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

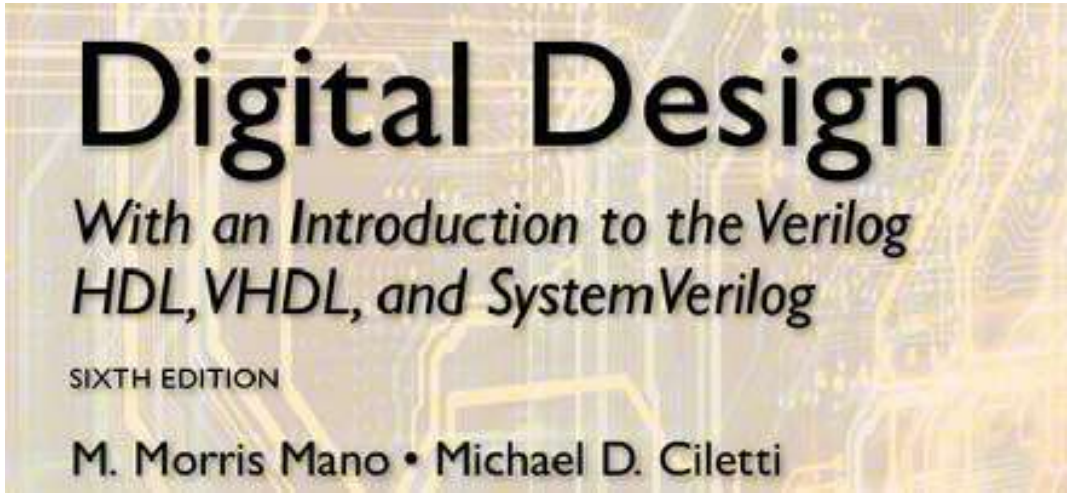
Sakarya University
Doctor of Philosophy - PhD
2009 - 2014

ABOUT

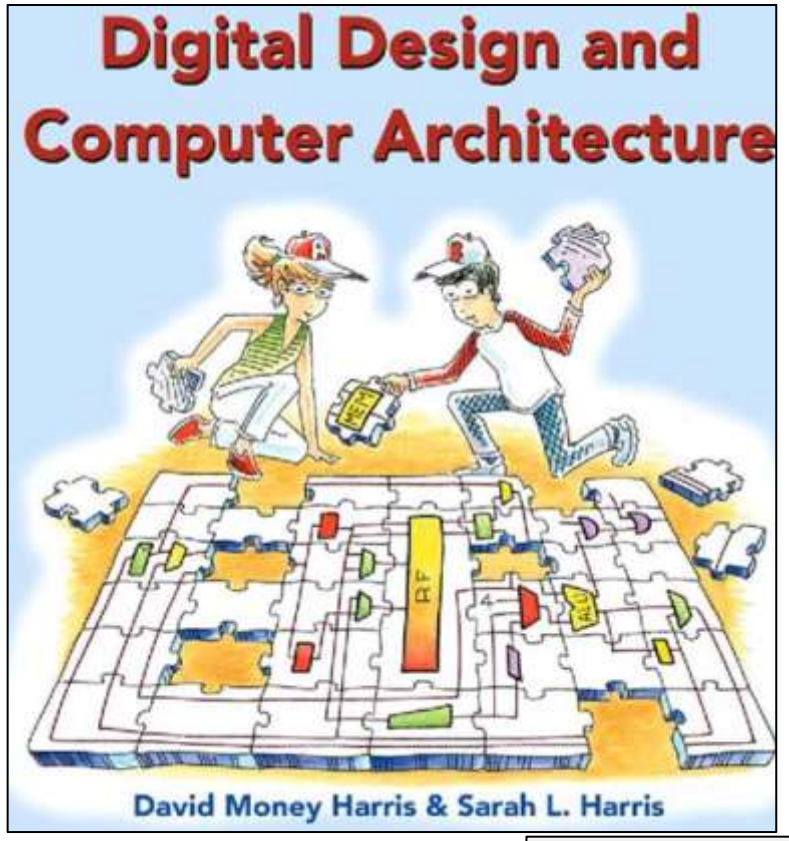
- Text Book: Digital Design, Global Edition Mano, M. Morris, Ciletti, Michael

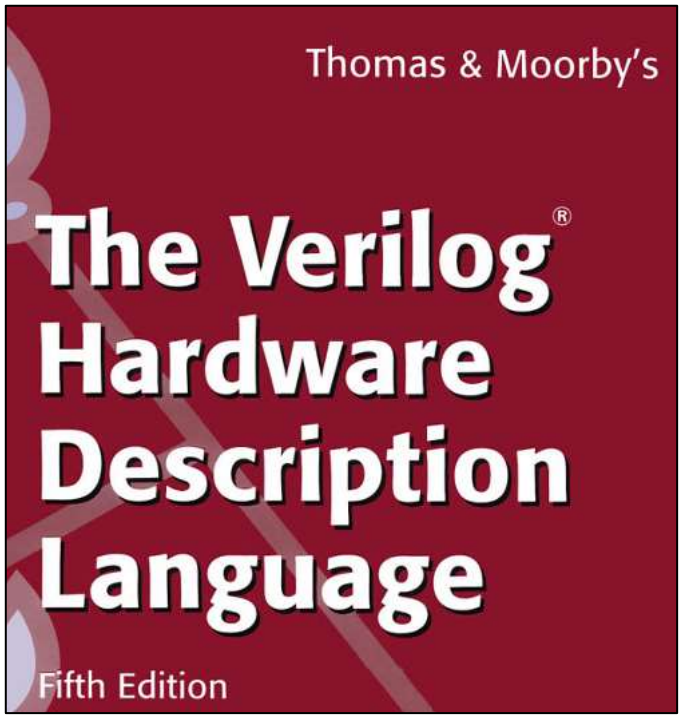
- Grades: 20% Midterm + 20% Project + 60% Final

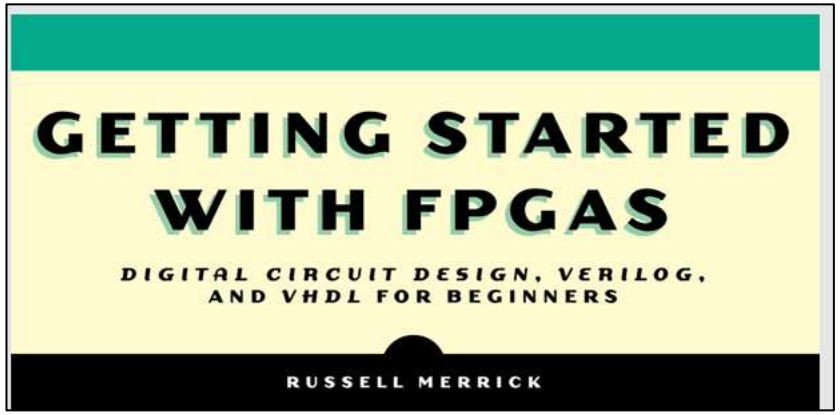
- Course Objectives: Learning Verilog Hardware Description Language for logic circuit testing, design and applications using FPGAs



Additional Resources







WHY

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

SYLLABUS

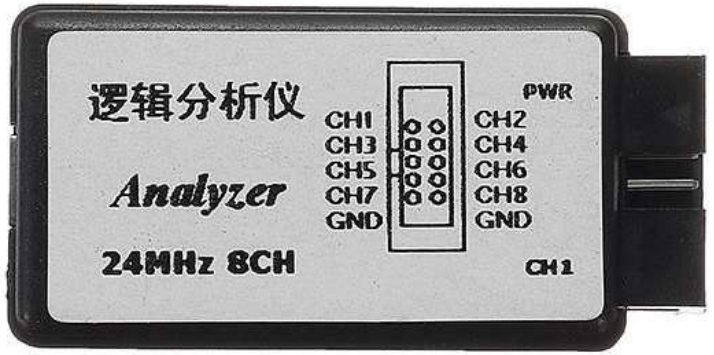
Week	Subject
1	Introduction
2	Gate Level Modeling and testbench
3	Gate Level Modeling and testbench
4	Data Flow Modeling
5	Data Flow Modeling
6	**Midterm**
7	Behavioral Modeling
8	Sequential logic circuits using Verilog
9	Sequential logic circuits using Verilog
10	Verilog and FPGA Implementation
11	Verilog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

DEVELOPMENT ENVIRONMENT

- EDA playground online Verilog IDE
- [Edit code - EDA Playground]

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

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



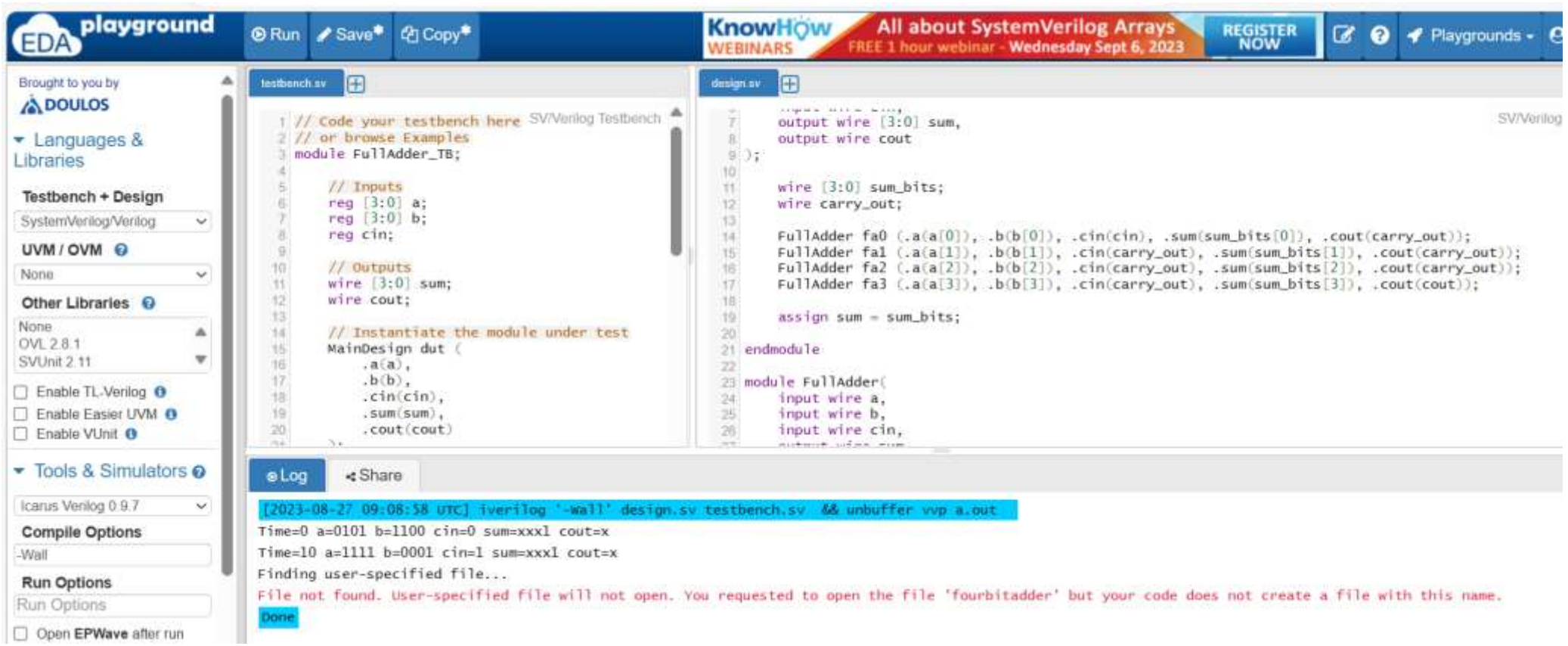
- ChatGPT - Copilot

Additional Resources

- Onur Mutlu Lectures – YouTube

EDAplayground for Verilog

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

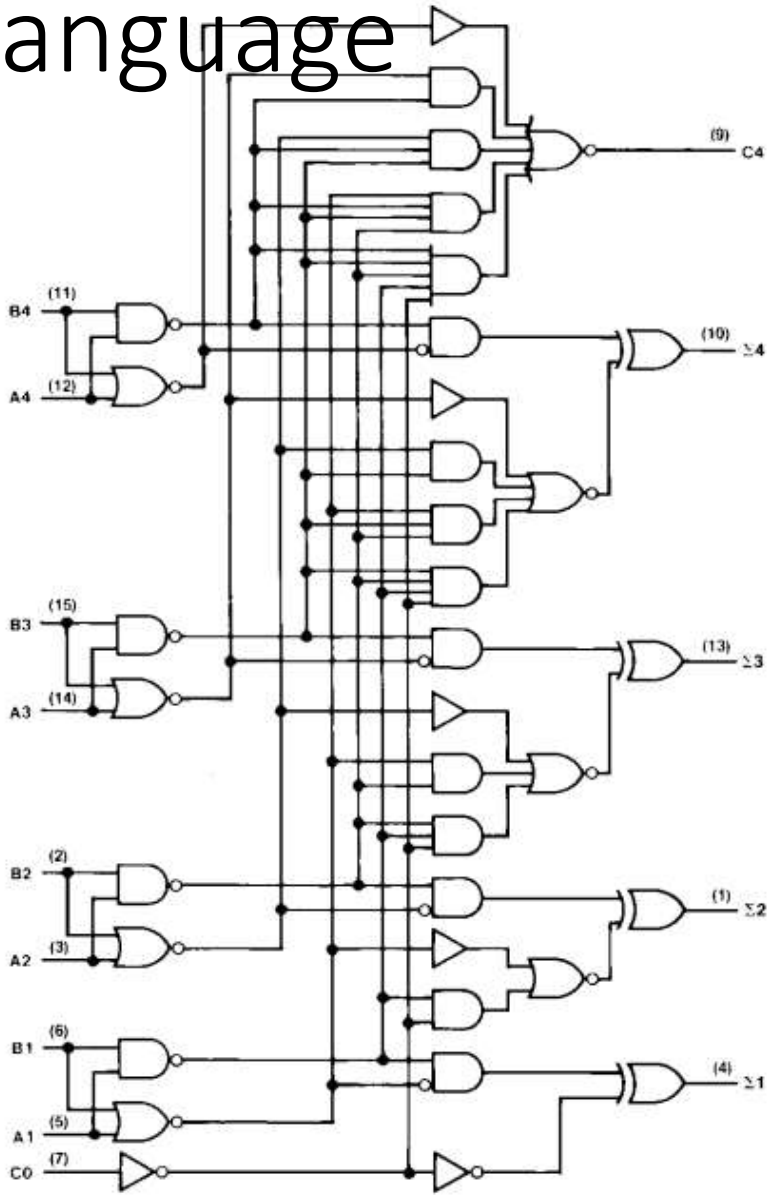


Verilog : Hardware Description Language

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?

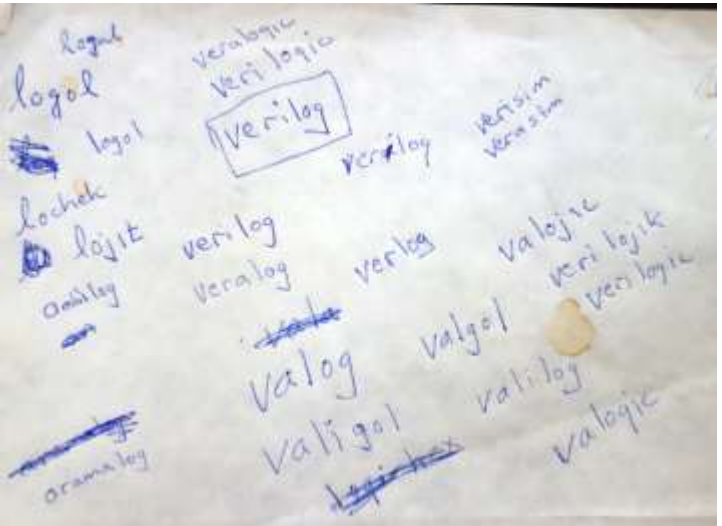
1994
Prof. Thomas Nicely reports bug in Pentium
Restoring Division
Logic error not caught until > 1M units shipped
Recall cost $450M (!!!)

