STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

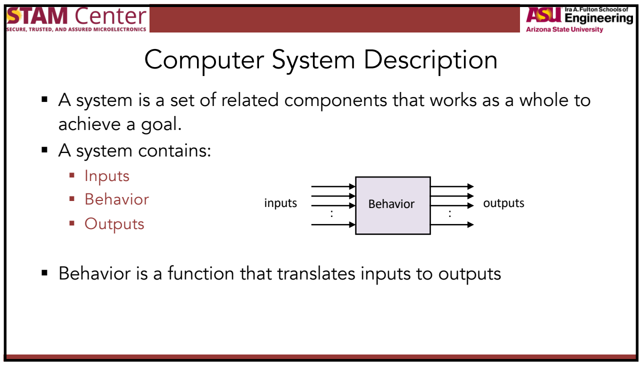
Ira A. Fulton Schools of Engineering
Arizona State University

CSE/CEN 598
Hardware Security & Trust

Trusted Digital System Design:
Verilog Fundamentals I

Prof. Michel A. Kinsy & Mishel Paul

1



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

Ira A. Fulton Schools of Engineering
Arizona State University

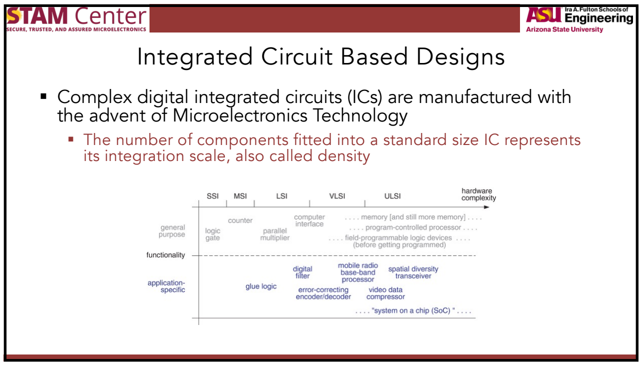
Computer System Description

- A system is a set of related components that works as a whole to achieve a goal.
- A system contains:
  - Inputs
  - Behavior
  - Outputs

inputs
Behavior
outputs

- Behavior is a function that translates inputs to outputs

2



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

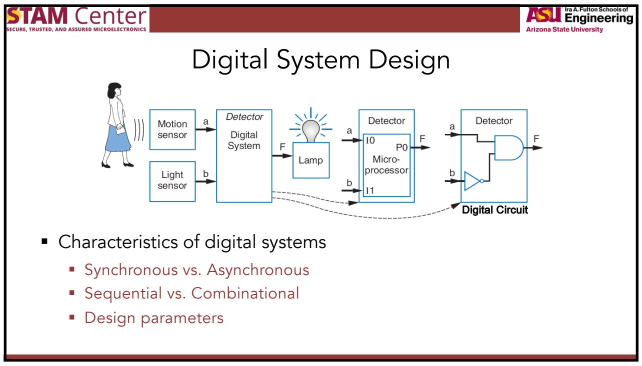
Ira A. Fulton Schools of Engineering
Arizona State University

Integrated Circuit Based Designs

- Complex digital integrated circuits (ICs) are manufactured with the advent of Microelectronics Technology
  - The number of components fitted into a standard size IC represents its integration scale, also called density

SSI
MSI
LSI
VLSI
ULSI
hardware complexity
general purpose
logic gate
counter
parallel multiplier
computer interface
. . . . memory [and still more memory] . . . .
. . . . program-controlled processor . . . .
. . . . field-programmable logic devices (before getting programmed) . . . .
functionality
application-specific
glue logic
digital filter
error-correcting encoder/decoder
mobile radio base-band processor
video data compressor
spatial diversity transceiver
. . . . "system on a chip (SoC) " . . . .

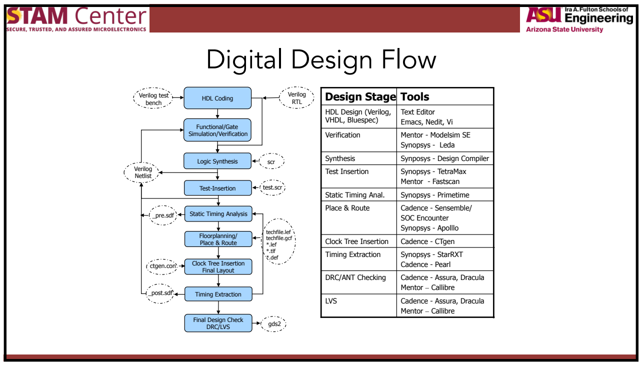
3



Digital System Design

- Characteristics of digital systems
  - Synchronous vs. Asynchronous
  - Sequential vs. Combinational
  - Design parameters

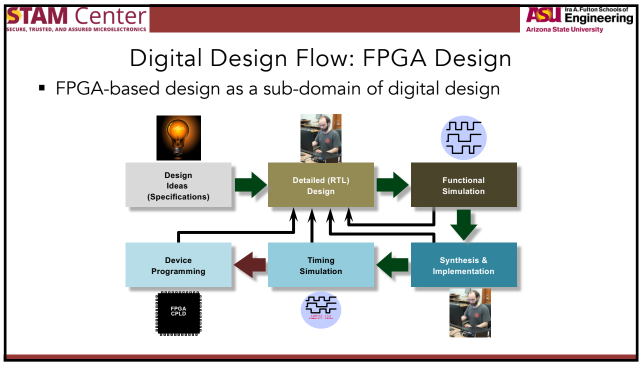
4



Digital Design Flow

Design Stage	Tools
HDL Design (Verilog, VHDL, Bluespec)	Text Editor Emacs, Nedit, Vi
Verification	Mentor - Modelsim SE Synopsys - Leda
Synthesis	Synposys - Design Compiler
Test Insertion	Synopsys - TetraMax Mentor - Fastscan
Static Timing Anal.	Synopsys - Primetime
Place & Route	Cadence - Sensemble/ SOC Encounter Synopsys - Apolllo
Clock Tree Insertion	Cadence - CTgen
Timing Extraction	Synopsys - StarRXT Cadence - Pearl
DRC/ANT Checking	Cadence - Assura, Dracula Mentor – Callibre
LVS	Cadence - Assura, Dracula Mentor – Callibre

5



Digital Design Flow: FPGA Design

- FPGA-based design as a sub-domain of digital design

6

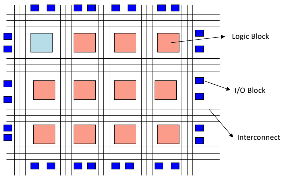
RTL Design Tools

- In this class, we will learn the principles of RTL (register level transfer) coding for synthesis tools through the Verilog hardware description language (HDL) for the design and documentation of out electronic systems.
  - Verilog allows designers to design at various levels of abstraction.
  - It is the most widely used HDL

7

Programmable Logics

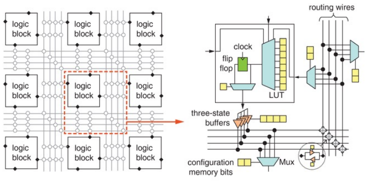
- Field Programmable Gate Arrays (more on it later)
  - Each cell in array contains a programmable logic function



