UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE MATEMÁTICA ASTRONOMÍA, FÍSICA Y COMPUTACIÓN.

ARQUITECTURA DE COMPUTADORAS

TEÓRICOS: PABLO A. FERREYRA

PRÁCTICOS:
DELFINA VELEZ
AGUSTÍN LAPROVITA
GONZALO VODANOVICK

CÁTEDRA DE ARQUITECTURA DE COMPUTADORAS

- Aprenderemos en detalle el funcionamiento interno y el incremento de performance de computadoras y un uso intenso de lógica programable (FPGAs, HDLS)…

ACTIVIDADES Y CRONOGRAMA

- I . DOS PARCIALES Y DOS RECUPERATORIOS
-
- II. LABORATORIOS
- III . 2 PROYECTOS FINALES

EL CRONOGRAMA CON LA DESCRIPCIÓN DE TAREAS SE SUBIRÁ A MOODLE, SEMANALMENTE.

SE PROMOCIONA CON DOS PARCIALES APROBADOS, PROMEDIO SUPERIOR A SIETE NOTA NO MENOR A SEIS. LOS LABORATORIOS APROBADOS Y LOS PROYECTOS FINALES APROBADOS.

SE APRUEBA CON 4 (CUATRO) PROYECTOS PODRÍAN EVENTUALMENTE SUMAR NOTA A LOS PARCIALES, (0,1,2).

PROGRAMA

"2015 – Año del Bicentenario del Congreso de los Pueblos Libres"

UNC Universidad Nacional de Córdoba

FAMAF Facultad de Matemática, Astronomía y Física

EXP-UNC: 49517/2015

Resolución CD N° 361/2015

PROGRAMA DE ASIGNATURA

ASIGNATURA: Arquitectura de Computadoras	**AÑO**: 2015
CARÁCTER: Obligatoria	
CARRERA: Licenciatura en Ciencias de la Computación	
RÉGIMEN: Cuatrimestral	**CARGA HORARIA:** 120 hs.
UBICACIÓN en la CARRERA: 3er Año – 2do Cuatrimestre	

PROGRAMA

FUNDAMENTACIÓN Y OBJETIVOS

Que el alumno sea capaz de interpretar el funcionamiento de los bloques "internos" asociados a Arquitectura de Computadoras No Convencionales (No “Von Neumann”, Procesadores de Alta Prestación y Computadoras Reconfigurables).

PROGRAMA

CONTENIDO

-Unidad 1: Computación SISD ("Single Instruction, Single Data")

1.1.-Arquitecturas tipo SISD, subtipo RISC (Reduced Instruction Set Computer).
1.2.-Arquitecturas tipo SISD, subtipo CISC (Complex Instruction Set Computer).
1.3.-Ejemplos de Arquitecturas SISD tipo No-Von Neuman.
1.4.- Concepto de Segmentación Encausada, (Pipe-Line).
1.5.- Ejemplos de Arquitecturas SISD, RISC.
1.6.- Ejemplos de Arquitecturas SISD, CISC.
1.7.- Arquitectura y Set de Instrucciones de Procesadores con Pipe-line.
1.8.- Ejemplos de Procesadores con Pipe-Line.

PROGRAMA

1.9.- Organización, Jerarquía y Administración de Memorias en Sistemas con Pipe-Line.
1.10.- Arquitectura y Set de Instrucciones de Procesadores Vectoriales.
1.11.- Características de los lenguajes para procesamiento Vectorial.
1.12.- Características de los Compiladores para Procesadores Vectoriales.
1.13.- Ejemplos de Procesadores con Segmentación Encausada y Vectoriales.
1.14.- Caso de estudio práctico para integración de conceptos.

PROGRAMA

-Unidad 2: Computación SIMD ("Single Instruction, Multiple Data")

2.1.-Procesadores Matriciales o SIMD. Concepto.
2.2.-Arquitecturas de los Procesadores SISD.
2.3.-Redes de Interconexión.
2.4.- Algoritmos para procesadores SIMD. Ejemplos.
2.5.- Procesadores SIMD Asociativos.
2.6.- Memorias Asociativas.
2.7.- Algoritmos para procesadores SIMD, Asociativos.
2.8.- GPGPU (General Purpose Graphic Processor Unit) y Computación heterogénea: Origen y Conceptos.
2.9.- Implementación: OpenCL (Open Computer Language)
2.10.- Arquitectura y Modelo de Plataforma.
2.11.- Modelo de Ejecución y de Memoria.
2.12.- Lenguaje e Interfaces.
2.13.- Operaciones con Matrices en OpenCL
2.14.- Caso de estudio Práctico para integración de conceptos.

PROGRAMA

-Unidad 3: Computación MIMD (Multiple Instruction, Multiple Data) y de Flujo de Datos

3.1.-Computadores MIMD ligeramente acoplados.
3.2.-Computadores MIMD estrechamente acoplados.
3.3.-Distintos tipos de Buses y Redes de Interconexión.
3.4.-Estructuración y Organización de la Memoria.
3.5.-La Problemática de los Sistemas Multicaché.
3.6.-Características de los S.O. para Sistemas MIMD.
3.7.-Algoritmos en Procesadores SIMD.
3.8.- Ejemplos de sistemas MIMD.
3.9.- Nociones de computadores de Flujo de Datos.
3.10.- Arquitecturas de computadoras de Flujo de Datos.
3.11.- Casos de estudio de ejemplo.

PROGRAMA

-Unidad 4: Nociones de Computación Reconfigurable (C. R.) y de Alta Performance (HPC)

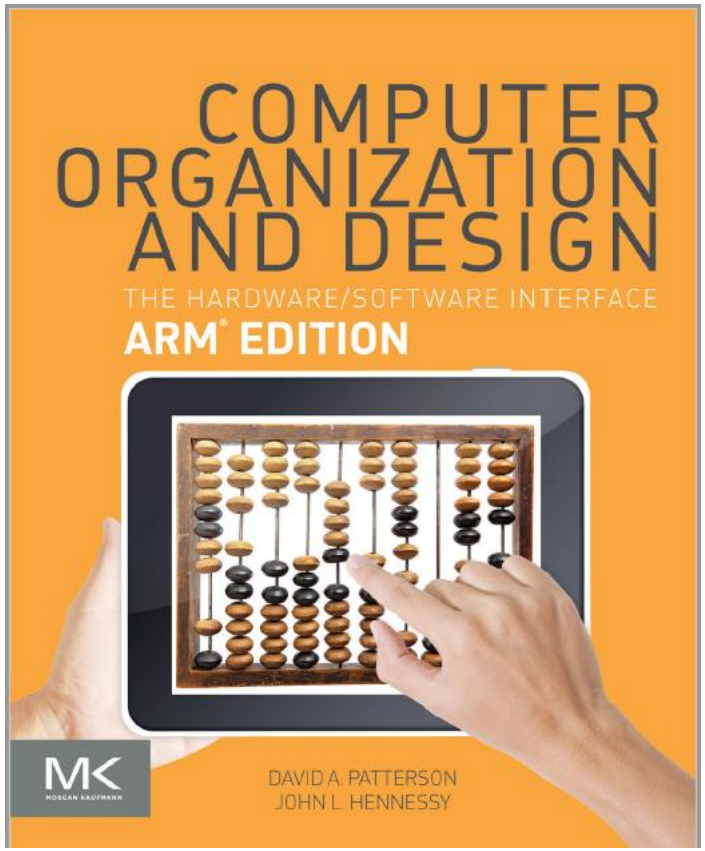
4.1.-Conceptos generales, historia y estado del arte de la C. R.
4.2.-El uso de HDL en computación reconfigurable.
4.3.-Nociones de Codiseño Hardware-Software y su aplicación en C.R.
4.4.- Conceptos generales de HPC.
4.5.- Historia y estado del arte HPC.
4.6.- Nociones de HPC y distribuida. Nociones básicas de Clusters.
4.7.- Nociones básicas de Arquitecturas Grid.

