

EEE 303

Digital System Design

Assist. Prof. Dr. Şenol GÜLGÖNÜL
2024-2025 Fall Semester



Manager of Control&Electronic Systems

BMC Power · Full-time
2017 - 2022 · 5 yrs

-Managing Engine Control Unit (ECU) and (Transmission Control Unit) development of ALTAY Main Battle Tank and other military diesel ...see more

Skills: OpenECU · MIL-STD-461 · MIL-STD-464 · MIL-STD-1275 · MIL-STD-810 · ISO 26262 · ECE R10 · IPC 620 · Simulink · Embedded Systems · Project



Parttime Instructor

Çankaya Üniversitesi · Part-time
2017 - Less than a year

ECE 439: Satellite and Mobile Communication Systems
ECE 246: Fundamentals of Electronics

Skills: Electronics - Satellite Communications (SATCOM)



Technical Manager

Turksat Uydu Haberleşme Kablo TV ve İşletme A.Ş. · Full-time
2004 - 2016 · 12 yrs
Ankara, Turkey

-VP of Satellite Operations: Involved in Turksat 4A&4B, Turksat 5A&5B and Turksat 6A satellite projects projects. Daily operation of teleport ...see more

Skills: Satellite Systems Engineering · Satellite Ground Systems · Satellite TV Global Navigation Satellite System (GNSS) · Satellite Communications



Network Manager

KoçSistem · Full-time
Mar 2000 - Jul 2000 · 5 mos

Network Manager of Cisco routers

Skills: Cisco Routers



Senior Network Engineer

TurkNet · Full-time
1996 - 1999 · 3 yrs

Network manager of Turnet which was the first commercial internet backbone of SATCOM, Cisco routers

Skills: VSAT · Cisco Routers · Satellite Communications (SATCOM)



Chief Engineer

Turk Telekom · Full-time
1993 - 1995 · 2 yrs

Chief Ground Control Systems engineer of Turksat Satellite Control Center. The first communication satellite of Turkey: Turksat-1A, Turksat-1B and Turksat-1C



Bilkent University

Bachelor of Science - BS
1986 - 1992



Gebze Technical University

Master's degree
1996 - 1998

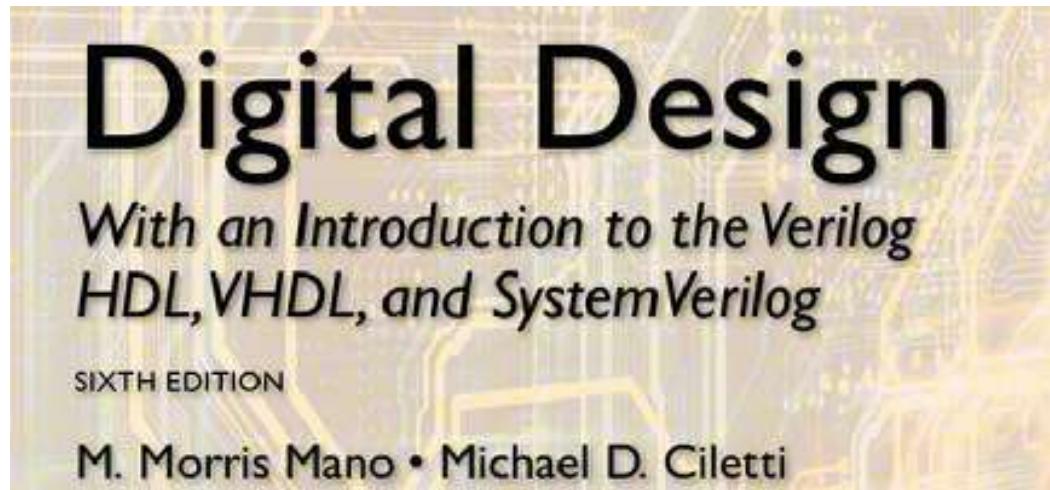


Sakarya University

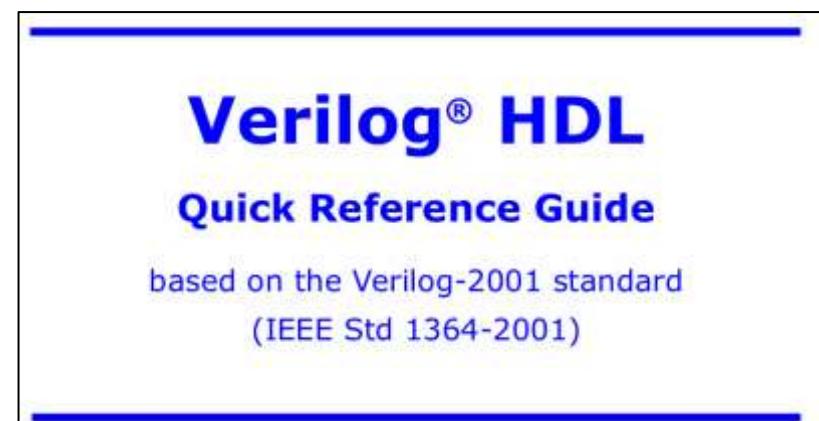
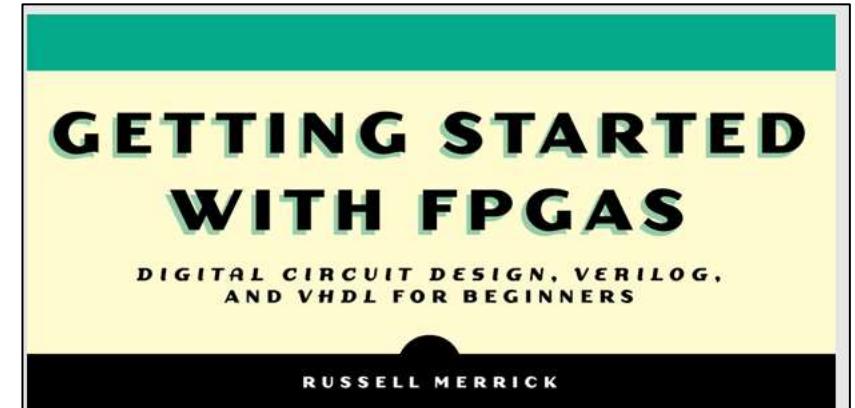
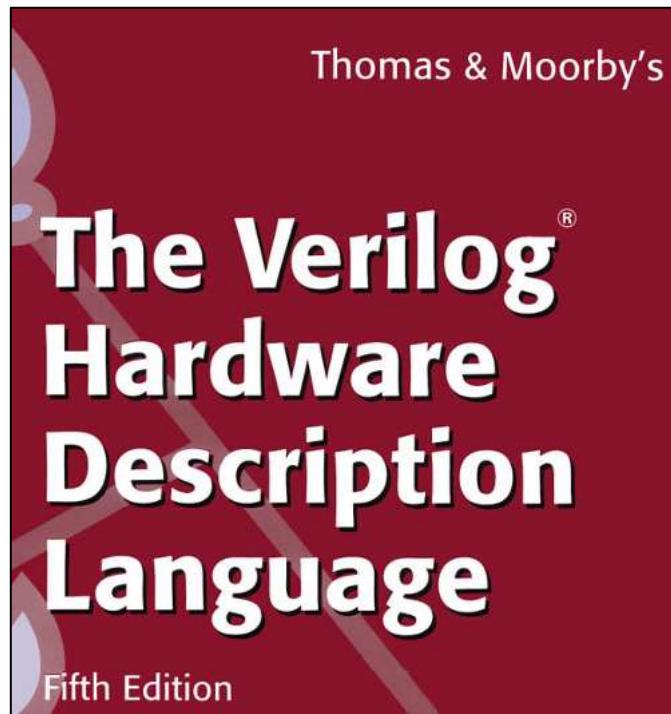
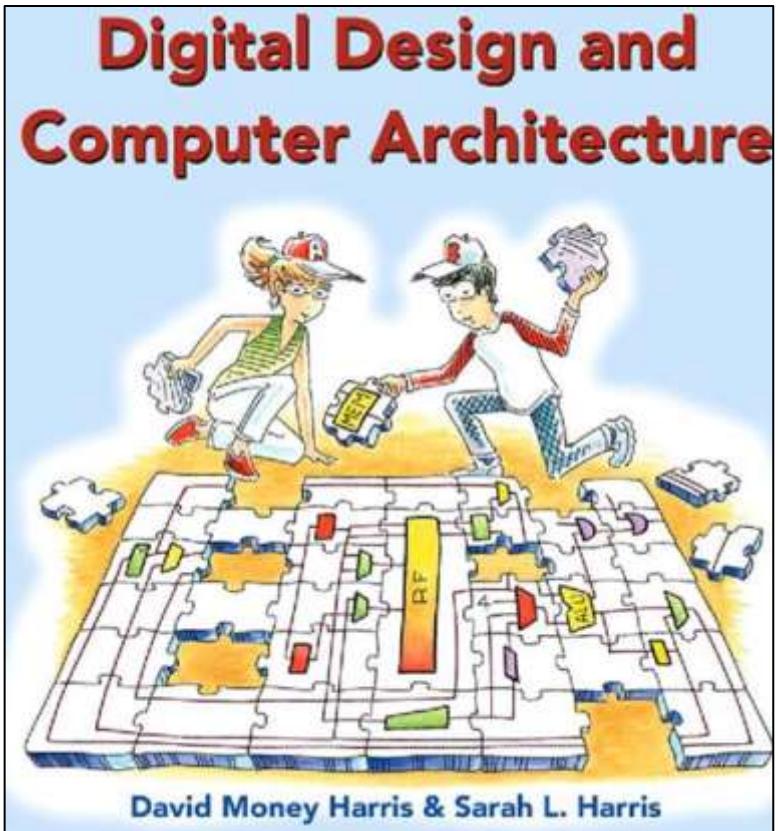
Doctor of Philosophy - PhD
2009 - 2014

ABOUT

- Text Book: Digital Design, Global Edition Mano, M. Morris, Ciletti, Michael
- Grades: 20% Midterm + 20% Project + 60% Final
- Course Objectives: Learning Verilog Hardware Description Language for logic circuit testing, design and applications using FPGAs



Additional Resources



Digital Design and Computer Architecture -
Spring 2024

WHY

- Electronics
 - Analog: RLC, Transistors, Amplifiers, Filters, Electromagnetic, Antenna etc.
 - Digital: Logic Gates, Logic IC's, Microcontrollers, FPGA etc.

SYLLABUS

Week	Subject
1	Introduction
2	Gate Level Modeling and testbench
3	Gate Level Modeling and testbench
4	Data Flow Modeling
5	Data Flow Modeling
6	Midterm
7	Behavioral Modeling
8	Sequential logic circuits using Verilog
9	Sequential logic circuits using Verilog
10	Verilog and FPGA Implementation
11	Verilog and FPGA Implementation
12	Microcontroller architecture
13	Microcontroller architecture
14	Final
15	Final
16	

DEVELOPMENT ENVIRONMENT

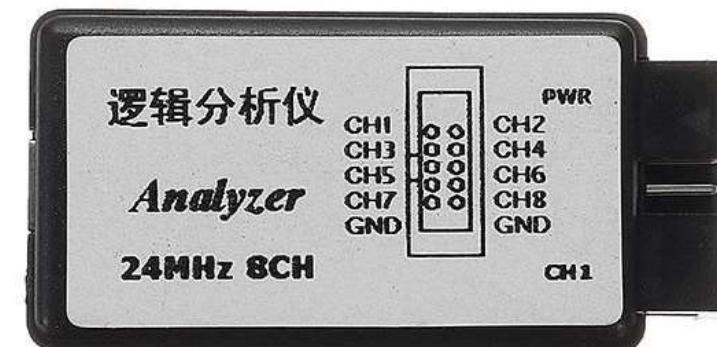
- EDA playground online Verilog IDE
- [Edit code - EDA Playground](#)



- Icarus Simulator + GTKWave
- [Icarus Verilog for Windows \(bleyer.org\)](#)
- [GTKWave \(sourceforge.net\)](#)



- Gowin TANG NANO 9K for FPGA implementation
- Saleae Logic Analyzer
- Breadboard
- M-M jumper cables
- ChatGPT - Copilot



Additional Resources

- [Onur Mutlu Lectures – YouTube](#)

EDApalyground for Verilog

- by using EDApalyground we can test our verilog code using ICARUS simulator.

The screenshot shows the EDApalyground web-based simulation environment. It features two main code editors and a log window at the bottom.

Left Editor (testbench.sv):

```
1 // Code your testbench here SV/Verilog Testbench
2 // or Browse Examples
3 module FullAdder_TB;
4
5   // Inputs
6   reg [3:0] a;
7   reg [3:0] b;
8   reg cin;
9
10  // Outputs
11  wire [3:0] sum;
12  wire cout;
13
14  // Instantiate the module under test
15  MainDesign dut (
16    .a(a),
17    .b(b),
18    .cin(cin),
19    .sum(sum),
20    .cout(cout)
21 );
```

Right Editor (design.sv):

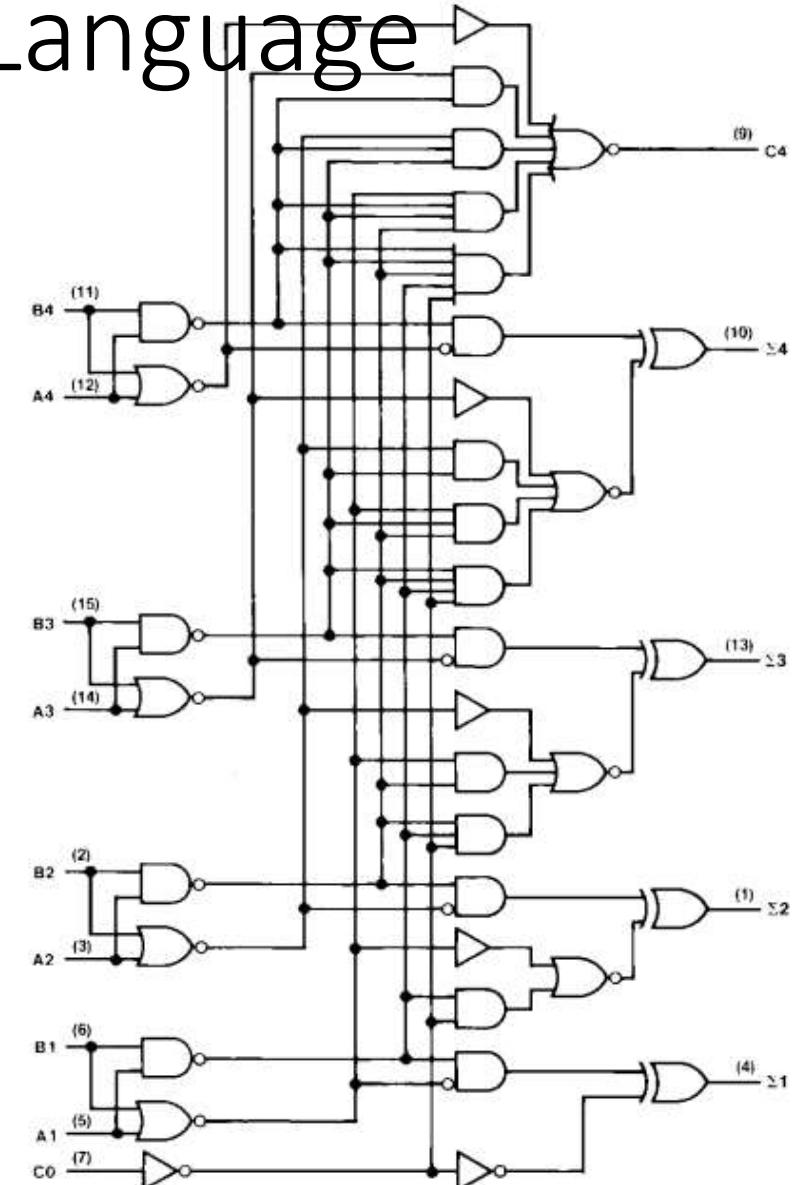
```
1 // Code your design here SV/Verilog Design
2 // or Browse Examples
3
4 module FullAdder(
5   input wire a,
6   input wire b,
7   input wire cin,
8   output wire sum,
9   output wire cout
10 );
11
12 wire [3:0] sum_bits;
13 wire carry_out;
14
15 FullAdder fa0 (.a(a[0]), .b(b[0]), .cin(cin), .sum(sum_bits[0]), .cout(carry_out));
16 FullAdder fa1 (.a(a[1]), .b(b[1]), .cin(carry_out), .sum(sum_bits[1]), .cout(carry_out));
17 FullAdder fa2 (.a(a[2]), .b(b[2]), .cin(carry_out), .sum(sum_bits[2]), .cout(carry_out));
18 FullAdder fa3 (.a(a[3]), .b(b[3]), .cin(carry_out), .sum(sum_bits[3]), .cout(cout));
19
20 assign sum = sum_bits;
21
22 endmodule
```

Log Window:

```
[2023-08-27 09:08:58 UTC] iverilog '-Wall' design.sv testbench.sv && unbuffer vvp a.out
Time=0 a=0101 b=1100 cin=0 sum=xxxx cout=x
Time=10 a=1111 b=0001 cin=1 sum=xxx1 cout=x
Finding user-specified file...
File not found. User-specified file will not open. You requested to open the file 'fourbitadder' but your code does not create a file with this name.
Done
```

Verilog : Hardware Description Language

1. The initial idea was to create a language that could **simulate digital circuits** efficiently and be easy to use.
2. Later Verilog used to design Logic Circuits using **Synthesis** of HDL code
3. Finally Verilog can be used to develop **FPGA Applications**



Verilog=Verification of Logic

Why Testing is Important?

1994

Prof. Thomas Nicely reports bug in Pentium
Restoring Division
Logic error not caught until > 1M units shipped
Recall cost \$450M (!!!)



1997-2000

All major micro-processor manufacturers adopt
formal verification.

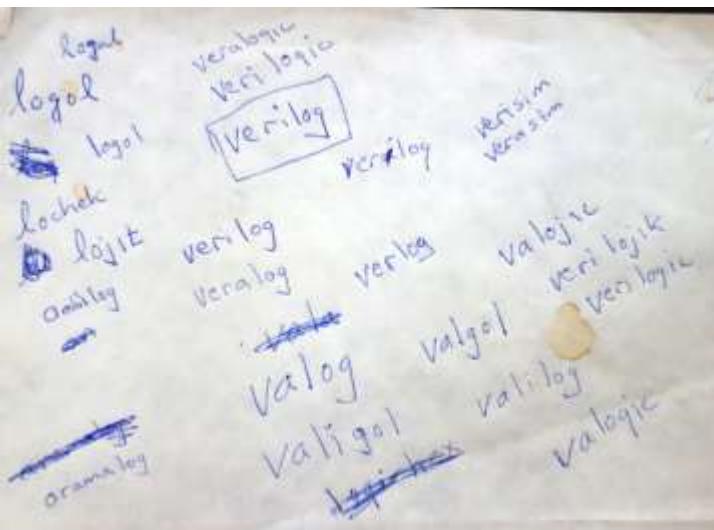
- Verilog allows logic circuit designer to test

Verilog



PHILIP MOORBY

1953-2022



Verilog History

- Introduced in 1984 by Gateway Design Automation
 - 1989 Cadence purchased Gateway and subsequently released Verilog to the public
 - Open Verilog International (OVI) was formed to control the language specifications
 - 1995 IEEE accepted OVI Verilog as a standard
 - 2001 and 2005 IEEE revised standard
 - 2009 Merged with SystemVerilog becoming IEEE Standard 1800-2009

© 2013 Altra Corporation—PubN

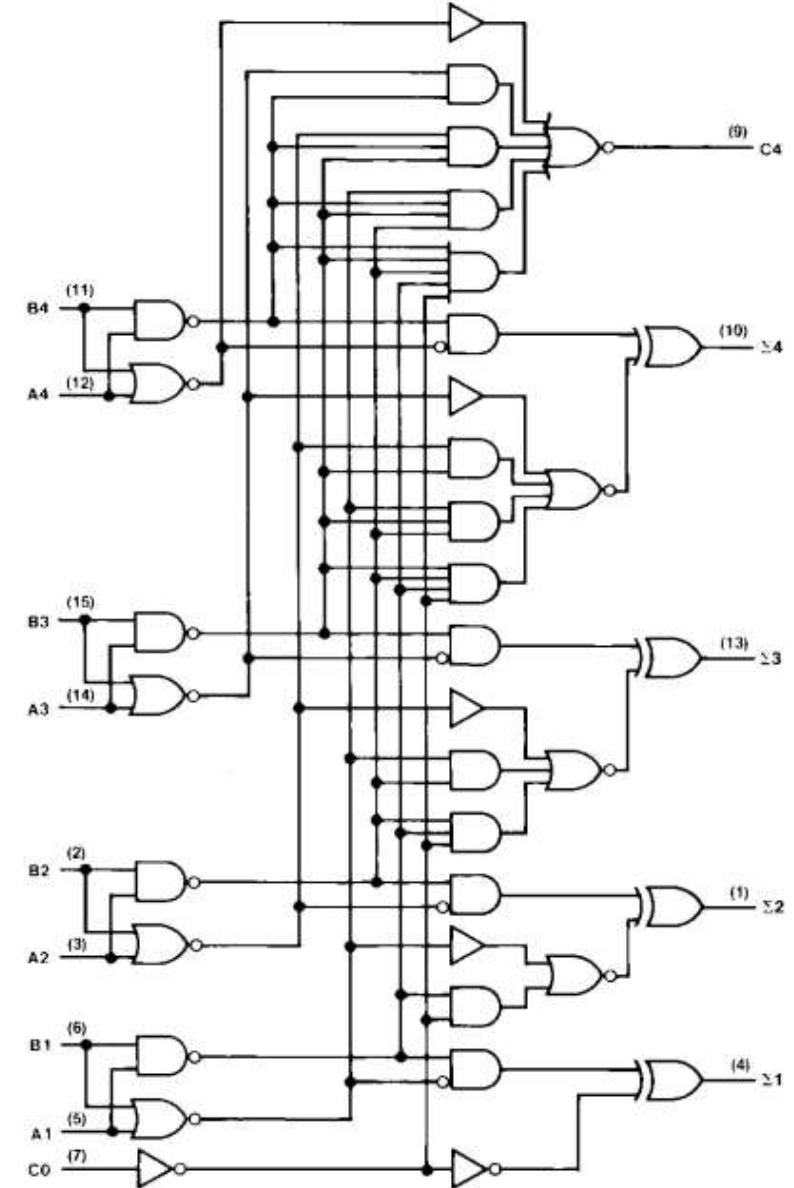
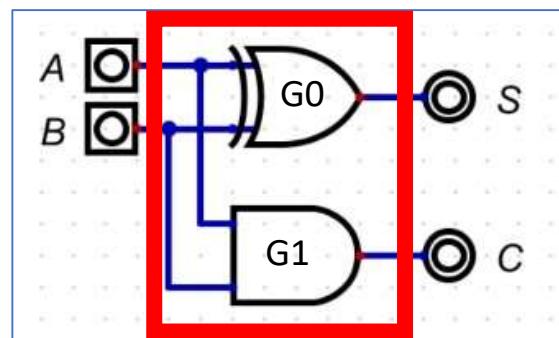
5

ALTERA
MEASURABLE ADVANTAGE™

Verilog HDL Basics (intel.com)

Verilog Module

- Module is fundamental block of Verilog
- Module has **input, output** ports



Verilog Module

- This simple module has only one NAND gate and NOR gate
- we use built-in defined modules nand() and nor()
- These built-in defined modules are called as **primitives**
- **if nothing declared; A,B,C,D data types are 1-bit**

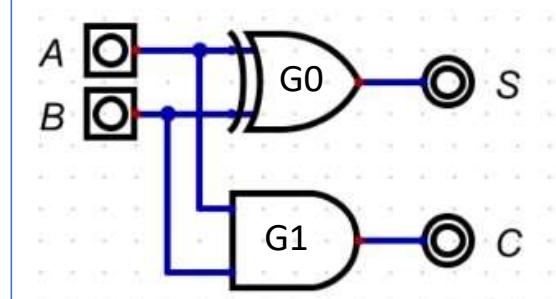
Gate Primitives	Terminal Order and Quantity
and	nand
or	nor
xor	xnor

5.2 Port Declarations

Combined Declarations (*added in Verilog-2001*)

```
port_direction data_type signed range port_name, port_name, ... ;
```

```
// half adder
module halfadder (
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```



Verilog Ports

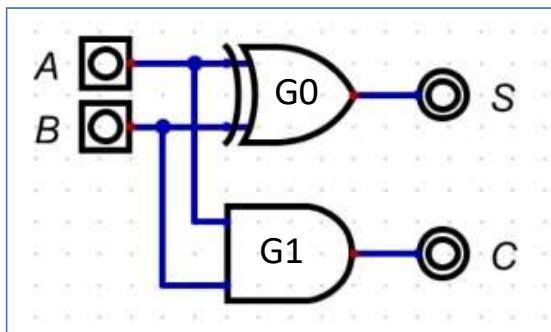
- **port direction** is either input, output
- If no **data type** is declared, it is implicitly declared as a **wire**
- **no comma at the last port**

5.2 Port Declarations

Combined Declarations (*added in Verilog-2001*)

```
port_direction data_type signed range port_name, port_name, ... ;
```

Port Declaration Examples	Notes
input a, b, sel;	three scalar (1-bit) ports

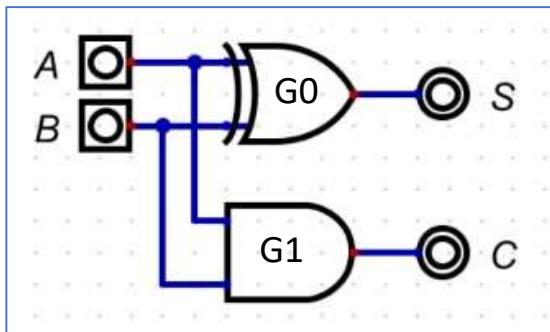


```
// half adder

module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

Verilog Data Types

- Verilog has two major data type classes:
- **Net data types** are used to make connections between parts of a design. (**wire**)
- **Variable data types** are used as temporary storage of programming data. (**reg, integer**)



5.2 Port Declarations

Combined Declarations (*added in Verilog-2001*)

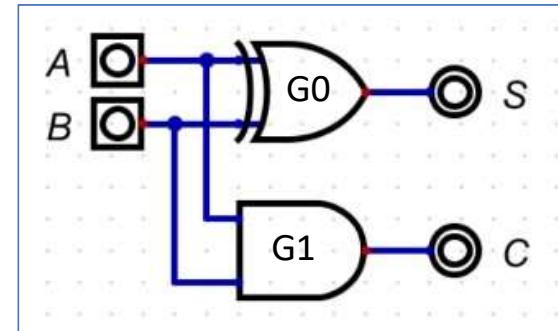
```
port_direction data_type signed range port_name, port_name, ... ;
```

```
// half adder

module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

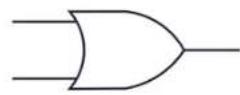
Verilog Primitives

- Verilog has **built-in primitives** to define **basic gates**
- **not, and, nand, or, nor, xor, xnor**
- We need to give a unique instance_name, because we can use the same gate many times within module

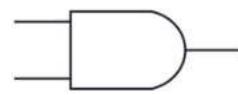


8.0 Primitive Instances

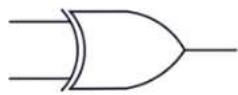
```
gate_type (drive_strength) #(delay) instance_name  
[instance_array_range] (terminal, terminal, ...);
```



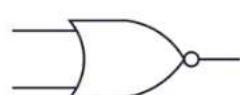
or



and



xor



nor



nand



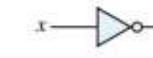
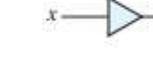
xnor

```
// half adder  
  
module halfadder(  
    output S,C,  
    input A,B  
>;  
    xor G0 (S,A,B);  
    and G1 (C,A,B);  
endmodule
```

Verilog Primitives

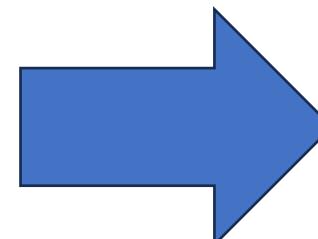
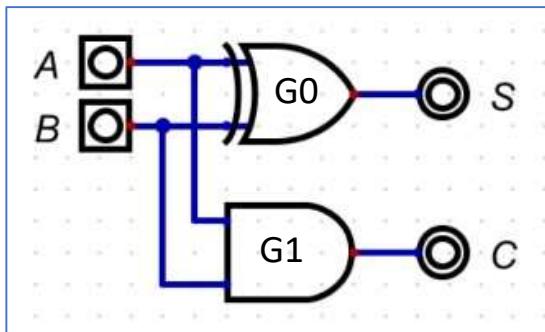
- Verilog has built-in **primitives** to define **basic gates**
- not, and, nand, or, nor, xor, xnor**

Gate Primitives		Terminal Order and Quantity
and	nand	(1-output, 1-or-more-inputs)
or	nor	
xor	xnor	

Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table border="1"> <thead> <tr> <th>x</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table border="1"> <thead> <tr> <th>x</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> </tr> </tbody> </table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y = x \oplus y$	<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y' = (x \oplus y)'$	<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Verilog=Hardware Description Language

- Keep in mind always that Verilog is not a **programming** language
- Verilog is a language to **describe** logic circuits



```
// half adder

module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

Gate-Level Modeling

- **Gate-level** models describe how a circuit is composed of other interconnected elements, such as logic gates or functional blocks.

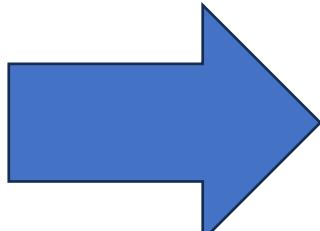
```
// half adder

module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

Verilog 1995-2001 Syntax Changes

- There may be changes in syntax of Verilog 1995 vs 2001.
- We will prefer 2001.
- Both are working, no problem

```
// Verilog 1995  
  
module halfadder(S,C,A,B);  
    output S,C;  
    input A,B;  
  
    xor G0 (S,A,B);  
    and G1 (C,A,B);  
endmodule
```



```
// Verilog 2001  
  
module halfadder(  
    output S,C,  
    input A,B  
)  
    xor G0 (S,A,B);  
    and G1 (C,A,B);  
endmodule
```

ANSI-C Style Port List (*added in Verilog-2001*)

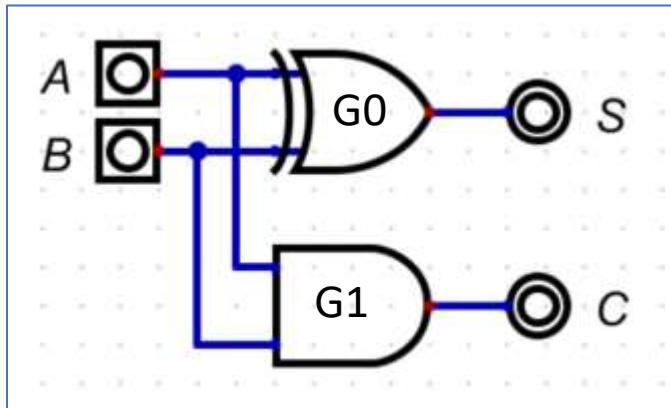
```
module module_name  
#(parameter_declaration, parameter_declaration, ... )  
(port_declaration port_name, port_name, ... ,  
 port_declaration port_name, port_name, ... );  
module items  
endmodule
```

Old Style Port List

```
module module_name (port_name, port_name, ... );  
port_declaration port_name, port_name, ... ;  
port_declaration port_name, port_name, ... ;  
module items  
endmodule
```

Verilog: Half Adder

- Using **gate-level modeling** we can define any **combinatorial logic circuit**



```
// half adder  
  
module halfadder(  
    output S,C,  
    input A,B  
);  
    xor G0 (S,A,B);  
    and G1 (C,A,B);  
endmodule
```

Gate Primitives	Terminal Order and Quantity
and	nand
or	nor
xor	xnor

Verilog Net Data Types

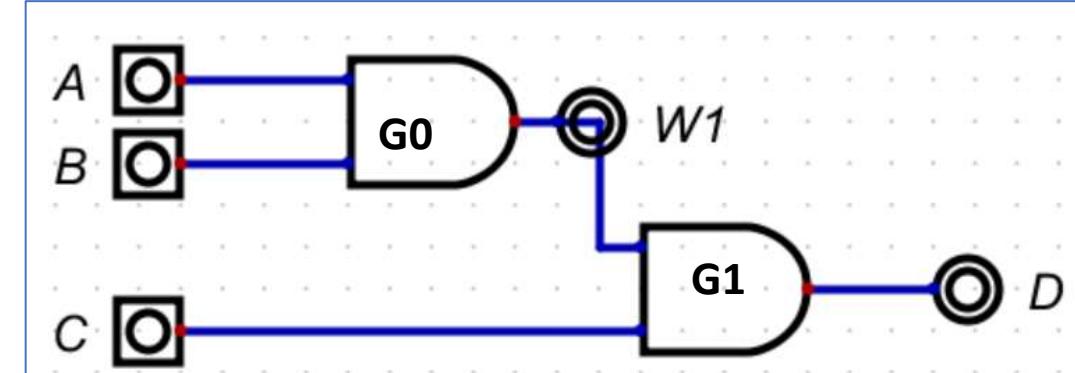
- consider following circuit, having two and gates
- **how can we define in Verilog?**
- We need to give a name to the output of the first gate W1
- input and outputs and wires between gates are called as **wire**.
- **wire is mostly used net data type**

```
module twoand (
    output D,
    input A,B,C
);
    wire W1; //wire
    and G0 (W1,A,B);
    and G1 (D,W1,C);

endmodule
```

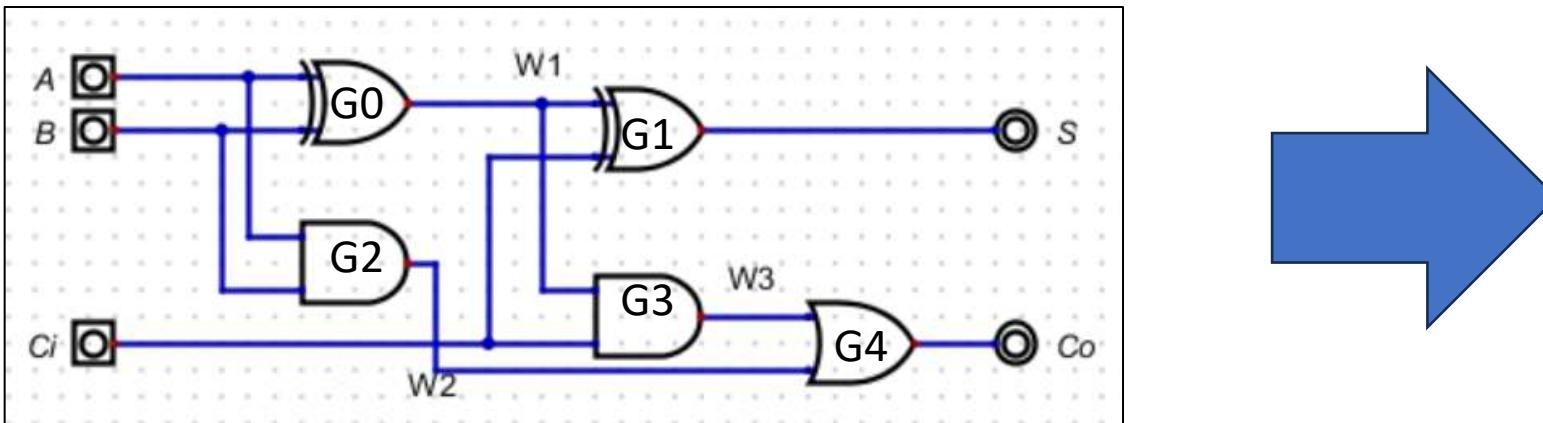
Net Data Type

Net data types connect structural components together.



Complex Combinatorial Circuits: Full Adder

- Using Gate-level modeling we can define any **combinatorial logic** circuit
- name gates G0, G1, etc.
- name wires as W1, W2, etc. for the intermediate connection points
- Sunday puzzle sudoku ☺



```
//full adder
module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;

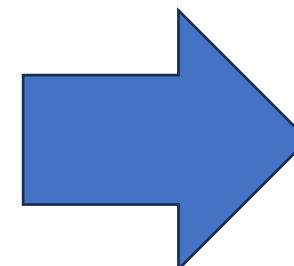
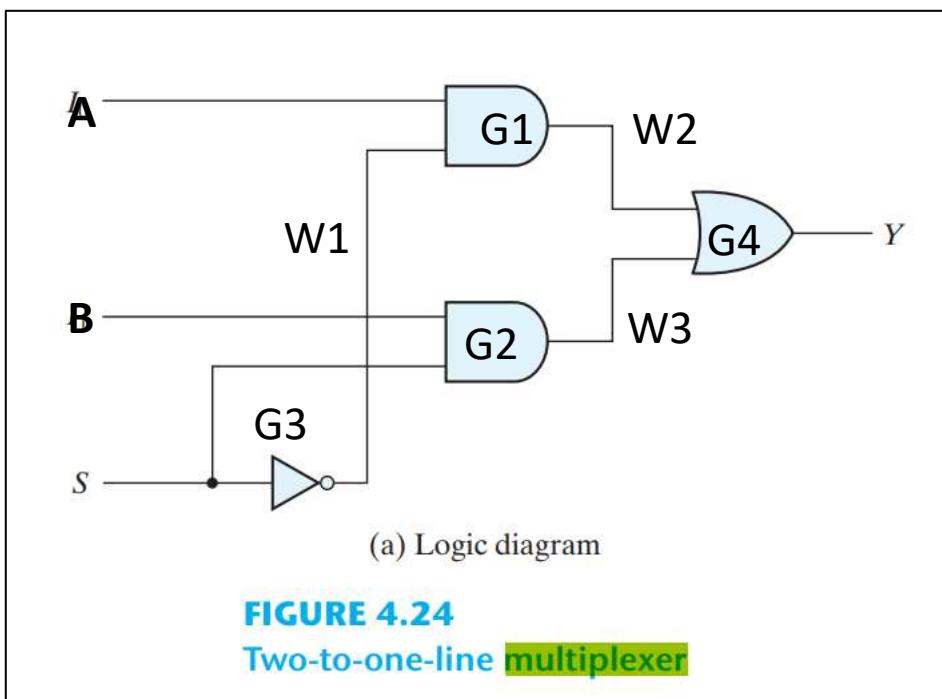
    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);

    and G2 (W2,A,B);
    and G3 (W3,Ci,W1);

    or G4 (Co,W2,W3);
endmodule
```

Complex Combinatorial Circuits: MUX

- Using Gate-level modeling we can define any **combinatorial logic** circuit
- name gates G0, G1, etc.
- name wires as W1, W2, etc. for the intermediate connection points
- **Sunday puzzle sudoku** ☺



```
//MUX
module mux2to1(
    output Y,
    input A,B,S
);
    wire W1,W2,W3;
    and G1 (W2,A,W1);
    and G2 (W3,B,S);
    not G3 (W1,S);
    or G4 (Y,W2,W3);
endmodule
```

Complex Combinatorial Circuits: MUX

- Using Gate-level modeling we can define any **combinatorial logic** circuit
- name gates G0, G1, etc.
- name wires as W1, W2, etc. for the intermediate connection points
- **Sunday puzzle sudoku** ☺

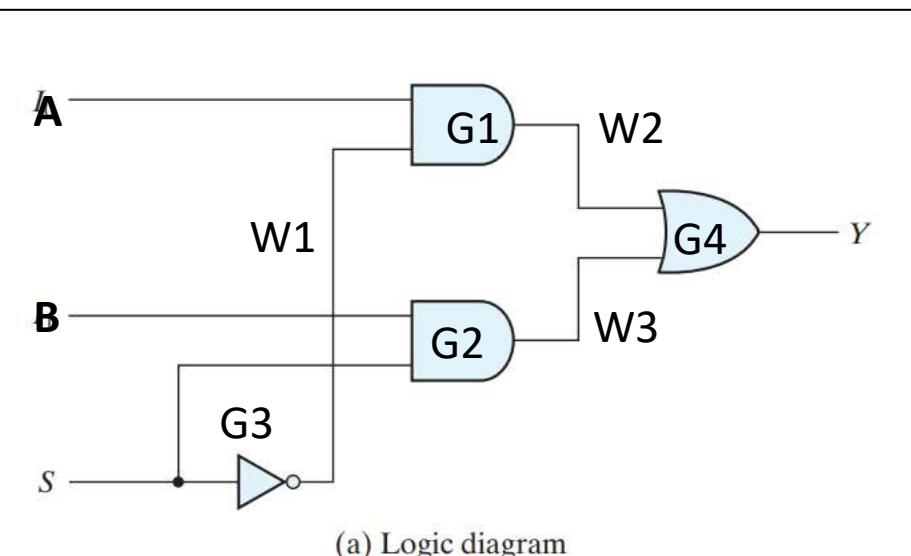
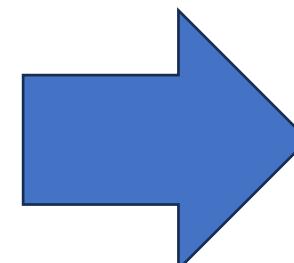


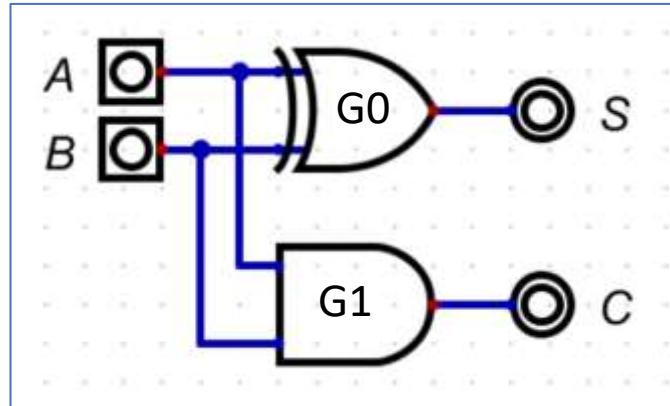
FIGURE 4.24
Two-to-one-line **multiplexer**



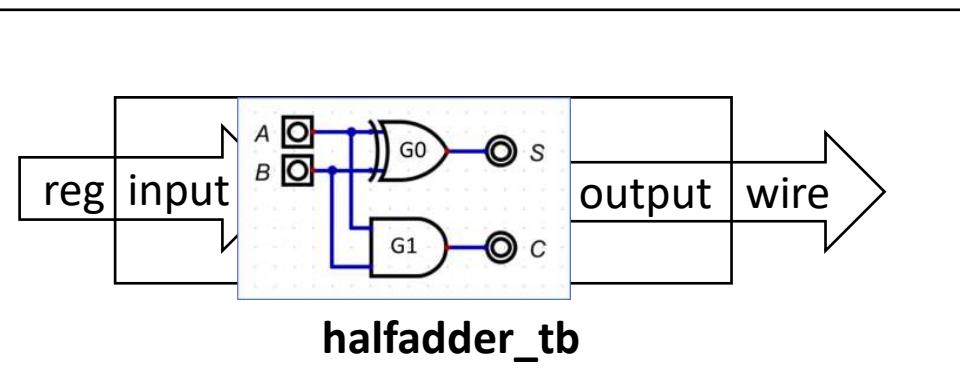
```
module twotoone(  
    input A,B, S,  
    output Y);  
  
    wire W1,W2,W3; //wires  
  
    and G1 (W2, A,W1);  
    and G2 (W3, B, S);  
    not G3 (W1, S);  
    or G4 (Y, W2,W3);  
  
endmodule
```

Verilog Test Bench

- Test bench is also written in Verilog
- test bench module has **no input or output ports**, because **it does not interact with its environment.**



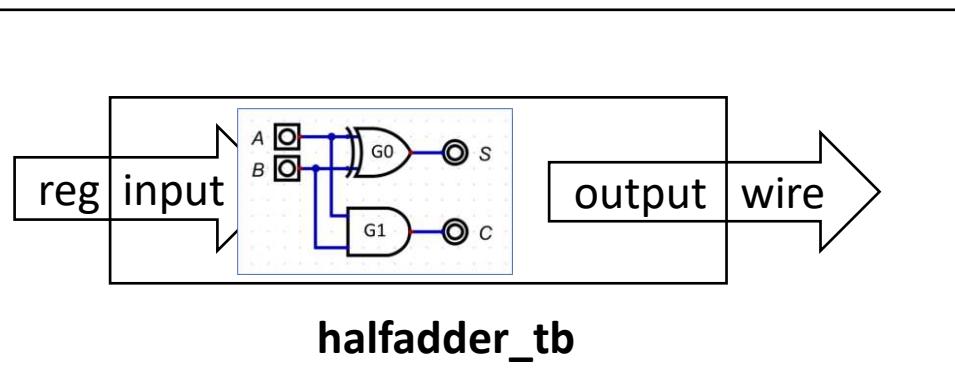
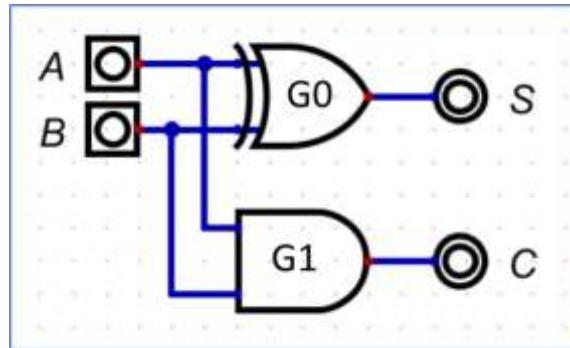
```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```



```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cs");
    $monitor("%b%b:%b%b",a,b,c,s);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
end // initial
endmodule
```

Verilog Test Bench: input output

- **inputs** to the circuit under test are declared with keyword **reg**
- **outputs** of the circuit under test are declared with the keyword **wire**



```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Variable Data Types

- *variable_type* is one of the following:

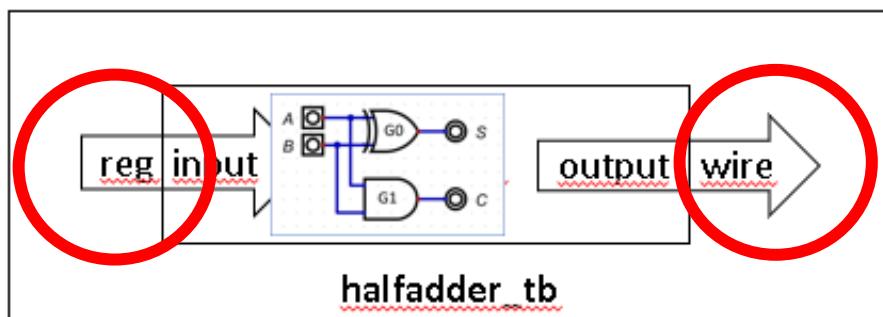
reg

a variable of any bit size; unsigned unless explicitly declared as signed

- *net_type* is one of the following keywords:

wire

interconnecting wire; CMOS resolution



```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Test Bench: Instantiation

- The module to be tested is **instantiated** with the user-chosen **unique instance** module names; example M0
- **Verilog is case sensitive** 'A' and 'a' are not the same
- We prefer **port connections defined by name**

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,Ali,B);
    and G1 (C,A,B);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.Ali(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Test Bench: Instantiate

- in instantiation, order or parameters not important
- parameters are **Case Sensitive Ali is not equal ali**
- **port connections are defined by name .Ali(c)**

```
//instantiation  
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
```



```
//half adder  
module halfadder(  
    output S,C,  
    input A,B  
);  
    xor G0 (S,A,B);  
    and G1 (C,A,B);  
endmodule
```

Verilog Test Bench: initial

- **initial** is a procedural block
- **runs once only**

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Test Bench: waveform

- besides text printout of \$display and \$monitor we can examine time waveform analysis of module under test
- add \$dumpfile and \$dumpvars
- \$dumpfile will dump the changes in a file named .vcd
- \$dumpvars selects which variables to be dumped into dumpfile. no variable declared means **dump all variables** in the test bench module
- add a \$finish to set where to stop the waveform. otherwise it will dump infinitely and create a big dumpfile
- select open EPWave after run option

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
reg a,b; //reg for inputs
wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("ab:cs");
$monitor("%b%b:%b%b",a,b,c,s);
#1 a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
#10 $finish;
end // initial
endmodule
```

Run Options

Run Options

Open EPWave after run

Verilog Test Bench: \$display(), \$monitor()

- **\$display()** prints arguments at formatted text
- **\$monitor()** prints only if arguments changes

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Test Bench \$display(), \$monitor()

17.1 Text Output System Tasks

```
$display("text_with_format_specifiers", list_of_arguments);
```

Prints the formatted message when the statement is executed. A newline is automatically added to the message. If no format is specified, the routines default to decimal, binary, octal and hexadecimal formats, respectively.

```
$monitor("text_with_format_specifiers", list_of_arguments);
```

Invokes a background process that continuously monitors the arguments listed, and prints the formatted message whenever one of the arguments changes. A newline is automatically added to the message.

Text Formatting Codes			
%b	binary values	%m	hierarchical name of scope
%o	octal values	%l	configuration library binding
%d	decimal values	\t	print a tab
%h	hex values	\n	print a newline
%e	real values—exponential	\"	print a quote
%f	real values—decimal	\\"	print a backslash
%t	formatted time values	%%	print a percent sign
%s	character strings		

Verilog Test Bench: timescale, #(delay)

- **`timescale 1 ns / 1 ns**
- Specifies the time units and precision for delays
- **Delays** execution of the next statement for a specific amount of time.

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cs");
    $monitor("%b%b:%b%b",a,b,c,s);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
end // initial
endmodule
```

\$display(), \$monitor()

- we can print \$time within \$monitor to see changes in #tics
- notice that timescale 10ns / 1ns will not change the time ticks
- there is better way to see response in time

```
$monitorh("text_with_format_specifiers", list_of_arguments);
```

Invokes a background process that continuously monitors the arguments listed, and prints the formatted message whenever one of the arguments changes. A newline is automatically added to the message.

Text Formatting Codes

%b	binary values	%m	hierarchical name of scope
%o	octal values	%l	configuration library binding
%d	decimal values	\t	print a tab
%h	hex values	\n	print a newline
%e	real values-exponential	\"	print a quote
%f	real values-decimal	\\	print a backslash
%t	formatted time values	%%	print a percent sign
%s	character strings		

%Ob, %Oo, %Od and %Oh truncates any leading zeros in the value.

%e and %f may specify field widths (e.g. %.5.2f).

%m and %l do not take an argument; they have an implied argument value.

The format letters are not case sensitive (i.e. %b and %B are equivalent).