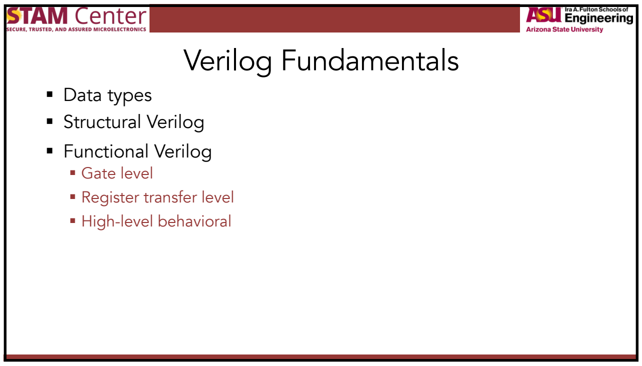
8

Programmable Logics

- Field Programmable Gate Arrays (more on it later)
  - Array has programmable interconnect between logic functions



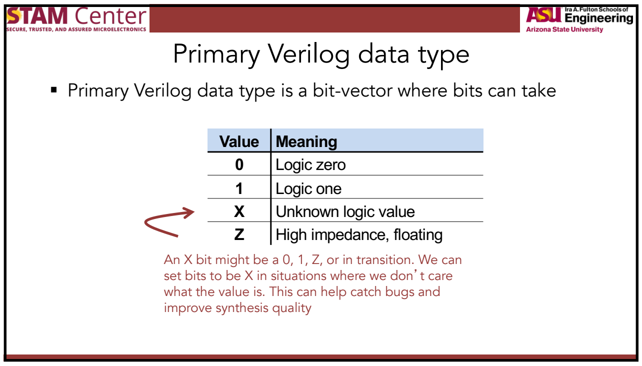
9



Verilog Fundamentals

- Data types
- Structural Verilog
- Functional Verilog
  - Gate level
  - Register transfer level
  - High-level behavioral

10



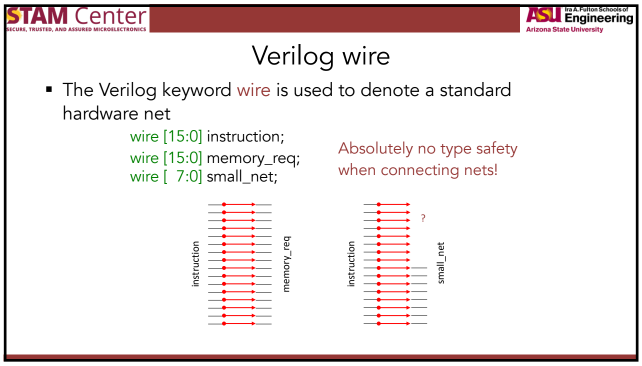
Primary Verilog data type

- Primary Verilog data type is a bit-vector where bits can take

Value	Meaning
0	Logic zero
1	Logic one
X	Unknown logic value
Z	High impedance, floating

An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’ t care what the value is. This can help catch bugs and improve synthesis quality

11



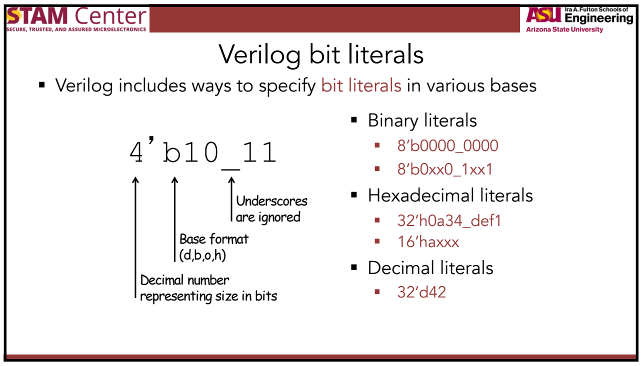
Verilog wire

- The Verilog keyword wire is used to denote a standard hardware net

```
wire [15:0] instruction;
wire [15:0] memory_req;
wire [ 7:0] small_net;
```

Absolutely no type safety when connecting nets!

12



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

Ira A. Fulton Schools of Engineering
Arizona State University

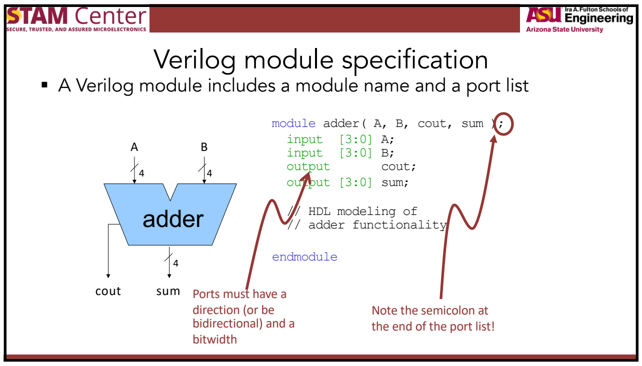
Verilog bit literals

- Verilog includes ways to specify bit literals in various bases

4'b10_11
Underscores are ignored
Base format (d,b,o,h)
Decimal number representing size in bits

- Binary literals
  - 8'b0000_0000
  - 8'b0xx0_1xx1
- Hexadecimal literals
  - 32'h0a34_def1
  - 16'haxxx
- Decimal literals
  - 32'd42

13



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

Ira A. Fulton Schools of Engineering
Arizona State University

Verilog module specification

- A Verilog module includes a module name and a port list

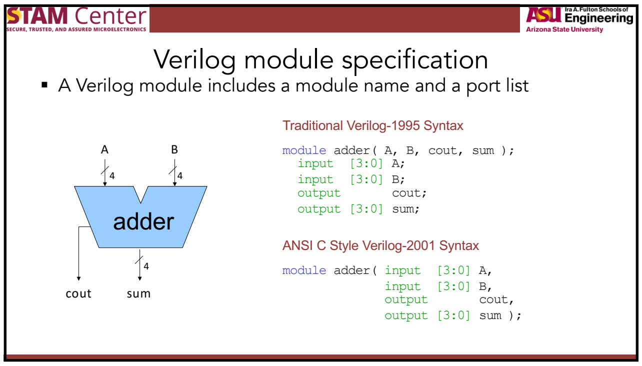
A
B
4
4
adder
4
cout
sum

```
module adder( A, B, cout, sum );
  input  [3:0] A;
  input  [3:0] B;
  output       cout;
  output [3:0] sum;

  // HDL modeling of
  // adder functionality

endmodule
```

Ports must have a direction (or be bidirectional) and a bitwidth

Note the semicolon at the end of the port list!

14



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

Ira A. Fulton Schools of Engineering
Arizona State University

Verilog module specification

- A Verilog module includes a module name and a port list

A
B
4
4
adder
4
cout
sum

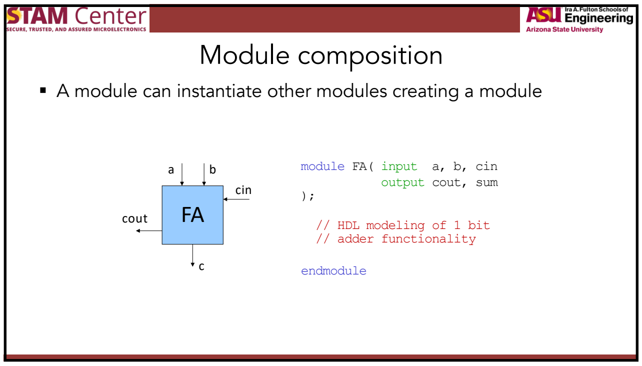
Traditional Verilog-1995 Syntax

```
module adder( A, B, cout, sum );
  input  [3:0] A;
  input  [3:0] B;
  output       cout;
  output [3:0] sum;
```

