SoC Verification

ELEN 613

Bruce S. Greene

Hello!

Instructor Bruce S. Greene, PhD
BSEE : Boston University
MSEE: University of Illinois, Urbana-Champaign
    Electromagnetic / Antenna Theory
PhD : Santa Clara University
    DFT & Partial Scan Techniques
Employment
    Lockheed Martin : Microwave/ Simulation
    Synopsys : Test Automation
    C level Design : C based verification & synthesis
    Synopsys : Verification (SystemVerilog, Emulation)
        Principal Engineer
    SCU : Adjunct Faculty : 2004 -> Present
Email:
    bruceg@synopsys.com
    bsgreene@scu.edu

Class
    Wednesday 7:10->9 O'Conner Hall ~~[illegible]~~ 106
Office Hours
    45 minutes before class

- Take home assignment 20%
- Quiz 20%
- Final Project 60%

Course Objective

A typical System-On-Chip consists of 1 or more embedded processors, on-chip bus, off-the-shelf peripherials (camera, display, wifi, Bluetooth, dsp, ethernet,…) , memory (DDR), low-power management, and other custom components.

Over 70% of the time is spent verifying SoC chips before fabrication. Functional bugs found later in the manufacturing cycle will be very costly, and may require a re-spin

This course presents several logical-verification techniques that is commonly used in industry today to ensure there are no bug-escapes

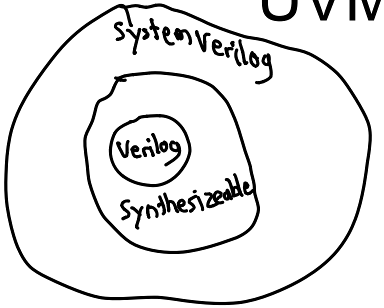
The techniques covered in this class will use the SystemVerilog design and verification language and will include code coverage, functional coverage, assertions, constraint random stimulus generation, and UVM (Universal Verification Methodology)

Expected Learning Outcomes

- Understand verification techniques such as coverage, assertions, test stimulus, and UVM
- Learn how to verify Verilog building blocks such as ALU, memory, FIFO’s
- Demonstrate how to write a testbench for a typical logic block
- Verify a reference SoC design with a custom peripheral

Course Outline

SoC Verification Overview
SystemVerilog Intro
Assertion-Based Verification (SVA)
Constraint-Based Random Testing
Functional Coverage
OOP + Classes in SV
General Verification Methodology
Verification Planning
UVM (Universal Verification Methodology)



Misc

Grading Policy

- 2-Take home Quizzes : 40%
- Project : 60%

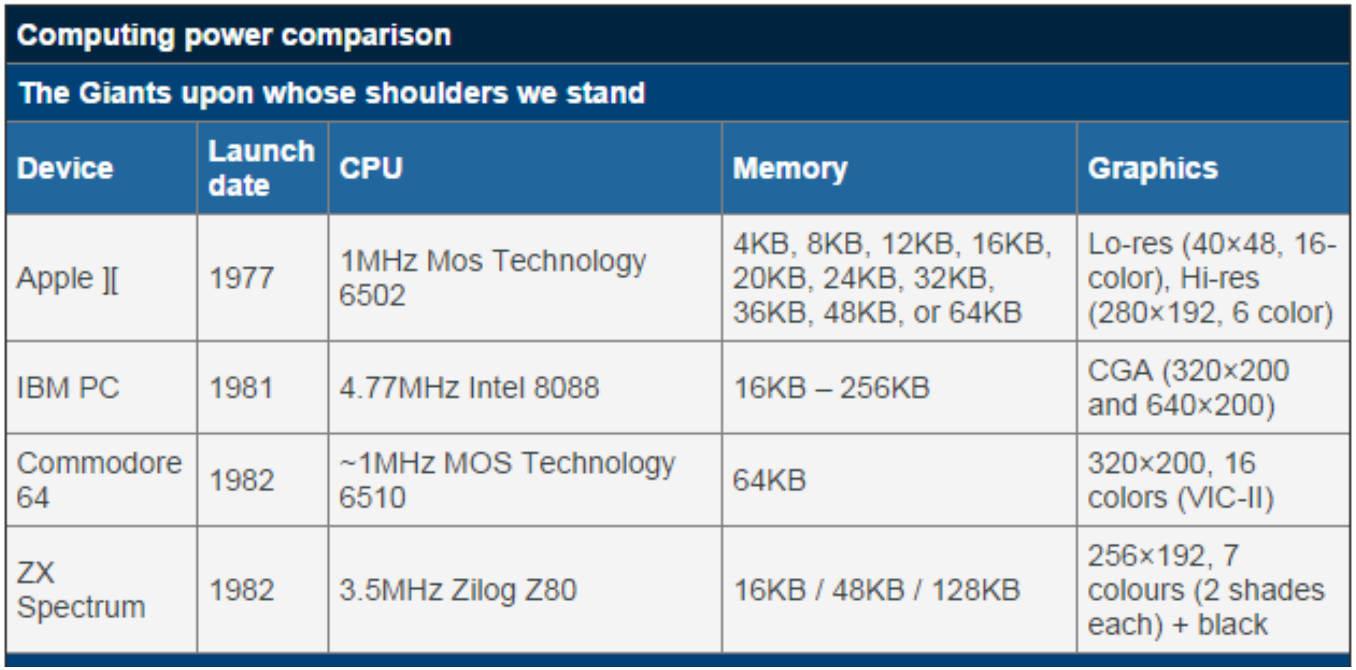
Lectures to be posted on Camino!

