ECE 327/627
Digital Hardware Systems

Lecture 2: SystemVerilog Basics I

Andrew Boutros

andrew.boutros@uwaterloo.ca

Some Logistics

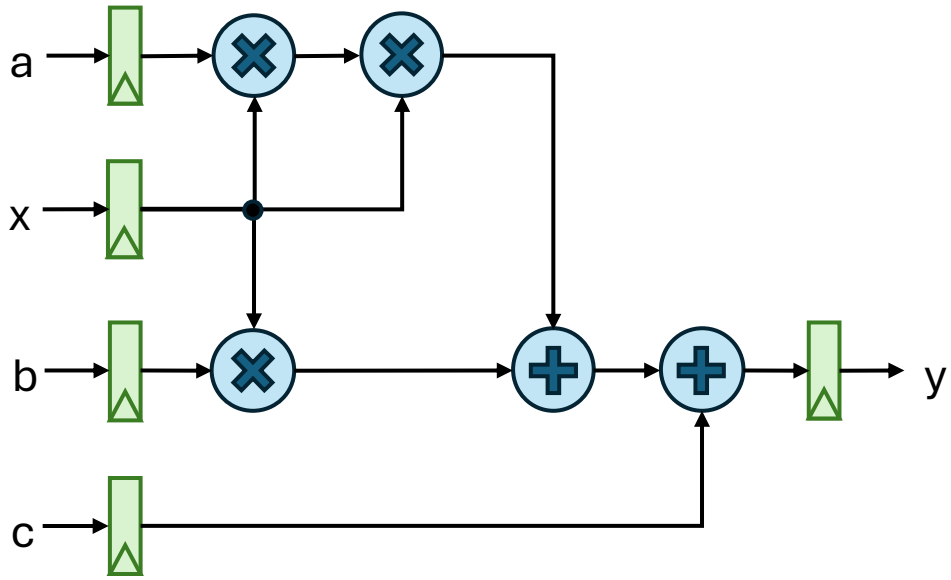
- Lab group registration is open on LEARN
  - Connect > Groups > View Categories (drop-down)
    - Lab Groups ECE327
    - Lab Groups ECE627
  - Deadline is Jan 16 @ 11:59 pm

- Lab 1 will be released tomorrow … stay tuned!

In Previous Lecture of ECE 327 …

- Dennard scaling is broken & Moore's law slowing down
- Need to think outside the box to advance computer systems
  - Architecture innovations
  - New technologies and compute paradigms
  - Domain-specific computers
- Golden time for learning how to design computer hardware!
- Different layers of abstraction when designing hardware
  - We will focus on register transfer level (RTL) design
- Hardware description languages (VHDL or Verilog) to …
  - Describe structure and behavior of circuits (synthesizable)
  - Simulate & verify functionality of circuits (non-synthesizable)
- Hardware design flow (HDL → Circuit implementation)

Example: Polynomial Circuit

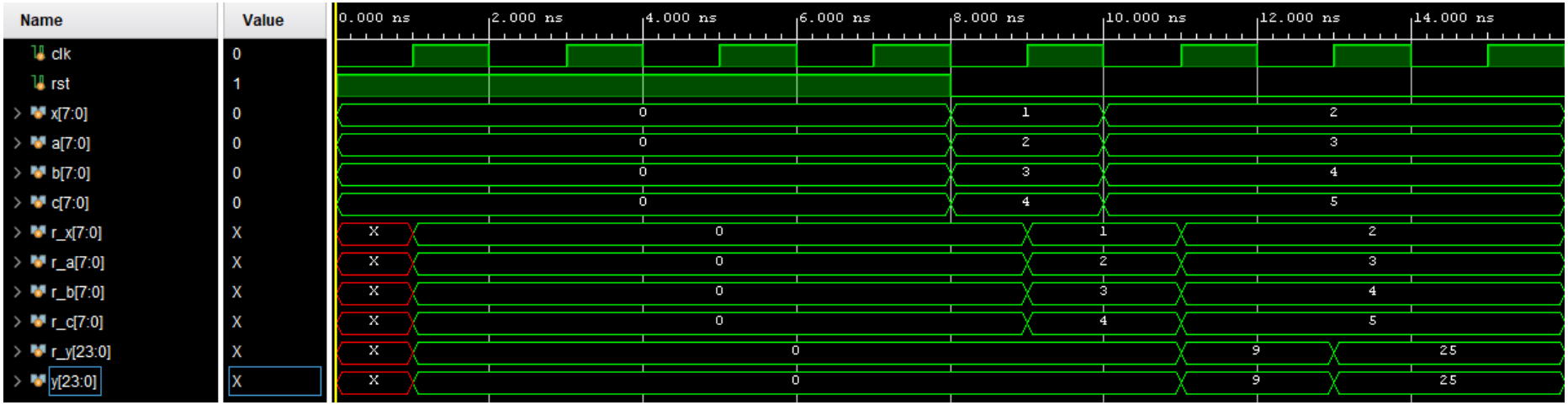



```
module poly (
     input clk,
     input rst,
     input [7:0] x,
     input [7:0] a,
     input [7:0] b,
     input [7:0] c,
     output [23:0] y
);

logic [7:0] r_x, r_a, r_b, r_c;
logic [23:0] r_y;

always_ff @(posedge clk) begin
 if(rst) begin
  r_x <= 0; r_a <= 0;
  r_b <= 0; r_c <= 0; r_y <= 0;
 end else begin
  // register inputs
  r_x <= x; r_a <= a;
  r_b <= b; r_c <= c;
  // register output
  r_y <= r_a*r_x*r_x + r_b*r_x + r_c;
 end
end

assign y = r_y;

endmodule
```

Example: Polynomial Circuit

Functional (Behavioral) simulation:

```
always_ff @(posedge clk) begin
 if(rst) begin
  r_x <= 0; r_a <= 0;
  r_b <= 0; r_c <= 0; r_y <= 0;
 end else begin
  // register inputs
  r_x <= x; r_a <= a;
  r_b <= b; r_c <= c;
  // register output
  r_y <= r_a*r_x*r_x + r_b*r_x + r_c;
 end
end
```