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

- Verilog allows logic circuit designer to test

Verilog

PHILIP MOORBY
1953-2022



Verilog History

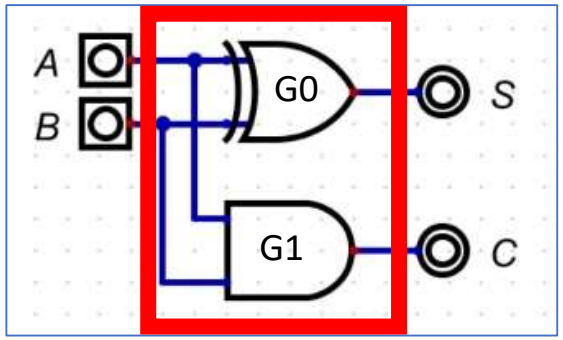
- Introduced in 1984 by Gateway Design Automation
- 1989 Cadence purchased Gateway and subsequently released Verilog to the public
- Open Verilog International (OVI) was formed to control the language specifications
- 1995 IEEE accepted OVI Verilog as a standard
- 2001 and 2005 IEEE revised standard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

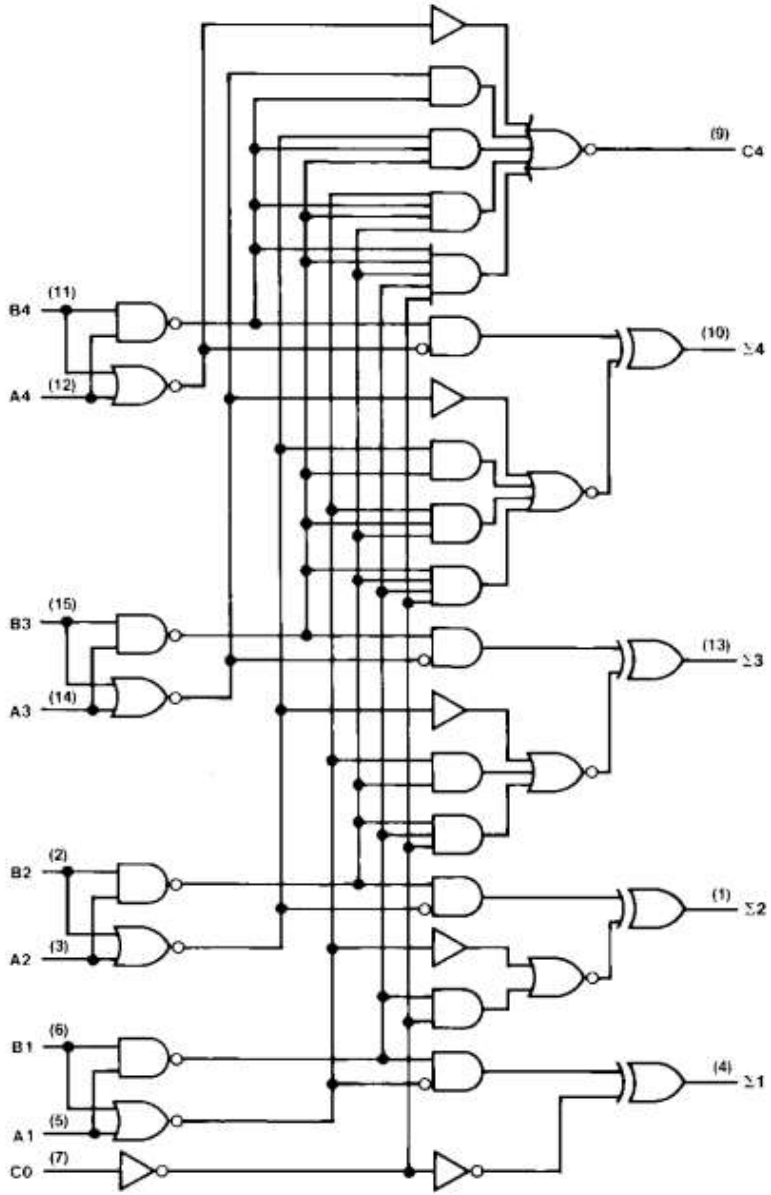
ALTERA.
MEASURABLE ADVANTAGE™

[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**

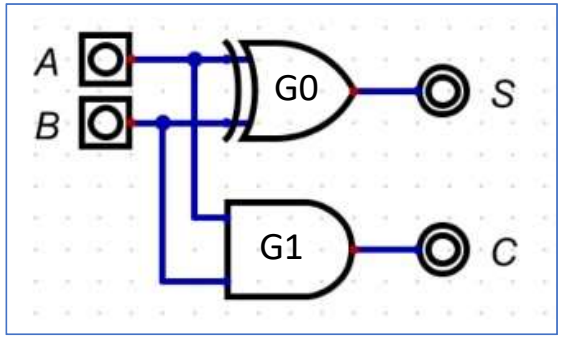
```
// half adder
module halfadder (
  output S,C,
  input A,B
);
  xor G0 (S,A,B);
  and G1 (C,A,B);
endmodule
```

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

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

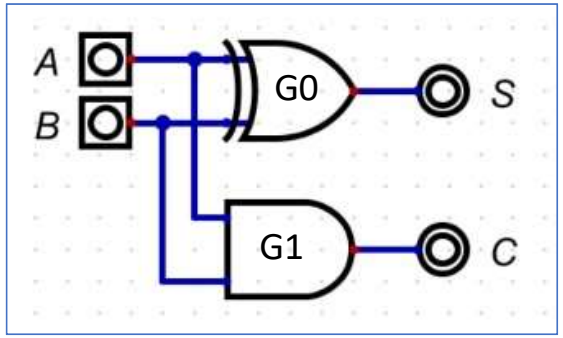
- **port direction** is either input, output
- If no **data type** is declared, it is implicitly declared as a **wire**
- **no comma at the last port**

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




```
// half adder

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

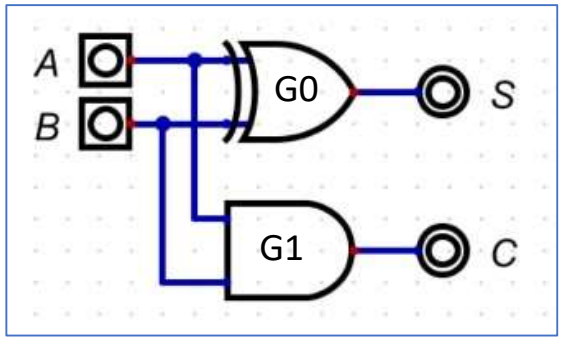
Verilog Data Types

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

5.2 Port Declarations

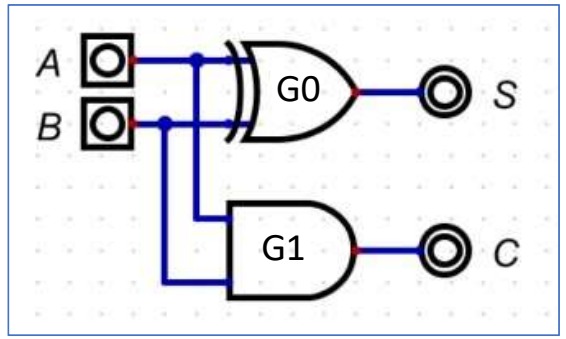
Combined Declarations *(added in Verilog-2001)*

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




```
// half adder

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

Verilog Primitives



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

8.0 Primitive Instances

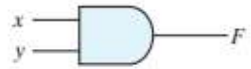
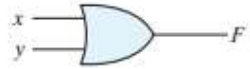
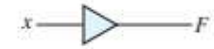
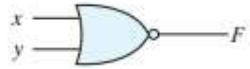
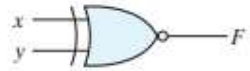
gate_type (*drive_strength*) #(*delay*) *instance_name*
[*instance_array_range*] (*terminal*, *terminal*, ...) ;


```
// half adder

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

Verilog Primitives

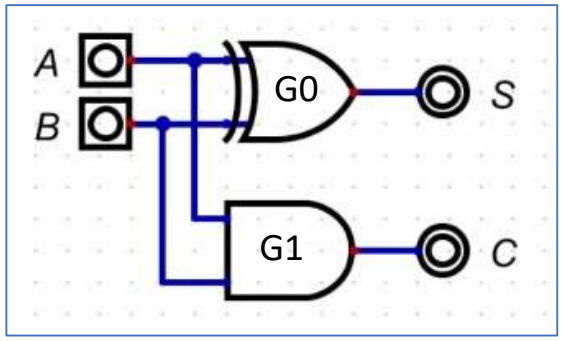
- Verilog has built-in **primitives** to define **basic gates**
- **not, and, nand, or, nor, xor, xnor**

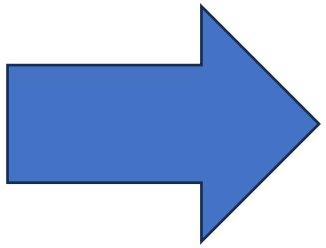
Gate Primitives		Terminal Order and Quantity
`and`	`nand`	(1–output, 1-or-more–inputs)
`or`	`nor`	
`xor`	`xnor`	

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	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





```
// half adder

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

Gate-Level Modeling

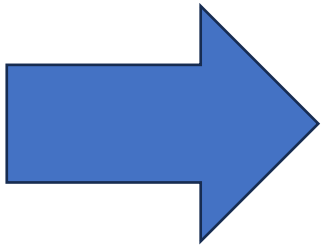
- **Gate-level** models describe how a circuit is composed of other interconnected elements, such as logic gates or functional blocks.

```
// half adder

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

Verilog 1995-**2001** Syntax Changes

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

```
// Verilog 1995

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

  xor G0 (S,A,B);
  and G1 (C,A,B);
endmodule
```



```
// Verilog 2001

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

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

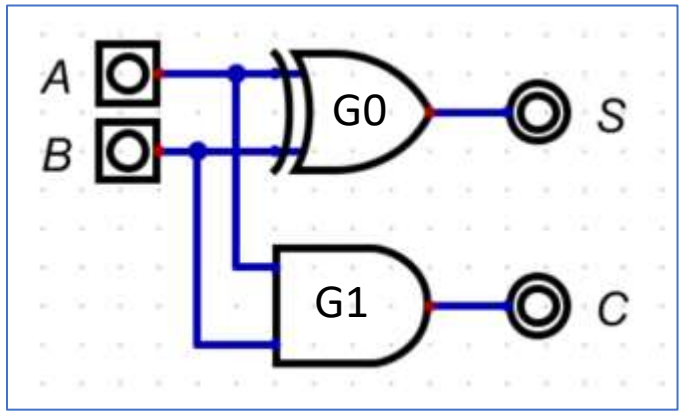
```
module module_name
    #(parameter_declaration, parameter_declaration,... )
    (port_declaration port_name, port_name,...,
     port_declaration port_name, port_name,...);
    module items
endmodule
```

Old Style Port List

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

Verilog: Half Adder

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



```
// half adder

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

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

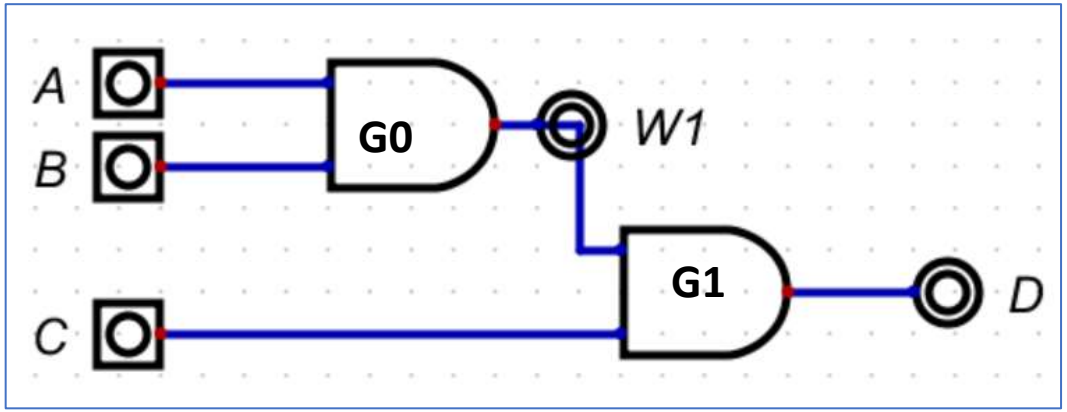
Verilog Net Data Types

- 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
```

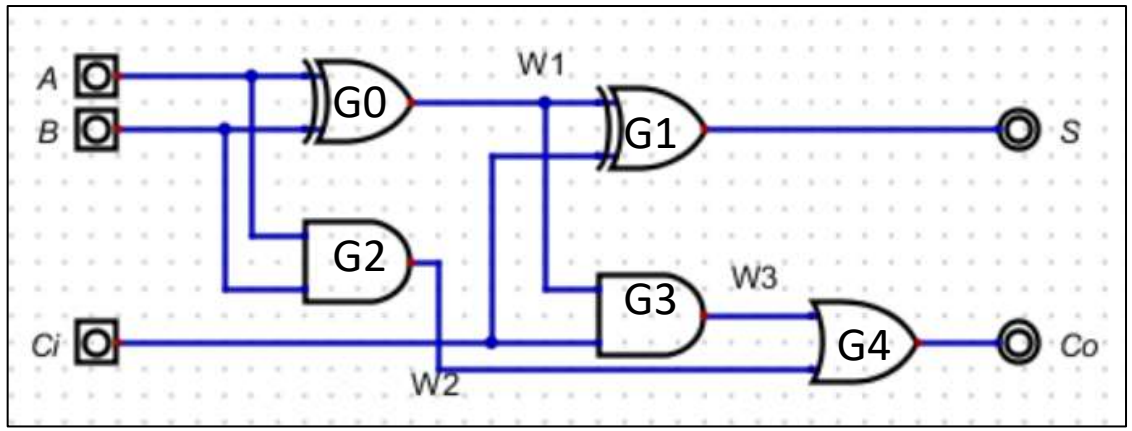
Net Data Type

Net data types connect structural components together.



Complex Combinatorial Circuits: Full Adder

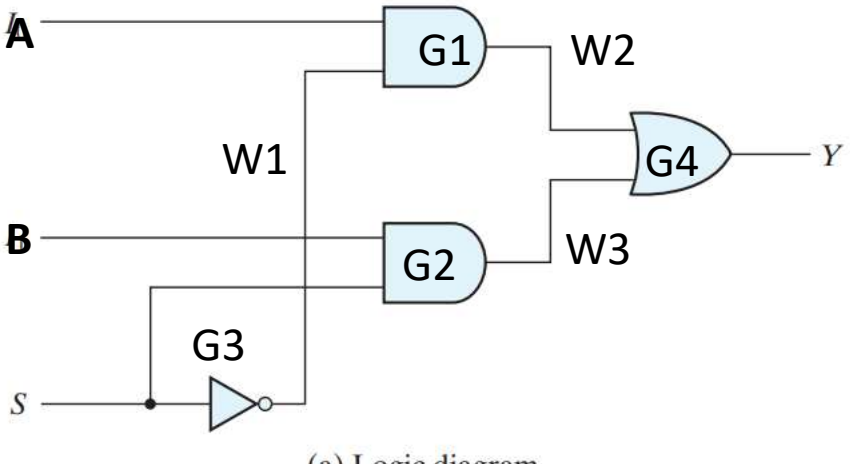
- Using Gate-level modeling we can define any **combinatorial logic** circuit
- name gates G0, G1, etc.
- name wires as W1, W2, etc. for the intermediate connection points
- Sunday puzzle sudoku ☺




```
//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
```

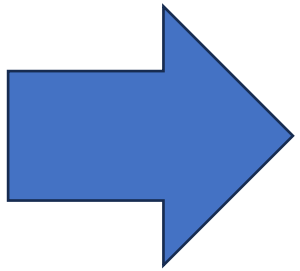
Complex Combinatorial Circuits: MUX

- 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

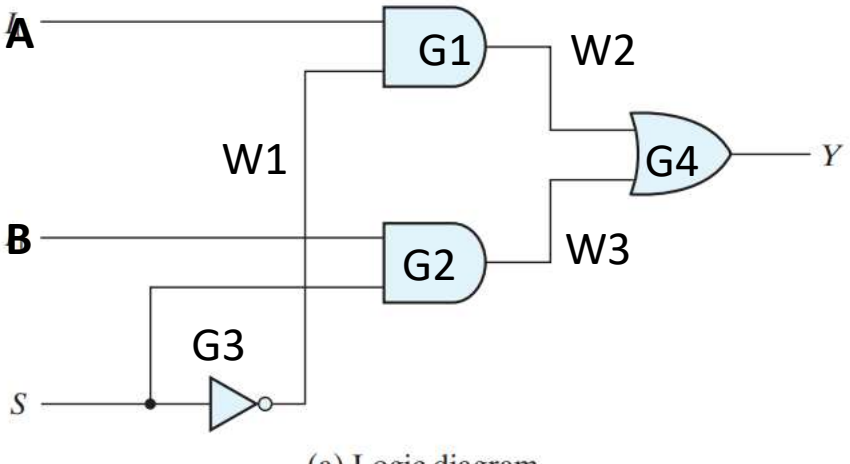
FIGURE 4.24
Two-to-one-line multiplexer



```
//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** ☺



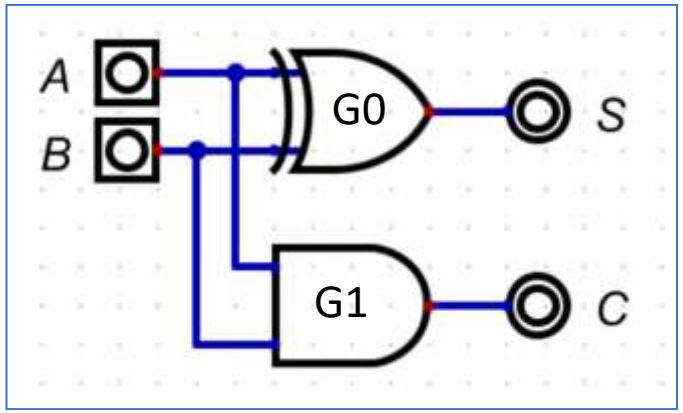
(a) Logic diagram

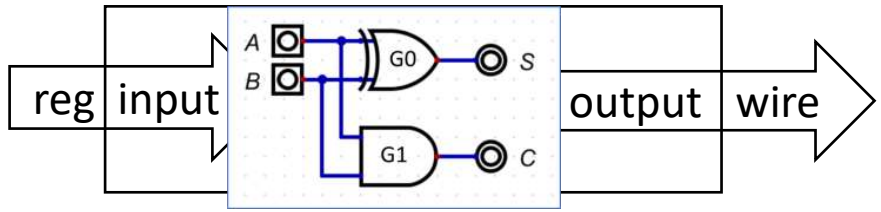
FIGURE 4.24
Two-to-one-line multiplexer

```
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**.





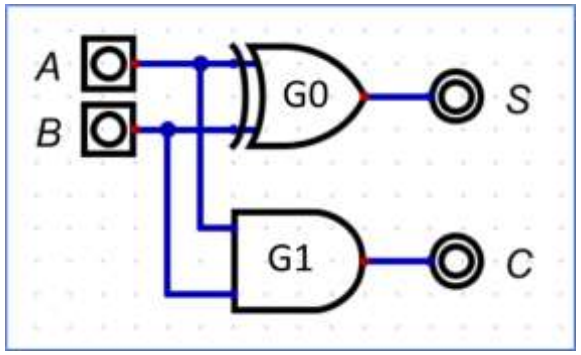
halfadder_tb

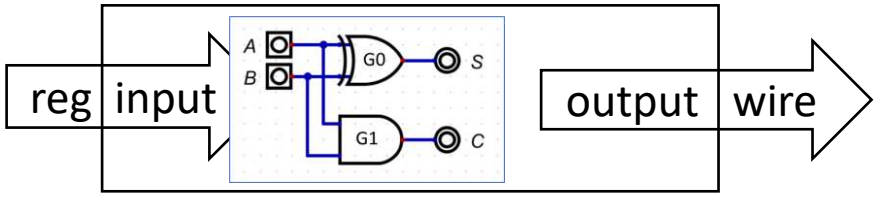
```
//half adder
module halfadder(
  output S,C,
  input A,B
);
  xor G0 (S,A,B);
  and G1 (C,A,B);
endmodule
```

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

Verilog Test Bench: input output

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





halfadder_tb

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

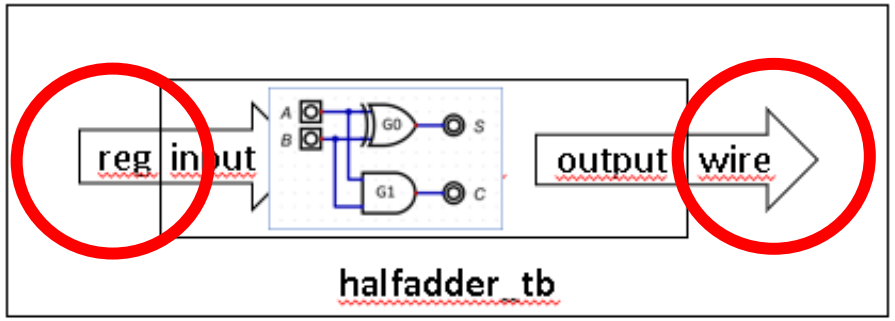
Verilog Variable Data Types

- *variable_type* is one of the following:

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

- *net_type* is one of the following keywords:

wire	interconnecting wire; CMOS resolution




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

Verilog Test Bench: Instantiation

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

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

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

Verilog Test Bench: Instantiate

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

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

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

Verilog Test Bench: initial

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

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

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

Verilog Test Bench: waveform

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


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

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

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

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

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

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

17.1 Text Output System Tasks

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

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

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

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

Text Formatting Codes

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

`%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.