Design Center : Account Required

```
source /home/bgreene/setvcs
```

Computing Power

the old

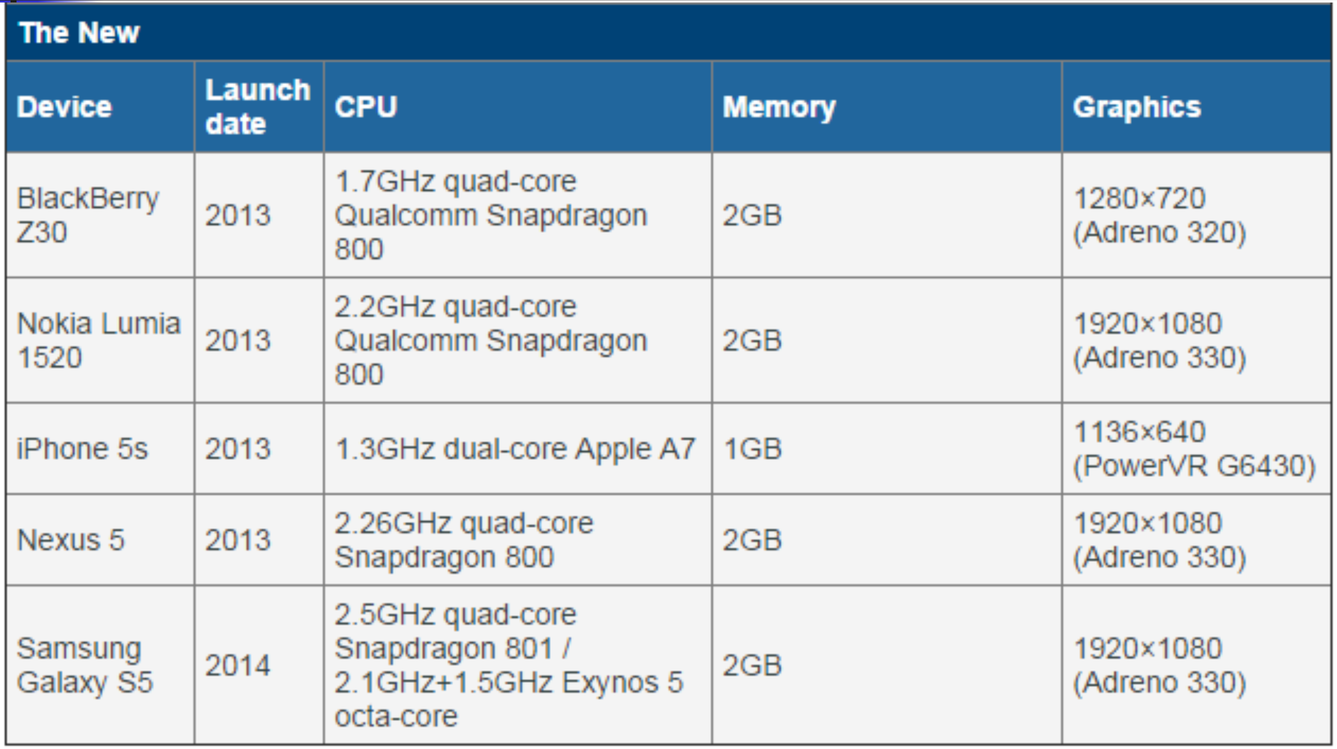


Computing power comparison

The Giants upon whose shoulders we stand

Device	Launch date	CPU	Memory	Graphics
Apple][	1977	1MHz Mos Technology 6502	4KB, 8KB, 12KB, 16KB, 20KB, 24KB, 32KB, 36KB, 48KB, or 64KB	Lo-res (40×48, 16-color), Hi-res (280×192, 6 color)
IBM PC	1981	4.77MHz Intel 8088	16KB – 256KB	CGA (320×200 and 640×200)
Commodore 64	1982	~1MHz MOS Technology 6510	64KB	320×200, 16 colors (VIC-II)
ZX Spectrum	1982	3.5MHz Zilog Z80	16KB / 48KB / 128KB	256×192, 7 colours (2 shades each) + black

Computing Power
the new



The New				
Device	**Launch date**	**CPU**	**Memory**	**Graphics**
BlackBerry Z30	2013	1.7GHz quad-core Qualcomm Snapdragon 800	2GB	1280×720 (Adreno 320)
Nokia Lumia 1520	2013	2.2GHz quad-core Qualcomm Snapdragon 800	2GB	1920×1080 (Adreno 330)
iPhone 5s	2013	1.3GHz dual-core Apple A7	1GB	1136×640 (PowerVR G6430)
Nexus 5	2013	2.26GHz quad-core Snapdragon 800	2GB	1920×1080 (Adreno 330)
Samsung Galaxy S5	2014	2.5GHz quad-core Snapdragon 801 / 2.1GHz+1.5GHz Exynos 5 octa-core	2GB	1920×1080 (Adreno 330)

Jan Vermeulen (mybroadband.co.za)