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
reg a,b; //reg for inputs
wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("ab:cs");
$monitor("%b%b:%b%b",a,b,c,s);
#1 a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
#10 $finish;
end // initial
endmodule
```

```
ab : cs
xx : xx
00 : 00
01 : 01
10 : 01
11 : 10
```

Verilog Test Bench: assignment

10.3 Procedural Assignment Statements

variable = expression;

Blocking procedural **assignment**. Expression is evaluated and assigned when the statement is encountered. In a **begin—end** sequential statement group, execution of the next statement is blocked until the **assignment** is complete. In the sequence **begin m=n; n=m; end**, the first **assignment** changes **m** before the second **assignment** reads **m**.

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Test Bench: \$finish

\$finish(*n*) ;

Finishes a simulation and exits the simulation process. *n* (optional) is 0, 1 or 2, and may cause extra information about the simulation to be displayed.

- \$finish does not have an affect inside initial begin-end because initial block runs ONCE
- but it is important in loop procedural block always @

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

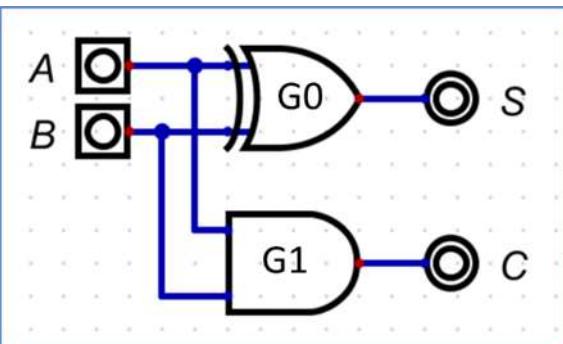
```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("ab:cs");
        $monitor("%b%b:%b%b",a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

Verilog Test Bench: review results

Table 4.3
Half Adder

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

ab:cs
xx:xx
00:00
01:01
10:01
11:10

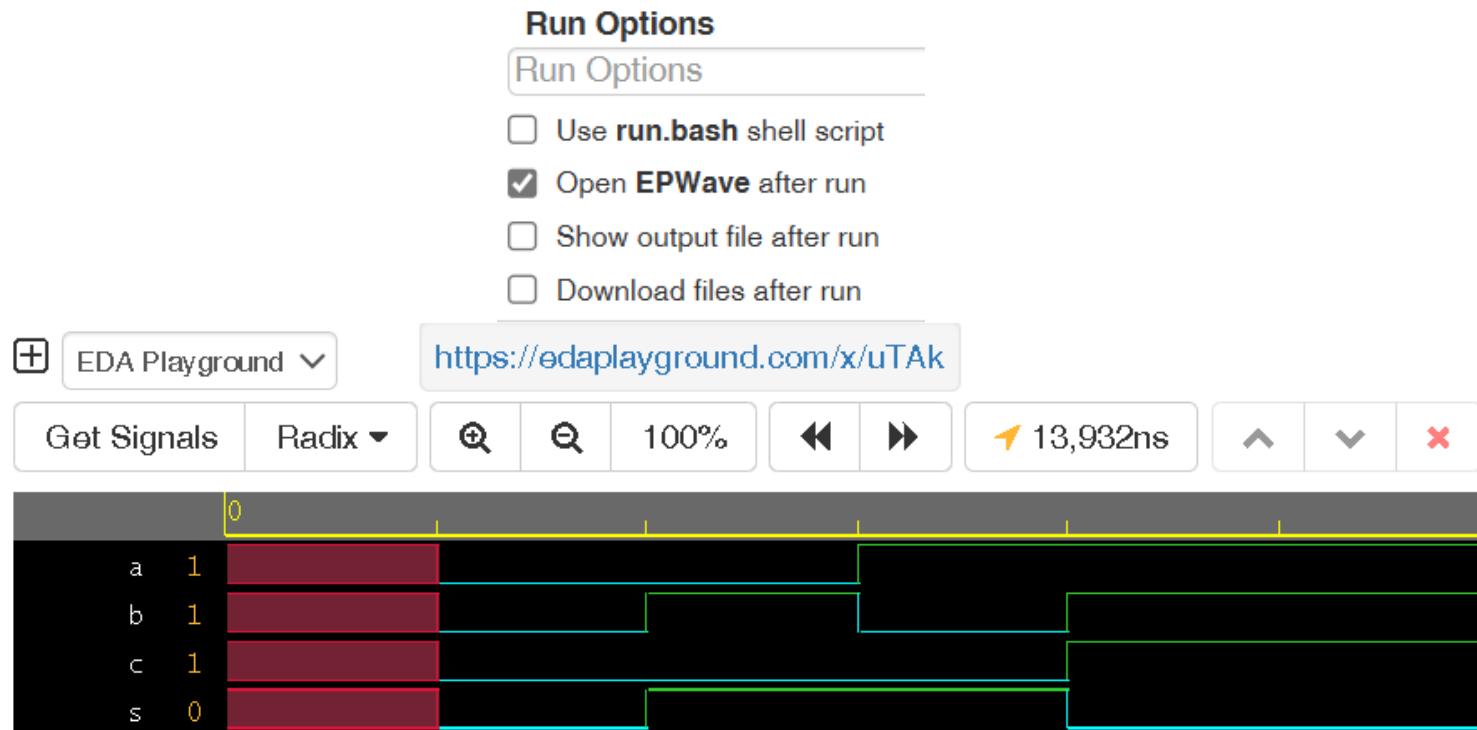


```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
reg a,b; //reg for inputs
wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cs");
    $monitor("%b%b:%b%b",a,b,c,s);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
end // initial
endmodule
```

Verilog Test Bench: waveform

- remove the signals you don't want. remember that we dumped all
- select Radix to binary
- values of variables are show at time selected by cursor



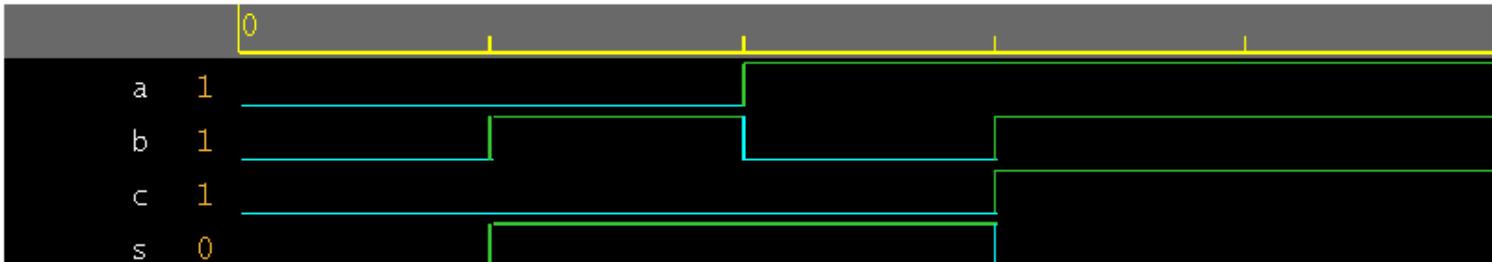
```
//testbench
`timescale 1 ns / 1 ns
module testbench;
reg a,b; //reg for inputs
wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("ab:cs");
$monitor("%b%b:%b%b",a,b,c,s);
#1 a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
#10 $finish;
end // initial
endmodule
```

Verilog Test Bench: waveform

- Logic values 1,0,X. x means unknown
- `initial` (optional) is used to define the initial (power-up) state for sequential UDP's. Only the logic values 0, 1, and X may be used. The default state is X. In Verilog-2001, the initial value can be assigned in the declaration.



```
$monitor("%b%b:%b%b",a,b,c,s);  
#1 a=0; b=0;  
#1 a=0; b=1;
```



```
$monitor("%b%b:%b%b",a,b,c,s);  
a=0; b=0;  
#1 a=0; b=1;
```

4.8 Logic Values

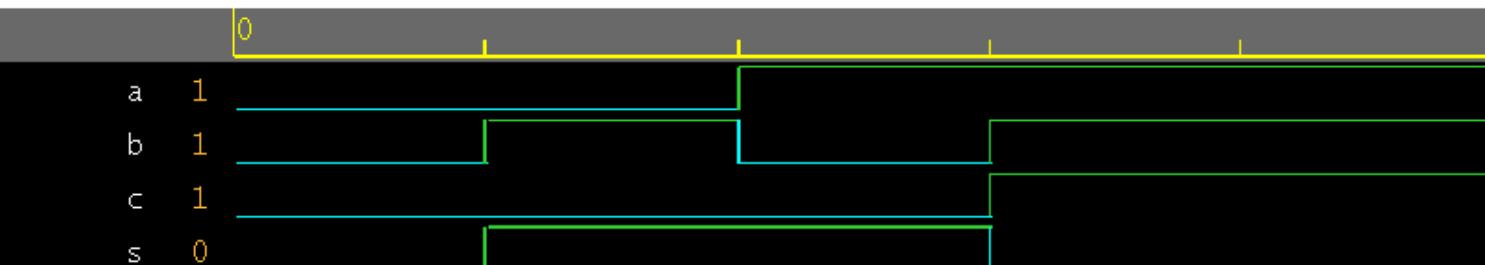
Verilog uses a 4 value logic system for modeling. There are two additional unknown logic values that may occur internal to the simulation, but which cannot be used for modeling.

Z is only for wires cannot be used for reg data type

Logic Value	Description
0	zero, low, or false
1	one, high, or true
z or Z	high impedance (tri-stated or floating)
x or X	unknown or uninitialized



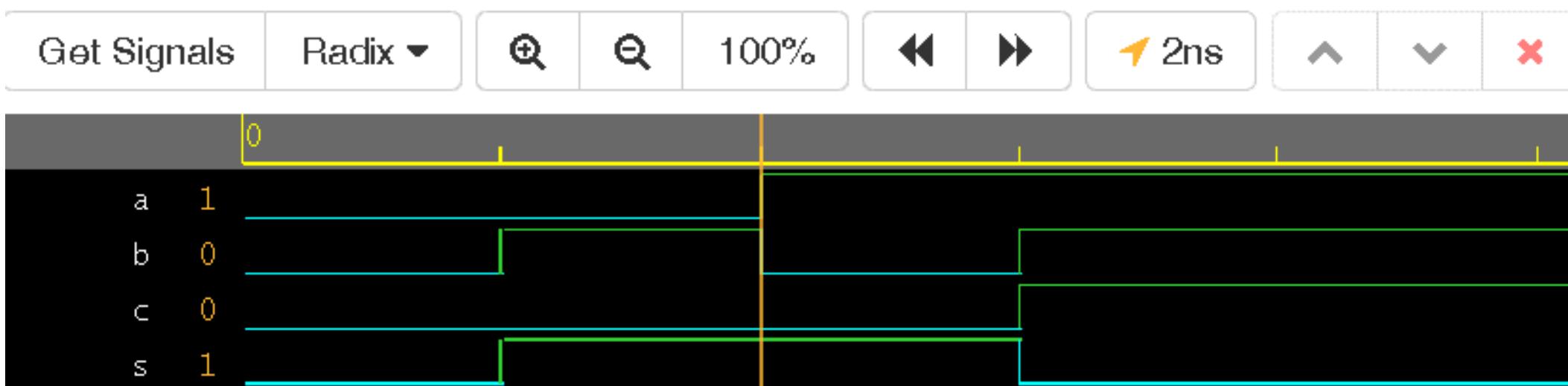
```
$monitor("%b%b:%b%b",a,b,c,s);  
#1 a=0; b=0;  
#1 a=0; b=1;
```



```
$monitor("%b%b:%b%b",a,b,c,s);  
a=0; b=0;  
#1 a=0; b=1;
```

Waveform: timescale

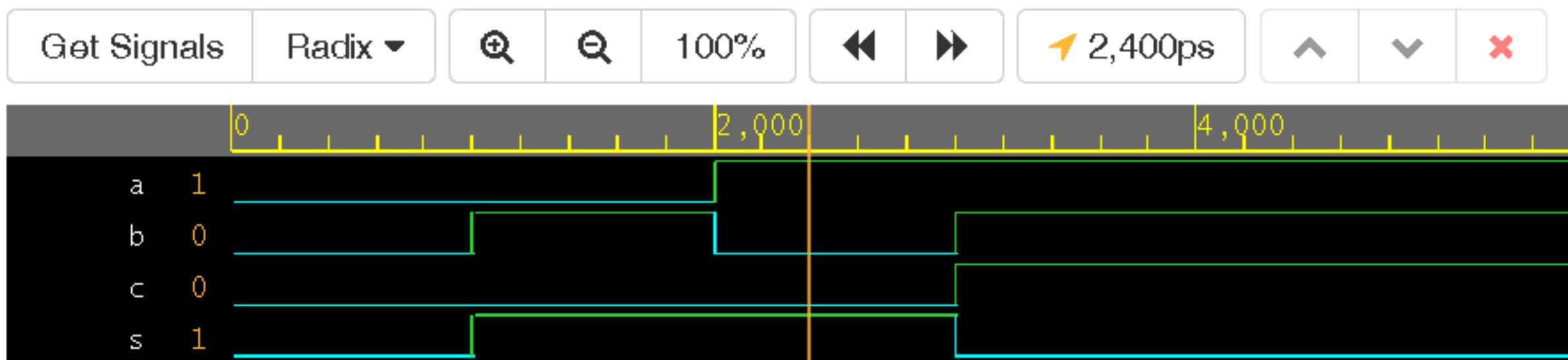
notice that values at 2ns are values on the right means values for time > 2ns



Waveform: timescale precision

We can decrease the precision to analyze in detail

```
//testbench  
`timescale 1 ns / 100 ps
```



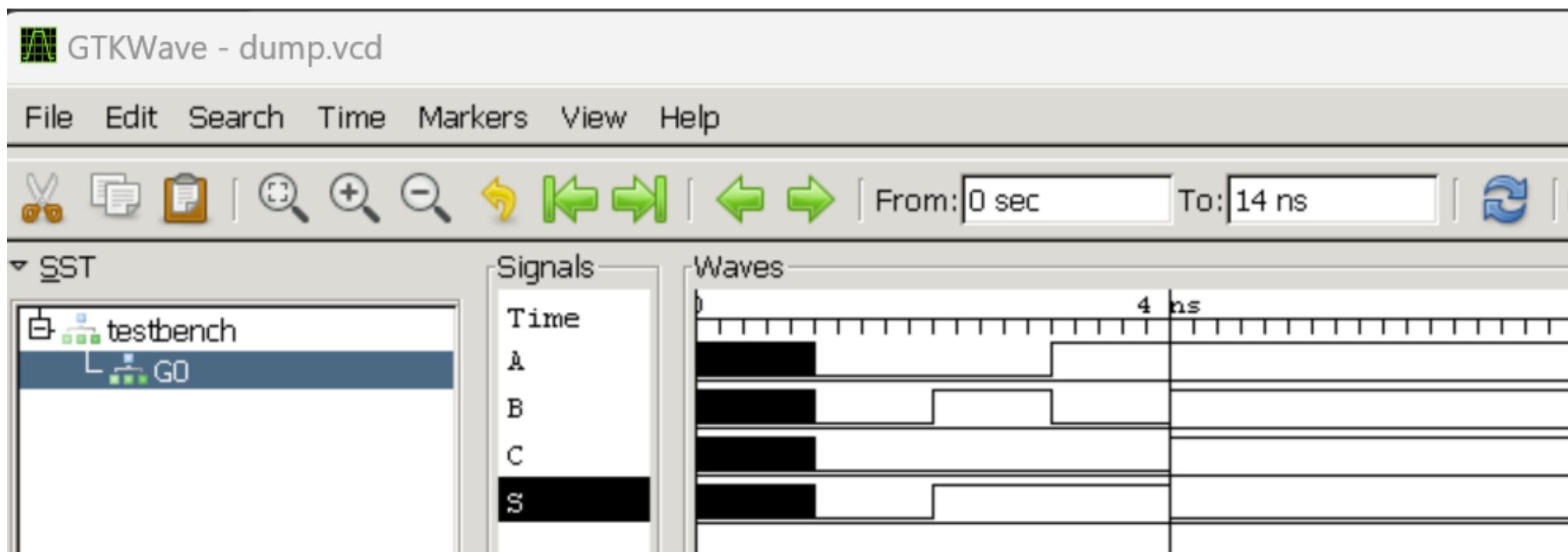
Verilog Test Bench: iverilog

- we can run simulation in local computer using **iverilog** and **gtkwave**
- save design and testbench files using a text editor (**notepad**)
- in command line:
- **iverilog -o halfadder.out halfadder.v halfadder_tb.v; vvp halfadder.out**
- will print:

```
PS D:\docs\ostim\digital\verilog_codes> iverilog -o halfadder.out halfadder.v halfadder_tb.v; vvp halfadder.out
VCD info: dumpfile dump.vcd opened for output.
ab:cs
xx:xx
00:00
01:01
10:01
11:10
halfadder_tb.v:16: $finish called at 14 (1ns)
```

Verilog Test Bench: gtkwave

- **in command line:**
- **gtkwave dump.vcd**
- **will start gtkwave with dump.vcd**



Verilog Test Bench: iverilog and gtkwave

- command line copy shown here

```
PS D:\docs\ostim\digital\EEE303> iverilog -o twogates.out twogates.v twogates_tb.v; vvp twogates.out
VCD info: dumpfile dump.vcd opened for output.
ab:cd
xx:xx
00:11
01:10
10:10
11:00
twogates_tb.v:16: $finish called at 14 (1ns)
PS D:\docs\ostim\digital\EEE303> gtkwave dump.vcd
```

Verilog : using VS Code Editor

The screenshot shows the VS Code interface with two files open:

- halfadder_tb.v**:

```
3 module testbench;
5   wire s,c;      //wire for outputs
6   //instantiation
7   halfadder G0 (.S(s),.C(c),.A(a),.B(b));
8   initial begin
9     $dumpfile("dump.vcd"); $dumpvars;
10    $display("ab:cs");
11    $monitor("%b%b:%b%b",a,b,c,s);
12    #1 a=0; b=0;
13    #1 a=0; b=1;
14    #1 a=1; b=0;
15    #1 a=1; b=1;
16    #10 $finish;
17  end // initial
18 endmodule
```
- halfadder.v**:

```
1 module halfadder (A,B,S,C);
2   input A,B;
3   output S,C;
4   xor M0 (S,A,B);
5   and M1 (C,A,B);
6   endmodule
7
```

A floating panel on the right displays:

- Icon: FPGA
- Text: Verilog-HDL/SystemVe... ⏲ 101ms
- Text: Verilog-HDL/SystemVerilog/Blues...
- Text: Masahiro Hiramori
- Icon: gear

Bottom navigation bar:

- PROBLEMS
- OUTPUT
- DEBUG CONSOLE
- TERMINAL**
- PORTS

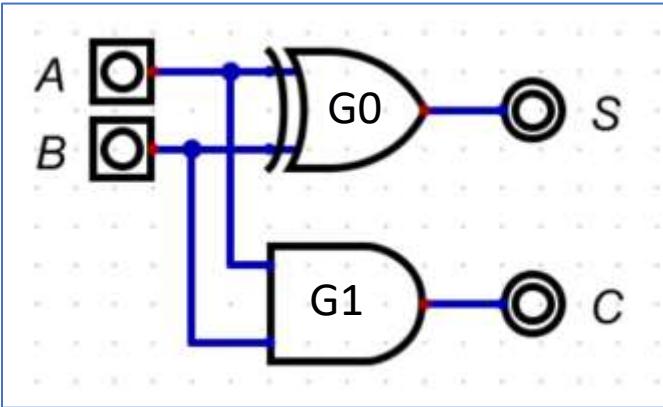
Terminal output:

```
[0] start time.
[18] end time.
PS D:\docs\ostim\digital\verilog_codes> iverilog -o halfadder.out halfadder.v halfadder_tb.v; vvp halfadder.out
VCD info: dumpfile dump.vcd opened for output.
```

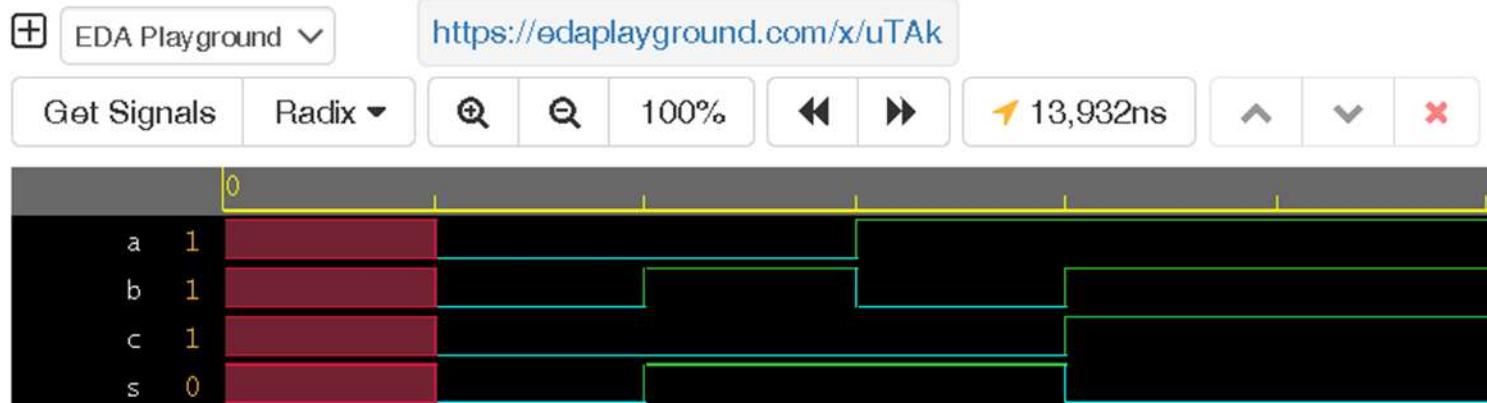
Verilog: Half Adder

- Using this **Gate-Level modeling** we can define any **combinatorial logic** circuit

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```



ab	:	cs
xx	:	xx
00	:	00
01	:	01
10	:	01
11	:	10



```
//testbench
`timescale 1 ns / 1 ns
module testbench;
reg a,b; //reg for inputs
wire s,c; //wire for outputs
//instantiation
halfadder G0 (.S(s),.C(c),.A(a),.B(b));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("ab:cs");
$monitor("%b%b:%b%b",a,b,c,s);
#1 a=0; b=0;
#1 a=0; b=1;
#1 a=1; b=0;
#1 a=1; b=1;
#10 $finish;
end // initial
endmodule
```

Why is Verification difficult?

- **How long would it take to test a 32-bit adder?**
 - In such an adder there are 64 inputs = 2^{64} possible inputs
 - That makes around 1.85×10^{19} possibilities
 - If you test one input in 1ns, you can test 10^9 inputs per second
 - or 8.64×10^{14} inputs per day
 - or 3.15×10^{17} inputs per year
 - we would still need **58.5 years** to test all possibilities
- **Brute force testing is not feasible for all circuits, we need alternatives**

Simple Testbench

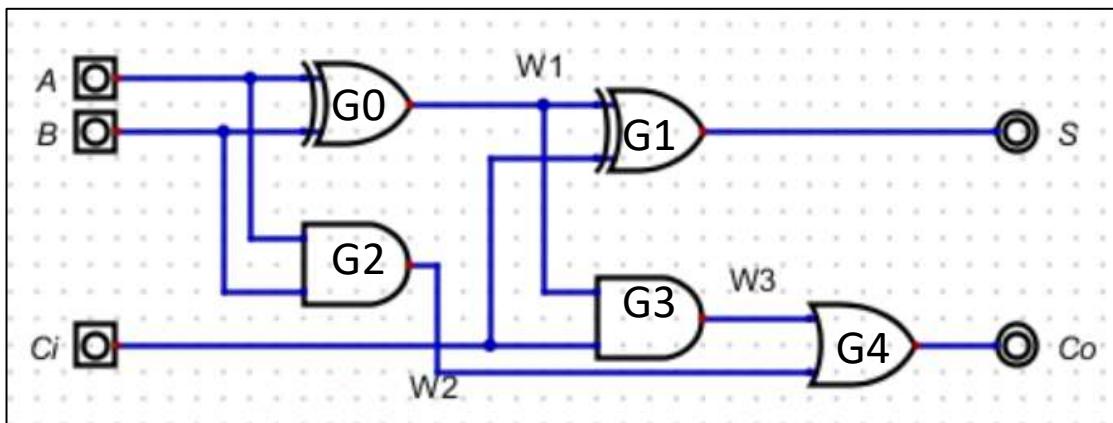
```
module testbench1(); // Testbench has no inputs, outputs
    reg a, b, c;      // Will be assigned in initial block
    wire y;

    // instantiate device under test
    sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );d

    // apply inputs one at a time
    initial begin          // sequential block
        a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
        c = 1; #10;           // apply inputs, wait 10ns
        b = 1; c = 0; #10;     // etc .. etc..
        c = 1; #10;
        a = 1; b = 0; c = 0; #10;
    end
endmodule
```

Complex Combinatorial Circuits: Full Adder

- Using this methodology we can define any **combinatorial logic** circuit
- give names to the gates G0, G1, etc.
- give wire names W1, W2, etc. to the intermediate connection points
- Sunday puzzle sudoku ☺



```
//full adder
module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;

    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);
    and G2 (W2,A,B);
    and G3 (W3,Ci,W1);
    or G4 (Co,W2,W3);
endmodule
```

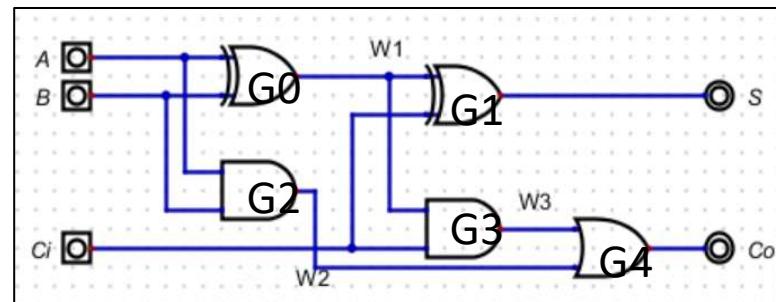
Full Adder

```
//full adder
module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;
    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);
    and G2 (W2,A,B);
    and G3 (W3,Ci,W1);
    or G4 (Co,W2,W3);
endmodule
```

Table 4.4
Full Adder

x	y	z	c	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

000:00
001:01
010:01
011:10
100:01
101:10
110:10
111:11



```
//testbench
`timescale 1 ns / 1 ns
module tb_fulladder;
    reg a,b,ci; //reg for inputs
    wire s,co; //wire for outputs
    //instantiation
    fulladder M0 (.S(s),.Co(co),.A(a),.B(b),.Ci(ci));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("abci:sco");
        $monitor("%b%b%b:%b%b",a,b,ci,s,co);
        #1 a=0; b=0; ci=0;
        #1 a=0; b=0; ci=1;
        #1 a=0; b=1; ci=0;
        #1 a=0; b=1; ci=1;
        #1 a=1; b=0; ci=0;
        #1 a=1; b=0; ci=1;
        #1 a=1; b=1; ci=0;
        #1 a=1; b=1; ci=1;
        #10 $finish;
    end // initial
endmodule
```

Full Adder – Wrong Design

```
//full adder
module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;

    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);
    and G2 (W2,A,B);
    and G3 (W3,Ci,W2);
    or G4 (Co,W2,W3);
endmodule
```

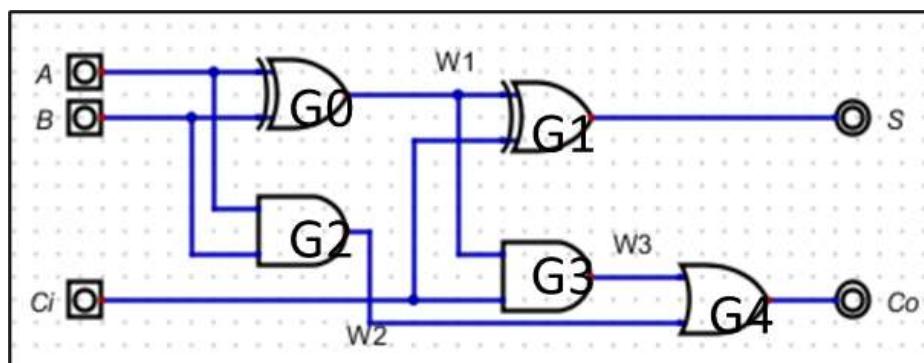
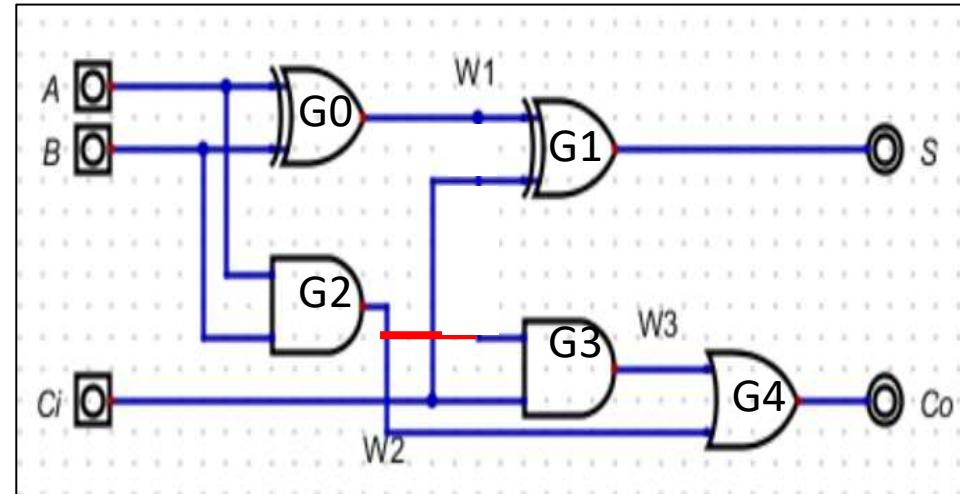


Table 4.4
Full Adder

x	y	z	c	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

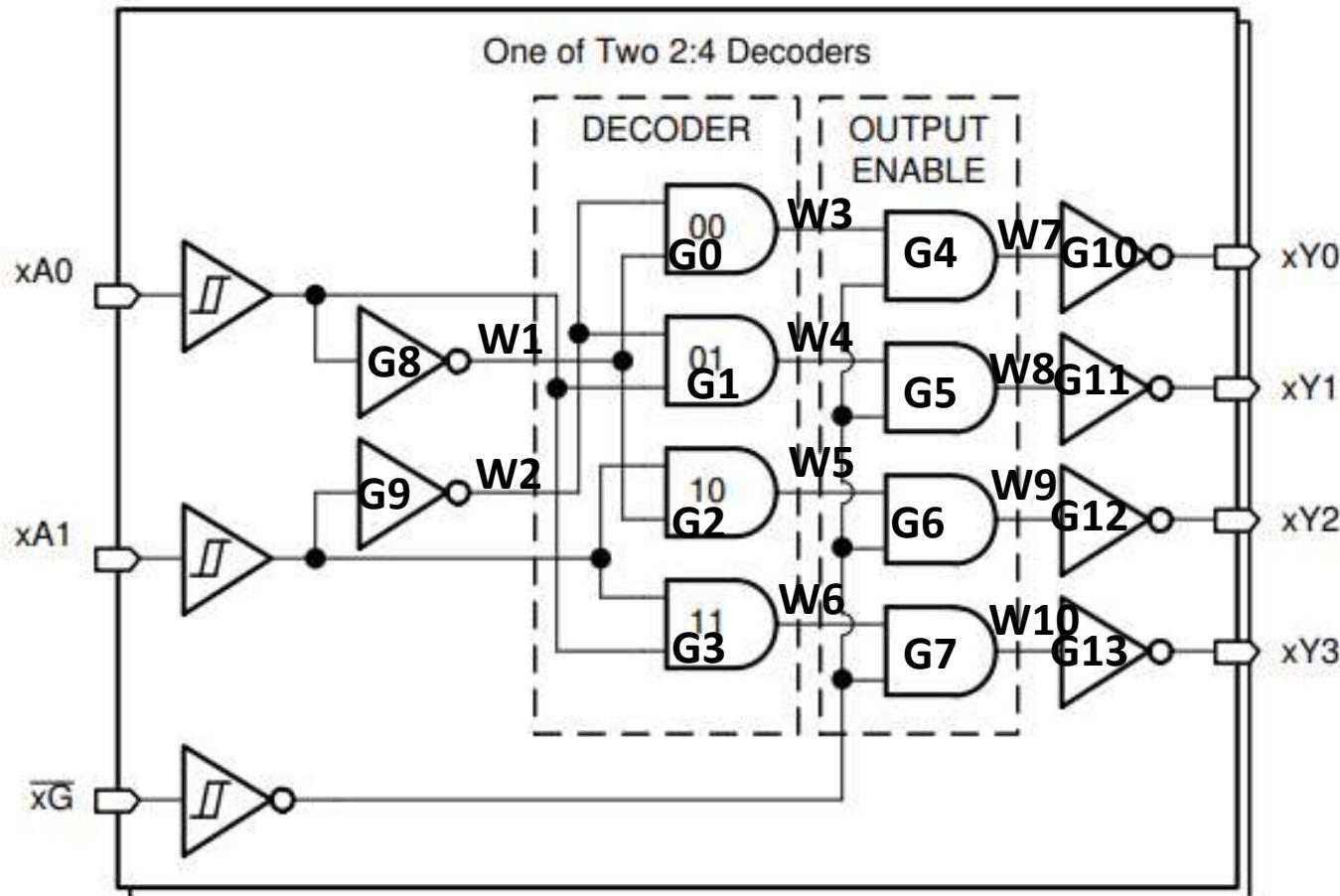
Full Adder – Wrong Design

Table 4.4
Full Adder

x	y	z	c	s	
0	0	0	0	0	000:00
0	0	1	0	1	001:01
0	1	0	0	1	010:01
0	1	1	1	0	011:00
1	0	0	0	1	100:01
1	0	1	1	0	101:00
1	1	0	1	0	110:10
1	1	1	1	1	111:11

simulation output and truth table outputs
are not the same so there is something
WRONG

```
//testbench
`timescale 1 ns / 1 ns
module tb_fulladder;
reg a,b,ci; //reg for inputs
wire s,co; //wire for outputs
//instantiation
fulladder M0 (.S(s),.Co(co),.A(a),.B(b),.Ci(ci));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("abci:sco");
$monitor("%b%b%b:%b%b",a,b,ci,s,co);
#1 a=0; b=0; ci=0;
#1 a=0; b=0; ci=1;
#1 a=0; b=1; ci=0;
#1 a=0; b=1; ci=1;
#1 a=1; b=0; ci=0;
#1 a=1; b=0; ci=1;
#1 a=1; b=1; ci=0;
#1 a=1; b=1; ci=1;
#10 $finish;
end // initial
endmodule
```

**SN74HCS139-Q1 Automotive Qualified Dual 2- to 4-Line Decoders/Demultiplexers
with Schmitt-Trigger Inputs**


```
//2 to 4 line decoder
module decoder(
    output Y0,Y1,Y2,Y3,
    input A0,A1,G
);
    wire W1,W2,W3,W4,W5,
    W6,W7,W8,W9,W10;
    and G0 (W3,W2,W1);
    and G1 (W4,W2,A0);
    and G2 (W5,A1,W1);
    and G3 (W6,A1,A0);
    and G4 (W7,W3,G);
    and G5 (W8,W4,G);
    and G6 (W9,W5,G);
    and G7 (W10,W6,G);
    not G8 (W1,A0);
    not G9 (W2,A1);
    not G10 (Y0,W7);
    not G11 (Y1,W8);
    not G12 (Y2,W9);
    not G13 (Y3,W10);
endmodule
```

Figure 8-1. Logic Diagram (Positive Logic) for SN74HCS139-Q1 -Q1

SN74HCS139-Q1 Automotive Qualified Dual 2- to 4-Line Decoders/Demultiplexers with Schmitt-Trigger Inputs

INPUTS ⁽¹⁾			OUTPUTS			
nG	nA1	nA0	nY0	nY1	nY2	nY3
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L
H	X	X	H	H	H	H

If it's an active-low pin, you must "pull" that pin LOW by connecting it to ground. For an active high pin, you connect it to your HIGH voltage (usually 3.3V/5V).

The CE pin would need to be pulled to GND in order for the chip to become enabled. If, however, the CE pin doesn't have a line over it, then it is active high, and it needs to be pulled HIGH in order to enable the pin.

GA1A0	:	Y0Y1Y2Y3					
x	x	x	:	x	x	x	x
1	0	0	:	0	1	1	1
1	0	1	:	1	0	1	1
1	1	0	:	1	1	0	1
1	1	1	:	1	1	1	0
0	0	0	:	1	1	1	1
0	0	1	:	1	1	1	1

```
//testbench
`timescale 1 ns / 1 ns
module testbench;
    wire y0,y1,y2,y3; //wire for outputs
    reg a0,a1,g;      //reg for inputs
    //instantiation
    decoder M0
    (.Y0(y0),.Y1(y1),.Y2(y2),.Y3(y3),
     .A0(a0),.A1(a1),.G(g));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("GA1A0 : Y0Y1Y2Y3");
        $monitor("%b %b %b : %b %b %b %b"
                 ,g,a1,a0,y0,y1,y2,y3);
        #1 g=1; a1=0; a0=0;
        #1 g=1; a1=0; a0=1;
        #1 g=1; a1=1; a0=0;
        #1 g=1; a1=1; a0=1;
        #1 g=0; a1=0; a0=0;
        #1 g=0; a1=0; a0=1;
        #10 $finish;
    end // initial
endmodule
```

Self-checking Testbench

```
module testbench2();
    reg a, b, c;
    wire y;

    // instantiate device under test
    sillyfunction dut(.a(a), .b(b), .c(c), .y(y));

    // apply inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10; // apply input, wait
        if (y !== 1) $display("000 failed."); // check
        c = 1; #10; // apply input, wait
        if (y !== 0) $display("001 failed."); // check
        b = 1; c = 0; #10; // etc.. etc..
        if (y !== 0) $display("010 failed."); // check
    end
endmodule
```

Self-checking Testbench

```
self checking
testbench:
ab:cs:test
00:00:1
01:01:1
10:01:1
11:10:1
```

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
truth table
00:00
01:10
10:10
11:01
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_halfadder ();
    reg a,b,test; //reg for inputs
    wire s,c; //wire for outputs
//instantiation
halfadder M0 (.S(s),.C(c),.A(a),.B(b));
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("self checking testbench:");
    $display("ab:cs:test");

    a=0; b=0; #1 test=(c==0)&&(s==0);
    $display("%b%b:%b%b:%b",a,b,c,s,test);

    a=0; b=1; #1 test=(c==0)&&(s==1);
    $display("%b%b:%b%b:%b",a,b,c,s,test);

    a=1; b=0; #1 test=(c==0)&&(s==1);
    $display("%b%b:%b%b:%b",a,b,c,s,test);

    a=1; b=1; #1 test=(c==1)&&(s==0);
    $display("%b%b:%b%b:%b",a,b,c,s,test);

    #10 $finish;
end // initial
endmodule
```

Verilog: Self-checking Testbench

```
self checking  
testbench:  
ab:cs:test  
ab=11 failed
```

truth table
00:00
01:10
10:10
11:01

```
//testbench  
'timescale 1 ns / 1 ns  
module tb_halfadder();  
    reg a,b,test; //reg for inputs  
    wire s,c; //wire for outputs  
    //instantiation  
    halfadder M0 (.S(s),.C(c),.A(a),.B(b));  
    initial begin  
        $dumpfile("dump.vcd"); $dumpvars;  
        $display("self checking testbench:");  
        $display("ab:cs:test");  
  
        a=0; b=0; #10  
        if ((c==0)&&(s==0) !=1) $display("ab=00 failed");  
  
        a=0; b=1; #10  
        if ((c==0)&&(s==1) !=1) $display("ab=01 failed");  
  
        a=1; b=0; #10  
        if ((c==0)&&(s==1) !=1) $display("ab=10 failed");  
  
        a=1; b=1; #10  
        if ((c==1)&&(s==0) !=1) $display("ab=11 failed");  
  
        #20 $finish;  
    end // initial  
endmodule
```

```
//half adder  
module halfadder(  
    output S,C,  
    input A,B  
);  
    xor G0 (S,A,B);  
    and G1 (C,A,B);  
endmodule
```

if (expression) statement or statement_group
Executes the next statement or statement group if the expression evaluates as true.

Testbench with Testvectors

- **The more elaborate testbench**
- **Write testvector file: inputs and expected outputs**
 - Usually can use a high-level model (golden model) to produce the 'correct' input output vectors
- **Testbench:**
 - Generate clock for assigning inputs, reading outputs
 - Read testvectors file into array
 - Assign inputs, get expected outputs from DUT
 - Compare outputs to expected outputs and report errors

Testbench with Testvectors

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

Table 4.3
Half Adder

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
'timescale 1 ns / 1 ns
module halfadder_tb;
// Inputs to the DUT (Device Under Test)
reg A,B;
// Outputs from the DUT
wire C,S; // Carry, Sum
// Instantiate the DUT
halfadder dut ( .C(C), .S(S), .A(A), .B(B) );
// Test vector storage
reg [3:0] test_vectors [0:3]; // 4 bits wide, 4 test cases
integer i;
initial begin
    // Read test vectors from file
    $readmemb("testvectors.txt", test_vectors);
    // Apply test vectors
    for (i = 0; i < 4; i = i + 1) begin
        // Assign inputs from the test vectors
        {A, B} = test_vectors[i][3:2];
        // Wait for the output to stabilize
        #10;
        // Check the results
        if (C != test_vectors[i][1] || S != test_vectors[i][0]) begin
            $display("Test failed for vector %0d: A=%b, B=%b, Expected_C=%b, Expected_S=%b",
                    Got_C=%b, Got_S=%b",
                    i, A, B, test_vectors[i][1], test_vectors[i][0], C, S);
        end else begin
            $display("Test passed for vector %0d: A=%b, B=%b, C=%b, S=%b",
                    i, A, B, C, S);
        end //if else
    end //for
    $finish;
end //initial
endmodule
```

for (*initial_assignment*; *expression*; *step_assignment*)
statement or statement group

```
$readmemb ("file_name", variable_array, start_address, end_address) ;
$readmemh ("file_name", variable_array, start_address, end_address) ;
Loads the contents of a file into a memory array. The file must be an ASCII
file with values represented in binary ($readmemb) or hex ($readmemh).
Start and end address are optional.
```

Testbench with Testvectors

reg	a variable of any bit size; unsigned unless explicitly declared as signed
integer	a signed 32-bit variable

```
for (initial_assignment; expression; step_assignment)  
statement or statement group
```

6.6 Reading and Writing Arrays

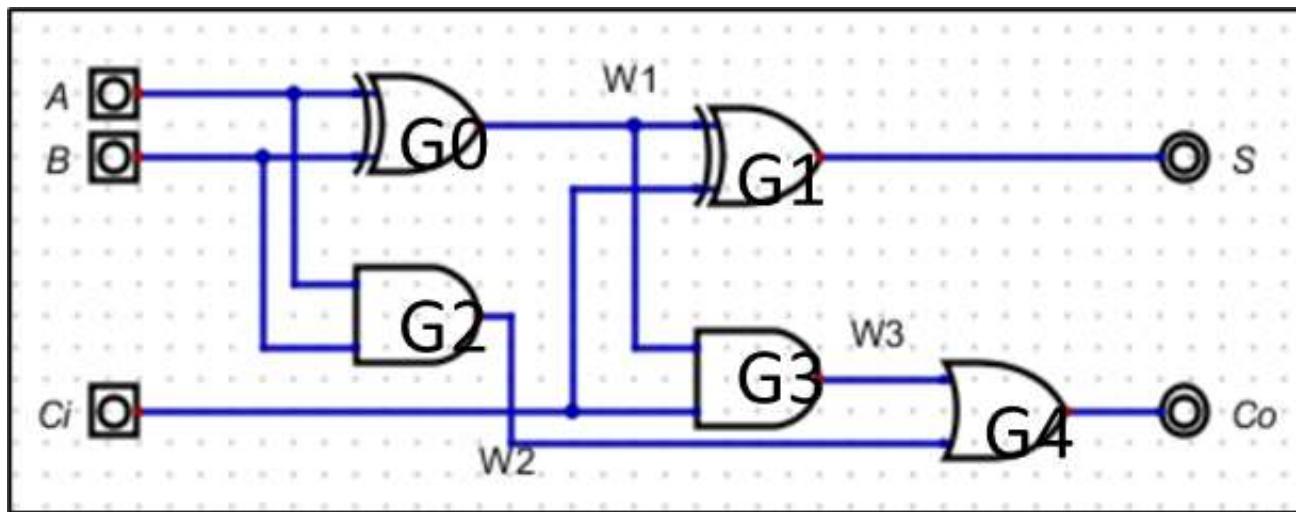
- Only one element at a time within an array can be read from or written to.
- A memory array (a one-dimensional array of reg variables) can be loaded using the \$readmemb, \$readmemh, \$sreadmemb, or \$sreadmemh system tasks.

{}	{m, n}	concatenate m to n, creating a larger vector
----	--------	--

```
// Test vector storage  
reg [3:0] test_vectors [0:3]; // 4 bits wide, 4 test cases  
integer i;  
  
// Read test vectors from file  
$readmemb("testvectors.txt", test_vectors);  
// Apply test vectors  
for (i = 0; i < 4; i = i + 1) begin  
    // Assign inputs from the test vectors  
    {A, B} = test_vectors[i][3:2];  
  
    if (C != test_vectors[i][1] || S != test_vectors[i][0]) begin  
        $display("Test failed for vector %0d: A=%b, B=%b,  
Expected_C=%b, Expected_S=%b, Got_C=%b, Got_S=%b",  
i, A, B, test_vectors[i][1], test_vectors[i][0], C, S);  
    end else begin  
        $display("Test passed for vector %0d: A=%b, B=%b, C=%b,  
S=%b",  
i, A, B, C, S);
```

Hierarchical Modeling: Module In Module

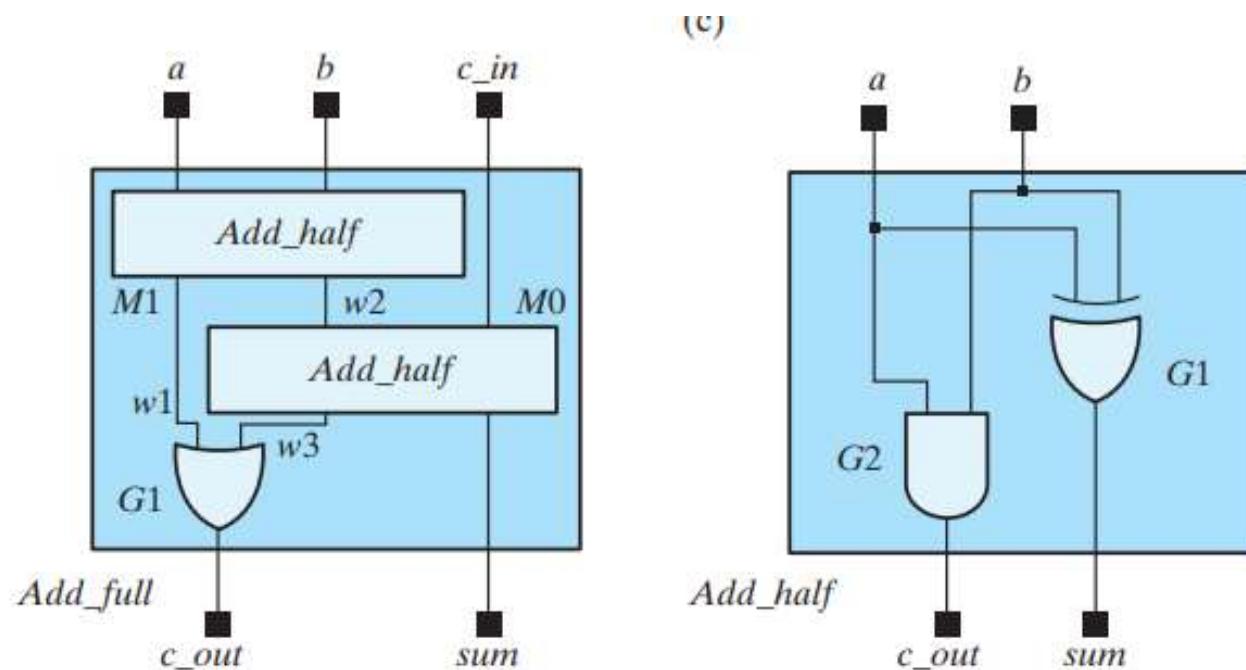
- full adder without hierachical design



```
//full adder
module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;
    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);
    and G2 (W2,A,B);
    and G3 (W3,Ci,W1);
    or G4 (Co,W2,W3);
endmodule
```

Hierarchical Modeling: Module In Module

- built-in primitives actually modules instantiated inside modules
- similarly we can instantiate modules defined inside another module
- This is called as **hierarchical modeling**



```
//full adder
module fulladder (
    input A,B,Ci,
    output S,Co);
    wire W1,W2,W3;

halfadder M0 (.A(W2),.B(Ci),.S(S),.C(W3));
halfadder M1 (.Ali(A),.B(B),.S(W2),.C(W1));
or G1 (Co,W1,W3);
endmodule //fulladder

module halfadder (
    input Ali,B,
    output S,C);
    xor M0 (S,A,B);
    and M1 (C,A,B);
endmodule //halfadder
```

Hierarchical Modeling: Module In Module

```
//full adder
module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;

    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);
    and G2 (W2,A,B);
    and G3 (W3,Ci,W1);
    or G4 (Co,W2,W3);
endmodule
```

```
//full adder
module fulladder (
    input A,B,Ci,
    output S,Co);

    wire W1,W2,W3;
    halfadder M0 (.A(W2),.B(Ci),.S(S),.C(W3));
    halfadder M1 (.A(A),.B(B),.S(W2),.C(W1));
    or G1 (Co,W1,W3);
endmodule //fulladder

module halfadder (
    input A,B,
    output S,C);
    xor M0 (S,A,B);
    and M1 (C,A,B);
endmodule //halfadder
```

Hierarchical Modeling: 4-bit Adder

- built-in primitives actually modules instantiated inside modules
- similarly we can instantiate modules defined inside another module
- This is called as **hierarchical modeling**

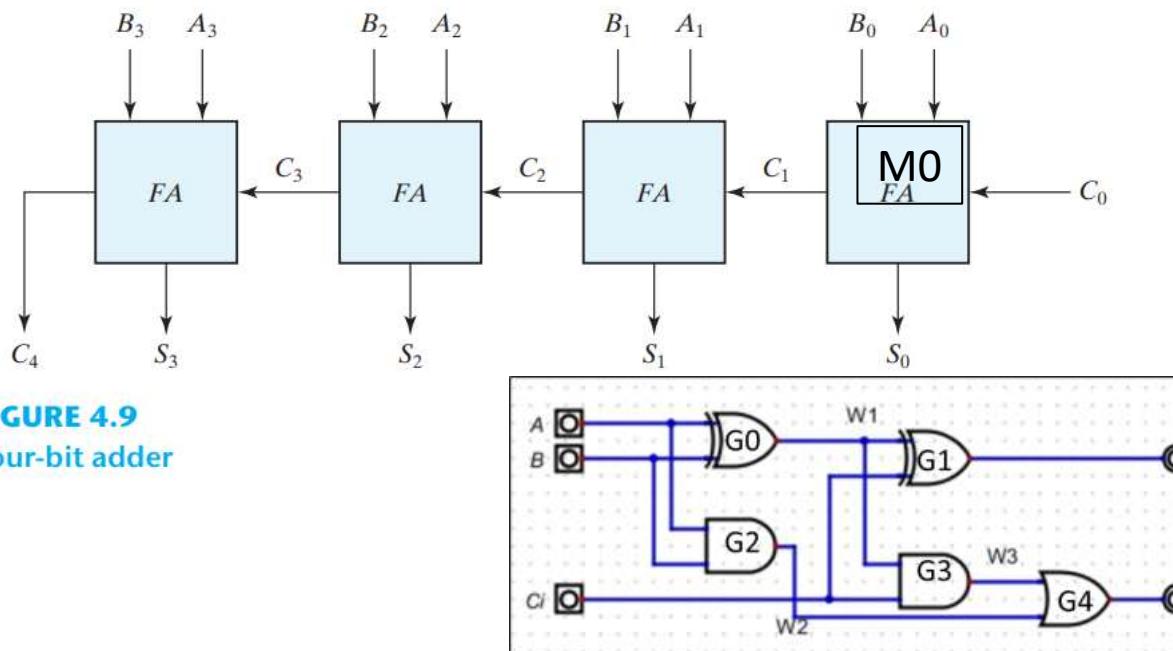


FIGURE 4.9
Four-bit adder

```
//full adder
module fourbitadder(
    output S0,S1,S2,S3,C4,
    input A0,A1,A2,A3,B0,B1,B2,B3,C0);
    wire C1,C2,C3;
    fulladder M0 (.S(S0),.Co(C1),.A(A0),.B(B0),.Ci(C0));
    fulladder M1 (.S(S1),.Co(C2),.A(A1),.B(B1),.Ci(C1));
    fulladder M2 (.S(S2),.Co(C3),.A(A2),.B(B2),.Ci(C2));
    fulladder M3 (.S(S3),.Co(C4),.A(A3),.B(B3),.Ci(C3));
endmodule

module fulladder(
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;
    xor G0 (W1,A,B);
    xor G1 (S,W1,Ci);
    and G2 (W2,A,B);
    and G3 (W3,Ci,W1);
    or G4 (Co,W2,W3);
endmodule
```

Hierarchical Modeling: 4-bit Adder

- built-in primitives actually modules instantiated inside modules
- similarly we can instantiate modules defined inside another module
- This is called as **hierarchical modeling**

```
a3a2a1a0+b3b2b1b0+c0=c4
s3s2s1s0
xxxx+xxxx+x=x  xxxx
0100+0010+0=0  0110
1100+0011+0=0  1111
1110+1010+0=1  1000
```

```
//testbench
`timescale 1 ns / 1 ns
module tb ();
    reg a0,a1,a2,a3,b0,b1,b2,b3,c0; //reg for inputs
    wire s0,s1,s2,s3,c4; //wire for outputs
    //instantiation
    fourbitadder G0 (.S0(s0),.S1(s1),.S2(s2),.S3(s3),.C4(c4),
                      .A0(a0),.A1(a1),.A2(a2),.A3(a3),
                      .B0(b0),.B1(b1),.B2(b2),.B3(b3),.C0(c0));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("a3a2a1a0+b3b2b1b0+c0=c4 s3s2s1s0");
        $monitor("%b%b%b%b%b+%b%b%b%b%b=%b %b%b%b%b",
                 a3,a2,a1,a0,b3,b2,b1,b0,c0,
                 c4,s3,s2,s1,s0);
        #1 a3=0;a2=1;a1=0;a0=0; b3=0;b2=0;b1=1;b0=0; c0=0;
        #1 a3=1;a2=1;a1=0;a0=0; b3=0;b2=0;b1=1;b0=1; c0=0;
        #1 a3=1;a2=1;a1=1;a0=0; b3=1;b2=0;b1=1;b0=0; c0=0;
        #10 $finish;
    end // initial
endmodule
```

Hierarchical Modeling:

- General practice for big projects define each module as a separate file. You can use include directive for this file and instantiate this module
- top.v is top level module, you dont need to use include directive and you can instantiate other modules inside other files

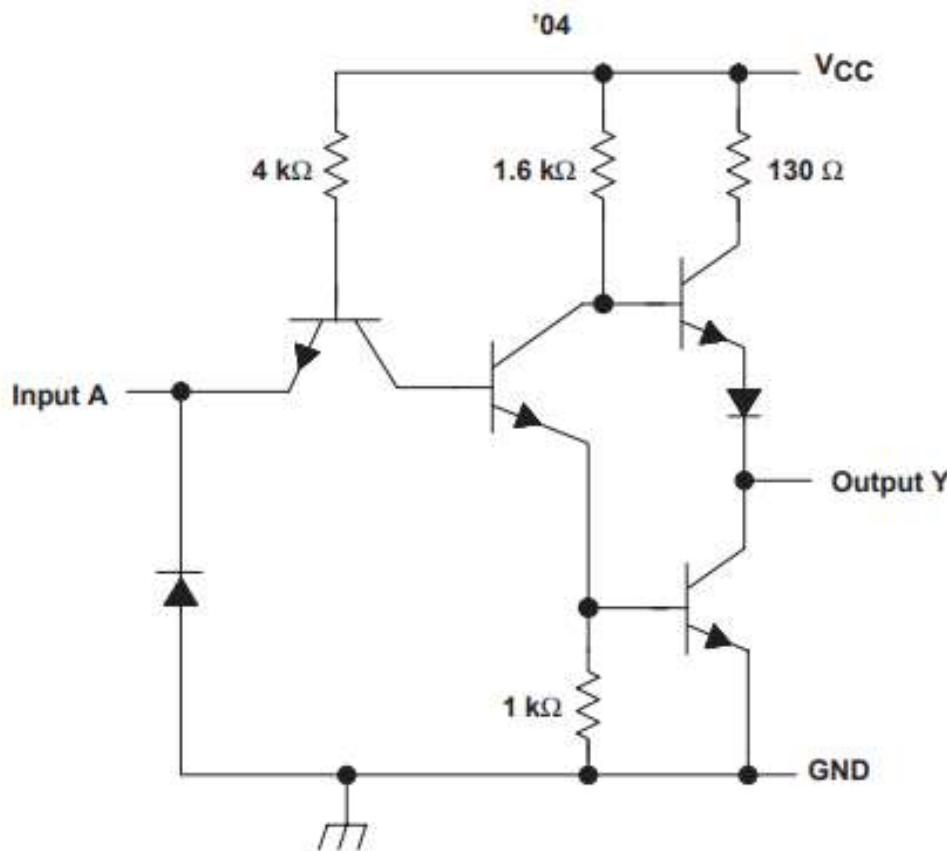
```
a3a2a1a0+b3b2b1b0+c0=c4  
s3s2s1s0  
xxxx+xxxx+x=x  xxxx  
0100+0010+0=0  0110  
1100+0011+0=0  1111  
1110+1010+0=1  1000
```

```
//full adder  
'include "fulladder.v"  
  
module fourbitadder(  
    output S0,S1,S2,S3,C4,  
    input A0,A1,A2,A3,B0,B1,B2,B3,C0);  
    wire C1,C2,C3;  
    fulladder M0 (.S(S0),.Co(C1),.A(A0),.B(B0),.Ci(C0));  
    fulladder M1 (.S(S1),.Co(C2),.A(A1),.B(B1),.Ci(C1));  
    fulladder M2 (.S(S2),.Co(C3),.A(A2),.B(B2),.Ci(C2));  
    fulladder M3 (.S(S3),.Co(C4),.A(A3),.B(B3),.Ci(C3));  
endmodule
```

```
module fulladder(  
    output S,Co,  
    input A,B,Ci  
>;  
    wire W1,W2,W3;  
    xor G0 (W1,A,B);  
    xor G1 (S,W1,Ci);  
    and G2 (W2,A,B);  
    and G3 (W3,Ci,W1);  
    or G4 (Co,W2,W3);  
endmodule
```

Timing in Combinatorial Circuits

- in reality logic gate outputs are not changing immediately
- there is a delay due to internal electronics, transistors, diodes etc

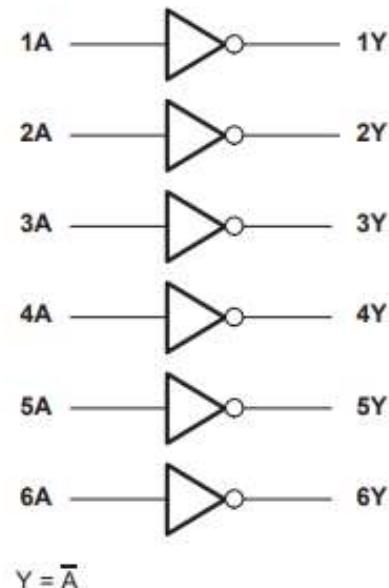
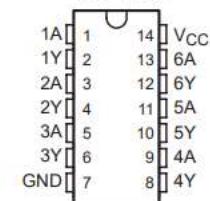


● Dependable Texas Instruments Quality and Reliability
description/ordering information
These devices contain six independent inverters.