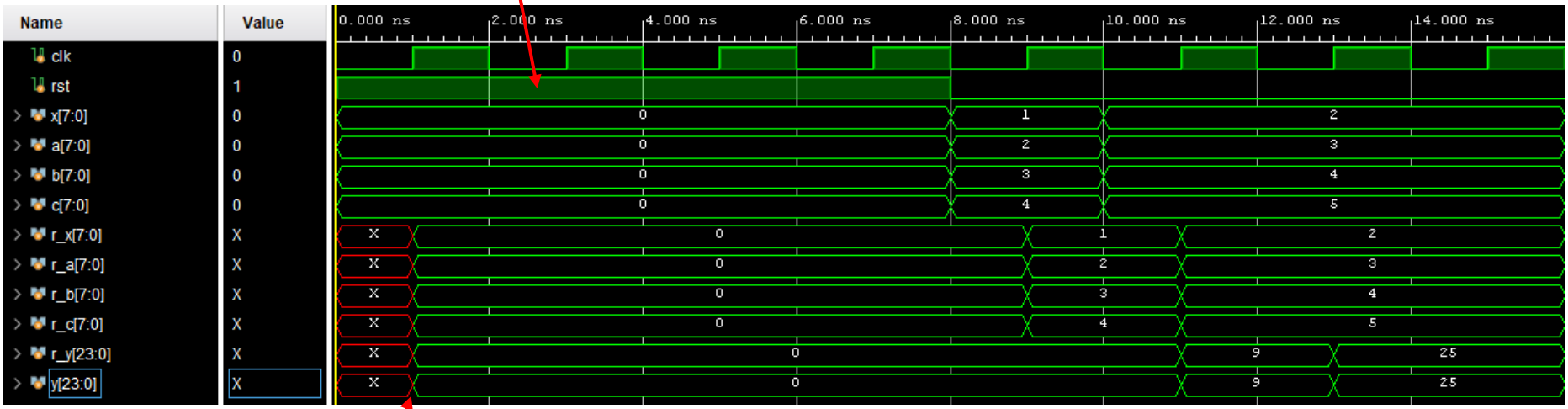


Example: Polynomial Circuit

```
always_ff @(posedge clk) begin
 if(rst) begin
  r_x <= 0; r_a <= 0;
  r_b <= 0; r_c <= 0; r_y <= 0;
 end else begin
  // register inputs
  r_x <= x; r_a <= a;
  r_b <= b; r_c <= c;
  // register output
  r_y <= r_a*r_x*r_x + r_b*r_x + r_c;
 end
end
```

Functional (Behavioral) simulation:

`rst` signal asserted for 4 cycles



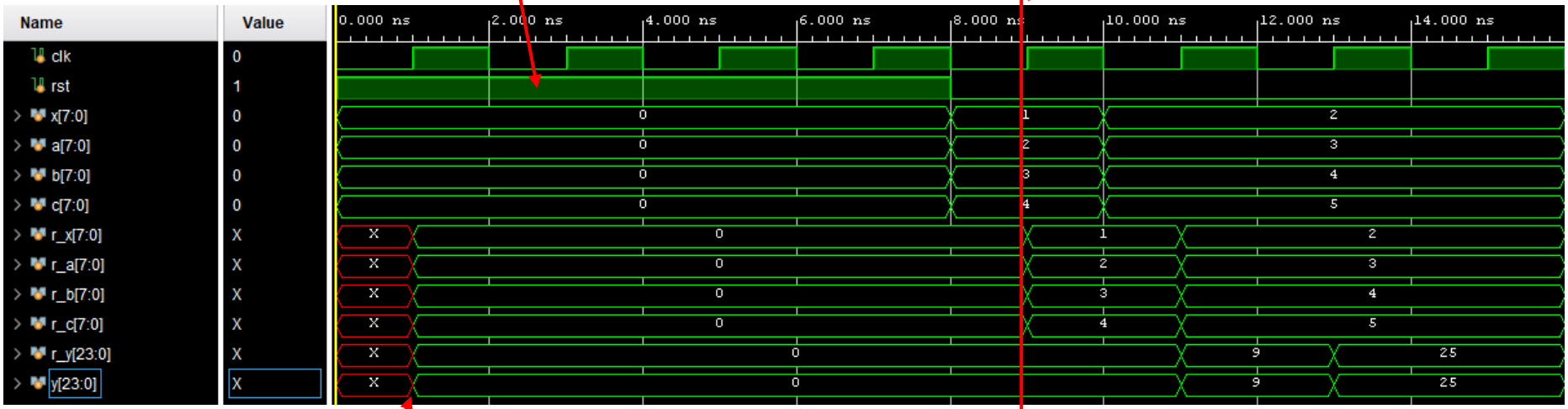
Input & Output register values are reset to zero on rising clock edge. Before that, they are undefined (X)

Example: Polynomial Circuit

Input registers capture first values on first rising clock edge after reset is low. Output register is concurrently set to result of input register values (zeros)

Functional (Behavioral) simulation:

`rst` signal asserted for 4 cycles



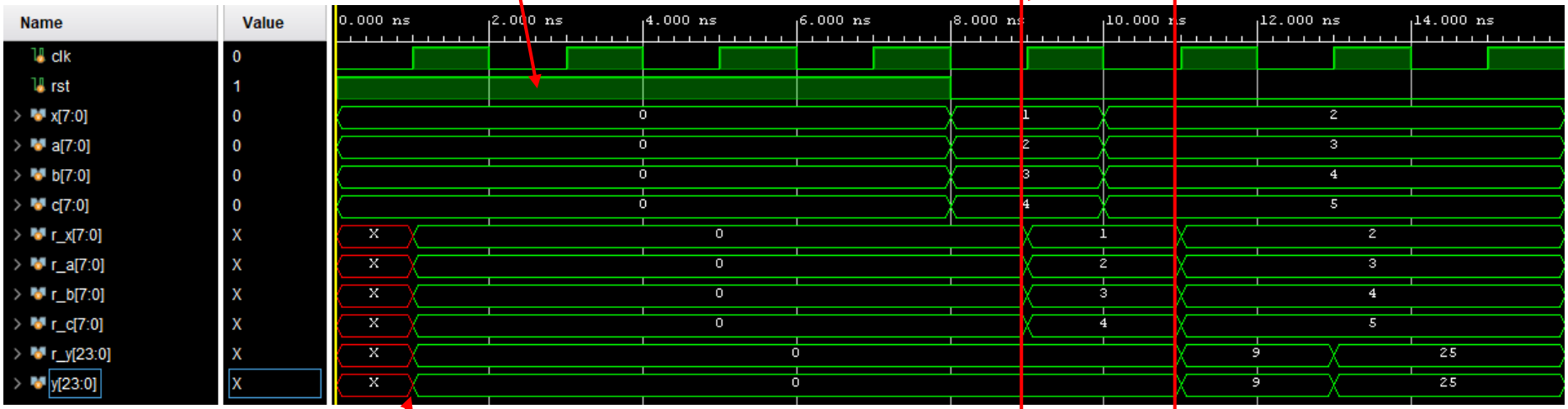
Input & Output register values are reset to zero on rising clock edge. Before that, they are undefined (X)

Example: Polynomial Circuit

Input registers capture first values on first rising clock edge after reset is low. Output register is concurrently set to result of input register values (zeros)

Functional (Behavioral) simulation:

`rst` signal asserted for 4 cycles



Input & Output register values are reset to zero on rising clock edge. Before that, they are undefined (X)

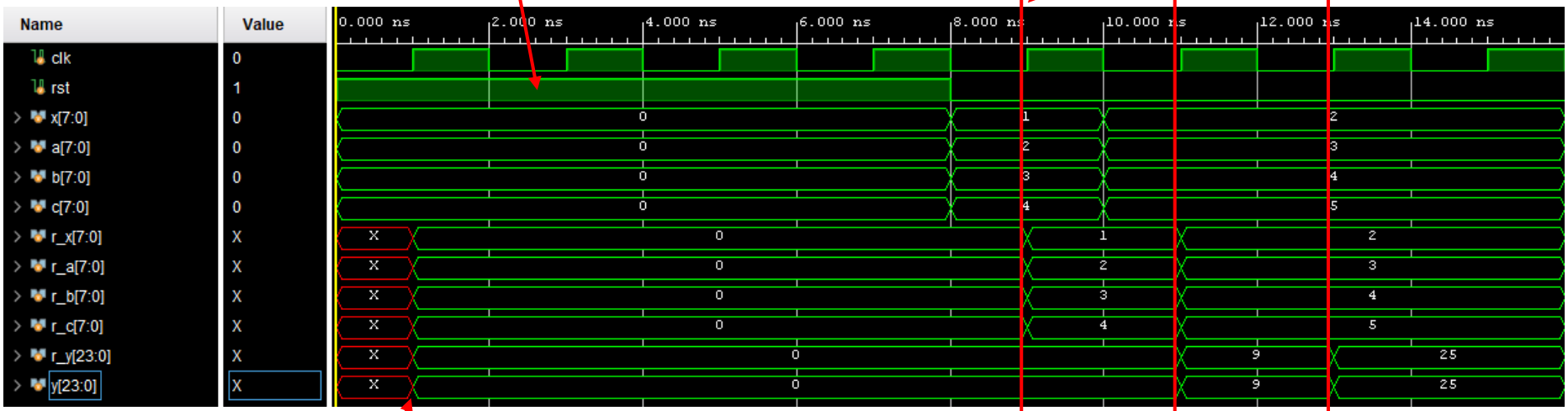
Input registers capture new values on second rising clock edge. Output register is concurrently set to result of input register values (2*1*1+3*1+4)