Evolution of IC Design

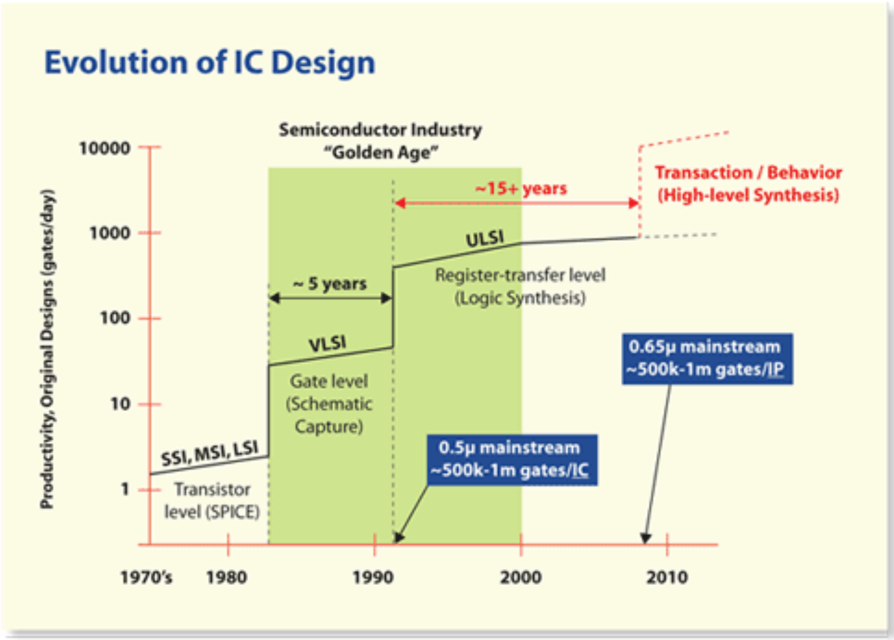
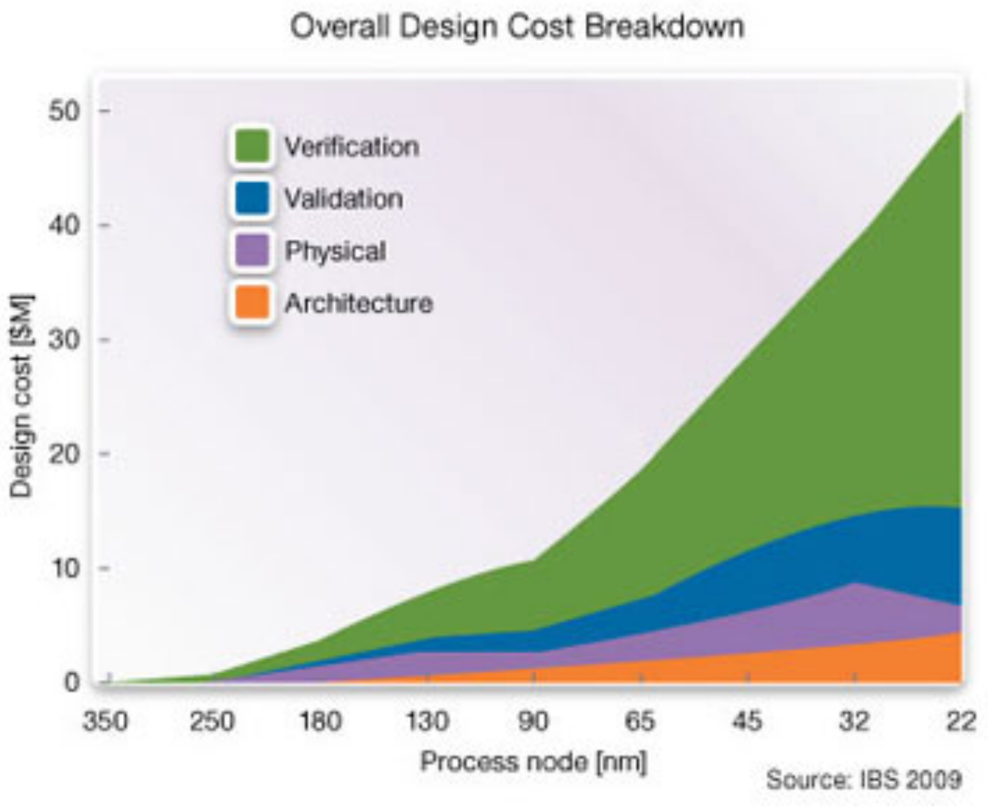


Figure 1. This chart shows the evolution of IC design from the mid-1970s to the present. The mid-1980s to 2000 was the semiconductor industry's period of fastest innovation, growth, and value-creation. This was driven in very large part by an over-100x increase in designer productivity.

Design Cost


Overall Design Cost Breakdown
Verification
Validation
Physical
Architecture
Design cost [$M]
50
40
30
20
10
0
350
250
180
130
90
65
45
32
22
Process node [nm]
Source: IBS 2009

Validation vs Verification

- Validation
  - Are we building the right product for today’s market?
- Verification
  - Is the product that we are build equivalent to the original spec?

Why Verify ?

82% of designs with re-spins resulting from logic and functional flows had design errors.
47% of designs with re-spins resulting from logic and functional flaws had incorrect or incomplete specification
32% of designs with re-spins resulting from logic and functional flaws had changes in specifications.
14% of all chips that failed had bugs in reused components or imported IP

Types of Verification

- Formal
  - Vector-less, uses formal mathematical properties to ascertain correctness against reference
- Dynamic
  - Vector-based scheme to verify a design against some reference model

Types of Verification

Logical Design Verification
Physical Verification
Timing Verification
Power Verification
Test Verification

Verification Complexity

SoC design
Growing number of features, modes, configurations
Power states
Increasing number of re-useable IP
HW/SW interaction
Increasing state-space
# of interfaces
..

How to reduce complexity?

