)$ FA's plus the delay of a single AND gate.

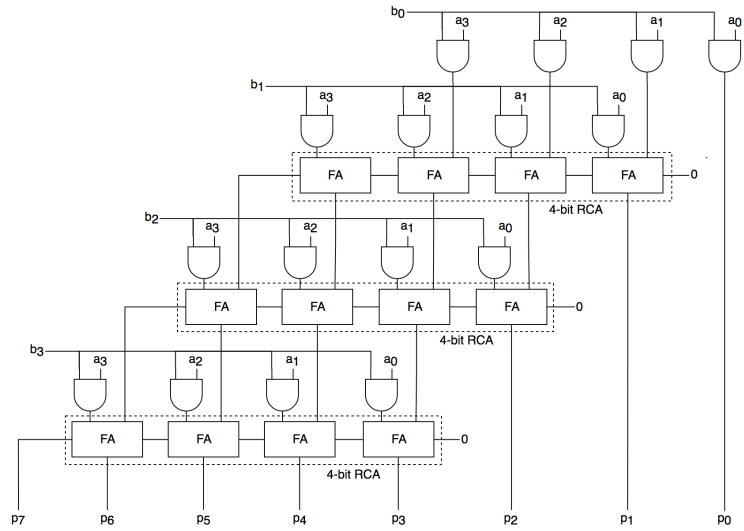


Figure 18. The Array Multiplier

```
module array_mul_4x4  (a, b, p);
    input[3:0] a;
    input[3:0] b;
    output[7:0] p;

    reg [3:0] pp[3:0];

    wire[3:0] s[3:1];
    wire[4:2] c;

    // generate the partial products
    integer i;
    always @ *
        for(i=0; i< 4; i=i+1)
            pp[i] = a & {4{b[i]}};

    RCA #(.n(4)) uu0 (    .X({1'b0,pp[0][3:1]}), .Y(pp[1]),
                       .Ci(1'b0), .S(s[1]), .Co(c[2]) );
    RCA #(.n(4)) uu1 (    .X({c[2],s[1][3:1]}), .Y(pp[2]),
                       .Ci(1'b0), .S(s[2]), .Co(c[3]) );
    RCA #(.n(4)) uu2 (    .X({c[3],s[2][3:1]}), .Y(pp[3]),
                       .Ci(1'b0), .S(s[3]), .Co(c[4]) );

    assign p = {c[4], s[3], s[2][0], s[1][0], pp[0][0]};

endmodule
```

Figure 18. The Array Multiplier Verilog Model

7.2 Signed Array Multiplier

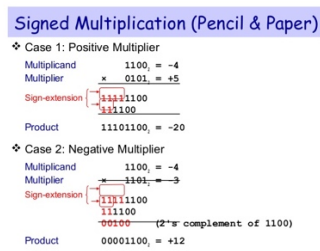
With a two's complement multiplier (m bits) and multiplicand (n bits), the high-order bit of each has negative weight. So when adding together the partial products, we'll need to sign-extend each of the n-bit partial products to the full

n+m-bit width of the addition. This will ensure that a negative partial product is properly treated when doing the addition. And, of course, since the high-order bit of the multiplier has a negative weight, we'd subtract instead of add the last partial product. This is illustrated in Figure 18. Describe the examples for 2's complement multiplications

