

Lecture Series:
Basics of Digital Design at RT Level with Verilog

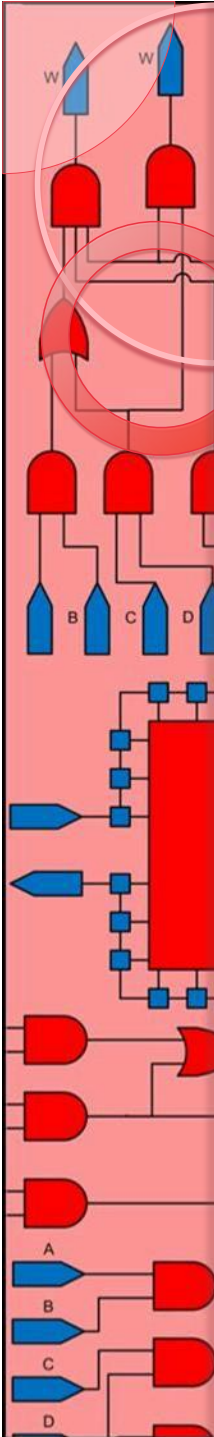
Z. Navabi

Basics of Digital Design at RT Level with Verilog

Lecture 1: Verilog as a Design Tool

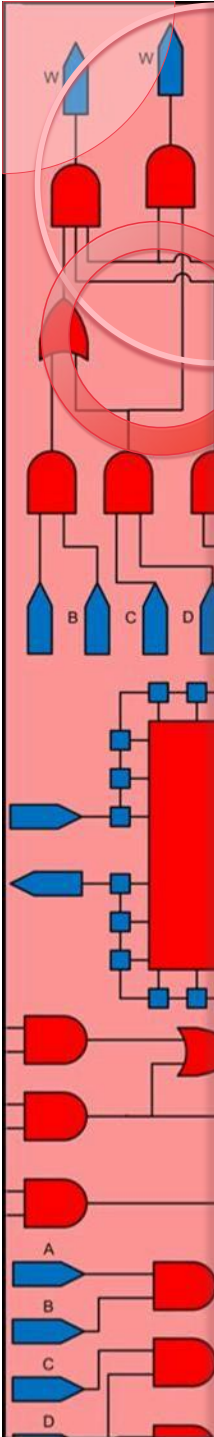
July 2014

© 2013-2014 Zain Navabi



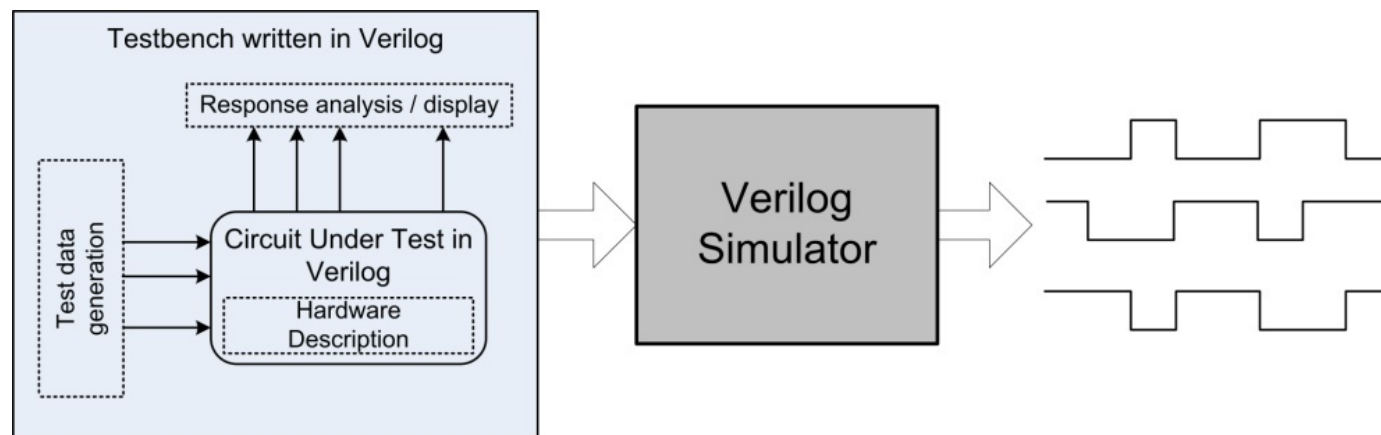
Verilog as a Design Tool

- Simulation
- Synthesis
- Language
- A simulation Environment
- Summary



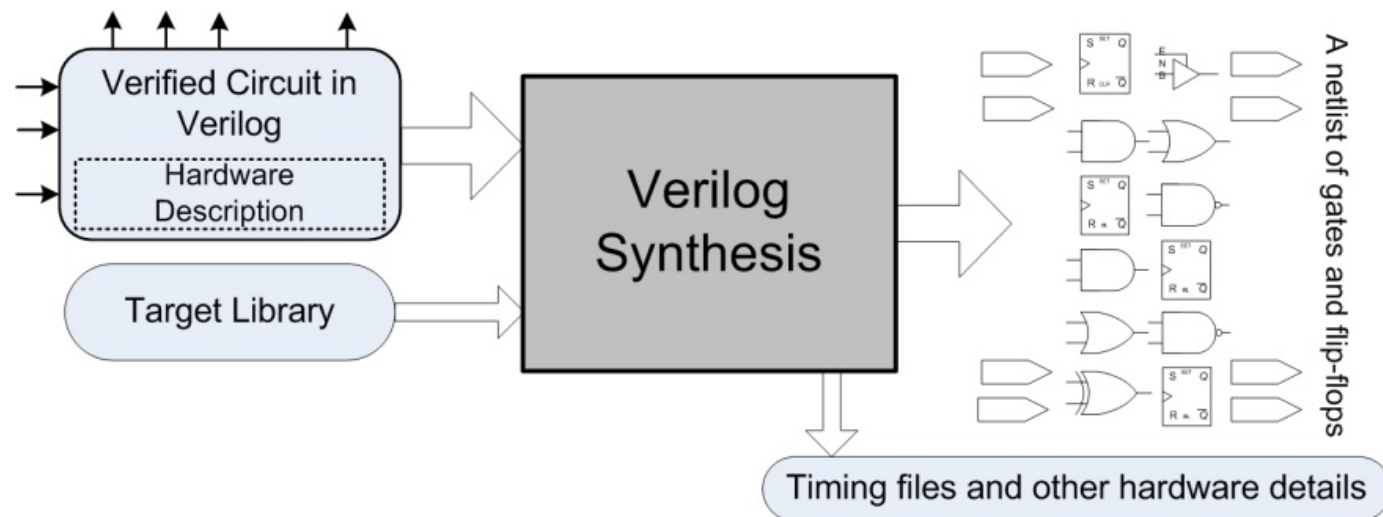
Verilog as a Design Tool

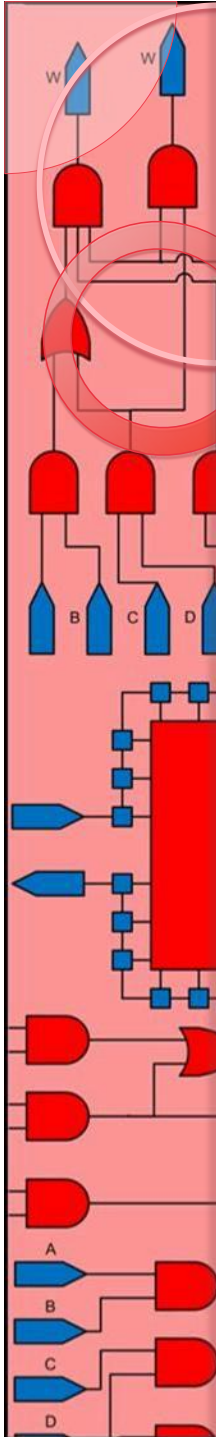
Verilog for simulation



Verilog as a Design Tool

Verilog for Synthesis



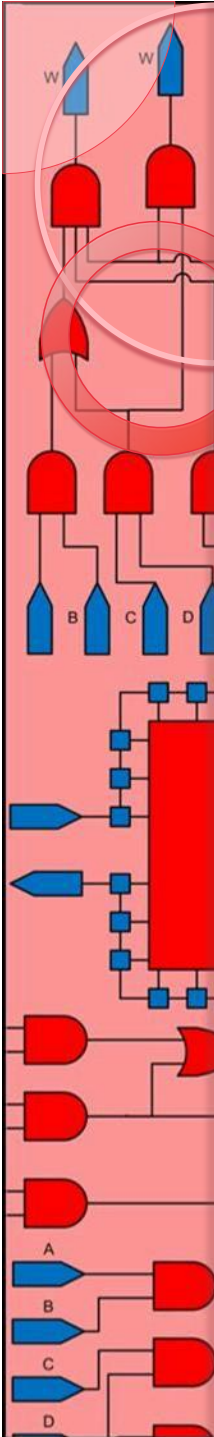


Verilog as a Design Tool

- Modules

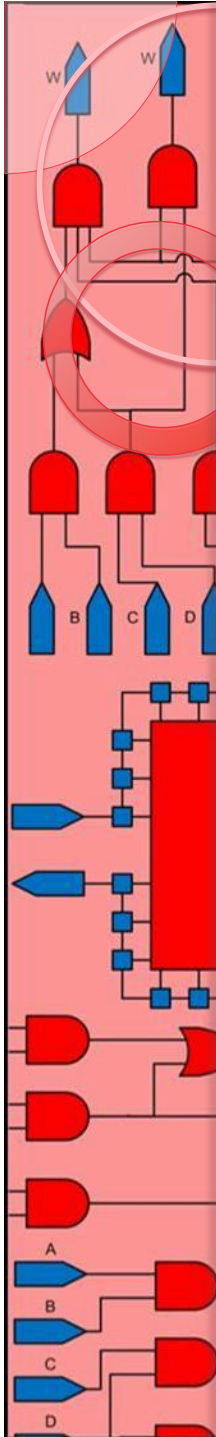
Design File: design1.v

```
module design1
    . . .
    . . .
    . . .
endmodule
```



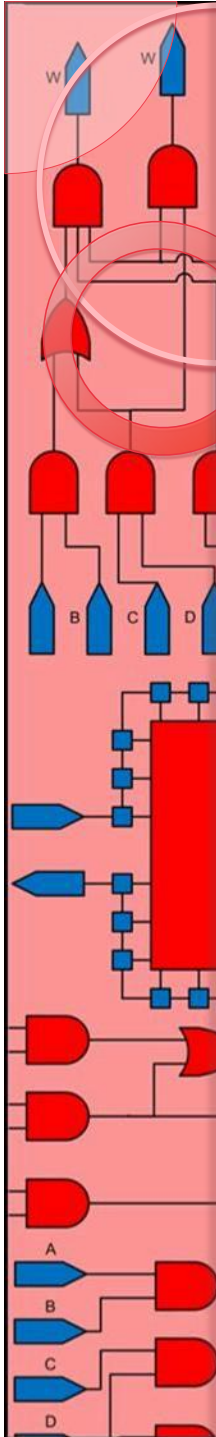
Verilog as a Design Tool

- ModelSim simulation environment
 - Define a project
 - Enter your design
 - Generate a testbench providing inputs
 - Simulate your design
 - Observe waveforms and outputs
 - Verify the operation of the design
 - Finish simulation



Verilog as a Design Tool

- Summary
 - Covered terminologies
 - A simple simulation tool
 - Became familiar with running the software



Lecture Series:

Basics of Digital Design at RT Level with Verilog

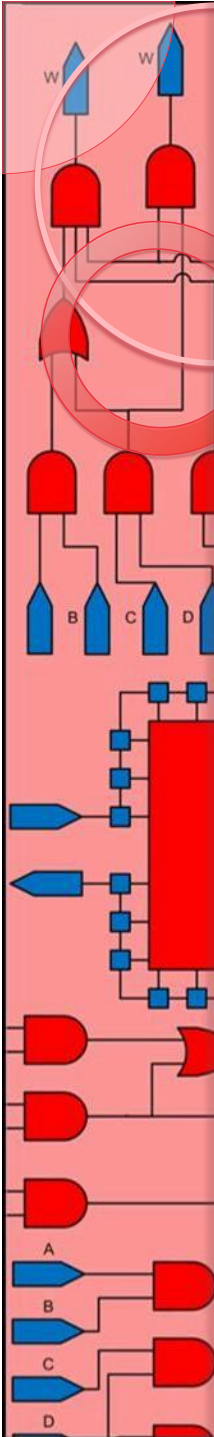
Z. Navabi, University of Tehran

Basics of Digital Design at RT Level with Verilog

Lecture 2: Gate Level Description

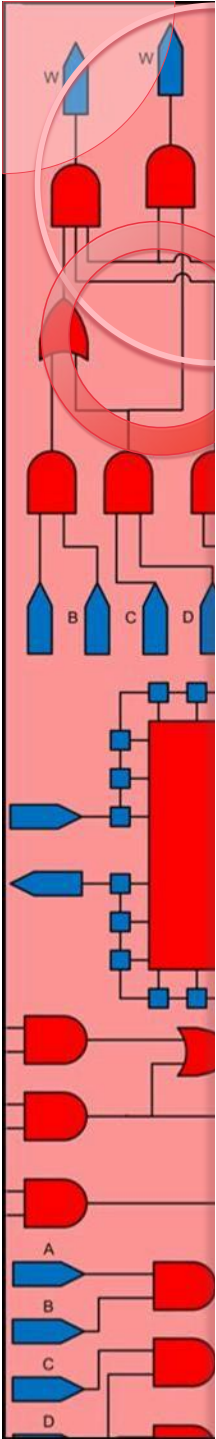
July 2014

© 2013-2014 Zain Navabi



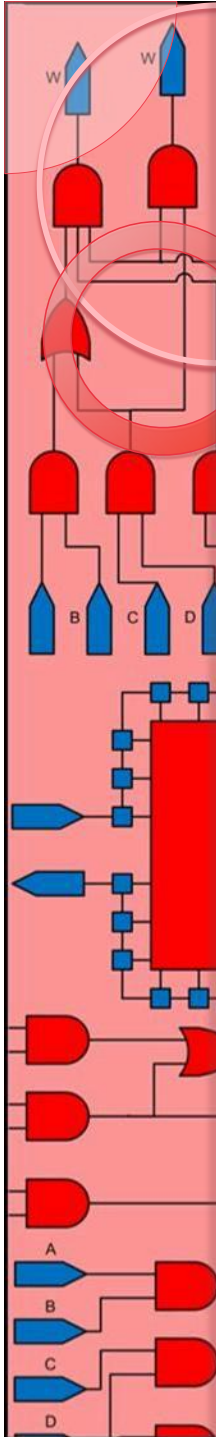
Verilog for Gate Level Descriptions

- Basic Structures of Verilog
- Levels of Abstraction
- Combinational Gates
- Writing Testbenches
- Summary



Basic Structures of Verilog

- Modules
- Module Outline
- Module Ports
- Module Variables
- Wires



Basic Structures of Verilog

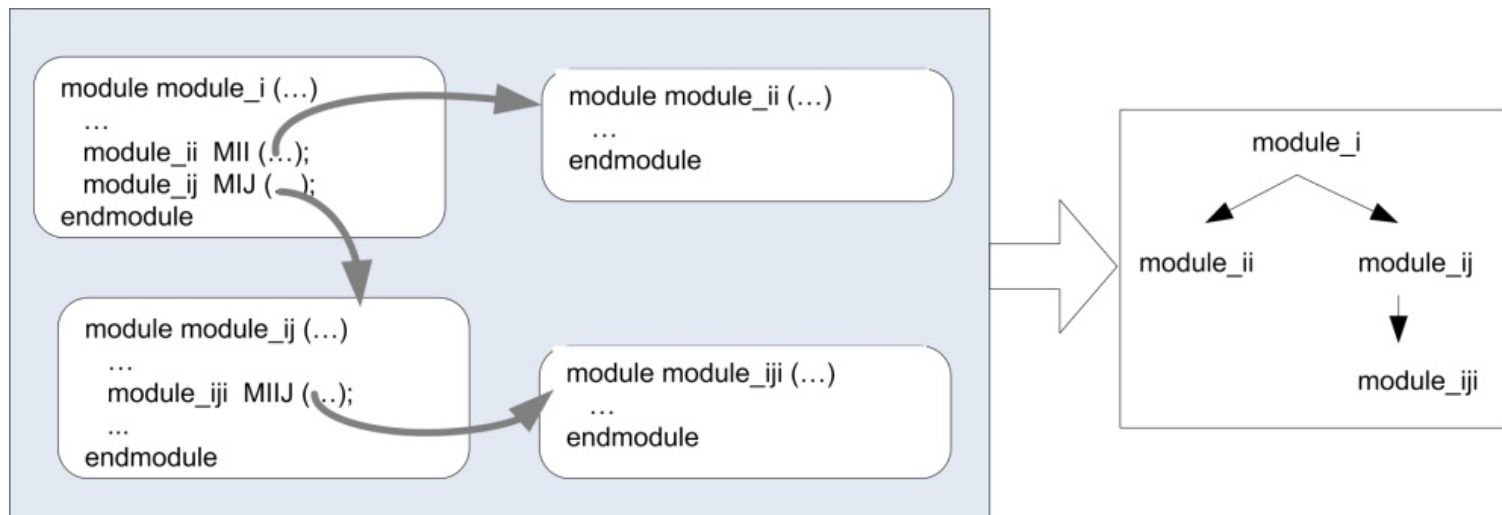
- Modules

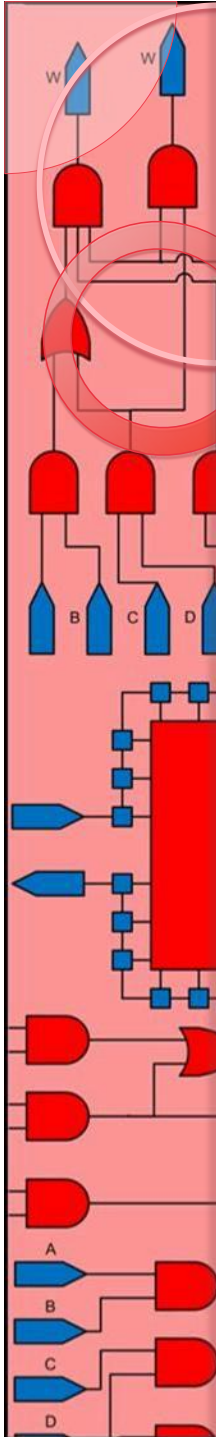
Design File: design1.v

```
module design1
    . . .
    . . .
    . . .
endmodule
```

Basic Structures of Verilog

- Modules

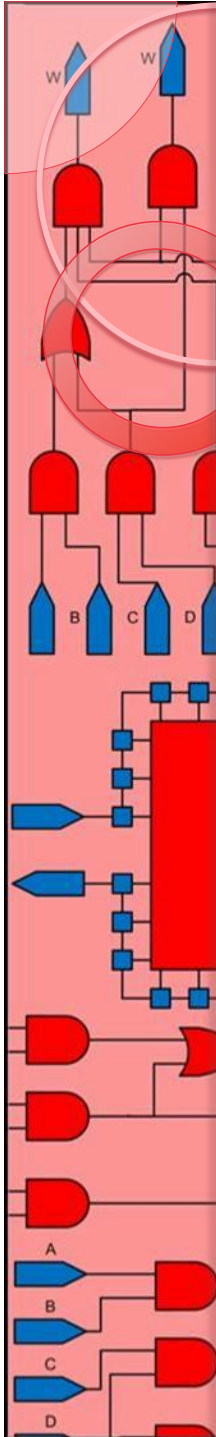




Basic Structures of Verilog

- Module Outline

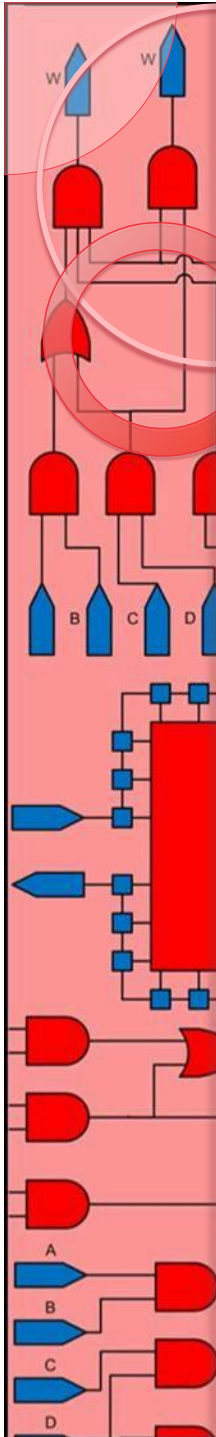
```
module name (ports or ports and their declarations);  
    port declarations if not in the header;  
    other declarations;  
    . . .  
    statements  
    . . .  
endmodule
```



Basic Structures of Verilog

- Module Ports

```
module acircuit (input a, b, output w);  
  // a comment  
  // wires and variable declarations  
  // operation of the circuit  
  . . .  
endmodule
```



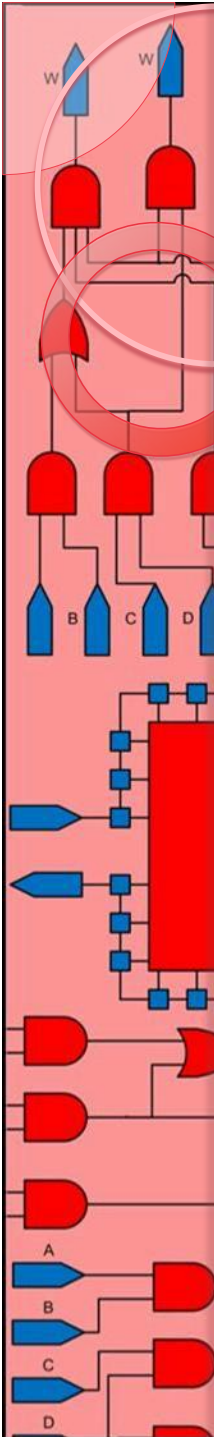
Basic Structures of Verilog

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);  
  wire c1;  
  nor g1 (c1, i1, i2);  
  and g2 (w1, c1, i3);  
  xor g3 (w2, i1, i2, i3);  
endmodule
```

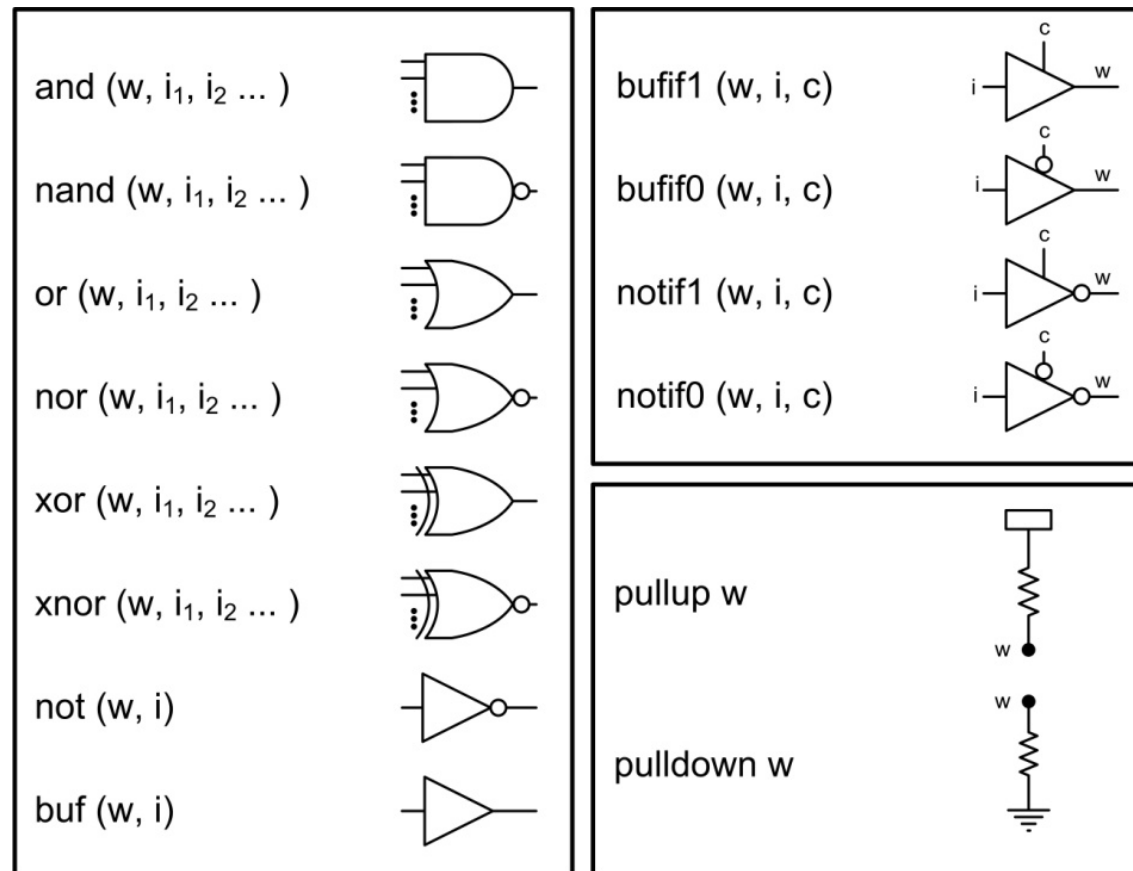
```
module simple_1b (input i1, i2, i3, output w1, w2);  
  assign w1 = i3 & ~(i1 | i2);  
  assign w2 = i1 ^ i2 ^ i3;  
endmodule
```

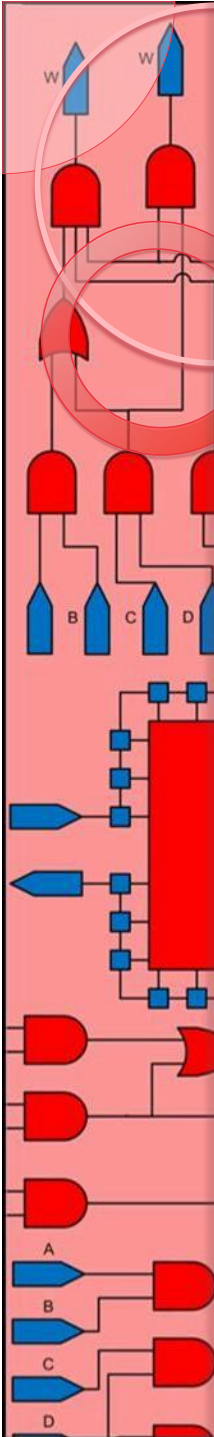
```
module simple_1c (input i1, i2, i3, output w1, w2);  
  reg w1, w2;  
  always @(i1, i2, i3) begin  
    if (i1 | i2 ) w1 = 0; else w1 = i3;  
    w2 = i1 ^ i2 ^ i3;  
  end  
endmodule
```



Combinational Circuits

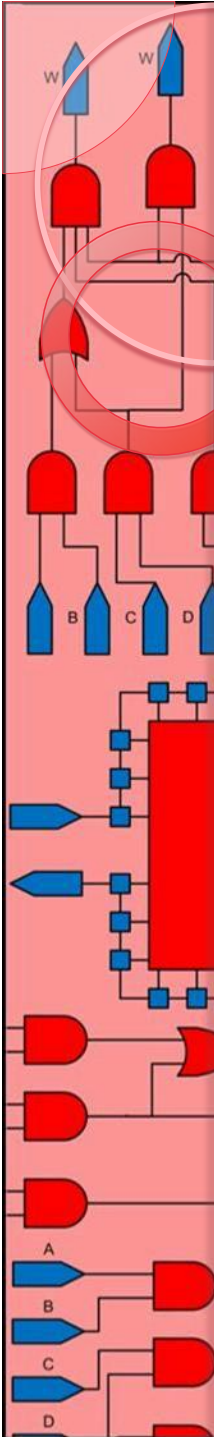
- Gate Level Combinational Circuits





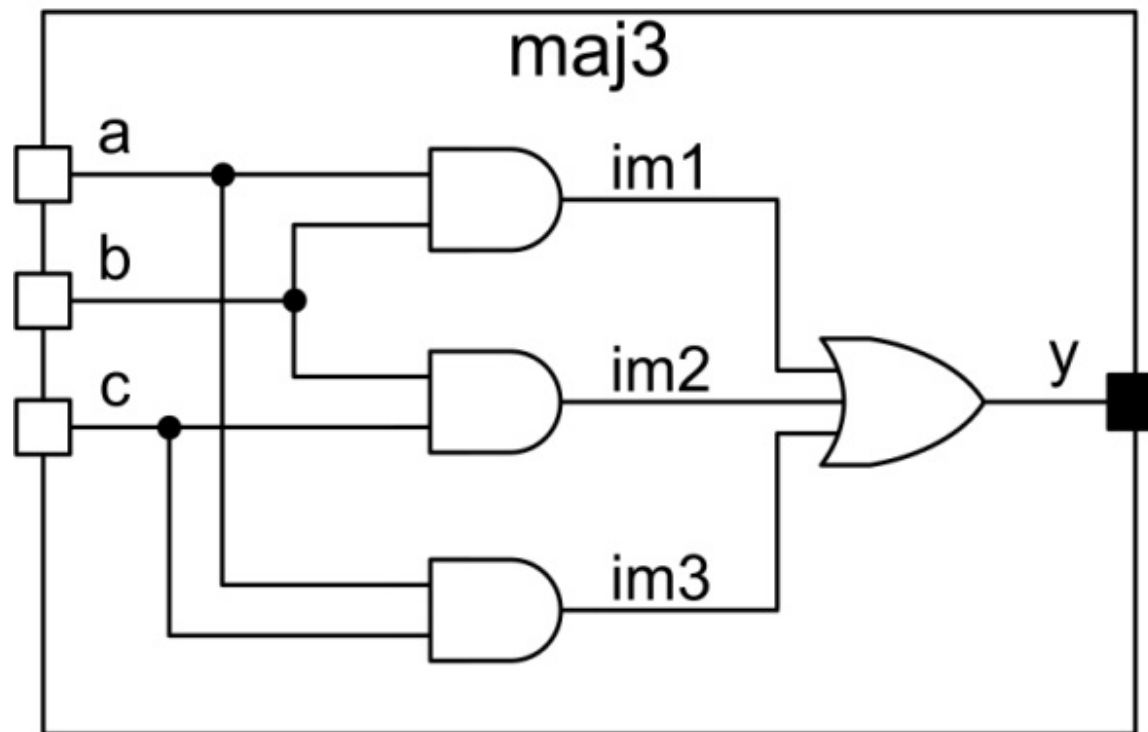
Combinational Circuits

- Gate Level Combinational Circuits
 - Majority Example
 - Multiplexer Example



Combinational Circuits

- Majority Example



Combinational Circuits

- Majority Example

```
`timescale 1ns/1ns

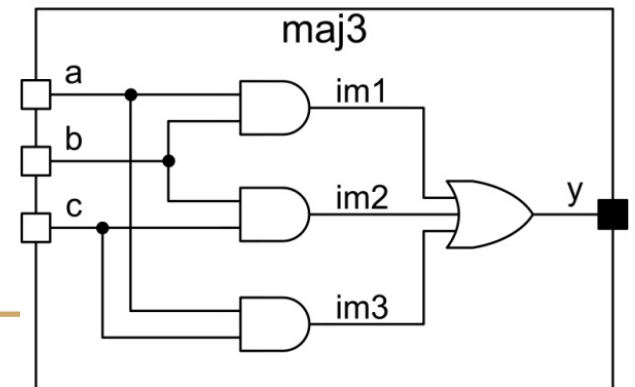
module maj3 ( input a, b, c, output y );

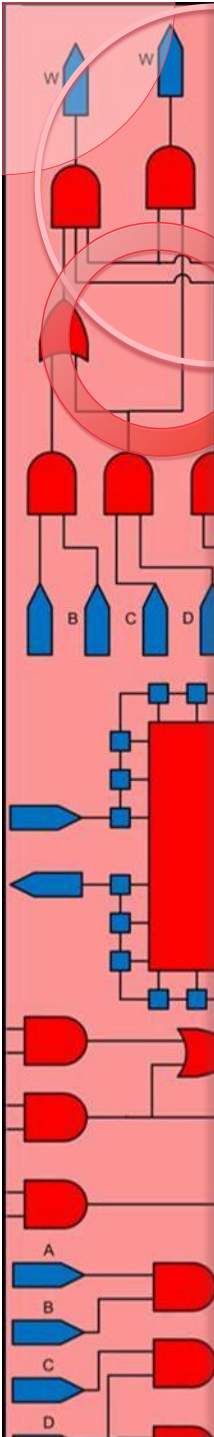
    wire im1, im2, im3;

    and U1 ( im1, a, b );
    and U2 ( im2, b, c );
    and U3 ( im3, c, a );
    or  U4 ( y, im1, im2, im3 );

endmodule
```

Verilog Code for the Majority Circuit





Writing Testbenches

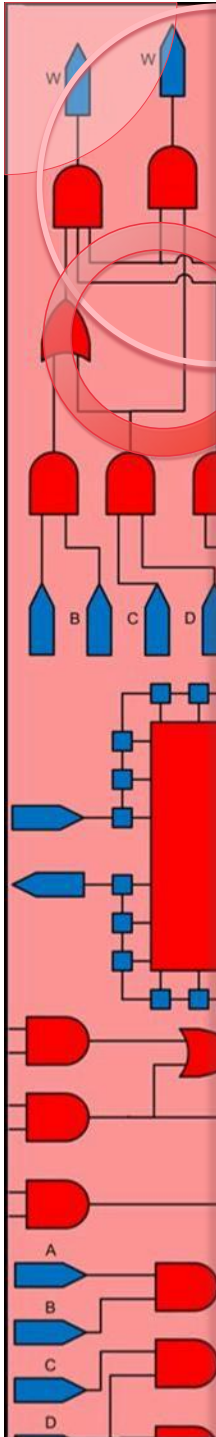
- Apply random data to test maj3

```
`timescale 1ns/1ns

module maj3Tester ();
    reg ai=0, bi=0, ci=0;
    wire yo;

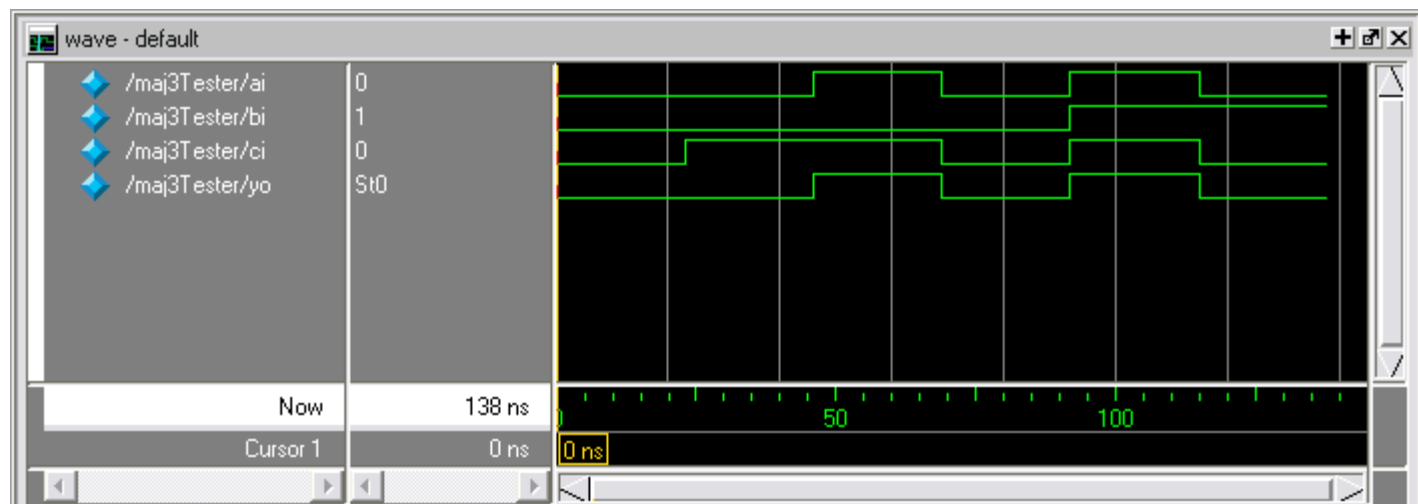
    maj3 MUT ( ai, bi, ci, yo );

    initial begin
        #23; ai=0; bi=0; ci=1;
        #23; ai=1; bi=0; ci=1;
        #23; ai=0; bi=0; ci=0;
        #23; ai=1; bi=1; ci=1;
        #23; ai=0; bi=1; ci=0;
        #23; $stop;
    end
endmodule
```



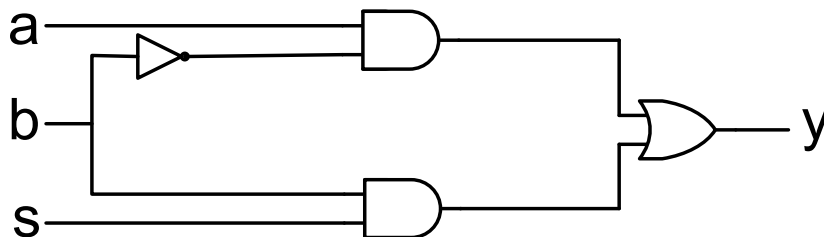
Simulation Results

- Simulating maj3



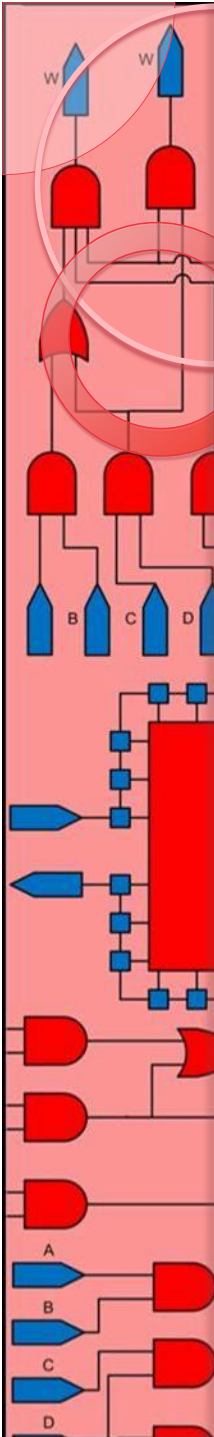
Combinational Circuits

- Multiplexer Example



Multiplexer Schematic

```
`timescale 1ns/1ns\n\nmodule mux_2to1 (a, b, s, y);\n    input a, b, s;\n    output y;\n    wire is;\n    wire aa, bb;\n\n    not U1 (is, s);\n    and U2 (aa, a, is),\n        U3 (bb, b, s);\n    or U4 (y, aa, bb);\n\nendmodule
```



Combinational Circuits

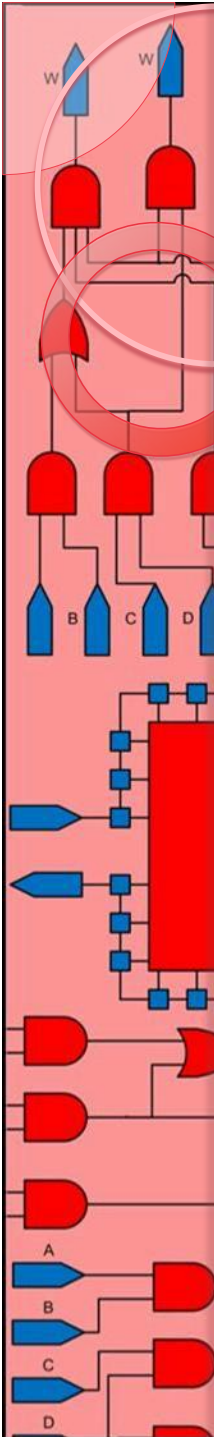
- Multiplexer Example

```
`timescale 1ns/1ns

module mux_2to1 (a, b, s, y);
    input a, b, s;
    output y;
    wire is;
    wire aa, bb;

    not U1 (is, s);
    and U2 (aa, a, is),
        U3 (bb, b, s);
    or U4 (y, aa, bb);

endmodule
```



Multiplexer Testbench

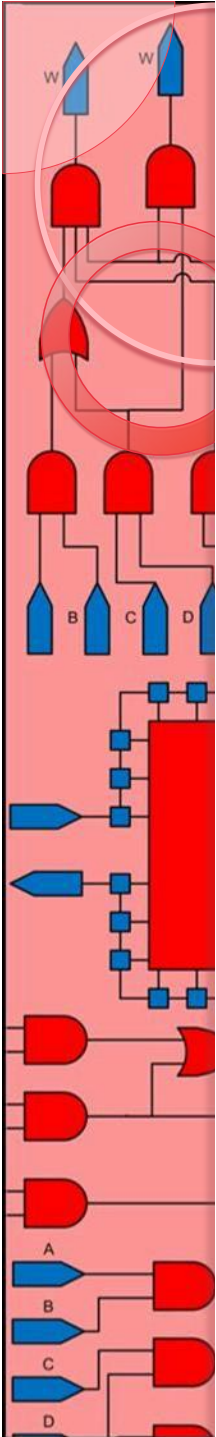
- Apply random test data to test mux

```
`timescale 1ns/1ns

module muxTester ();
    reg ai=0, bi=0, ci=0;
    wire yo;

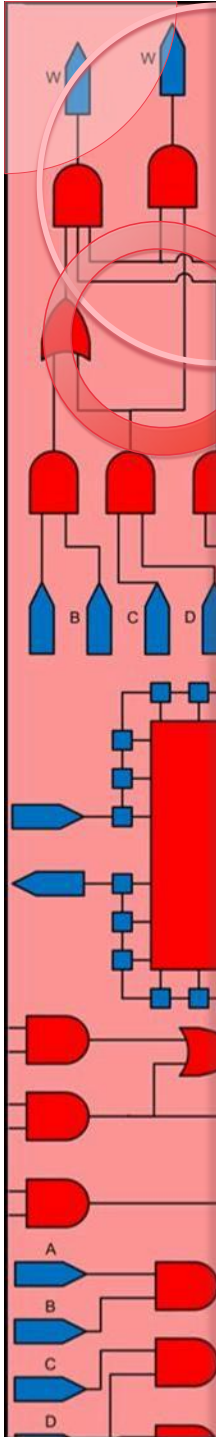
    mux_2to1 MUT ( ai, bi, ci, yo );

    initial begin
        #23; ai=0; bi=0; ci=1;
        #23; ai=1; bi=0; ci=1;
        #23; ai=0; bi=0; ci=0;
        #23; ai=1; bi=1; ci=1;
        #23; ai=0; bi=1; ci=0;
        #23; $stop;
    end
endmodule
```



Summary

- The focus of this lecture was on gate level in Verilog
- Some Basic Verilog concepts were presented
- We showed examples of levels of abstraction
- We used Verilog primitives
- Two examples were used, used somewhat different styles
- Simple testbenches were used
- ModelSim simulation run results were shown



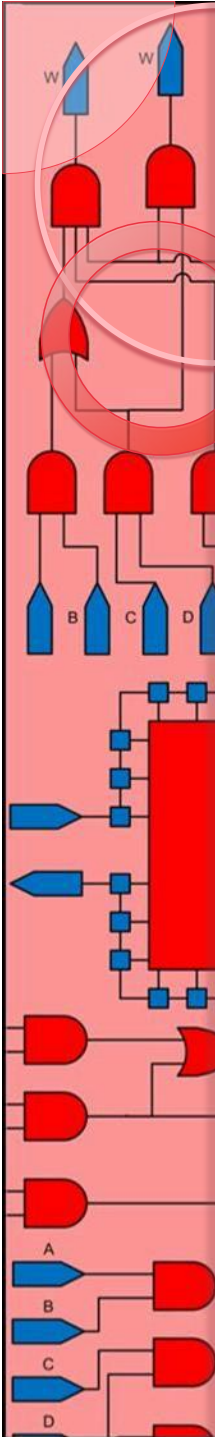
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

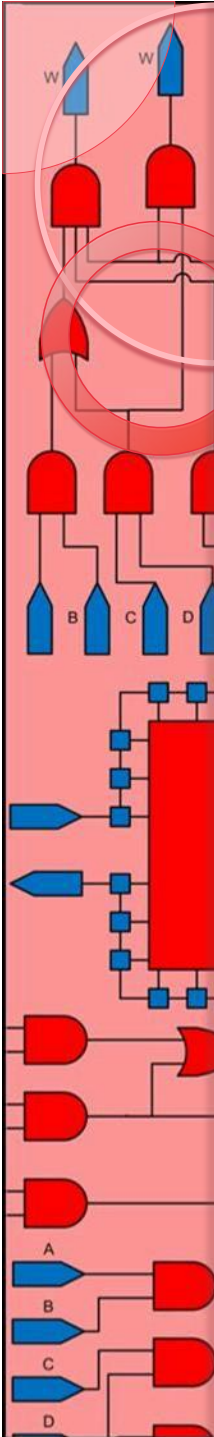
Basics of Digital Design at RT Level with Verilog

Lecture 3: Gate Delays



Gate Delays

- Switch Level Primitives
- Logic Value System
- Transistor (Switch) Level Description
- Combinational Gates with Delay
- Three-state Logic
- Summary

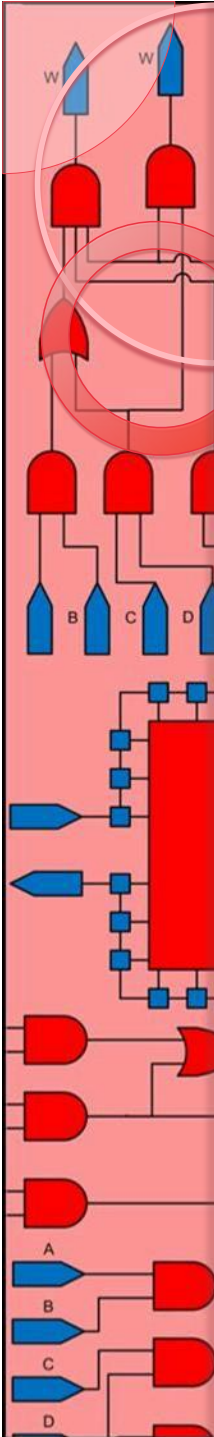


Gate Delays

- A module using transistors

Design File: design1.v

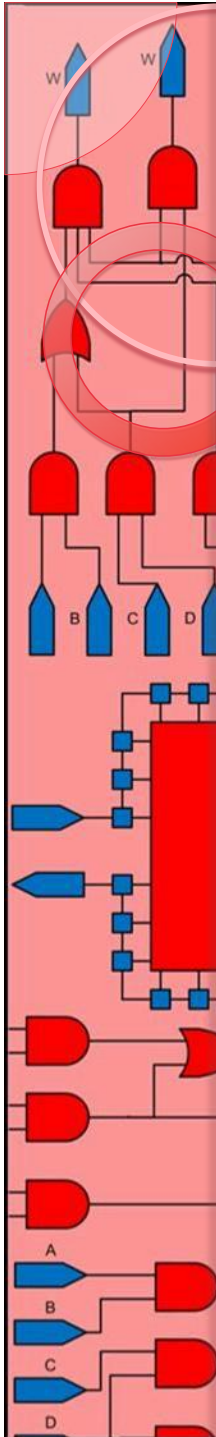
```
module design1
    . . .
    . . .
    . . .
endmodule
```



Basic Structures of Verilog

- Module Ports

```
module acircuit (input a, b, output w);  
  // a comment  
  // wires and variable declarations  
  // operation of the circuit using transistors  
  . . .  
endmodule
```



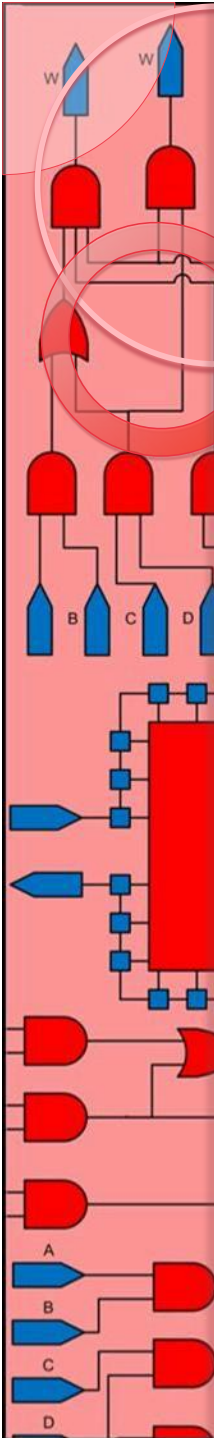
Basic Structures of Verilog

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);  
  wire c1;  
  nor g1 (c1, i1, i2);  
  and g2 (w1, c1, i3);  
  xor g3 (w2, i1, i2, i3);  
endmodule
```

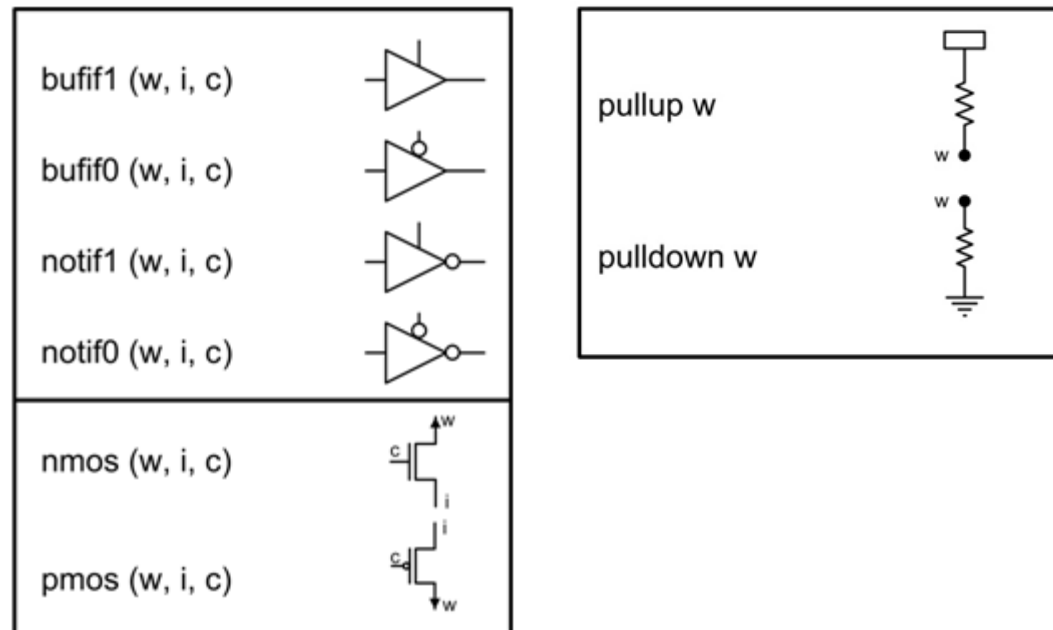
```
module simple_1b (input i1, i2, i3, output w1, w2);  
  assign w1 = i3 & ~(i1 | i2);  
  assign w2 = i1 ^ i2 ^ i3;  
endmodule
```

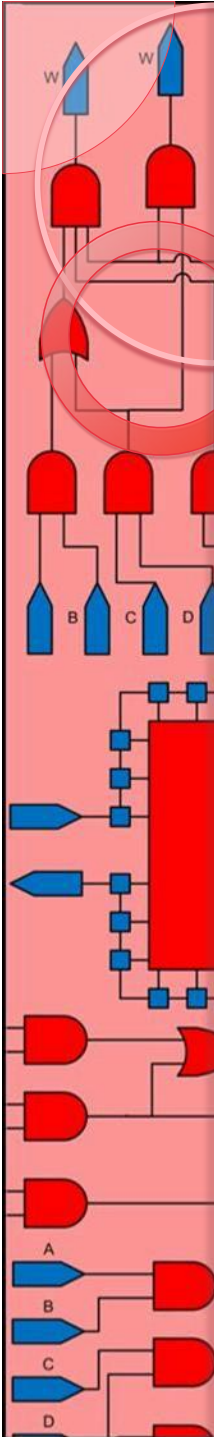
```
module simple_1c (input i1, i2, i3, output w1, w2);  
  reg w1, w2;  
  always @(i1, i2, i3) begin  
    if (i1 | i2 ) w1 = 0; else w1 = i3;  
    w2 = i1 ^ i2 ^ i3;  
  end  
endmodule
```



Switch Level Primitives

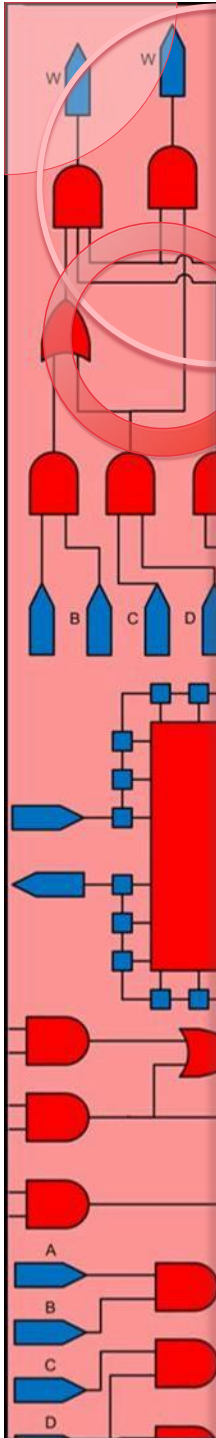
- Switches and three-state gates





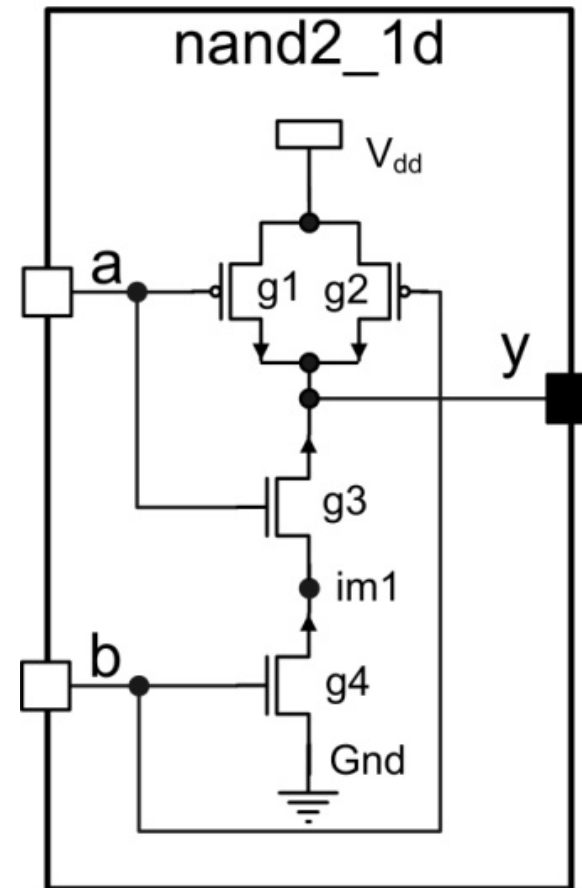
Switch Level Primitives

- Logic Value System
 - 0: forced low
 - 1: forced high
 - Z: open
 - X: unknown



Transistor (Switch) Level Description

- CMOS NAND Example



CMOS NAND at the Switch Level

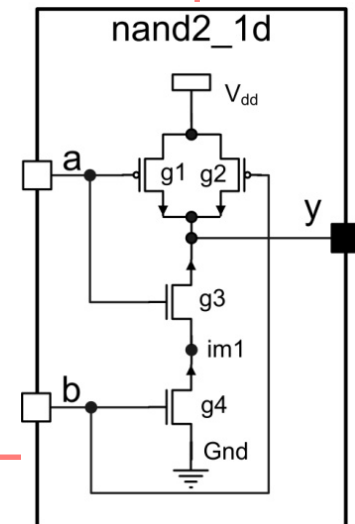
Transistor (Switch) Level Description

- CMOS NAND Example

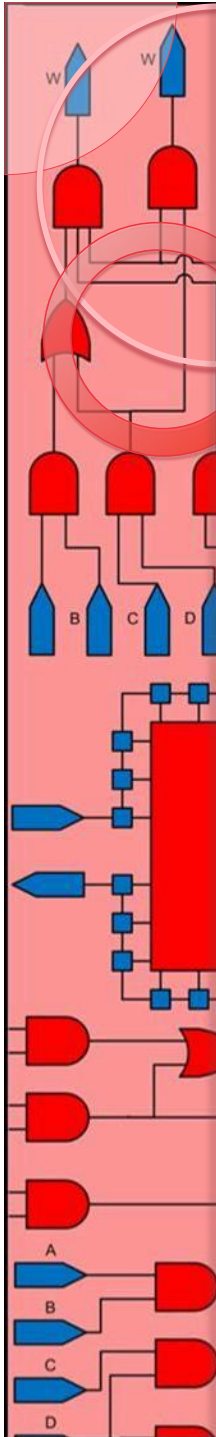
```
`timescale 1ns/1ns

module nand2_1d ( input a, b, output y);
wire im1;
supply1 vdd;
supply0 gnd;
    nmos #(3, 4)
        g4 (im1, gnd, b),
        g3 (y, im1, a);

    pmos #(4, 5)
        g1 (y, vdd, a),
        g2 (y, vdd, b);
endmodule
```



CMOS NAND Verilog Description



Transistor (Switch) Level Description

- CMOS NAND Example

```
`timescale 1ns/1ns

module nand2Tester ();
    reg ai=0, bi=0;
    wire yo;

    nand2_1d MUT ( ai, bi, yo );

    initial begin
        #25; ai=0; #25; bi=0;
        #25; ai=0; #25; bi=1;
        #25; ai=1; #25; bi=0;
        #25; ai=1; #25; bi=1;
        #25; ai=0; #25; bi=1;
        #25; $stop;
    end
endmodule
```

```
`timescale 1ns/1ns

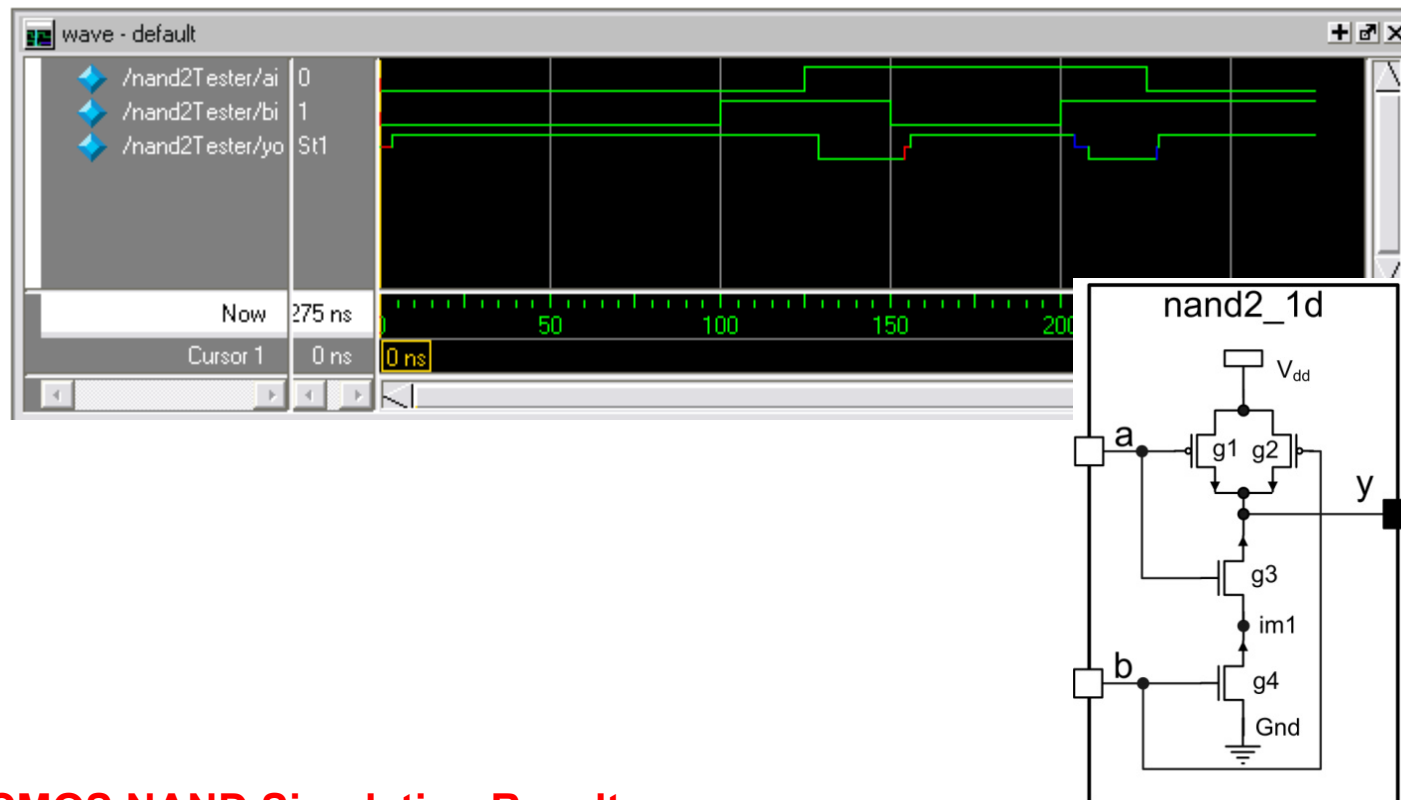
module nand2_1d ( input a, b, output y);
    wire im1;
    supply1 vdd;
    supply0 gnd;
    nmos #(3, 4)
        g4 (im1, gnd, b),
        g3 (y, im1, a);

    pmos #(4, 5)
        g1 (y, vdd, a),
        g2 (y, vdd, b);
endmodule
```