ANSI C Style Verilog-2001 Syntax

```
module adder( input  [3:0] A,
              input  [3:0] B,
              output       cout,
              output [3:0] sum );
```

15

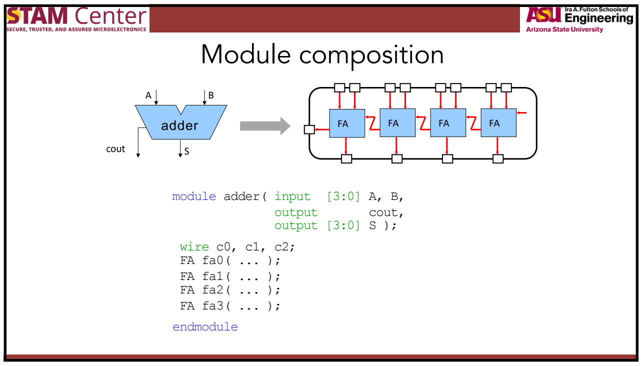


Module composition

- A module can instantiate other modules creating a module

```
module FA( input  a, b, cin
           output cout, sum
);

  // HDL modeling of 1 bit
  // adder functionality

endmodule
```

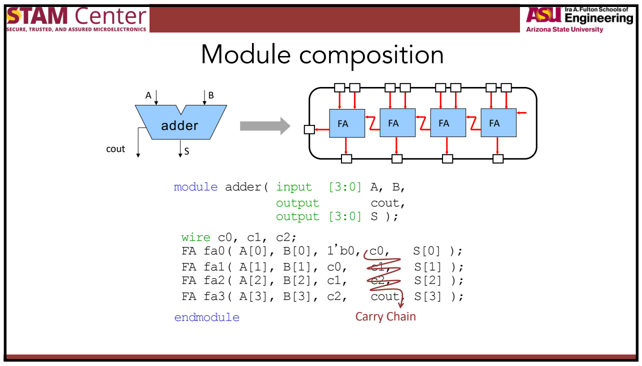
16



Module composition

```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( ... );
 FA fa1( ... );
 FA fa2( ... );
 FA fa3( ... );
endmodule
```

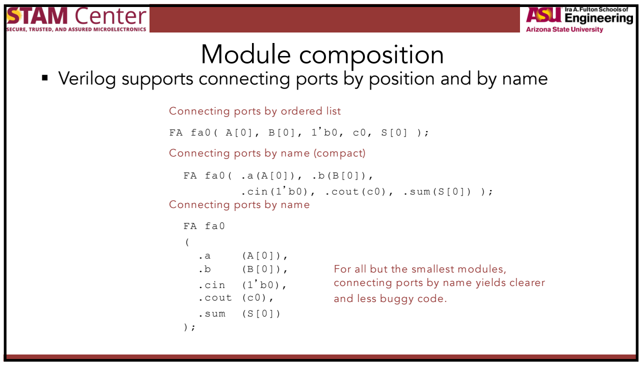
17



Module composition

```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
 FA fa1( A[1], B[1], c0,   c1,   S[1] );
 FA fa2( A[2], B[2], c1,   c2,   S[2] );
 FA fa3( A[3], B[3], c2,   cout, S[3] );
endmodule
```

18



Module composition

- Verilog supports connecting ports by position and by name

Connecting ports by ordered list

```
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
```

Connecting ports by name (compact)

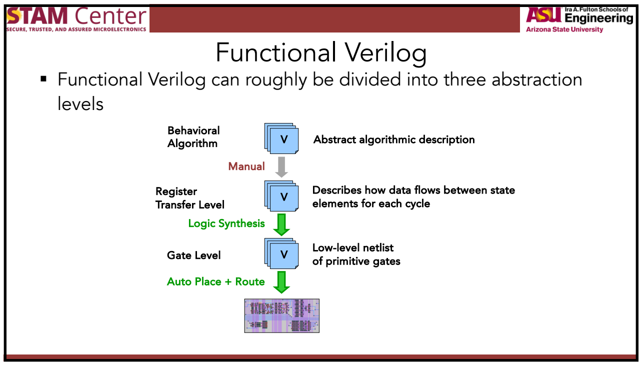
```
FA fa0( .a(A[0]), .b(B[0]),
        .cin(1'b0), .cout(c0), .sum(S[0]) );
```

Connecting ports by name

```
FA fa0
(
  .a    (A[0]),
  .b    (B[0]),
  .cin  (1'b0),
  .cout (c0),
  .sum  (S[0])
);
```

For all but the smallest modules, connecting ports by name yields clearer and less buggy code.

19

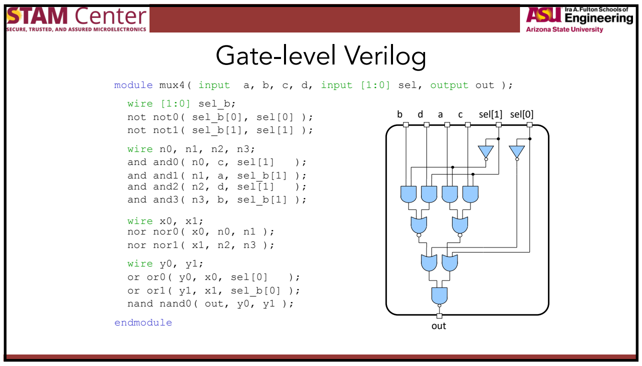


Functional Verilog

- Functional Verilog can roughly be divided into three abstraction levels

Behavioral Algorithm
V
Abstract algorithmic description
Manual
Register Transfer Level
V
Describes how data flows between state elements for each cycle
Logic Synthesis
Gate Level
V
Low-level netlist of primitive gates
Auto Place + Route

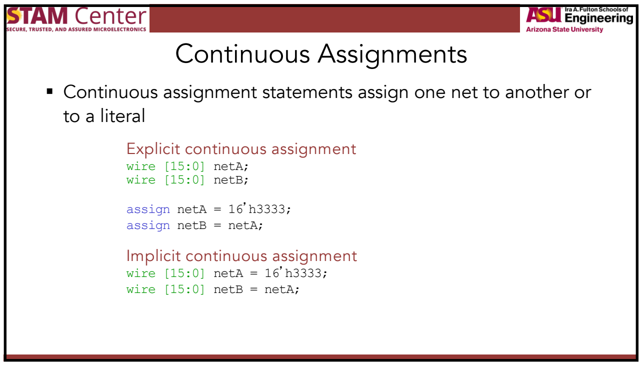
20



Gate-level Verilog

```
module mux4( input  a, b, c, d, input [1:0] sel, output out );

  wire [1:0] sel_b;
  not not0( sel_b[0], sel[0] );
  not not1( sel_b[1], sel[1] );

  wire n0, n1, n2, n3;
  and and0( n0, c, sel[1]   );
  and and1( n1, a, sel_b[1] );
  and and2( n2, d, sel[1]   );
  and and3( n3, b, sel_b[1] );

  wire x0, x1;
  nor nor0( x0, n0, n1 );
  nor nor1( x1, n2, n3 );

  wire y0, y1;
  or or0( y0, x0, sel[0]   );
  or or1( y1, x1, sel_b[0] );
  nand nand0( out, y0, y1 );

endmodule
```