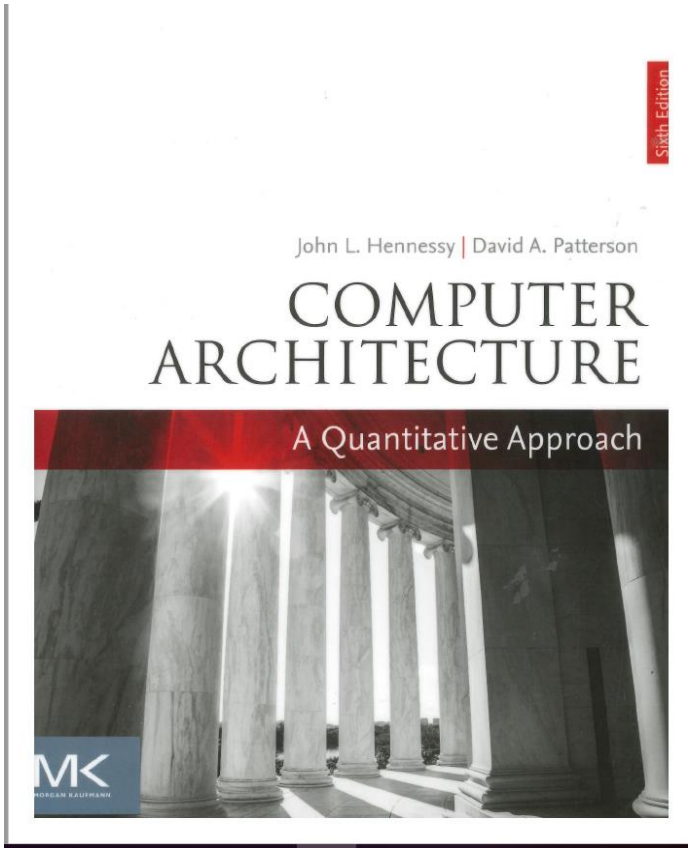
PROGRAMA

BIBLIOGRAFÍA

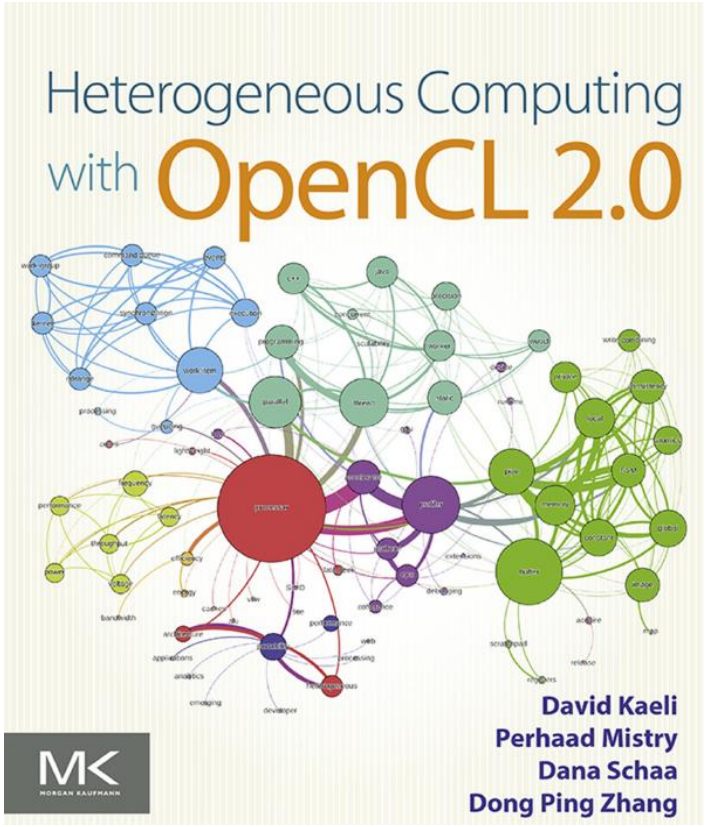
- Arquitectura de Computadoras y Procesamiento Paralelo. Kai Hwang y Faye A. Briggs. Mc Graw-Hill (1988).
- David A. Patterson and John L Hennessy: "Computer Organization and Design – The Hardware/Software Interface". Fourth Edition. Elsevier – Morgan Kaufmann (ISBN 978-0-12-374493-7).
- John L Hennessy and David A. Patterson: "Computer Architectura – A cuantitative Approach". Fourth Edition. Elsevier – Morgan Kaufmann.
- Volnei. Pedroni. Circuit Design Using VHDL. MIT Press, Cambridge, Massachusetts, 2004.
- Douglas Perry. VHDL: Programming by Example. Mc. Graw Hill, NY, 2002.
- Enoch Hwang. Microprocessor Design: Principles and Practices with VHDL. Brooks/Coole. 2004.

