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 mor

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 mor

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 o 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 sate 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

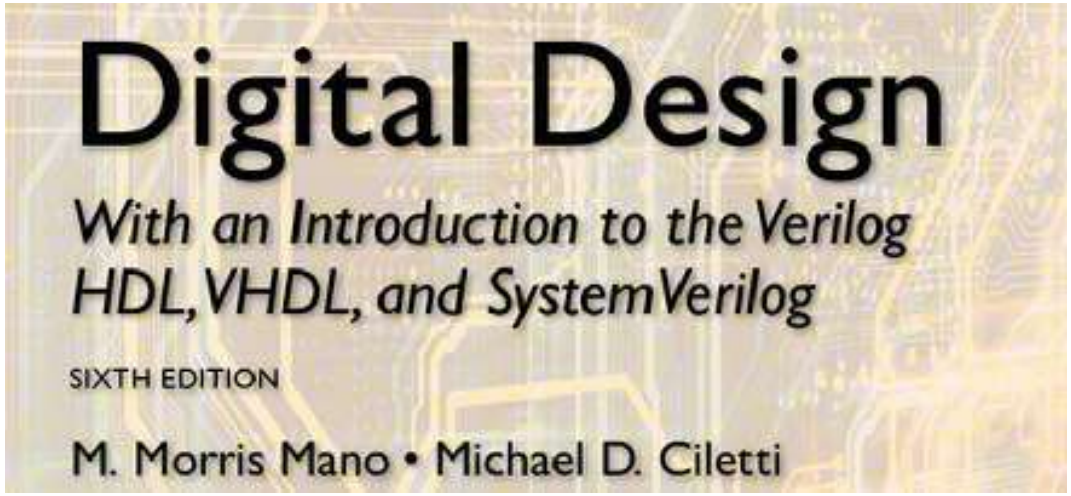
Sakarya University
Doctor of Philosophy - PhD
2009 - 2014

ABOUT

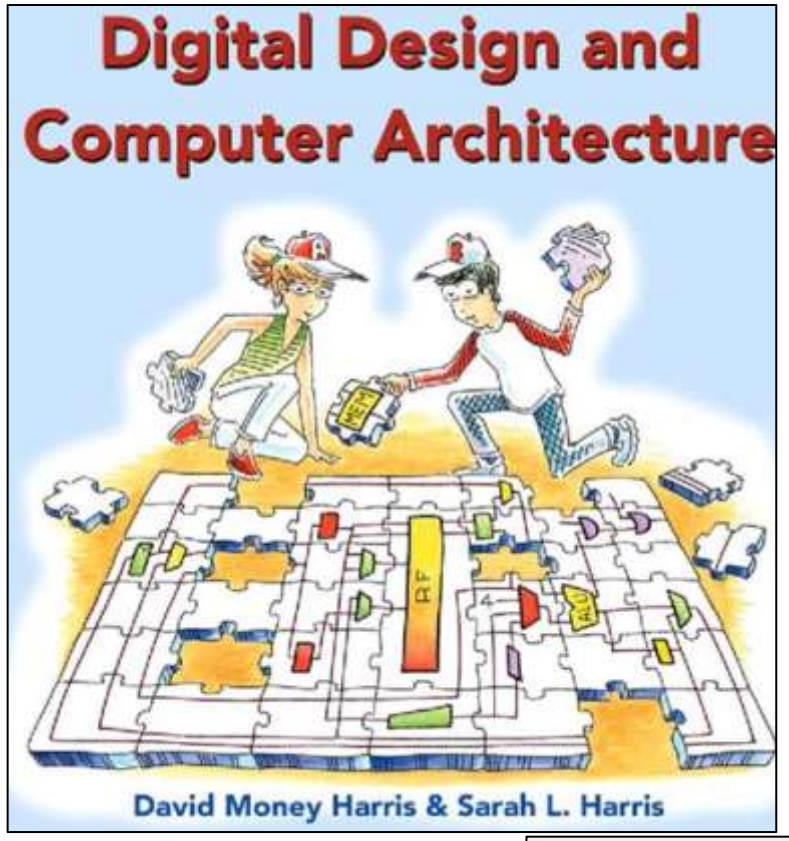
- Text Book: Digital Design, Global Edition Mano, M. Morris, Ciletti, Michael

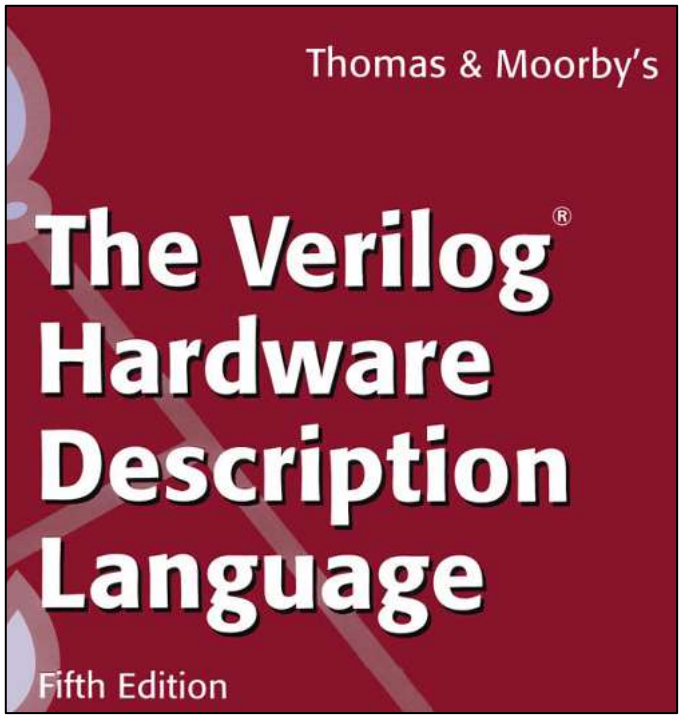
- Grades: 20% Midterm + 20% Project + 60% Final

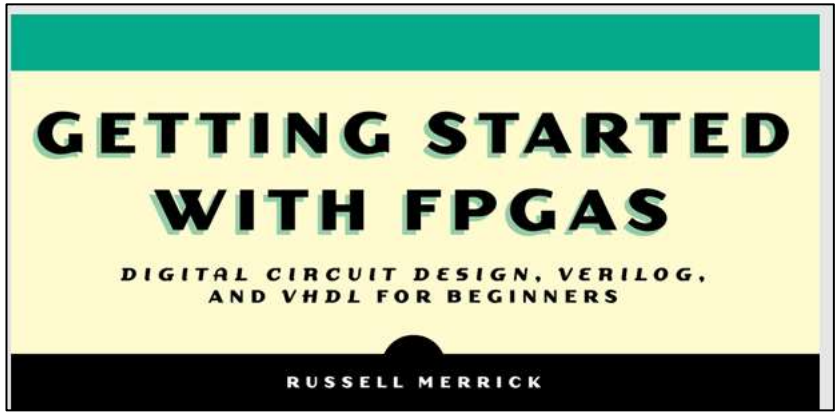
- Course Objectives: Learning Verilog Hardware Description Language for logic circuit testing, design and applications using FPGAs



Additional Resources







WHY

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

SYLLABUS

Week	Subject
1	Boolean Algebra and Logic Gates
2	Combinational Logic
3	Sequential Logic
4	Combinatorial logic circuits using Verilog
5	Combinatorial logic circuits using Verilog
6	**Midterm**
7	Sequential logic circuits using Verilog
8	Sequential logic circuits using Verilog
9	Sequential logic circuits using Verilog
10	Sequential logic circuits using Verilog
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]

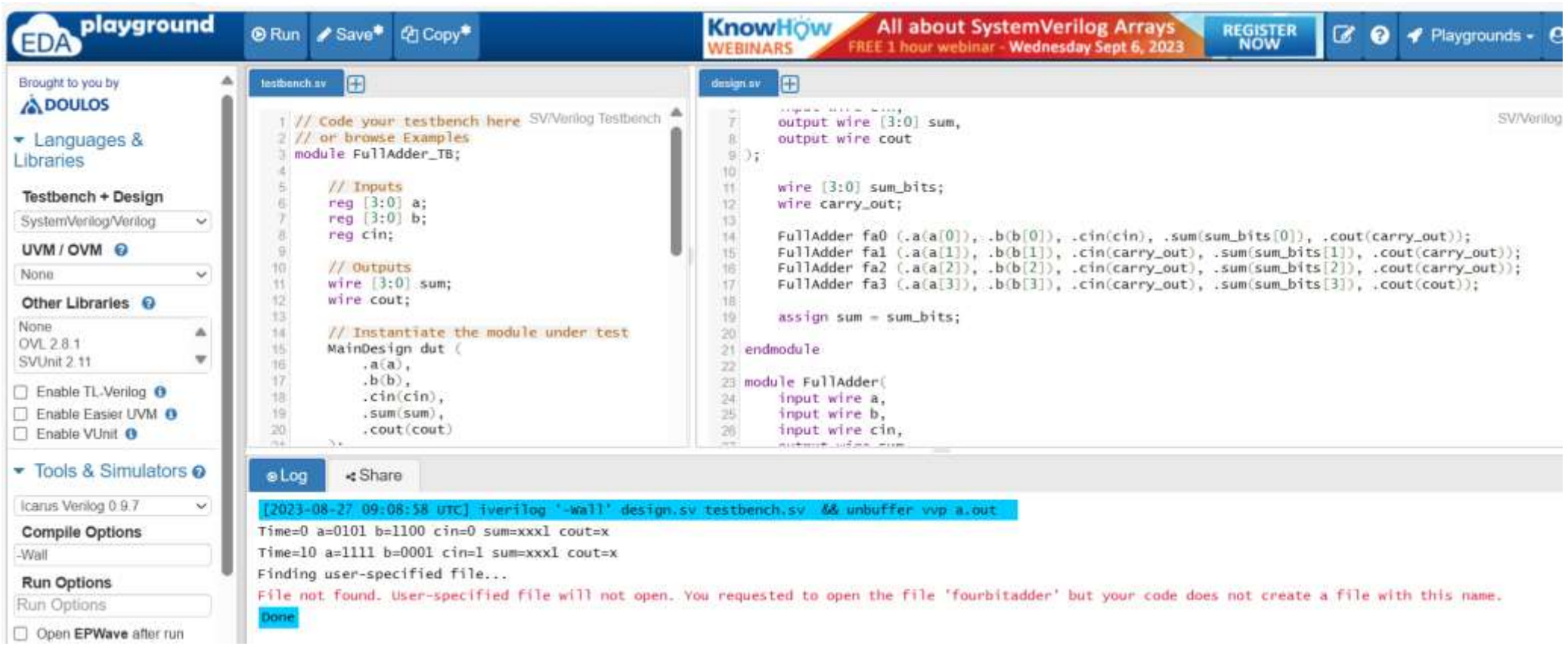
- Icarus Simulator + GTKWave
- [Icarus Verilog for Windows (bleyer.org)]
- [GTKWave (sourceforge.net)]

- Gowin TANG NANO 9K for FPGA implementation
- Saleae Logic Analyzer
- Breadboad
- M-M jumper cables

- ChatGPT - Copilot

EDAplayground for Verilog

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

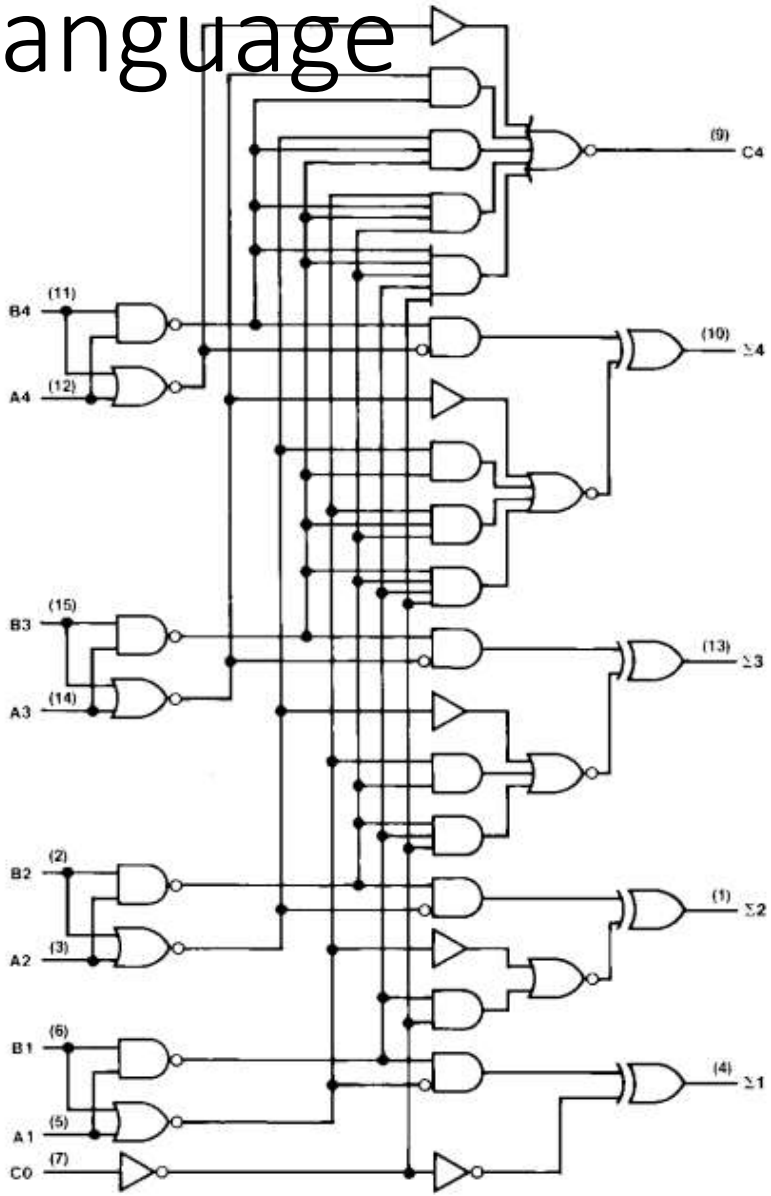


Verilog : Hardware Description Language

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?

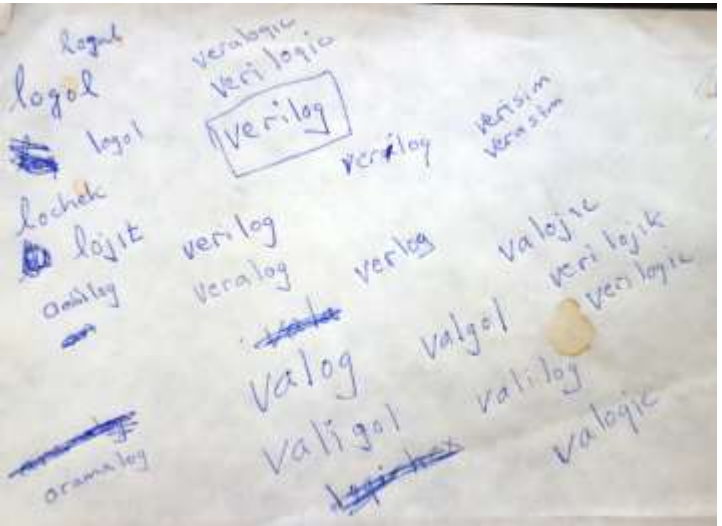
1994
Prof. Thomas Nicely reports bug in Pentium
Restoring Division
Logic error not caught until > 1M units shipped
Recall cost $450M (!!!)

1997-2000
All major micro-processor manufacturers adopt formal verification.

- Verilog allows logic circuit designer to test

Verilog

PHILIP MOORBY
1953-2022



Verilog History

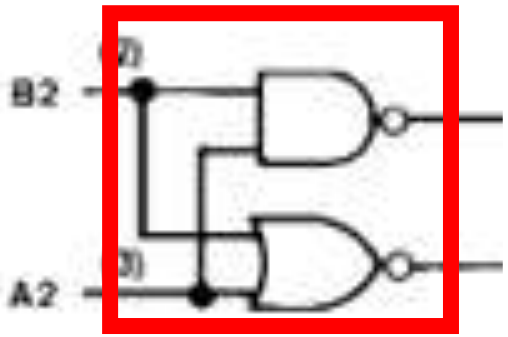
- Introduced in 1984 by Gateway Design Automation
- 1989 Cadence purchased Gateway and subsequently released Verilog to the public
- Open Verilog International (OVI) was formed to control the language specifications
- 1995 IEEE accepted OVI Verilog as a standard
- 2001 and 2005 IEEE revised standard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

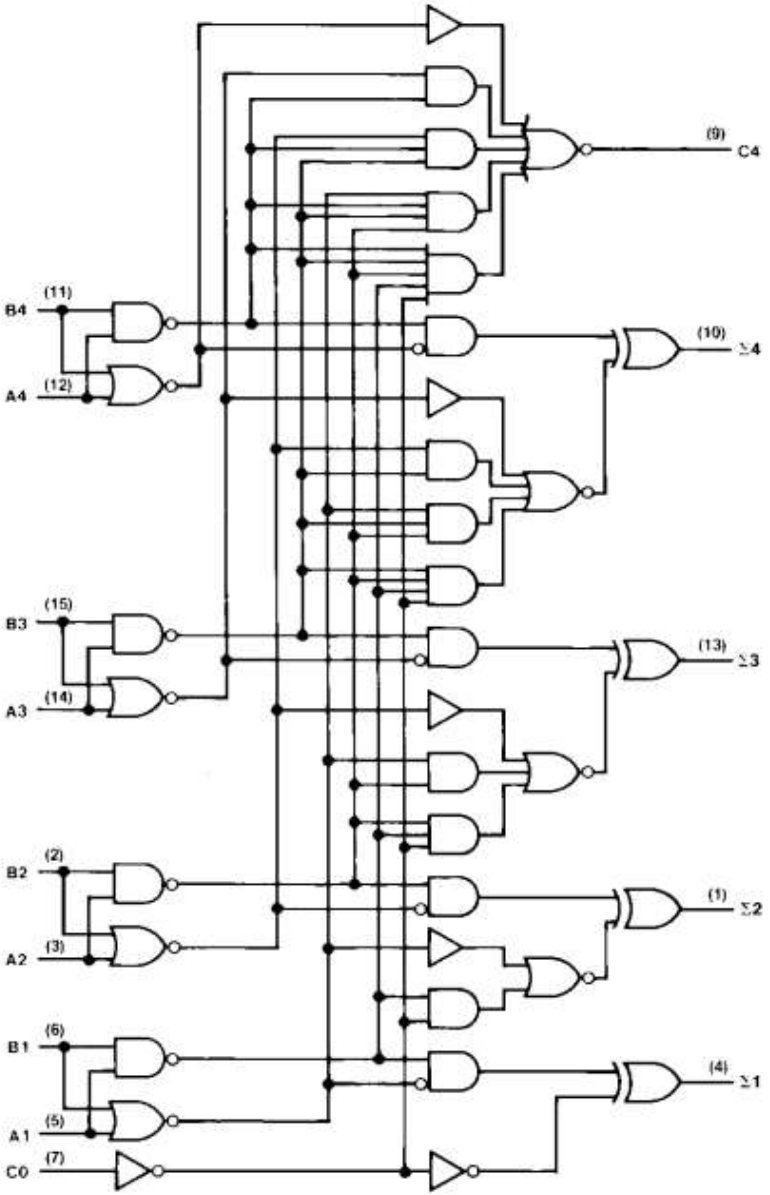
ALTERA
MEASURABLE ADVANTAGE™

[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**

```
module twogates (
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

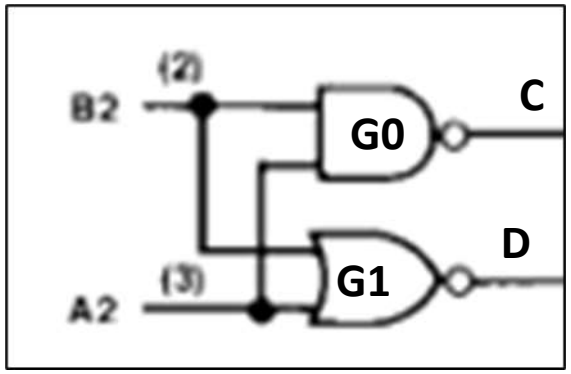
ANSI-C Style Port List *(added in Verilog-2001)*

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

5.2 Port Declarations

Combined Declarations *(added in Verilog-2001)*

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



verilog_2001_ref_guide.pdf (sutherland-hdl.com)

Verilog Ports

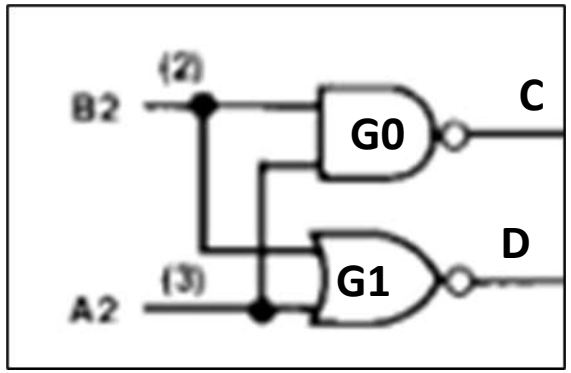
- **port direction** is either input, output
- If no **data type** is declared, it is implicitly declared as a **wire**
- **no comma at the last port**

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




```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

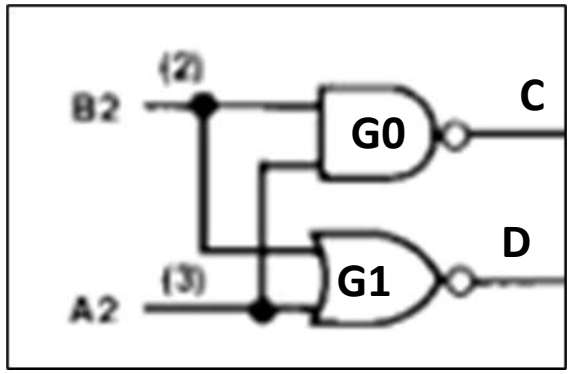
Verilog Data Types

- 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

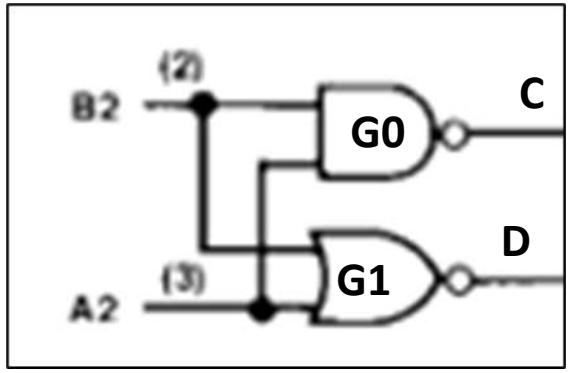
Combined Declarations *(added in Verilog-2001)*

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




```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

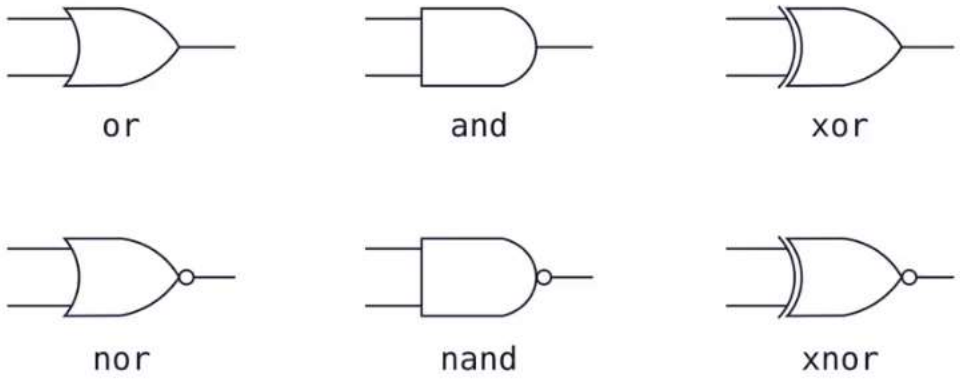
Verilog Primitives



- 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*, ...) ;

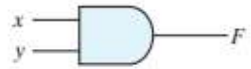
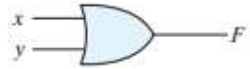
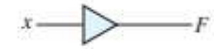
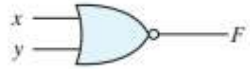
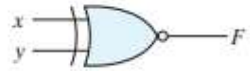



```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

Verilog Primitives

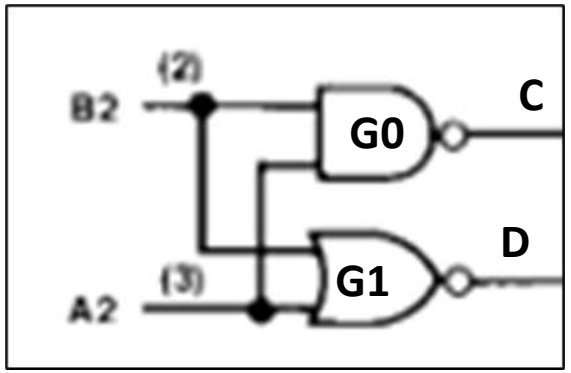
- Verilog has built-in **primitives** to define **basic gates**
- **not, and, nand, or, nor, xor, xnor**

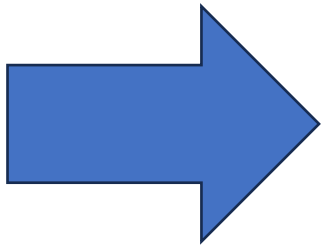
Gate Primitives		Terminal Order and Quantity
`and`	`nand`	(1–output, 1-or-more–inputs)
`or`	`nor`	
`xor`	`xnor`	

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	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





```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

Gate-Level Modeling

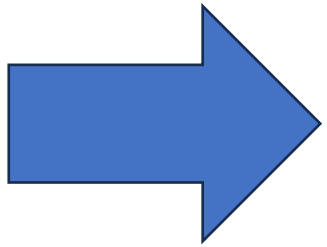
- **Gate-level** models describe how a circuit is composed of other interconnected elements, such as logic gates or functional blocks.

```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

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 twogates(C,D,A,B);
  output C,D;
  input A,B;


nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```



```
//Verilog 2001
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

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

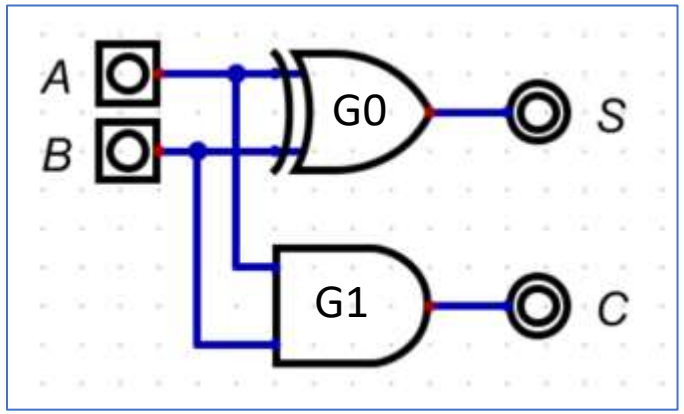
```
module module_name
    #(parameter_declaration, parameter_declaration,... )
    (port_declaration port_name, port_name,...,
     port_declaration port_name, port_name,...);
    module items
endmodule
```

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	(1–output, 1-or-more–inputs)
or	nor	
xor	xnor	

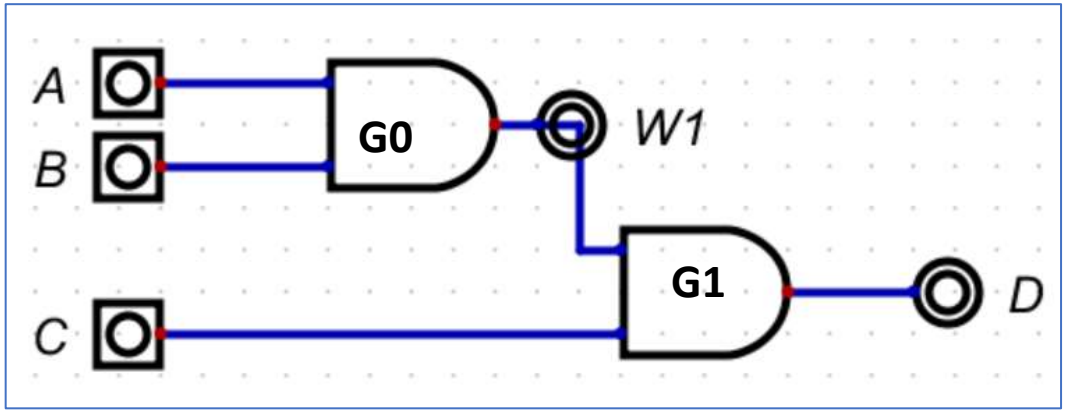
Verilog Net Data Types

- 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
```

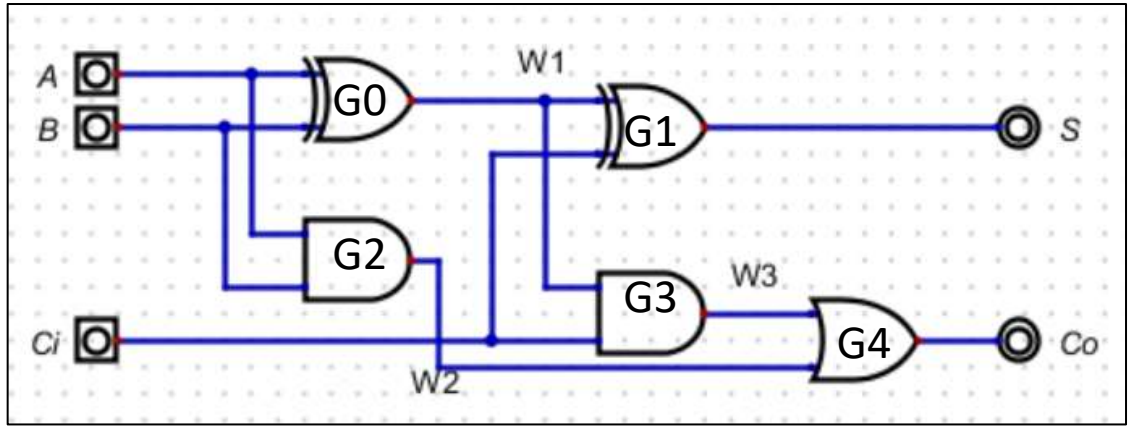
Net Data Type

Net data types connect structural components together.



Complex Combinatorial Circuits: Full Adder

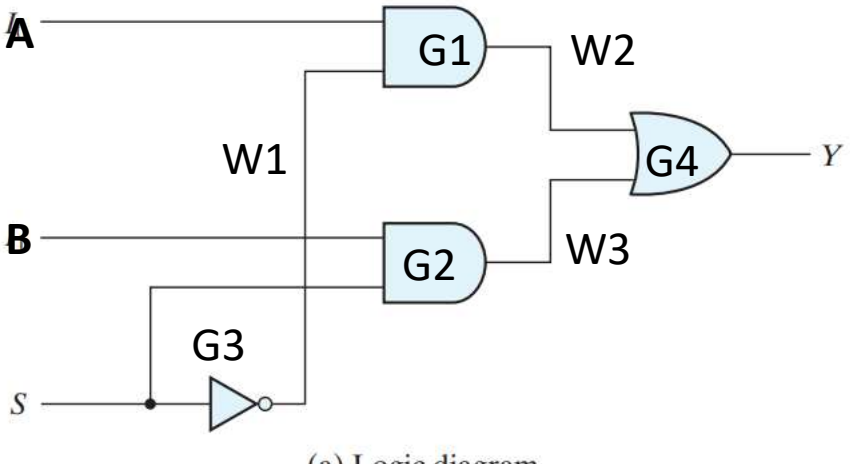
- Using Gate-level modeling we can define any **combinatorial logic** circuit
- name gates G0, G1, etc.
- name wires as W1, W2, etc. for the intermediate connection points
- Sunday puzzle sudoku ☺




```
//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
```

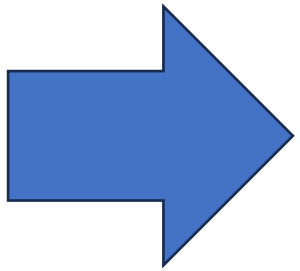
Complex Combinatorial Circuits: MUX

- 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** ☺



(a) Logic diagram

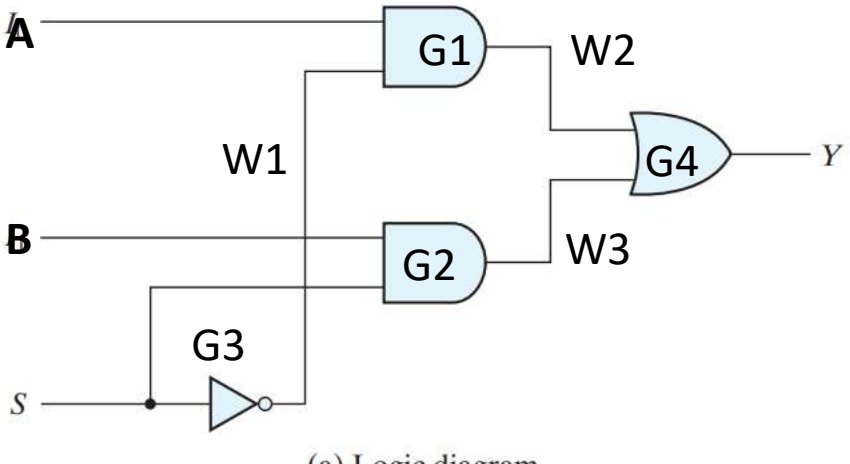
FIGURE 4.24
Two-to-one-line multiplexer



```
//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** ☺



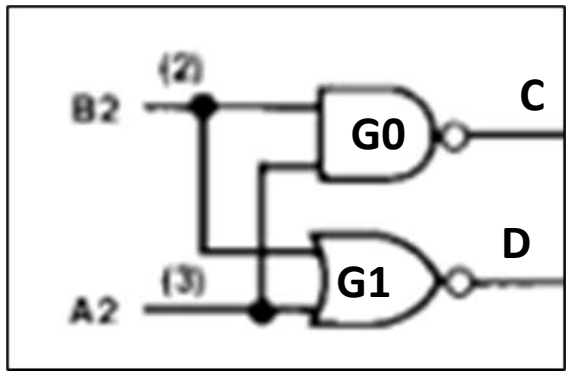
(a) Logic diagram

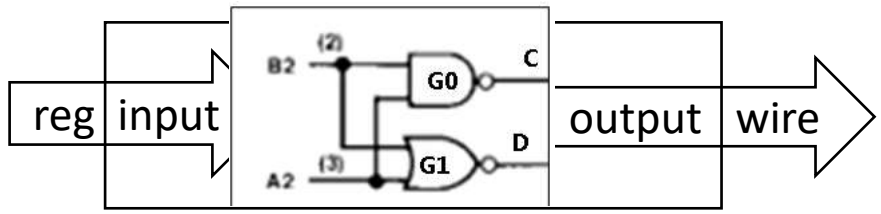
FIGURE 4.24
Two-to-one-line multiplexer

```
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**.





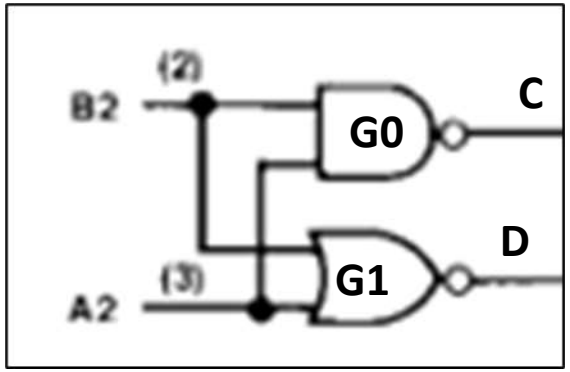
tb_twogates

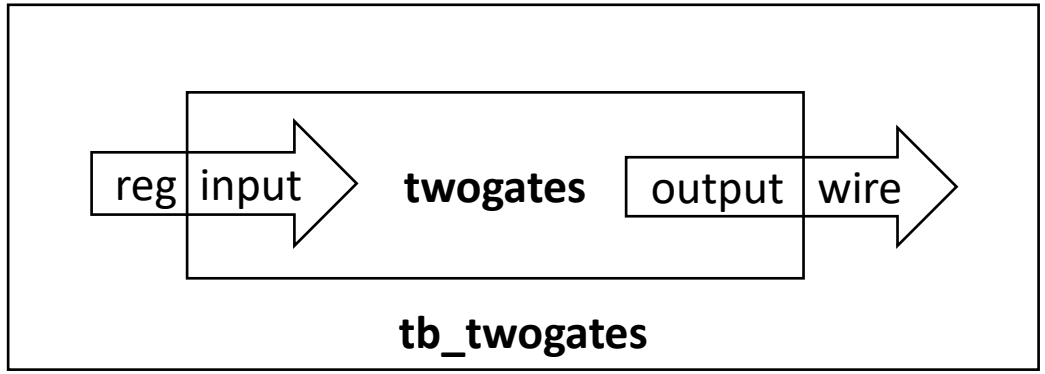
```
//two gates
module twogates(
  output C,D,
  input A,B
);
nand G0 (C,B,A);
nor G1 (D,B,A);
endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
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 tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

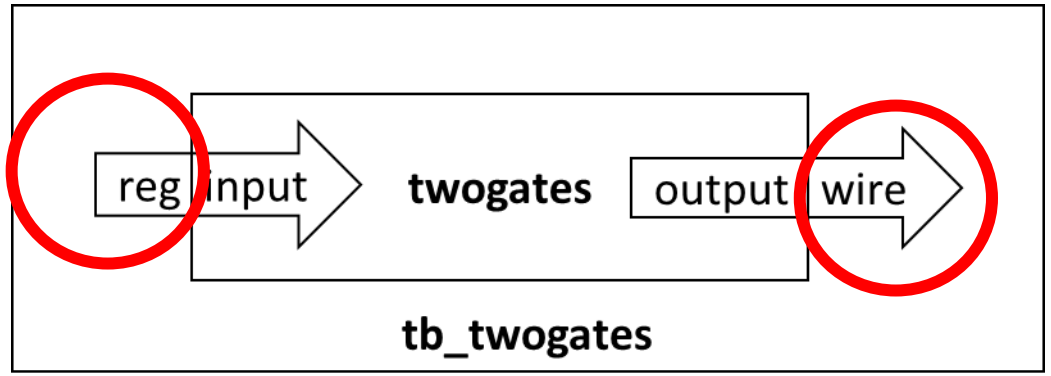
Verilog Variable Data Types

- *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:

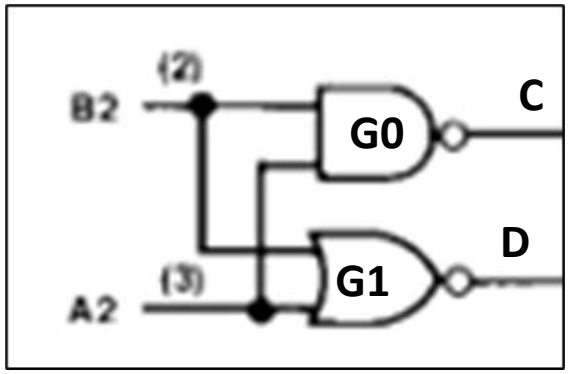
wire	interconnecting wire; CMOS resolution

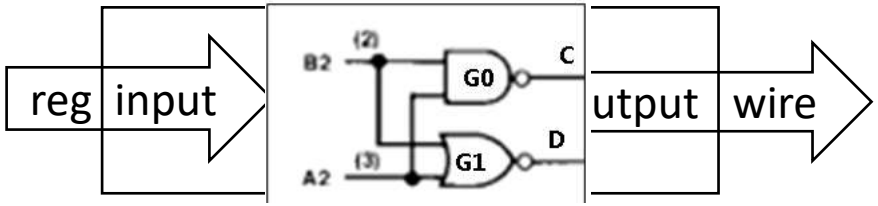



```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
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**





tb_twogates

```
//two gates
module twogates(
  output Ali,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.Ali(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

Verilog Test Bench: Instantiate

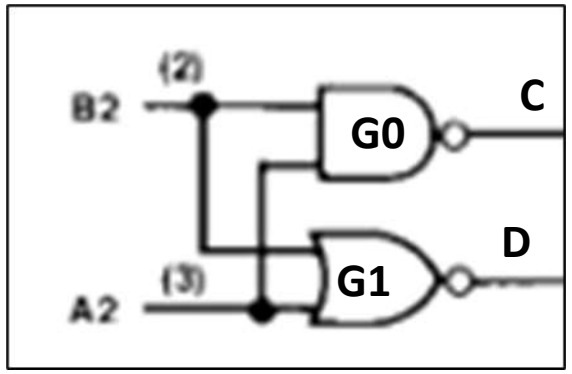
- in instantiation, order or parameters not important
- parameters are Case Sensitive Ali is not equal ali
- port connections are defined by name .Ali(c)

```
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;  //wire for outputs

  //instantiation
  twogates M0 (.Ali(c),.D(d),.A(a),.B(b));
```

```
//two gates
module twogates(
  output D,Ali,
  input A,B
);
nand G0 (Ali,B,A);
nor G1 (D,B,A);
endmodule
```

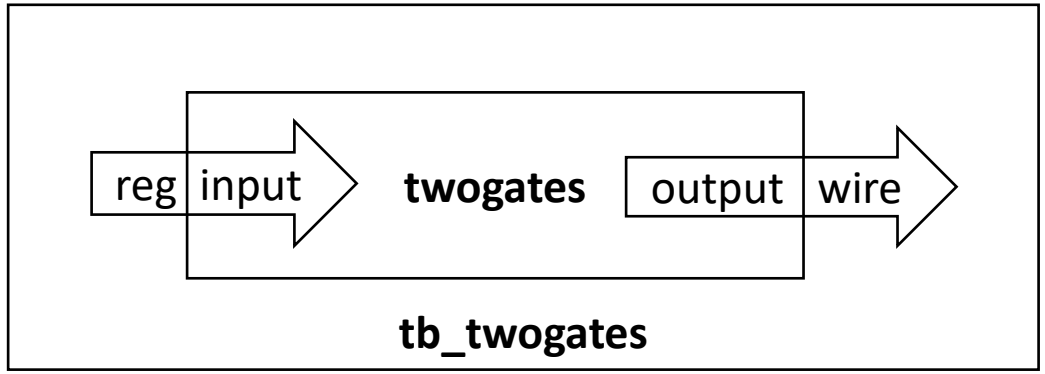
Verilog Test Bench: initial

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




```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```



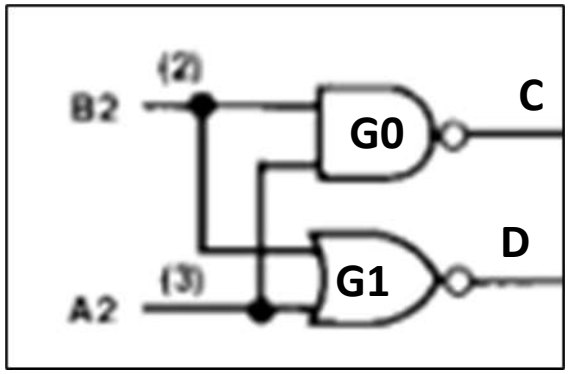
Verilog Test Bench: waveform

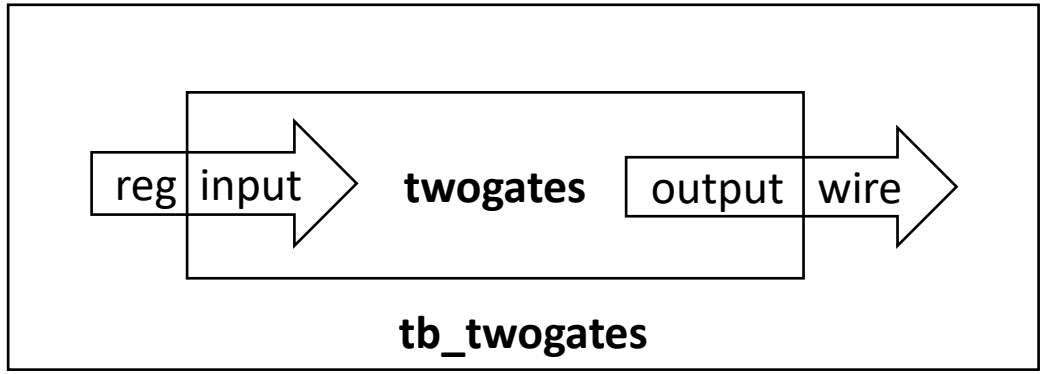
- besides text printout of $display and $monitor we can examine time waveform analysis of module under test
- add $dumpfile and $dumpvars
- $dumpfile will dump the changes in a file named .vcd
- $dumpvars selects which variables to be dumped into dumpfile. no variable declared means **dump all variables** 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 tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

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

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






```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
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

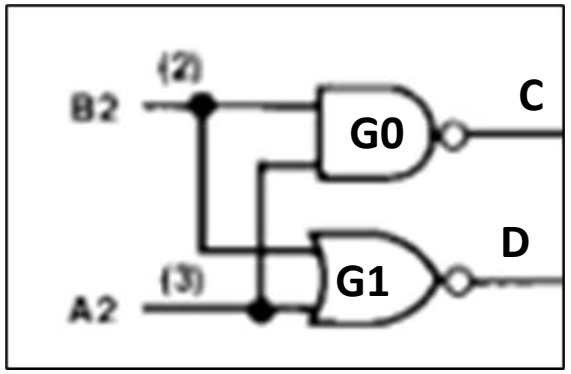
`%b`	binary values	`%m`	hierarchical name of scope
`%o`	octal values	`%l`	configuration library binding
`%d`	decimal values	`\t`	print a tab
`%h`	hex values	`\n`	print a newline
`%e`	real values–exponential	`\"`	print a quote
`%f`	real values–decimal	`\\`	print a backslash
`%t`	formatted time values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

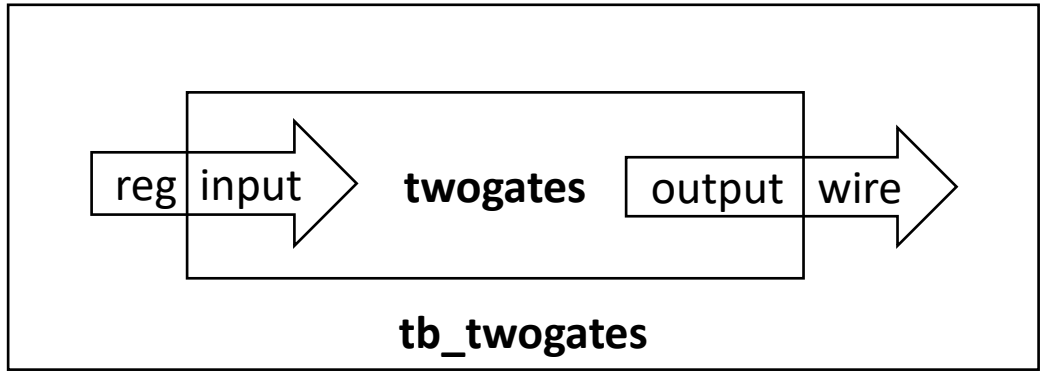
`%0b`, `%0o`, `%0d` and `%0h` truncates any leading zeros in the value.
`%e` and `%f` may specify field widths (e.g. `%5.2f`).
`%m` and `%l` do not take an argument; they have an implied argument value.
The format letters are not case sensitive (i.e. `%b` and `%B` are equivalent).

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.