21



Continuous Assignments

- Continuous assignment statements assign one net to another or to a literal

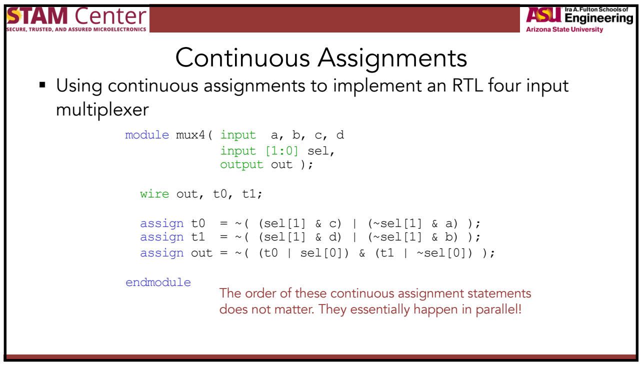
Explicit continuous assignment

```
wire [15:0] netA;
wire [15:0] netB;

assign netA = 16'h3333;
assign netB = netA;
```

Implicit continuous assignment

```
wire [15:0] netA = 16'h3333;
wire [15:0] netB = netA;
```

22



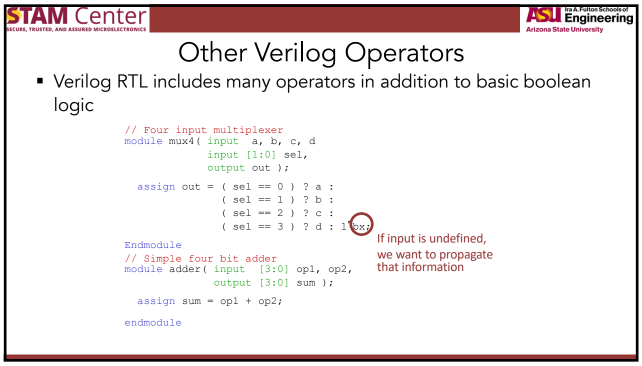
Continuous Assignments

- Using continuous assignments to implement an RTL four input multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire out, t0, t1;

  assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

The order of these continuous assignment statements does not matter. They essentially happen in parallel!

23



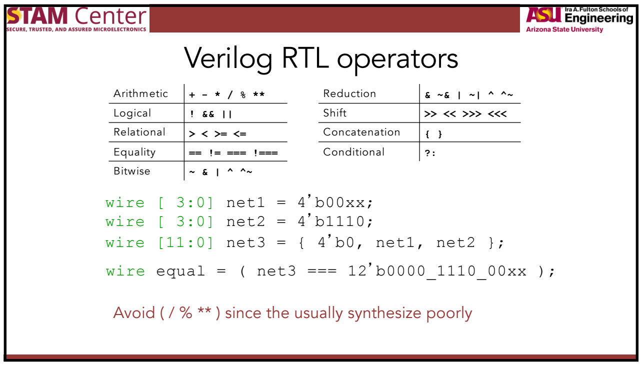
Other Verilog Operators

- Verilog RTL includes many operators in addition to basic boolean logic

```
// Four input multiplexer
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  assign out = ( sel == 0 ) ? a :
               ( sel == 1 ) ? b :
               ( sel == 2 ) ? c :
               ( sel == 3 ) ? d : 1'bx;

Endmodule
// Simple four bit adder
module adder( input  [3:0] op1, op2,
              output [3:0] sum );

  assign sum = op1 + op2;

endmodule
```

If input is undefined, we want to propagate that information

24



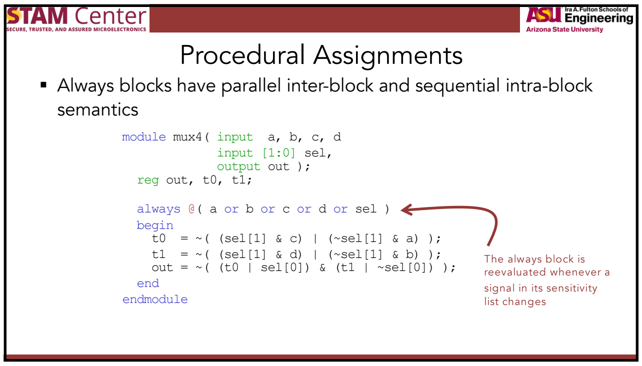
Verilog RTL operators

Arithmetic	+ - * / % **	Reduction	& ~& | ~| ^ ^~
Logical	! && ||	Shift	>> << >>> <<<
Relational	> < >= <=	Concatenation	{ }
Equality	== != === !===	Conditional	?:
Bitwise	~ & | ^ ^~		

```
wire [ 3:0] net1 = 4’b00xx;
wire [ 3:0] net2 = 4’b1110;
wire [11:0] net3 = { 4’b0, net1, net2 };
wire equal = ( net3 === 12’b0000_1110_00xx );
```

Avoid (/ % **) since the usually synthesize poorly

25



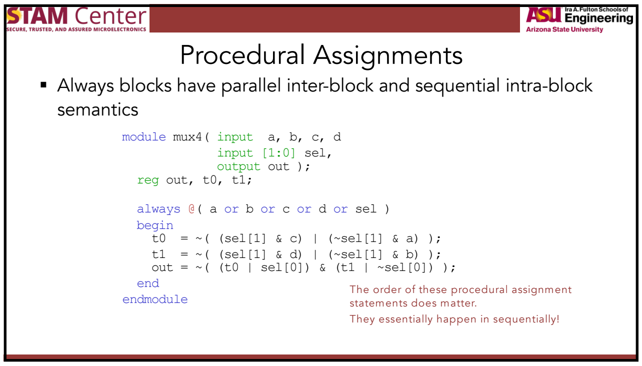
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
  reg out, t0, t1;

  always @( a or b or c or d or sel )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end
endmodule
```

The always block is reevaluated whenever a signal in its sensitivity list changes

26



Procedural Assignments

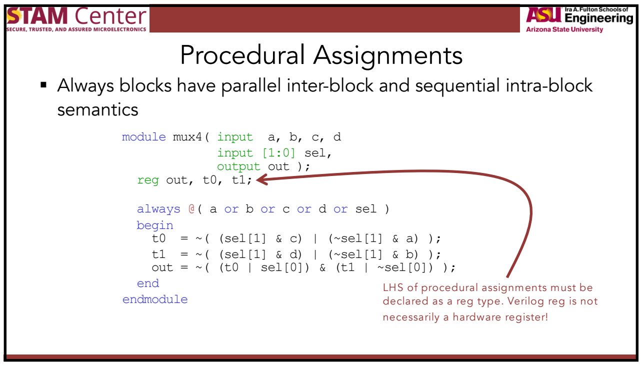
- Always blocks have parallel inter-block and sequential intra-block semantics

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
  reg out, t0, t1;

  always @( a or b or c or d or sel )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end
endmodule
```

The order of these procedural assignment statements does matter.

They essentially happen in sequentially!