**SN5404, SN54LS04, SN54S04,
SN7404, SN74LS04, SN74S04
HEX INVERTERS**

SDLS029C - DECEMBER 1983 - REVISED JANUARY 2004

SN5404 . . . J PACKAGE
SN54LS04, SN54S04 . . . J OR W PACKAGE
SN7404, SN74S04 . . . D, N, OR NS PACKAGE
SN74LS04 . . . D, DB, N, OR NS PACKAGE
(TOP VIEW)



Timing in Combinatorial Circuits

- in reality outputs are not changing immediately, there is a delay
- can we simulate delay ? YES

8.0 Primitive Instances

```
gate_type (drive_strength) #(delay) instance_name
    [instance_array_range] (terminal, terminal, ...);
```

- *delay* (optional) represents the propagation delay through a primitive. The default delay is zero. Integers or real numbers may be used.
- Separate delays for 1, 2 or 3 transitions may be specified.
 - Each transition may have a single delay or a min:typ:max delay range.

Delays	Transitions represented (in order)
1	all output transitions
2	rise, fall output transitions
3	rise, fall, turn-off output transitions (turn-off delay is the time for a tri-state primitive to transition to Z)

```
module inverter(
    output Y,
    input A
);
    not G0 (Y,A);
endmodule
```

```
`timescale 1 ns / 1 ns
module tb_halfadder ();
    reg a; //reg for inputs
    wire y; //wire for outputs
    //instantiation
    inverter G0 (.Y(y),.A(a));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("a:y");
        $monitor("%b:%b",a,y);
        #1 a=0;
        #1 a=1;
        #10 $finish;
    end // initial
endmodule
```

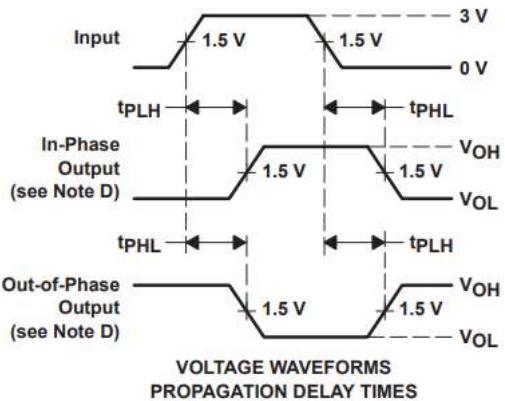
Timing in Combinatorial Circuits

SN5404, SN54LS04, SN54S04,
SN7404, SN74LS04, SN74S04
HEX INVERTERS

- can we simulate delay ? YES #3ns delay

switching characteristics, $V_{CC} = 5 \text{ V}$, $T_A = 25^\circ\text{C}$ (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN54S04 SN74S04			UNIT
				MIN	TYP	MAX	
t_{PLH}	A	Y	$R_L = 280 \Omega$, $C_L = 15 \text{ pF}$	3	4.5	5	ns
t_{PHL}				3	5		



```
'timescale 1 ns / 1 ns
module inverter(
    output Y,
    input A
);
    not #3 G0 (Y,A); //3ns rise&fall
endmodule
```

```
'timescale 1 ns / 1 ns
module testbench ();
    reg a; //reg for inputs
    wire y; //wire for outputs
    //instantiation
    inverter G0 (.Y(y),.A(a));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("a:y");
        $monitor("%b:%b",a,y);
        #1 a=0;
        #10 a=1;
        #10 a=0;
        #10 $finish;
    end // initial
endmodule
```



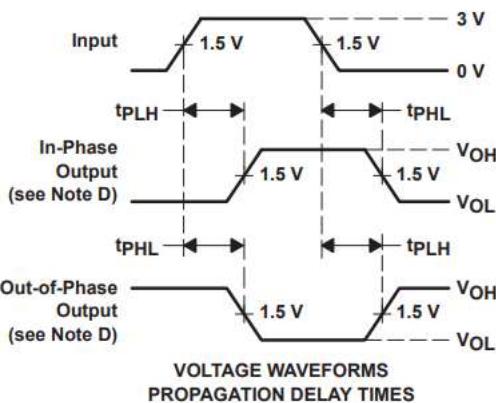
Timing in Combinatorial Circuits

SN5404, SN54LS04, SN54S04,
SN7404, SN74LS04, SN74S04
HEX INVERTERS

- can we simulate delay ? YES
- #1ns LH, #3ns HL delay of OUTPUT

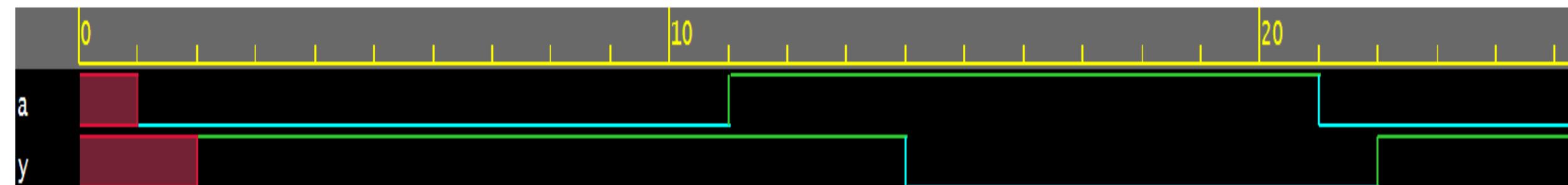
switching characteristics, $V_{CC} = 5$ V, $T_A = 25^\circ\text{C}$ (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN54S04			UNIT
				MIN	TYP	MAX	
t_{PLH}	A	Y	$R_L = 280 \Omega$, $C_L = 15 \text{ pF}$	3	4.5		
t_{PHL}				3	5		ns



```
'timescale 1 ns / 1 ns
module inverter(
    output Y,
    input A
);
not #(1,3) G0 (Y,A); //1ns rise&3ns fall delay
endmodule
```

```
'timescale 1 ns / 1 ns
module testbench ();
reg a; //reg for inputs
wire y; //wire for outputs
//instantiation
inverter G0 (.Y(y),.A(a));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("a:y");
$monitor("%b:%b",a,y);
#1 a=0;
#10 a=1;
#10 a=0;
#10 $finish;
end // initial
endmodule
```



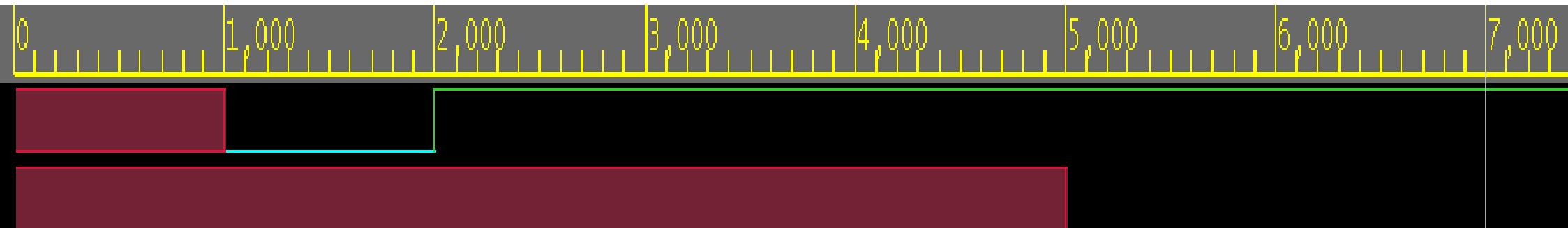
Timing in Combinatorial Circuits

SN5404, SN54LS04, SN54S04,
SN7404, SN74LS04, SN74S04
HEX INVERTERS

- If input changes faster than 3ns it is a problem
- last value a=1 will be affective $2+3 = 5\text{ns}$ we will see the output

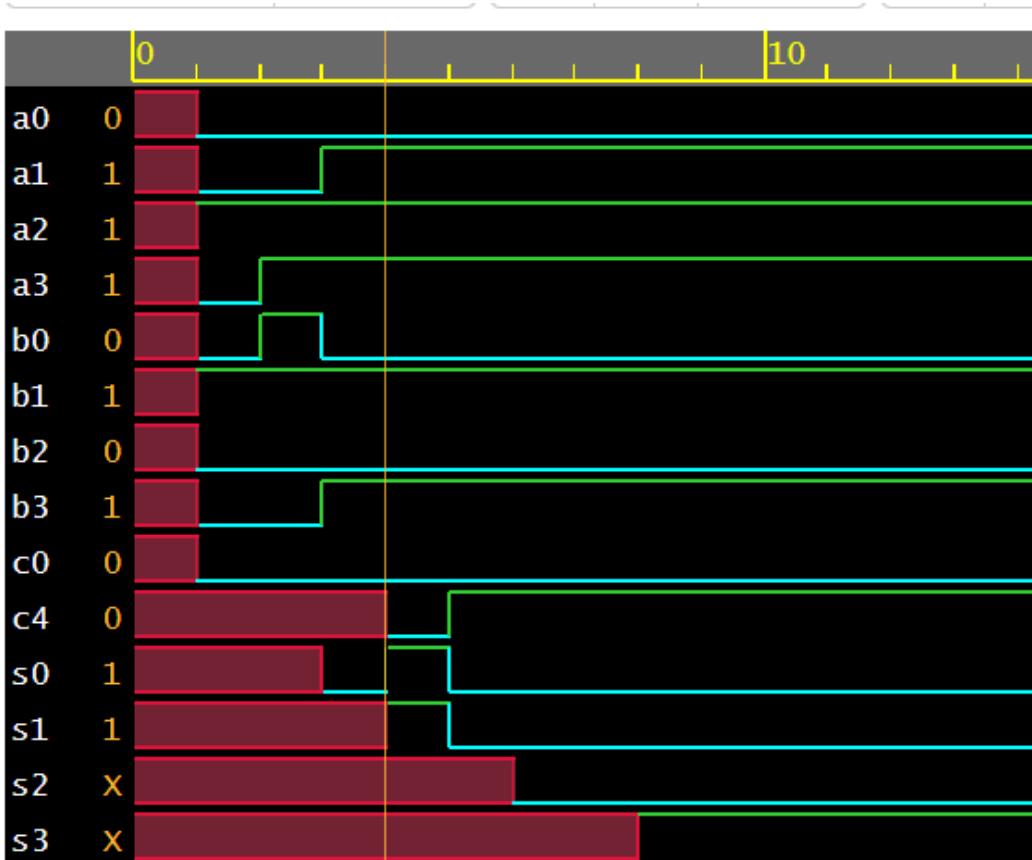
```
'timescale 1 ns / 1 ns
module inverter(
    output Y,
    input A
);
    not #3 G0 (Y,A); //1ns rise&3ns fall delay
endmodule
```

```
'timescale 1 ns / 1 ns
module testbench ();
    reg a; //reg for inputs
    wire y; //wire for outputs
    //instantiation
    inverter G0 (.Y(y),.A(a));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("a:y");
        $monitor("%b:%b",a,y);
        #1 a=0;
        #1 a=1;
        #10 $finish;
    end // initial
endmodule
```



Hierarchical Modeling: 4-bit Adder (delay)

- assume that each gate has 1ns delay
- delay of S3 becomes 8ns
- S3 changes value and becomes stable at 17ns



```
//full adder
`timescale 1 ns / 1 ns
module fourbitadder(
    output S0,S1,S2,S3,C4,
    input A0,A1,A2,A3,B0,B1,B2,B3,C0);
    wire C1,C2,C3;
    fulladder M0 (.S(S0),.Co(C1),.A(A0),.B(B0),.Ci(C0));
    fulladder M1 (.S(S1),.Co(C2),.A(A1),.B(B1),.Ci(C1));
    fulladder M2 (.S(S2),.Co(C3),.A(A2),.B(B2),.Ci(C2));
    fulladder M3 (.S(S3),.Co(C4),.A(A3),.B(B3),.Ci(C3));
endmodule

module fulladder (
    output S,Co,
    input A,B,Ci
);
    wire W1,W2,W3;
    xor #1 G0 (W1,A,B);
    xor #1 G1 (S,W1,Ci);
    and #1 G2 (W2,A,B);
    and #1 G3 (W3,Ci,W1);
    or #1 G4 (Co,W2,W3);
endmodule
```

Hierarchical Modeling: 4-bit Adder (delay)

- assume that each gate has 1ns delay
- delay of S3 becomes 8ns
- S0 changes value and becomes stable 01000000
- so input has to change slower than 17ns

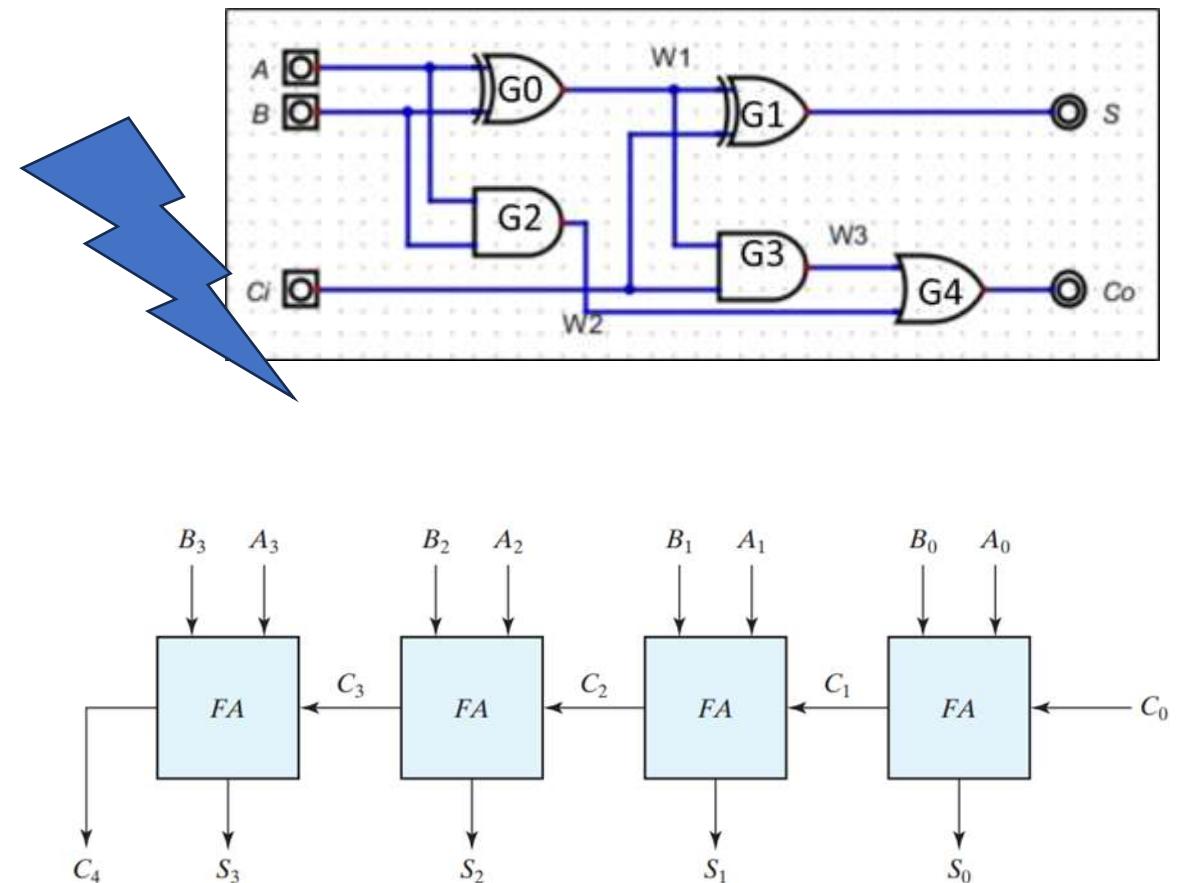
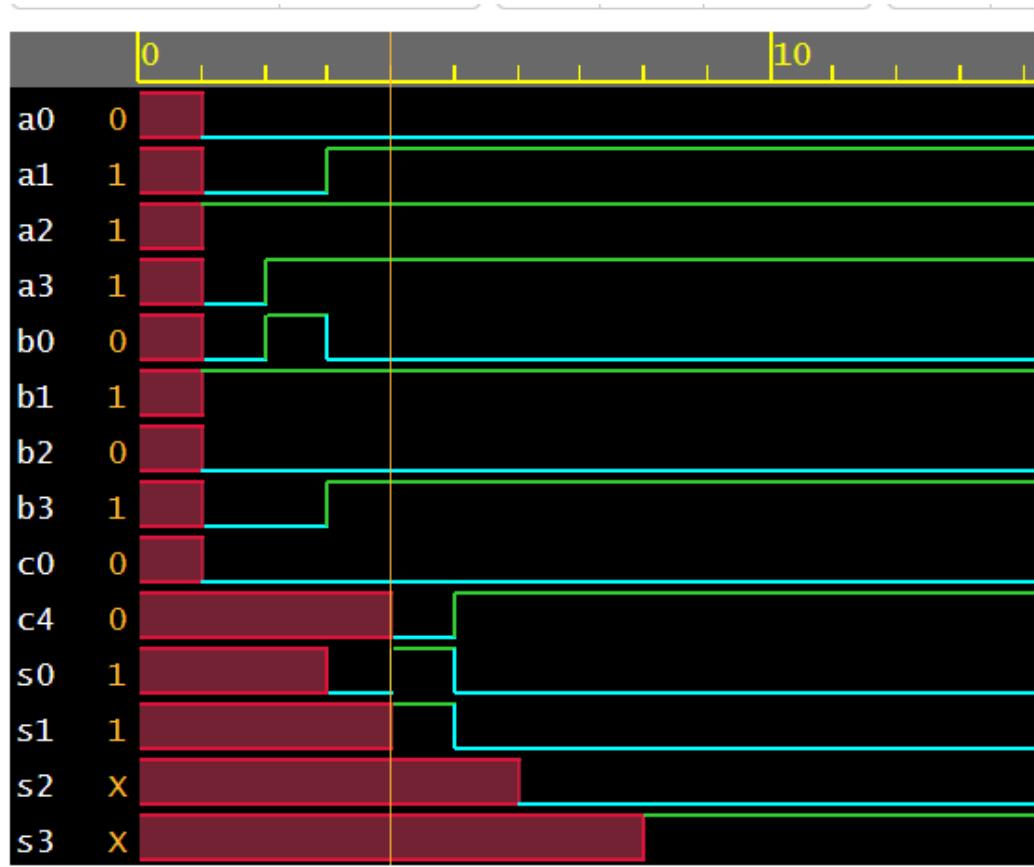
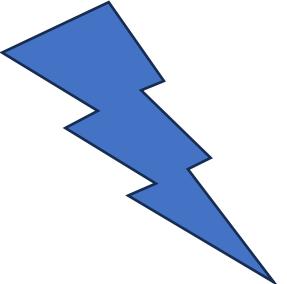


FIGURE 4.9
Four-bit adder

Hierarchical Modeling: 4-bit Adder (delay)

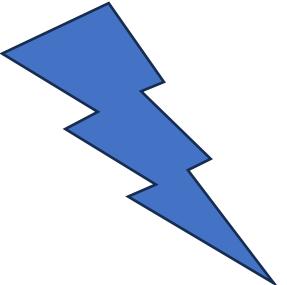


- assume that each gate has 1ns delay
- delay of S3 becomes 17ns
- so input has to change slower than 17ns

```
a3a2a1a0+b3b2b1b0+c0=c4  
s3s2s1s0  
  
xxxx+xxxx+x=x xxxx  
0100+0010+0=0 0110  
1100+0011+0=0 1111  
1110+1010+0=1 1000
```

0:xxxx+xxxx+x=x	xxxx
10:0100+0010+0=x	xxxx
12:0100+0010+0=x	xxx0
13:0100+0010+0=0	xx10
15:0100+0010+0=0	x110
17:0100+0010+0=0	0110
20:1100+0011+0=0	0110
22:1100+0011+0=0	1111
30:1110+1010+0=0	1111
32:1110+1010+0=1	0100
33:1110+1010+0=1	0000
35:1110+1010+0=1	1000

Hierarchical Modeling: 4-bit Adder (delay)



- so input has to change slower than max delay

**SN54F283, SN74F283
4-BIT BINARY FULL ADDERS
WITH FAST CARRY**

switching characteristics (see Note 2)

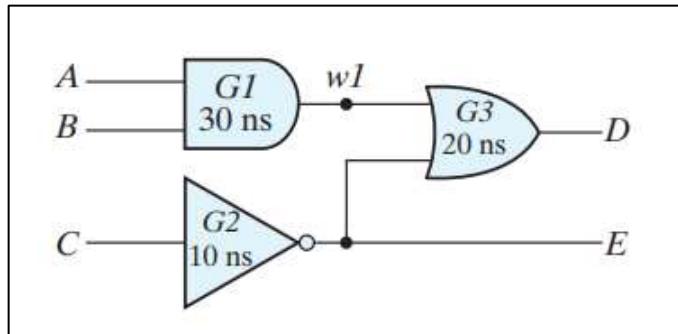
PARAMETER	FROM (INPUT)	TO (OUTPUT)	$V_{CC} = 5 \text{ V}$, $C_L = 50 \text{ pF}$, $R_L = 500 \Omega$, $T_A = 25^\circ\text{C}$			$V_{CC} = 4.5 \text{ V to } 5.5 \text{ V}$, $C_L = 50 \text{ pF}$, $R_L = 500 \Omega$, $T_A = \text{MIN to MAX}^{\$}$			UNIT	
			'F283			SN54F283		SN74F283		
			MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{PLH}	C0	Σ	2.7	6.6	9.5	2.7	14	2.7	10.5	ns
t_{PHL}			3.2	6.6	9.5	3.2	14	3.2	10.5	
t_{PLH}	A or B	Σ	3.2	6.6	9.5	3.2	14	3.2	10.5	ns
t_{PHL}			2.7	6.6	9.5	2.7	14	2.7	10.5	
t_{PLH}	C0	C4	2.7	5.3	7.5	2.7	10.5	2.7	8.5	ns
t_{PHL}			2.2	5	7	2.2	10	2.2	8	
t_{PLH}	A or B	C4	2.7	5.3	7.5	2.7	10.5	2.7	8.5	ns
t_{PHL}			2.2	4.9	7	2.2	10	2.2	8	

^{\$} For conditions shown as MIN or MAX, use the appropriate value specified under recommended operating conditions.

NOTE 2: Load circuits and waveforms are shown in Section 1.

Output Glitches

- Verilog can be used also to test propagation delay problems of the logic circuits such as output glitches
- Notice that D value oscillates from 1 to 0 and to 1 again (glitch)



```
'timescale 1 ns / 1 ns
module andor(
    input A, B, C,
    output D, E);
    wire w1;
    and #30 G1 (w1, A, B);
    not #10 G2 (E, C);
    or #20 G3 (D, w1, E);
endmodule
```

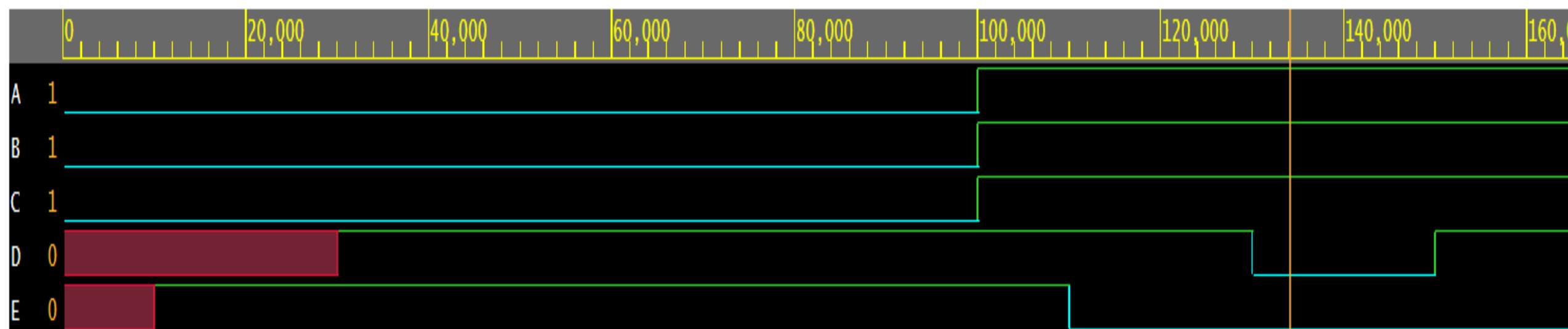
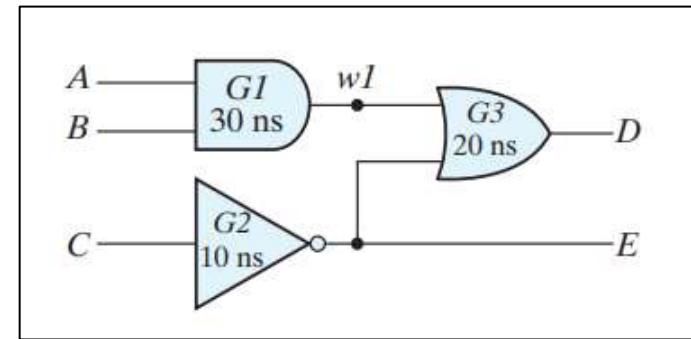
time	A	B	C	:	D	E
0	0	0	0	:	x	x
10	0	0	0	:	x	1
30	0	0	0	:	1	1
100	1	1	1	:	1	1
110	1	1	1	:	1	0
130	1	1	1	:	0	0
150	1	1	1	:	1	0

```
//testbench
`timescale 1 ns / 1 ns
module testbench ();
reg A,B,C;
wire D,E;
andor M0 (.A(A),.B(B),.C(C),.D(D),.E(E));
initial begin
$dumpfile("dump.vcd");
$dumpvars;
$display("\t\time A B C : D E");
$monitor("%d %b %b %b : %b %b",$time,A,B,C,D,E);
A=0; B=0; C=0;
#100 A=1; B=1; C=1;
#200 $finish;
end //initial
endmodule
```

Output Glitches

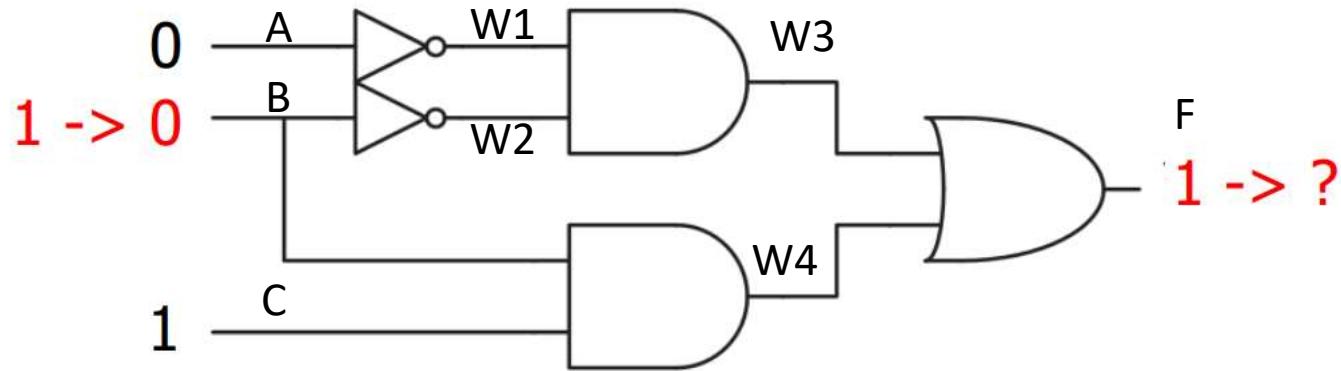
- Here we can observe timing problems
- Follow the graph after A=1; B=1; C=1;
- 100ns: w1=0; D=1; E=1;
- 110ns: E becomes 0 after 10ns
- 130ns: w1 was still 0, thus D becomes 0
- 150ns: w1 turns to 1 makes D=1 after 20ns and stabilize

time	A	B	C	:	D	E
0	0	0	0	:	x	x
10	0	0	0	:	x	1
30	0	0	0	:	1	1
100	1	1	1	:	1	1
110	1	1	1	:	1	0
130	1	1	1	:	0	0
150	1	1	1	:	1	0



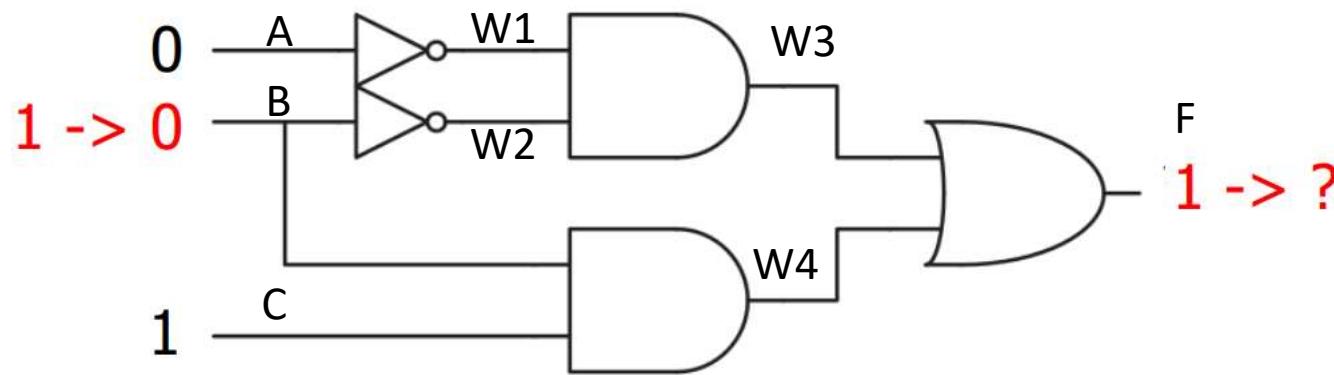
Output Glitches

*Onur Mutlu lecture notes



```
'timescale 1 ns / 1 ns
module glitch(
    output F,
    input A,B,C
);
    wire W1,W2,W3,W4;
    not #3 G0 (W1,A);
    not #3 G1 (W2,B);
    and #3 G2 (W3,W1,W2);
    and #3 G3 (W4,B,C);
    or #3 G4 (F,W3,W4);
endmodule
```

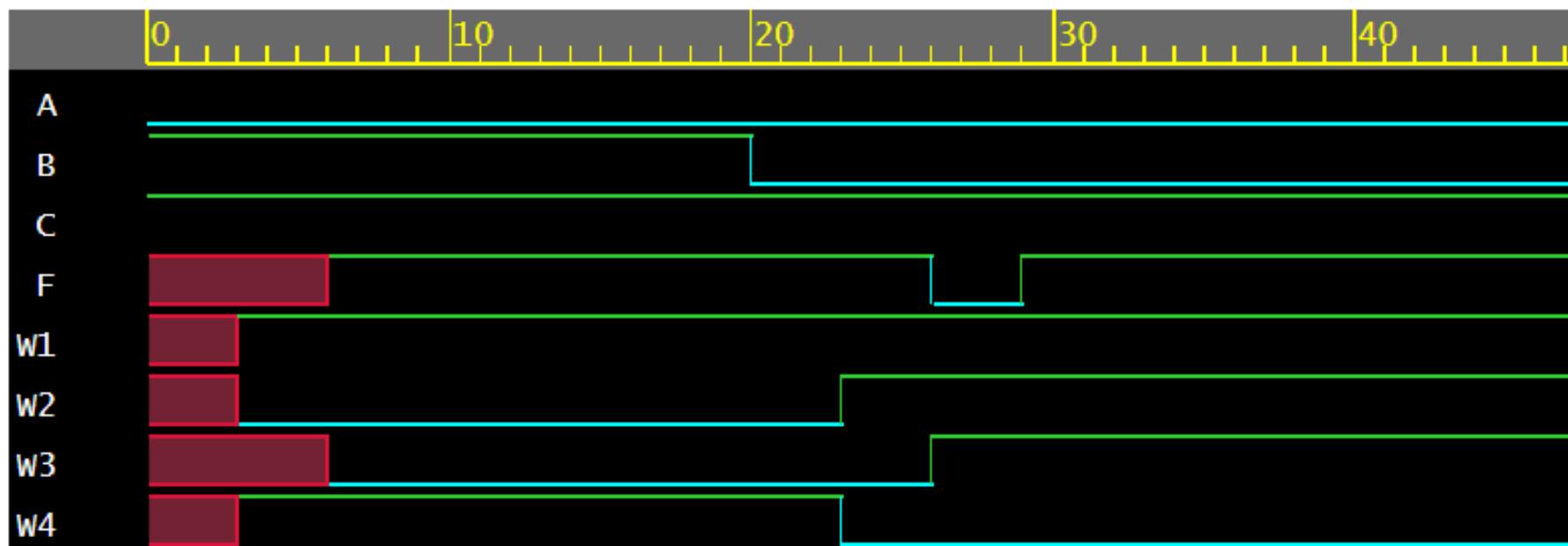
Output Glitches



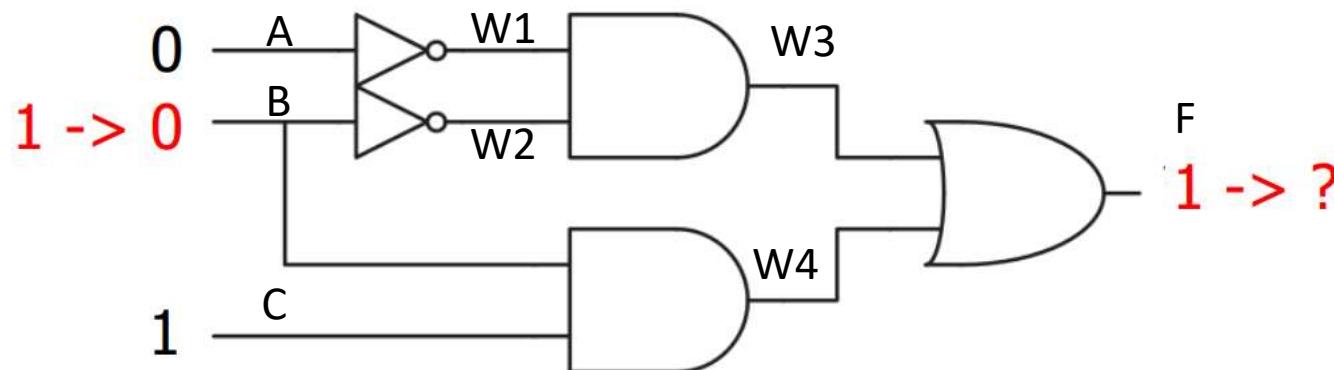
```

`timescale 1 ns / 1 ns
module testbench;
reg a,b,c; //reg for inputs
wire f; //wire for outputs
//instantiation
glitch G0 (.F(f),.A(a),.B(b),.C(c));
initial begin
$dumpfile("dump.vcd"); $dumpvars;
$display("a,b,c:f");
$monitor("%b%b%b:%b",a,b,c,f);
a=0; b=1; c=1;
#20 a=0; b=0; c=1;
#100 $finish;
end // initial
endmodule

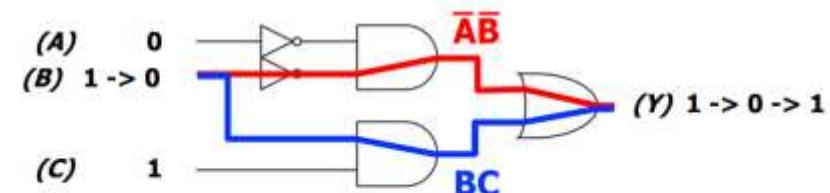
```



Output Glitches



- Glitches are **visible in K-maps**
 - Recall: K-maps show the results of a change in a **single input**
 - A glitch occurs when **moving between prime implicants**

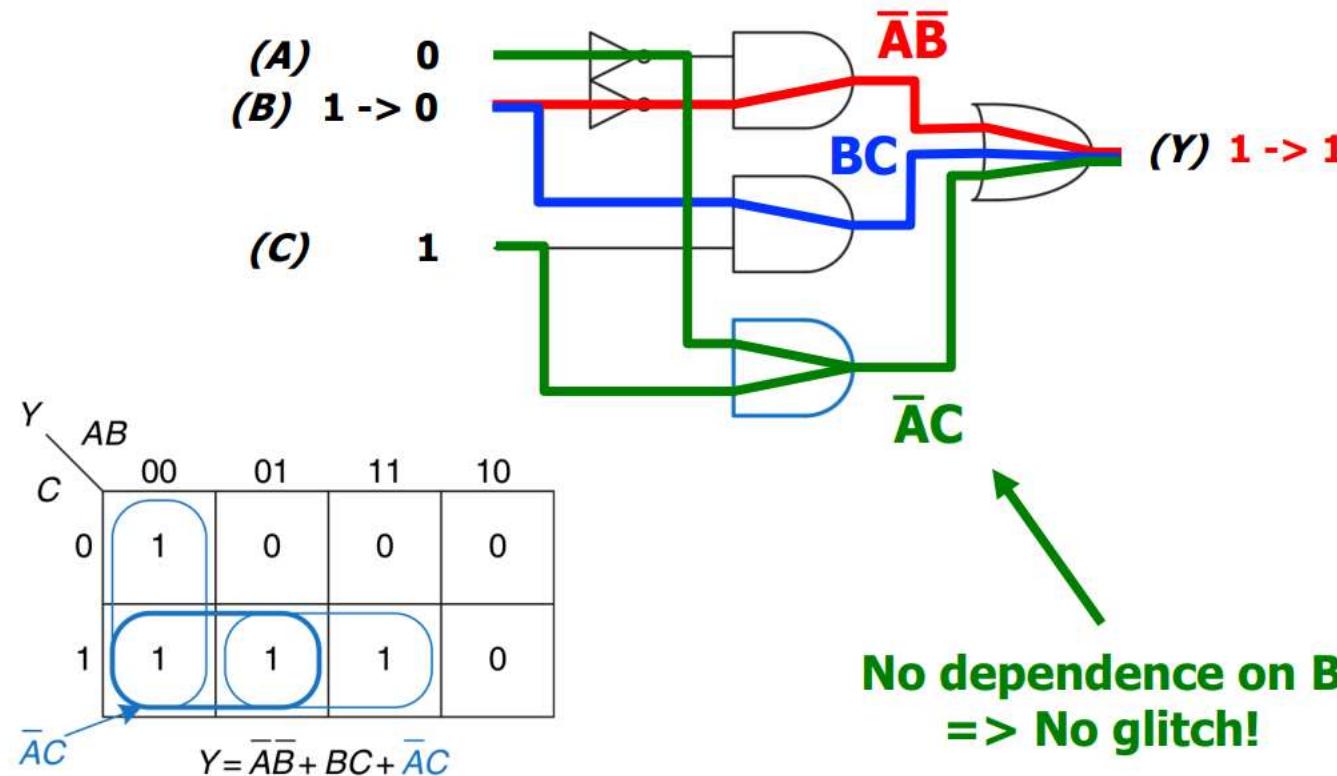


	AB	00	01	11	10
C	0	1	0	0	0
	1	1	1	1	0

$$Y = \bar{AB} + BC$$

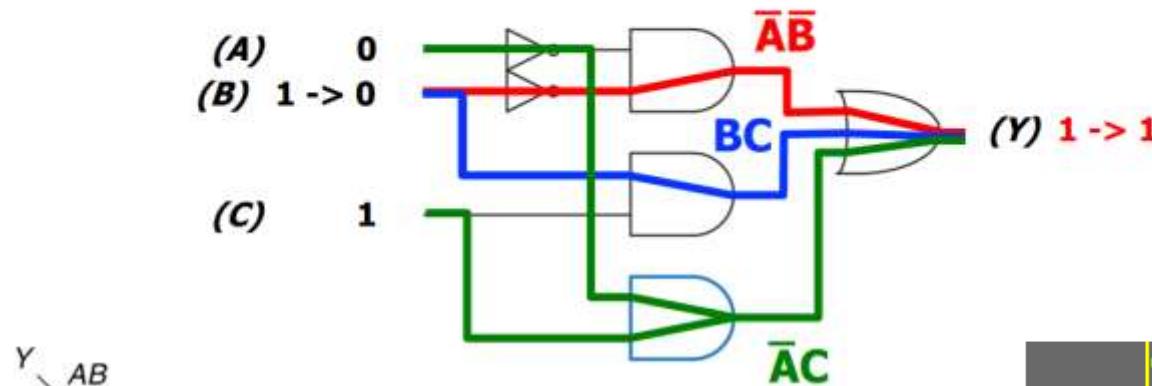
Output Glitches

- We can **fix** the issue by adding in the **consensus** term
 - Ensures **no transition** between different **prime implicants**

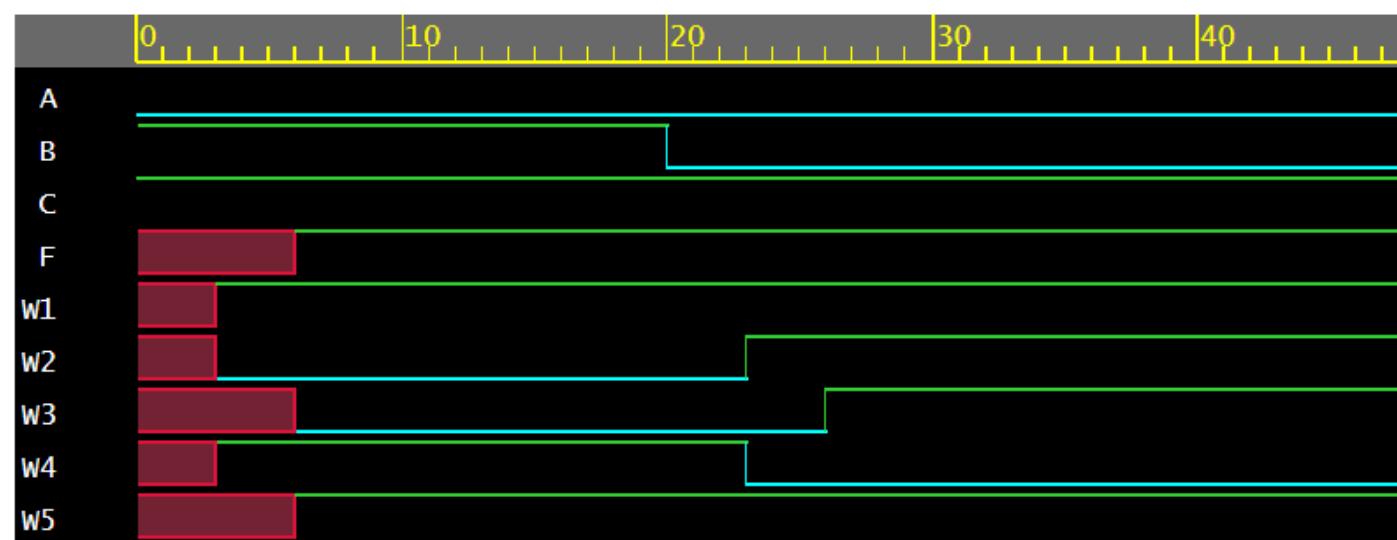


Output Glitches

- We can **fix** the issue by adding in the **consensus** term
 - Ensures **no transition** between different **prime implicants**



```
'timescale 1 ns / 1 ns
module glitch(
    output F,
    input A,B,C
);
    wire W1,W2,W3,W4,W5;
    not #3 G0 (W1,A);
    not #3 G1 (W2,B);
    and #3 G2 (W3,W1,W2);
    and #3 G3 (W4,B,C);
    or #3 G4 (F,W3,W4,W5);
    and #3 G5 (W5,C,W1);
endmodule
```



Output Glitches

Avoiding Glitches

- **Q:** Do we **always** care about glitches?
 - **Fixing** glitches is **undesirable**
 - More chip **area**
 - More **power consumption**
 - More **design effort**
 - The circuit is **eventually** guaranteed to **converge** to the **right value** regardless of glitchiness
- **A:** No, not always!
 - If we only care about the **long-term steady state output**, we can **safely ignore** glitches
 - Up to the **designer to decide** if glitches matter in their application
 - When examining simulation output, important to recognize glitches

if change of input is not faster than the clock it should not be a problem

Logic Synthesis

- Logic synthesis: The process of converting a high-level description of a design into an optimized gate-level representation.
- Convert the high-level description into a gate-level logical netlist.
- introduction of logic synthesis put HDLs at the forefront of digital design [Brayton et al. 1984].
- A logic synthesis tool compiles HDL code into an optimized physical netlist suitable for manufacture.
- At first, designers using the labor-intensive **schematic capture** and **manual layout** were able to create smaller and faster circuits than those produced by synthesis.
- **productivity advantages** offered by **synthesis** quickly **replaced manual techniques**, turning synthesis into the preeminent method of **digital circuit design**

Logic Synthesis

1.2 An Example Design Flow Using a Modern HDL

To understand how HDLs are used, it is useful to consider a design flow for a circuit. There are four major steps to complete when designing an integrated circuit:

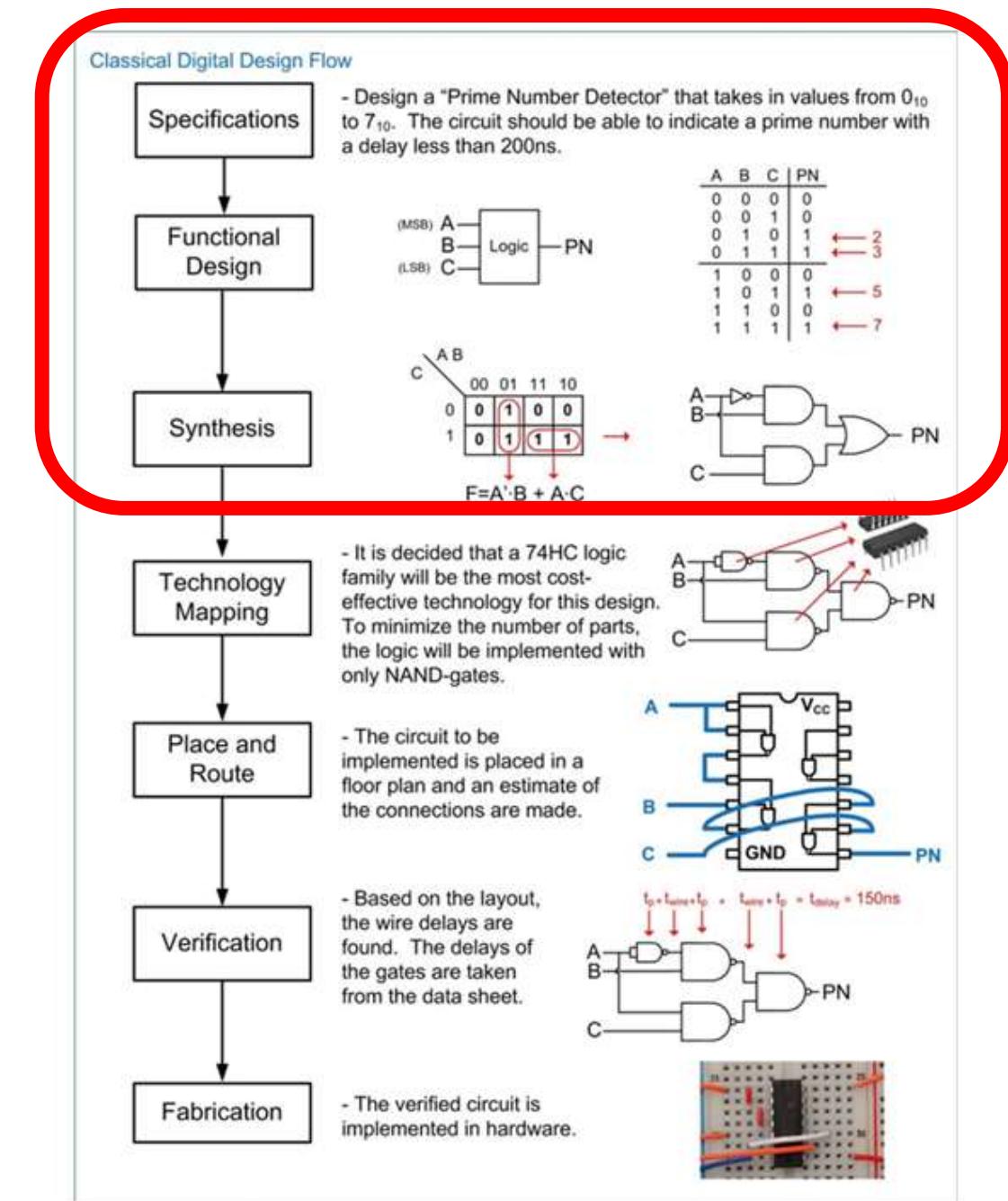
High-Level and Logic Design: Create the architecture. Specify and implement the required system behavior.