```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
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		

%0b, %0o, %0d and %0h 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).

```
test results:
t=   ab:cd
t= 0 00:11
t= 1 01:10
t= 3 10:10
t= 6 11:00
```

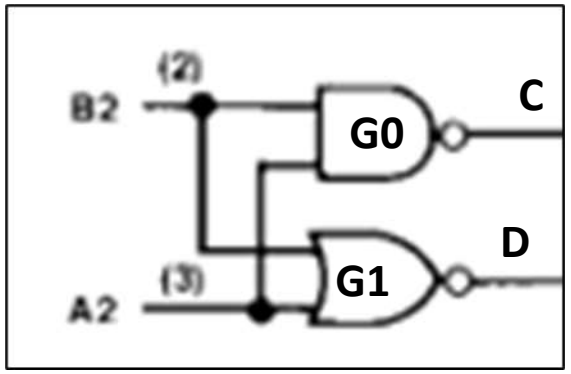
```
//testbench
`timescale 10 ns / 1 ns
module tb_twogates ();
 reg  a,b;  //reg for inputs
 wire c,d;  //wire for outputs

 //instantiation
 twogates M0 (.C(c),.D(d),.A(a),.B(b));

 initial begin
  $dumpfile("dump.vcd"); $dumpvars;
  $display("test results:");
  $display("t=   ab:cd");

  $monitor("t=%2d %b%b:%b%b",$time,a,b,c,d);
   a=0; b=0;
//    #1 a=0; b=0;
  #1 a=0; b=1;
  #2 a=1; b=0;
  #3 a=1; b=1;
  #10 $finish;
 end
endmodule
```

Verilog Test Bench: assignment

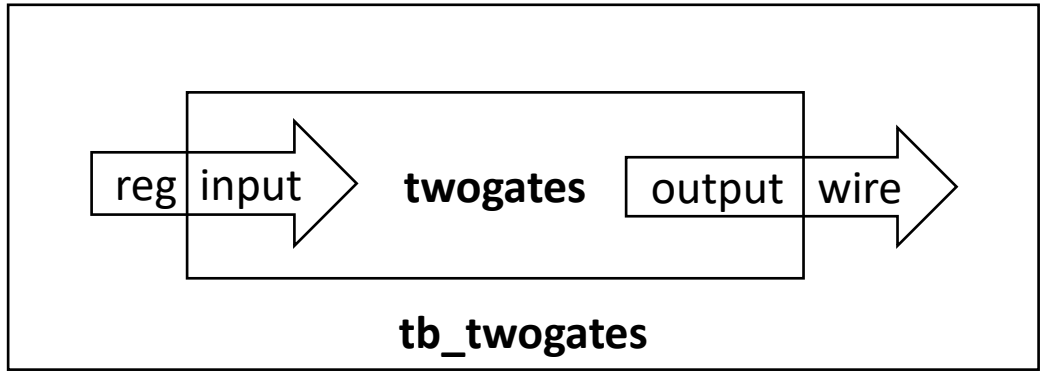
10.3 Procedural Assignment Statements

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`.




```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```




```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;   //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
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 @

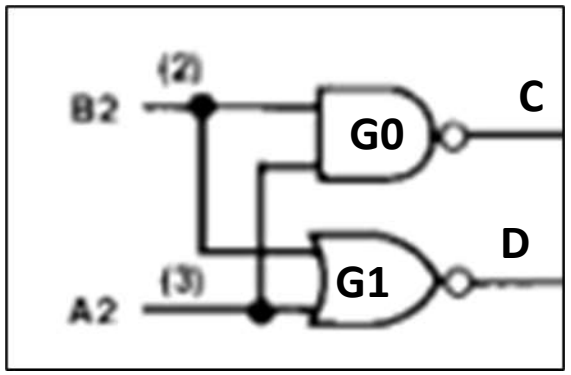
```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

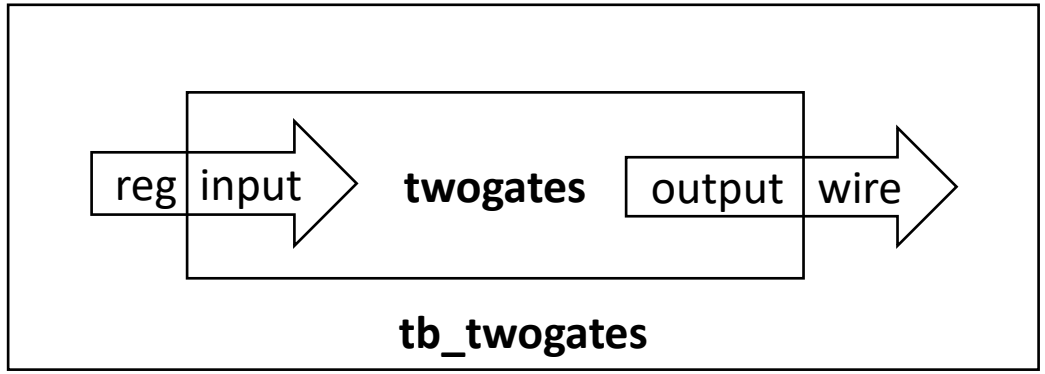
```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

Verilog Test Bench: review results

truth table:
ab:cd
00:11
01:10
10:10
11:00

ab:cd
xx:xx
00:11
01:10
10:10
11:00



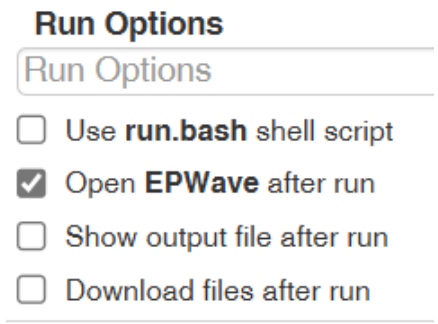


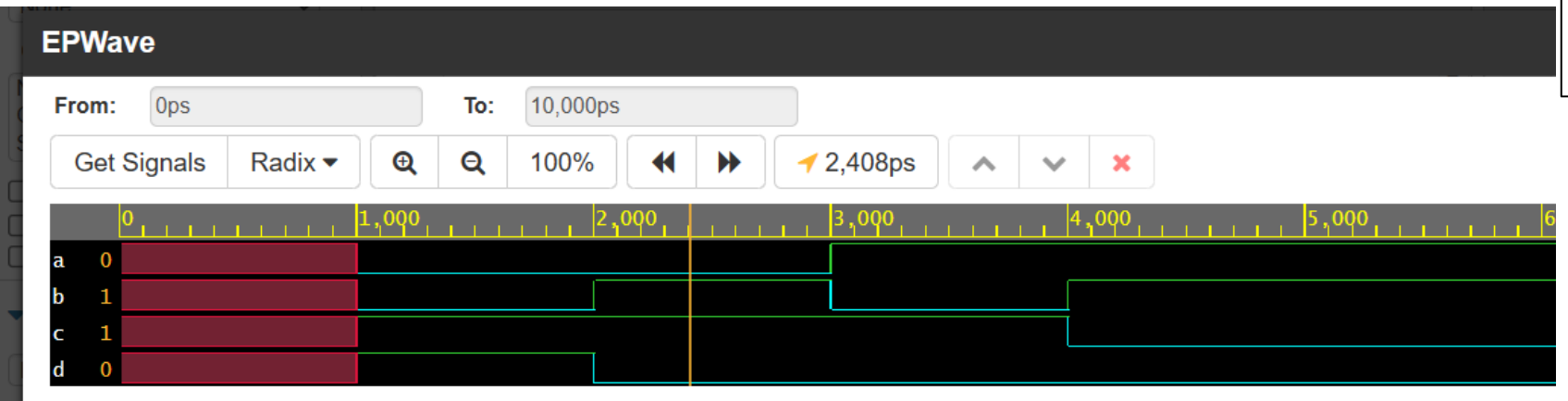

```
//two gates
module twogates(
  output C,D,
  input A,B
);

nand G0 (C,B,A);
nor G1 (D,B,A);

endmodule
```

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
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 tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

Verilog Test Bench: iverilog

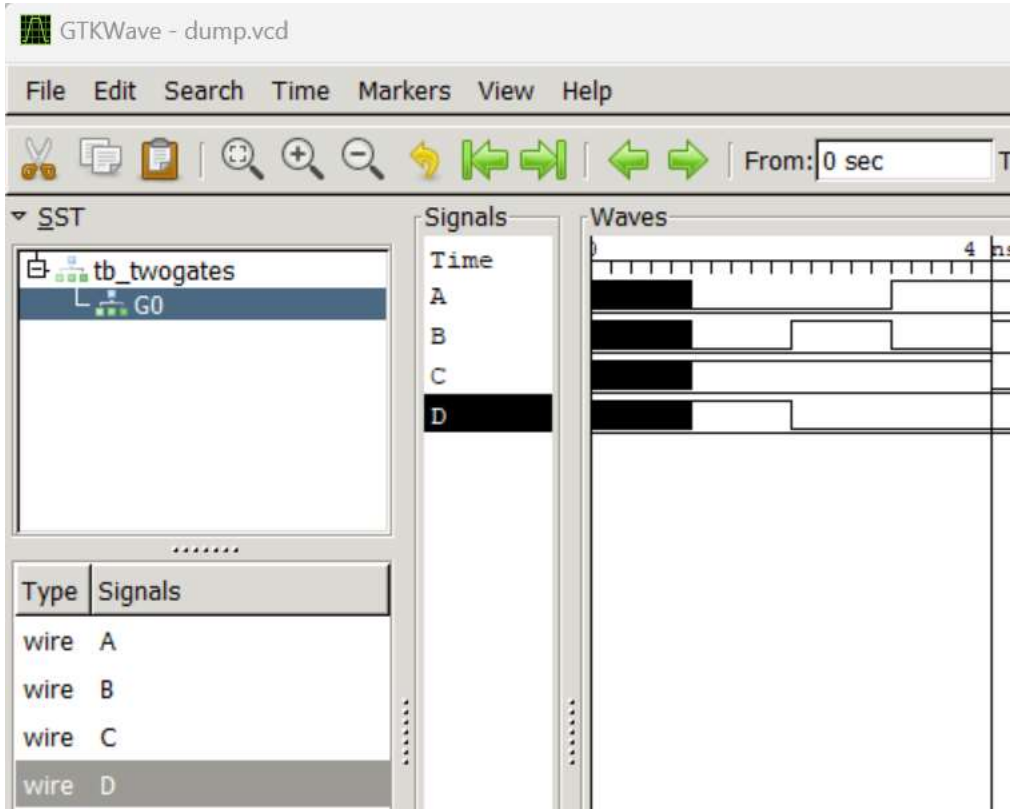
- we can run simulation in local computer using **iverilog** and **gtkwave**

- **save design and testbench files using a text editor (notepad)**
- **in command line:**
- **iverilog -o twogates.out twogates.v twogates_tb.v; vvp twogates.out**
- **will print:**
- **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)**

```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

Verilog Test Bench: gtkwave

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




```
//testbench
`timescale 1 ns / 1 ns
module tb_twogates ();
  reg  a,b;  //reg for inputs
  wire c,d;    //wire for outputs
  //instantiation
  twogates M0 (.C(c),.D(d),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cd");
    $monitor("%b%b:%b%b",a,b,c,d);
    #1 a=0; b=0;
    #1 a=0; b=1;
    #1 a=1; b=0;
    #1 a=1; b=1;
    #10 $finish;
  end
endmodule
```

Verilog Test Bench: iverilog and gtkwave

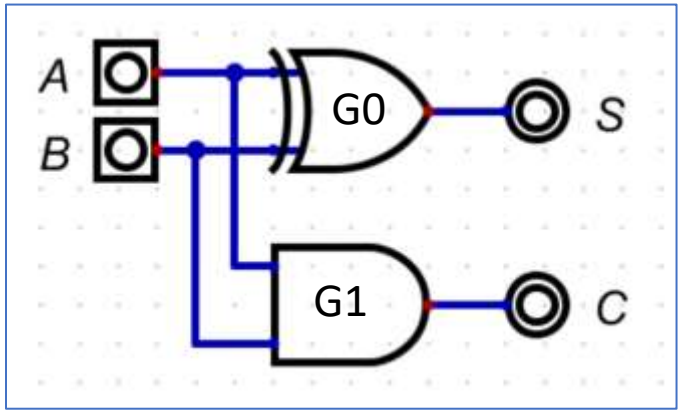
- **command line copy shown here**

```
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
```

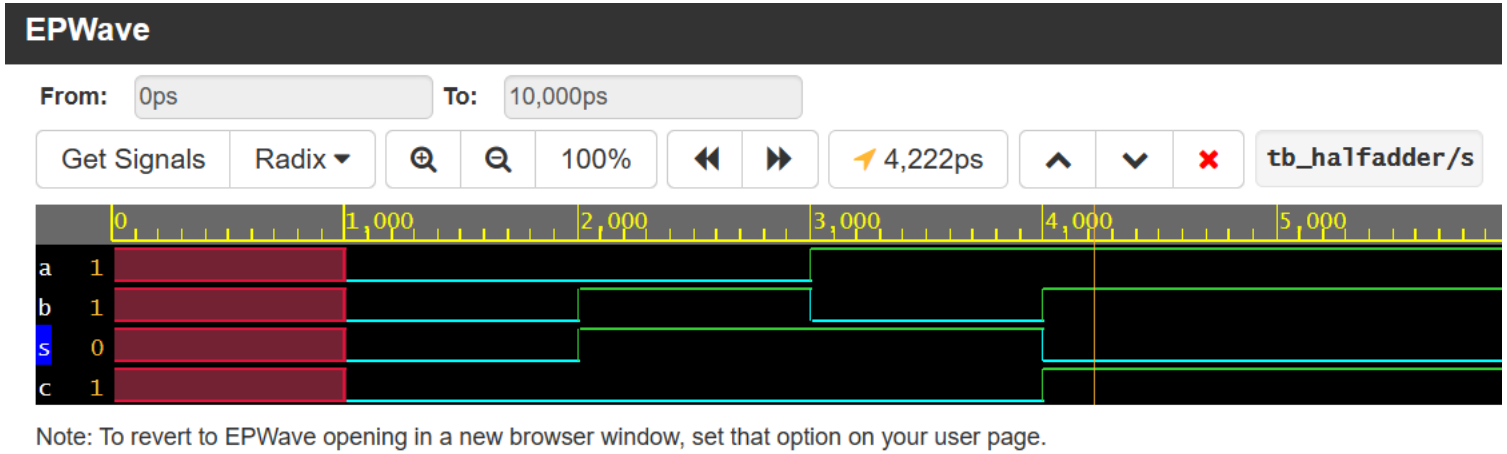
Verilog: Half Adder

- 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
```




```
xx:xx
00:00
01:10
10:10
11:01
```



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

```
//testbench
`timescale 1 ns / 1 ns
module tb_halfadder ();
  reg  a,b;  //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("ab:sc");
    $monitor("%b%b:%b%b",a,b,s,c);
    #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 x 10^{14} inputs per day
    - or 3.15 x 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
```

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
```

Verilog: Self-checking Testbench

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

```
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
```

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

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
```

```
0000
0101
1001
1110
```

```
Test passed for vector 0: A=0, B=0, C=0, S=0
Test passed for vector 1: A=0, B=1, C=0, S=1
Test passed for vector 2: A=1, B=0, C=0, S=1
Test passed for vector 3: A=1, B=1, C=1, S=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
```

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.

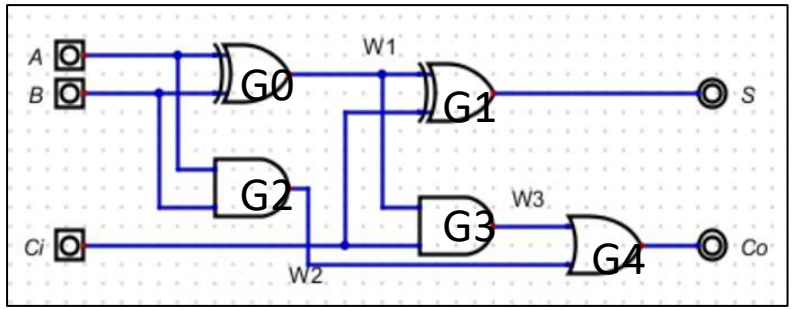
{}	{m,n}	concatenate m to n, creating a larger vector

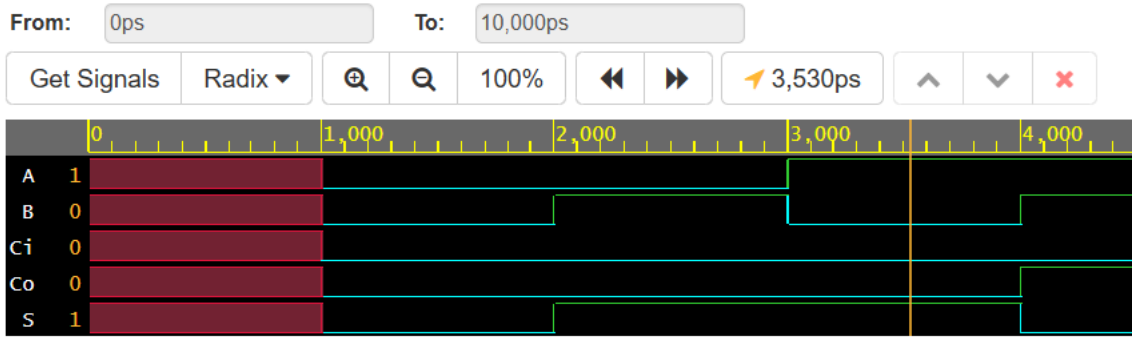
```
// Test vector storage
   reg [3:0] test_vectors [0:3]; // 4 bits wide, 4 test cases
   integer i;

// 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);
```

Complex Combinatorial Circuits: 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
```

```
abci:sco
xxx:xx
000:00
010:10
100:10
110:01
```



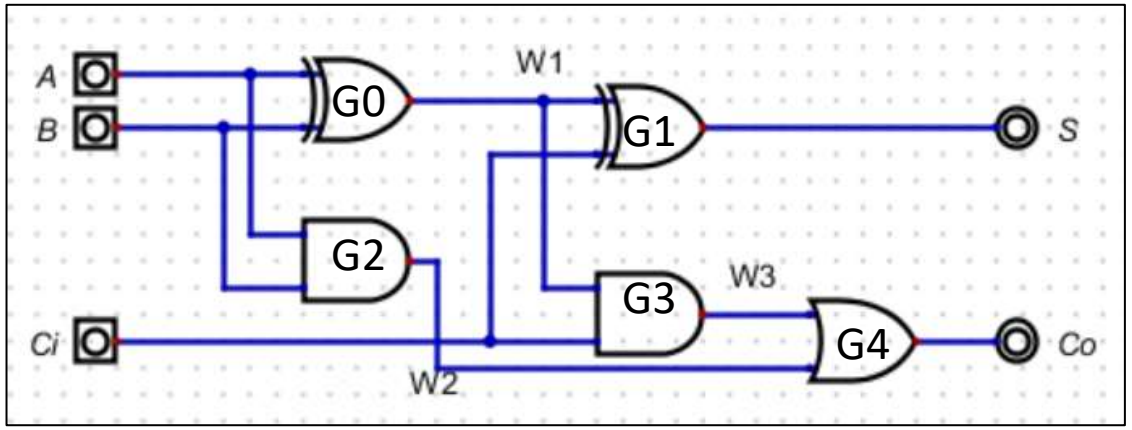



```
//testbench
`timescale 1 ns / 1 ns
module tb_fulladder ();
  reg  a,b,ci;  //reg for inputs
  wire s,co;    //wire for outputs
  //instantiation
  fulladder M0 (.S(s),.Co(co),.A(a),.B(b),.Ci(ci));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("abci:sco");
    $monitor("%b%b%b:%b%b",a,b,ci,s,co);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

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

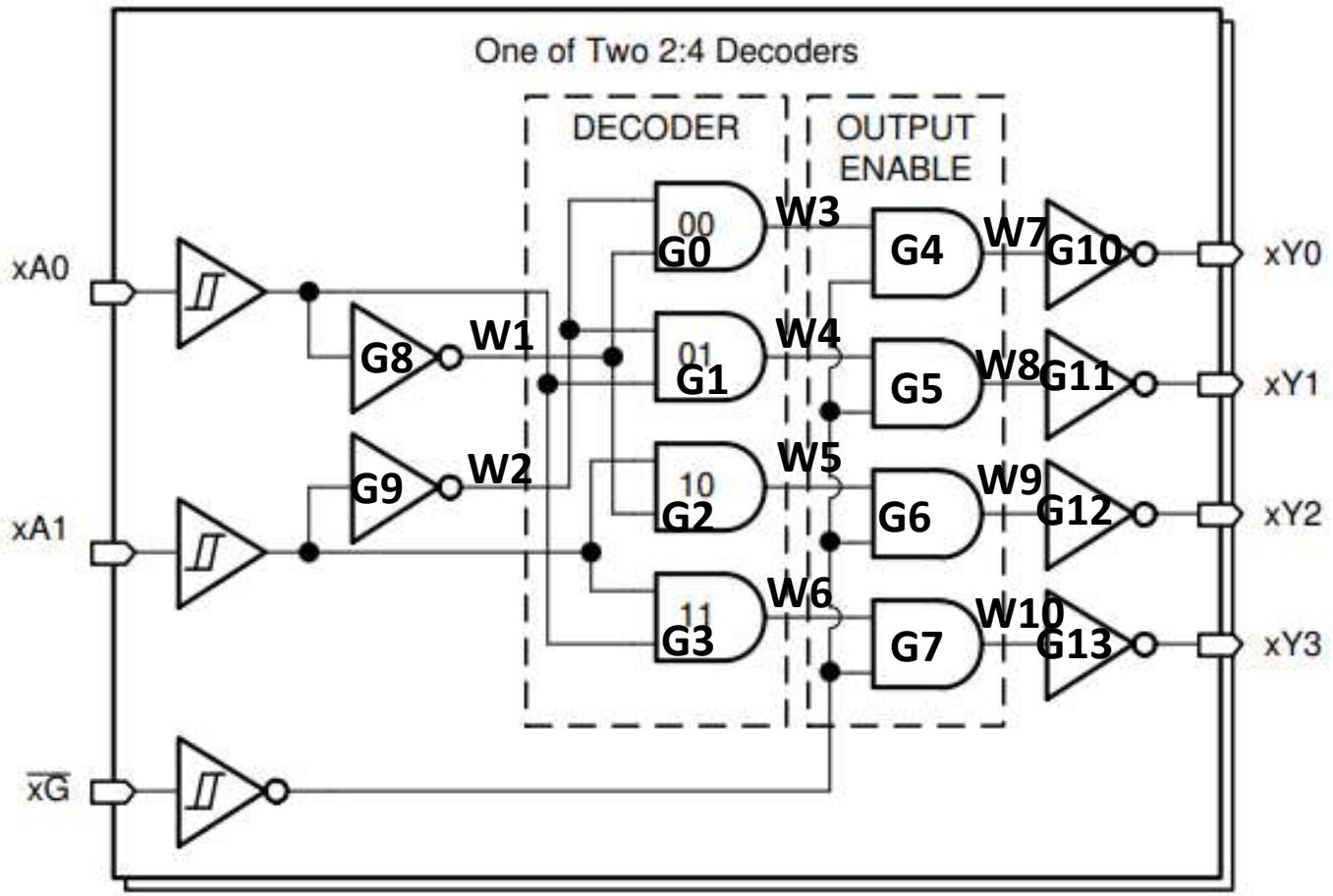


Figure 8-1. Logic Diagram (Positive Logic) for SN74HCS139-Q1 -Q1

```
//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
```

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

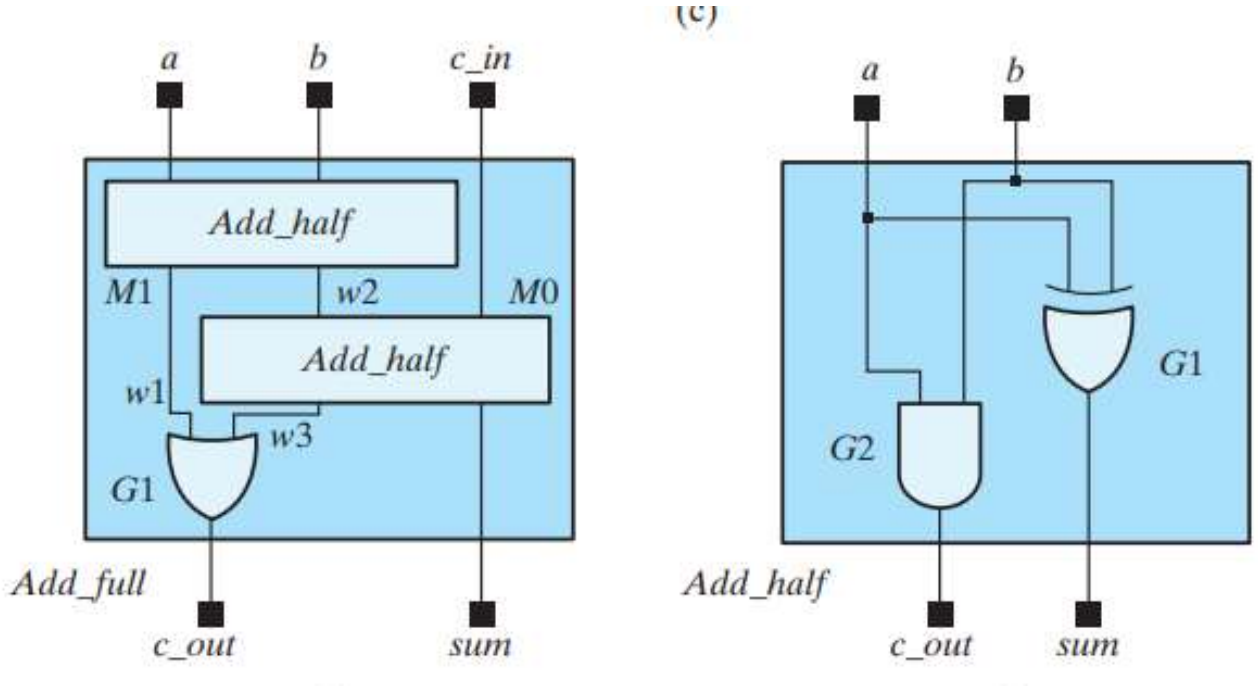
INPUTS[1]			OUTPUTS			
$n\overline{G}$	nA1	nA0	nY0	nY1	nY2	nY3
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L
H	X	X	H	H	H	H

```
A1A0G : Y3Y2Y1Y0
x x x : x x x x
0 0 1 : 1 1 1 0
0 1 1 : 1 1 0 1
1 0 1 : 1 0 1 1
1 1 1 : 0 1 1 1
0 0 0 : 1 1 1 1
0 1 0 : 1 1 1 1
1 0 0 : 1 1 1 1
1 1 0 : 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("A1A0G : Y3Y2Y1Y0");
    $monitor("%b %b %b : %b %b %b %b"
           ,a1,a0,g,y3,y2,y1,y0);
    #1 a1=0; a0=0; g=1;
    #1 a1=0; a0=1; g=1;
    #1 a1=1; a0=0; g=1;
    #1 a1=1; a0=1; g=1;
    #1 a1=0; a0=0; g=0;
    #1 a1=0; a0=1; g=0;
    #1 a1=1; a0=0; g=0;
    #1 a1=1; a0=1; g=0;
    #10 $finish;
  end // initial
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 (.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: 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**

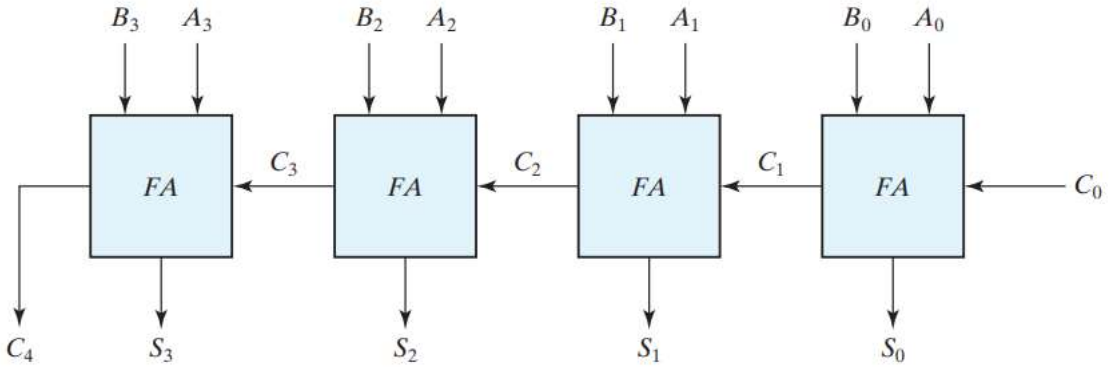
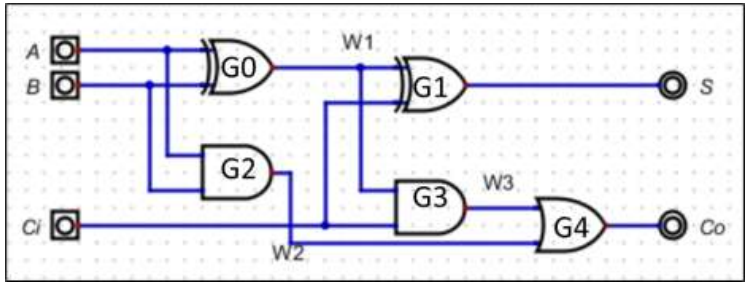


FIGURE 4.9
Four-bit adder




```
//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

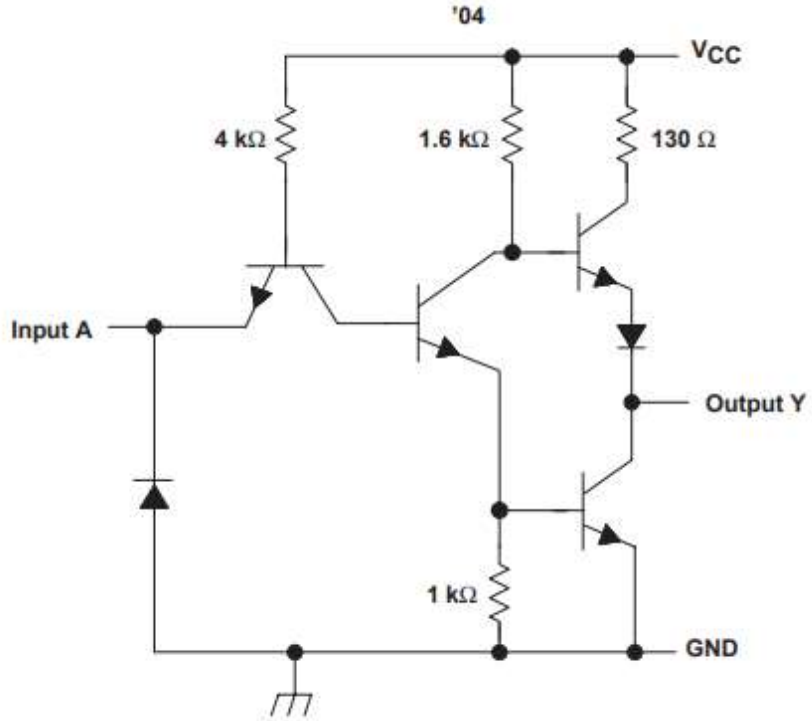
- built-in primitives actually modules instantiated inside modules
- similarly we can instantiate modules defined inside another module
- This is called as **hierarchical modeling**

```
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",
          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
```

Timing in Combinatorial Circuits

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



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

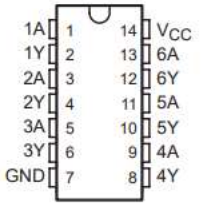
SDLS029C – DECEMBER 1983 – REVISED JANUARY 2004

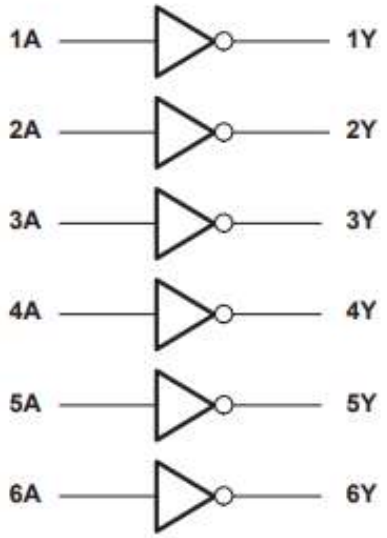
- Dependable Texas Instruments Quality and Reliability

description/ordering information

These devices contain six independent inverters.