Reduce chip complexity
Reduce number of designs
Increase resources
Increase productivity of engineers
Increase verification productivity
Increase size of compute farms

Design Challenges

- Create products fast
  - Time to market
  - Time to volume
- Create differentiation
- Put lots of features in your product

What is HDL?

- HDL = “Hardware Description Language”
- Describes hardware of digital systems in a text form
  - Logic diagrams
  - Boolean expressions
- Language is specialized to model digital circuits
  - Concurrency
  - Timing

What can I do with HDL??

- Logic Synthesis
  - "process of transforming a circuit from one level of abstraction to another"
  - Deriving a set of components and their interconnects with same functionality
  - Similar to compiling C code down to assembler code
    - Instead of object code, you get gates

What can I do with HDL??

- Logic Simulation
  - Reads in HDL code, interprets it and predicts how the digital circuit will perform in actual hardware
  - Allow one to test out the functional correctness of the circuit before fabricating it

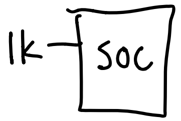
HDL’s in use today

- Standard HDL’s from IEEE
  - Verilog
    - Developed initially by Gateway and then donated to standards committee by Cadence
  - VHDL (Very High Speed Integrated Circuit Hardware Description Language)
    - Developed initially as a Department of Defense language, but now used in industry also

What is HVL?

HVL = "Hardware Verification Language"
Language tailored to building re-usable, flexible testbenches

- Higher level data-types
- Testbench – DUT connectivity
- → Constrained random stimulus generation
- → Functional coverage
- → Assertions



data {

address {

Vera, Specman "E", TestBuilder,…

Origins of SystemVerilog

1983/85 – Automated Integrated Design Systems (later as Gateway Design Automation)
1989/90 – acquired by Cadence Design Systems
1990/91 – opened to the public in 1990 (because of VHDL pressure) - OVI (Open Verilog International) was born
1992 – the first simulator by another company, more to follow
1993 – IEEE working group to produce the IEEE Verilog standard 1364
May 1995 – IEEE Standard 1364-1995
2001 – IEEE Standard 1364-2001 – revised version
2005 – IEEE Standard 1364-2005 – clarifications; Verilog-AMS
2005 – IEEE Standard 1800-2005 – SystemVerilog 2005
2009 – Verilog and SystemVerilog merged – IEEE Standard 1800-2009
2013 – IEEE Standard 1800-2012 – SystemVerilog 2012
2017 – IEEE Standard 1800-2017 – SystemVerilog 2017

SystemVerilog Intro

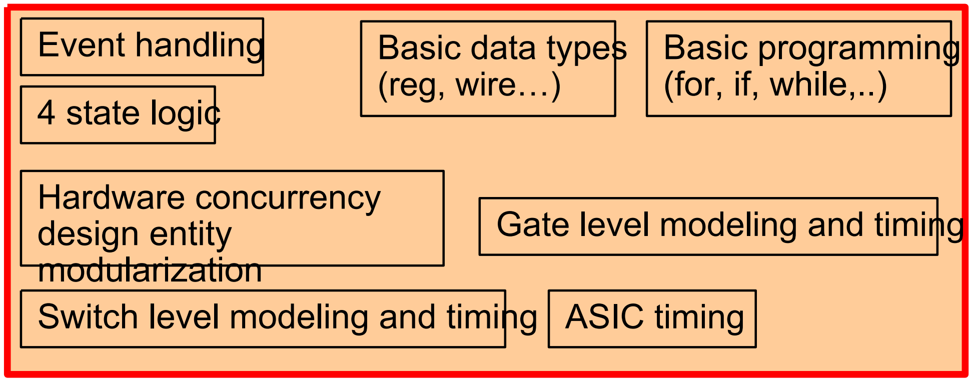
SystemVerilog is a superset of Verilog language