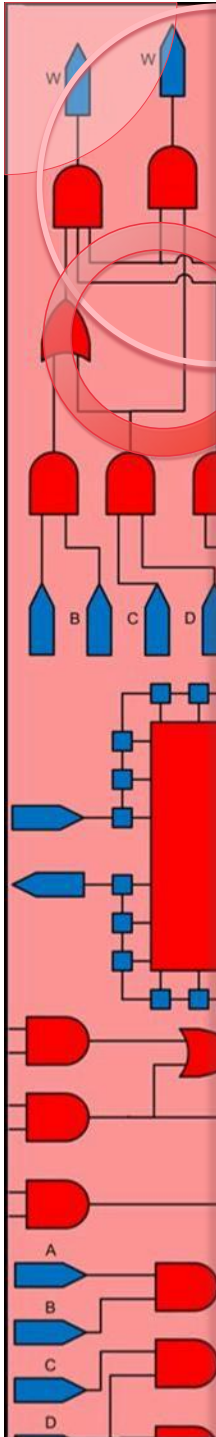
CMOS NAND Verilog Description

Transistor (Switch) Level Description

- CMOS NAND Example

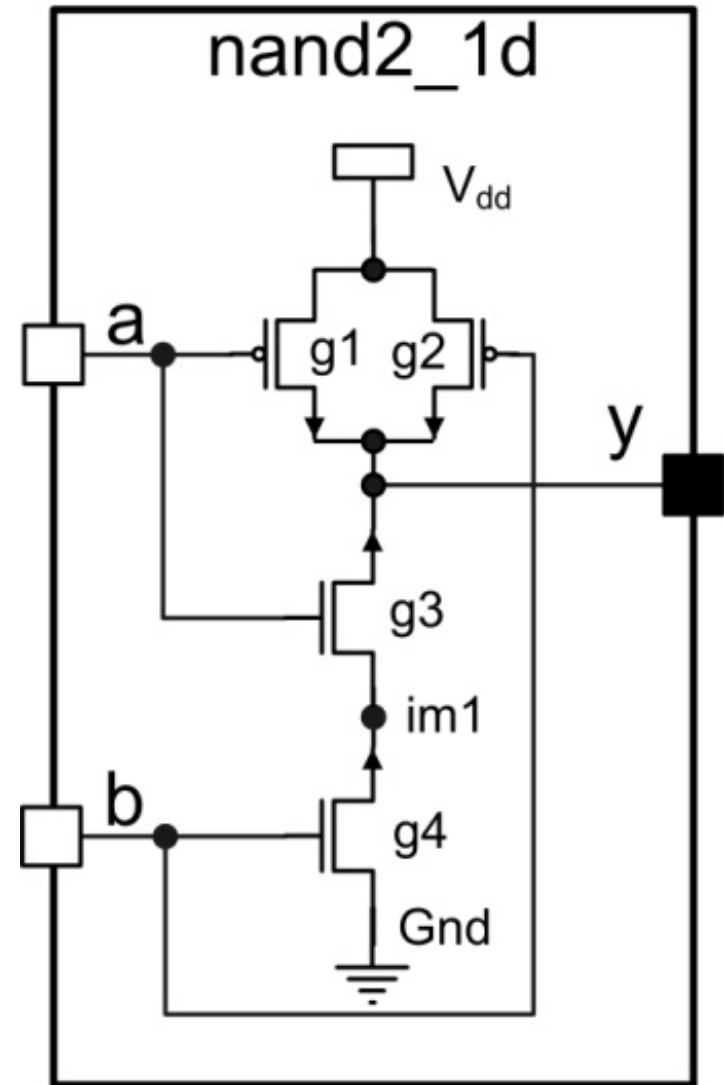


CMOS NAND Simulation Result

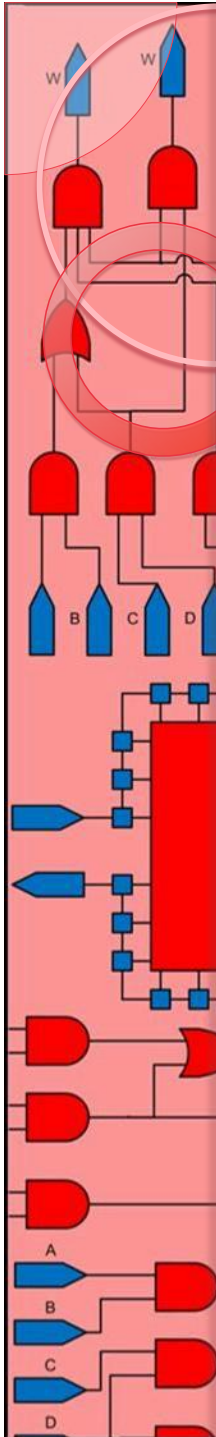


Transistor (Switch) Level Description

- Timing analysis

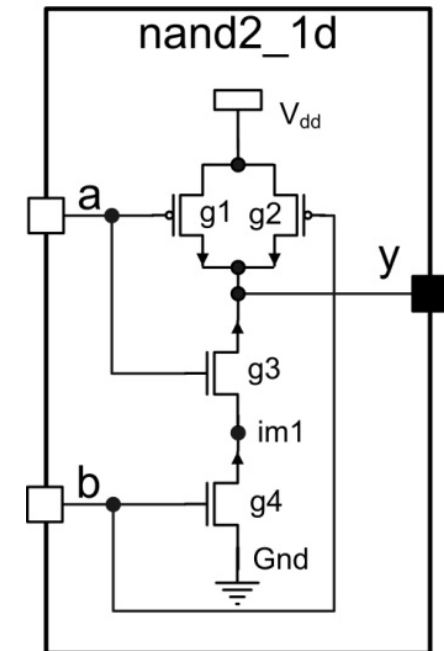


CMOS NAND nmos #(3,4); pmos #(4,5)

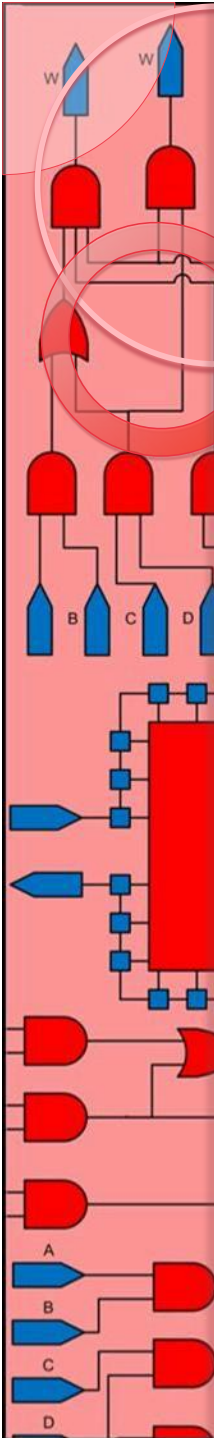


Transistor (Switch) Level Description

- Timing analysis, gate equivalent

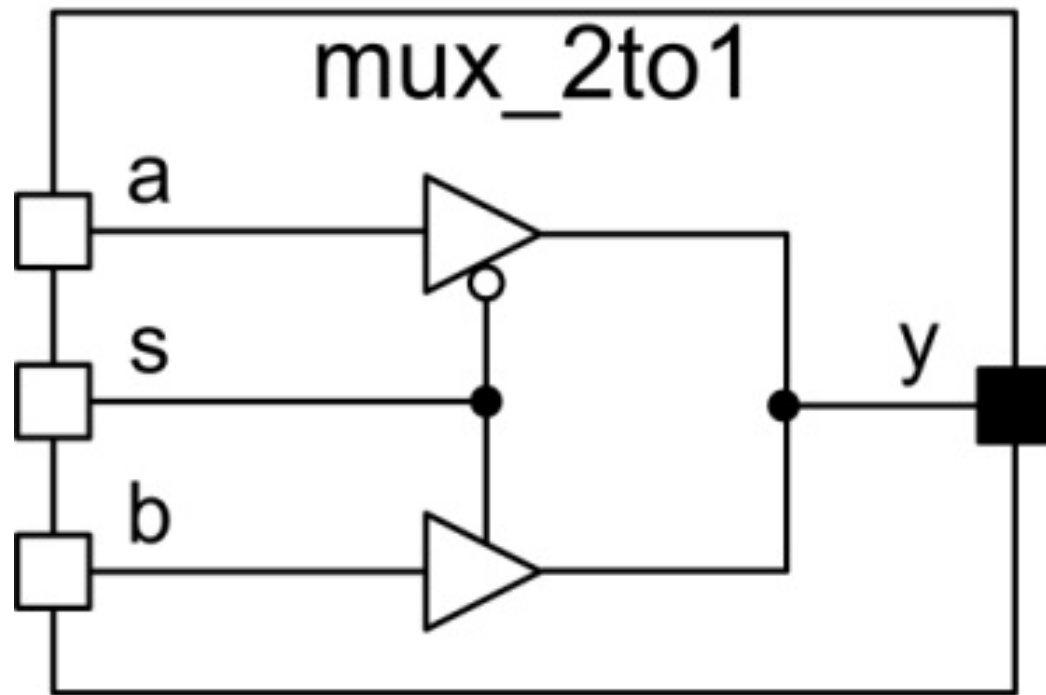


CMOS NAND nmos #(3,4); pmos #(4,5)



Three-state Logic

- Multiplexer Example



Combinational Circuits

- Multiplexer Example – use three-state gates

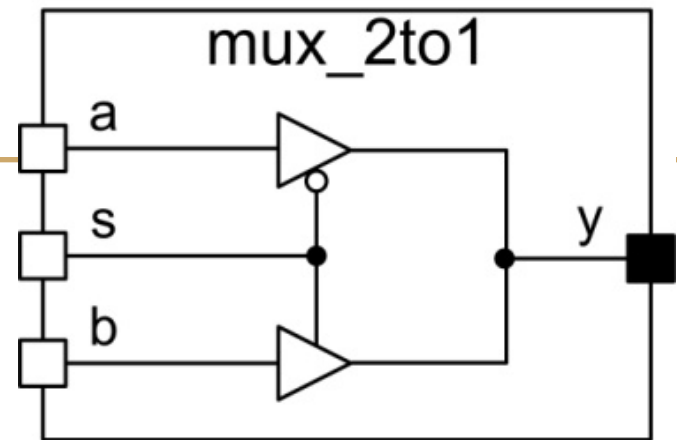
```
`timescale 1ns/1ns

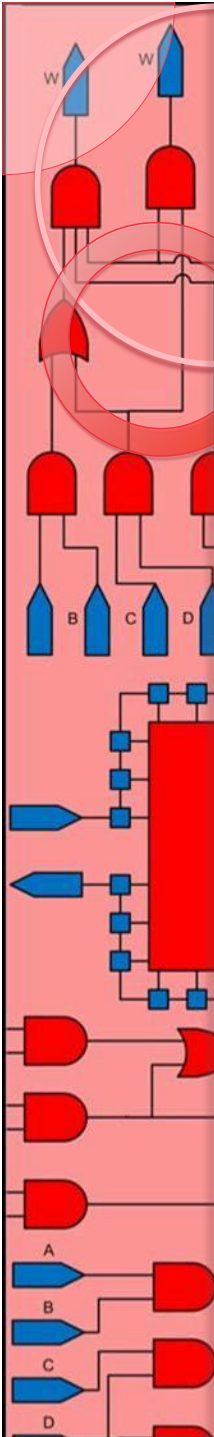
module mux_2to1 ( input a, b, s, output y );

    bufif1 #(3) ( y, b, s );
    bufif0 #(5) ( y, a, s );

endmodule
```

Multiplexer Verilog Code





Combinational Circuits

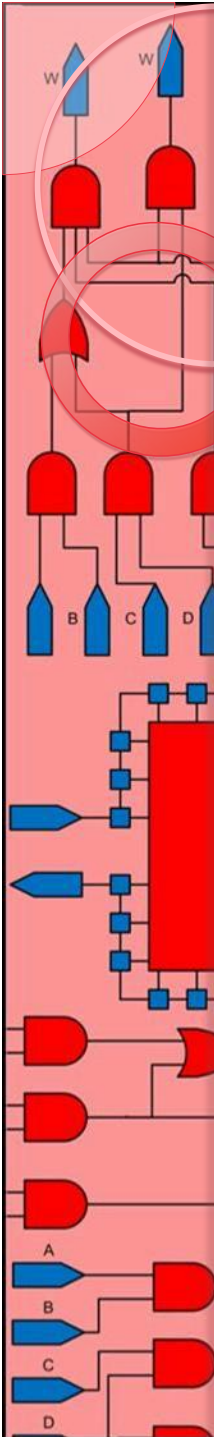
- Multiplexer Example

```
`timescale 1ns/1ns

module mux_2to1 (a, b, s, y);
    input a, b, s;
    output y;
    wire is;
    wire aa, bb;

    not U1 #(3,4) (is, s);
    and U2 #(3,4) (aa, a, is),
        U3 #(3,4) (bb, b, s);
    or U4 #(3,4) (y, aa, bb);

endmodule
```



Multiplexer Testbench

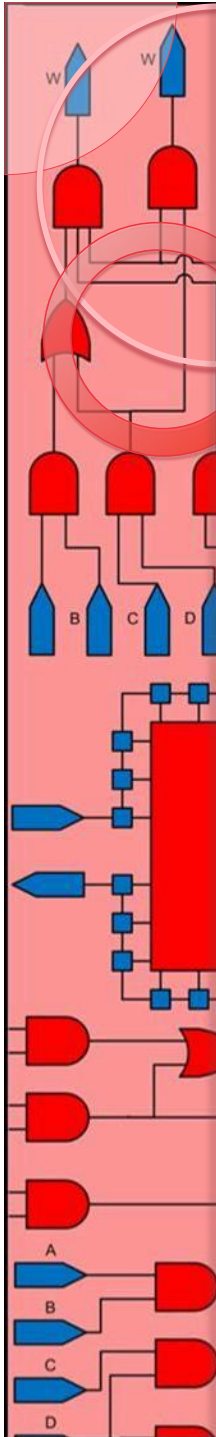
- Random data at some time intervals

```
`timescale 1ns/1ns

module muxTester ();
    reg ai=0, bi=0, ci=0;
    wire yo;

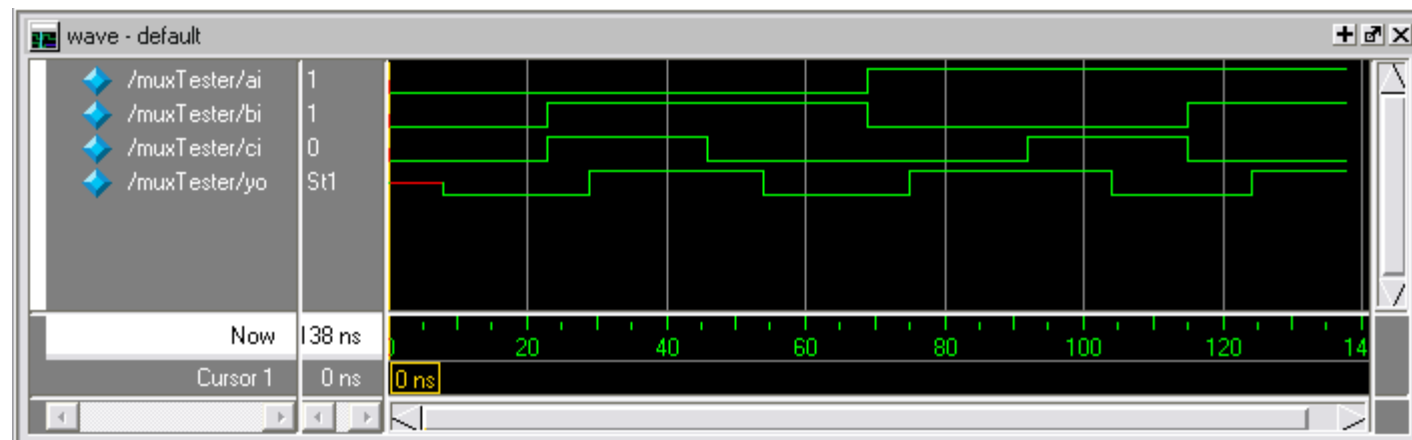
    mux_2to1 MUT ( ai, bi, ci, yo );

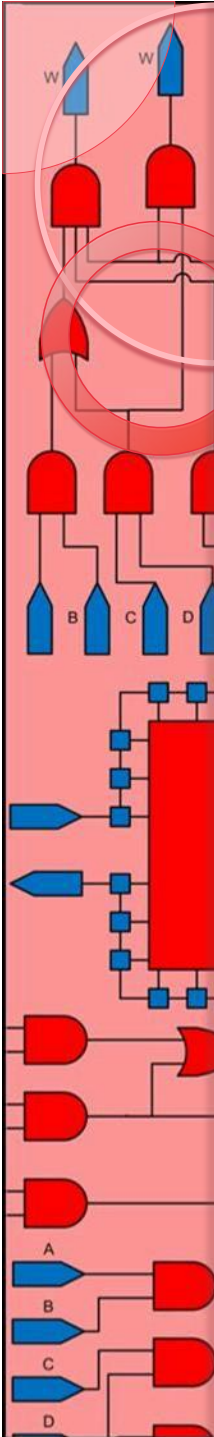
    initial begin
        #23; ai=0; bi=0; ci=1;
        #23; ai=1; bi=0; ci=1;
        #23; ai=0; bi=0; ci=0;
        #23; ai=1; bi=1; ci=1;
        #23; ai=0; bi=1; ci=0;
        #23; $stop;
    end
endmodule
```

Multiplexer Testbench

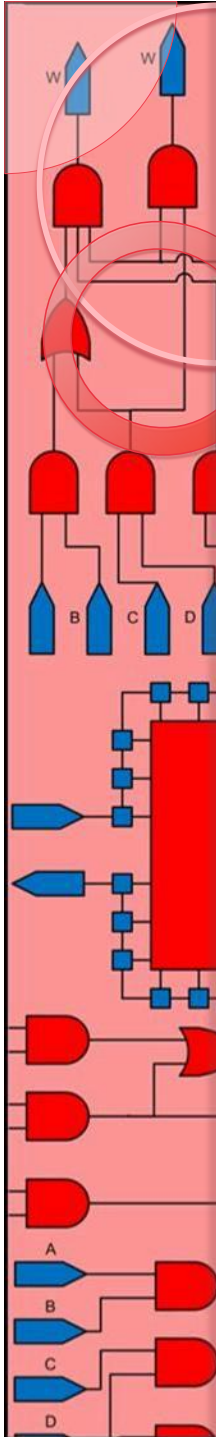
- Multiplexer output waveform





Summary

- Discussed the origin of delays
- Showed 4-value logic value system
- Performed switch level simulation
- Simulated gates with delay values
- ModelSim simulation run results were shown



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

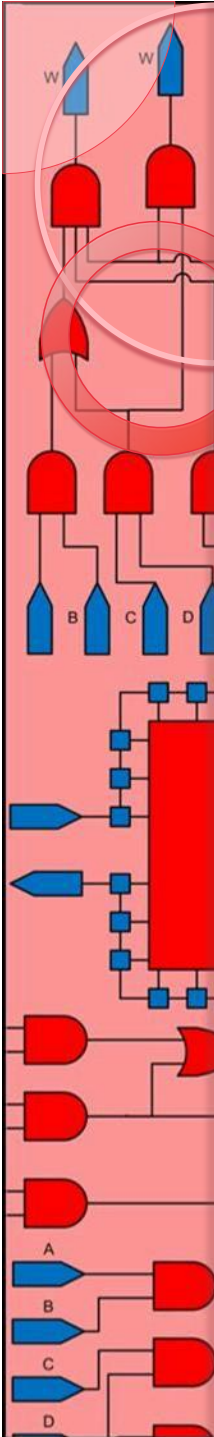
Basics of Digital Design at RT Level with Verilog

Lecture 4: Hierarchical Gate Level Design

July 2014

© 2013-2014 Zain Navabi

45

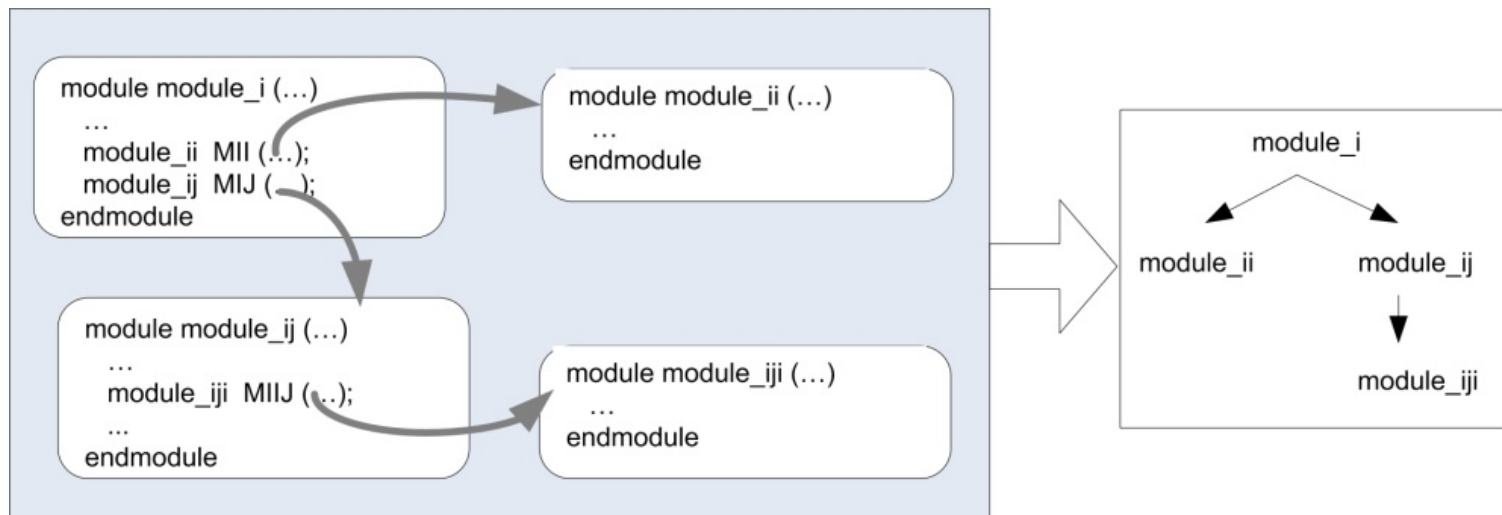


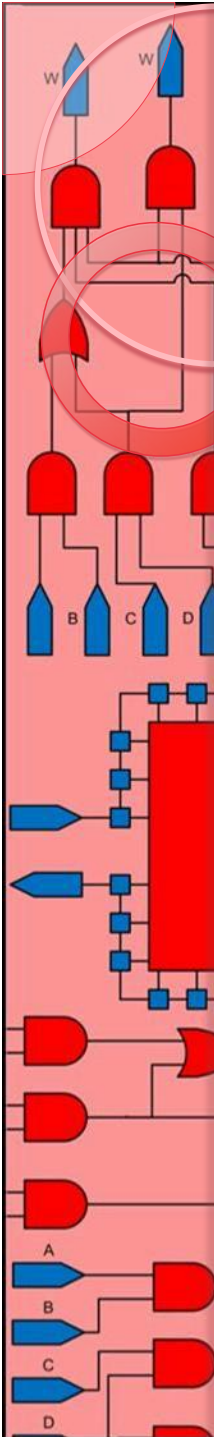
Hierarchical Gate Level Design

- Gate Level Design
- Module Instantiation
- Top Level Module
- More on Testbenches
- Summary

Hierarchical Gate Level Design

- Module Hierarchy





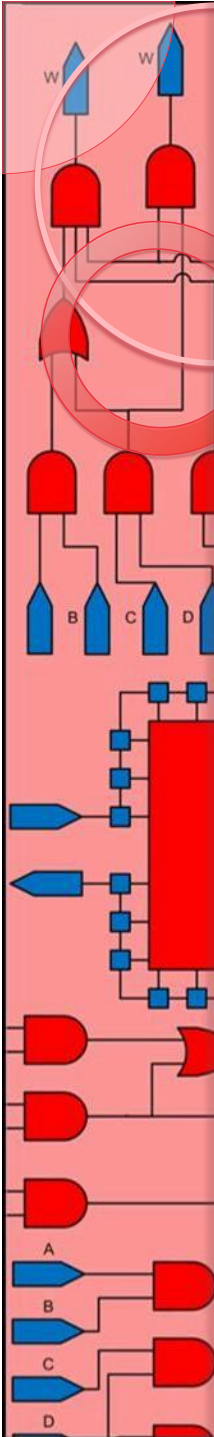
Hierarchical Gate Level Design

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);  
  wire c1;  
  nor g1 (c1, i1, i2);  
  and g2 (w1, c1, i3);  
  xor g3 (w2, i1, i2, i3);  
endmodule
```

```
module simple_1b (input i1, i2, i3, output w1, w2);  
  assign w1 = i3 & ~(i1 | i2);  
  assign w2 = i1 ^ i2 ^ i3;  
endmodule
```

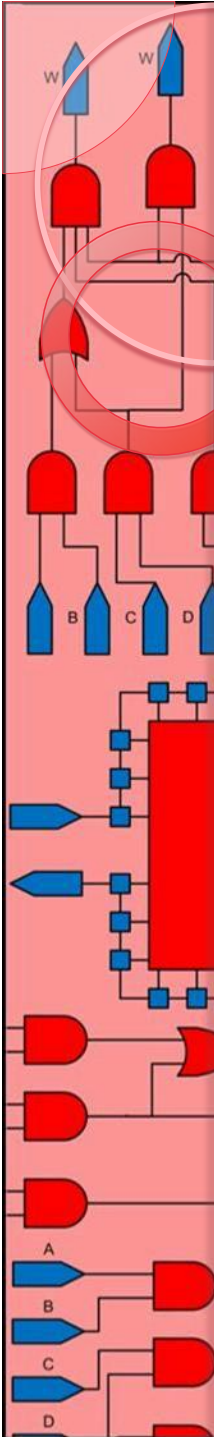
```
module simple_1c (input i1, i2, i3, output w1, w2);  
  reg w1, w2;  
  always @(i1, i2, i3) begin  
    if (i1 | i2 ) w1 = 0; else w1 = i3;  
    w2 = i1 ^ i2 ^ i3;  
  end  
endmodule
```



Hierarchical Gate Level Design

- Module Instantiation

```
module a_hierarchical_circuit (input a, b, output w);  
  // a comment  
  // wires and variable declarations  
  // operation of the circuit  
  
  AnotherTestedModule UUT (port connection according to module being instantiated);  
  // any number of the above  
  . . .  
endmodule
```



Basic Structures of Verilog

- Module Instantiation
 - Top level design instantiates Majority and Mux

Combinational Circuits

- Majority Example

```
`timescale 1ns/1ns

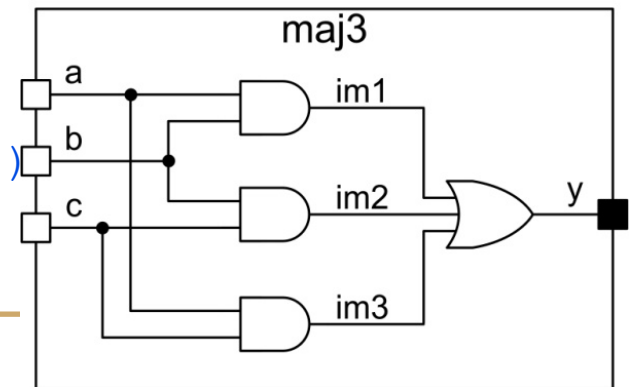
module maj3 ( input a, b, c, output y );

    wire im1, im2, im3;

    and U1 #(4,5) ( im1, a, b );
    and U2 #(4,5) ( im2, b, c );
    and U3 #(4,5) ( im3, c, a );
    or  U4 #(4,5) ( y, im1, im2, im3 );

endmodule
```

Verilog Code for the Majority Circuit



Combinational Circuits

- Where delays come from

```
`timescale 1ns/1ns
module some_circuit ( input a, b, c, output y );
    . . .
    nand2_1d U1 #(6,8) ( im1, a, b );
    . . .
endmodule
```

```
`timescale 1ns/1ns

module nand2_1d ( input a, b, output y);
wire im1;
supply1 vdd;
supply0 gnd;
    nmos #(3, 4)
        g4 (im1, gnd, b),
        g3 (y, im1, a);

    pmos #(4, 5)
        g1 (y, vdd, a),
        g2 (y, vdd, b);
endmodule
```

Combinational Circuits

- Multiplexer Example – use three-state gates

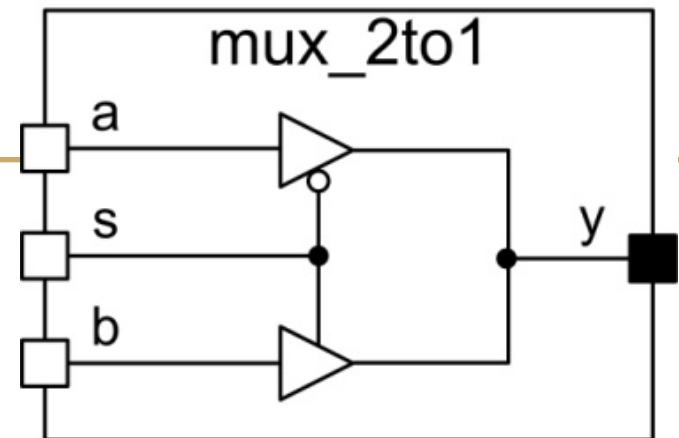
```
`timescale 1ns/1ns

module mux_2to1 ( input a, b, s, output y );

    bufif1 #(3) ( y, b, s );
    bufif0 #(5) ( y, a, s );

endmodule
```

Multiplexer Verilog Code





Hierarchical Gate Level Design

- Multiplexer Example

```
`timescale 1ns/1ns

module RateSelector (input a, b, c, d, e, output w);

    wire mj;

    maj3 U1 (c, d, e, mj);

    mux_2to1 U2 (a, b, mj, w);

endmodule
```

Top level module



Hierarchical Gate Level Design

- Multiplexer Example

```
`timescale 1ns/1ns

module RateSelector (input a, b, c, d, e, output w);

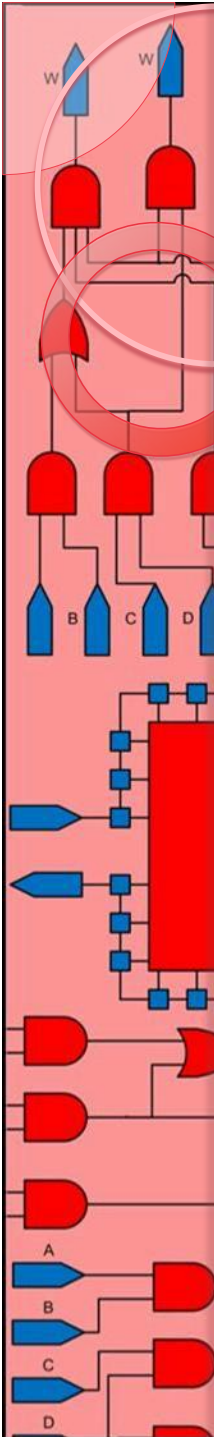
    wire mj;

    maj3 U1 (c, d, e, mj);

    mux_2to1 U2 (a, b, mj, w);

endmodule
```

Top level module



Hierarchical Gate Level Design

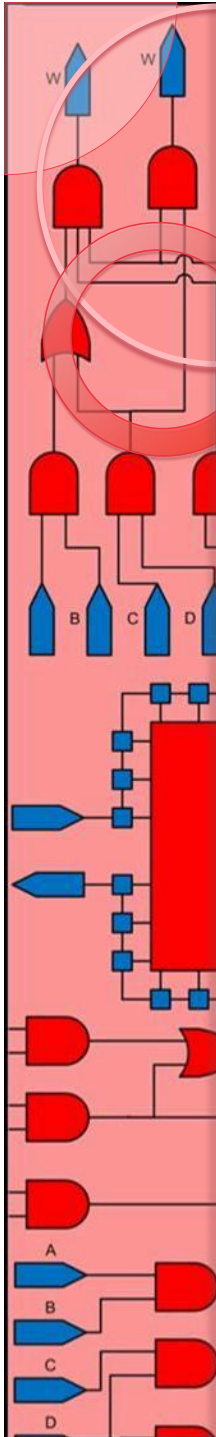
- Periodic data application

```
`timescale 1ns/1ns

module RateSelectoreTester ();
    reg ai=0, bi=0, ci=0, di=0, ei=0;
    wire yo;

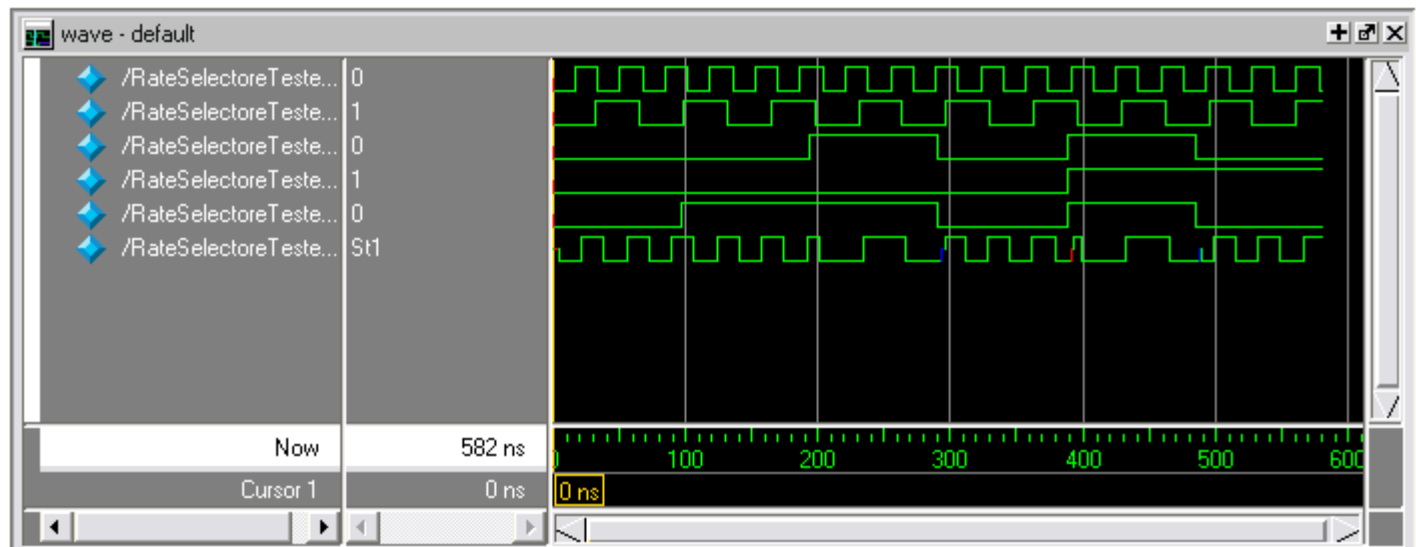
    RateSelector MUT ( ai, bi, ci, di, ei, yo );

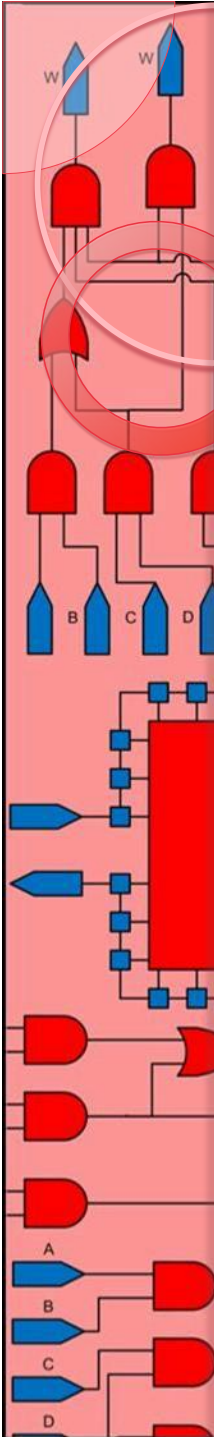
    always #17 ai = ~ai;
    always #33 bi = ~bi;
    initial begin
        #97; ci=0; di=0; ei=1;
        #97; ci=1; di=0; ei=1;
        #97; ci=0; di=0; ei=0;
        #97; ci=1; di=1; ei=1;
        #97; ci=0; di=1; ei=0;
        #97; $stop;
    end
endmodule
```



Hierarchical Gate Level Design

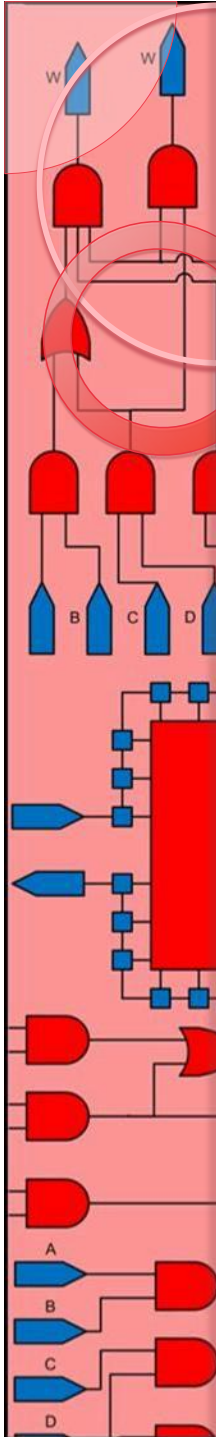
- Selecting one of the two frequencies





Summary

- Talked about module hierarchy
- Any design can have a hierarchy and instantiation of other modules
- The design remains at the gate level
- Talked about more constructs to use in a testbench



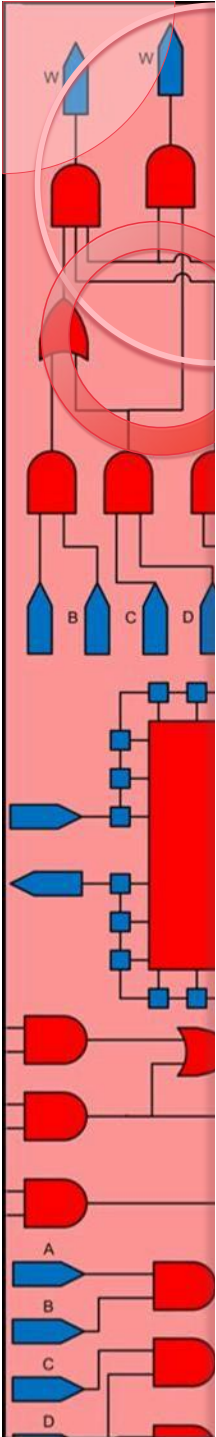
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

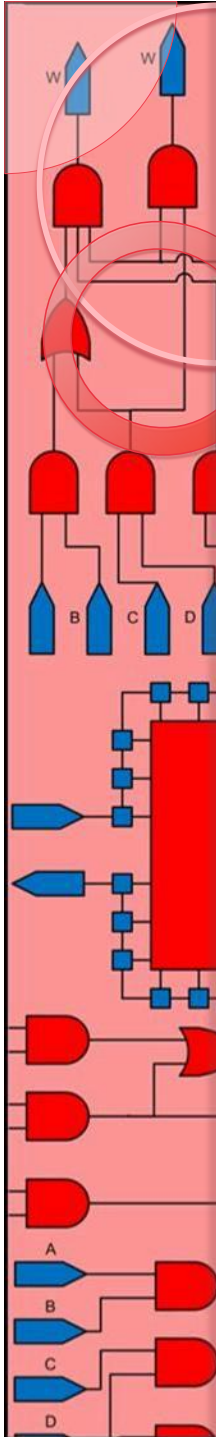
Basics of Digital Design at RT Level with Verilog

Lecture 5: Using Expressions in Modules



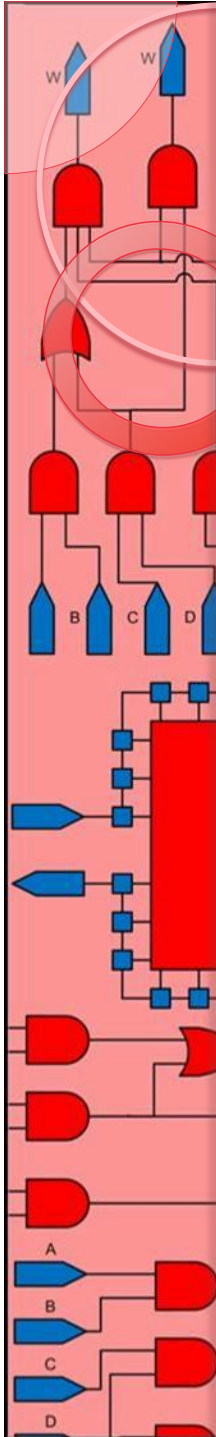
Using Expressions in Modules

- Boolean expressions
- Two familiar examples
- Module Instantiation
- Top Level Module
- More on Testbenches
- Summary



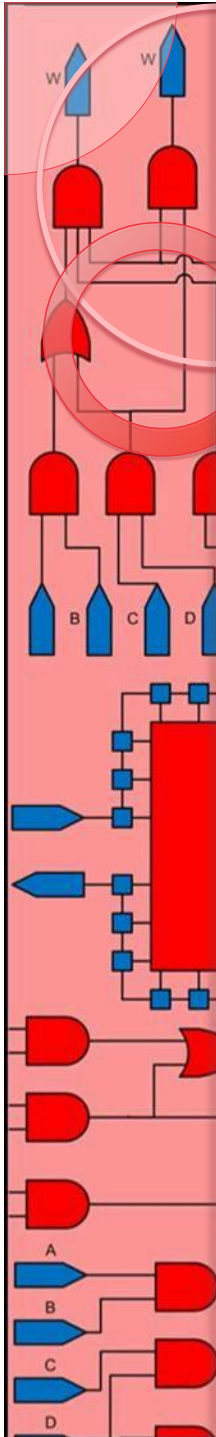
Boolean Expressions

- Use Verilog **assign** statement
- **assign** statements in a module are considered as logic blocks
- Position in module code is irrelevant
- Use the following notations:
 - For AND use: $\&$
 - For OR use: $|$
 - For NOT use: \sim
 - For XOR use: \wedge
 - Use parenthesis for enforcing precedence



Boolean Expressions

- Use Verilog **assign** statement
- Format:
`assign #(0 to I_Delay, Ito0_Delay) w = a & b & c;`
`assign #(Delay) w = a & b & c;`
`assign #Delay w = a & b & c;`
`assign w = a & b & c;`
- Optional delay parameters
- Output is w
- Inputs are a, b, c



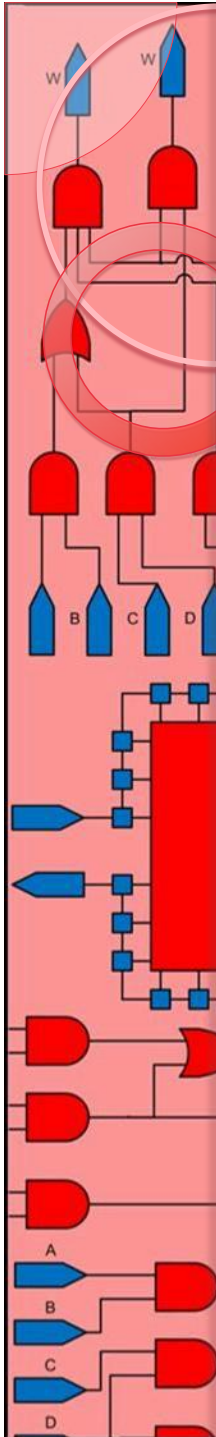
Boolean Expressions

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);
  wire c1;
  nor g1 (c1, i1, i2);
  and g2 (w1, c1, i3);
  xor g3 (w2, i1, i2, i3);
endmodule
```

```
module simple_1b (input i1, i2, i3, output w1, w2);
  assign w1 = i3 & ~(i1 | i2);
  assign w2 = i1 ^ i2 ^ i3;
endmodule
```

```
module simple_1c (input i1, i2, i3, output w1, w2);
  reg w1, w2;
  always @(i1, i2, i3) begin
    if (i1 | i2 ) w1 = 0; else w1 = i3;
    w2 = i1 ^ i2 ^ i3;
  end
endmodule
```



Boolean Expressions

- From KM
 - A circuit that produces a 1 if $abcd$ is divisible by 3 or 4
 - Input is $abcd$ and treated as a 4-bit binary number
 - Assume non-zero input

c,d \ a,b	a,b			
	00	01	11	10
00	-	1	1	1
01				1
11	1		1	
10		1		

W

$$w = c'.d' + a.b'.c' + a'.b'.c.d + a'.b.d' + a.b.c.d$$

$$w = (\sim c \ \& \ \sim d) \mid (a \ \& \ \sim b \ \& \ \sim c) \mid (\sim a \ \& \ \sim b \ \& \ c \ \& \ d) \mid (\sim a \ \& \ b \ \& \ \sim d) \mid (a \ \& \ b \ \& \ c \ \& \ d);$$

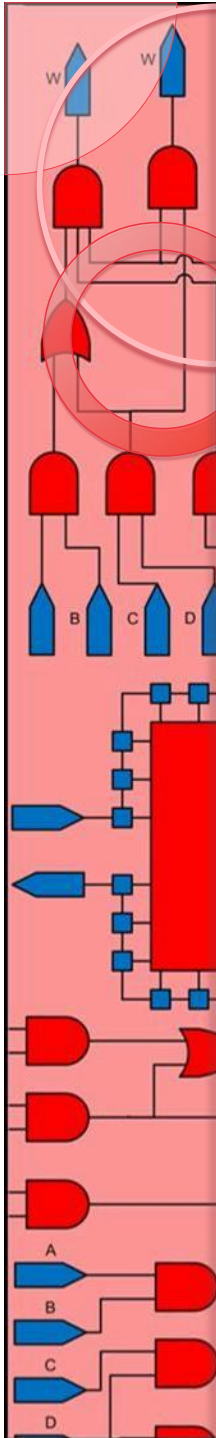


Boolean Expressions

$$w = c'.d' + a.b'.c' + a'.b'.c.d + a'.b.d' + a.b.c.d$$

$$w = (\sim c \ \& \ \sim d) \mid (a \ \& \ \sim b \ \& \ \sim c) \mid (\sim a \ \& \ \sim b \ \& \ c \ \& \ d) \mid (\sim a \ \& \ b \ \& \ \sim d) \mid (a \ \& \ b \ \& \ c \ \& \ d);$$

```
module div3or4 (input a, b, c, d, output w);  
  
    assign w = (~c & ~d) |  
               (a & ~b & ~c) |  
               (~a & ~b & c & d) |  
               (~a & b & ~d) |  
               (a & b & c & d);  
  
endmodule
```



Boolean Expressions

- From KM
 - A circuit that produces a 1 if $abcd$ is divisible by 3 or 5
 - Input is $abcd$ and treated as a 4-bit binary number
 - It turns out that output is 1 if even number of 1's are in $abcd$

c,d \ a,b	a,b			
	00	01	11	10
00	1		1	
01		1		1
11	1		1	
10		1		1

w

$$w = a' \oplus b \oplus c \oplus d$$

$$w = \sim a \wedge b \wedge c \wedge d$$



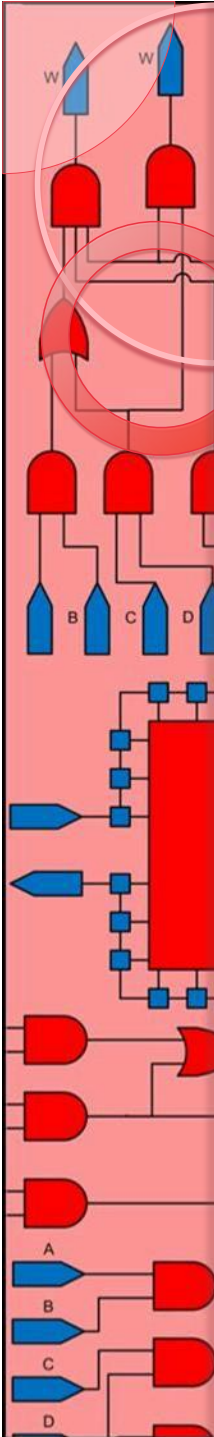
Boolean Expressions

- Module Description

$$w = c'.d' + a.b'.c' + a'.b'.c.d + a'.b.d' + a.b.c.d$$

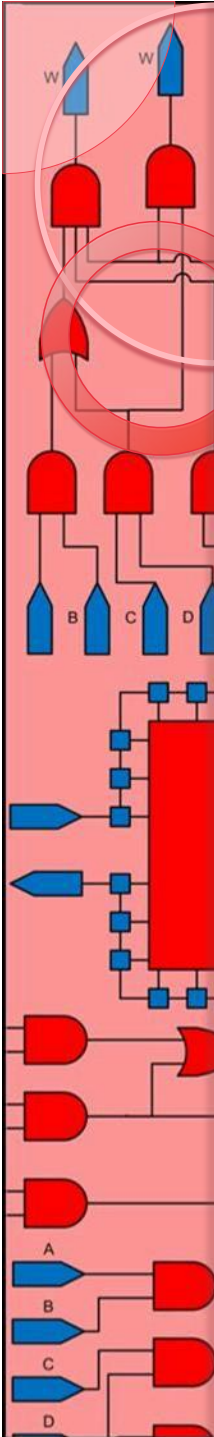
$$w = (\sim c \ \& \ \sim d) \mid (a \ \& \ \sim b \ \& \ \sim c) \mid (\sim a \ \& \ \sim b \ \& \ c \ \& \ d) \mid (\sim a \ \& \ b \ \& \ \sim d) \mid (a \ \& \ b \ \& \ c \ \& \ d);$$

```
module div3or5 (input a, b, c, d, output w);  
  
    assign #(68,70) w = ~a ^ b ^ c ^ d;  
  
endmodule
```



Boolean Expressions

- Where delays come from?



Two Familiar Examples

- Module Instantiation
 - Top level design instantiates Majority and Mux

Two Familiar Examples

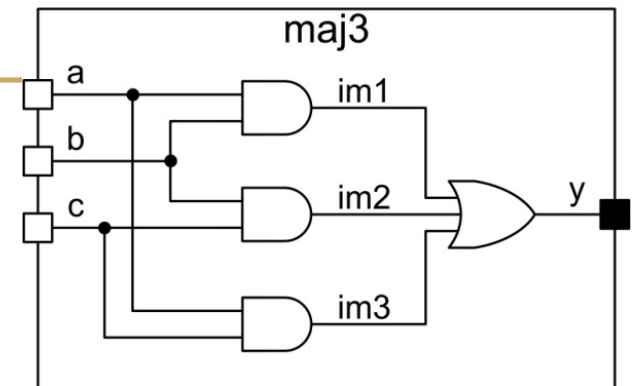
- Majority Example

```
`timescale 1ns/1ns

module maj3 ( input a, b, c, output y );

    assign y = (a & b) | (a & c) | (b & c);

endmodule
```



Verilog Code for the Majority Circuit

Two Familiar Examples

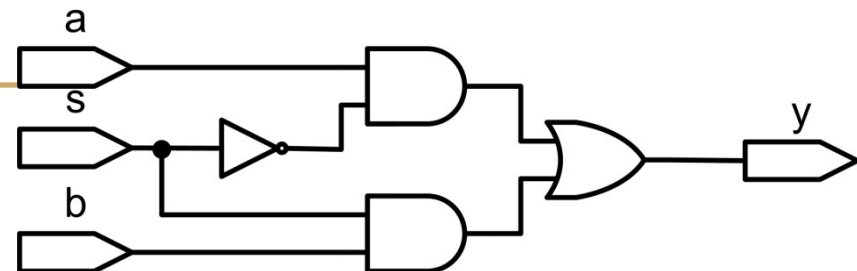
- Multiplexer Example – Use AND OR gates

```
`timescale 1ns/1ns

module mux_2to1 ( input a, b, s, output y );

    assign #(5) y = (a & ~s) | (b & s);

endmodule
```



Verilog Code for the Multiplexer

Two Familiar Examples

- Multiplexer Example – Use three-state gates

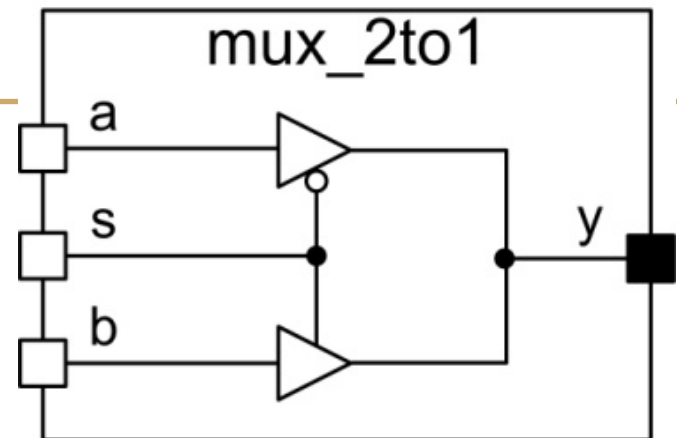
```
`timescale 1ns/1ns

module mux_2to1 ( input a, b, s, output y );

    assign #(5) y = ~s ? a : b;

endmodule
```

Multiplexer Verilog Code





Hierarchical Expression Level Design

- Rate Selector Example

```
`timescale 1ns/1ns

module RateSelector (input a, b, c, d, e, output w);

    wire mj;

    maj3 U1 (c, d, e, mj);

    mux_2to1 U2 (a, b, mj, w);

endmodule
```

Top level module



Multiple Expressions

- Rate Selector Example

```
`timescale 1ns/1ns

module RateSelector (input a, b, c, d, e, output w);

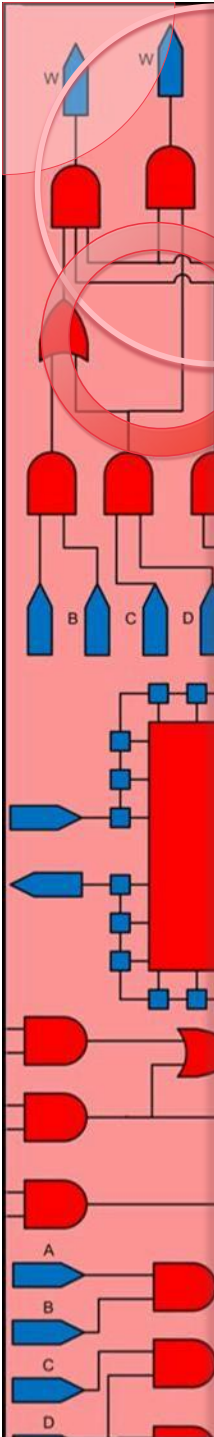
    wire mj;

    assign mj = (c & d) | (c & e) | (d & e);

    assign #(5) w = (a & ~mj) | (b & mj);

endmodule
```

Module with two expressions



Hierarchical Expression Level Design

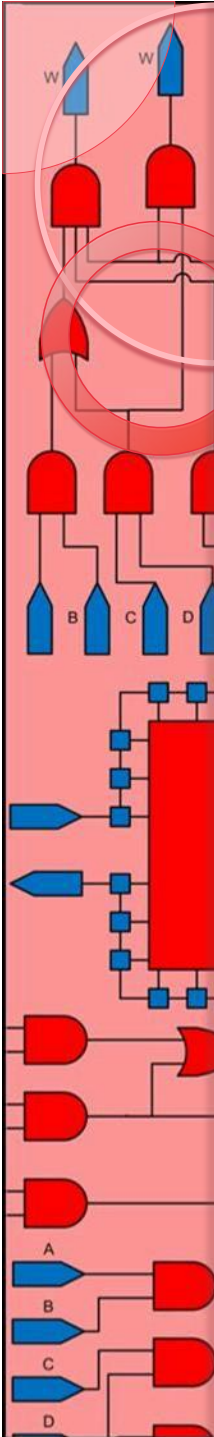
- Periodic data application

```
`timescale 1ns/1ns

module RateSelectoreTester ();
    reg ai=0, bi=0, ci=0, di=0, ei=0;
    wire yo;

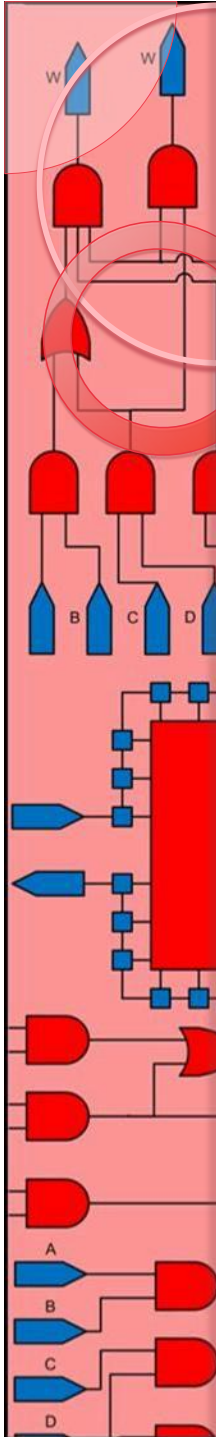
    RateSelector MUT ( ai, bi, ci, di, ei, yo );

    always #17 ai = ~ai;
    always #33 bi = ~bi;
    initial begin
        #97; ci=0; di=0; ei=1;
        #97; ci=1; di=0; ei=1;
        #97; ci=0; di=0; ei=0;
        #97; ci=1; di=1; ei=1;
        #97; ci=0; di=1; ei=0;
        #97; $stop;
    end
endmodule
```



Summary

- Expression in Verilog
- Assignments from Boolean
- Use of assign statements
- Multiple assignments
- Delay values
- Two versions of Rate Selector



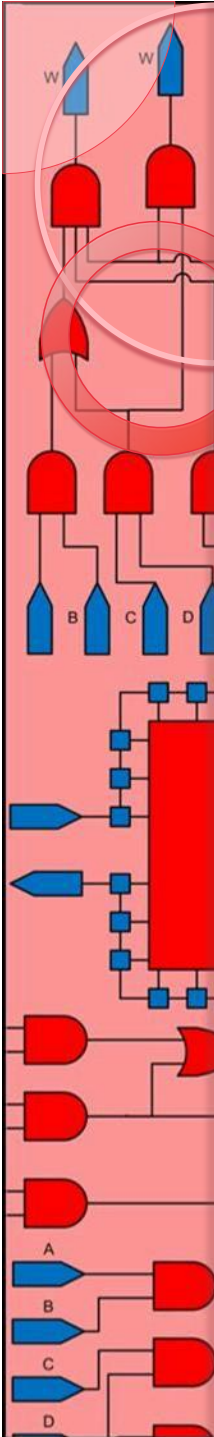
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

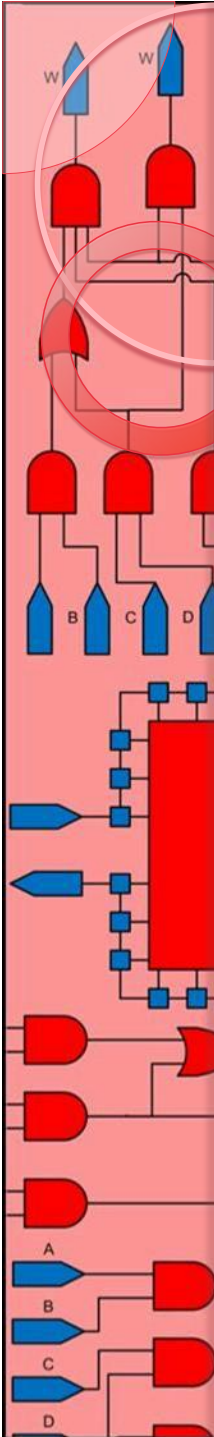
Basics of Digital Design at RT Level with Verilog

Lecture 6: Timing and Dynamic Hazards



Timing and Static Hazards

- Hazards
- Multiplexer Example
- Hazards on Karnaugh Maps
- Preventing Hazards
- Four Variable Example
- Summary



Hazards

- Change in one input
- 1-Hazard
 - Should be 1, but a 0-glitch
- 0-Hazard
 - Should be 0, but a 1-glitch

Multiplexer Example

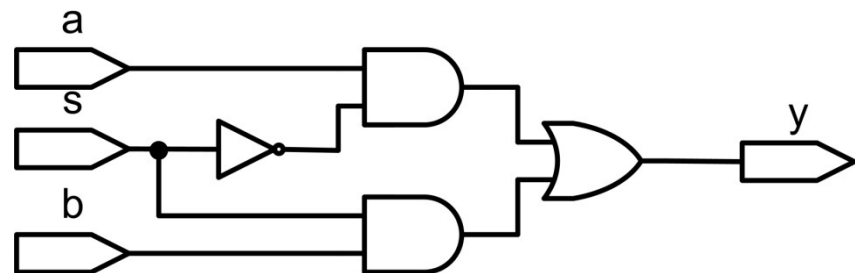
- Multiplexer Example – Using AND OR gates

```
`timescale 1ns/1ns

module mux2to1 ( input a, b, s, output y );
    wire sn, asn, bs;

    not #7 (sn, s);
    and #4 (asn, a, sn);
    and #4 (bs, b, s);
    or #3 (y, asn, bs);

endmodule
```