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)





$Y = \overline{A}$

Timing in Combinatorial Circuits

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

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

```
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
```

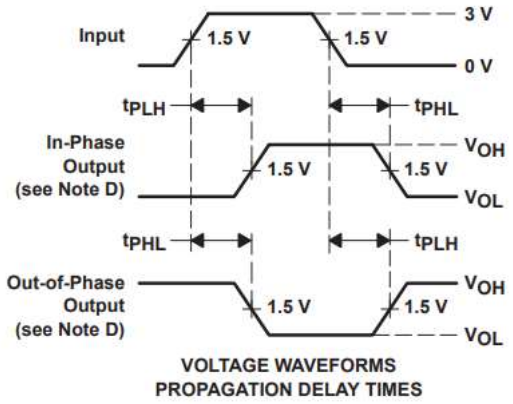
Timing in Combinatorial Circuits

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

- can we simulate delay ? YES #3ns delay

switching characteristics, V_{CC} = 5 V, T_A = 25°C (see Figure 1)

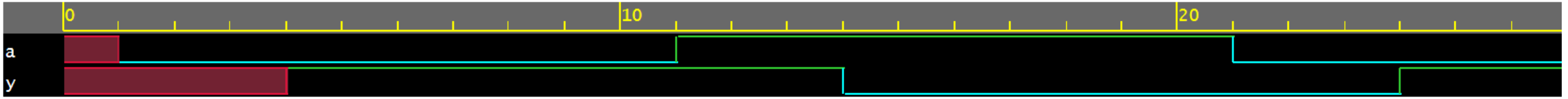
PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN54S04 SN74S04 MIN	TYP	MAX	UNIT
t_{PLH}	A	Y	R_L = 280 Ω, C_L = 15 pF		3	4.5	ns
t_{PHL}					3	5	



VOLTAGE WAVEFORMS
PROPAGATION DELAY TIMES

```
`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
```



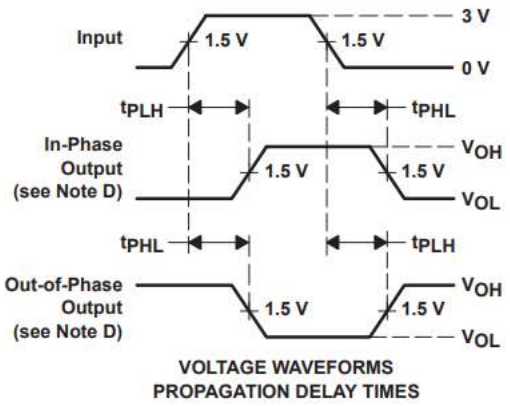
Timing in Combinatorial Circuits

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

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

switching characteristics, V_{CC} = 5 V, T_A = 25°C (see Figure 1)

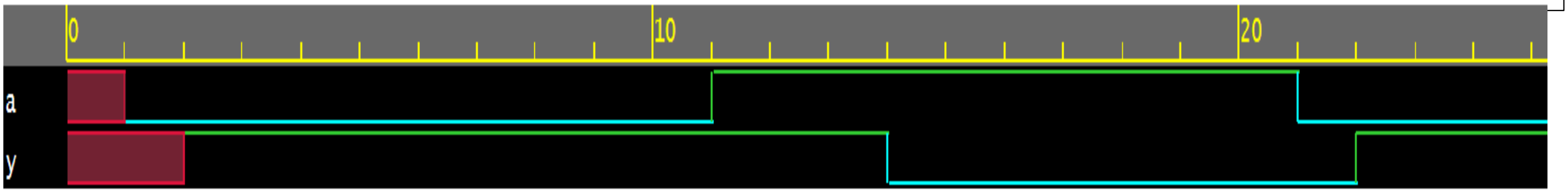
PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN54S04 SN74S04			UNIT
				MIN	TYP	MAX	
t_{PLH}	A	Y	R_L = 280 Ω, C_L = 15 pF		3	4.5	ns
t_{PHL}					3	5	



VOLTAGE WAVEFORMS
PROPAGATION DELAY TIMES

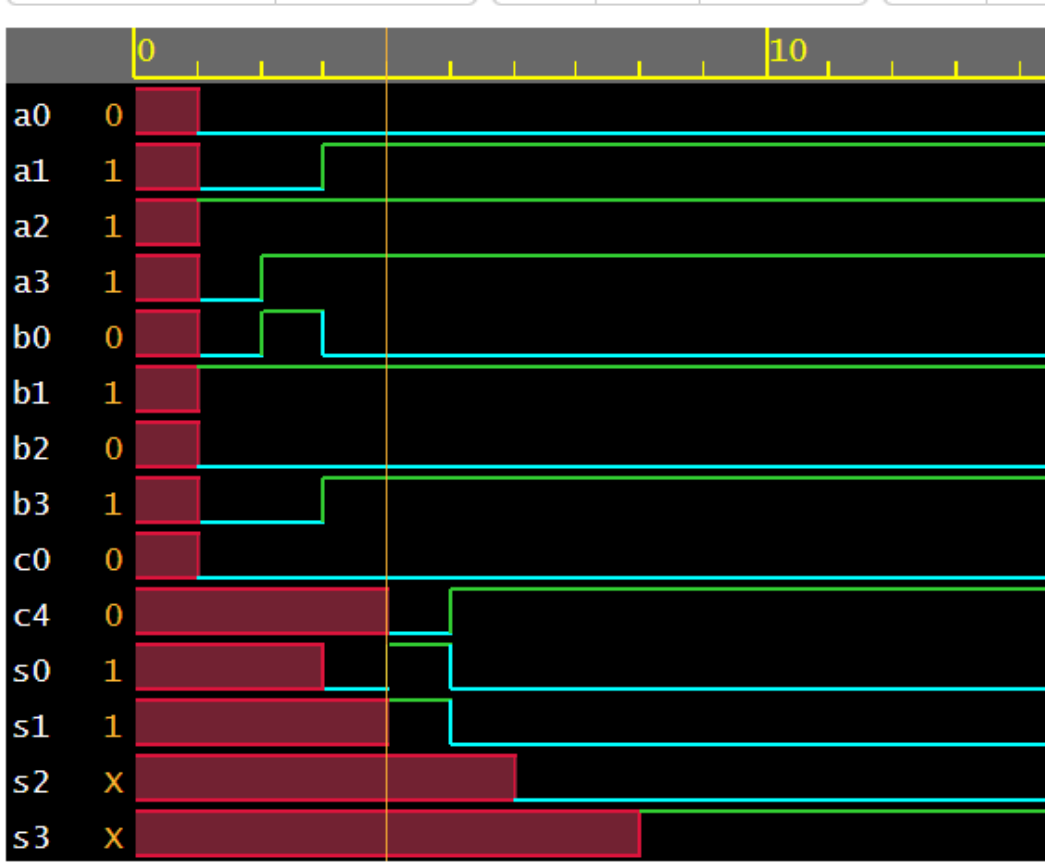
```
`timescale 1 ns / 1 ns
module inverter(
  output Y,
  input A
);
  not #(1,3) G0 (Y,A);  //1ns rise&3ns fall delay
endmodule
```

```
`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
```



Hierarchical Modeling: 4-bit Adder (delay)

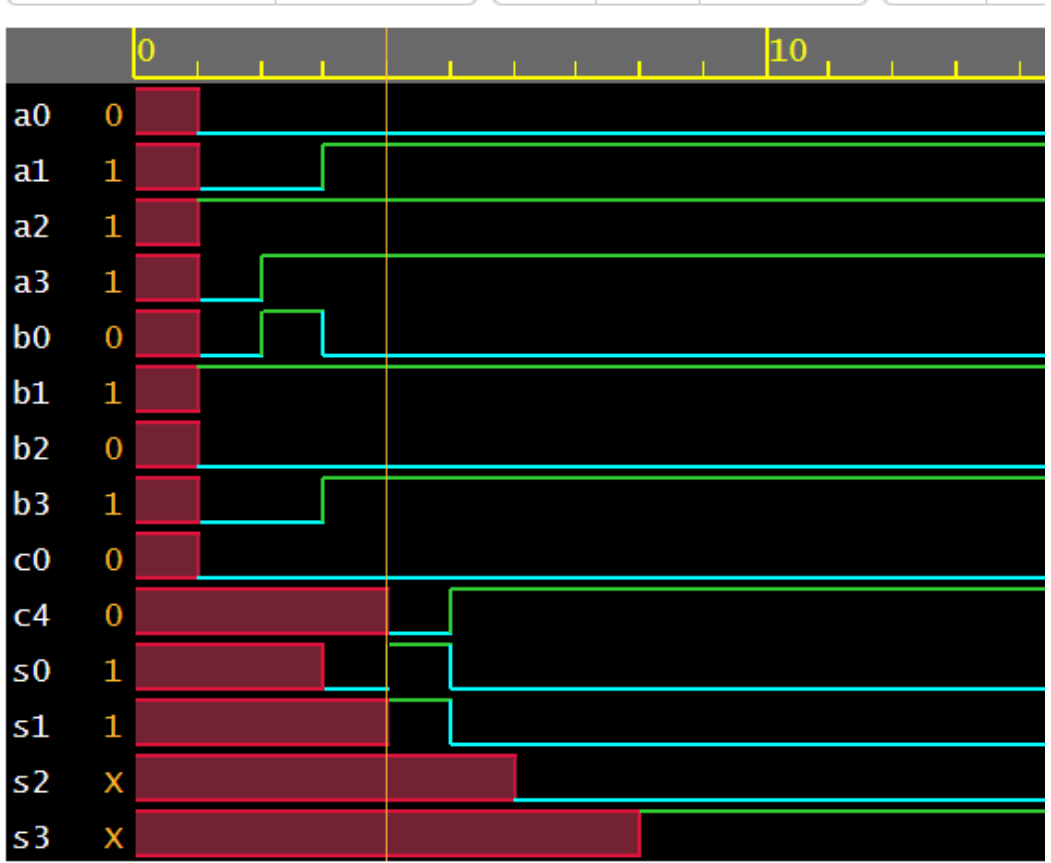
- assume that each gate has 1ns delay
- delay of S3 becomes 8ns
- S0 changes value and becomes stable 0100000

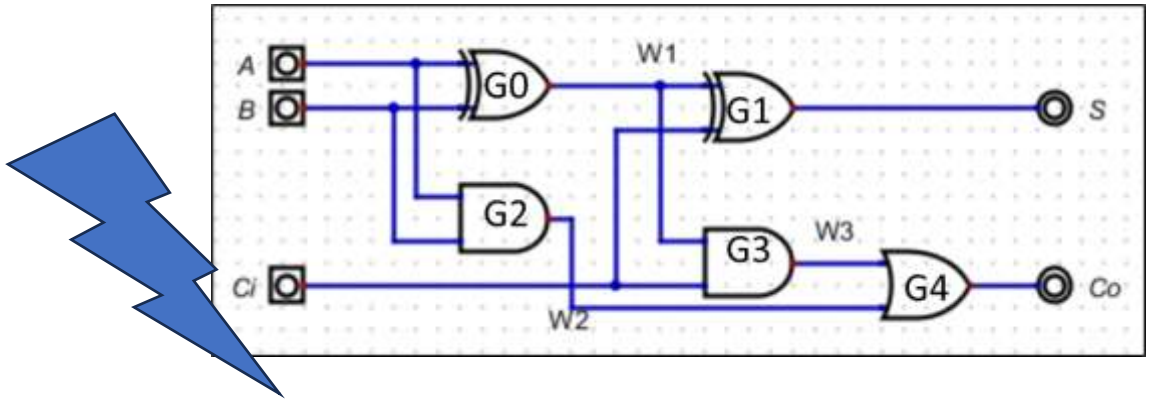



```
//full adder
`timescale 1 ns / 1 ns
module fourbitadder(
  output S0,S1,S2,S3,C4,
  input A0,A1,A2,A3,B0,B1,B2,B3,C0);
  wire C1,C2,C3;
  fulladder M0 (.S(S0),.Co(C1),.A(A0),.B(B0),.Ci(C0));
  fulladder M1 (.S(S1),.Co(C2),.A(A1),.B(B1),.Ci(C1));
  fulladder M2 (.S(S2),.Co(C3),.A(A2),.B(B2),.Ci(C2));
  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 0100000





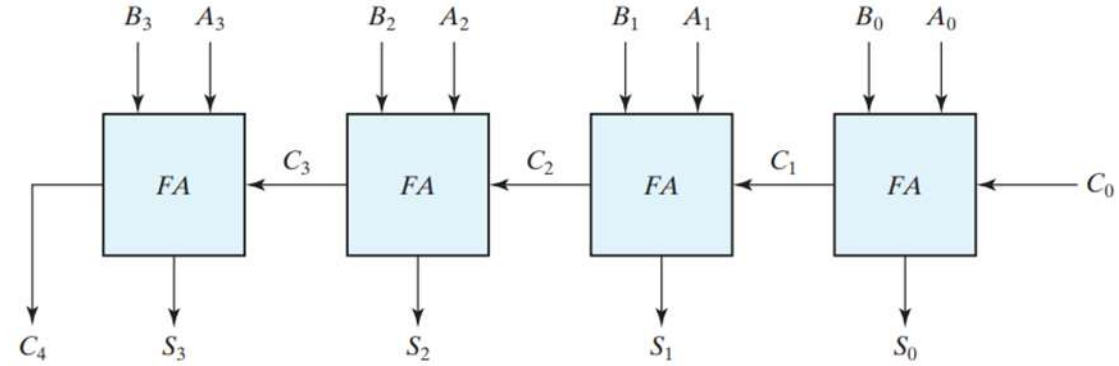
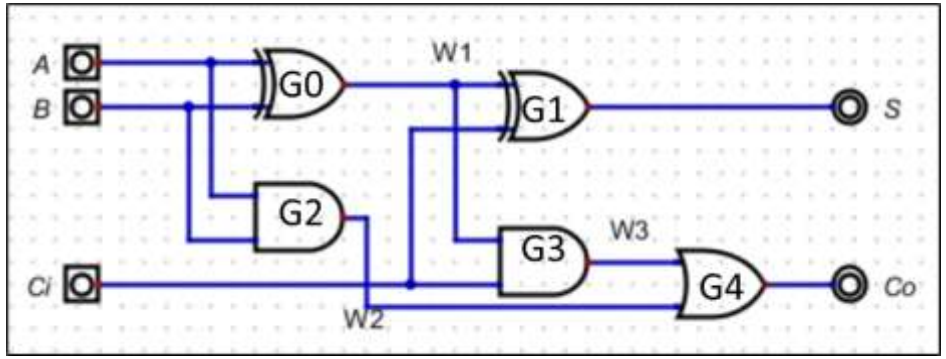
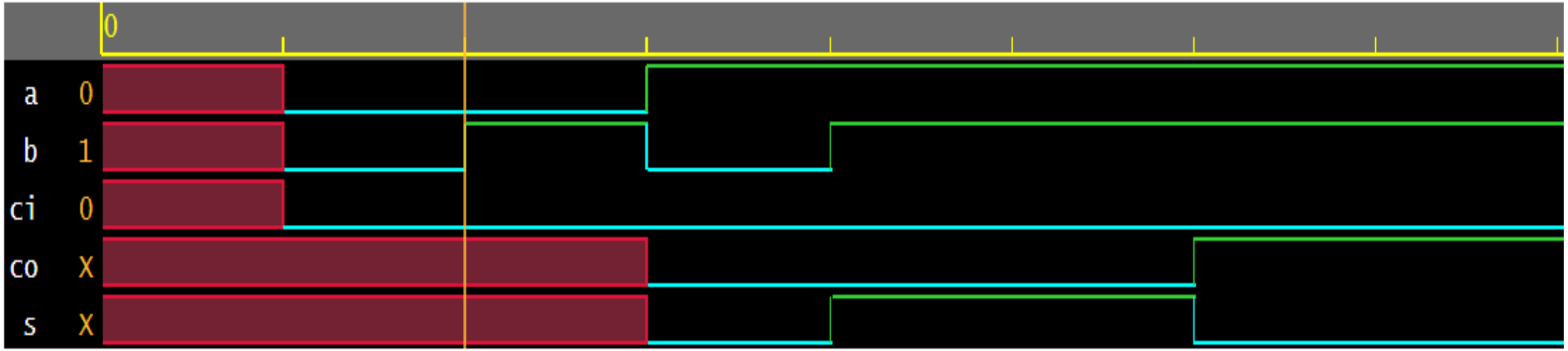


FIGURE 4.9
Four-bit adder

Full Adder (delay)

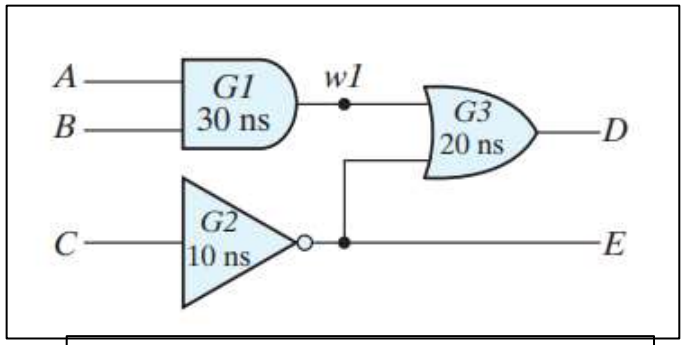
- assume that each gate has 1ns delay





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
```

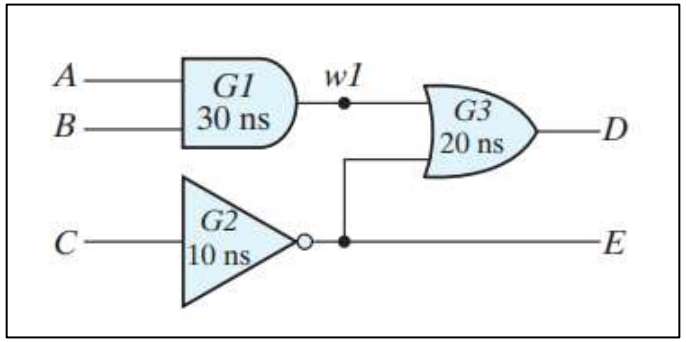
```
time A B C : D E
0 0 0 0 : x x
10 0 0 0 : x 1
30 0 0 0 : 1 1
100 1 1 1 : 1 1
110 1 1 1 : 1 0
130 1 1 1 : 0 0
150 1 1 1 : 1 0
```

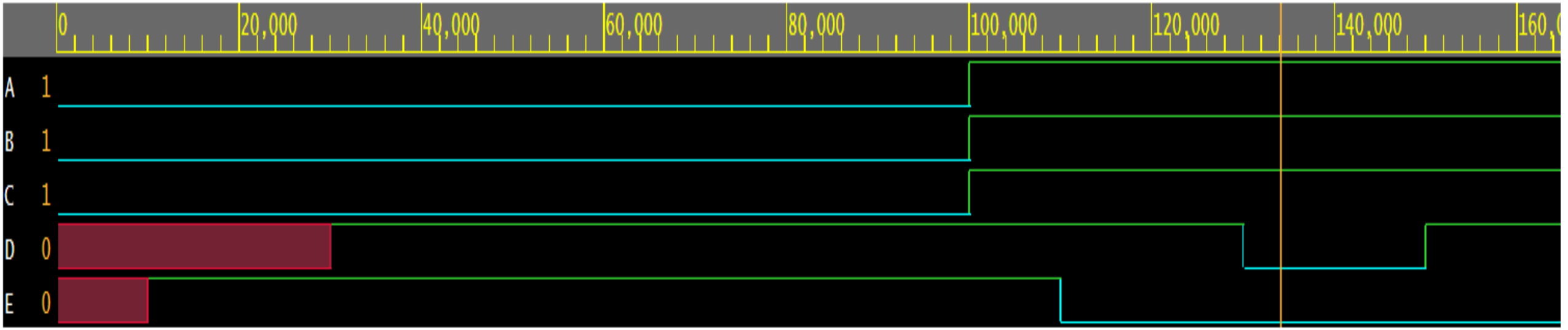
```
//testbench
`timescale 1 ns / 1 ns
module testbench ();
reg A,B,C;
wire D,E;
  andor M0 (.A(A),.B(B),.C(C),.D(D),.E(E));
initial begin
  $dumpfile("dump.vcd");
  $dumpvars;
  $display("\t\ttime A B C : D E");
  $monitor("%d %b %b %b : %b %b",$time,A,B,C,D,E);
  A=0; B=0; C=0;
  #100 A=1; B=1; C=1;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

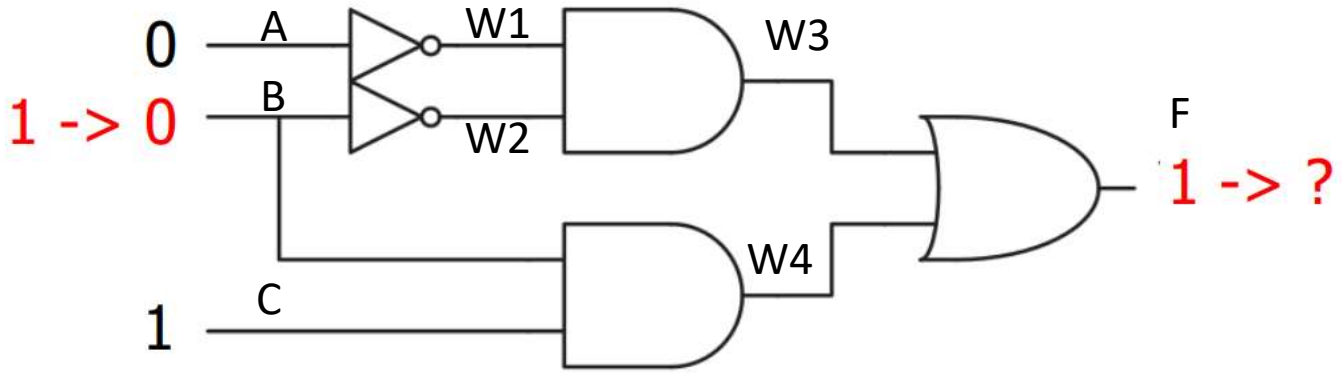
time A B C : D E
0 0 0 0 : x x
10 0 0 0 : x 1
30 0 0 0 : 1 1
100 1 1 1 : 1 1
110 1 1 1 : 1 0
130 1 1 1 : 0 0
150 1 1 1 : 1 0





Output Glitches

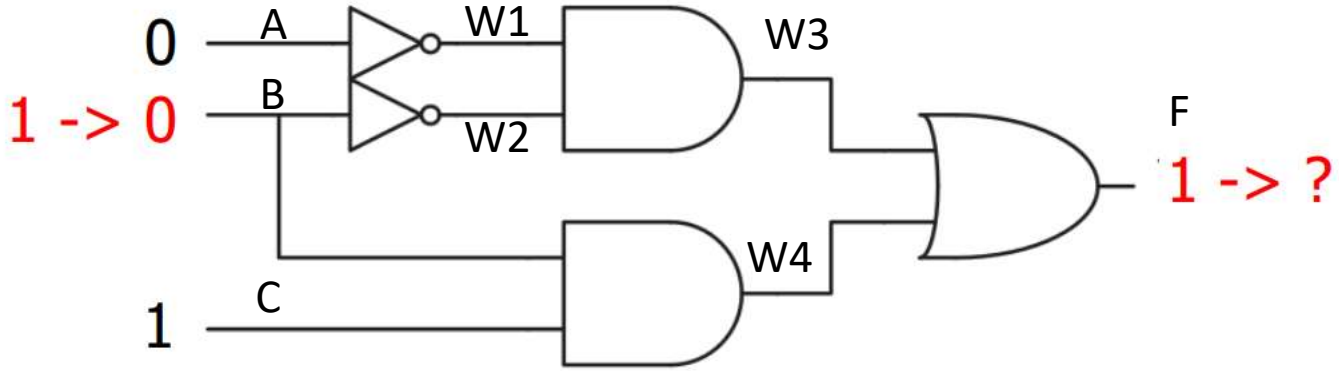
*Onur Mutlu lecture notes

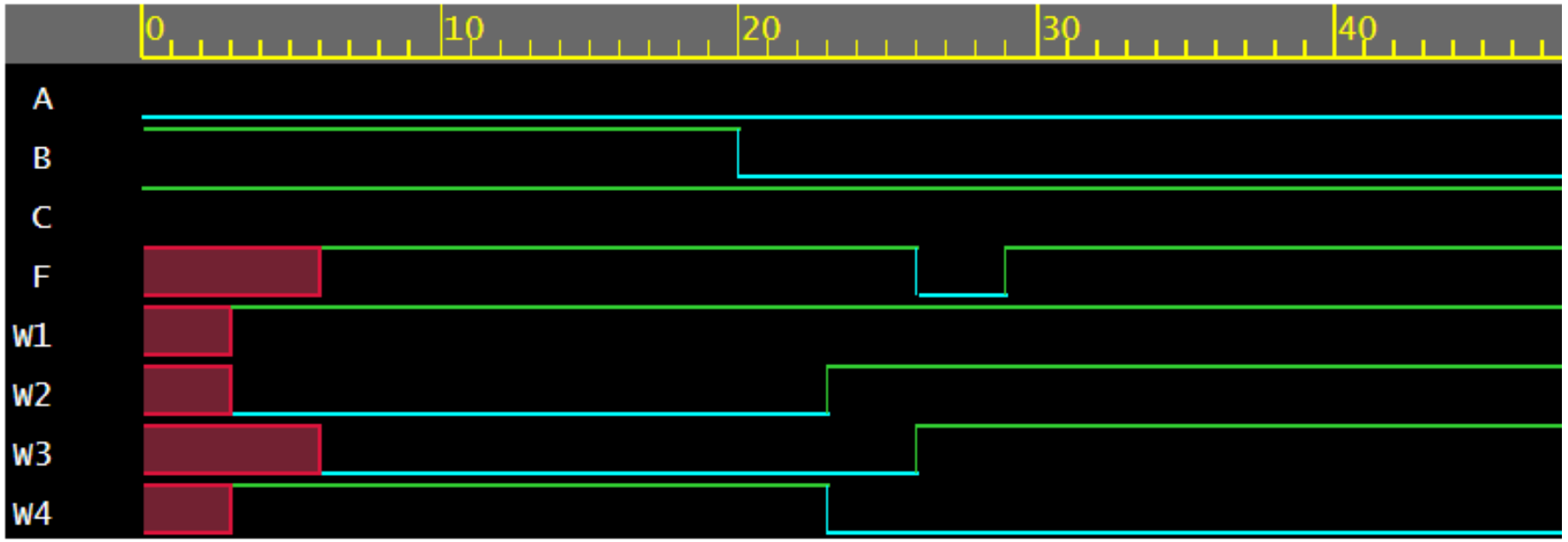



```
`timescale 1 ns / 1 ns
module glitch(
  output F,
  input A,B,C
);
  wire W1,W2,W3,W4;
  not #3 G0 (W1,A);
  not #3 G1 (W2,B);
  and #3 G2 (W3,W1,W2);
  and #3 G3 (W4,B,C);
  or  #3 G4 (F,W3,W4);
endmodule
```

Output Glitches

*Onur Mutlu lecture notes

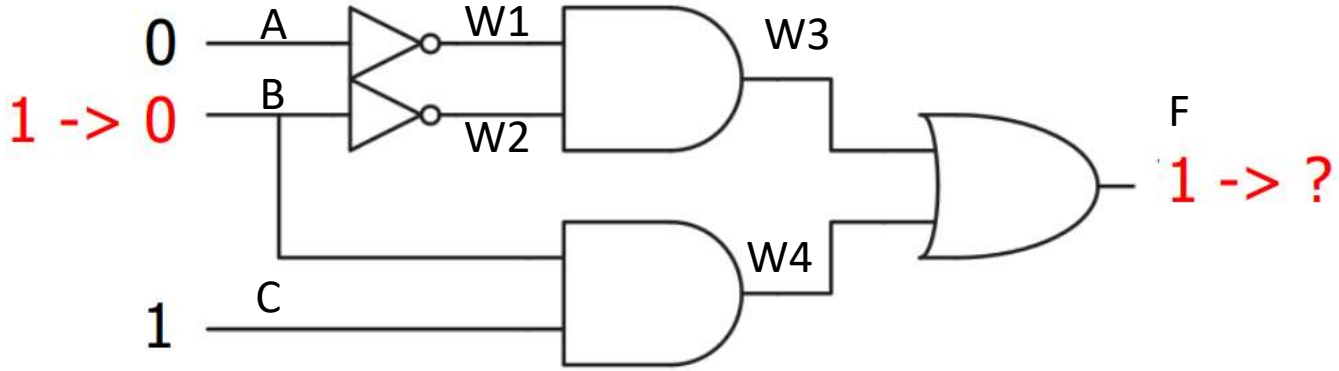




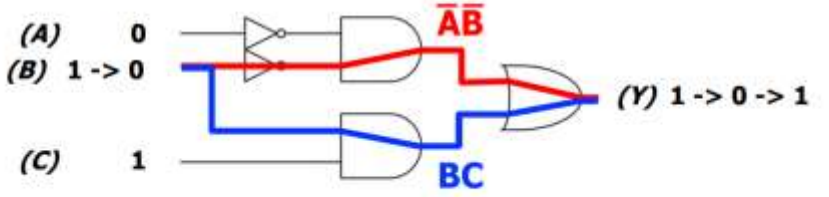

```
`timescale 1 ns / 1 ns
module testbench;
  reg  a,b,c;  //reg for inputs
  wire f;    //wire for outputs
  //instantiation
  glitch G0 (.F(f),.A(a),.B(b),.C(c));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("a,b,c:f");
    $monitor("%b%b%b:%b",a,b,c,f);
       a=0; b=1; c=1;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

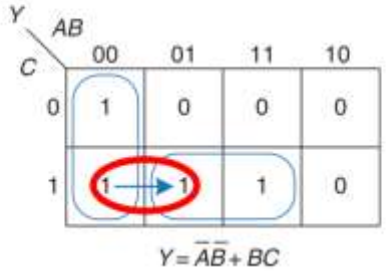
*Onur Mutlu lecture notes

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**

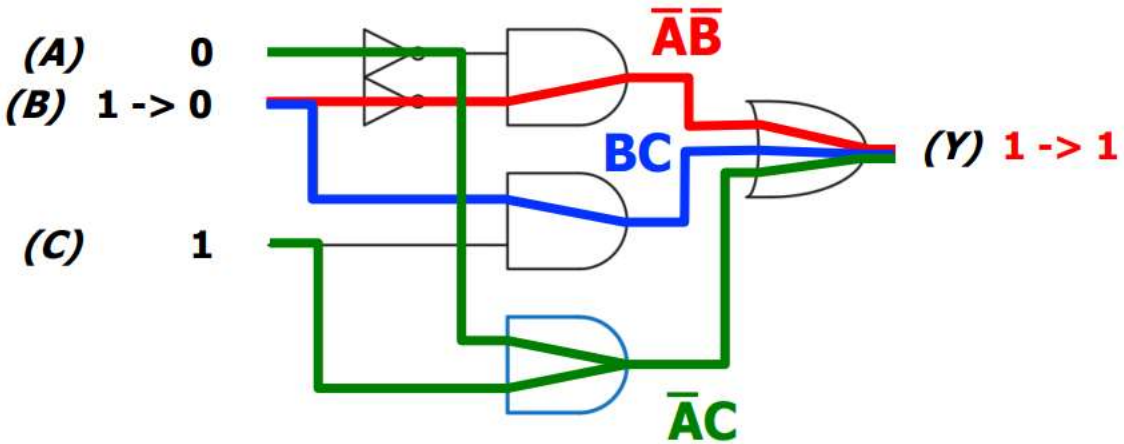




*Onur Mutlu lecture notes

Output Glitches

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



Y, AB \ C	00	01	11	10
0	1	0	0	0
1	1	1	1	0

$\bar{A}C$

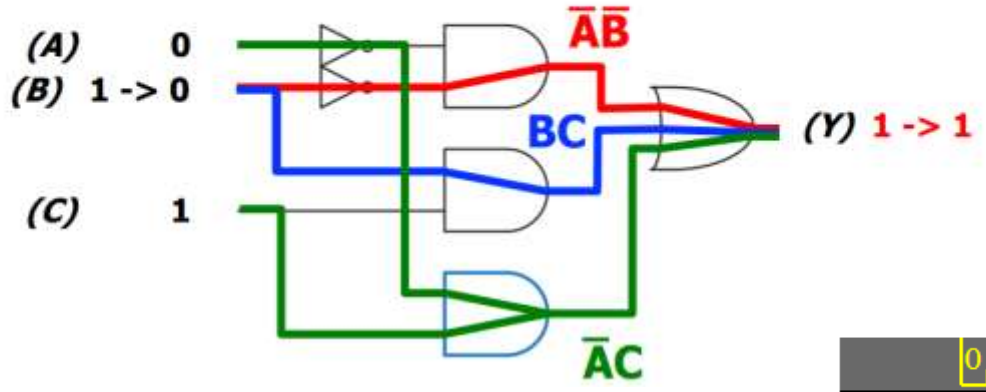
$Y = \bar{A}\bar{B} + BC + \bar{A}C$

No dependence on B
=> No glitch!

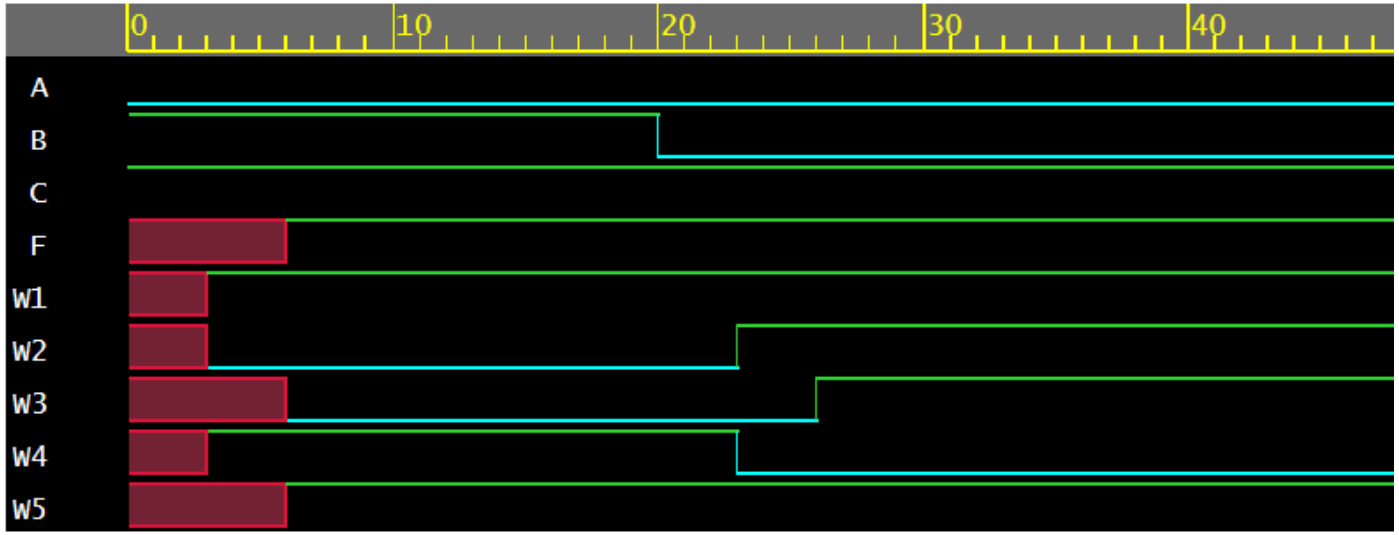
Output Glitches

*Onur Mutlu lecture notes

- 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

*Onur Mutlu lecture notes

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:

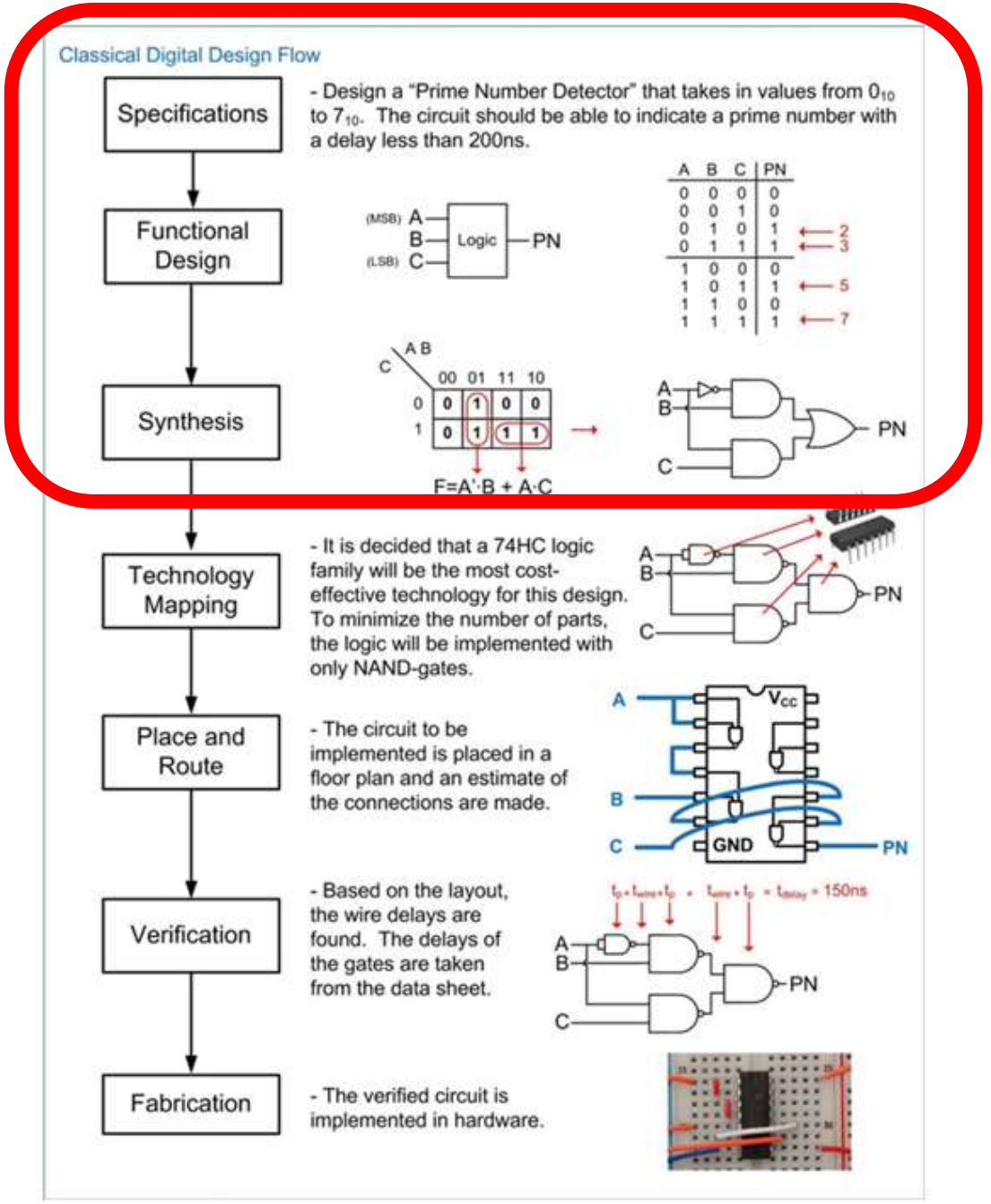
High-Level and Logic Design: Create the architecture. Specify and implement the required system behavior.

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

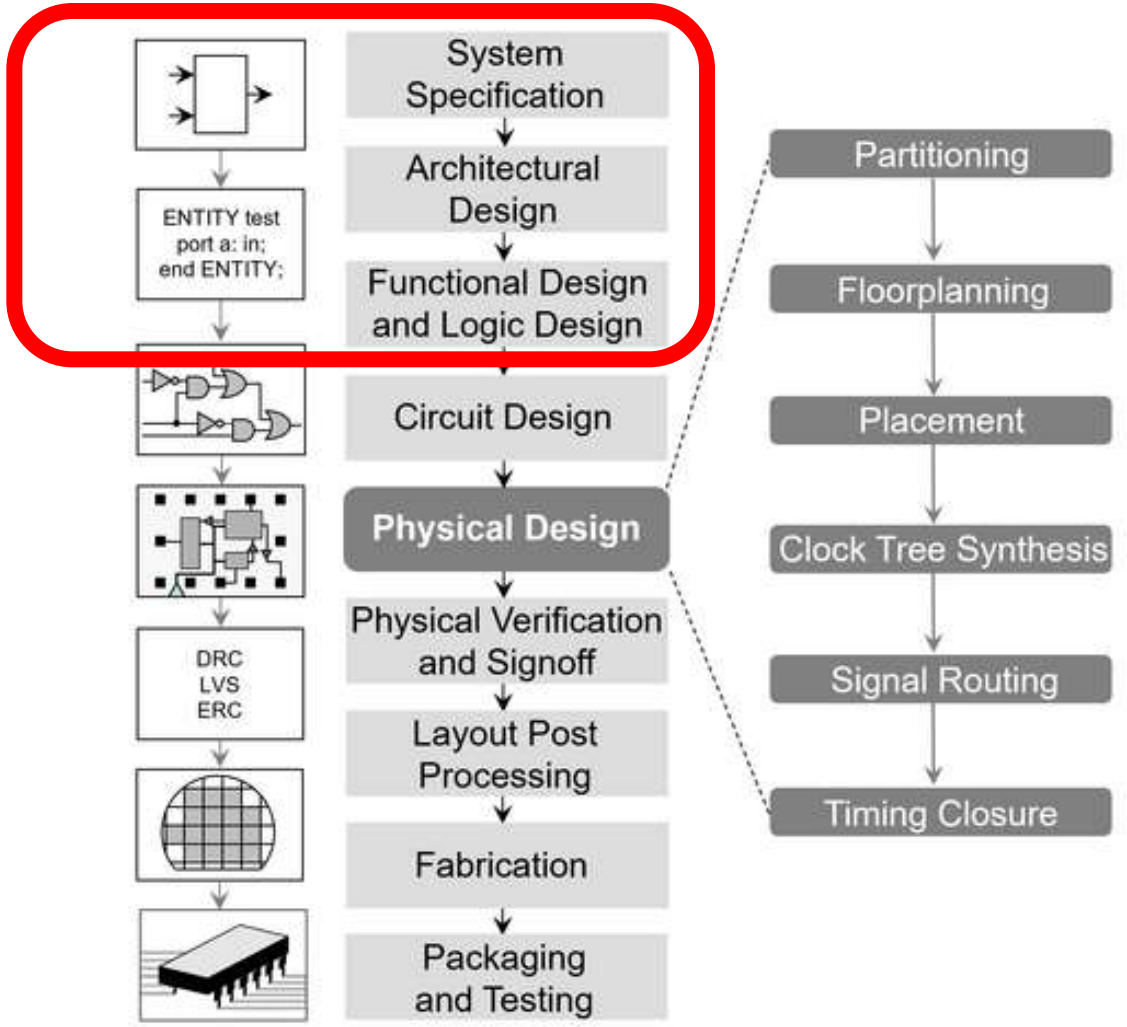
Logic Synthesis: Convert the high-level description into a gate-level logical netlist.

Physical Implementation: Create a physical layout suitable for fabrication.

