ENCS5337: Chip Design Verification

Spring 2023/2024

Hardware Design Languages

Dr. Ayman Hroub

Many thanks to Dr. K. Eder and Dr. M. Mudawar for most of the slides

Hardware Description Languages (1)

- Describe the hardware structure and behavior of digital systems in a textual form
- NOT another software programming language
- Most popular hardware description languages today (all are IEEE standards) are **Verilog**, **SystemVerilog**, and **VHDL**
- They were built with simulation in mind
  - Synthesis and other back-end purposes were added at a later stage

Hardware Description Languages (2)

- Designers used to draw schematics, then gate level netlist, then Register Transfer Level
- Raising abstraction levels as designs get more complex and faster ways to find bugs were required
- Abstracted design – Behavioral vs Synthesizable - for faster simulation leading to finding bugs sooner than later
- Synthesis tools allow direct implementation from HDL code (higher productivity, earlier bug finding, and easier debugging)

HDL vs. Programming Languages

3 major new concepts of HDLs compared to software programming languages:

- **Connectivity:**
  - Ability to describe a design using simpler blocks and then connecting them together.
- **Time:**
  - Can specify a delay (in time units of simulator): (WHY?)
    - **and #2 (Y3, In3, Sel1, Sel0);**
- **Concurrency** is always assumed! (for structural style)
  - No matter in which order primitives/components are specified, a change in value of any input signal **activates** the component.
  - If 2 or more components are activated **concurrently**, they perform their actions **concurrently**.
  - Order of specification does not influence order of activation!
  - (NOTE: Statements inside behavioral blocks may be sequential)
- **synthesizability**

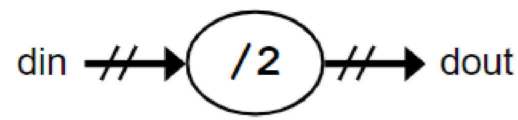
Example: Divide by 2

Behavioral

Block executed on every change in din

```
always @(din)
    dout = din/2 ;
```

"always" code block loops forever



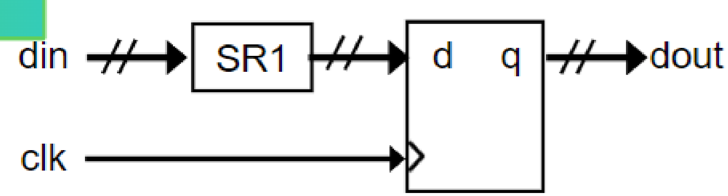
RTL

"Posedge " filters transitions to just at the positive edge of clk

```
always @(posedge clk)
  dout <= din >> 1 ;
```

Right shift

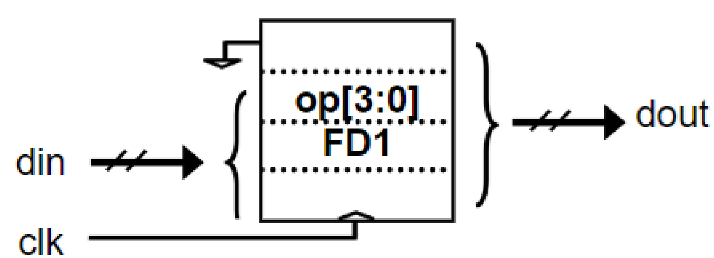
Special "nonblocking" assignment



Structural

Instance array "op" of cell type "FD1"

```
FD1 op[3:0] (
    .D({1'b0,din[3:1]}),
    .CP(clk), .Q(dout) );
```



Resource: Cadence

VHDL Background

- **VHSIC Hardware Description Language.**
  - VHSIC - Very High Speed Integrated Circuit.
- •Developed by the U.S. Department of Defense (1981)
  - 1986 rights where given to IEEE
  - Became a standard and published in 1987

Verilog (1)

- Verilog = "Verifying Logic "
- Invented as a simulation language in 1983 by Phil Moorby
- Opened to public in 1990 by Cadence Design Systems
- Became an IEEE standard in 1995 (Verilog-95)
- Revised and upgraded in 2001 (Verilog-2001)
- Revised also in 2005 (Verilog-2005)

Verilog (2)

- Verilog allows designers to describe hardware at different levels
  - Can describe anything from a single gate to a full computer system
- Verilog is supported by the majority of electronic design tools
- Verilog can be used for logic simulation and synthesis