Verilog Code for the Multiplexer

Multiplexer Example

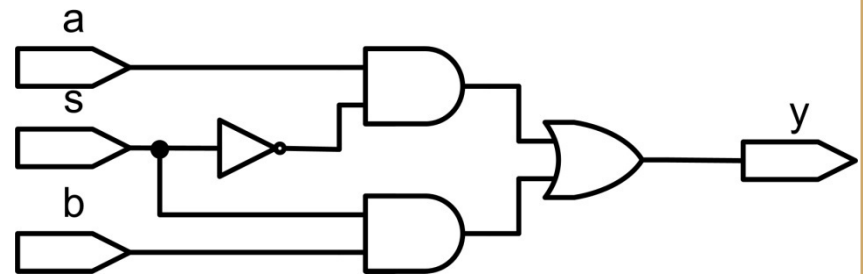
- Multiplexer Testbench

```
`timescale 1ns/1ns

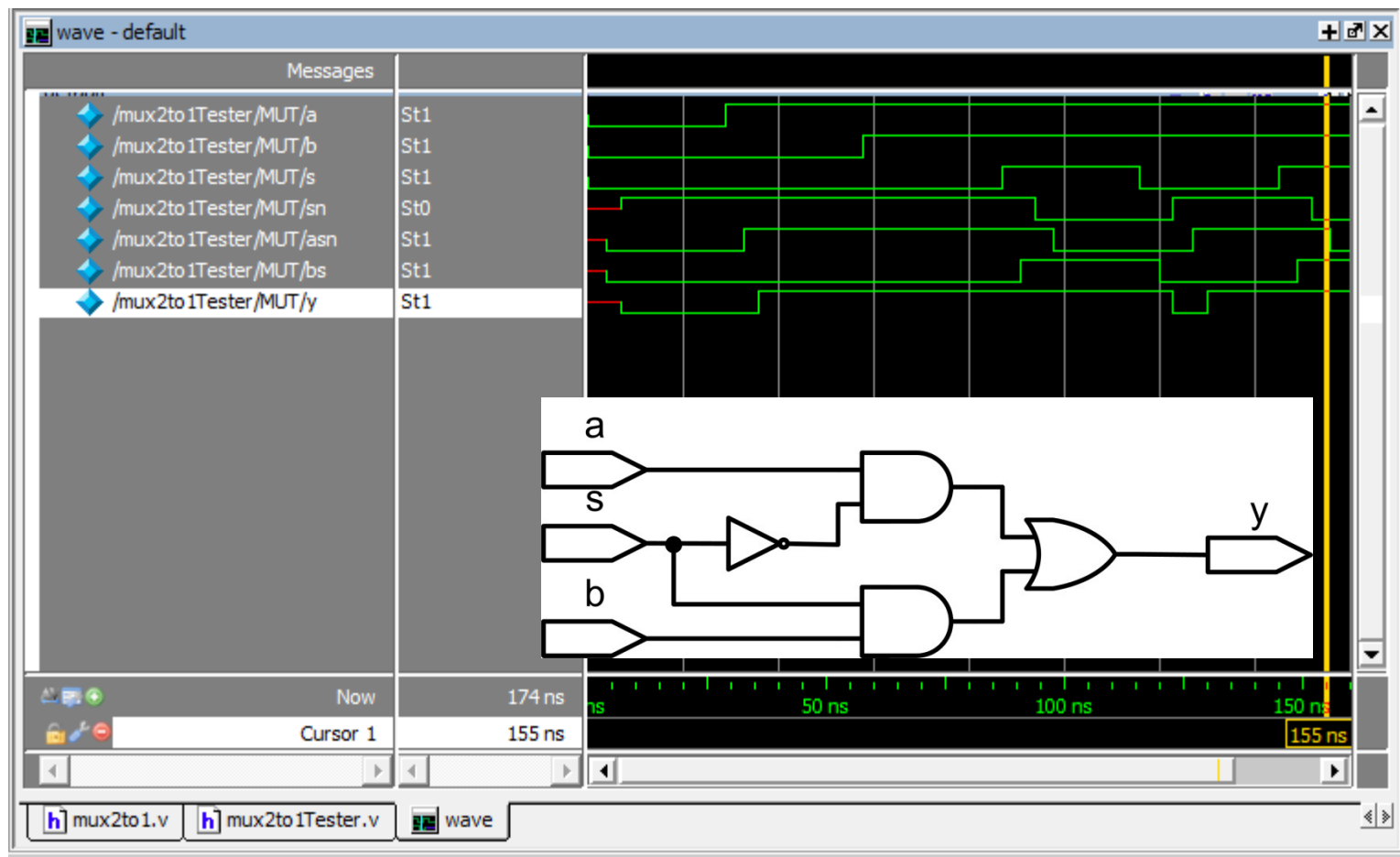
module mux2to1Tester ();
    reg ai=0, bi=0, si=0;
    wire yo;

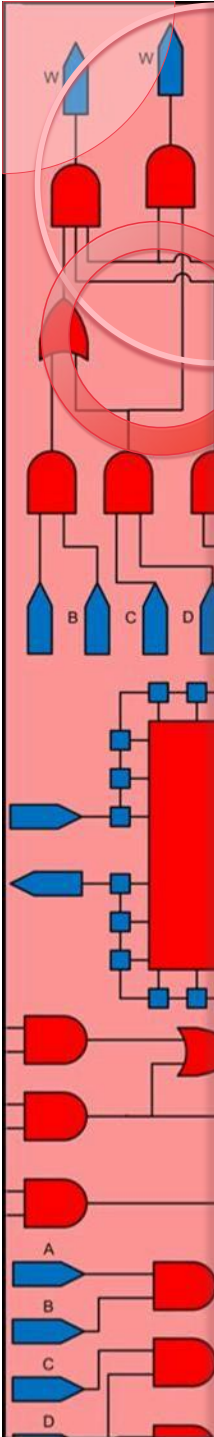
    mux_2to1 MUT ( ai, bi, si, yo );

    initial begin
        #29; ai=1; bi=0; si=0;
        #29; ai=1; bi=1; si=0;
        #29; ai=1; bi=1; si=1;
        #29; ai=1; bi=1; si=0;
        #29; ai=1; bi=1; si=1;
        #29; $stop;
    end
endmodule
```



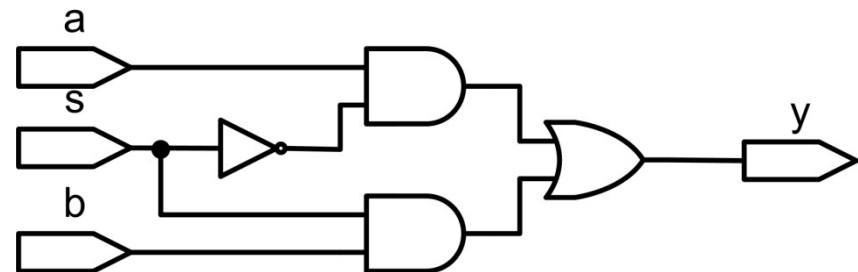
Multiplexer Example

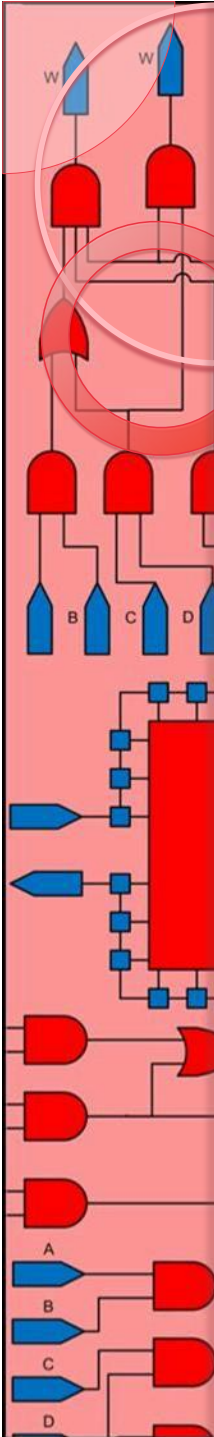




Multiplexer Example

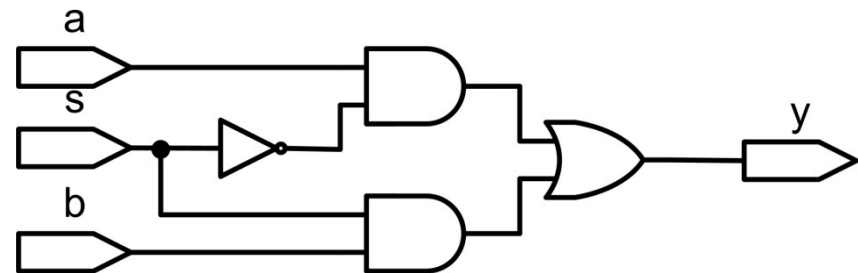
- Hazard Correction K-Map

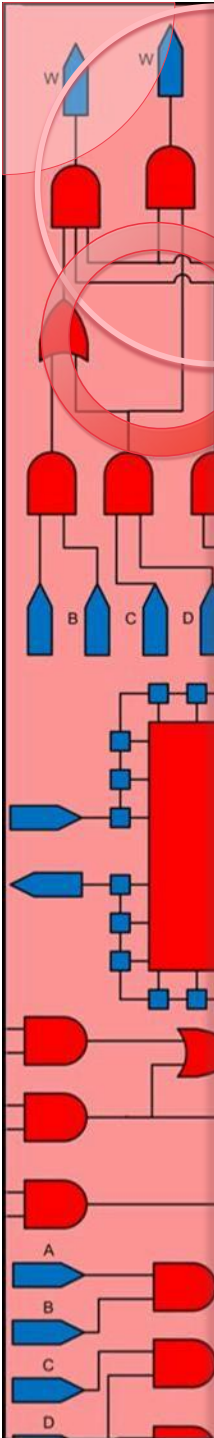




Preventing Hazards

- Map adjacent 1's in AND-OR circuits
- Map adjacent 0's in OR-AND circuits

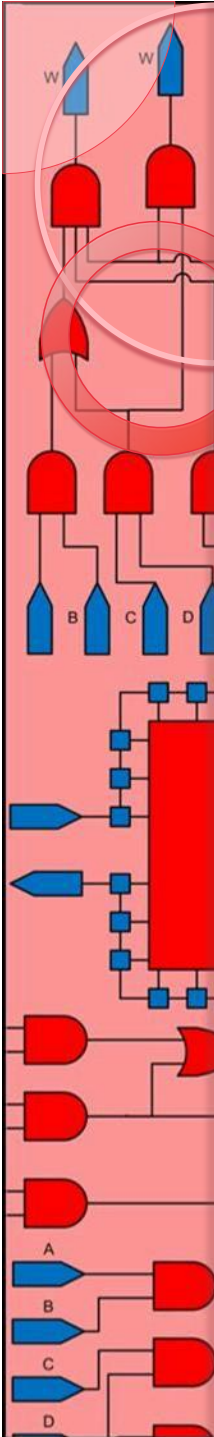




Four Variable Maps

- From KM
 - Show circuit
 - Add gates to remove hazards

a,b \ c,d	00	01	11	10
00	1	1	1	1
01			1	1
11		1	1	1
10		1		

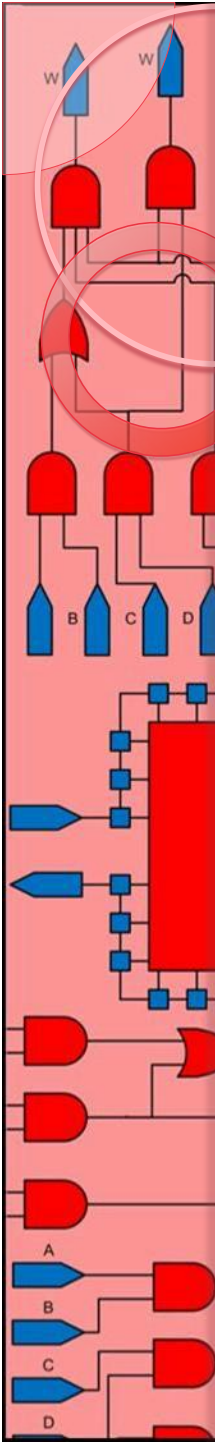


Four Variable Maps

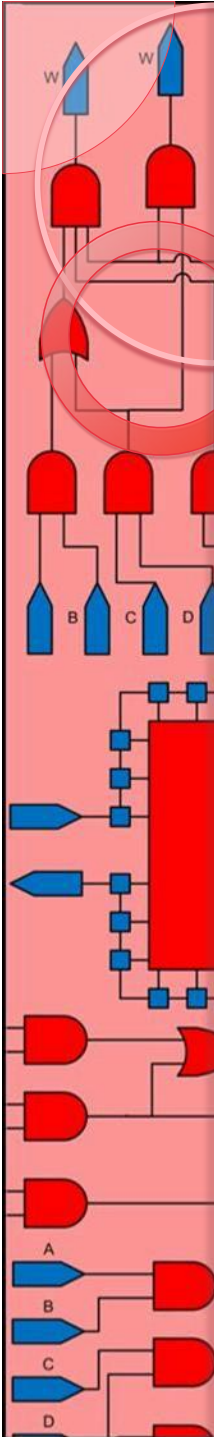
- From KM
 - Show circuit
 - Add gates to remove hazards

a,b \ c,d	00	01	11	10
00	1	1	1	1
01			1	1
11		1	1	1
10		1		

Four Variable Maps

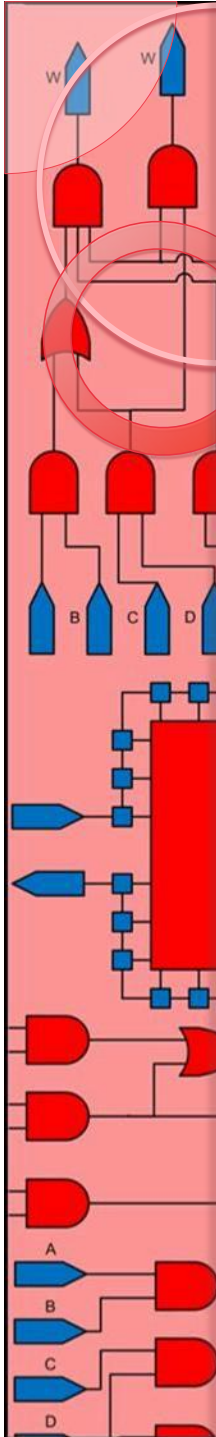


```
module HazardNotSeen (input a, b, c, d, output w);  
  
    assign w = (~c & ~d) |  
               (a & d) |  
               (~a & b & c);  
  
endmodule
```



Summary

- Hazards
- Multiplexer Example
- Hazards on Karnaugh Maps
- Preventing Hazards
- Four Variable Example
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

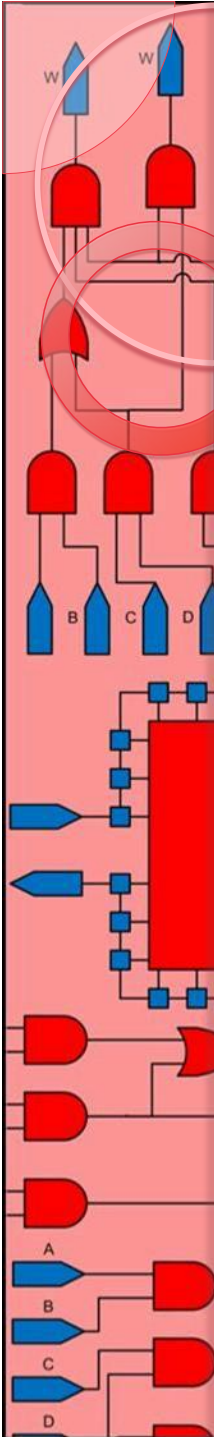
Basics of Digital Design at RT Level with Verilog

Lecture 7: Arithmetic Circuits

July 2014

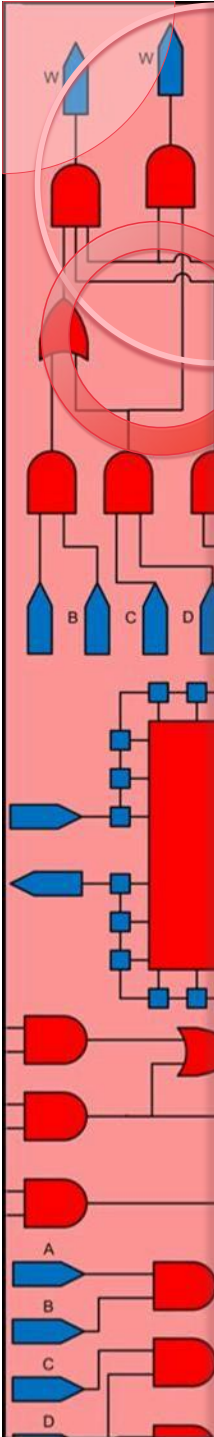
© 2013-2014 Zain Navabi

89



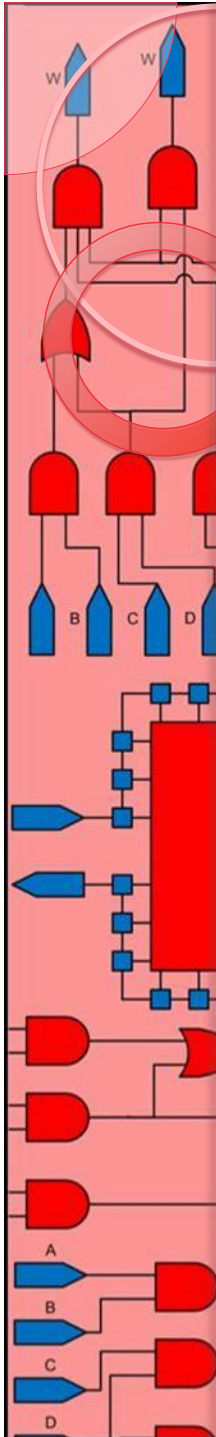
Arithmetic Circuits

- New operations
- Full adder
- Vector inputs
- Structural adder
- Adder using expressions
- A multi-function module
- More on Testbenches
- Summary



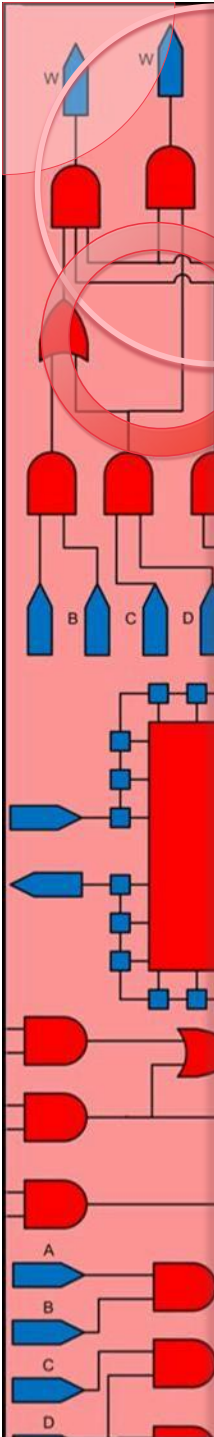
New Operations

- Use Verilog **assign** statement
- Position in module code is irrelevant
- Use the following notations:
 - For Adding use: +
 - For Subtracting use: -
 - For Comparing use: < > <= >= ==
 - For Condition use: ? :
 - For Concatenation use: { }
 - Use parenthesis for enforcing precedence



New Operation

- Use Verilog **assign** statement
- Format:
`assign w = a + b;`
`assign w = a - b`
`assign w = (f==1) ? y : z;`
`assign w = (a > b) ? a : b;`
- Vector format [7:0]
- Output is w and is a vector, e.g., w[7:0]
- Inputs are a, b, f
 - a and b are vectors
 - f is a scalar



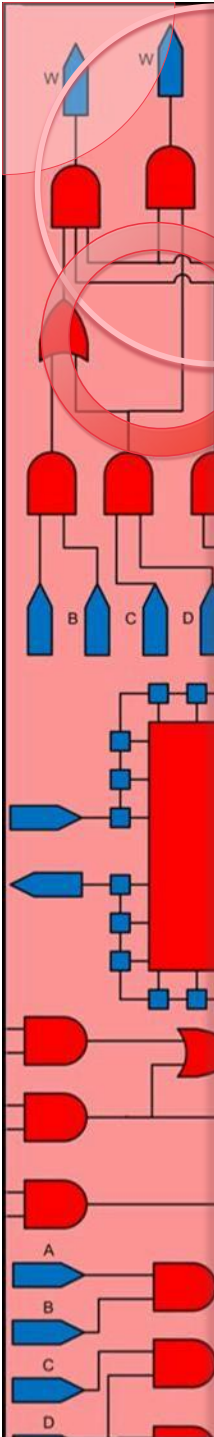
New Operations

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);
  wire c1;
  nor g1 (c1, i1, i2);
  and g2 (w1, c1, i3);
  xor g3 (w2, i1, i2, i3);
endmodule
```

```
module simple_1b (input i1, i2, i3, output w1, w2);
  assign w1 = i3 & ~(i1 | i2);
  assign w2 = i1 ^ i2 ^ i3;
endmodule
```

```
module simple_1c (input i1, i2, i3, output w1, w2);
  reg w1, w2;
  always @(i1, i2, i3) begin
    if (i1 | i2 ) w1 = 0; else w1 = i3;
    w2 = i1 ^ i2 ^ i3;
  end
endmodule
```



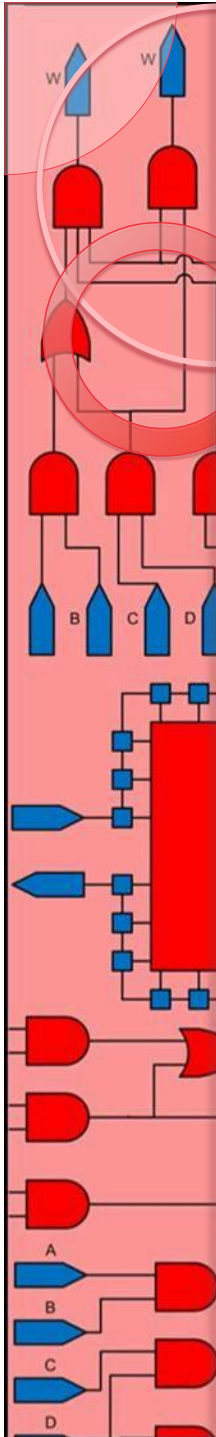
Full Adder

- Descriptions by Use of Equations
 - Sum output

```
`timescale 1ns/1ns

module xor3 ( input a, b, c, output y );
    assign y = a ^ b ^ c;
endmodule
```

XOR Verilog Code



Full Adder

- Descriptions by Use of Equations
 - Carry output

```
`timescale 1ns/1ns
```

```
module maj3 ( input a, b, c, output y );  
    assign y = (a & b) | (a & c) | (b & c);  
endmodule
```

Majority Verilog Code



Full Adder

- Descriptions by Use of Equations
- Full-Adder Example

```
`timescale 1ns/1ns

module fulladder ( input a, b, ci, output s, co );
    assign #(10) s = a ^ b ^ ci;
    assign #(8) co = ( a & b ) | ( b & ci ) | ( a &
ci );
endmodule
```

Full Adder Verilog Code

Structural Adder

- A simpler design of full-adder
- Using 8 full-adders to build an 8-bit adder

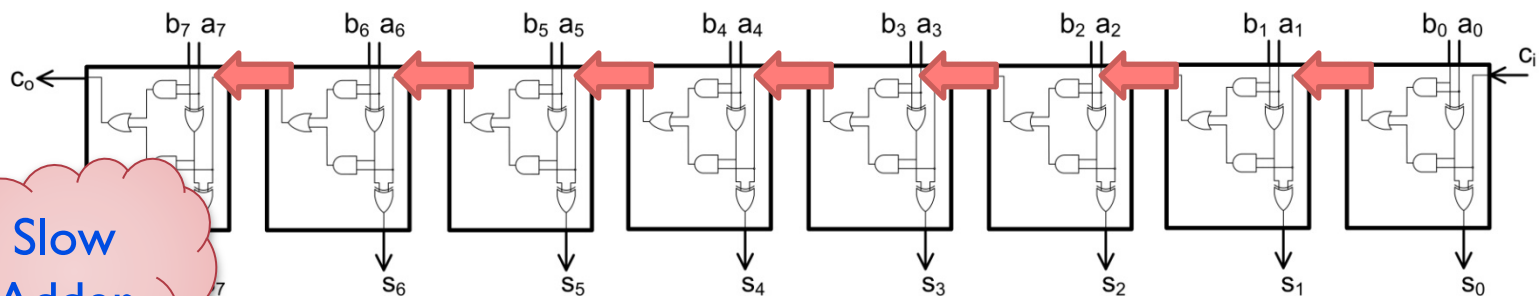
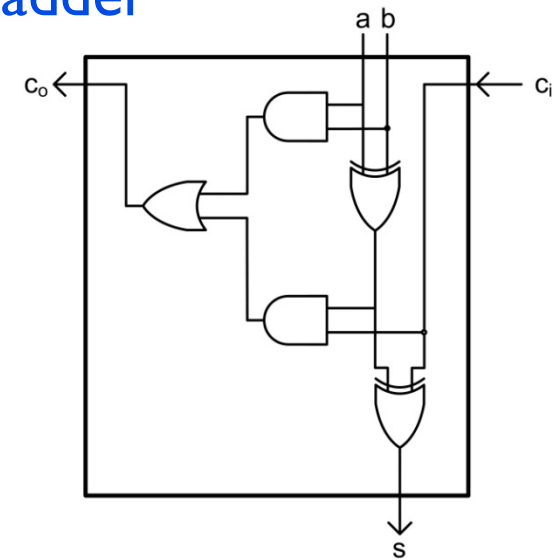
a	b	c_i	c_0	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c_i \ a b	00	01	11	10
0			1	
1		1	1	1

$$c_o = ab + ac_i + bc_i$$

c_i \ a b	00	01	11	10
0		1		1
1	1		1	

$$s = a \oplus b \oplus c_i$$



Slow
Adder

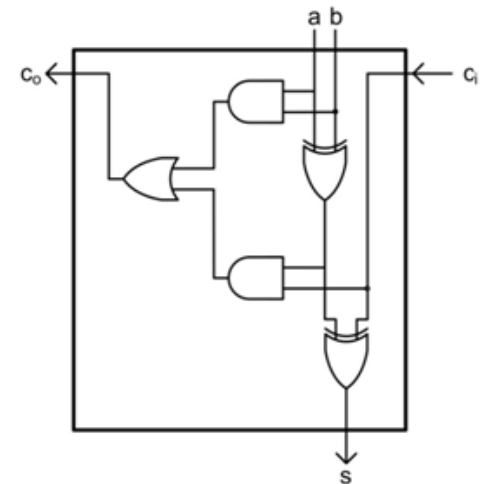
July 2014

Structural Adder

- Descriptions by Use of Equations
 - Full-Adder using fewer gates

```
`timescale 1ns/1ns

module fulladder ( input a, b, ci, output s, co );
    wire axb;
    assign axb = a ^ b;
    assign s = axb ^ ci;
    assign co = ( a & b ) | ( axb & ci );
endmodule
```



Full Adder Verilog Code

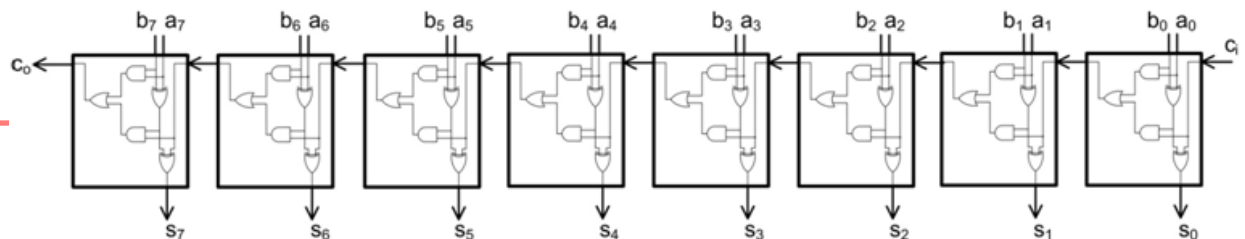
Structural Adder

- Instantiate 8 full-adders

```
`timescale 1ns/1ns

module adder8 ( input [7:0] a, b, input ci,
               output [7:0] s, output co );

  wire [7:0] c;
  assign c[0] = ci;
  fulladder FA0 (a[0], b[0], c[0], s[0], c[1]);
  fulladder FA1 (a[1], b[1], c[1], s[1], c[2]);
  fulladder FA2 (a[2], b[2], c[2], s[2], c[3]);
  fulladder FA3 (a[3], b[3], c[3], s[3], c[4]);
  fulladder FA4 (a[4], b[4], c[4], s[4], c[5]);
  fulladder FA5 (a[5], b[5], c[5], s[5], c[6]);
  fulladder FA6 (a[6], b[6], c[6], s[6], c[7]);
  fulladder FA7 (a[7], b[7], c[7], s[7], co);
endmodule
```



Eight-bit Adder Verilog Code



Adder Using Expressions

- Use assign statement and concatenation

```
`timescale 1ns/1ns

module adder8 ( input [7:0] a, b, input ci,
               output [7:0] s, output co );

    assign {co, s} = a + b + ci;

endmodule
```

Eight-bit Functional Adder Verilog Code



Multi Function Modules

- Descriptions by Use of Equations
- ALU Example

```
module ALU ( input [7:0] a, b, input addsub,  
             output gt, zero, co, output [7:0] r );  
  
    assign {co, r} = addsub ? (a + b) : (a - b);  
    assign gt = (a>b);  
    assign zero = (r == 0);  
endmodule
```

ALU Verilog Code Using a Mix of Operations



Testbenches

- Use initial repeat

```
`timescale 1ns/1ns

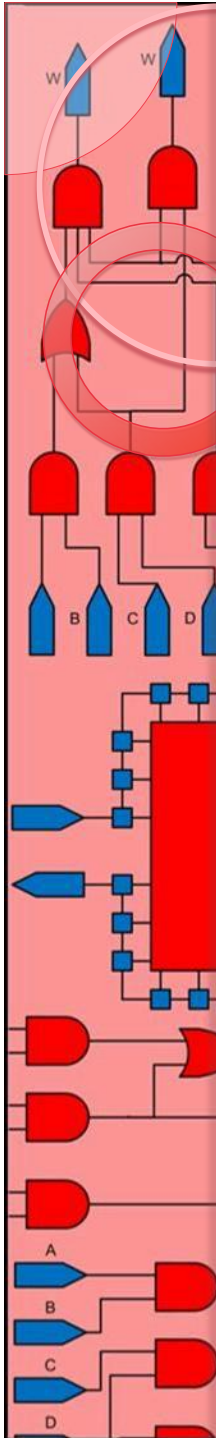
module adder8testbench ();

    reg [7:0] ai, bi;
    reg ci=0;
    wire [7:0] so;
    wire co;

    adder8 UUT (ai, bi, ci, so, co);
    initial repeat (20) #17 ai = $random;
    initial repeat (16) #23 bi = $random;

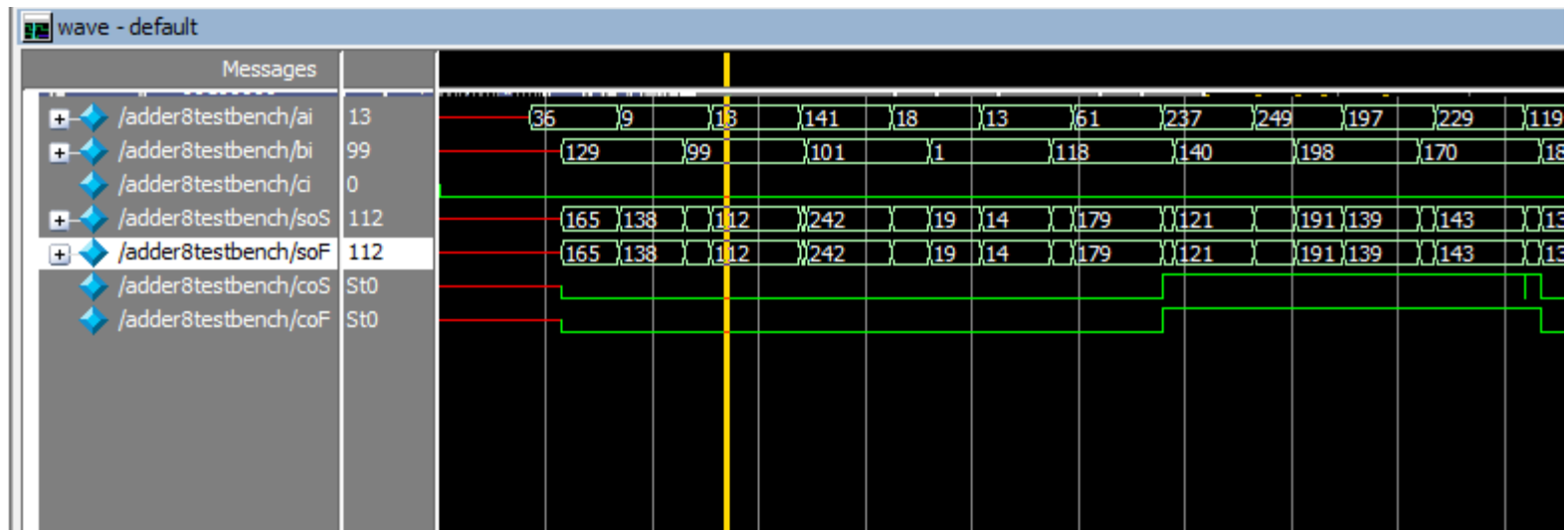
endmodule
```

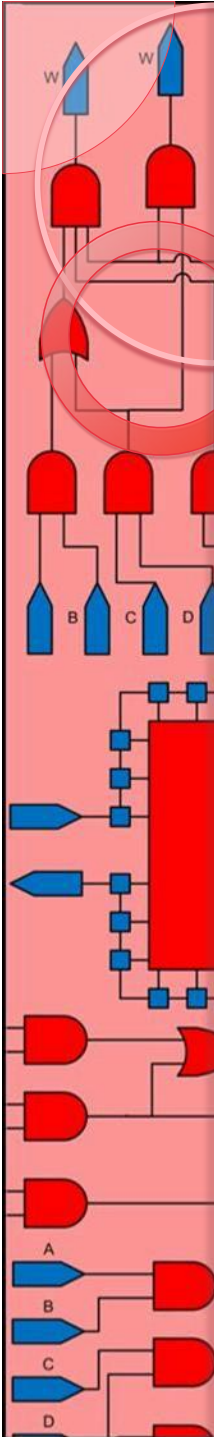
Eight-bit Adder Verilog Testbench



Testbenches

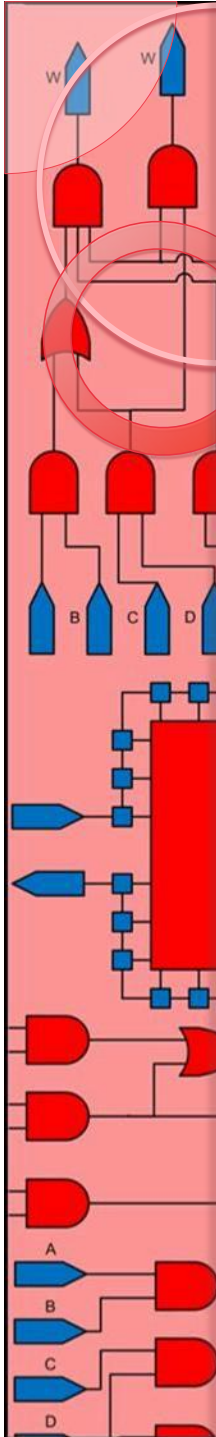
- Simulation results





Summary

- New operations
- Full adder
- Vector inputs
- Structural adder
- Adder using expressions
- A multi-function module
- More on Testbenches
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

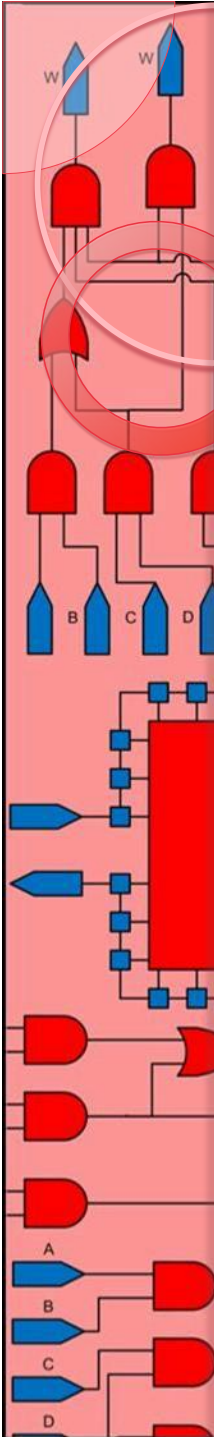
Basics of Digital Design at RT Level with Verilog

Lecture 8: Behavioral Modeling

July 2014

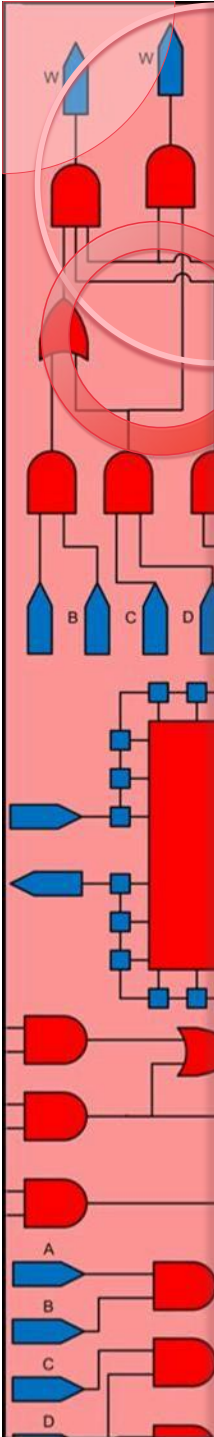
© 2013-2014 Zain Navabi

105



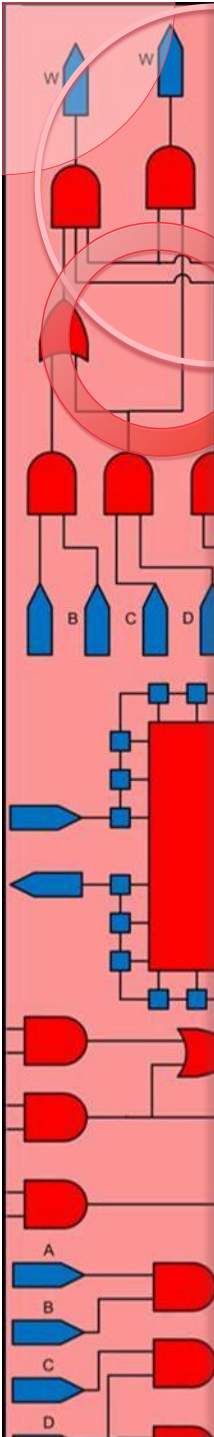
Behavioral Modeling

- New constructs
 - Procedural blocks
 - Procedural statements
- Procedural always block
 - Majority circuit
 - Full adder
- Delay options
- Procedural statements
- A behavioral ALU
- Summary



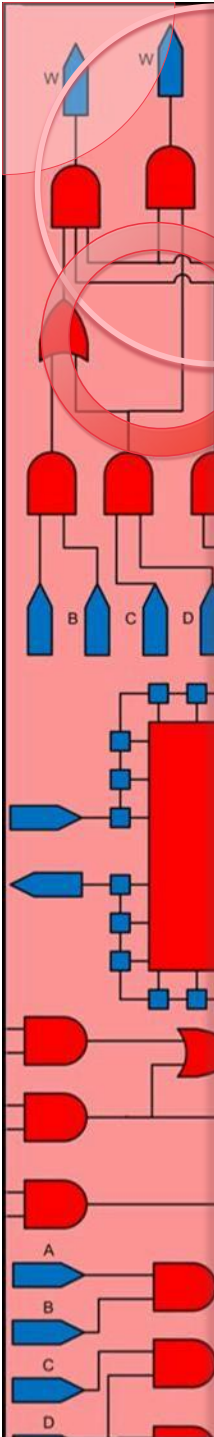
New Constructs

- Use Verilog **always** statement
- Position of various always statements in module code is irrelevant
- Position of statements within an always statements is important



New Constructs

- Use the following constructs:
 - Procedural if: if (a==b) DoThis else DoThat;
 - Procedural if: if (a==b) begin ... end
 else begin ... end
 - Procedural case: case (expr)
 0: ...
 1: ...
 2: ...
 endcase



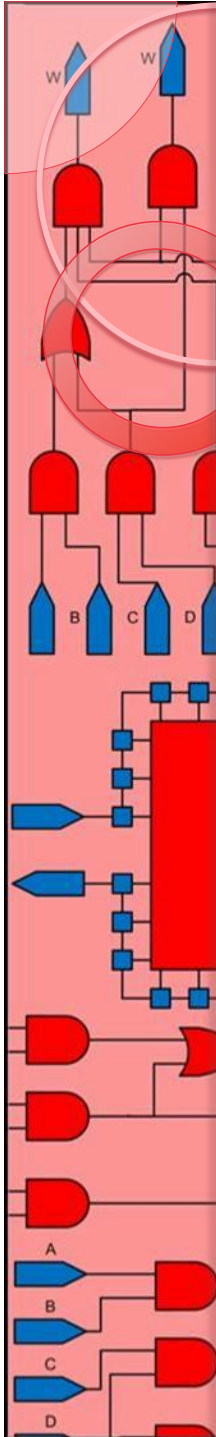
New Operations

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);
  wire c1;
  nor g1 (c1, i1, i2);
  and g2 (w1, c1, i3);
  xor g3 (w2, i1, i2, i3);
endmodule
```

```
module simple_1b (input i1, i2, i3, output w1, w2);
  assign w1 = i3 & ~(i1 | i2);
  assign w2 = i1 ^ i2 ^ i3;
endmodule
```

```
module simple_1c (input i1, i2, i3, output w1, w2);
  reg w1, w2;
  always @(i1, i2, i3) begin
    if (i1 | i2 ) w1 = 0; else w1 = i3;
    w2 = i1 ^ i2 ^ i3;
  end
endmodule
```



Procedural always Block

- XOR example

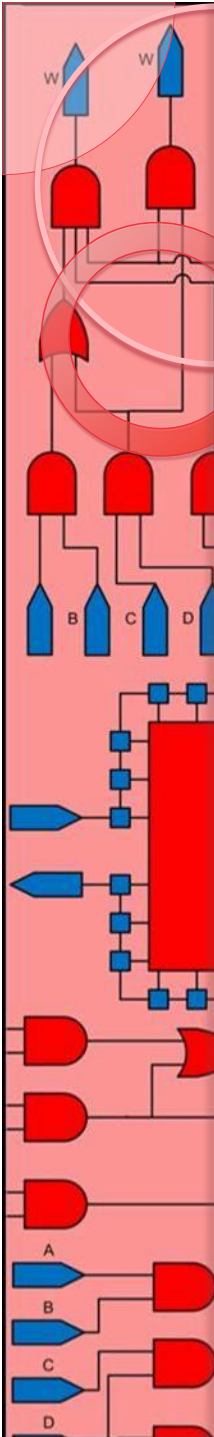
```
`timescale 1ns/1ns

module xor3 ( input a, b, c,
              output reg y );

    always @(a, b, c) y = a ^ b ^ c;

endmodule
```

XOR Verilog Code



Procedural always Block

- Majority Example

```
module maj3 ( input a, b, c, output reg y );  
  
    always @( a or b or c )  
    begin  
        y = (a & b) | (b & c) | (a & c);  
    end  
  
endmodule
```

Procedural Block Describing a Majority Circuit



Delay Options

- Majority example with delay

```
`timescale 1ns/100ps

module maj3 ( input a, b, c, output reg y );

    always @(a, b, c )
        #5 y = (a & b) | (b & c) | (a & c);

endmodule
```

Majority Gate with Delay



Delay Options

- Descriptions with procedural statements
 - Full-Adder example

```
`timescale 1ns/100ps
module add_1bit ( input a, b, ci,
                  output reg s, co );

    always @( a, b, ci )
    begin
        s = #5 a ^ b ^ ci;
        co = #3 (a & b) | (b & ci) | (a & ci);
    end

endmodule
```

Full-Adder Using Procedural Assignments



Delay Options

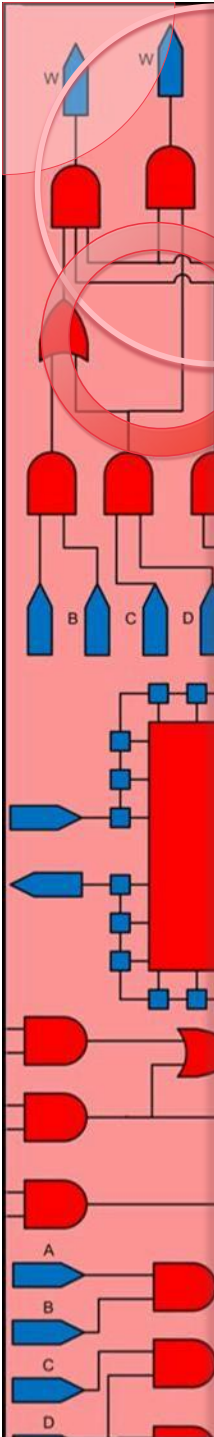
- Descriptions with procedural statements
 - Full-Adder example

```
`timescale 1ns/100ps
module add_1bit ( input a, b, ci,
                  output reg s, co );

    always @( a, b, ci )
    begin
        s <= #5 a ^ b ^ ci;
        co <= #8 (a & b) | (b & ci) | (a & ci);
    end

endmodule
```

Full-Adder Using Procedural Assignments



Procedural Statements

- Descriptions with Procedural Statements
 - Procedural Multiplexer Example

```
module mux2_1 (input i0, i1, s, output reg y );  
  always @( i0 or i1 or s ) begin  
    if ( s==1'b0 )  
      y = i0;  
    else  
      y = i1;  
    end  
endmodule
```

Procedural Multiplexer

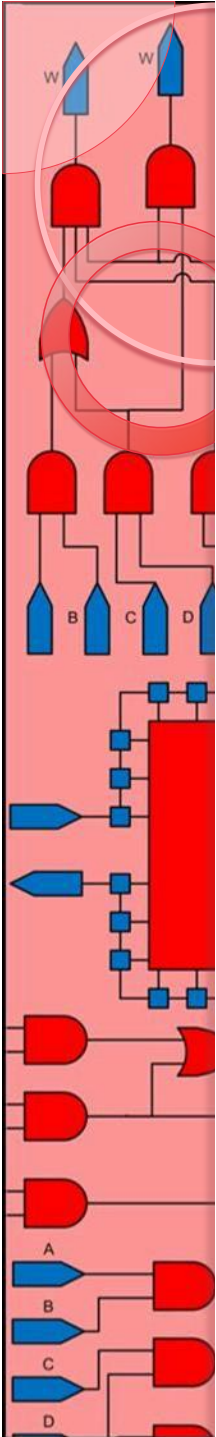


Procedural Statements

- Descriptions with Procedural Statements
 - Procedural ALU Example

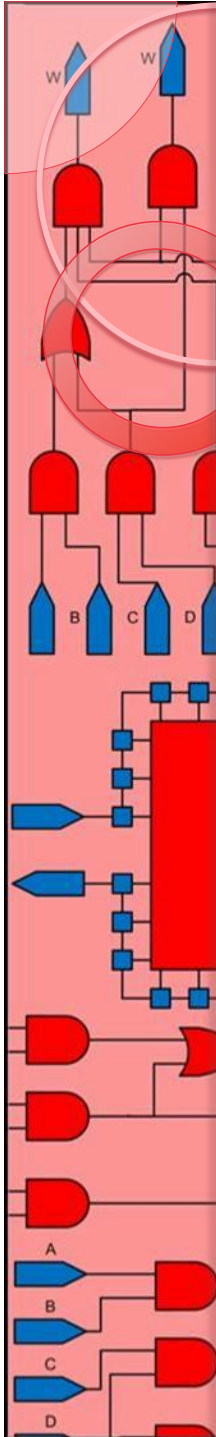
```
module alu_4bit (input [3:0] a, b, input [1:0] f,  
                output reg [3:0] y );  
    always @ ( a or b or f ) begin  
        case ( f )  
            2'b00 : y = a + b;  
            2'b01 : y = a - b;  
            2'b10 : y = a & b;  
            2'b11 : y = a ^ b;  
            default: y = 4'b0000;  
        endcase  
    end  
endmodule
```

Procedural ALU



Summary

- New constructs
 - Procedural blocks
 - Procedural statements
- Procedural always block
 - Majority circuit
 - Full adder
- Delay options
- Procedural statements
- A behavioral ALU
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

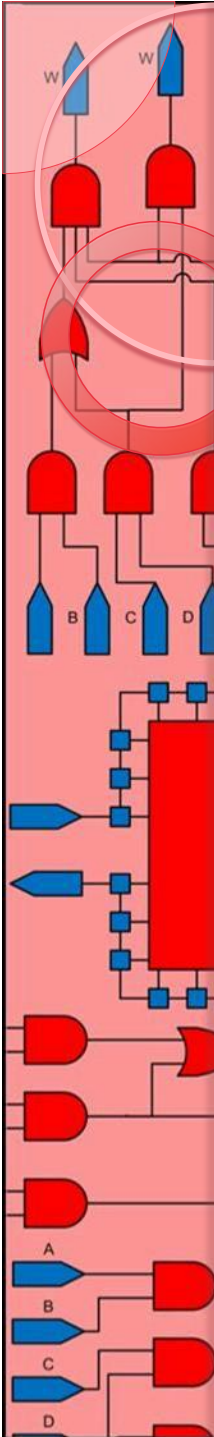
Basics of Digital Design at RT Level with Verilog

Lecture 9: Flip-Flop Modeling

July 2014

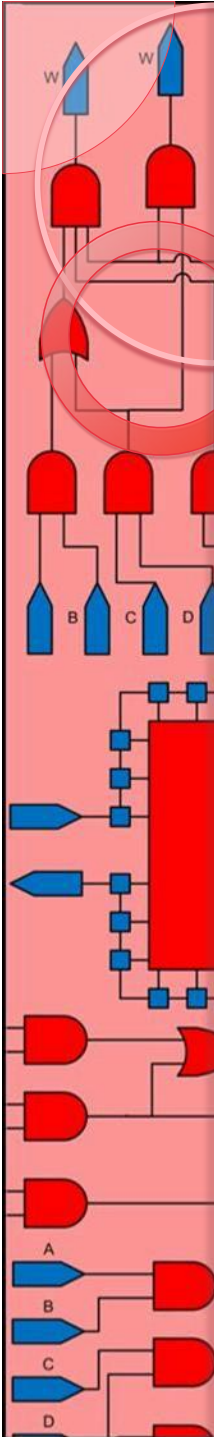
© 2013-2014 Zain Navabi

118



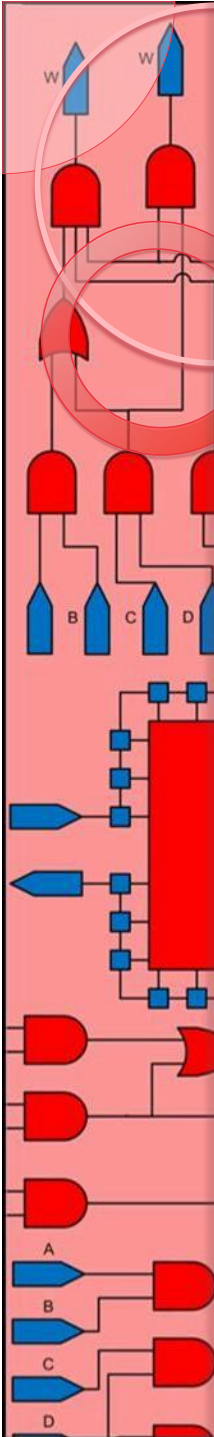
Flip-flop Modeling

- New constructs
 - Procedural blocks
 - Posedge
 - Nonblocking assignment
- Edge sensitive **always** block
 - Clock edge
 - Asynchronous control
 - Clock enable
- Delay options
- D-type flip-flop
- More on Testbenches
- Summary



New Constructs

- Use Verilog **always** statement
- Position of various always statements in module code is irrelevant
- Use **posedge** or **negedge** for edge
- Position of statements within an always statements is important
- Check for flip-flop functions according to their priority
- Assign flip-flop output using **<=**



Edge Sensitive **always** Block

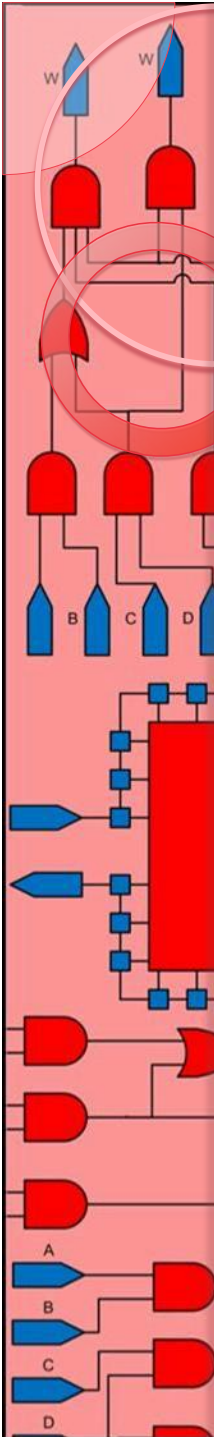
- Use in the following forms:
 - Clock detection:

```
always @(posedge C) begin  
    Q <= D;  
end
```
 - Clock detection:

```
always @(posedge C) Q <= D;
```
 - Asynch reset:

```
always @(posedge C, posedge R) begin  
    if (R) Q <= 0;  
    ...  
end
```
 - Clock enable:

```
always @(posedge C) if (E) Q <= D;
```



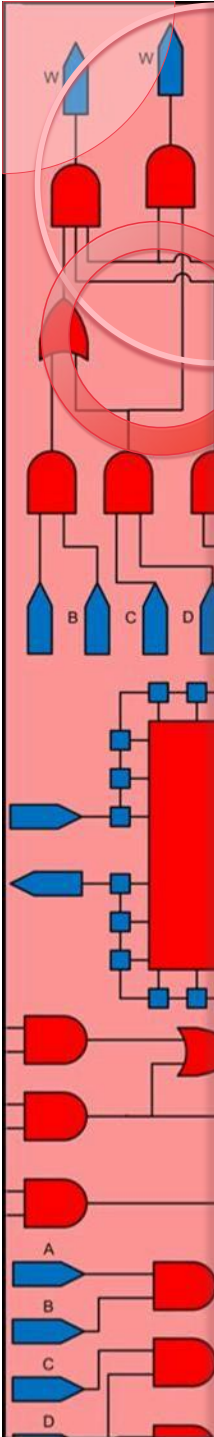
Flip-flop Modeling

- Module Outline

```
module simple_1a (input i1, i2, i3, output w1, w2);  
  wire c1;  
  nor g1 (c1, i1, i2);  
  and g2 (w1, c1, i3);  
  xor g3 (w2, i1, i2, i3);  
endmodule
```

```
module simple_1b (input i1, i2, i3, output w1, w2);  
  assign w1 = i3 & ~(i1 | i2);  
  assign w2 = i1 ^ i2 ^ i3;  
endmodule
```

```
module simple_1c (input i1, i2, i3, output w1, w2);  
  reg w1, w2;  
  always @(i1, i2, i3) begin  
    if (i1 | i2 ) w1 = 0; else w1 = i3;  
    w2 = i1 ^ i2 ^ i3;  
  end  
endmodule
```

Delay Options

- Delay after edge of C, before reading D, before scheduling into Q
 $\#6 Q \leq D$
- Delay after edge of C, does not affect reading D, delay scheduling D into Q
 $Q \leq \#5 D$
- After edge of C, wait 5, read D, wait 6, Q gets D
 $\#6 Q \leq \#5 D$



D-type flip-flop

- Basic D-type flip-flop

```
`timescale 1ns/1ns

module basic_dff ( input d, clk,
                  output reg q, output q_b );

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

    assign #3 q_b = ~q;
endmodule
```

A Positive-Edge D Flip-Flop



D-type flip-flop

- D-type flip-flop with reset input

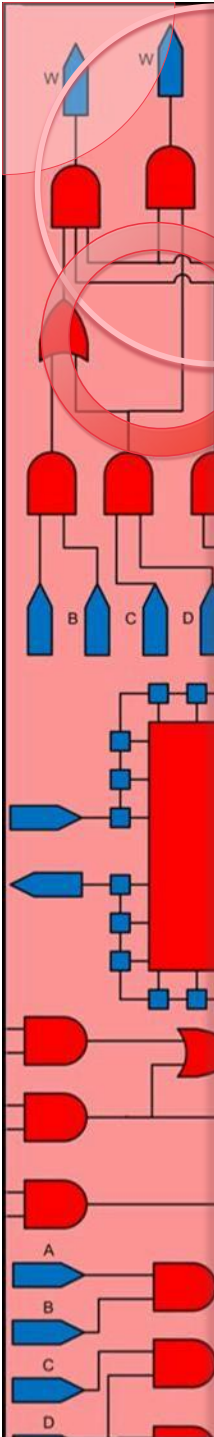
```
`timescale 1ns/1ns

module dff_reset ( input d, clk, reset
                  output reg q);

    always @( posedge clk, posedge reset ) begin
        if (reset) q <= 1'b0;
        else q <= d;
    end

endmodule
```

A Positive-Edge D Flip-Flop with active high reset



D-type flip-flop

- D-type flip-flop with reset input

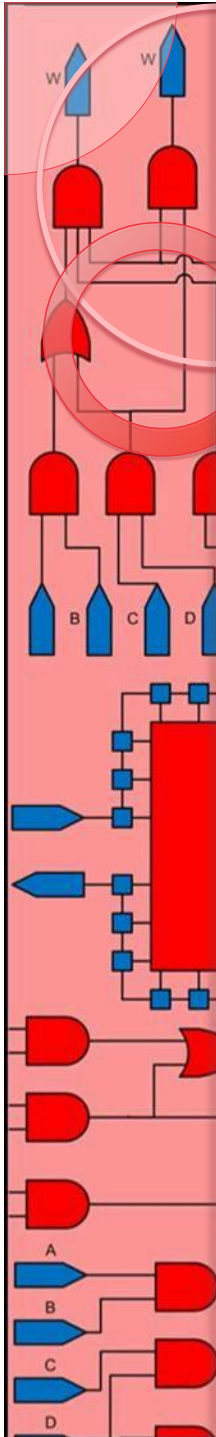
```
`timescale 1ns/1ns

module dff_reset ( input d, clk, reset
                  output reg q);

    always @( posedge clk, negedge reset ) begin
        if (~reset) q <= 1'b0;
        else q <= d;
    end

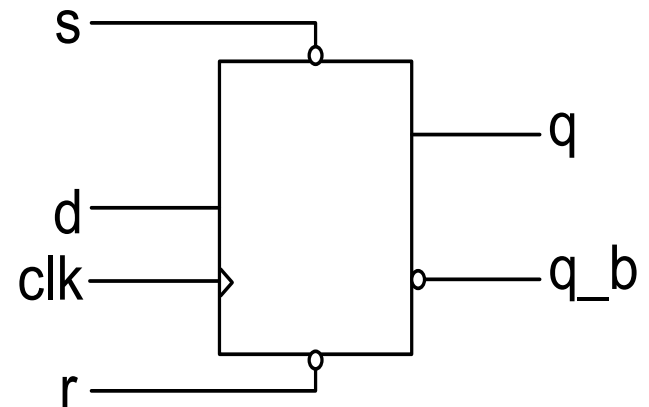
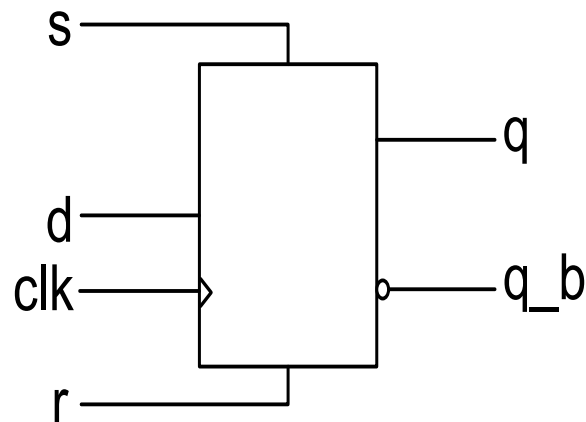
endmodule
```

A Positive-Edge D Flip-Flop with active low reset



D-type flip-flop

- Memory Elements Using Procedural Statements
- Asynchronous Control





D-type flip-flop

- D-type flip-flop with set and reset input

```
`timescale 1ns/1ns

module dff_reset ( input d, clk, set, reset
                  output reg q);

    always @(posedge clk, posedge reset, posedge set)
        if (reset) q <= 1'b0;
        else if (set) q <= 1'b1;
        else q <= d;

endmodule
```

A Positive-Edge D Flip-Flop with active high set and reset



D-type flip-flop

- D-type flip-flop with clock enable

```
`timescale 1ns/1ns

module dff_enable ( input d, clk, en,
                   output reg q);

    always @(posedge clk)
        if (en) q <= d;

endmodule
```

A Positive-Edge D Flip-Flop with active high enable



More on Testbenches

- Apply periodic clock

```
`timescale 1ns/1ns
```

```
module dffTester ();
```

```
  reg di=0, clki=0, eni=0;
```

```
  wire qo;
```

```
  dff_enable MUT ( di, clki, eni, qo );
```

```
  always #29 clki = ~ clki;
```

```
  initial begin
```

```
    #23 di=0; #37 di=1; #23 di=0; #37 di=1;
```

```
    #23 di=0; #37 di=1; #23 di=0; #37 di=1;
```

```
    $stop;
```

```
  end
```

```
  initial begin #37 eni=1; #187 eni=0; end
```

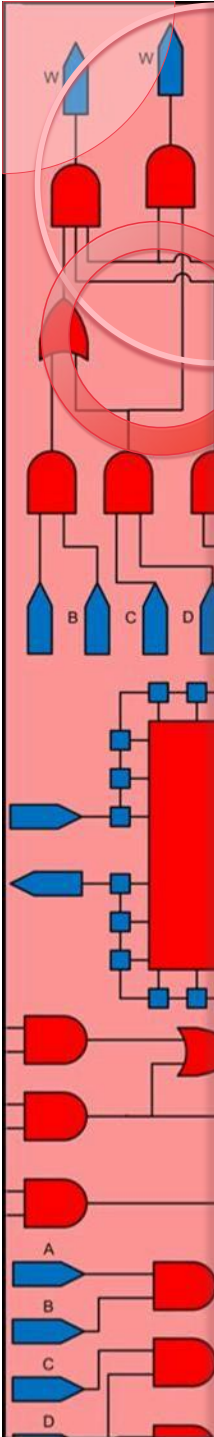
```
endmodule
```

```
`timescale 1ns/1ns
```

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

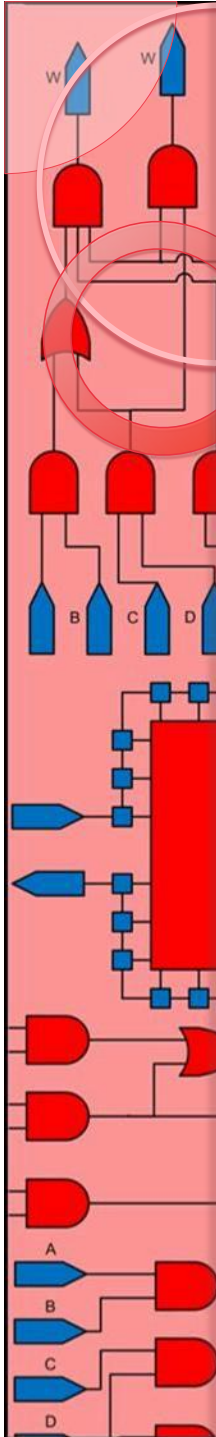
```
  always @(posedge clk)  
    if (en) q <= d;
```

```
endmodule
```

Summary

- New constructs
 - Procedural blocks
 - Posedge
 - Nonblocking assignment
- Edge sensitive **always** block
 - Clock edge
 - Asynchronous control
 - Clock enable
- Delay options
- D-type flip-flop
- More on Testbenches
- Summary