Verification: Confirm that the design description meets the specification. Run tests that verify correct operation. Simulate, check, and prove correctness using formal methods.

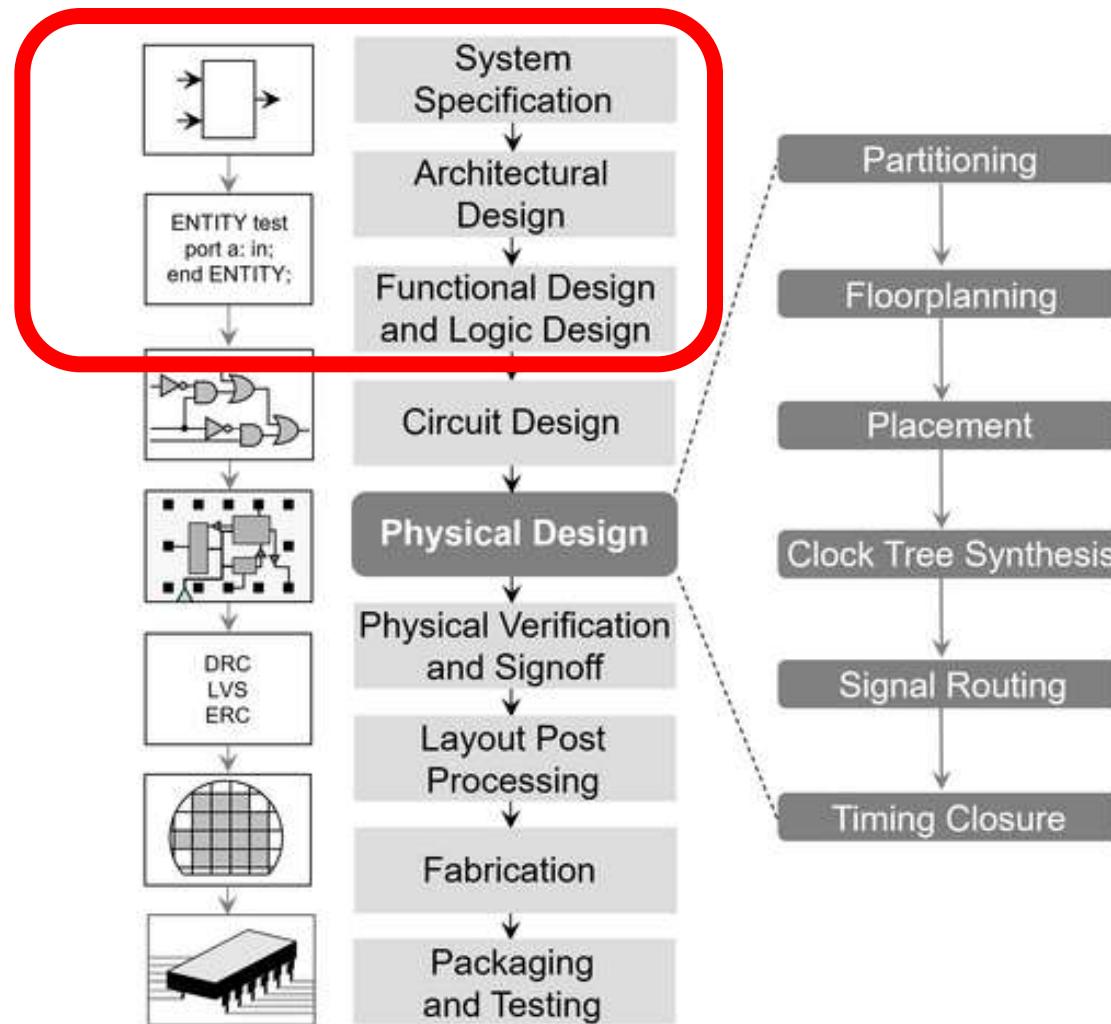
Logic Synthesis: Convert the high-level description into a gate-level logical netlist.

Physical Implementation: Create a physical layout suitable for fabrication.

Physical Chip Design



Physical Chip Design



Gowin Synthesis

GOWIN FPGA Designer - [D:\docs\ostim\digital\gowin\digital_lecture\src\circuit.v]

File Edit Project Tools Window Help

Design

- digital_lecture -
 - GW1NR-LV
- Verilog Files
 - src\circu

Process

- Design Summary
- User Constraint
- Options...

Start Page

Gowin Analyzer Oscilloscope

Schematic Viewer

IP Core Generator

Programmer

FloorPlanner

Timing Constraints Editor

DSim Cloud

RTL Design Viewer

ns / 1 ns

Serial data
clock and reset

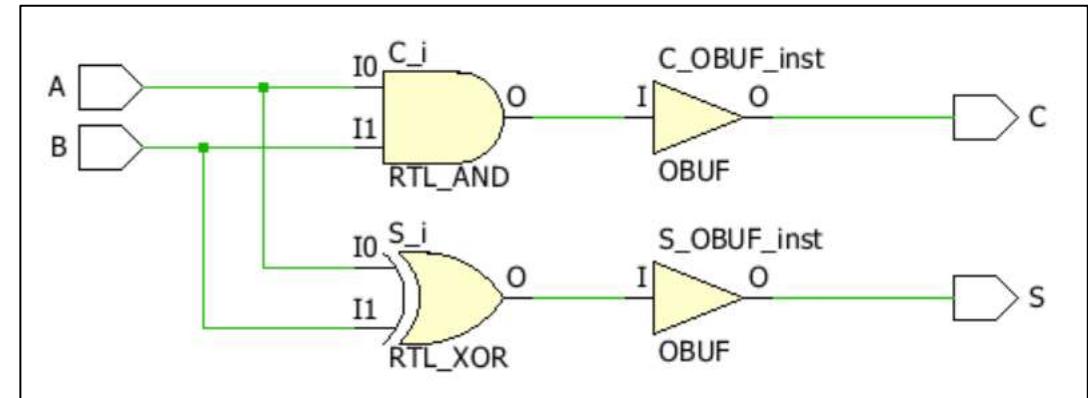
posedge clk, posedge
begin
b0;
b0;
b0;

```
graph LR; A((A)) --> i4((i4)); A --> i5((i5)); B((B)) --> i4; B --> i5; i4 --> S((S)); i5 --> C((C))
```

Vivado Synthesis (AMD FPGA) Quartus (INTEL FPGA)

Vivado quick start:

- New Project
- RTL Project
- Default Part: xc7a50ticsg325 Artix7 Spartan
- Next-Next
- Design Sources – Right Click-Add Sources
- Add Design Sources
- Create File – halfadder.v
- Define Module- OK
- double Click halfadder.v and write the code or copy/paste from EDAPlayground
- Flow-Open Elaborated Design-OK
- Schematic will be created



Verilog for Synthesis: Yosys

- So far we have seen gate-level modeling using Verilog **primitives**
- Behavioral modeling allows us to describe the functionality of logic circuit using programming like structures of Verilog (operators, loops, if-else, case etc.)
- **Synthesis** tools convert the behavioral description into a gate-level logical netlist.
- it maybe different than what you expected
- Example: Select **Yosys Open Synthesis** and **Show diagram**

▼ Tools & Simulators ?

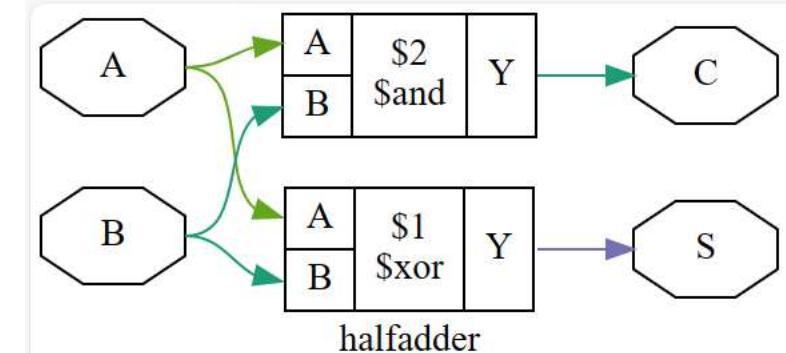
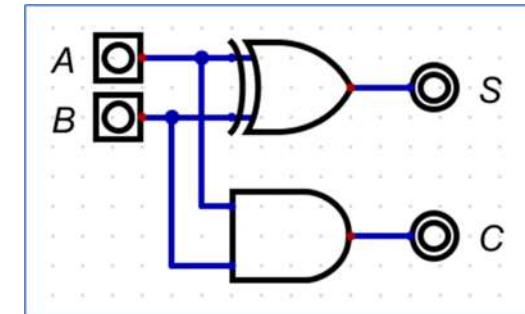
Yosys 0.9.0

Synthesis options

Use **run.yos** file instead

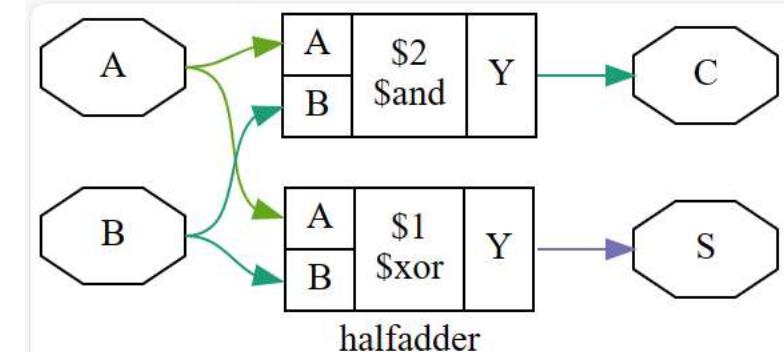
Show **diagram** after run ?

```
module halfadder (
    output C,S,
    input A,B);
    assign S=A^B;
    assign C=A&B;
endmodule
```

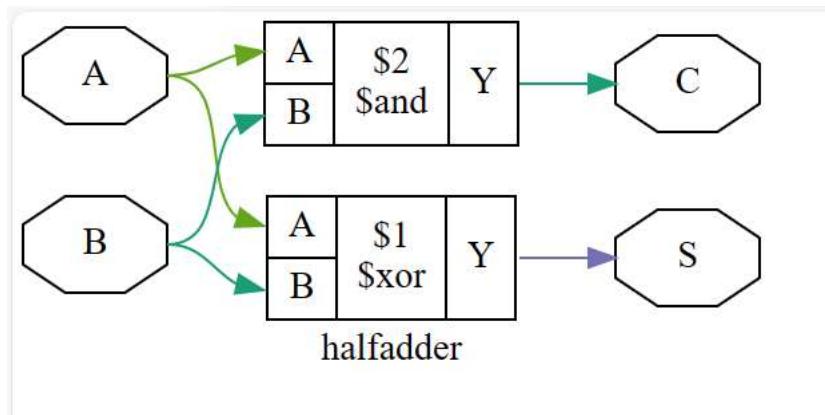


Verilog for Synthesis: Yosys

- A Hardware Description Language (HDL) is a computer language used to describe circuits.
- A HDL synthesis tool is a computer program that takes a formal description of a circuit written in an HDL as input and generates a netlist that implements the given circuit as output
- Square boxes are cells
- Octagon-shaped nodes are ports



Verilog for Synthesis: Yosys Netlist



- At the logical gate level the design is represented by a netlist that uses only cells from a small number of single-bit cells, such as basic logic gates (AND, OR, NOT, XOR, etc.) and registers (usually D-Type Flip-flops).
- A number of netlist formats exists that can be used on this level, e.g. the **Electronic Design Interchange Format (EDIF)**, but for ease of simulation often a **HDL netlist** is used.
- The latter is a HDL file (**Verilog** or VHDL) that only uses the most basic language constructs for **instantiation** and **connecting of cells**

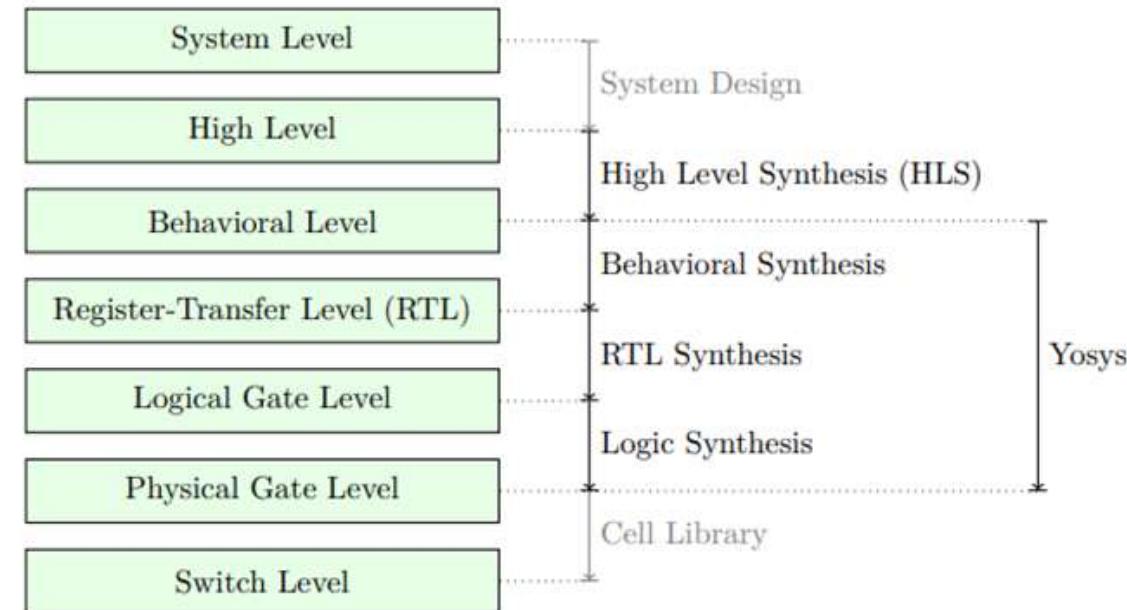


Figure 2.1: Different levels of abstraction and synthesis.

1. Gate Level Modeling: primitives (and or not xor etc)
2. Dataflow Modeling : continuous assignment, logic and arithmetic and conditional operators
3. Behavioral Modeling: always @*, if-else, case,

Dataflow Modeling : Continuous Assignment

Dataflow modeling style is mainly used to describe combinational circuits. The basic mechanism used is the continuous assignment. In a continuous assignment, a value is assigned to a data type called net. The syntax of a continuous assignment is:

assign [delay] LHS_net = RHS_expression;

Where LHS_net is a destination net of one or more bit, and RHS_expression is an expression consisting of various operators. The statement is evaluated at any time any of the source operand value changes and the result is assigned to the destination net after the delay unit.

The gate level modeling examples can be described in dataflow modeling using the continuous assignment. For example,

```
assign out1 = in1 & in2; // perform and function on in1 and in2 and assign the result to out1  
assign out2 = not in1;  
assign #2 z[0] = ~(ABAR & BBAR & EN); // perform the desired function and assign the result  
after 2 units
```

Dataflow Modeling: Continuous Assignment

- Continuous Assignments useful for behavioral description of combinatorial logic circuits
- Assign **wires** EXPRESSION to WIRE
- it is wire connection and immediately changes with the expression

```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
module halfadder (
    output C,S,
    input A,B);
    assign S=A^B;
    assign C=A&B;
endmodule
```

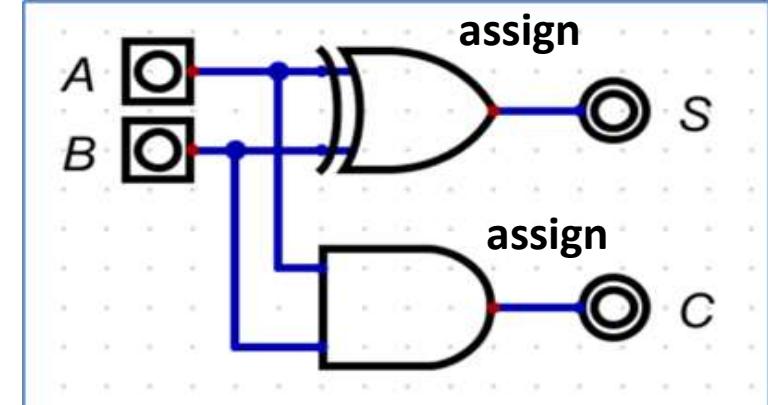
11.0 Continuous Assignments

Explicit Continuous Assignment

```
net_type [size] net_name;
assign # (delay) net_name = expression;
```

Implicit Continuous Assignment

```
net_type (strength) [size] # (delay) net_name = expression;
```



Dataflow Modeling: Net Types

- Continuous Assignments useful for behavioral description of combinatorial logic circuits
- remember input and output datatypes are **wire**
- **wire** is the most used but there are more

```
module halfadder (
    output C,S,
    input A,B);

    assign S=A^B;
    assign C=A&B;
endmodule
```

Nets are used to connect structural components together.

- A net data type must be used when a signal is:
 - Driven by the output of a module instance or primitive instance.
 - Connected to an input or inout port of the module in which it is declared.
 - On the left-hand side of a continuous assignment.
- **net_type** is one of the following keywords:

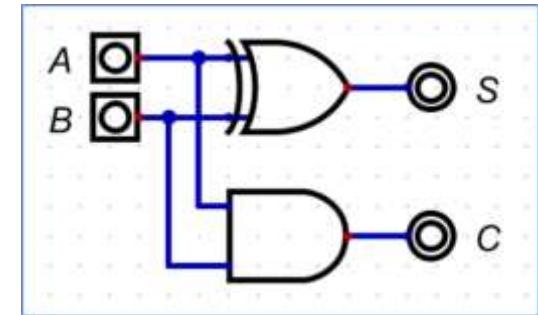
wire	interconnecting wire; CMOS resolution
-------------	---------------------------------------

11.0 Continuous Assignments

Explicit Continuous Assignment
net_type [size] net_name ;
assign #(delay) net_name = expression;
Implicit Continuous Assignment
net_type (strength) [size] #(delay) net_name = expression;

Dataflow Modeling: Bitwise Operators

- We can describe half adder using Bitwise operators XOR and AND
- Continuous assignments drive net types with the result of an expression.
- result is automatically updated anytime a value on the right-hand side changes.



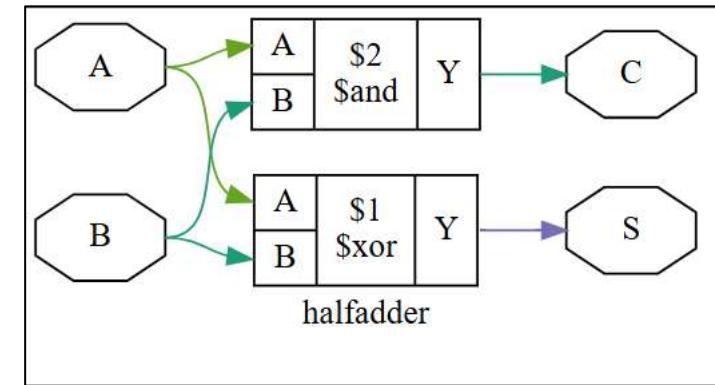
```
module halfadder (
    output C,S,
    input A,B);
    assign S=A^B;
    assign C=A&B;
endmodule
```

Bitwise Operators		
\sim	$\sim m$	invert each bit of m
$\&$	$m \& n$	AND each bit of m with each bit of n
$ $	$m n$	OR each bit of m with each bit of n
$^$	$m ^ n$	exclusive-OR each bit of m with n
$\sim^$ or $\sim\sim$	$m \sim^ n$	exclusive-NOR each bit of m with n
$<<$	$m << n$	shift m left n-times and fill with zeros
$>>$	$m >> n$	shift m right n-times and fill with zeros

Dataflow Modeling: Synthesize

```
module halfadder (
    output C,S,
    input A,B);

    assign S=A^B;
    assign C=A&B;
endmodule
```



Dataflow Modeling: Arithmetic Operators

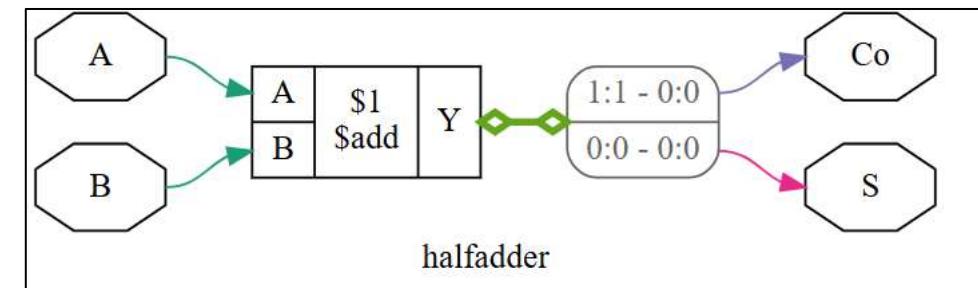
- We can use arithmetic operators also to define half adder
- Notice that synthesizer generated the half adder circuit based on our modeling

Miscellaneous Operators		
<code>? :</code>	<code>sel?m:n</code>	conditional operator; if sel is true, return m; else return n
<code>{}</code>	<code>{m, n}</code>	concatenate m to n, creating a larger vector

Arithmetic Operators		
<code>+</code>	<code>m + n</code>	add n to m
<code>-</code>	<code>m - n</code>	subtract n from m
<code>-</code>	<code>-m</code>	negate m (2's complement)
<code>*</code>	<code>m * n</code>	multiply m by n
<code>/</code>	<code>m / n</code>	divide m by n
<code>%</code>	<code>m % n</code>	modulus of m / n
<code>**</code>	<code>m ** n</code>	m to the power n (new in Verilog-2001)
<code><<<</code>	<code>m <<< n</code>	shift m left n-times, filling with 0 (new in Verilog-2001)
<code>>>></code>	<code>m >>> n</code>	shift m right n-times; fill with value of sign bit if expression is signed, otherwise fill with 0 (Verilog-2001)

```
module halfadder (
    output Co,S,
    input A,B);
    assign {Co,S} = A+B;
endmodule
```

- Boxes with round corners with lines such as 0:0 - 42:42 are used to break out and re-combine nets from busses.
- The numbers tell you which bits on which side are connected. for example '3:0 - 7:4' means that the bits 3:0 from the left net are connected to bits 7:4 from the right net



Dataflow Modeling: Arithmetic Operators

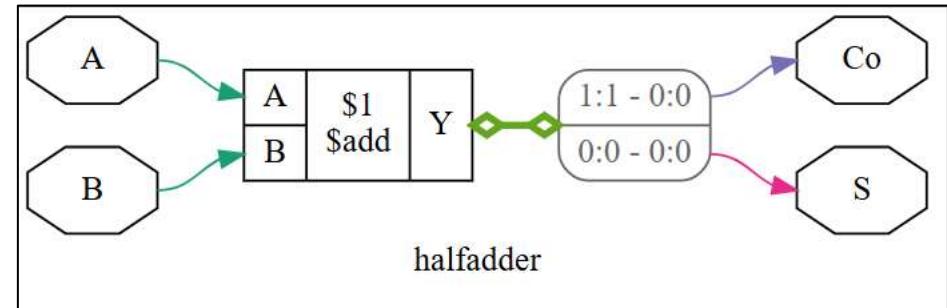
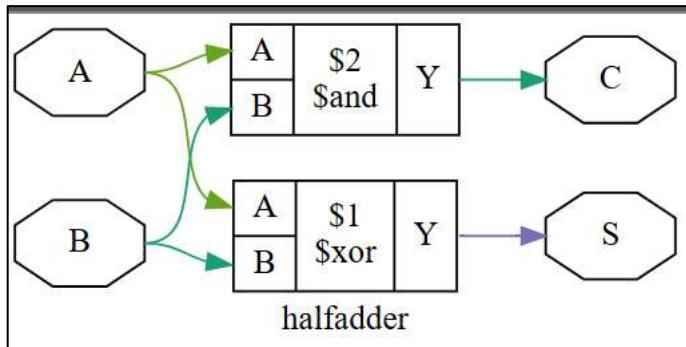
- We can use arithmetic operators also to define half adder

```
module halfadder (
    output C,S,
    input A,B);

    assign S=A^B;
    assign C=A&B;
endmodule
```

```
module halfadder (
    output Co,S,
    input A,B);

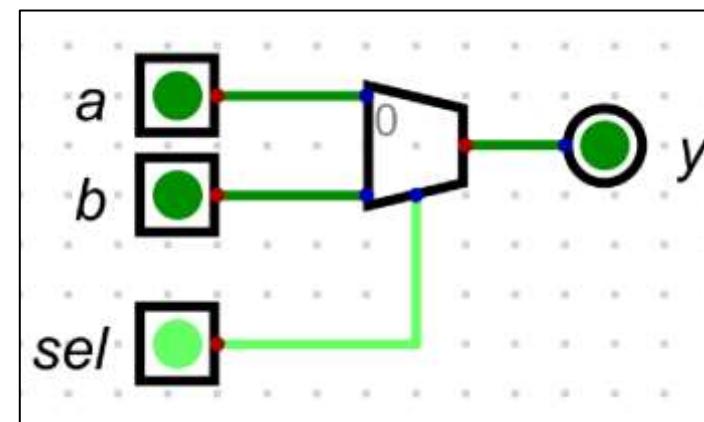
    assign {Co, S}=A+B;
endmodule
```



Dataflow Modeling: Conditional Operator

- We can use conditional operator to define a 2to1 MUX

```
module mux2to1(  
    input sel,  
    input a, b,  
    output y);  
    assign y = sel ? b : a;  
endmodule
```



Miscellaneous Operators

? : sel ? m : n conditional operator; if sel is true, return m: else return n

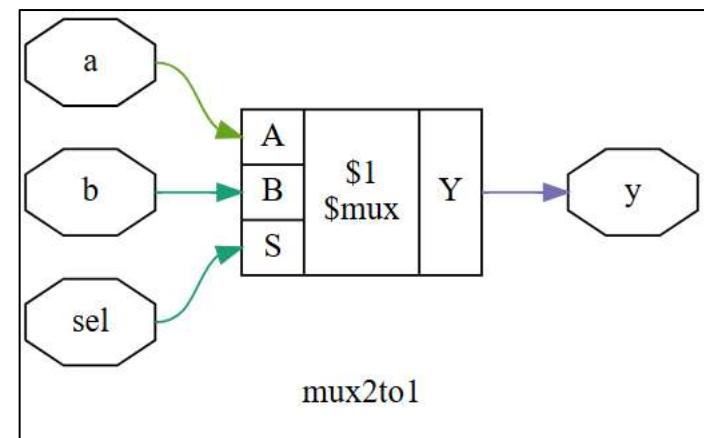
Dataflow Modeling : MUX with conditional

- We can use conditional operator to define a 2to1 MUX
- testbench for mux2to1 module

```
// testbench
module testbench;
  reg a,b,sel; //inputs
  wire y; //outputs
//intantiate
mux2to1 M0 (sel,a,b,y);
initial begin
  $display("a b:sel y");
  $monitor("%b %b:%b %b", a,b,sel,y);
  a= 0; b= 1; sel = 1;
#1; sel = 0;
end
endmodule
```

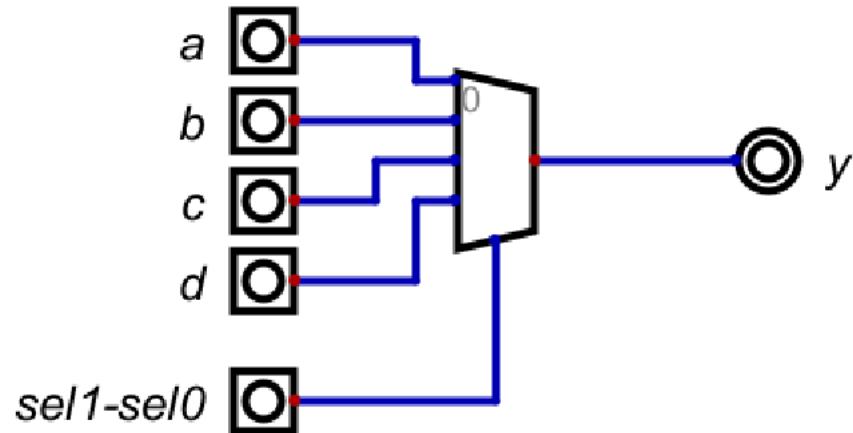
a	b:sel	y
0	1:1	1
0	1:0	0

```
module mux2to1(
  input sel,
  input a, b,
  output y);
  assign y = sel ? b : a;
endmodule
```



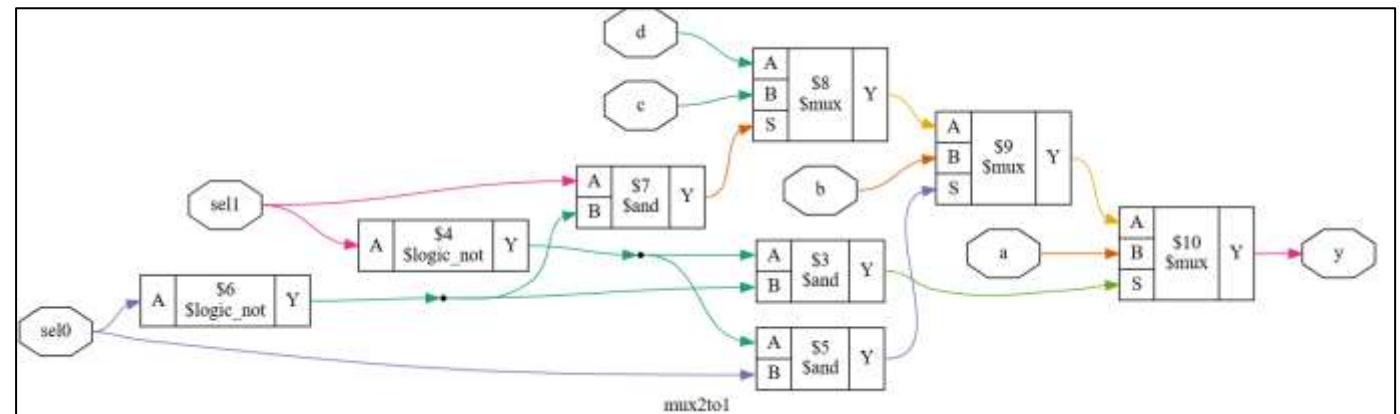
Dataflow Modeling : MUX4to1 with Conditional

- We can use conditional operator to define a 4to1 MUX



sel1	sel0	y
0	0	a
0	1	b
1	0	c
1	1	d

```
module mux2to1(  
    input sel0, sel1,  
    input a, b, c,d,  
    output y);  
    assign y = !sel1 & !sel0 ? a :  
              !sel1 & sel0 ? b :  
              sel1 & !sel0 ? c :  
              sel1 & sel0 ? d;  
endmodule
```



Dataflow Modeling : MUX4to1 with Conditional

- We can use conditional operator to define a 4to1 MUX

sel1	sel0	y
0	0	1
0	1	1
1	0	0
1	1	0

$$y = sel1' * sel0' + sel1' * sel0$$

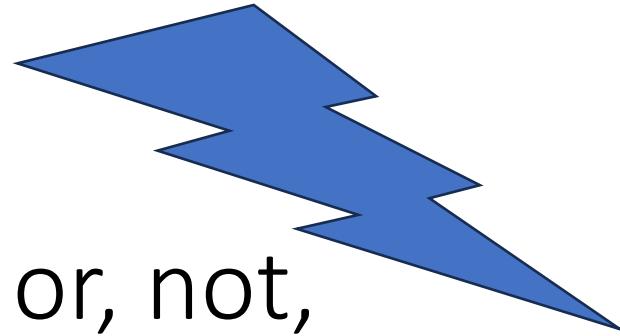
Bitwise Operators		
\sim	$\sim m$	invert each bit of m
$\&$	$m \& n$	AND each bit of m with each bit of n
$ $	$m n$	OR each bit of m with each bit of n
$^$	$m ^ n$	exclusive-OR each bit of m with n
$\sim^ or \sim~$	$m \sim^ n$	exclusive-NOR each bit of m with n
$<<$	$m << n$	shift m left n-times and fill with zeros
$>>$	$m >> n$	shift m right n-times and fill with zeros

```
module mux2to1(  
    input sel0, sel1,  
    input a, b, c,d,  
    output y);
```

```
assign y = ~sel1&~sel0 | ~sel1&sel0;
```

```
endmodule
```

VERILOG MODELING



1. Gate Level Modeling: primitives (and, or, not, xor, etc), UDP
2. Dataflow Modeling : continuous assignment, logic, arithmetic and conditional operators
3. Behavioral Modeling: always @*, if-else, case,

EDAPLAYGROUND VS ICARUS VERILOG

- EDAPlayground is nice to get started quickly
- Icarus Verilog (iverilog) running at the background for simulation
- Yosys is running at the back ground for synthesis
- GTKWave for waveform simulation
- VSCode is well accepted code development environment
- We can use VScode for Verilog coding and simulations

VSCode

- We can create an IDE similar to EDAPlayground by using VSCode
- Design file is at the right, testbench is at the left and command line at the bottom

The screenshot shows the Visual Studio Code interface with several tabs open:

- `rom_tb.v`: Testbench for the ROM module.
- `rom.v`: The Verilog code for the ROM module.
- `mux21_case_tb.v`: Another testbench tab.
- `mux21_case.v`: Another Verilog code tab.

The `rom.v` tab contains the following Verilog code:

```
module rom (
    output reg [15:0] data_out // 16-bit data output
);
    // Memory array (16 locations of 16-bit data)
    reg [15:0] mem [0:15];
    // Initialize memory from file
    initial begin
        $readmemb("program.txt", mem);
    end
    always @* begin
        data_out = mem[address];
    end
endmodule
```

The `rom_tb.v` tab contains the following Verilog code:

```
'timescale 1ns / 1ns
`include "rom.v"
module tb_rom;
    reg [3:0] pcounter;
    wire [15:0] data_out;

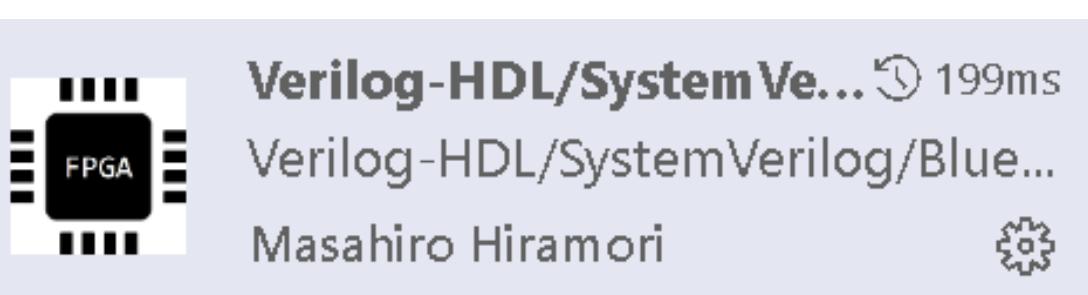
    rom M0 (
        .address(pcounter),
        .data_out(data_out)
    );
    initial begin
        $monitor("Time=%t pcounter=%d data_out=%b",
                $time, pcounter, data_out);
    end
endmodule
```

At the bottom, the terminal window shows the execution of the Verilog code:

```
PS D:\docs\ostim\digital\verilog_codes> iverilog mux21_case_tb.v; vvp ./a.out
a b:sel y
0 1:1 1
0 1:0 0
PS D:\docs\ostim\digital\verilog_codes> iverilog mux21_case_tb.v; vvp ./a.out
a b:sel y
0 1:1 1
0 1:0 0
```

VSCode

- Install Masahiro Hiramori extension for syntax checking and formating
- Ctrl-S will run iverilog automatically and check the syntax errors



A screenshot of the VSCode editor showing a Verilog module. The code is as follows:

```
17      pcounter = + ui, #10,
18      pcounter = 4'd2; #10;
19
20
21
22      end
23
24      endmodule
```

A tooltip is displayed over the word 'end' at line 22, indicating a 'syntax error iverilog(iverilog)'. Below the tooltip, there are links to 'View Problem (Alt+F8)' and 'No quick fixes available'.

Icarus Verilog

- Icarus Verilog developed by Stephen Williams:
- [Icarus Verilog — Icarus Verilog documentation \(steveicarus.github.io\)](https://steveicarus.github.io/)
- Windows compiled version :
- [iverilog-v12-20220611-x64 setup \[18.2MB\]](#)
- [Icarus Verilog for Windows \(bleyer.org\)](#)
- Command line
- **iverilog -o mux21case.out mux21_case.v mux21_case_tb.v will create mux21case.out. default output file is a.out**
- **vvp mux21case.out will simulate the verilog code and print simulation results**
- We can include design file into testbench using include, and vvp a.out to make our life easy
- **iverilog mux21_case_tb.v; vvp .\a.out**



GTKWave

- [GTKWave \(sourceforge.net\)](http://sourceforge.net)



Icarus Verilog Documentation

- [Icarus Verilog — Icarus Verilog documentation \(steveicarus.github.io\)](https://steveicarus.github.io)



```
module hello;
    initial
        begin
            $display("Hello, World");
            $finish ;
        end
    endmodule
```

```
> iverilog -o hello.out hello.v; vvp hello.out
Hello, World
hello.v:5: $finish called at 0 (1s)
```

User Defined Primitives (UDP)

- If we know/given the truth table of the circuit we can use UDPs to define and use this primitive inside modules

```
iverilog .\mux21_udp_tb.v; vvp .\a.out  
time absel y
```

```
0 xxx x  
1 000 0  
2 001 0  
3 010 0  
4 011 1  
5 100 1  
6 101 0  
7 110 1  
8 111 1
```

```
primitive mux2to1  
(Y,A,B,SEL);  
output Y;  
input A, B, SEL;  
table  
// A B sel : Y  
0 0 0 : 0;  
0 0 1 : 0;  
0 1 0 : 0;  
0 1 1 : 1;  
1 0 0 : 1;  
1 0 1 : 0;  
1 1 0 : 1;  
1 1 1 : 1;  
endtable  
endprimitive
```

```
// Test bench  
\timescale 1 ns / 100 ps  
\include "mux21_udp.v"  
module tb_top;  
reg a1, b1, sel1; //reg for  
inputs  
wire y1; //wire for outputs  
mux2to1 M0  
(.Y(y1),.A(a1),.B(b1),.SEL(sel1));  
//instantiation  
initial begin  
$dumpfile("dump.vcd");  
$dumpvars;  
$display("time absel y");  
$monitor("%0d %b%b%b %b",  
$time,a1,b1,sel1,y1);  
#1 a1=0; b1=0; sel1=0;  
#1 a1=0; b1=0; sel1=1;  
#1 a1=0; b1=1; sel1=0;  
#1 a1=0; b1=1; sel1=1;  
#1 a1=1; b1=0; sel1=0;  
#1 a1=1; b1=0; sel1=1;  
#1 a1=1; b1=1; sel1=0;  
#1 a1=1; b1=1; sel1=1;
```

Behavioral Modeling

Behavioral modeling is used to describe complex circuits. It is primarily used to model sequential circuits, but can also be used to model pure combinatorial circuits. The mechanisms (statements) for modeling the behavior of a design are:

- **initial Statements**
- **always Statements**

10.0 Procedural Blocks

```
type_of_block @ (sensitivity_list)
  statement_group : group_name
  local_variable_declarations
  time_control procedural_statements
end_of_statement_group
```

- *type_of_block* is either **initial** or **always**
 - **initial** blocks process statements one time.
 - **always** blocks are an infinite loop which process statements repeatedly.

Procedural Statements : always @ ()

- **always @(signal, signal, ...) infers combinational logic** if the list of signals contains all signals read within the procedure.

10.2 Sensitivity Lists

The sensitivity list is used at the beginning of an always procedure to infer combinational logic or sequential logic behavior in simulation.

- **always @ (signal, signal, ...)** infers combinational logic if the list of signals contains all signals read within the procedure.
- **always @*** infers combinational logic. Simulation and synthesis will automatically be sensitive to all signals read within the procedure. @* was added in Verilog-2001.
- **always @ (posedge signal, negedge signal, ...)** infers sequential logic. Either the positive or negative edge can be specified for each signal in the list. A specific edge should be specified for each signal in the list.

Procedural Statements: if-else

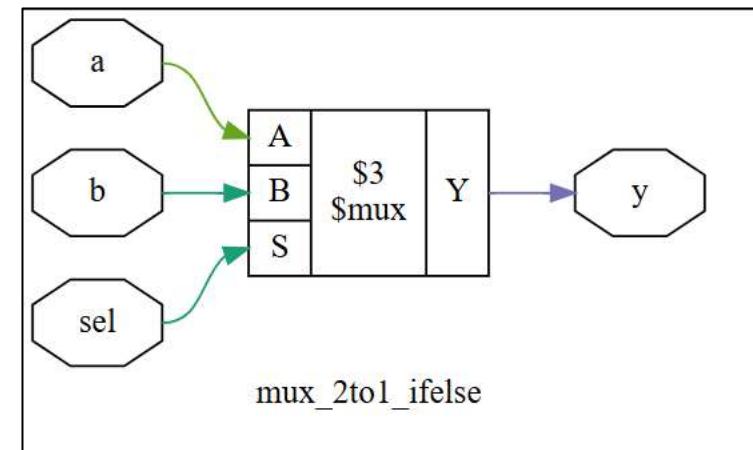
- In addition to using **continuous assign statements** and **primitive gate instantiations** to specify **combinational logic**, **procedural statements** may be used.
- if outputs are assigned within a procedural block it must be set to **reg** (register) variable data type

```
if ( expression ) statement or statement_group  
else statement or statement_group
```

Executes the first statement or statement group if the expression evaluates as true. Executes the second statement or statement group if the expression evaluates as false or unknown.

NOTICE that although module is defined differently
synthesized circuit is the SAME

```
module mux_2to1_ifelse (  
    input wire a,      // Input 0  
    input wire b,      // Input 1  
    input wire sel,    // Select line  
    output reg y       // Output  
);  
  
    always @* begin  
        if (sel)  
            y = b;  
        else  
            y = a;  
    end  
  
endmodule
```



Procedural Statements: if-else

```
// Test bench
`timescale 1 ns / 1 ns
`include "mux_2to1_ifelse.v"
module tb_top;
    reg a1, b1, sel1; //reg for inputs
    wire y1; //wire for outputs
    mux_2to1_ifelse M0
(.y(y1),.a(a1),.b(b1),.sel(sel1));
//instantiation
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("time absel y");
    $monitor("%0t %b%b%b %b",
$time,a1,b1,sel1,y1);
    #1 a1=0; b1=0; sel1=0;
    #1 a1=0; b1=0; sel1=1;
    #1 a1=0; b1=1; sel1=0;
    #1 a1=0; b1=1; sel1=1;
    #1 a1=1; b1=0; sel1=0;
    #1 a1=1; b1=0; sel1=1;
```

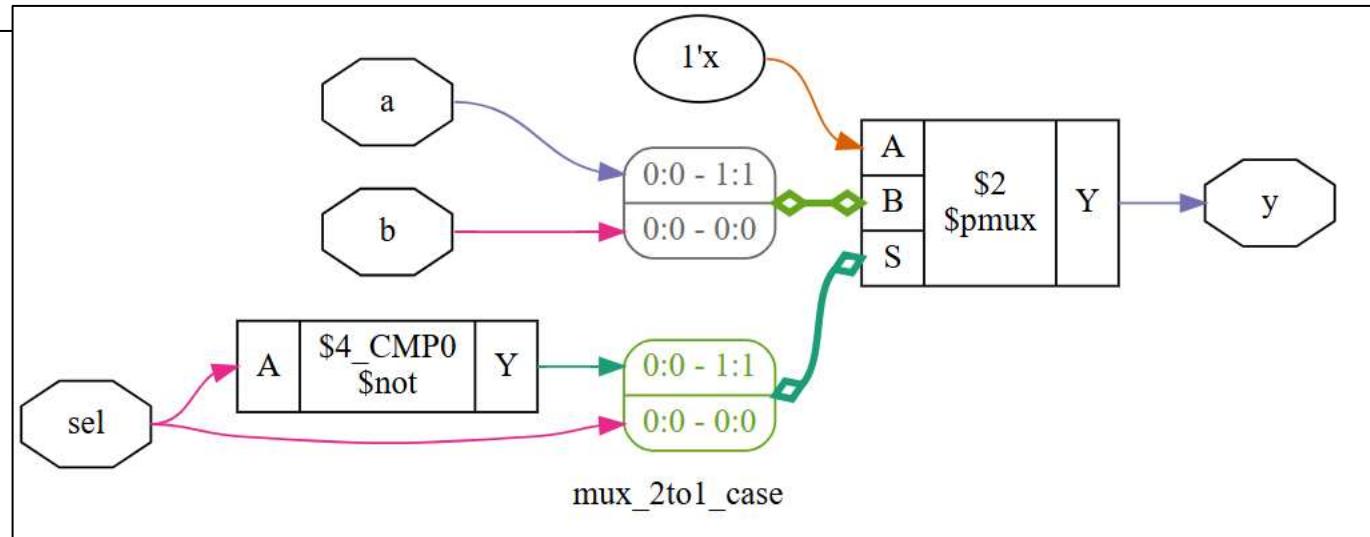
```
module mux_2to1_ifelse (
    input wire a,           // Input 0
    input wire b,           // Input 1
    input wire sel,         // Select line
    output reg y            // Output
);
    always @* begin
        if (sel)
            y = b;
        else
            y = a;
    end
endmodule
```

Procedural Statements: case

```
module mux_2to1_case (
    input wire a,      // Input 0
    input wire b,      // Input 1
    input wire sel,    // Select line
    output reg y       // Output
);

    always @* begin
        case (sel)
            1'b0: y = a; // When sel is 0, output y is assigned input a
            1'b1: y = b; // When sel is 1, output y is assigned input b
            default: y = a; // Default case (though not strictly necessary here)
        endcase
    end

endmodule
```



```
case ( expression )
    case_item:
    case_item, case_item:
    default:
endcase
```

statement or statement_group
statement or statement_group
statement or statement_group

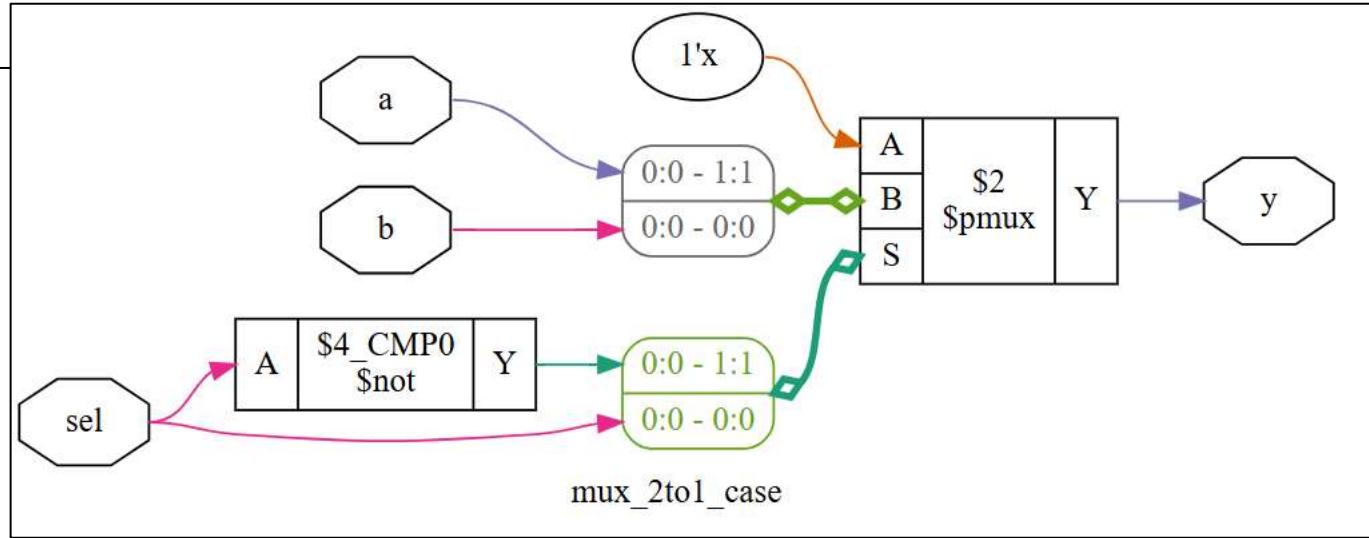
Compares the value of the expression to each **case item** and executes the *statement or statement_group* associated with the first matching **case**. Executes the **default** if none of the **cases** match (the **default case** is optional).

Procedural Statements: case

```
module mux_2to1_case (
    input wire a,      // Input 0
    input wire b,      // Input 1
    input wire sel,    // Select line
    output reg y       // Output
);

    always @* begin
        case (sel)
            1'b0: y = a; // When sel is 0, output y is assigned input a
            1'b1: y = b; // When sel is 1, output y is assigned input b
            default: y = a; // Default case (though not strictly necessary here)
        endcase
    end

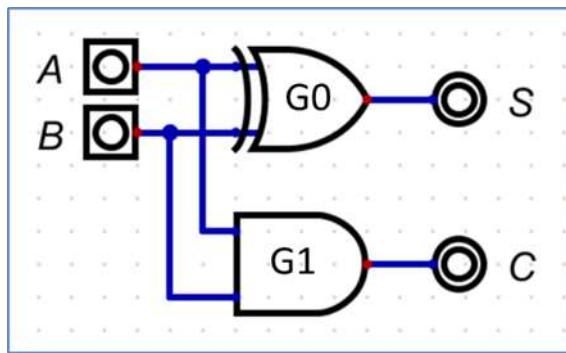
endmodule
```



The `$pmux` cell is used to multiplex between many inputs using a one-hot select signal. Cells of this type have `a` and a parameter and inputs `, ,` and `,` and an output `.` The input is `* bits wide`. The input and the output are both `* bits wide` and the input is `* bits wide`. When all bits of `are zero`, the value from `input` is sent to the output. If the `n'th bit from` is set, the value `n'th bits wide` slice of the input is sent to the output. When more than one bit from is set the output is undefined. Cells of this type are used to model “parallel cases” (defined by using the `parallel_case` attribute or detected by an optimization).

Half Adder with Primitives

```
//testbench
`timescale 1 ns / 100 ps
`include "halfadder_primitives.v"
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("time ab:cs");
        $monitor("%0d %b%b:%b%b",$time,a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

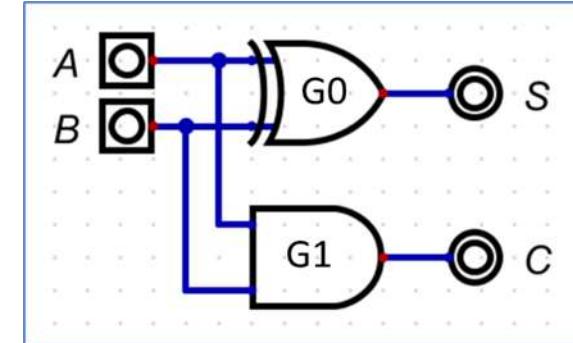


```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
iverilog .\halfadder_primitives_tb.v; vvp a.out
time ab:cs
0 xx:xx
1 00:00
2 01:01
3 10:01
4 11:10
```

Half Adder with User Defined Primitives

```
//testbench
`timescale 1 ns / 100 ps
`include "halfadder_primitives.v"
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("time ab:cs");
        $monitor("%0d %b%b:%b%b",$time,a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

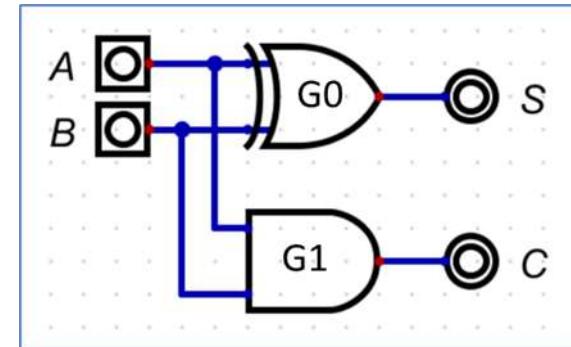


```
//half adder
module halfadder(
    output S,C,
    input A,B
);
    xor G0 (S,A,B);
    and G1 (C,A,B);
endmodule
```

```
iverilog .\halfadder_primitives_tb.v; vvp a.out
time ab:cs
0 xx:xx
1 00:00
2 01:01
3 10:01
4 11:10
```

Half Adder with Data Flow Modeling: Logic Operators

```
//testbench
`timescale 1 ns / 100 ps
`include "halfadder_logic.v"
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("time ab:cs");
        $monitor("%0d %b%b:%b%b",$time,a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```

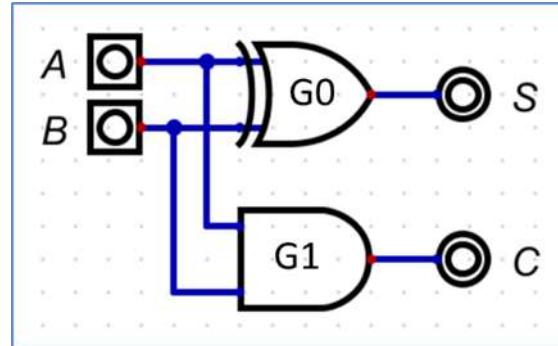


```
module halfadder
(
    output C,S,
    input A,B);
    assign S=A^B;
    assign C=A&B;
endmodule
```

```
iverilog .\halfadder_logic_tb.v; vvp a.out
time ab:cs
0 xx:xx
1 00:00
2 01:01
3 10:01
4 11:10
```

Half Adder with Data Flow Modeling: Arithmetic Operators

```
//testbench
`timescale 1 ns / 100 ps
`include "halfadder_arithmetic.v"
module testbench;
    reg a,b; //reg for inputs
    wire s,c; //wire for outputs
    //instantiation
    halfadder G0 (.S(s),.C(c),.A(a),.B(b));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
        $display("time ab:cs");
        $monitor("%0d %b%b:%b%b",$time,a,b,c,s);
        #1 a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #10 $finish;
    end // initial
endmodule
```



```
module halfadder (
    output C,S,
    input A,B);
    assign {C,S}=A+B;
endmodule
```

```
iverilog .\halfadder_arithmetic_tb.v; vvp a.out
time ab:cs
0 xx:xx
1 00:00
2 01:01
3 10:01
4 11:10
```

Half Adder with Data Flow Modeling: Conditional Assignment

```
//testbench
`timescale 1 ns / 100 ps
`include "halfadder_conditional.v"
module testbench;
  reg a,b; //reg for inputs
  wire s,c; //wire for outputs
  //instantiation
  halfadder G0 (.S(s),.C(c),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("time ab:cs");
    $monitor("%0d %b%b:%b%b",$time,a,b,c,s);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end // initial
endmodule
```

```
module halfadder (
  output C,S,
  input A,B);

  assign S = (A == 1'b1 && B == 1'b1) ? 1'b0 :
             (A == 1'b0 && B == 1'b0) ? 1'b0 :
             1'b1;
  assign C = (A == 1'b1 && B == 1'b1) ? 1'b1 :
             1'b0;
endmodule
```

```
iverilog .\halfadder_conditional_tb.v; vvp .\a.out
time ab:cs
0 xx:xx
1 00:00
2 01:01
3 10:01
4 11:10
```

Binary Numbers in Verilog

- If data is given as just names it means 1-bit
- multiple bit binary numbers can be represented like 1'b0 4'b0101
8'b01010101
- array representation : 4bit input A [3:0]

Port Range -Vector

5.2 Port Declarations

Combined Declarations (<i>added in Verilog-2001</i>) <code>port_direction data_type signed range port_name, port_name, ... ;</code>	
Old Style Declarations <code>port_direction signed range port_name, port_name, ... ;</code> <code>data_type_declarations (see section 6.0)</code>	

- *range* (optional) is a range from `[msb :lsb]` (most-significant-bit to least-significant-bit).
 - If no range is specified, ports are 1-bit wide.