BIBLIOGRAFÍA





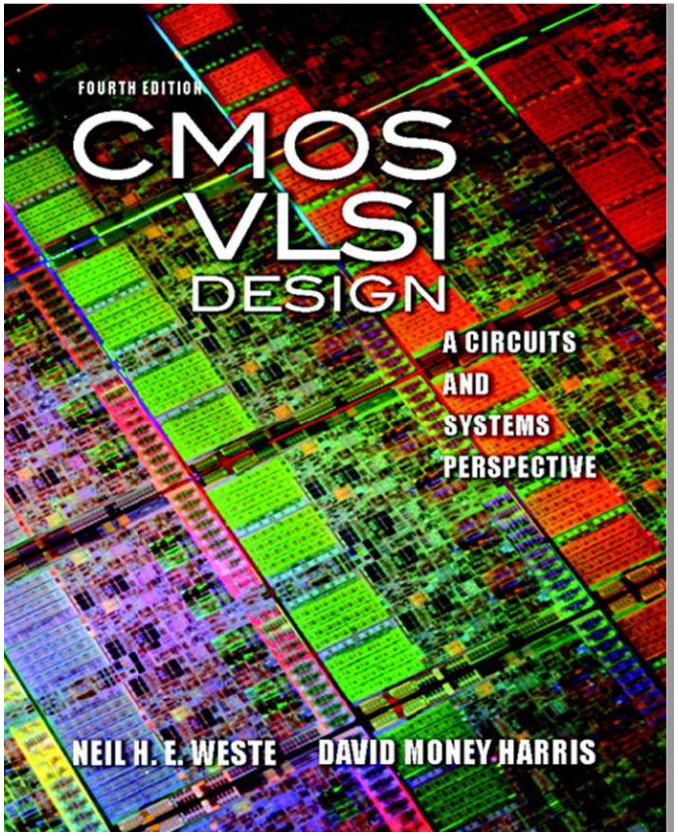
BIBLIOGRAFÍA



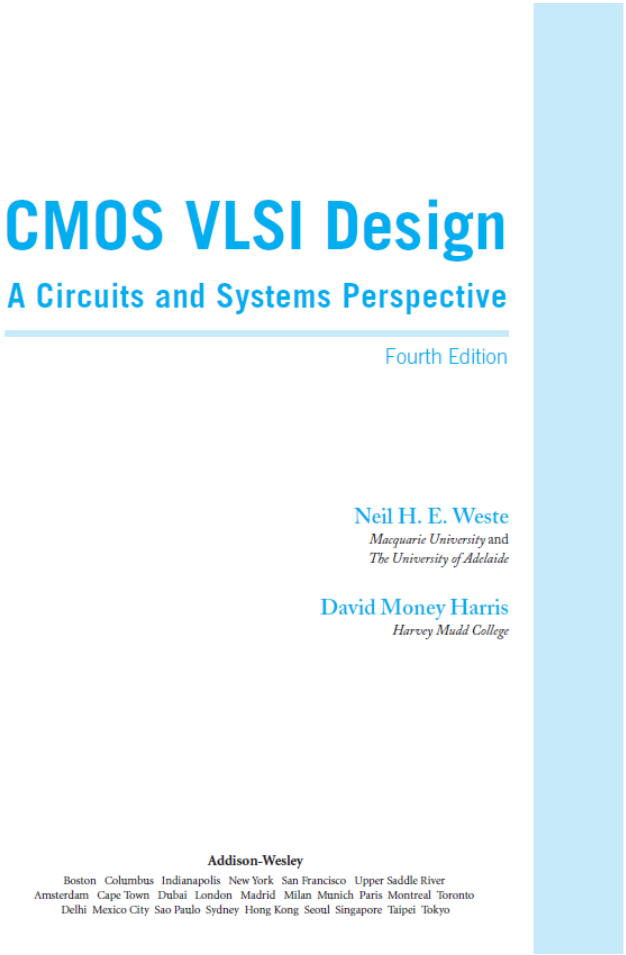
T1: HARDWARE DESCRIPTION LANGUAGES

- HARDWARE DESCRIPTION LANGUAGES
- **VHDL Y SYSTEM VERILOG**
- **UN ESTUDIO COMPARATIVO**

T1: HARDWARE DESCRIPTION LANGUAGES



T1: HARDWARE DESCRIPTION LANGUAGES



T1: HARDWARE DESCRIPTION LANGUAGES

APPENDIX A

Hardware Description Languages

A.1 Introduction

This appendix gives a quick introduction to the SystemVerilog and VHDL Hardware Description Languages (HDLs). Many books treat HDLs as programming languages, but HDLs are better understood as a shorthand for describing digital hardware. It is best to begin your design process by planning, on paper or in your mind, the hardware you want. (For example, the MIPS processor consists of an FSM controller and a datapath built from registers, adders, multiplexers, etc.) Then, write the HDL code that implies that hardware to a synthesis tool. A common error among beginners is to write a program without thinking about the hardware that is implied. If you don't know what hardware you are implying, you are almost certain to get something that you don't want. Sometimes, this means extra latches appearing in your circuit in places you didn't expect. Other times, it means that the circuit is much slower than required or it takes far more gates than it would if it were more carefully described.

The treatment in this appendix is unusual in that both SystemVerilog and VHDL are covered together. Discussion of the languages is divided into two columns for literal side-by-side comparison with SystemVerilog on the left and VHDL on the right. When you read the appendix for the first time, focus on one language or the other. Once you know one, you'll quickly master the other if you need it. Religious wars have raged over which HDL is superior. According to a large 2007 user survey [Cooley07], 73% of respondents primarily used Verilog/SystemVerilog and 20% primarily used VHDL, but 41% needed to use both on their project because of legacy code, intellectual property blocks, or because Verilog is better suited to netlists. Thus, many designers need to be bilingual and most CAD tools handle both.

In our experience, the best way to learn an HDL is by example. HDLs have specific ways of describing various classes of logic; these ways are called *idioms*. This appendix will teach you how to write the proper HDL idiom for each type of block and put the blocks together to produce a working system. We focus on a *synthesizable subset* of HDL sufficient to describe any hardware function. When you need to describe a particular kind of hardware, look for a similar example and adapt it to your purpose. The languages contain many other capabilities that are mostly beneficial for writing test fixtures and that are beyond the scope of this book. We do not attempt to define all the syntax of the HDLs rigorously because that is deathly boring and because it tends to encourage thinking of HDLs as programming languages, not shorthand for hardware. Be careful when experimenting with other features in code that is intended to be synthesized. There are many ways to write HDL code whose behavior in simulation and synthesis differ, resulting in improper chip operation or the need to fix bugs after synthesis is complete. The subset of the language covered here has been carefully selected to minimize such discrepancies.

T1: HARDWARE DESCRIPTION LANGUAGES (HDLS)

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language*. In turn, VHSIC is an acronym for the *Very High Speed Integrated Circuits* project. VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The IEEE standardized VHDL in 1987 and updated the standard several times since [IEEE1076-08]. The language was first envisioned for documentation, but quickly was adopted for simulation and synthesis.

VHDL is heavily used by U.S. military contractors and European companies. By some quirk of fate, it also has a majority of university users.

[Pedroni10] offers comprehensive coverage of the language.

T1: HARDWARE DESCRIPTION LANGUAGES (HDLS)

Verilog and SystemVerilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard in 1995 and was updated in 2001 [IEEE1364-01]. In 2005, it was updated again with minor clarifications; more importantly, SystemVerilog [IEEE 1800-2009] was introduced, which streamlines many of the annoyances of Verilog and adds high-level programming language features that have proven useful in verification. This appendix uses some of SystemVerilog's features.

There are many texts on Verilog, but the IEEE standard itself is readable as well as authoritative.

T1: HARDWARE DESCRIPTION LANGUAGES (MODULES)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= ((not a) and (not b) and (not c)) or
       (a and (not b) and (not c)) or
       (a and (not b) and c);
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (MODULES)

VHDL code has three parts: the `library` use clause, the `entity` declaration, and the `architecture` body. The `library` use clause is required and will be discussed in Section A.7. The `entity` declaration lists the module's inputs and outputs. The `architecture` body defines what the module does.

VHDL signals such as inputs and outputs must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of '0' or '1,' as well as floating and undefined values that will be described in Section A.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations, so Boolean equations should be parenthesized.

T1: HARDWARE DESCRIPTION LANGUAGES (MODULES)

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES

A module begins with a listing of the inputs and outputs. The `assign` statement describes combinational logic. ~ indicates NOT, `&` indicates AND, and | indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values that will be discussed in Section A.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on nets with multiple drivers, as will be explained in Section A.7.

T1: HARDWARE DESCRIPTION LANGUAGES (MODULES)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity adder is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of adder is
begin
  y <= a + b;
end;
```

Observe that the inputs and outputs are 32-bit vectors. They must be declared as `STD_LOGIC_VECTOR`.

T1: HARDWARE DESCRIPTION LANGUAGES (MODULES)

SystemVerilog

```
module adder(input  logic [31:0] a,
             input  logic [31:0] b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

Note that the inputs and outputs are 32-bit busses.

T1: HARDWARE DESCRIPTION LANGUAGES SIMULATION)