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

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

$display(), $monitor()

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

$monitorh ("text_with_format_specifiers", list_of_arguments) ;

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

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

%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).

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

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

Verilog Test Bench: assignment

10.3 Procedural Assignment Statements

variable = expression;

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

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

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

Verilog Test Bench: $finish

$finish(*n*) ;

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

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

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

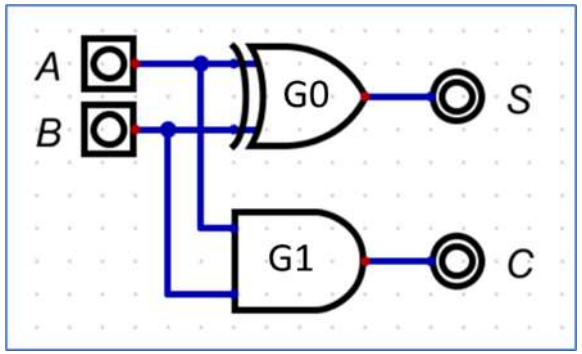
```
//testbench
`timescale 1 ns / 1 ns
module testbench;
  reg  a,b;  //reg for inputs
  wire s,c;    //wire for outputs
  //instantiation
  halfadder G0 (.S(s),.C(c),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cs");
    $monitor("%b%b:%b%b",a,b,c,s);
    #1 a=0;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

Verilog Test Bench: review results

Table 4.3
Half Adder

x	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

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

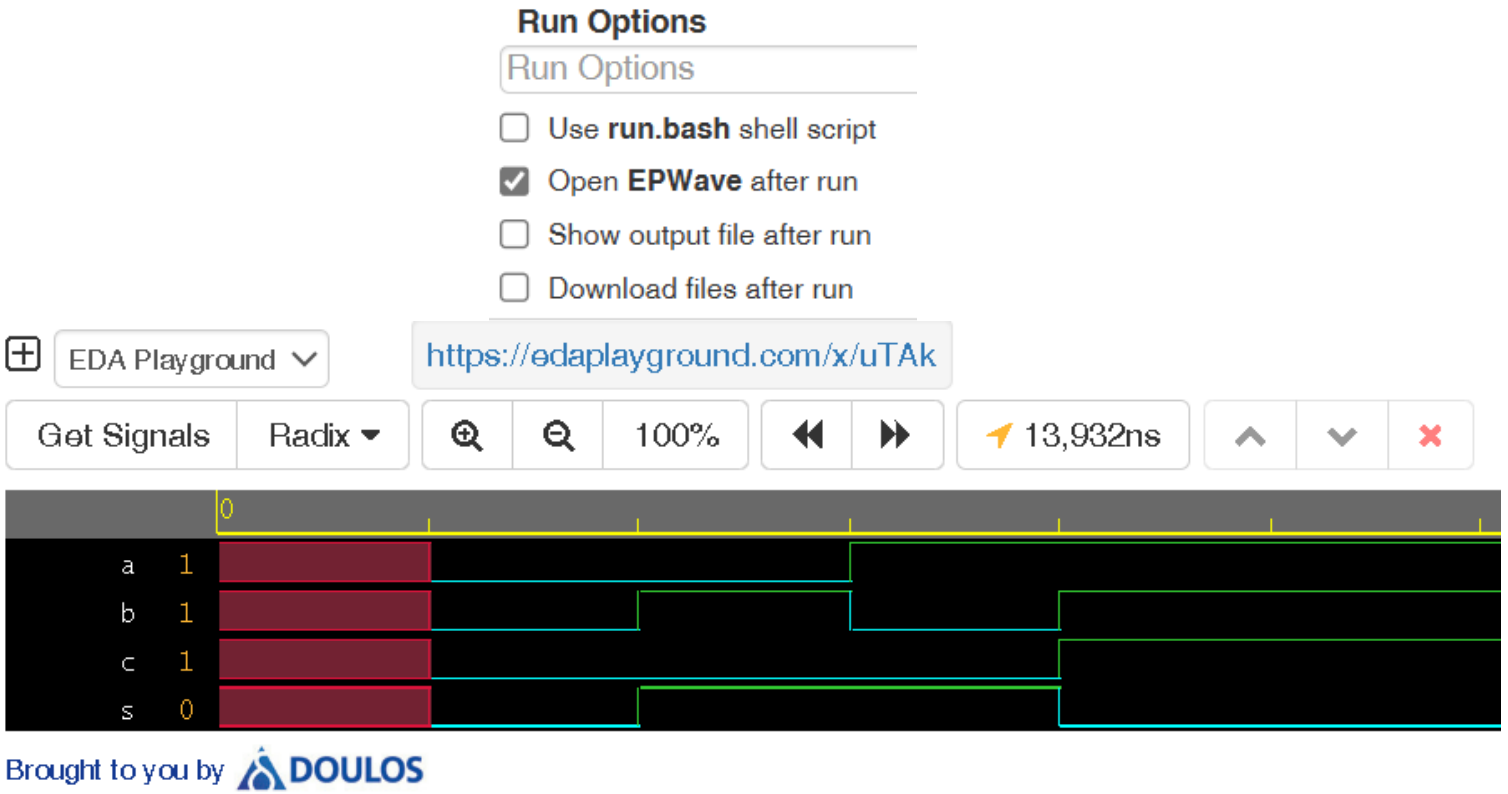



```
//half adder
module halfadder(
  output S,C,
  input A,B
);
  xor G0 (S,A,B);
  and G1 (C,A,B);
endmodule
```

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

Verilog Test Bench: **waveform**

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

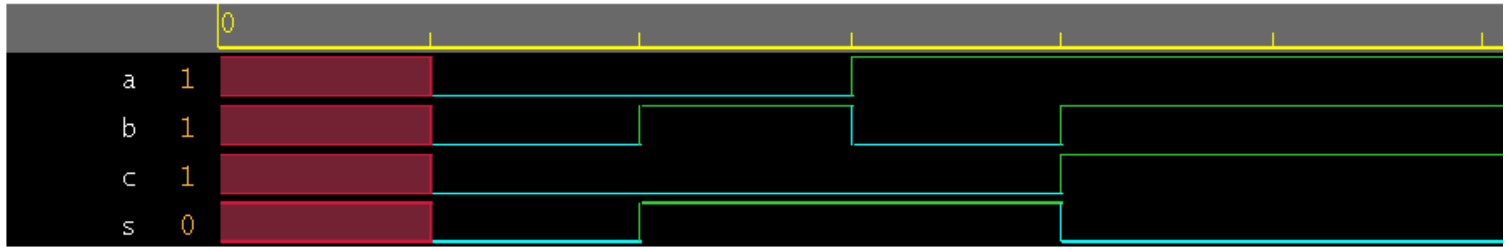



```
//testbench
`timescale 1 ns / 1 ns
module testbench;
  reg  a,b;  //reg for inputs
  wire s,c;    //wire for outputs
  //instantiation
  halfadder G0 (.S(s),.C(c),.A(a),.B(b));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("ab:cs");
    $monitor("%b%b:%b%b",a,b,c,s);
    #1 a=0;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

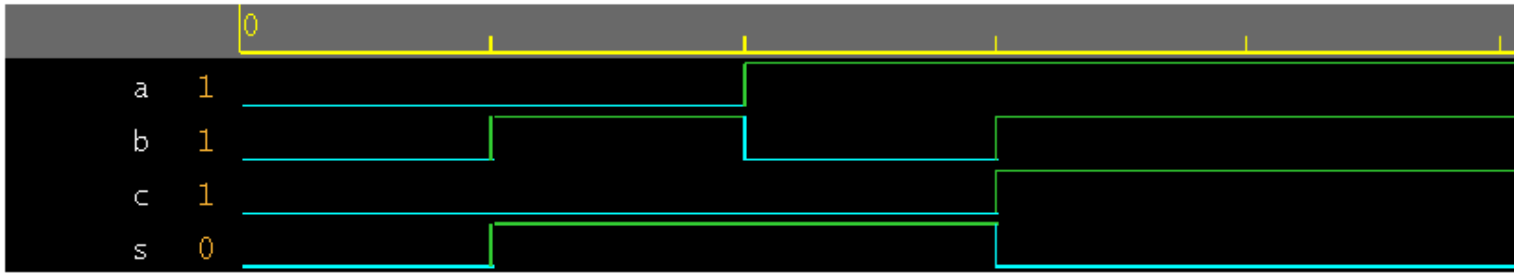
Verilog Test Bench: **waveform**

- Logic values 1,0,X. x means unknown

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




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



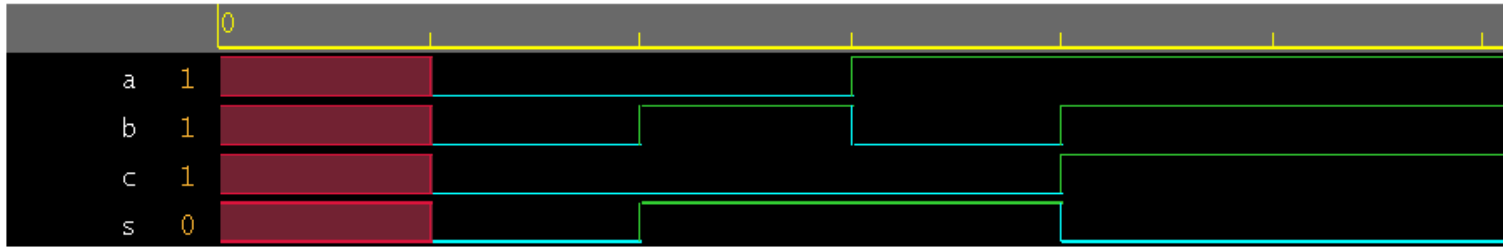

```
$monitor("%b%b:%b%b",a,b,c,s);
    a=0; b=0;
   #1 a=0; b=1;
```

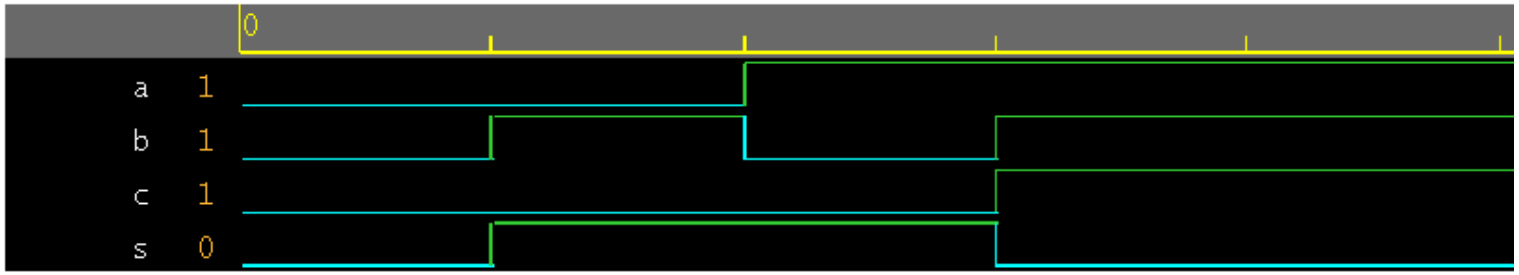
4.8 Logic Values

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

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

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



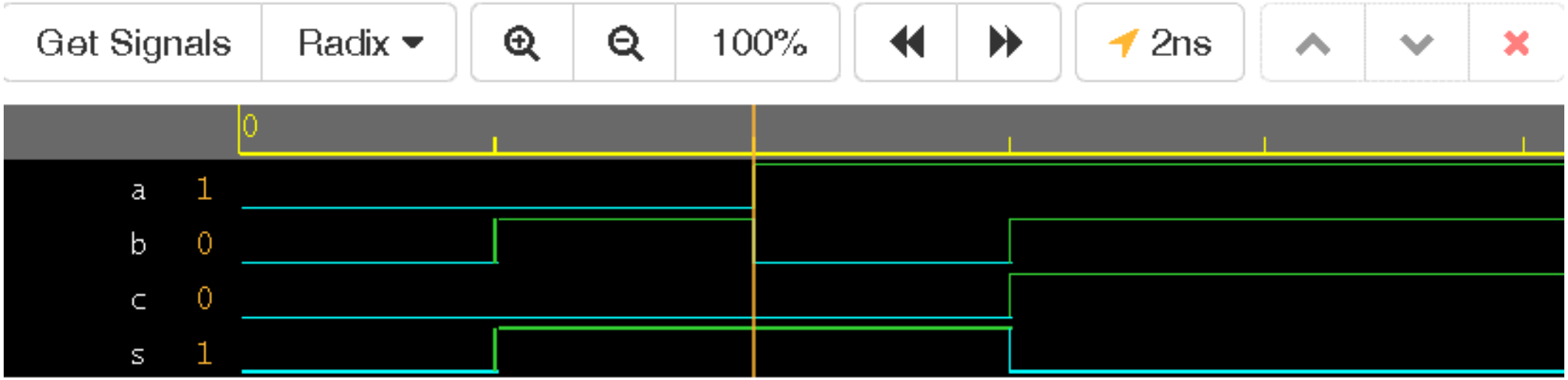



```
$monitor("%b%b:%b%b",a,b,c,s);
   #1 a=0; b=0;
   #1 a=0; b=1;
```

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

Waveform: timescale

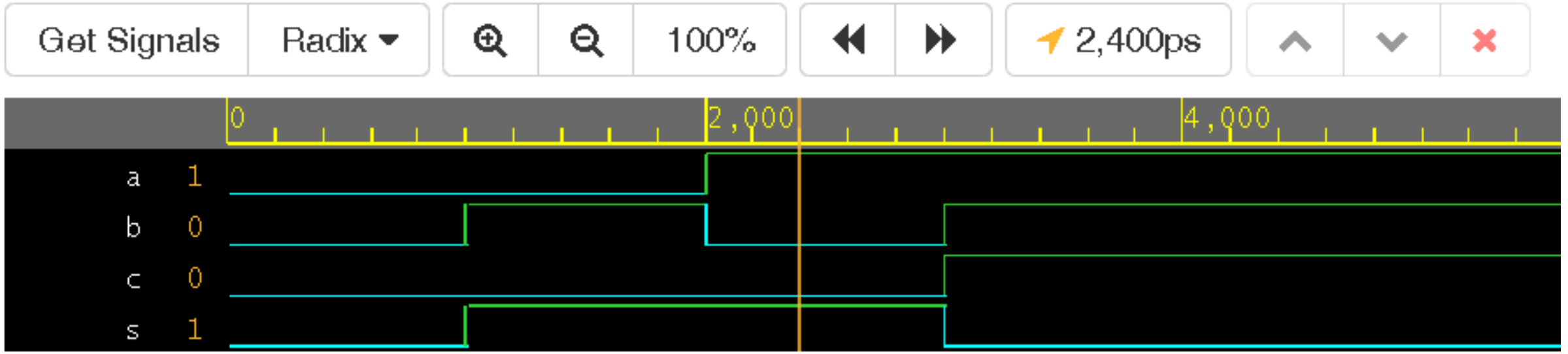
notice that values at 2ns are values on the right means values for time > 2ns



Waveform: timescale precision

We can decrease the precision to analyze in detail

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



Verilog Test Bench: iverilog

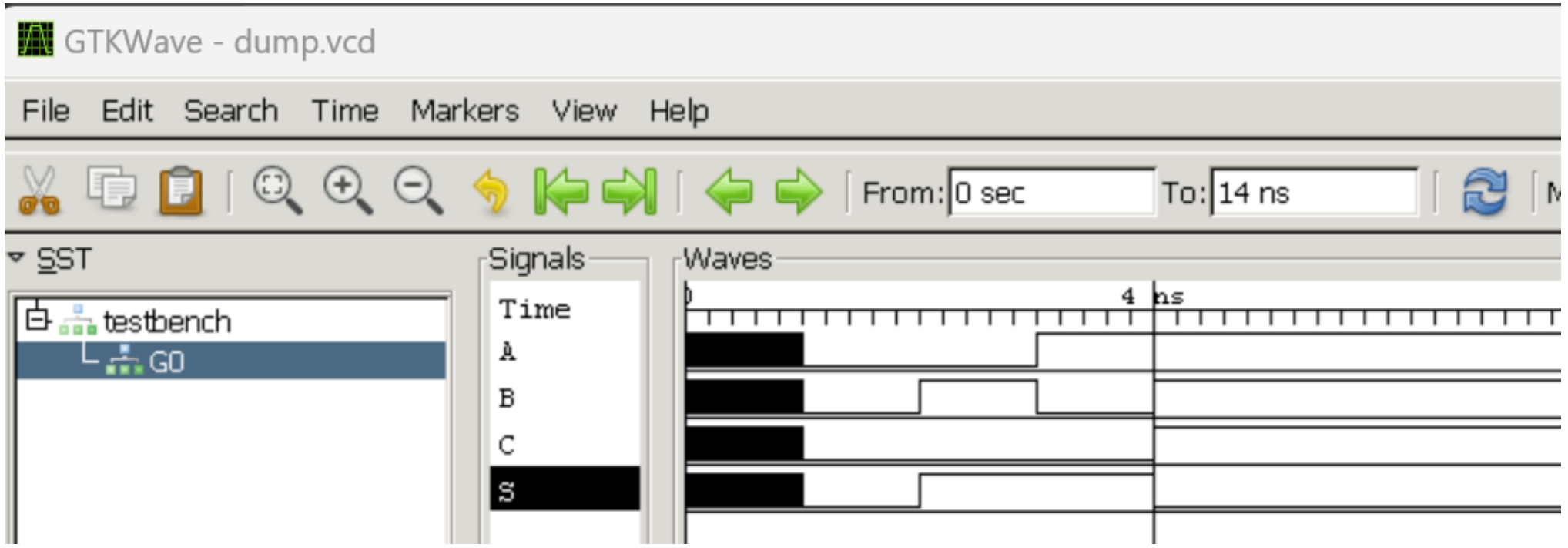
- we can run simulation in local computer using **iverilog** and **gtkwave**