Example: Polynomial Circuit

Functional (Behavioral) simulation:

Input registers capture first values on first rising clock edge after reset is low. Output register is concurrently set to result of input register values (zeros)

`rst` signal asserted for 4 cycles



Input & Output register values are reset to zero on rising clock edge. Before that, they are undefined (X)

Input registers capture new values on second rising clock edge. Output register is concurrently set to result of input register values (2*1*1+3*1+4)

Output register is set to result for input register values (3*2*2+4*2+5)

Dive into More SystemVerilog Details

- But remember …
  - We will not learn all the features of the language
  - Sometimes the easiest way to know the answer for “what-if” questions is to write a toy example and synthesize/simulate it
  - Always think in hardware! HDL is not a programming language ☺

Typical SystemVerilog Implementation Skeleton

```
module <module_name> # (
    1 Module parameters
)
(
    2 Definition of module ports
);

3 Declaration of signals

4 Procedural blocks to describe some behavior of the module

5 Continuous assignments & instantiation of subcomponents

endmodule
```

Ports

- Inputs & outputs of a module
  - For top-level module = pins of the chip
  - For subcomponents = interface to other subcomponents or top-level ports
- Port type must be specified
  - `input`
  - `output`
  - `inout` (bidirectional – not common)
- For multi-bit ports (vectors), the bitwidth must be specified
  - 1-bit, if not specified

```
module adder8b (
     input  [7:0] in1,
     input  [7:0] in2,
     output [8:0] out
);



assign out = in1 + in2;

endmodule
```

Continuous Assignment

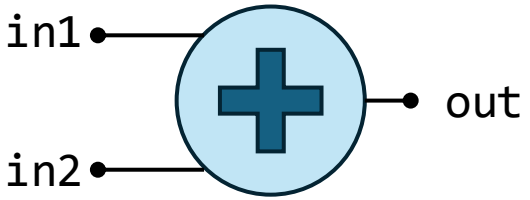
- Connects port or signal on the LHS to the output of the hardware described by the RHS expression
  - Can be as simple as a wire

```
assign a = b;
```

  - Can generate combinational circuits

```
// Arithmetic
assign a = b + c;
// Logic
assign a = (b & c) ^ (d | e);
// Conditional
assign a = (b)? c : d;
```

```
module adder8b (
     input  [7:0] in1,
     input  [7:0] in2,
     output [8:0] out
);

assign out = in1 + in2;

endmodule
```



`logic` Signals

- Signals represent a net (i.e., wire) or a register in the circuit depending on how it is assigned in the HDL code
  - Continuous assignment → net
  - Connecting subcomponents → net
  - Combinational logic → net
  - Sequential logic → register

- Example signal declarations

```
logic sel;  // 1b signal
logic [3:0] op0, op1; // two 4b signals
logic [7:0] data [0:3]; // four 8b signals
```

```
module adder8b (
     input  [7:0] in1,
     input  [7:0] in2,
     output [8:0] out
);

logic [8:0] temp;

assign temp = in1 + in2;
assign out = temp;

endmodule
```

What does that mean for the generated hardware?

`logic` Signals

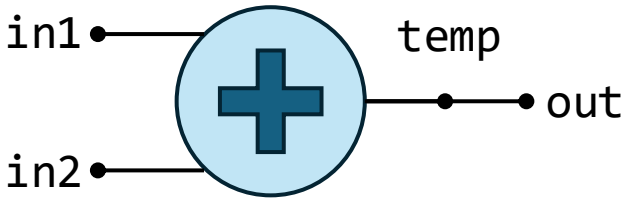
- Signals represent a net (i.e., wire) or a register in the circuit depending on how it is assigned in the HDL code
  - Continuous assignment → net
  - Connecting subcomponents → net
  - Combinational logic → net
  - Sequential logic → register

- Example signal declarations

```
logic sel;  // 1b signal
logic [3:0] op0, op1; // two 4b signals
logic [7:0] data [0:3]; // four 8b signals
```

```
module adder8b (
     input  [7:0] in1,
     input  [7:0] in2,
     output [8:0] out
);

logic [8:0] temp;

assign temp = in1 + in2;
assign out = temp;

endmodule
```

What does that mean for the generated hardware?



`logic` Signals

- Signals represent a net (i.e., wire) or a register in the circuit depending on how it is assigned in the HDL code
  - Continuous assignment → net
  - Connecting subcomponents → net
  - Combinational logic → net
  - Sequential logic → register

```
module adder8b (
     input  [7:0] in1,
     input  [7:0] in2,
     output [8:0] out
);

logic [8:0] temp;

assign out = temp;
assign temp = in1 + in2;

endmodule
```

What happens if we flip the order of assignments?

- Example signal declarations

```
logic sel;  // 1b signal
logic [3:0] op0, op1; // two 4b signals
logic [7:0] data [0:3]; // four 8b signals
```

`logic` Values

- Signals can have four possible values:
  - 0 = logic 0
  - 1 = logic 1
  - Z = tri-state or high impedance → not connected to anything
  - X = unknown at the time → uninitialized or has multiple drivers
    - Symbolic – In real hardware, value could be 0 or 1
    - Useful for tracing bugs in simulation
- A value is specified as `[size][’radix]constant`
  - size is the number of bits
  - radix is the number base (d:decimal, b:binary, h:hexadecimal, o:octal)

`11`	Decimal number 11
`’b11`	Binary number $(11)_2 = (3)_{10}$
`’h1C`	Hex number $(1C)_{16} = (28)_{10}$
`4’b10`	Binary number $(0010)_2 = (2)_{10}$
`4’bX`	Undefined 4-bit value XXXX
`8’b1000_0011`	_ can be used for readability

SystemVerilog Operators

Example: A = 4'b0101 (5), B = 4'b1000 (8), C = 4'b0001 (1), n = 2

Operator	Meaning	Example Eval.
~A	Bitwise Negation	4'b1010
-A	2's complement	4'b1011
A & B	Bitwise AND	4'b0000
A | B	Bitwise OR	4'b1101
A ^ B	Bitwise XOR	4'b1101
!A	Logical NOT	1'b0
A && B	Logical AND	1'b1
A || B	Logical OR	1'b1
&A	Reduction AND	1'b0
|A	Reduction OR	1'b1
A == B	Logical equality	1'b0
A != B	Logical inequality	1'b1
A > B, A >= B	Greater than (or =)	1'b0
A < B, A <= B	Less than (or =)	1'b1

Operator	Meaning	Example Eval.
A + B	Addition	4'b1101
B - A	Subtraction	4'b0011
A * B	Multiplication	8'b00101000
B / A	Division	4'b0001
B % A	Modulus	4'b0011
A << C	Logical shift left	4'b1010
A >> C	Logical shift right	4'b0010
{A, B}	Concatenate	8'b01011000
{n {A}}	Replicate	8'b01010101
A ? B : C	Condition	4'b1000

There are more operators. This is only a subset of the most commonly-used ones.