All usage, coding style that you learned in Verilog class still apply

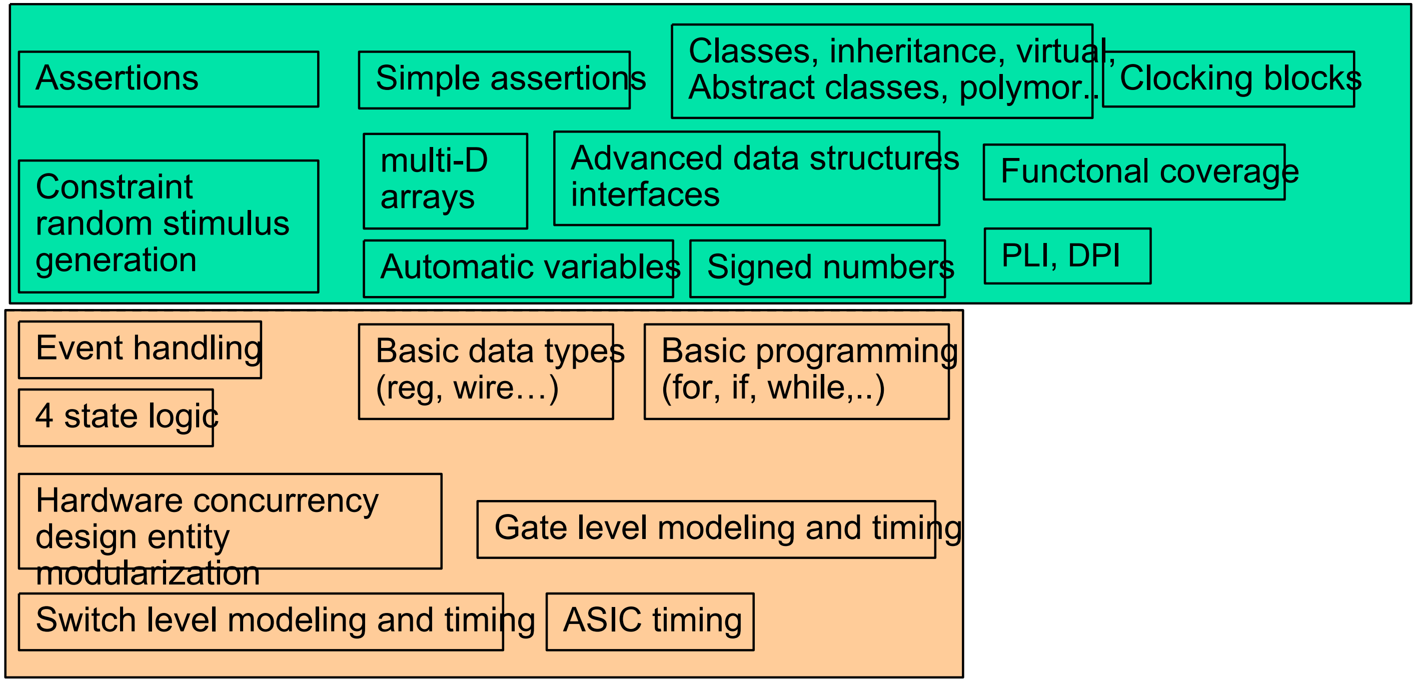
Part of SystemVerilog language is synthesizable for modeling hardware

Race conditions still exist in SystemVerilog, but there are addition coding styles/constructs that help prevent them

Verilog 1995/2001


Event handling
4 state logic
Basic data types (reg, wire…)
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modeling and timing
Switch level modeling and timing
ASIC timing

SystemVerilog


Assertions
Simple assertions
Classes, inheritance, virtual, Abstract classes, polymor..
Clocking blocks
Constraint random stimulus generation
multi-D arrays
Advanced data structures interfaces
Functonal coverage
Automatic variables
Signed numbers
PLI, DPI
Event handling
4 state logic
Basic data types (reg, wire…)
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modeling and timing
Switch level modeling and timing
ASIC timing

Data Types in Verilog

Simplest net is wire

- Nets
  - Represents physical connections between gates, modules,..
  - Doesn't store a value
  - It's value determined by the value of its drivers (gate or continuous assignment), If no driver, then it's high-impedance
  - net can be written by one or more continuous assignments, by primitive outputs, or through module ports.
  - The resultant value of multiple drivers is determined by the resolution function of the net type.
  - A net cannot be procedurally assigned
  - Built-in nettypes
    - wire tri tri0 supply0
    - wand triand tri1 supply1
    - wor trior trireg uwire