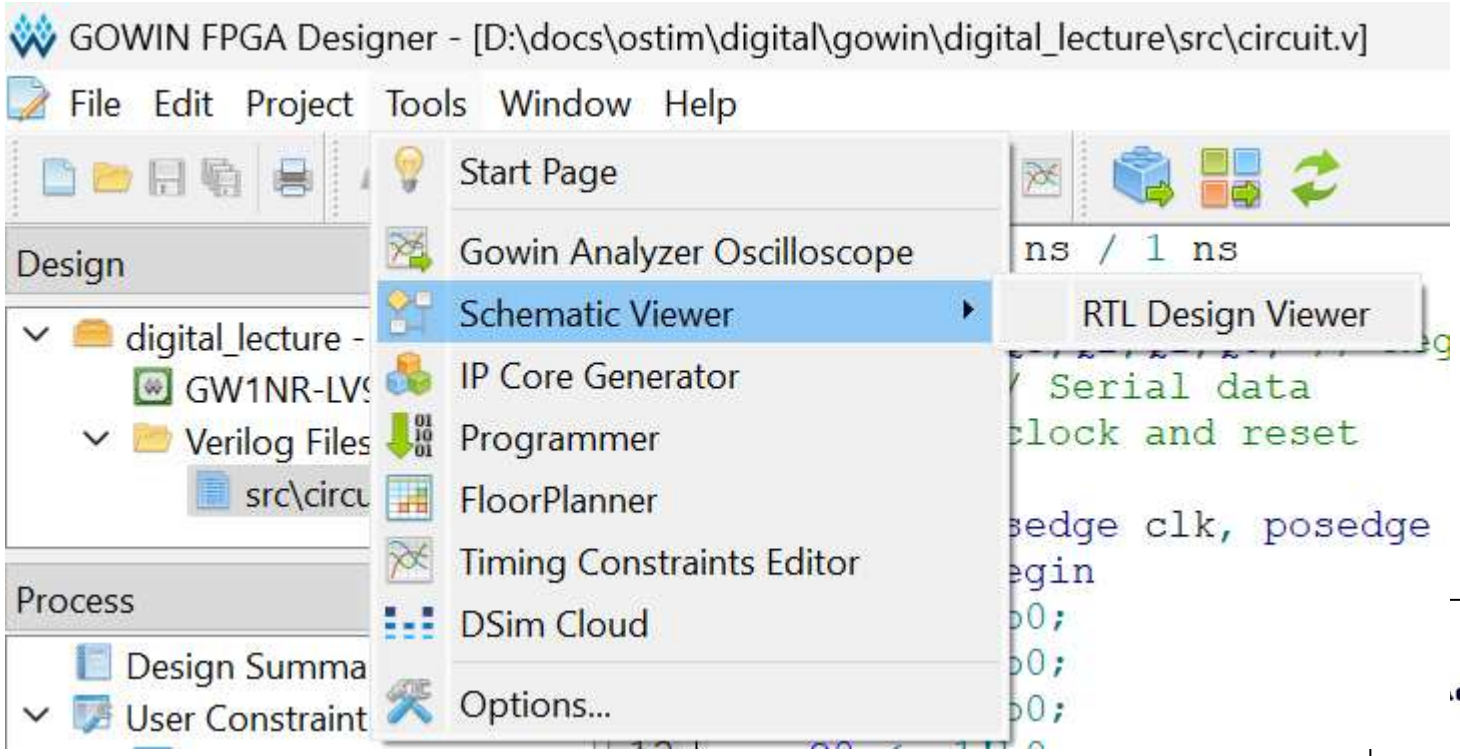
Physical Chip Design

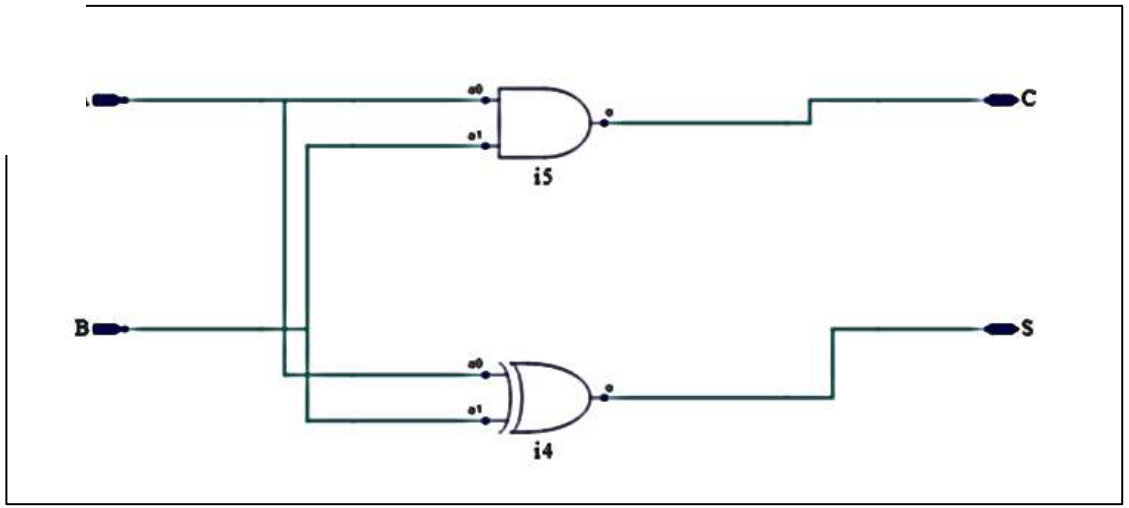


Physical Chip Design



Gowin Synthesis

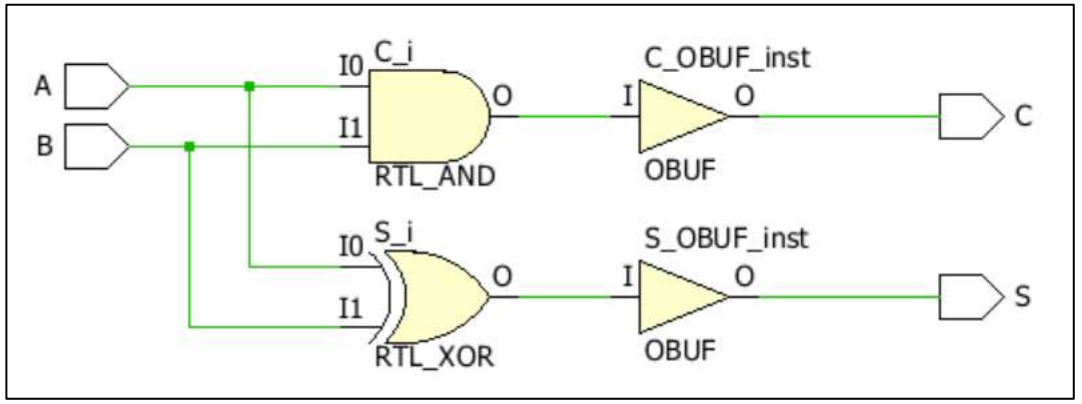




Vivado Synthesis

Vivado quick start:

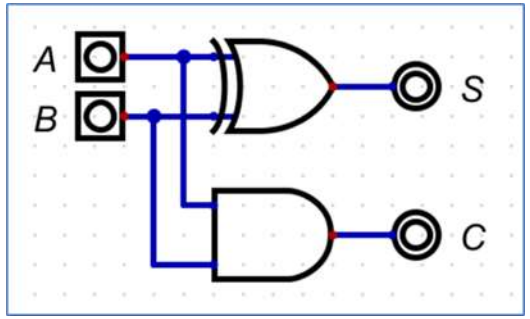
- New Project
- RTL Project
- Default Part: xc7a50ticsg325 Artix7 Spartan
- Next-Next
- Design Sources – Right Click-Add Sources
- Add Design Sources
- Create File – halfadder.v
- Define Module- OK
- double Click halfadder.v and write the code or copy/paste from EDAplayground
- Flow-Open Elaborated Design-OK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

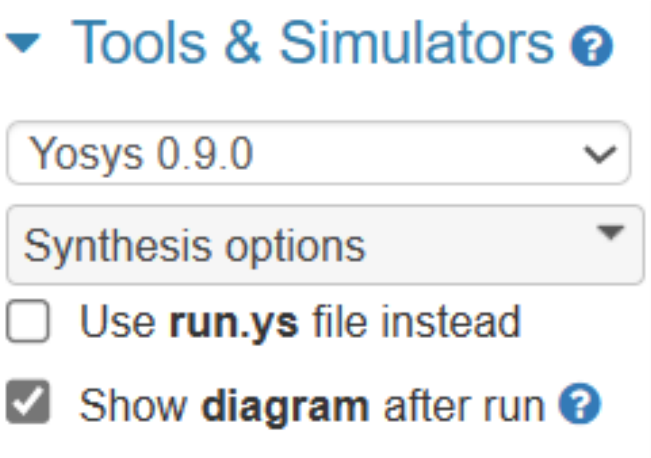


[Basys 3 - Digilent Reference]

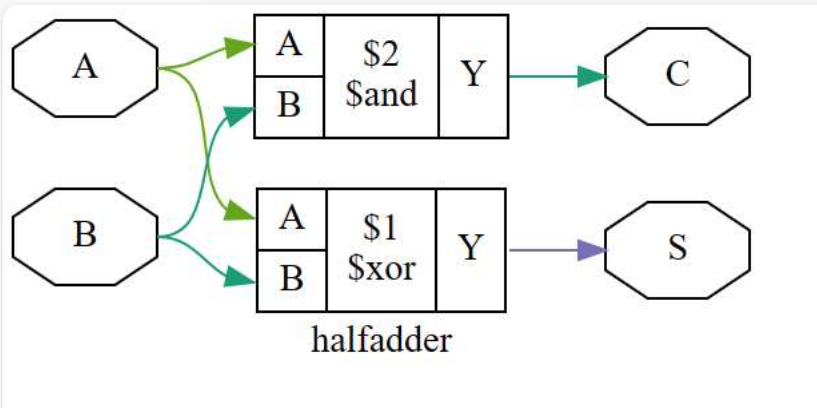
Verilog for Synthesis: Yosys

- 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**





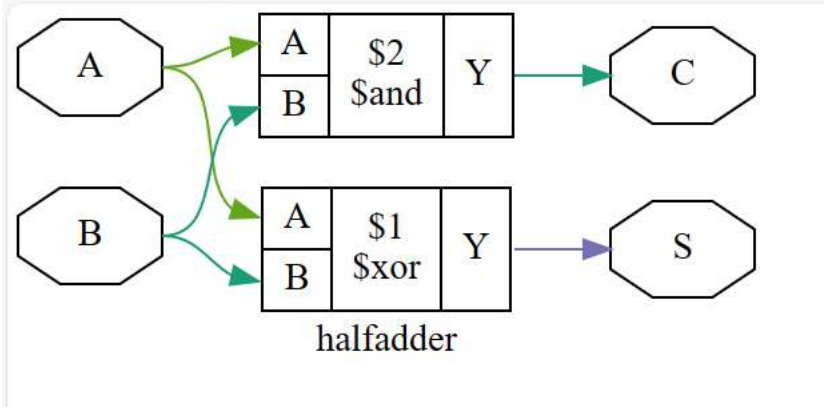

```
module halfadder (
  output C,S,
  input A,B);

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



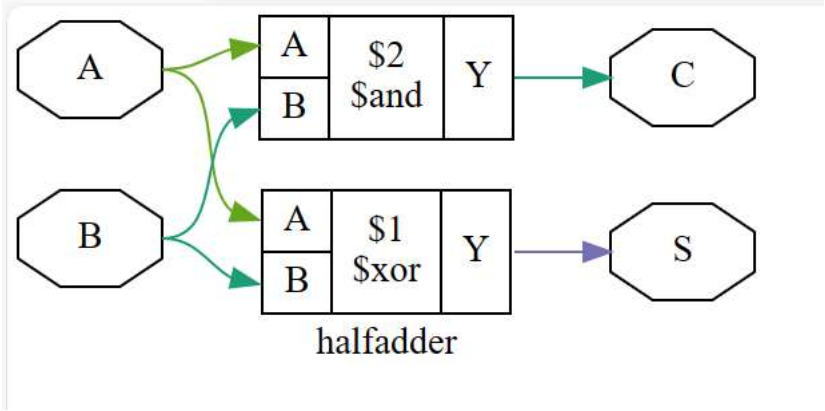
Verilog for Synthesis: Yosys

- 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**

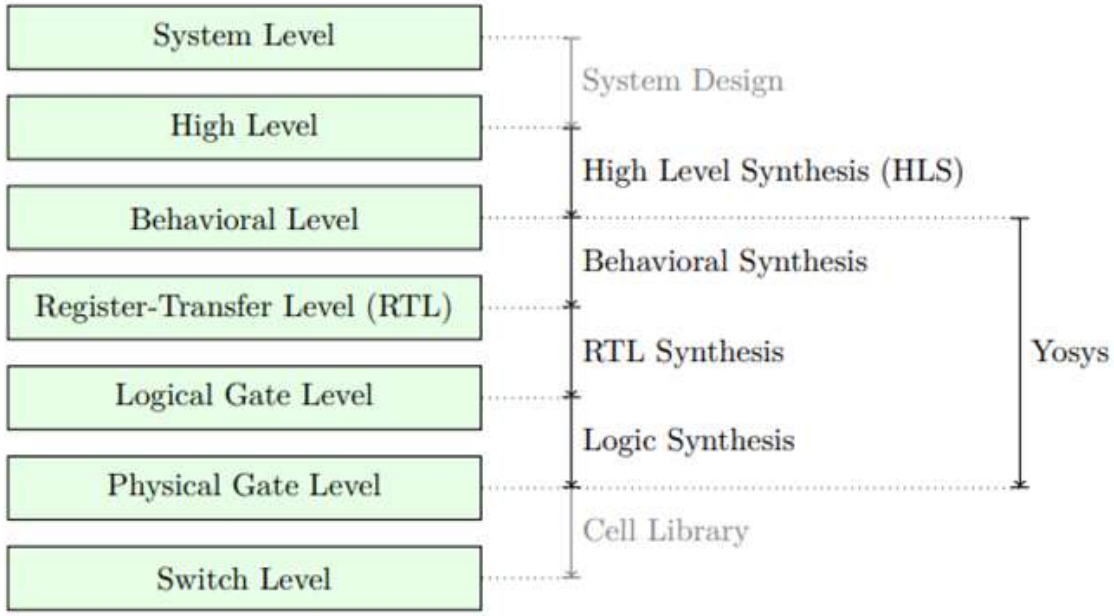


Figure 2.1: Different levels of abstraction and synthesis.

Dataflow Modeling : Continous 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

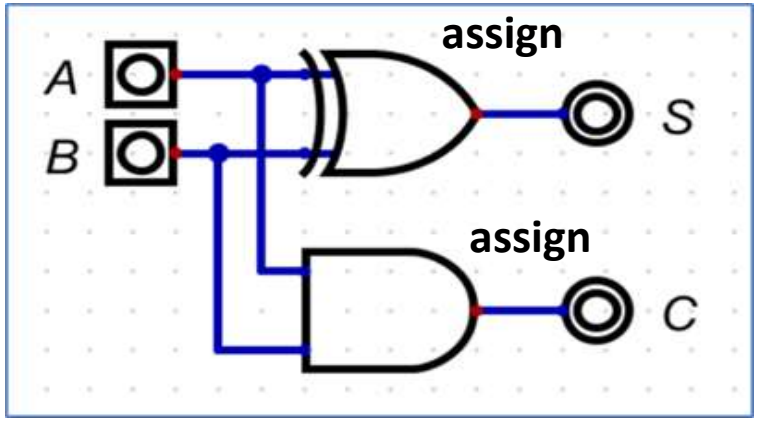
- Continuous Assignments useful for behavioral description of combinatorial logic circuits
- Assign **wires** EXPRESSION to WIRE
- it is wire connection and immediately changes with the expression

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* ;

```
//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
```



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

Nets are used 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

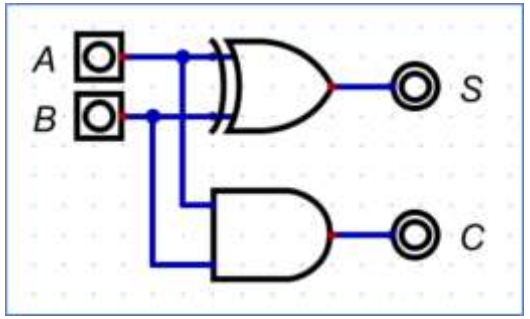
```
module halfadder (
  output C,S,
  input A,B);

  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: 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.

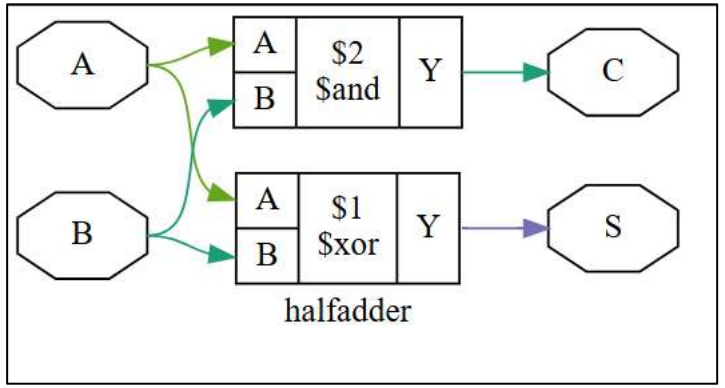



```
module halfadder (
  output C,S,
  input A,B);

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

Bitwise Operators		
~	~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
~^ *or* ^~	m ~^ 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

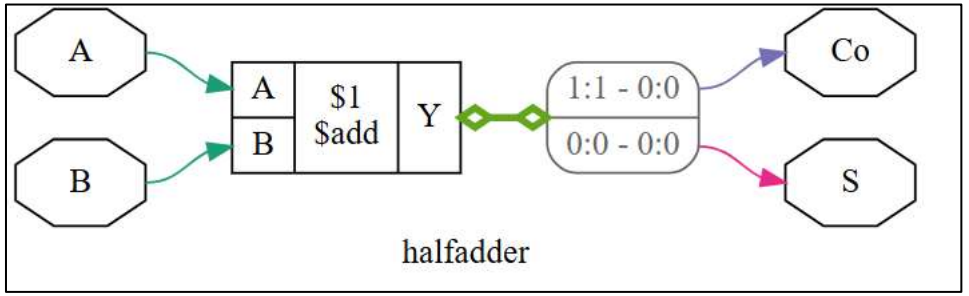
- We can use arithmetic operators also to define half adder
- Notice that synthesizer generated the half adder circuit based on our modeling

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

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

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

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

- 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

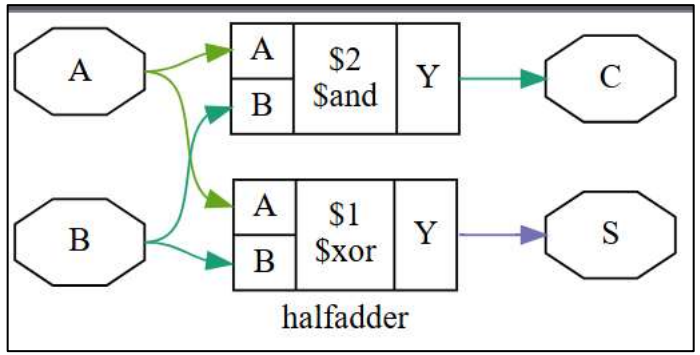


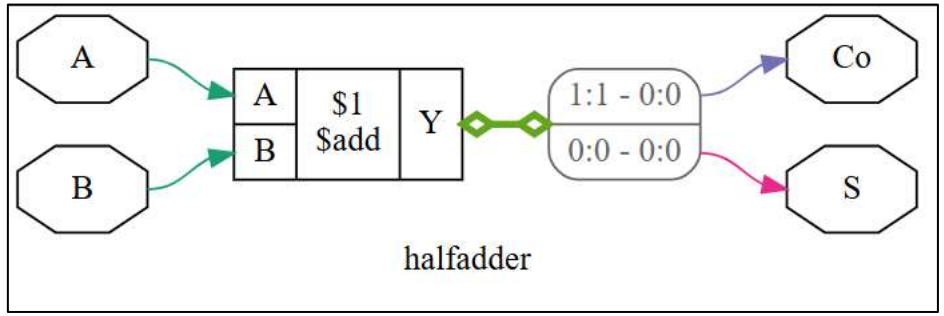
Dataflow Modeling: Arithmetic Operators

- 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

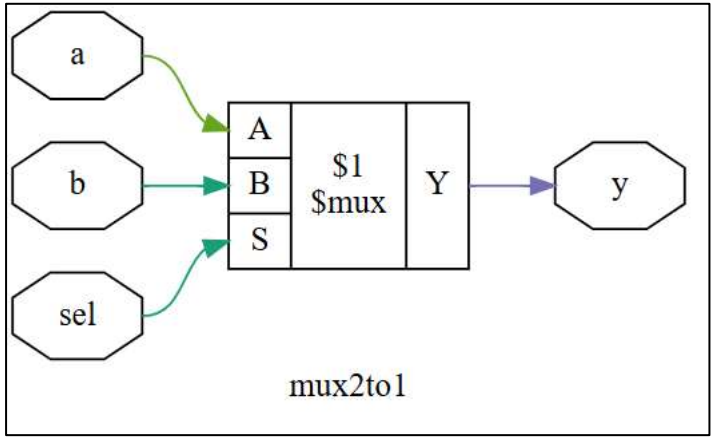
Dataflow Modeling : MUX with conditional

- 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
```

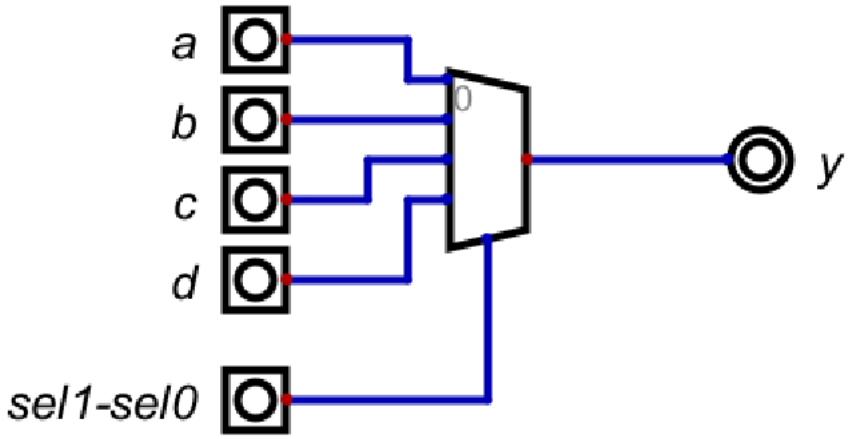
```
a b:sel y
0 1:1 1
0 1:0 0
```

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



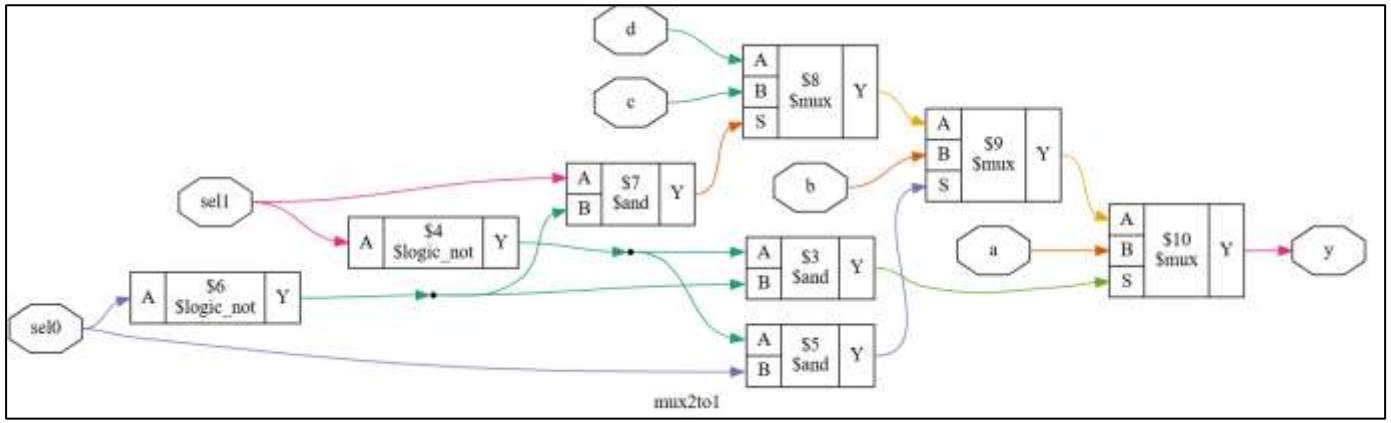
Dataflow Modeling : MUX4to1 with Conditional

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




```
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
```

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



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

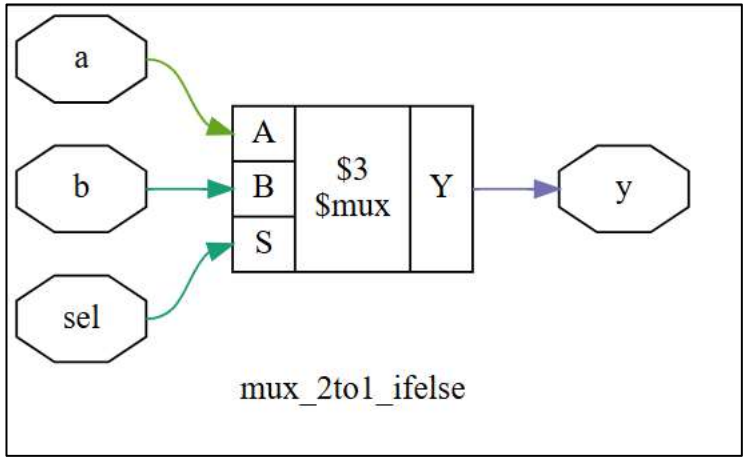
```
module mux_2to1_ifelse (
   input wire a,        // Input 0
   input wire b,        // Input 1
   input wire sel,      // Select line
   output reg y         // Output
);

   always @* begin
      if (sel)
         y = b;
      else
         y = a;
   end

endmodule
```

`if ( expression ) statement or statement_group`

`else` *statement or statement_group*

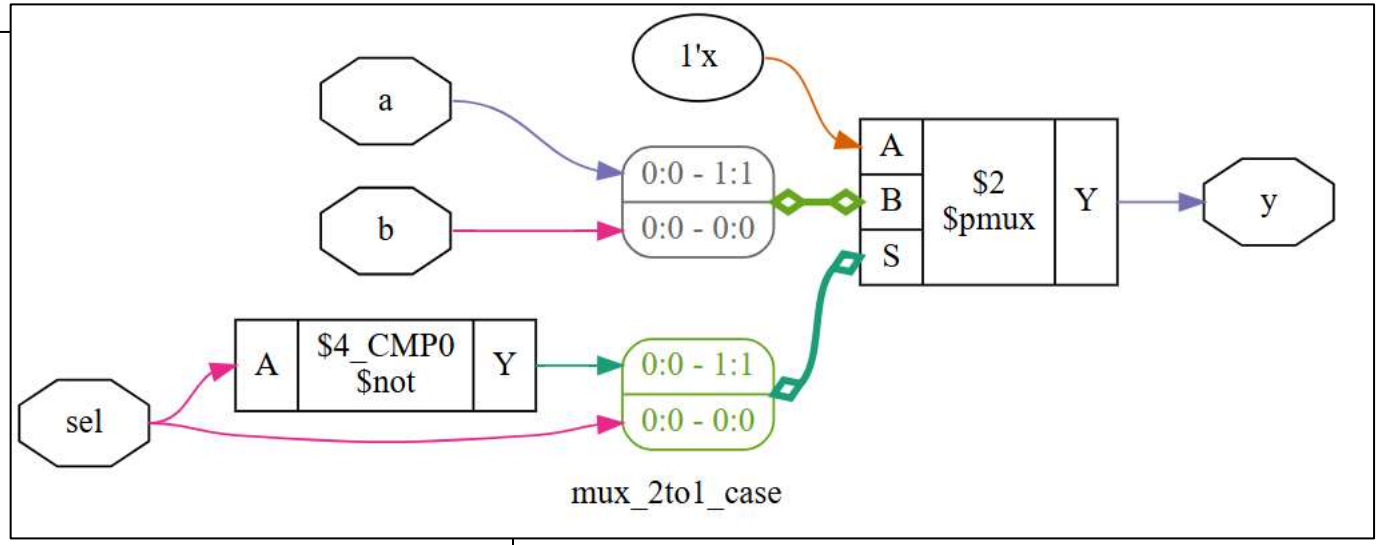
Executes the first statement or statement group if the expression evaluates as true. Executes the second statement or statement group if the expression evaluates as false or unknown.

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



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
```



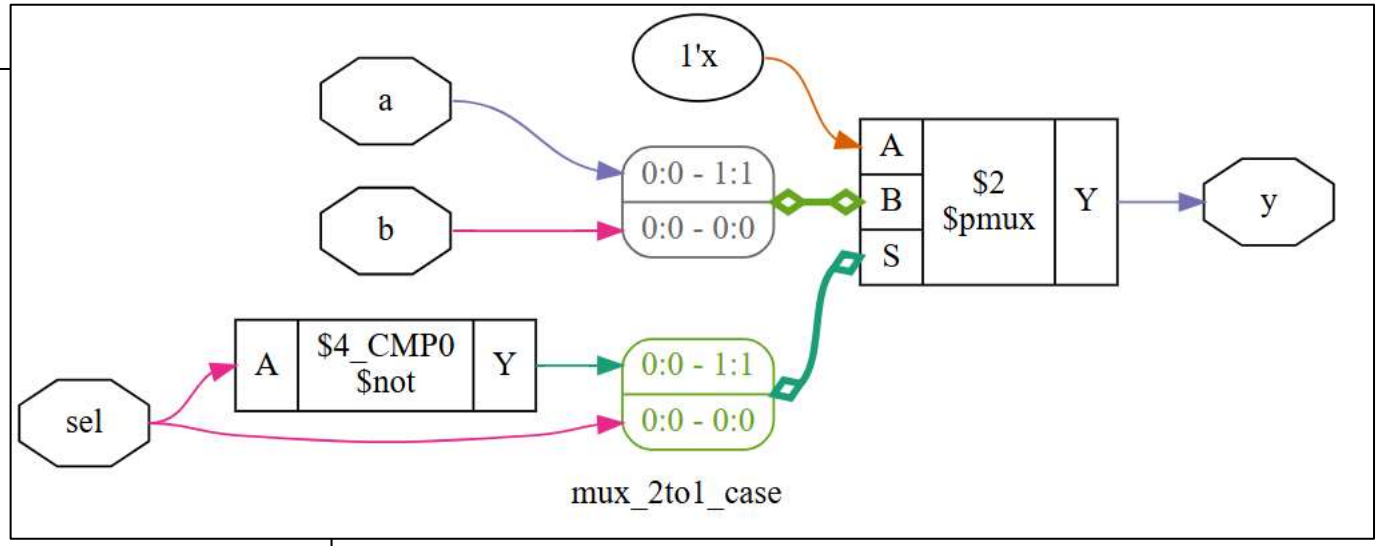
`case` (*expression*)
- *case_item*: *statement or statement_group*
- *case_item*, *case_item*: *statement or statement_group*
- `default`: *statement or statement_group*

`endcase`

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).

Binary Numbers in Verilog

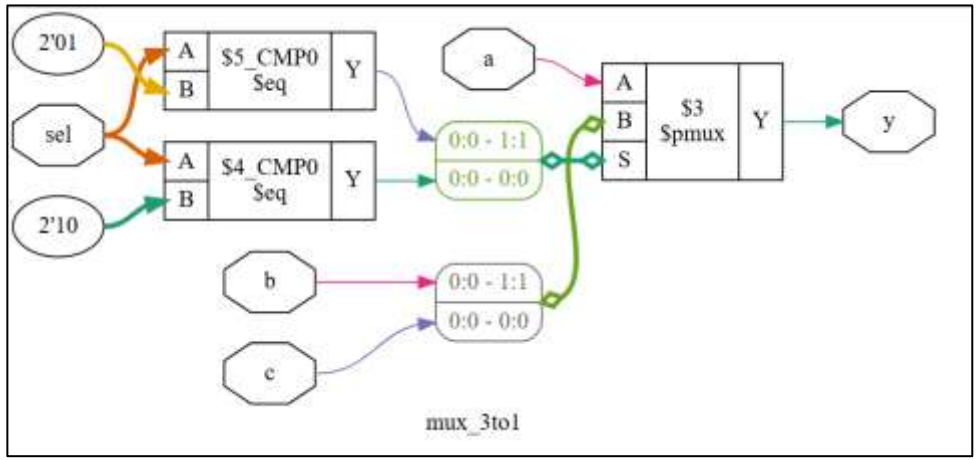
- If data is given as just names it means 1-bit
- multiple bit binary numbers can be represented like 1’b0 4’b0101 8’b01010101
- array representation : 4bit input A [3:0]

Port Range -Vector

5.2 Port Declarations

Combined Declarations *(added in Verilog-2001)*
port_direction data_type **`signed`** *range port_name , port_name , ... ;*
Old Style Declarations
port_direction **`signed`** *range port_name , port_name , ... ;* *data_type_declarations* (see section 6.0)

- *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.

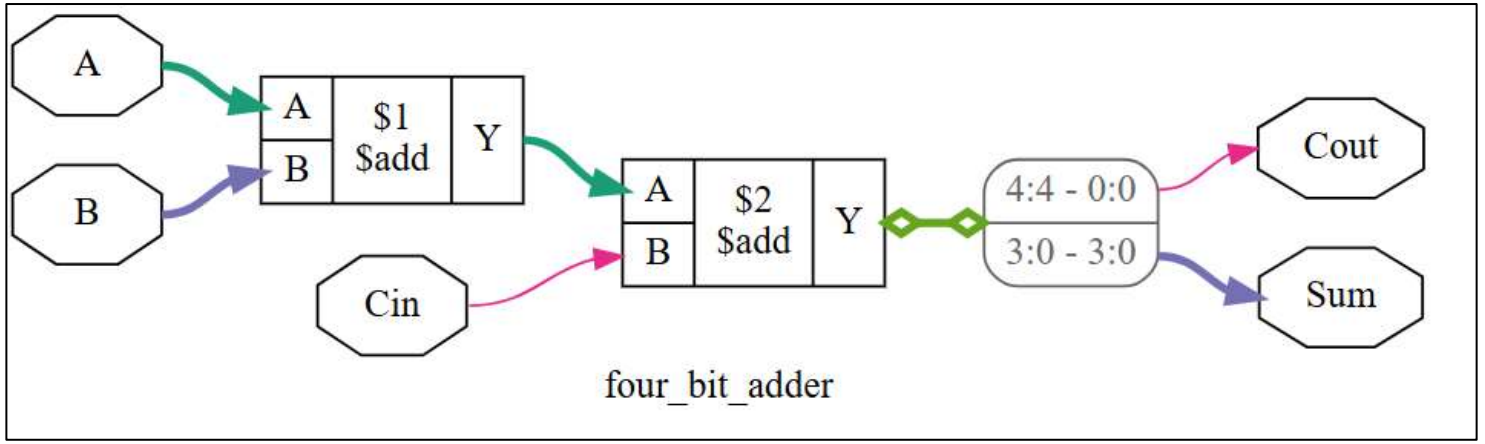



```
module mux_3to1 (
   input wire a,         // Input 0
   input wire b,         // Input 1
   input wire c,         // Input 2
   input wire [1:0] sel,  // 2-bit Select line
   output reg y          // Output
);

   always @* begin
      case (sel)
         2'b00: y = a;   // 00, output y is assigned input a
         2'b01: y = b;   // 01, output y is assigned input b
         2'b10: y = c;   // 10, output y is assigned input c
         default: y = a; // Default case
      endcase
   end

endmodule
```

4bit adder

```
module four_bit_adder(
    input [3:0] A, B,
    input Cin,
    output [3:0] Sum,
    output Cout
);
  assign {Cout, Sum} = A + B + Cin;
endmodule
```

NOTICE two adders
A+B first and output added with Cin



4bit ALU

```
module four_bit_alu(
   input [3:0] A, B,
   input M,
   output [3:0] S,
   output C
);
  assign {C,S} = M ? (A - B): (A + B);
endmodule
```

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=12 B=13 M=0 S= 9 C=1
Time=3 A= 9 B= 2 M=1 S= 7 C=0
Time=4 A= 9 B=12 M=1 S=13 C=1

NOTICE that 4'd2, 4bit size DECIMAL 2

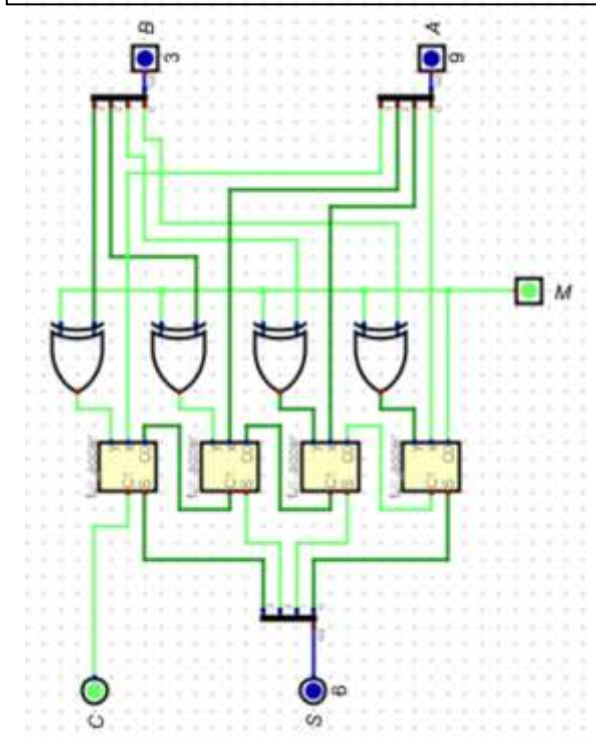
4.11 Literal Integer Numbers

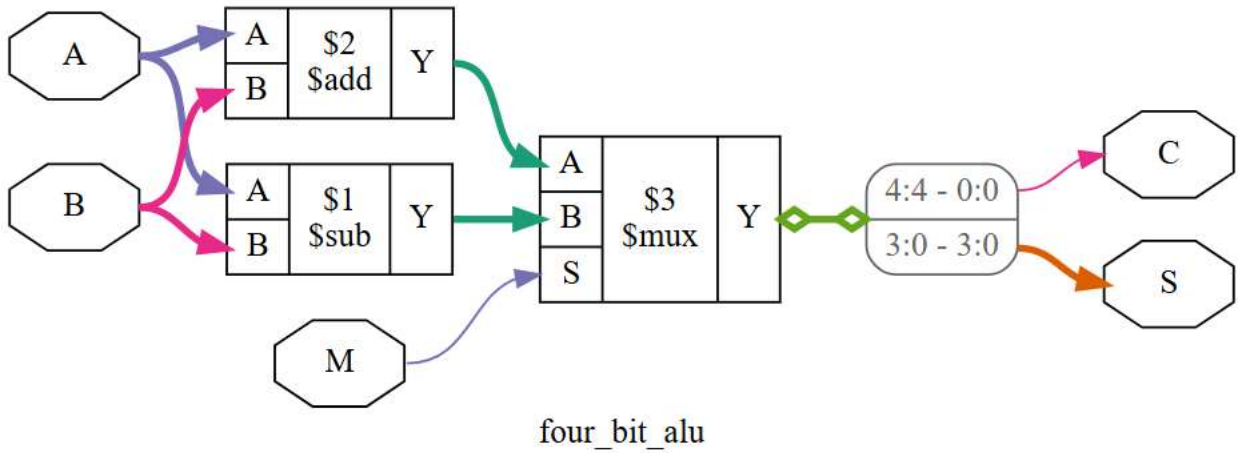
value	unsized decimal integer
size ' base value	sized integer in a specific radix (base)

```
module test_four_bit_alu_tb;
reg [3:0] A, B;
reg M;
wire [3:0] S;
wire C;
four_bit_alu uut(
   .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 =4'd2; B =4'd3; M = 0;
   #1; A =4'd12; B =4'd13; M = 0;
   #1; A = 4'd9; B = 4'd2; M = 1;
   #1; A = 4'd9; B = 4'd12; M = 1;

end
endmodule
```

4bit ALU

```
module four_bit_alu(
    input [3:0] A, B,
    input M,
    output [3:0] S,
    output C
);
  assign {C,S} = M ? (A - B): (A + B);
endmodule
```



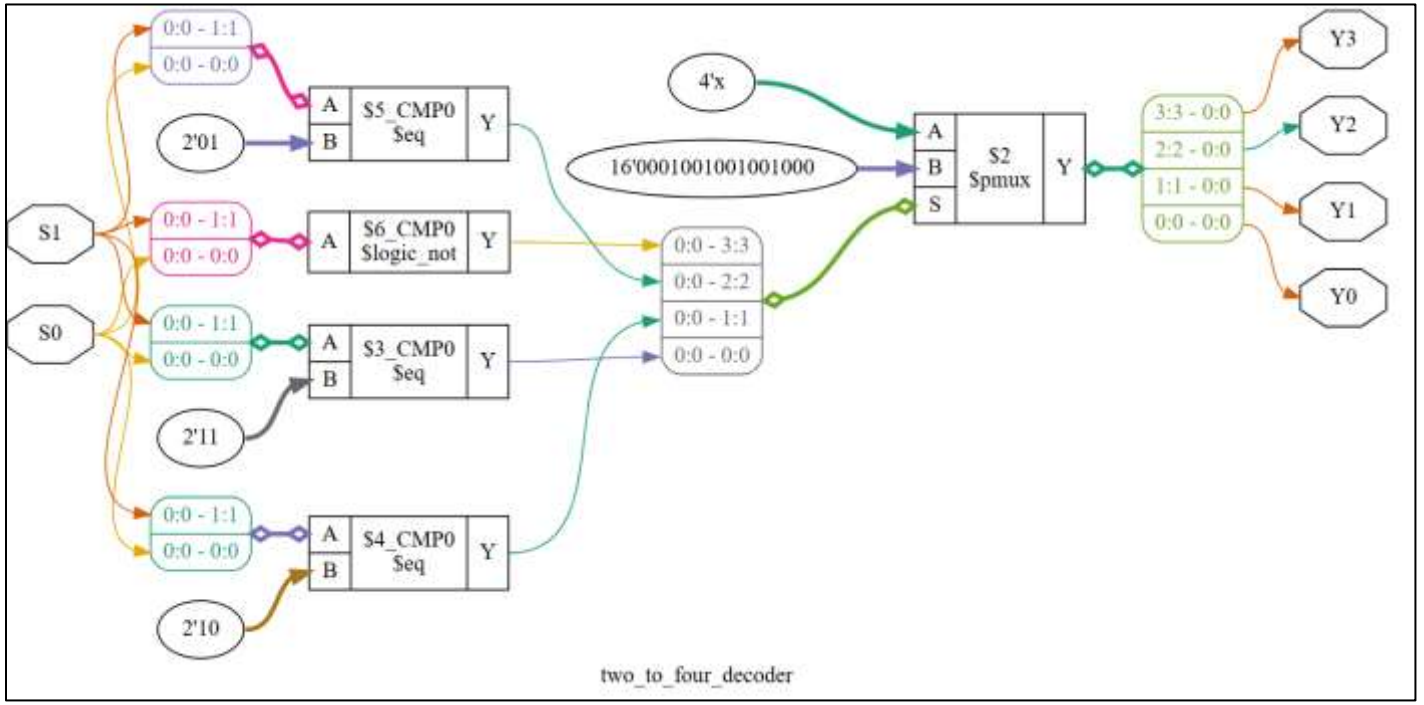


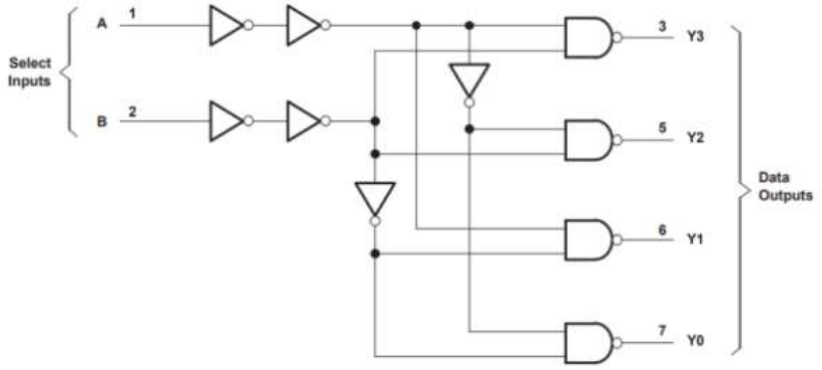
four_bit_alu

There are two adders (A-B) and (A+B) output is selected with MUX

2 to 4 bit decoder

```
module two_to_four_decoder(
    input S0, S1,
    output reg Y0, Y1, Y2, Y3
);
always @* begin
    case ({S1, S0})
        2'b00: {Y3, Y2, Y1, Y0} = 4'b0001;
        2'b01: {Y3, Y2, Y1, Y0} = 4'b0010;
        2'b10: {Y3, Y2, Y1, Y0} = 4'b0100;
        2'b11: {Y3, Y2, Y1, Y0} = 4'b1000;
        default: {Y3, Y2, Y1, Y0} = 4'b0000;
    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**.