Parameters

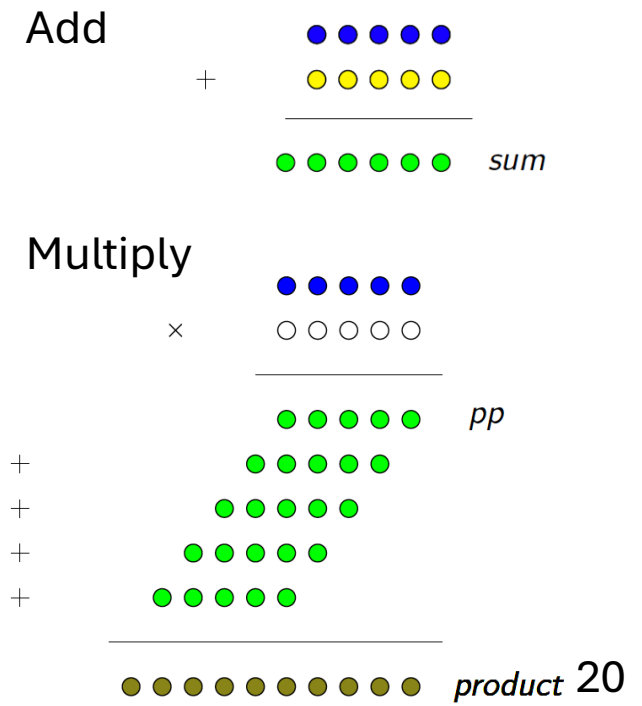
- Allow the same code to be reused to implement different instances of the same module in a bigger design using different configurations
- Parameter values must be known at synthesis ("compile") time
- Somewhat analogous to templated C++ functions

```
module adder #(
    parameter N = 8
)(
    input  [N-1:0] in1,
    input  [N-1:0] in2,
    output [N:0] out
);

logic [N:0] temp;

assign out = temp;
assign temp = in1 + in2;

endmodule
```

An Aside: Numerical Precision

- SystemVerilog is not strongly typed
- Precision problems do not flag errors
  - Some synthesis engines produce a warning
- Could be a source of bugs!
- Need to remember some rules:
  - Add/Sub: 1 extra bit than largest operand
  - Multiply: sum of bits of both operands
  - Other ops: analyze input/output ranges
- Sign bit handling
  - By default, all ports/signals are unsigned
  - Need to explicitly use `signed` keyword
    - `logic signed`
    - `input signed` or `output signed`

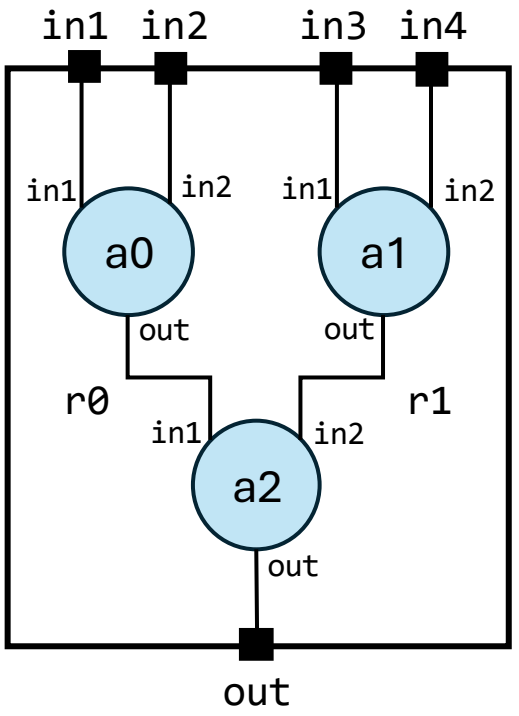
```
module poly (
     input  [7:0] x,
     input  [7:0] a,
     input  [7:0] b,
     input  [7:0] c,
     output [7:0] y
);

assign y = a*x*x + b*x + c;

endmodule
```



Hierarchical Instantiation

- Modules can instantiate other modules as subcomponents
  - Define module parameters for each instance & connect ports
  - *"Instantiate"* and not *"call"* – not a software function

```
module adder_tree #(
     parameter IWIDTH = 8,
     parameter OWIDTH = 10
)(
     input  [IWIDTH-1:0] in1,
     input  [IWIDTH-1:0] in2,
     input  [IWIDTH-1:0] in3,
     input  [IWIDTH-1:0] in4,
     output [OWIDTH-1:0] out
);

logic [IWIDTH:0] r0, r1;

adder # (IWIDTH) a0 (in1, in2, r0);
adder # (IWIDTH) a1 (in3, in4, r1);
adder # (IWIDTH+1) a2 (r0, r1, out);

endmodule
```



Hierarchical Instantiation

- Modules can instantiate other modules as subcomponents
  - Define module parameters for each instance & connect ports
  - *"Instantiate"* and not *"call"* – not a software function

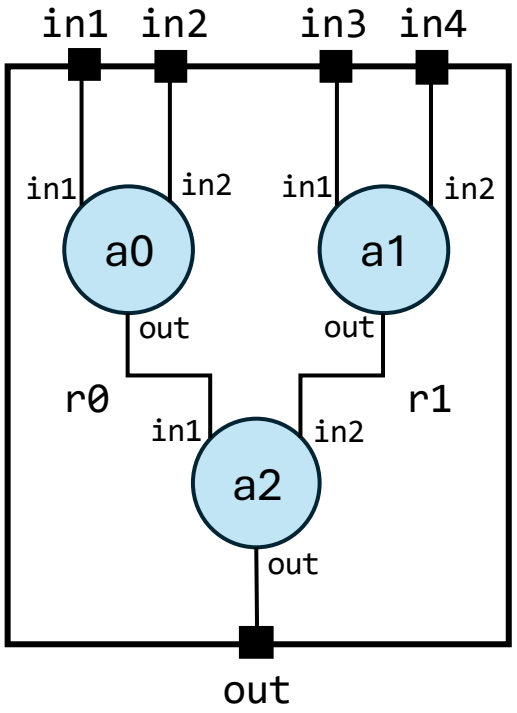
```
module adder_tree #(
     parameter IWIDTH = 8,
     parameter OWIDTH = 10
)(
     input  [IWIDTH-1:0] in1,
     input  [IWIDTH-1:0] in2,
     input  [IWIDTH-1:0] in3,
     input  [IWIDTH-1:0] in4,
     output [OWIDTH-1:0] out
);

logic [IWIDTH:0] r0, r1;

adder # (IWIDTH) a0 (in1, in2, r0);
adder # (IWIDTH) a1 (in3, in4, r1);
adder # (IWIDTH+1) a2 (r0, r1, out);

endmodule
```

```
module adder #(
     parameter N = 8
)(
     input  [N-1:0] in1,
     input  [N-1:0] in2,
     output [N:0] out
);
```