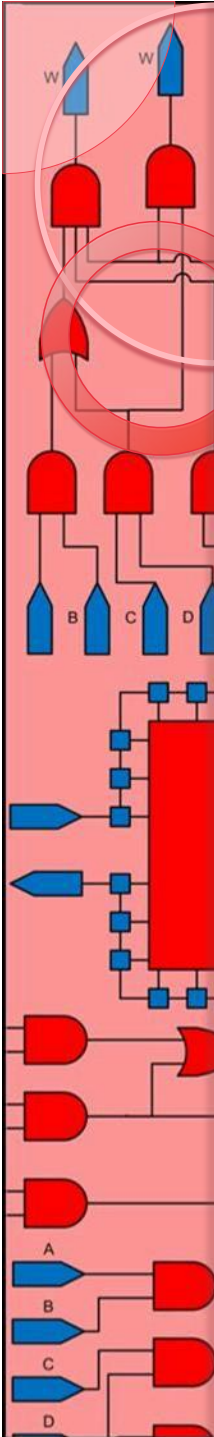
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

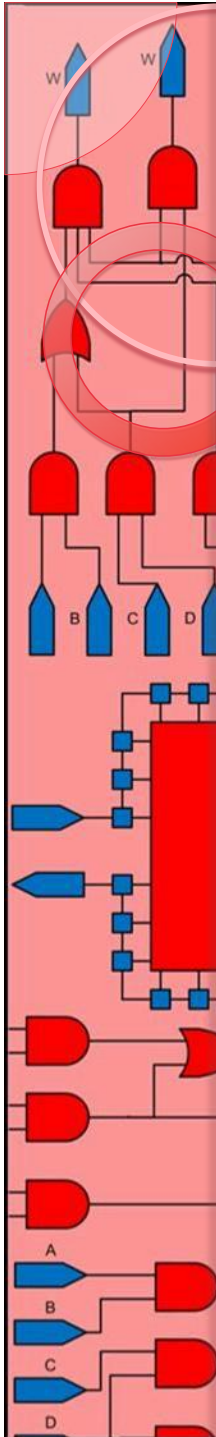
Basics of Digital Design at RT Level with Verilog

Lecture 10: Elements of RT Level Design



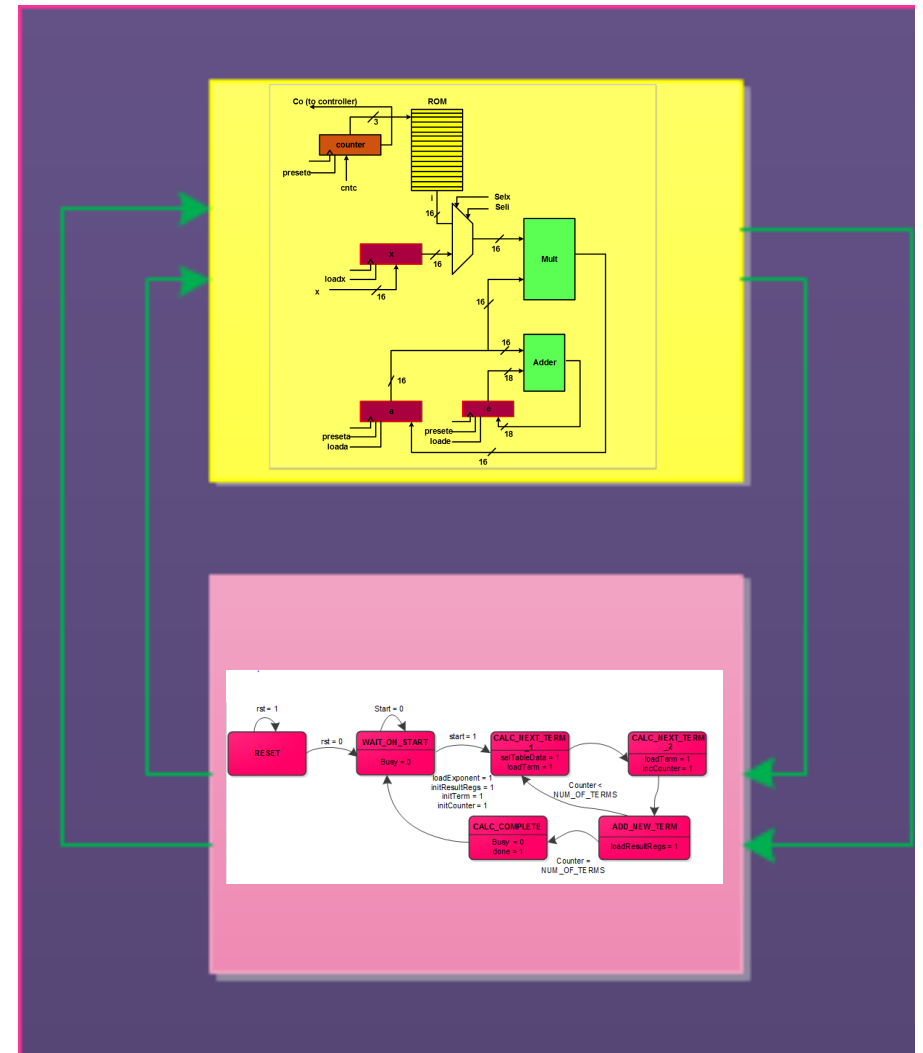
Outline

- RTL Big Picture View
- Processing Elements
- Bussing System
- Controller
- Summary



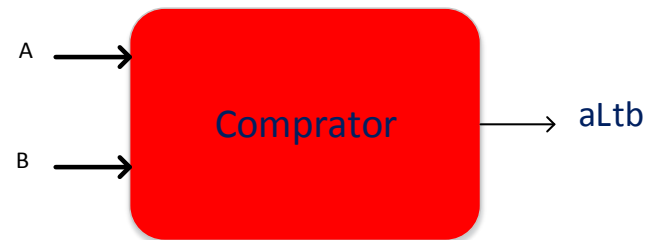
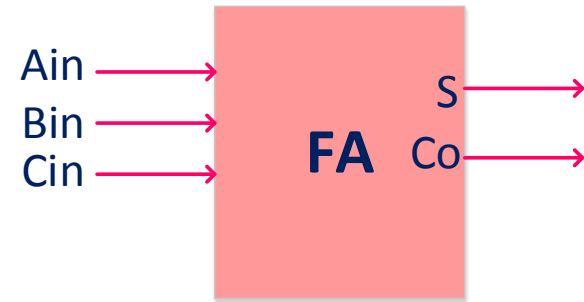
RTL Big Picture View

- Datapath
 - Combinational elements
 - Sequential elements
 - Bussing structure
- Controller
 - State machines

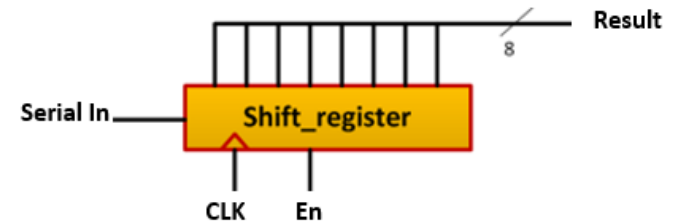
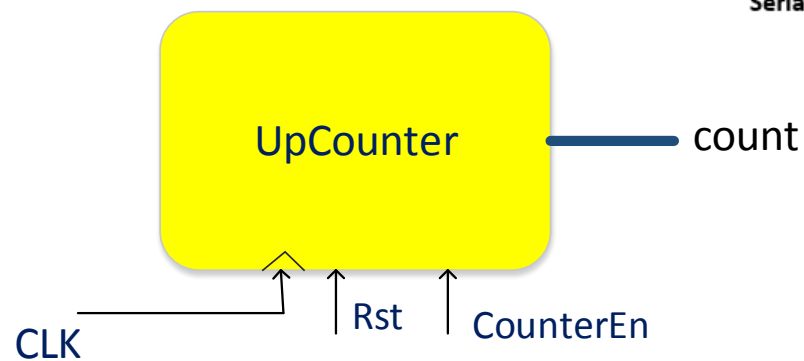


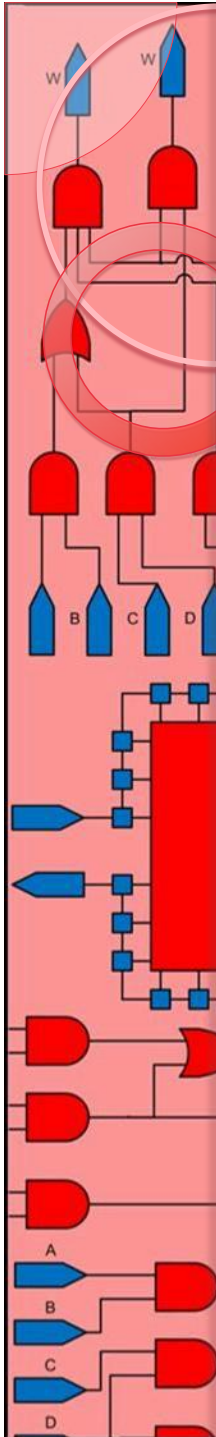
Processing Elements

- Combinational elements



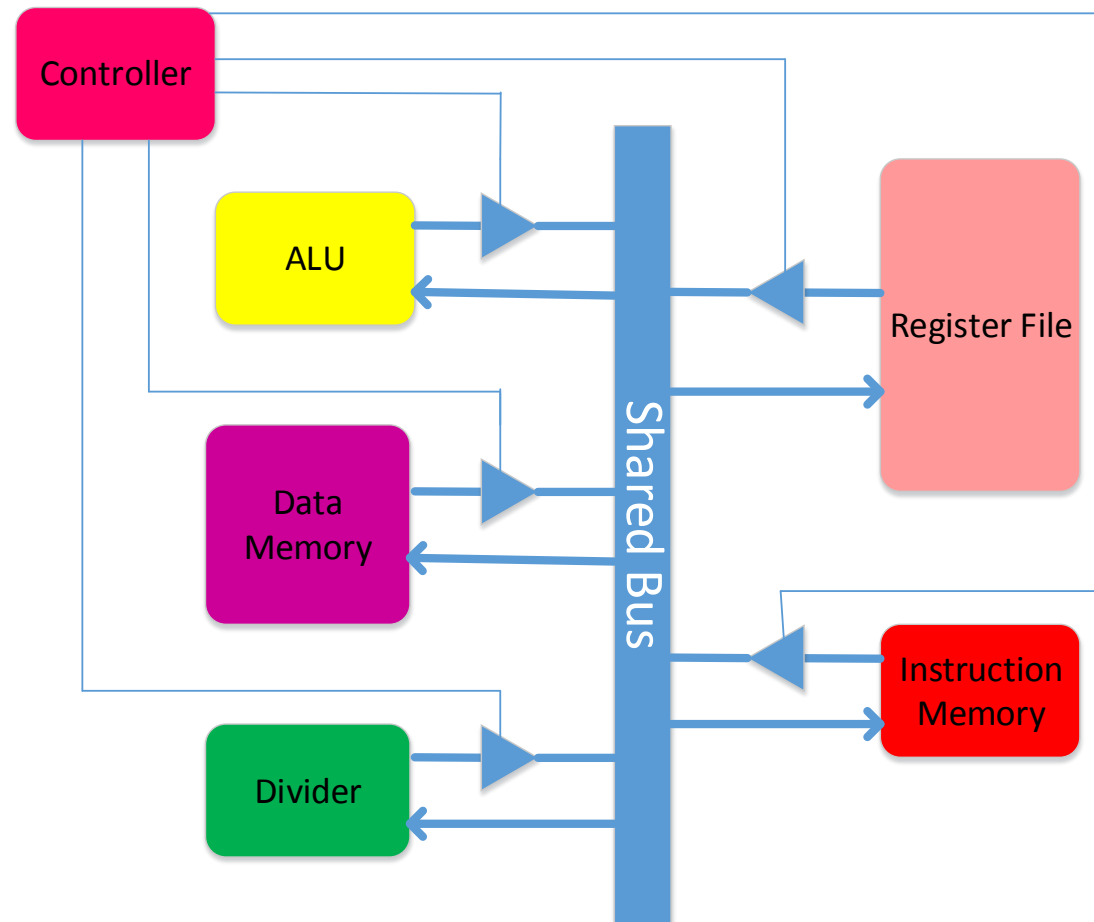
- Sequential elements

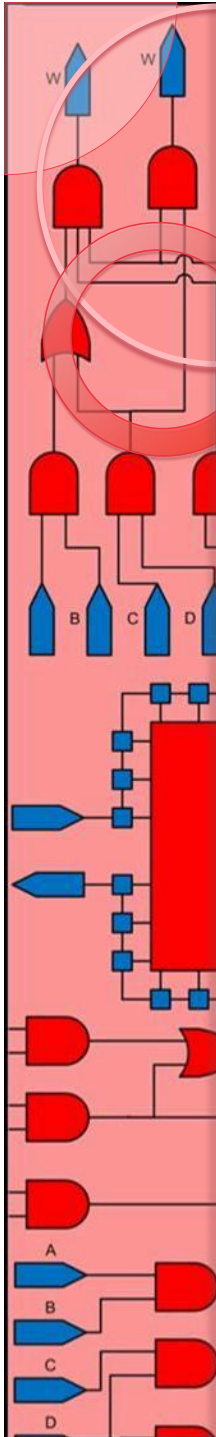




Bussing System: Intra RTL Communication

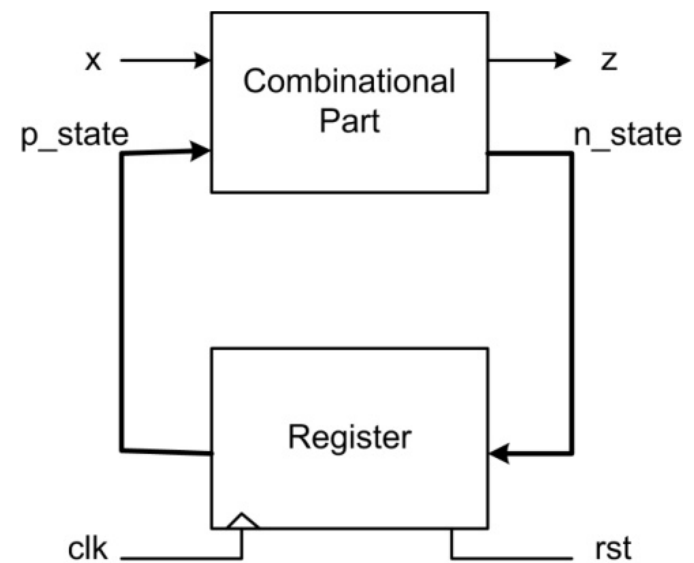
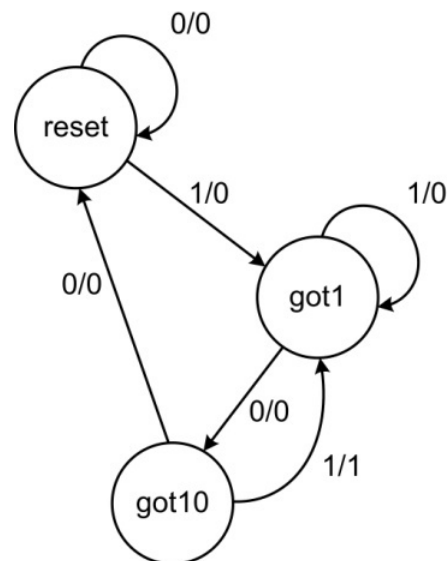
- Bussing structure

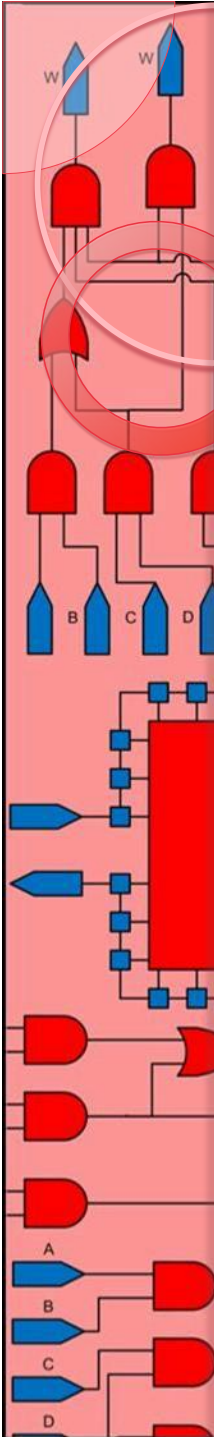




Controller:

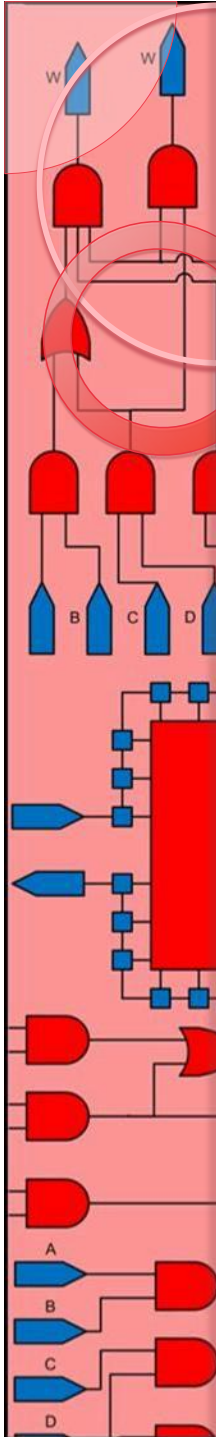
- State machines
 - Mealy/ Moore
 - Huffman coding style





Outline

- RTL Big Picture View
- Processing Elements
- Bussing System
- Controller
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

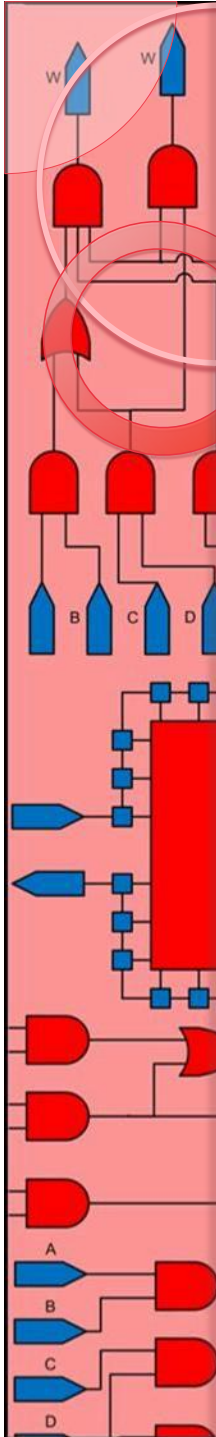
Basics of Digital Design at RT Level with Verilog

Lecture 11: Combinational Elements of RTL Design (Datapath building blocks)

July 2014

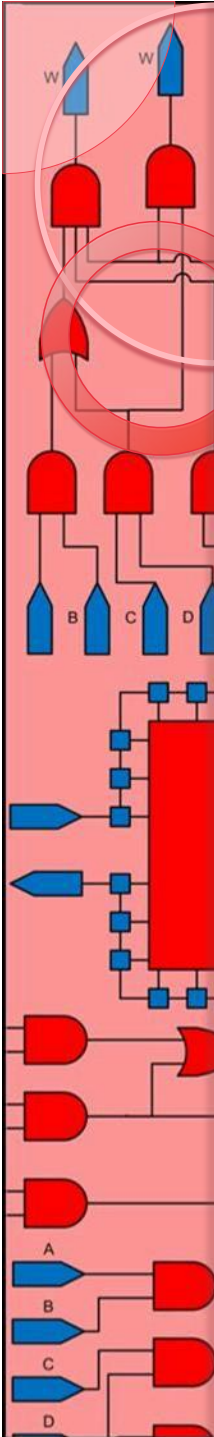
© 2013-2014 Zain Navabi

139



Combinational Elements of RTL Design

- Logic units, arithmetic units, busses
 - Random logic
 - Iterative hardware
 - Bussing system
 - General logic units

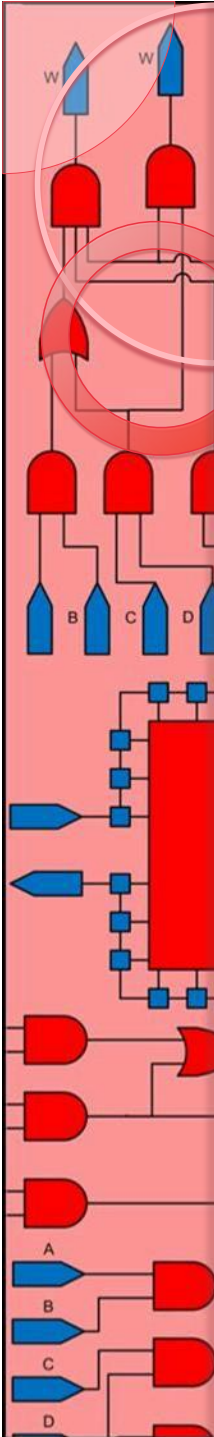


Random logic

- A Boolean expression example

```
module someLogic ( input a, b, c, d, output y );  
  
    assign y = ((a&b) | (b&c) | (a&c)) ^ d;  
  
endmodule
```

A concurrent assign statement



Random logic

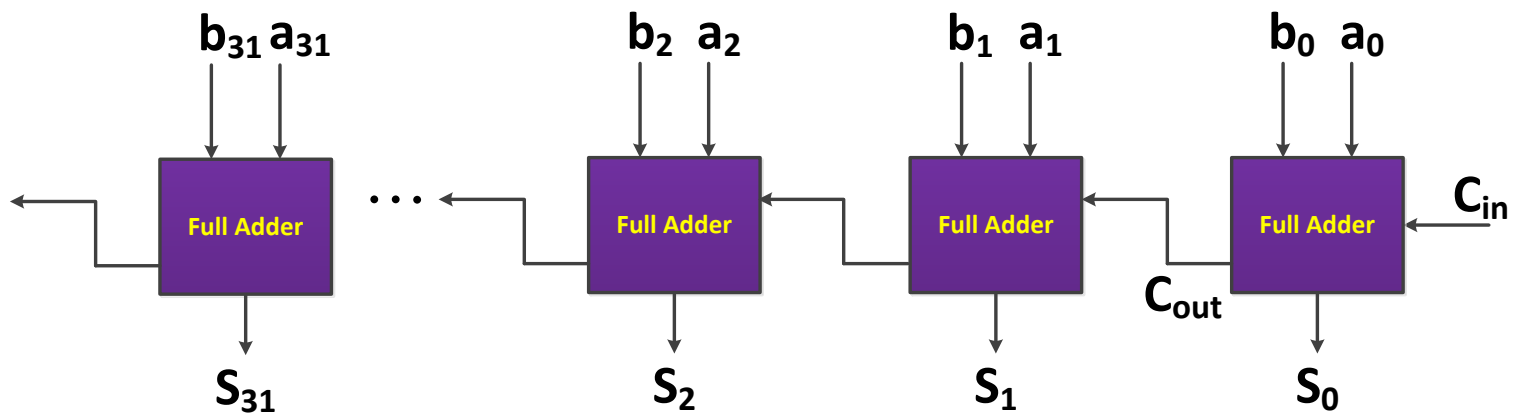
- A conditional expression example

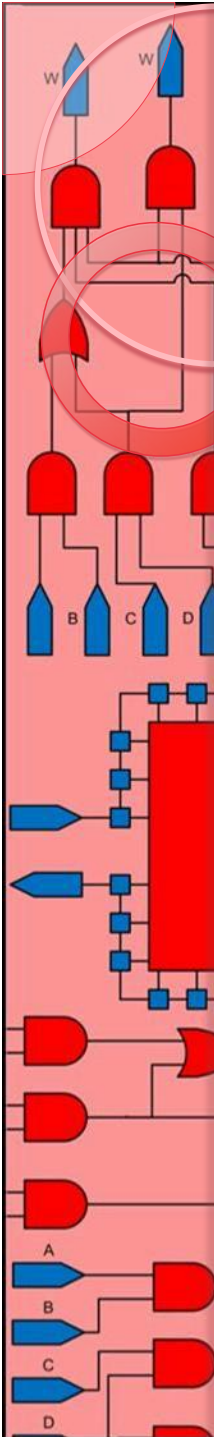
```
module sameLogic ( input a, b, c, d, output y );  
  
    assign y = d ? ~( (a&b) | (b&c) | (a&c) ) :  
                ( (a&b) | (b&c) | (a&c) );  
  
endmodule
```

A concurrent assign statement

Iterative logic

- 32-bit ripple carry adder

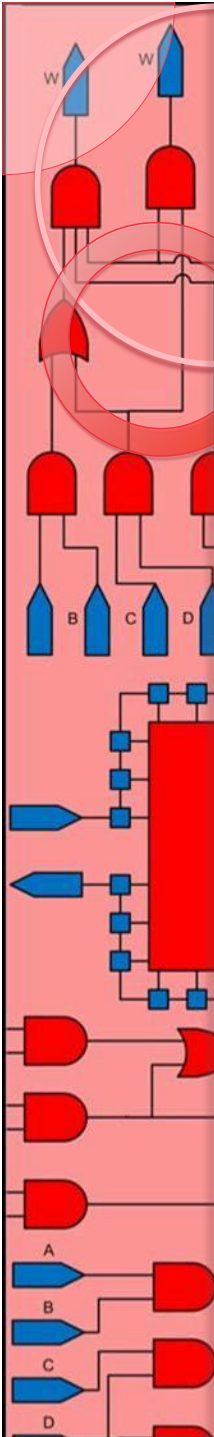




Iterative logic

- 8-bit ripple carry adder, 8 instantiations

```
module RippleCarryAdder (a, b, c0, sum, carry);  
    input [7:0] a, b;  
    input c0;  
    output [7:0] sum;  
    output carry;  
    wire [7:0] c;  
    FullAdder FA0(a[0],b[0],c0 ,sum[0],c[0]);  
    FullAdder FA1(a[1],b[1],c[0],sum[1],c[1]);  
    FullAdder FA2(a[2],b[2],c[1],sum[2],c[2]);  
    FullAdder FA3(a[3],b[3],c[2],sum[3],c[3]);  
    FullAdder FA4(a[4],b[4],c[3],sum[4],c[4]);  
    FullAdder FA5(a[5],b[5],c[4],sum[5],c[5]);  
    FullAdder FA6(a[6],b[6],c[5],sum[6],c[6]);  
    FullAdder FA7(a[7],b[7],c[6],sum[7],c[7]);  
    assign carry = c[7];  
endmodule
```



Iterative logic

- 32-bit ripple carry adder, generate

```
module GenericRippleCarryAdder32 (a,b,c0,sum,carry);
    parameter BIT = 32;
    input [BIT-1:0] a, b;
    input c0;
    output [BIT-1:0] sum;
    output carry;
    wire [BIT-1:0] c;
    FullAdder FA0(a[0],b[0],c0,sum[0],c[0]);
    genvar i;
    generate
        for(i = 1; i < BIT; i = i+1)
            FullAdder FA1(a[i],b[i],c[i-1],sum[i],c[i]);
    endgenerate
    assign carry = c[BIT-1];
endmodule
```



Bussing system

- A three-bus system

```
module bussingSystem (input [7:0] abus, bbus, cbus,  
                      input sa, sb, sc,  
                      output [7:0] ybus );  
  
    assign ybus = sa ? abus : 8'bz;  
    assign ybus = sb ? bbus : 8'bz;  
    assign ybus = sc ? cbus : 8'bz;  
  
endmodule
```

Using assign statements



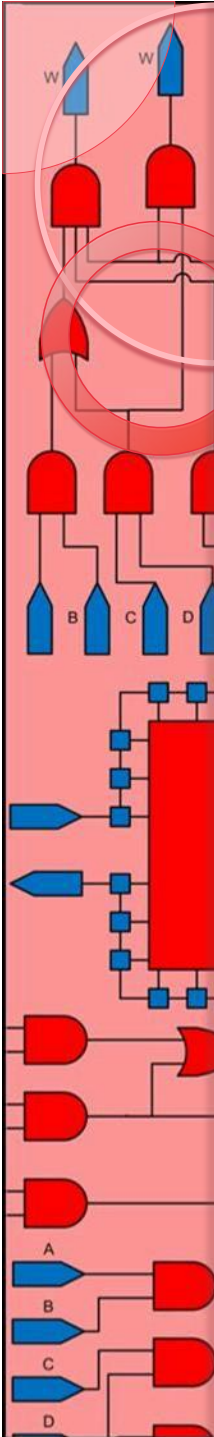
General logic units

- An ALU with flags

```
module alu_8bit (input [7:0] a, b, input [1:0] f,
                output gt, zero,
                output reg [7:0] y );

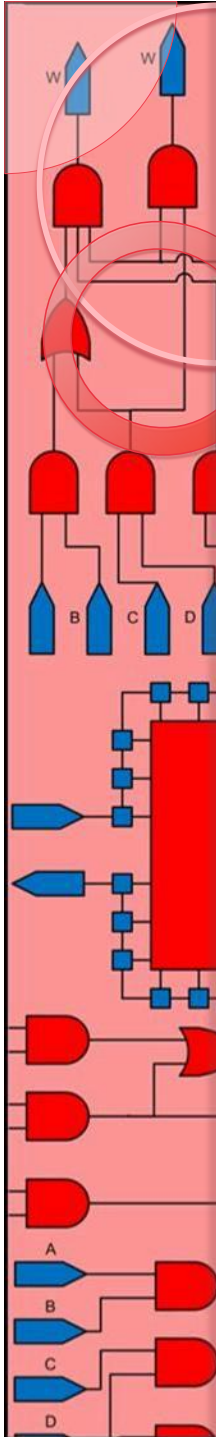
    always @ ( a, b, f ) begin
        case ( f )
            2'b00 : y = a + b;
            2'b01 : y = a > b ? a : b;
            2'b10 : y = a & b;
            2'b11 : y = a ^ b;
            default: y = 4'b0000;
        endcase
    end

    assign gt = a > b;
    assign zero = (y==8'b0) ? 1'b1 : 1'b0;
endmodule
```



Summary

- Logic units, arithmetic units, busses
- Random logic
- Iterative hardware
- Bussing system
- General logic units



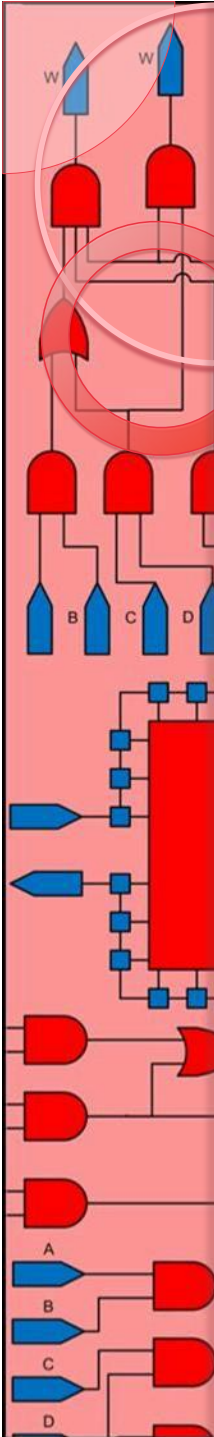
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

Basics of Digital Design at RT Level with Verilog

Lecture 12: Sequential Elements of RTL Design



Sequential Elements of RTL Design

- Registers, Shifters and Counters
 - Registers
 - Shift-Register
 - Counters



Sequential Elements of RTL Design

- Registers, Shifters and Counters - Registers

```
module register (input [7:0] d, input clk, set, reset,
                 output reg [7:0] q);
    always @ ( posedge clk ) begin
        if ( set )
            #5 q <= 8'b1;
        else if ( reset )
            #5 q <= 8'b0;
        else
            #5 q <= d;
    end
endmodule
```

An 8-bit Register

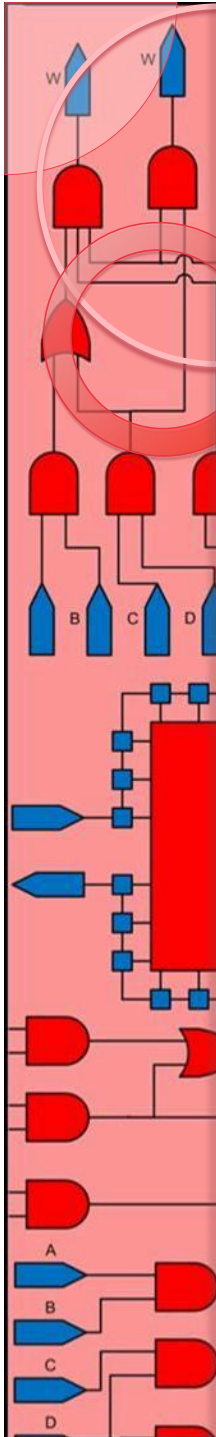


Sequential Elements of RTL Design

- Registers, Shifters and Counters - Shift-Registers

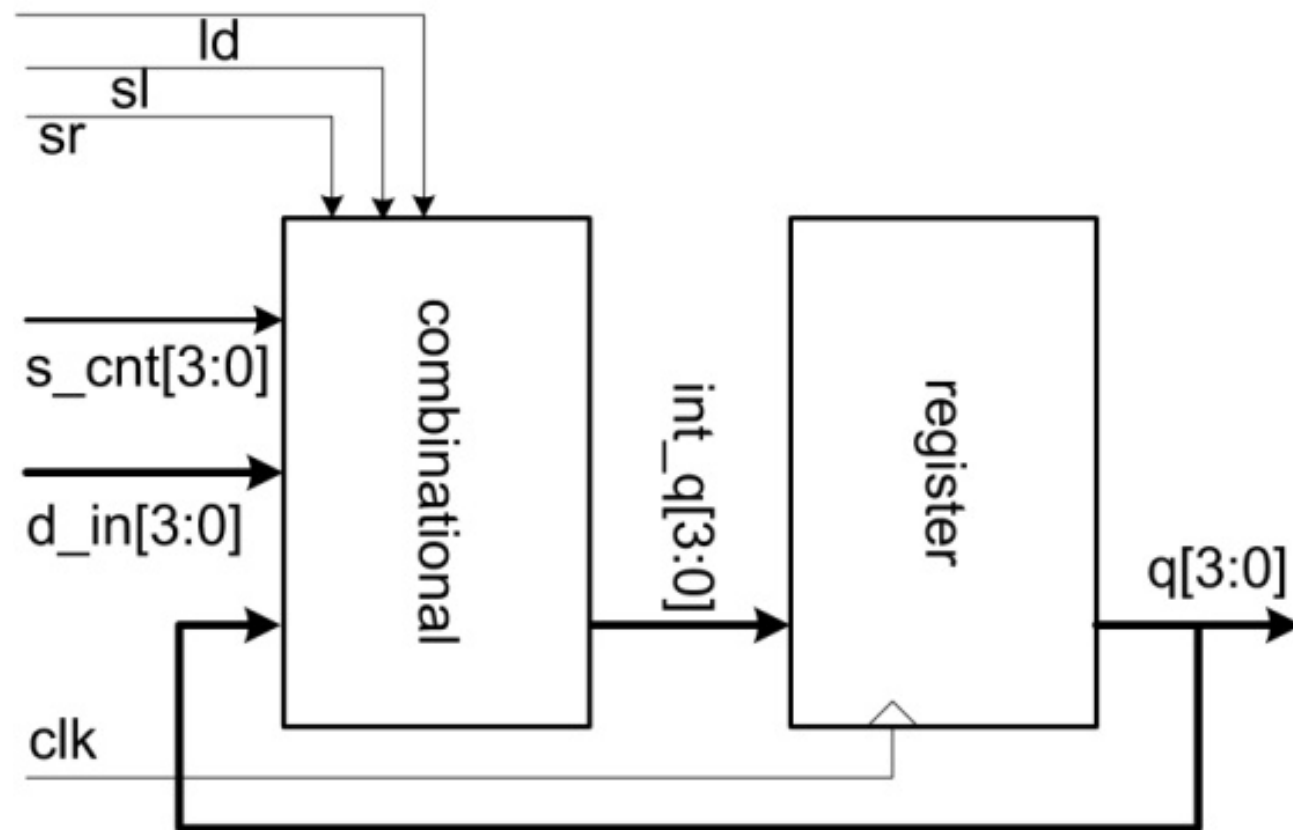
```
module shift_reg (input [3:0] d, input clk, ld, rst,
                  l_r, s_in, output reg [3:0] q);
    always @( posedge clk ) begin
        if ( rst )
            #5 q <= 4'b0000;
        else if ( ld )
            #5 q <= d;
        else if ( l_r )
            #5 q <= {q[2:0], s_in};
        else
            #5 q <= {s_in, q[3:1]};
    end
endmodule
```

A 4-bit Shift Register



Sequential Elements of RTL Design

- Registers, Shifters and Counters - Shift-Registers





Sequential Elements of RTL Design

- Registers, Shifters and Counters - Shift-Registers

```
module shift_reg (input [3:0] d_in, input clk, input [1:0]
                  s_cnt, sr, sl, ld, output reg [3:0] q );
  reg [3:0] int_q;
  always @(d_in, s_cnt, sr, sl, ld, q) begin:combinational
    if ( ld )   int_q = d_in;
    else if ( sr )   int_q = q >> s_cnt;
    else if ( sl )  int_q = q << s_cnt;
    else int_q = q;
  end
  always @ ( posedge clk ) begin:register
    q <= int_q;
  end
endmodule
```

Shift-Register Using Two Procedural Blocks



Sequential Elements of RTL Design

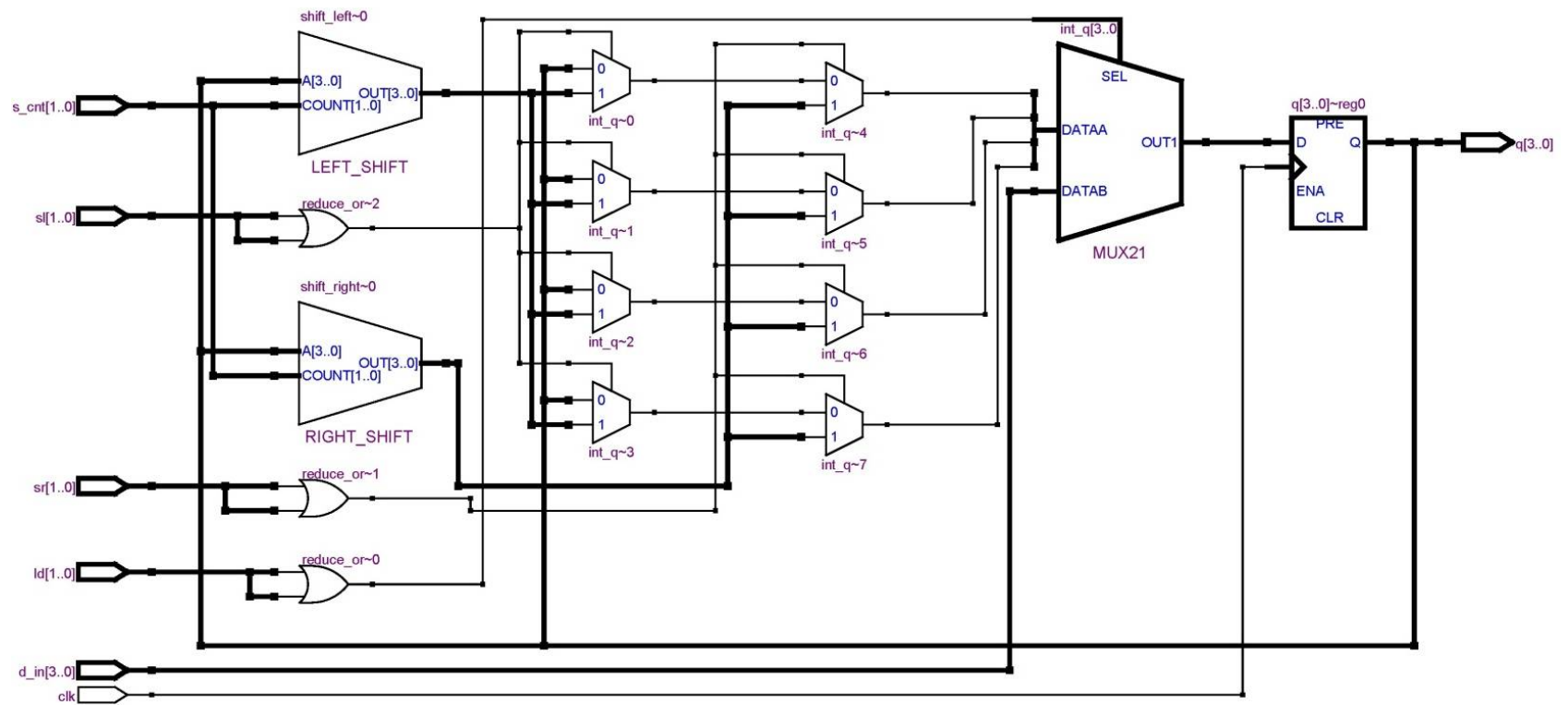
- Registers, Shifters and Counters - Counters

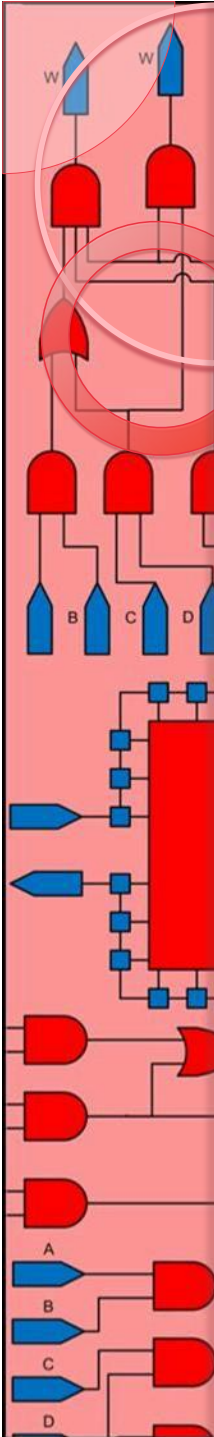
```
module counter (input [3:0] d_in, input clk, rst, ld,
                u_d, output reg [3:0] q );
    always @ ( posedge clk ) begin
        if ( rst )
            q <= 4'b0000;
        else if ( ld )
            q <= d_in;
        else if ( u_d )
            q <= q + 1;
        else
            q <= q - 1;
    end
endmodule
```

An Up-Down Counter

Sequential Elements of RTL Design

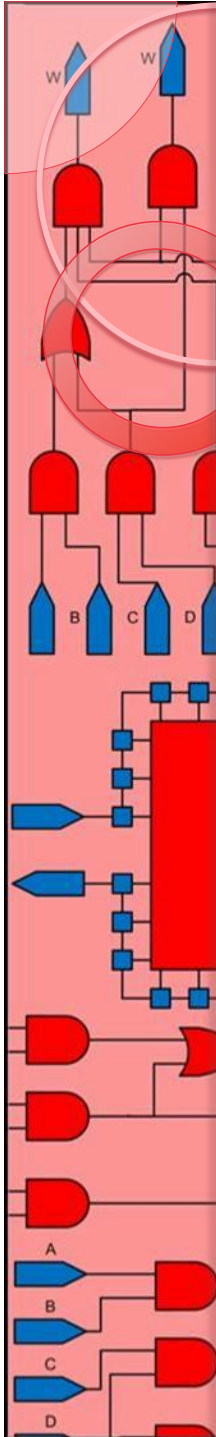
- Synthesis of Shifters and Counters





Summary

- Sequential Elements
- Registers
- Counters
- Shift registers



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

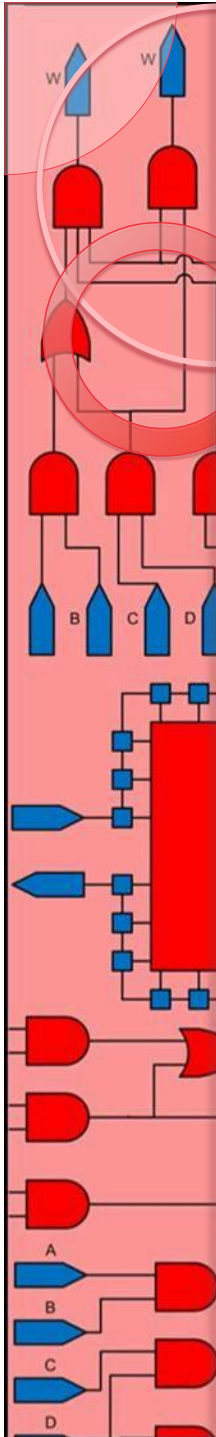
Basics of Digital Design at RT Level with Verilog

Lecture 13: State Machines (Controller building blocks)

July 2014

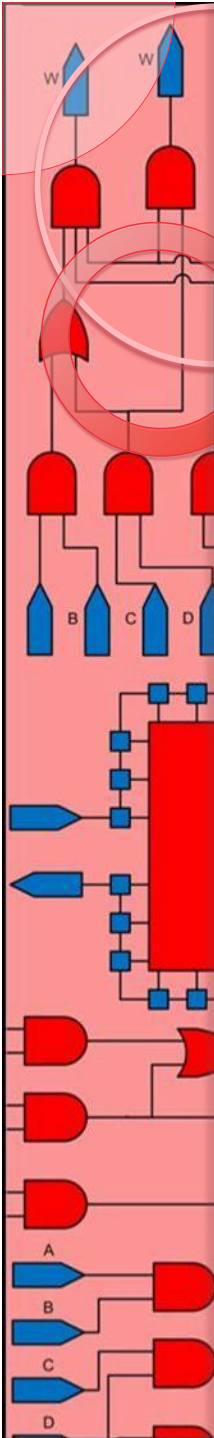
© 2013-2014 Zain Navabi

158



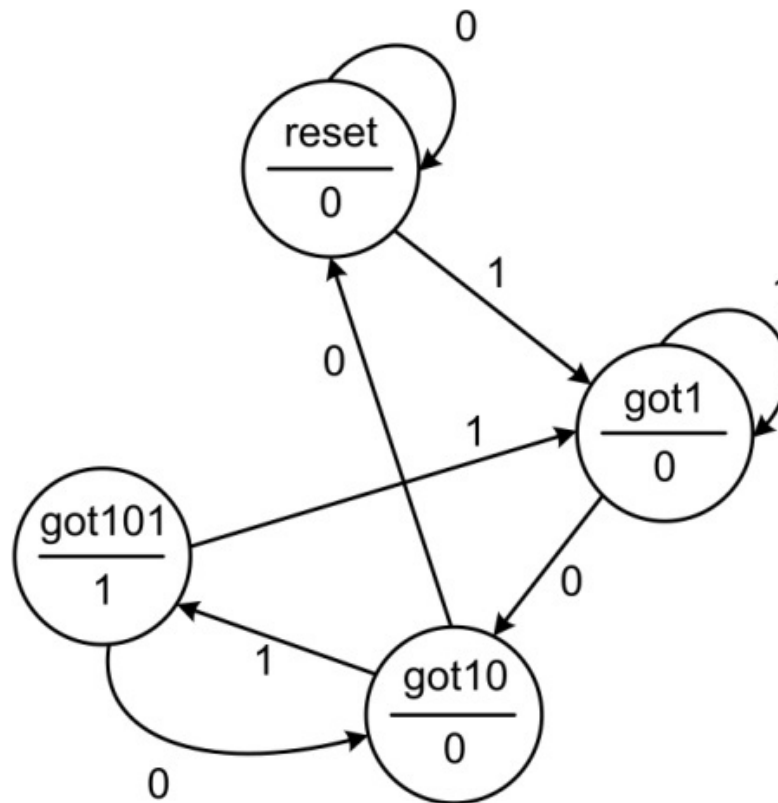
State Machines

- State Machine Coding
 - Moore Detector
 - A Mealy Machine Example
 - Huffman Coding Style
 - More Complex Outputs



State Machines

- State Machine Coding





State Machines

- State Machine Coding - Moore Detector

```
module moore_detector (input x, rst, clk, output z);
  parameter [1:0] reset = 0, got1 = 1, got10 = 2, got101 = 3;
  reg [1:0] current;
  always @ ( posedge clk ) begin
    if ( rst ) begin
      current <= reset;
    end
    else case ( current )
      reset: begin
        if ( x==1'b1 ) current <= got1;
        else current <= reset;
      end
      got1: begin
        if ( x==1'b0 ) current <= got10;
        else current <= got1;
      end
    end
  end
end
```

Moore Machine Verilog Code . . .



State Machines

- State Machine Coding - Moore Detector

```
got10:  begin
    if ( x==1'b1 ) begin
        current <= got101;
    end else begin
        current <= reset;
    end
end
got101: begin
    if ( x==1'b1 ) current <= got1;
    else current <= got10;
end
default: current <= reset;
endcase
end
assign z = (current == got101) ? 1 : 0;
endmodule
```

... Moore Machine Verilog Code

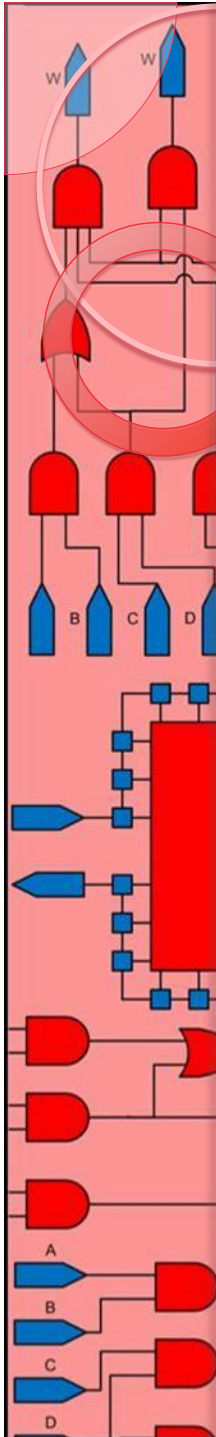


State Machines

- State Machine Coding
- Moore Detector

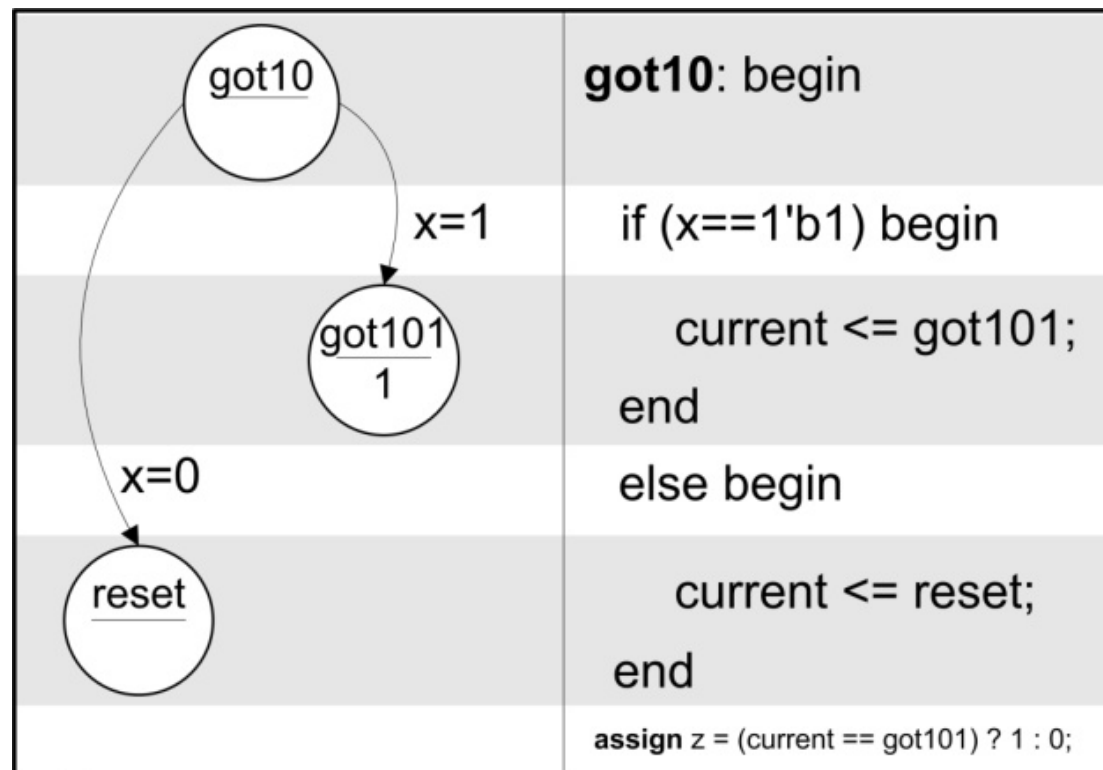
```
case ( current )  
    reset:      begin . . . end  
    got1:       begin . . . end  
    got10:      begin . . . end  
    got101:     begin . . . end  
    default:    begin . . . end  
endcase
```

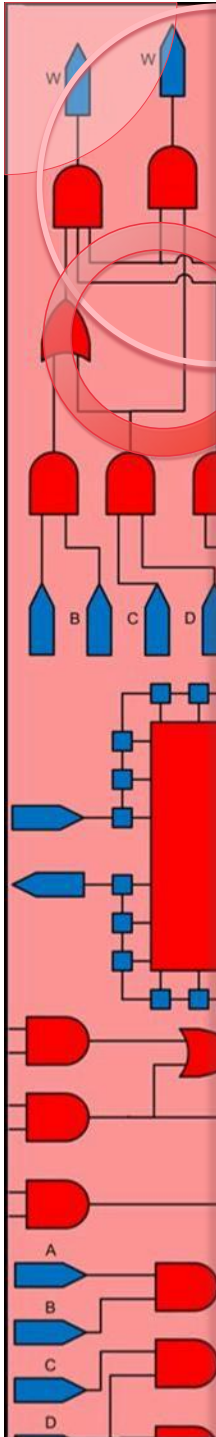
case-Statement Outline



State Machines

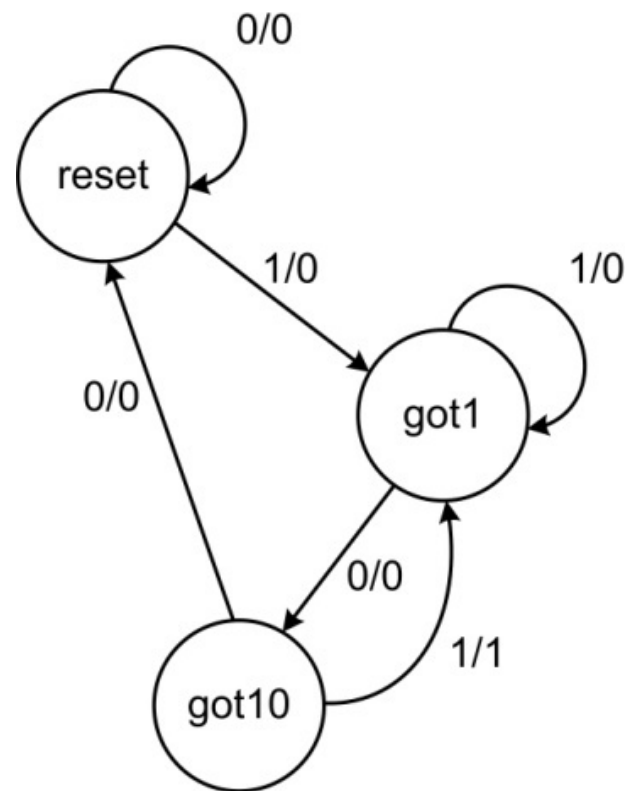
- State Machine Coding
- Moore Detector





State Machines

- State Machine Coding
 - A Mealy Machine Example





State Machines

- State Machine Coding - A Mealy Machine Example

```
module mealy_detector ( input x, clk, output z );
  parameter [1:0]
    reset    = 0,    // 0 = 0 0
    got1     = 1,    // 1 = 0 1
    got10    = 2;    // 2 = 1 0
  reg [1:0] current;
  initial current = reset;
  always @ ( posedge clk )
  begin
    case ( current )
      reset:
        if( x==1'b1 ) current <= got1;
        else current <= reset;
      got1:
        if( x==1'b0 ) current <= got10;
        else current <= got1;
    endcase
  end
endmodule
```

Verilog Code of 101 Mealy Detector . . .



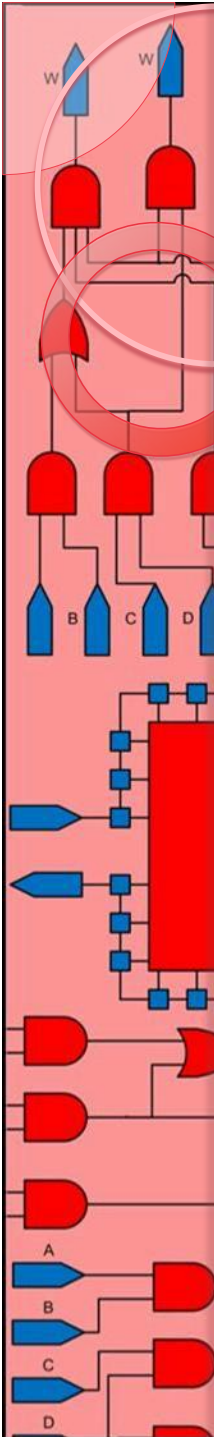
State Machines

- State Machine Coding - A Mealy Machine Example

```
got10:
    if( x==1'b1 ) current <= got1;
    else current <= reset;
    default: current <= reset;
endcase
end
assign z= ( current==got10 && x==1'b1 ) ? 1'b1 : 1'b0;

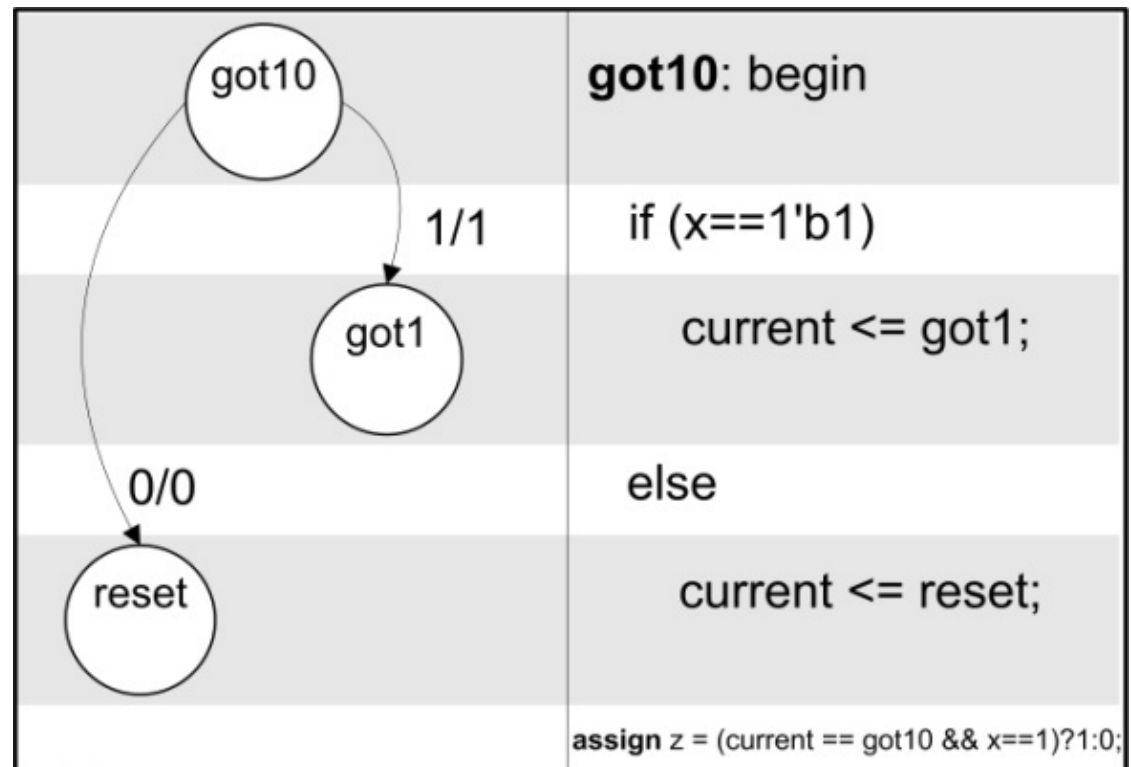
endmodule

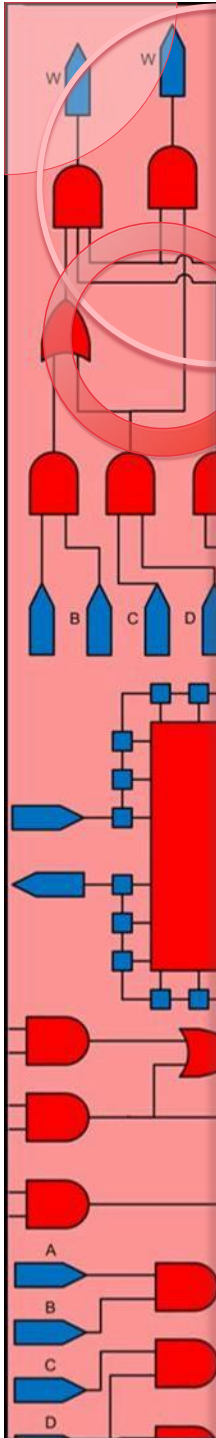
... Verilog Code of 101 Mealy Detector
```



State Machines

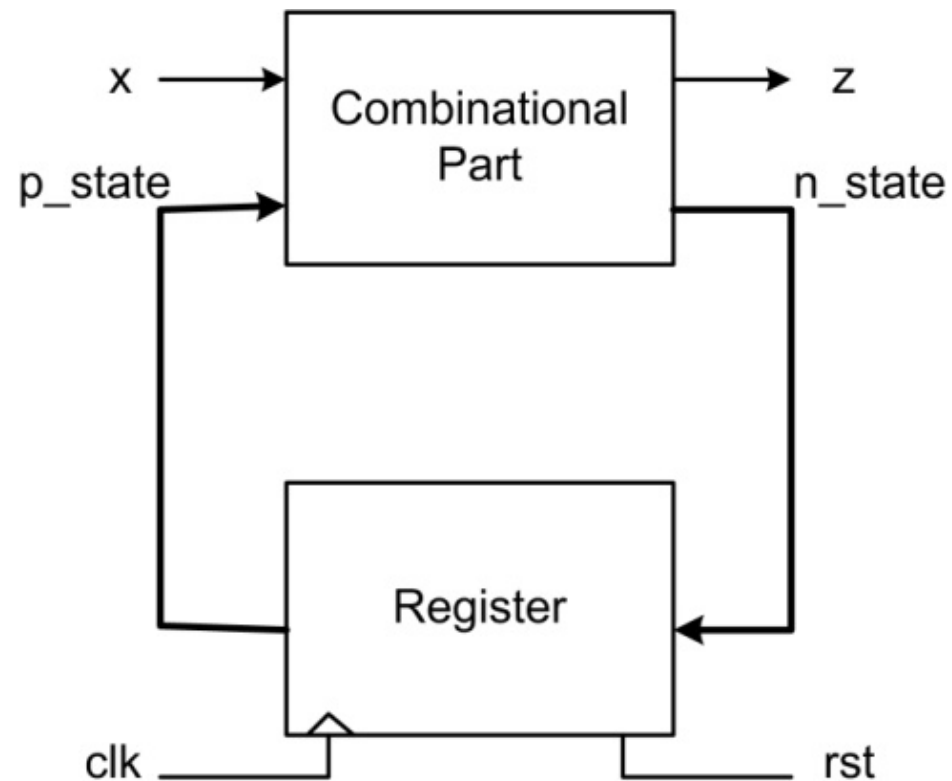
- State Machine Coding
- Huffman Coding Style





State Machines

- State Machine Coding
 - Huffman Coding Style





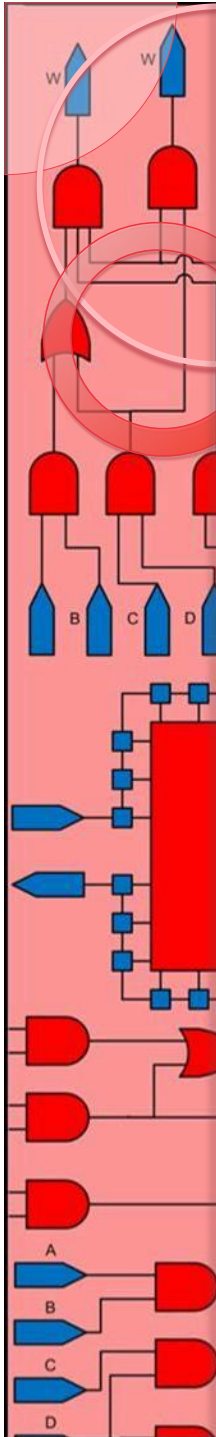
State Machines

- State Machine Coding - More Complex Outputs

```
module moore_detector ( input x, rst, clk, output z );
    parameter [1:0] reset = 2'b00, got1 = 2'b01,
                  got10 = 2'b10, got101 = 2'b11;
    reg [1:0] p_state, n_state;

    always @ ( p_state or x ) begin : combinational
        n_state = 0;
        case ( p_state )
            reset: begin
                if( x==1'b1 ) n_state = got1;
                else n_state = reset;
            end
            got1: begin
                if( x==1'b0 ) n_state = got10;
                else n_state = got1;
            end
        end
    end
```

Verilog Huffman Coding Style . . .