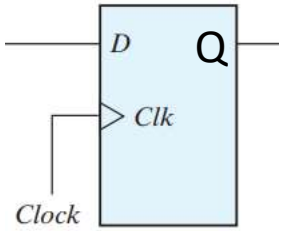
always @ (sensitivity list) begin
statement;
end

- 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<=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

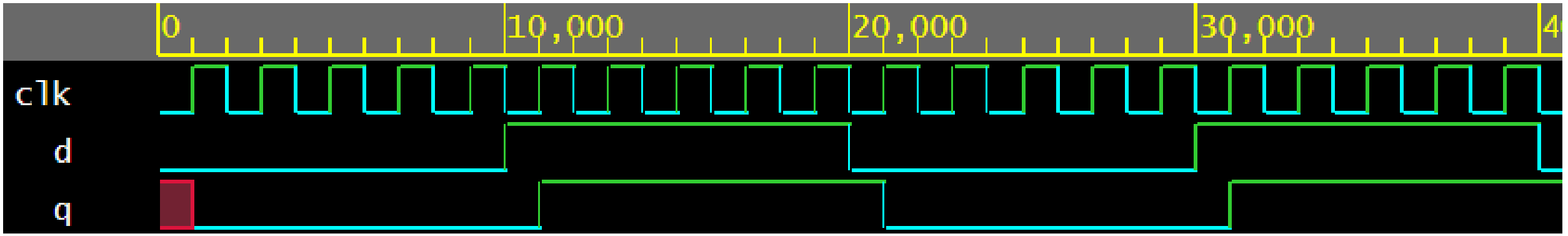
```
initial begin
    clk = 0;
    forever  #25  clk = ~clk;
end
```

```
// 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
```

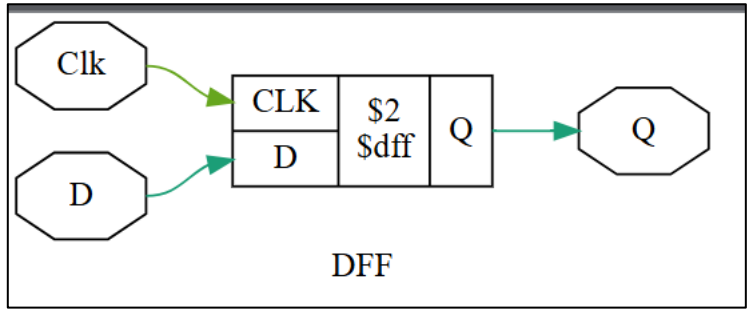
Verilog timescale

- 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

[12]The 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.

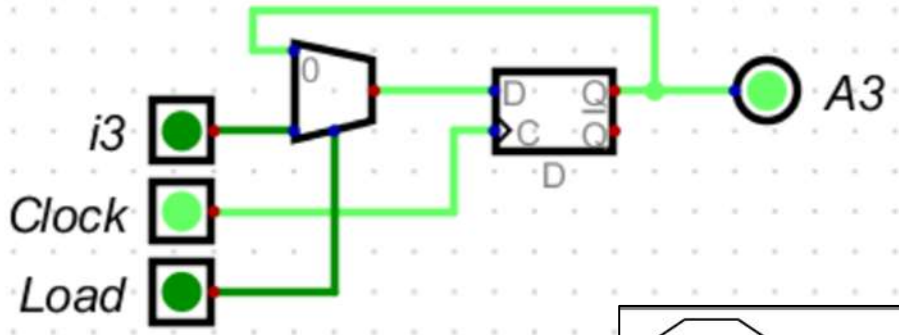


D-FlipFlop: Synthesis

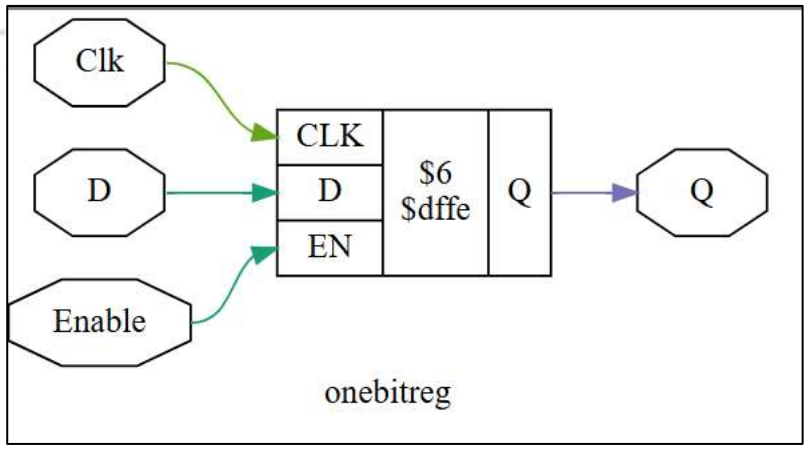



```
// D flip-flop without reset
`timescale 1 ns / 1 ps
module DFF (Q, D, Clk);
output Q;
input D, Clk;
reg Q;
always @ (posedge Clk)
 Q <= D;
endmodule
```

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

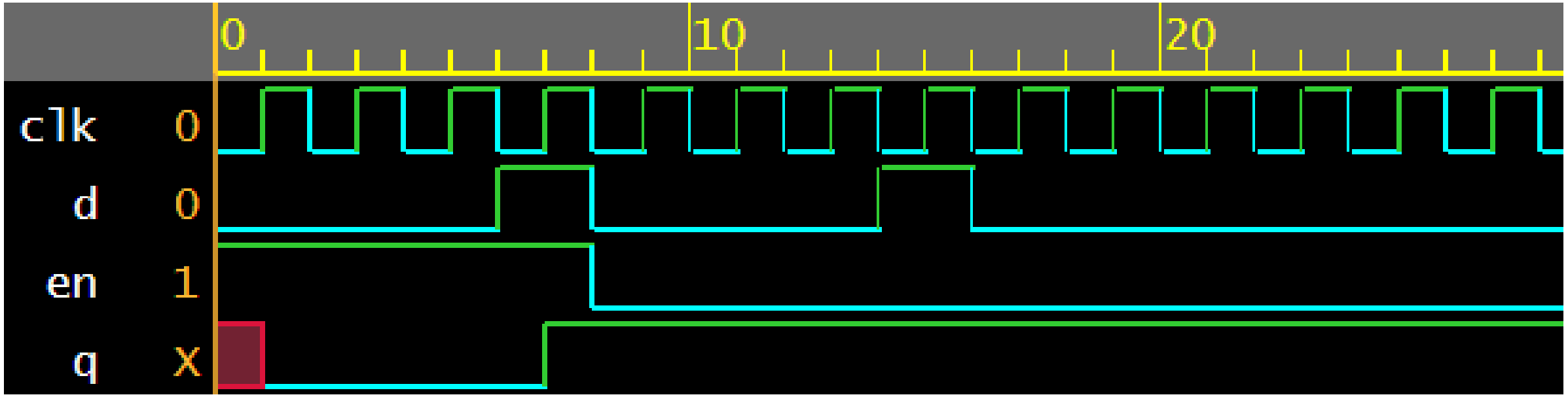
```
`timescale 1 ns / 1 ns
module onebitreg (
output reg Q,
input D,Enable,Clk);

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

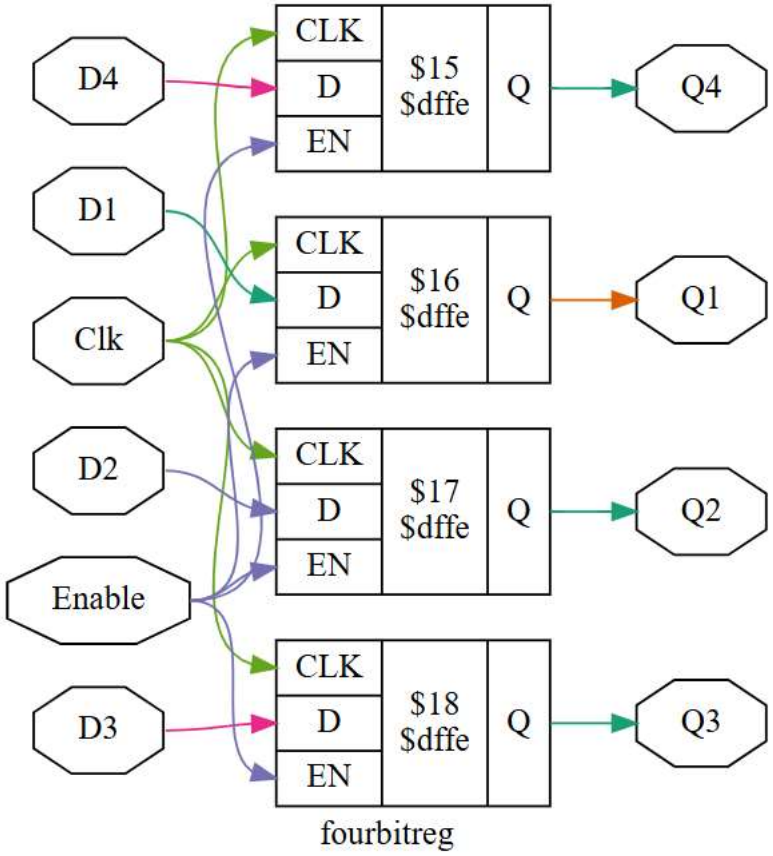
```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
  reg d,en,clk;  //inputs
  wire q; //outputs

  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

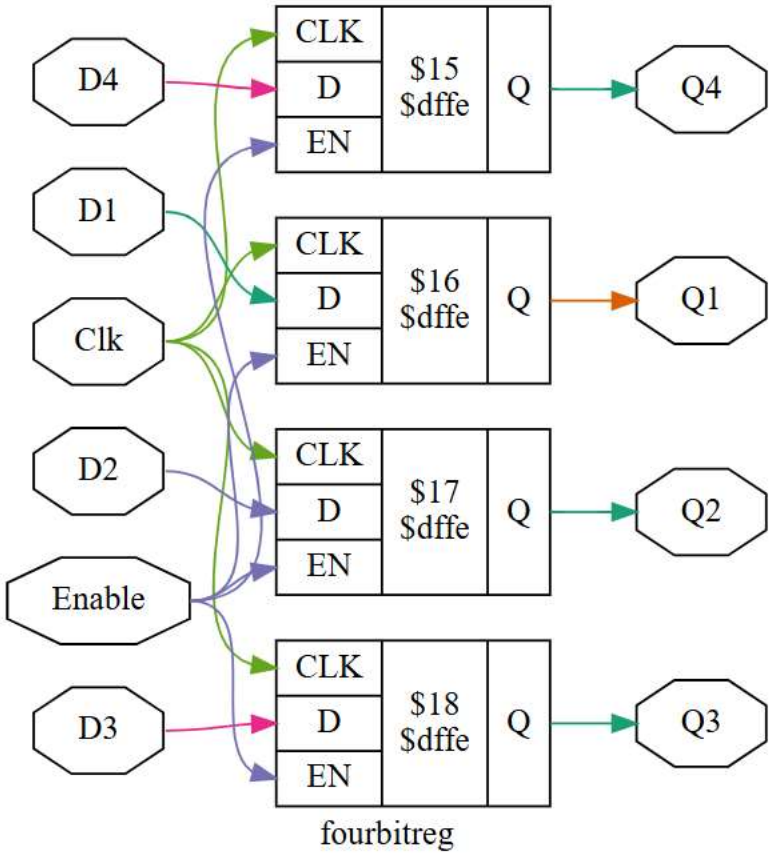


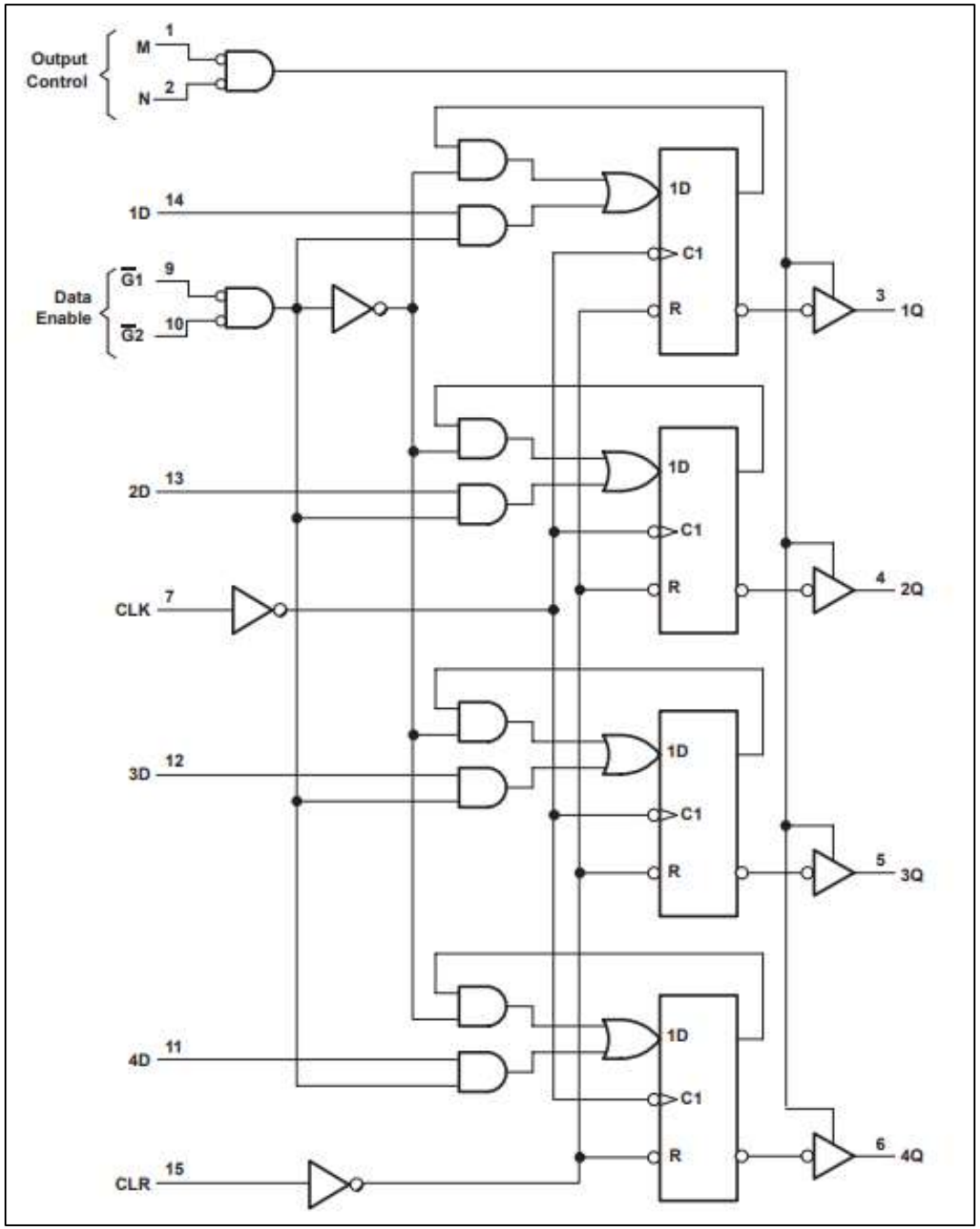
D-FlipFlop: 4bit register



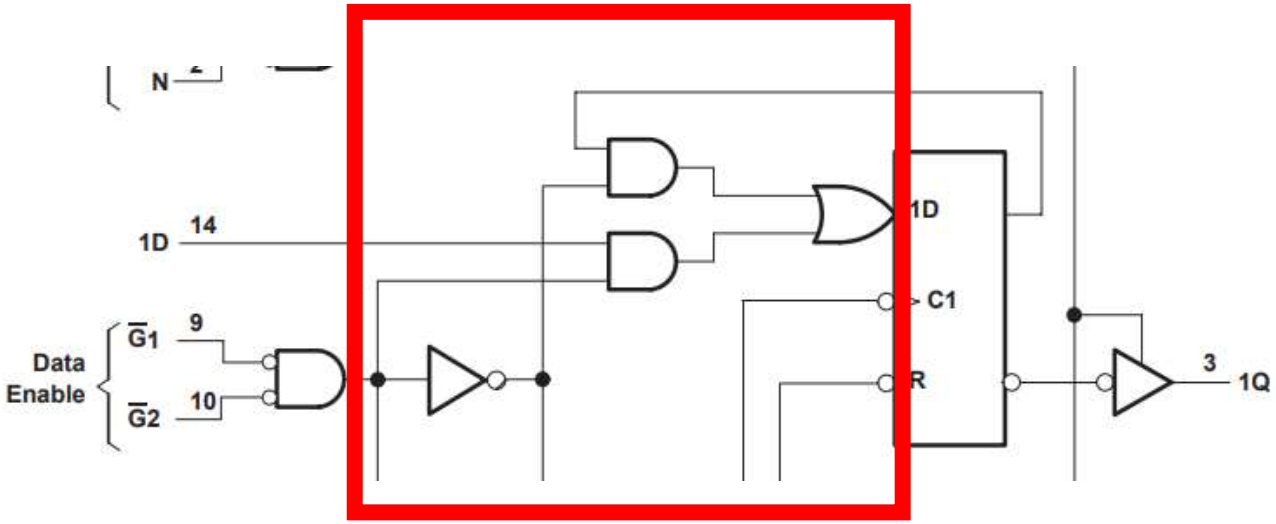

```
`timescale 1 ns / 1 ns
module fourbitreg (
output reg Q1,Q2,Q3,Q4,
input D1,D2,D3,D4,Enable,Clk);

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



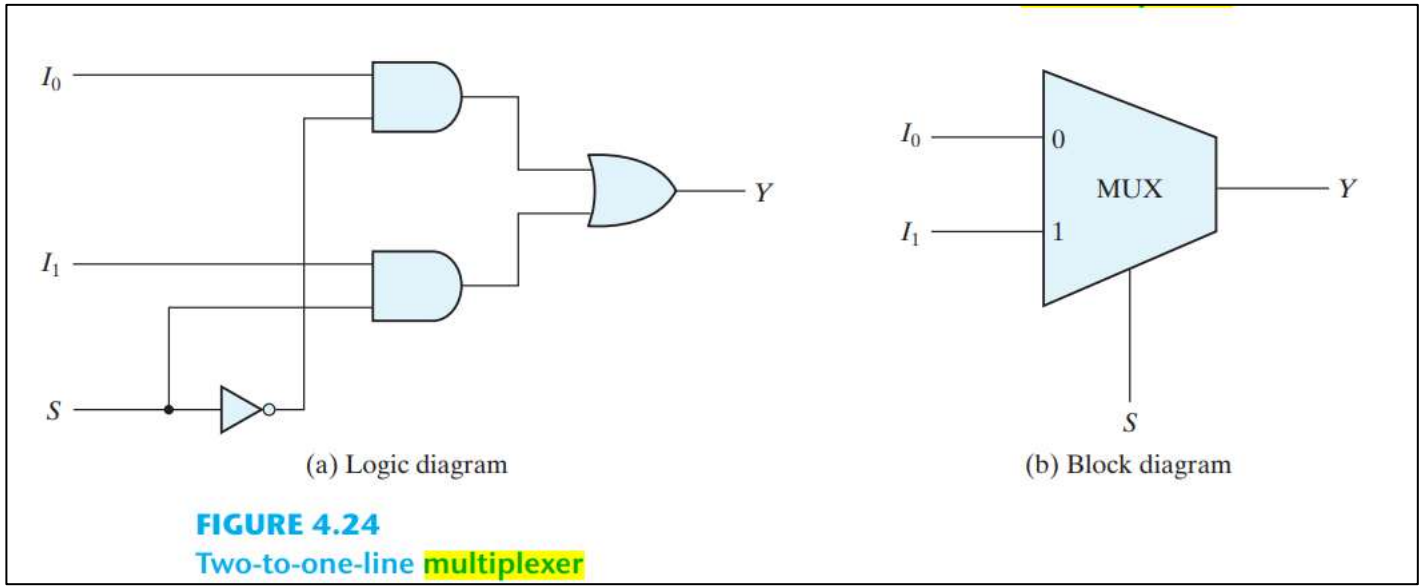
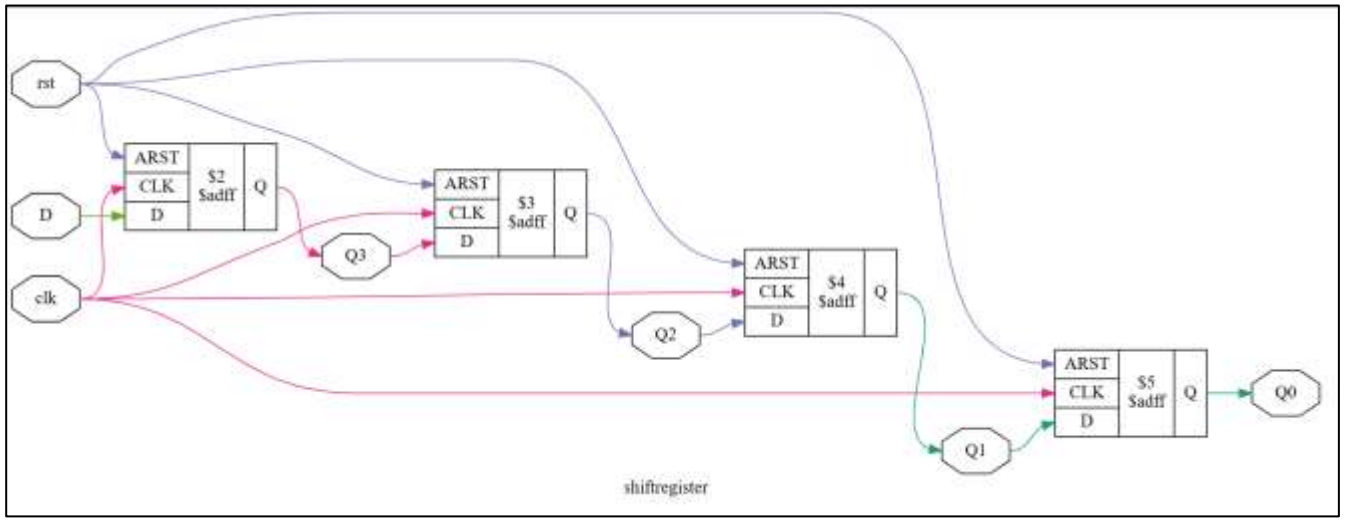


FIGURE 4.24
Two-to-one-line multiplexer

Shift Registers

- input data is serial and bit is shifted right at each clk posedge
- D→Q3 Q2 Q1 Q0



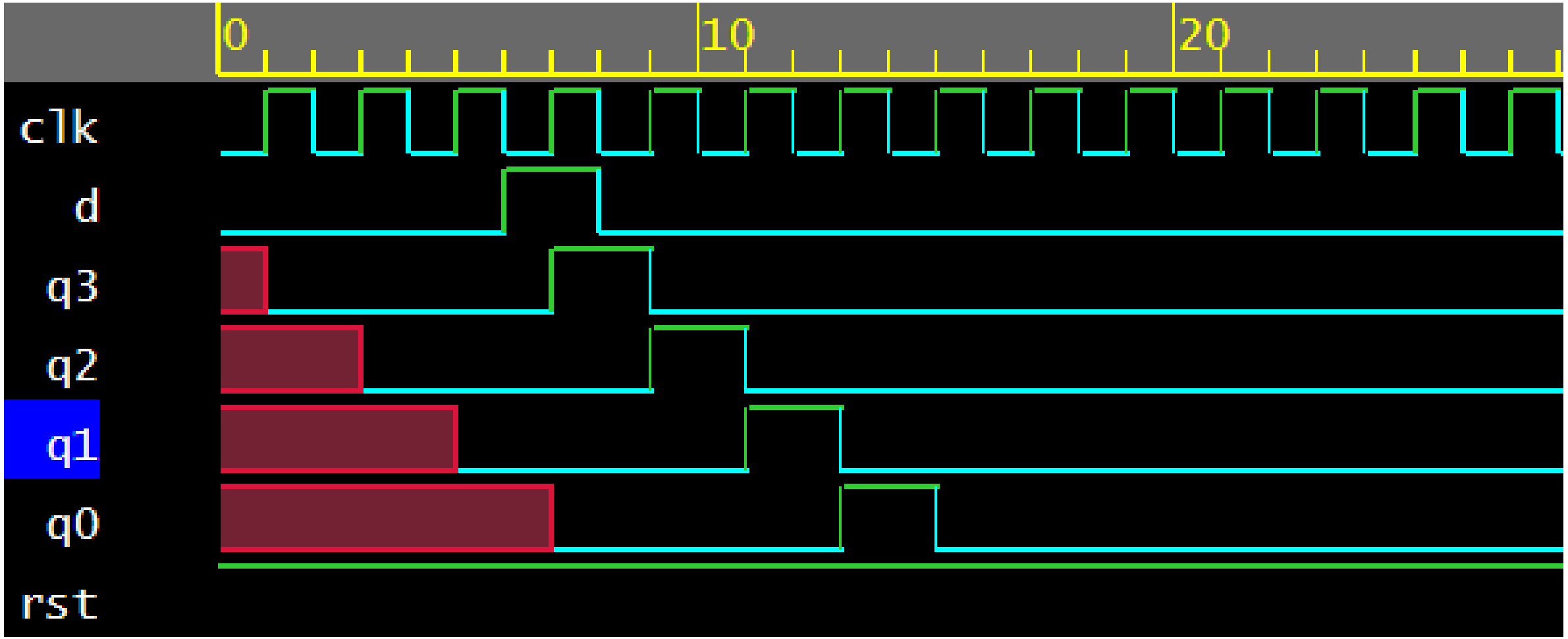

```
`timescale 1 ns / 1 ns
module shiftregister (
output reg  Q3,Q2,Q1,Q0, // Register output
input        D, // Serial data
clk, rst // clock and reset
);
always @ (posedge clk, posedge rst)
  if (rst) begin
    Q3 <= 1'b0;
    Q2 <= 1'b0;
    Q1 <= 1'b0;
    Q0 <= 1'b0;
    end
  else begin
    Q3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

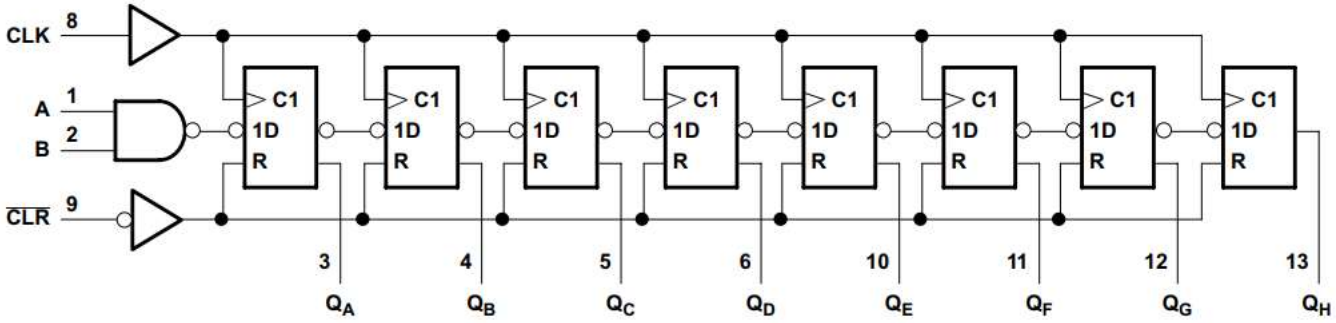
```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
  reg clk, rst;  //inputs
  reg d; //inputs
  wire q3,q2,q1,q0; //outputs
  // Instantiate
  shiftregister D0 (.clk(clk),.D(d), .Q3(q3),.Q2(q2),.Q1(q1),.Q0(q0), .rst(rst));
   initial begin
                    clk=0;
     forever #1 clk=~clk;
   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
```

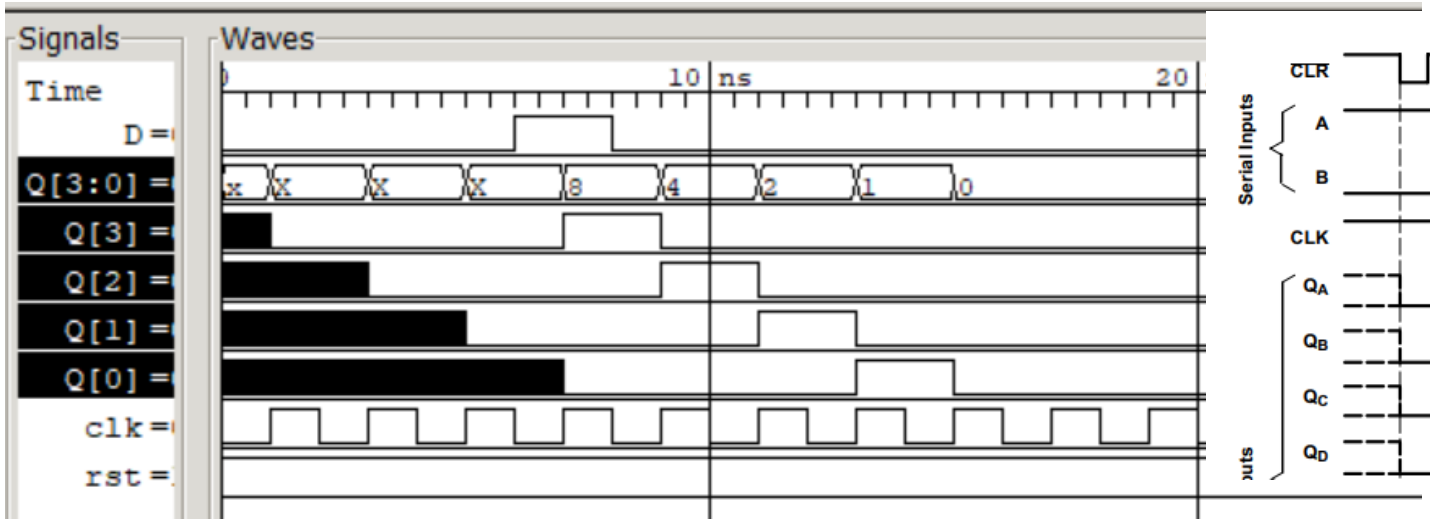
Shift Registers

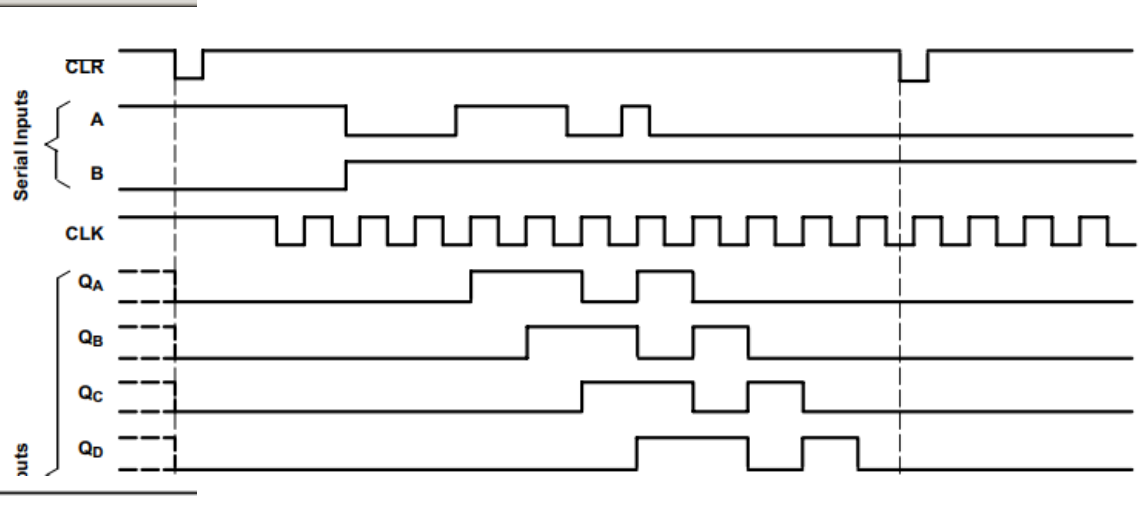


SNx4HC164 8-Bit Parallel-Out Serial Shift Registers

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

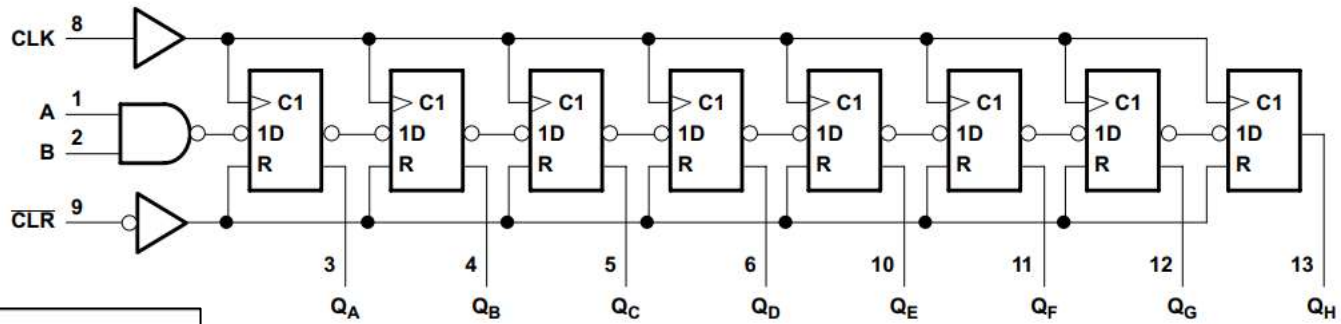


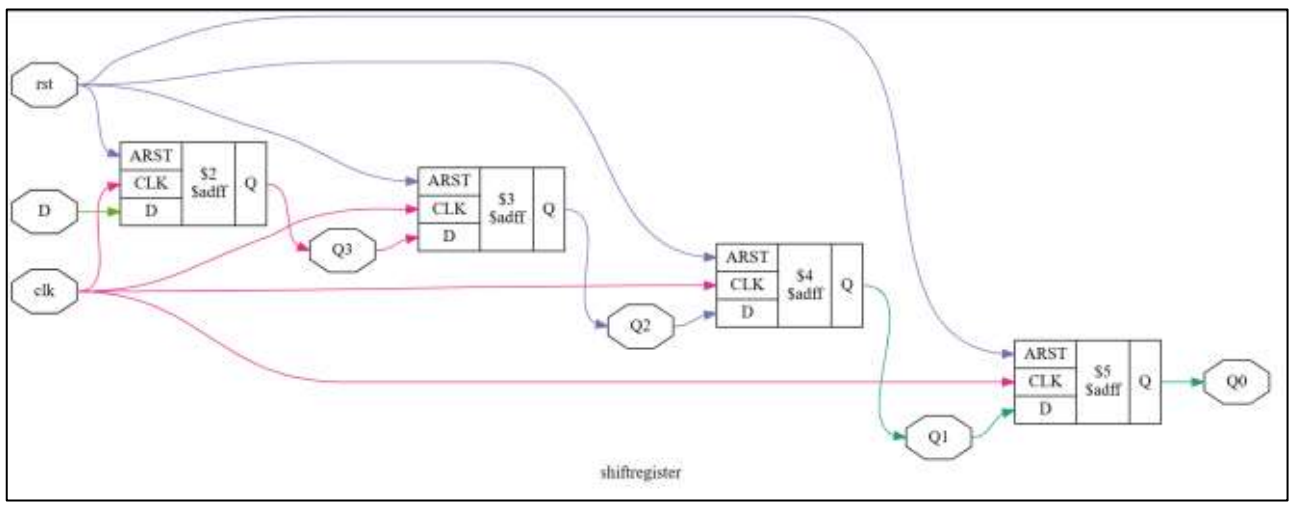




SNx4HC164 8-Bit Parallel-Out Serial Shift Registers

- 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

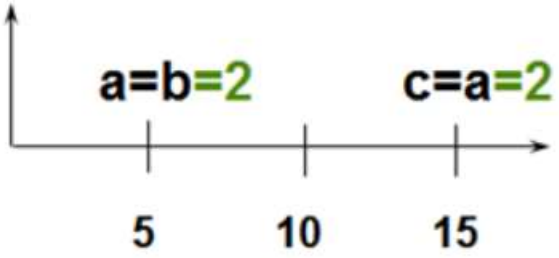
Blocking (=)

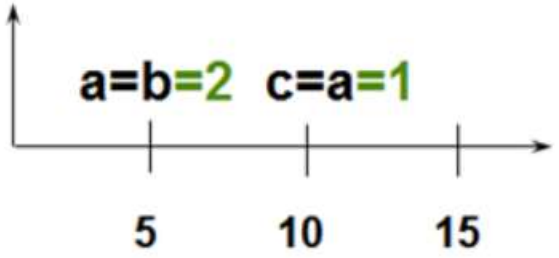
```
initial
  begin
      a = #5 b;
      c = #10 a;
  end
```

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
```

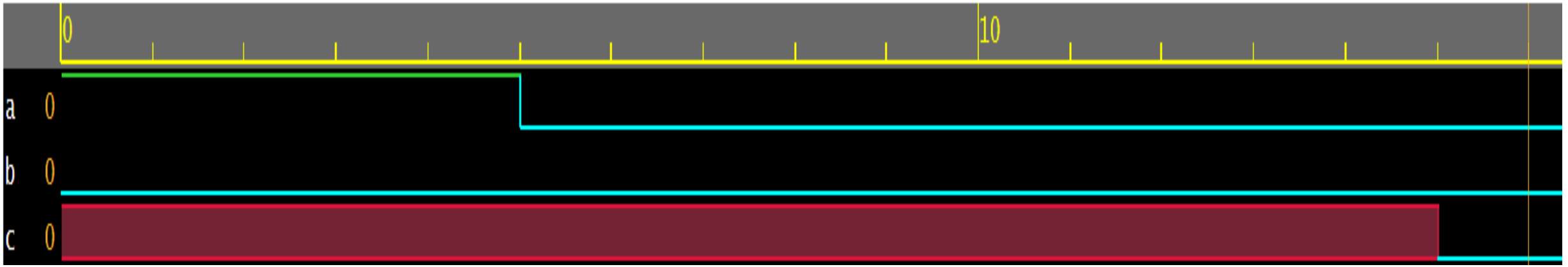
time=0 a=1 b=0 c=x
time=5 a=0 b=0 c=x
time=10 a=0 b=0 c=1

Blocking and Nonblocking Assignments

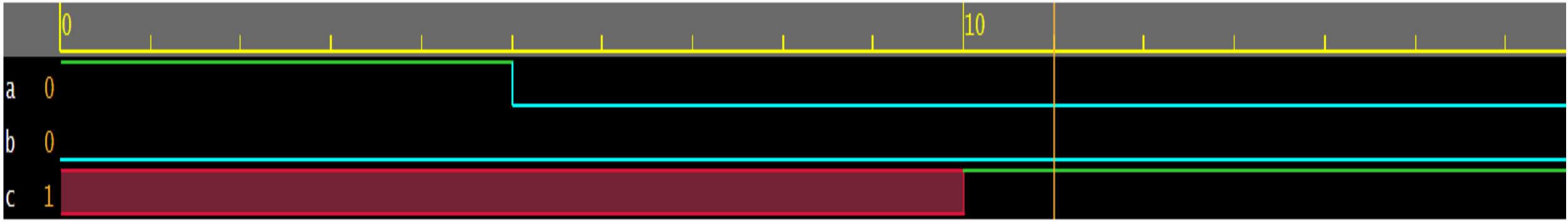
```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
  reg  a,b,c;  //inputs

  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



a=1;b=0; **a=#5b;** **c=#10a;**

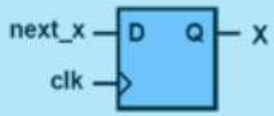


a=1;b=0; **a<=#5b;** **c<=#10a;**

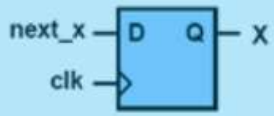
Blocking and Nonblocking Assignments

Blocking vs. Nonblocking Assignments

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

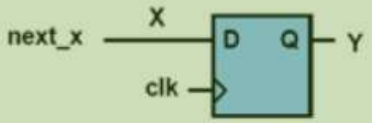



```
always @( posedge clk )
begin
  x <= next_x;
end
```