```
module mux4to1 (
    input wire [1:0] sel, // 2-bit selection input
    input wire a,         // 1-bit data input a
    input wire b,         // 1-bit data input b
    input wire c,         // 1-bit data input c
    input wire d,         // 1-bit data input d
    output reg y          // 1-bit output
);
    always @* begin
        case(sel)
            2'b00: y = a; // If sel is 00, output a
            2'b01: y = b; // If sel is 01, output b
            2'b10: y = c; // If sel is 10, output c
            2'b11: y = d; // If sel is 11, output d
        endcase
    end
endmodule
```

Mux 4 to 1 Tetbench

```
iverilog mux4to1_tb.v; vvp .\a.out
Time=0 sel=00 a=0 b=1 c=1 d=1 y=0
Time=10 sel=01 a=0 b=1 c=1 d=1 y=1
Time=20 sel=10 a=0 b=1 c=1 d=1 y=1
Time=30 sel=11 a=0 b=1 c=1 d=1 y=1
```

```
`timescale 1ns / 1ns
`include "mux4to1.v"
module tb_mux4to1;
    reg [1:0] sel;
    reg a, b, c, d;
    wire y;
    mux4to1 M0 (
        .sel(sel), .a(a), .b(b), .c(c), .d(d),
        .y(y)
    );
    initial begin
        $monitor("Time=%0t sel=%b a=%b b=%b
c=%b d=%b y=%b", $time, sel, a, b, c, d, y);
        a = 1'b0; b = 1'b1; c = 1'b1; d = 1'b1;
        sel = 2'b00; #10;
        sel = 2'b01; #10;
        sel = 2'b10; #10;
        sel = 2'b11; #10;
        $finish;
    end
endmodule
```

4bit adder

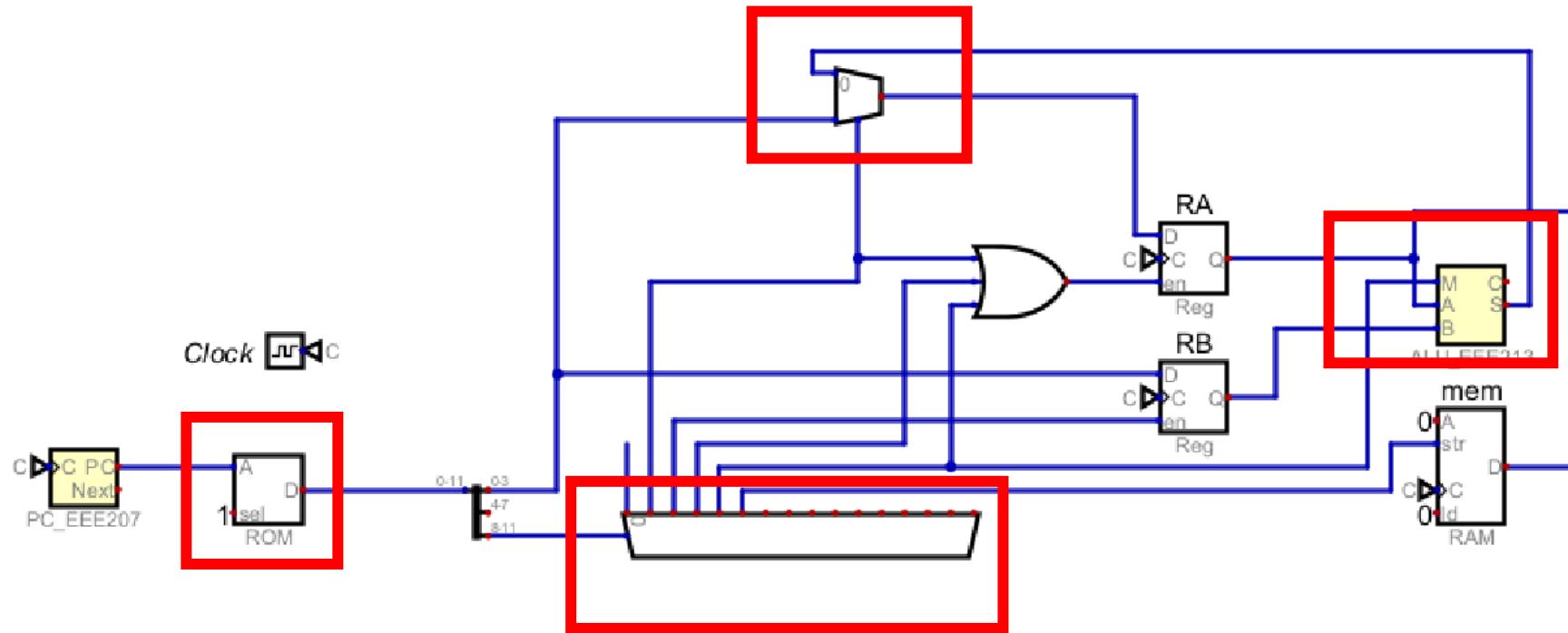
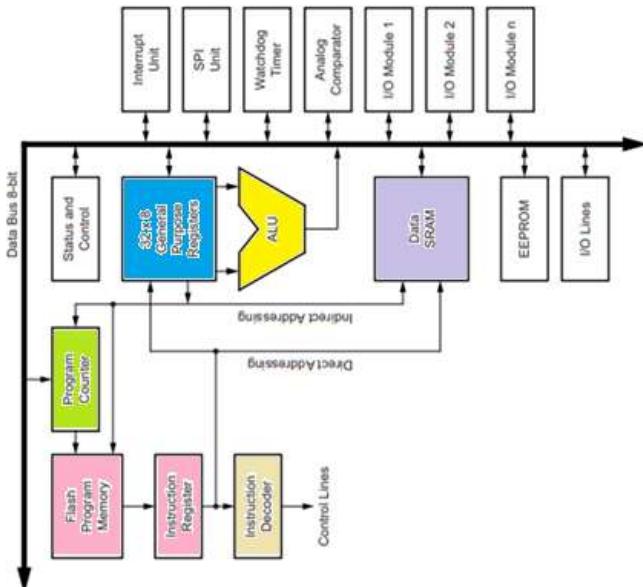
```
module adder4bit (
    input wire [3:0] a,          // 4-bit input a
    input wire [3:0] b,          // 4-bit input b
    input wire cin,             // Carry-in input
    output wire [3:0] sum,       // 4-bit sum output
    output wire cout            // Carry-out output
);
    // Calculate the sum and carry out
    assign {cout, sum} = a + b + cin;
endmodule
```

4bit adder testbench

```
iverilog adder4bit_tb.v; vvp .\a.out
Time=0 a= 2 b= 3 cin=0 sum= 5 cout=0
Time=10 a= 9 b= 8 cin=0 sum= 1 cout=1
```

```
`timescale 1ns / 1ns
`include "adder4bit.v"
module tb_adder4bit;
    reg [3:0] a, b;
    reg cin;
    wire [3:0] sum;
    wire cout;
    adder4bit M0 (
        .a(a),
        .b(b),
        .cin(cin),
        .sum(sum),
        .cout(cout)
    );
    initial begin
        $monitor("Time=%0t a=%d b=%d cin=%b
sum=%d cout=%b", $time, a, b, cin, sum, cout);
        a = 4'd2; b = 4'd3; cin = 1'b0; #10;
        a = 4'd9; b = 4'd8; cin = 1'b0; #10;
        $finish;
    end
endmodule
```

MCU Blocks



Combinational

- MUX
 - Decoder
 - ROM
 - Async RAM
 - ALU

Sequential

- Program Counter
 - Registers
 - Synch RAM

2 to 1 MUX

```
`timescale 1 ns / 1 ns
`include "mux21_case.v"
module testbench;
    reg a,b,sel; //inputs
    wire y; //outputs
//intantiate
mux_2to1_case M0
(.sel(sel),.a(a),.b(b),.y(y));
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("a b:sel y");
    $monitor("%0t %b %b:%b %b", $time,
a,b,sel,y);
    #10 a= 0; b= 1; sel = 1;
    #10 sel = 0;
    #10 $finish;
end
endmodule
```

```
module mux_2to1_case (
    input wire a,           // Input 0
    input wire b,           // Input 1
    input wire sel,         // Select line
    output reg y            // Output
);
    always @* begin
        case (sel)
            1'b0: y = a;    // sel is 0
            1'b1: y = b;    // sel is 1
        endcase
    end
endmodule
```

```
iverilog .\mux21_case_tb.v; vvp .\a.out
a b:sel y
0 x x:x x
10 0 1:1 1
20 0 1:0 0
```

4 bit decoder

```
module decoder4to16 (
    input wire [3:0] in,          // 4-bit input
    output reg out0,              // 16 individual
    1-bit outputs
    output reg out1,
    output reg out2,
    output reg out3,
    output reg out4,
    output reg out5,
    output reg out6,
    output reg out7,
    output reg out8,
    output reg out9,
    output reg out10,
    output reg out11,
    output reg out12,
    output reg out13,
    output reg out14,
    output reg out15
);
always @* begin
    // Initialize all outputs to 0
    out0 = 1'b0;
    out1 = 1'b0;
    out2 = 1'b0;
```

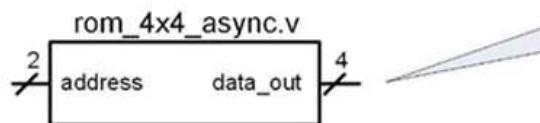
4 bit decoder Testbench

```
iverilog ./fourbitdecoder_tb.v; vvp a.out
Time=0 in= 0 out=0000000000000001
Time=10 in= 1 out=0000000000000010
Time=20 in=14 out=0100000000000000
Time=30 in=15 out=1000000000000000
```

```
`timescale 1ns / 1ns
`include "fourbitdecoder.v"
module tb_decoder4to16;
    reg [3:0] in; //inputs
    wire out0, out1, out2, out3, out4, out5,
    out6, out7, out8, out9;
    wire out10, out11, out12, out13, out14,
    out15;
    decoder4to16 M0 (
        .in(in),
        .out0(out0), .out1(out1), .out2(out2),
        .out3(out3), .out4(out4), .out5(out5),
        .out6(out6), .out7(out7), .out8(out8),
        .out9(out9), .out10(out10), .out11(out11),
        .out12(out12), .out13(out13),
        .out14(out14), .out15(out15)
    );
    initial begin
        $monitor("Time=%0t in=%d
out=%b%b%b%b%b%b%b%b%b%b%b%b%b",
        $time,
        in,
                    out15, out14, out13, out12,
        out11, out10, out9, out8, out7, out6, out5,
        out4, out3, out2, out1, out0);
        // Apply test inputs
```

ROM Initialized

Example: Behavioral Models of a 4x4 Asynchronous Read Only Memory in Verilog



ROM contents for this example:

Address	Data
0	1 1 1 0
1	0 0 1 0
2	1 1 1 1
3	0 1 0 0

```
module rom (
    input wire [3:0] address, // 4-bit address input
    output reg [15:0] data_out // 16-bit data output
);
    reg [15:0] mem [0:15];
    initial begin
        mem[0] = 16'b1110000100000010;
        mem[1] = 16'b1110001100010100;
        mem[2] = 16'b0001111100000001;
        mem[3] = 16'b1001000000000010;
        mem[4] = 16'b0;
        mem[5] = 16'b0;
        mem[6] = 16'b0;
        mem[7] = 16'b0;
        mem[8] = 16'b0;
        mem[9] = 16'b0;
        mem[10] = 16'b0;
        mem[11] = 16'b0;
        mem[12] = 16'b0;
        mem[13] = 16'b0;
        mem[14] = 16'b0;
        mem[15] = 16'b0;
    end
    always @* begin
        data_out = mem[address];
    end
endmodule
```

- compiled code in binary stored in ROM/Flash
- program counter is the address of the ROM

ROM Initialized Testbench

- compiled code in binary stored in ROM/Flash
- program counter is the address of the ROM

```
iverilog rom_init_tb.v; vvp .\a.out
```

```
Time=0 pcounter= 0 data_out=1110000100000010
Time=10 pcounter= 1 data_out=1110001100010100
Time=20 pcounter= 2 data_out=0001111100000001
Time=30 pcounter= 3 data_out=1001000000000010
Time=40 pcounter= 4 data_out=0000000000000000
```

```
`timescale 1ns / 1ns
`include "rom_init.v"
module tb_rom;
    reg [3:0] pcounter;
    wire [15:0] data_out;
    rom M0 (
        .address(pcounter),
        .data_out(data_out)
    );
    initial begin
        $monitor("Time=%0t pcounter=%d
data_out=%b", $time, pcounter, data_out);
    end
    initial begin
        // Read and print data from the first 5
        addresses using pcounter
        pcounter = 4'd0; #10;
        pcounter = 4'd1; #10;
        pcounter = 4'd2; #10;
        pcounter = 4'd3; #10;
        pcounter = 4'd4; #10;
        $finish;
    end
endmodule
```

ROM from File

```
module rom (
    input wire [3:0] address, // 4-bit
address input
    output reg [15:0] data_out // 16-bit
data output
);
    // Memory array (16 locations of 16-
bit data)
    reg [15:0] mem [0:15];
    // Initialize memory from file
    initial begin
        $readmemb("program.txt", mem);
    end
    always @* begin
        data_out = mem[address];
    end
endmodule
```

ROM from File Testbench

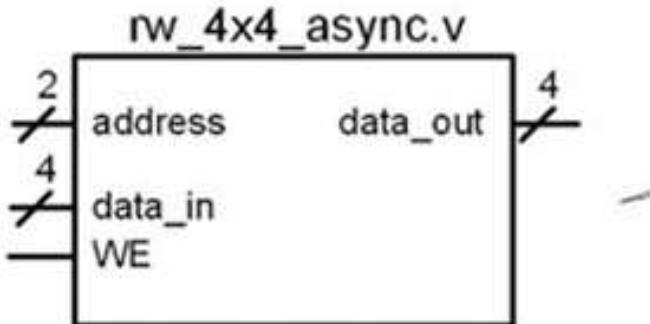
```
iverilog rom_tb.v; vvp .\a.out
Time=0 pcounter= 0 data_out=1110000100000010
Time=10 pcounter= 1 data_out=1110001100010100
Time=20 pcounter= 2 data_out=0001111100000001
Time=30 pcounter= 3 data_out=1001000000000010
Time=40 pcounter= 4 data_out=0000000000000000
```

```
`timescale 1ns / 1ns
`include "rom.v"
module tb_rom;
    reg [3:0] pcounter;
    wire [15:0] data_out;

    rom M0 (
        .address(pcounter),
        .data_out(data_out)
    );
    initial begin
        $monitor("Time=%0t pcounter=%d
data_out=%b", $time, pcounter, data_out);
    end
    initial begin
        pcounter = 4'd0; #10;
        pcounter = 4'd1; #10;
        pcounter = 4'd2; #10;
        pcounter = 4'd3; #10;
        pcounter = 4'd4; #10;
        $finish;
    end
endmodule
```

Async RAM

- We can WRITE/READ to RAM
- Address is the input
- Asynchronous DRAM (ADRAM)
- example arduino
- Synchronous DRAM (SDRAM)
- notebooks



```
module memory (
    input wire we,           // Write enable
    input wire [4:0] address, // Address input (5-bit)
    input wire [15:0] data_in, // Data input (16-bit)
    output reg [15:0] data_out // Data output (16-bit)
);
    // Memory array (32 address of 16-bit data)
    reg [15:0] mem [31:0];
    always @* begin
        if (we) begin
            // Write operation
            mem[address] = data_in;
            data_out = 16'bzz; // high-impedance
        end else begin
            // Read operation
            data_out = mem[address];
        end
    end
endmodule
```

Contents to be written:

Address	Data
0	1 1 1 0
1	0 0 1 0
2	1 1 1 1
3	0 1 0 0

Async RAM Test Bench

- READ or WRITE to RAM by using WE write enable

```
`timescale 1ns / 1ns
`include "memory.v"
module tb_memory;
    reg we;
    reg [4:0] address;
    reg [15:0] data_in;
    wire [15:0] data_out;
    memory M0 (
        .we(we),
        .address(address),
        .data_in(data_in),
        .data_out(data_out)
    );
    initial begin
        $monitor("Time=%0t we=%b address=%d data_in=%b
data_out=%b", $time, we, address, data_in, data_out);
        // Write data to addresses
        we = 1'b1;
        address = 5'd0; data_in = 16'b1110000100000010; #10;
        address = 5'd1; data_in = 16'b1110001100010100; #10;
        address = 5'd2; data_in = 16'b0001111100000001; #10;
        address = 5'd3; data_in = 16'b100100000000010; #10;
        // Switch to read mode
        we = 1'b0;
```

Async RAM Test Bench

- Memory is empty
- first part: write values to address
- second part: read from address and print

```
iverilog memory_tb.v; vvp .\a.out
Time=0 we=1 address= 0 data_in=1110000100000010 data_out=zzzzzzzzzzzzzz
Time=10 we=1 address= 1 data_in=1110001100010100 data_out=zzzzzzzzzzzzzzzz
Time=20 we=1 address= 2 data_in=000111100000001 data_out=zzzzzzzzzzzzzzzz
Time=30 we=1 address= 3 data_in=1001000000000010 data_out=zzzzzzzzzzzzzzzz
Time=40 we=0 address= 0 data_in=1001000000000010 data_out=1110000100000010
Time=50 we=0 address= 1 data_in=1001000000000010 data_out=1110001100010100
Time=60 we=0 address= 2 data_in=1001000000000010 data_out=0001111000000001
Time=70 we=0 address= 3 data_in=1001000000000010 data_out=1001000000000010
```

8-bit ALU for ADD, SUB

```
module eight_bit_alu(
    input [7:0] A, B,
    input M,
    output [7:0] S,
    output C);
    assign {C,S} = M ? (A - B): (A + B);
endmodule
```

```
iverilog alu8bit_tb.v; vvp .\a.out
Time=0 A= x B= x M=x S= x C=x
Time=1 A= 2 B= 3 M=0 S= 5 C=0
Time=2 A=250 B= 13 M=0 S= 7 C=1
Time=3 A= 9 B= 2 M=1 S= 7 C=0
Time=4 A= 9 B= 12 M=1 S=253 C=1
```

NOTICE that 8'd2, 4bit size DECIMAL 2

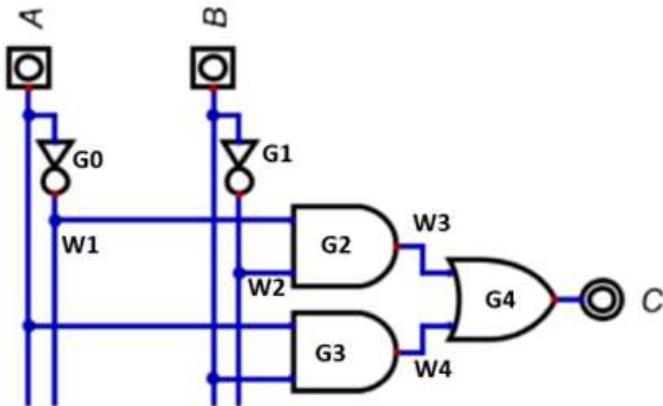
```
`timescale 1ns / 1ns
`include "alu8bit.v"
module test_four_bit_alu_tb;
reg [7:0] A, B;
reg M;
wire [7:0] S;
wire C;
eight_bit_alu M0
(.A(A),.B(B),.M(M),.S(S),.C(C));
initial begin
    $monitor("Time=%0t A=%d B=%d M=%d S=%d C=%b",
$time, A, B, M, S, C);
    #1; A=8'd2; B=8'd3; M=0;
    #1; A=8'd250; B=8'd13; M=0;
    #1; A=8'd9; B=8'd2; M=1;
    #1; A=8'd9; B=8'd12; M=1;
end
endmodule
```

? : sel?m:n conditional operator; if sel is true, return m: else return n

4.11 Literal Integer Numbers

value	unsized decimal integer
size 'base value	sized integer in a specific radix (base)

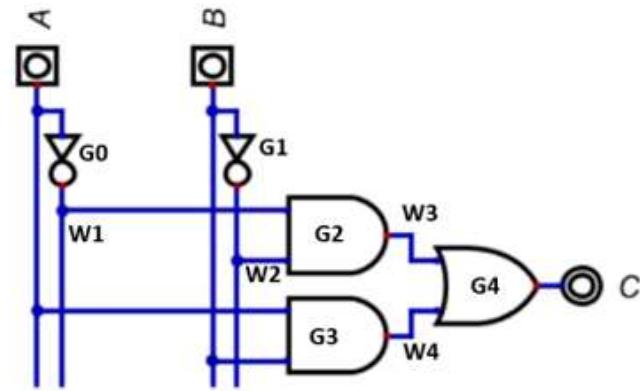
EXERCISES- Primitives



1.(20 points) Define the module for the given logic circuit?

```
module AeqB(
    output _____,
    input  _____
);
    not G0 (_____,_____);
    not G1 (_____,_____);
    and G2 (_____,_____,_____);
    and G3 (_____,_____,_____);
    or  G4 (_____,_____,_____);
endmodule
```

Gate Primitives	Terminal Order and Quantity	
and or xor	nand nor xnor	(1-output, 1-or-more-inputs)



1.(20 points) Define the module for the given logic circuit?

```
module AeqB(
    output C,
    input  A,B
);
    wire W1,W2,W3,W4;
    not G0 (W1,A);
    not G1 (W2,B);
    and G2 (W3,W1,W2);
    and G3 (W4,A,B);
    or  G4 (C,W3,W4);
endmodule
```

EXERCISES-testbench

4. (10 points) Instantiate the module AeqB() inside the testbench module.

```
//testbench
`timescale 1 ns / 1 ns
module tb_AeqB;
    reg ___,____; //reg for inputs
    wire ____; //wire for outputs
    //instantiation
    AeqB G0 (____,____,____);
```

4. (10 points) Instantiate the module AeqB() inside the testbench module.

```
//testbench
`timescale 1 ns / 1 ns
module tb_AeqB;
    reg a,b; //reg for inputs
    wire c; //wire for outputs
    //instantiation
    AeqB G0 (.C(c), .A(a), .B(b));
    // AeqB G0 (c,a,b);
```

```
module AeqB(
    output C,
    input A,B
);
    wire W1,W2,W3,W4;
    not G0 (W1,A);
    not G1 (W2,B);
    and G2 (W3,W1,W2);
    and G3 (W4,A,B);
    or G4 (C,W3,W4);
endmodule
```

EXERCISES-testbench

- **%0t time format will print time no leading spaces**
- `timescale 1 ns / 1 ns
- #10 \$finish;

```
`timescale 1ns/1ps

a b:sel y
0 x x:x x
10000 0 1:1 1
20000 0 1:0 0
```

```
`timescale 1ns/1ns

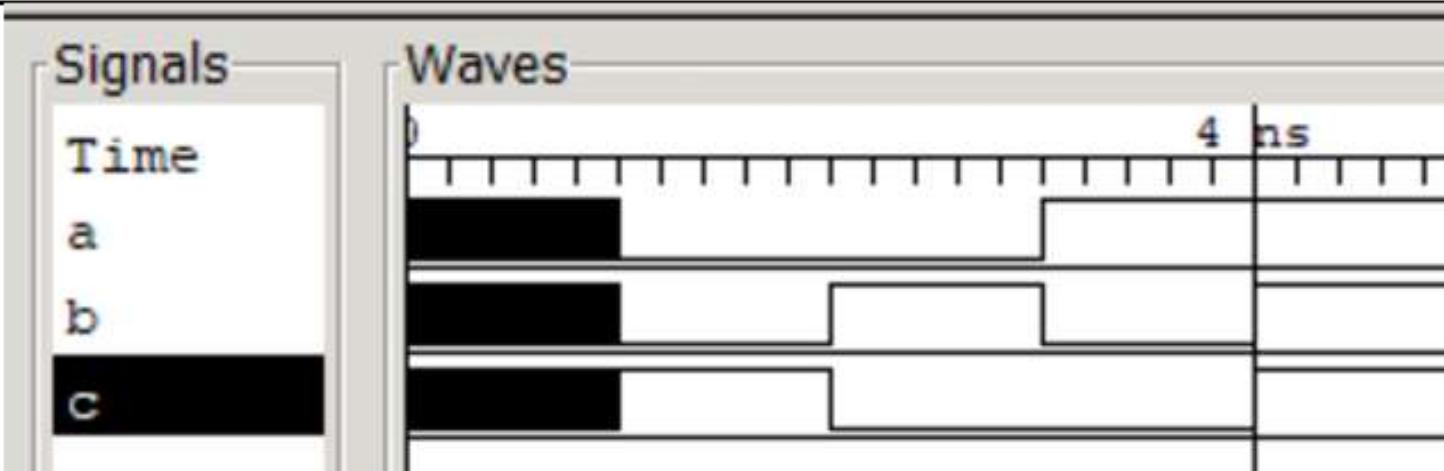
a b:sel y
0 x x:x x
10 0 1:1 1
20 0 1:0 0
```

```
`timescale 1 ns / 1 ns
`include "mux21_case.v"
module testbench;
    reg a,b,sel; //inputs
    wire y; //outputs
//instantiate
mux_2to1_case M0
(.sel(sel),.a(a),.b(b),.y(y));
initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("a b:sel y");
    $monitor("%0t %b %b:%b %b", $time,
a,b,sel,y);
    #10 a= 0; b= 1; sel = 1;
    #10 sel = 0;
    #10 $finish;
end
endmodule
```

EXERCISES-testbench

```
`timescale 1ns / 1ns
`include "alu8bit.v"
module test_four_bit_alu_tb;
reg [7:0] A, B;
reg M;
wire [7:0] S;
wire C;
eight_bit_alu M0
(.A(A),.B(B),.M(M),.S(S),.C(C));
initial begin
$monitor("Time=%0t A=%d B=%d M=%d S=%d C=%b",
$time, A, B, M, S, C);
#1; A=8'd2; B=8'd3; M=0;
#1; A=8'd250; B=8'd13; M=0;
#1; A=8'd9; B=8'd2; M=1;
#1; A=8'd9; B=8'd12; M=1;
#10 $finish;
end
endmodule
```

EXERCISES-Waveform



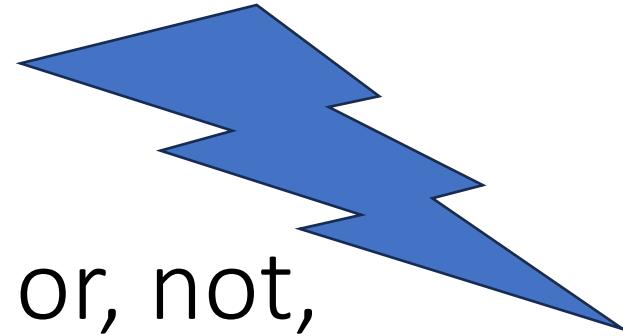
8. (10 points) What are the initial values of a, b and c in simulation

a=_____ b=_____ c=_____

What are the values of a, b and c at 4ns

a=_____ b=_____ c=_____

VERILOG MODELING



1. Gate Level Modeling: primitives (and, or, not, xor, etc), UDP
2. Dataflow Modeling : continuous assignment, logic, arithmetic and conditional operators
3. Behavioral Modeling: always @*, if-else, case,

EXERCISES- Assignment with operators

```
module sample_module(  
    input wire a, b,      // 1-bit inputs  
    output wire logic_out, // 1-bit output for logical operations  
    output wire [1:0] arith_out // 2-bit output for arithmetic operations  
);  
  
    // Logical operations  
    assign logic_out = a & b; // Logical AND  
  
    // Arithmetic operations  
    assign arith_out = a + b; // Arithmetic addition  
  
endmodule
```

EXERCISES-Conditional Assignment

```
module conditional_assignment(
    input wire a, b, sel,
    output wire out
);

    // Conditional assignment using ternary
    operator
    assign out = (sel) ? a : b;

endmodule
```

EXERCISES: always @* and if-else

```
module if_else_example(
    input wire [3:0] in1, in2, // 4-bit inputs
    output reg [3:0] max     // 4-bit output for the maximum value
);

    always @* begin
        if (in1 > in2)
            max = in1;
        else
            max = in2;
    end

endmodule
```

EXERCISES-Case

```
module mux_2to1_case (
    input wire a,           // Input 0
    input wire b,           // Input 1
    input wire sel,         // Select line
    output reg y            // Output
);
    always @* begin
        case (sel)
            1'b0: y = a;    // sel is 0
            1'b1: y = b;    // sel is 1
        endcase
    end

endmodule
```

Verilog for Sequential Circuits

- Behavior is controlled by a **positive or negative edge** that occurs on a special input, called the **clock**
- Sequential elements are the **latches** and **flip flops**
- **Flip flops** are **edge-triggered storage devices**
- **THERE IS CLOCK IN SEQUENTIAL CIRCUITS**
- **EXECUTION AT POSEDGE or NEGEDGE of the CLOCK**

Verilog Sequential Circuit Design: D-FlipFlop

- inputs are data, clock and reset. output is Q
- Sequential circuits performs actions **at clock signals.**

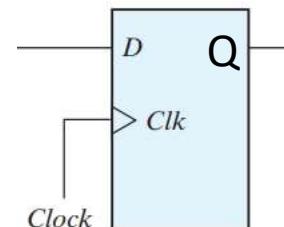
```
always @ (sensitivity list) begin  
statement;  
end
```

- Whenever the event in the sensitivity list occurs, the statement is executed

```
always @ (posedge clk)
```

- The posedge defines a rising edge (transition from 0 to 1).
- Once the clk signal rises: the value of D will be copied to Q **$Q \leq D;$**

```
// D flip-flop without reset  
'timescale 1 ns / 1 ps  
module DFF (Q, D, Clk);  
output Q;  
input D, Clk;  
reg Q;  
always @ (posedge Clk)  
Q <= D;  
endmodule
```



D-FlipFlop: Test Bench

- **instantiating** module D flip-flop inside the test bench
- generate the clock at every #1 toggle, means 2ns clock period
- **initial** blocks process statements **one time**.
- **forever** statement or statement _group An infinite loop that continuously executes the statement or statement group
- **#1 delay 1 timescale=1ns**
- Change data D at every **#10 10ns**
- **\$dumpfile("dump.vcd");** dump to file for gtkwave
- finish the simulation at **#100 ticks**, otherwise dump file can become BIG due to forever statement

■ forever loop - executes continually

```
initial begin
    clk = 0;
    forever #25 clk = ~clk;
end
```

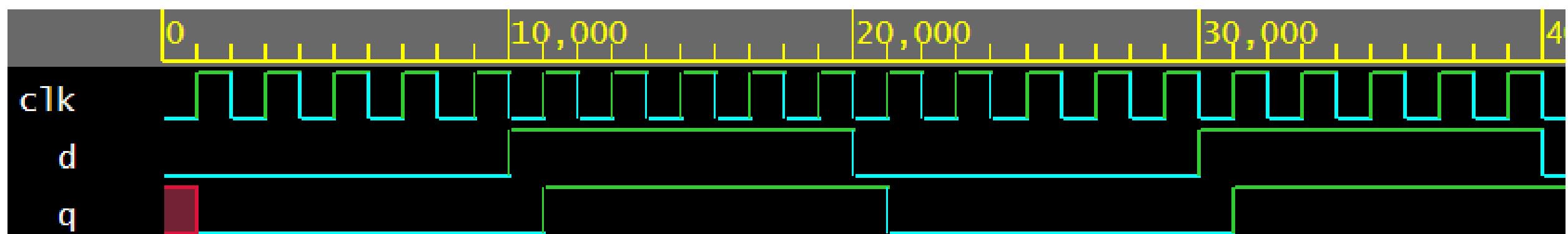
```
// Testbench
`timescale 1 ns / 1 ps
module testbench;
reg d,clk; //inputs
wire q; //outputs

// Instantiate
DFF D0 (.Clk(clk),.D(d), .Q(q));
initial begin
    clk=0;
    forever #1 clk=~clk;
end
initial begin
    d=0;
    forever #10 d=~d;
end
initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
end
initial #100 $finish;
endmodule
```

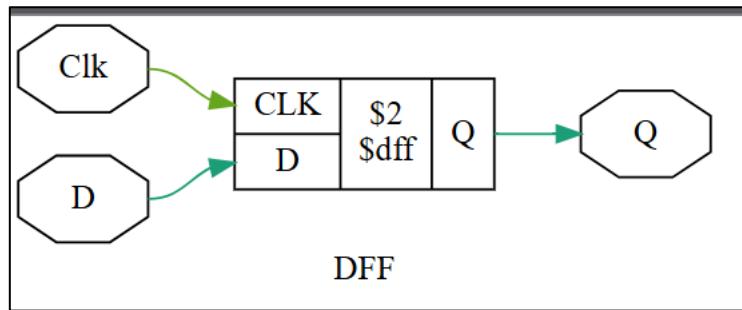
Verilog timescale

- In Verilog, the propagation delay of a gate is specified in terms of time units and by the symbol #.
- The numbers associated with time delays in Verilog are dimensionless.
- The association of a time unit with physical time is made with the ‘timescale compiler directive.
- (Compiler directives start with the (‘) back quote, or grave accent, symbol.)
- Such a directive is specified before the declaration of a module and applies to all numerical values of time in the code that follows

¹²The **timescale** directive (‘timescale 1 ns / 1 ps) specifies that the numerical values in the model are to be interpreted in units of nanoseconds, with a precision of picoseconds. This information would be used by a simulator.

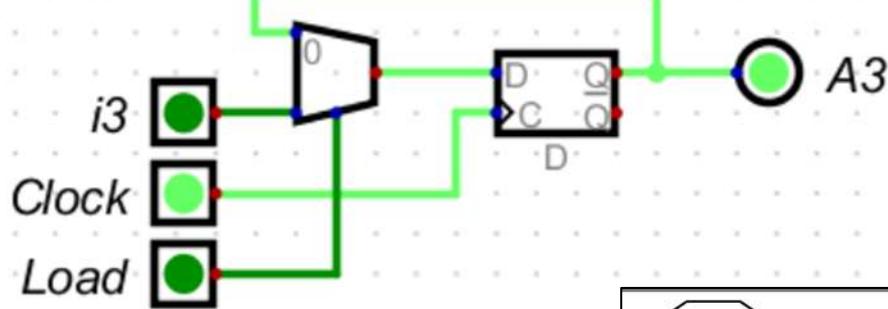


D-FlipFlop: Synthesis

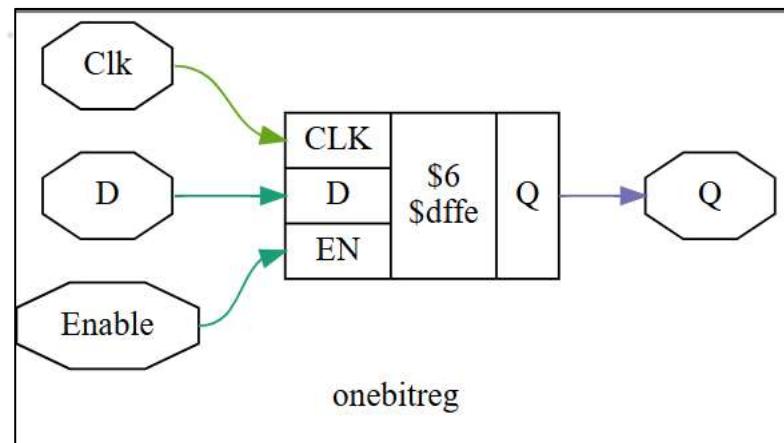


```
// D flip-flop without reset
`timescale 1 ns / 1 ps
module DFF (Q, D, Clk);
output Q;
input D, Clk;
reg Q;
always @ (posedge Clk)
Q <= D;
endmodule
```

D-FlipFlop: 1bit register



```
'timescale 1 ns / 1 ns
module onebitreg (
    output reg Q,
    input D,Enable,Clk);
    always @ (posedge Clk)
        if (Enable==1) Q <= D;
endmodule
```



D-FlipFlop: 1bit register

```
`timescale 1 ns / 1 ns
module onebitreg (
    output reg Q,
    input D,Enable,Clk);

    always @ (posedge Clk)
        if (Enable==1) Q <= D;
endmodule
```

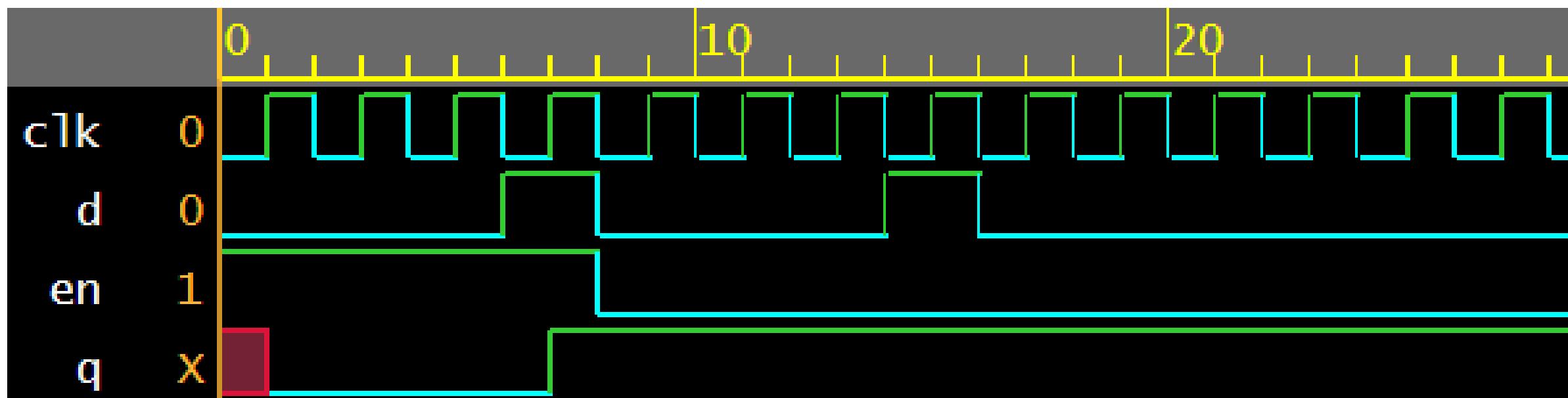
- clk will cycle 2ns
- enable=1 to write into 1bit register
- write the data=1 and disable writing
- next data will not be taken into register, register keeps 1

```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
    reg d,en,clk; //inputs
    wire q; //outputs

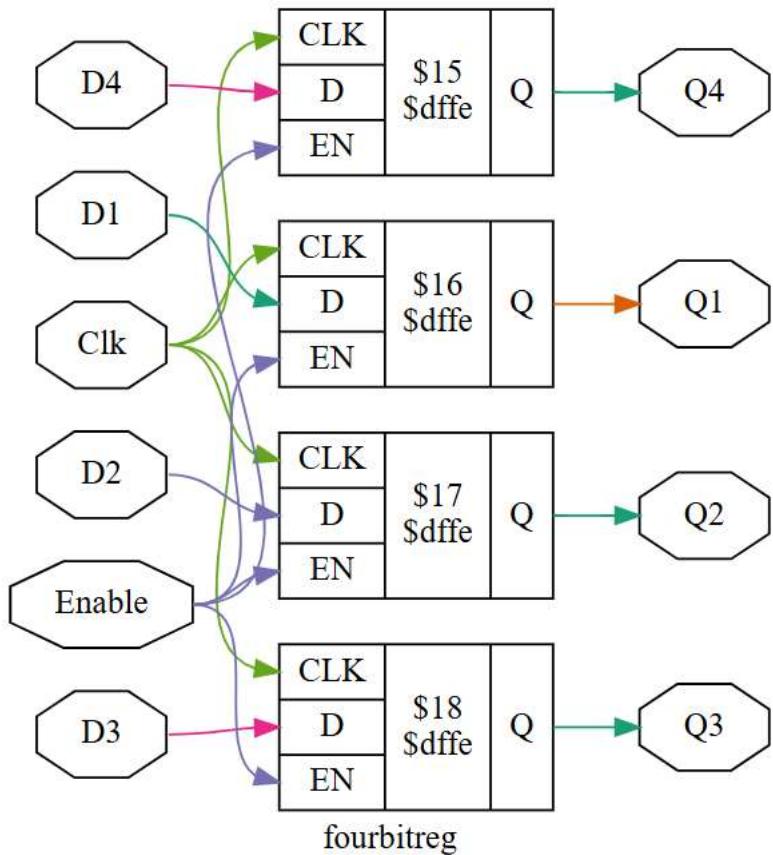
    onebitreg D0
        (.Q(q),.D(d),.Enable(en),.Clk(clk));
    initial begin
        clk=0;
        forever #1 clk=~clk;
    end
    initial begin
        d=0; en=1;
        #6 d=1;
        #2 d=0;  en=0;
        #6 d=1;
        #2 d=0;
    end
    initial begin
        $dumpfile("dump.vcd"); $dumpvars;
    end
    initial #100 $finish;
endmodule
```

D-FlipFlop: 1bit register

- clk will cycle 2ns
- enable=1 to write into 1bit register
- write the data=1 and disable writing
- next data will not be taken into register, register keeps 1



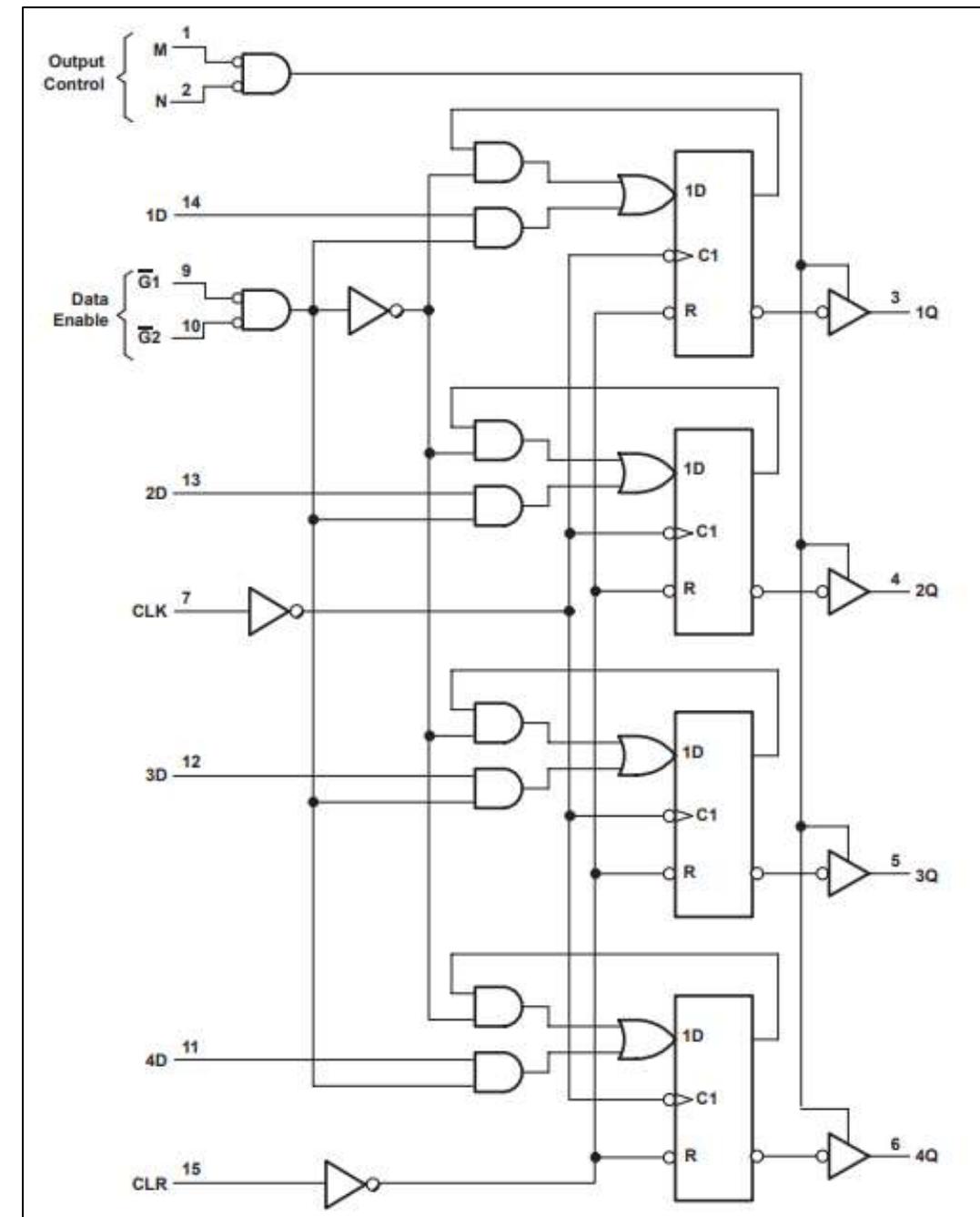
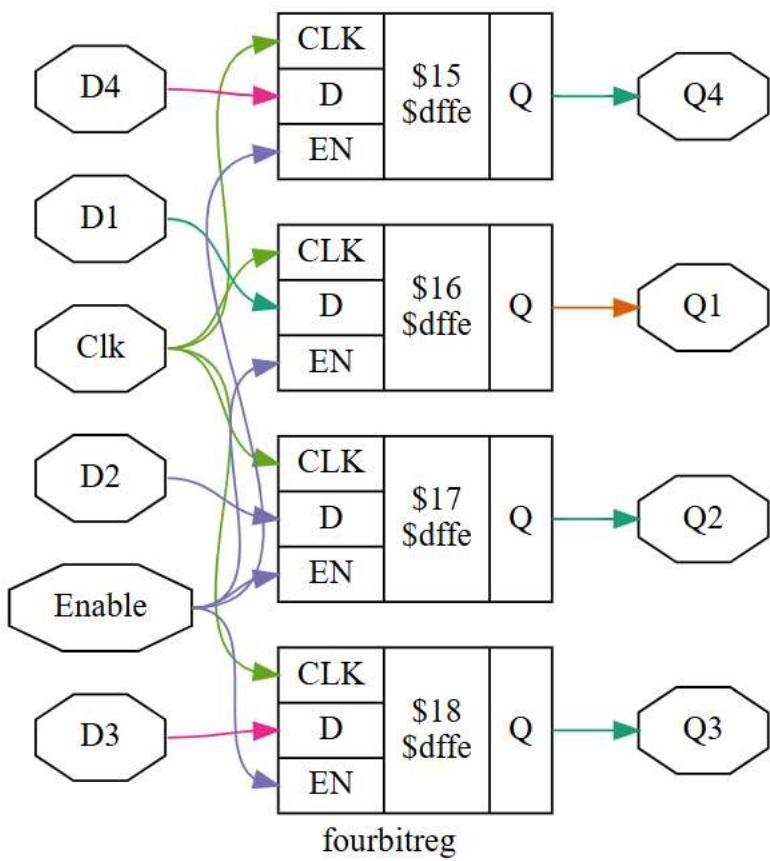
D-FlipFlop: 4bit register