Positional association

Same order of parameter & port definitions in adder

Harder to understand & maintain



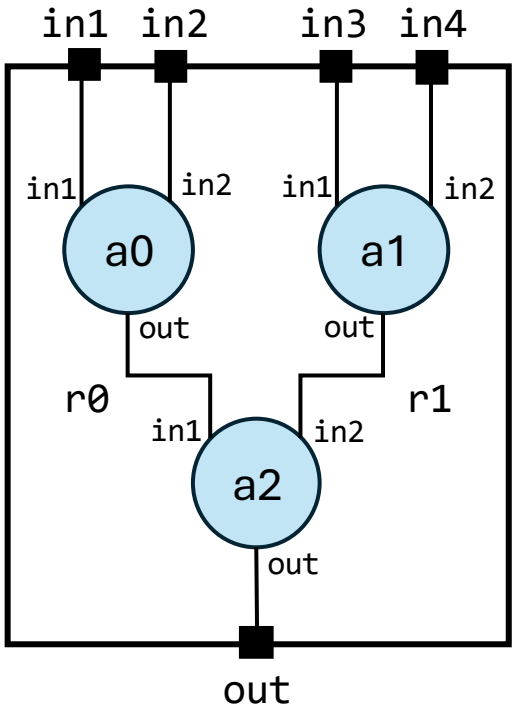
Hierarchical Instantiation

- Modules can instantiate other modules as subcomponents
  - Define module parameters for each instance & connect ports
  - *"Instantiate"* and not *"call"* – not a software function

```
module adder_tree #(
     parameter IWIDTH = 8,
     parameter OWIDTH = 10
)(
     input  [IWIDTH-1:0] in1,
     input  [IWIDTH-1:0] in2,
     input  [IWIDTH-1:0] in3,
     input  [IWIDTH-1:0] in4,
     output [OWIDTH-1:0] out
);

logic [IWIDTH:0] r0, r1;

adder # (.N(IWIDTH)) a0 (.in1(in1), .in2(in2), .out(r0));
adder # (.N(IWIDTH)) a1 (.in1(in3), .in2(in4), .out(r1));
adder # (.N(IWIDTH+1)) a2 (.in1(r0), .in2(r1), .out(out));

endmodule
```

```
module adder #(
     parameter N = 8
)(
     input  [N-1:0] in1,
     input  [N-1:0] in2,
     output [N:0] out
);
```

Named association
Explicit binding of ports



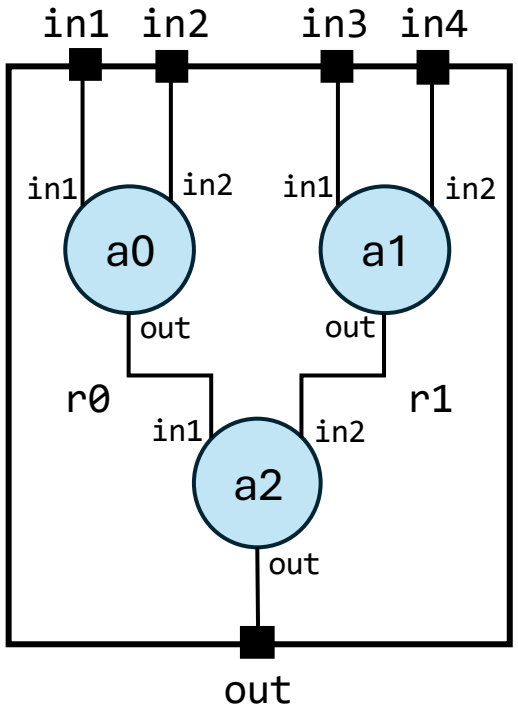
Hierarchical Instantiation

- Modules can instantiate other modules as subcomponents
  - Define module parameters for each instance & connect ports
  - *"Instantiate"* and not *"call"* – not a software function

```
module adder_tree #(
     parameter IWIDTH = 8,
     parameter OWIDTH = 10
)(
     input  [IWIDTH-1:0] in1,
     input  [IWIDTH-1:0] in2,
     input  [IWIDTH-1:0] in3,
     input  [IWIDTH-1:0] in4,
     output [OWIDTH-1:0] out
);

logic [IWIDTH:0] r0, r1;

adder # (.N(IWIDTH)) a0 (.in2(in2), .in1(in1), .out(r0));
adder # (.N(IWIDTH)) a1 (.out(r1), .in1(in3), .in2(in4));
adder # (.N(IWIDTH+1)) a2 (.in1(r0), .in2(r1), .out(out));

endmodule
```

```
module adder #(
     parameter N = 8
)(
     input  [N-1:0] in1,
     input  [N-1:0] in2,
     output [N:0] out
);
```

Named association
Explicit binding of ports
Can have arbitrary order
Recommended best practice



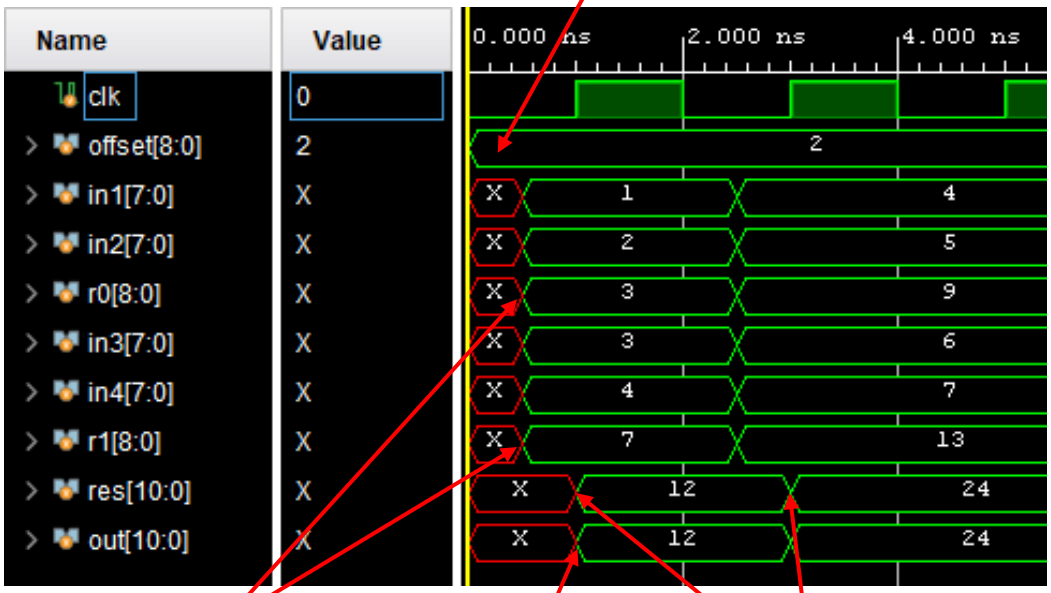
Procedural Blocks

- Procedural blocks describe the behavior of the module
- Two types of procedural blocks:
  - initial: behavior that happens only once at the beginning of time
  - always: behavior that happens every time a signal in sensitivity list changes
- There are 2 commonly-used types of always blocks in SystemVerilog
  - `always_comb` – combinational logic
    - Sensitivity list is any RHS signal
  - `always_ff` – sequential logic
    - Sensitivity list is clock `posedge` or `negedge`

```
module example_circuit (
     input  clk,
     input  [7:0] in1,
     input  [7:0] in2,
     input  [7:0] in3,
     input  [7:0] in4,
     output [10:0] out
);

logic [8:0] offset, r0, r1;
logic [10:0] res;

initial begin
     offset = 9'd2;
end

always_comb begin
     r0 = in1 + in2;
     r1 = in3 + in4;
end

always_ff @ (posedge clk) begin
     res <= r0 + r1 + offset;
end

assign out = res;

endmodule
```