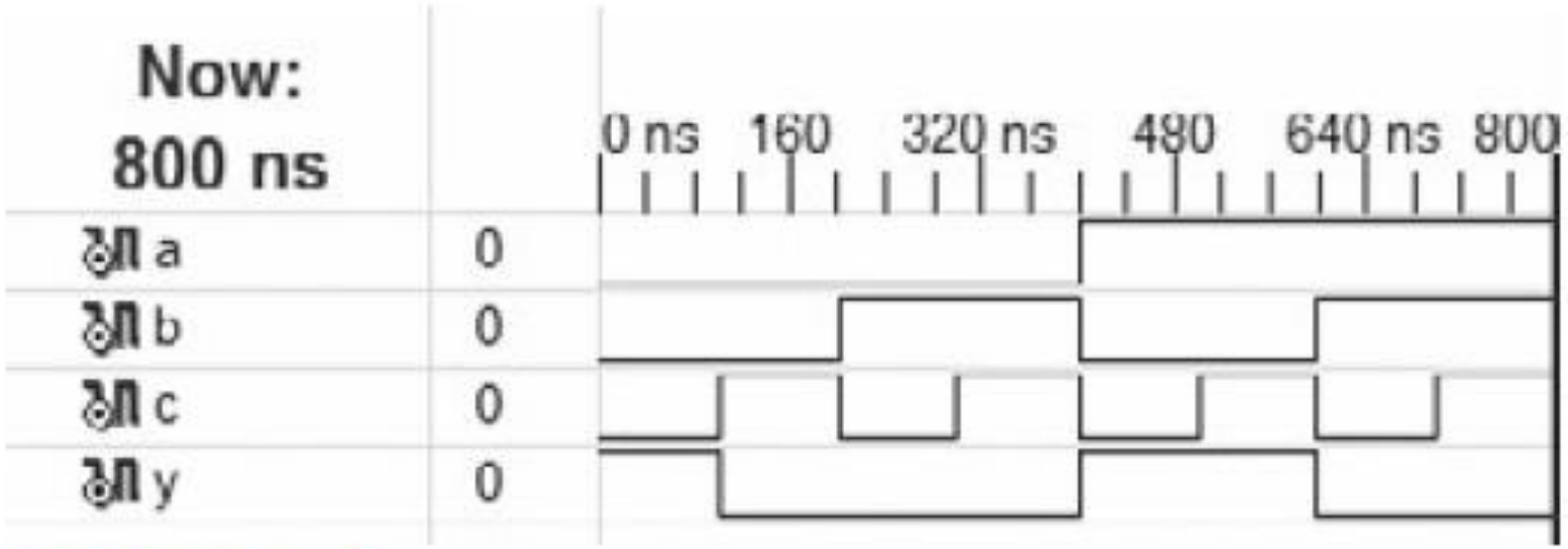


FIGURE A.1 Simulation waveforms

T1: HARDWARE DESCRIPTION LANGUAGES (SYNTHESIS)

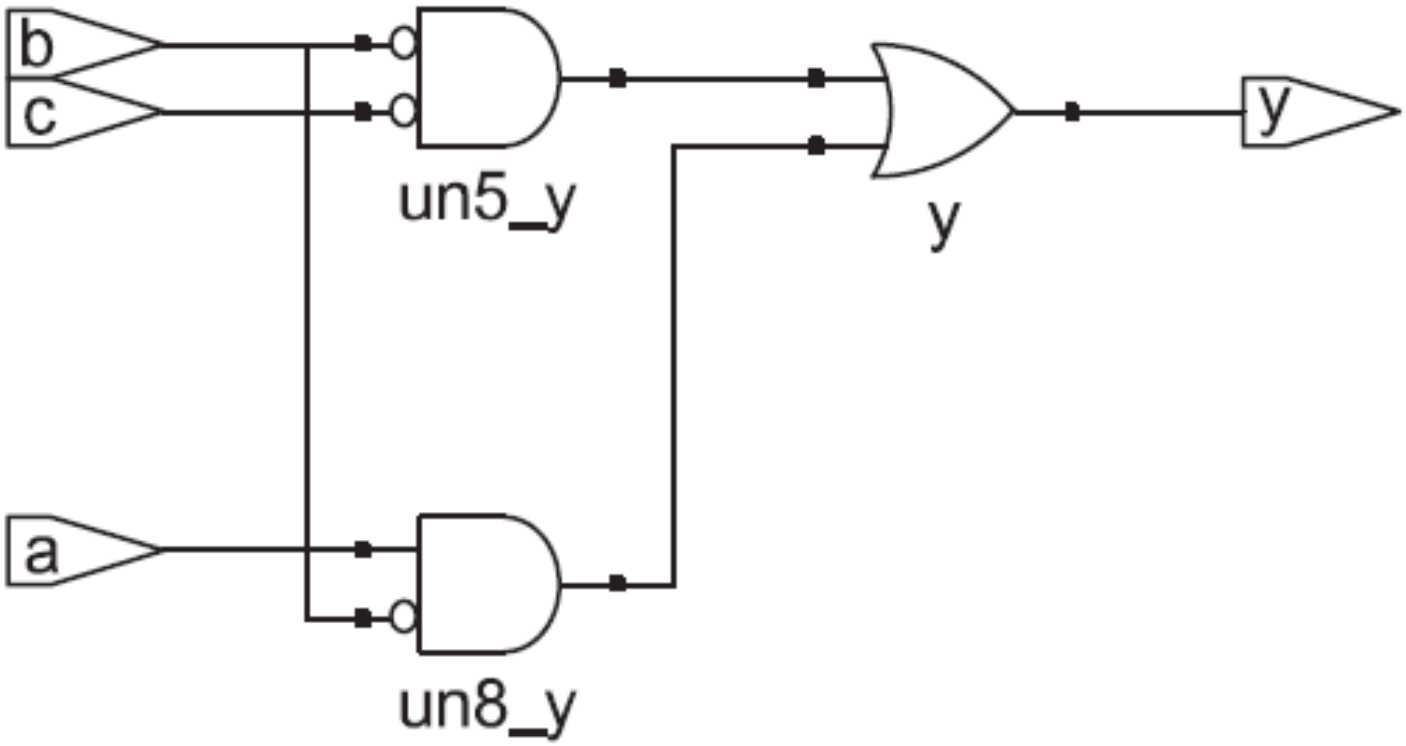


FIGURE A.2 Synthesized silly_function circuit

T1: HARDWARE DESCRIPTION LANGUAGES (SYNTHESIS)

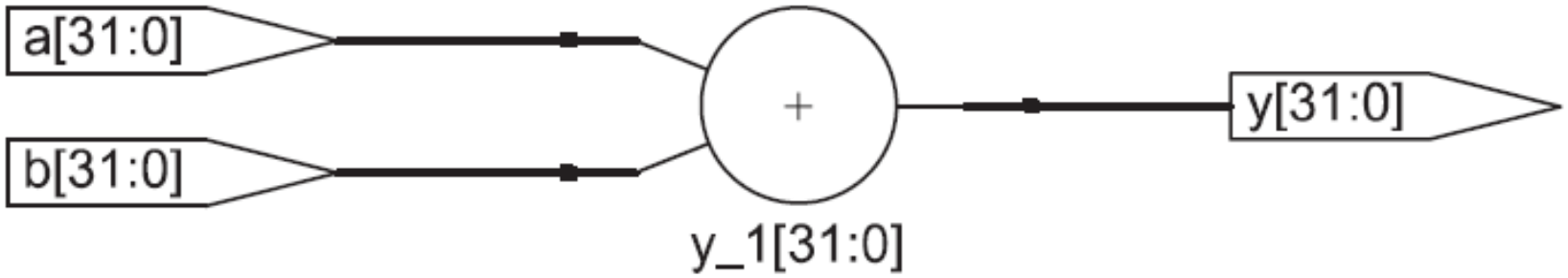


FIGURE A.3 Synthesized adder

T1: HARDWARE DESCRIPTION LANGUAGES (combinational bit wise operators)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

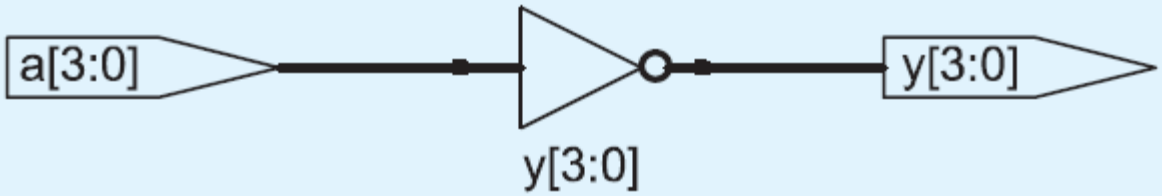


FIGURE A.4 inv

T1: HARDWARE DESCRIPTION LANGUAGES (combinational bit wise operators)