```
'timescale 1 ns / 1 ns
module fourbitreg (
    output reg Q1,Q2,Q3,Q4,
    input D1,D2,D3,D4,Enable,Clk);

    always @ (posedge Clk)
        if (Enable==1) begin
            Q1 <= D1;
            Q2 <= D2;
            Q3 <= D3;
            Q4 <= D4;
        end
    endmodule
```

SN54173, 4-BIT D-TYPE REGISTERS



SN54173, 4-BIT D-TYPE REGISTERS

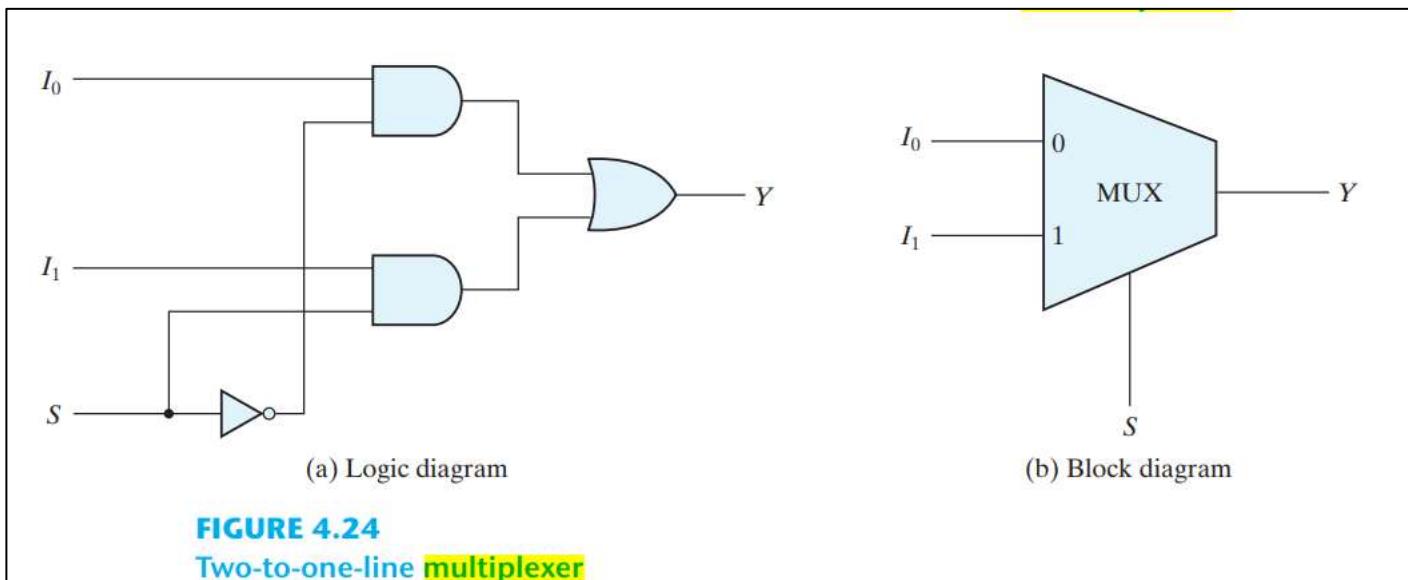
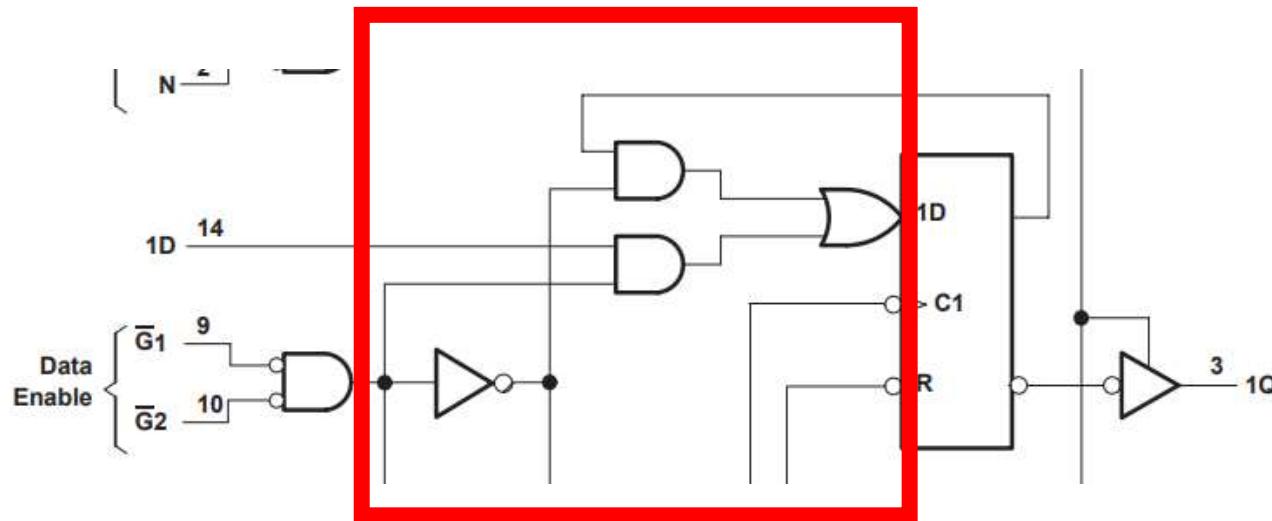
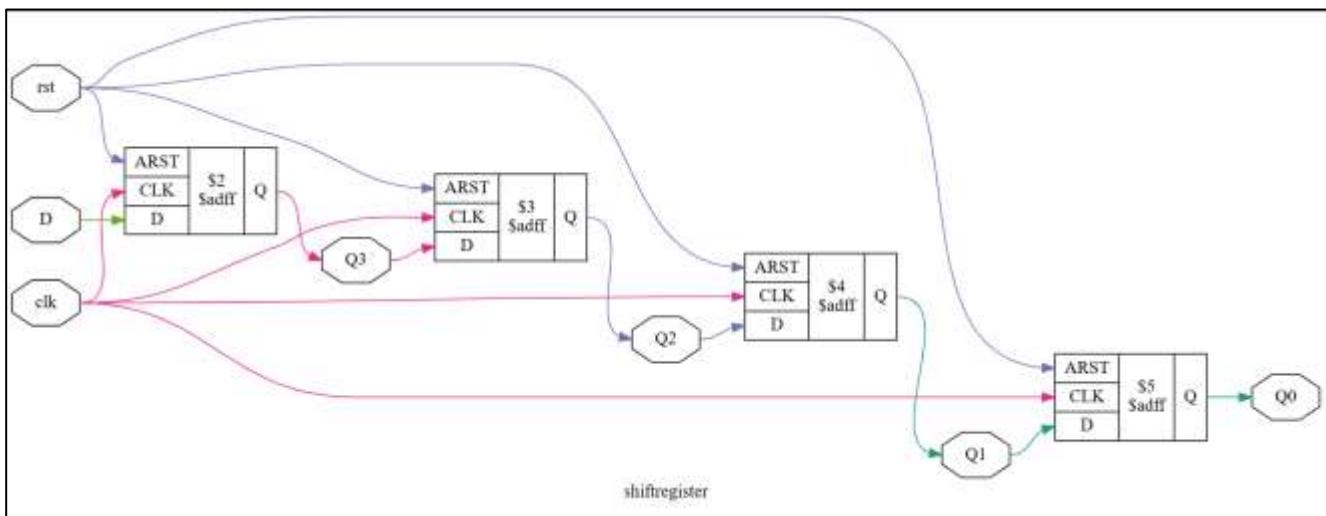


FIGURE 4.24
Two-to-one-line multiplexer

Shift Registers

- input data is serial and bit is shifted right at each clk posedge
- $D \rightarrow Q_3\ Q_2\ Q_1\ Q_0$



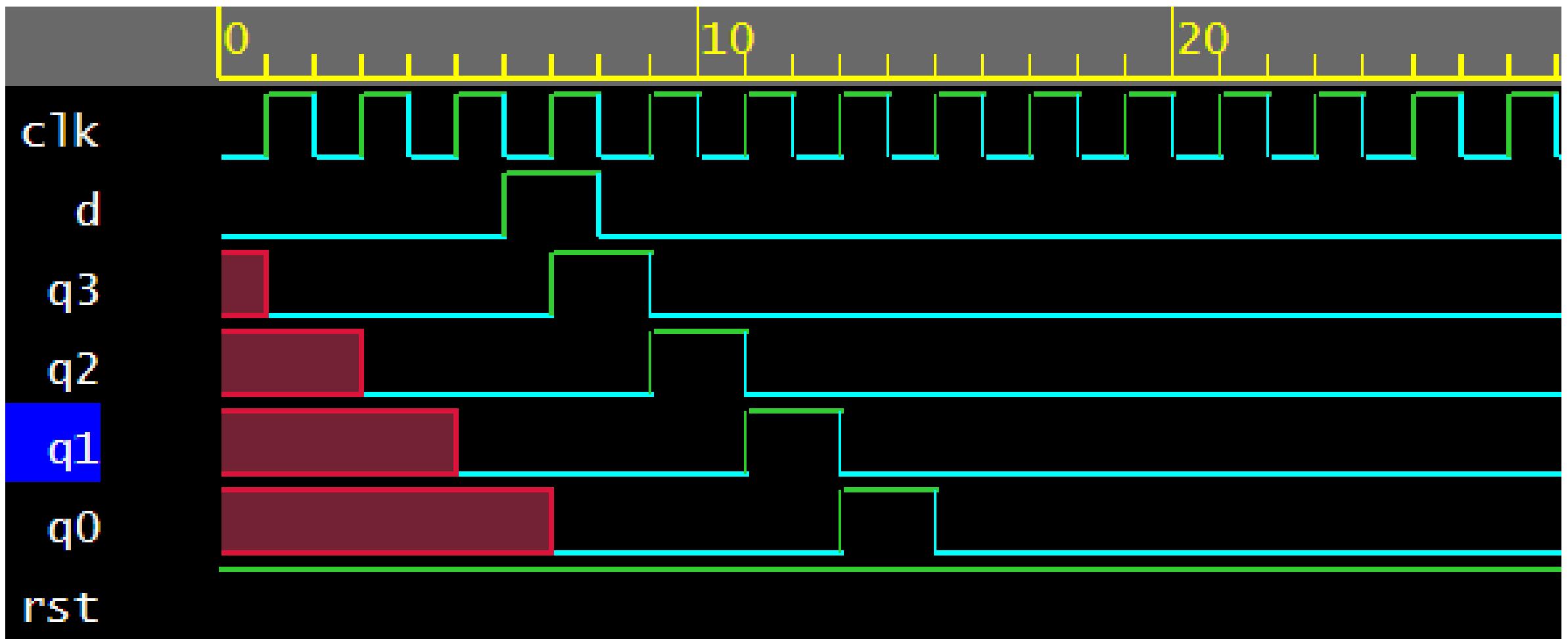
```
'timescale 1 ns / 1 ns
module shiftregister (
    output reg Q3,Q2,Q1,Q0, // Register output
    input      D, // Serial data
    clk, rst // clock and reset
);
    always @ (posedge clk, posedge rst)
        if (rst) begin
            Q3 <= 1'b0;
            Q2 <= 1'b0;
            Q1 <= 1'b0;
            Q0 <= 1'b0;
        end
        else begin
            Q3 <= D;
            Q2 <= Q3;
            Q1 <= Q2;
            Q0 <= Q1;
        end
    endmodule
```

Shift Registers

```
'timescale 1 ns / 1 ns
module shiftregister (
output reg Q3,Q2,Q1,Q0, // Register output
input D, // Serial data
clk, rst // clock and reset
);
always @ (posedge clk, negedge rst)
if (rst == 0) begin
Q3 <= 1'b0;
Q2 <= 1'b0;
Q1 <= 1'b0;
Q0 <= 1'b0;
end
else begin
Q3 <= D;
Q2 <= Q3;
Q1 <= Q2;
Q0 <= Q1;
end
endmodule
```

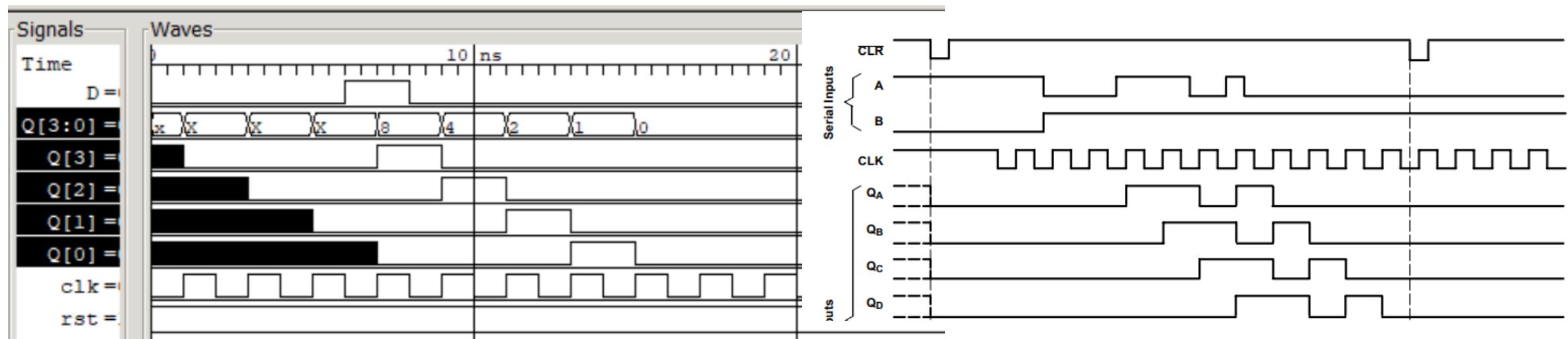
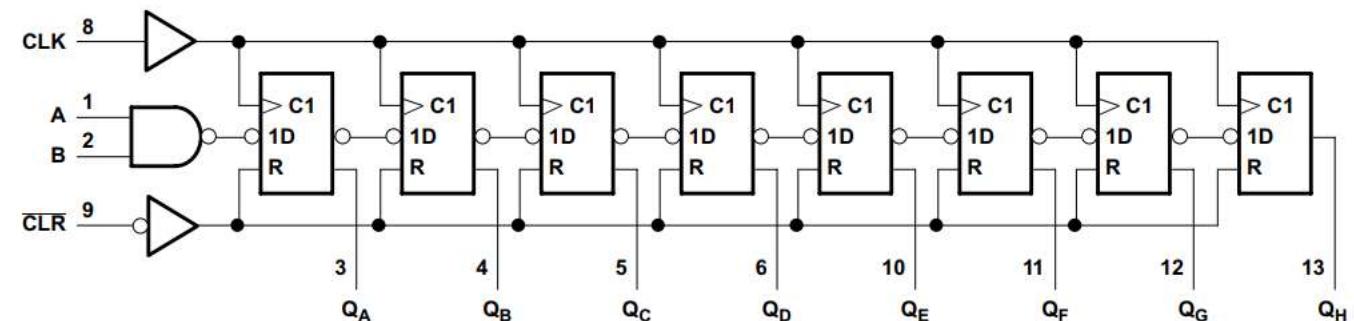
```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
reg clk, rst; //inputs
reg d; //inputs
wire q3,q2,q1,q0; //outputs
// Instantiate
shiftregister D0 (.clk(clk),.D(d), .Q3(q3),.Q2(q2),.Q1(q1),.Q0(q0), .rst(rst));
initial begin
clk=0;
forever #1 clk=~clk;
end
initial begin
d=0;  rst=1;
#6 d=1;
#2 d=0;
end
initial begin
$dumpfile("dump.vcd");
$dumpvars;
end
initial #100 $finish;
endmodule
```

Shift Registers



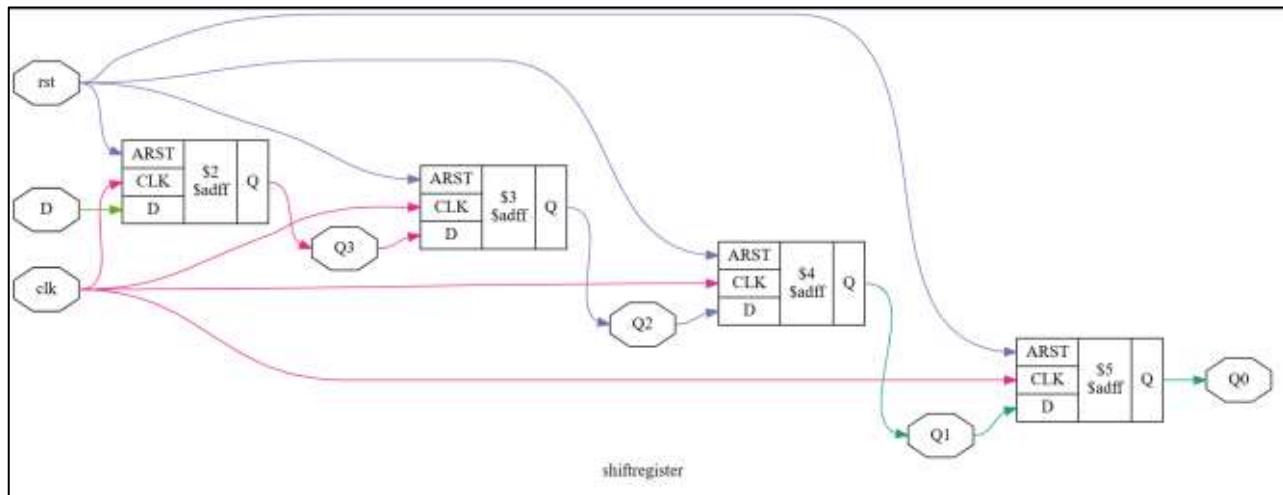
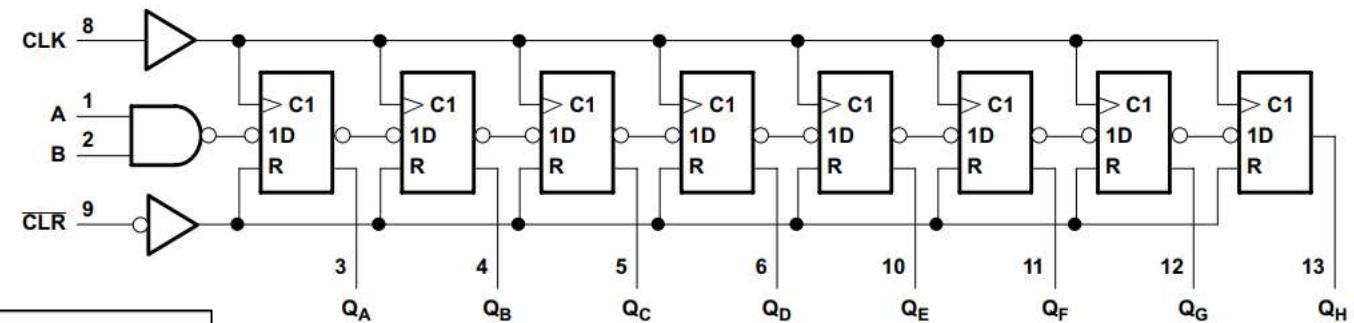
SNx4HC164 8-Bit Parallel-Out Serial Shift Registers

- lets look at datasheet of a real shift register
- we have similar waveform



SNx4HC164 8-Bit Parallel-Out Serial Shift Registers

- lets look at datasheet of a real shift register
- output of the first DFF feed into Data of the next



Blocking and Nonblocking Assignments

Blocking vs. Nonblocking Assignments

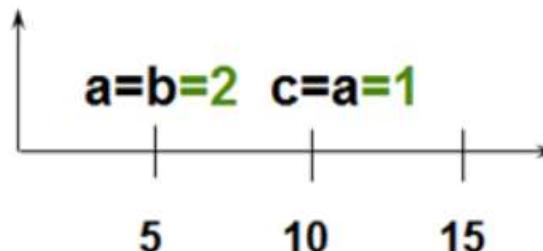
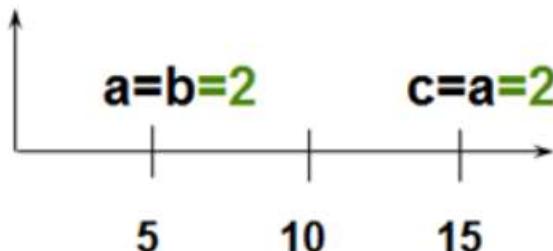
Blocking (=)

```
initial  
begin  
    a = #5 b;  
    c = #10 a;  
end
```

Nonblocking (<=)

```
initial  
begin  
    a <= #5 b;  
    c <= #10 a;  
end
```

Assuming initially a=1 and b=2



```
initial begin  
    a=1;b=0;  
    //non-blocking  
    a<=#5b;  
    c<=#10a;  
    //blocking  
    //a=#5b;  
    //c=#10a;  
end
```

Blocking and Nonblocking Assignments

Blocking vs. Nonblocking Assignments

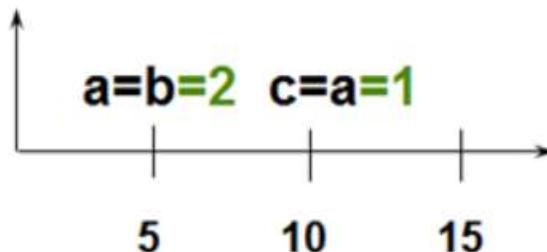
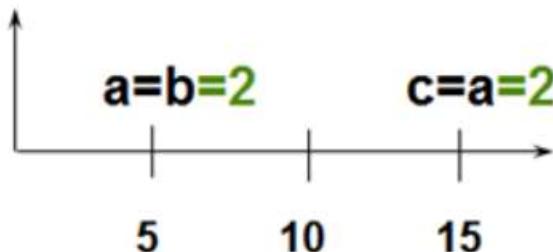
Blocking (=)

```
initial  
begin  
    a = #5 b;  
    c = #10 a;  
end
```

Nonblocking (<=)

```
initial  
begin  
    a <= #5 b;  
    c <= #10 a;  
end
```

Assuming initially a=1 and b=2



```
initial begin  
    a=1;b=0;  
    //non-blocking  
    a<=#5b;  
    c<=#10a;  
    //blocking  
    //a=#5b;  
    //c=#10a;  
end
```

time=0 a=1 b=0 c=x
time=5 a=0 b=0 c=x
time=10 a=0 b=0 c=1

Blocking and Nonblocking Assignments

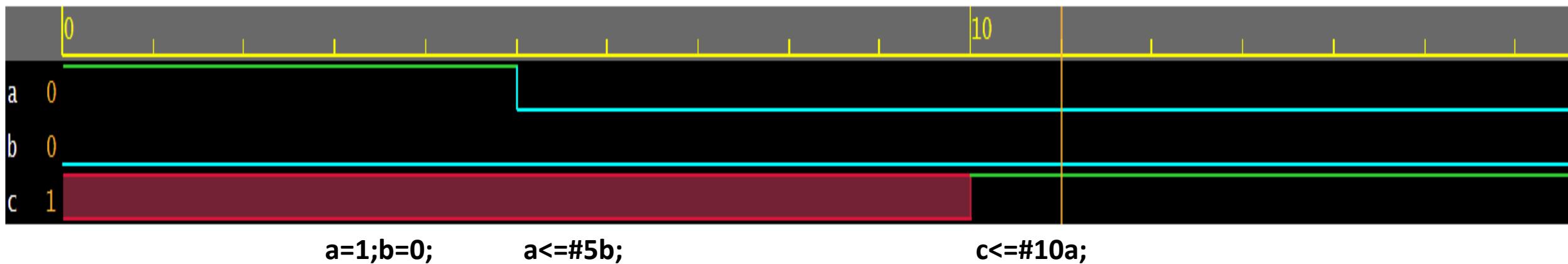
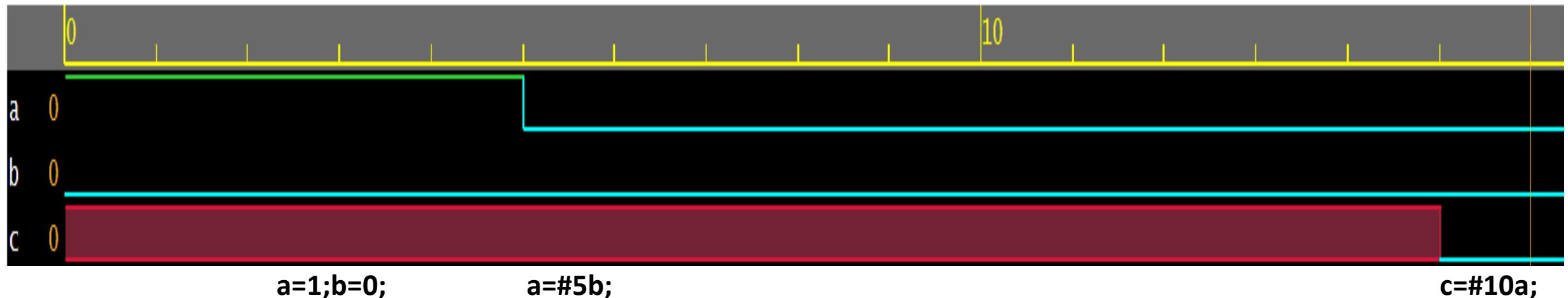
```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
  reg a,b,c; //inputs

initial begin
  a=1;b=0;
  a=#5b;
  c=#10a;
end
initial begin
  $dumpfile("dump.vcd");
  $dumpvars;
  end
initial #20 $finish;
endmodule
```

```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
  reg a,b,c; //inputs

initial begin
  a=1;b=0;
  a<=#5b;
  c<=#10a;
end
initial begin
  $dumpfile("dump.vcd");
  $dumpvars;
  end
initial #20 $finish;
endmodule
```

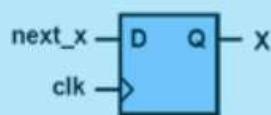
Blocking and Nonblocking Assignments



Blocking and Nonblocking Assignments

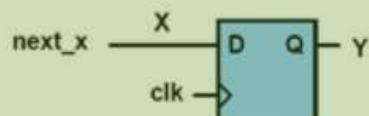
Blocking vs. Nonblocking Assignments

```
always @ ( posedge clk )
begin
  x = next_x;
end
```



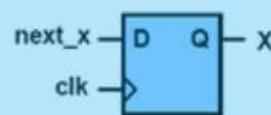
Same Behavior

```
always @ ( posedge clk )
begin
  x = next_x;
  y = x;
end
```

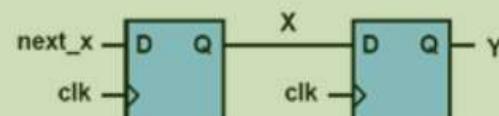


Different Behavior

```
always @ ( posedge clk )
begin
  x <= next_x;
end
```



```
always @ ( posedge clk )
begin
  x <= next_x;
  y <= x;
end
```

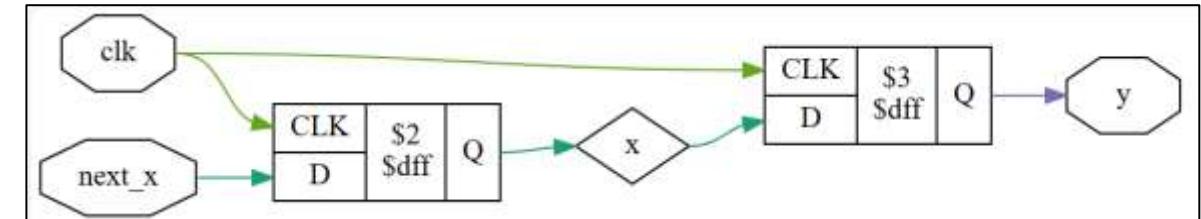
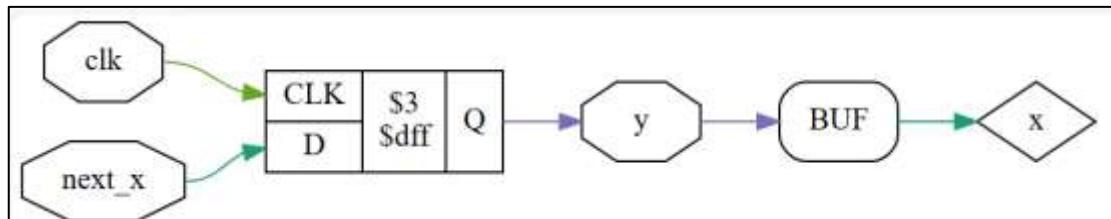


wrong use of blocking and non-blocking statements may cause synthesis to generate different logic circuits

Blocking and Nonblocking Assignments

```
module blocking(  
    input next_x, clk,  
    output y  
);  
    always @ (posedge clk)  
        begin  
            x=next_x;  
            y=x;  
        end  
    endmodule
```

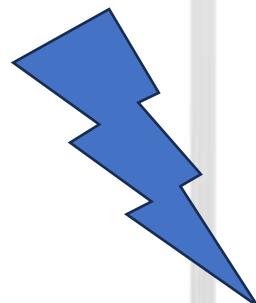
```
module blocking(  
    input next_x, clk,  
    output y  
);  
    always @ (posedge clk)  
        begin  
            x<=next_x;  
            y<=x;  
        end  
    endmodule
```



Blocking and Nonblocking Assignments

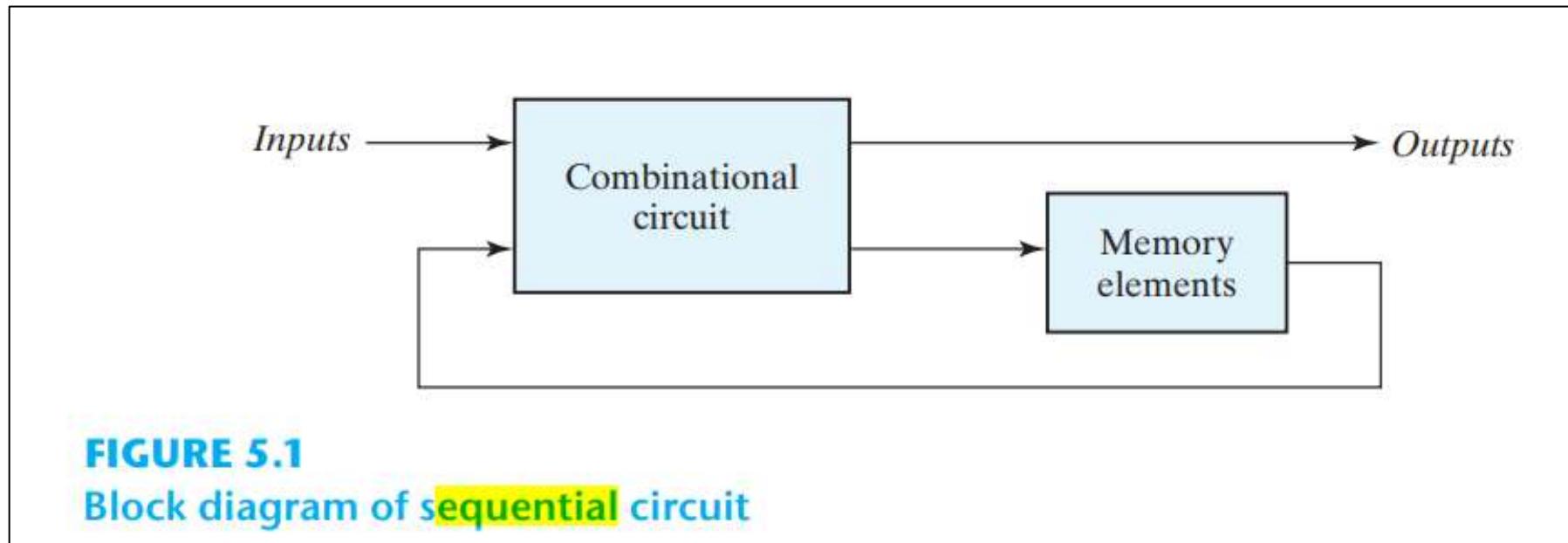
Blocking/Nonblocking Rule of Thumb

- Use blocking operator ($=$) for combinatorial logic
- Use nonblocking operator ($<=$) for sequential logic
- This avoids confusion and unintended hardware implementations during RTL synthesis



Where to GO NOW

- We created module for D-Flip Flop
- NOW there are two ways to go
 1. using combinatorial design and using D-Flip Flops we can model any sequential logic circuit.
 2. we can define completely behavioral description of the circuit which have clock and data inputs
- Remember that sequential logic is combination of combinatorial and D-Flip Flops



Clock Divider

- We may need to generate signals at lower divided clock rates, example is UART transmission, FPGA clock rate in MHz but UART is 9600, 57600, 115200 etc

```
'timescale 1 ns / 1 ns
module clockdivider (input clk, output out1);
  reg [3:0] c1;
  initial c1=4'b0000;
  always @(posedge clk) begin
    c1 <= c1 + 1'b1;
  end
  assign out1 = c1[3:3];
endmodule
```

Clock Divider

```
'timescale 1 ns / 1 ns
module clockdivider (input clk, output
out1);
 reg [3:0] c1;
initial c1=4'b0000;
always @(posedge clk) begin
c1 <= c1 + 1'b1;
end
 assign out1 = c1[3:3];
endmodule
```

```
'timescale 1 ns / 1 ns
module test;
 reg clockin;
 wire clockout;

clockdivider M0 (
 .clk(clockin),
 .out1(clockout)
);

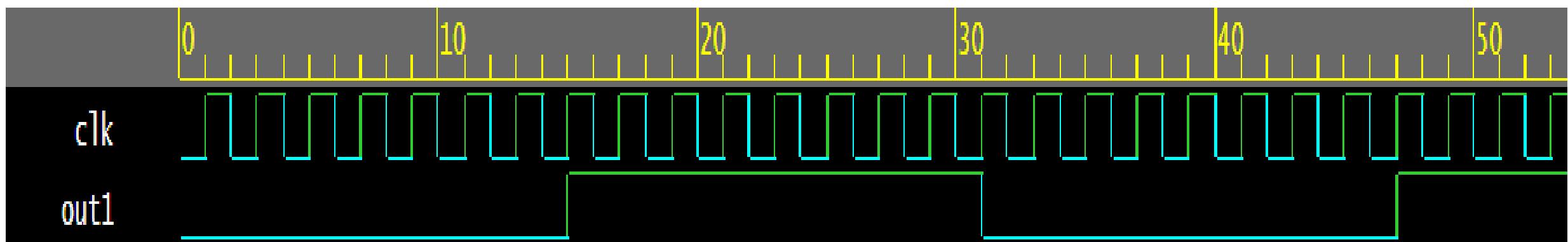
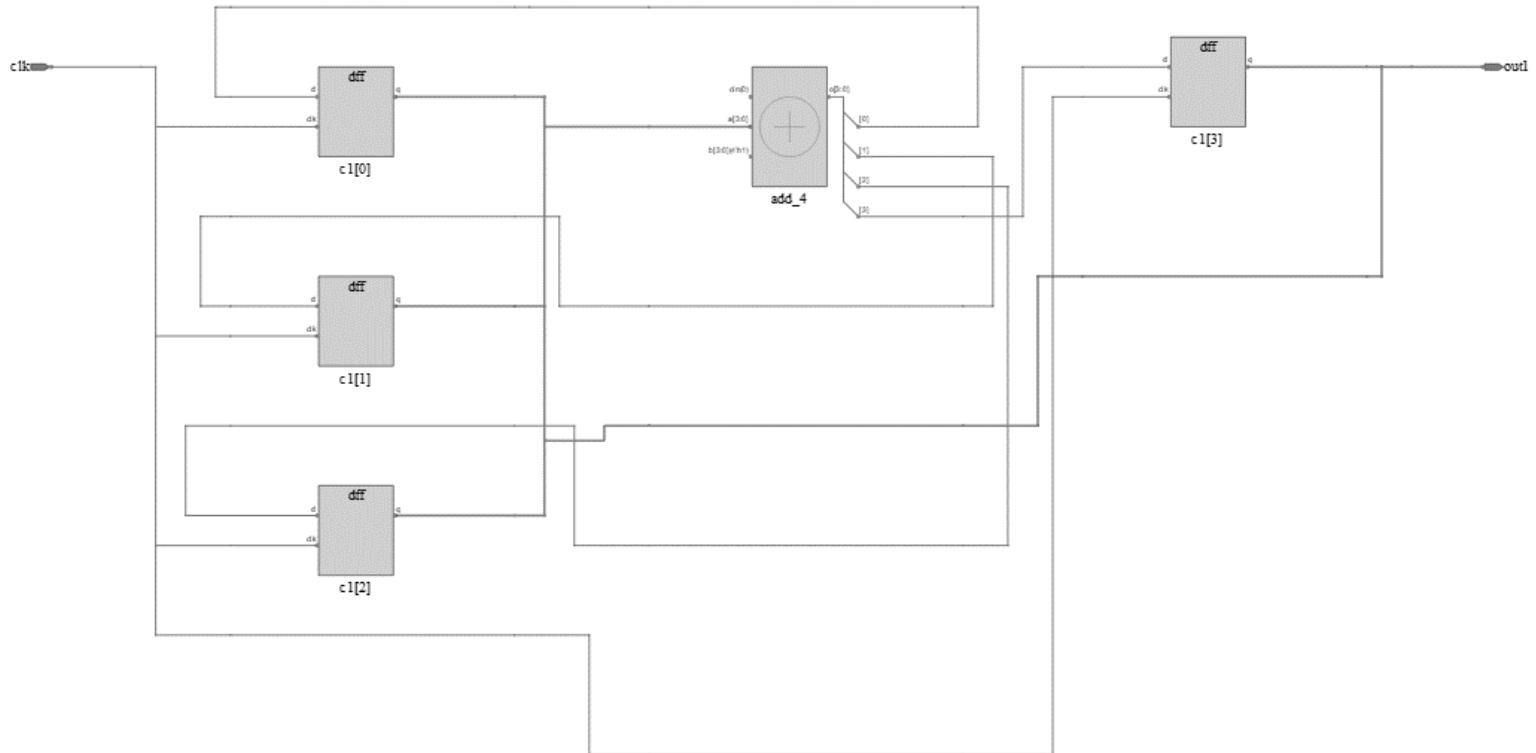
initial begin
clockin=0;
forever #1 clockin = ~clockin;
end
initial #100 $finish;

initial begin
$dumpfile("dump.vcd");
$dumpvars();
end

endmodule
```

Clock Divider

- observe that at every 8 clk, out1 toggles



Ripple Counter: Behavioral

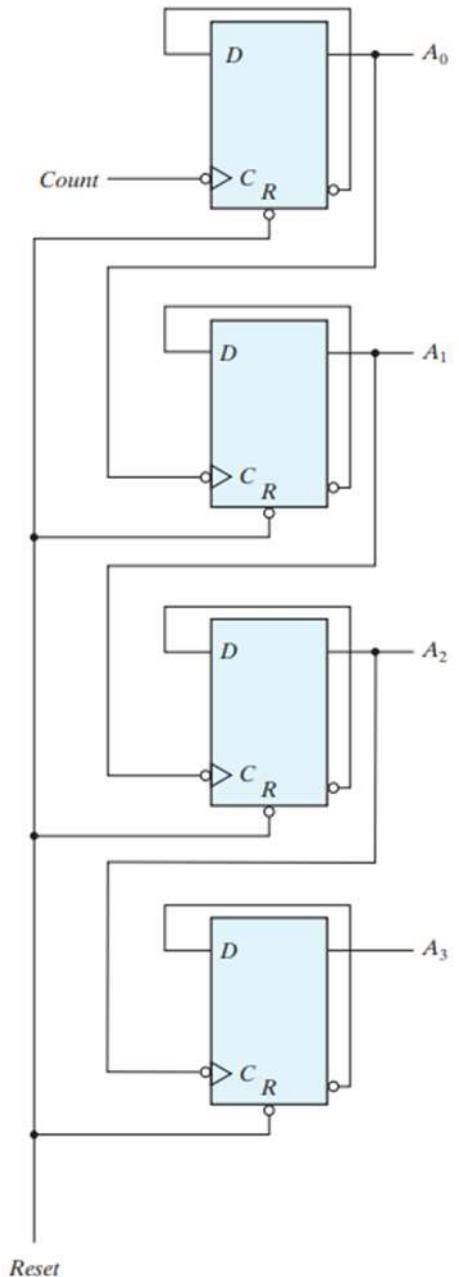
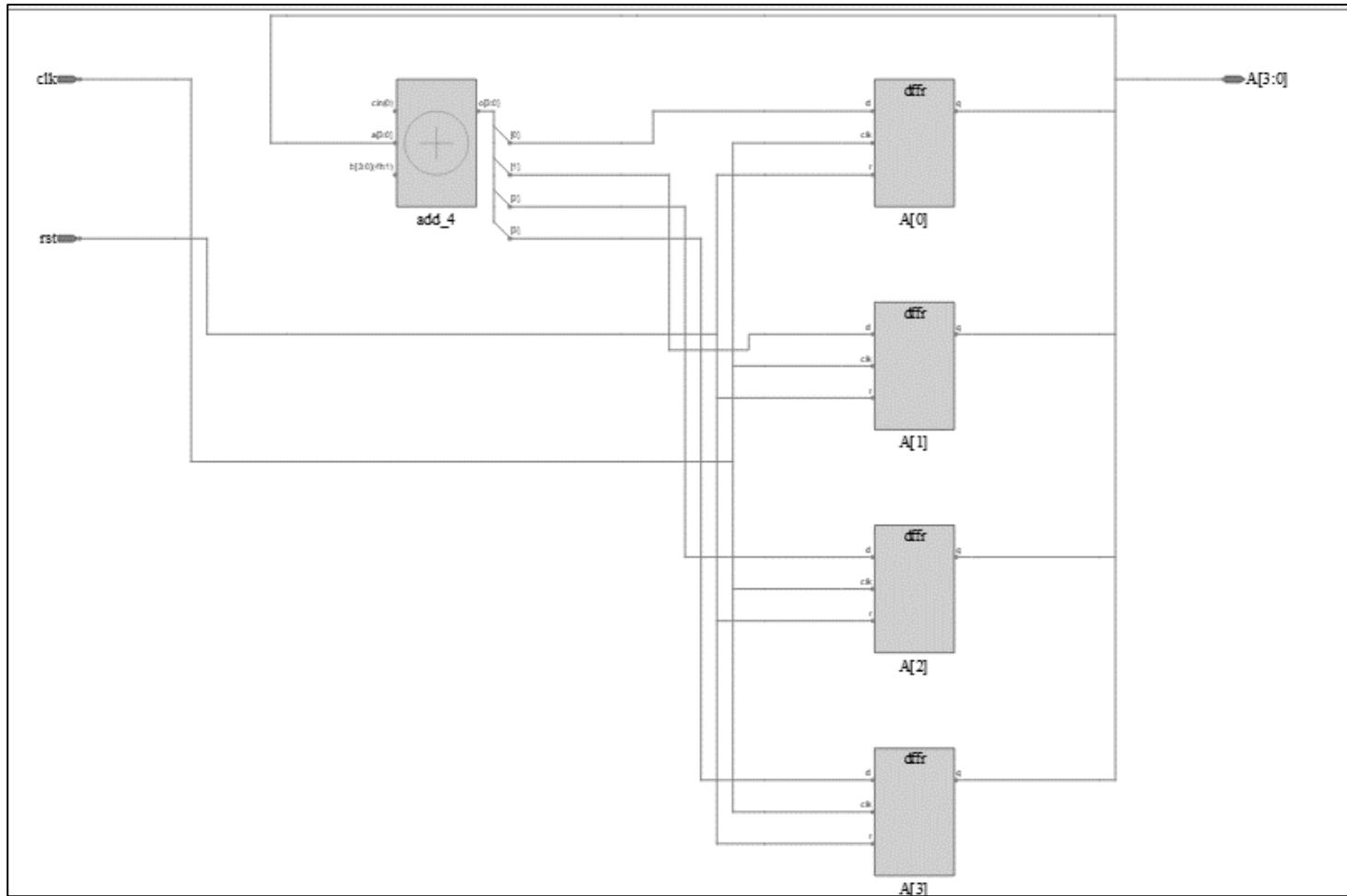
- If I have no idea about ripple counter design with D flip flops how can I define ripple counter
- I have clk and rst inputs
- output is 4bit
- at each clk, increment A+1
- just describe it Verilog will handle the rest

```
'timescale 1 ns / 1 ps
module ripplecounter(A,clk,rst);
input  clk;
input  rst;
output reg [3:0] A=4'b0000;

always @ (posedge clk, posedge rst)
begin
  if (rst)
    A <= 4'b0000;
  else
    A <= A + 4'b0001;
end
endmodule
```

Ripple Counter: Synthesis

- Synthesis generates a D Flip Flop and adder
- adder output becomes input to adder at next cycle



Verilog HDL: Adder/Subtractor

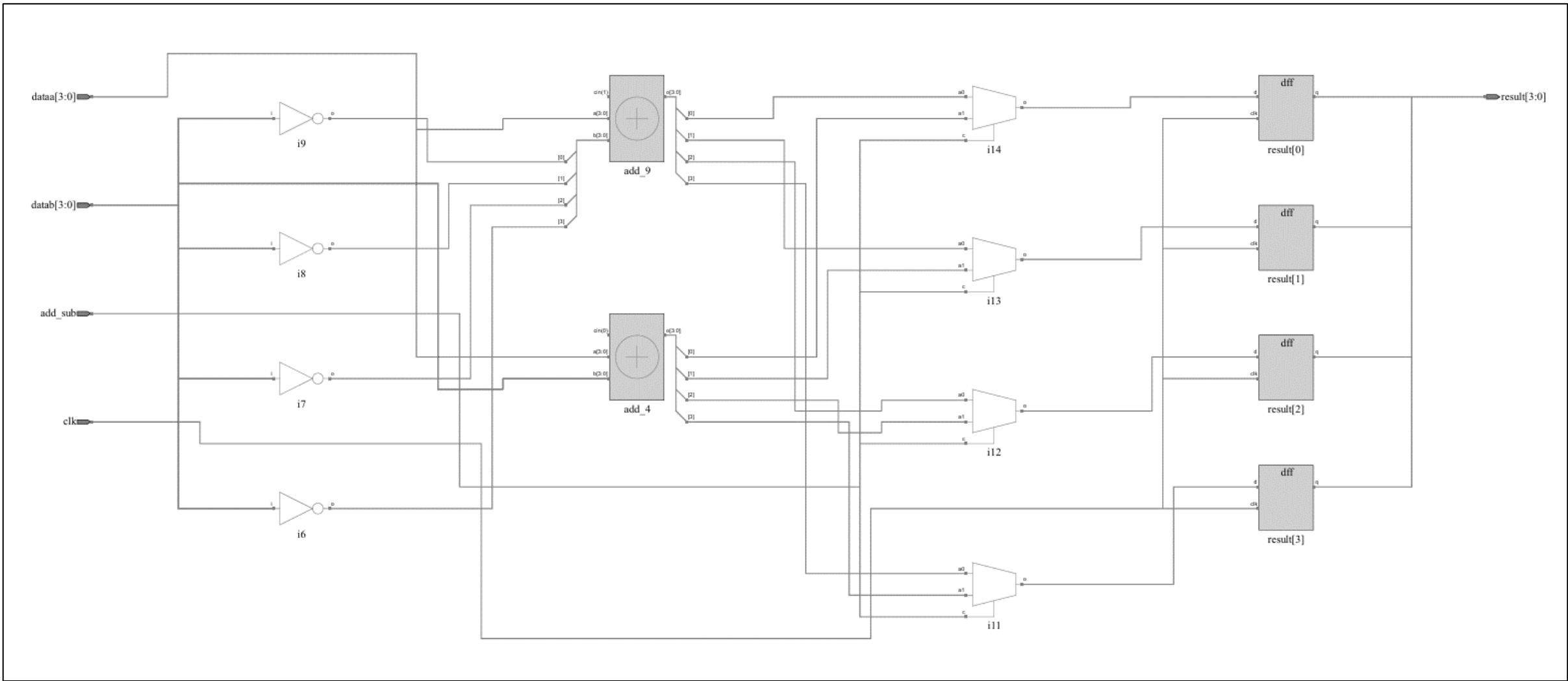
- adder subtractor is a mini version of ALU Arithmetic Logic Unit
- add_sub 1-bit selects the operator add or sub

```
'timescale 1 ns / 1 ns
module addsub
(
    input [3:0] dataa,
    input [3:0] datab,
    input add_sub,           // if this is 1, add;
    else subtract
    input clk,
    output reg [3:0] result
);

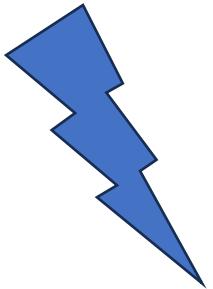
always @ (posedge clk)
begin
    if (add_sub)
        result <= dataa + datab;
    else
        result <= dataa - datab;
end

endmodule
```

Verilog HDL: Adder/Subtractor



Forever and Repeat Loops



Forever and Repeat Loops

- **forever** loop - executes continually

```
initial begin  
    clk = 0;  
    forever #25 clk = ~clk;  
end
```

*Clock with period
of 50 time units*

Not synthesizable!

- **repeat** loop - executes a fixed number of times

```
if (rotate == 1)  
    repeat (8) begin  
        tmp = data[15];  
        data = {data << 1, tmp};  
    end
```

*Repeats a rotate
operation 8 times*

While Loop

While Loop

- while loop - executes if expression is true

```
initial begin
    count = 0;
    while (count < 101) begin
        $display ("Count = %d", count);
        count = count + 1;
    end
end
```

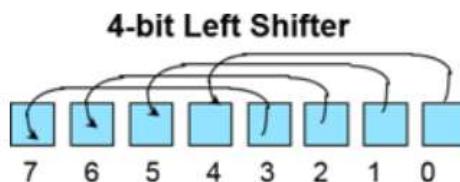
*Counts from 0 to 100
Exits loop at count 101*

Not synthesizable!

For Loop

For Loop

- **for** loop - executes initial assignment at the start of the loop and then executes loop body if expression is true

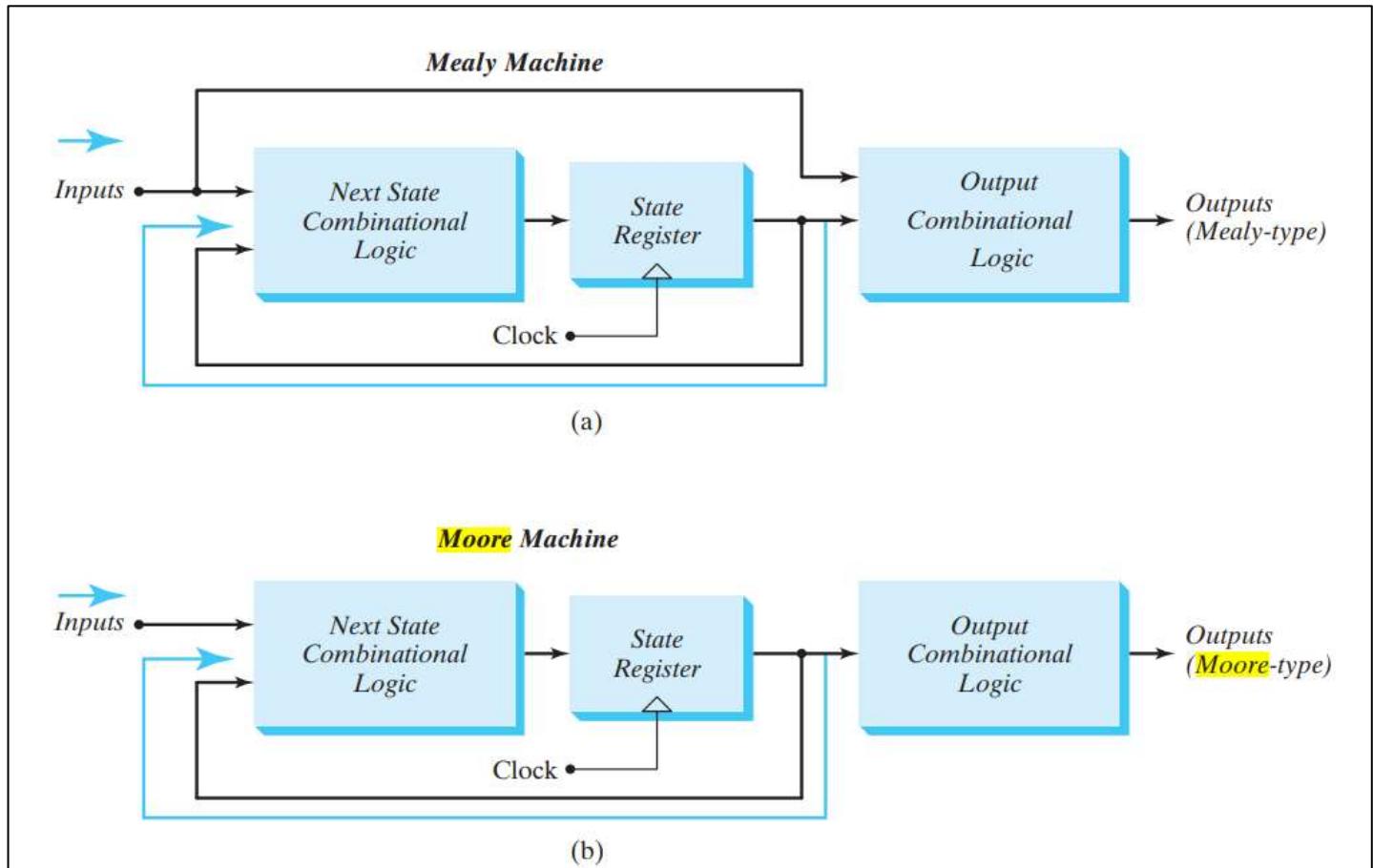


```
// declare the index for the FOR loop
integer i;

always @(inp, cnt) begin
    result[7:4] = 0;
    result[3:0] = inp;
    if (cnt == 1) begin
        {
            for (i = 4; i <= 7; i = i + 1) begin
                result[i] = result[i-4];
            end
            result[3:0] = 0;
        }
    end
end
```

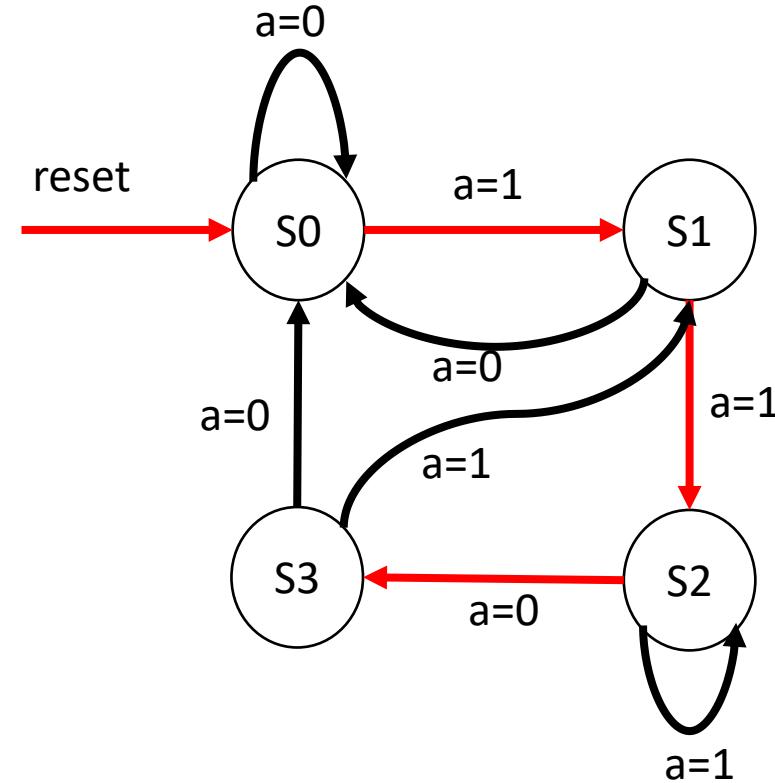
Finite State Machines

- The most general model of a sequential circuit has **inputs, outputs, and internal states**.
- In the **Mealy model**, the output is a function of **both the present state and the input**.
- In the **Moore model**, the output is a function of **only the present state**.
- The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated **FSM**



Finite State Machines: Pattern Detection

- In the **Moore model**, the output is a function of **only the present state**.
- Lets design a circuit to catch serial input a pattern 110
- First draw state diagram
- initial state is S0
- Final state is S3 means we catched the pattern 110

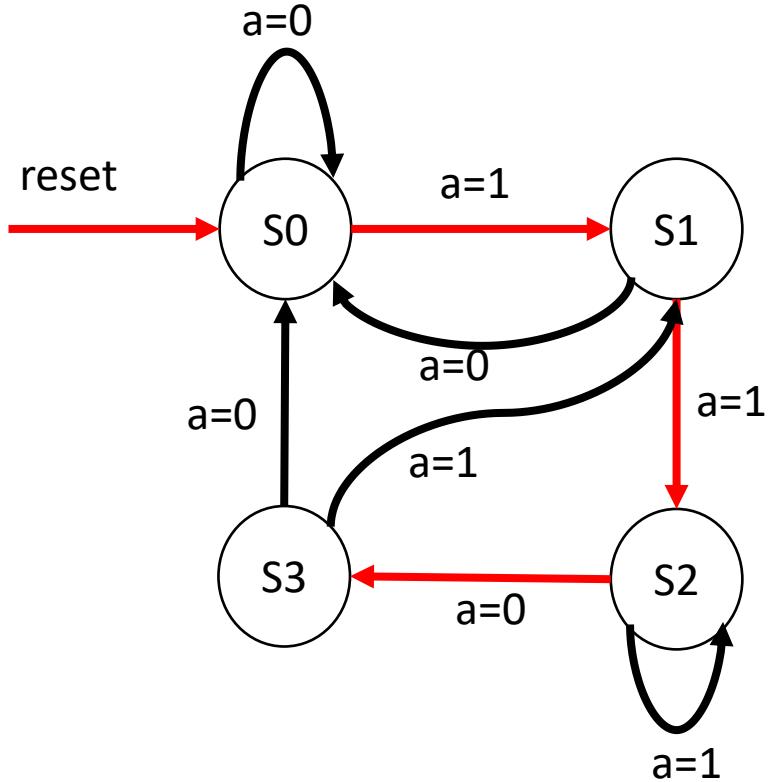