Procedural Blocks

offset is set at beginning of time (initial)



r0, r1 change when any of the inputs change (always_comb)

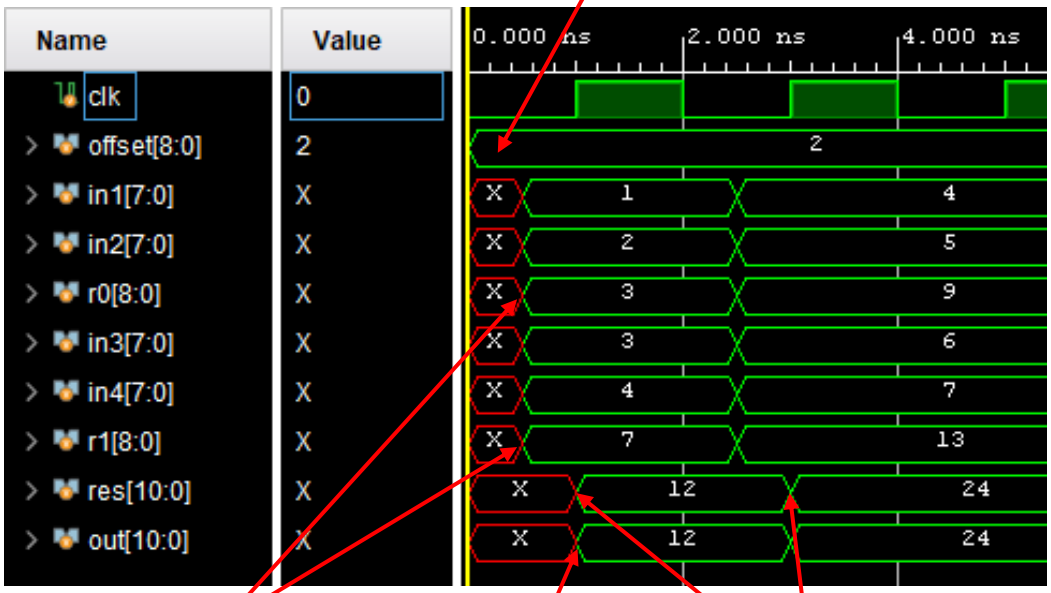
res changes at rising clock edge (always_ff)

out changes when res changes (assign)

```
module example_circuit (
     input  clk,
     input  [7:0] in1,
     input  [7:0] in2,
     input  [7:0] in3,
     input  [7:0] in4,
     output [10:0] out
);

logic [8:0] offset, r0, r1;
logic [10:0] res;

initial begin
     offset = 9'd2;
end

always_comb begin
     r0 = in1 + in2;
     r1 = in3 + in4;
end

always_ff @ (posedge clk) begin
     res <= r0 + r1 + offset;
end

assign out = res;

endmodule
```

Procedural Blocks

offset is set at beginning of time (initial)



r0, r1 change when any of the inputs change (always_comb)

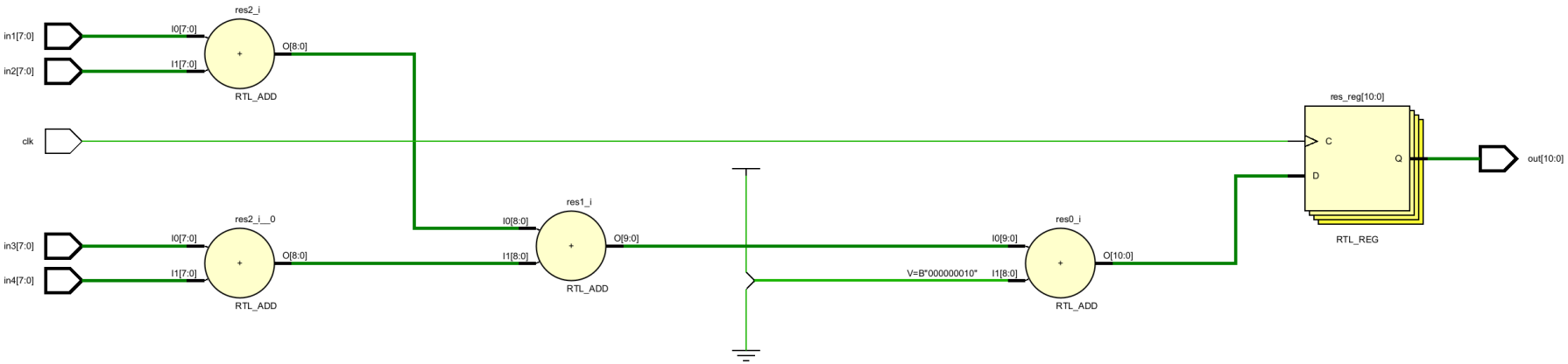
res changes at rising clock edge (always_ff)

out changes when res changes (assign)

```
module example_circuit (
     input  clk,
     input  [7:0] in1,
     input  [7:0] in2,
     input  [7:0] in3,
     input  [7:0] in4,
     output [10:0] out
);

logic [8:0] offset, r0, r1;
logic [10:0] res;

initial begin
     offset = 9'd2;
end

always_comb begin
     r0 = in1 + in2;
     r1 = in3 + in4;
end

always_ff @ (posedge clk) begin
     res <= r0 + r1 + offset;
end

assign out = res;

endmodule
```

Can you guess the hardware?

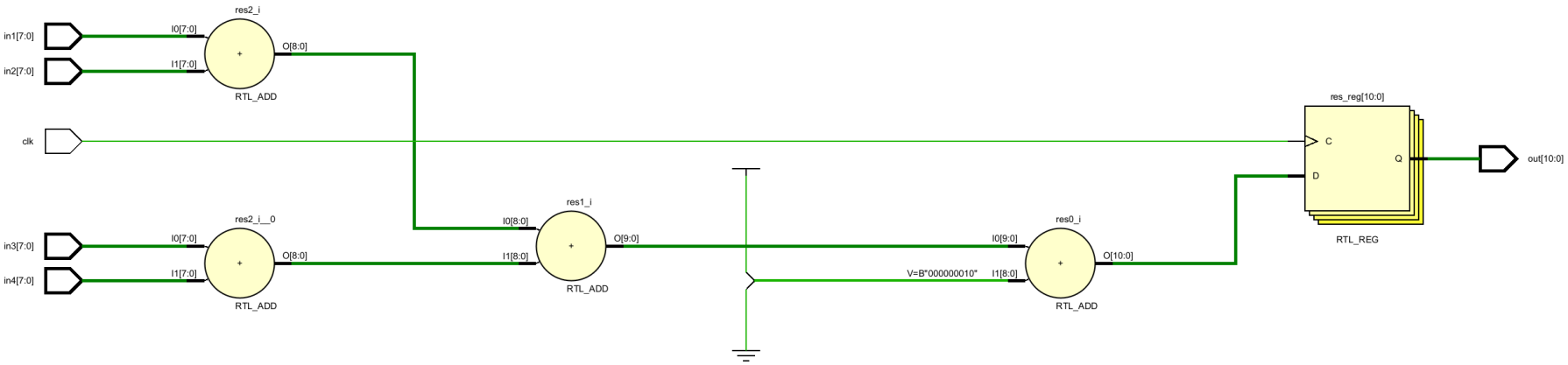
Procedural Blocks




```
module offset_adder_tree (
     input  clk,
     input  [7:0] in1,
     input  [7:0] in2,
     input  [7:0] in3,
     input  [7:0] in4,
     output [10:0] out
);

logic [8:0] offset, r0, r1;
logic [10:0] res;

initial begin
     offset = 9'd2;
end
```

```
always_comb begin
     r0 = in1 + in2;
     r1 = in3 + in4;
end

always_ff @ (posedge clk) begin
     res <= r0 + r1 + offset;
end

assign out = res;

endmodule
```