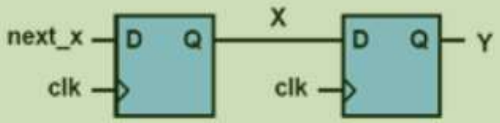


Same Behavior

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




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



Different Behavior

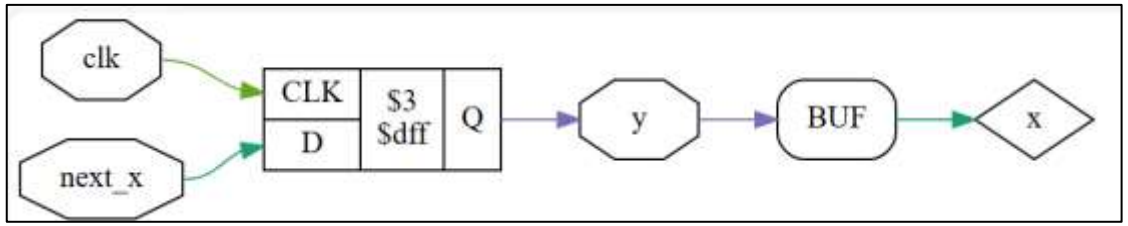
ALTERA. MEASURABLE ADVANTAGE™

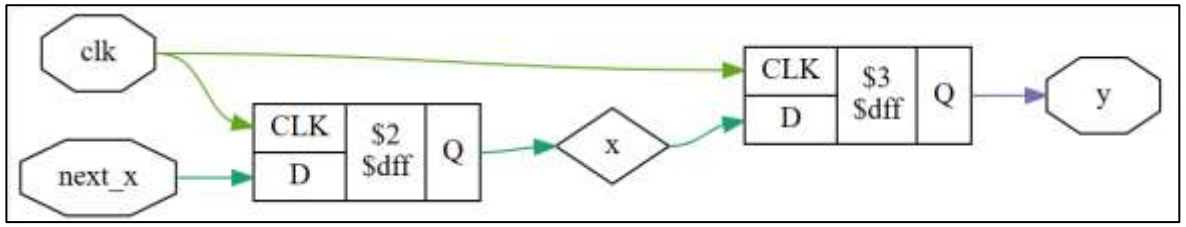
wrong use of blocking and non-blocking statements may cause synthesis to generate different logic circuits

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
```



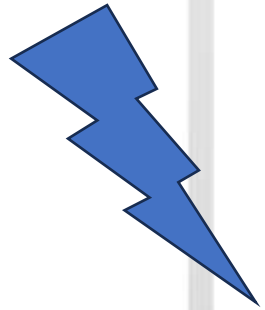


Blocking and Nonblocking Assignments

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

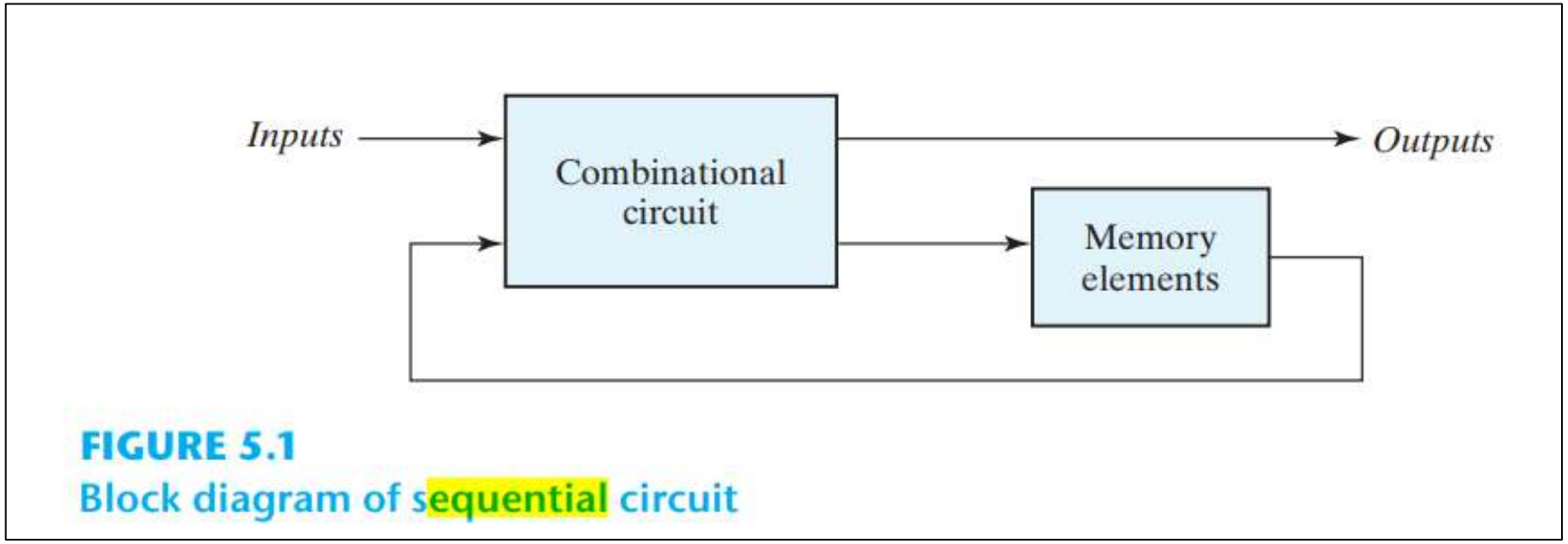


FIGURE 5.1
Block diagram of sequential circuit

Clock Divider

- We may need to generate signals at lower divided clock rates, example is UART transmission, FPGA clock rate in 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
```

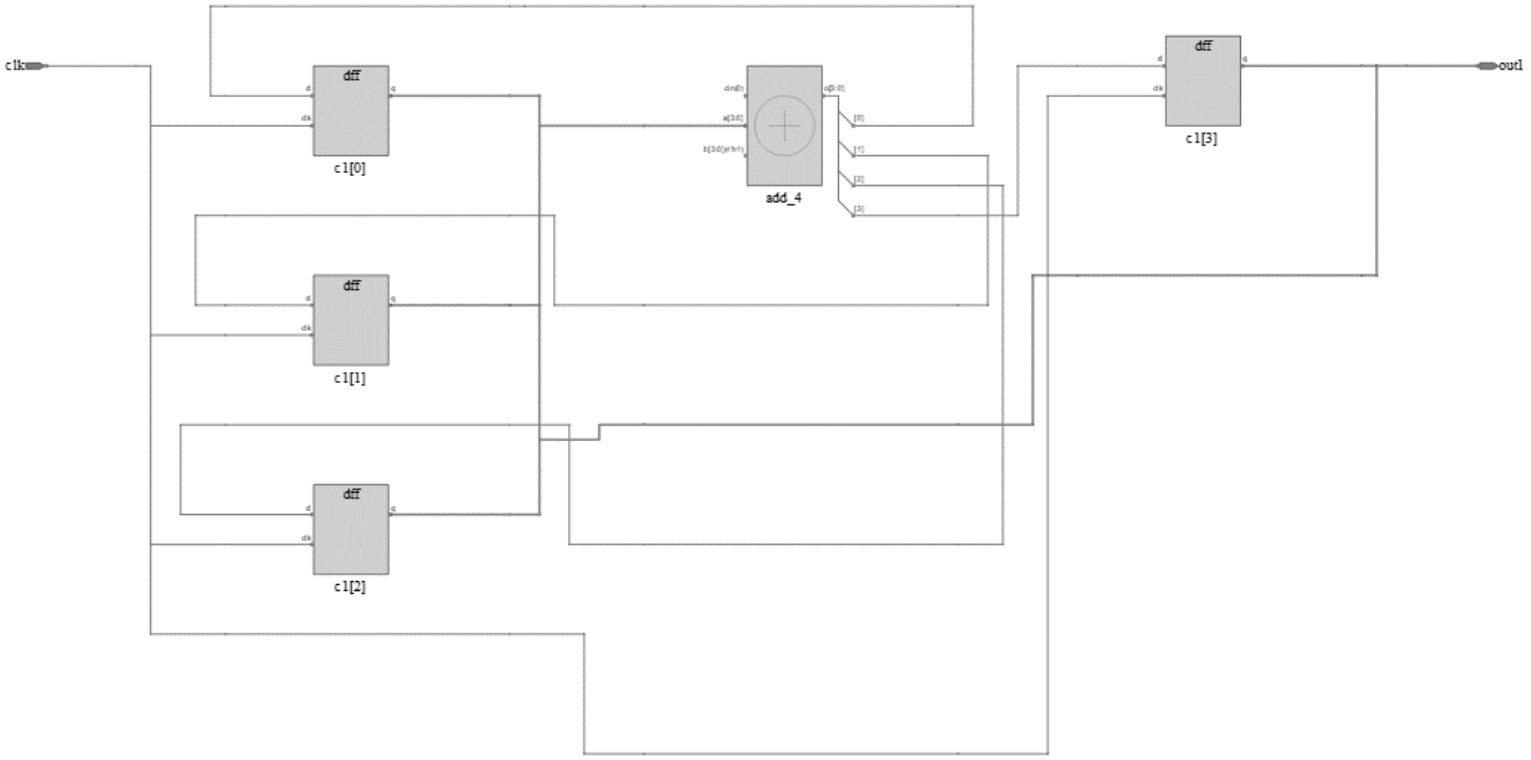
Clock Divider

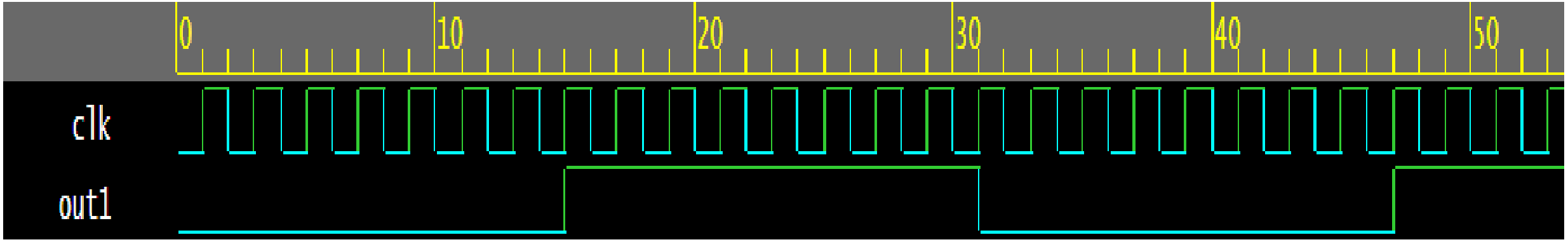
```
`timescale 1 ns / 1 ns
module clockdivider (input  clk, output
out1);
  reg [3:0] c1;
initial c1=4'b0000;
always @(posedge clk) begin
c1 <= c1 + 1'b1;
end
  assign out1 = c1[3:3];
endmodule
```

```
`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





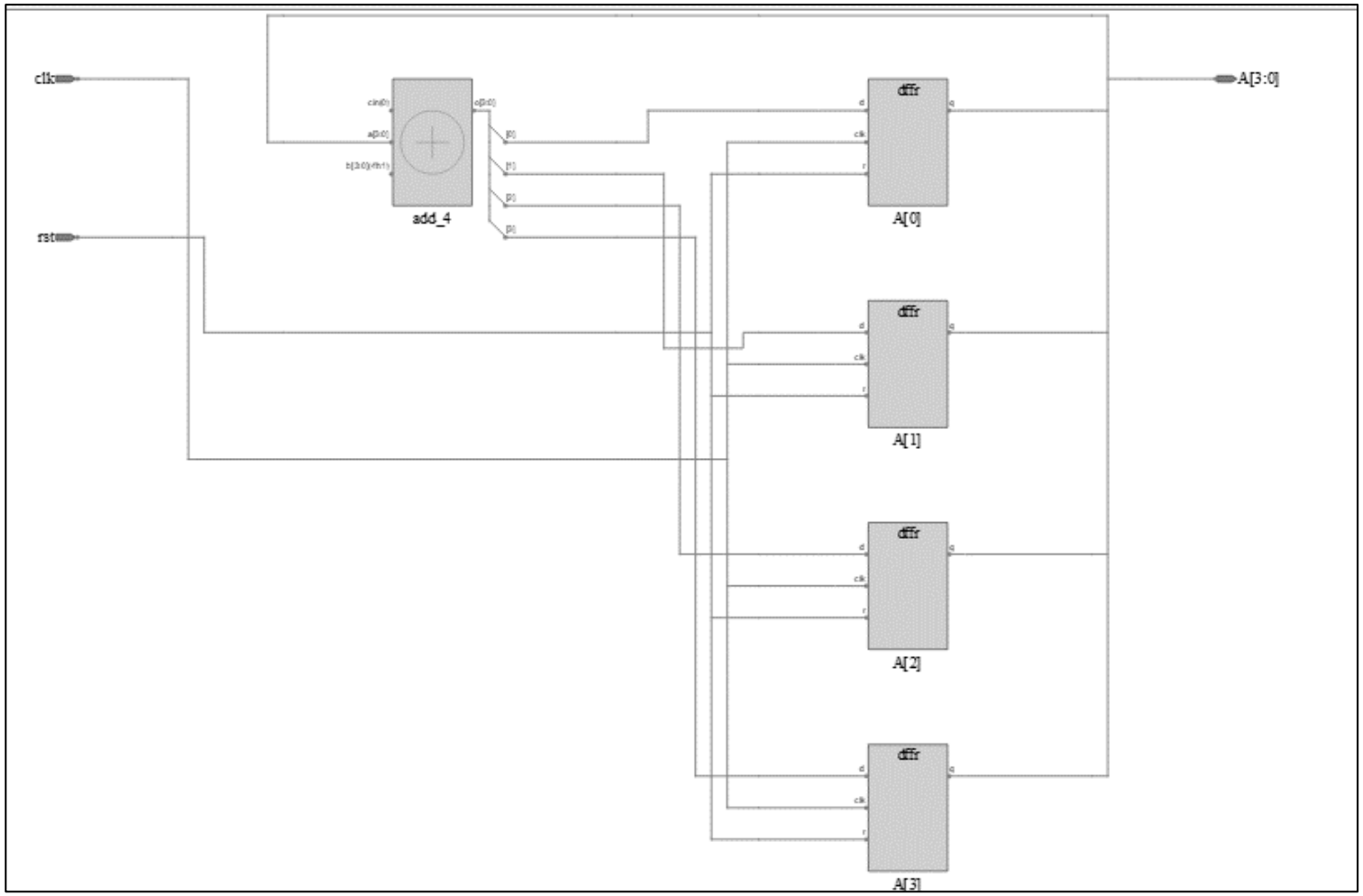
Ripple Counter: Behavioral

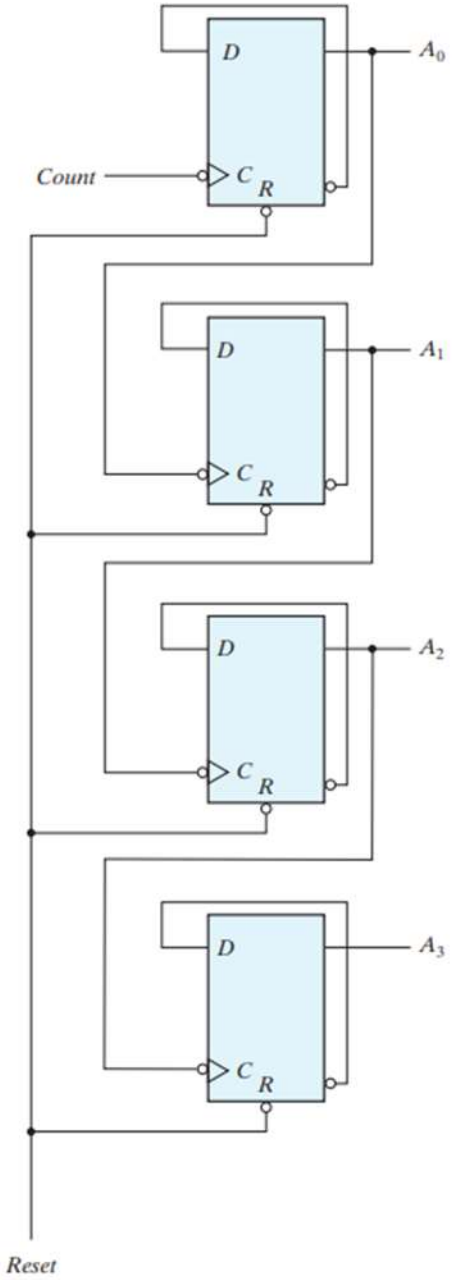
- If I have no idea about ripple counter design with D flip flops how can I define ripple counter
- I have clk and rst inputs
- output is 4bit
- at each clk, increment A+1
- just describe it Verilog will handle the rest

```
`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



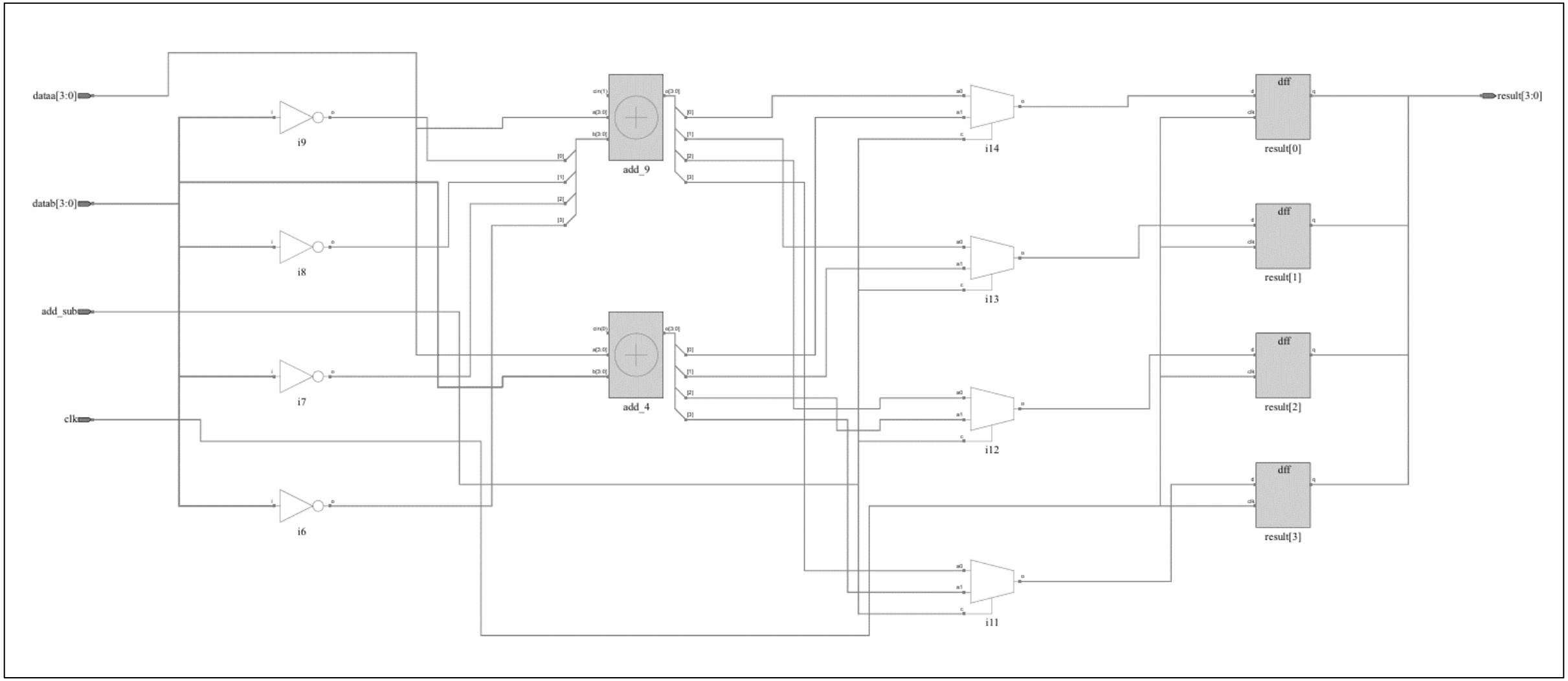


Verilog HDL: Adder/Subtractor

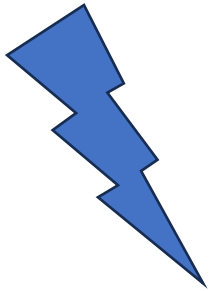
- adder subtractor is a mini version of ALU
  Arithmetic Logic Unit
- add_sub 1-bit selects the operator add or sub

```
`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

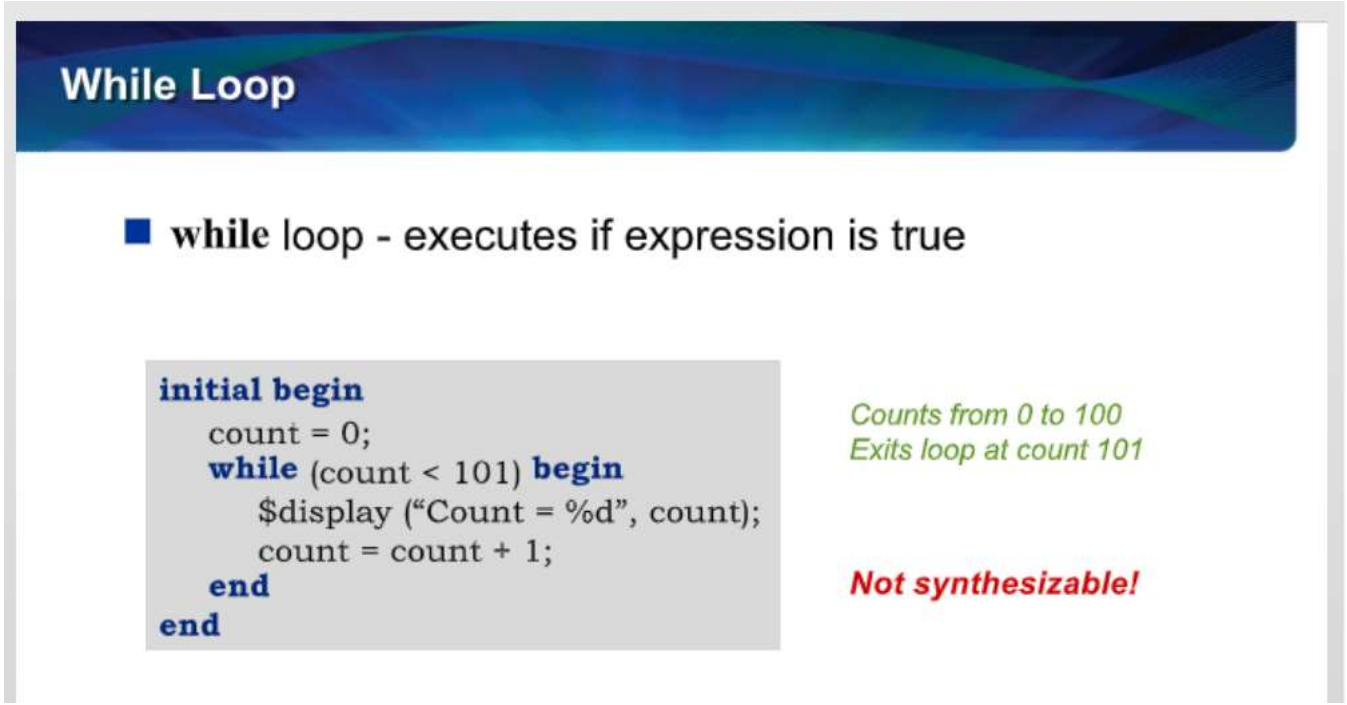
Not synthesizable!

- **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

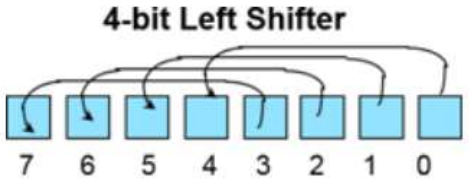
While Loop


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
```


ALTERA
MEASURABLE ADVANTAGE™

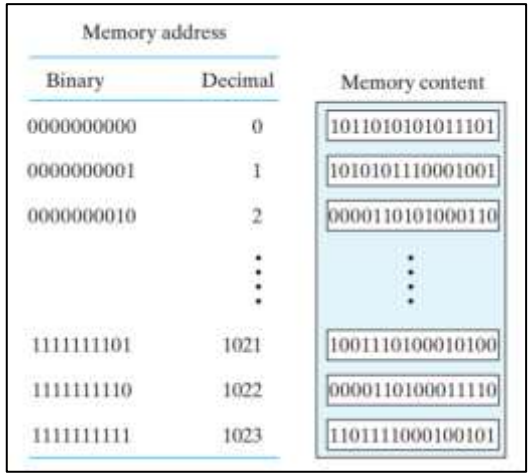
Memory

- A memory in Verilog is declared with a **reg** keyword, using a two-dimensional array.
- The first number specified in the array determines the number of bits in a word (the **word length**), and the second gives the number of **words in memory** (address).
- For example, a memory of 4 words with 8 bits per word is declared as

```
reg [7: 0] Mem [3:0]; // 8 x 4 memory
```

- Enable=1 required for Read/Write operation
- ReadWrite=1 Reads to Mem
- ReadWrite=0 Writes to Mem
- Enable=0; DataOut is high impedance

```
module memory (Enable, ReadWrite, Address, DataIn,
DataOut);
input Enable, ReadWrite;
  input [7:0] DataIn;
  input [3:0] Address;
  output reg [7:0] DataOut;
reg [7:0] Mem [3:0]; // 8 x 4 memory

always @ (Enable or ReadWrite or DataIn)
 if (Enable) begin
              if (ReadWrite) DataOut = Mem [Address]; // Read
              else Mem [Address] = DataIn; // Write
              end // if enable
  else DataOut = 8'bzzzzzzzz; // High impedance state
endmodule
```



Memory address		Memory content
Binary	Decimal	
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
⋮	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

Memory Test Bench

- Memory is empty
- first, display DataOut=xxxx, not defined value
- second, write DataIn=4’b0111 into Address=6’b000011
- #10 read the Address=6’b0011 and display DataOut=0111 is what was written before
- #10 disable access to Mem, check DataOut=zzzz means high impedance state
- so memory module is working as expected

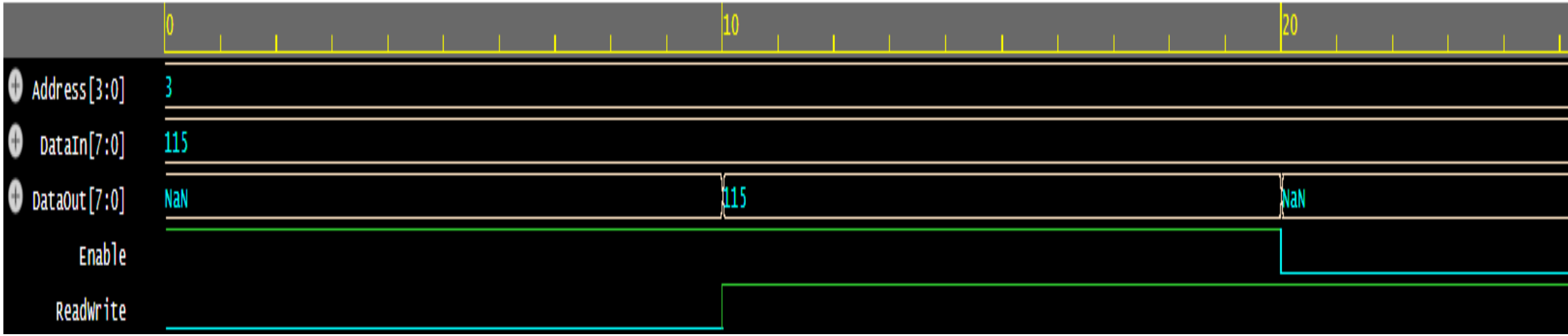
```
time= 0 DataOut=xxxxxxxx
time= 10 DataOut=01110011
time= 20 DataOut=zzzzzzzz
```

```
// Testbench
`timescale 1 ns / 1 ns
module tb_memory;
  reg Enable, ReadWrite;
  reg [7: 0] DataIn;
  reg [3: 0] Address;
  wire [7: 0] DataOut;

 memory M0 (Enable, ReadWrite, Address, DataIn, DataOut);
 initial begin
 $dumpfile("dump.vcd");
 $dumpvars();
   $monitor("time= %0t DataOut=%8b",$time, DataOut);
 Address=4'b0011;
 DataIn=8'b01110011;
 Enable=1;  //enable
 ReadWrite=0; //write
#10 ReadWrite=1; //read
#10 Enable=0;
#10 $finish;
end
endmodule
```

Memory Test Bench

- Memory is empty
- first, display DataOut=xxxxxxxx, not defined value
- second, write DataIn=4’b01110011 into Address=4’b0011
- #10 read the Address=4’b0011 and display DataOut=01110011 is what was written before
- #10 disable access to Mem, check DataOut=zzzzzzzz means high impedance state
- so memory module is working as expected

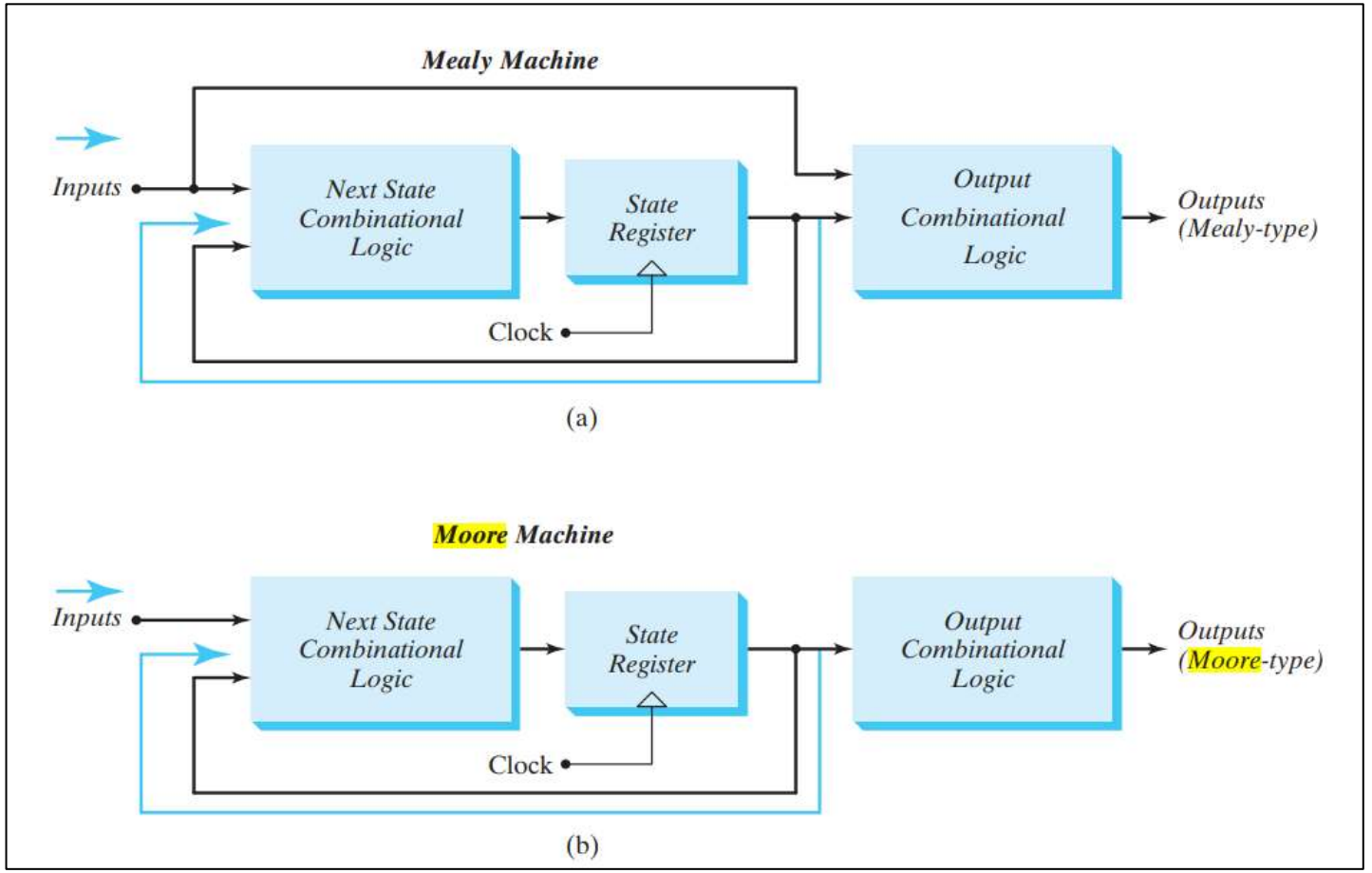
```
time= 0 DataOut=xxxxxxxx
time= 10 DataOut=01110011
time= 20 DataOut=zzzzzzzz
```



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**.

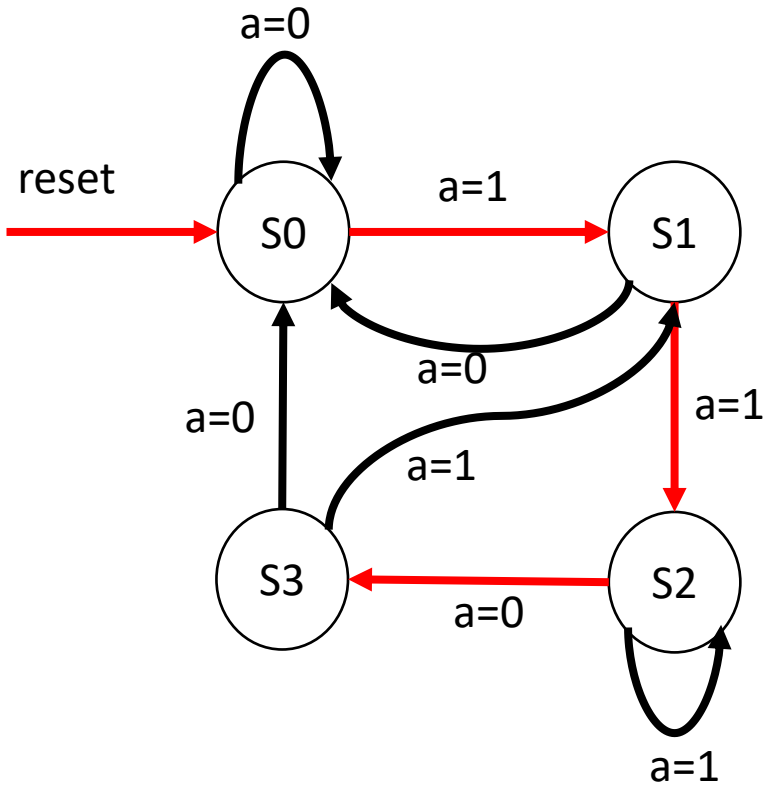
- The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated **FSM**



Finite State Machines: Pattern Detection

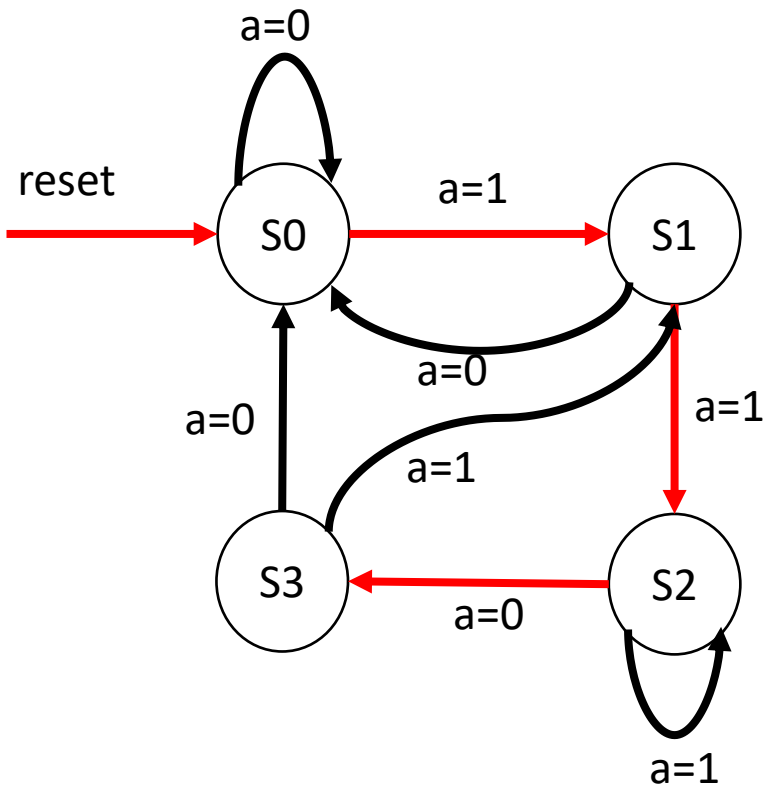
- In the **Moore model**, the output is a function of **only the present state**.