Data Types in Verilog

- Variables
  - Abstraction of a data storage element
  - Written to in initial & always blocks
  - Stores a value from one assignment to the next
  - Variables can be written by one or more procedural statements, including procedural continuous assignments. The last write determines the value.
  - Alternatively, variables can be written by one continuous assignment or one port.

variables can act as net

nets	variables
wire	reg ≠≠ register
	logic == reg

Variable data types

shortint 2-state data type, 16-bit signed integer
int 2-state data type, 32-bit signed integer
longint 2-state data type, 64-bit signed integer
byte 2-state data type, 8-bit signed integer or ASCII character
only 0,1 $\rightarrow$ bit 2-state data type, user-defined vector size, unsigned
0,1,x,z $\rightarrow$ logic 4-state data type, user-defined vector size, unsigned
reg 4-state data type, user-defined vector size, unsigned
integer 4-state data type, 32-bit signed integer
time 4-state data type, 64-bit unsigned integer

Unsized literal constants

An unsized single-bit value can be specified by preceding the single-bit value with an apostrophe ('), but without the base specifier. All bits of the unsized value shall be set to the value of the specified bit. In a

'0, '1, 'X, 'x, 'Z, 'z // sets all bits to specified value

no radix and it will fill

logic [1023:0] x = ‘1; //easier in SV

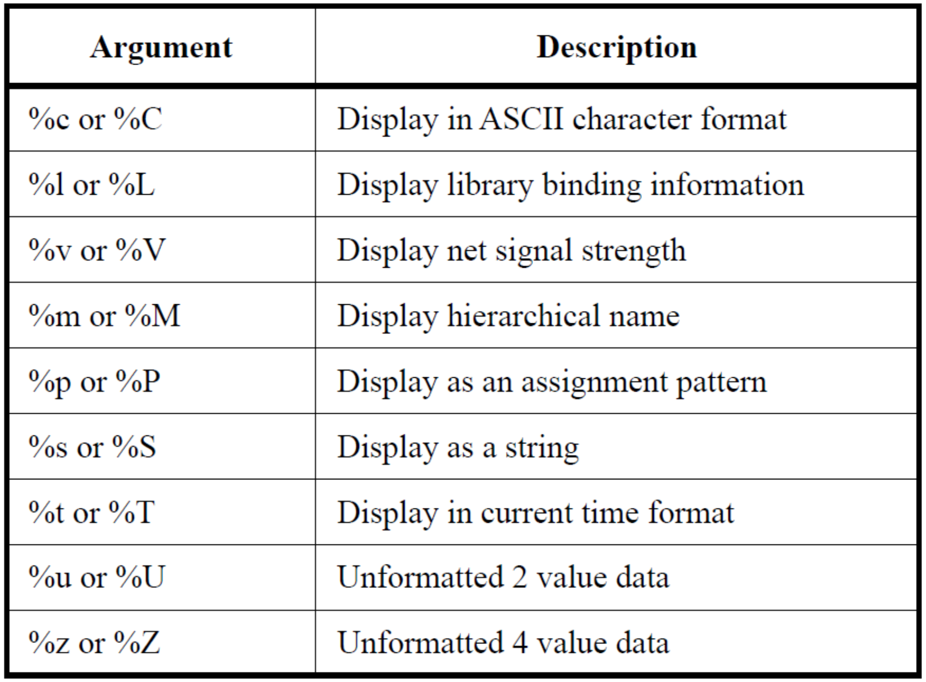
padding with 1's for all 1024 bits

Different than

logic [1023:0] x = ‘h1; only 1 bit is a 1

logic [63:0] = 0;
= 32'd0; will pad upper 32 bits w/ 0
= 64'h FFFF_FFFF

Format String



Argument	Description
%c or %C	Display in ASCII character format
%l or %L	Display library binding information
%v or %V	Display net signal strength
%m or %M	Display hierarchical name
%p or %P	Display as an assignment pattern
%s or %S	Display as a string
%t or %T	Display in current time format
%u or %U	Unformatted 2 value data
%z or %Z	Unformatted 4 value data