Procedural Blocks




```
module offset_adder_tree (
     input  clk,
     input  [7:0] in1,
     input  [7:0] in2,
     input  [7:0] in3,
     input  [7:0] in4,
     output [10:0] out
);

logic [8:0] offset, r0, r1;
logic [10:0] res;

initial begin
     offset = 9'd2;
end
```

```
always_comb begin
     r0 = in1 + in2;
     r1 = in3 + in4;
end

always_ff @ (posedge clk) begin
     res <= r0 + r1 + offset;
end

assign out = res;

endmodule
```

Have you noticed anything strange?
= vs. <= ?

Blocking & Non-Blocking Assignments

- Inside a procedural block, it is possible to use blocking (=) or non-blocking (<=) assignments

- **Blocking** assignments describe **sequential** behavior
  - The behavior described by one statement happens before the subsequent statements (in imaginary infinitesimally-small time steps)

- **Non-blocking** assignments describe **concurrent** behavior
  - All statements happen simultaneously

Blocking & Non-Blocking Assignments

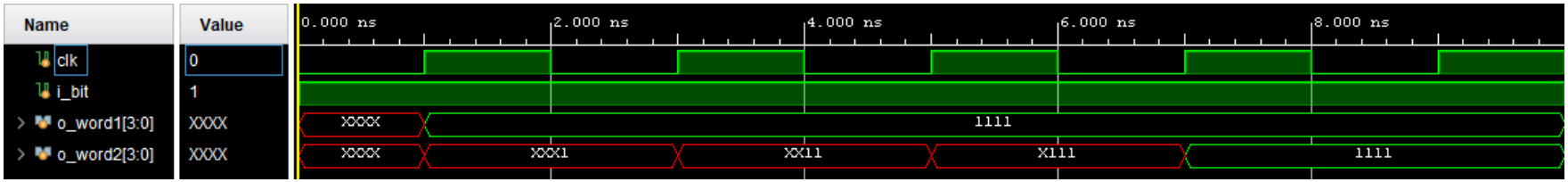
```
module blocking_vs_non_blocking(
     input  clk,
     input  i_bit,
     output [3:0] o_word1,
     output [3:0] o_word2
);

logic [3:0] o1, o2;

always_ff @ (posedge clk) begin
     o1[0] = i_bit;
     o1[1] = o1[0];
     o1[2] = o1[1];
     o1[3] = o1[2];
end

always_ff @ (posedge clk) begin
     o2[0] <= i_bit;
     o2[1] <= o2[0];
     o2[2] <= o2[1];
     o2[3] <= o2[2];
end

assign o_word1 = o1;
assign o_word2 = o2;

endmodule
```

o_word1 and o_word2 start as 4'bXXXX
Let's assume we provide i_bit = 1'b1

What would be the values of o_word1 and o_word2 after the rising clock edge?

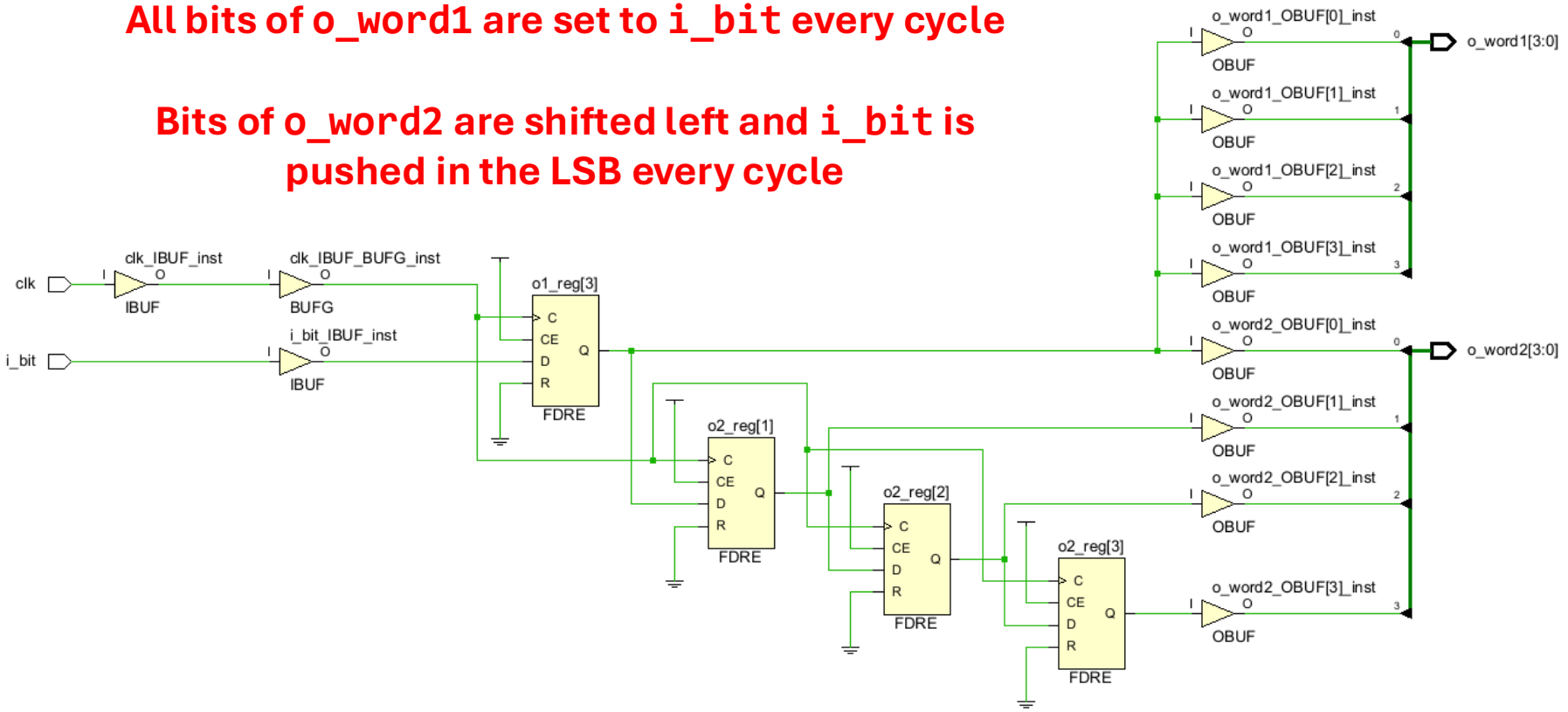
Is the generated hardware circuit the same or different?