- 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

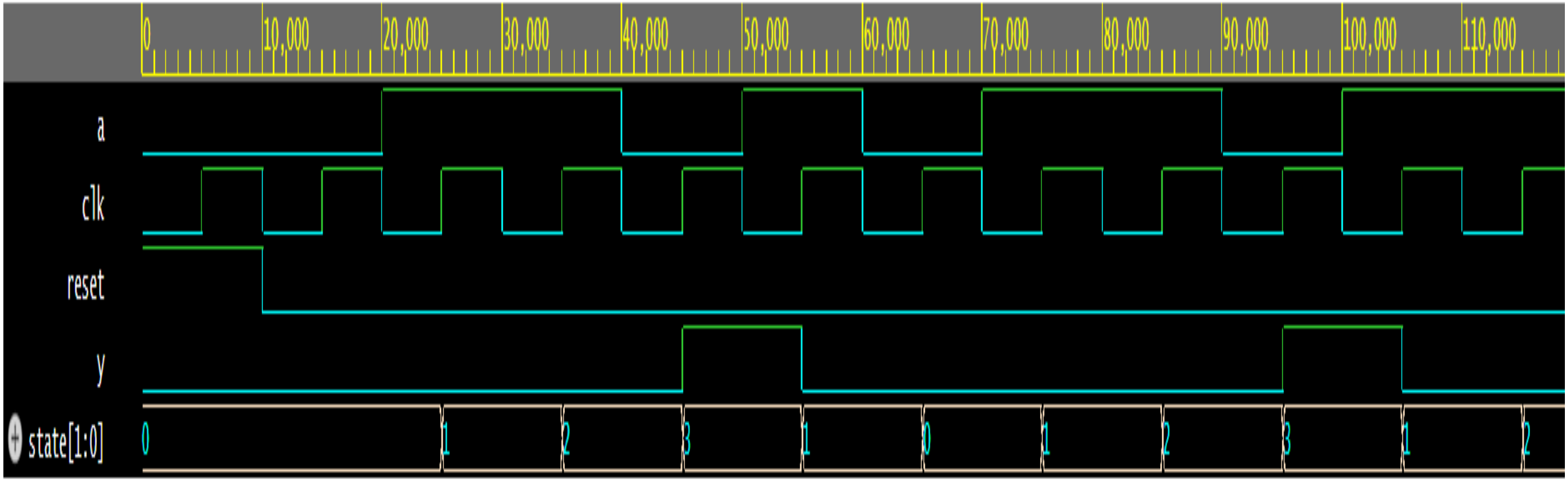
```
`timescale 1ns / 1ps

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);
```

```
    // 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

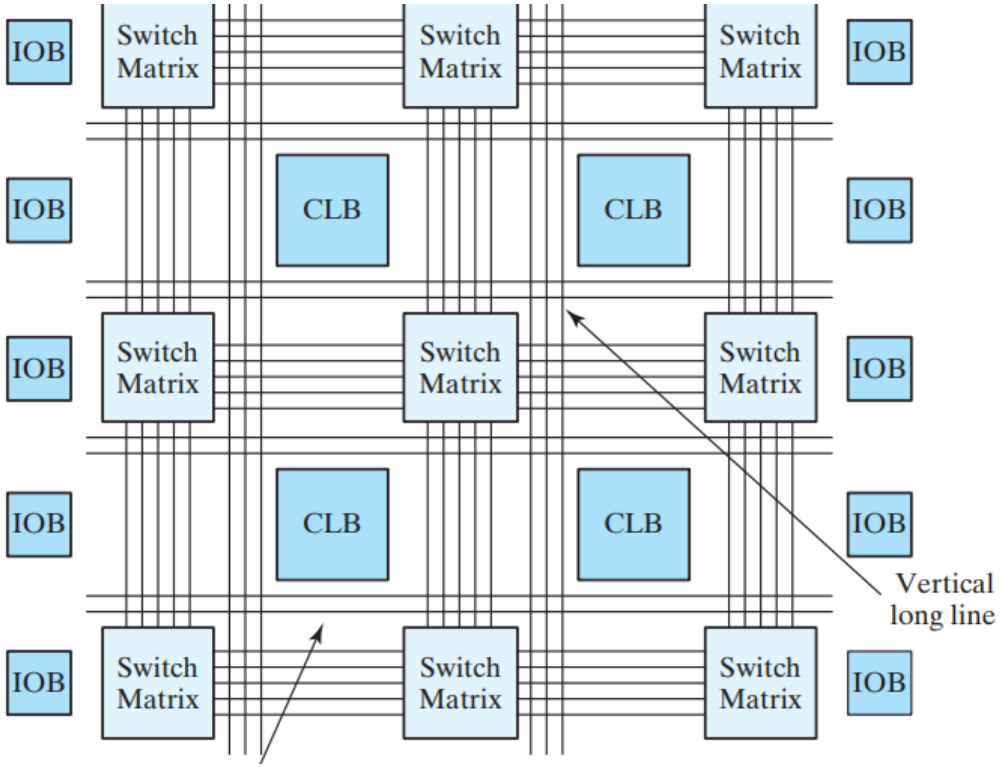
```
Time = 0 | clk = 0 | reset = 1 | a = 0 | state = 00 | y = 0
Time = 5000 | clk = 1 | reset = 1 | a = 0 | state = 00 | y = 0
Time = 10000 | clk = 0 | reset = 0 | a = 0 | state = 00 | y = 0
Time = 15000 | clk = 1 | reset = 0 | a = 0 | state = 00 | y = 0
Time = 20000 | clk = 0 | reset = 0 | a = 1 | state = 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
```

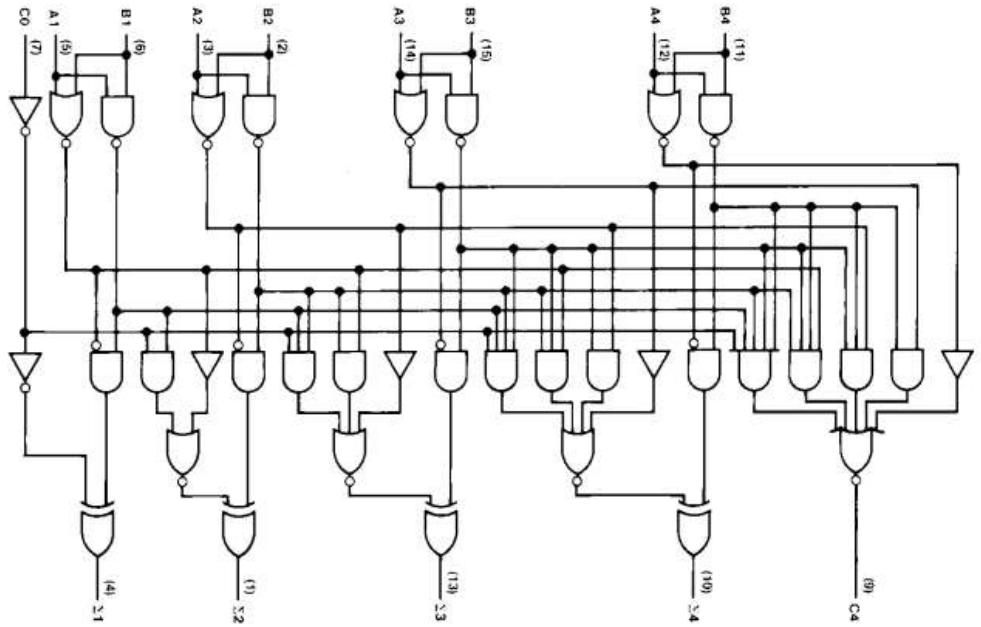
FSM: Pattern Detection



Logic Circuits vs FPGA

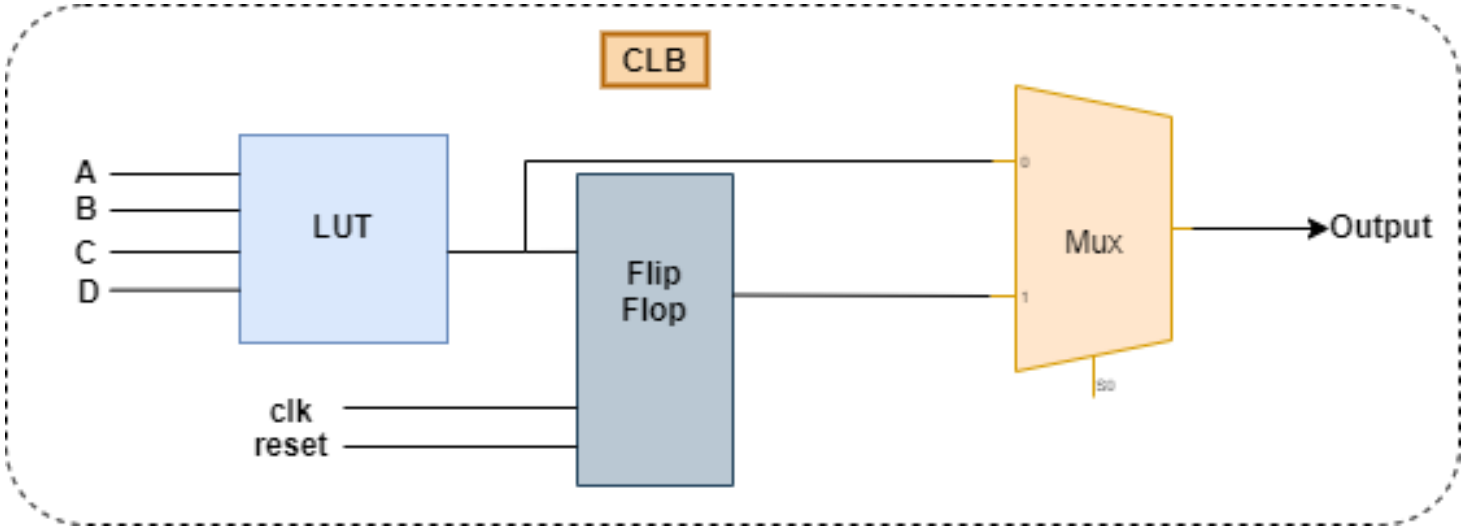
- Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects.
- FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes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

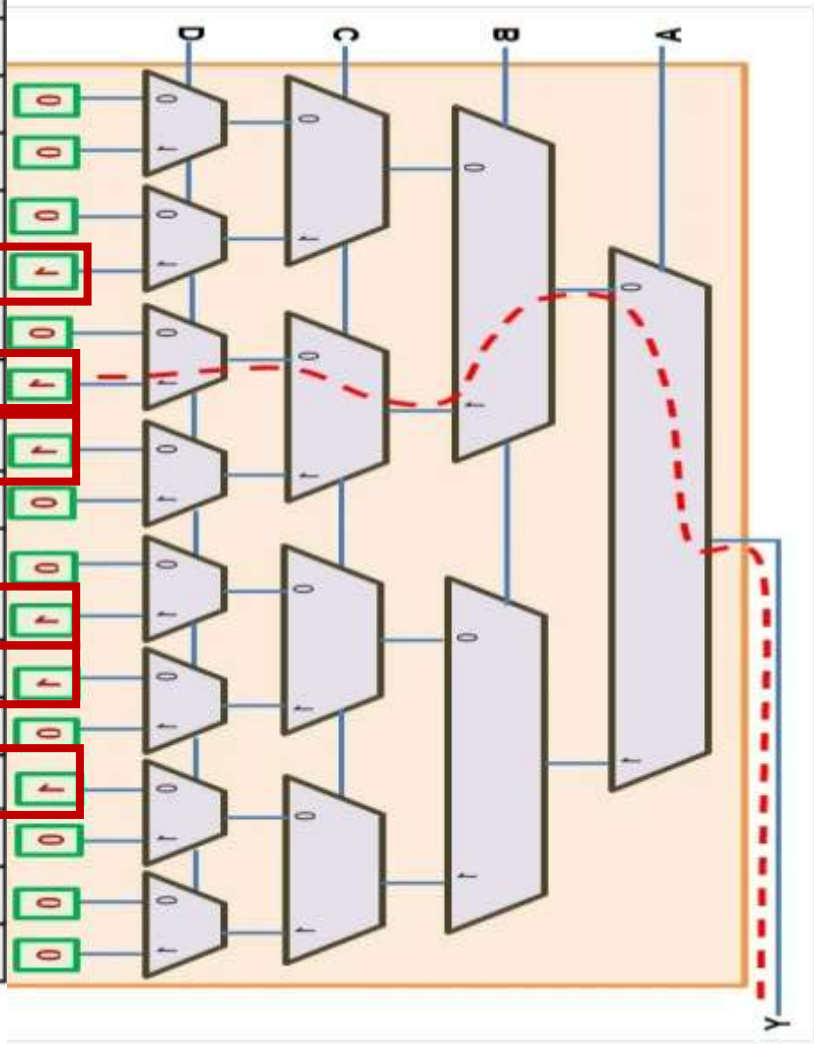


Implementation of Logic Functions using LUT

- 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).

Truth Table

Inputs				Output
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



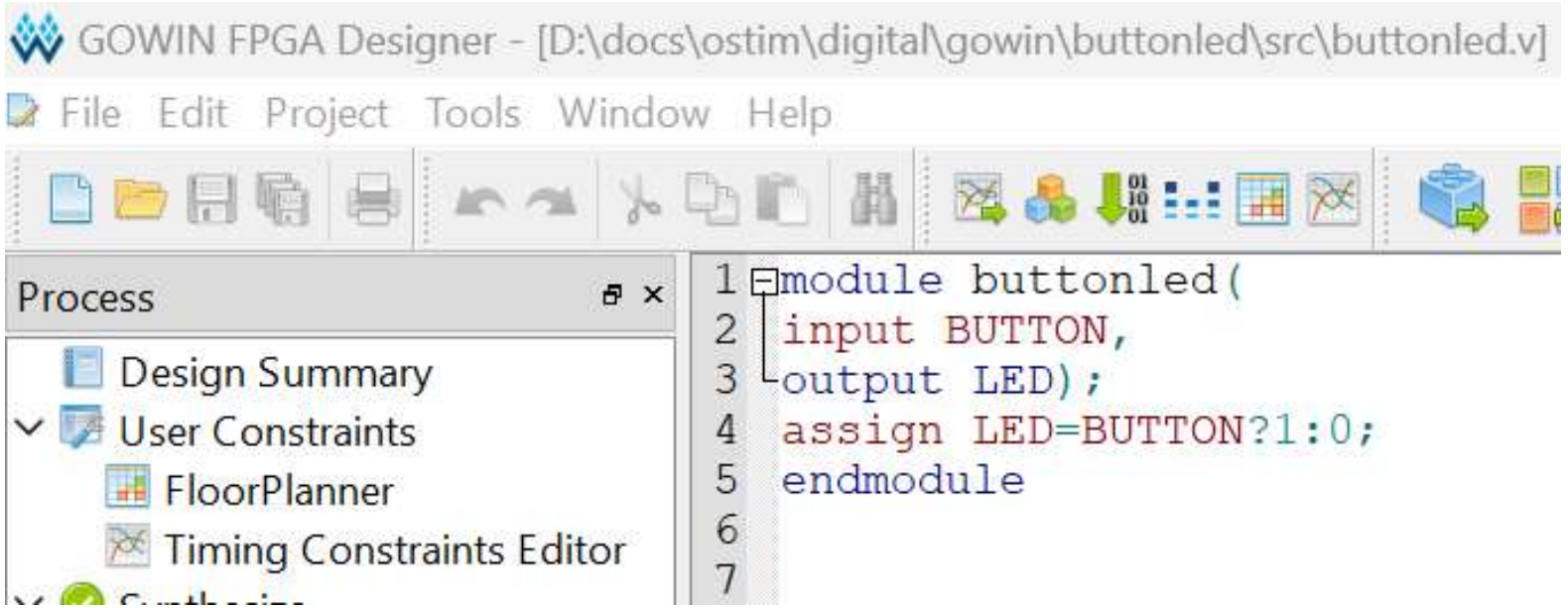
Verilog FPGA Implementation

- 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]

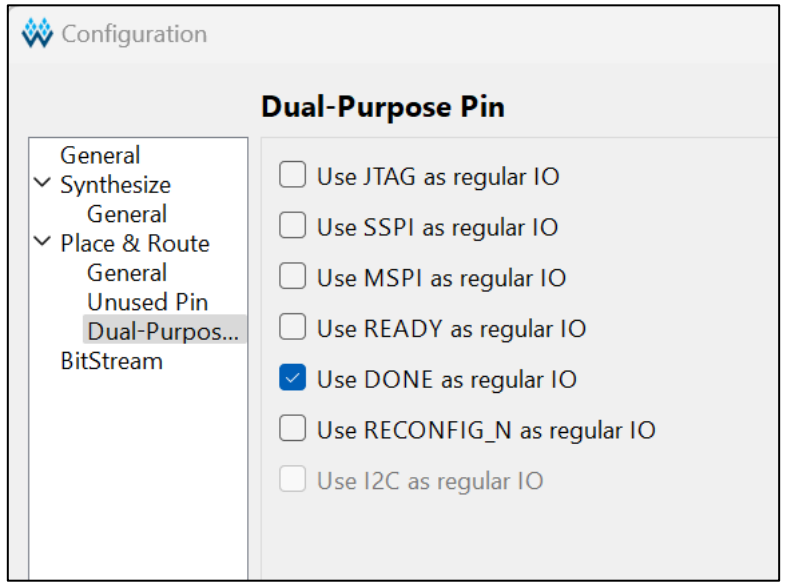


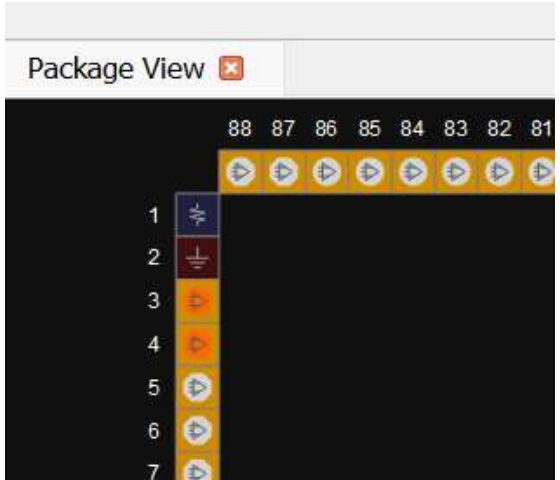
TANG NANO 9K Board: Button LED

- 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.



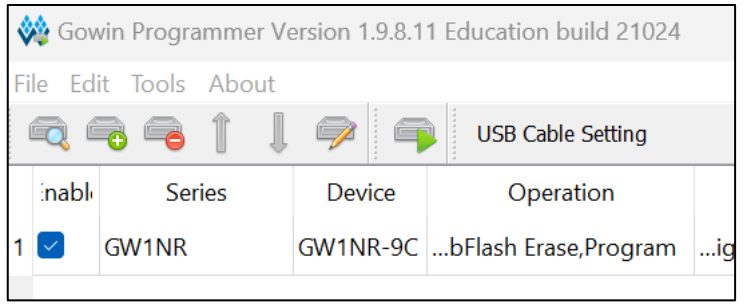


I/O Constraints

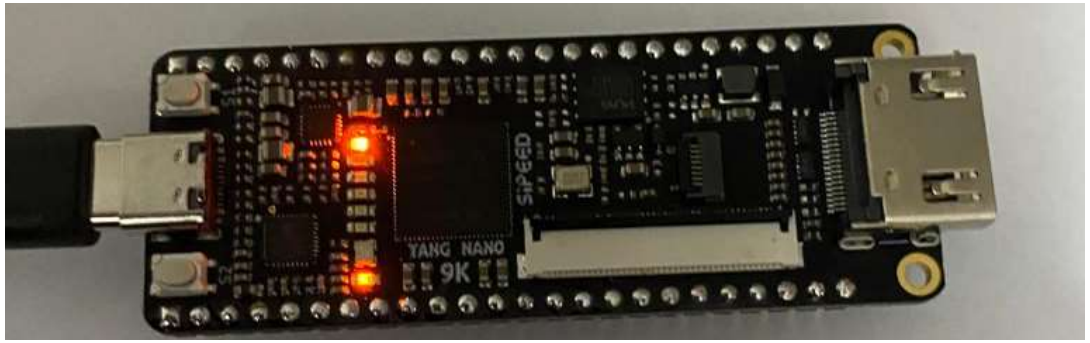
	Port	Direction	Diff Pair	Location	B
1	BUTTON	input		3	
2	LED	output		10	

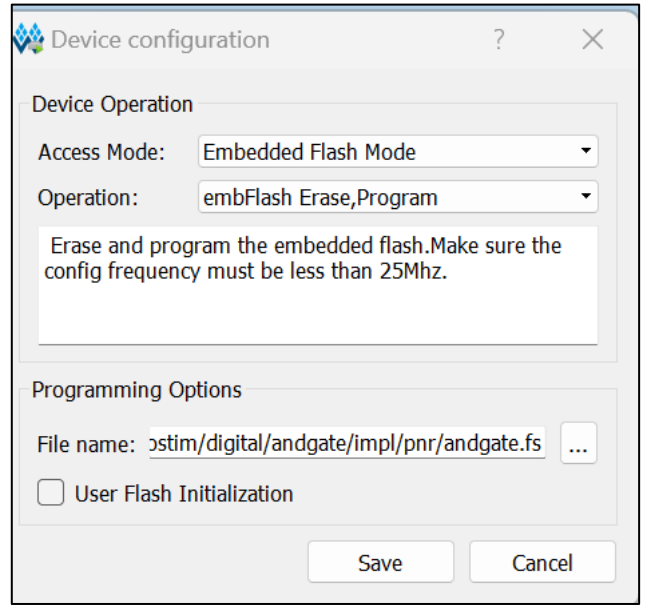
TANG NANO 9K Board: Button LED

- 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



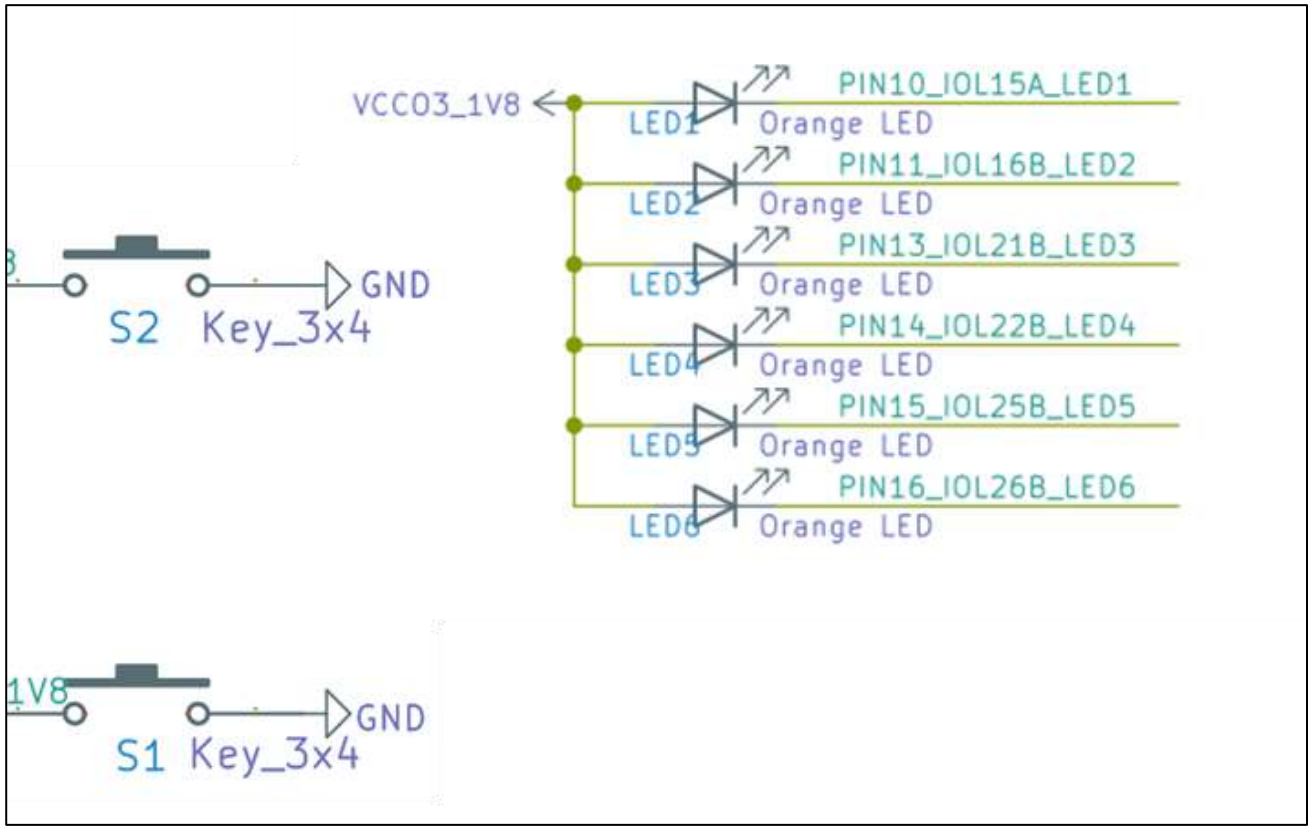


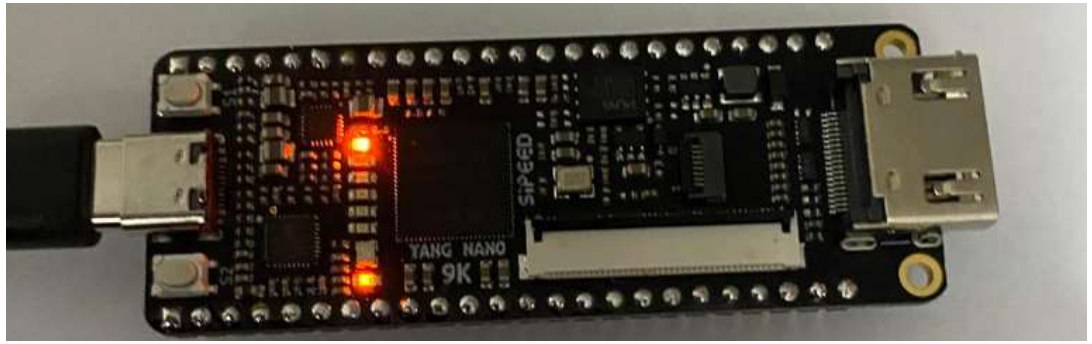
TANG NANO 9K Board: Button LED

- LED pins are connected ports as drain means:
- LED ligthining if LEDpin=0

- pressed BUTTON=1, notpressed BUTTON=0,

- Thus:
- LED=BUTTON?1:0

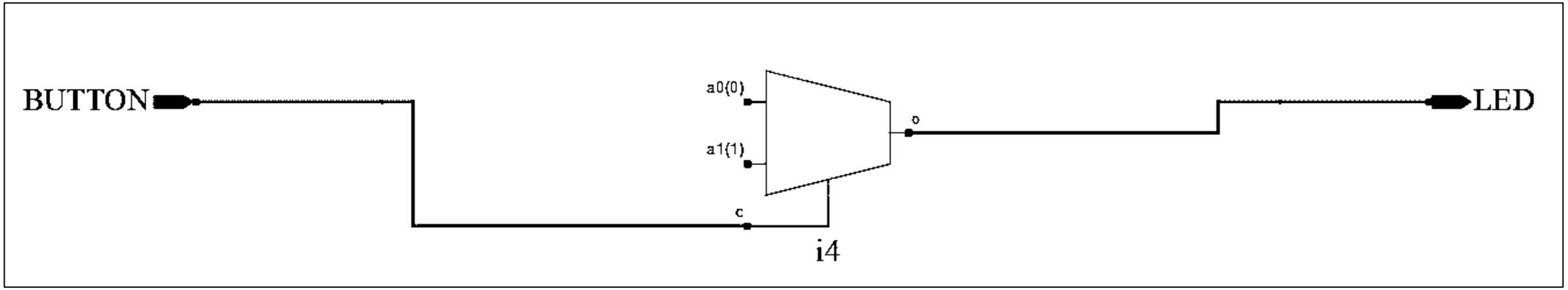




TANG NANO 9K Board: Syhthesis

- Tools-Shematic 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

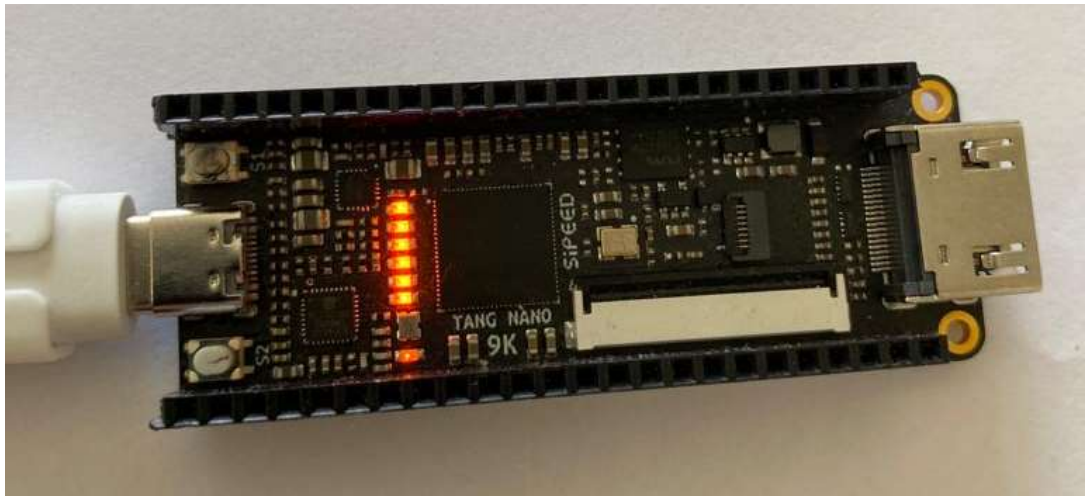
- Sequential circuits performs actions **at clock signals**. Thus we have clock input
- Output leds defined as 6-bit [5:0]
- localparam is local to the module, number of clock cycles that we we will wait for changing leds
- **localparam**: a local constant for storing integers, real numbers, time, delays, 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



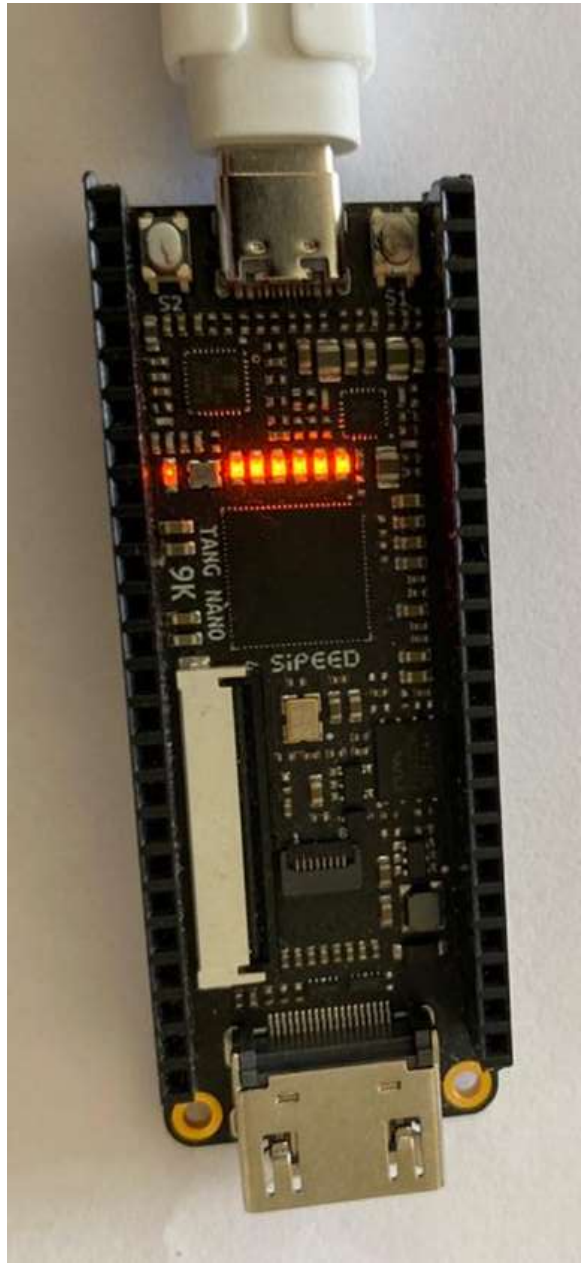

```
module blinkled
(
    input clk,
    output reg [5:0]  leds
);
localparam WAIT_TIME = 15000000;
reg [23:0] clockCounter = 0;
always @(posedge clk) begin
    clockCounter <= clockCounter + 1;
    if (clockCounter == WAIT_TIME) begin
        clockCounter <= 0;
        leds=~leds;
    end
end
endmodule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

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**

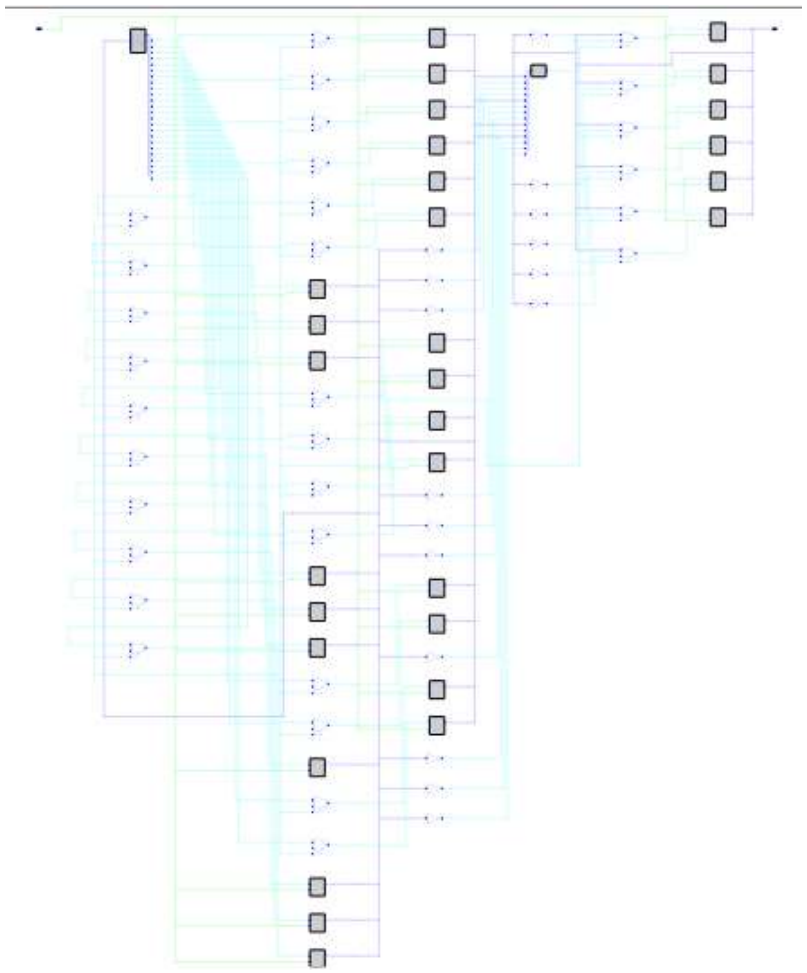
I/O Constraints

	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

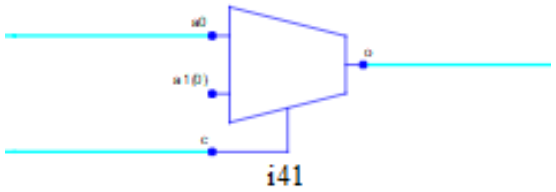
- a little complicated ☺
- lets zoom in

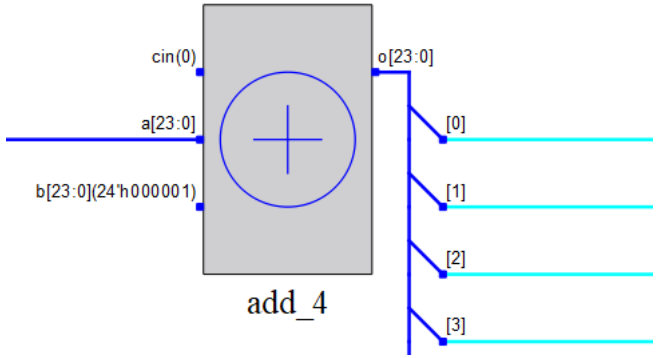


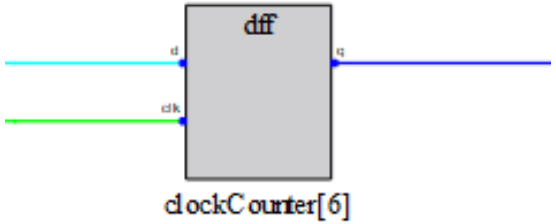
Tang Nano 9K: Blink LED RTL Shematic

- 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
```

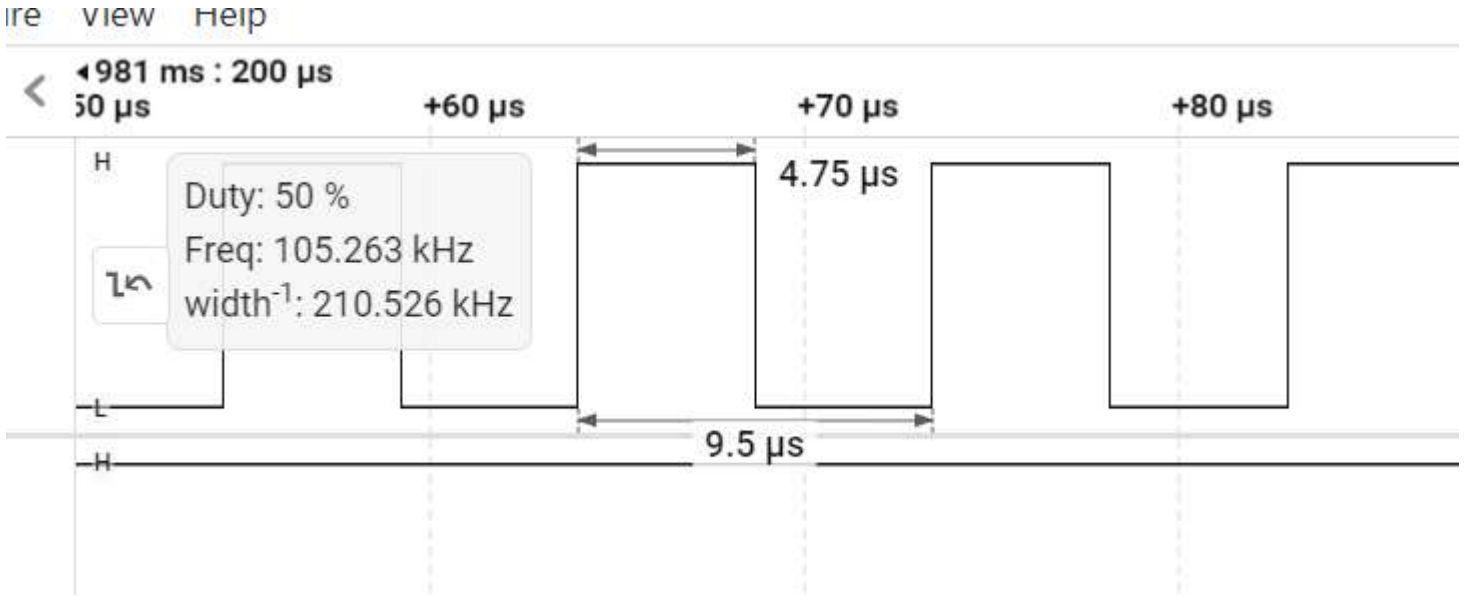






Tang Nano 9K: Clock Divider

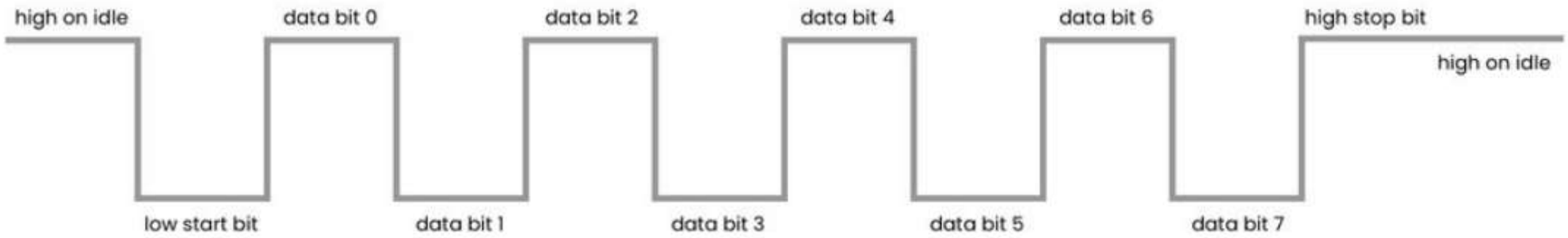
Divides 27MHz clock to 8bits=128 / 2 = 105KHz




```
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

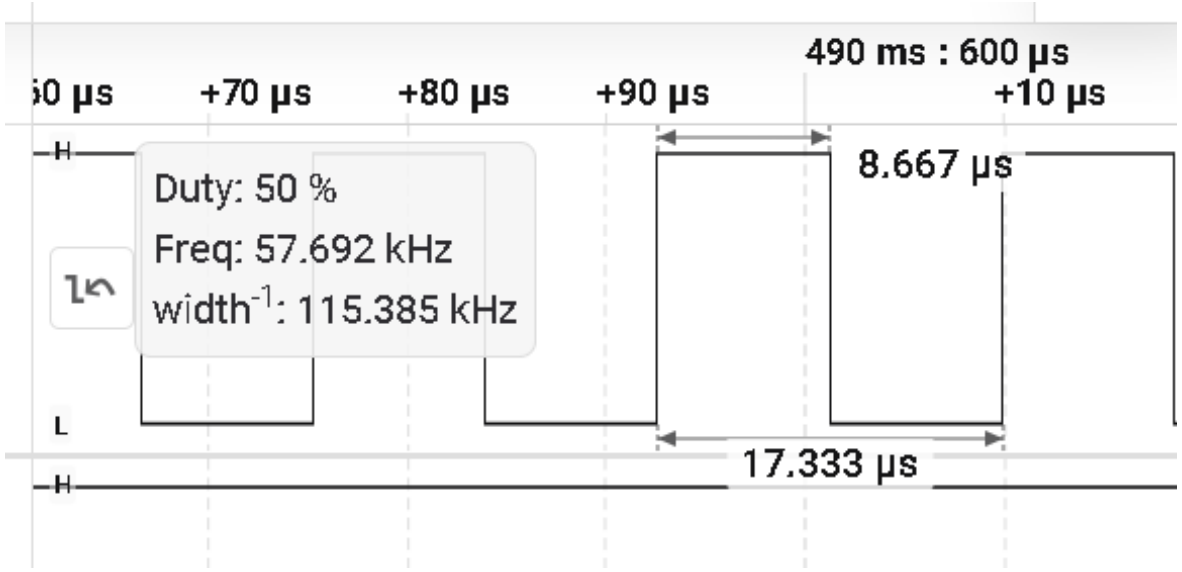
- FPGA’s are also used to test communication protocols
- You can generate any digital communication UART, I2C, CAN etc
- lets flash UART transmit example
- 8N1, 0start - 8bit data – 1stop total 10bits



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
input
   output reg baud_clk    // Baud rate clock
output
);
   // 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;
```

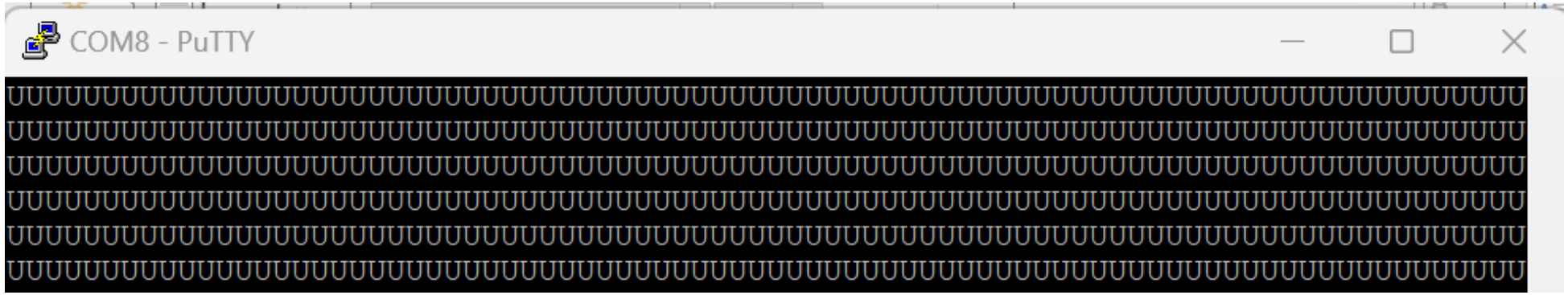
```
  always @(posedge clk) begin
    // Increment baud counter
    if (baud_counter == BAUD_DIVISOR - 1) begin
      baud_counter <= 11'b0;
      // Transmit data on the rising edge of the start bit
      case (bit_counter)
        0: begin
          tx_reg <= 0; // Data LSB
          bit_counter <= 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
  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

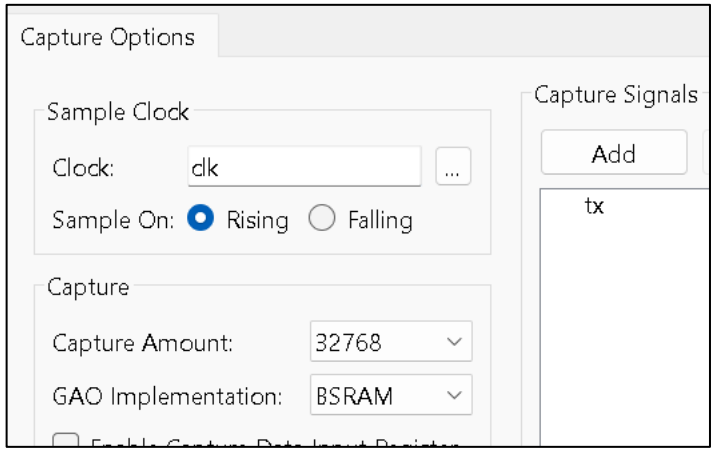
I/O Constraints

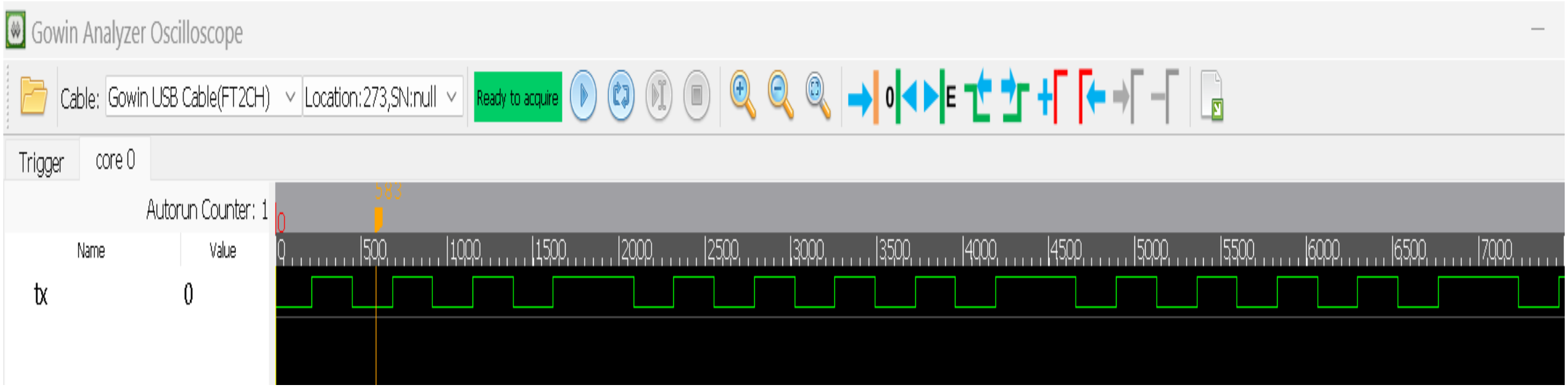
	Port	Direction	Diff Pair	Location
1	clk	input		52
2	tx	output		17