Writing '{en:'h0, data:'hde}
Reading '{en:'h1, data:'hef}

Logic , reg

In pre-SystemVerilog version, variables could only be driven procedurally

Reminder : reg != register

In SystemVerilog, variables can be written to procedural or through one-continuous assignment

logic == reg

Recommended to use “logic” (better name), but reg is equivalent

```
reg good
assign good=x;
assign good=y;
```

Cannot do this!!
If want, make "good" a wire

Examples

```
logic x;
initial x = 1'b1;
```

wire
~~reg~~ bad;
assign bad = x;

Syntax Error in non-System Verilog

```
logic y;
assign y = x;
```

```
reg z;
assign z = !y;
```

assign concurrently which is fine is SystemVerilog

Better Example

```
logic [7:0] count;
always @(posedge clk, negedge rst)
  if (!rst) count <= 8’b0000_0000;
  else
   count <= count + 1’b0;
```

Structures

structure represents a collection of data types that can be referenced as a whole , or the individual data
types that make up the structure can be referenced by name. By default, structures are unpacked

- Meaning: that there is an implementation-dependent packing of the data types. Unpacked structures can contain any data type.

Structures

```
typedef struct packed {
 logic [9:0]  head;
 logic [9:0]  tail;
 logic [9:0]  cnt;
 } t_fifo;

typedef struct packed {
  logic en;
  logic[7:0] data ;
  }  t_fifo_data;
```

packed means you can access as a single thing (an aggregate assignment

```
t_fifo control
:
control <= 0; ✓
```

Probably not recommeded though

Fifo example

```
module fifo  (output full, output empty, output t_fifo_data  dout, input t_fifo_data din,
                                               input clk, input reset, input push, input pop);
 t_fifo_data mem[1023:0];
 t_fifo fifo;
 assign dout = mem[fifo.tail];
 assign empty = (fifo.cnt == 0);
 assign full = (fifo.cnt == 1024);
always @(posedge clk or negedge reset) if (!reset) begin
          fifo.head <= 0;
          fifo.tail <= 0;
          fifo.cnt <= 0;
        end
        else if (pop && (!empty)) begin
          fifo.tail <= (fifo.tail + 1);
          fifo.cnt <= (fifo.cnt - 1);
        end
        else if (push && (!full)) begin
          mem[fifo.head] <= din;
          fifo.head <= (fifo.head + 1);
          fifo.cnt <= (fifo.cnt + 1);
        end
endmodule
```

Assignment for Next Week

Write a simple testbench that writes 4 random elements, then reads them, and checks that they are identical

Do Not Submit

assert push and in same cycle

```
vcs -full64 -sverilog test.v
                            ^
                          or .sv
```

Assignment to Structs

```
typedef struct {
   int addr;
   int crc;
   byte data[4];
} T_packet1;

// Use C-like means to initialize structure
T_packet1 pi = '{1,2,'{2,3,4,5}};
```

array of bytes

```
// Alternatively
pi.addr = 1;
Pi.crc = 2;
Pi.data[0] = 2;
```

reg [31:0] mem [1024]

mem [0] = ...
mem [1] = ...

~~mem~~ = 0;

reg == logic

logic [31:0] [1023:0] mem;

both packed

so:
mem = 0; is legal

Packed

① logic [8:0] data

② data.en = 1'b0
data.data = 8'b1111_0000;

③ data = '{1'b0, 8'b1111_0000};

Enumerated Types

Use for State Machine

```
typedef enum {red, yellow, green} t_light;
t_light light;
// by default enum’s are of type int (can override)

always @(posedge clk) begin
    case (light)
      red : light <= yellow;
      yellow : light <= green;
      green : light <= red;
    endcase
end
```

Enumerated Types

```
initial $monitor ("light is %s %d " , light.name(),light);
```

```
light is red           0
light is yellow          1
light is green          2
```

More Examples

```
typedef enum {blue=8, red=4, yellow=2, green=1} t_
light;
t_light light;

$display(light.name());
light = 8;    //  ???

light = blue;
$display(light.name());
end
```

Verilog - weakly typed

logic [7:0] x= 1'b0

SV: new types are strongly typed

Methods for Enum

First
Last
Next
Prev
Num
name

Casting

Verilog = weakly typed language
SystemVerilog = more strongly typed

Similar to C-style casting

light = t_light'(8);