```

`timescale 1ns / 1ps
module patternMealy (
    input clk,
    input reset,
    input a,
    output y
);
    reg [1:0] state, nextstate;
    // State encoding
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;
    // State register
    always @ (posedge clk or posedge reset) begin
        if (reset)
            state <= S0;
        else
            state <= nextstate;
    end
    // Next state logic
    always @ (*) begin
        case (state)
            S0: if (a) nextstate = S1;
                  else nextstate = S0;
            S1: if (a) nextstate = S2;
                  else nextstate = S0;
            S2: if (a) nextstate = S2;
                  else nextstate = S3;
            S3: if (a) nextstate = S1;
                  else nextstate = S0;
            default: nextstate = S0;
        endcase
    end
    // Output logic
    assign y = (state == S3);
endmodule

```

FSM: Pattern Detection



FSM: Pattern Detection

```
'timescale 1ns / 1ps

module patternMealy_tb;
    // Inputs
    reg clk;
    reg reset;
    reg a;
    // Outputs
    wire y;
    // Instantiate the Unit Under Test (UUT)
    patternMealy uut (
        .clk(clk),
        .reset(reset),
        .a(a),
        .y(y)
    );
    // Clock generation: 10ns period (100MHz)
    always #5 clk = ~clk;

    // Test sequence
    initial begin
        // Initialize inputs
        clk = 0;
        reset = 0;
        a = 0;
        // Setup waveform dump
        $dumpfile("dump.vcd"); // Specify the output file name
        $dumpvars(); // Dump all variables
        // Monitor changes in a, state, and y
        $monitor("Time = %0t | clk = %b | reset = %b | a = %b | state = %b | y = %b",
            $time, clk, reset, a, uut.state, y);
    end
endmodule
```

```
// Apply reset
reset = 1;
#10;
reset = 0;
#10;

// Apply test pattern 1101
// Sequence: 1 -> 1 -> 0 -> 1 should result in y = 1
a = 1; #10; // Transition from S0 to S1
a = 1; #10; // Transition from S1 to S2
a = 0; #10; // Transition from S2 to S3
a = 1; #10; // Transition from S3 to S1, y should be 1

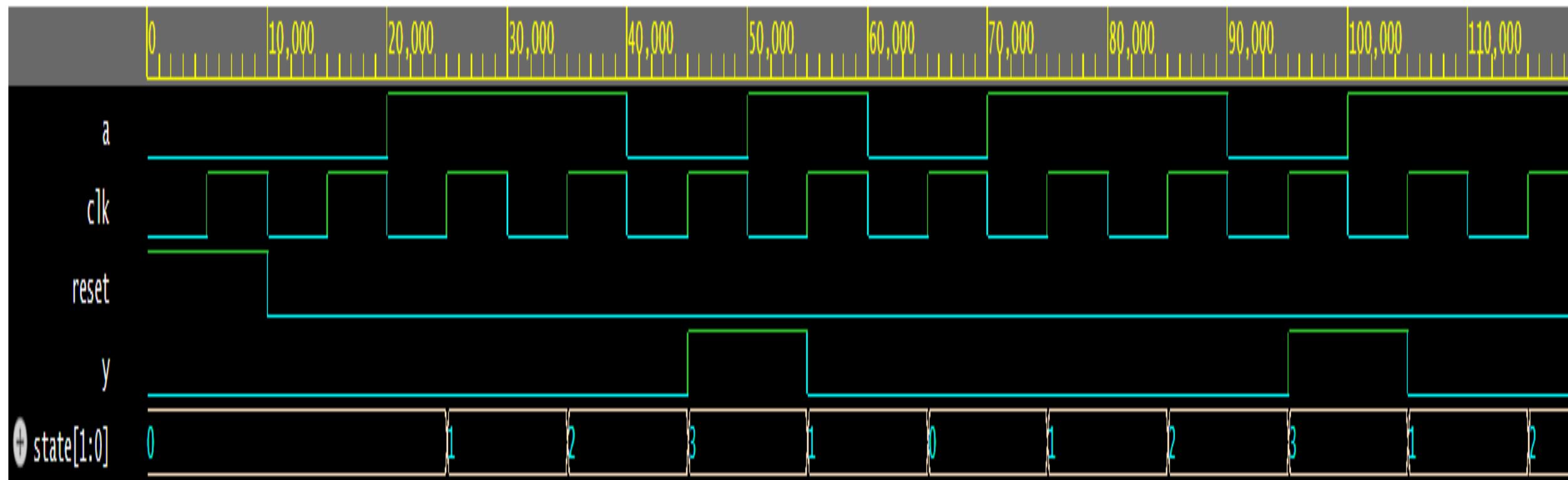
// Apply another test pattern (e.g., 01101)
// This pattern also results in y = 1
a = 0; #10; // Stay in S0
a = 1; #10; // Transition from S0 to S1
a = 1; #10; // Transition from S1 to S2
a = 0; #10; // Transition from S2 to S3
a = 1; #10; // Transition from S3 to S1, y should be 1 again

// End the simulation
#10;
$finish;
end
endmodule
```

FSM: Pattern Detection

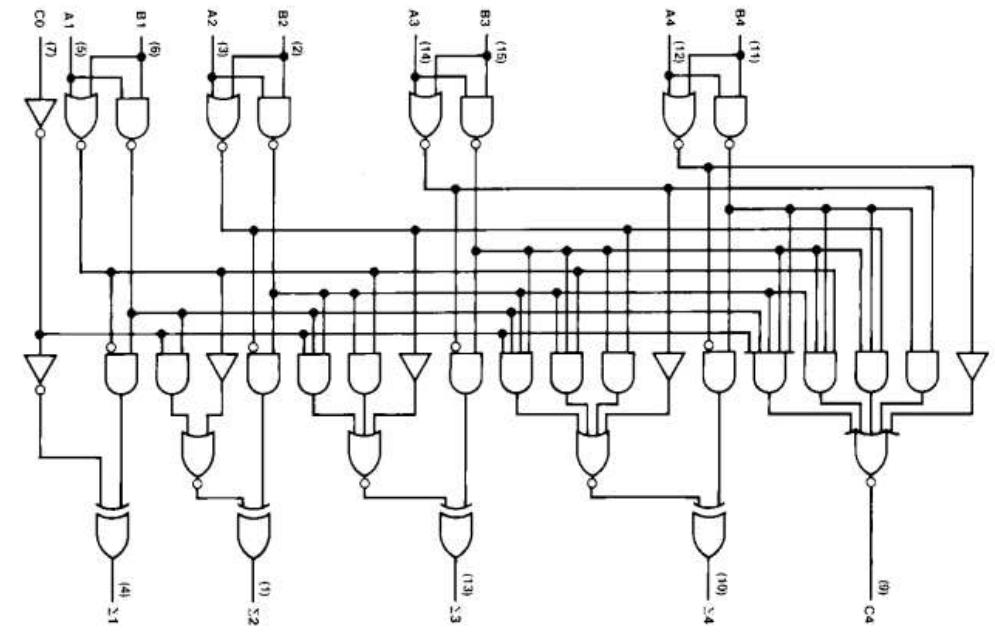
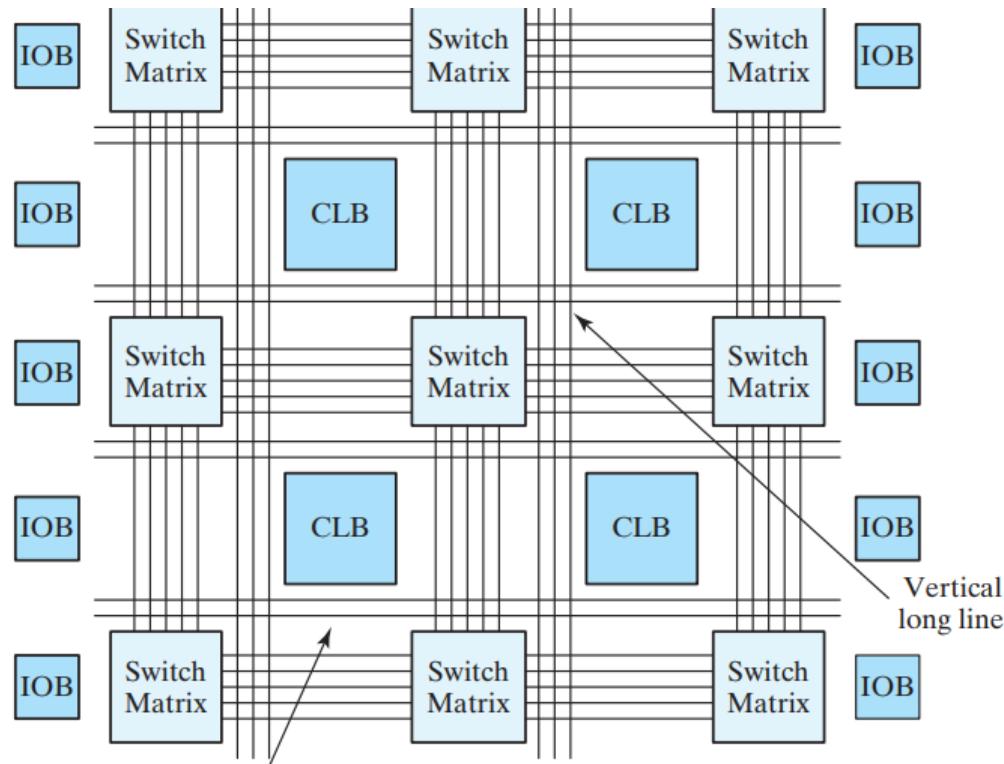
```
Time = 0 | clk = 0 | reset = 1 | a = 0 | state = 00 | y = 0
Time = 5000 | clk = 1 | reset = 1 | a = 0 | state = 00 | y = 0
Time = 10000 | clk = 0 | reset = 0 | a = 0 | state = 00 | y = 0
Time = 15000 | clk = 1 | reset = 0 | a = 0 | state = 00 | y = 0
Time = 20000 | clk = 0 | reset = 0 | a = 1 | state = 00 | y = 0
Time = 25000 | clk = 1 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 30000 | clk = 0 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 35000 | clk = 1 | reset = 0 | a = 1 | state = 10 | y = 0
Time = 40000 | clk = 0 | reset = 0 | a = 0 | state = 10 | y = 0
Time = 45000 | clk = 1 | reset = 0 | a = 0 | state = 11 | y = 1
Time = 50000 | clk = 0 | reset = 0 | a = 1 | state = 11 | y = 1
Time = 55000 | clk = 1 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 60000 | clk = 0 | reset = 0 | a = 0 | state = 01 | y = 0
Time = 65000 | clk = 1 | reset = 0 | a = 0 | state = 00 | y = 0
Time = 70000 | clk = 0 | reset = 0 | a = 1 | state = 00 | y = 0
Time = 75000 | clk = 1 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 80000 | clk = 0 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 85000 | clk = 1 | reset = 0 | a = 1 | state = 10 | y = 0
Time = 90000 | clk = 0 | reset = 0 | a = 0 | state = 10 | y = 0
Time = 95000 | clk = 1 | reset = 0 | a = 0 | state = 11 | y = 1
Time = 100000 | clk = 0 | reset = 0 | a = 1 | state = 11 | y = 1
Time = 105000 | clk = 1 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 110000 | clk = 0 | reset = 0 | a = 1 | state = 01 | y = 0
Time = 115000 | clk = 1 | reset = 0 | a = 1 | state = 10 | y = 0
testbench.sv:61: $finish called at 120000 (1ps)
Time = 120000 | clk = 0 | reset = 0 | a = 1 | state = 10 | y = 0
```

FSM: Pattern Detection



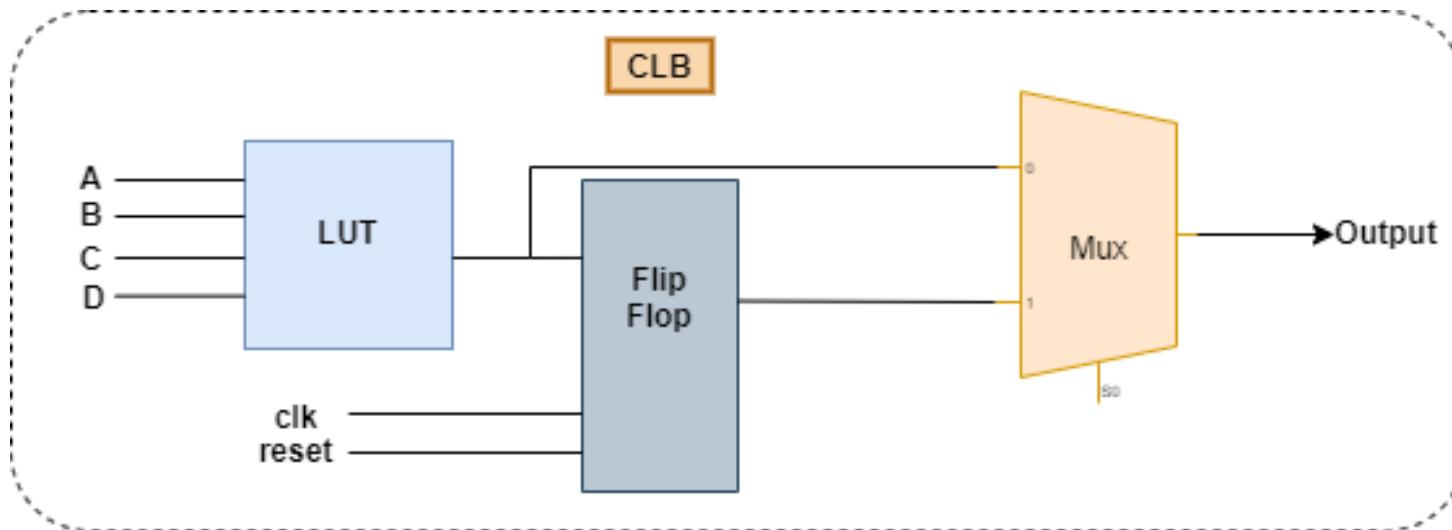
Logic Circuits vs FPGA

- Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects.
- FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from discrete logic IC's



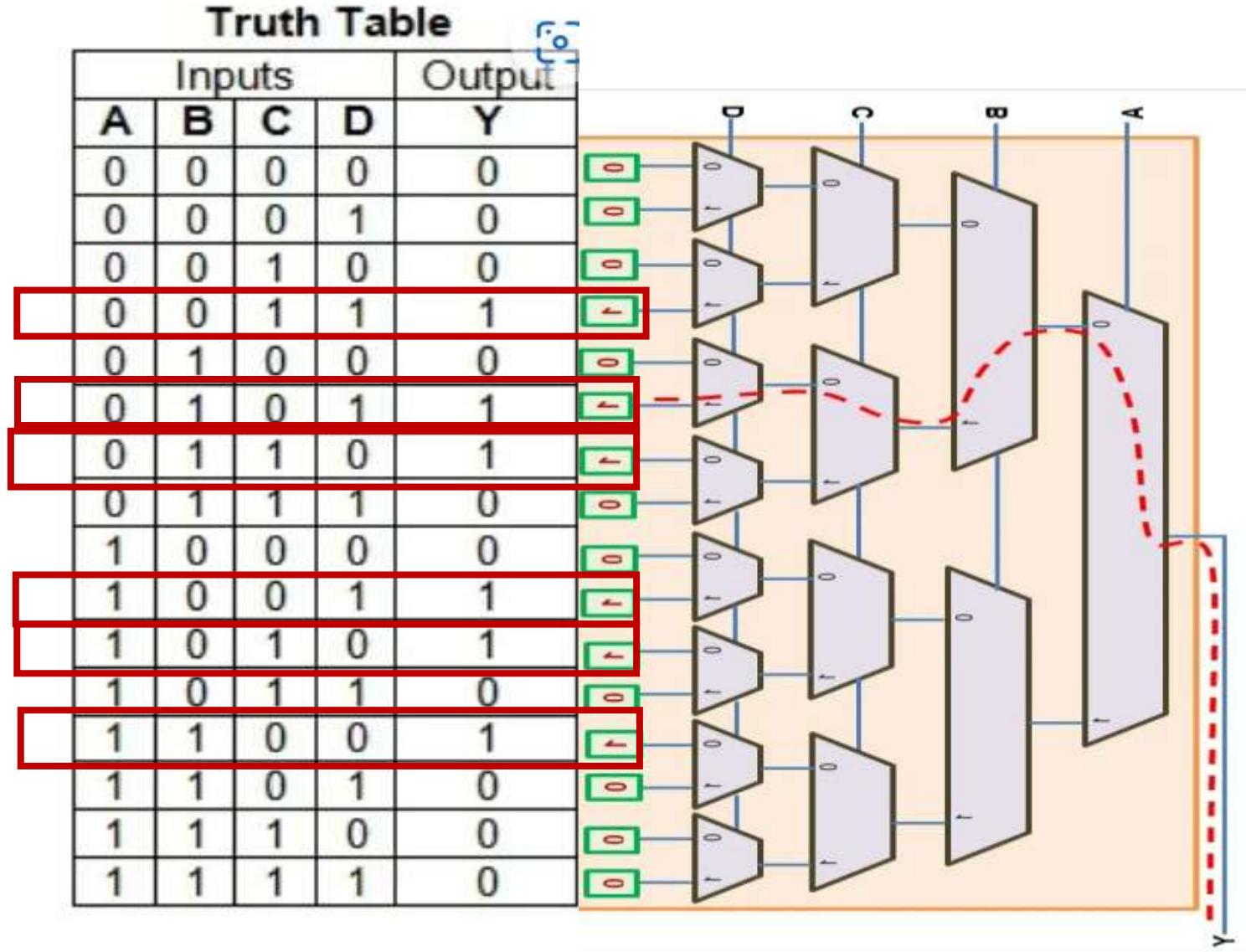
Configurable Logic Blocks (CLB)

- The CLBs consist of three essential elements: LUTs, multiplexer, Flipflop.
- The **LUT** is LookUpTable the primary element that can implement the logical function (combinatorial)
- multiplexer is used to select the data output between **combinational and sequential logic**
- **So having both combinatorial logic and flip-flop together we can implement all type of logic functions**



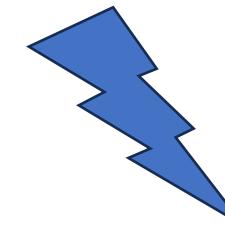
Implementation of Logic Functions using LUT

- Suppose we want to realize a Boolean Function of four input variables A, B, C and D using a 4-input LUT
- While realizing this function using an FPGA, A, B, C, and D will be the inputs to LUT.
- Next, the values of the output variable for each of their combination (available in the last column of the truth table) will be stored in the Flash RAM
- if $ABCD = 0101$, then the output of the LUT, Y, will take the value of 1 as the content of the sixth memory cell makes its way to the output pin (as shown by the red discontinuous line in Figure 3).



Verilog FPGA Implementation

- to implement into TANG NANO 9K board. Follow Steps:
- Install Gowin IDE Education Version
- Create New Project, Select GW1NR or Project → Set Device
- Create New Verilog buttonled.v
- Copy paste from a previous file or write the code
- or use Add Files



Use of Verilog:

1. **test** a given logic circuit
2. **design** a logic circuit
3. use FPGA for **applications**

[Blink LED - Sipeed Wiki](#)

GOWIN FPGA Designer - [D:\docs\ostim\digital\gowin\buttonled\src\buttonled.v]

File Edit Project Tools Window Help

Process

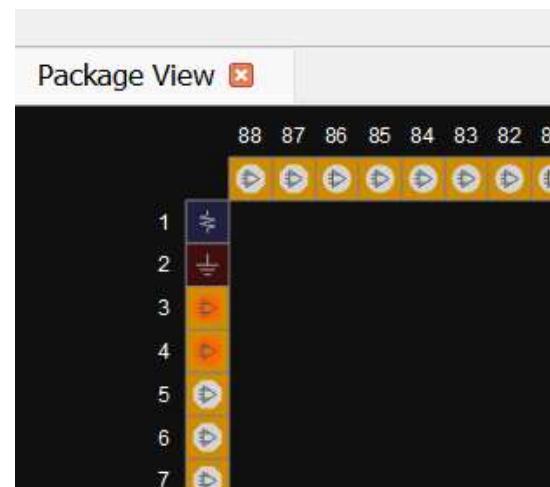
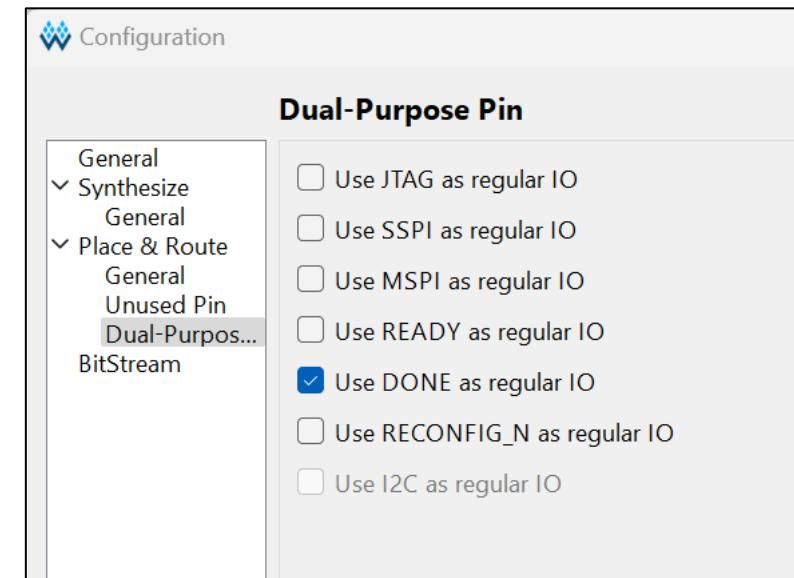
- Design Summary
- User Constraints
- FloorPlanner
- Timing Constraints Editor

```
1 module buttonled(
2   input BUTTON,
3   output LED);
4   assign LED=BUTTON?1:0;
5 endmodule
6
7
```



TANG NANO 9K Board: Button LED

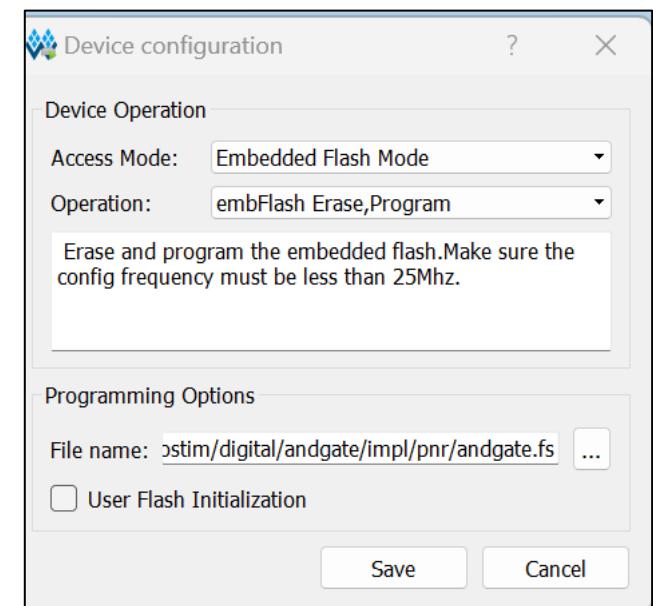
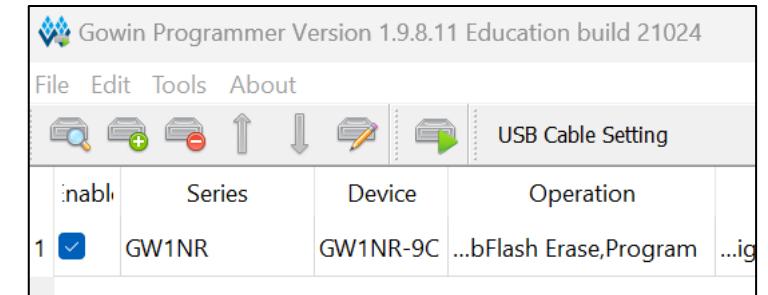
- Process tab:
- Synthesize Configuration (right click): Use DONE as Regular IO
- double click Synthesize. check if no errors
- double click User Constraints: Floor Planner.
- will ask to create .cst file, click YES
- select Package View tab
- drag and drop from ports to package view
- BUTTON→3 LED→10
- double click Place Route.. check if no errors.



I/O Constraints				
	Port	Direction	Diff Pair	Location
1	BUTTON	input		3
2	LED	output		10

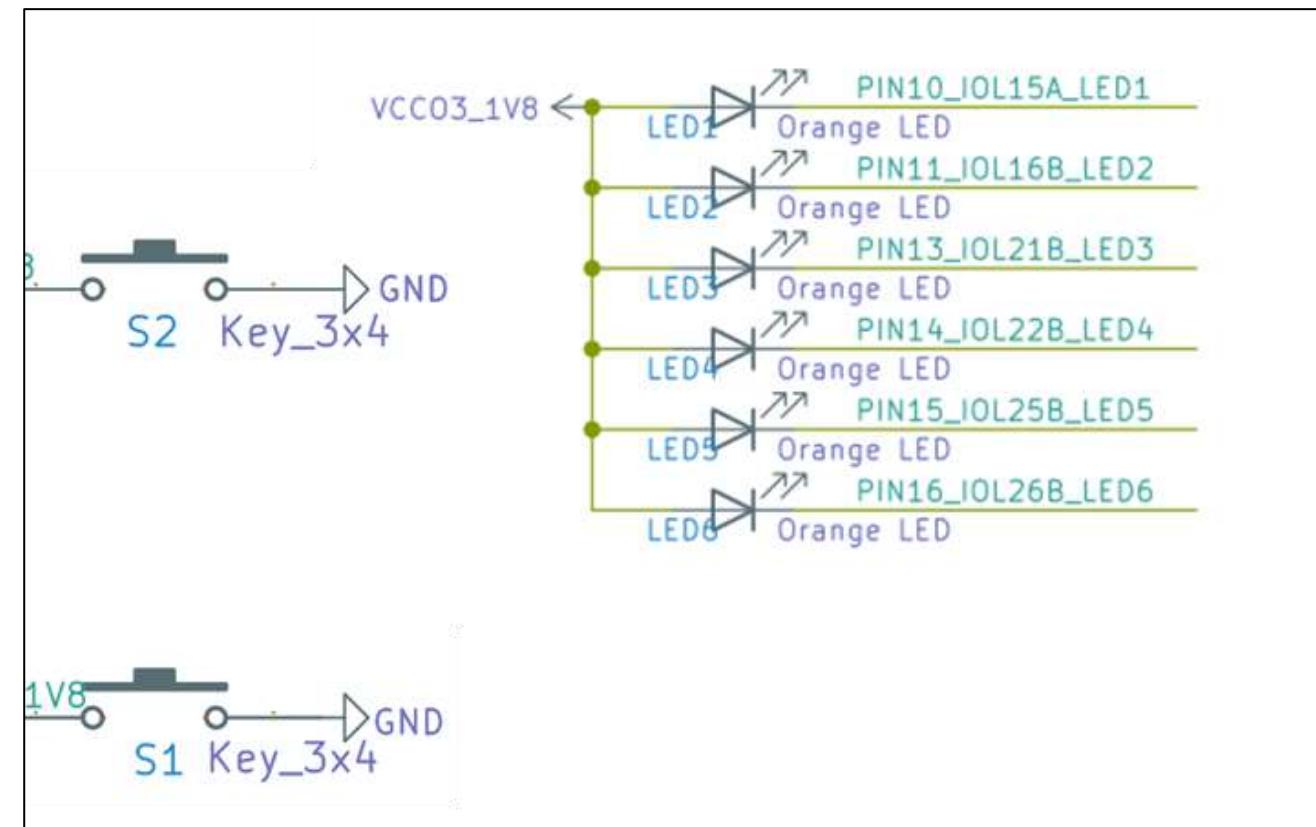
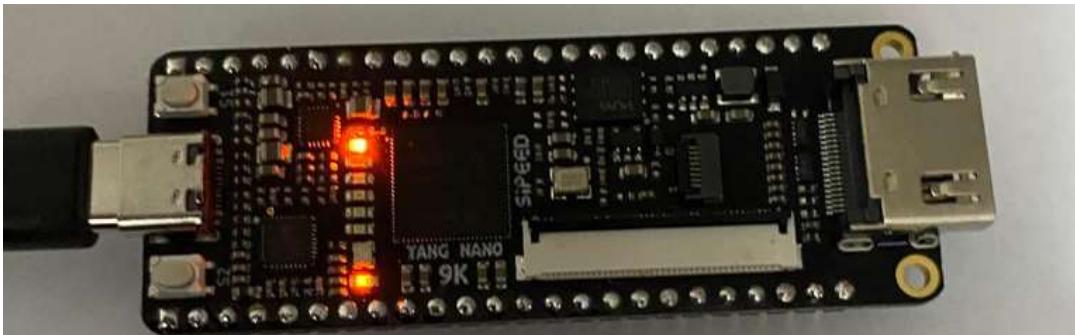
TANG NANO 9K Board: Button LED

- double click Program device
 - Select Embedded Flash Mode
 - Click Program Configure button
 - Verilog program will be flashed into FPGA board
-
- TESTING
 - Press Button1 on the board, check if LED is ON



TANG NANO 9K Board: Button LED

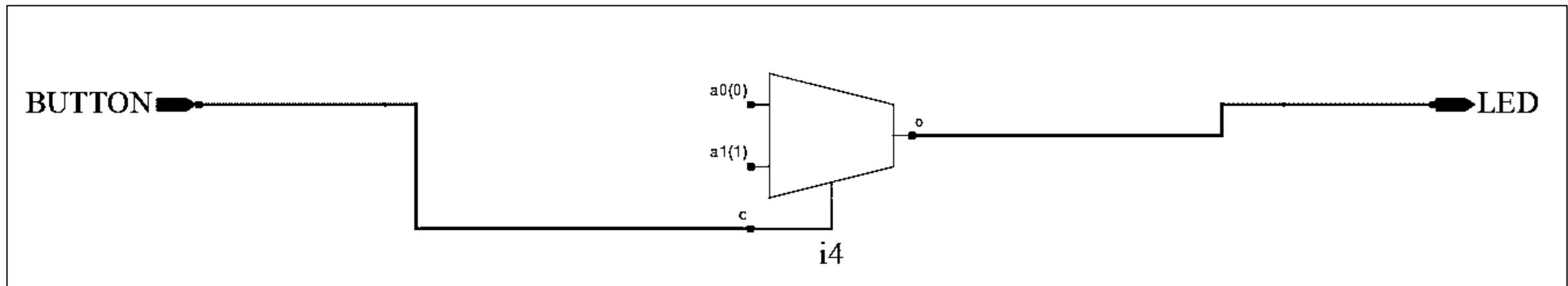
- LED pins are connected ports as drain means:
- LED lighthing if LEDpin=0
- pressed BUTTON=1, notpressed BUTTON=0,
- Thus:
- LED=BUTTON?1:0



TANG NANO 9K Board: Synthesis

- Tools-Schematic Viewer- RTL Design Viewer
- Behavioral model description implemented using a 2to1 MUX

```
1 module buttonled(
2   input BUTTON,
3   output LED);
4   assign LED=BUTTON?1:0;
5 endmodule
6
7
```



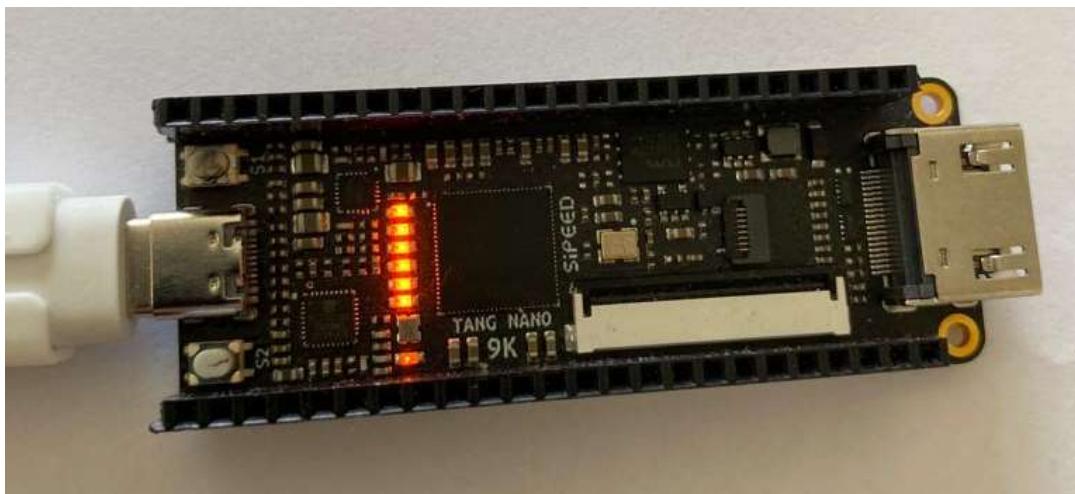
Clock Usage: Blink LED

- Sequential circuits performs actions **at clock signals**. Thus we have clock input
- Output leds defined as 6-bit [5:0]
- localparam is local to the module, number of clock cycles that we will wait for changing leds
- **localparam**: a local constant for storing integers, real numbers, time, delays, or ASCII strings.
- clockCounter is VARIABLE can change within the module, defined with reg (register)
- **reg**: A reg represents a variable in Verilog. A reg can be a 1-bit quantity or a vector of bits. For a vector of bits, the range indicates the most significant bit (msb) and least significant bit (lsb) of the vector.
- The following example shows reg declarations.
- reg x; // single bit
- reg a,b,c; // 3 1-bit quantities
- **reg [7:0] q;** // an 8-bit vector

```
1 module blinkled
2 (
3     input clk,
4     output reg [5:0]    leds
5 );
6 localparam WAIT_TIME = 15000000;
7 reg [23:0] clockCounter = 0;
8 always @ (posedge clk) begin
9     clockCounter <= clockCounter + 1;
10    if (clockCounter == WAIT_TIME) begin
11        clockCounter <= 0;
12        leds = ~leds;
13    end
14 end
15 endmodule
```

Clock Usage: Blink LED

- always @(posedge clk) begin
- ...statements...
- end
- executes statements if condition event occurs. Example clk positive edge, (rising edge)
- if clockCounter reaches WAIT_TIME, clockCounter<=0; resets and leds toggle
- uploaded into TANG NANO 9K and 6leds toggles
- clk is 27MHz, means #1 is 1/50MHz 20ns 1500000 is 0.3sec

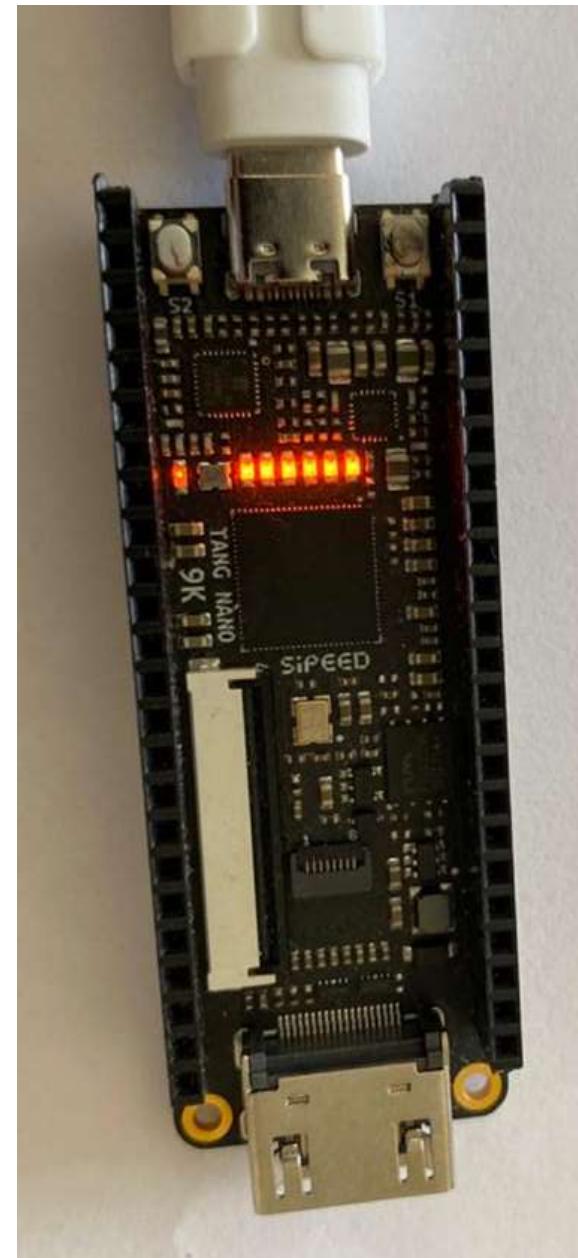


```
module blinkled
(
    input clk,
    output reg [5:0] leds
);
localparam WAIT_TIME = 15000000;
reg [23:0] clockCounter = 0;
always @(posedge clk)
    begin
        clockCounter <= clockCounter + 1;
        if (clockCounter == WAIT_TIME)
            begin
                clockCounter <= 0;
                leds=~leds;
            end
    end
end
endmodule
```

Tang Nano 9K: Blink LED

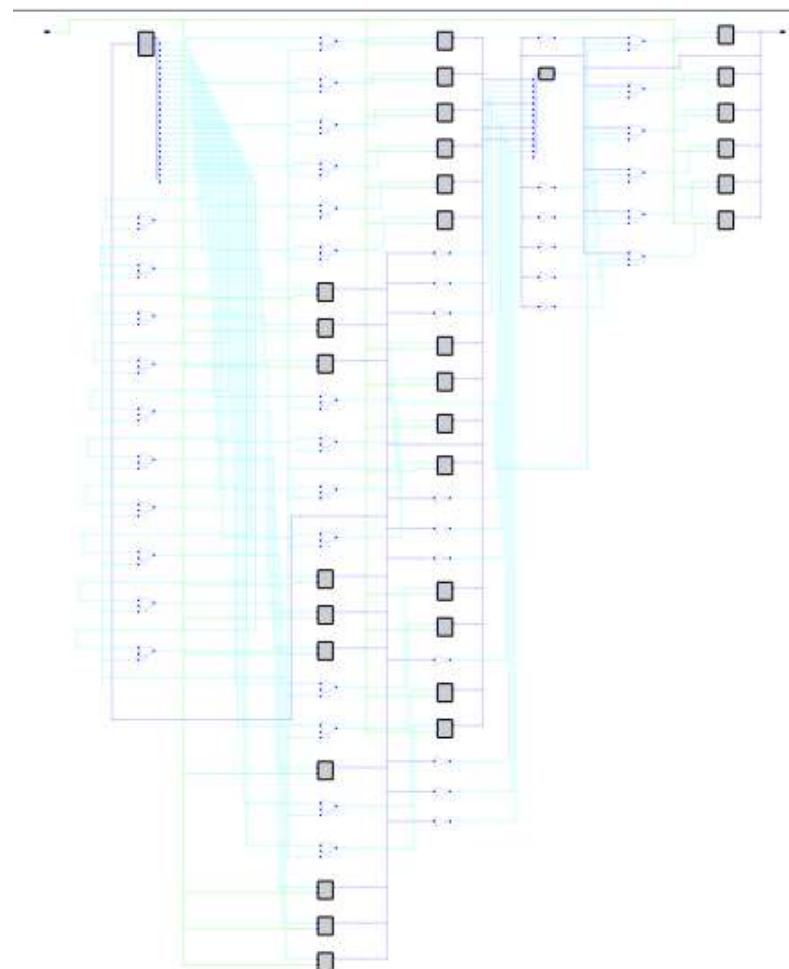
- connect pins as below:
- clock is 52 fix
- 6 leds are 10,11,13,14,15,16
- buttons 3,4

I/O Constraints				
	Port	Direction	Diff Pair	Location
1	clk	input		52
2	leds[0]	output		10
3	leds[1]	output		11
4	leds[2]	output		13
5	leds[3]	output		14
6	leds[4]	output		15
7	leds[5]	output		16



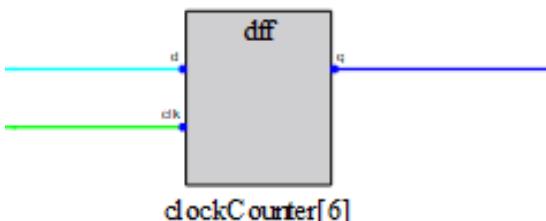
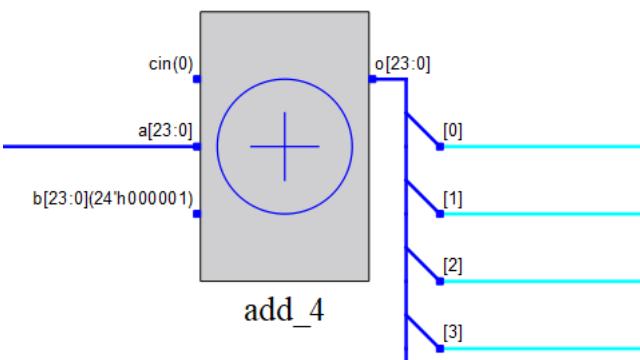
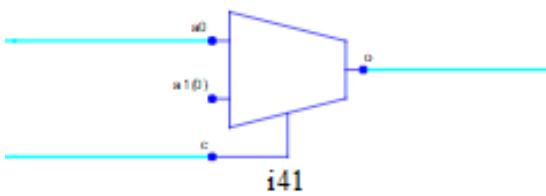
Tang Nano 9K: Blink LED RTL Shematic

- a little complicated ☺
- lets zoom in



Tang Nano 9K: Blink LED RTL Schematic

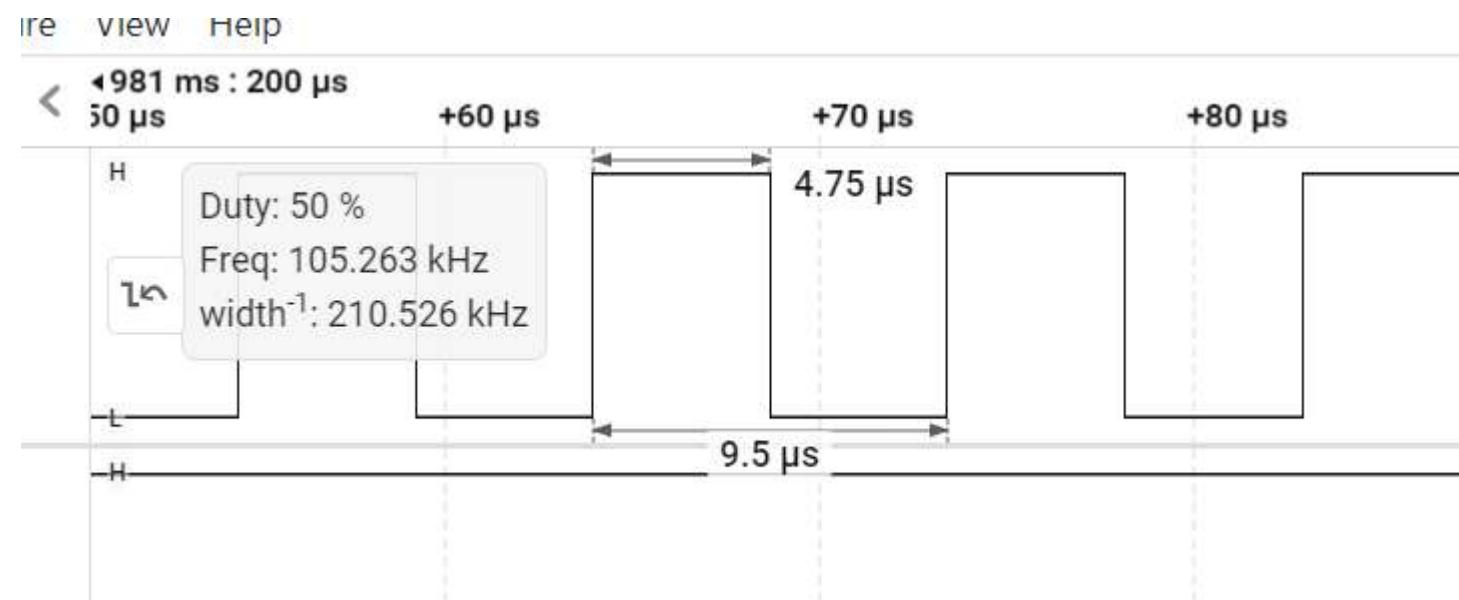
- a little complicated 😊
- lets zoom in
- adder for clockCounter +1
- DFF for clockCounter, leds (register)
- MUX, DFF, NOT



```
1 module blinkled
2 (
3     input clk,
4     output reg [5:0] leds
5 );
6 localparam WAIT_TIME = 15000000;
7 reg [23:0] clockCounter = 0;
8 always @ (posedge clk) begin
9     clockCounter <= clockCounter + 1;
10    if (clockCounter == WAIT_TIME) begin
11        clockCounter <= 0;
12        leds = ~leds;
13    end
14 end
15 endmodule
```

Tang Nano 9K: Clock Divider

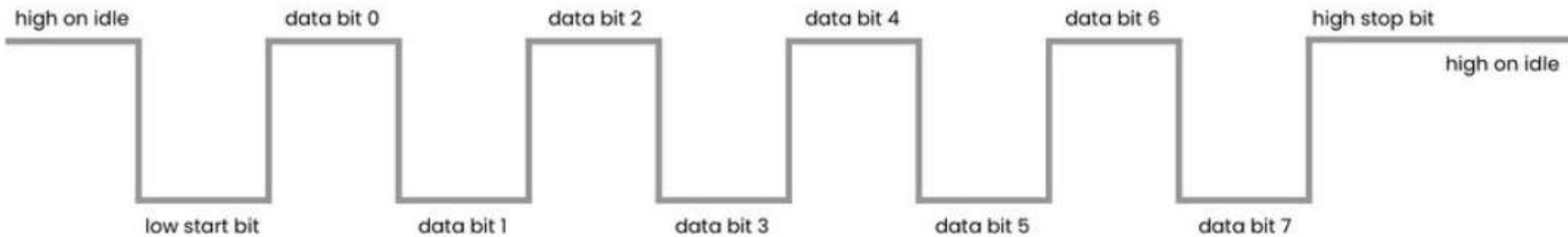
Divides 27MHz clock to 8bits=128 / 2 = 105KHz



```
module blink (input clk, output out1);
reg [7:0] c1;
always @(posedge clk) begin
c1 <= c1 + 1'b1;
end
assign out1 = c1[7:7];
endmodule
```

Tang Nano:UART

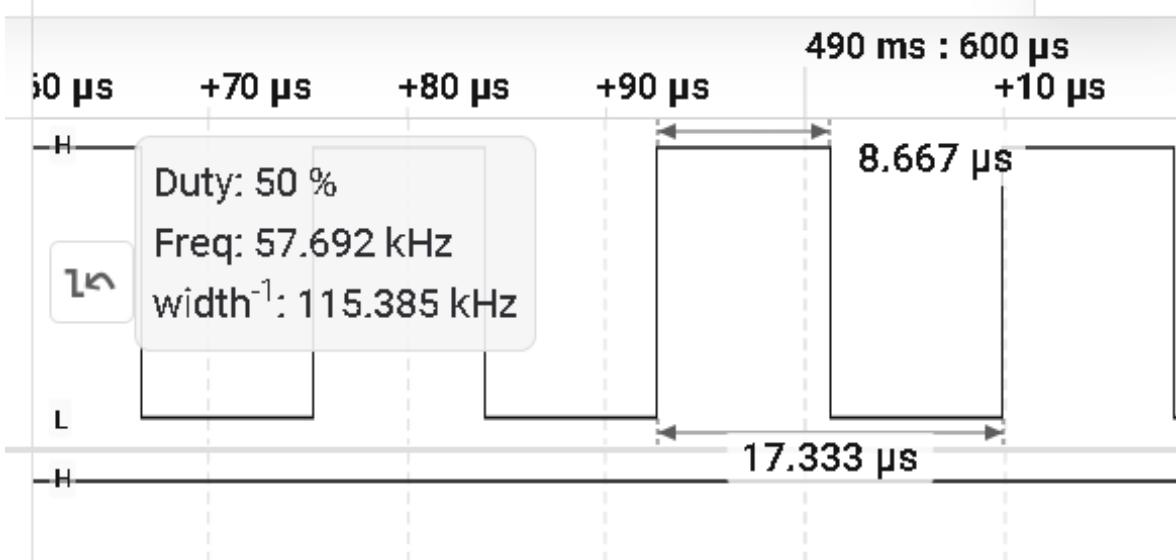
- FPGA's are also used to test communication protocols
- You can generate any digital communication UART, I2C, CAN etc
- lets flash UART transmit example
- 8N1, 0start - 8bit data – 1stop total 10bits



Example sending the byte 01010101 or 85 in decimal

Tang Nano 9K: UART Clock Divider

- How to get 115200bps from 27MHz clock



```
module baud_rate_generator (
    input wire clk_27mhz, // 27 MHz clock
    output reg baud_clk // Baud rate clock
);
    // Calculate the divisor
    localparam integer DIVISOR = 234;
    reg [7:0] counter = 0;
    always @(posedge clk_27mhz) begin
        if (counter >= DIVISOR - 1) begin
            counter <= 0;
            baud_clk <= ~baud_clk;
        end else begin
            counter <= counter + 1'b1;
        end
    end
endmodule
```

Tang Nano:UART

- lets flash UART transmit example

I/O Constraints			
Port	Direction	Diff Pair	Location
1 clk	input		52
2 tx	output		17

```
module uart_transmitter (
    input wire clk,      // System clock
    output wire tx       // UART Tx
);

reg [10:0] baud_counter = 11'b0; // Initialize baud_counter to zero
reg [3:0] bit_counter = 4'b0000; // Initialize bit_counter to zero
reg tx_reg;                // Transmitter shift register

// Baud rate divisor (adjust for your desired baud rate)
parameter BAUD_DIVISOR = 234; // 234 for 115200 bps

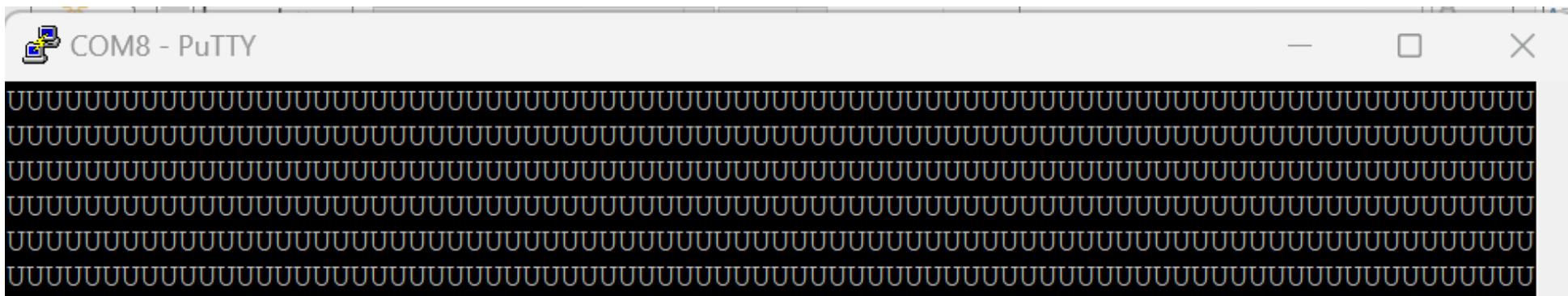
// Fixed data to transmit: 8'b01010101 (binary)
reg [7:0] fixed_data = 8'b01010101;
```