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

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

Verilog Test Bench: gtkwave

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

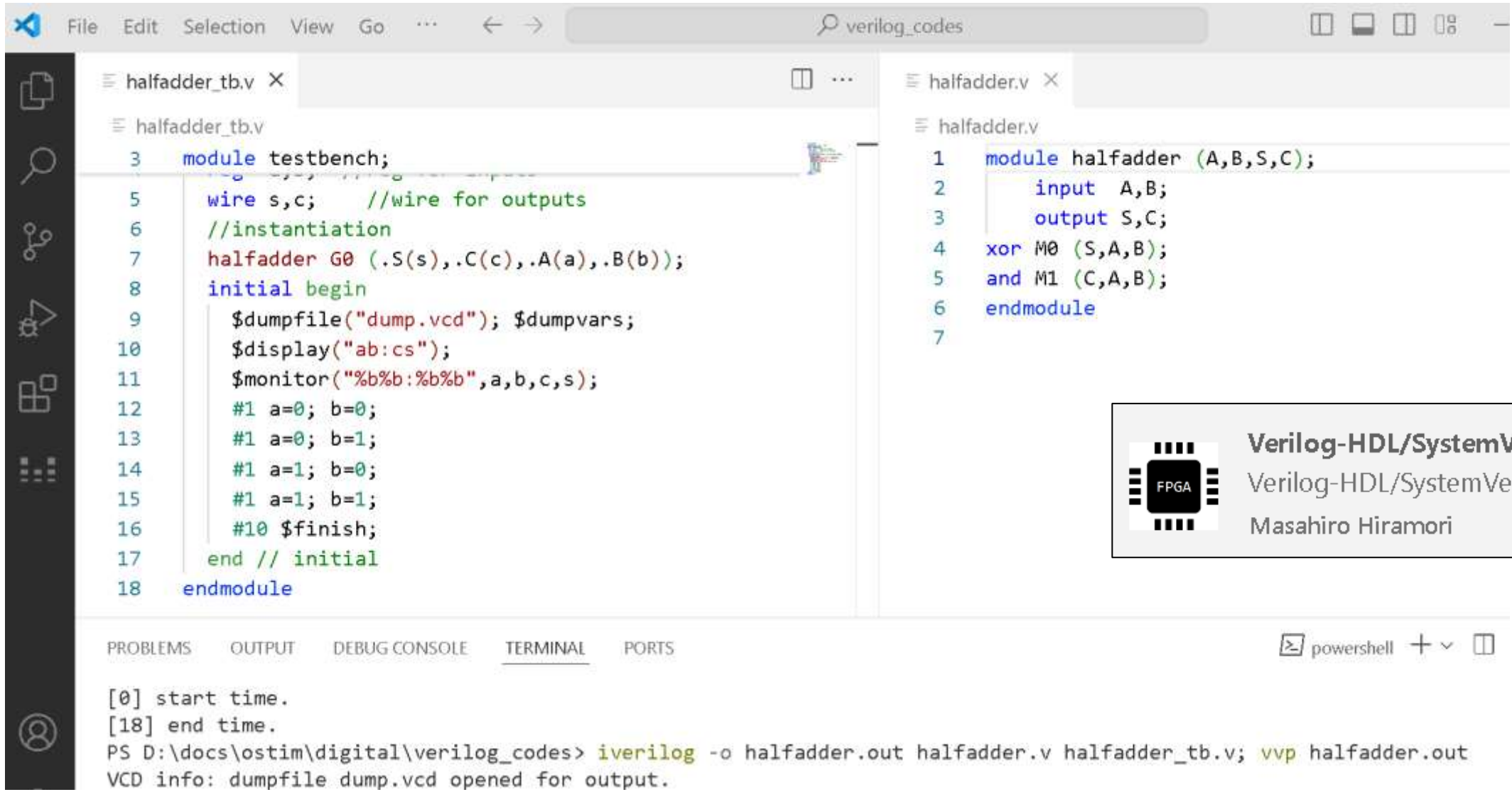


Verilog Test Bench: iverilog and gtkwave

- **command line copy shown here**

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

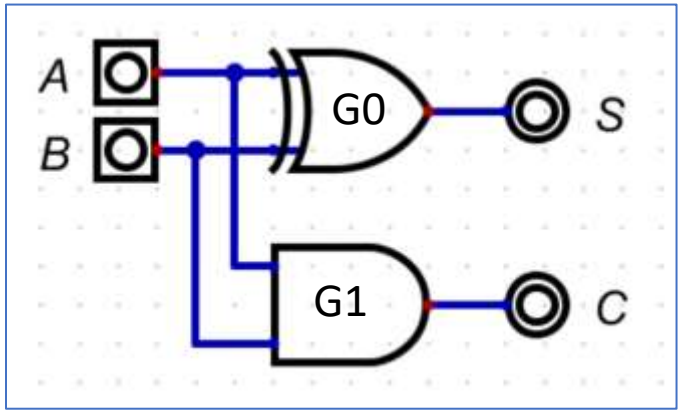
Verilog : using VS Code Editor



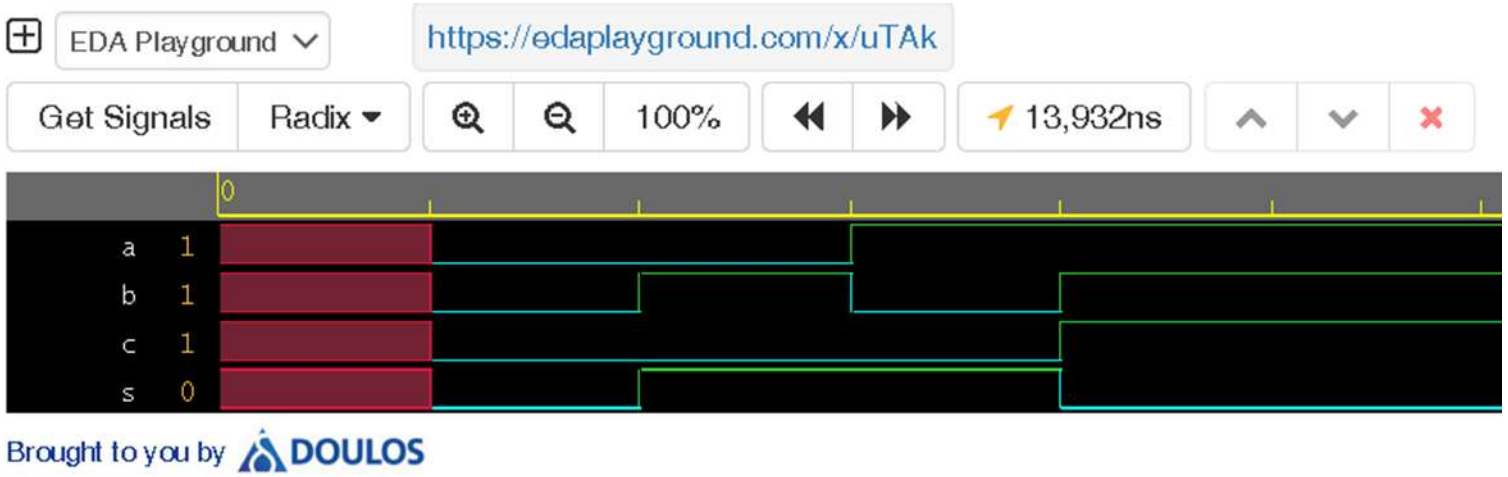
Verilog: Half Adder

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

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




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




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

Why is Verification difficult?

- **How long would it take to test a 32-bit adder?**
  - In such an adder there are 64 inputs = 2^{64} possible inputs
  - That makes around 1.85 10^{19} possibilities
  - If you test one input in 1ns, you can test 10^{9} inputs per second
    - or 8.64 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

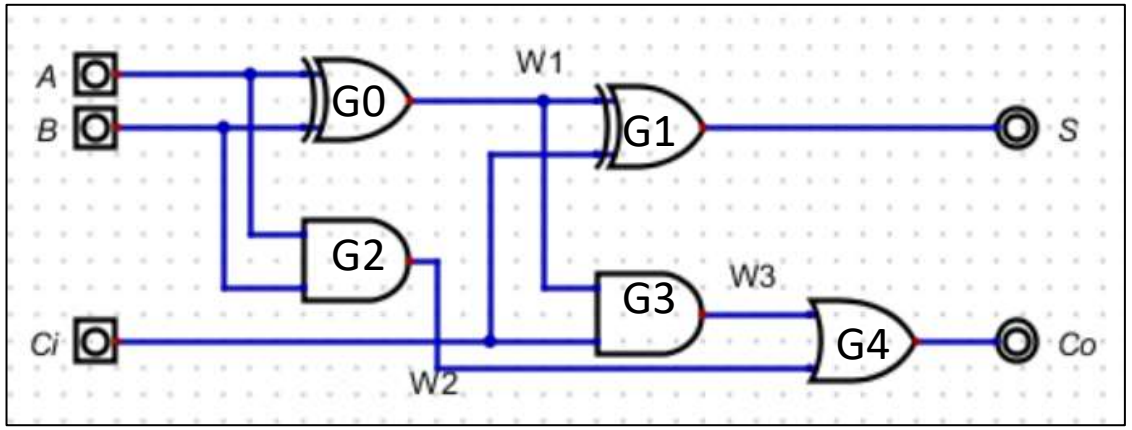
```
module testbench1(); // Testbench has no inputs, outputs
  reg  a, b, c;      // Will be assigned in initial block
  wire y;

  // 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
```

(1) Onur Mutlu | LinkedIn

Complex Combinatorial Circuits: Full Adder

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

- Sunday puzzle sudoku ☺




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

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

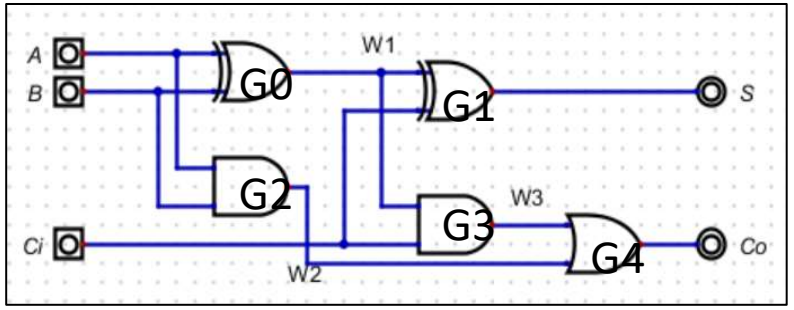
Full Adder

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

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

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

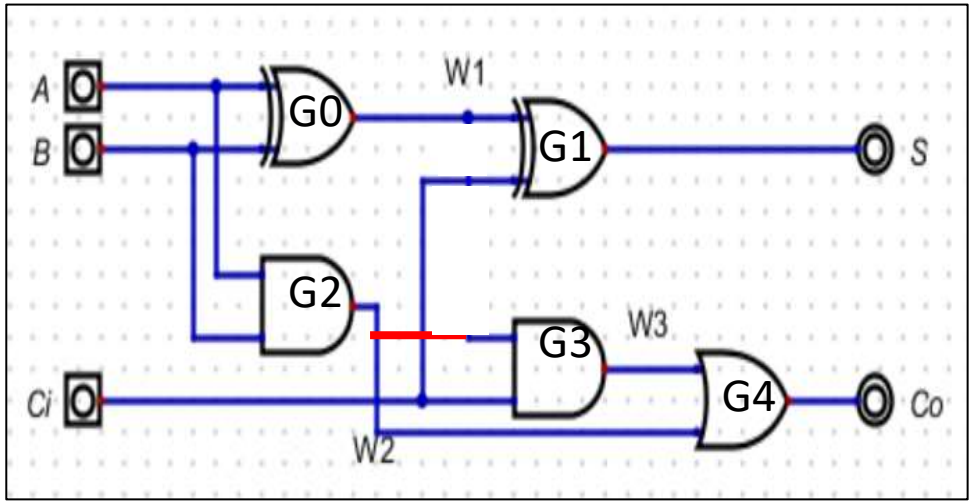
```
000:00
001:01
010:01
011:10
100:01
101:10
110:10
111:11
```




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

Full Adder – Wrong Design

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

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



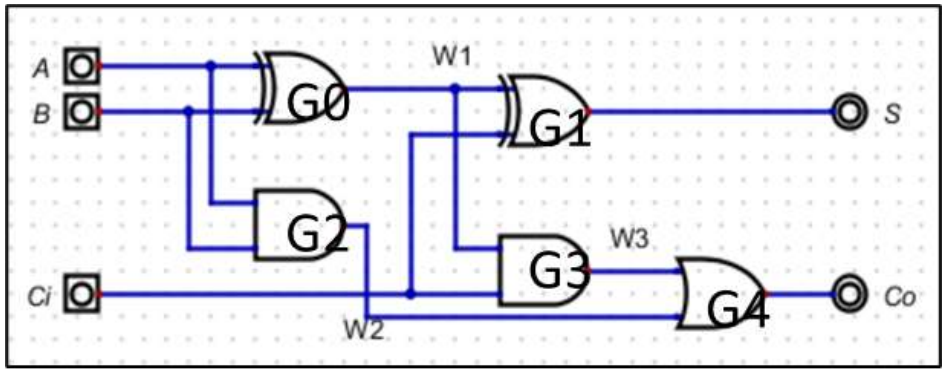


Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder – Wrong Design

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

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

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

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

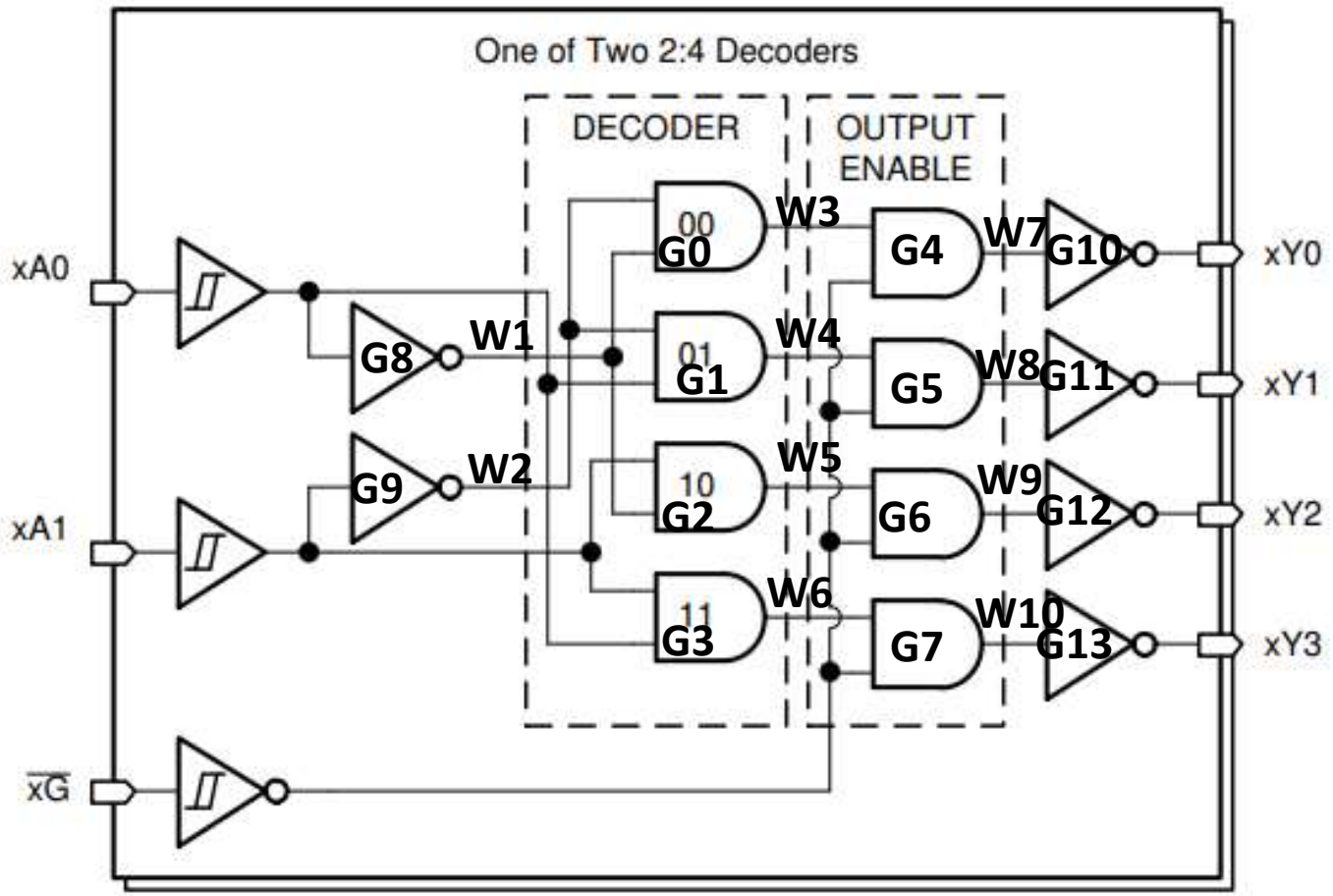


Figure 8-1. Logic Diagram (Positive Logic) for SN74HCS139-Q1 -Q1

```
//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

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

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

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

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

Self-checking Testbench

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

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

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

Self-checking Testbench

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

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

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

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

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

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

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

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

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

Verilog: Self-checking Testbench

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

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

`if ( expression ) statement or statement_group`

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

```
//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
```

Table 4.3
Half Adder

x	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
ABCS
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
```

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

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

```
`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

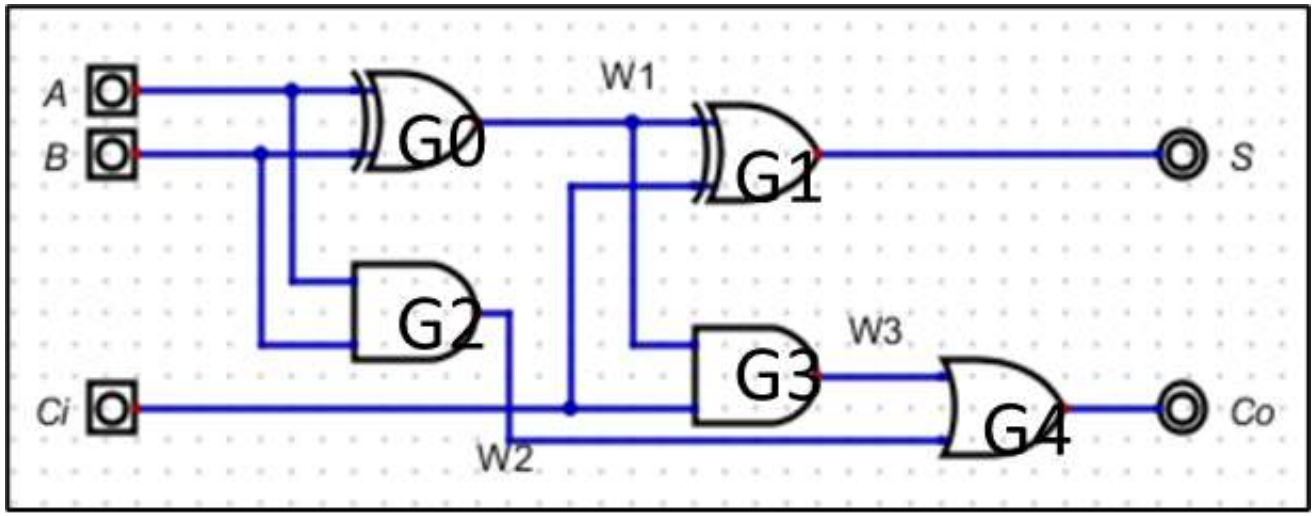
- Only one element at a time within an array can be read from or written to.
- A memory array (a one-dimensional array of reg variables) can be loaded using the $readmemb, $readmemh, $sreadmemb, or $sreadmemh system tasks.

{}	{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);
```