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

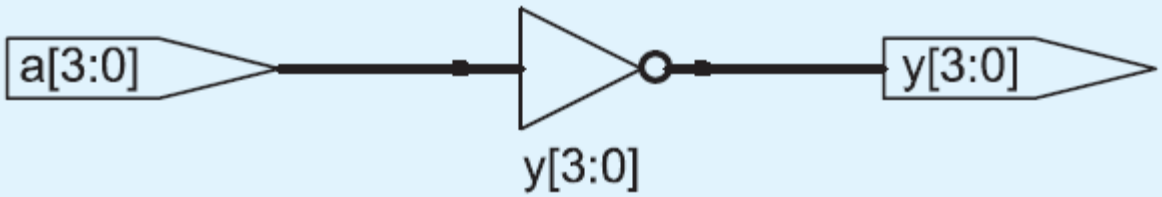


FIGURE A.4 inv

T1: HARDWARE DESCRIPTION LANGUAGES (combinational bit wise operators, comments)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
  port(a, b:   in  STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4,
       y5: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- Five different two-input logic gates
  -- acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (combinational bit wise operators, comments)

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (combinational bit wise operators)

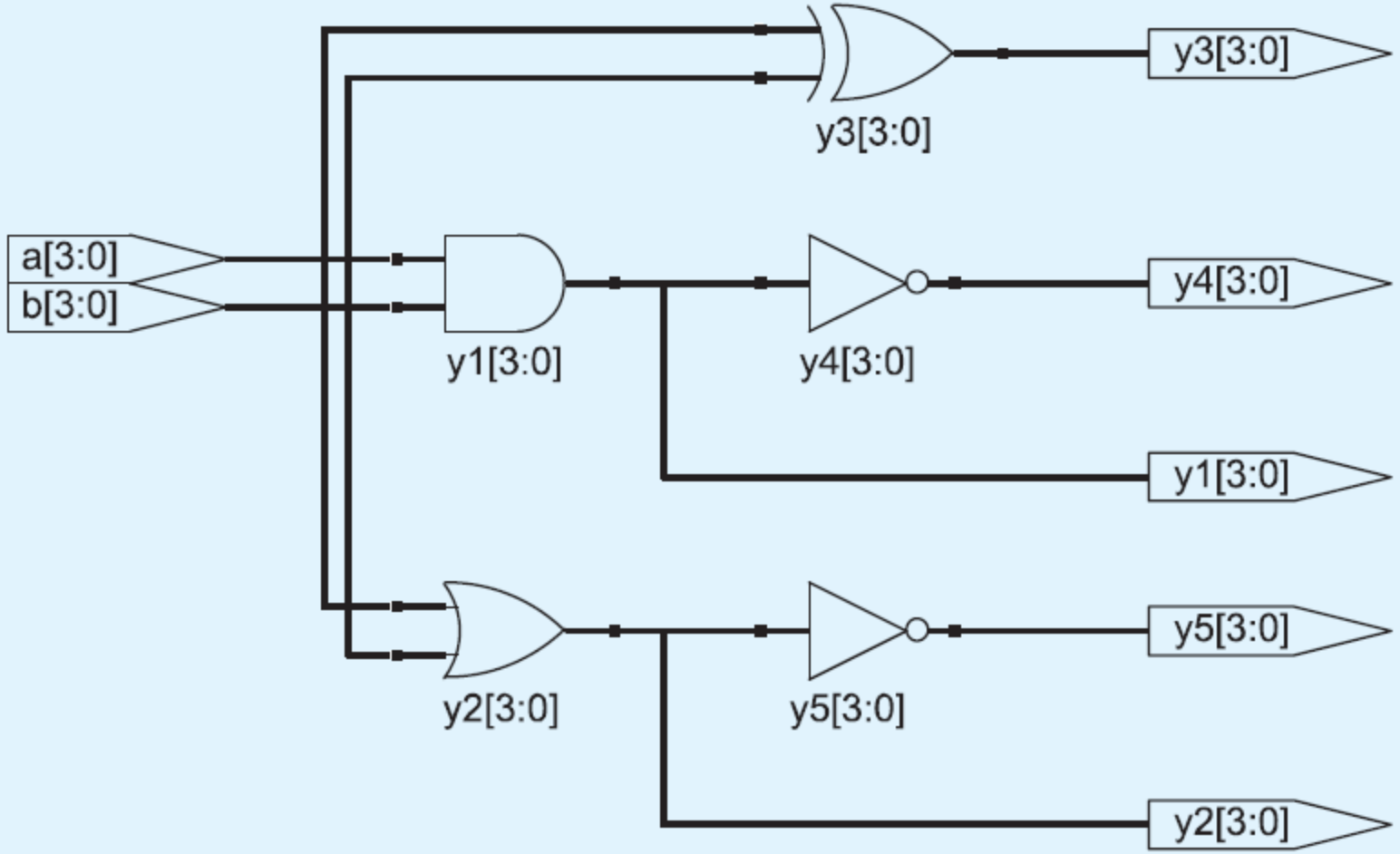


FIGURE A.5 Gates

T1: HARDWARE DESCRIPTION LANGUAGES (Reduction Operators / Generate)

VHDL

VHDL does not have reduction operators. Instead, it provides the `generate` command (see Section A.8). Alternately, the operation can be written explicitly:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= a(7) and a(6) and a(5) and a(4) and
       a(3) and a(2) and a(1) and a(0);
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Reduction Operators / Generate)

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

   assign y = &a;

   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```

As one would expect, |, ^, ~&, and ~| reduction operators are available for OR, XOR, NAND, and NOR as well. Recall that a multi-input XOR performs parity, returning TRUE if an odd number of inputs are TRUE.

T1: HARDWARE DESCRIPTION LANGUAGES (Reduction Operators / Generate)

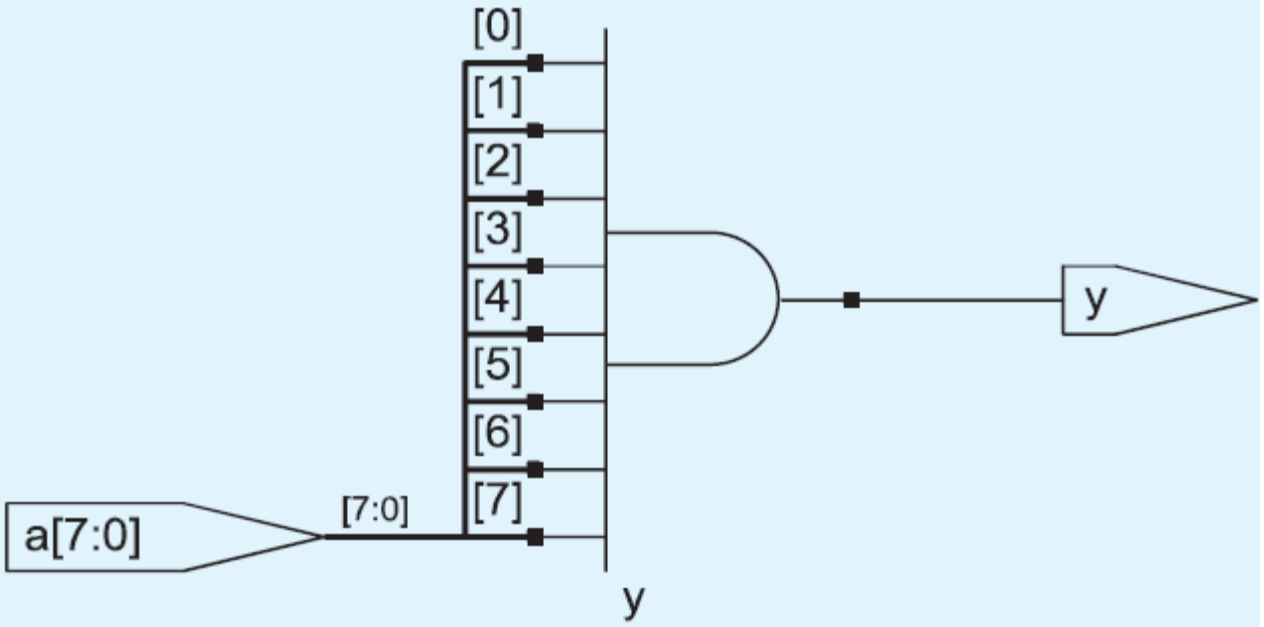


FIGURE A.6 and8

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1:in  STD_LOGIC_VECTOR(3 downto 0);
       s:     in  STD_LOGIC;
       y:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