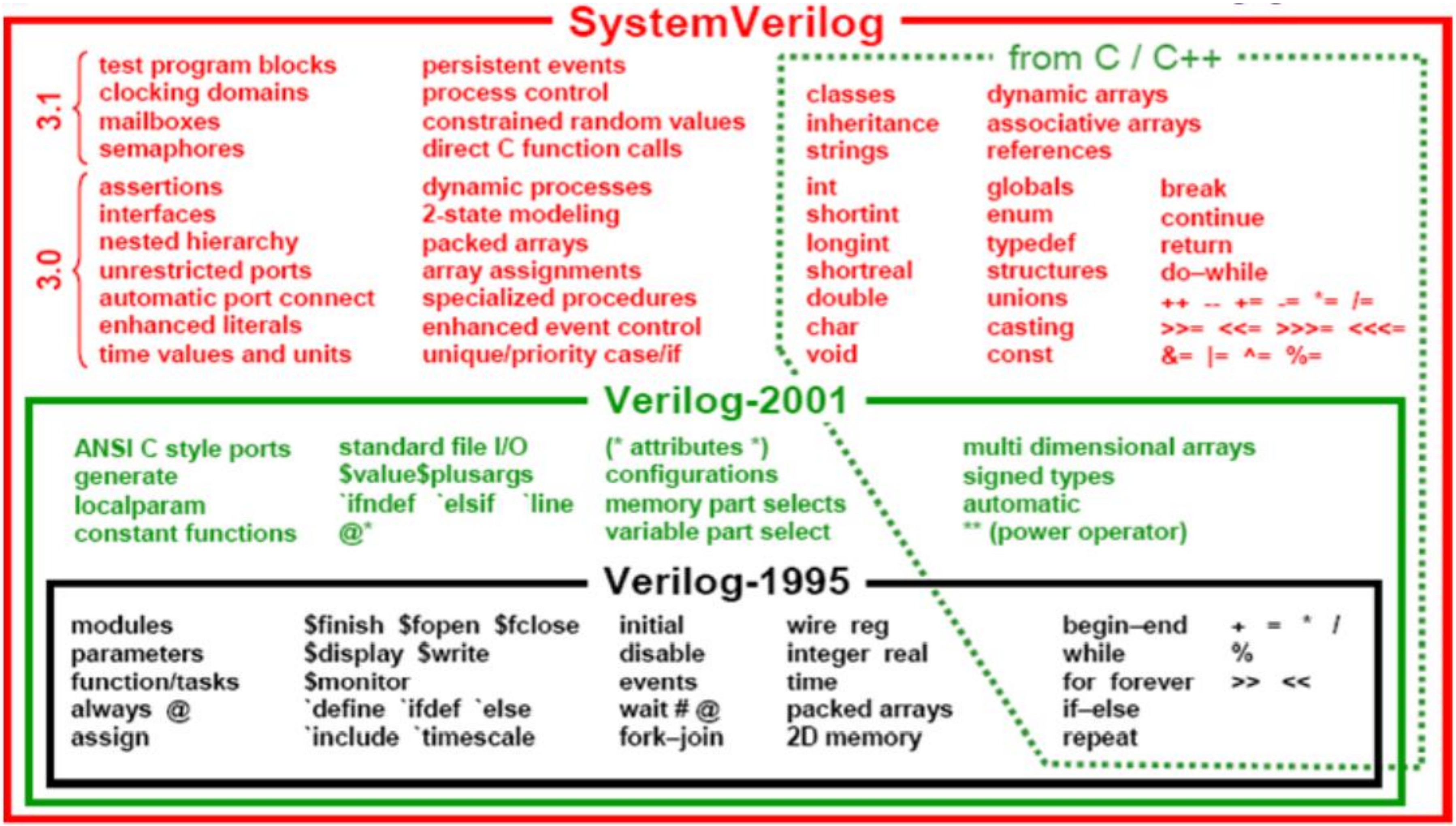
SystemVerilog (1)

- SystemVerilog is a Hardware **specification**, **design** and **Verification** language having features inherited from Verilog and C++ - Object Oriented Programming
- It is a solution to close the gap between design and verification language
- It is the (next generation of Verilog) designed to improve abstraction of Verilog
  - Abstraction levels
  - Data types
  - Verification constructs

SystemVerilog (2)

- Complete verification environment to improve the verification process:
  - Constrained Random Generation
  - Assertion Based Verification
  - Coverage Driven Verification
- Provides hardware-modeling features, which improve the RTL design productivity and simplify the design process

Verilog vs SystemVerilog



Logic Simulation

- Logic simulator interprets the Verilog (HDL) description
- Produces **timing diagrams**
- Predicts how the hardware will behave before it is fabricated
- Simulation allows the detection of functional errors in a design without having to physically implement the circuit
- Errors detected during the simulation can be corrected by modifying the appropriate statements in the Verilog code
- Simulating and verifying a design requires a **test bench**
- The test bench is also written in Verilog

Logic Synthesis

- Logic synthesis is similar to translating a program
- However, the output of logic synthesis is a digital circuit
- A digital circuit modeled in Verilog can be translated into a list of components and their interconnections, called **netlist**
- Synthesis can be used to fabricate an integrated circuit
- Synthesis can also target a Field Programmable Gate Array (FPGA)
- Logic simulation and synthesis are automated using special software, called Electronic Design Automation (EDA) tools

Verilog Code Structure

- Verilog only has one building block
  - **Module:** modules are connected through their ports similarly as in VHDL
  - Usually there is only one module per file.
  - A top level invokes instances of other modules.
  - Modules can be specified behaviorally or structurally.