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;
```

Tang Nano: UART

- Send ‘Hello World’
- Written by ChatGPT

```
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] = "l";
    message[10] = "d";
    message[11] = "!";



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
```

```
// 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
```

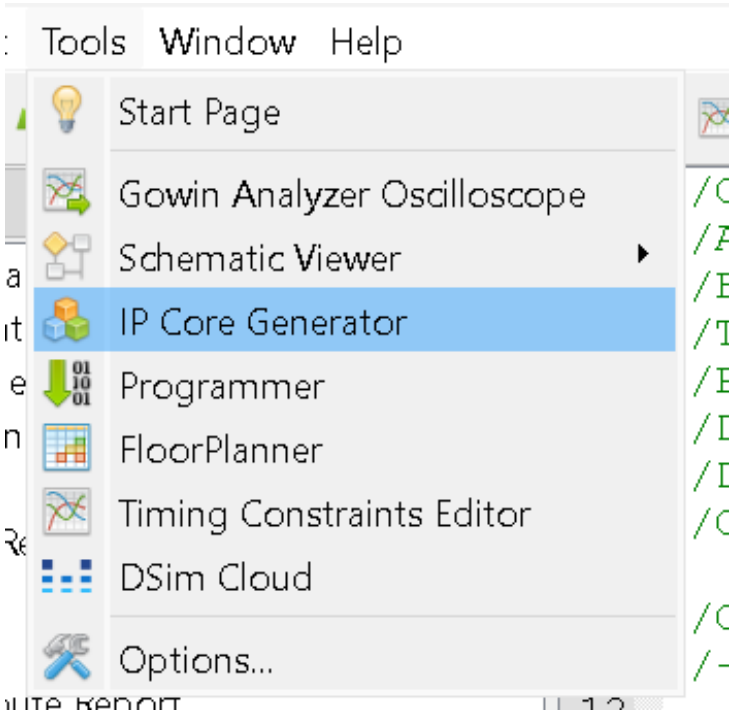
Tang Nano:Memory

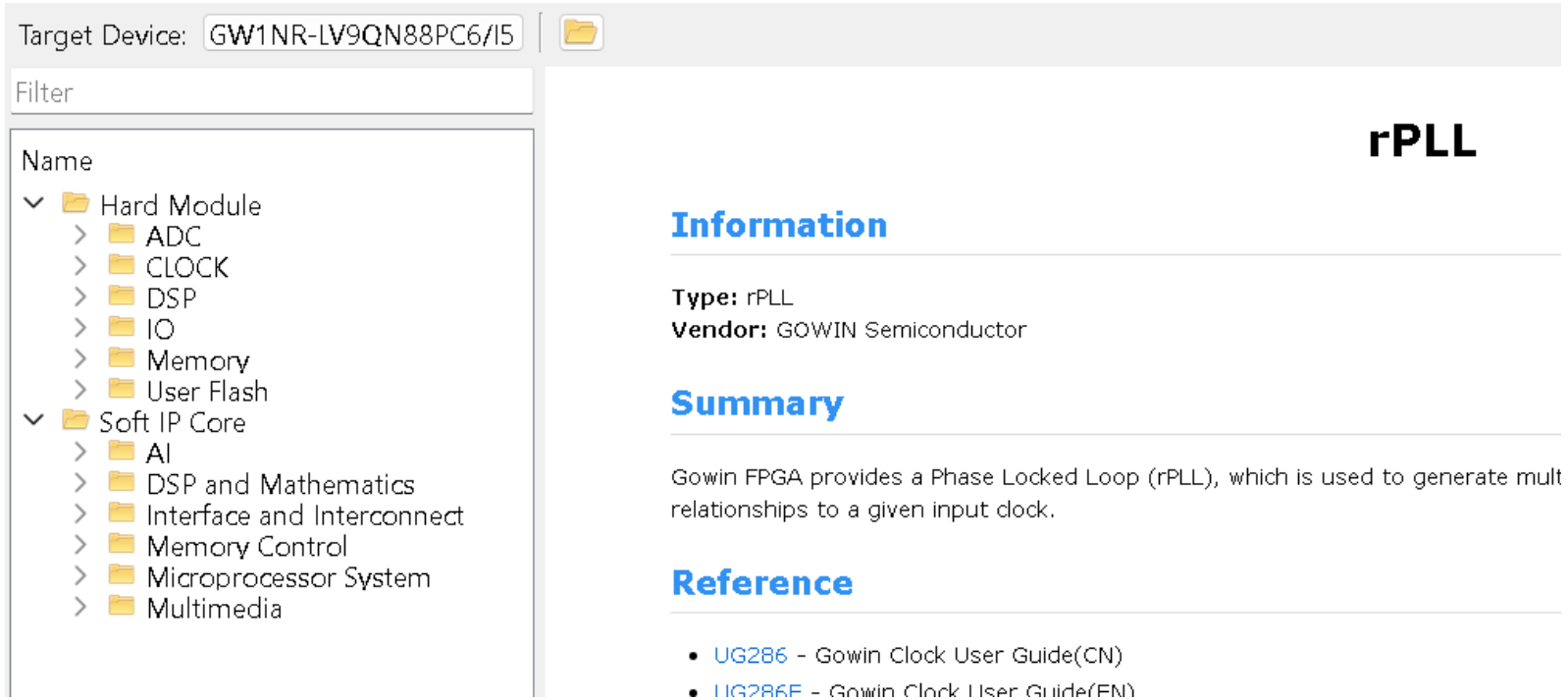
- lets write an example which reads led[n] status from memory and outputs to leds
- 64x6 bits memory defined
- Fill the memory first 6 address
- in always @() block increments Address counter with modulo 6
-
- flash into Tang Nano 9k
- Leds will turn on as described in memory

```
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.

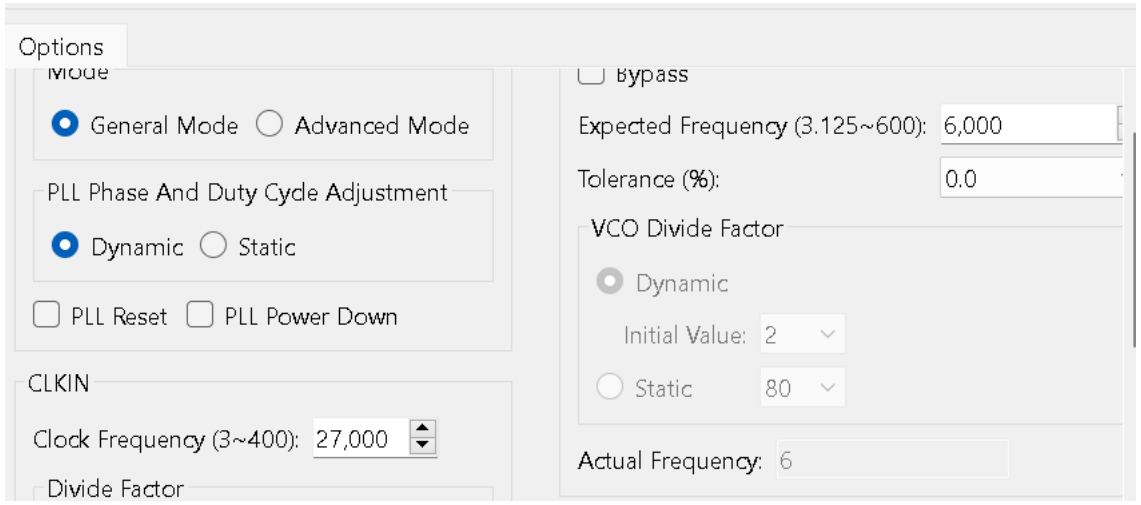




Tang Nao IP Core Example rPLL

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 genrated 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

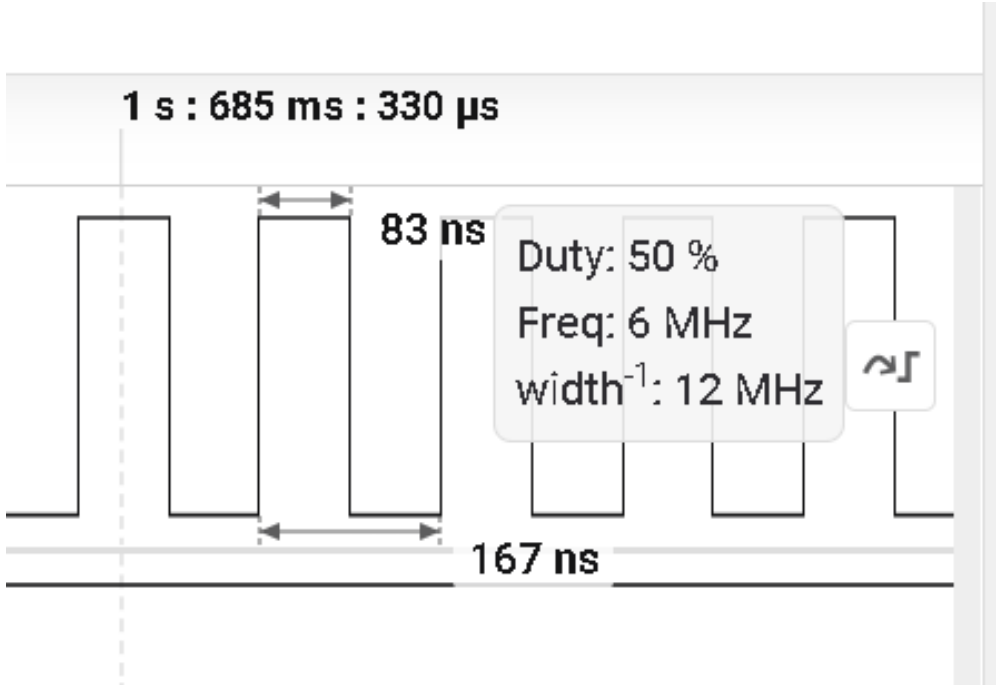
	Port	Direction	Diff Pair	Location
1	clkin	input		52
2	clkout	output		32

Tang Nao IP Core Example rPLL

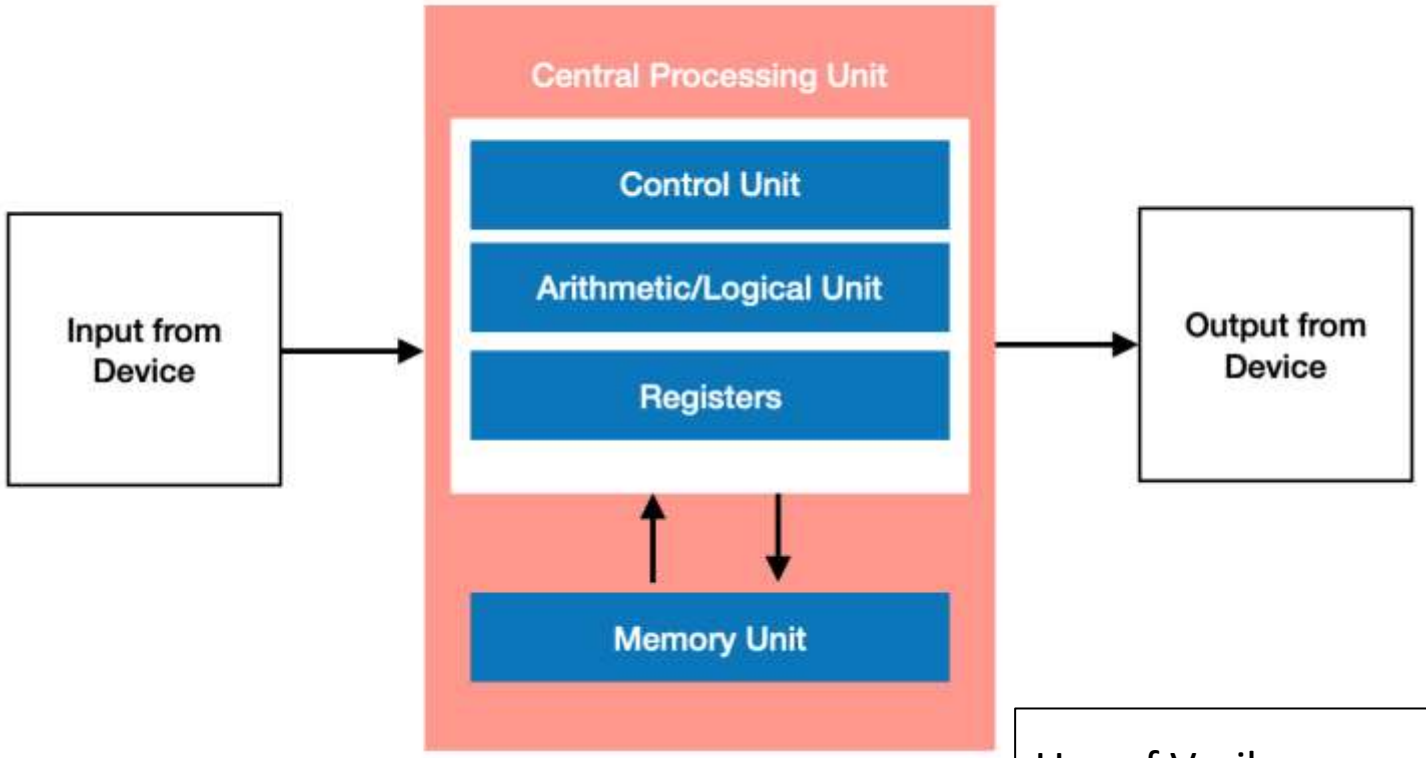
We will generate 6MHz clock at pin 32 from internal 27MHz vlock at pin 52

gowin_rpll.v automatically genrated 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)



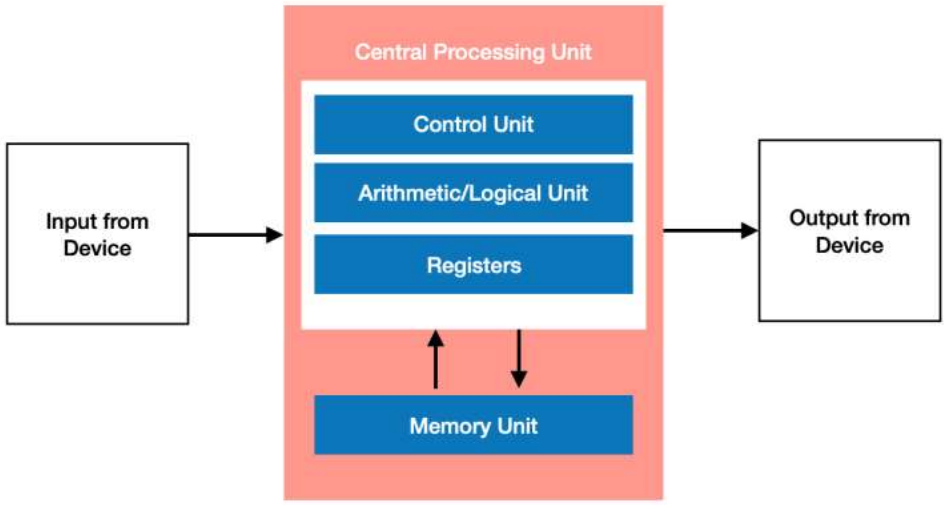
The **von Neumann** architecture—also known as the von Neumann model or Princeton architecture—is a computer architecture based on a 1945 description by **John von Neumann**, and by others, in the First Draft of a Report on the EDVAC.

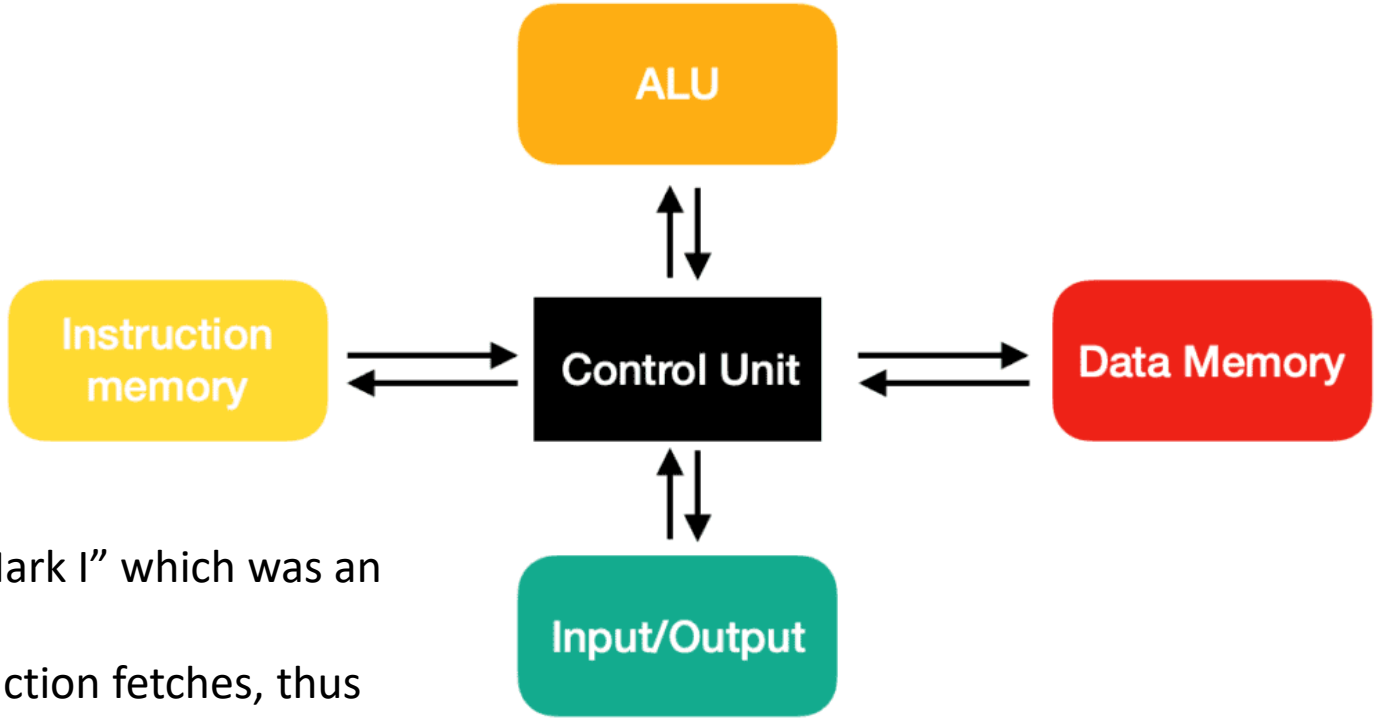
The document describes a design architecture for an electronic digital computer with these components:

Use of Verilog:

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

Microcontroller (MCU): Harvard Architecture

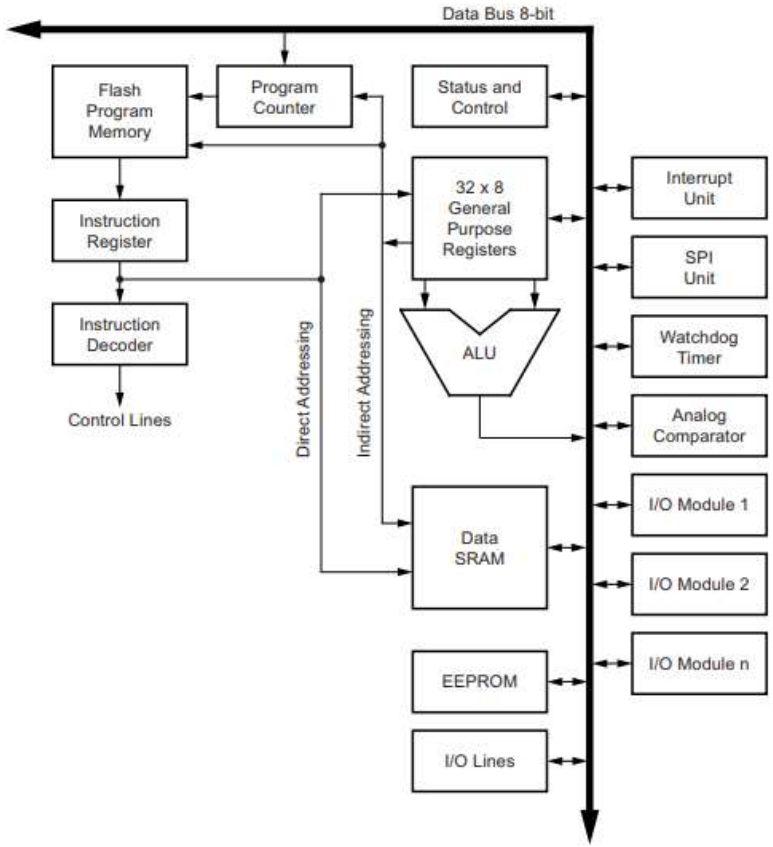




- Harvard architecture is named after the “Harvard Mark I” which was an IBM computer in the University of Harvard
- two buses, one for **data transfers** and one for instruction fetches, thus faster

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 ← K

	Syntax:	Operands:	Program Counter:
(i)	LDI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255	PC ← 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 ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← 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 ← 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 ← k

	Syntax:	Operands:	Program Counter:	Stack:
(i)	JMP k	0 ≤ k < 4M	PC ← k	Unchanged

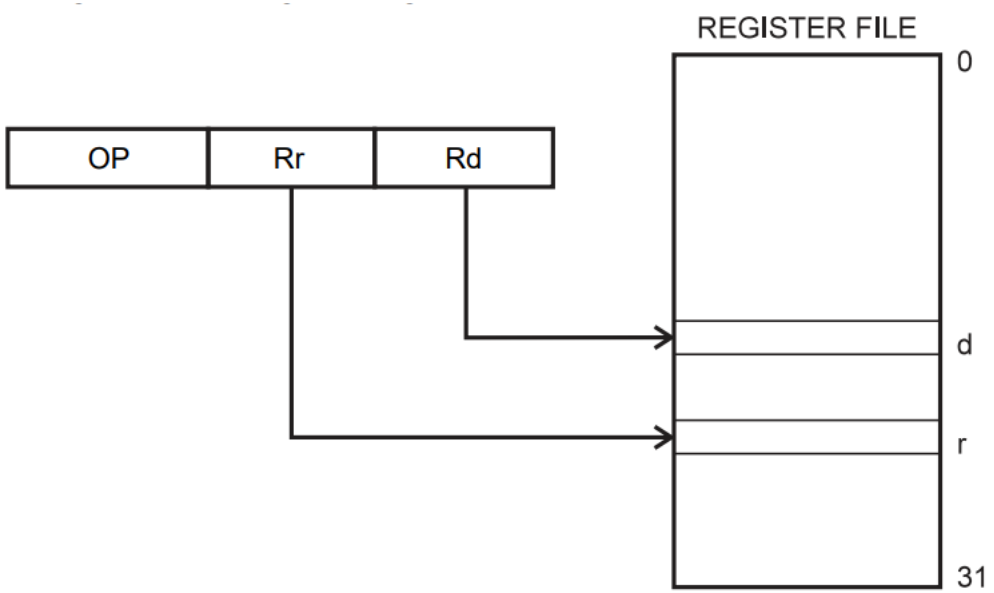
32-bit Opcode:

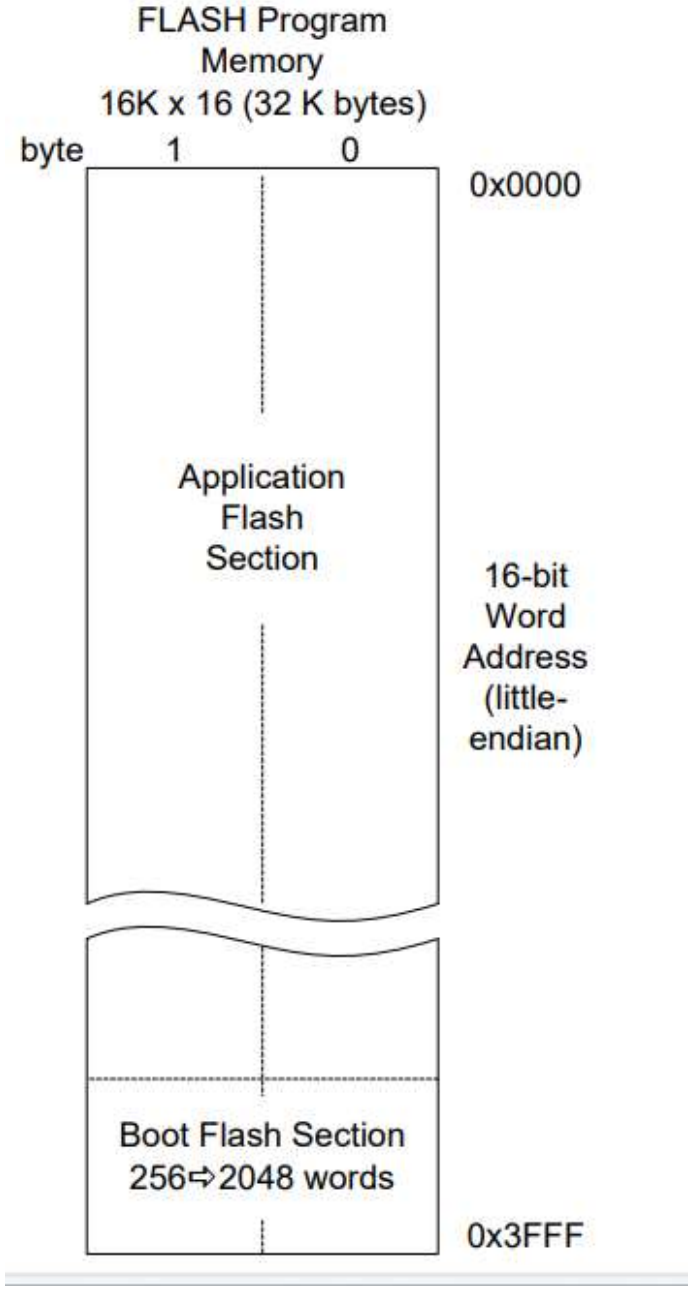
1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

MCU: Program Memory

- General Purpose Registers for fast add-sub operations

- opcodes are kept in flash memory. example 16bit address
- 0x3FFF=16K x 2 bytes = 32Kbytes of flash





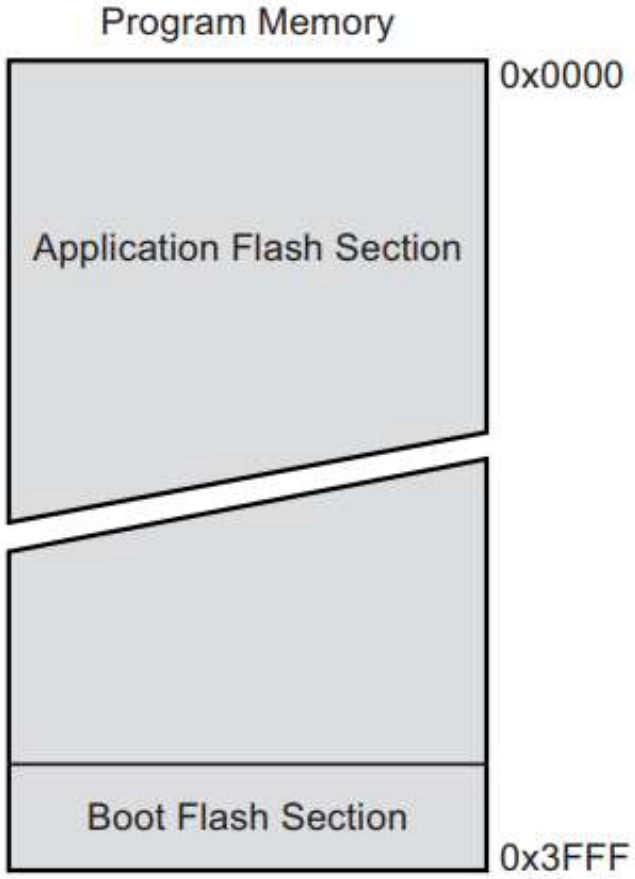
MCU: Program Memory

- 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
```

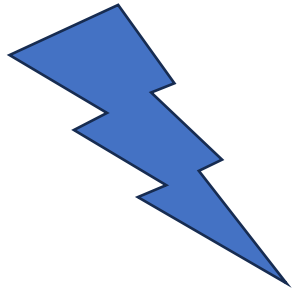
```
1110000100000010  // (1110 KKKK dddd KKKK)
1110001100010100  // (1110 KKKK dddd KKKK)
0001111100000001  // (0001 11rd  dddd rrrr)
1001000000000010  //jmp 0x02
0000000000000000
0000000000000000
0000000000000000
```




```
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,



```
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'b0001111100000001;
   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
```

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 111**1** 0000 **0001** (0001 11**r**d dddd **rrrr**)
jmp 0x02 : 16'b1001 **0000 0000 0010** (1001 **kkkk kkkk kkkk**)
nop :16'b**0000 0000 0000 0000** (**0000 0000 0000 0000**)
nop
…

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

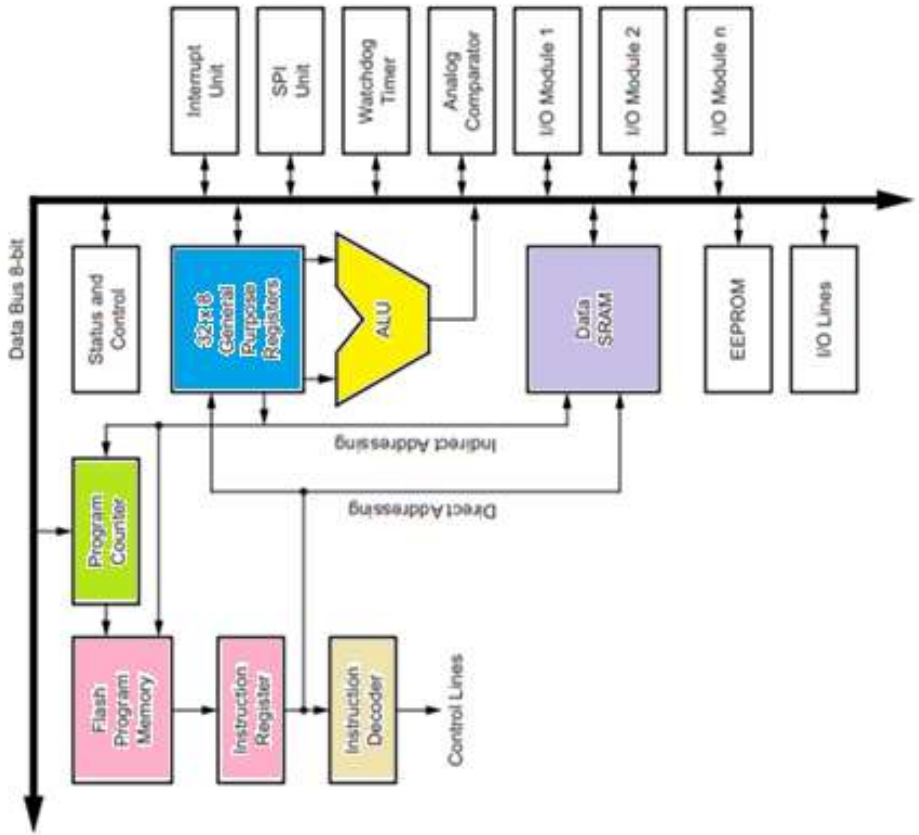
- $display() for debugging will print where we are

$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
1110000000010010
ldi PC= 2 d=  2
0001111100000001
add PC= 3 rd= 19
1001000000000010
jmp PC= 2
0001111100000001
add PC= 3 rd= 21
1001000000000010
jmp PC= 2
0001111100000001
add PC= 3 rd= 23
1001000000000010
jmp PC= 2
0001111100000001
add PC= 3 rd= 25
1001000000000010
jmp PC= 2
```

AVR vs MIPS



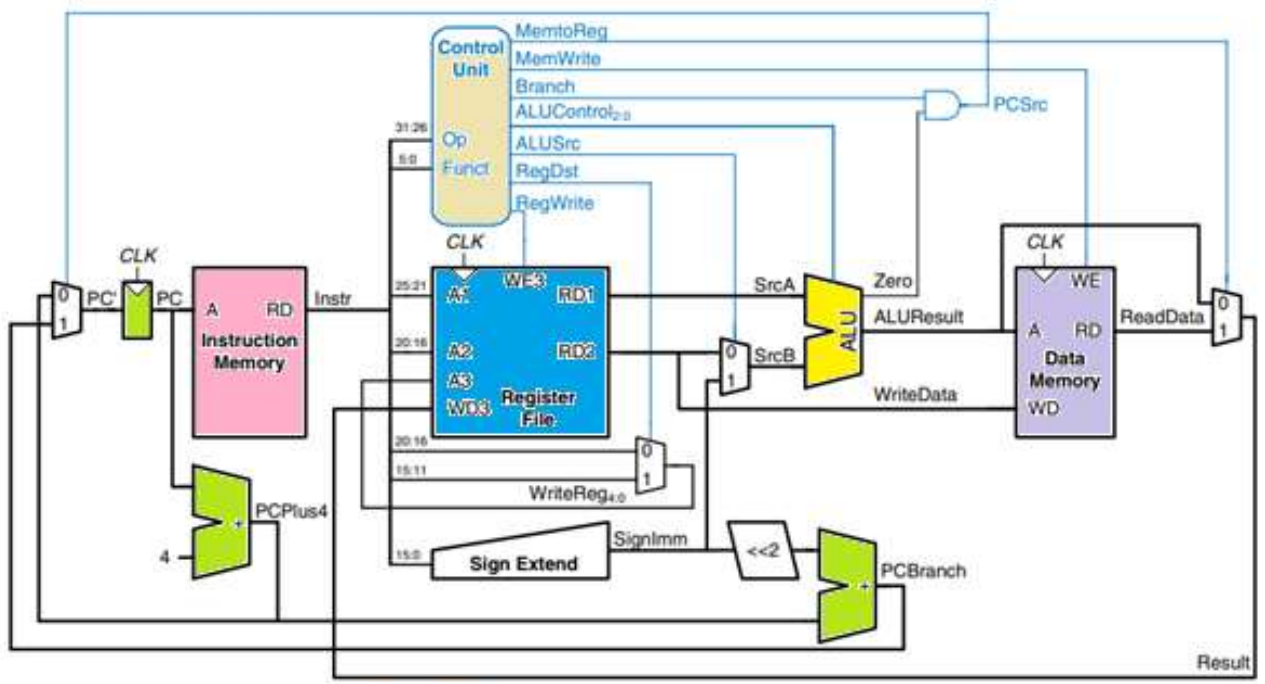
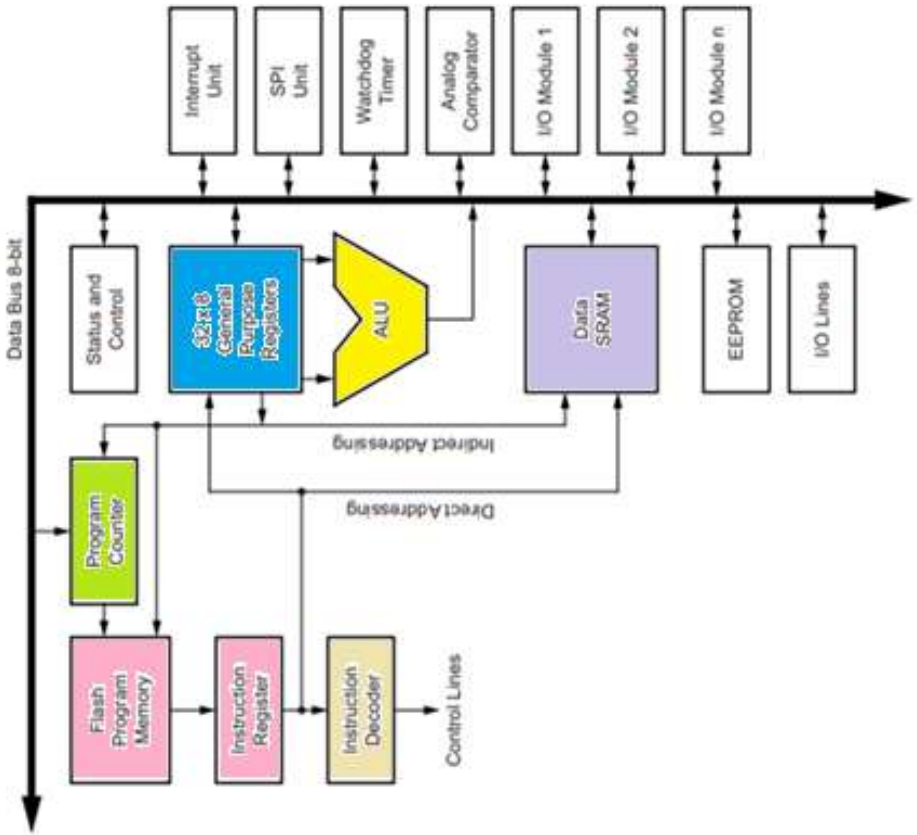
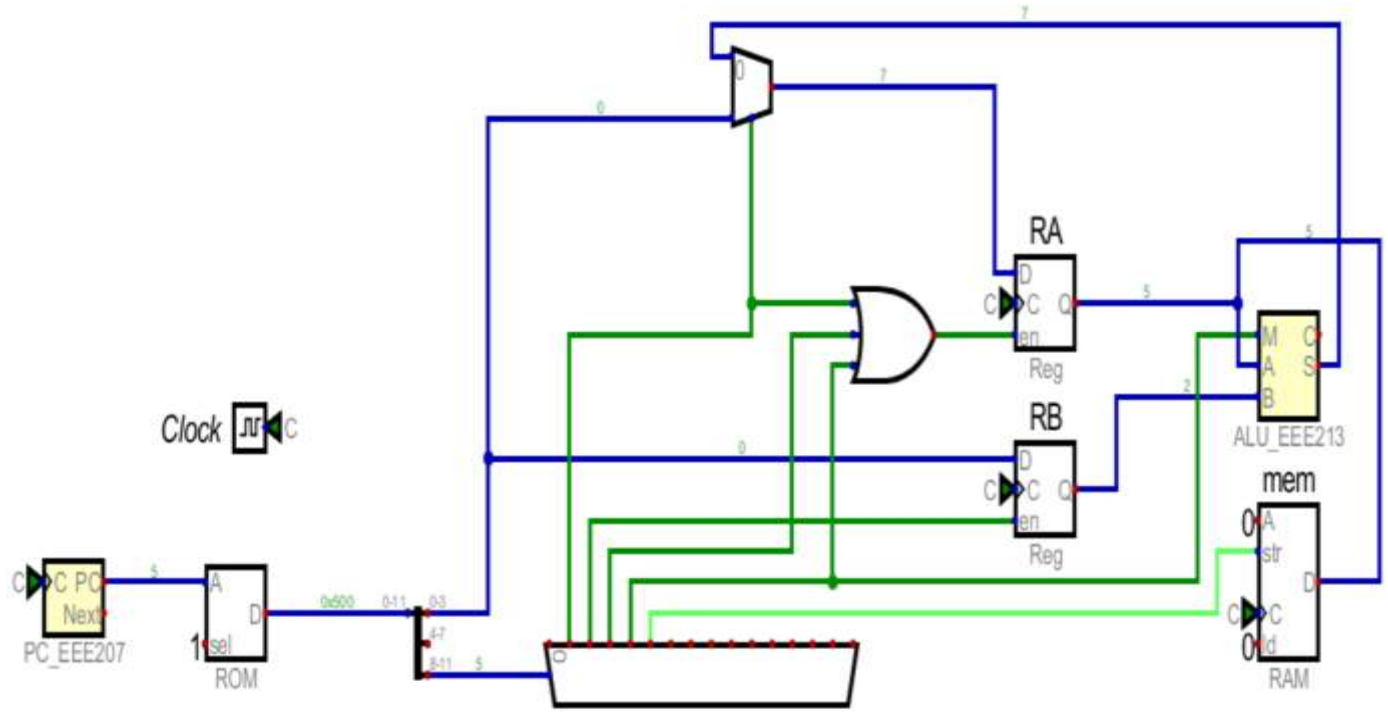


Figure 7.11 **Complete single-cycle MIPS processor**

AVR vs MyCPU


Interrupt Unit
SPI Unit
Watchdog Timer
Analog Comparator
I/O Module 1
I/O Module 2
I/O Module n
Data Bus 8-bit
Status and Control
32x8 General Purpose Registers
ALU
Data SRAM
EEPROM
I/O Lines
Indirect Addressing
Direct Addressing
Program Counter
Flash Program Memory
Instruction Register
Instruction Decoder
Control Lines


Clock
PC_EEE207
ROM
RA
RB
Reg
Reg
ALU_EEE213
mem
RAM

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
```