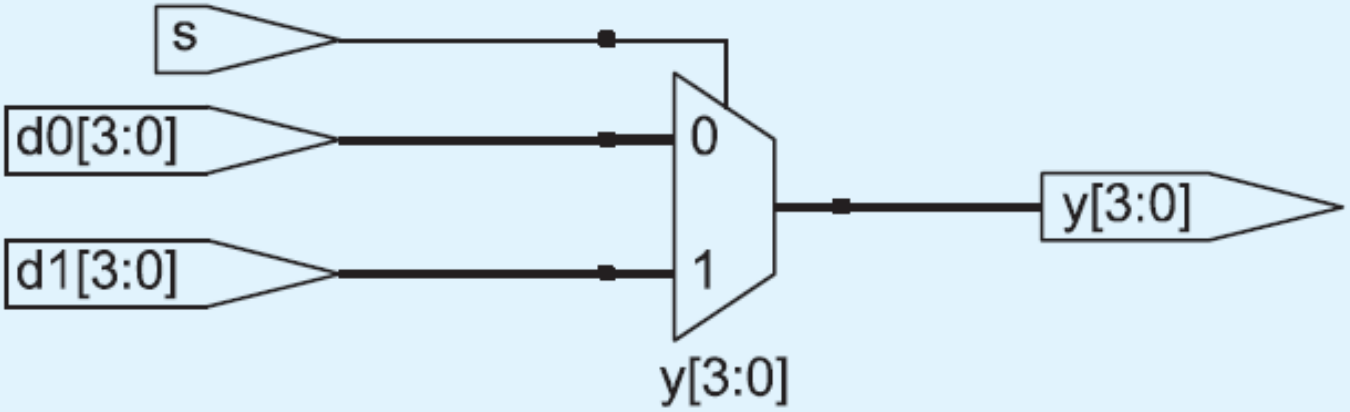


FIGURE A.7 mux2

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port(d0, d1,
    d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
    s:      in  STD_LOGIC_VECTOR(1 downto 0);
    y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

   assign y = s[1] ? (s[0] ? d3 : d2)
                   : (s[0] ? d1 : d0);
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (Conditional Assignments)

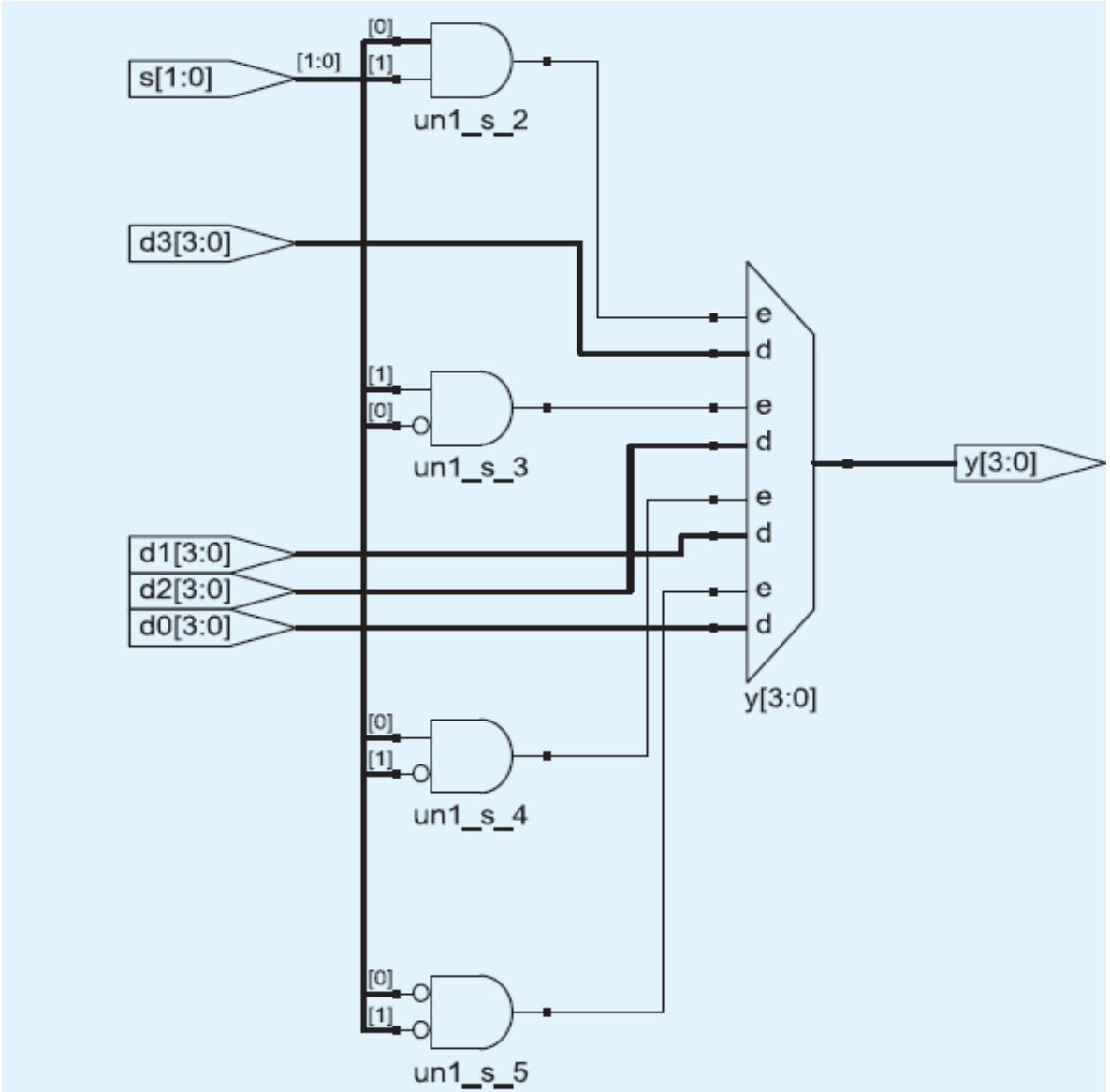


FIGURE A.8 mux4

T1: HARDWARE DESCRIPTION LANGUAGES (Internal Variables/Signals/Concurrence)

$$S = A \oplus B \oplus C_{\text{in}}$$
$$C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}} \tag{A.1}$$

If we define intermediate signals P and G

$$P = A \oplus B$$
$$G = AB \tag{A.2}$$

we can rewrite the full adder as

$$S = P \oplus C_{\text{in}}$$
$$C_{\text{out}} = G + PC_{\text{in}} \tag{A.3}$$

P and G are called *internal variables* because they are neither inputs nor outputs but are only used internal to the module. They are similar to local variables in programming languages. Example A.8 shows how they are used in HDLs.