27



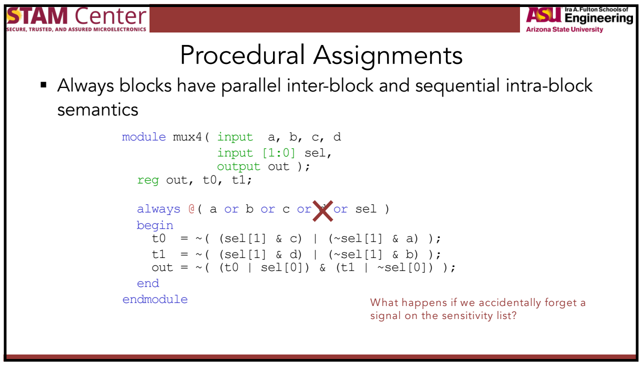
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
  reg out, t0, t1;

  always @( a or b or c or d or sel )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end
endmodule
```

LHS of procedural assignments must be declared as a reg type. Verilog reg is not necessarily a hardware register!

28



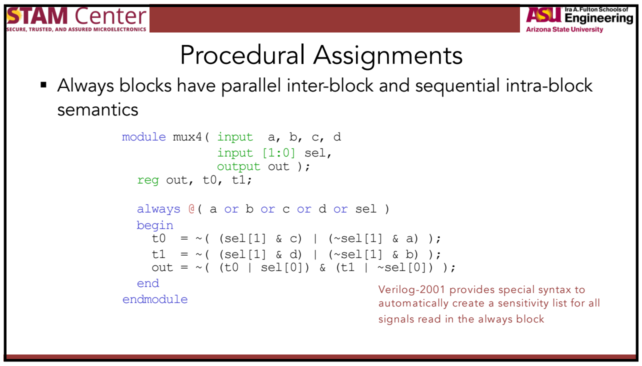
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
  reg out, t0, t1;

  always @( a or b or c or ~~d~~ or sel )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end
endmodule
```

What happens if we accidentally forget a signal on the sensitivity list?

29



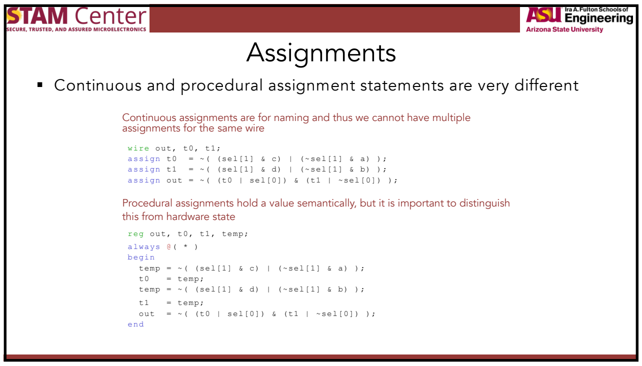
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
  reg out, t0, t1;

  always @( a or b or c or d or sel )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end
endmodule
```

Verilog-2001 provides special syntax to automatically create a sensitivity list for all signals read in the always block

30



Assignments

- Continuous and procedural assignment statements are very different

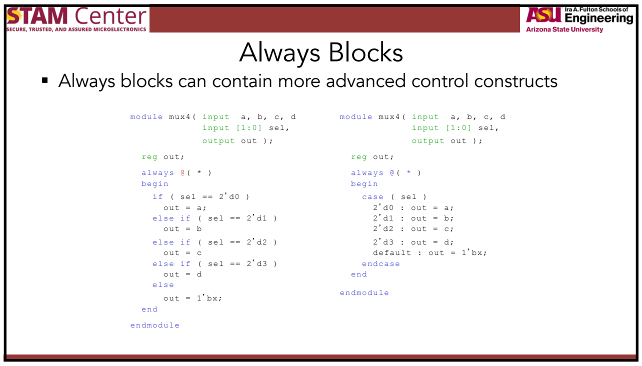
Continuous assignments are for naming and thus we cannot have multiple assignments for the same wire

```
wire out, t0, t1;
assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );
assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
```

Procedural assignments hold a value semantically, but it is important to distinguish this from hardware state

```
reg out, t0, t1, temp;
always @( * )
begin
  temp = ~( (sel[1] & c) | (~sel[1] & a) );
  t0   = temp;
  temp = ~( (sel[1] & d) | (~sel[1] & b) );
  t1   = temp;
  out  = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
```

31



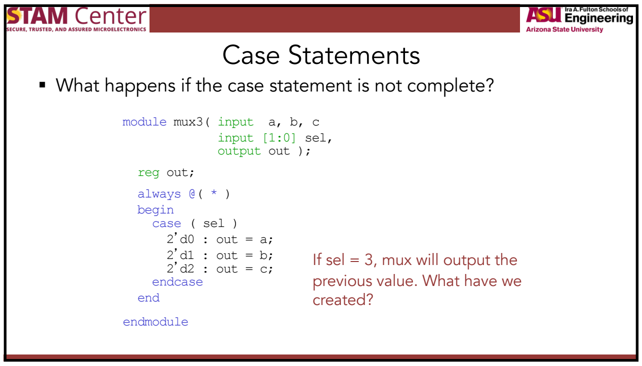
Always Blocks

- Always blocks can contain more advanced control constructs

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    if ( sel == 2'd0 )
      out = a;
    else if ( sel == 2'd1 )
      out = b
    else if ( sel == 2'd2 )
      out = c
    else if ( sel == 2'd3 )
      out = d
    else
      out = 1'bx;
  end

endmodule
```

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2'd0 : out = a;
      2'd1 : out = b;
      2'd2 : out = c;
      2'd3 : out = d;
      default : out = 1'bx;
    endcase
  end

endmodule
```

32



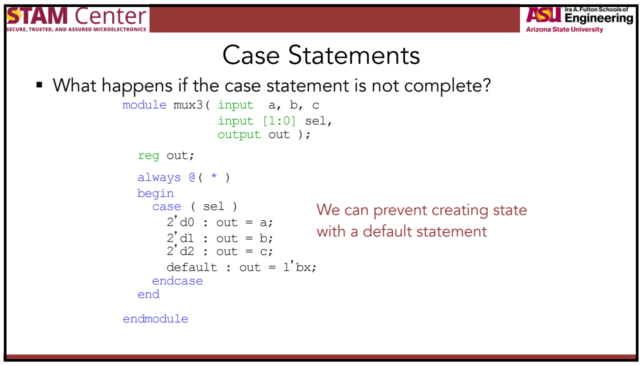
Case Statements

- What happens if the case statement is not complete?

```
module mux3( input  a, b, c
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2'd0 : out = a;
      2'd1 : out = b;
      2'd2 : out = c;
    endcase
  end

endmodule
```

If sel = 3, mux will output the previous value. What have we created?

33



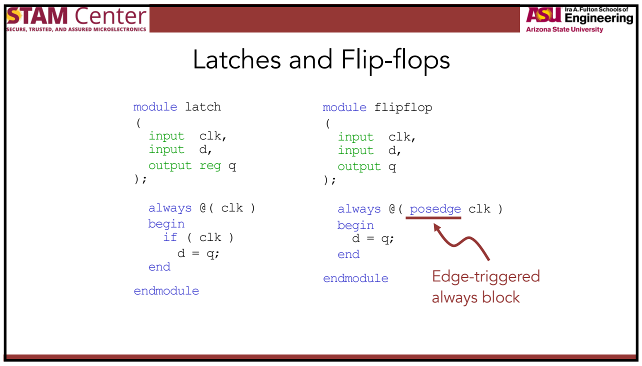
Case Statements

- What happens if the case statement is not complete?