It is obvious that the adder hardware constructed after this algorithm does not have a regular structure because of the last subtraction. However, instead of subtracting the last partial product, we can add its 2's complement. Baugh and Wooley devised an algorithm around this idea to construct a 2's complement multiplier with a regular structure. The Baugh-Wooley multiplier is shown in Figure 19.

```
                          X3   X2   X1   X0
                      *   Y3   Y2   Y1   Y0
                      ---------------------
   X3Y0 X3Y0 X3Y0 X3Y0 X3Y0 X2Y0 X1Y0 X0Y0
 + X3Y1 X3Y1 X3Y1 X3Y1 X2Y1 X1Y1 X0Y1
 + X3Y2 X3Y2 X3Y2 X2Y2 X1Y2 X0Y2
 - X3Y3 X3Y3 X2Y3 X1Y3 X0Y3
 ----------------------------------------
    Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```

Figure 18. 2's complement multiplication



2's Complement Multiplication Examples

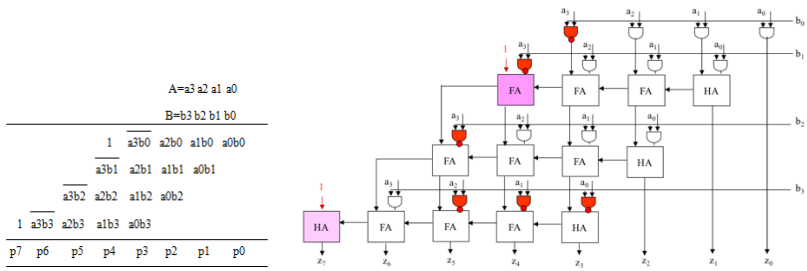


Figure 19. Baugh-Wooley Multiplier

7.3 Carry Save Adder (CSA) Multiplier

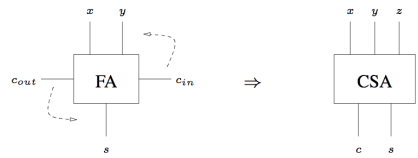
There are many cases where it is desired to add more than two numbers together. The straightforward way of adding together m numbers (all n bits wide) is to add

the first two, then add that sum to the next, and so on. This requires a total of m-1 additions.

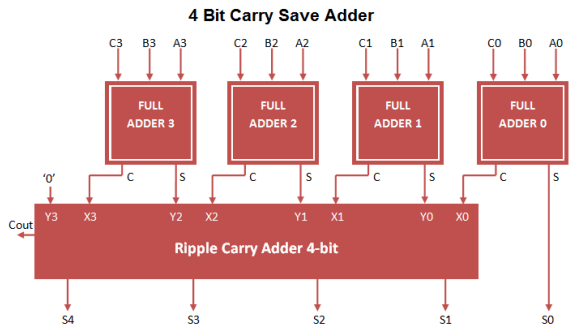
To add three numbers by hand, we typically align the three operands, and then proceed column by column in the same fashion that we perform addition with two numbers. The three digits in a row are added, and any overflow goes into the next column. Observe that when there is some non-zero carry, we are really adding four digits (the digits of x, y and z, plus the carry).

The carry save approach breaks this process down into two steps. The first is to compute the sum ignoring any carries. Then, separately, we can compute the carry on a column by column basis. Finally, we add the sum to the carries.

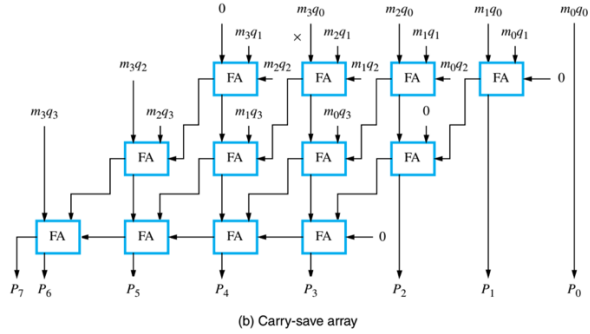
x:			1	0	0	1	1
y:			1	1	0	0	1
z:	+		0	1	0	1	1
s:			0	0	0	0	1
c:	+	1	1	0	1	1	
sum:		1	1	0	1	1	1



To add 3 numbers we construct a CSA to generate the sum and the carries then we use a RCA to add them up as shown in Figure ww.



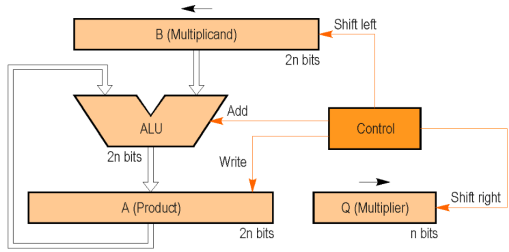
The idea of carry save adder can be applied to the partial products additions needed for the array multiplier to reduce the delay. Figure zz shows that for unsigned 4x4 multiplier. The delay of this multiplier is the delay of 6 FA's + the delay of one AND gate. This is less than the delay of the array multiplier, in Figure 18, built using RCA's (the delay is the delay of 8 FA's + delay of 1 AND gate).



(b) Carry-save array

Figure 9.16 Ripple-carry and carry-save arrays for a 4 × 4 multiplier.

In general the Delay = (m-2)+n FA's plus the delay of an AND gate



7.4 Sequential Multiplier

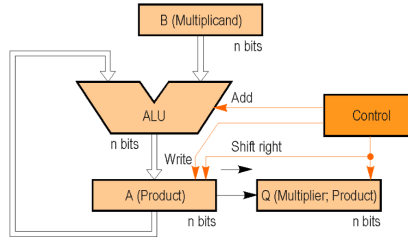
The paper and pencil multiplication can be implemented as it is using a simple sequential circuit. Figure xyz shows the implementation of such circuit. It consists of 2 shift registers, 2n-bit shift left register (B) to hold the n-bit multiplicand and m-bit shift right register (Q) to hold the m-bit multiplier. Also, there is n+m-bit register (A) to hold the product and is initialized to zero. The addition is done in steps. Each step takes one clock cycle. In the first clock cycle, the content of B (multiplicand) is added to P if the first bit of Q (multiplier) is not zero. In other words, the product gets the first partial product which is resulted from multiplying the multiplicand by the first bit of the multiplier. Then we shift B to the left and we shift Q to the right. We repeat these steps for m clock cycles (number of multiplier bits) to generate all the partial products, one every clock cycle, then add each partial product to A only if it is not zero (the first bit of Q is not 0). Loading and shifting the registers is controlled using a control signal generated by a FSM we call the control unit. The following pseudo code illustrates the multiplication algorithm done using this circuit.