T1: HARDWARE DESCRIPTION LANGUAGES (Internal Variables/Signals/Concurrence)

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Internal Variables/Signals/Concurrence)

SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (Internal Variables/Signals/Concurrence)

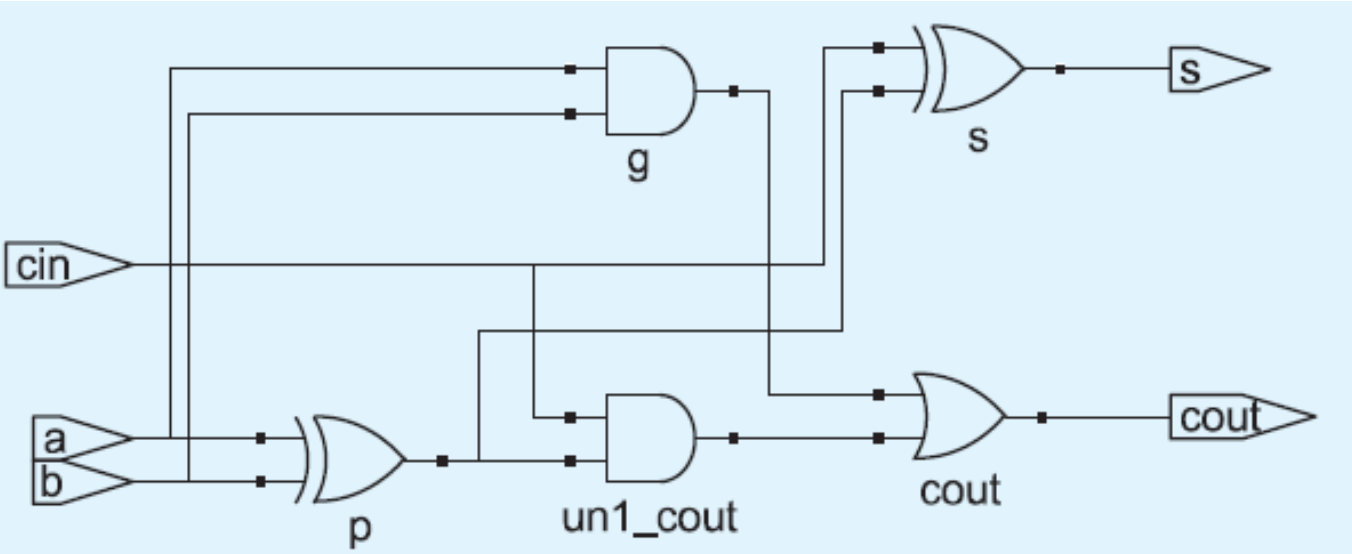


FIGURE A.9 fulladder

T1: HARDWARE DESCRIPTION LANGUAGES (Precdence and other operators)

VHDL

TABLE A.2 VHDL operator precedence

	Op	Meaning
Highest	not	NOT
	*, /, mod, rem	MUL, DIV, MOD, REM
	+, -, &	PLUS, MINUS, CONCATENATE
	rol, ror, srl, sll, sra, sla	Rotate, Shift logical, Shift arithmetic
Lowest	=, /=, <, <=, >, >=	Comparison
	and, or, nand, nor, xor	Logical Operations

T1: HARDWARE DESCRIPTION LANGUAGES (Precdence and other operators)

SystemVerilog

TABLE A.1 SystemVerilog operator precedence

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, –	PLUS, MINUS
	<<, >>	Logical Left / Right Shift
	<<<, >>>	Arithmetic Left / Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Lowest	?:	Conditional

T1: HARDWARE DESCRIPTION LANGUAGES (Numbers)

VHDL

In VHDL, STD_LOGIC numbers are written in binary and enclosed in single quotes. `'0'` and `'1'` indicate logic 0 and 1.

STD_LOGIC_VECTOR numbers are written in binary or hexadecimal and enclosed in double quotes. The base is binary by default and can be explicitly defined with the prefix `x` for hexadecimal or `B` for binary, as shown in Table A.4.

TABLE A.4 VHDL numbers

Numbers	Bits	Base	Val	Stored
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	161	10101011

T1: HARDWARE DESCRIPTION LANGUAGES (Numbers)

SystemVerilog

As shown in Table A.3, SystemVerilog numbers can specify their base and size (the number of bits used to represent them). The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is the base, and `value` gives the value. For example `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary (base 2), `'o` for octal (base 8), `'d` for decimal (base 10), and `'h` for hexadecimal (base 16). If the base is omitted, the base defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog shorthands for filling a bus with all 0s and all 1s.

T1: HARDWARE DESCRIPTION LANGUAGES (Numbers)

TABLE A.3 SystemVerilog numbers

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010
'1	?	n/a		11...111

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= "ZZZZ" when en = '0' else a;
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that `y` is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `trireg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (`z`), while a `trireg` retains the previous value. If no type is specified for an input or output, `tri` is assumed.

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

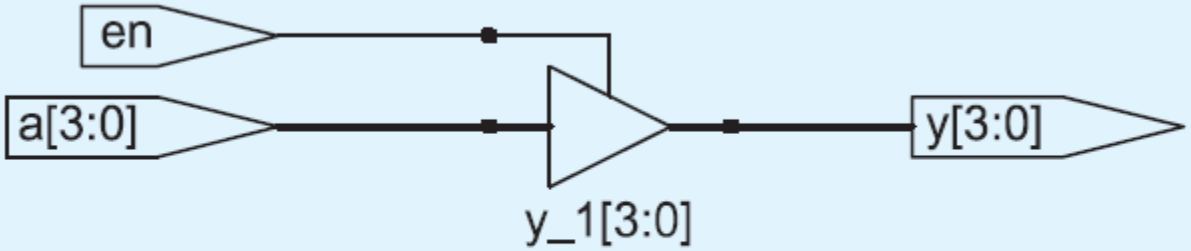


FIGURE A.10 tristate

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

VHDL

VHDL `STD_LOGIC` signals are `'0'`, `'1'`, `'z'`, `'x'`, and `'u'`.

Table A.6 shows a truth table for an AND gate using all five possible signal values. Notice that the gate can sometimes determine the output despite some inputs being unknown. For example, `'0'` and `'z'` returns `'0'` because the output of an AND gate is always `'0'` if either input is `'0'`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `'x'` in VHDL. Uninitialized inputs cause uninitialized outputs, displayed as `'u'` in VHDL.

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

TABLE A.6 VHDL AND gate truth table with `z`, `x`, and `u`

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

SystemVerilog

SystemVerilog signal values are `0`, `1`, `z`, and `x`. Constants starting with `z` or `x` are padded with leading `z`s or `x`s (instead of 0s) to reach their full length when necessary.

Table A.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example `0 & z` returns 0 because the output of an AND gate is always 0 if either input is 0. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x`.

T1: HARDWARE DESCRIPTION LANGUAGES (Zs and Xs)

TABLE A.5 SystemVerilog AND gate truth table with `z` and `x`

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

VHDL

```
y <= c(2 downto 1) & d(0) & d(0) & d(0) &
     c(0) & "101";
```

The `&` operator is used to *concatenate* (join together) busses. `y` must be a 9-bit `STD_LOGIC_VECTOR`. Do not confuse `&` with the `and` operator in VHDL.

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The `{}` operator is used to concatenate busses.

`{3{d[0]}}` indicates three copies of `d[0]`.

Don't confuse the 3-bit binary constant `3'b101` with bus `b`.

Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of `y`.

If `y` were wider than 9 bits, zeros would be placed in the most significant bits.