```
module mux3( input  a, b, c
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2'd0 : out = a;
      2'd1 : out = b;
      2'd2 : out = c;
      default : out = 1'bx;
    endcase
  end

endmodule
```

We can prevent creating state with a default statement

34



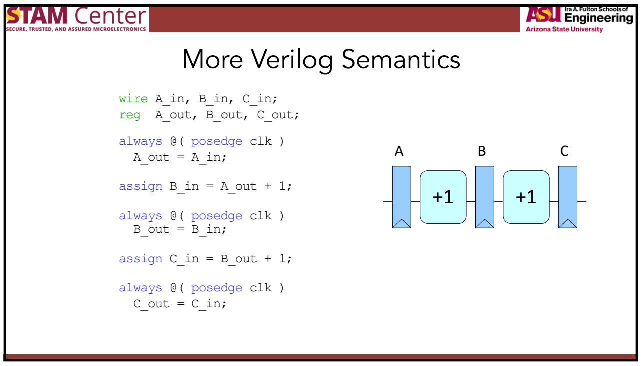
Latches and Flip-flops

```
module latch
(
  input  clk,
  input  d,
  output reg q
);

  always @( clk )
  begin
    if ( clk )
      d = q;
  end

endmodule
```

```
module flipflop
(
  input  clk,
  input  d,
  output q
);

  always @( posedge clk )
  begin
    d = q;
  end

endmodule
```

Edge-triggered always block

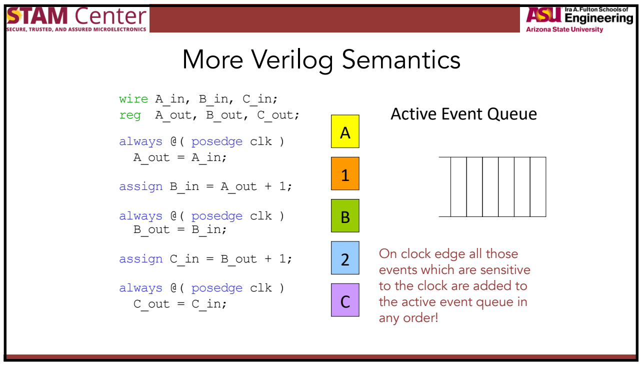
35



More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

36



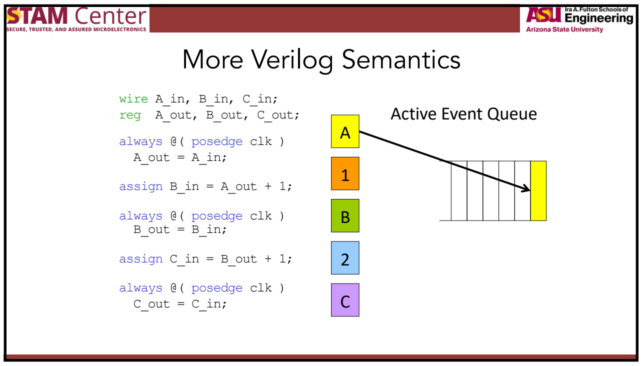
More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!

37

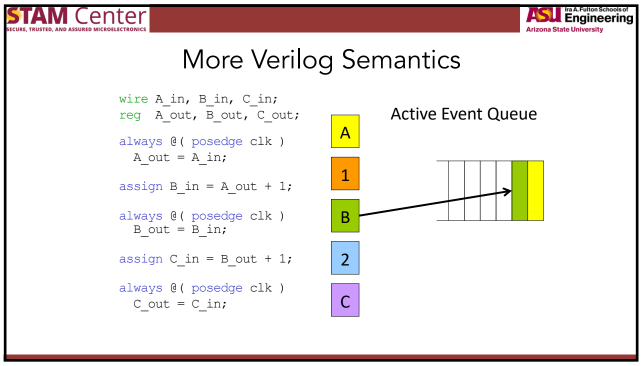


More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

38

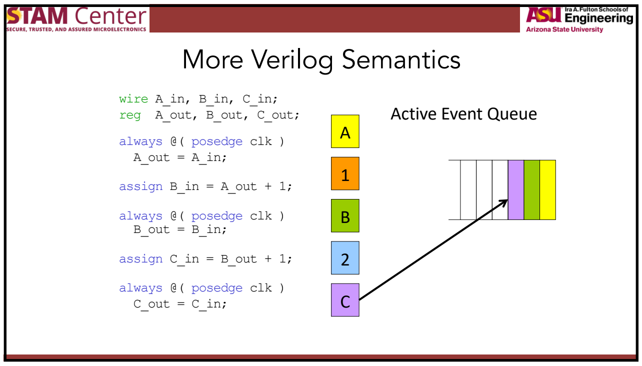


More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

39

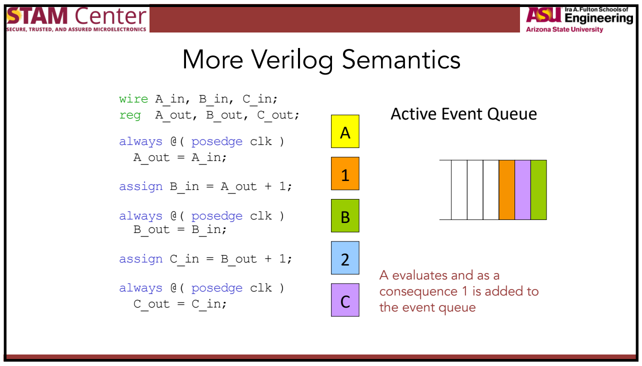


More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

40



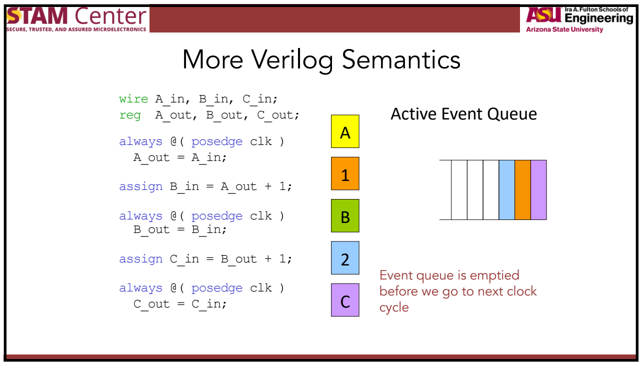
More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

A evaluates and as a consequence 1 is added to the event queue

41



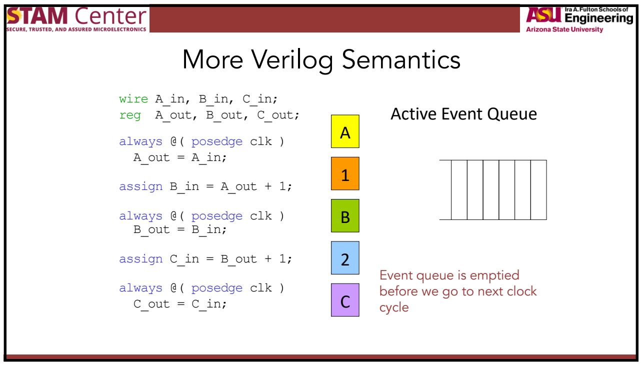
More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

Event queue is emptied before we go to next clock cycle

42



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS
ASU Ira A. Fulton Schools of Engineering
Arizona State University

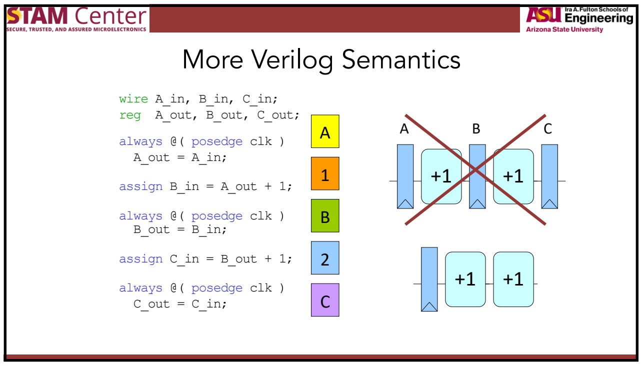
More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

Active Event Queue

Event queue is emptied before we go to next clock cycle

43



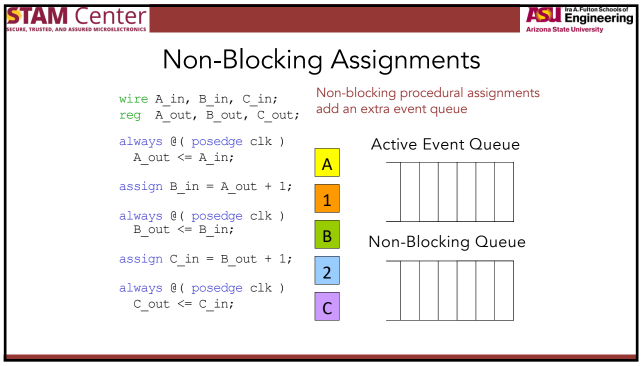
STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS
ASU Ira A. Fulton Schools of Engineering
Arizona State University

More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

A
1
B
2
C
A
B
C
+1
+1
+1
+1

44



STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS
ASU Ira A. Fulton Schools of Engineering
Arizona State University

Non-Blocking Assignments

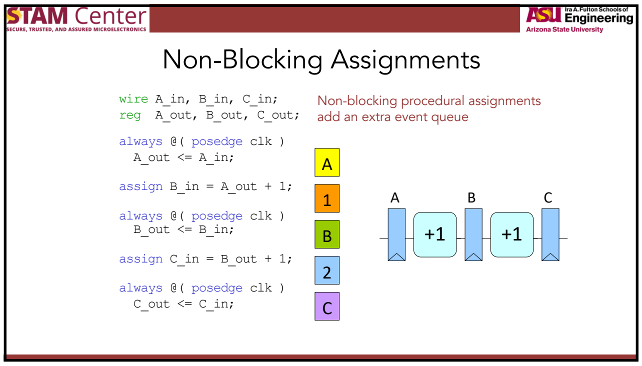
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```

Non-blocking procedural assignments add an extra event queue

Active Event Queue

Non-Blocking Queue

45

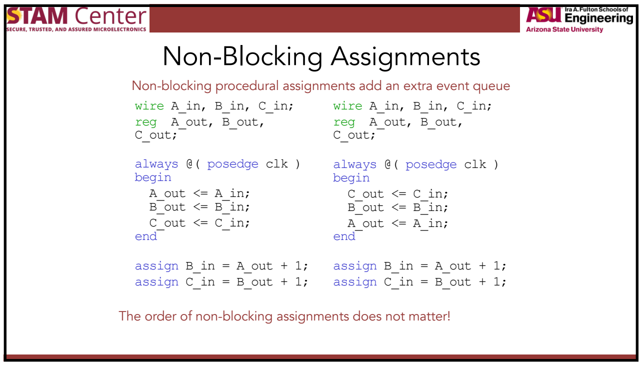

STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS
Ira A. Fulton Schools of Engineering
Arizona State University
A
1
B
2
C
A
B
C
+1
+1

Non-Blocking Assignments

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```

Non-blocking procedural assignments add an extra event queue

46


STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS
Ira A. Fulton Schools of Engineering
Arizona State University

Non-Blocking Assignments

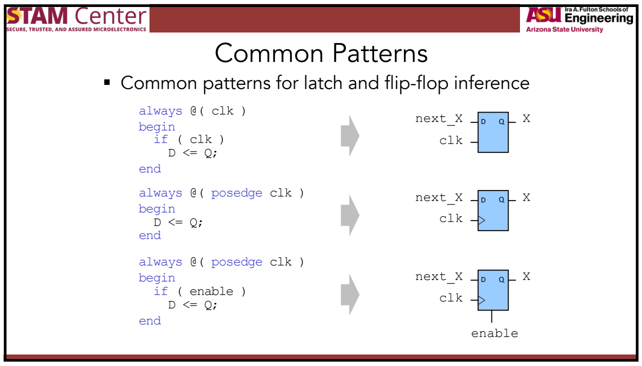
Non-blocking procedural assignments add an extra event queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out,
C_out;

always @( posedge clk )
begin
  A_out <= A_in;
  B_out <= B_in;
  C_out <= C_in;
end

assign B_in = A_out + 1;
assign C_in = B_out + 1;
```

```
wire A_in, B_in, C_in;
reg  A_out, B_out,
C_out;

always @( posedge clk )
begin
  C_out <= C_in;
  B_out <= B_in;
  A_out <= A_in;
end

assign B_in = A_out + 1;
assign C_in = B_out + 1;
```