```
Wait for go to be 1
Set done to 0
Load Multiplicand into B
```

```
Load Multiplier into Q
Clear A
Repeat m times
        If(Q[0]) load A+ B into A
        Shift B to the left
        Shift Q to the right
Set done to 1
```

Step	A	Q	B	Operation
0	0000 0000	1100	0000 1001	Initialization
1	0000 0000	1100	0001 0010	Shift left B
	0000 0000	0110	0001 0010	Shift right Q
2	0000 0000	0110	0010 0100	Shift left B
	0000 0000	0011	0010 0100	Shift right Q
3	0010 0100	0011	0010 0100	Add B to A
	0010 0100	0011	0100 1000	Shift left B
	0010 0100	0001	0100 1000	Shift right Q
4	0110 1100	0001	0100 1000	Add B to A
	0110 1100	0001	1001 0000	Shift left B
	0110 1100	0000	1001 0000	Shift right Q

Table 2 shows the operation of the multiplier while multiplying 1001 x 1100.



We can reduce the size of register A by exploiting the fact that register Q most significant bits are becoming useless as it is being shifted to the right. Shifting Q one bit to the right make the most significant bit of Q useless, shifting it twice makes the 2 most significant bits useless and so on. Hence, we can reduce the size of A to n bits only and connect it to the Q register. This way both A and Q will be used to store the n+m product.

Moreover, The original algorithm shifts the multiplicand left with zeros inserted in the new positions, so the least significant bits of the product cannot change after they are formed. Instead of shifting the multiplicand left, we can shift the product to the right. Therefore the multiplicand is fixed relative to the product, and since we are adding only n bits, the adder needs to be only n bits wide. Only the left half of the 2n-bit product register is changed during the addition. Figure 3.13 shows the new version of the circuit.

Step	A	Q	B	Operation
0	0000	1100	1001	Initialization
1	0000	0110	1001	Shift right A_Q
2	0000	0011	1001	Shift right A_Q
3	1001	0011	1001	Add B to A
	0100	1001	1001	Shift right A_Q
4	1101	1001	1001	Add B to A
	0110	1100	1001	Shift right A_Q

8. Summary