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mul is
  port(a, b: in  STD_LOGIC_VECTOR(7 downto 0);
       upper, lower:
            out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture behave of mul is
  signal prod: STD_LOGIC_VECTOR(15 downto 0);
begin
  prod <= a * b;
  upper <= prod(15 downto 8);
  lower <= prod(7 downto 0);
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

SystemVerilog

```
module mul(input  logic [7:0] a, b,
           output logic [7:0] upper, lower);

  assign {upper, lower} = a*b;
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

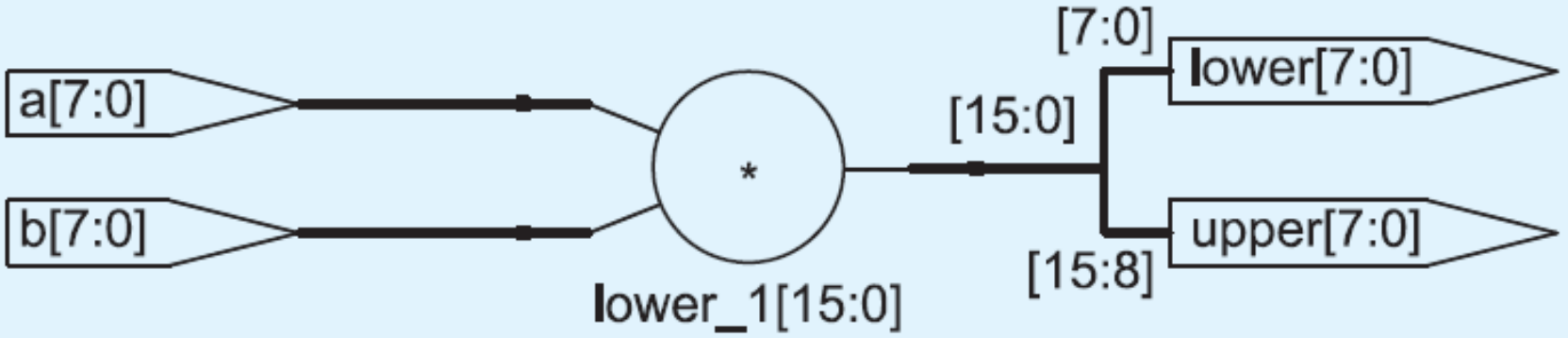


FIGURE A.11 Multipliers

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity signext is -- sign extender
  port(a: in  STD_LOGIC_VECTOR (15 downto 0);
       y: out STD_LOGIC_VECTOR (31 downto 0));
end;
architecture behave of signext is
begin
  y <= X"0000" & a when a (15) ='0' else X"ffff" & a;
end;
```

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

SystemVerilog

```
module signextend(input  logic [15:0] a,
                  output logic [31:0] y);

   assign y = {{16{a[15]}}, a[15:0]};
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (Bit Swizzling)

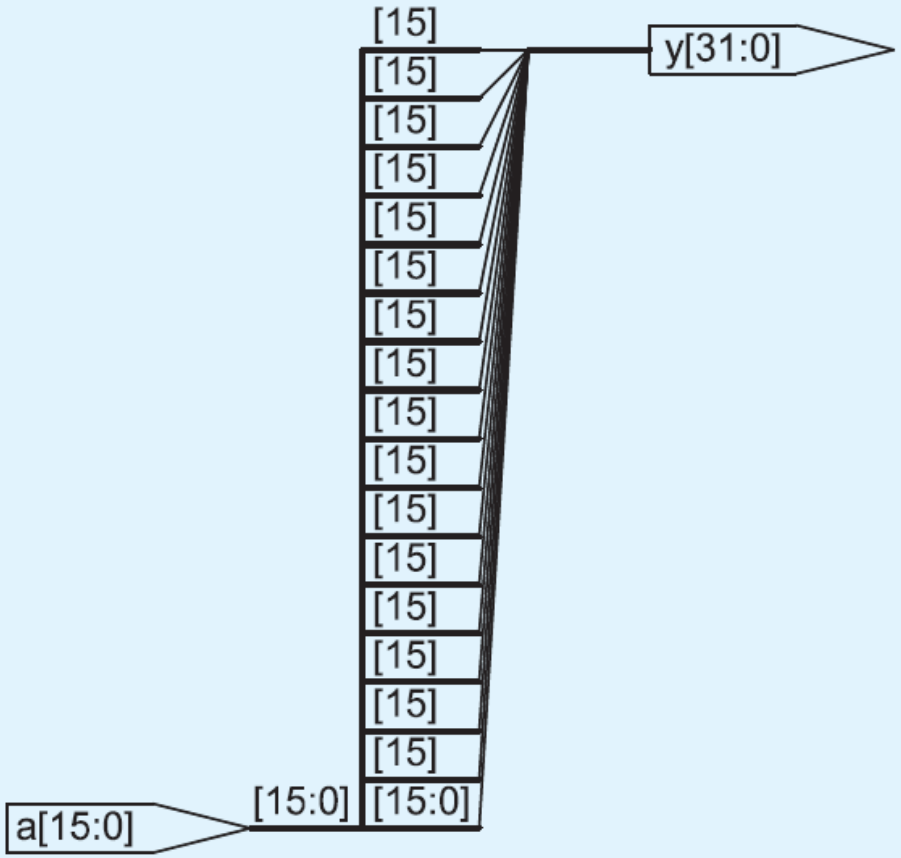


FIGURE A.12 Sign extension

T1: HARDWARE DESCRIPTION LANGUAGES (Delays)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;


architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the `after` clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

T1: HARDWARE DESCRIPTION LANGUAGES (Delays)

SystemVerilog

```
`timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

T1: HARDWARE DESCRIPTION LANGUAGES (Delays)

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `` `timescale unit/step ``. In this file, each unit is 1ns, and the simulation has 1 ps resolution. If no timescale directive is given in the file, a default unit and step (usually 1 ns for both) is used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as nonblocking (<=) and blocking (=) assignments that will be discussed in Section A.5.4.

T1: HARDWARE DESCRIPTION LANGUAGES (Delays)

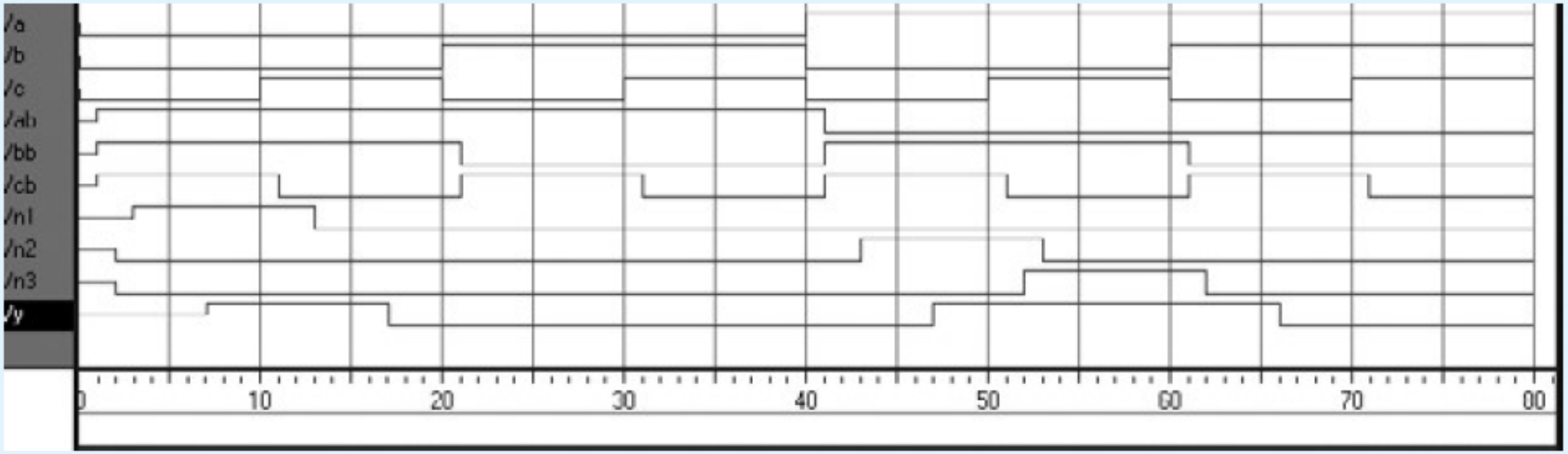


FIGURE A.13 Example simulation waveforms with delays