Verilog Module

- A digital circuit is described in Verilog as a set of modules
- A module is declared using the **module** keyword
- A module is terminated using the **endmodule** keyword
- A module has a name and a list of **input** and **output** ports
- A module is described by a group of statements

Verilog Coding Styles

- **Structural**
- **Dataflow**
- **Behavioural** (best for verification)

Structural and Dataflow Modeling

- **Structural Modeling** using module instantiation. It describes the structure of a circuit with modules at different levels
- **Dataflow Modeling** using **Continuous Assignment**
  - Used mostly for describing Boolean equations and combinational logic
  - Verilog provides a rich set of operators
  - **Can describe:** adders, comparators, multiplexers, etc.
  - Synthesis tool can map a dataflow model into a target technology

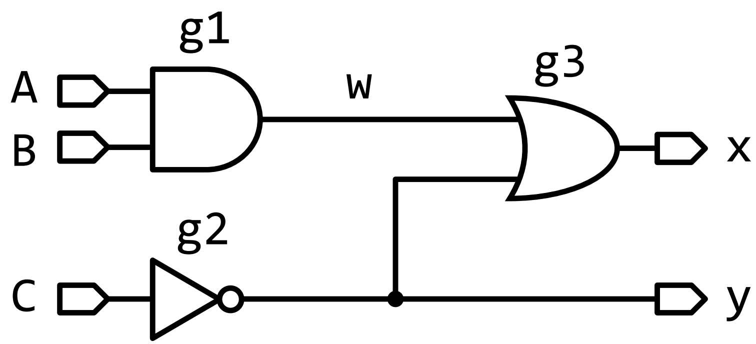
Behavioral Modeling

- Behavioral Modeling using Procedural Blocks and Statements
  - Describes what the circuit does at a functional and algorithmic level
  - Encourages designers to rapidly create a prototype
  - Can be verified easily with a simulator
  - Some procedural statements are synthesizable (Others are NOT)

Example of a Module in Verilog

```
// Description of a simple circuit
module simple_circuit(input A, B, C, output x, y);
  wire w;
  and g1(w, A, B);
  not g2(y, C);
  or  g3(x, w, y);
endmodule
```

Order is not important



The **input** keyword defines the input ports: A, B, C

The **output** keyword defines the output ports: x, y

The **wire** keyword defines an internal connection: w

The structure of `simple_circuit` is defined by three gates: **and**, **not**, **or**

Each gate has an optional name, followed by the gate output then inputs

Verilog Syntax

- **Keywords:** have special meaning in Verilog
  - Many keywords: module, input, output, wire, and, or, etc.
  - Keywords cannot be used as identifiers
- **Identifiers:** are user-defined names for modules, ports, etc.
- Verilog is **case-sensitive**: A and a are different names
- **Comments:** can be specified in two ways (similar to C)
  - Single-line comments begin with // and terminate at end of line
  - Multi-line comments are enclosed between /* and */
- **White space:** space, tab, newline can be used freely in Verilog
- **Operators:** operate on variables (similar to C: **~ & | ^ + -** etc.)

Verilog Operators

<table>
<tr><th colspan="2">Bitwise Operators</th></tr>
<tr><td>~a</td><td>Bitwise NOT</td></tr>
<tr><td>a & b</td><td>Bitwise AND</td></tr>
<tr><td>a | b</td><td>Bitwise OR</td></tr>
<tr><td>a ^ b</td><td>Bitwise XOR</td></tr>
<tr><td>a ~^ b</td><td>Bitwise XNOR</td></tr>
<tr><td>a ^~ b</td><td>Same as ~^</td></tr>
</table>

Arithmetic Operators	
a + b	ADD
a - b	Subtract
-a	Negate
a * b	Multiply
a / b	Divide
a % b	Remainder

Relational Operators	
a == b	Equality
a != b	Inequality
a < b	Less than
a > b	Greater than
a <= b	Less or equal
a >= b	Greater or equal

<table>
<tr><th colspan="2">Reduction Operators</th></tr>
<tr><td>&a</td><td>AND all bits</td></tr>
<tr><td>|a</td><td>OR all bits</td></tr>
<tr><td>^a</td><td>XOR all bits</td></tr>
<tr><td>~&a</td><td>NAND all bits</td></tr>
<tr><td>~|a</td><td>NOR all bits</td></tr>
<tr><td>~^a</td><td>XNOR all bits</td></tr>
</table>

Shift Operators	
a << n	Shift Left
a >> n	Shift Right

Miscellaneous Operators	
sel?a:b	Conditional
{a, b}	Concatenate

Reduction operators produce a 1-bit result

Relational operators produce a 1-bit result

{a, b} concatenates the bits of **a** and **b**

Bit Vectors in Verilog

- A Bit Vector is multi-bit declaration that uses a single name
- A Bit Vector is specified as a Range **`[msb:lsb]`**
- **`msb`** is ***most-significant bit*** and **`lsb`** is ***least-significant bit***
- Examples:

```
input  [15:0] A;      // A is a 16-bit input vector
output [0:15] B;      // Bit 0 is most-significant bit
wire   [3:0]  W;      // Bit 3 is most-significant bit
```

- **Bit select**: **`W[1]`** is bit **`1`** of **`W`**
- **Part select**: **`A[11:8]`** is a 4-bit select of **`A`** with range **`[11:8]`**
- The part select range must be consistent with vector declaration