State Machines

- State Machine Coding - More Complex Outputs

```
got101:  begin
    if( x==1'b1 ) n_state = got1;
    else n_state = got10;
end
default: n_state = reset;
endcase
end
```

... Verilog Huffman Coding Style ...



State Machines

- State Machine Coding - More Complex Outputs

```
always @( posedge clk ) begin : register
    if( rst ) p_state = reset;
    else p_state = n_state;
end

assign z = (current == got101) ? 1 : 0;

endmodule
```

... Verilog Huffman Coding Style



State Machines

- State Machine Coding - More Complex Outputs

```
module mealy_detector ( input x, en, clk, rst, output reg z );
    parameter [1:0]  reset = 0,
                   got1 = 1, got10 = 2, got11 = 3;
    reg [1:0] p_state, n_state;

    always @( p_state or x ) begin : Transitions
        . . .
    end
    always @( p_state or x ) begin: Outputting
        . . .
    end
    always @ ( posedge clk ) begin: Registering
        . . .
    end
endmodule
```

Separate Transition and Output Blocks . . .



State Machines

- State Machine Coding - More Complex Outputs

```
always @( p_state or x ) begin : Transitions
    n_state = reset;
    case ( p_state )
    reset:
        if ( x == 1'b1 ) n_state = got1;
        else n_state = reset;
    got1:
        if ( x == 1'b0 ) n_state = got10;
        else n_state = got11;
    got10:
        if ( x == 1'b1 ) n_state = got1;
        else n_state = reset;
    got11:
        if ( x == 1'b1 ) n_state = got11;
        else n_state = got10;
    default: n_state = reset;
    endcase
end
```

... Separate Transition and Output Blocks ...

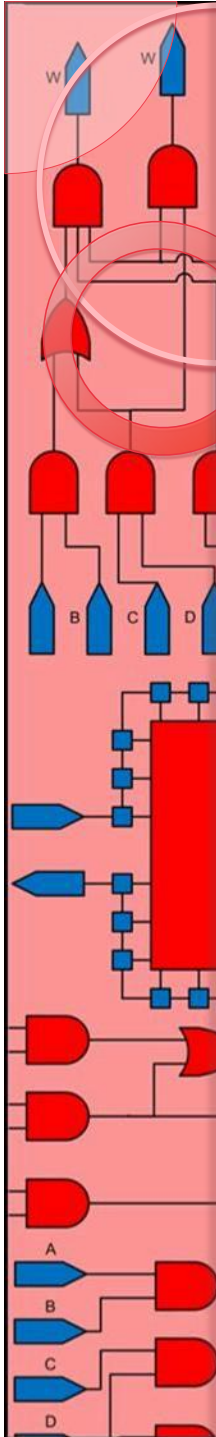


State Machines

- State Machine Coding - More Complex Outputs

```
always @( p_state or x ) begin: Outputting
    z = 0;
    case ( p_state )
        reset:  z = 1'b0;
        got1:   z = 1'b0;
        got10:  if ( x == 1'b1 ) z = 1'b1;
                else z = 1'b0;
        got11:  if ( x==1'b1 ) z = 1'b0;
                else z = 1'b1;
        default: z = 1'b0;
    endcase
end
```

... Separate Transition and Output Blocks ...



State Machines

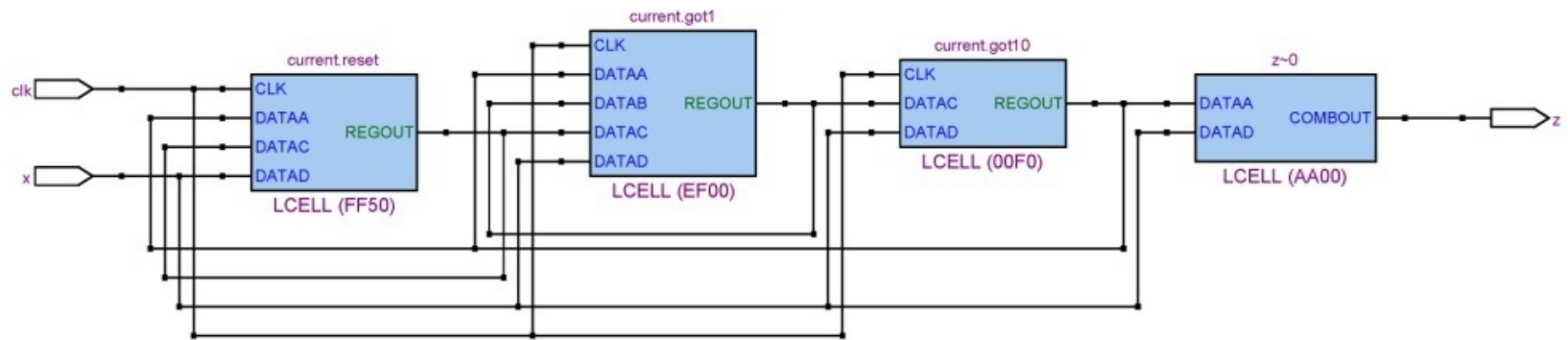
- State Machine Coding - More Complex Outputs

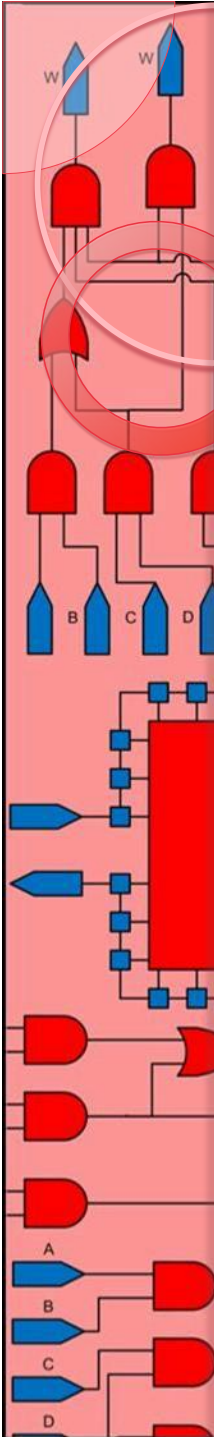
```
always @ ( posedge clk ) begin: Registering
    if( rst ) p_state <= reset;
    else if( en ) p_state <= n_state;
end
```

... Separate Transition and Output Blocks

State Machines

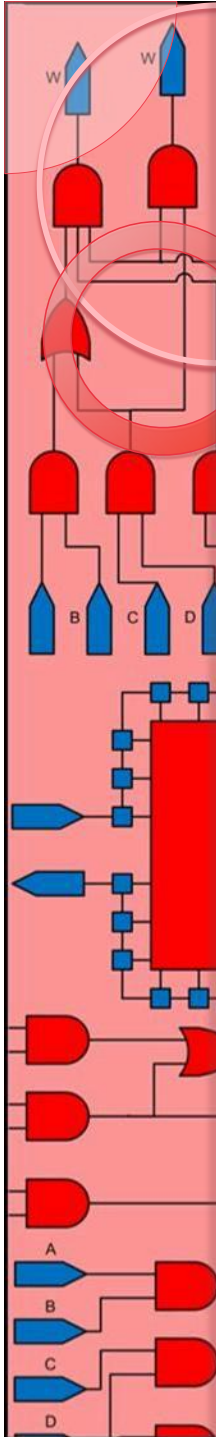
- State Machine Synthesis





Summary

- State Machine Coding
 - Moore Detector
 - A Mealy Machine Example
 - Huffman Coding Style
 - More Complex Outputs



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

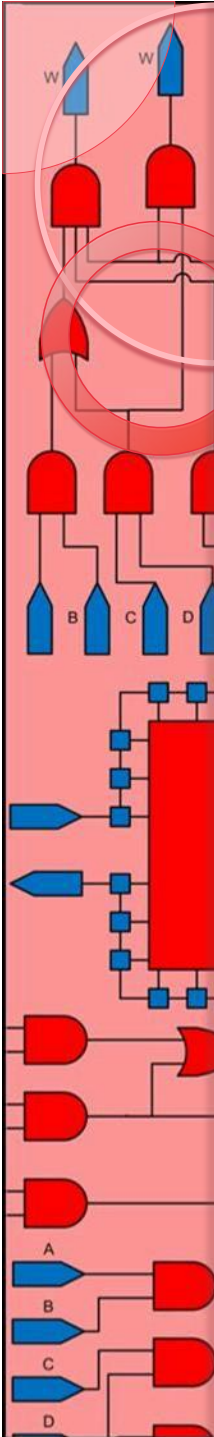
Basics of Digital Design at RT Level with Verilog

Lecture 14: Memories (Used in RTL Datapath)

July 2014

© 2013-2014 Zain Navabi

179



Memories

- Memory declaration
- Memory part selection
- Memory module
- Summary



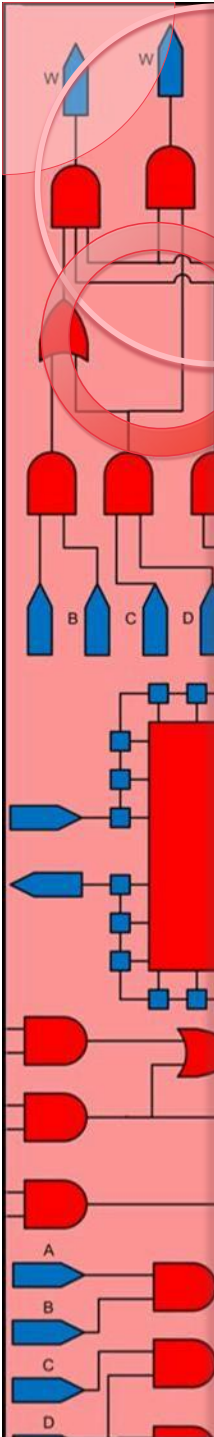
Memories

- Memory declaration

```
reg [7:0] a_array [0:1023][0:511];
```

```
reg [7:0] Areg;           // An 8-bit vector  
reg Amem [7:0];          // A memory of 8 1-bit elements  
reg Dmem [7:0][0:3];      // 2-Dim mem, 1-bit elements  
reg [7:0] Cmem [0:3];     // A memory of four 8-bit words  
reg [2:0] Dmem [0:3][0:4]; // 2-Dim mem, 3-bit elements  
reg [7:0] Emem [0:1023];  // A memory of 1024 8-bit words
```

Array Declaration Examples



Memories

- Memory part selection
 - Array Indexing

```
Areg [5:3] selects bits 5, 4, and 3
Areg [5-:4] selects bits 5, 4, 3, and 2
Areg [2+:4] selects bits 5, 4, 3, and 2
```

```
Cmem [Areg[7:6]] // extracts Cmem word
                  // addressed by Areg[7:6]
```

```
Emem [Emem[0]] // extracts Emem word
                // addressed by Emem[0]
```



Memories

- Memory part selection
 - Array Indexing

```
Emem [355][3:0] // 4 LSB of location 355
```

```
Emem [355][3-:4] // 4 bits starting from 3, down
```

```
Emem [355:358] // Illegal.  
                // This is not a 4-word block
```

```
Dmem [0][2]58] // Illegal.  
                // This is not a 4-word block
```

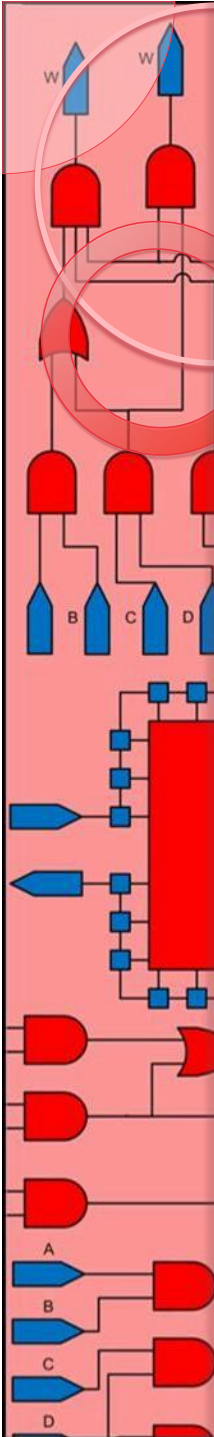


Memories

- Memory module
 - Standard clocked register file

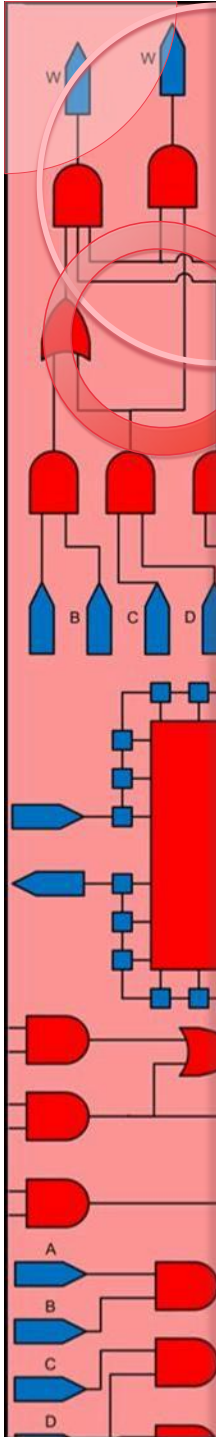
```
module memory (input [7:0] inbus,  
               output [7:0] outbus,  
               input [9:0] addr, input clk, rw);  
  reg [7:0] mem [0:1023];  
  assign outbus = rw ? mem [addr] : 8'bz;  
  always @ (posedge clk)  
    if (rw == 0) mem [addr] <= inbus;  
endmodule
```

Memory Description



Summary

- Memory declaration
- Memory part selection
- Memory module
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

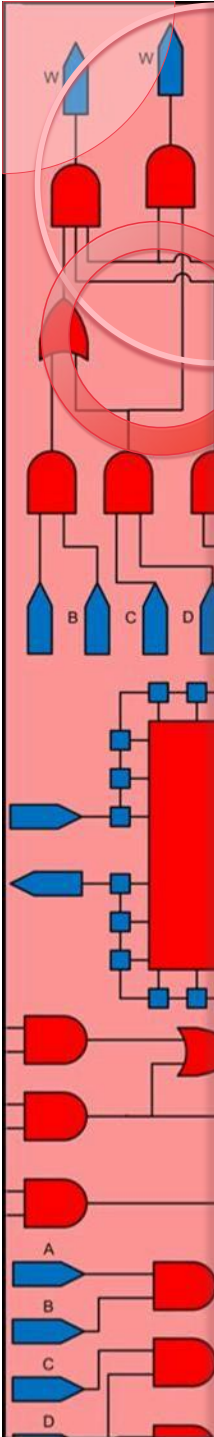
Basics of Digital Design at RT Level with Verilog

Lecture 15: Writing Testbenches

July 2014

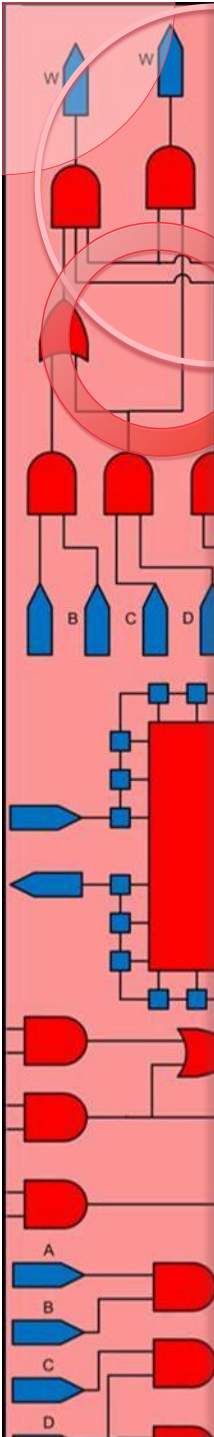
© 2013-2014 Zain Navabi

186



Writing Testbenches

- Periodic data
- Random data
- Timed data
- Summary



Writing Testbenches

- Generating Periodic Data
- Random Input Data
- Timed Data

```
module moore_detector ( input x, rst, clk, output z );
    parameter [1:0] a=0, b=1, c=2, d=3;
    reg [1:0] current;
    always @( posedge clk )
        if ( rst )    current = a;
        else case ( current )
            a : current <= x ? b : a ;
            b : current <= x ? b : c ;
            c : current <= x ? d : a ;
            d : current <= x ? b : c ;
            default : current <= a;
        endcase
    assign z = (current==d) ? 1'b1 : 1'b0;
endmodule
```

Circuit Under Test



Writing Testbenches

- Generating Periodic Data

```
`timescale 1 ns / 100 ps
module test_moore_detector;
    reg x, reset, clock;
    wire z;

    moore_detector uut ( x, reset, clock, z );
    initial begin
        clock=1'b0; x=1'b0; reset=1'b1;
    end
    initial #24 reset=1'b0;
    always #5 clock=~clock;
    always #7 x=~x;
endmodule
```

Generating Periodic Data



Writing Testbenches

- Random Input Data

```
`timescale 1 ns / 100 ps

module TESTER_test_moore_detector;
    reg x, reset, clock;
    wire z;
    moore_detector uut( x, reset, clock, z );
    initial    begin
        clock=1'b0; x=1'b0; reset=1'b1;
        #24 reset=1'b0;
    end
    initial #165 $finish;
    always #5 clock=~clock;
    always #7 x=$random;
endmodule
```

Random Data Generation

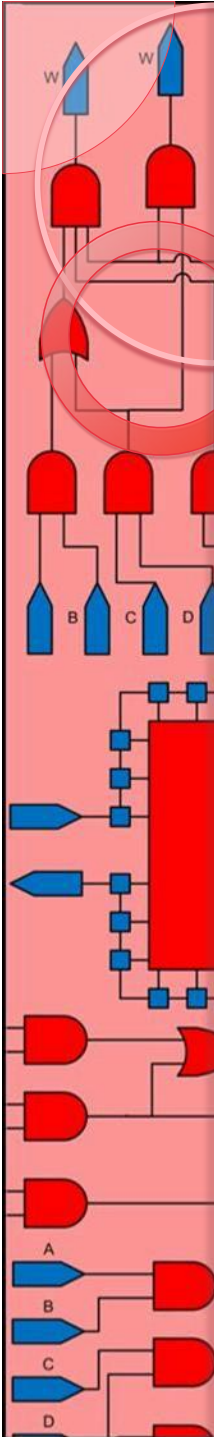


Writing Testbenches

- Timed Data

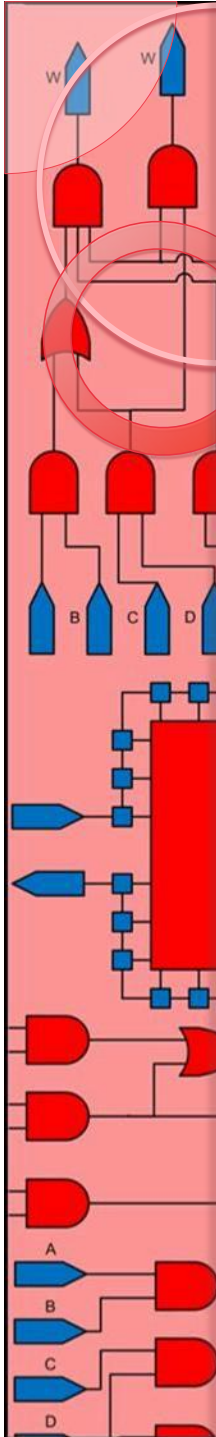
```
`timescale 1ns/100ps
module test_moore_detector;
    reg x, reset, clock;
    wire z;
    moore_detector uut( x, reset, clock, z );
    initial begin
        clock=1'b0; x=1'b0; reset=1'b1;
        #24 reset=1'b0;
    end
    always #5 clock=~clock;
    initial begin
        #7  x=1;          #5  x=0;          #18 x=1;
        #21 x=0;          #11 x=1;          #13 x=0;
        #33 $stop;
    end
end
endmodule
```

Timed Test Data Generation



Summary

- Periodic data
- Random data
- Timed data
- Summary



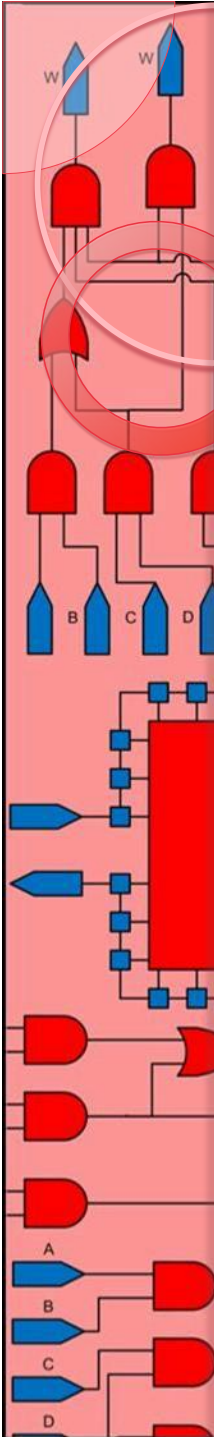
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

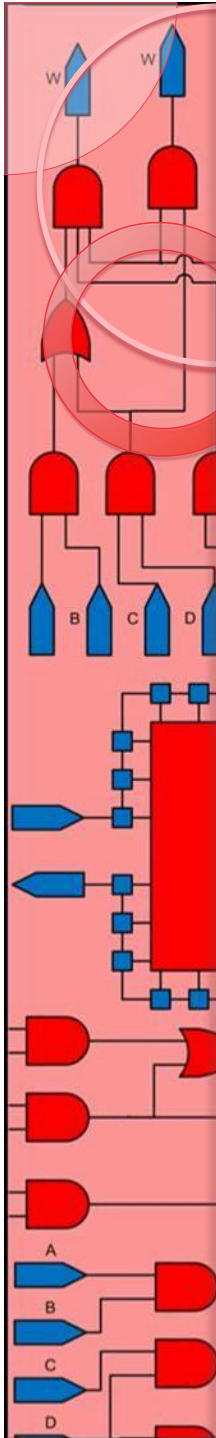
Basics of Digital Design at RT Level with Verilog

Lecture 16 : RTL Methodology

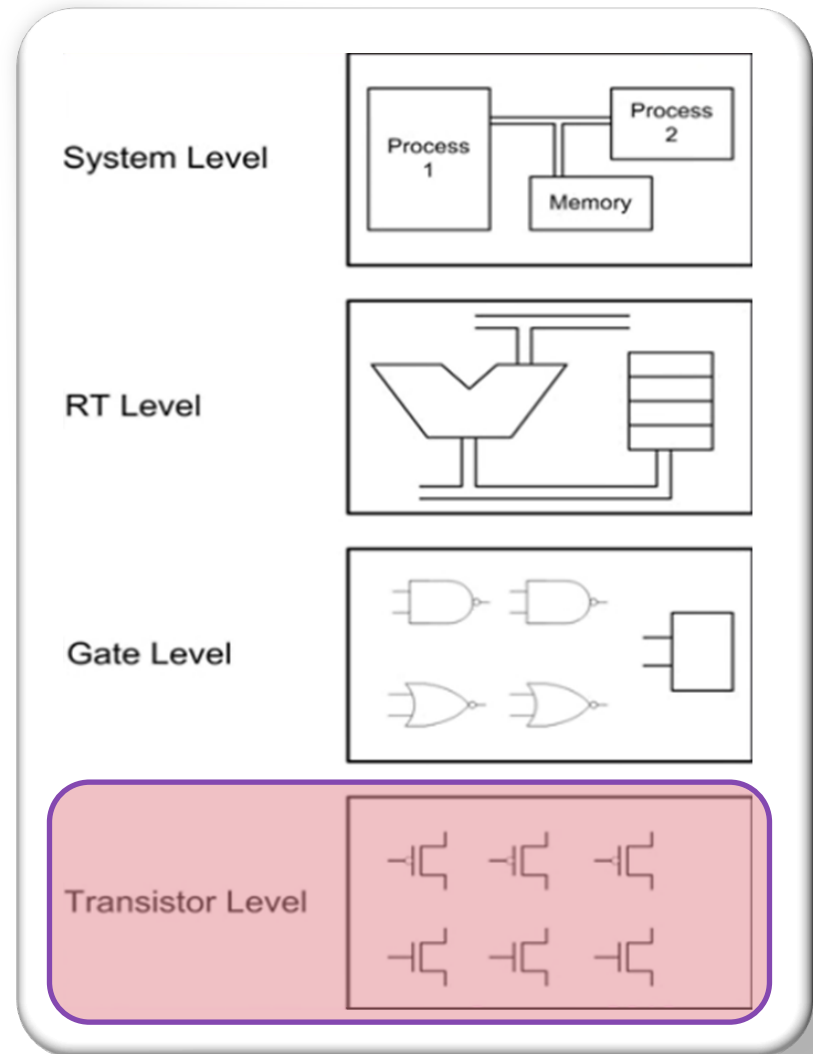
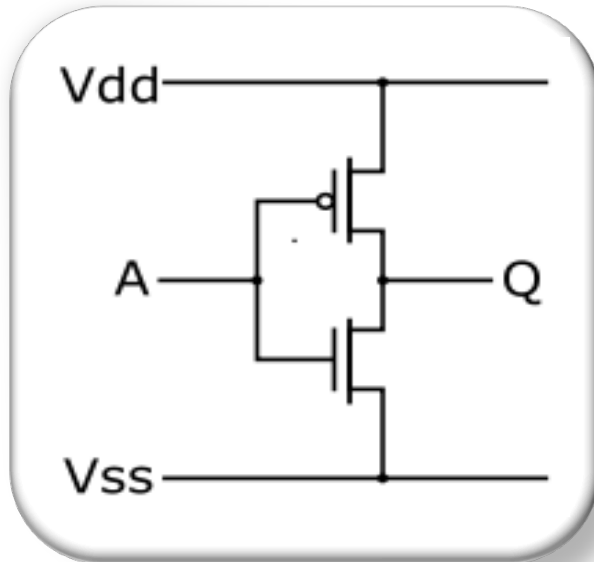


RTL Methodology

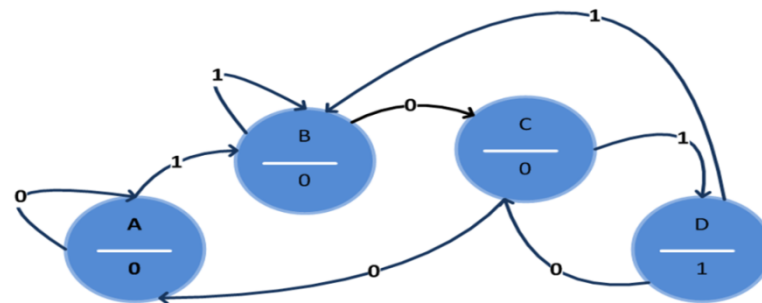
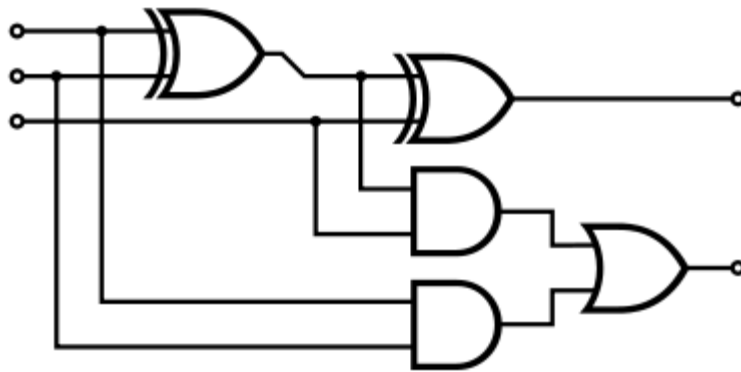
- Transistors to processing elements
- RTL design partitioning
- Example: serial adder
- Summary



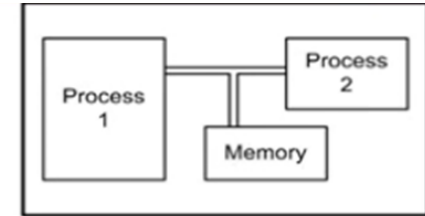
Transistors Level Design



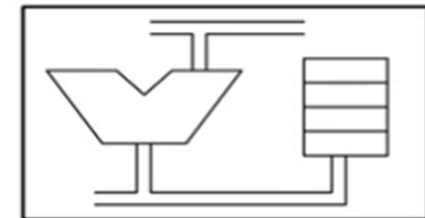
Gate Level Design



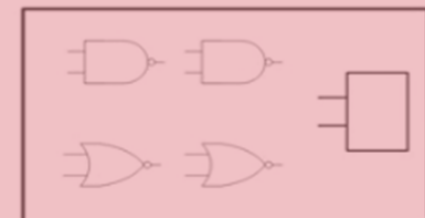
System Level



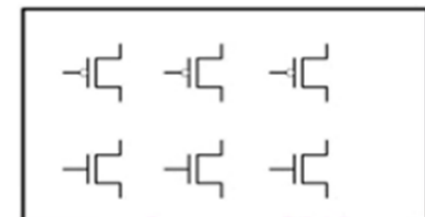
RT Level



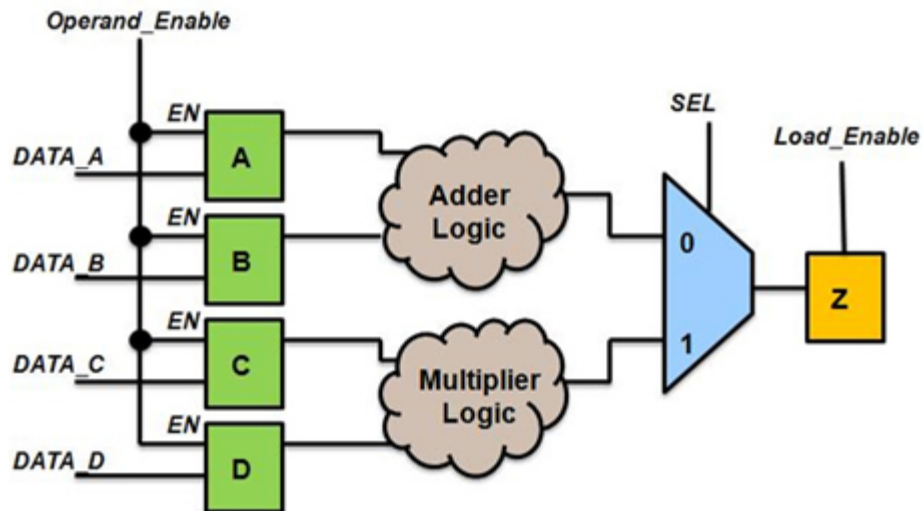
Gate Level



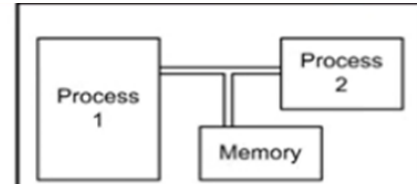
Transistor Level



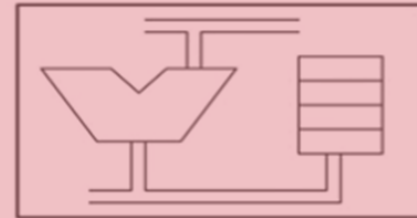
RT Level Design



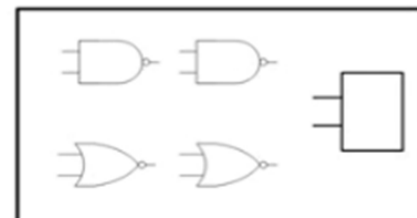
System Level



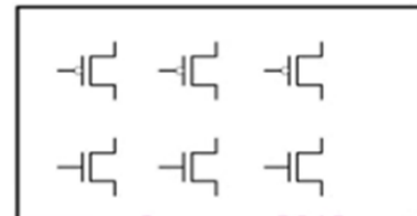
RT Level



Gate Level

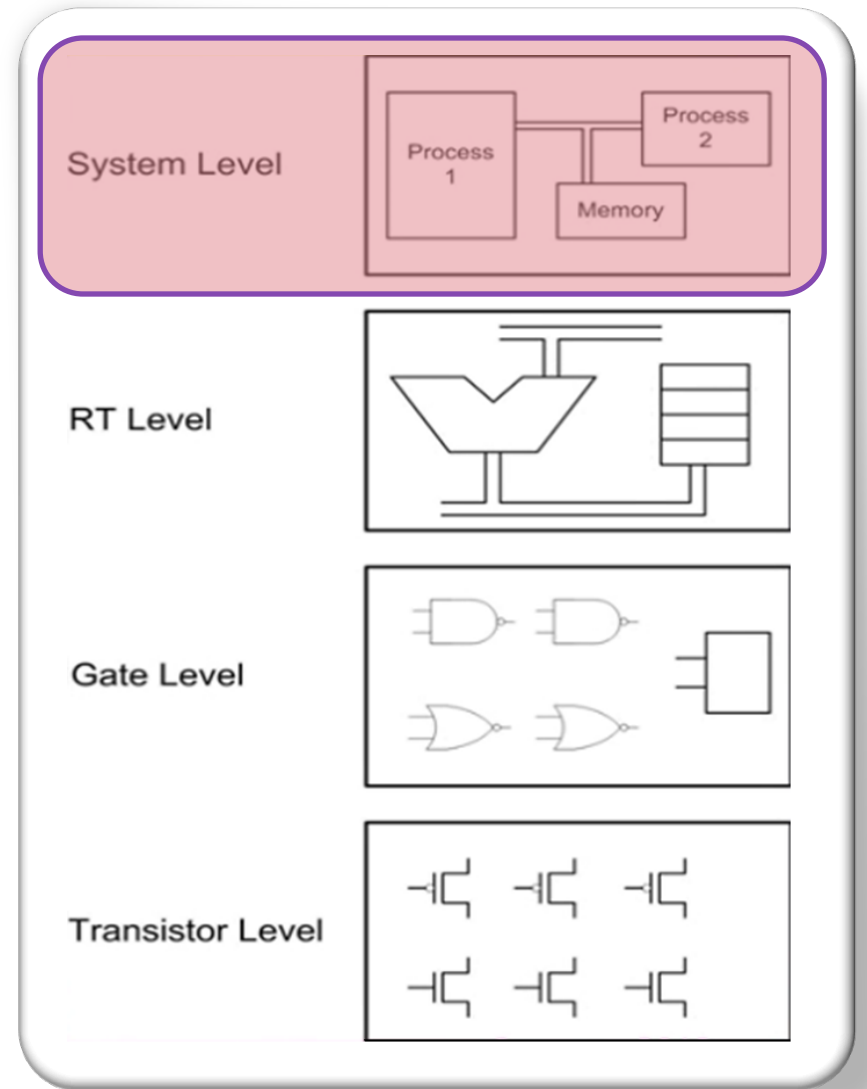
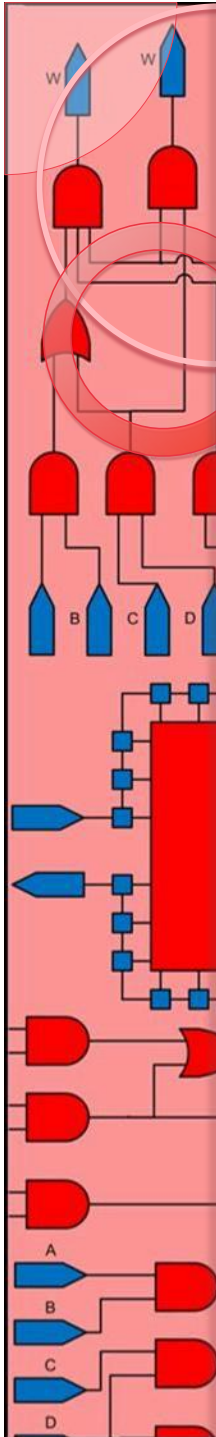


Transistor Level



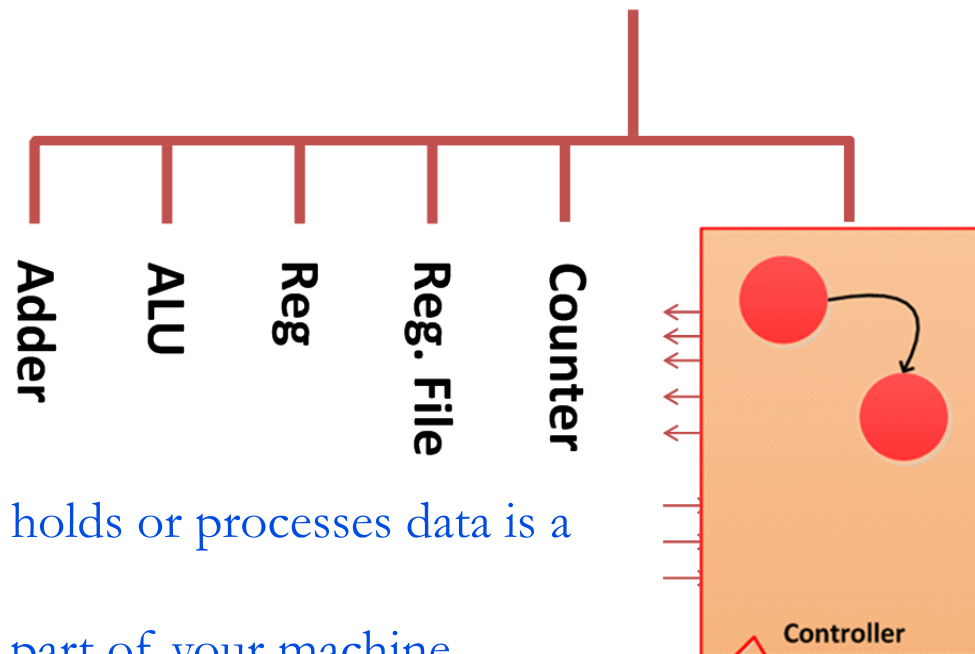
System Level Design

Connecting CPU's, memories,
etc. together via channel
interconnections



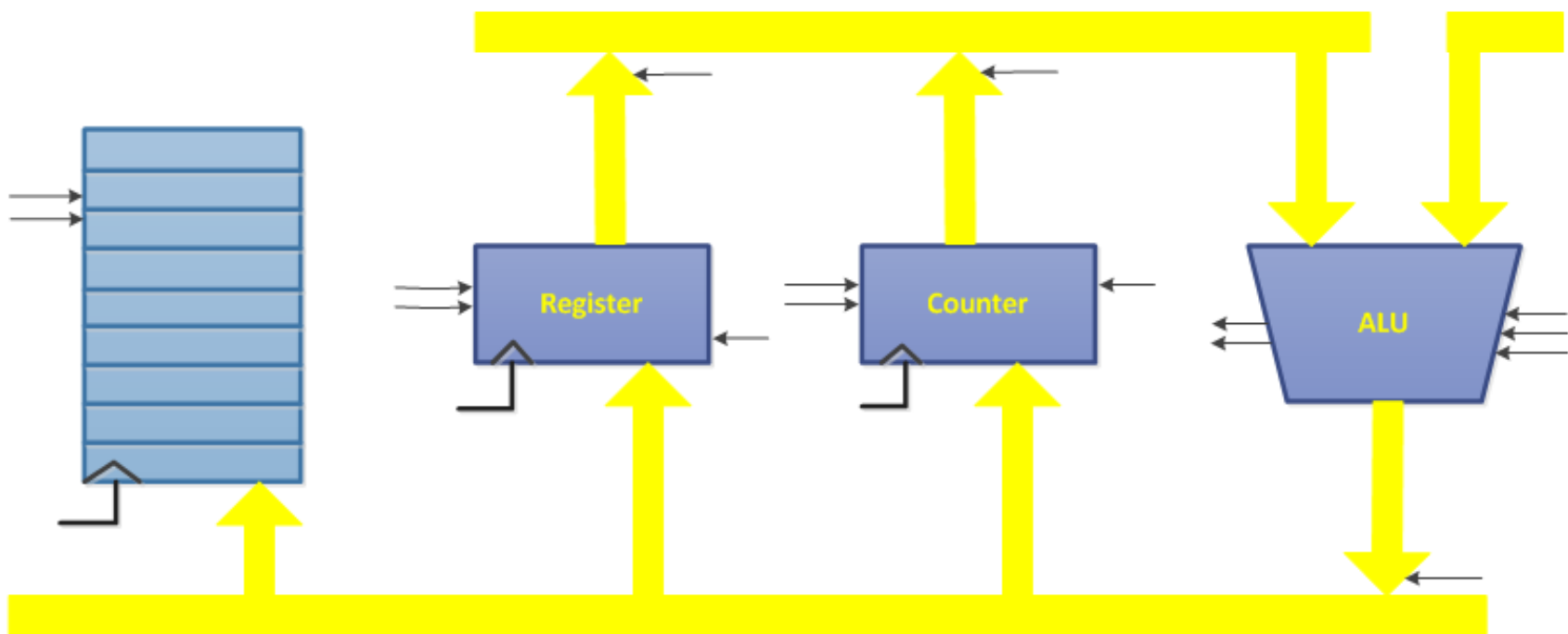
Datapath And Controller

RTL Design



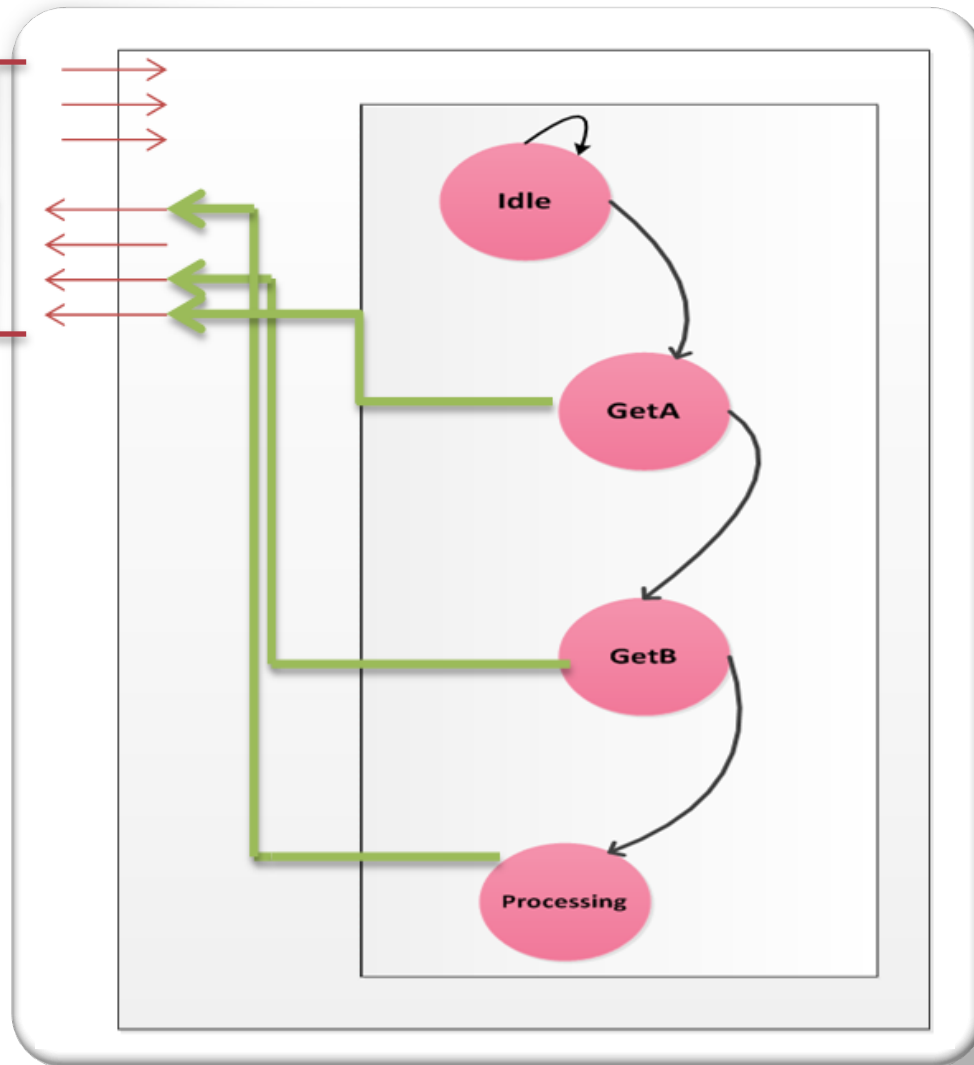
- Any element that passes, holds or processes data is a datapath element
- Controller is the thinking part of your machine
- You should decide how to wire datapath elements
- When designing datapath, don't be concern about how control signals are Issued

RTL Datapath Example



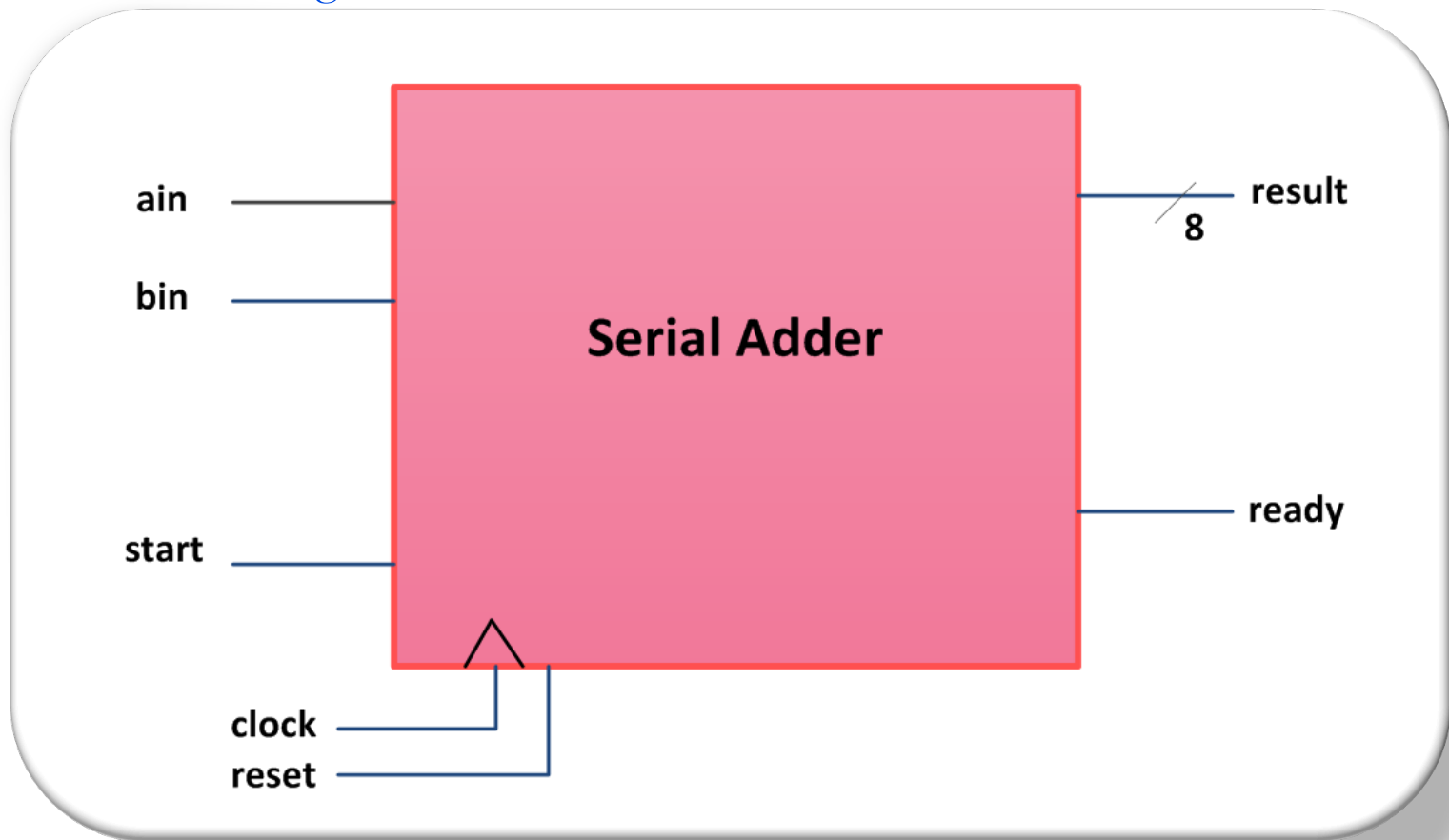
RTL Controller Example

Signals that go to or
Come from datapath

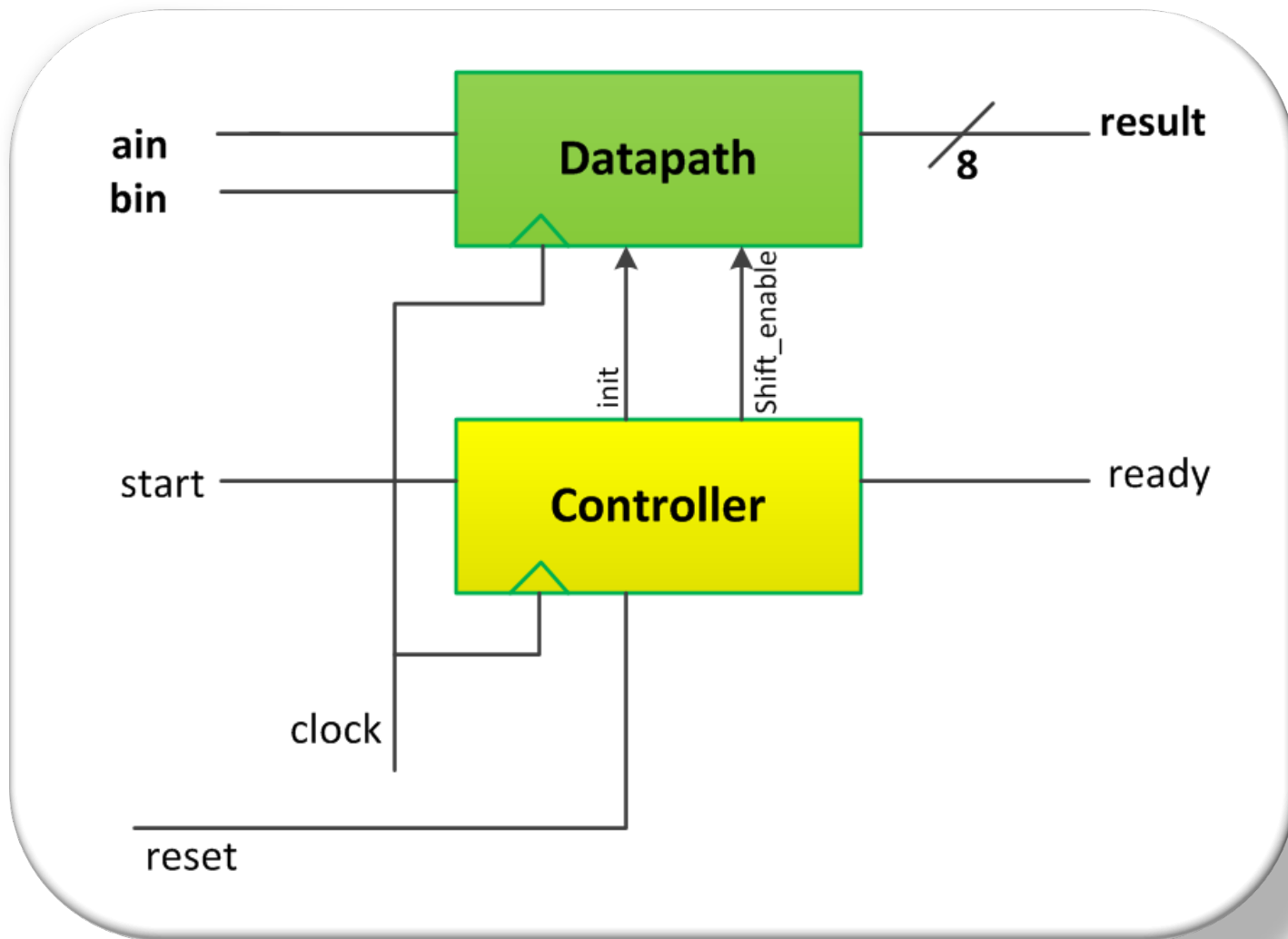


Example: Serial Adder

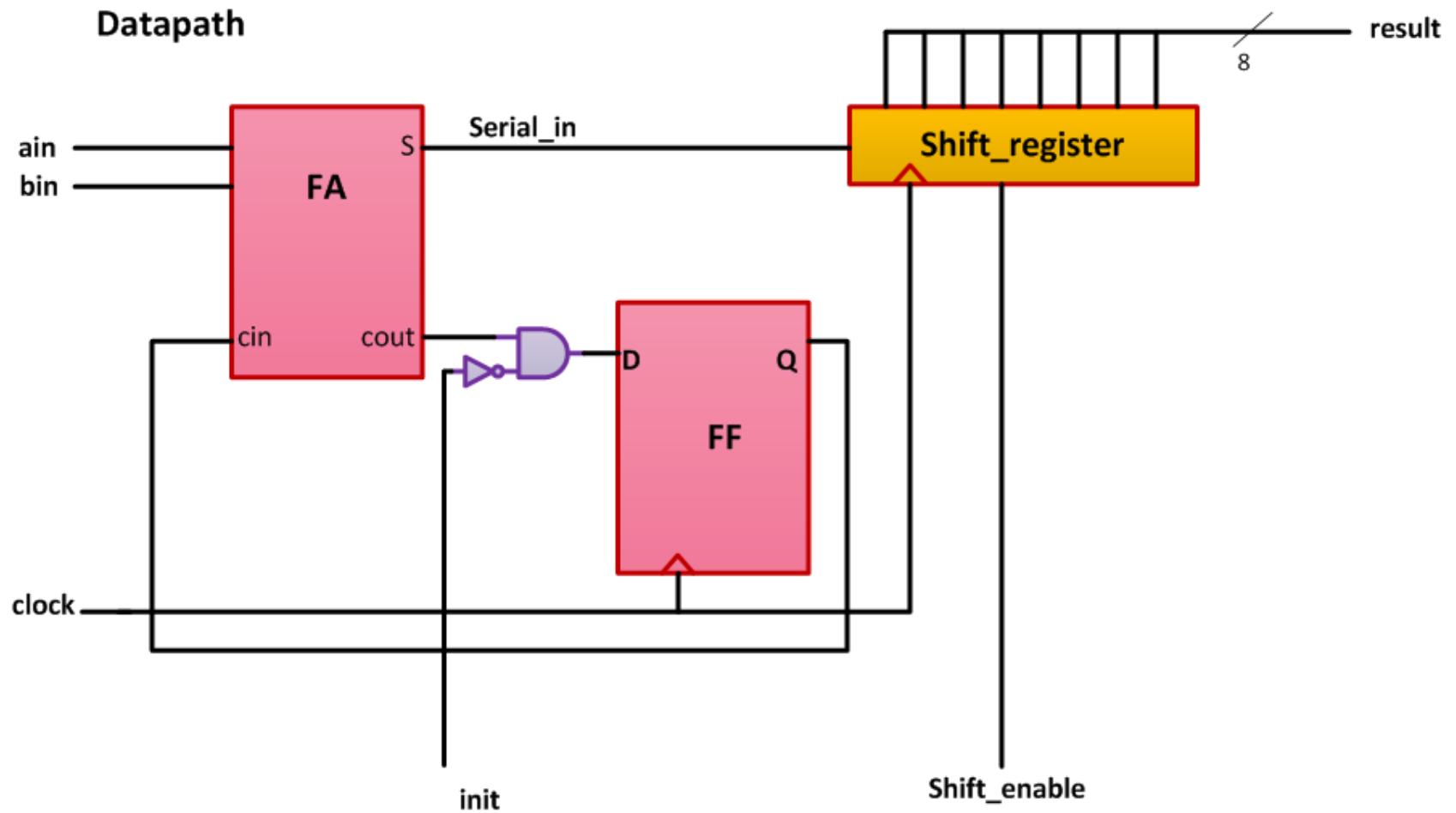
After “start” signal is issued, 8 pairs of serial bits on ain and bin are added and an 8-bit result is generated.