```
always @(posedge clk) begin
    // Increment baud counter
    if (baud_counter == BAUD_DIVISOR - 1) begin
        baud_counter <= 11'b0;
        // Transmit data on the rising edge of the start bit
        case (bit_counter)
            0: begin
                tx_reg <= 0; // Data LSB
                bit_counter <= bit_counter + 1;
            end
            1, 2, 3, 4, 5, 6, 7, 8: begin
                tx_reg <= fixed_data[bit_counter - 1];
                bit_counter <= bit_counter + 1;
            end
            9: begin
                tx_reg <= 1; // Stop bit (always high)
                bit_counter <= bit_counter + 1;
            end
            default: begin
                tx_reg <= 1; // Idle state
                bit_counter <= 0;
            end
        endcase
    end else begin
        baud_counter <= baud_counter + 1;
    end
    assign tx = tx_reg;
endmodule
```

Tang Nano: UART

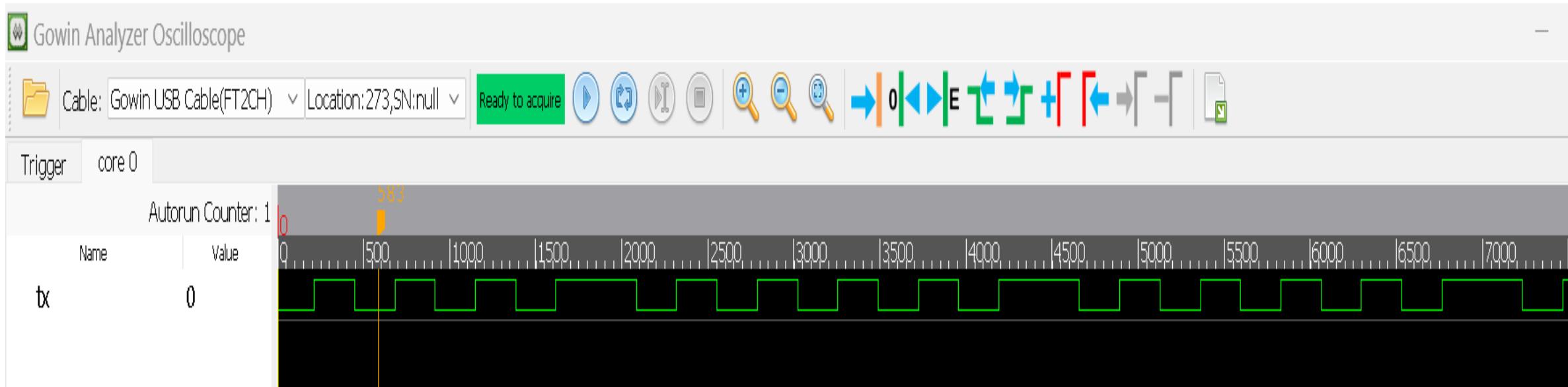
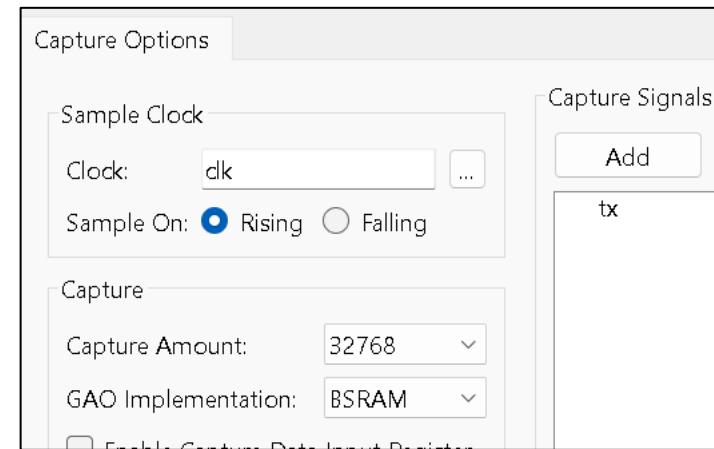
- lets flash UART transmit example
- use PUTTY, COM8, 115200
- 01010101 , ASCII hex55, decimal 85 is character **U** printed continuously

I/O Constraints				
	Port	Direction	Diff Pair	Location
1	clk	input		52
2	tx	output		17



Tang Nano: UART GAO

- Gowin Analyzer Oscilloscope, use Light configuration
- lets flash UART transmit example
- use PUTTY, COM8, 115200
- 01010101 , ASCII hex55, decimal 85 is character **U** printed continuously
- including 1 stop 010101011



Tang Nano: UART

- Send ‘Hello World’ at every 1 seconds
- Written by ChatGPT

```
module uart_tx_hello_world (
    input clk,      // 27 MHz clock
    output reg tx   // UART transmit output
);

// Parameters for 115200 baud rate with 27 MHz clock
parameter CLK_FREQ = 27000000;
parameter BAUD_RATE = 115200;
parameter BAUD_DIV = CLK_FREQ / BAUD_RATE;
parameter SECOND_DIV = CLK_FREQ; // Divider for 1-second interval

// States for the state machine
parameter IDLE = 0;
parameter START_BIT = 1;
parameter DATA_BITS = 2;
parameter STOP_BIT = 3;
parameter DONE = 4;
parameter WAIT = 5;

// String to be sent: "Hello World"
reg [7:0] message [0:10]; // 11 characters
initial begin
    message[0] = "H";
    message[1] = "e";
    message[2] = "l";
    message[3] = "l";
    message[4] = "o";
    message[5] = " ";
    message[6] = "W";
    message[7] = "o";
    message[8] = "r";
    message[9] = "l";
    message[10] = "d";
end

reg [3:0] state = WAIT;
```

```

reg [7:0] message [0:NUM_CHARS-1];
initial begin
    message[0] = "H";
    message[1] = "e";
    message[2] = "l";
    message[3] = "l";
    message[4] = "o";
    message[5] = " ";
    message[6] = "W";
    message[7] = "o";
    message[8] = "r";
    message[9] = "!";
    message[10] = "d";
end

reg [3:0] char_index = 0;
reg [7:0] txdata;
reg send_char = 0;

// Instantiate the UART transmitter module
uart_tx uart_inst (
    .clk(clk),
    .txdata(txdata),
    .tx(tx)
);

// State machine for managing the transmission of characters
always @(posedge clk) begin
    if (uart_inst.state == uart_inst.IDLE && send_char) begin
        txdata <= message[char_index];
        char_index <= (char_index < NUM_CHARS - 1) ? (char_index + 1) : 0;
        send_char <= 0; // Clear the send_char flag
    end else if (uart_inst.state == uart_inst.IDLE && !send_char) begin
        send_char <= 1; // Set the send_char flag to send the next character
    end
end
endmodule

```

Tang Nano: UART

- Send Hello World
- Written by ChatGPT

```

// uart_tx.v
module uart_tx (
    input clk,           // 27 MHz clock
    input [7:0] txdata, // 8-bit data to be transmitted
    output reg tx      // UART transmit output
);

// Parameters
parameter BAUD_RATE = 115200;
parameter CLK_FREQ = 27000000;
parameter DIVISOR = CLK_FREQ / BAUD_RATE;
parameter IDLE = 1;
parameter START = 2;
parameter DATA = 3;
parameter STOP = 4;

reg [3:0] state = IDLE;
reg [15:0] baud_counter = 0;
reg [3:0] bit_count = 0;
reg [7:0] tx_shift_reg = 0;
reg [7:0] txdata_reg = 0; // Register for txdata
reg send = 0; // Signal to indicate when to start sending

// State machine for UART transmission
always @(posedge clk) begin
    case (state)
        IDLE: begin
            tx <= 1; // Idle state is high
            if (send) begin
                txdata_reg <= txdata; // Update txdata
                state <= START; // Move to START state
            end
        end

        START: begin
            tx <= 0; // Start bit
            if (baud_counter < DIVISOR - 1) begin
                baud_counter <= baud_counter + 1;
            end else begin
                baud_counter <= 0;
                tx_shift_reg <= txdata_reg; // Load data to shift register
                bit_count <= 0; // Reset bit count
                state <= DATA; // Move to DATA state
            end
        end

        DATA: begin
            tx <= tx_shift_reg[0]; // Transmit current bit
            if (baud_counter < DIVISOR - 1) begin
                baud_counter <= baud_counter + 1;
            end else begin
                baud_counter <= 0;
                tx_shift_reg <= tx_shift_reg >> 1; // Shift data
                bit_count <= bit_count + 1;
                if (bit_count == 7) begin
                    state <= STOP; // Move to STOP state after 8 bits
                end
            end
        end

        STOP: begin
            tx <= 1; // Stop bit
            if (baud_counter < DIVISOR - 1) begin

```

Tang Nano:Memory

- lets write an example which reads led[n] status from memory and outputs to leds
- 64x6 bits memory defined
- Fill the memory first 6 address
- in always @() block increments Address counter with modulo 6
-
- flash into Tang Nano 9k
- Leds will turn on as described in memory

```
module tb_memory(
    input clk,
    output reg [5:0] leds
);
reg [5: 0] Address=6'b000000;
localparam WAIT_TIME = 13500000;
reg [23:0] clockCounter=0;
reg [5: 0] Mem [0: 5]; // 6 x 6 memory
initial begin
Mem[0]=6'b000001;
Mem[1]=6'b000011;
Mem[2]=6'b000111;
Mem[3]=6'b001111;
Mem[4]=6'b011111;
Mem[5]=6'b111111;
end
always @(posedge clk) begin
clockCounter <= clockCounter + 1;
if (clockCounter == WAIT_TIME) begin
clockCounter <= 0;
Address<=(Address +1) % 6;
leds=~Mem[Address];
end
end
endmodule
```

Tang Nao IP Core

GOWIN and our design partners provide proven Intellectual Property (IP) for various market segment and application to accelerate your design innovation, simply your work and let you focus on your key competence.

The screenshot shows the GOWIN FPGA Development Software interface. The menu bar includes Tools, Window, and Help. The toolbar on the left contains icons for Start Page, Gowin Analyzer Oscilloscope, Schematic Viewer, IP Core Generator (which is selected and highlighted in blue), Programmer, FloorPlanner, Timing Constraints Editor, DSim Cloud, and Options... The main window displays the Target Device as GW1NR-LV9QN88PC6/I5. A tree view under the Filter section shows categories like Hard Module (ADC, CLOCK, DSP, IO, Memory, User Flash) and Soft IP Core (AI, DSP and Mathematics, Interface and Interconnect, Memory Control, Microprocessor System, Multimedia). To the right, a detailed view for the rPLL IP Core is shown with sections for Information (Type: rPLL, Vendor: GOWIN Semiconductor), Summary (describing it as a Phase Locked Loop used for generating multiple relationships to a given input clock), and Reference (links to UG286 and UG286F user guides).

Target Device: GW1NR-LV9QN88PC6/I5

Filter

Name

- ✓ Hard Module
 - > ADC
 - > CLOCK
 - > DSP
 - > IO
 - > Memory
 - > User Flash
- ✓ Soft IP Core
 - > AI
 - > DSP and Mathematics
 - > Interface and Interconnect
 - > Memory Control
 - > Microprocessor System
 - > Multimedia

rPLL

Information

Type: rPLL
Vendor: GOWIN Semiconductor

Summary

Gowin FPGA provides a Phase Locked Loop (rPLL), which is used to generate multiple relationships to a given input clock.

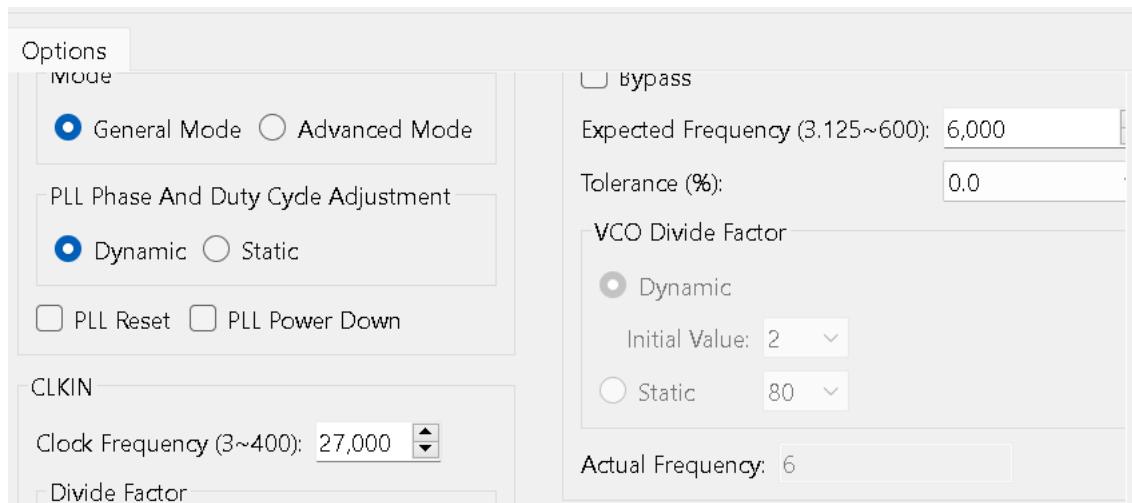
Reference

- [UG286 - Gowin Clock User Guide\(CN\)](#)
- [UG286F - Gowin Clock User Guide\(FN\)](#)

Tang Nao IP Core Example rPLL

Gowin FPGA provides a Phase Locked Loop (rPLL), which is used to generate multiple clocks with defined phase and frequency relationships to a given input clock.

We will generate 6MHz clock at pin 32 from internal 27MHz vlock



Tang Nao IP Core Example rPLL

We will generate 6MHz clock at pin 32 from internal 27MHz vlock at pin 52

gowin_rpll.v automatically generated by GUI. We just instantiate the Gowin_rPLL module

```
module top (
    input clkin,
    output clkout
);
    Gowin_rPLL your_instance_name(
        .clkout(clkout), //output clkout
        .clkin(clkin) //input clkin
    );
endmodule
```

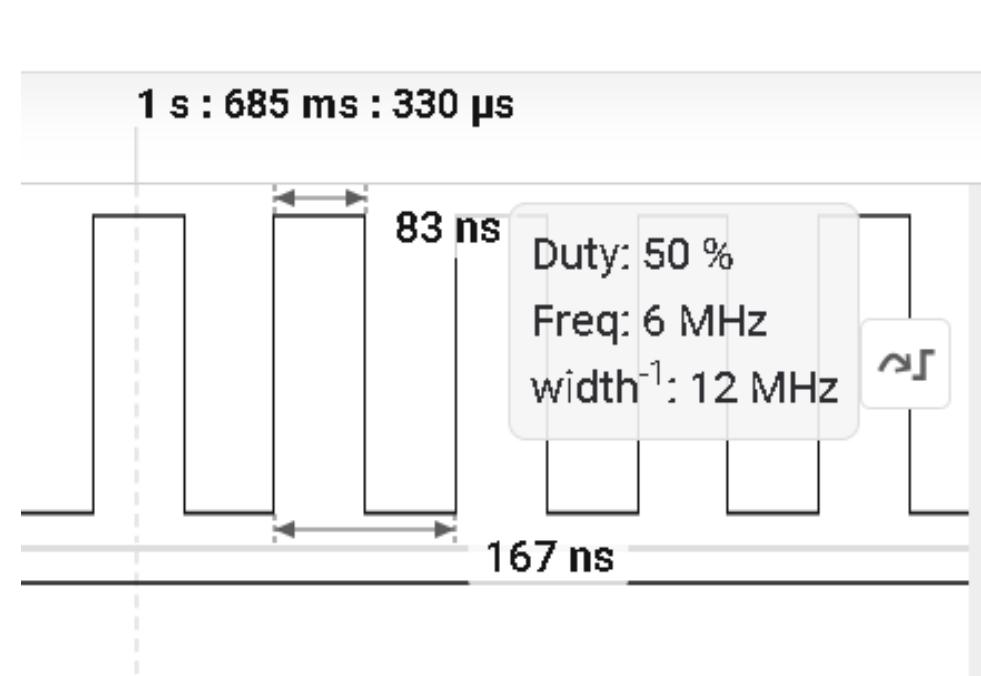
I/O Constraints				
	Port	Direction	Diff Pair	Location
1	clkin	input		52
2	clkout	output		32

Tang Nao IP Core Example rPLL

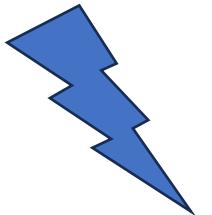
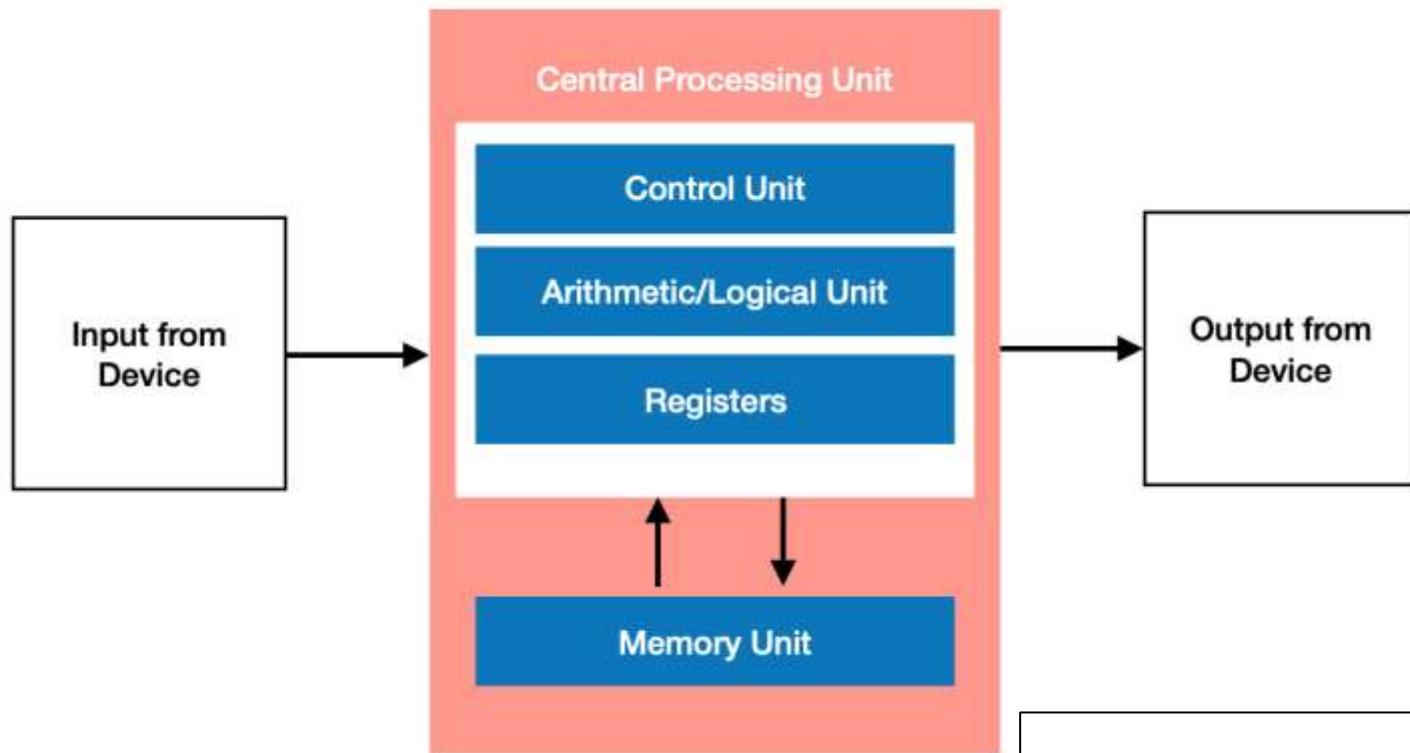
We will generate 6MHz clock at pin 32 from internal 27MHz vlock at pin 52

gowin_rpll.v automatically generated by GUI. We just instantiate the Gowin_rPLL module

```
module top (
    input clkin,
    output clkout
);
    Gowin_rPLL your_instance_name(
        .clkout(clkout), //output clkout
        .clkin(clkin) //input clkin
    );
endmodule
```



Microcontroller (MCU)



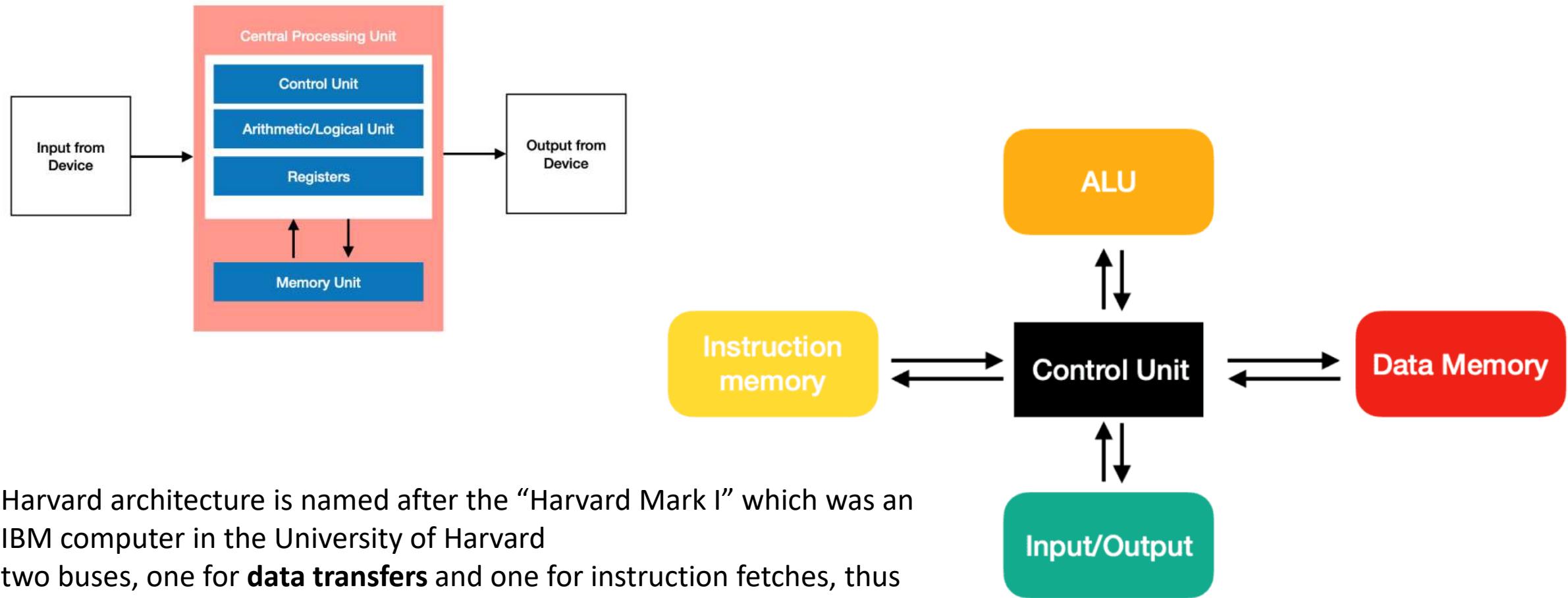
The **von Neumann** architecture—also known as the von Neumann model or Princeton architecture—is a computer architecture based on a 1945 description by **John von Neumann**, and by others, in the First Draft of a Report on the EDVAC.

Use of Verilog:

1. **test** a given logic circuit
2. **design** a logic circuit
3. use FPGA for **applications**

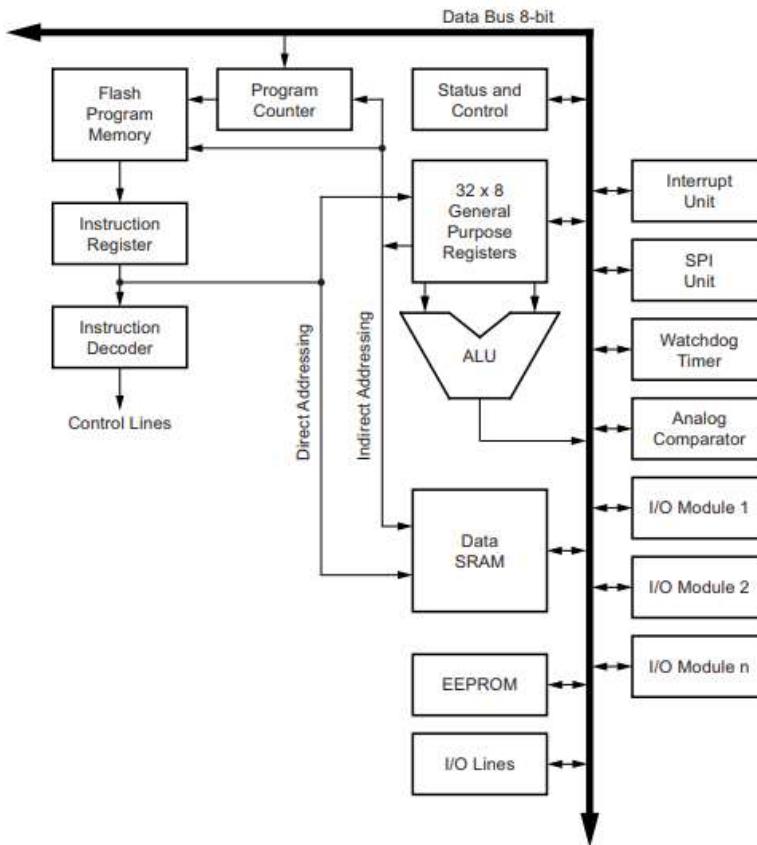
The document describes a design architecture for an electronic digital computer with these components:

Microcontroller (MCU): Harvard Architecture



Microcontroller (MCU) Parts: Arithmetic and Logic Unit (ALU)

- ALU is critical part of MCU performs arithmetic and logical operations
- AluOp performs the operation to execute; ADD, SUB, etc.
- example ADD, SUB without Carry



```
module alu8bit (result,a,b,aluop);
output [7:0] result;
input [7:0] a,b;
input [5:0] aluop;
assign result = (aluop==6'b000011) ? a+b:
              (aluop==6'b000110) ? a-b:
              8'b00000000;
endmodule

example a=sel?1:0;
```

```
'timescale 1ns / 1ns
module alu8bit_tb;
reg [7:0] a,b;
reg [5:0] aluop;
wire [7:0] result;
alu8bit M0(result,a,b,aluop);
initial begin
$display(" a b aluop : result");
$monitor("%d %d %b : %d",a,b,aluop,result);
#1 $display("--add testing--");
#1 a=1; b=2; aluop=6'b000011;
#1 a=13; b=4; aluop=6'b000011;
//sub testing
#1 $display("--sub testing--");
#1 a=4; b=2; aluop=6'b000110;
#1 a=8; b=3; aluop=6'b000110;
end
endmodule
```

```
a b aluop : result
x x xxxxxx : x
--add testing--
1 2 000011 : 3
13 4 000011 : 17
--sub testing--
4 2 000110 : 2
8 3 000110 : 5
```

Microcontroller (MCU) Instructions

- Assembly instructions **ADD, SUB, LDI, NOP, JMP** etc. converted into **16bit Opcode**
- 4-6bit instruction , 2 x 5bit register address (ADD)
- general purpose registers: 5bit address, 8-bit data
- Example is from AVR Instruction Set (Arduino MCU)

6.69 LDI – Load Immediate

6.69.1 Description

Loads an 8-bit constant directly to register 16 to 31.

Operation:

(i) $Rd \leftarrow K$

Syntax:

Operands:

$16 \leq d \leq 31, 0 \leq K \leq 255$

Program Counter:

$PC \leftarrow PC + 1$

16-bit Opcode:

1110	KKKK	dddd	KKKK
------	------	------	------

6.2 ADD – Add without Carry

6.2.1 Description

Adds two registers without the C flag and places the result in the destination register Rd.

Operation:

(i) $(i) Rd \leftarrow Rd + Rr$

Syntax:

Operands:

Program Counter:

(i) $ADD Rd,Rr$

$0 \leq d \leq 31, 0 \leq r \leq 31$

$PC \leftarrow PC + 1$

16-bit Opcode:

0000	11rd	dddd	rrrr
------	------	------	------

6.81 NOP – No Operation

6.81.1 Description

This instruction performs a single cycle No Operation.

Operation:

(i) No

Syntax:

Operands:

Program Counter:

(i) **NOP**

None

$PC \leftarrow PC + 1$

16-bit Opcode:

0000	0000	0000	0000
------	------	------	------

6.62 JMP – Jump

6.62.1 Description

Jump to an address within the entire 4M (words) program memory. See also **RJMP**.

This instruction is not available on all devices. Refer to [Appendix A](#).

Operation:

(i) $PC \leftarrow k$

Syntax:

Operands:

Program Counter:

Stack:

(i) $JMP k$

$0 \leq k < 4M$

$PC \leftarrow k$

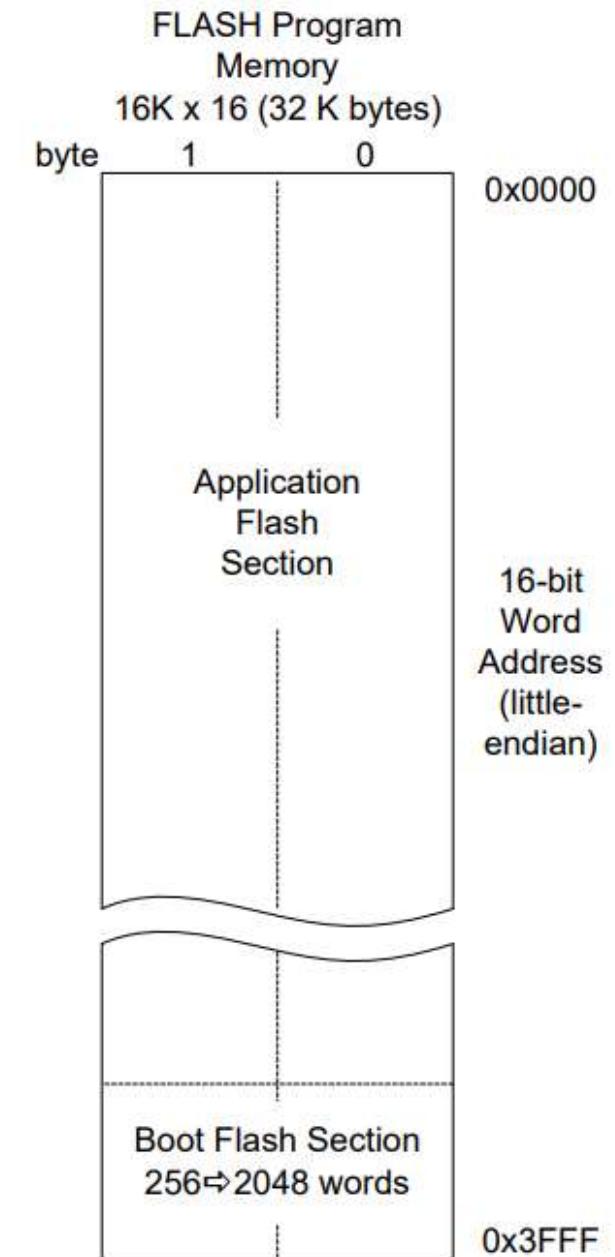
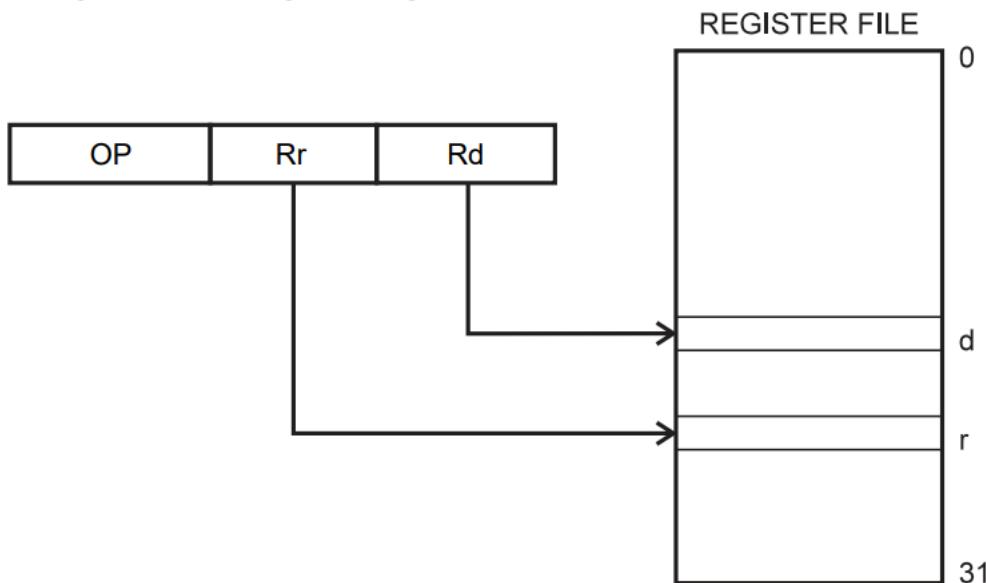
Unchanged

32-bit Opcode:

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

MCU: Program Memory

- General Purpose Registers for fast add-sub operations
- opcodes are kept in flash memory. example 16bit address
- $0x3FF = 16K \times 2 \text{ bytes} = 32\text{Kbytes}$ of flash



MCU: Program Memory

- example code for decimal 17+02=19 operation
- hex values 0x12+0x34
- opcodes can be loaded from a text file or **directly written into the code**

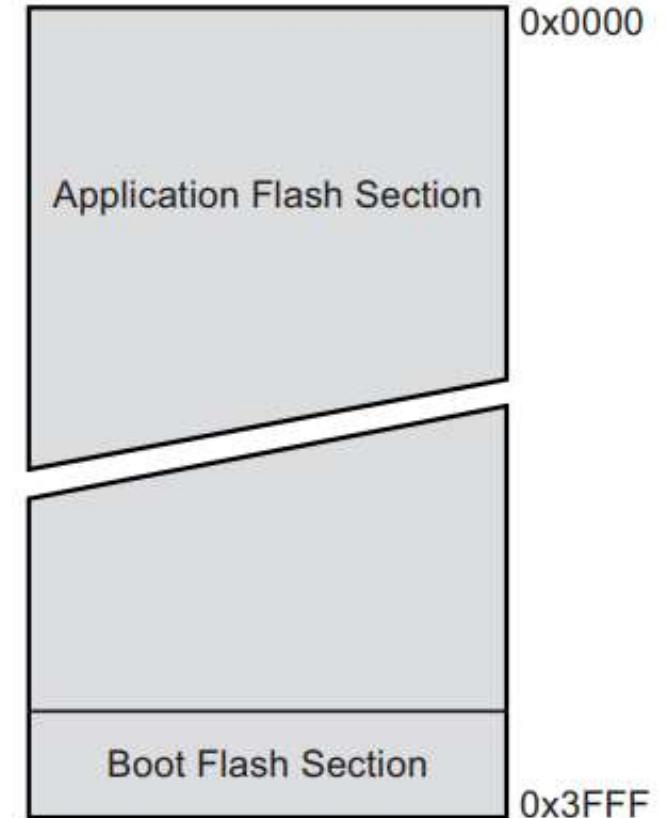
```
ldi r16, 0x12 : 16'b1110 0001 0000 0001 (1110 KKKK dddd KKKK)
ldi r17, 0x02 : 16'b1110 0000 0001 0010 (1110 KKKK dddd KKKK)
add r16, r17 : 16'b0001 1111 0000 0001 (0001 11rd dddd rrrr)
jmp 0x02    : 16'b1001 0000 0000 0010 (1001 kkkk kkkk kkkk)
nop        : 16'b0000 0000 0000 0000 (0000 0000 0000 0000)
```

```
module pmem_tb();
reg [15:0] pmem [0:15];
initial begin
$display("Loading Program Memory.");
$readmemb("prog.txt", pmem);
#10 $display("%b",pmem[0]);
#10 $display("%b",pmem[1]);
#10 $display("%b",pmem[2]);
#10 $display("%b",pmem[3]);
#10 $display("%b",pmem[4]);
end
endmodule
```

```
1110000100000010 // (1110 KKKK dddd KKKK)
1110001100010100 // (1110 KKKK dddd KKKK)
0001111100000001 // (0001 11rd dddd rrrr)
1001000000000010 //jmp 0x02
0000000000000000
0000000000000000
0000000000000000
```



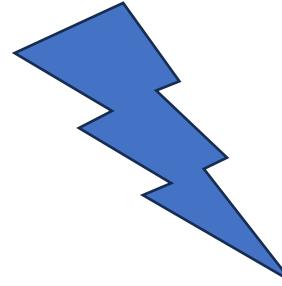
Program Memory



Loading Program Memory.
1110000100000010
1110001100010100
0001111100000001
1001000000000010
0000000000000000

MYMCU: FETCH→DECODE→EXECUTE

- lets try a basic mymcu implementation
- program is written into pmem[] memory
- FETCH: read instructions from memory,

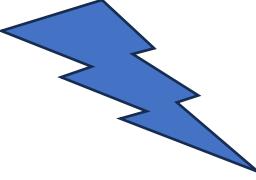


```
ldi r16, 0x12 : 16'b1110 0001 0000 0001 (1110 KKKK dddd KKKK)
ldi r17, 0x02 : 16'b1110 0000 0001 0010 (1110 KKKK dddd KKKK)
add r16, r17 : 16'b0001 1111 0000 0001 (0001 11rd dddd rrrr)
jmp 0x02    : 16'b1001 0000 0000 0010 (1001 kkkk kkkk kkkk)
nop       : 16'b0000 0000 0000 0000 (0000 0000 0000 0000)
nop
...
...
```

```
module mymcu(clk);
input clk;
reg [15:0] pmem [0:15]; //prog mem
reg [7:0] gpreg [0:31]; //gp registers
reg [5:0] PCounter = 0;
reg [15:0] instr;
reg [4:0] rega,rd,rr; //reg address
reg [7:0] data; //data to load
//load program memory
initial begin
pmem[0]=16'b1110000100000001;
pmem[1]=16'b1110000000010010;
pmem[2]=16'b0001111000000001;
pmem[3]=16'b1001000000000010;
pmem[4]=16'b0000000000000000;
pmem[5]=16'b0000000000000000;
pmem[6]=16'b0000000000000000;
pmem[7]=16'b0000000000000000;
pmem[8]=16'b0000000000000000;
pmem[9]=16'b0000000000000000;
pmem[10]=16'b0000000000000000;
pmem[11]=16'b0000000000000000;
pmem[12]=16'b0000000000000000;
pmem[13]=16'b0000000000000000;
pmem[14]=16'b0000000000000000;
pmem[15]=16'b0000000000000000;
end
```

MYMCU:

- you can put \$display() for debugging
- remove later for fpga implementation
- **FETCH**:read instructions from memory
- **instr=pmem[PCounter];**
- **DECODE**: decode the instruction
- **case (instr[15:12])**
- **rega={1'b1,instr[7:4]};**
- **data={instr[11:8],instr[3:0]};**
- **EXECUTE**: execute the function
- **gpreg[rega]=data;**
- **gpreg[rd] = gpreg[rd] + gpreg[rr];**
- **increment the PC**
- **PCounter=PCounter+1**



```
//execute at each waited clock cycle
always @(posedge clk) begin
    //get the instruction from program mem
    instr=pmem[PCounter];
    $display("%b",instr);
    case (instr[15:12])
        4'b1110 : begin //ldi
            rega={1'b1,instr[7:4]};
            data={instr[11:8],instr[3:0]};
            gpreg[rega]=data;
            PCounter=PCounter+1;
            $display("ldi PC=%d d=%d",PCounter,data);
        end
        4'b0001 : begin //add
            rd={1'b1,instr[7:4]};
            rr={1'b1,instr[3:0]};
            gpreg[rd] = gpreg[rd] + gpreg[rr];
            PCounter=PCounter+1;
            $display("add PC=%d
rd=%d",PCounter,gpreg[rd]);
        end
        4'b1001 : begin //jmp
            PCounter=instr[11:0];
            $display("jmp PC=%d",PCounter);
        end
        4'b0000 : begin //nop
            PCounter=PCounter+1;
            $display("nop PC=%d",PCounter);
        end
    endcase
end //always
endmodule
```

MYMCU: Test Bench

- \$display() for debugging will print where we are

```
$display("ldi PC=%d d=%d",PCounter,data);
$display("add PC=%d rd=%d",PCounter,gpreg[rd]);
$display("jmp PC=%d",PCounter);
$display("nop PC=%d",PCounter);
```

```
`timescale 1 ns / 1 ns
module mymcu_tb();
    reg clk;
    mymcu M0 (clk);
    initial begin
        clk=0;
        forever #1 clk = ~clk;
    end
    initial #20 $finish;
endmodule
```

```
1110000100000001
ldi PC= 1 d= 17
111000000010010
ldi PC= 2 d= 2
0001111000000001
add PC= 3 rd= 19
1001000000000010
jmp PC= 2
0001111000000001
add PC= 3 rd= 21
1001000000000010
jmp PC= 2
0001111000000001
add PC= 3 rd= 23
1001000000000010
jmp PC= 2
0001111000000001
add PC= 3 rd= 25
1001000000000010
jmp PC= 2
```

AVR vs MIPS

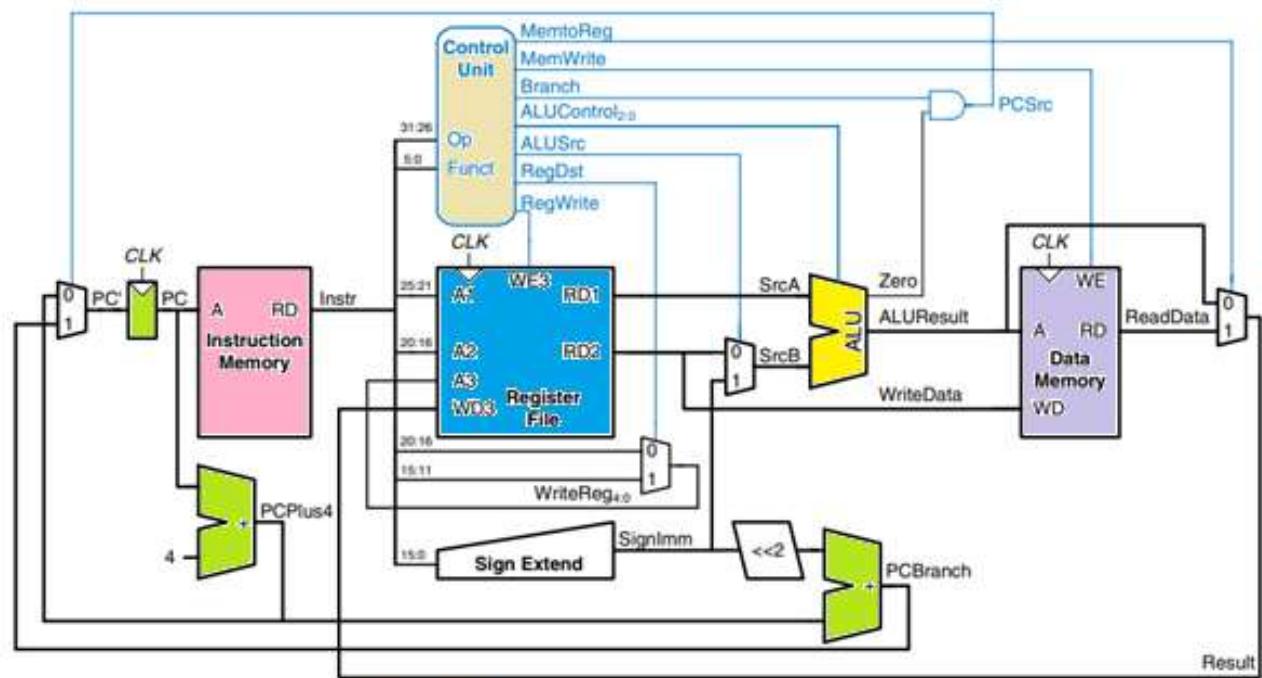
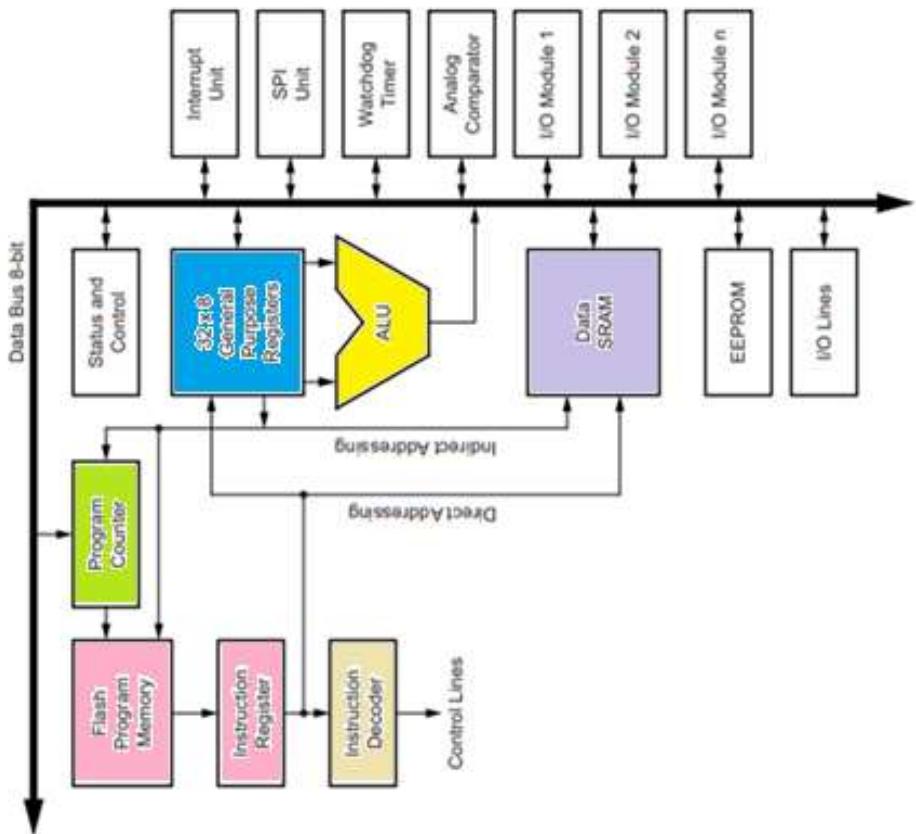
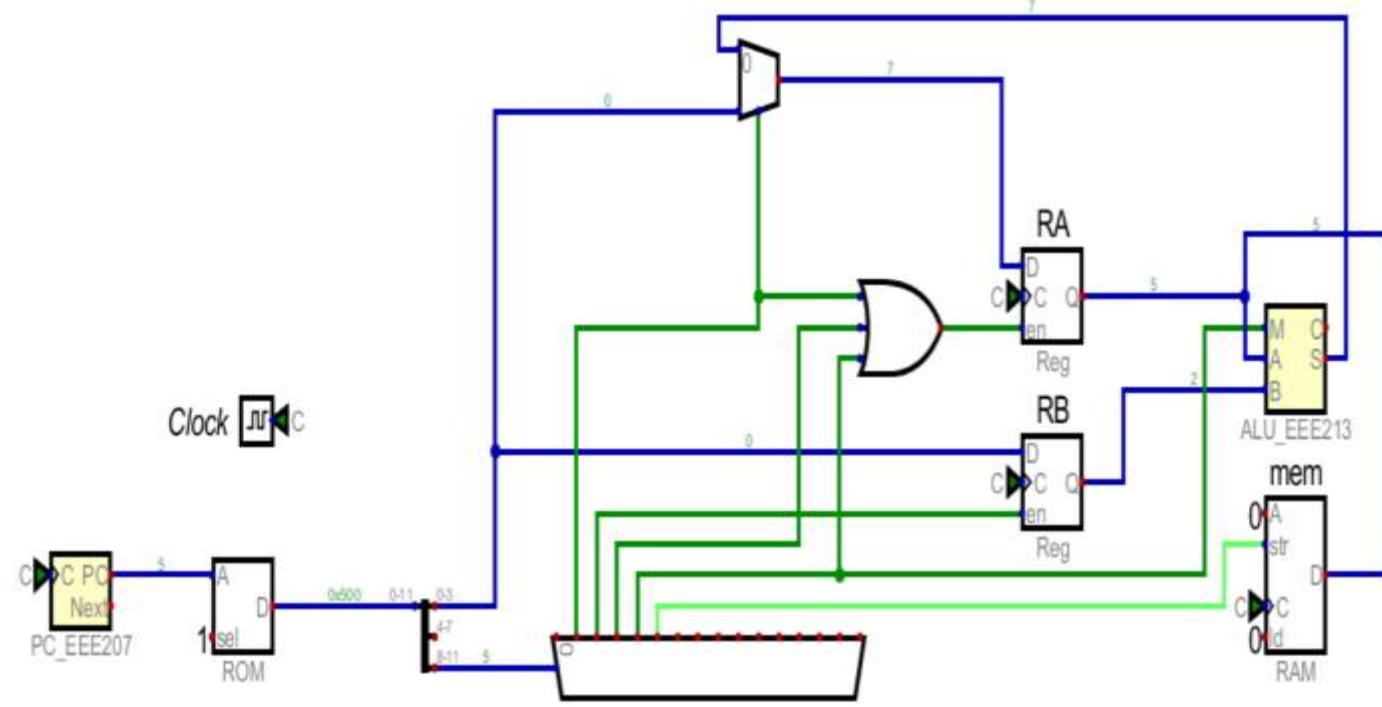
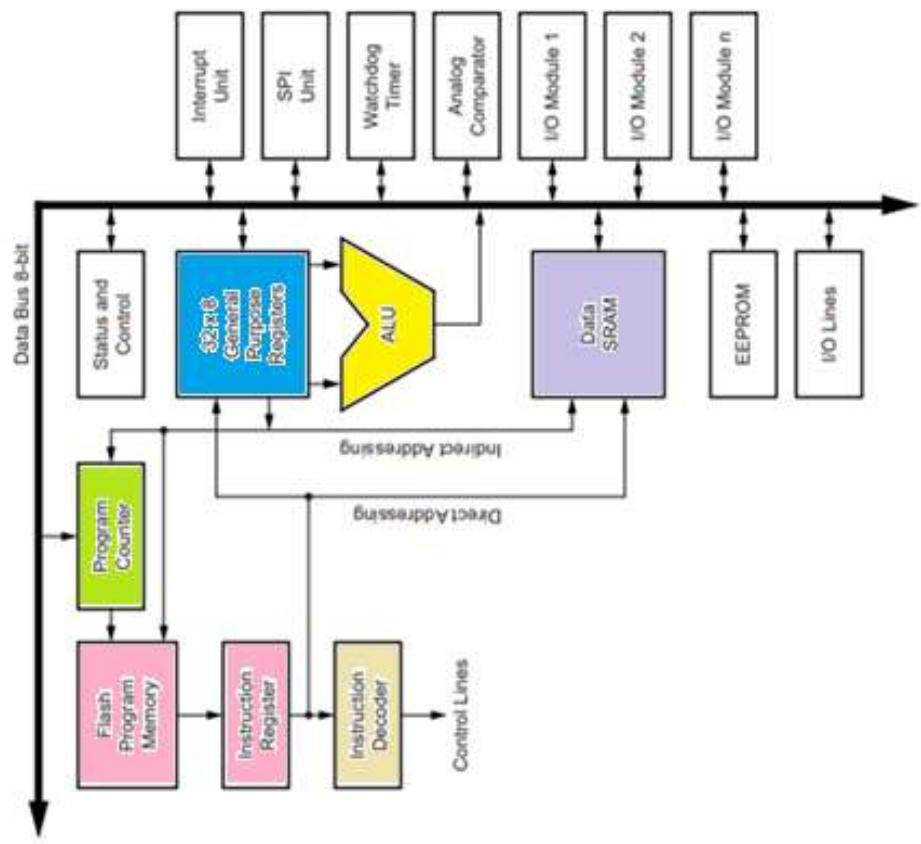


Figure 7.11 Complete single-cycle MIPS processor

AVR vs MyCPU



Verilog Summary

```
// Verilog Reference Code
// Senol Gulgonul
// 19.11.2023
module reference(
    input A,B,clk,
    output Fand,Fxor,Fcond,
    output reg Fif,
    output reg [3:0] counter
);
    and G1 (Fand,A,B);
    assign Fxor=(~A&B)|(A&~B);
    assign Fcond=A?1:0;
    initial begin
        counter=4'b0000;
    end // initial
    always @(posedge clk) begin
        counter<=counter+1;
        case (counter)
            4'b0001: if (A==0) Fif<=1; else Fif<=0;
            4'b0011: counter<=4'b0000;
            default: Fif<=0;
        endcase
    end //always
endmodule
```

```
// Verilog Reference Testbench
// Senol Gulgonul
// 19.11.2023
`timescale 1ns/1ns
module testbench;
    reg A,B,clk; //inputs
    wire Fand,Fxor,Fcond; //outputs
    wire Fif;
    wire [3:0] counter;
    reference M1 (.A(A),.B(B),.clk(clk),
    .Fand(Fand),.Fxor(Fxor),.Fcond(Fcond),.Fif(Fif),.counter(counter));
    initial begin
        clk=0;
        forever #1 clk = ~clk;
    end
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars();
    end
    initial begin
        $monitor("time=%0t A=%b B=%b Fand=%b Fxor=%b Fcond=%b Fif=%b
        counter=%d",$time,A,B,Fand,Fxor,Fcond,Fif,counter);
    end
    always @(posedge clk) begin
        A=counter[1];
        B=counter[0];
    end
    initial #20 $finish;
endmodule
```