Hierarchical Modeling: Module In Module

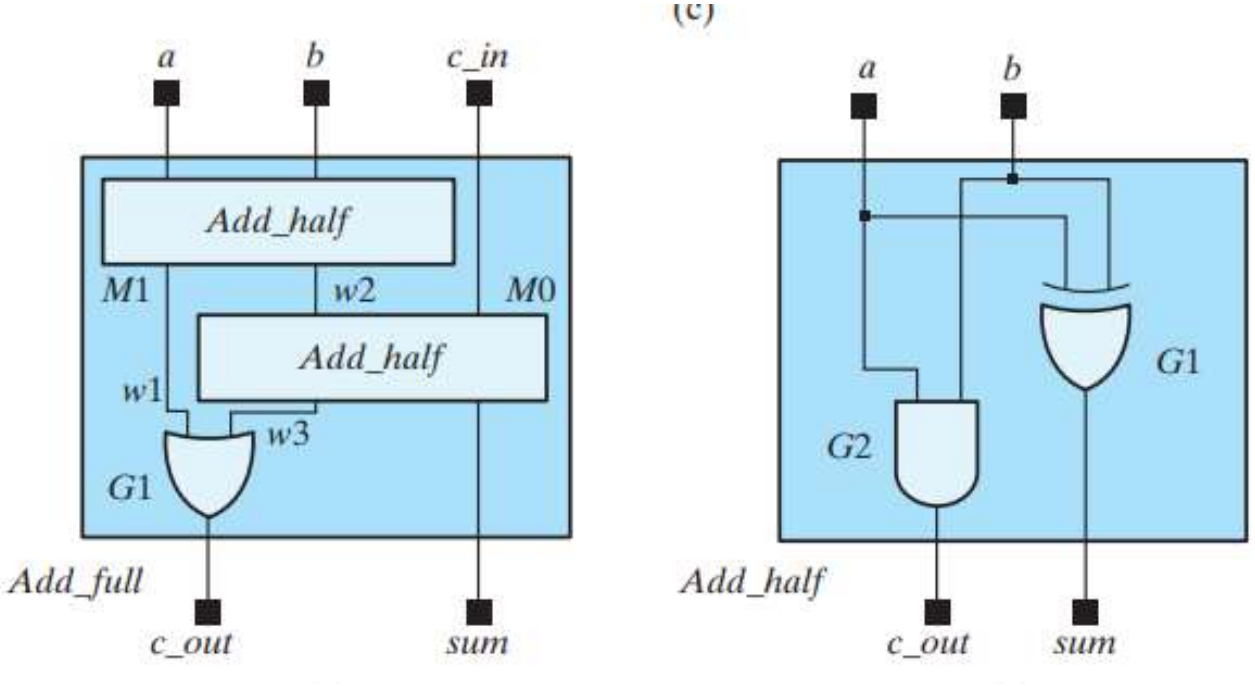
- full adder withıout hierachical design




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

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

endmodule
```

Hierarchical Modeling: Module In Module

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




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

 wire W1,W2,W3;

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

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

Hierarchical Modeling: Module In Module

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

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

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

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

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

Hierarchical Modeling: 4-bit Adder

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

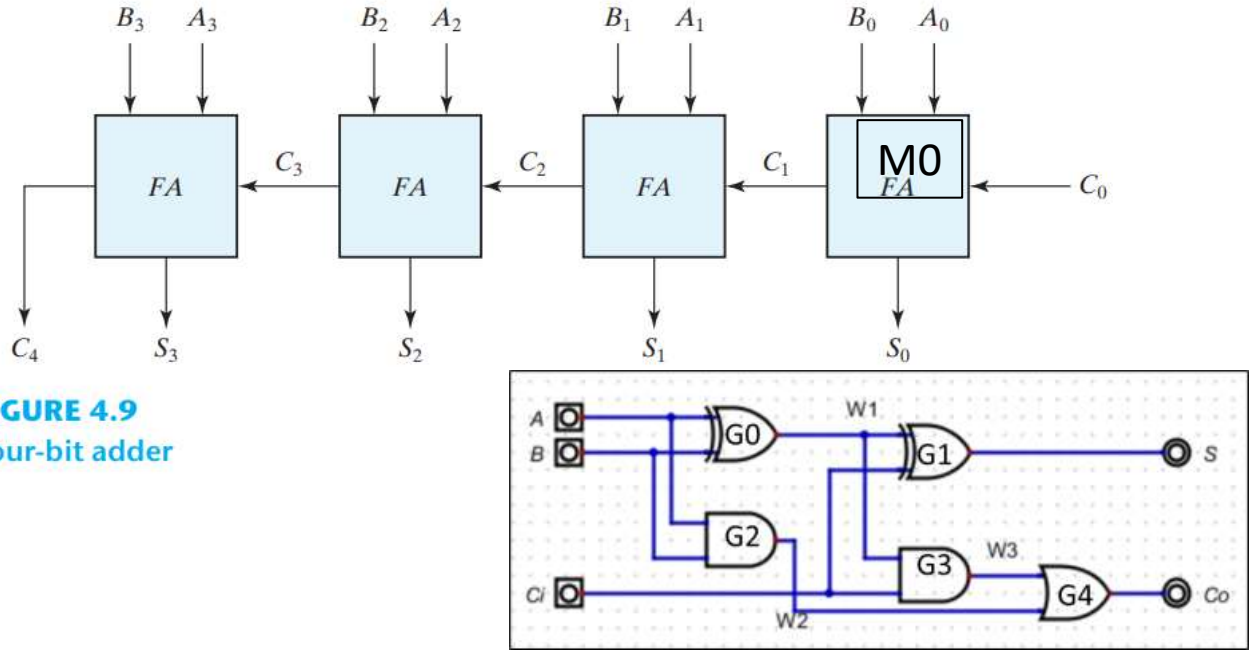


FIGURE 4.9
Four-bit adder

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

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

Hierarchical Modeling: 4-bit Adder

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

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

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

Hierarchical Modeling:

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

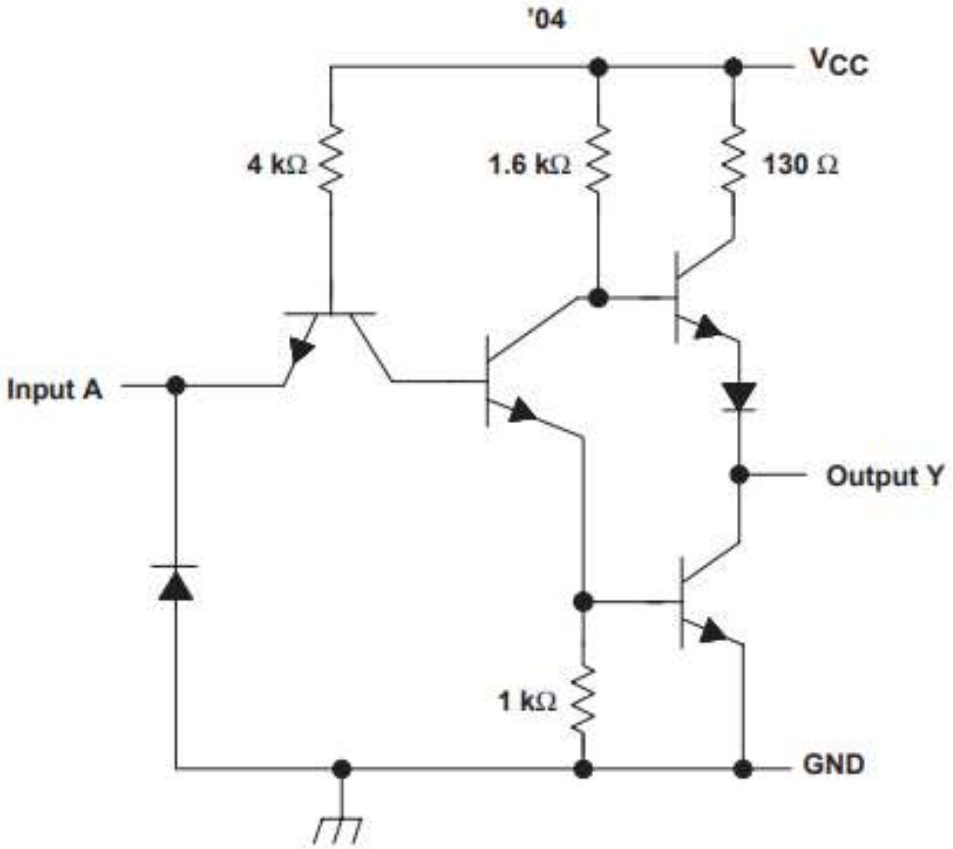
```
//full adder
`include "fulladder.v"

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

```
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
```

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

Timing in Combinatorial Circuits

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



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

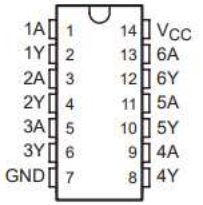
SDLS029C – DECEMBER 1983 – REVISED JANUARY 2004

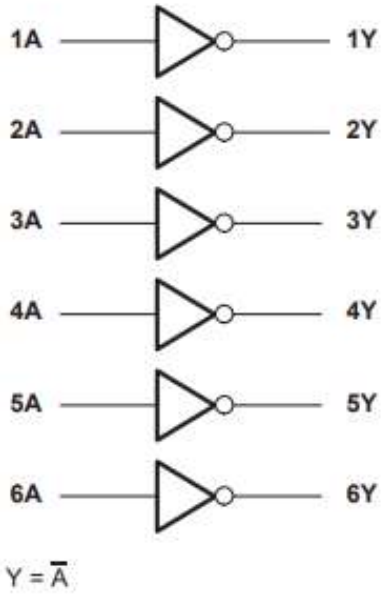
- Dependable Texas Instruments Quality and Reliability

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
```

8.0 Primitive Instances

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

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

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

```
`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
```

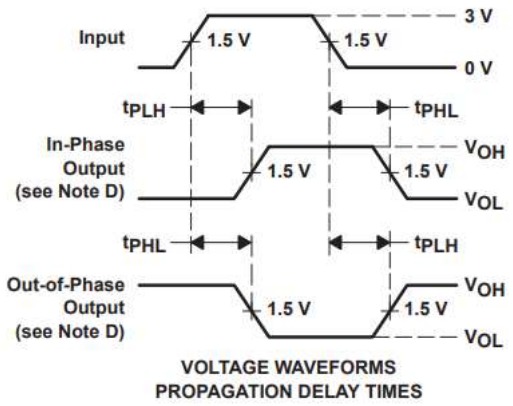
Timing in Combinatorial Circuits

**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)

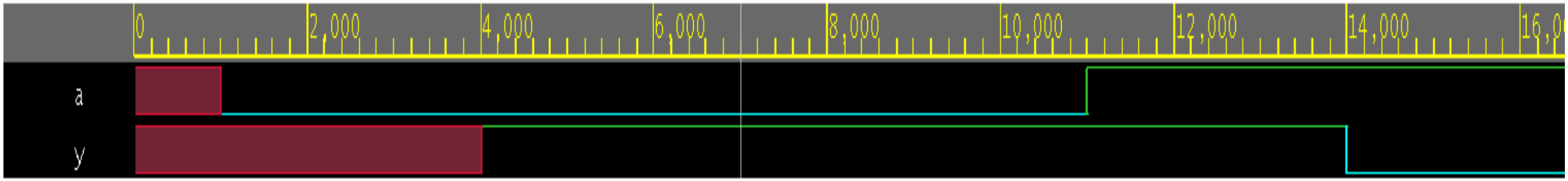
PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN54S04 SN74S04 MIN	TYP	MAX	UNIT
t_{PLH}	A	Y	R_L = 280 Ω, C_L = 15 pF		3	4.5	ns
t_{PHL}					3	5	



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
```



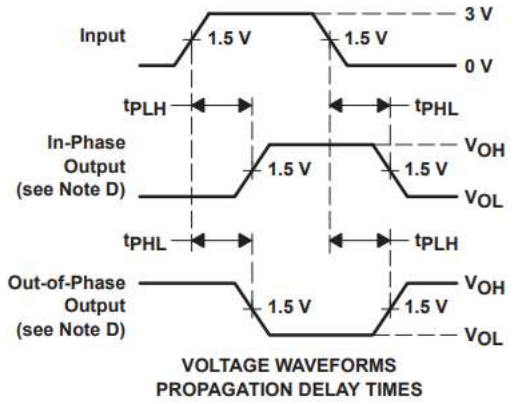
Timing in Combinatorial Circuits

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

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

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

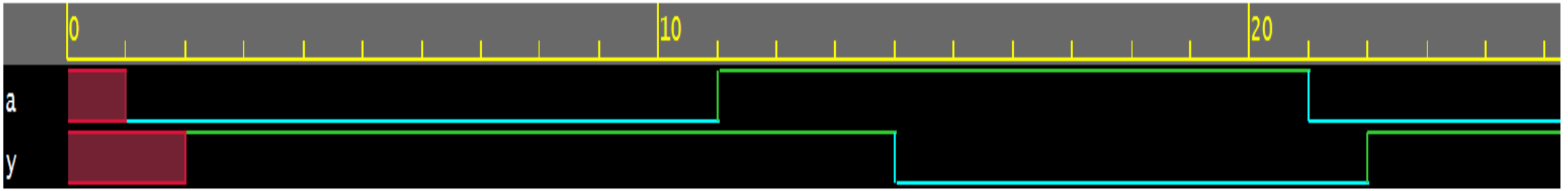
PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN54S04 SN74S04 MIN	TYP	MAX	UNIT
t_{PLH}	A	Y	R_L = 280 Ω, C_L = 15 pF		3	4.5	ns
t_{PHL}					3	5	



VOLTAGE WAVEFORMS
PROPAGATION DELAY TIMES

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

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



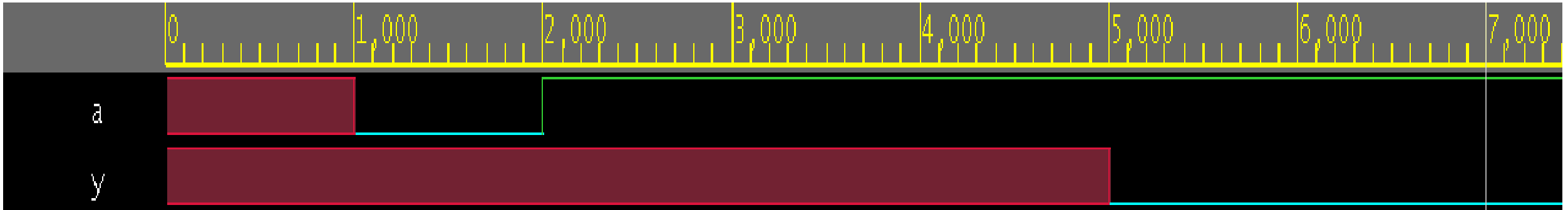
Timing in Combinatorial Circuits

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

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

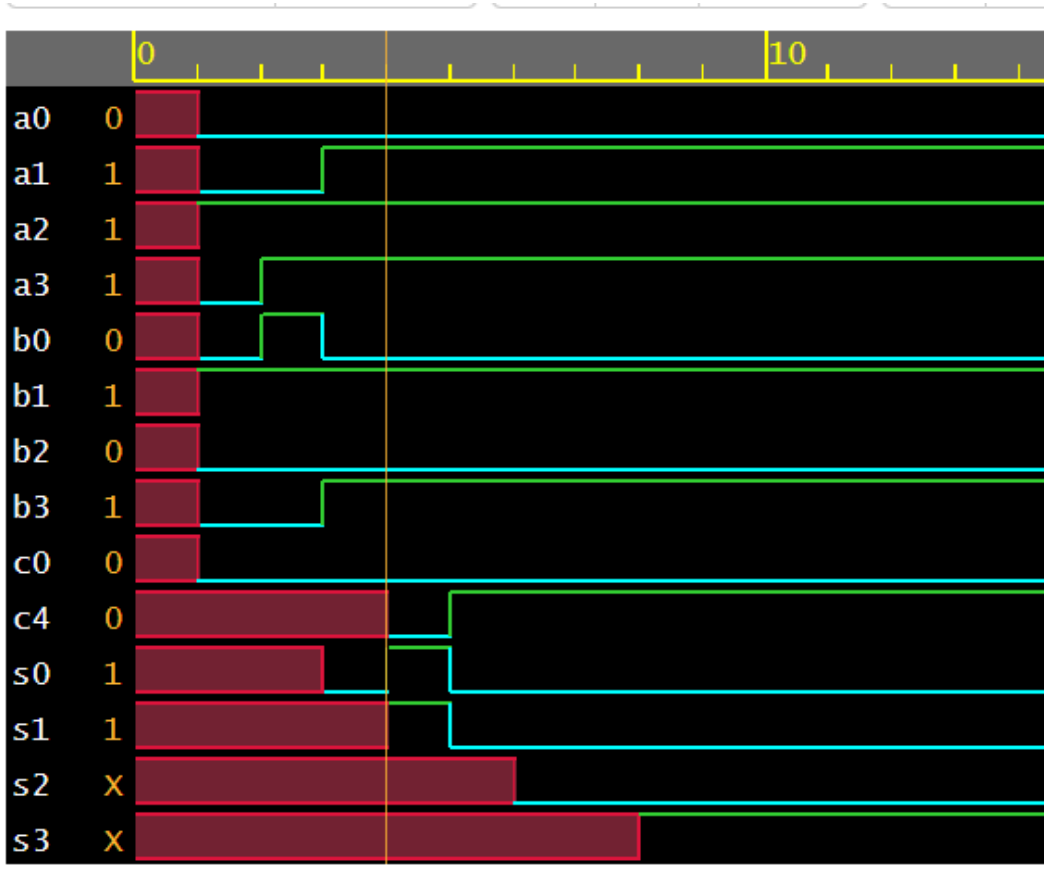
```
`timescale 1 ns / 1 ns
module inverter(
  output Y,
  input A
);
  not #3 G0 (Y,A);  //1ns rise&3ns fall delay
endmodule
```

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



Hierarchical Modeling: 4-bit Adder (delay)