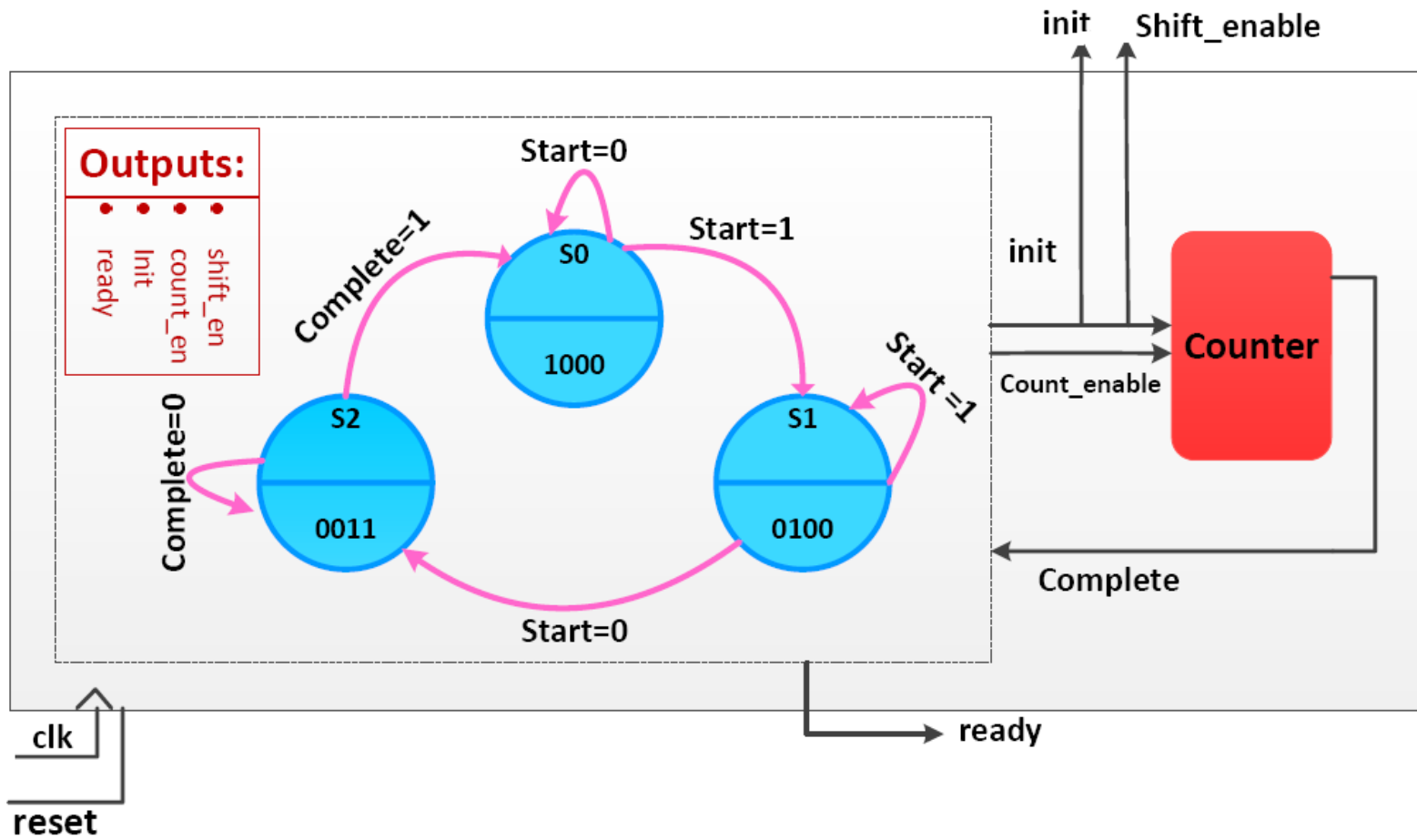
Design Partitioning



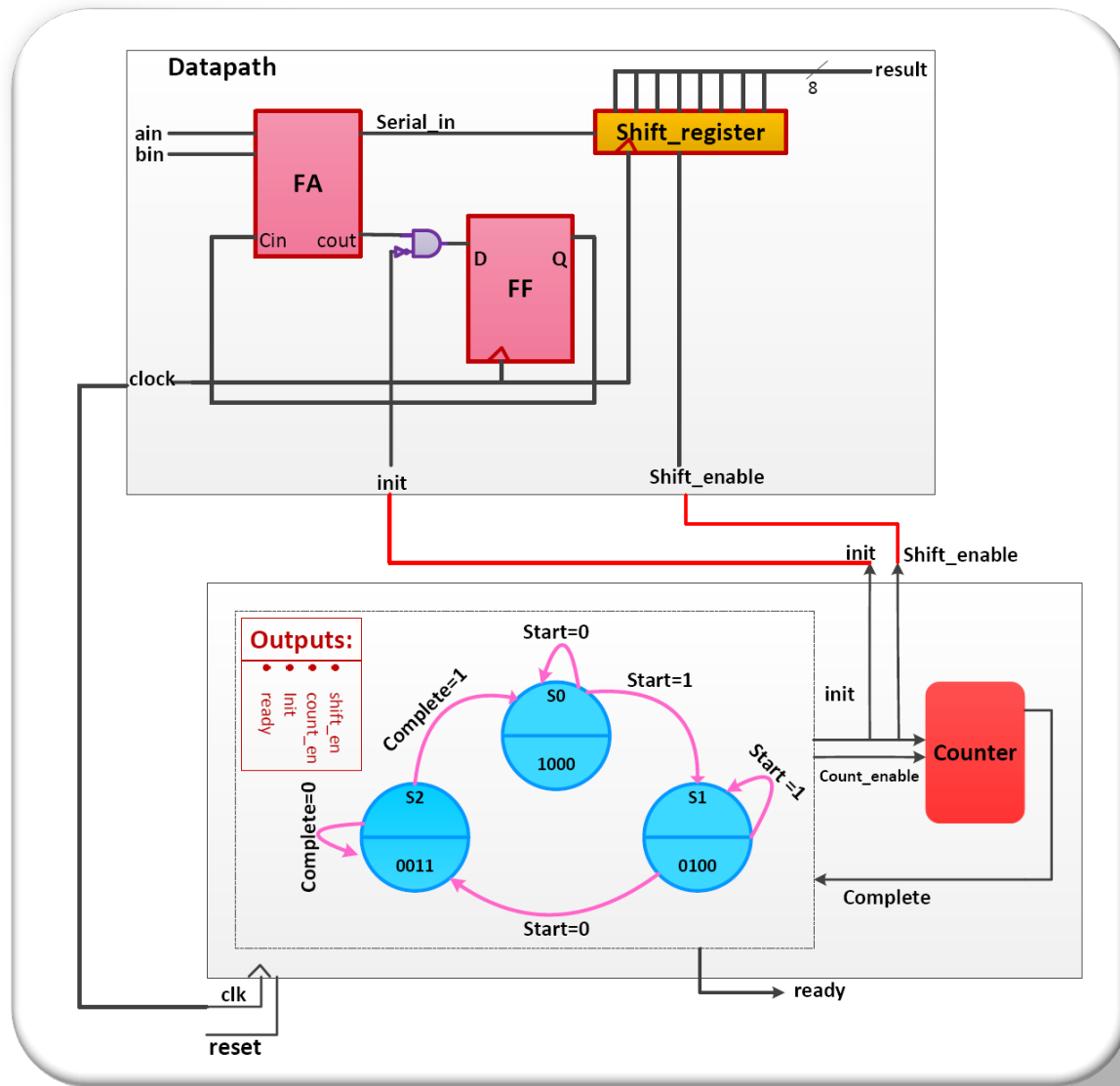
Datapath Design

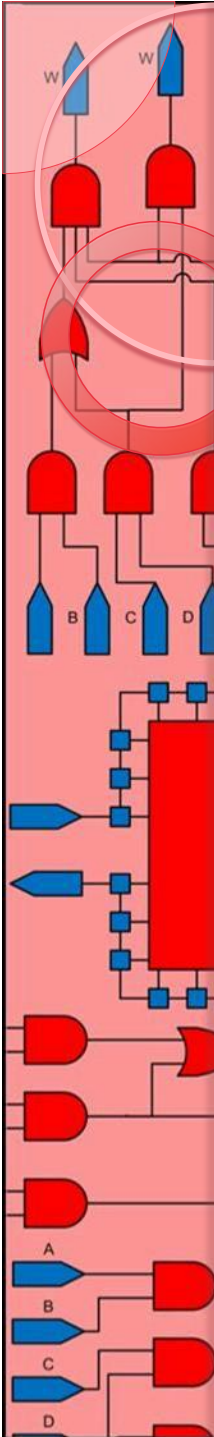


Controller Design



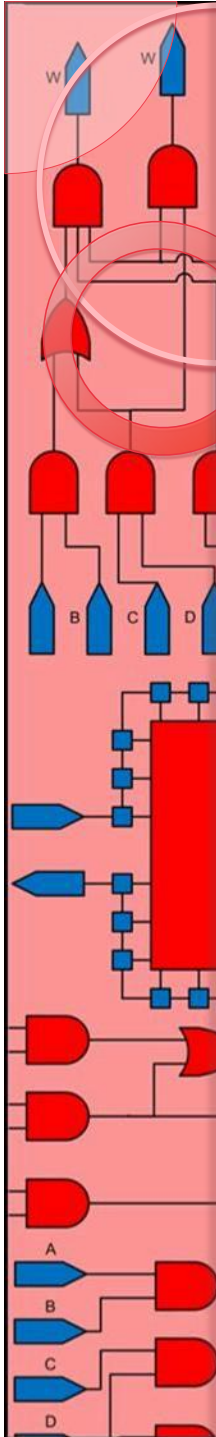
Wiring Of Datapath And Controller





Summary

- Transistors to processing elements
- Design abstraction levels
- Datapath/controller partitioning
- Example: serial adder
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

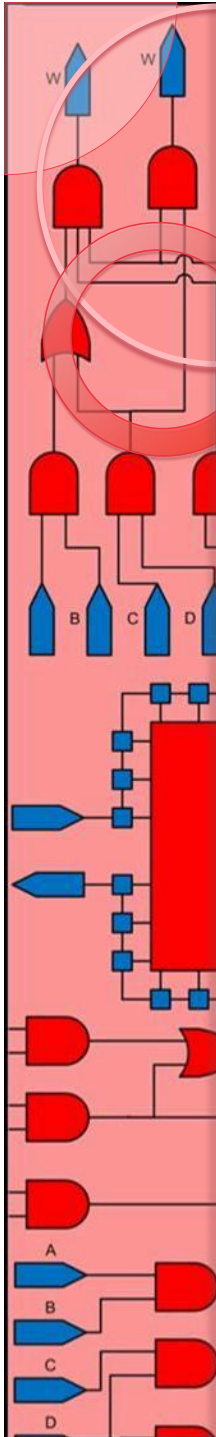
Basics of Digital Design at RT Level with Verilog

Lecture 17: RTL Timing

July 2014

© 2013-2014 Zain Navabi

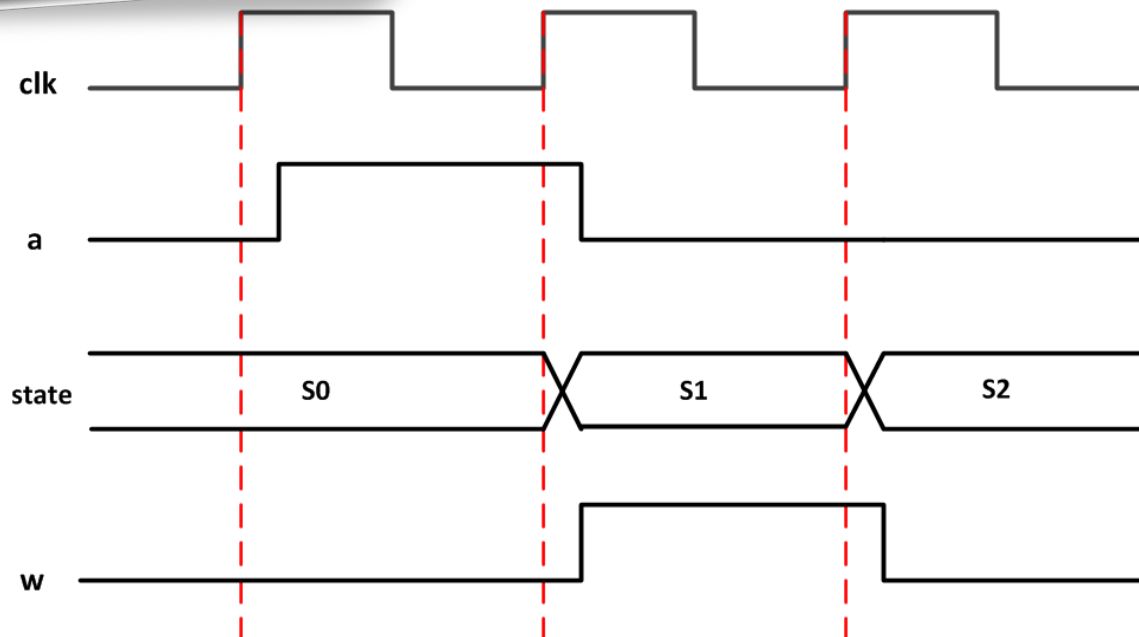
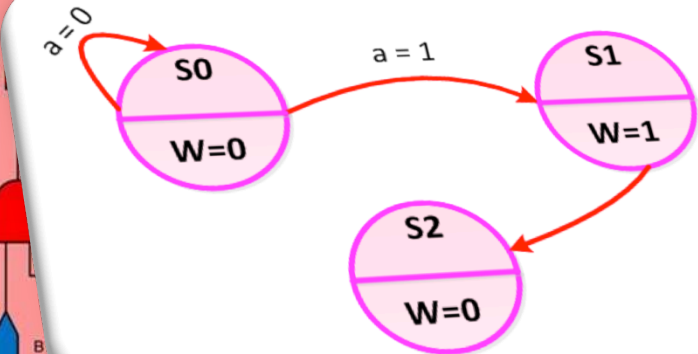
208



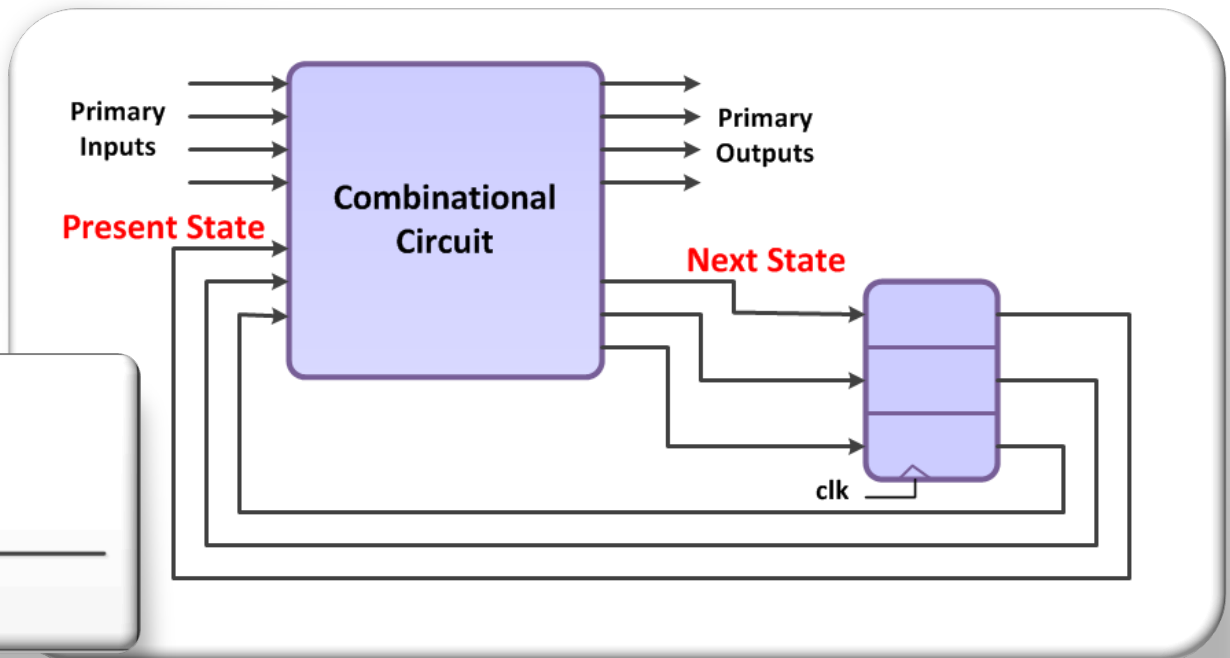
RTL Timing

- State machine timing
- Datapath/controller timing
- Datapath timing
- Summary

State Machine Timing

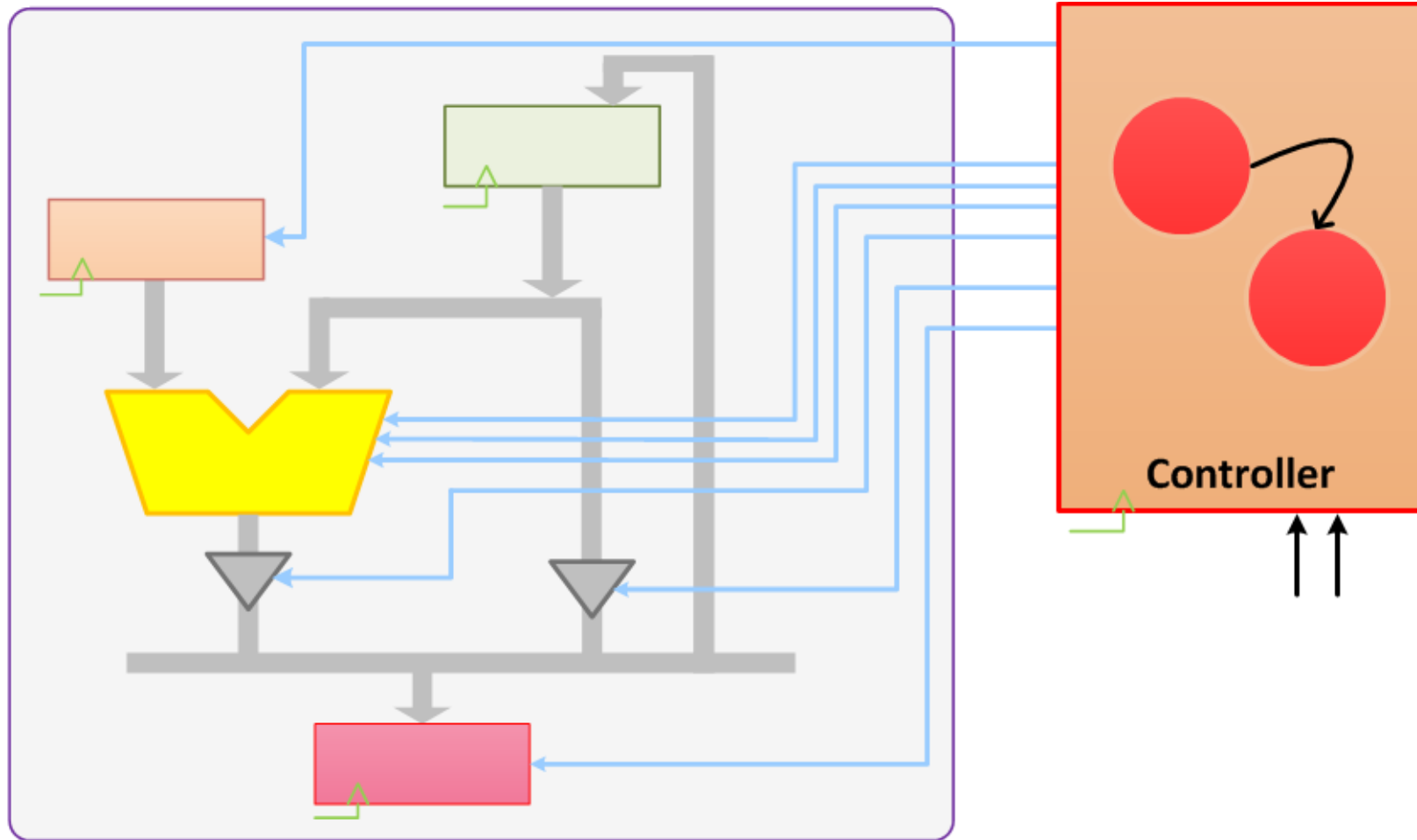


Huffman Model of a Sequential Circuit

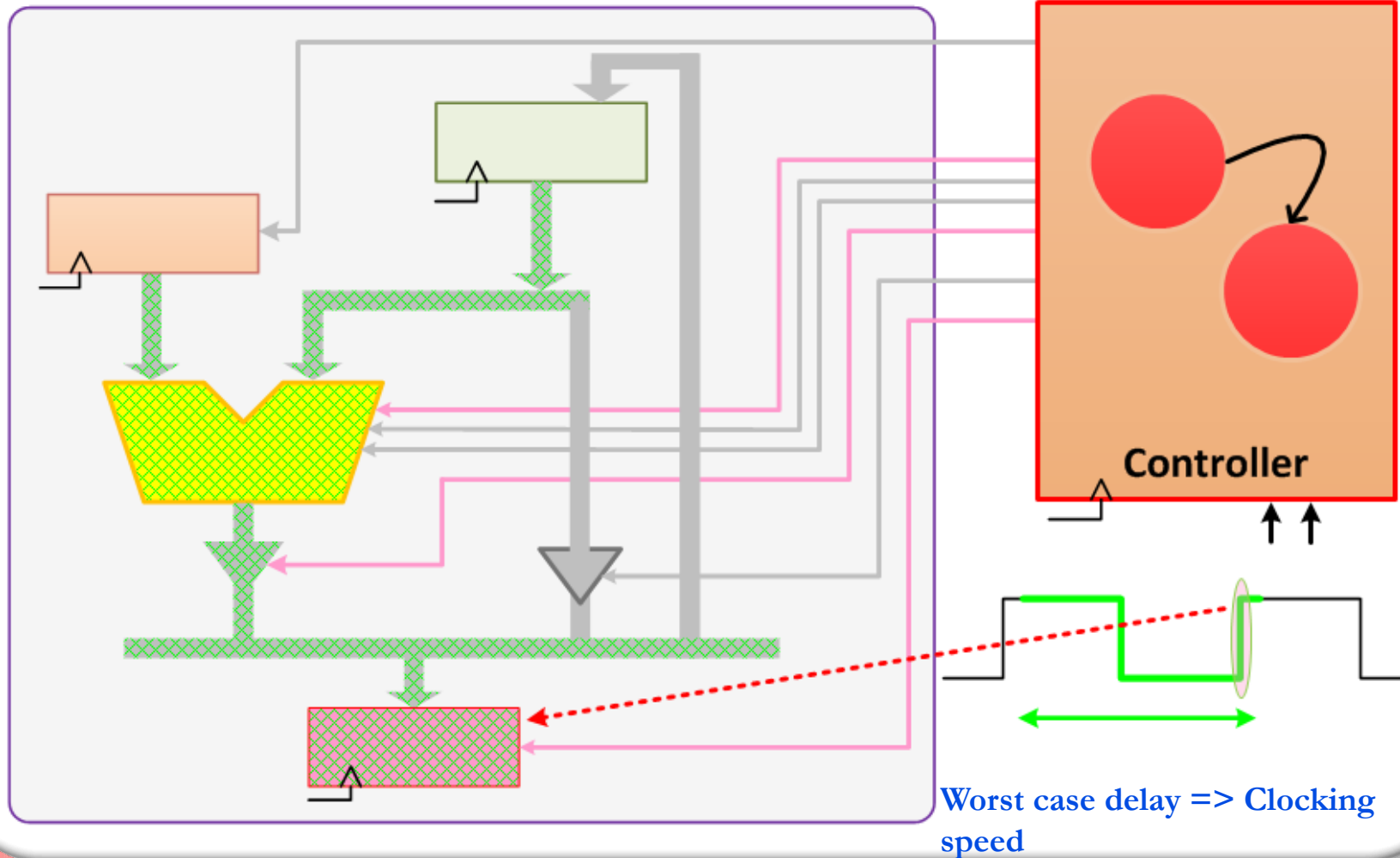


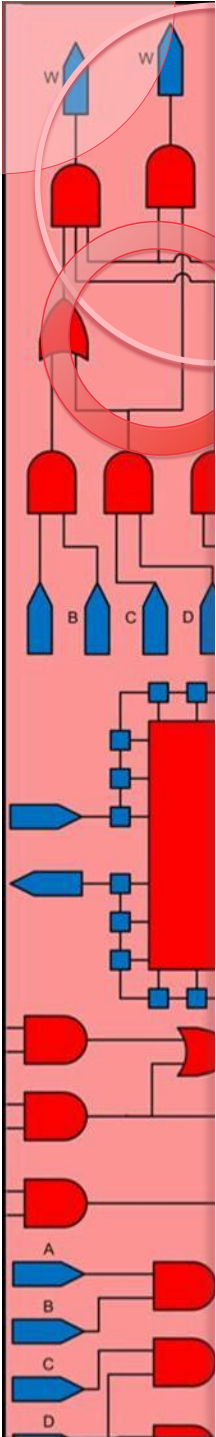
- This much of time allows the combination of PS and PI to propagate and reach to NS and PO output.
- Primary outputs producing their values right after the clock ticks and remain in that way for some time after the next clock tick.

Datapath/Controller Timing



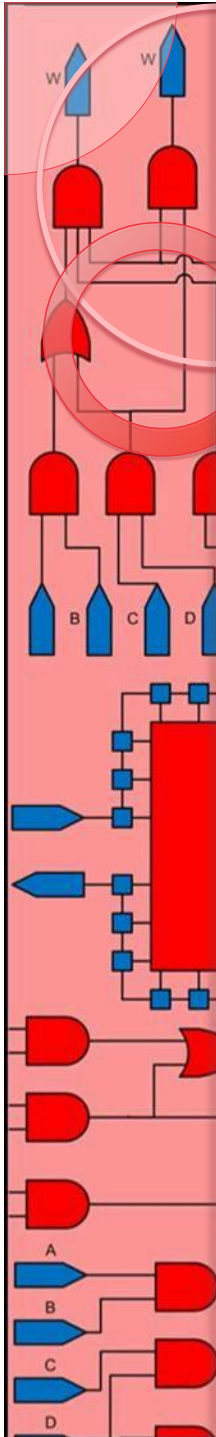
Datapath/Controller Timing





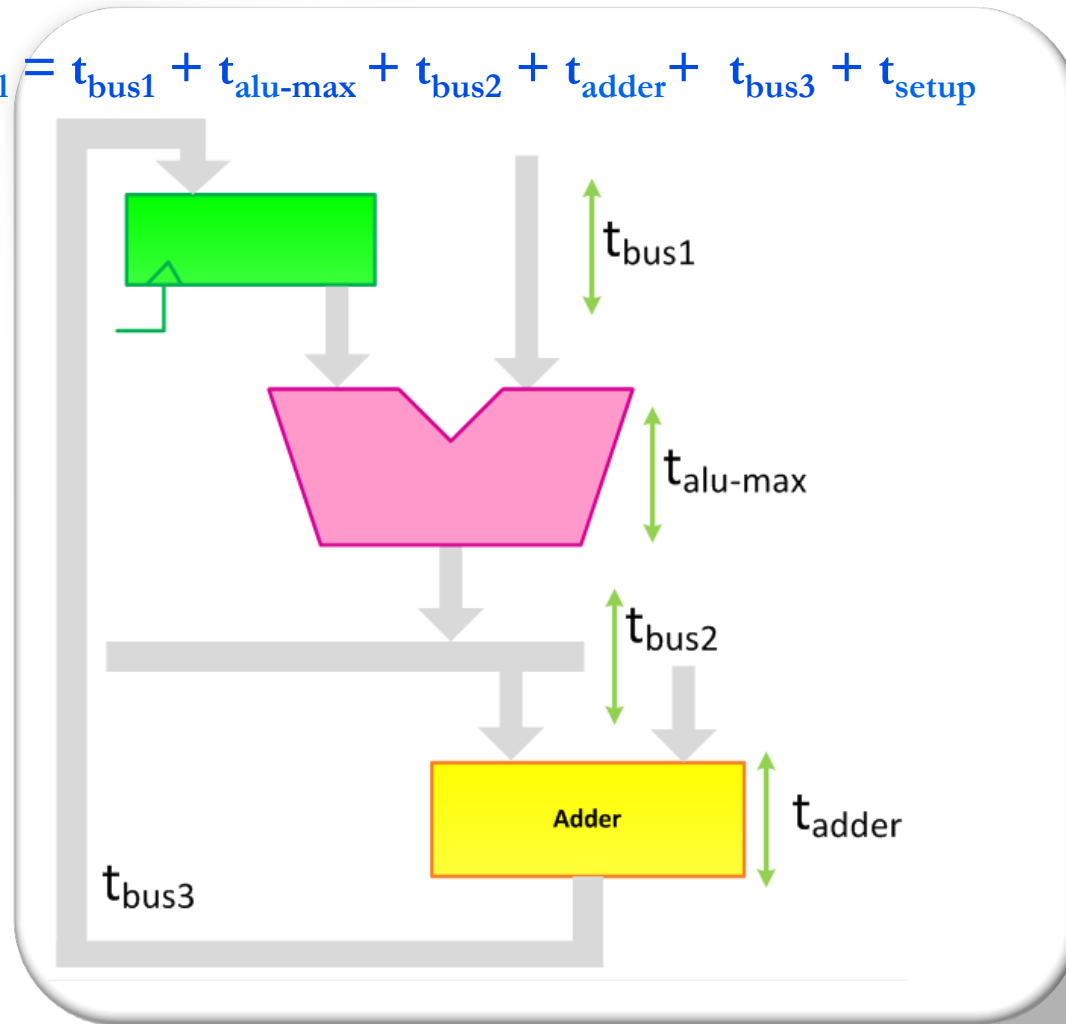
Datapath Timing

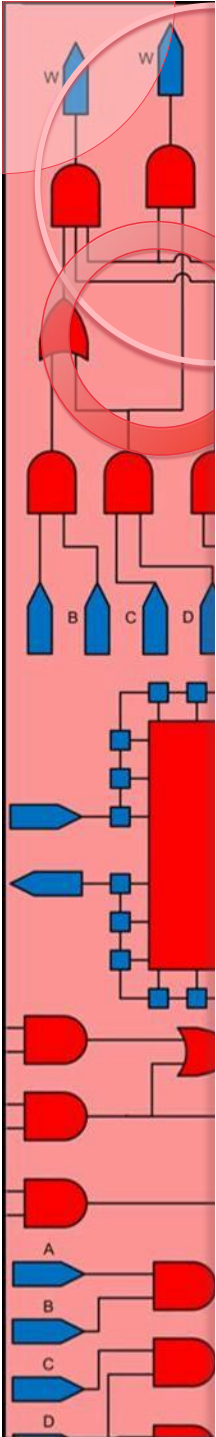
- For finding the clock frequency we should consider the worst case delay(bottleneck) of the datapath
- If we have ALU in datapath, we should take the delay of ALU operation with the most delay, as ALU delay
- Selection of each bus has also its own delay
- Each register has setup time delay



Datapath Timing

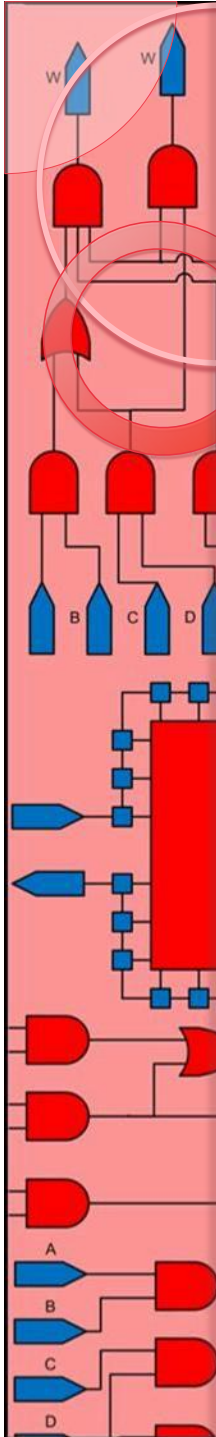
$$t_{\text{total}} = t_{\text{bus1}} + t_{\text{alu-max}} + t_{\text{bus2}} + t_{\text{adder}} + t_{\text{bus3}} + t_{\text{setup}}$$





Summary

- Signals from the controller
- Controller timing
- Datapath timing
- Datapath / Controller Synchronization
- Longest path propagation
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

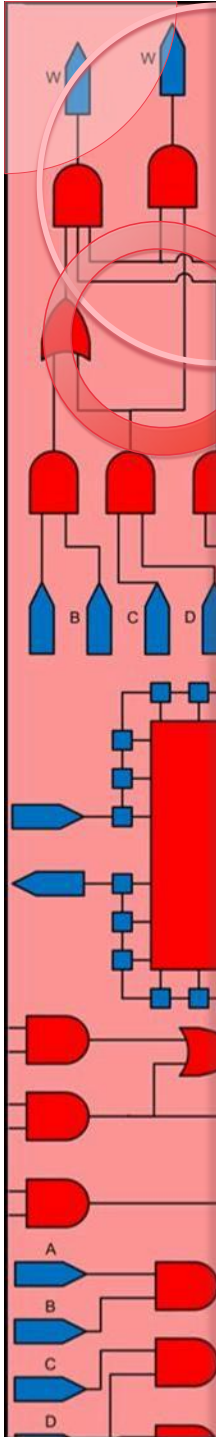
Basics of Digital Design at RT Level with Verilog

Lecture 18: RTL Processing Element Design

July 2014

© 2013-2014 Zain Navabi

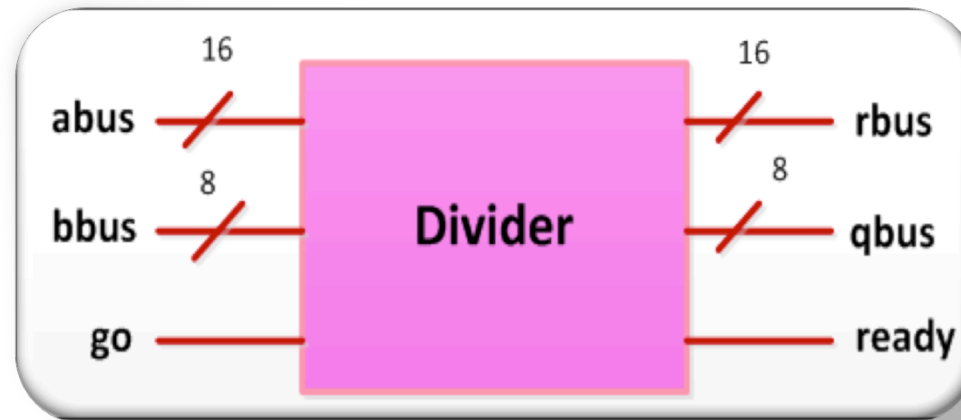
217



RTL Processing Element Design

- Simple unsigned integer divider
- Datapath
- Controller
- Verilog description
- Testbench
- Summary

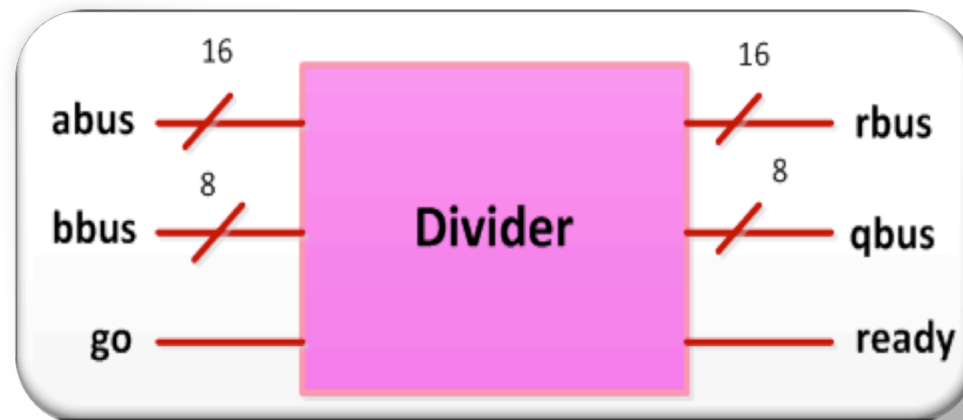
Unsigned Integer Divider



Emphasis is on a simple algorithm, and not very efficient.

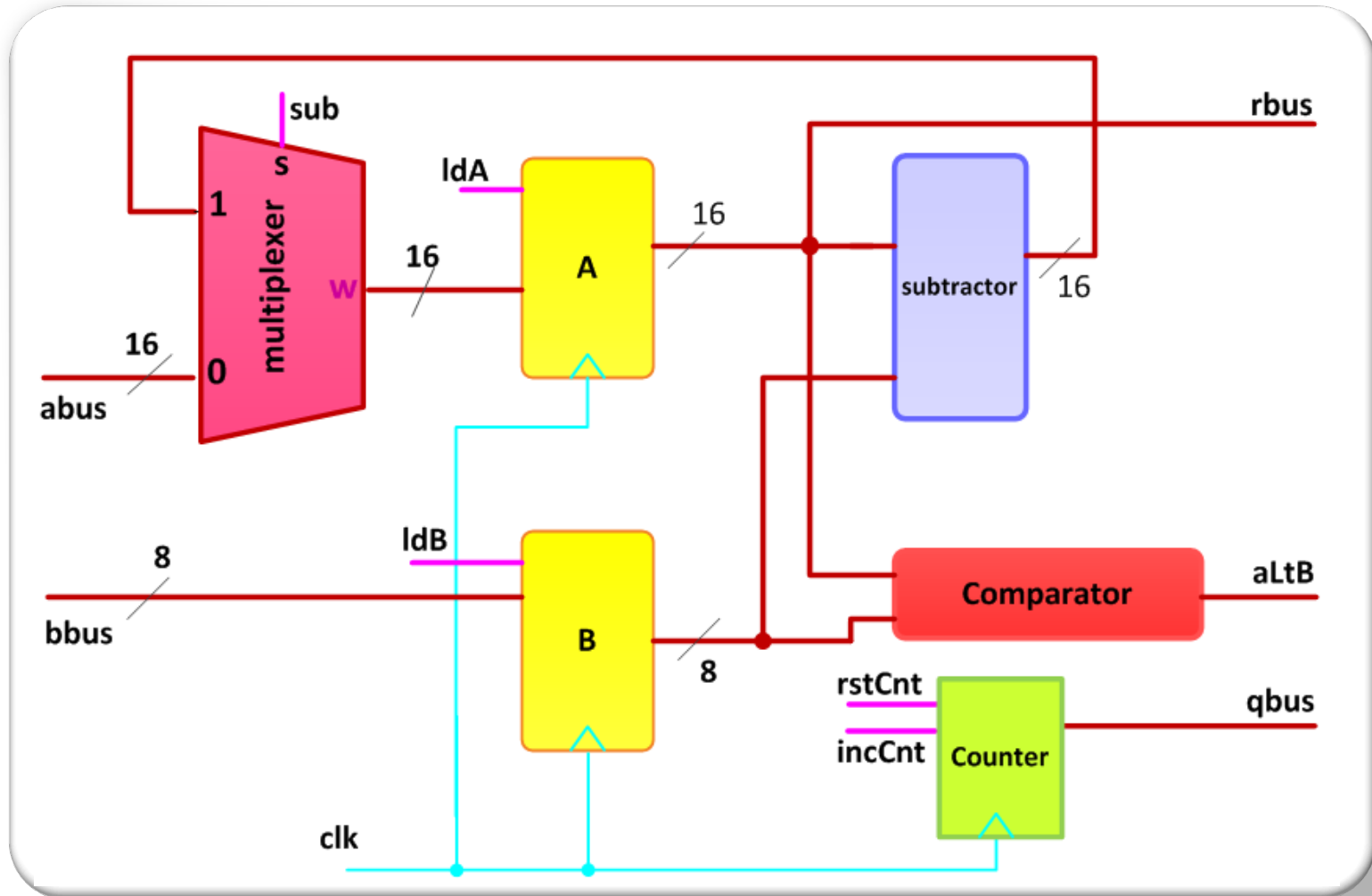
- Dividing A by B by subtracting B from A until the remainder is less than B
- The number of times that subtraction successfully happens is the quotient
- The value that is left behind is the remainder

Unsigned Integer Divider

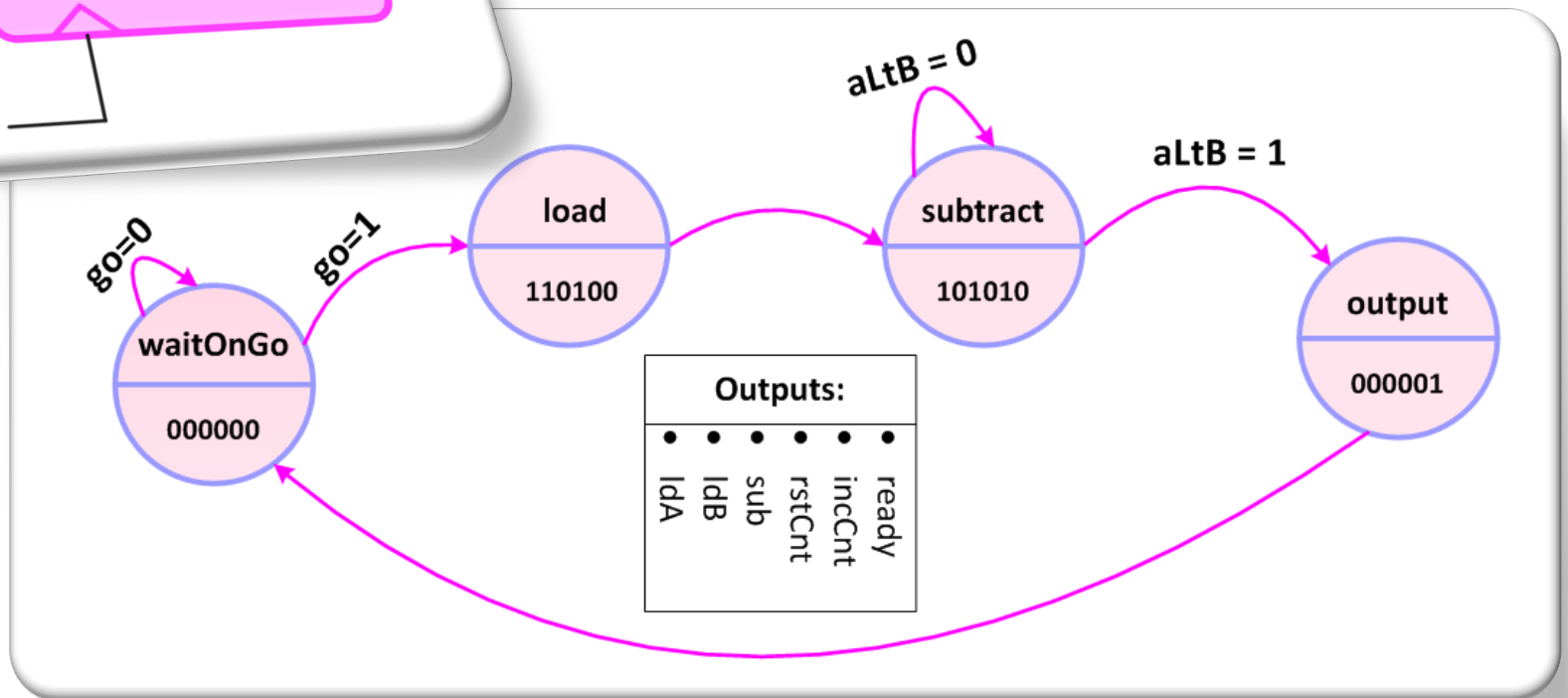
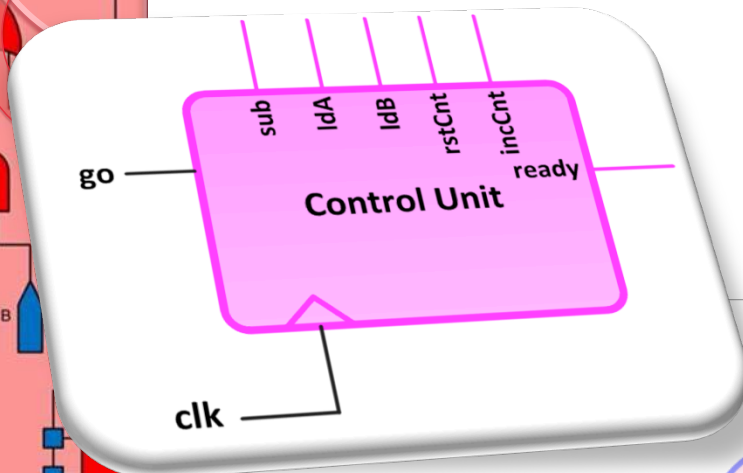
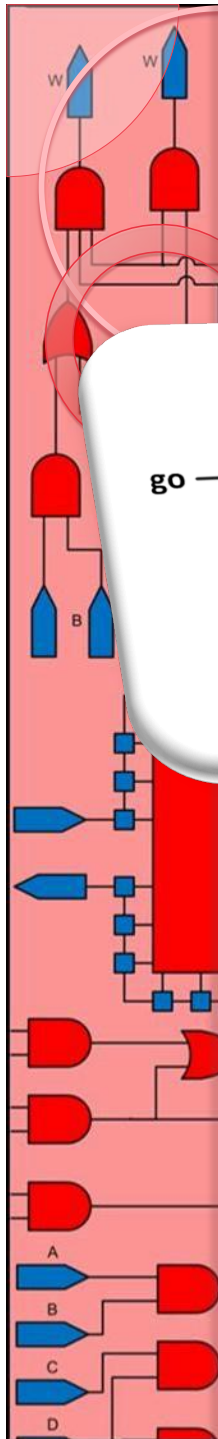


- The circuit has a 16-bit input bus for A, an 8-bit input bus for B and two 8-bit buses for remainder and quotient
- After a complete pulse happens on go, the circuit reads the A and B operands and division begins
- After division is completed, circuit places R and Q on the corresponding buses and issues a one-clock duration pulse on ready

Unsigned Integer Divider Datapath



Unsigned Integer Divider Controller



Unsigned Integer Divider

Verilog Code of Controller

```
`timescale 1ns/1ns
module controller( input clk, rst, go, aLtB, output reg sub ,ldA ,ldB ,rstCnt ,incCnt ,ready);
    parameter waitOnGo = 2'b00, load = 2'b01, subtract = 2'b10, getout = 2'b11;
    reg[1:0] ps, ns;

    always @( ps, go, aLtB )
    begin
        ns = waitOnGo; ldA = 1'b0; ldB = 1'b0; sub = 1'b0;
        rstCnt = 1'b0; incCnt = 1'b0; ready = 1'b0;
        case (ps)
            waitOnGo: begin
                if(go == 1'b1) ns = load; else ns = waitOnGo;
            end
            load: begin
                ldA = 1'b1; ldB = 1'b1; rstCnt = 1'b1; ns = subtract;
            end
            subtract: begin
                if(aLtB == 1'b1) ns = getout;
                else begin
                    ns = subtract; ldA = 1'b1; sub = 1'b1; incCnt = 1'b1;
                end
            end
        end
    end
```

Unsigned Integer Divider

Verilog Code of Controller

```
        getout: begin
            ready = 1'b1; ns = waitOnGo;
        end
    endcase
end

always @(posedge clk)
begin
    if (rst == 1'b1) ps = waitOnGo;
    else ps = ns;
end
```

2

```
`timescale
module cc
    param
    reg[1

    always
        h

        c
```

Unsigned Integer Divider

Verilog Code of Datapath

```
module datapath ( aBus, bBus, clk, rstCnt, incCnt, ldA, ldB, sub, rBus, qBus, aLtB);  
  
    input [15:0] aBus, bBus;  
    input clk, rstCnt, incCnt, ldA, ldB, sub;  
    output [15:0] rBus, qBus;  
    output aLtB;  
  
    wire [15:0] subOut, muxOut, aRegOut, bRegOut;  
  
    mux16_2to1 g0(aBus, subOut, sub, muxOut);  
    reg16 g1(clk, ldA, muxOut, aRegOut);  
    reg16 g2(clk, ldB, bBus, bRegOut);  
    subtractor g3(aRegOut, bRegOut, subOut);  
    comparator g4 (aRegOut, bRegOut, aLtB);  
    counter16 g5(clk, rstCnt, incCnt, qBus);  
  
    assign rBus = aRegOut;  
  
endmodule
```

Complete Design

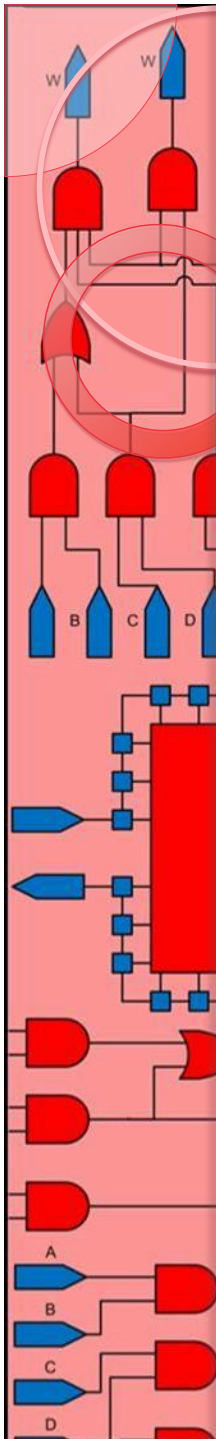
```
`timescale 1ns/1ns

module divider (clk, go, aBus, bBus, qBus, rBus, ready);
    input clk, go;
    input [15:0] aBus, bBus;
    output [15:0] qBus, rBus;
    output ready;

    wire rstCnt, incCnt, ldA, ldB, sub, aLtB;

    datapath DP(aBus, bBus, clk, rstCnt, incCnt, ldA, ldB, sub, rBus, qBus, aLtB);
    controller CTRL(clk, rst, go, aLtB, sub, ldA, ldB, rstCnt, incCnt, ready);

endmodule
```

Testbench

```

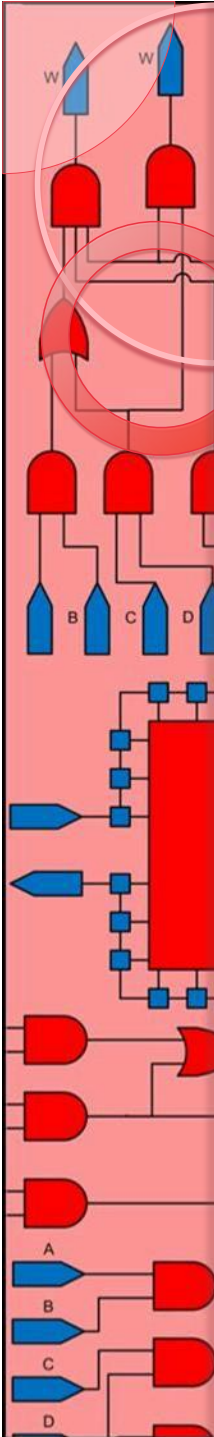
module divider_TB ();
    reg clk, go;
    reg [15:0] aBus, bBus;
    wire ready;
    wire [15:0] qBus, rBus;

    divider DIV(clk, go, aBus, bBus, qBus, rBus, ready);
    initial begin
        clk = 1'b0;
        aBus = 16'b0001000011100100;
        bBus = 16'b00000000101100101;
        go = 1'b1;
        #15;
        go = 1'b0;

        #50;
        aBus = 16'b011010100000001101;
        bBus = 16'b00001001011001100;
        go = 1'b1;
        #15;
        go = 1'b0;

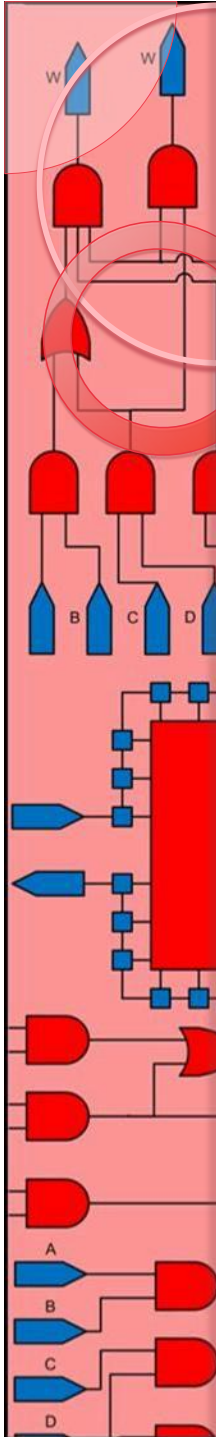
        #50;
        aBus = 16'b00001010011101111;
        bBus = 16'b00000010010111101;
        go = 1'b1;
        #15;
        go = 1'b0;
    end
end

```



Summary

- Simple unsigned integer divider
- Datapath
- Controller
- Verilog description
- Testbench
- Summary



Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

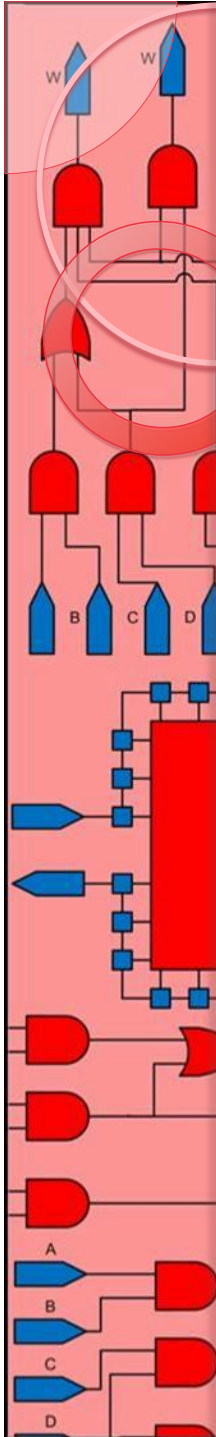
Basics of Digital Design at RT Level with Verilog

Lecture 19: Handshaking

July 2014

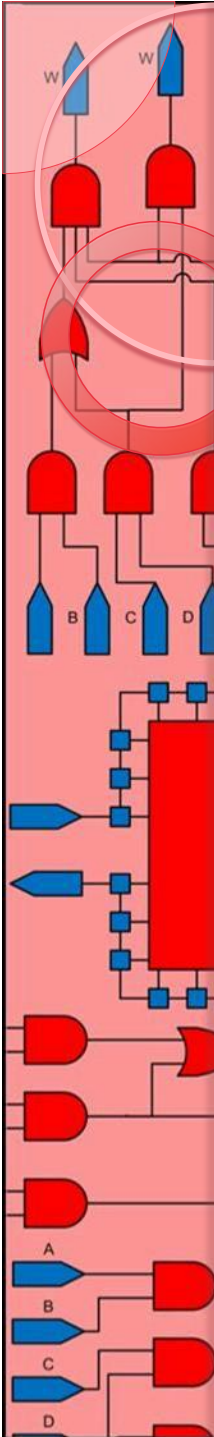
© 2013-2014 Zain Navabi

229



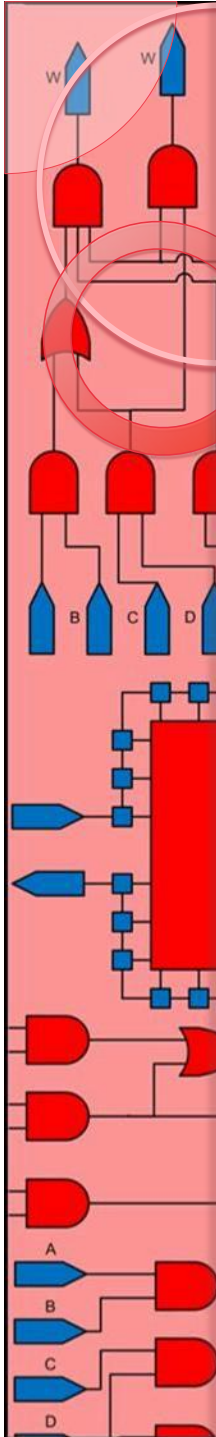
Handshaking

- Handshaking definition
- Types of handshaking
 - Handshaking between two systems
 - Handshaking for accessing a shared bus
 - Memory handshaking
 - DMA mode or burst mode
- Handshaking with timing details



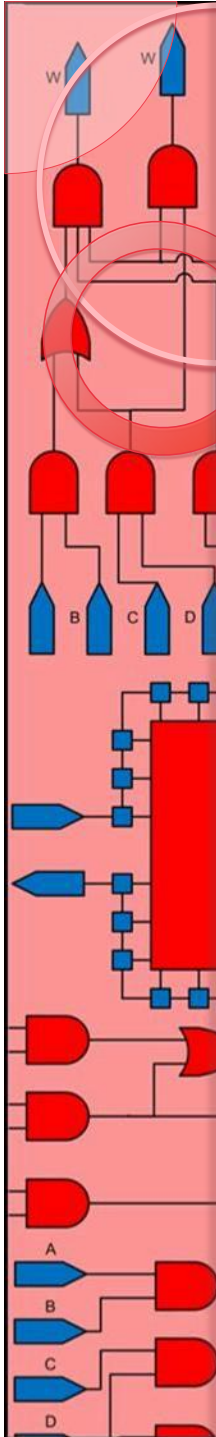
Why Handshaking?