Reduction Operators

```
module Reduce
  ( input [3:0] A, B, output X, Y, Z );
  // A, B are input vectors, X, Y, Z are 1-bit outputs

  // X = A[3] | A[2] | A[1] | A[0];
  assign X = |A;

  // Y = B[3] & B[2] & B[1] & B[0];
  assign Y = &B;

  // Z = X & (B[3] ^ B[2] ^ B[1] ^ B[0]);
  assign Z = X & (^B);

endmodule
```

Concatenation Operator { }

```
module Concatenate
  ( input [7:0] A, B, output [7:0] X, Y, Z );
  // A, B are input vectors, X, Y, Z are output vectors

  // X = A is right-shifted 3 bits using { } operator
  assign X = {3'b000, A[7:3]};

  // Y = A is right-rotated 3 bits using { } operator
  assign Y = {A[2:0], A[7:3]};

  // Z = selecting and concatenating bits of A and B
  assign Z = {A[5:4], B[6:3], A[1:0]};
endmodule
```

Integer Literals (Constant Values)

- Syntax: `[size]['base]value`

  `size` (optional) is the number of bits in the value

  `'base` can be: `'b`(binary), `'o`(octal), `'d`(decimal), or `'h`(hex)

  `value` can be in binary, octal, decimal, or hexadecimal

- If the `'base` is not specified then decimal `value`

- Examples:

  `8'b1011_1101` (8-bit binary), `'hA3F0` (16-bit hexadecimal)

  `16'o56377` (16-bit octal), `32'd999` (32-bit decimal)

- The underscore _ can be used to enhance readability of `value`

- When `size` is fewer bits than `value`, upper bits are truncated

If Statement

- The `if` statement is procedural
- Can only be used inside a procedural block
- Syntax:

```
if (expression) statement
[ else statement ]
```

- The `else` part is optional

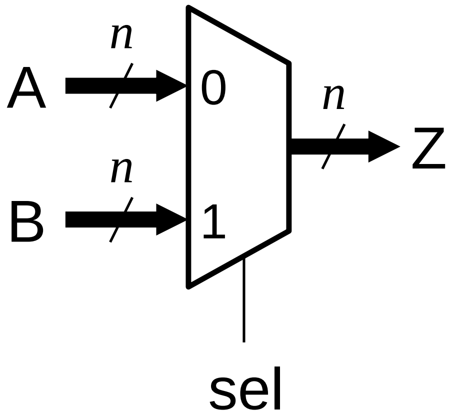
  A ***statement*** can be simple or compound

  A ***compound statement*** is surrounded by `begin ... end`
- `if` statements can be nested
- Can be nested under `if` or under `else` part

Modeling a 2x1 Multiplexer

```
// Behavioral Modeling of a Parametric 2x1 Mux
module Mux2 #(parameter n = 1)
  ( input [n-1:0] A, B, input sel,
    output reg [n-1:0] Z);

  // Output Z must be of type reg
  // Sensitivity list = @(A, B, sel)
  always @(A, B, sel) begin
    if (sel == 0) Z = A;
    else Z = B;
  end
endmodule
```



Case Statement

- The **case** statement is procedural (used inside **always** block)
- Syntax:

```
case (expression)
  case_item1:   statement
  case_item2:   statement
  . . .
  default:      statement
endcase
```

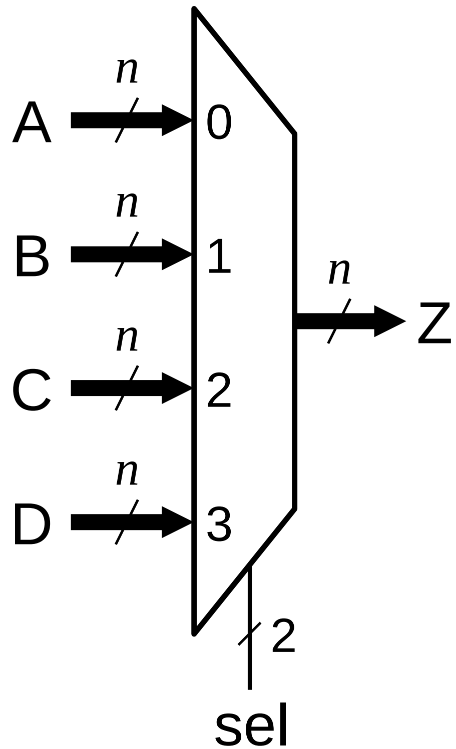
The **default** case is optional

A ***statement*** can be simple or compound

A ***compound statement*** is surrounded by **begin ... end**

Modeling a Mux with a Case Statement

```
module Mux4 #(parameter n = 1)
  ( input  [n-1:0] A, B, C, D, input [1:0] sel,
    output reg [n-1:0] Z );
  // @(*) is @(A, B, C, D, sel)
  always @(*) begin
    case (sel)
      2'b00:   Z = A;
      2'b01:   Z = B;
      2'b10:   Z = C;
      default: Z = D;
    endcase
  end
endmodule
```



Conditional Operator

- Syntax:

  Boolean_expr ? True_expression : False_expression

  If ***Boolean_expr*** is true then select ***True_expression***

  Else select ***False_Expression***

- Examples:

```
assign max = (a>b)? a : b;  // maximum of a and b

assign min = (a>b)? b : a;  // minimum of a and b
```

- Conditional operators can be nested

Basic Gates

- Basic gates: **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf**
- Verilog define these gates as **keywords**
- Each gate has an optional name
- Each gate has an output (listed first) and one or more inputs
- The **not** and **buf** gates can have only one input
- Examples:

```
and g1(x,a,b);     // 2-input and gate named g1
or  g2(y,a,b,c);   // 3-input or  gate named g2
nor g3(z,a,b,c,d); // 4-input nor gate named g3
```

name output inputs

Continuous Assignment

- **Used in Dataflow coding style.**
  - `assign #4 Out = In1 & In2;`
- Keyword **assign** followed by optional delay declaration
- LHS (target) can be net (scalar or vector) or concatenation of nets
  - NO registers allowed as target for assignment. Only output port or wire
- Assignment symbol: **=**
- RHS is an expression.
- **Implicit continuous assignment:** `wire x = ...;`
- **Conditional assignment:**
  - `assign Out = Sel ? In1 : In0;`
  - If `Sel` is 1 then `In1` is assigned to `Out`; if `Sel` is 0 then `Out` is `In0`.

Continuous Assignment: Execution

- Continuous assignments are **always active**.
- The order of the `assign` statements does not matter
- **Concurrency:**
  - When any of the operands on RHS changes, assignment is evaluated.
  - Several assignments can be executed concurrently.
  - **Race conditions** can occur!
    - Two or more assignments, which operate on the same data, read and write the data concurrently.
    - Result, which might be erroneous, depends on which assignment does what when.
- **Delays** specify time between change of operand on RHS and assignment of resulting value to LHS target.
  - `assign #4 Out = In1 & In2;`

TestBench

- In order to simulate a circuit, it is necessary to apply inputs to the circuit for the simulator to generate an output response
- A testbench is written to **verify the correctness** of a DUT
- A testbench is written as a **Verilog module with no ports**
- It **instantiates the** DUT
- It **provides inputs** to the DUT
- Test benches can be complex and lengthy, depending on the complexity of the design

Example of a Simple TestBench

```
module Test_Full_Adder;        // No need for Ports
  reg  a, b, c;  // variable inputs
  wire sum, cout;// wire outputs
  // Instantiate the module to be tested
  Full_Adder FA (a, b, c, cout, sum);
  initial begin  // initial block
    a=0; b=0; c=0;     // at t=0  time units
    #20 a=1; b=1;// at t=20 time units
    #20 a=0; b=0; c=1; // at t=40 time units
    #20 a=1; c=0;// at t=60 time units
    #20 $finish; // at t=80 finish simulation
  end // end of initial block
endmodule
```

Nets and Variables

Verilog has two major data types:

1. **Net data types:** are connections between parts of a design
2. **Variable data types:** can store data values

- The `wire` is a net data type (physical connection)
  - A wire cannot store the value of a procedural assignment
  - However, a wire can be driven by continuous assignment
- The `reg` is a variable data type
  - Can store the value of a procedural assignment
  - However, cannot be driven by continuous assignment
  - Other variable types: `integer`, `time`, `real`, and `realtime`

Verilog Signals

- Verilog signals:
  - nets (used for “connections”, no storage capacity)
  - registers (storage capacity, similar to variables in software programming languages)
- Verilog external signals:
  - ports (input, output or inout, port connecting rules)

Verilog Four-Valued Logic

- Verilog Value Set consists of four basic values:

  0 – represents a logic zero, or false condition

  1 – represents a logic one, or true condition

  X – represents an unknown logic value

  Z – represents a high-impedance value

x or **X** represents an unknown or uninitialized value

z or **Z** represents the output of a disabled tri-state buffer

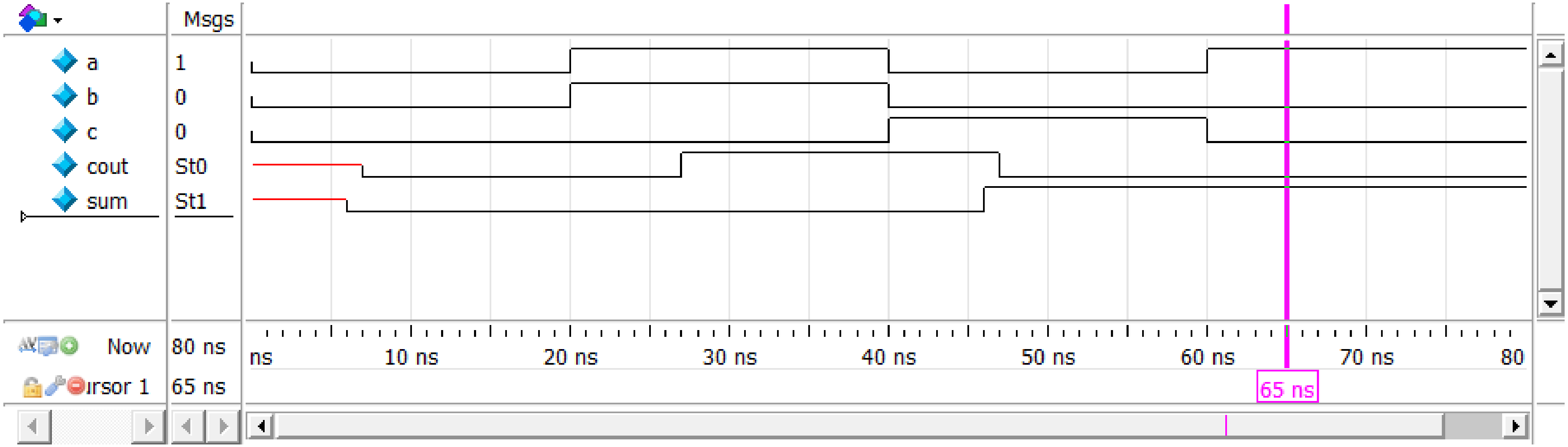
The `initial` Statement

- The **`initial`** statement is a procedural block of statements
- The body of the **`initial`** statement surrounded by **`begin`**-**`end`**
- It is sequential, like a sequential block in a programming language
- Procedural assignments are used inside the **`initial`** block
- Procedural assignment statements are executed in sequence

  Syntax: `#`*`delay variable = expression`*`;`

- Procedural assignment statements can be delayed
- The optional ***`#delay`*** indicates that the *`variable`* (of **`reg`** type) should be updated after the time delay

Running the Simulator



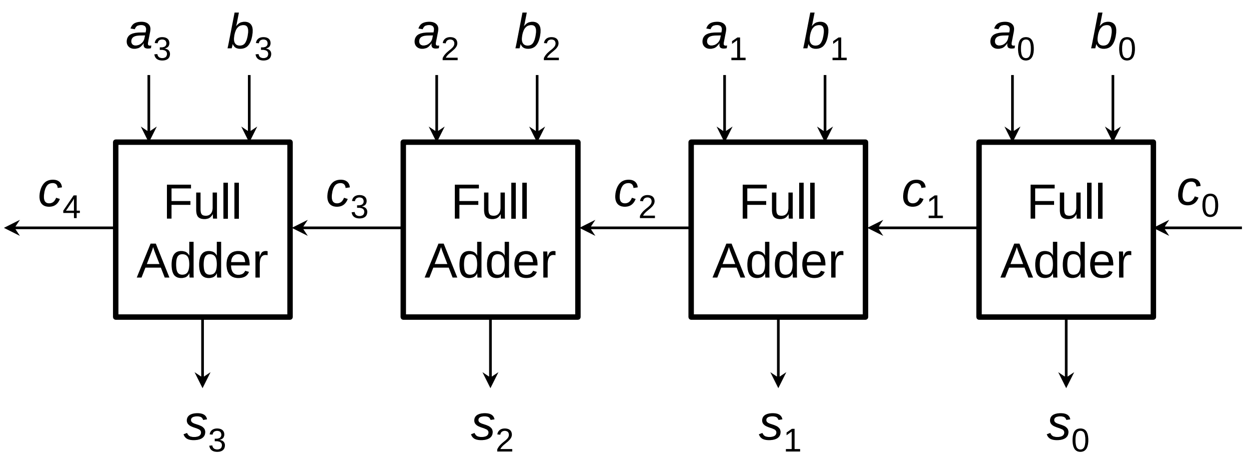
Examine the waveforms to verify the correctness of your design

At t = 0 ns, the values of **cout** and **sum** are unknown (shown in red)

The **cout** and **sum** signals are delayed by **7ns** and **6ns**, respectively

Modular Design: 4-bit Adder

- Uses **identical copies** of a full adder to build a large adder
- Simple to implement: the **cell** (iterative block) is a **full adder**
- Carry-out of cell *i* becomes carry-in to cell (*i* +1)
- Can be extended to add any number of bits



4-bit Adder using Module Instantiation

```
module Adder4 (input a0, a1, a2, a3, b0, b1, b2, b3, c0,
  output s0, s1, s2, s3, c4
);
  wire c1, c2, c3;  // Internal wires for the carries
 // Instantiate Four Full Adders: FA0, FA1, FA2, FA3
 // The ports are matched by position
  Full_Adder FA0 (a0, b0, c0, c1, s0);
  Full_Adder FA1 (a1, b1, c1, c2, s1);
  Full_Adder FA2 (a2, b2, c2, c3, s2);
  Full_Adder FA3 (a3, b3, c3, c4, s3);
 // Can also match the ports by name
 // Full Adder FA0 (.a(a0), .b(b0), .c(c0), .cout(c1),
.sum(s0));
endmodule
```

Module Instantiation

- Module declarations are like **templates**
- Module instantiation is like creating an **object**
- Modules are instantiated inside other modules at different levels
- Module instantiation defines the **structure** of a digital design
- It produces **module instances** at different levels
- The ports of a module instance must match those declared
- The matching of the ports can be done **by name** or **by position**