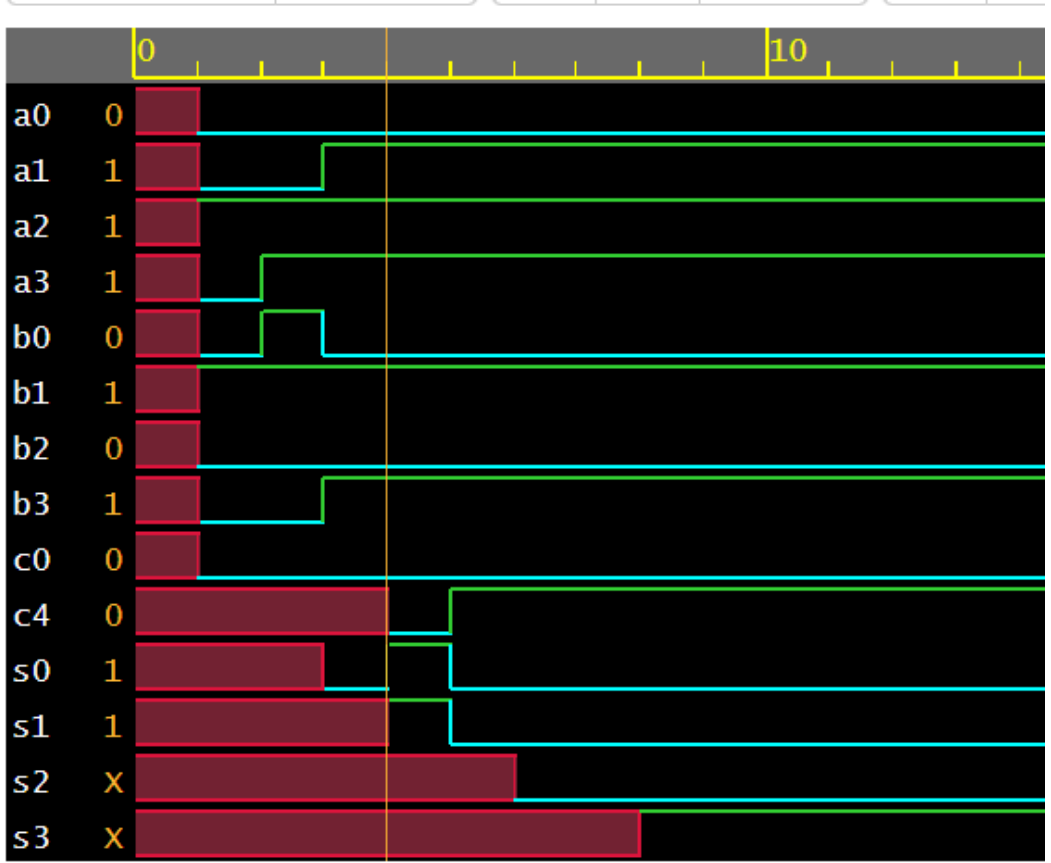
- assume that each gate has 1ns delay
- delay of S3 becomes 8ns
- S3 changes value and becomes stable at 17ns

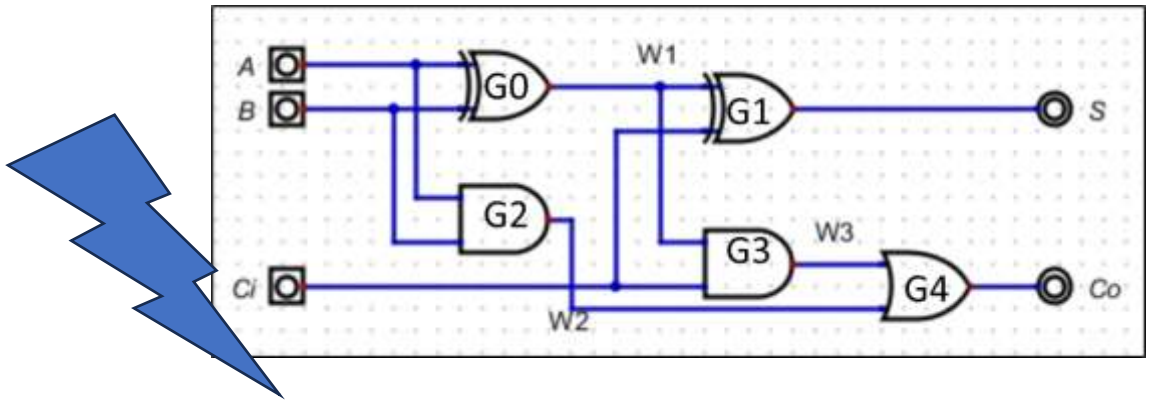



```
//full adder
`timescale 1 ns / 1 ns
module fourbitadder(
  output S0,S1,S2,S3,C4,
  input A0,A1,A2,A3,B0,B1,B2,B3,C0);
  wire C1,C2,C3;
  fulladder M0 (.S(S0),.Co(C1),.A(A0),.B(B0),.Ci(C0));
  fulladder M1 (.S(S1),.Co(C2),.A(A1),.B(B1),.Ci(C1));
  fulladder M2 (.S(S2),.Co(C3),.A(A2),.B(B2),.Ci(C2));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

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

Hierarchical Modeling: 4-bit Adder (delay)

- assume that each gate has 1ns delay
- delay of S3 becomes 8ns
- S0 changes value and becomes stable 0100000
- so input has to change slower than 17ns





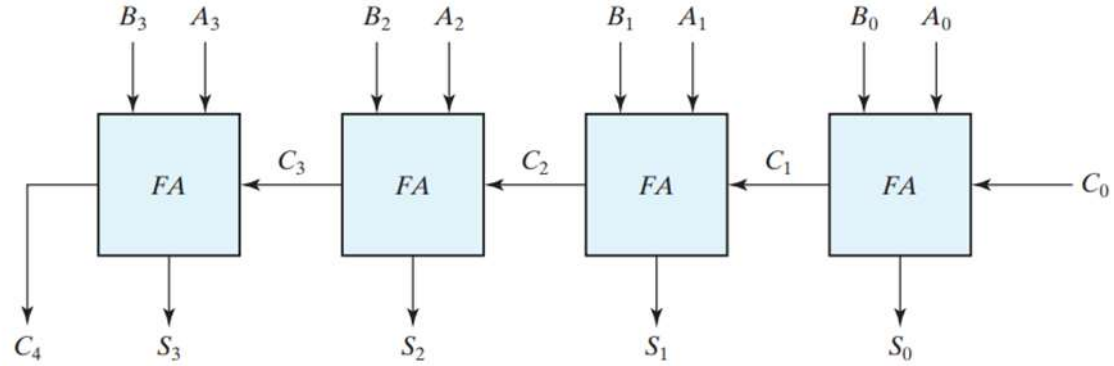
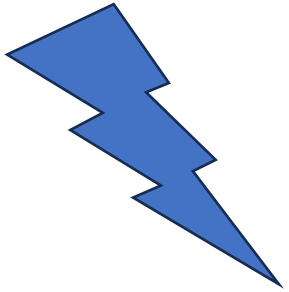


FIGURE 4.9
Four-bit adder

Hierarchical Modeling: 4-bit Adder (delay)



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

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

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

Hierarchical Modeling: 4-bit Adder (delay)

- so input has to change slower than max delay

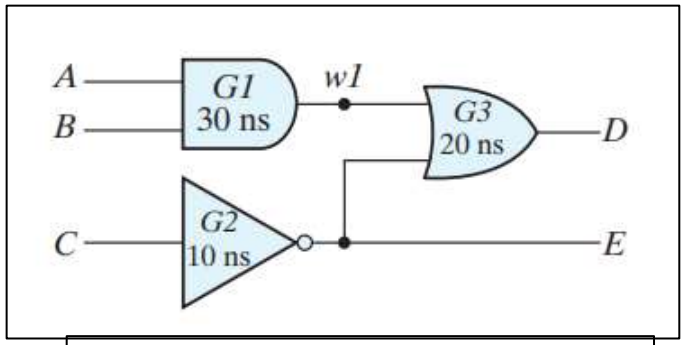
switching characteristics (see Note 2)

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

§ For conditions shown as MIN or MAX, use the appropriate value specified under recommended operating conditions.
NOTE 2: Load circuits and waveforms are shown in Section 1.

Output Glitches

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




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

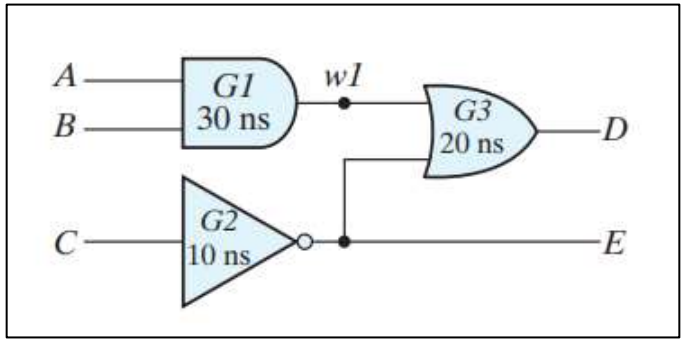
```
time A B C : D E
0 0 0 0 : x x
10 0 0 0 : x 1
30 0 0 0 : 1 1
100 1 1 1 : 1 1
110 1 1 1 : 1 0
130 1 1 1 : 0 0
150 1 1 1 : 1 0
```

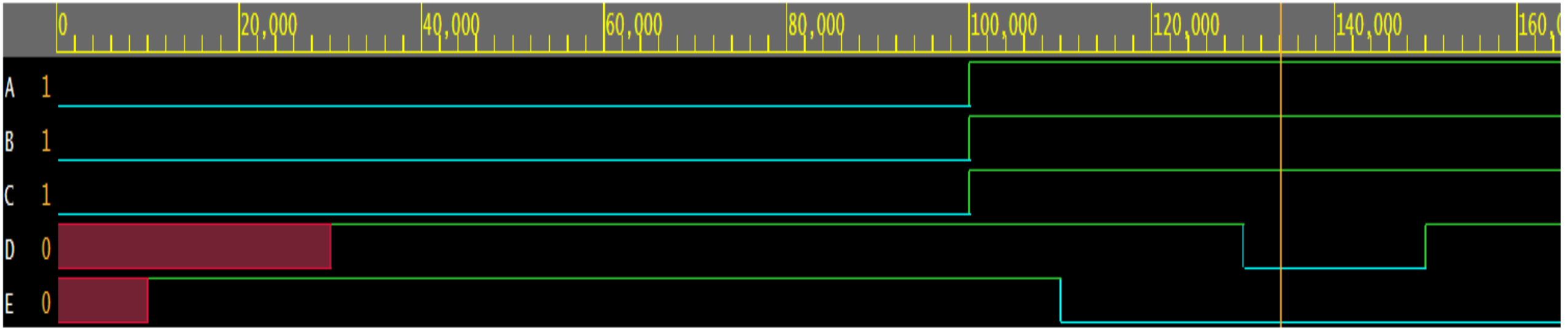
```
//testbench
`timescale 1 ns / 1 ns
module testbench ();
reg A,B,C;
wire D,E;
  andor M0 (.A(A),.B(B),.C(C),.D(D),.E(E));
initial begin
  $dumpfile("dump.vcd");
  $dumpvars;
  $display("\t\ttime A B C : D E");
  $monitor("%d %b %b %b : %b %b",$time,A,B,C,D,E);
  A=0; B=0; C=0;
  #100 A=1; B=1; C=1;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

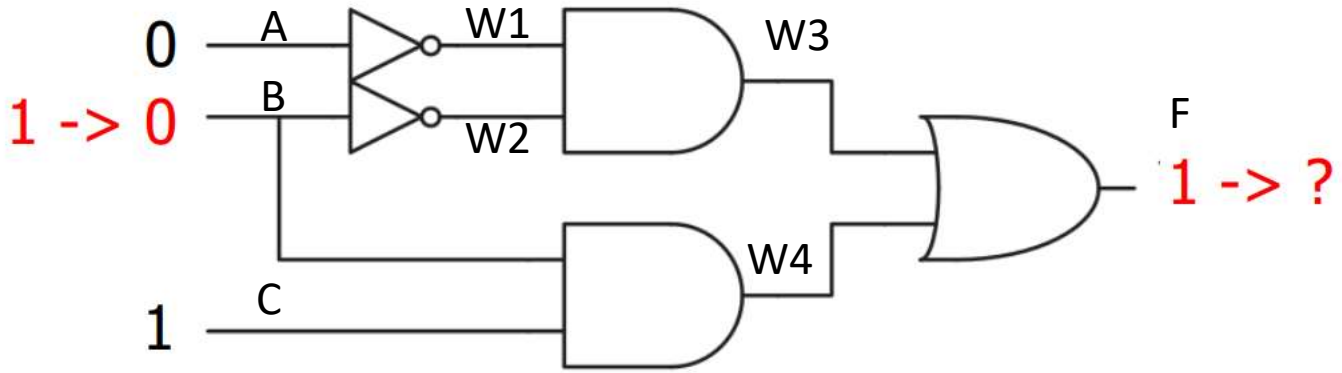
time A B C : D E
0 0 0 0 : x x
10 0 0 0 : x 1
30 0 0 0 : 1 1
100 1 1 1 : 1 1
110 1 1 1 : 1 0
130 1 1 1 : 0 0
150 1 1 1 : 1 0





Output Glitches

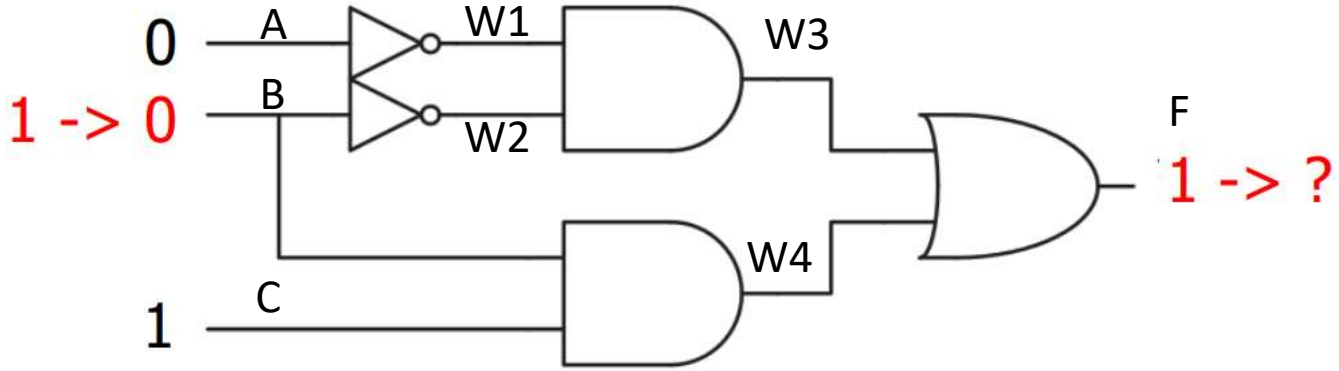
*Onur Mutlu lecture notes

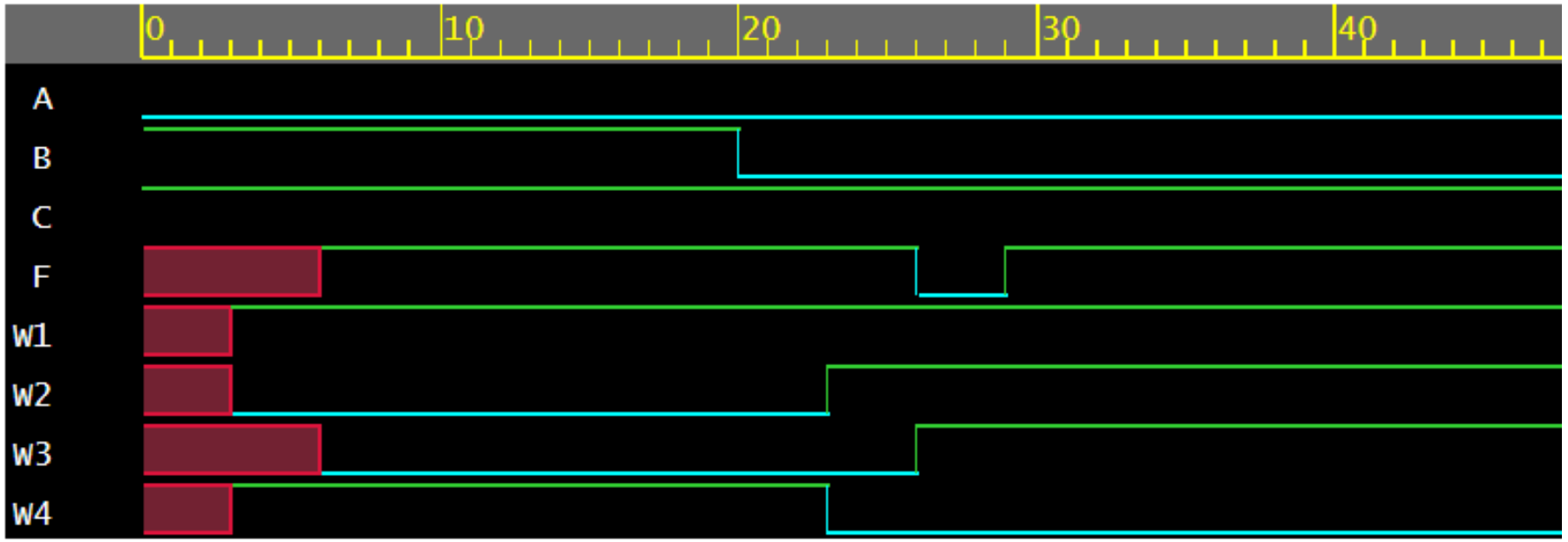



```
`timescale 1 ns / 1 ns
module glitch(
  output F,
  input A,B,C
);
  wire W1,W2,W3,W4;
  not #3 G0 (W1,A);
  not #3 G1 (W2,B);
  and #3 G2 (W3,W1,W2);
  and #3 G3 (W4,B,C);
  or  #3 G4 (F,W3,W4);
endmodule
```