The order of non-blocking assignments does not matter!

47


STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS
Ira A. Fulton Schools of Engineering
Arizona State University
next_X
clk
D
Q
X
next_X
clk
D
Q
X
next_X
clk
D
Q
X
enable

Common Patterns

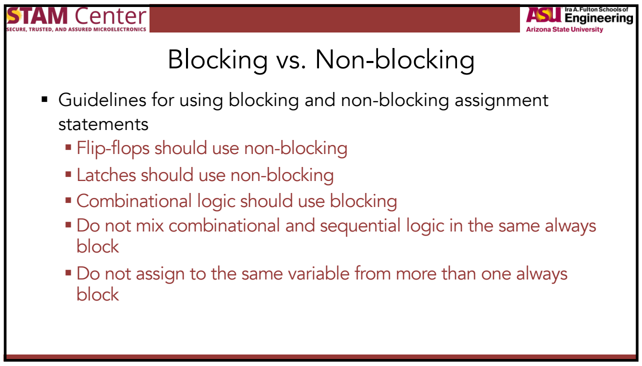
- Common patterns for latch and flip-flop inference

```
always @( clk )
begin
  if ( clk )
    D <= Q;
end
```

```
always @( posedge clk )
begin
  D <= Q;
end
```

```
always @( posedge clk )
begin
  if ( enable )
    D <= Q;
end
```

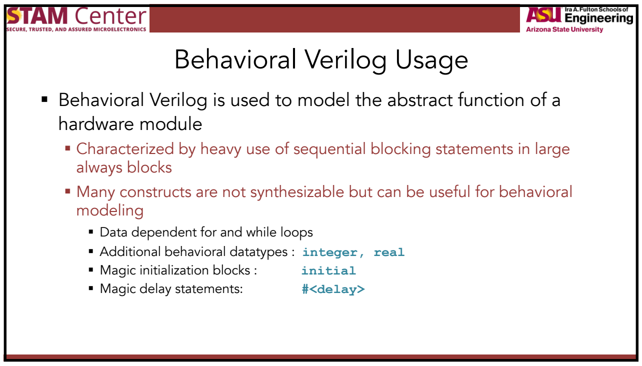
48



Blocking vs. Non-blocking

- Guidelines for using blocking and non-blocking assignment statements
  - Flip-flops should use non-blocking
  - Latches should use non-blocking
  - Combinational logic should use blocking
  - Do not mix combinational and sequential logic in the same always block
  - Do not assign to the same variable from more than one always block