- Two systems want to communicate data and they don't necessarily have the same timing
- The systems have to send some signals before the actual data is transmitted
- Handshaking implementation is a part of the control of the system



Between two RTLs Handshaking

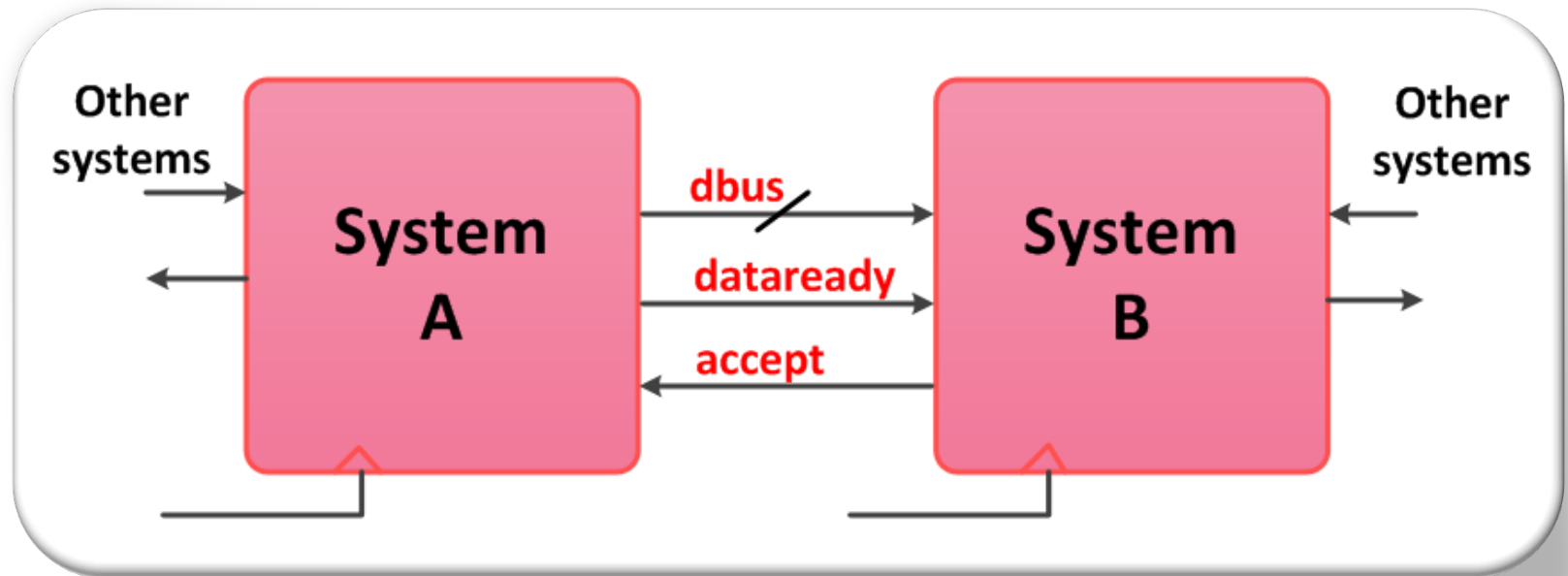
- Two systems want to communicate data and they don't necessarily have the same timing
- The systems have to send some signals before the actual data is transmitted
- Handshaking implementation is a part of the control of the system



Types of Handshaking

- Handshaking between two systems
- Handshaking for accessing a shared bus
- Memory handshaking
- DMA mode or burst mode

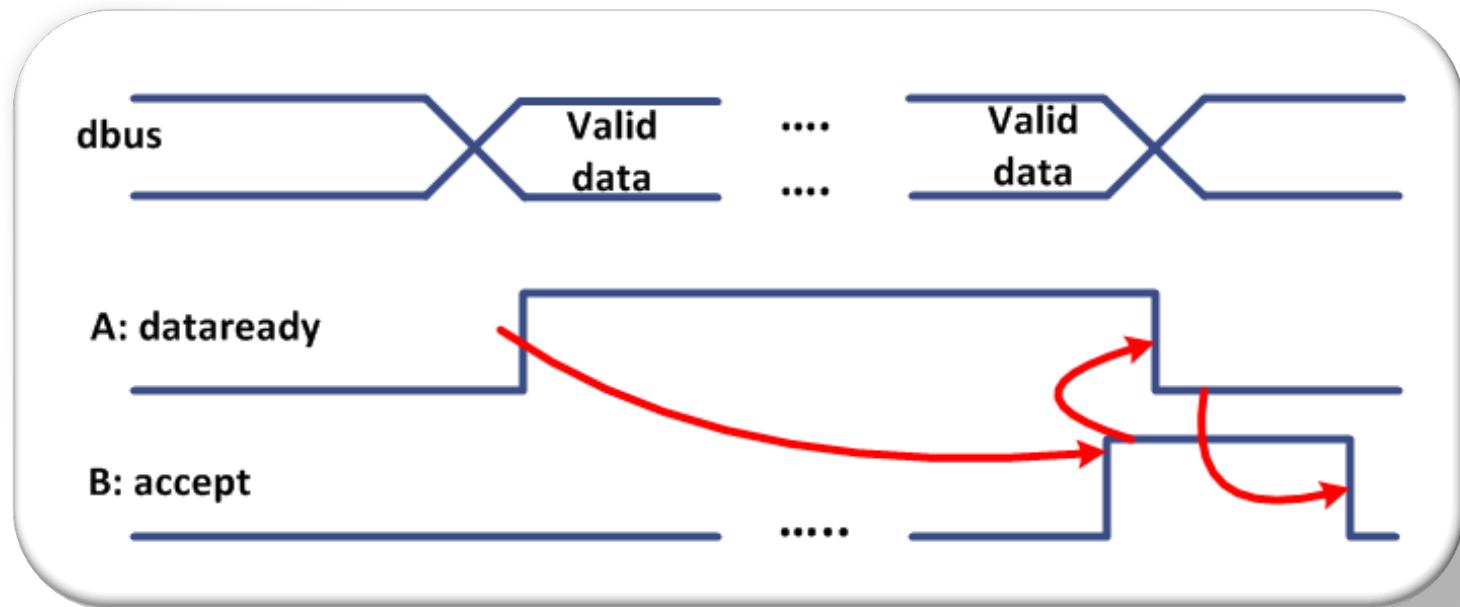
Handshaking Type One: Handshaking Between Two Systems



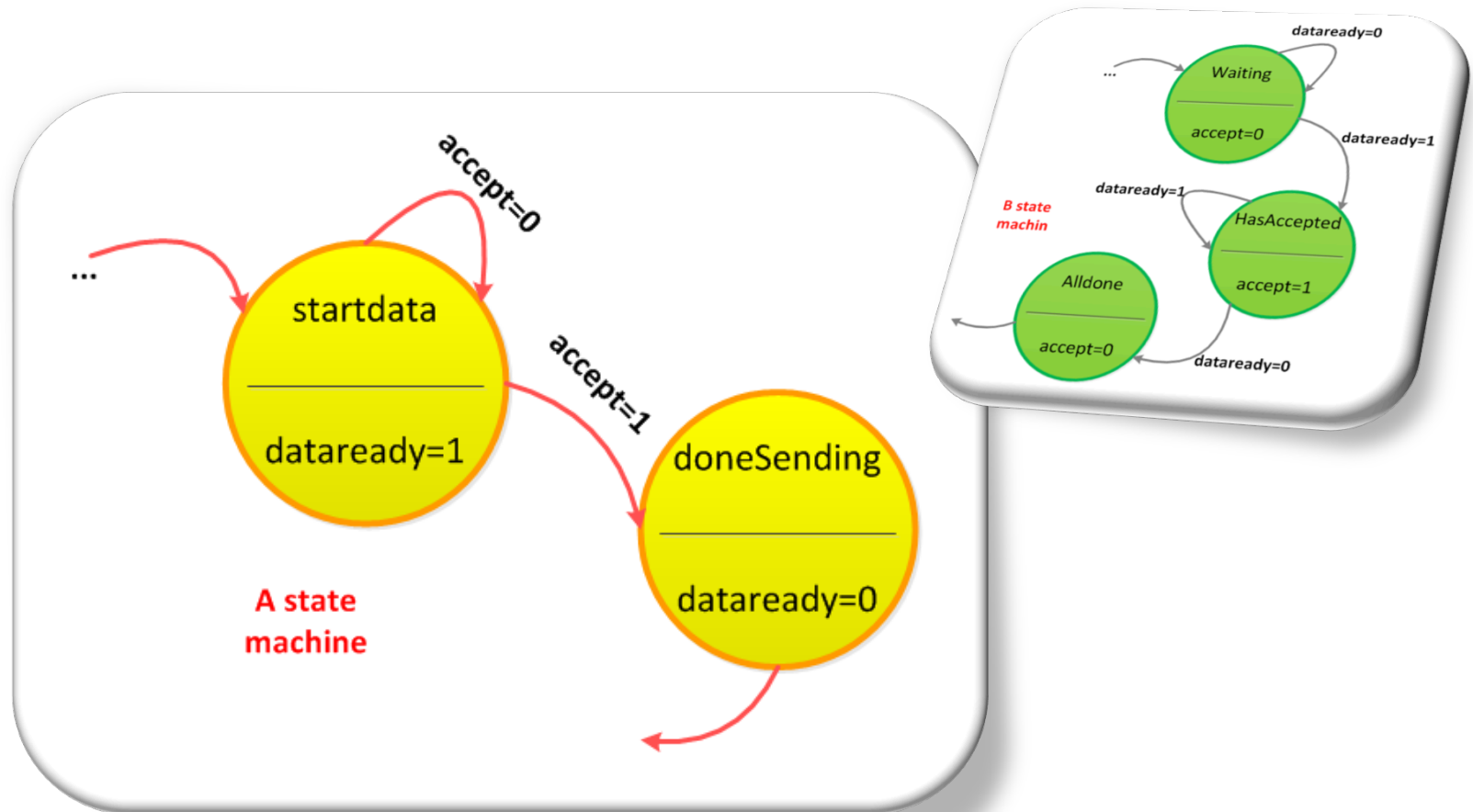
- Each system has its own clocking
- They have to have certain signals to talk

Handshaking Type One: Handshaking Between Two Systems

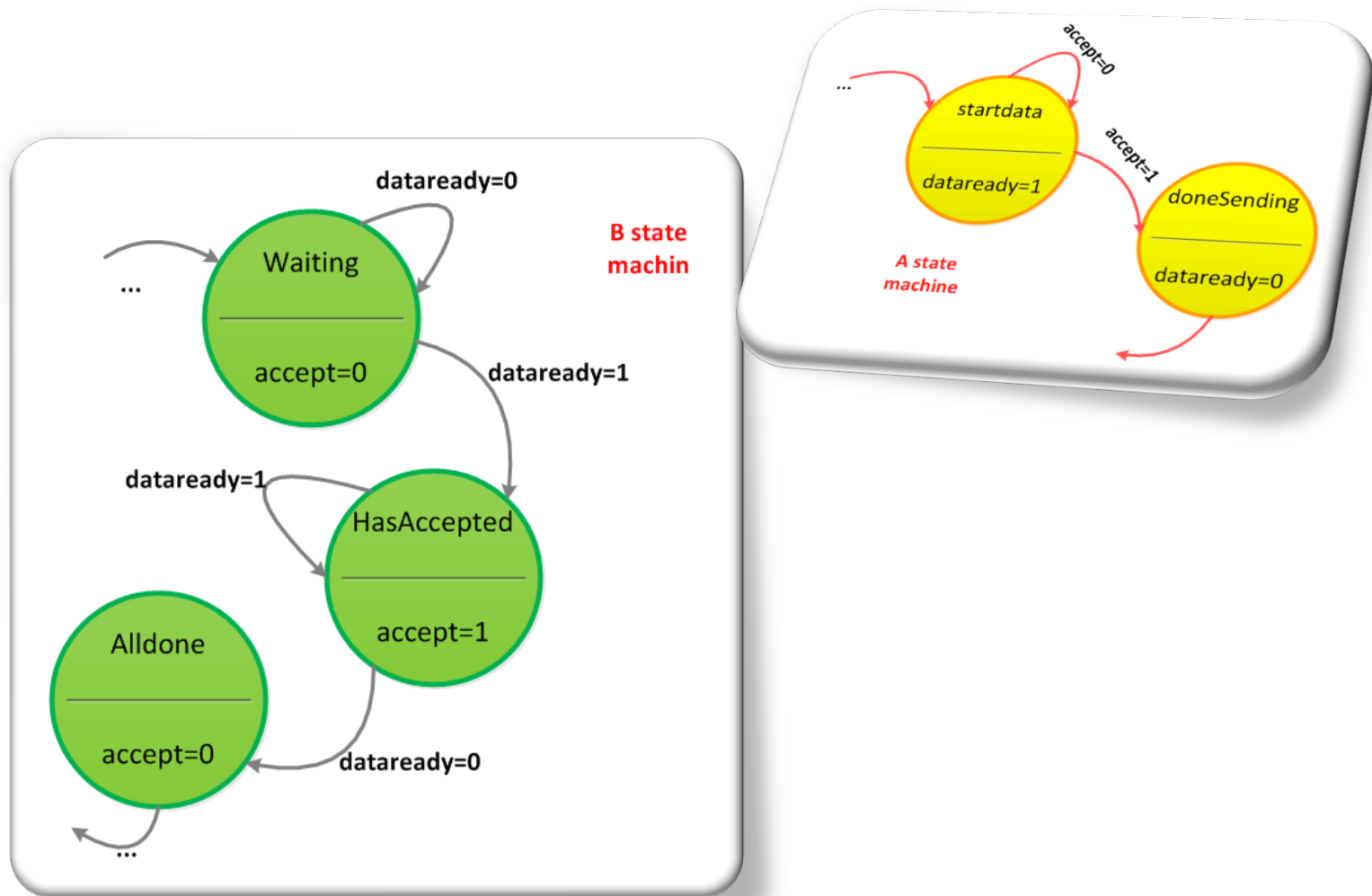
- Fully responsive handshaking



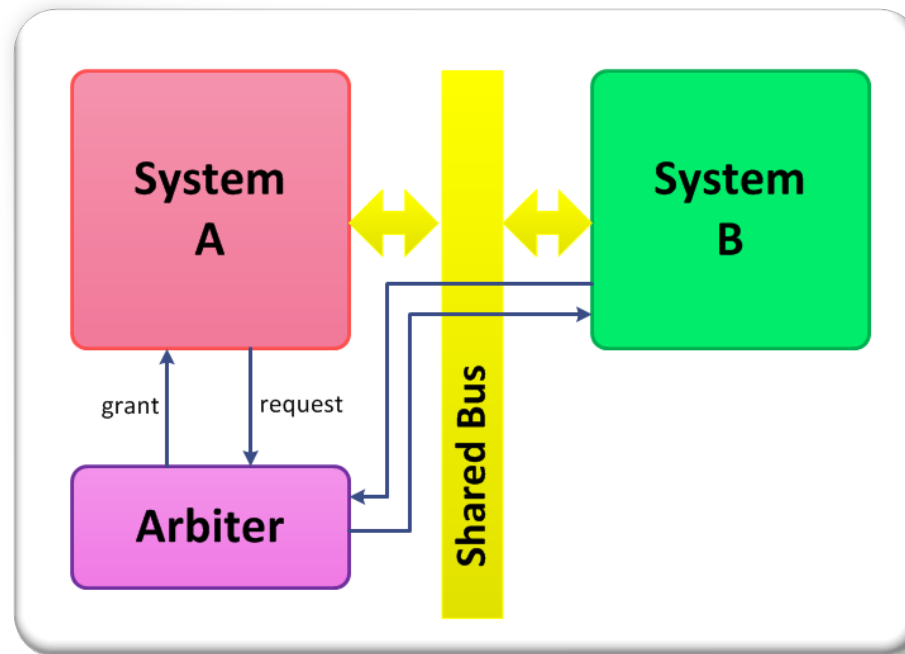
Handshaking Type One: Handshaking Between Two Systems



Handshaking Type One: Handshaking Between Two Systems

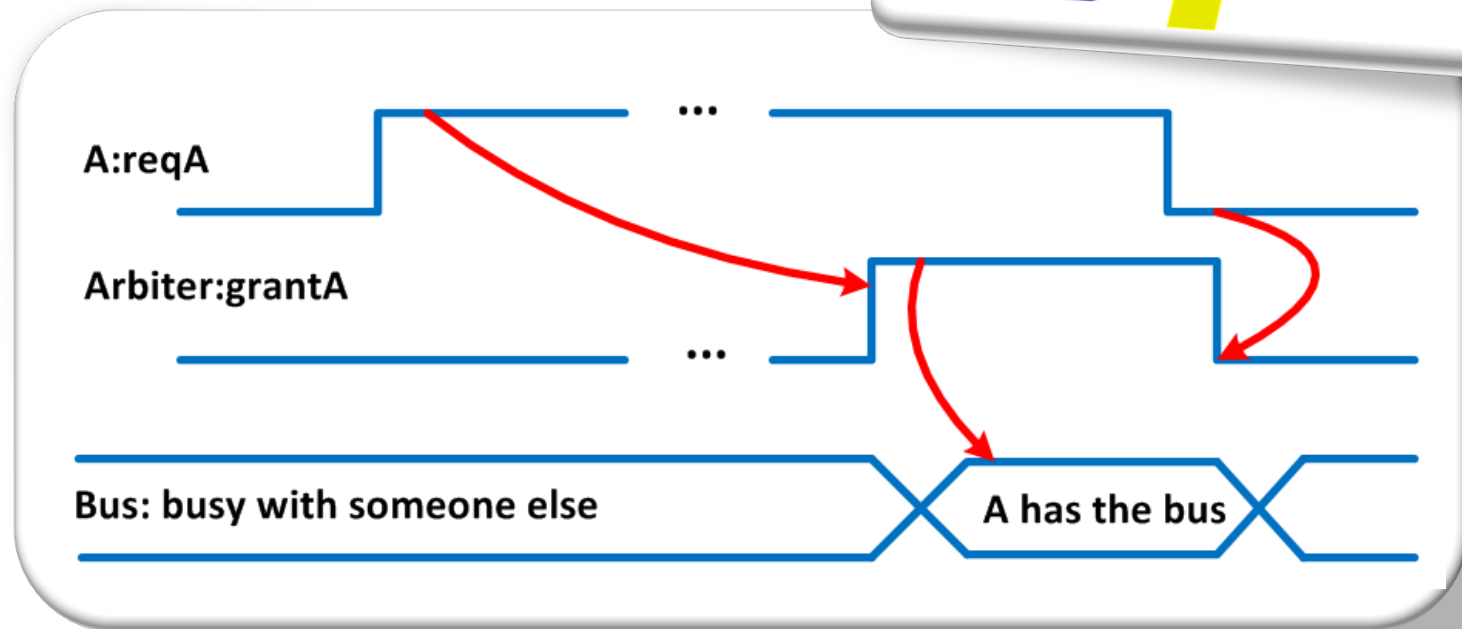
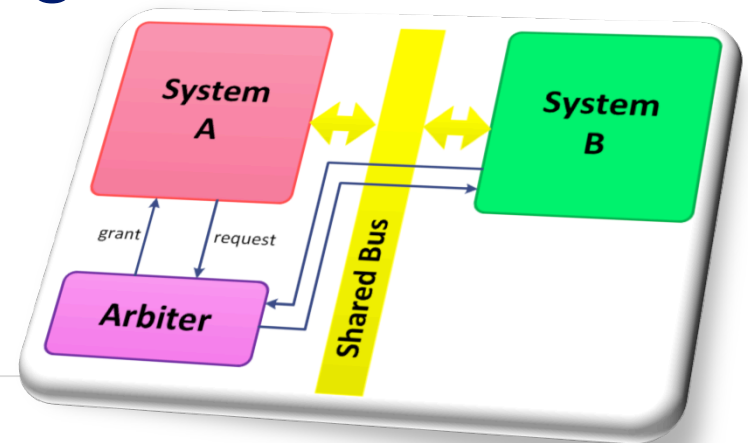


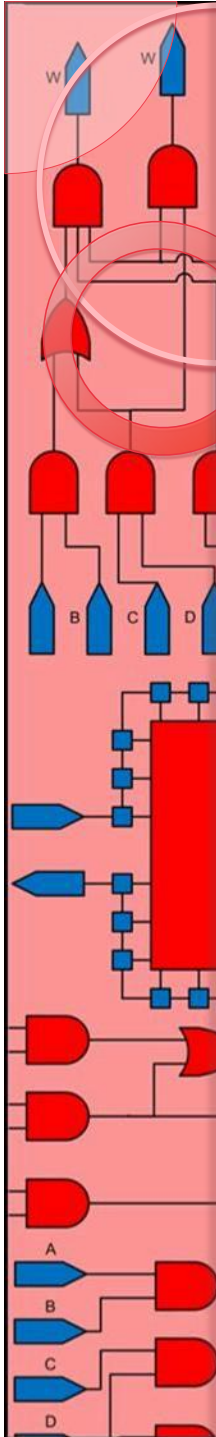
Handshaking Type Two: Handshaking for accessing a shared bus



- Using an arbiter to assure that none of the systems will simultaneously access the shared bus
- Each system has to have its own request and grant signals

Handshaking Type Two: Handshaking for accessing a shared bus

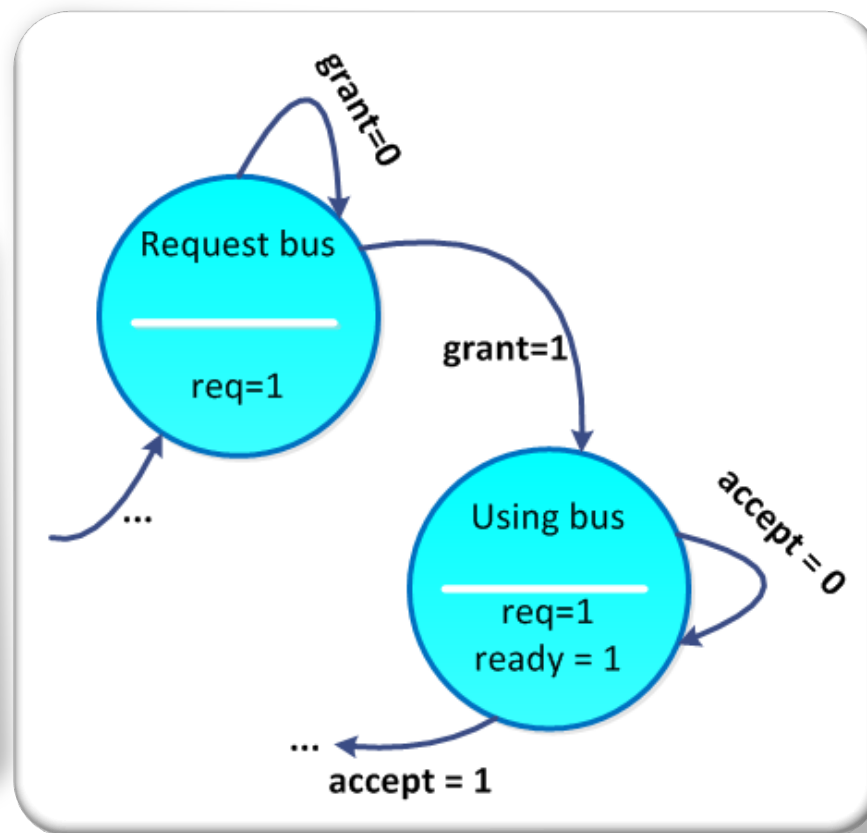
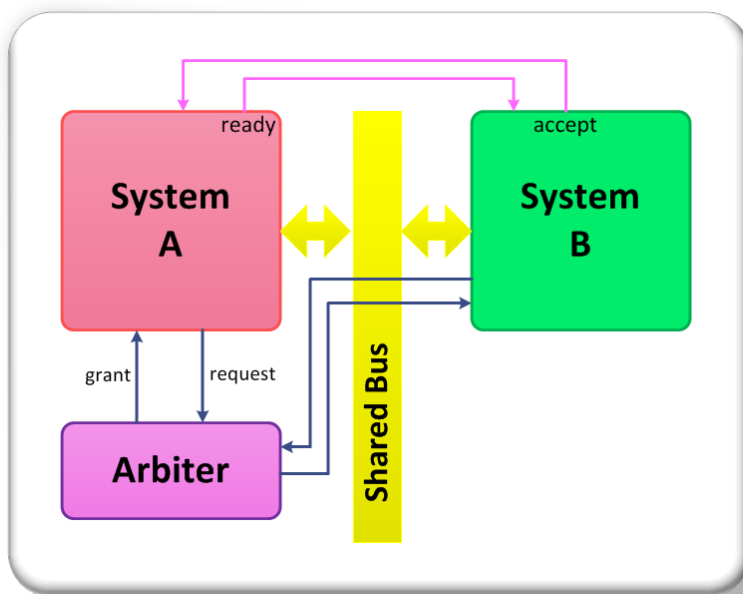




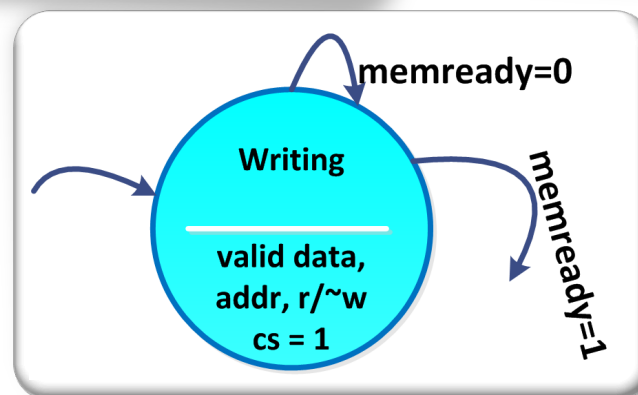
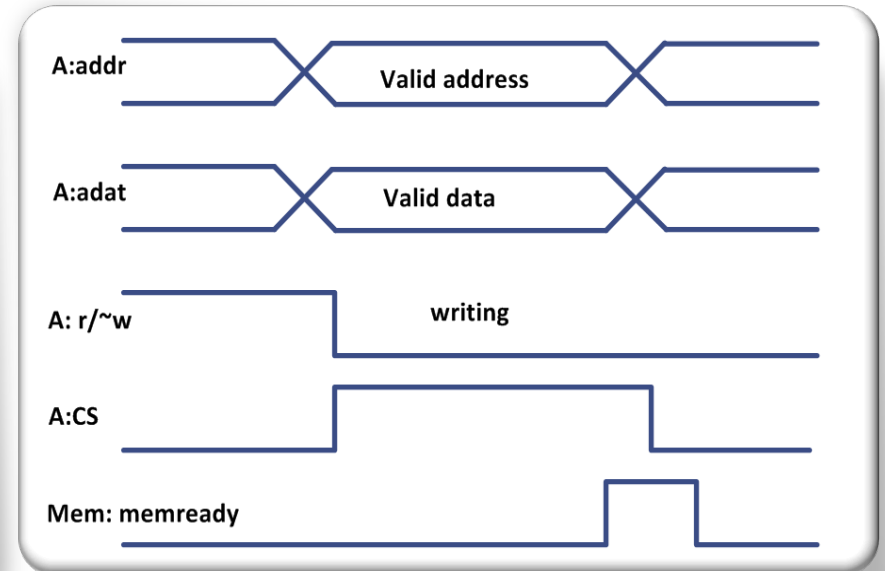
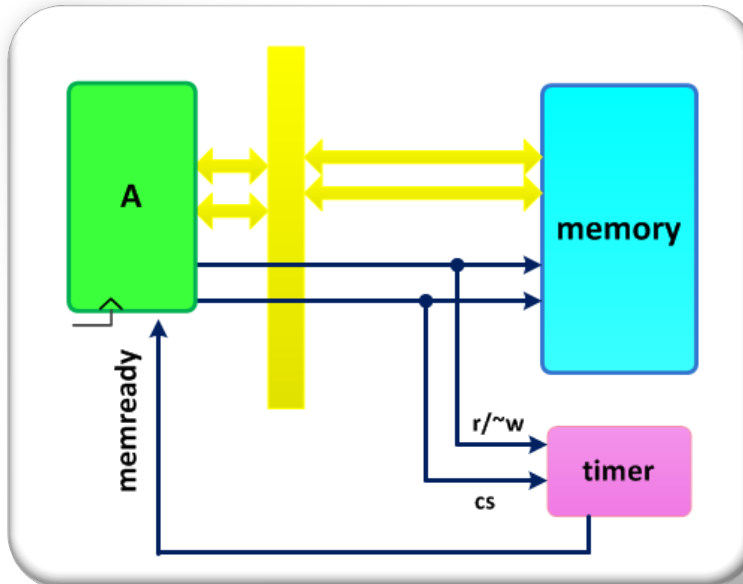
Two level handshaking

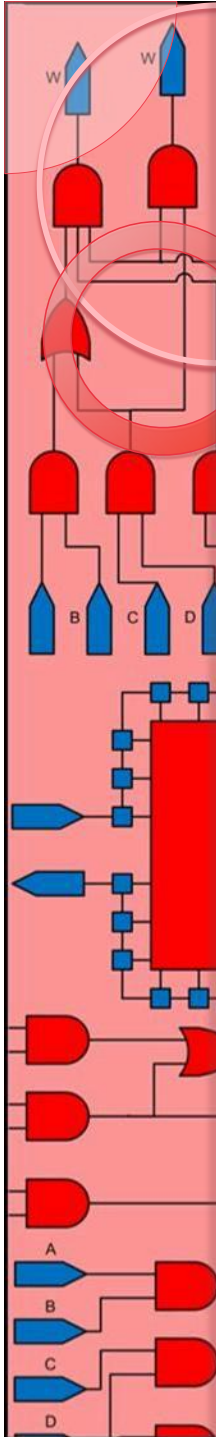
- Assume that system A wants to send some data to B through a shared bus
- At first A should talk to the arbiter and catches the bus by issuing a request .
- Once it puts the data on bus it informs system B by issuing ready
- After data is picked up by B, A removes its request.

Two level handshaking



Handshaking Type Three: memory handshaking

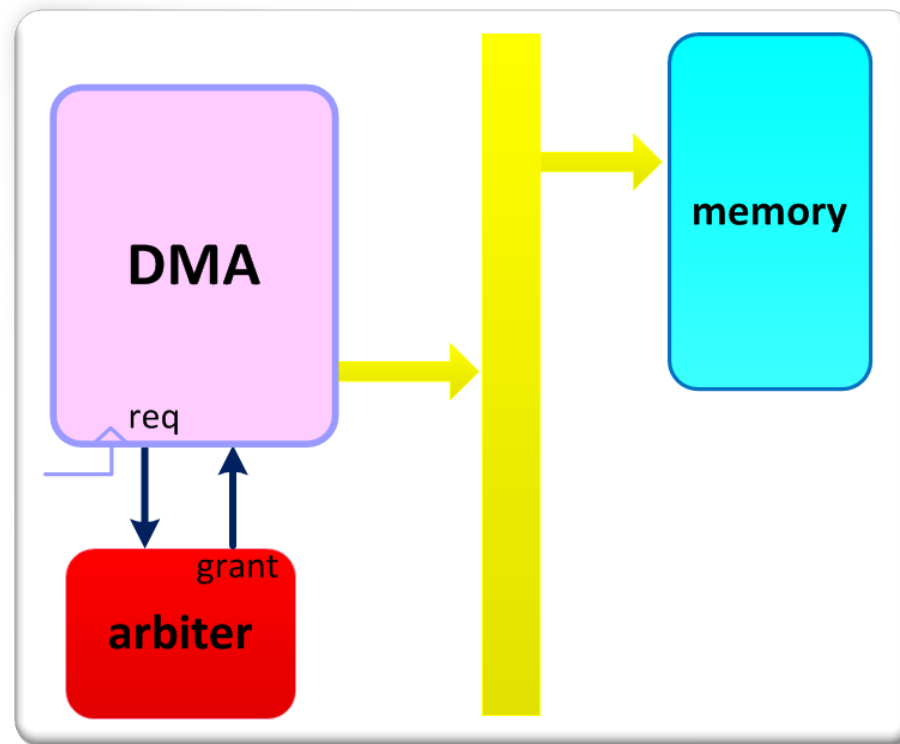


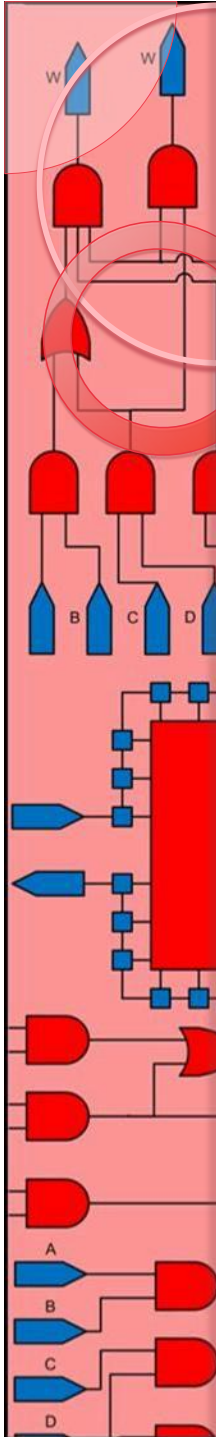


Combining Type Two and Three of Handshaking

- We can combine type two and three of handshaking
- At first A should deal with arbiter and gets the permission of using the bus
- Then it should send signals to the memory and waits for memready

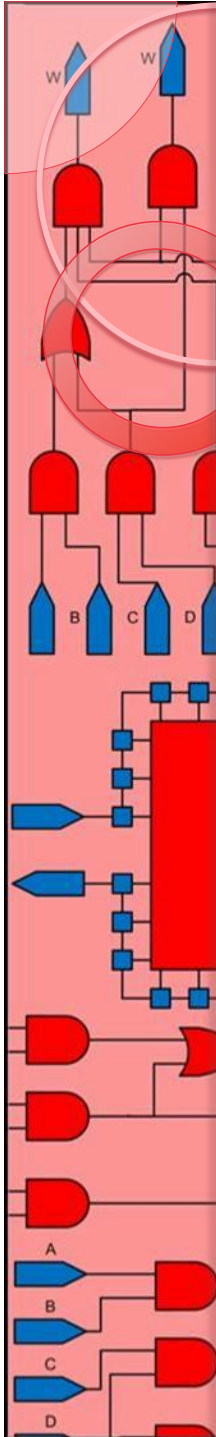
Handshaking Type Four: DMA Mode Or Burst Mode





Summary

- Handshaking definition
- Types of handshaking
 - Handshaking between two systems
 - Handshaking for accessing a shared bus
 - Memory handshaking
 - DMA mode or burst mode
- Handshaking with timing details
- Summary



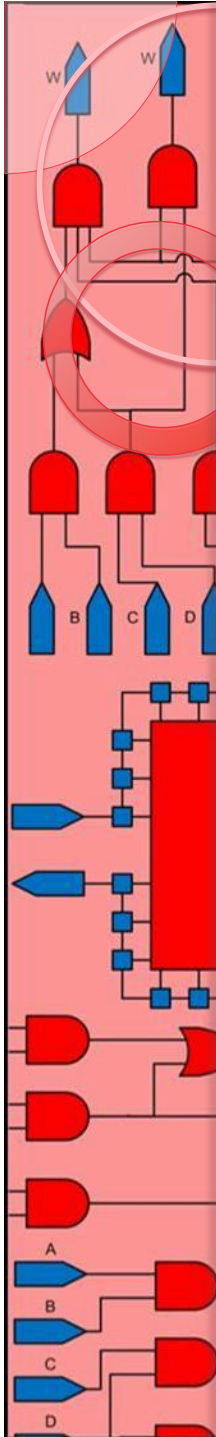
Lecture Series:

Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

Basics of Digital Design at RT Level with Verilog

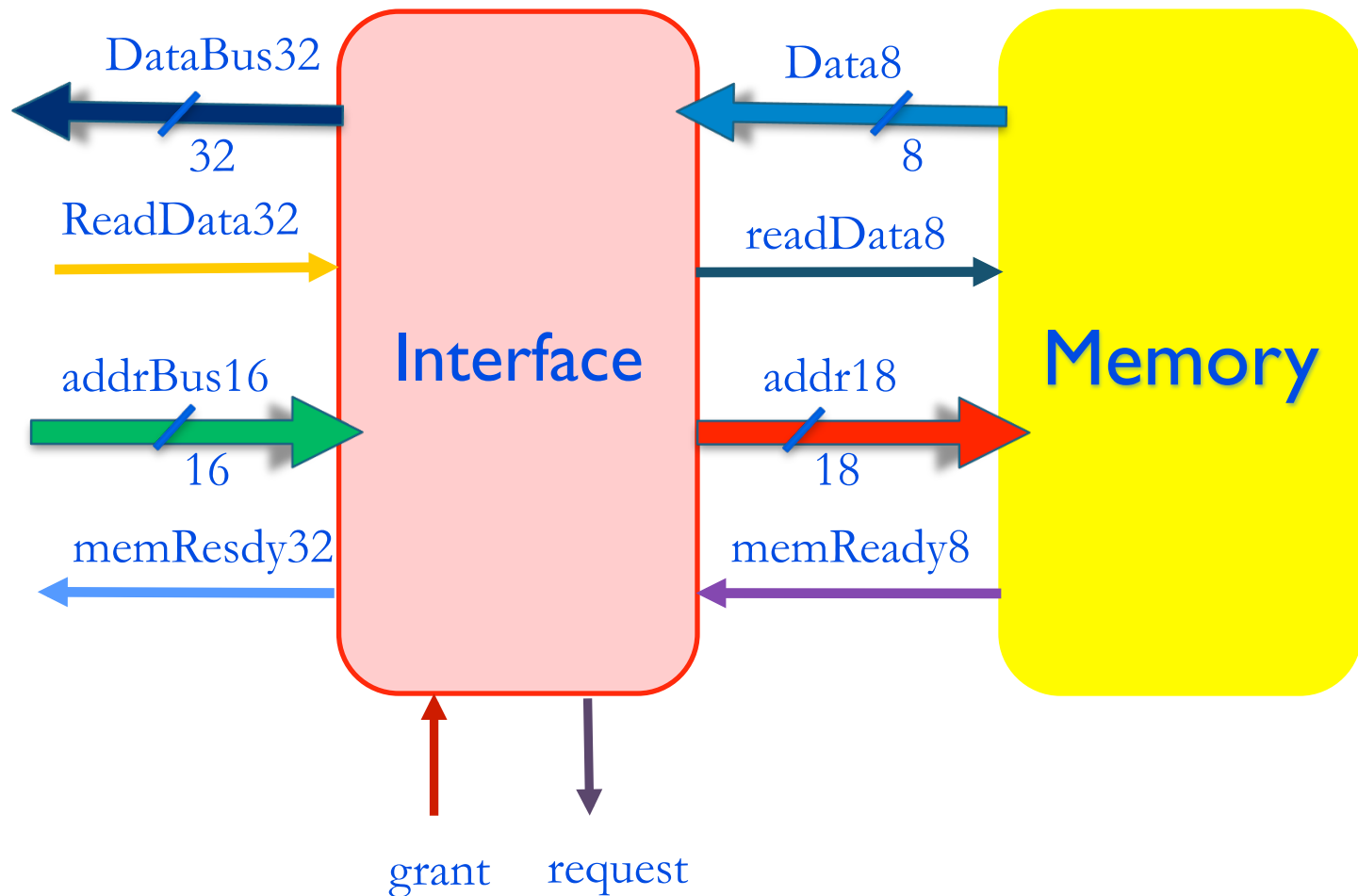
Lecture 20: Inter RTL Communications



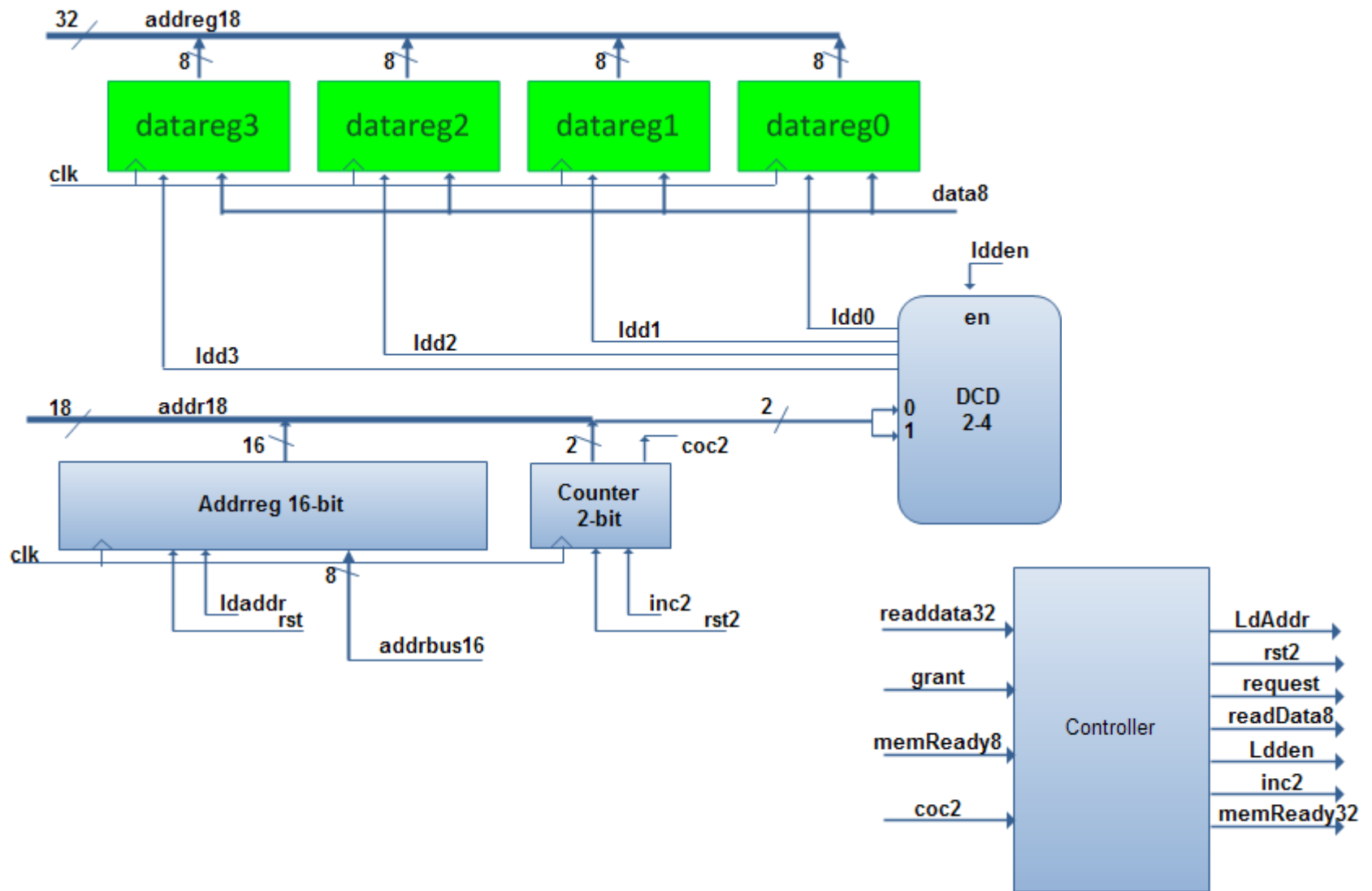
Inter RTL Communications

- RTL Handshaking
- Arbitration
- Inter-RTL device communications
- Interfacing
- Signals from the controller
- Datapath and Controller design
- Verilog descriptions
- Summary

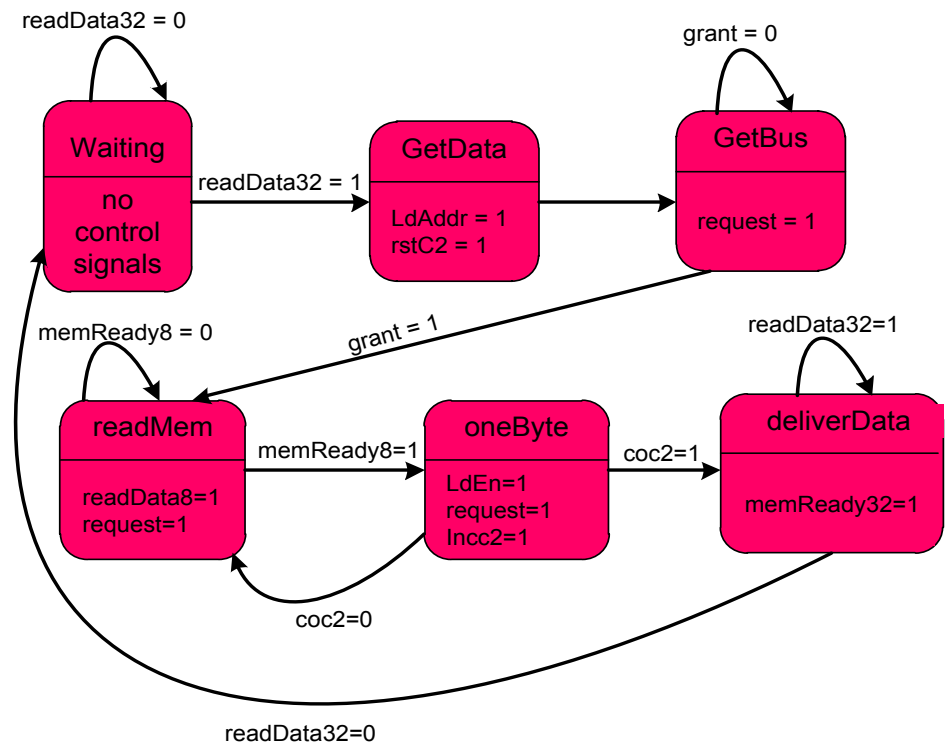
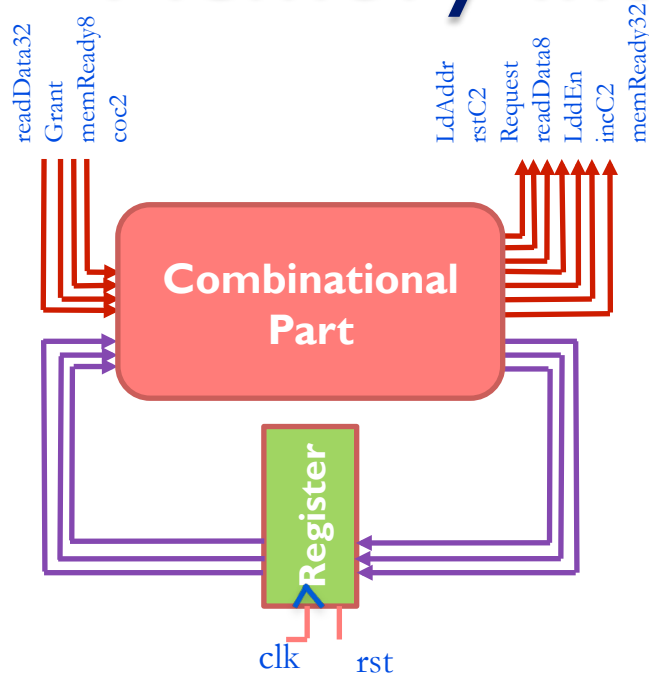
Memory Interface: Design Example

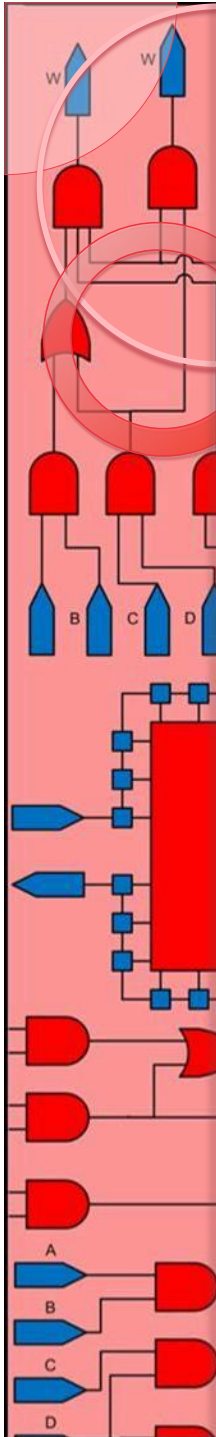


Memory Interface: Datapath&Controller Partitioning



Memory Interface: State Machine





Memory Interface: Datapath Verilog Code

```

module MI_datapath( input [15:0]addrbus16,input [7:0]data8, input clk, rst,
                   input ldaddr,incc2,rst2 ,ldden,
                   output [17:0]addr18,output [31:0]databus32,output reg coc2);

wire [17:0] addrreg18;
reg [15:0] addrreg16;
reg [1:0] counter2;
reg [3:0] ldd =0;
reg [7:0]datareg0,datareg1,datareg2,datareg3;

always @(posedge clk, posedge rst)begin // 16-bit address register
    if (rst)
        addrreg16 <=16'b0;
    if (ldaddr)
        addrreg16 <= addrbus16;
end

always @(posedge clk, posedge rst2)begin // 2-bit counter
    if (rst2)
        counter2 <=2'b0;
    else if(incc2)
        {coc2, counter2} <=counter2 +1;
end
end

```

Memory Interface: Datapath Verilog Code

```
//assign coc2={counter2,incc2};
assign addr18={addr16,counter2}; // 18-bit address to memory
assign addr18=addr18;

always @(ldden)begin // decoder
    ldd = 4'b0000;
    if(counter2==2'b00)
        ldd[0] = 1;
    else if(counter2==2'b01)
        ldd[1] = 1;
    else if(counter2==2'b10)
        ldd[2] = 1;
    else if(counter2==2'b11)
        ldd[3] = 1;
end
```

Memory Interface: Datapath Verilog Code

```
always @(posedge clk, posedge rst) begin
    if (rst)begin
        datareg0 <=8'b0;
        datareg1 <=8'b0;
        datareg2 <=8'b0;
        datareg3 <=8'b0;
    end
    else if(1dd[0])
        datareg0 <= data8;
    else if(1dd[1])
        datareg1 <= data8;
    else if(1dd[2])
        datareg2 <= data8;
    else if(1dd[3])
        datareg3 <= data8;
    end
    assign databus32={datareg3,datareg2,datareg1,datareg0};
endmodule
```

3

Memory Interface: Controller Verilog Code

```
module MI_cu(input clk,rst,readdata32,grant,memready8,coc2,
             output reg ldaddr, rst2, request, readdata8,ldden,
             incc2,memready32);

    reg [2:0] ns,ps;

    always @(ps,readdata32,grant,memready8,coc2)begin:comb
        ldaddr=1'b0;rst2=1'b0;request=1'b0;readdata8=1'b0;
        ldden=1'b0;incc2=1'b0;memready32=1'b0;
        case(ps)
            3'b000:
                if(readdata32)
                    ns=1;
                else
                    ns=0;
            3'b001:begin
                ns=2;ldaddr=1'b1;rst2=1'b1;
            end
            3'b010:begin
                if(grant)
                    ns=3;
                else
                    ns=2;request=1'b1;
            end
        end
    end
```

Memory Interface: Controller Verilog Code

```
3'b011:begin
    if(memready8)
        ns=4;
    else ns=3;
        request=1'b1;
    end
3'b100:begin
    if(coc2)
        ns=5;
    else
        ns=3;ldden=1'b1;request=1'b1;incc2=1'b1;
    end
3'b101:begin
    if(readdata32)
        ns=5;
    else
        ns=0;memready32=1'b1;
    end
default:begin
    ns=0;ldaddr=1'b0;rst2=1'b0;
    request=1'b0;readdata8=1'b0;
    ldden=1'b0;incc2=1'b0;memready32=1'b0;
end
endcase
end
```

```
module
...
req
...
alt
```



Memory Interface: Controller Verilog Code

```
always@(posedge clk,posedge rst)begin:seq
    if(rst)
        ps<=3'b0;
    else
        ps<=ns;
    end
endmodule
```

Memory Interface: Complete System Verilog Code

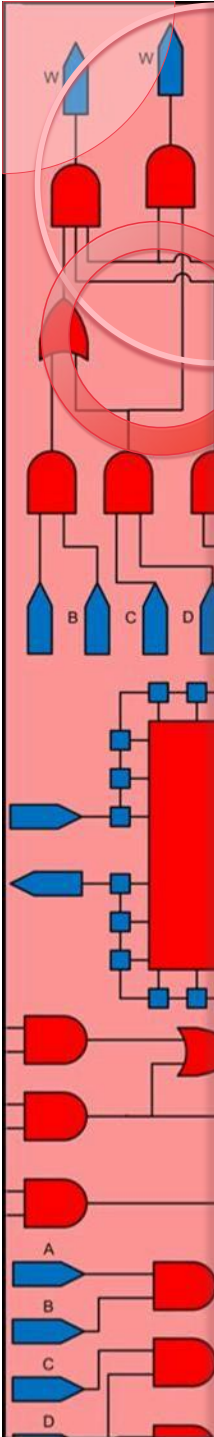
```
module MI_system(
    input readdata32, grant, memready8, clk, rst,
    input [15:0] addrbus16,
    input [7:0] data8, output [31:0] databus32,
    output [17:0] addr18,
    output memready32, request, readdata8);

    wire coc2;
    wire incc2, ldaddr, ldden, rst2;

    MI_cu U1 ( clk, rst, readdata32, grant, memready8, coc2, ldaddr,
               rst2, request, readdata8, ldden, incc2, memready32);

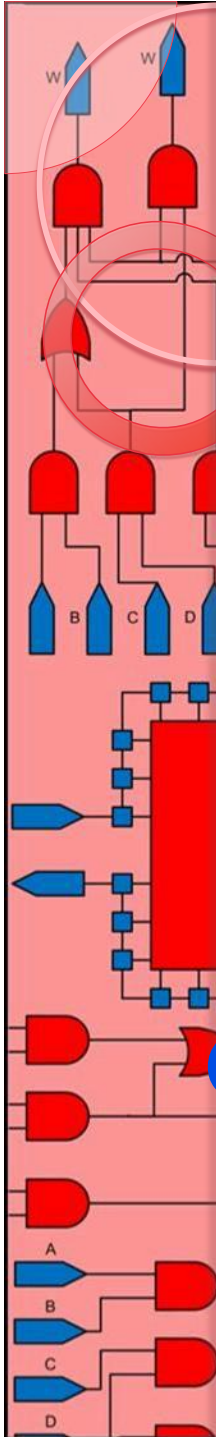
    MI_datapath U2 (addrbus16, data8, clk, rst, ldaddr, incc2, rst2,
                   ldden, addr18, databus32, coc2);

endmodule
```

Inter RTL Communications

- RTL Handshaking
- Arbitration
- Inter-RTL device communications
- Interfacing
- Signals from the controller
- Datapath and Controller design
- Verilog descriptions
- Summary



Lecture Series:

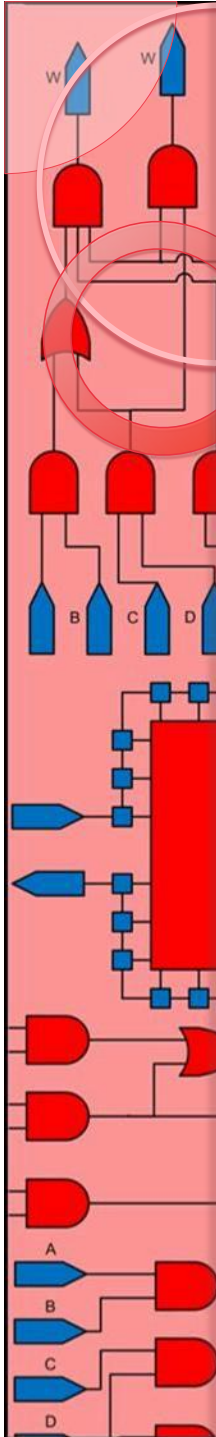
Basics of Digital Design at RT Level with Verilog

Z. Navabi, University of Tehran

Basics of Digital Design at RT Level with Verilog

Lecture 21:

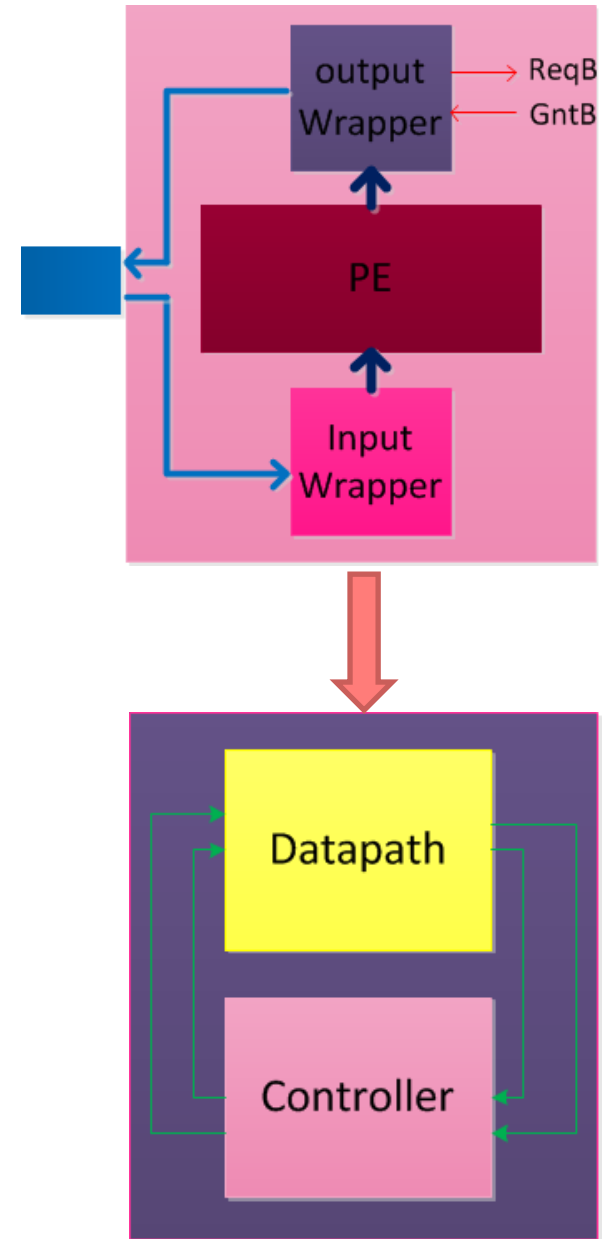
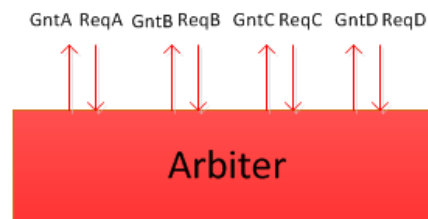
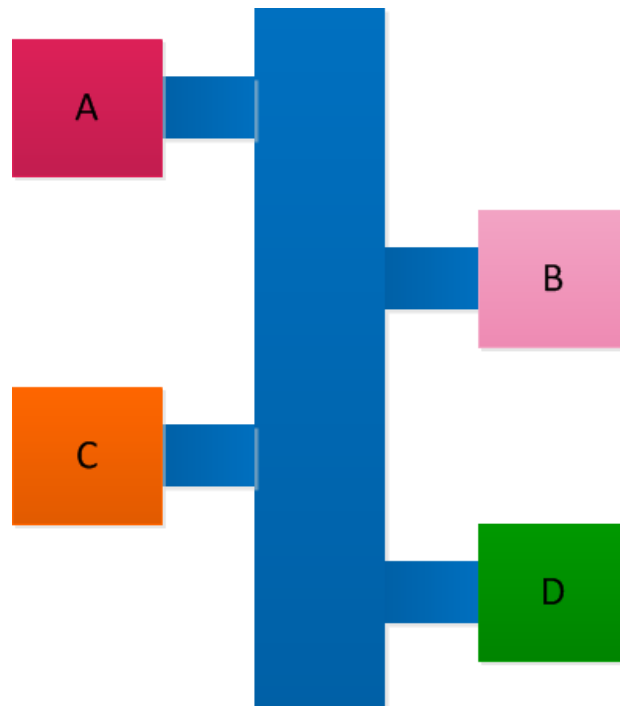
Complete RTL: Processing Elements and Communications



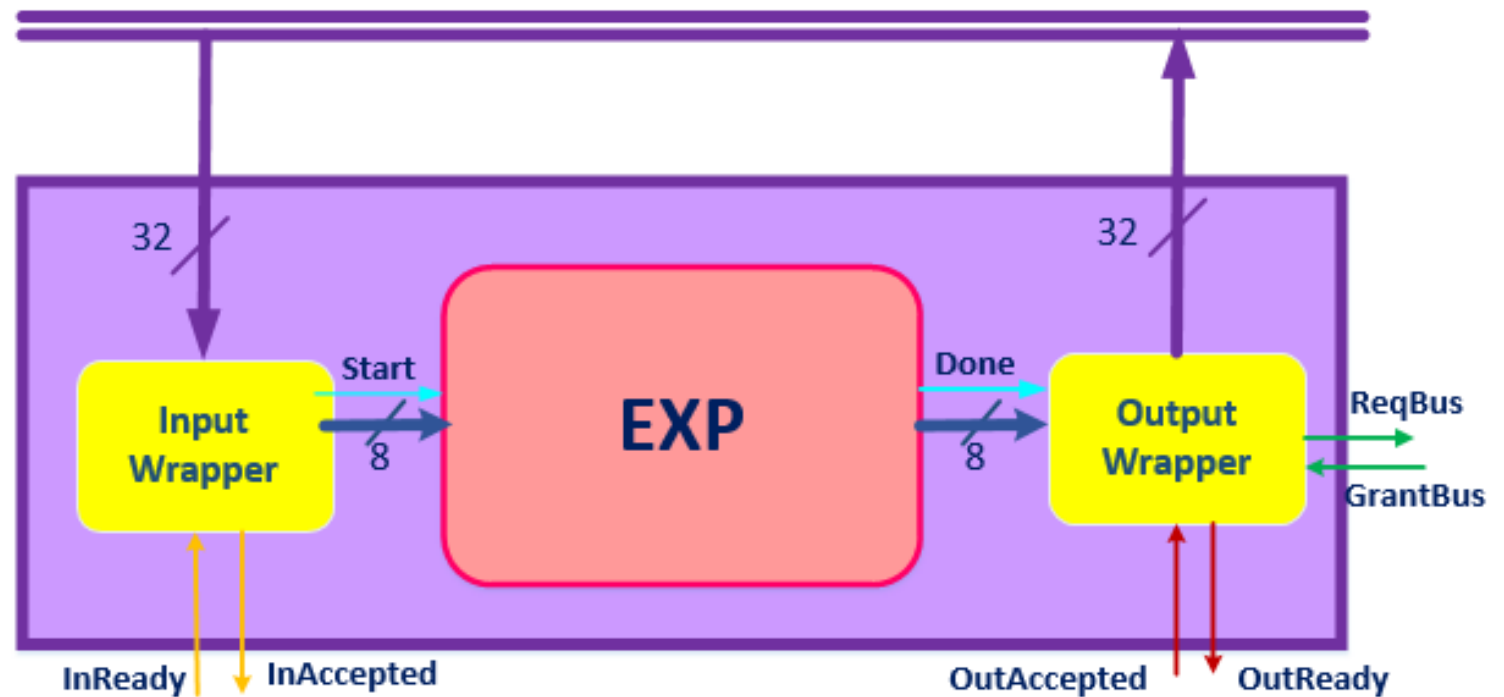
RTL Processing Elements and Communications

- Overview
- PE and IO wrappers
- A PE example (includes Verilog)
 - Datapath controller partitioning
 - Datapath design
 - Controller design
- IO wrappers
 - Datapath controller partitioning
 - Datapath design
 - Controller design
- Summary

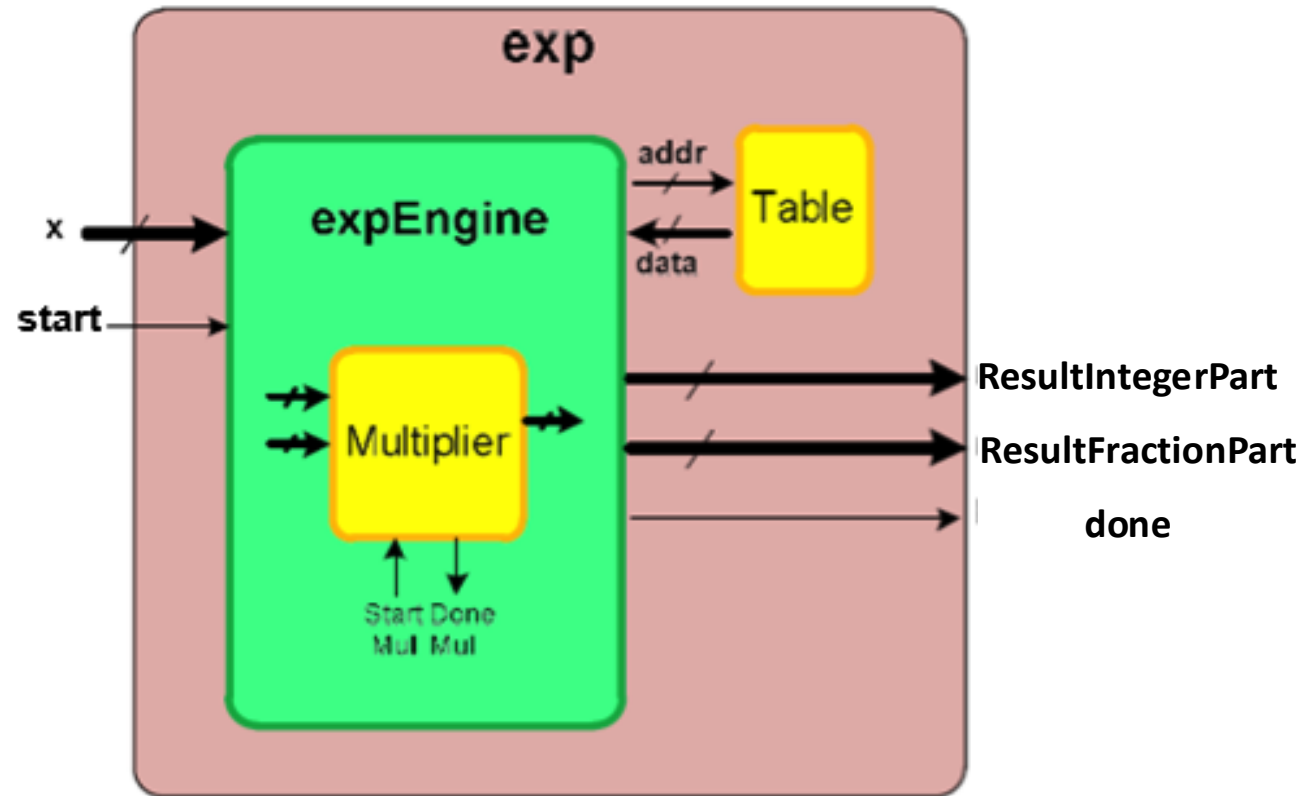
Bus Based Design



Exponentiation Module: Design Example



Exponentiation Module



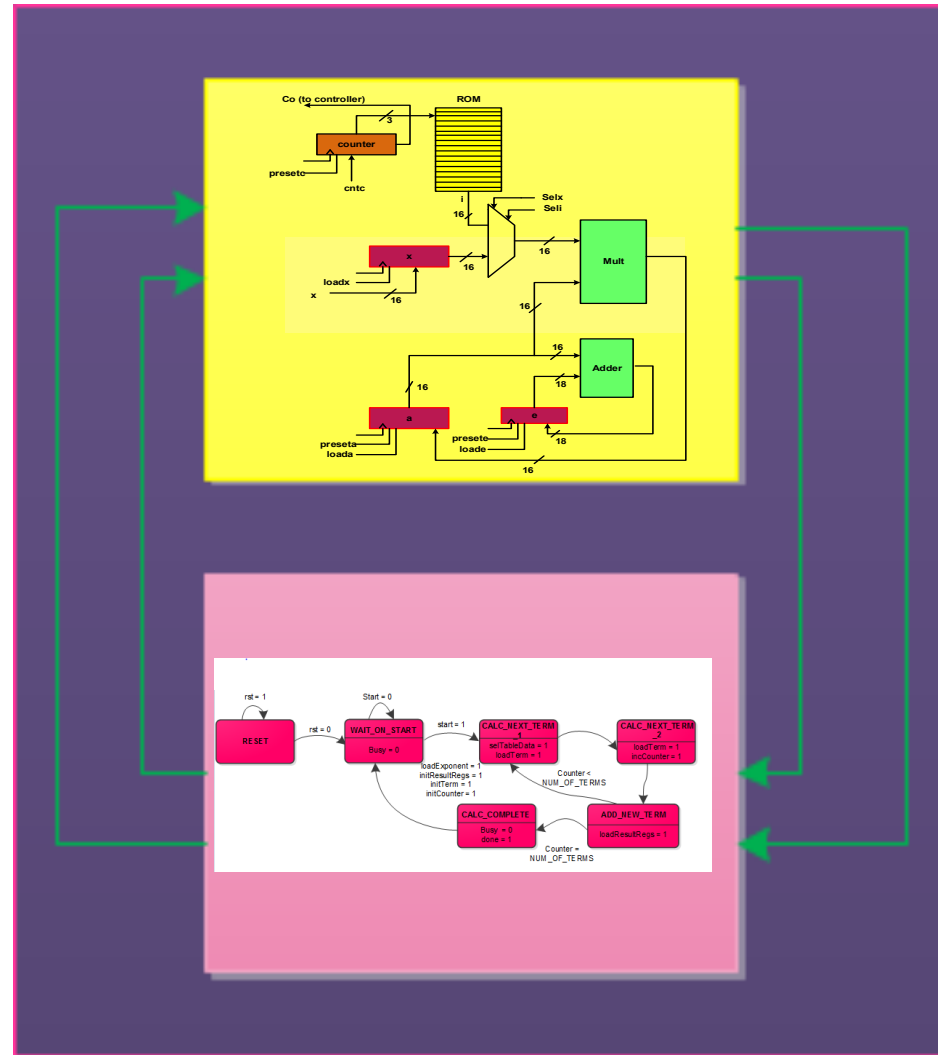
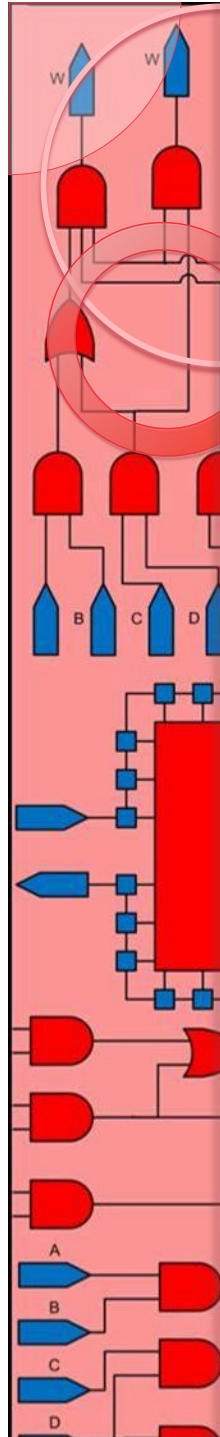