Output Glitches

*Onur Mutlu lecture notes

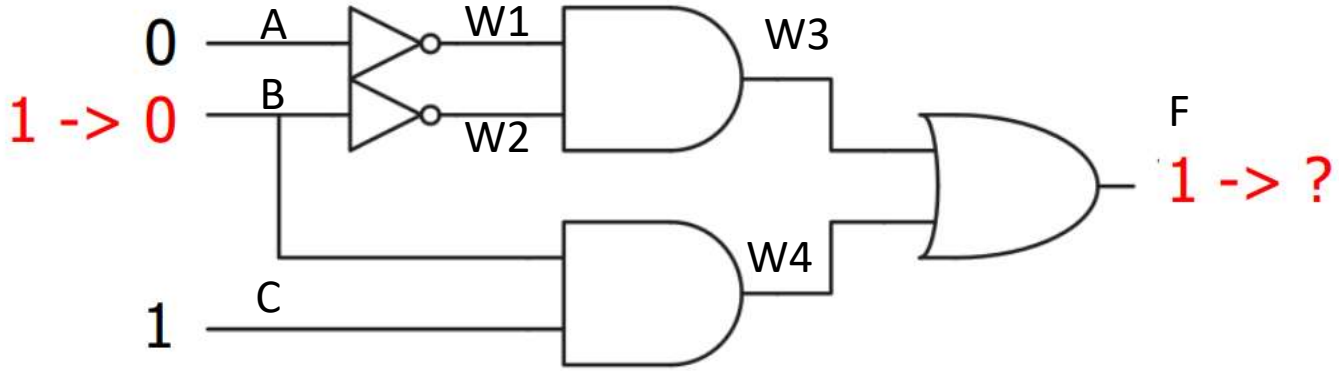




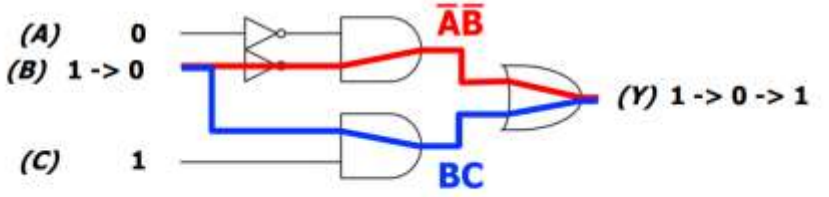

```
`timescale 1 ns / 1 ns
module testbench;
  reg  a,b,c;  //reg for inputs
  wire f;    //wire for outputs
  //instantiation
  glitch G0 (.F(f),.A(a),.B(b),.C(c));
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("a,b,c:f");
    $monitor("%b%b%b:%b",a,b,c,f);
       a=0; b=1; c=1;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

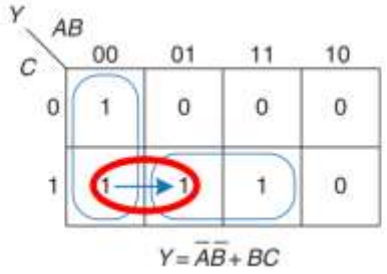
*Onur Mutlu lecture notes

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**

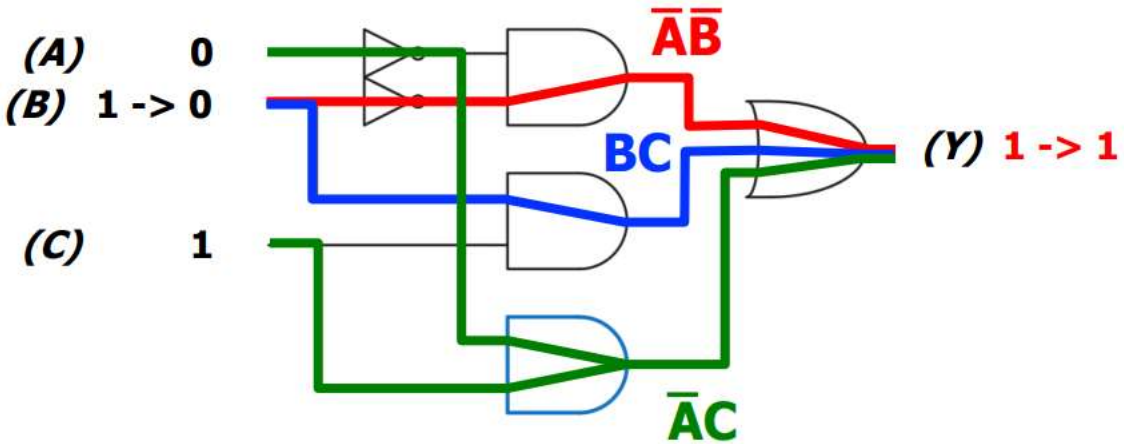




*Onur Mutlu lecture notes

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

$\bar{A}C$

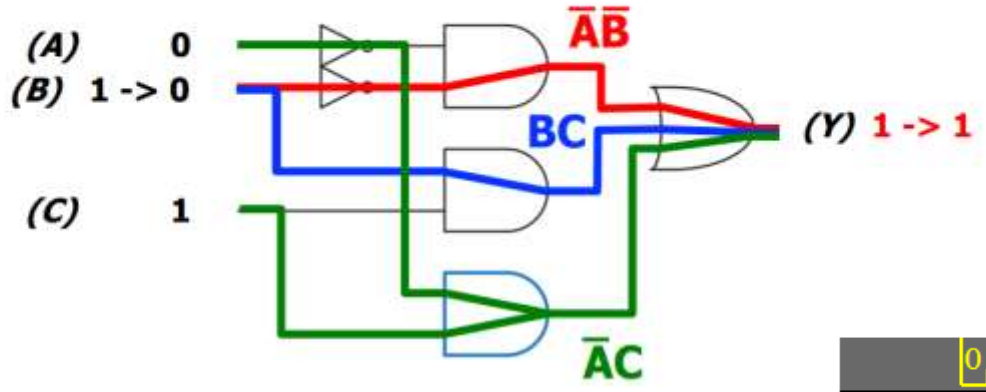
$Y = \bar{A}\bar{B} + BC + \bar{A}C$

No dependence on B
=> No glitch!

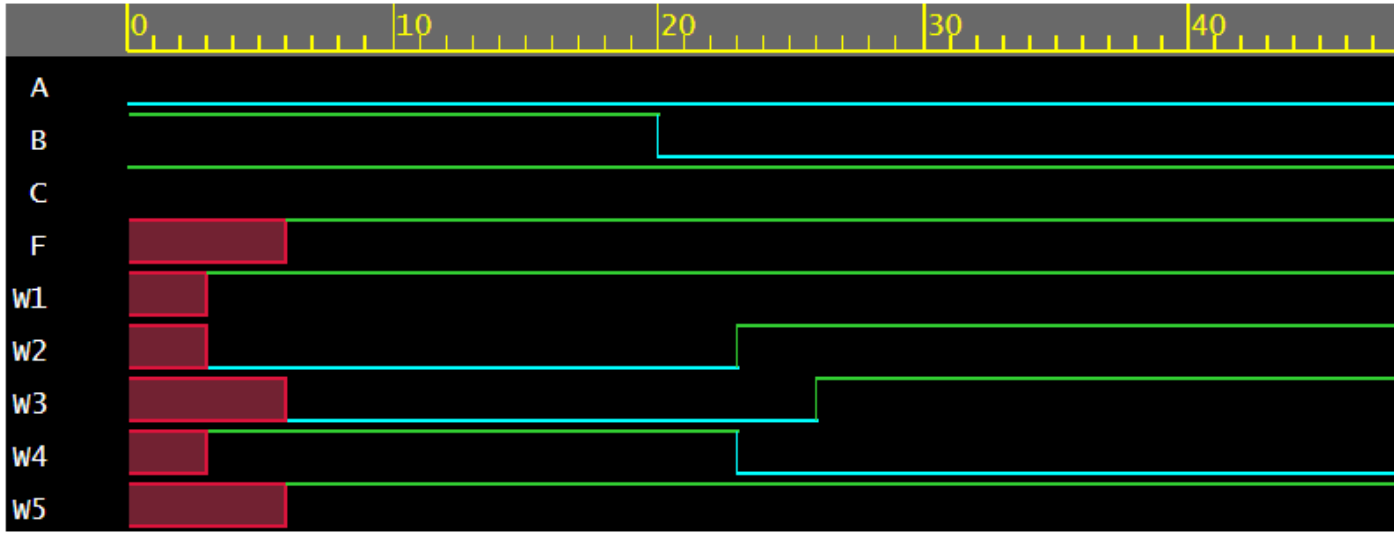
Output Glitches

*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:

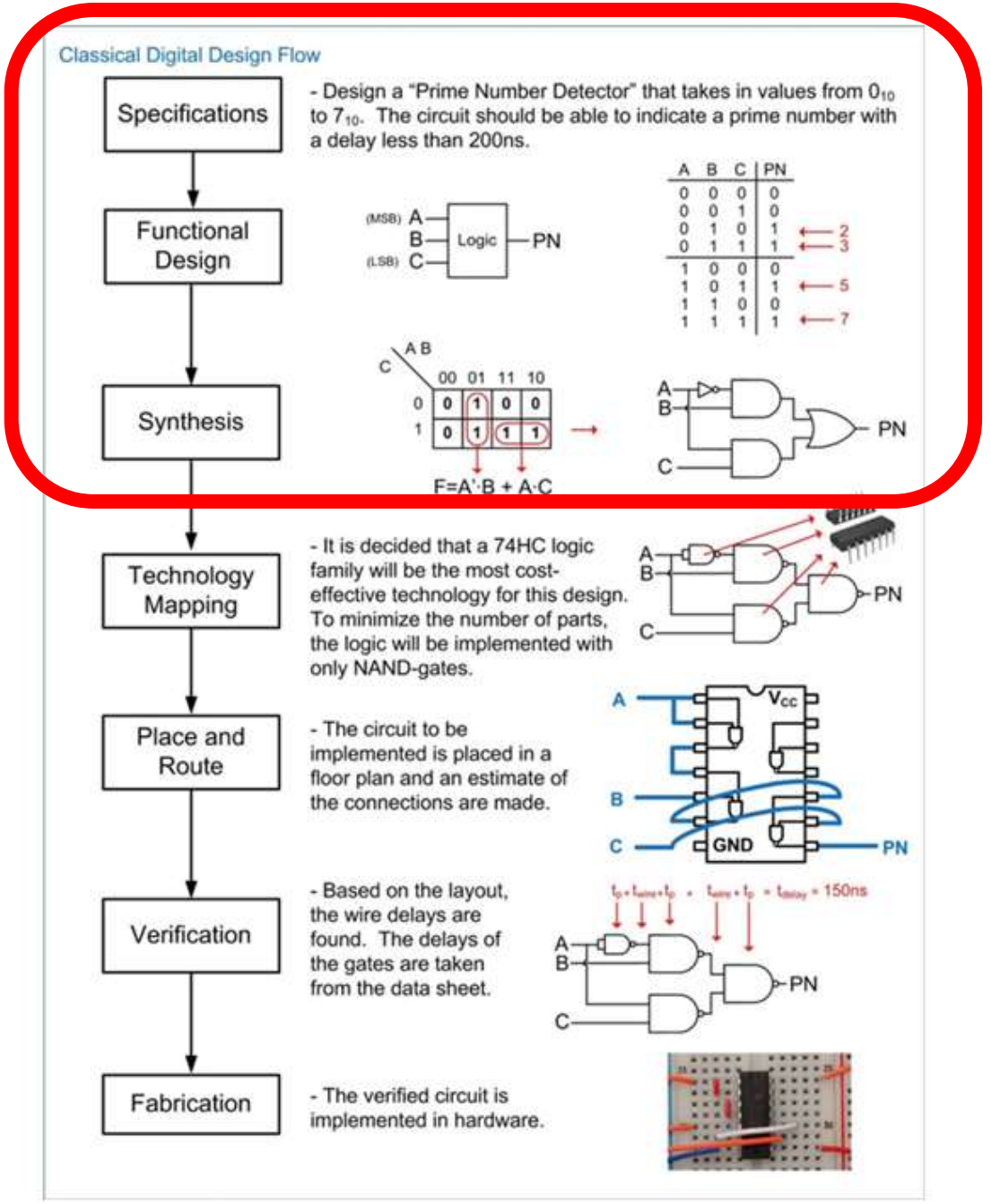
High-Level and Logic Design: Create the architecture. Specify and implement the required system behavior.

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

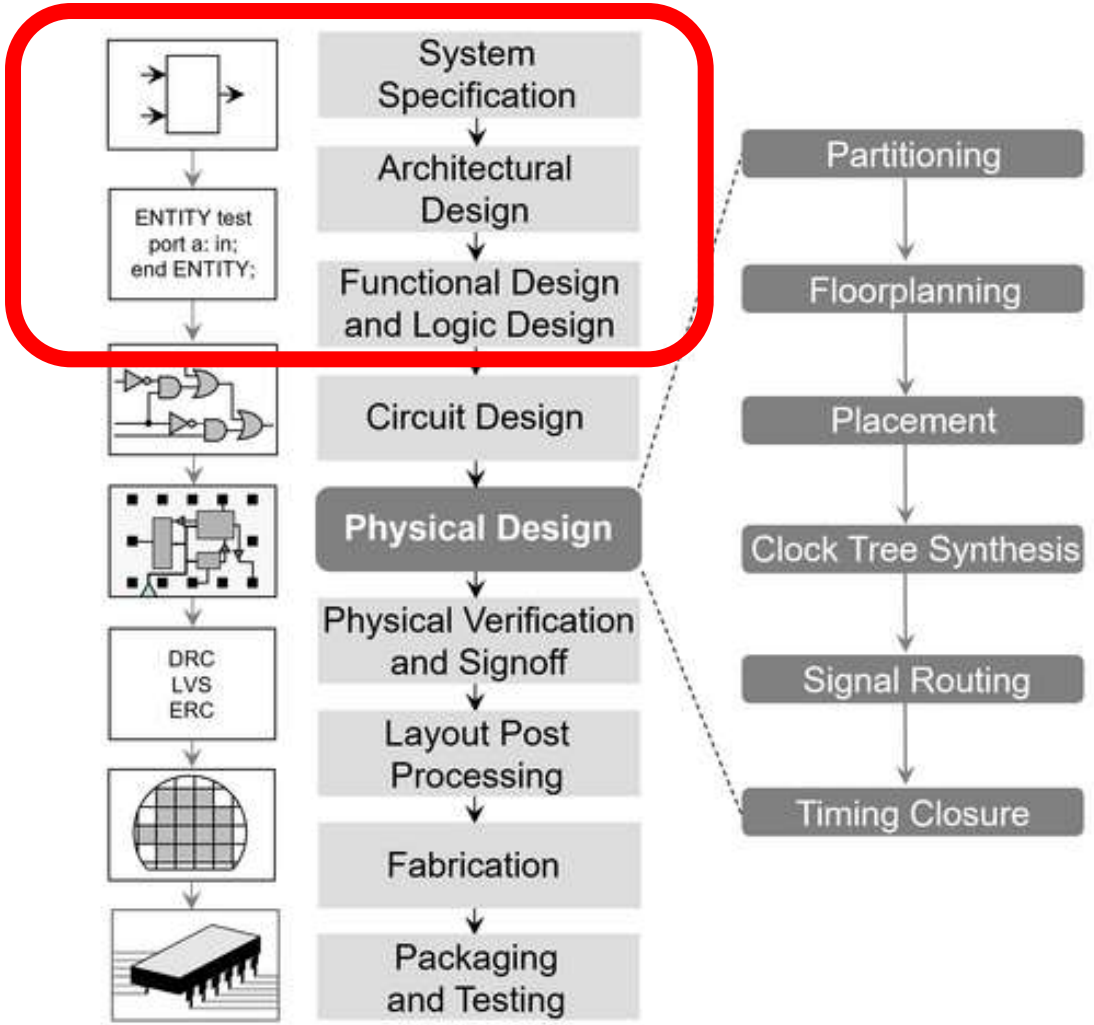
Logic Synthesis: Convert the high-level description into a gate-level logical netlist.

Physical Implementation: Create a physical layout suitable for fabrication.

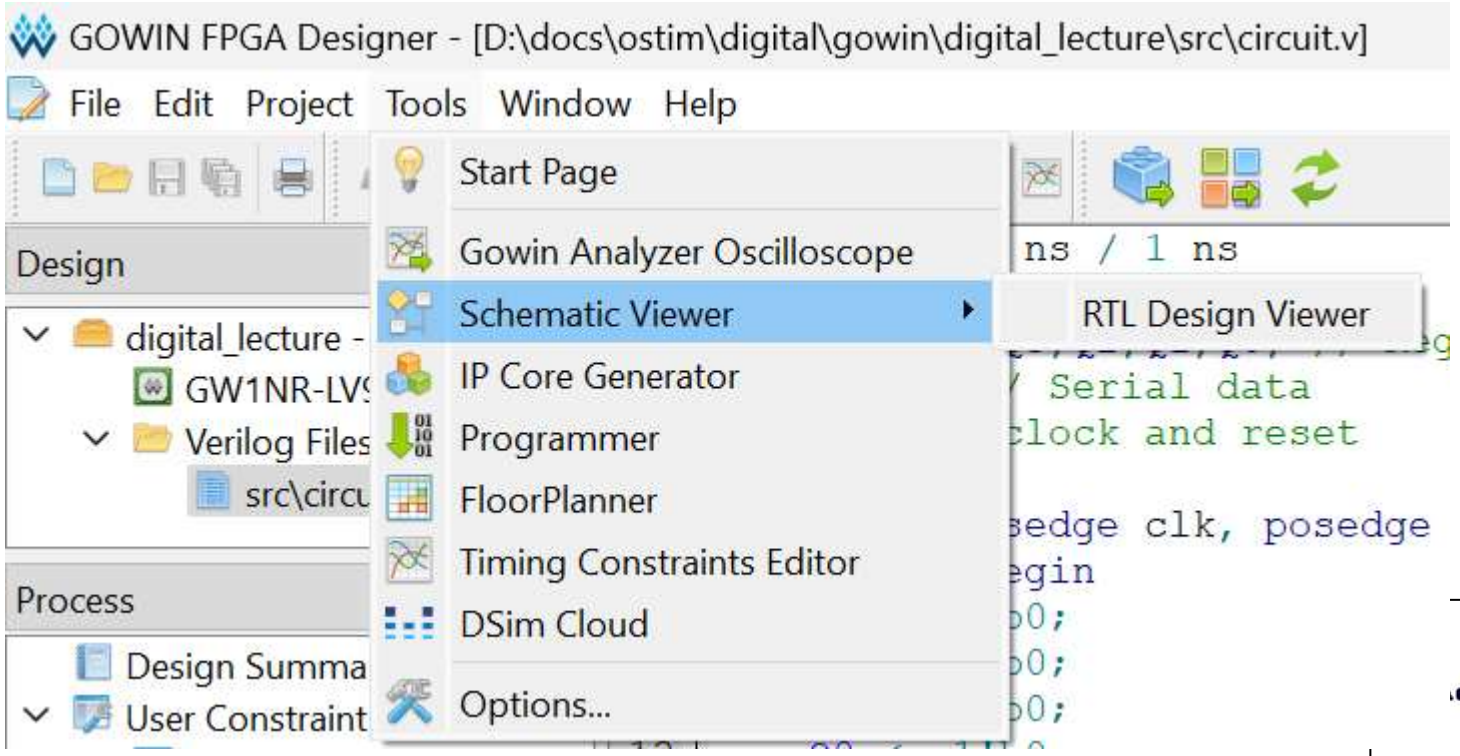
Physical Chip Design

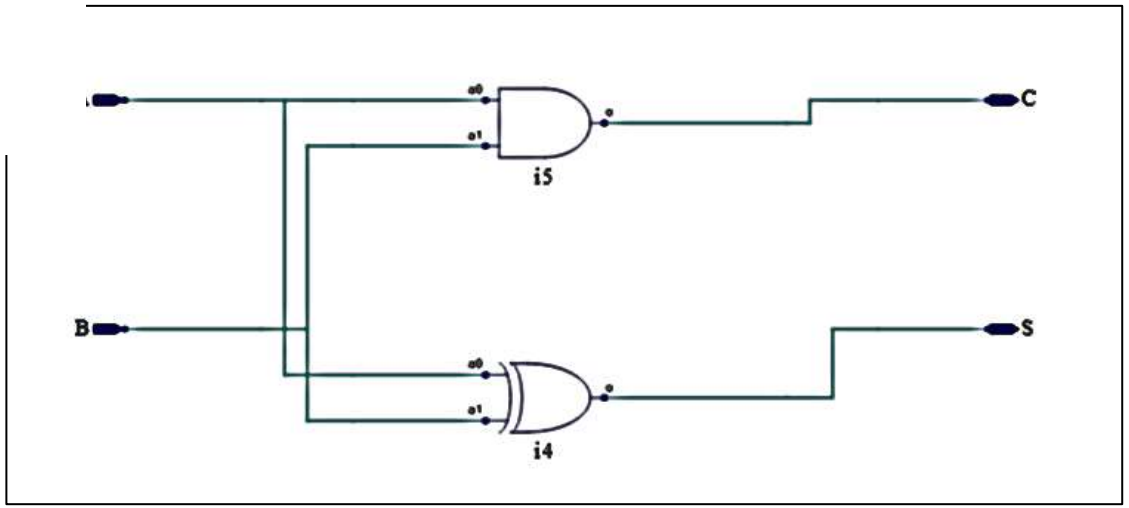


Physical Chip Design



Gowin Synthesis

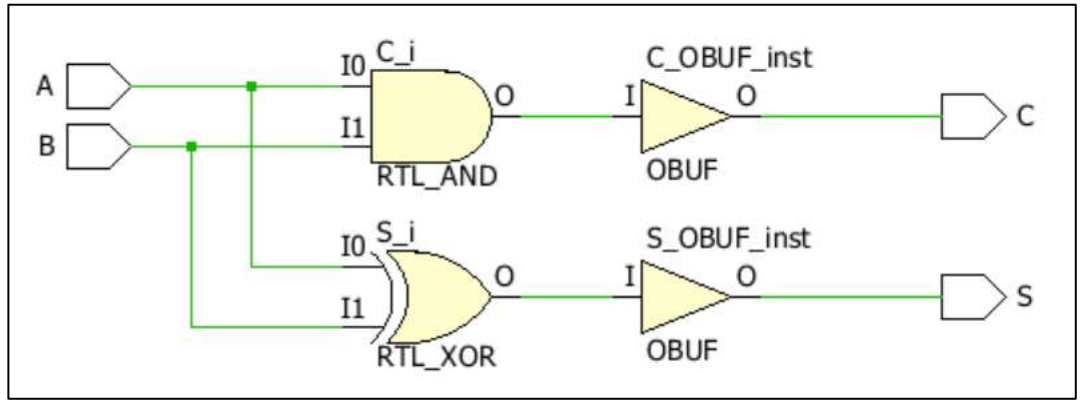




Vivado Synthesis (AMD FPGA)
Quartus (INTEL FPGA)

Vivado quick start:

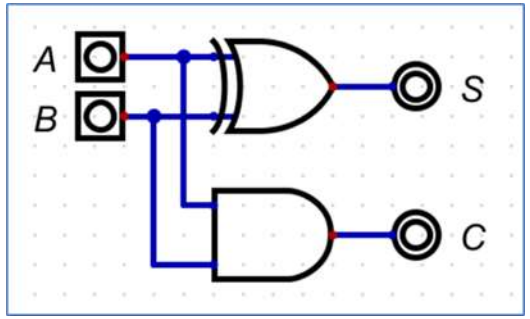
- New Project
- RTL Project
- Default Part: xc7a50ticsg325 Artix7 Spartan
- Next-Next
- Design Sources – Right Click-Add Sources
- Add Design Sources
- Create File – halfadder.v
- Define Module- OK
- double Click halfadder.v and write the code or copy/paste from EDAplayground
- Flow-Open Elaborated Design-OK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

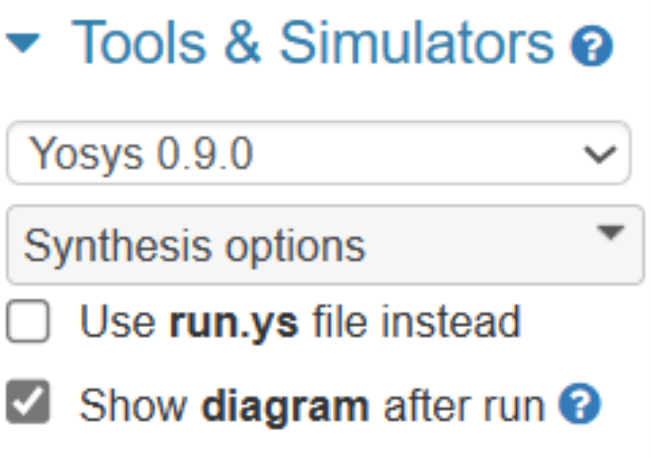


[Basys 3 - Digilent Reference]

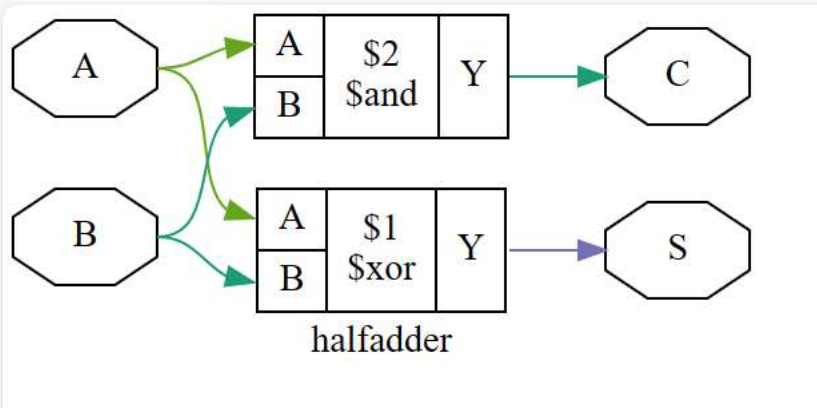
Verilog for Synthesis: Yosys

- 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**





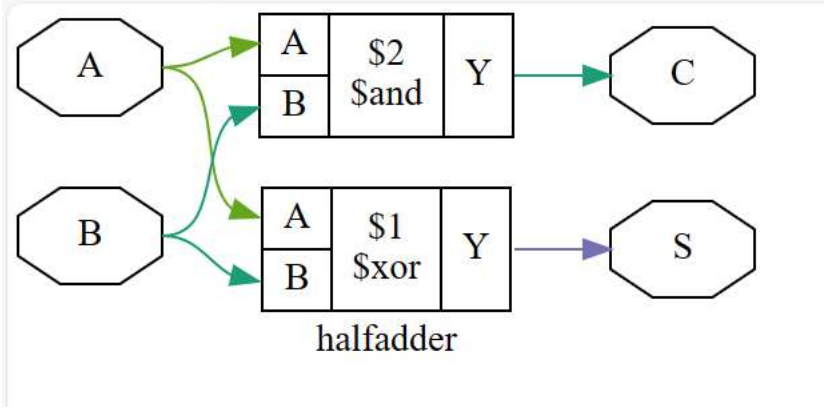