Writing a Test Bench for the 4-bit Adder

```
module Adder4_TestBench;      // No Ports
  reg  a0, a1, a2, a3;        // variable inputs
  reg  b0, b1, b2, b3, cin;  // variable inputs
  wire s0, s1, s2, s3, cout; // wire outputs
  // Instantiate the module to be tested
  Adder4 Add4 (a0,a1,a2,a3, b0,b1,b2,b3, cin, s0,s1,s2,s3, cout);
  initial begin        // initial block
    a0=0;a1=0;a2=0;a3=0;      // at t=0
    b0=0;b1=0;b2=0;b3=0;cin=0;      // at t=0
    #100 a1=1;a3=1;b2=1;b3=1;       // at t=100
    #100 a0=1;a1=0;b1=1;b2=0;       // at t=200
    #100 a2=1;a3=0;cin=1;     // at t=300
    #100 $finish;      // at t=400 finish simulation
  end  // end of initial block
endmodule
```

Behavioural Coding Style

- **Behavior:**
  - Actions a circuit is supposed to perform when it is active.

- Most advanced coding style: flexible and high-level
  - closest to programming languages
  - allows use of conditional statements, case statements, loops, etc.

Best for verification, but by no means ideal...

- **Algorithmic description:** Need ”variables” similar to PLs!
  - Abstraction of data storage elements - register objects:
    - `reg R;` one bit register - default value x before first assignment
    - `time T;` can store/manipulate simulation time
    - `integer N;` by default at least 32 bit - stores values signed
    - `real R;` default value is 0

Behavioural Constructs for Coding

- **Conditionals:**

```
if (expression true) true branch;
else false branch;
```

- **Case:**

```
    case ({_,...,_})
          pattern : ...;
          ...
          default : ...;
    endcase
```

- **Loops:** `forever, repeat, while, for`

Mux421: Behavioural Coding Example

```
module mux421_behavioural (Out, In0, In1, In2, In3, Sel0, Sel1);
  output Out;
  input In0, In1, In2, In3, Sel0, Sel1;
  reg Out;
  always @ (Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1,Sel0})
      2'b00 : Out = In0;
      2'b01 : Out = In1;
      2'b10 : Out = In2;
      2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end
endmodule // mux421_behavioural
```

Mux421: Behavioural Coding Example

```
module mux421_behavioural (Out, In0, In1, In2, In3, Sel0, Sel1);
  output Out;
  input In0, In1, In2, In3, Sel0, Sel1;
  reg Out;
  always @ (Sel1,Sel0,In0,In1,In2,In3) // Verilog 2001 style
  begin
    case ({Sel1,Sel0})
      2'b00 : Out = In0;
      2'b01 : Out = In1;
      2'b10 : Out = In2;
      2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end
endmodule // mux421_behavioural
```

Behavioural Blocks

- `initial` and `always`
  - Can't be nested.
  - Block containing several statements must be grouped using:
    - `begin ... end` (sequential)

- `initial` block:
  - Used to initialize variables (registers).
  - Executed at (simulation) time 0. Only once!

- `always` block:
  - Starts executing at time 0.
  - Contents is executed in infinite loop.
    - **Means:** Execution repeats as long as simulation is running.
  - Multiple blocks are all executed **concurrently** from time 0.

Assignment in Behavioural Coding

Assignment in behavioral coding style is **procedural**:

#5 C = #10 A+B;

- LHS (target) must be a register (reg, integer, real or time) - not a net, a bit or part of a vector of registers.
- NO **`assign`** keyword!
- Must be contained within a behavioral (i.e. `initial` or `always`) block.
- NOT always active!
  - Target register value is only changed when procedural assignment is executed according to sequence contained in block.
- **Delays:** indicate time that simulator waits from ”finding” the assignment to executing it.

Blocking Assignment

(... as opposed to continuous assignment from dataflow coding style.)

- Sequential initialization assignment.

```
reg A;
reg [7:0] Vector;
integer Count;
initial
begin
   A = 1'b0;
   Vector = 8'b0;
   Count = 0;
end
```

Timing Control Evaluation

#5 C = #10 A+B;

Assignment delay

Intra-assignment delay

1. Find procedural assignment
2. Wait 5 time units
3. Perform A+B
4. Wait 10 time units
5. Assign result to C

- So, what is the difference between:
  - `#10 C = A+B` and
  - `C = #10 A+B`?

Events and Wait

- **Events** mark changes in nets and registers, e.g. raising/falling edge of clock.
  - `@ negedge` means from any value to 0
  - `@ posedge` means from any value to 1
  - `@ clk` always activates when clock changes
- **Wait statement:**
  - `wait (condition) stmt;`
    - `wait (EN) #5 C = A + B;`
      - waits for `EN` to be 1 before `#5 C = A + B;`
- Use `wait` to block execution by not specifying a statement!
  - `wait (EN); ...`

Sensitivity List

```
always @(sensitivity list) <begin> <procedural stments> <end>

always @ (posedge Clk or EN)
begin ... end

always @ (Sel1,Sel2) // Verilog 2001 style
begin ... end
```

- Allows to **suspend `always` blocks.**
- Block executes and suspends until signal (one or more) in **sensitivity list** changes.
- NOTE: **or** is used to make statement **sensitive to multiple signals or events.**
- (Don't use sensitivity list to express a logical condition!)
- Common mistake:
  - Forgetting to add relevant signals to sensitivity list!