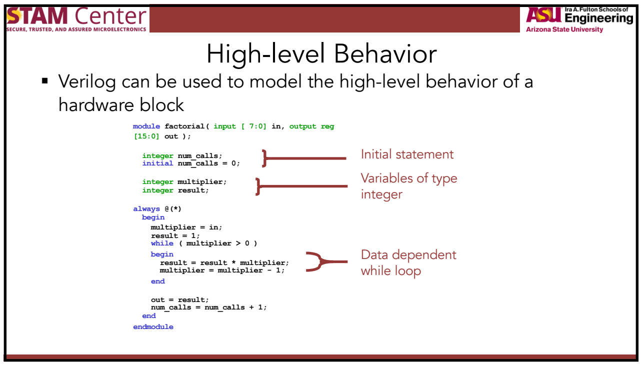
49



Behavioral Verilog Usage

- Behavioral Verilog is used to model the abstract function of a hardware module
  - Characterized by heavy use of sequential blocking statements in large always blocks
  - Many constructs are not synthesizable but can be useful for behavioral modeling
    - Data dependent for and while loops
    - Additional behavioral datatypes : `integer, real`
    - Magic initialization blocks : `initial`
    - Magic delay statements: `#<delay>`

50



High-level Behavior

- Verilog can be used to model the high-level behavior of a hardware block

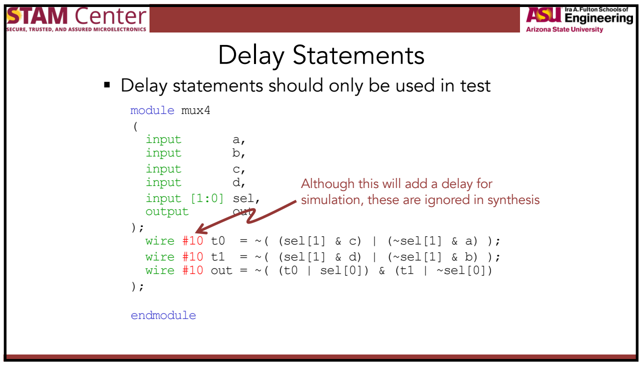
```
module factorial( input [ 7:0] in, output reg
[15:0] out );

  integer num_calls;
  initial num_calls = 0;

  integer multiplier;
  integer result;

always @(*)
  begin
    multiplier = in;
    result = 1;
    while ( multiplier > 0 )
    begin
      result = result * multiplier;
      multiplier = multiplier - 1;
    end

    out = result;
    num_calls = num_calls + 1;
  end
endmodule
```

Initial statement

Variables of type integer

Data dependent while loop

51



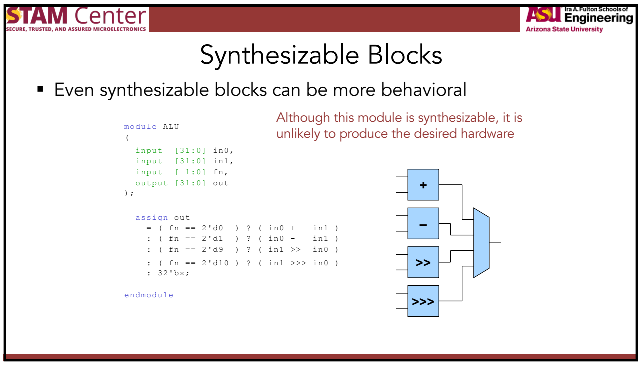
Delay Statements

- Delay statements should only be used in test

```
module mux4
(
  input       a,
  input       b,
  input       c,
  input       d,
  input [1:0] sel,
  output      out
);
  wire #10 t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  wire #10 t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  wire #10 out = ~( (t0 | sel[0]) & (t1 | ~sel[0])
);

endmodule
```

Although this will add a delay for simulation, these are ignored in synthesis

52



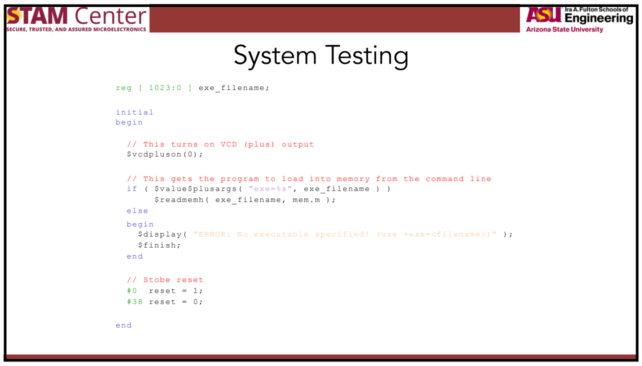
Synthesizable Blocks

- Even synthesizable blocks can be more behavioral

```
module ALU
(
  input  [31:0] in0,
  input  [31:0] in1,
  input  [ 1:0] fn,
  output [31:0] out
);

  assign out
    = ( fn == 2'd0  ) ? ( in0 +   in1 )
    : ( fn == 2'd1  ) ? ( in0 -   in1 )
    : ( fn == 2'd9  ) ? ( in1 >>  in0 )
    : ( fn == 2'd10 ) ? ( in1 >>> in0 )
    : 32'bx;

endmodule
```

Although this module is synthesizable, it is unlikely to produce the desired hardware

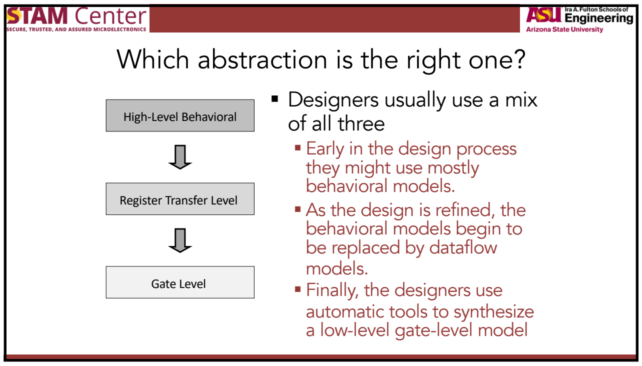
53



System Testing

```
reg [ 1023:0 ] exe_filename;

initial
begin

  // This turns on VCD (plus) output
  $vcdpluson(0);

  // This gets the program to load into memory from the command line
  if ( $value$plusargs( "exe=%s", exe_filename ) )
       $readmemh( exe_filename, mem.m );
  else
  begin
    $display( "ERROR: No executable specified! (use +exe=<filename>)" );
    $finish;
  end

  // Stobe reset
  #0  reset = 1;
  #38 reset = 0;

end
```

54



Which abstraction is the right one?

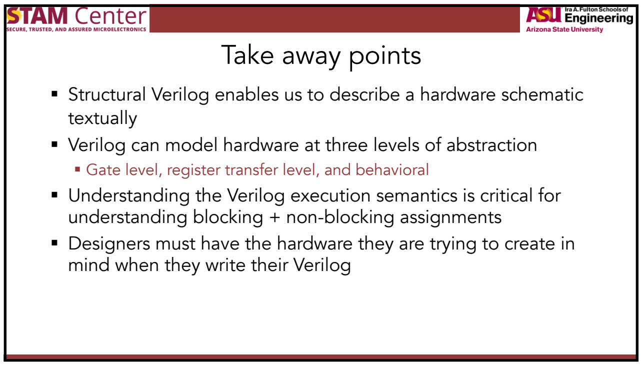
High-Level Behavioral

Register Transfer Level

Gate Level

- Designers usually use a mix of all three
  - Early in the design process they might use mostly behavioral models.
  - As the design is refined, the behavioral models begin to be replaced by dataflow models.
  - Finally, the designers use automatic tools to synthesize a low-level gate-level model

55



Take away points

- Structural Verilog enables us to describe a hardware schematic textually
- Verilog can model hardware at three levels of abstraction
  - Gate level, register transfer level, and behavioral
- Understanding the Verilog execution semantics is critical for understanding blocking + non-blocking assignments
- Designers must have the hardware they are trying to create in mind when they write their Verilog

56