Blocking & Non-Blocking Assignments



All bits of `o_word1` are set to `i_bit` every cycle

Bits of `o_word2` are shifted left and `i_bit` is pushed in the LSB every cycle



Blocking & Non-Blocking Assignments

```
module blocking_vs_non_blocking(
     input  clk,
     input  i_bit,
     output [3:0] o_word1,
     output [3:0] o_word2
);

logic [3:0] o1, o2;

always_ff @ (posedge clk) begin
     o1[0] = i_bit;
     o1[1] = o1[0];
     o1[2] = o1[1];
     o1[3] = o1[2];
end

always_ff @ (posedge clk) begin
     o2[0] <= i_bit;
     o2[1] <= o2[0];
     o2[2] <= o2[1];
     o2[3] <= o2[2];
end

assign o_word1 = o1;
assign o_word2 = o2;

endmodule
```

o_word1 and o_word2 start as 4'bXXXX
Let's assume we provide i_bit = 1'b1

What would be the values of o_word1 and o_word2 after the rising clock edge?

Is the generated hardware circuit the same or different?

Blocking & Non-Blocking Assignments

```
module blocking_vs_non_blocking(
     input  clk,
     input  i_bit,
     output [3:0] o_word1,
     output [3:0] o_word2
);

logic [3:0] o1, o2;

always_ff @ (posedge clk) begin
     o1[3] = o1[2];
     o1[2] = o1[1];
     o1[1] = o1[0];
     o1[0] = i_bit;
end

always_ff @ (posedge clk) begin
     o2[3] <= o2[2];
     o2[2] <= o2[1];
     o2[1] <= o2[0];
     o2[0] <= i_bit;
end

assign o_word1 = o1;
assign o_word2 = o2;

endmodule
```

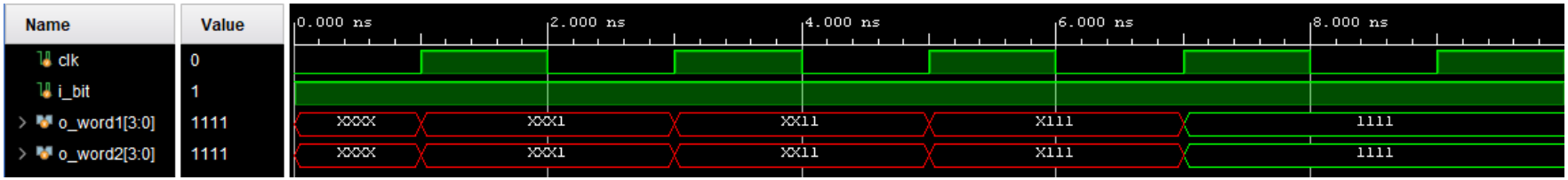
o_word1 and o_word2 start as 4'bXXXX
Let's assume we provide i_bit = 1'b1

What would be the values of o_word1 and o_word2 after the rising clock edge?

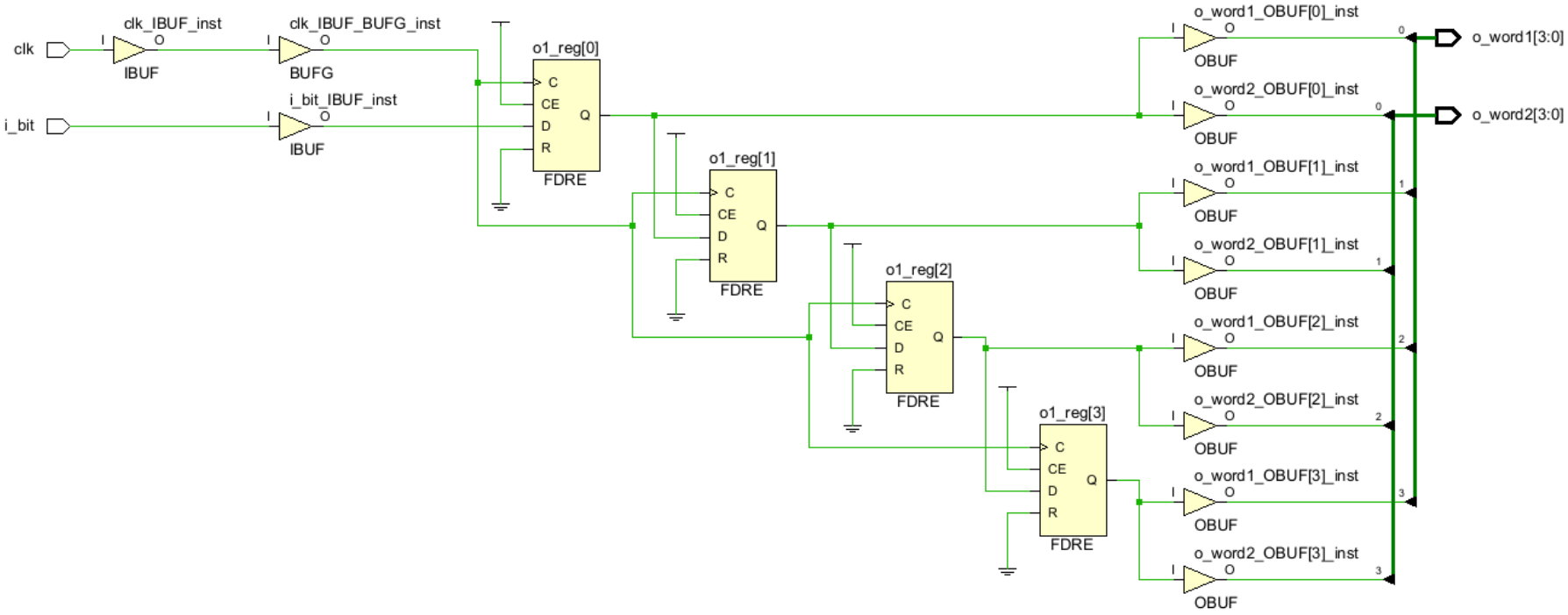
Is the generated hardware circuit the same or different?

Does the order of assignments make any difference?

Blocking & Non-Blocking Assignments



o_word1 and o_word2 are identical!



Note on `wire` and `reg`

- Verilog used keywords `wire` and `reg` for signals
  - A bit confusing when to use which
    - A `wire` can be driven by a continuous assignment and connect between ports of instantiated submodules
    - A `reg` can only be driven in procedural blocks (always, initial)
  - Even more confusing because `reg` does not necessarily generate a register in hardware – could be used for an intermediate node of a purely combinational circuit

- SystemVerilog replaced both with the `logic` keyword
  - Synthesis engine ("compiler") is smart enough to understand if a `logic` represents a net or a variable from how it is used in the code
  - You will still see `reg` and `wire` in legacy code in your career

Note on Code Comments

- Best way to document your code, is to include comments in it
- Good (general) coding practice to include …
  - Big picture comments about the functionality of a module
  - Description of the module's interface (i.e., input/output ports)
  - Short meaningful comments to describe complex functionality
- Short comment spans 1 line and begins with double slash //
- Long comment spans multiple lines and is contained in /* */

```
// This is a short comment
/* This is a long comment
   that spans more than one
   line */
```

Next Lecture ...

Continue with more SystemVerilog basics