Non-blocking Assignments

- **Concurrency** can be introduced into sequential statements.
  - Delay is counted down before assignment,
  - BUT **control is passed to next statement immediately.**
- **Non-blocking Assignments** allow to model multiple concurrent data transfers after common event.
- A blocking assignment would force sequential execution.

`A <= #1 1; B <= #2 0;` (non-blocking)

```
    A x 1 1 1
    B x x 0 0
Time: 0 1 2 3
```

`A = #1 1; B = #2 0;` (blocking)

```
    A x 1 1 1
    B x x x 0
Time: 0 1 2 3
```

Approaches to Assignment - II

Race Condition!

```
reg [7:0] MyReg;
initial
  begin
    MyReg <= #50 8'hFF; // pass control, wait, assign
    MyReg <= #50 8'h01;
    MyReg <= #50 8'h2F;
    MyReg <= #50 8'h00;
    #250 $finish;
  end
```

- **Sequential with non-blocking assignment (<=)**

```
Time:        0   50   100 150 200 250
MyReg[7:0] XX   ??    ??  ??  ??  ??
```

Important when driving input into a DUV in a testbench!

Approaches to Assignment - III

```
reg [7:0] MyReg;
initial
  begin
    MyReg <= #50  8'hFF; // pass control, wait, assign
    MyReg <= #100 8'h01;
    MyReg <= #150 8'h2F;
    MyReg <= #200 8'h00;
    #250 $finish;
  end
```

- **Sequential with non-blocking assignment (<=)**

```
Time:        0  50 100 150 200 250
MyReg[7:0] XX FF 01  2F  00  00
```

Important when driving input into a DUV in a testbench!

Approaches to Assignment - IV

```
reg [7:0] MyReg;
initial
  begin
    #50 MyReg = 8'hFF; // wait, assign, pass control
    #50 MyReg = 8'h01;
    #50 MyReg = 8'h2F;
    #50 MyReg = 8'h00;
    #250 $finish;
  end
```

- **Sequential with blocking assignment (=)**

```
Time:        0  50 100 150 200 250
MyReg[7:0] XX FF 01  2F  00  00
```

Important when driving input into a DUV in a testbench!

Verilog Coding Guidelines

1. When modeling **combinational** logic, use **blocking** assignments
2. When modeling **sequential** logic, use **non-blocking** assignments
3. When modeling **both sequential** and **combinational logic** within the same always block, use **non-blocking** assignments
4. Do **NOT** mix blocking with non-blocking assignments in the same always block
5. Do **NOT** make assignments to the same variable from more than one always block

Tasks and Functions

- Both are **purely behavioural.**
  - Can’t define nets inside them.
  - Can use logical variables, registers, integers and reals.
- **Must be declared within a module.**
  - Are local to this module.
  - To share tasks/functions in several modules, specify declaration in separate module and use **‘include** directive.
- **Timing** (simulation time)
  - **Tasks:**
    - No restriction on use of timing; engineer specifies execution.
  - **Functions:**
    - Execute in ZERO sim time units; no timing/event control allowed.

Task Example

```
task factorial;
  output [31:0] f;
  input [3:0] n;
  integer count; // local variable
  begin
    f = 1;
    for (count=n; count>0; count=count-1)
      f = f * count;
  end
endtask
```

- **Invoke task:** `< task name > (list of arguments);`
  - Declaration order determines order of arguments when task is called!

Function Example

```
function ParityCheck;
  input [3:0] Data;
  begin
    ParityCheck = ^Data; // bit-wise xor reduction
  end
endfunction
```

- **Result** is by default a 1 bit register assigned to implicitly declared local variable that has same name as function.

- **Function calls:**
  - Are either assigned to a variable, or
  - occur in an expression that is assigned to a variable,
  - or occur as an argument of another function call.

Comparing Tasks with Functions

	Tasks	Functions
Timing	can be non-zero sim time	execute in 0 sim time
Calling other tasks or functions	no limit; may enable functions	may not call tasks but may call another function **No recursion!**
Arguments	any number; any type; can't return result	at least one input; no output/inout; always results in single return value
Purpose	modularize code	react to some input with single response; only combinatorial code; use as operands in expressions

System Tasks and Functions

- More than 100 Verilog system tasks/functions.
- Can be used in any module without explicit include directive.
- **Syntax: `$`**`< keyword >`
- **Most important tasks for verification:**
  - **`$display, $monitor`**
  - **`$time, $stop, $finish`**
  - (Also with files: **`$fopen, $fdisplay`**)

Summary

- **Verilog HDL IEEE Standard 1364-2001**
  - Signals: internal and external (ports)
  - Different coding styles:
    - structural
    - dataflow
    - **behavioural**
- SystemVerilog builds on IEEE 1364-2005
- HDLs: **Connectivity, Time and Concurrency**