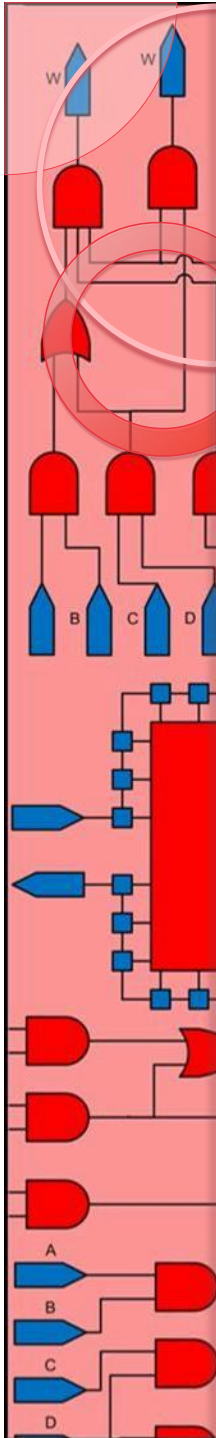
Exponential Function Algorithm

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

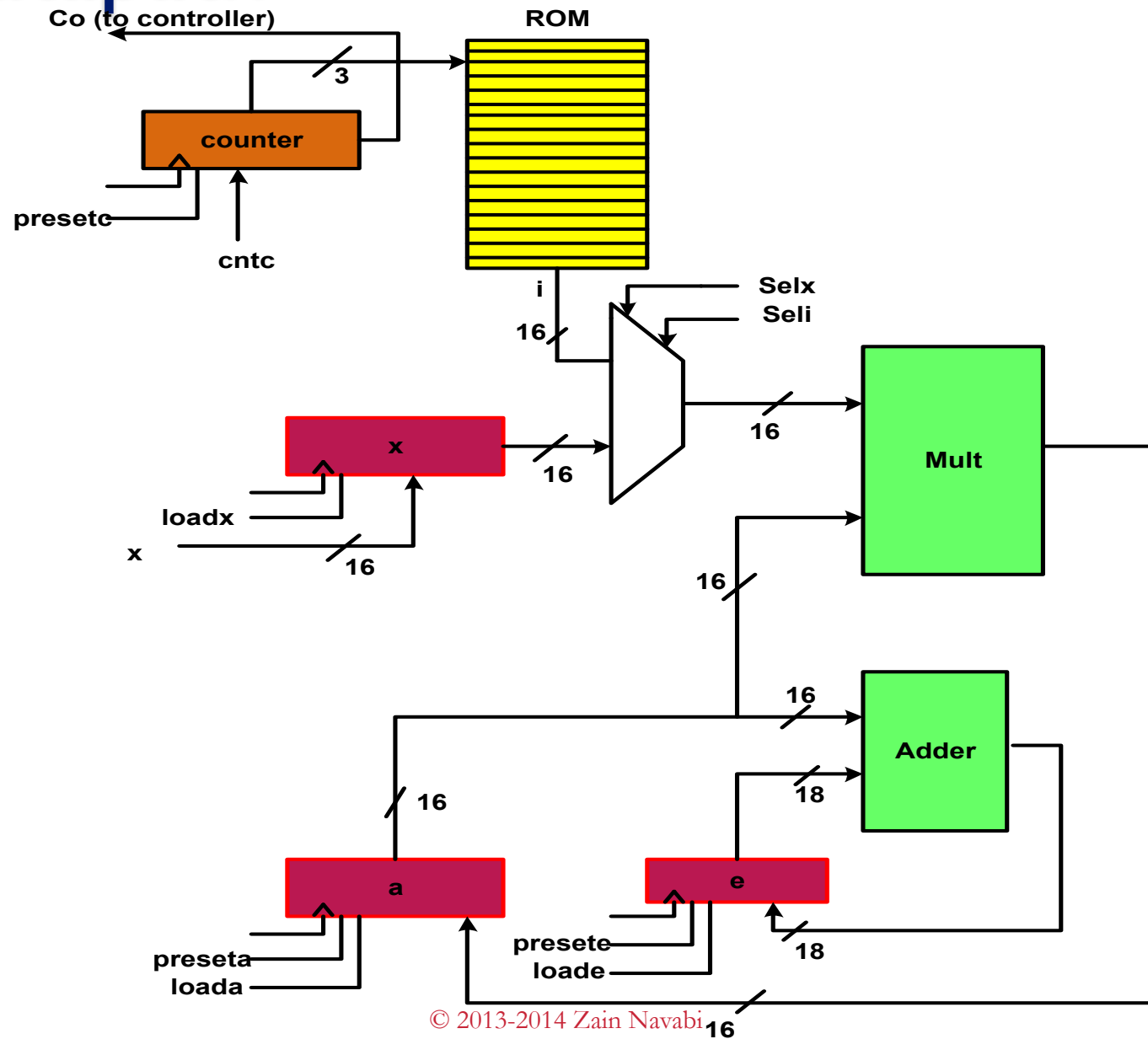
```
e = 1;  
a = 1;  
for( i = 1; i < n; i++ ) {  
    a = a * x * ( 1 / i );  
    e = e + a;  
}
```

Datapath / Controller

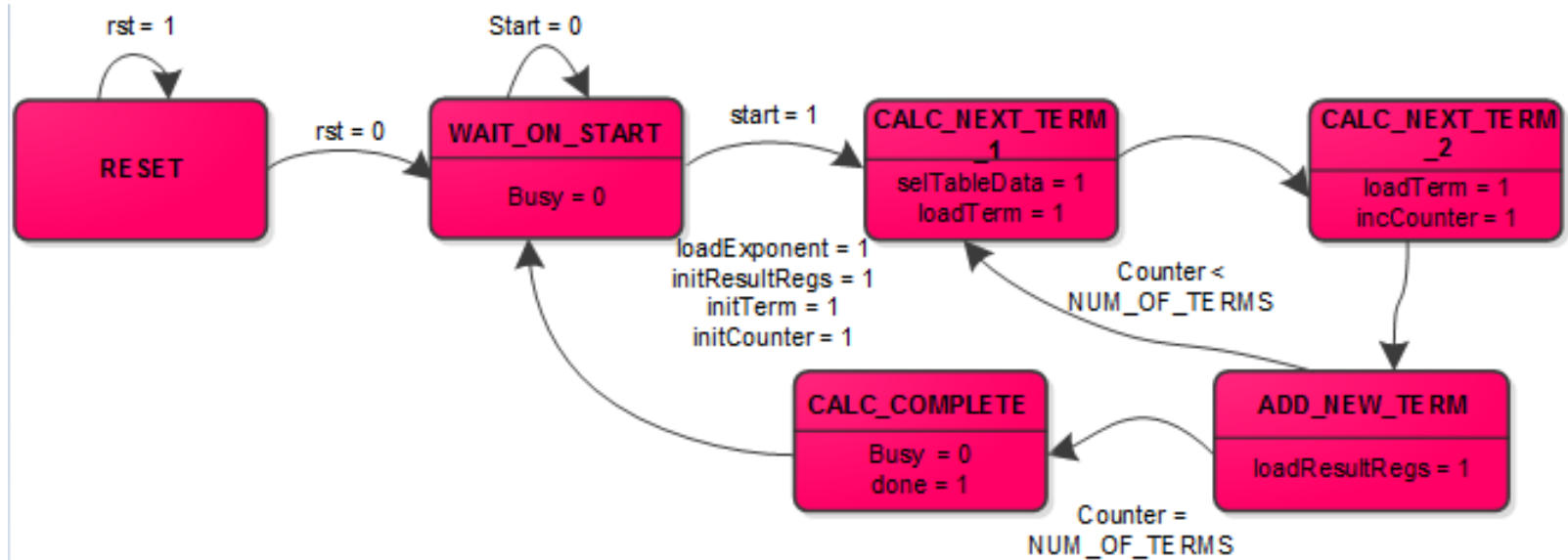




Datapath

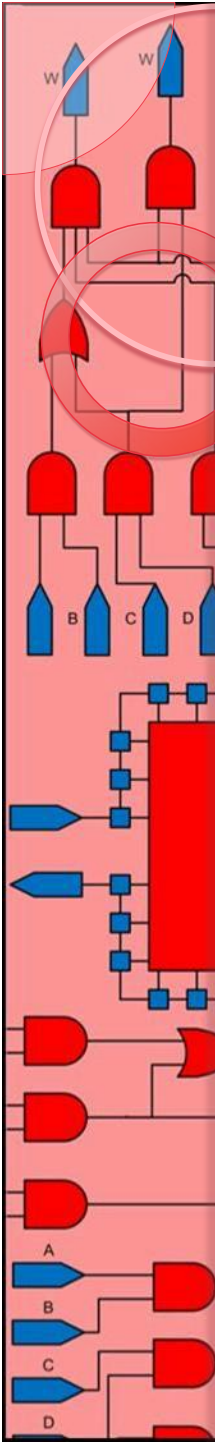


Controller



Exponentiation Module :

Datapath Verilog Code



```
`define RESET          0
`define WAIT_ON_START  1
`define CALC_NEXT_TERM_1  2
`define CALC_NEXT_TERM_2  3
`define ADD_NEW_TERM    4
`define CALC_COMPLETE   5

module expEngine      ( clk, rst, start, x, busy, done, addr,
                      tableData, resultIPart, resultFPart );

    parameter F_WIDTH = 16;
    parameter NUM_OF_TERMS = 8;
    parameter CNT_WIDTH = 4;
    input  clk, rst, start;
    input  [F_WIDTH - 1:0] x;
    output busy, done;
    output [CNT_WIDTH - 1:0] addr;
    input  [F_WIDTH - 1:0] tableData;
    output [1:0] resultIPart;
    output [F_WIDTH - 1:0] resultFPart;
    reg [F_WIDTH - 1:0] exponent;
    reg [F_WIDTH - 1:0] term;
    reg [1:0] resultIPart;
    reg [F_WIDTH - 1:0] resultFPart;
    wire [F_WIDTH * 2 - 1:0] multResult;
    reg [F_WIDTH - 1:0] multRInput;
    wire [F_WIDTH + 1:0] addResult;
    reg [CNT_WIDTH - 1:0] counter;
```

Exponentiation Module

Datapath Verilog Code

```
reg [2:0] pState, nState;
//=====
//Datapath
//=====

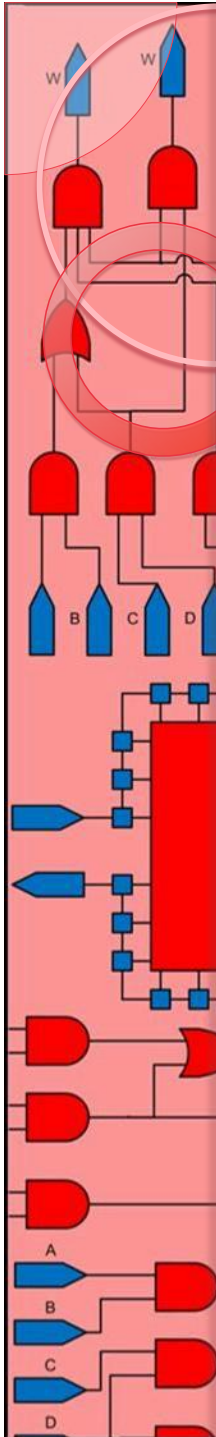
//Exponent register
always @(posedge clk ) begin
    if( rstExponent )
        exponent <= 0;
    else if( loadExponent )
        exponent <= x;
end

//Term register: register storing one of terms in Taylor series
always @( posedge clk ) begin
    if( rstTerm )
        term <= 0;
    else if( initTerm )
        term <= 32'hFFFFFFF;
    else if( loadTerm )
        term <= multResult [F_WIDTH * 2 - 1: F_WIDTH ];
end

//Multiplexer on the right input of the multiplier
always @( selTableData or tableData or exponent ) begin
    if( selTableData )
        multRInput <= tableData;
    else
        multRInput <= exponent;
end

//Multiplier
assign multResult = term * multRInput ;
```

2



Exponentiation Module

Datapath Verilog Code

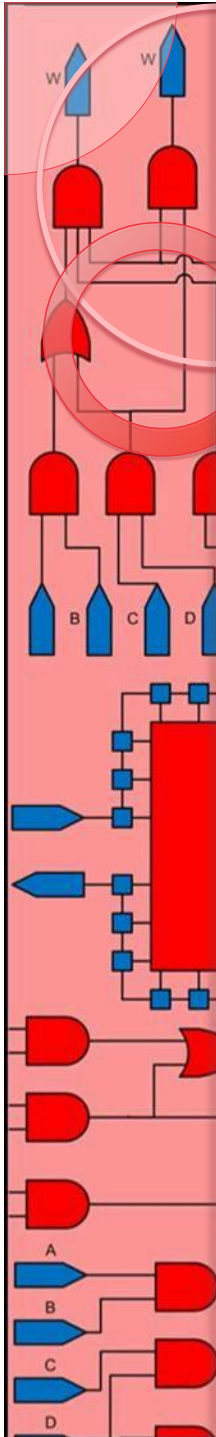
```

assign addResult = term + { resultIPart, resultFPart };
//Registers storing integral part and fractional parts of the result
always @(posedge clk ) begin
    if( rstResultRegs )
        { resultIPart, resultFPart } <= 0;
    else if( initResultRegs ) begin
        resultIPart <= 1;
        resultFPart <= 0;
    end
    else if( loadResultRegs )
        { resultIPart, resultFPart } <= addResult;
end

//Counter
always @(posedge clk ) begin
    if( rstCounter )
        counter <= 0;
    else if ( initCounter )
        counter <= 1;
    else if( incCounter )
        counter <= counter + 1;
end

assign addr = counter;

```



Exponentiation Module :

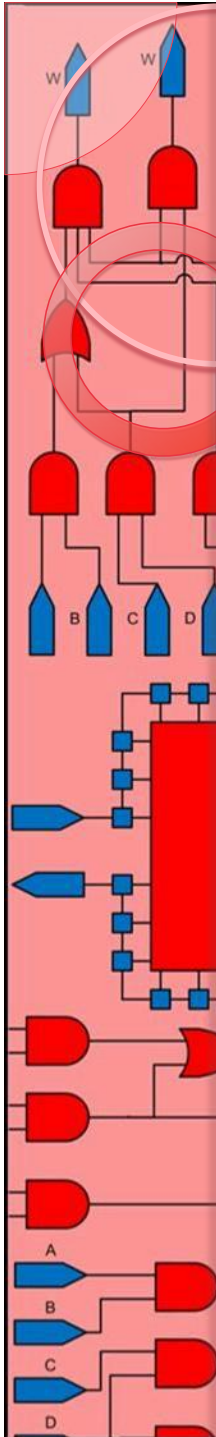
Combinational Table Verilog Code

```

module Table ( addr, tableData );
    parameter ADDR_WIDTH = 3;
    parameter F_WIDTH = 16;
    input [ADDR_WIDTH - 1:0] addr;
    output [F_WIDTH - 1: 0 ] tableData;
    reg [31:0]temp;
    always @( addr ) begin
        temp <= 0;
        case ( addr )
            0: temp <= 32'h00000000;
            1: temp <= 32'hFFFFFFF; //Fixed point representation of 1
            2: temp <= 32'h80000000; //Fixed point representation of 1/2
            3: temp <= 32'h55555555; //Fixed point representation of 1/3
            4: temp <= 32'h40000000; //Fixed point representation of 1/4
            5: temp <= 32'h33333333; //Fixed point representation of 1/5
            6: temp <= 32'h2AAAAAAA; //Fixed point representation of 1/6
            7: temp <= 32'h24924924; //Fixed point representation of 1/7
            8: temp <= 32'h20000000; //Fixed point representation of 1/8
            9: temp <= 32'h1C71C71C; //Fixed point representation of 1/9
            10: temp <= 32'h19999999; //Fixed point representation of 1/10
            11: temp <= 32'h1745D174; //Fixed point representation of 1/11
            12: temp <= 32'h15555555; //Fixed point representation of 1/12
            13: temp <= 32'h13B13B13; //Fixed point representation of 1/13
            14: temp <= 32'h12492492 ; //Fixed point representation of 1/14
            15: temp <= 32'h11111111; //Fixed point representation of 1/15

        endcase
    end
    assign tableData = temp[31: 31 - F_WIDTH + 1 ];
endmodule

```



Exponentiation Module :

Controller Verilog Code

```
//=====
//Controller
//=====

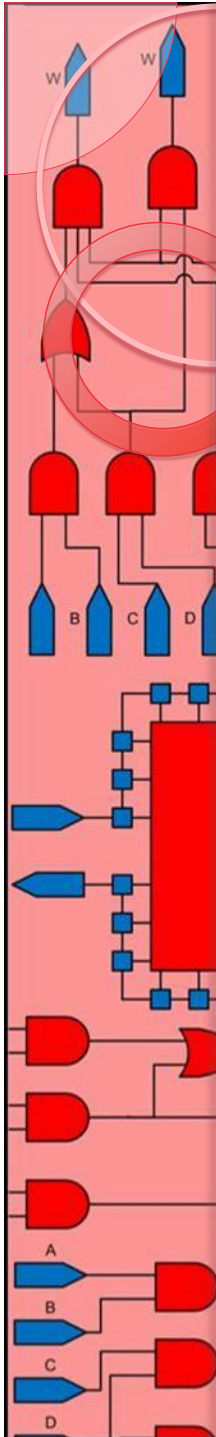
always @( pState or rst or start or counter ) begin
    nState <= `RESET;
    loadExponent <= 1'b0;
    rstExponent <= 1'b0;
    loadTerm <= 1'b0;
    rstTerm <= 1'b0;
    initTerm <= 1'b0;
    selTableData <= 1'b0;
    rstResultRegs <= 1'b0;
    initResultRegs <= 1'b0;
    loadResultRegs <= 1'b0;
    initResultRegs <= 1'b0;
    incCounter <= 1'b0;
    rstCounter <= 1'b0;
    initCounter <= 1'b0;
    busy <= 1'b1;
    done <= 1'b0;
end
```


Exponentiation Module :

Controller Verilog Code

```
case ( pState )
  `RESET : begin
    rstExponent <= 1'b1;
    rstTerm <= 1'b1;
    rstResultRegs <= 1'b1;
    rstCounter <= 1'b1;
    busy <= 1'b0;
    done <= 1'b0;
    if( rst )
      nState <= `RESET;
    else
      nState <= `WAIT_ON_START;
  end
  `WAIT_ON_START: begin
    busy <= 1'b0;
    if( start ) begin
      loadExponent <= 1'b1;
      initResultRegs <= 1'b1;
      initTerm <= 1'b1;
      initCounter <= 1'b1;
      nState <= `CALC_NEXT_TERM_1;
    end
    else
      nState <= `WAIT_ON_START;
  end
end
```

```
//==
//Cor
//==
z
```



Exponentiation Module :

Controller Verilog Code

```

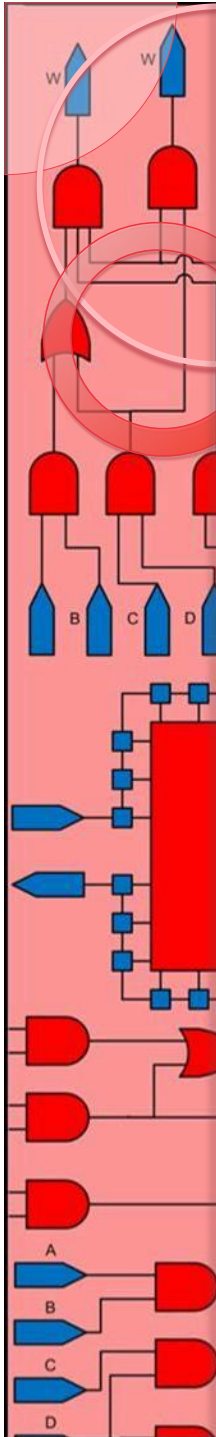
`CALC_NEXT_TERM_1: begin
    selTableData <= 1'b1;
    loadTerm <= 1'b1;
    nState <= `CALC_NEXT_TERM_2;
end
`CALC_NEXT_TERM_2: begin
    loadTerm <= 1'b1;
    incCounter <= 1'b1;
    nState <= `ADD_NEW_TERM;
end

`ADD_NEW_TERM: begin
    loadResultRegs <= 1'b1;
    if( counter < NUM_OF_TERMS )
        nState <= `CALC_NEXT_TERM_1;
    else
        nState <= `CALC_COMPLETE;
    end
end

`CALC_COMPLETE: begin
    busy <= 1'b0;
    done <= 1'b1;
    nState <= `WAIT_ON_START;
end

endcase
end

```

Exponentiation Module :

Complete System Verilog Code

```

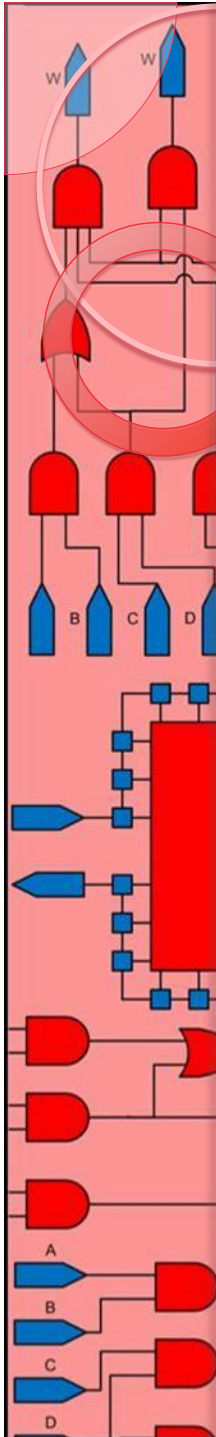
module exp ( clk, rst, start, x, busy, done, resultIPart, resultFPart );
    parameter F_WIDTH = 16;
    parameter NUM_OF_TERMS = 7;
    parameter CNT_WIDTH = 3;
    input  clk, rst, start;
    input  [F_WIDTH - 1:0] x;
    output busy, done;
    output [1:0 ] resultIPart;
    output [F_WIDTH - 1: 0 ]resultFPart;
    wire [CNT_WIDTH - 1:0] addr;
    wire [F_WIDTH - 1: 0 ] tableData;

    expEngine #( F_WIDTH, NUM_OF_TERMS, CNT_WIDTH )
    expEngine1( clk, rst, start, x, busy, done, addr,
    tableData, resultIPart, resultFPart );

    Table  #( CNT_WIDTH, F_WIDTH )
    Table1( addr, tableData );

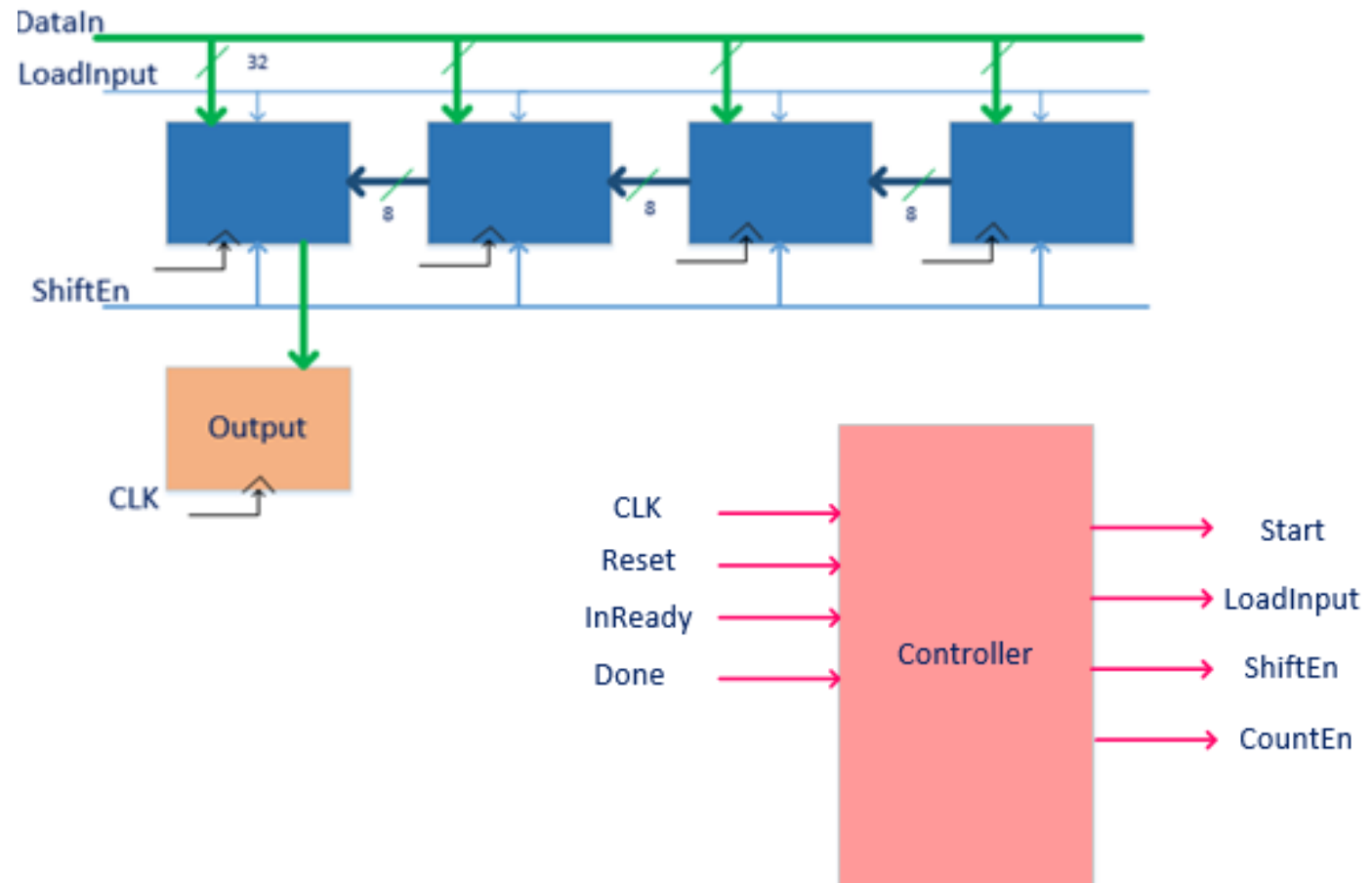
endmodule;

```



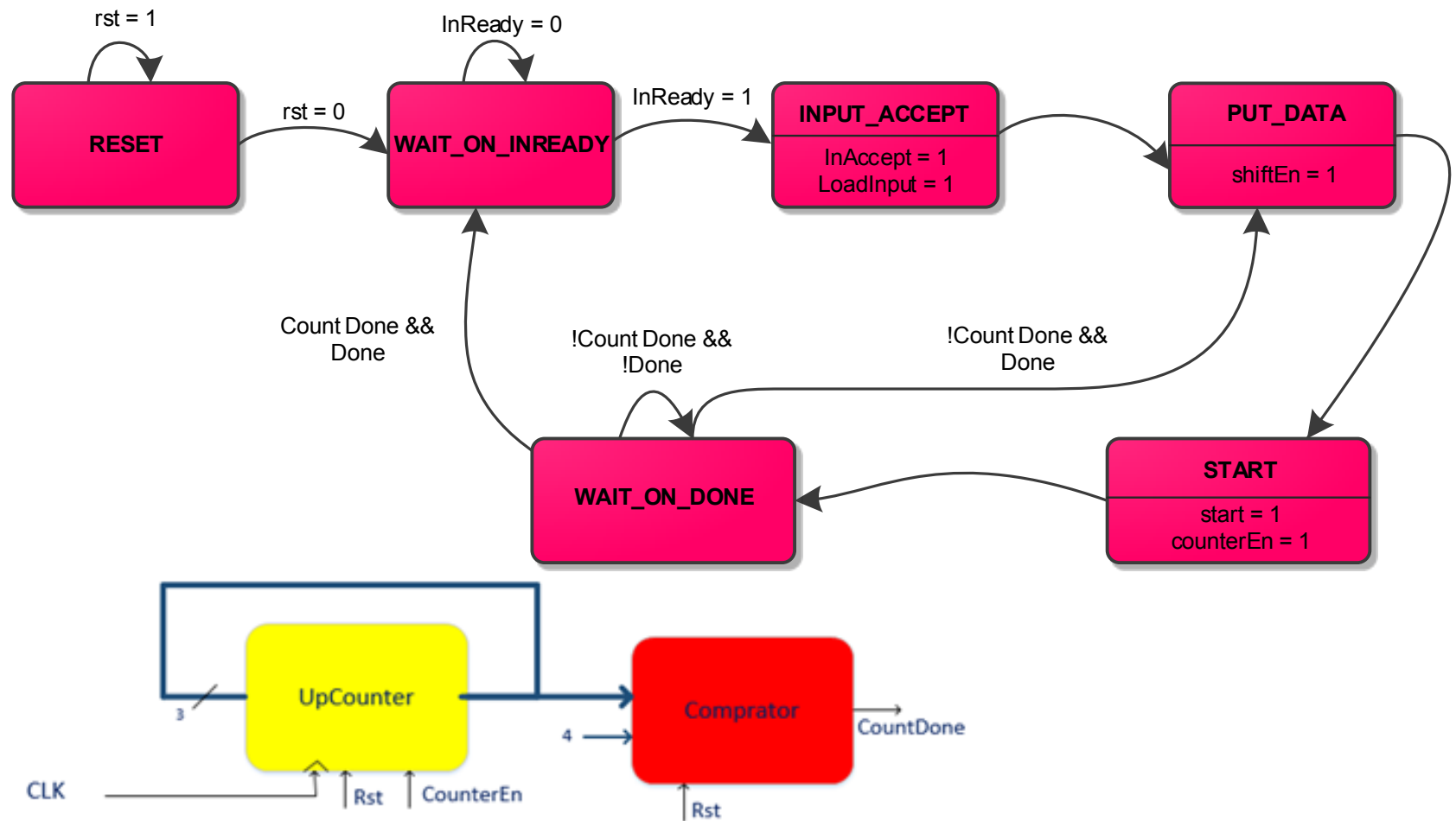
Input Wrapper:

Datapath & Controller Partitioning



Input Wrapper:

State Machine



Input Wrapper :

Datapath Verilog Code

```

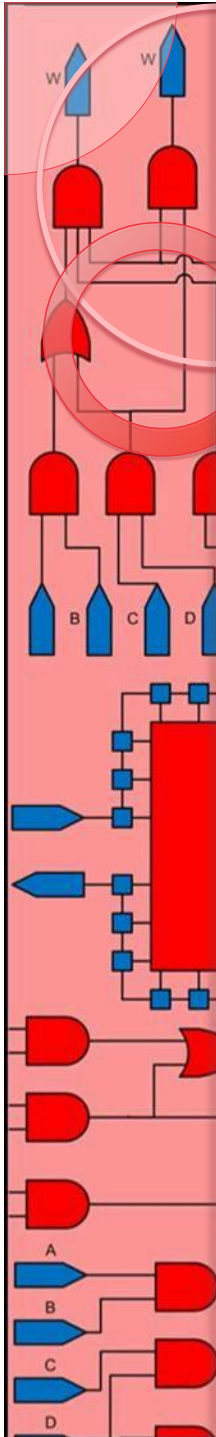
module inputDatapath( clk, rst, loadInput, shiftEn, countEn, dataIn
parameter BUS_SIZE = 32;
parameter DATA_SIZE = 8;
parameter NUM_REGS = 4;
input
input [BUS_SIZE - 1:0] dataIn;
output [DATA_SIZE - 1:0] dataOut;
reg [DATA_SIZE - 1:0] buffer [NUM_REGS-1:0];
reg first;
integer i;

//=====
//Datapath
//=====

//input buffer
always @(posedge clk ) begin
    if( rst ) begin
        for ( i = 0; i < NUM_REGS ; i = i+1 )begin
            buffer[i] <= 0;
        end
    end
    else if( loadInput ) begin
        {buffer[NUM_REGS-1], buffer[NUM_REGS-2],
        buffer[NUM_REGS-3], buffer[NUM_REGS-4]} <= dataIn;
    end
    else if( shiftEn && countEn!=0 ) begin
        for ( i = NUM_REGS-1; i > 0 ; i = i-1 )begin
            buffer[i] <= buffer[i-1];
        end
    end
end

assign dataOut = buffer[NUM_REGS-1];
endmodule;

```



Input Wrapper :

Controller Verilog Code

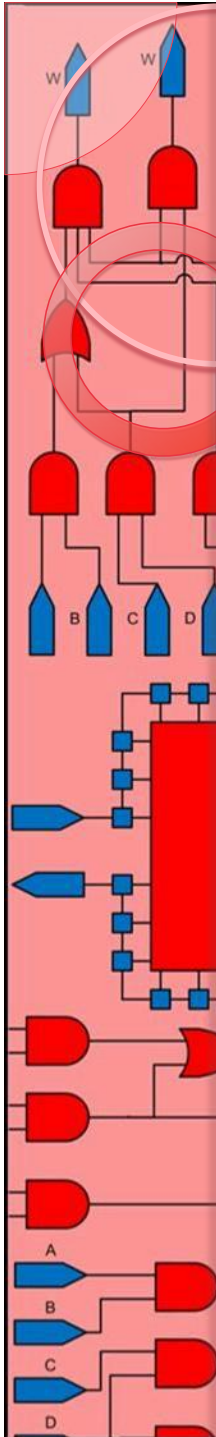
```

`define RESET          0
`define WAIT_ON_INREADY 1
`define INPUT_ACCEPT   2
`define PUT_DATA       3
`define START          4
`define WAIT_ON_DONE   5
module inputController ( clk, rst, inReady, done,
                        inAccepted, start, loadInput, shiftEn, countEn );
    parameter DATA_SIZE = 8;

    input          clk, rst, inReady, done;
    output reg     inAccepted, start, shiftEn, loadInput;
    output         countEn;
    reg [2:0]      ns, ps;
    reg [2:0]      count;
    reg            counterEn, countDone;

    //combinational part
    always @ ( ps or inReady or done or rst or countDone ) begin
        inAccepted = 0;
        start = 0;
        shiftEn = 0;
        counterEn = 0;
        loadInput = 0;
        case ( ps )
            `RESET: begin
                if ( rst == 0 )
                    ns = `WAIT_ON_INREADY;
                else
                    ns = `RESET;
            end
        end
    end

```



Input Wrapper :

Controller Verilog Code

```

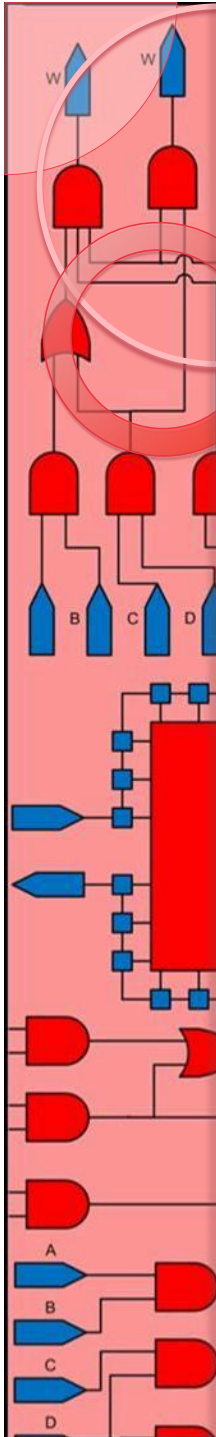
`WAIT_ON_INREADY: begin
    if ( inReady == 1 )
        ns = `INPUT_ACCEPT;
    else
        ns = `WAIT_ON_INREADY;
    end
`INPUT_ACCEPT: begin
    inAccepted = 1;
    loadInput = 1;
    ns = `PUT_DATA;
    end
`PUT_DATA:begin
    shiftEn = 1;
    ns = `START;
    end
`START:begin
    start = 1;
    counterEn = 1;
    ns = `WAIT_ON_DONE;
    end
`WAIT_ON_DONE:begin
    if (countDone && done)
        ns = `WAIT_ON_INREADY;
    else if (done)
        ns = `PUT_DATA;
    else if (!done)
        ns = `WAIT_ON_DONE;
    end
endcase
end

```

```

`define
`define
`define
`define
`define
module i
    para
        input
        output
        reg
        reg
        reg
        //cc
        alwa
        inAc
        star
        shif
        coun
        load
        case

```

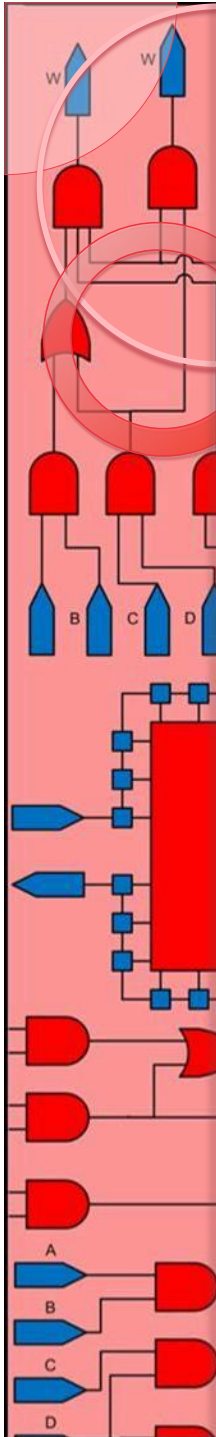


Input Wrapper :

Controller Verilog Code

```
//sequential part
always @ ( posedge clk ) begin
    if ( rst == 1 )
        ps <= `RESET;
    else
        ps <= ns;
    end
//Up Counter
always @( posedge clk ) begin
    if( rst )
        count <= 0;
    else if( counterEn )
        count <= count + 1;
    end

//Comparator
always @( count ) begin
    if( rst )
        countDone = 0;
    else if ( count==4 ) begin
        countDone = 1;
        count = 0;
    end
    else
        countDone = 0;
    end
    assign countEn = 1 ? (count!=0) : 0;
endmodule
```

Input Wrapper :

Complete Verilog Code

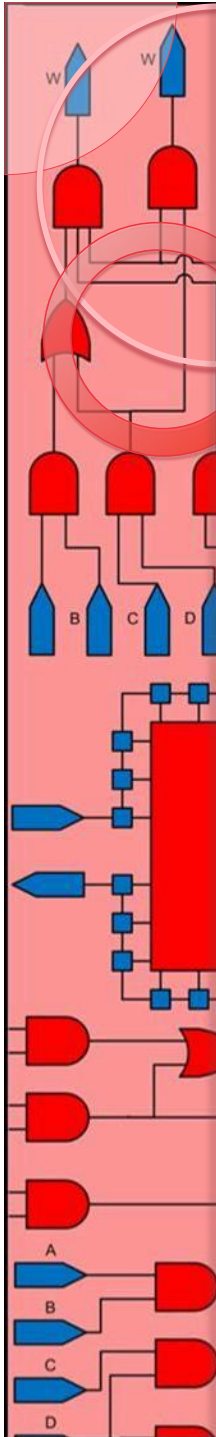
```
module inputWrapper ( clk, rst, dataIn, inReady, busy, done, inAccepted, start, dataOut );

    parameter BUS_SIZE = 32;
    parameter DATA_SIZE = 8;

    input          clk, rst, inReady, busy, done;
    input  [BUS_SIZE-1:0] dataIn;
    output [DATA_SIZE-1:0] dataOut;
    output          inAccepted, start;
    wire            shiftEn, loadInput, countEn;

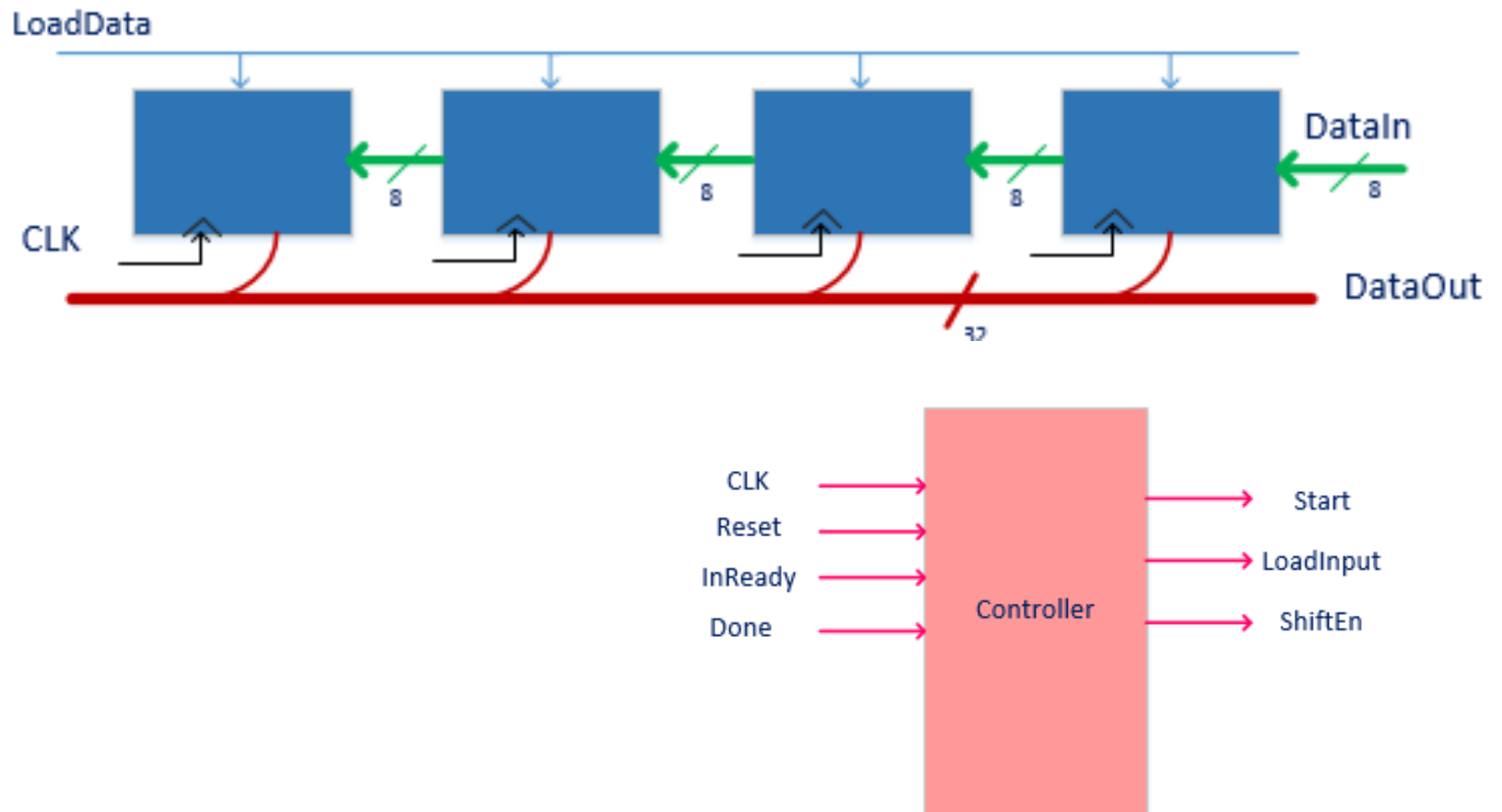
    inputDatapath  #(BUS_SIZE, DATA_SIZE) datapath
    ( clk, rst , loadInput, shiftEn, countEn, dataIn, dataOut );

    inputController #(DATA_SIZE)
    controller( clk, rst, inReady, done, inAccepted,
               start, loadInput, shiftEn, countEn);
endmodule;
```

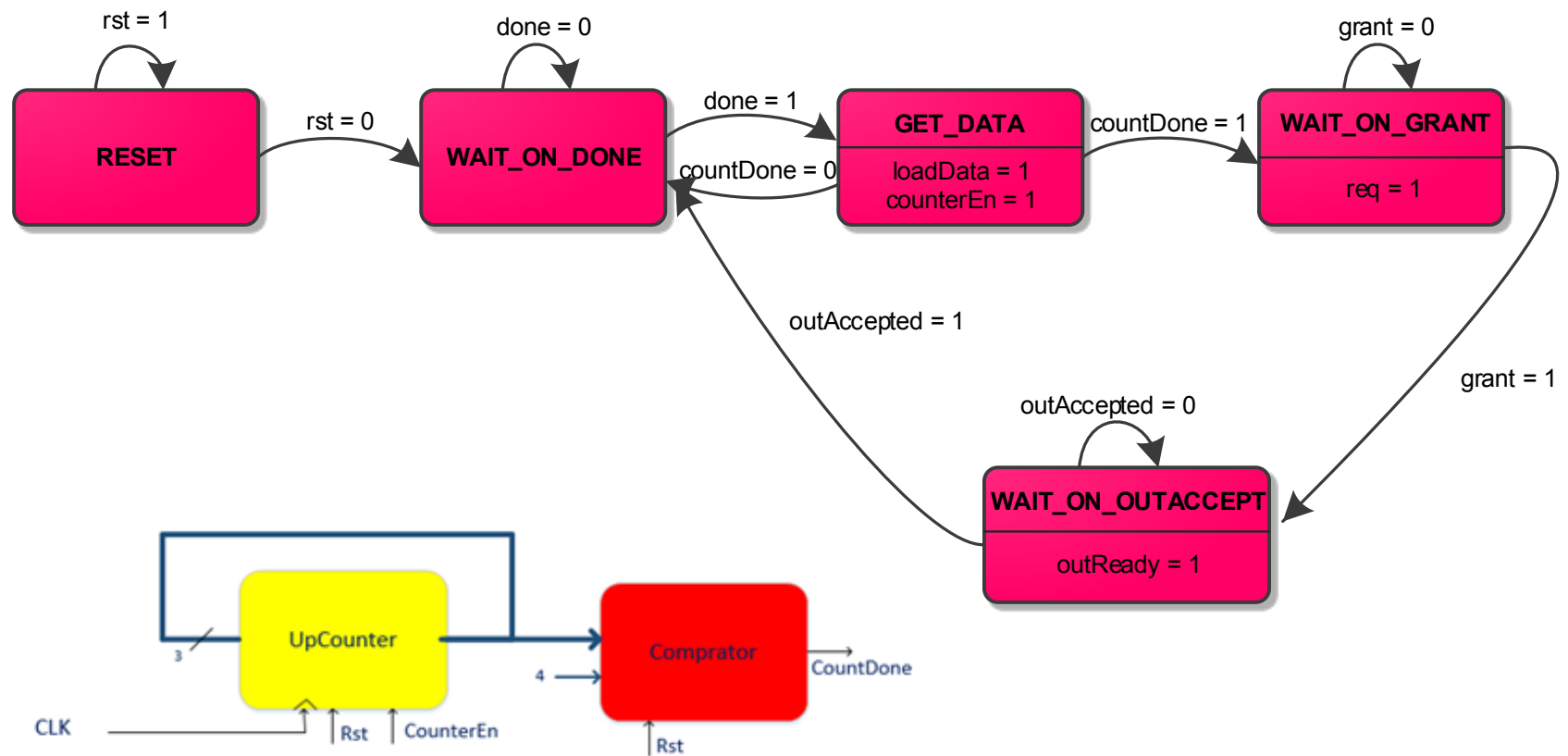
Output Wrapper:

Datapath & Controller Partitioning



Output Wrapper:

State Machine



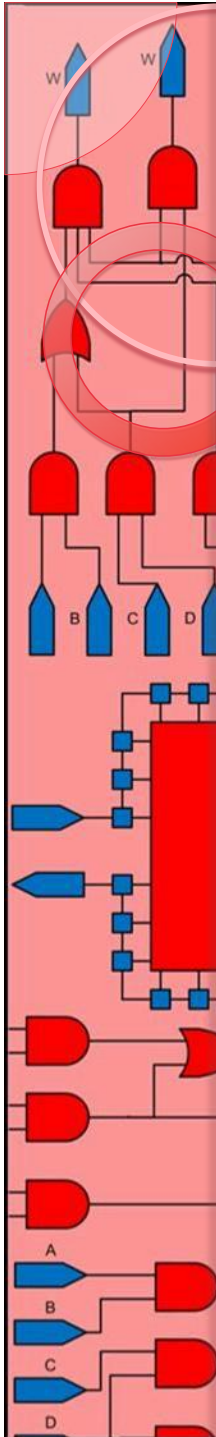
Output Wrapper :

Datapath Verilog Code

```
module outputDatapath ( clk, rst, done, outAccepted,
                      loadData, dataIn, dataOut );

    parameter BUS_SIZE = 32;
    parameter DATA_SIZE = 8;
    parameter NUM_REGS = 4;
    input clk, rst, done, outAccepted, loadData;
    input [DATA_SIZE - 1:0] dataIn;
    output [BUS_SIZE - 1:0] dataOut;
    reg [DATA_SIZE - 1:0] buffer [NUM_REGS-1:0];

    integer i;
```



Output Wrapper :

Datapath Verilog Code

```
//=====
//Datapath
//=====

//output register
always @(posedge clk ) begin
    if( rst ) begin
        for ( i = 0; i < NUM_REGS ; i = i+1 )begin
            buffer[i] <= 0;
        end
    end
    else if( loadData ) begin
        for ( i = NUM_REGS-1; i > 0 ; i = i-1 )begin
            buffer[i] <= buffer[i-1];
        end
        buffer[0] <= dataIn;
    end
end

assign dataOut={buffer[3],buffer[2],buffer[1],buffer[0]};

endmodule;
```

Output Wrapper :

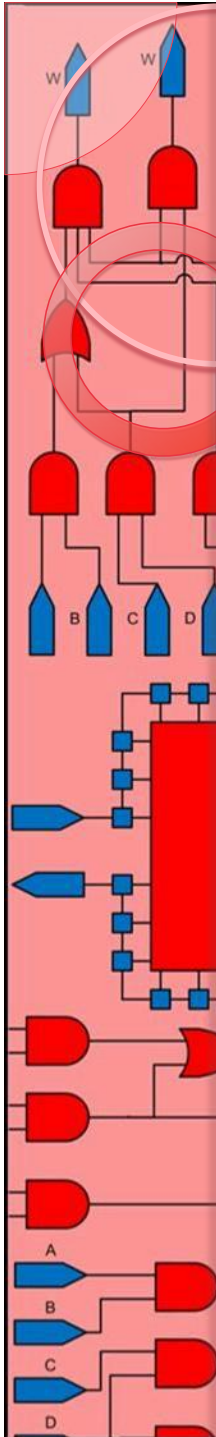
Controller Verilog Code

```
`define RESET                0
`define WAIT_ON_DONE         1
`define GET_DATA             2
`define WAIT_ON_GRANT        3
`define WAIT_ON_OUTACCEPT    4

module outputController ( clk, rst, done, outAccepted, grant, , dataIn,
                        outReady, loadData, req );

    parameter DATA_SIZE = 8;
    input      clk, rst, done, outAccepted, grant;
    input      [DATA_SIZE-1:0] dataIn;
    output reg  outReady, loadData, req;
    reg        [2:0] ns, ps;
    reg        [2:0] count;
    reg        countDone;
    reg        counterEn;

    //combinational part
    always @ ( ps or rst or done or outAccepted or countDone or grant ) begin
        outReady = 0;
        counterEn = 0;
        loadData = 0;
        req = 0;
    end
endmodule
```



Output Wrapper :

Controller Verilog Code

```

case ( ps )
`RESET: begin
    if ( rst == 0 )
        ns = `WAIT_ON_DONE;
    else
        ns = `RESET;
end
`WAIT_ON_DONE: begin
    if ( done == 1 )
        ns = `GET_DATA;
    else
        ns = `WAIT_ON_DONE;
end
`GET_DATA: begin
    loadData = 1;
    counterEn = 1;
    if ( countDone )
        ns = `WAIT_ON_GRANT;
    else
        ns = `WAIT_ON_DONE;
end
`WAIT_ON_GRANT: begin
    req = 1;
    if ( grant )
        ns = `WAIT_ON_OUTACCEPT;
    else
        ns = `WAIT_ON_GRANT;
end

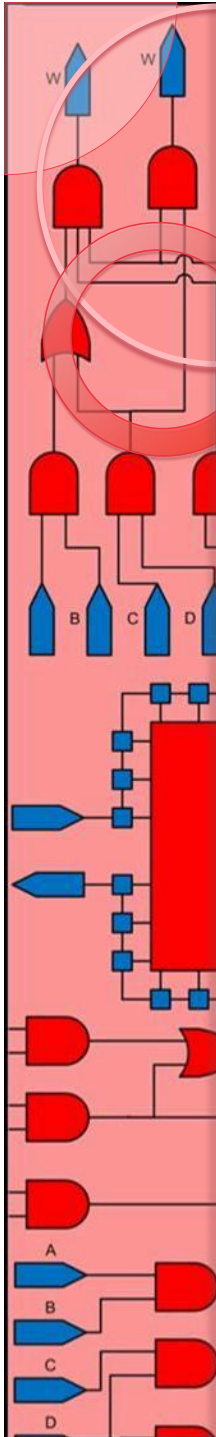
```

```

`define R
`define W
`define G
`define W
`define W

module ou
    .....
    param
    input
    input
    output
    reg
    reg
    reg
    reg
    //com
    always
    outRe
    count
    loadD
    req =

```



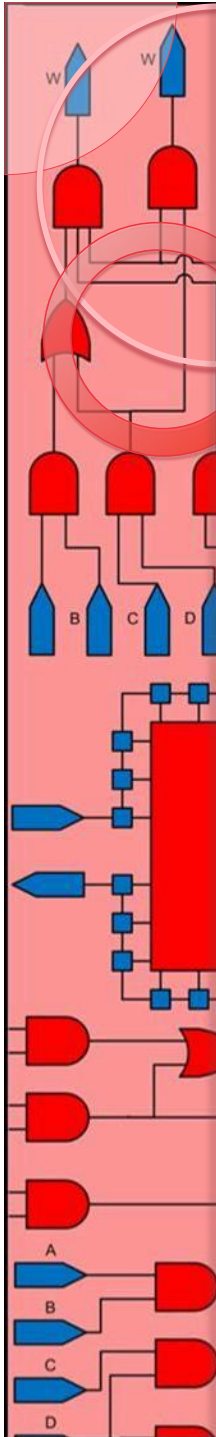
Output Wrapper :

Controller Verilog Code

```

`WAIT_ON_OUTACCEPT:begin
outReady = 1;
if ( outAccepted )
    ns = `WAIT_ON_DONE;
else
    ns = `WAIT_ON_OUTACCEPT;
end
endcase
end
//sequential part
always @ ( posedge clk ) begin
    if ( rst == 1 )
        ps <= `RESET;
    else
        ps <= ns;
    end
end
//Up Counter
always @( posedge clk ) begin
    if( rst )
        count <= 0;
    else if( counterEn )
        count <= count + 1;
    end
end
//Comparator
always @( count ) begin
    if( rst )
        countDone = 0;
    else if ( count==4 )
        countDone = 1;
    else
        countDone = 0;
    end
end

```

Output Wrapper :

Complete Verilog Code

```

module outputWrapper ( clk, rst, dataIn, outAccepted, grant, busy,
                      done, outReady, dataOut, req );

    .....
    parameter BUS_SIZE = 32;
    parameter DATA_SIZE = 8;

    input          clk, rst, outAccepted, busy, done, grant;
    input  [DATA_SIZE-1:0] dataIn;

    output [BUS_SIZE-1:0] dataOut;
    output          outReady, req;

    wire          countDone, loadData, counterEn, loadInput;

    outputDatapath #(BUS_SIZE, DATA_SIZE) datapath
    ( clk, rst, done, outAccepted, loadData,
      dataIn, dataOut );

    .....
    outputController #(DATA_SIZE)          controller
    ( clk, rst, done, outAccepted, grant, dataIn,
      outReady, loadData, req );

    .....
endmodule;

```


Complete System

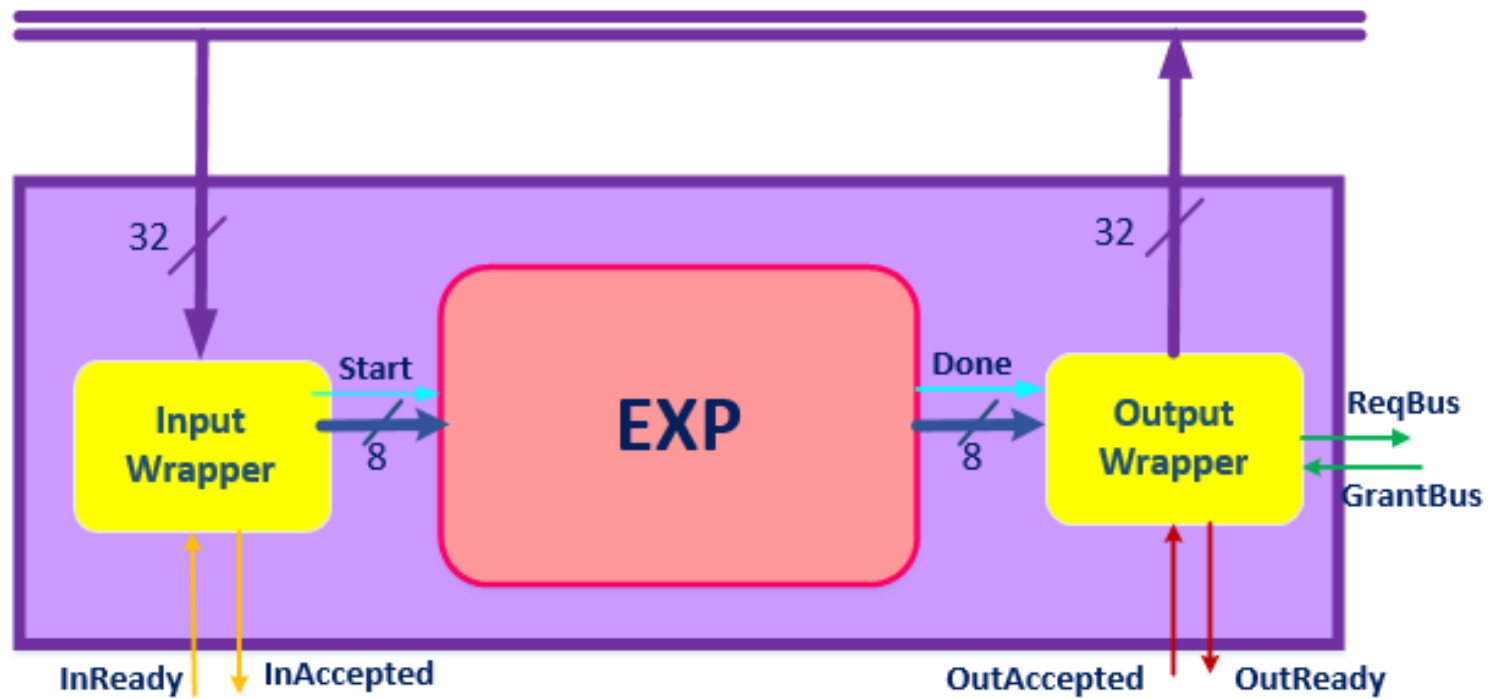
```

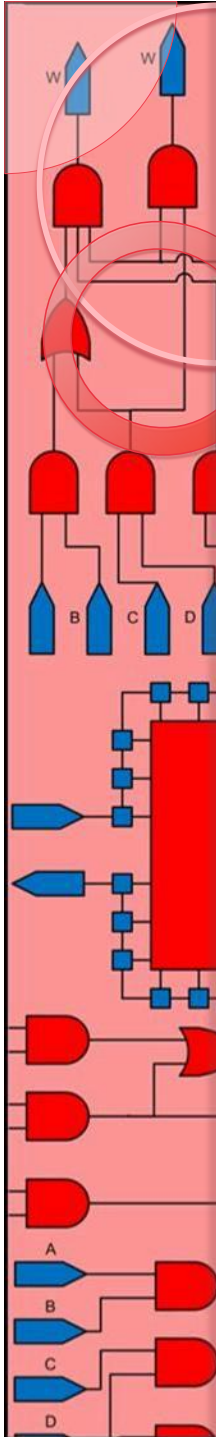
module top ( clk, rst, inReady, grant, dataIn,
            inAccepted, outAccepted, outReady, req, dataBus );

    parameter    BUS_SIZE = 32;
    parameter    DATA_SIZE = 8;
    input        clk, rst, inReady, req;
    input        [BUS_SIZE-1:0] dataIn;
    output       [BUS_SIZE-1:0] inAccepted, outAccepted, grant, outReady, grant;
    output       [BUS_SIZE-1:0] dataBus;
    wire         start, busy, done;
    wire         [DATA_SIZE-1:0] dataOut;
    wire         [DATA_SIZE-1:0] data;
    wire         [DATA_SIZE-1:0] FloatingPart;
    wire         [1:0] integerPart;
    inputWrapper #(BUS_SIZE, DATA_SIZE)    inw
    ( clk, rst, dataIn, inReady, busy, done, inAccepted, start, dataOut );
    exp      #(DATA_SIZE, 3, 3)    inst1
    ( clk, rst, start, dataOut, busy, done, integerPart, FloatingPart);
    outputWrapper #(BUS_SIZE, DATA_SIZE)    outw
    ( clk, rst, data, outAccepted, grant, busy, done, outReady, dataBus, req );
    //select most significant bits
    assign data = {integerPart, FloatingPart[DATA_SIZE-1:DATA_SIZE-6]};
endmodule

```

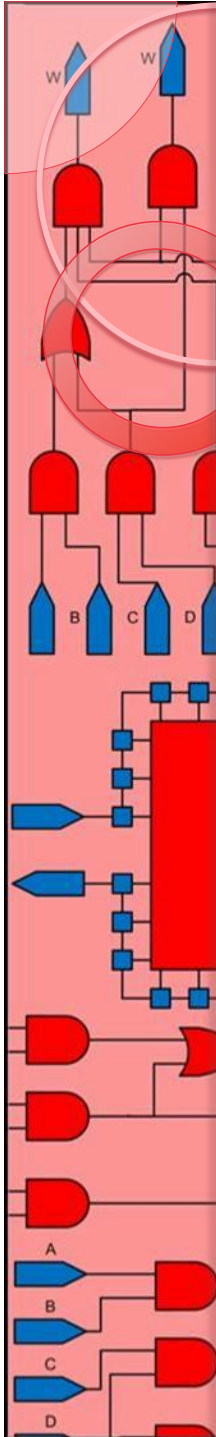
Complete System





Summary

- Overview
- PE and IO wrappers
- A PE example (includes Verilog)
 - Datapath controller partitioning
 - Datapath design
 - Controller design
- IO wrappers
 - Datapath controller partitioning
 - Datapath design
 - Controller design
- Summary



Series Summary

- Basics of logic - Verilog
- Transistor and gate level timing - Verilog
- Boolean expressions – Verilog
- Behavioral logic description – Verilog
- Flip-flops – Verilog
- Combinational RTL elements
- Sequential RTL elements
- RTL design methodology
- Handshaking
- RTL PEs
- RTL Communications
- Complete systems

A complete logic to
RTL course using
HDLs.