```
module halfadder (
  output C,S,
  input A,B);

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



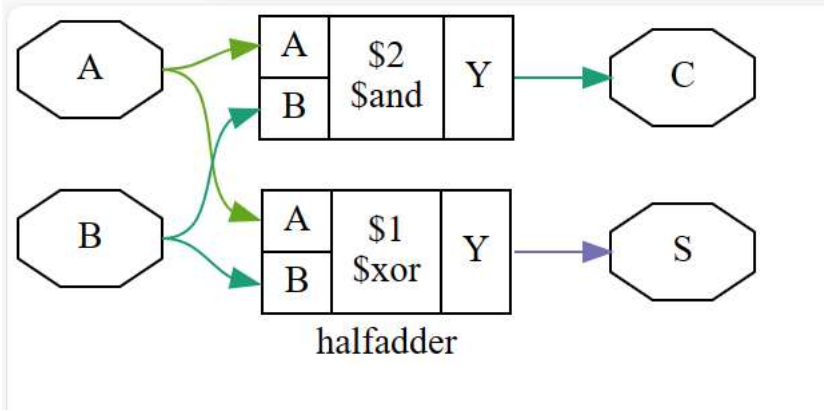
Verilog for Synthesis: Yosys

- 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**

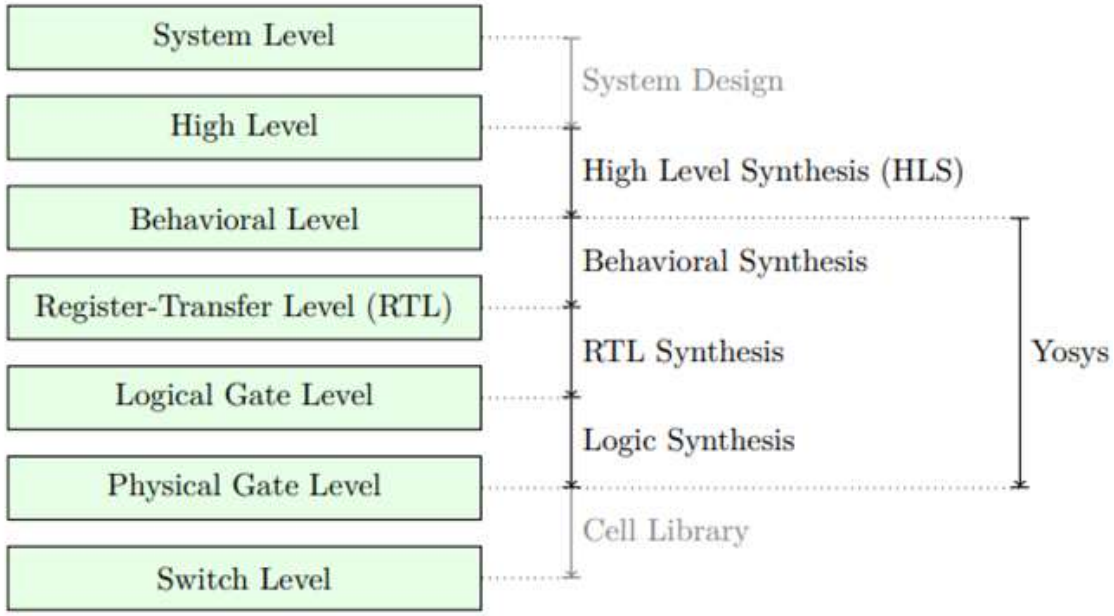


Figure 2.1: Different levels of abstraction and synthesis.

1. **Gate Level Modeling:** primitives (and or not xor etc)

2. **Dataflow Modeling** : continous assignment, logic and arithmetic and conditional operators

3. **Behavioral Modeling**: always @*, if-else, case,

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

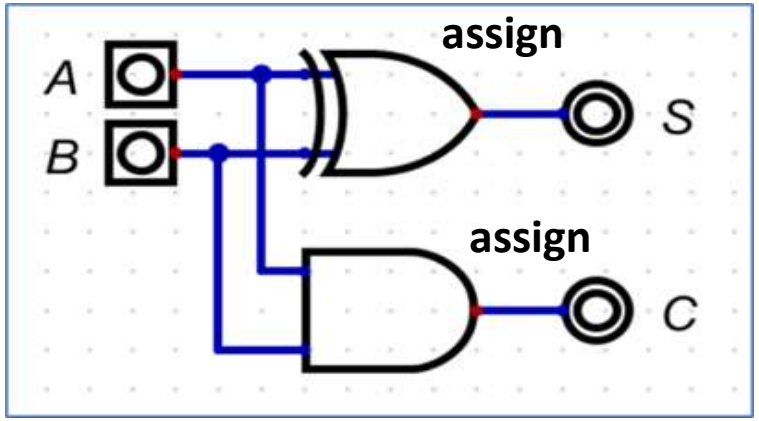
- Continuous Assignments useful for behavioral description of combinatorial logic circuits
- Assign **wires** EXPRESSION to WIRE
- it is wire connection and immediately changes with the expression

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

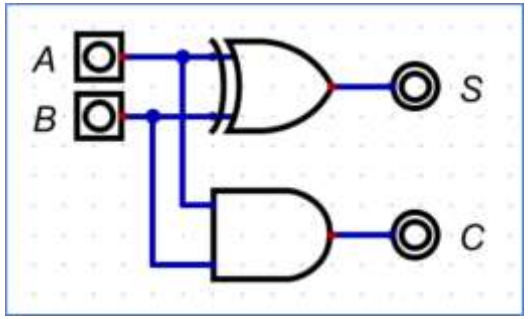
```
module halfadder (
  output C,S,
  input A,B);

  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.

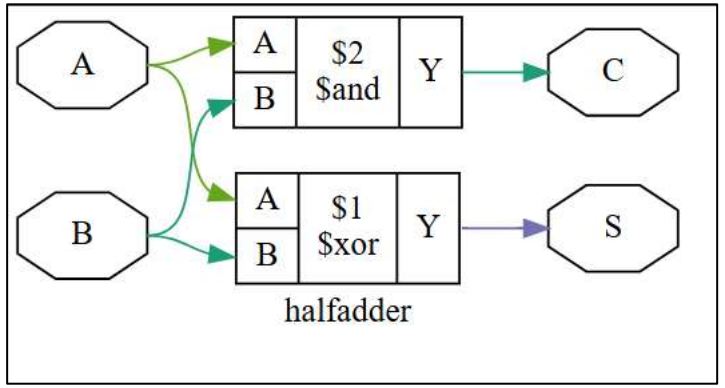



```
module halfadder (
  output C,S,
  input A,B);

  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

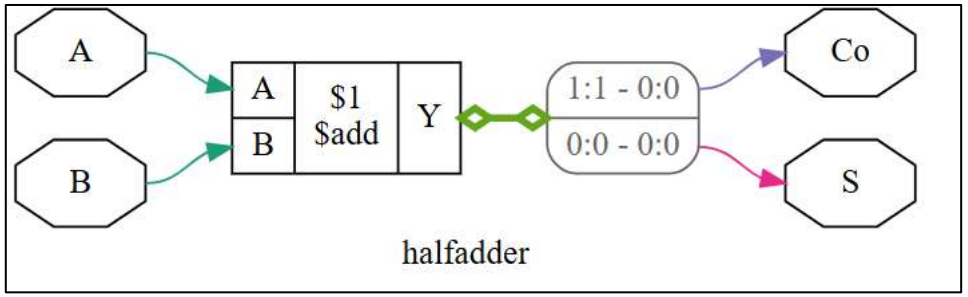
- We can use arithmetic operators also to define half adder
- Notice that synthesizer generated the half adder circuit based on our modeling

```
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

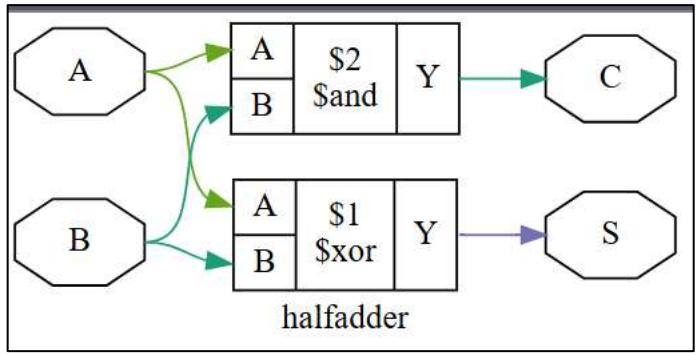


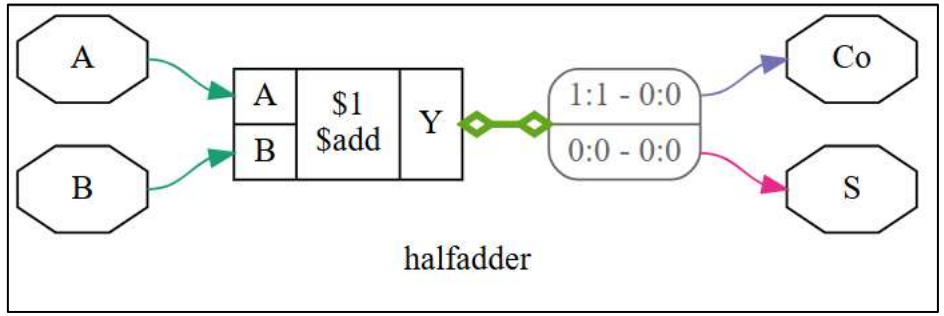
Dataflow Modeling: Arithmetic Operators

- 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
```

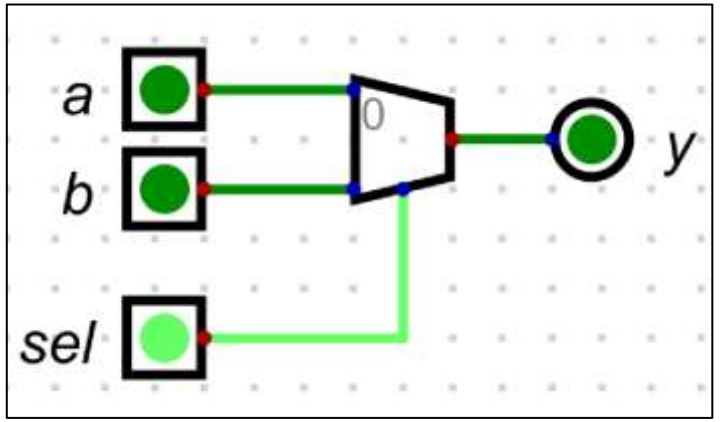




Dataflow Modeling: Conditional Operator

- 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

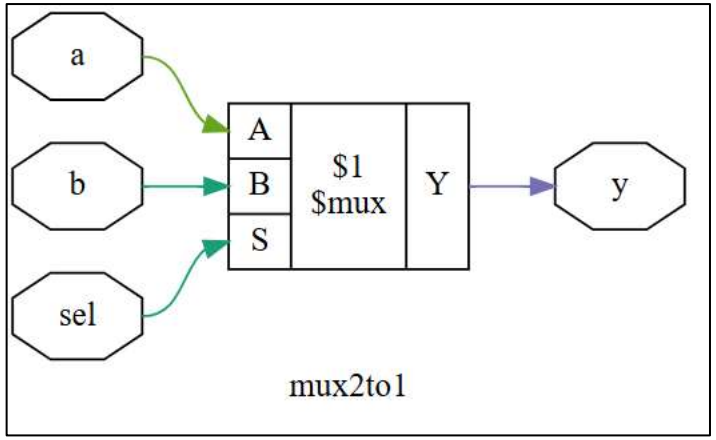
Dataflow Modeling : MUX with conditional

- 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
```

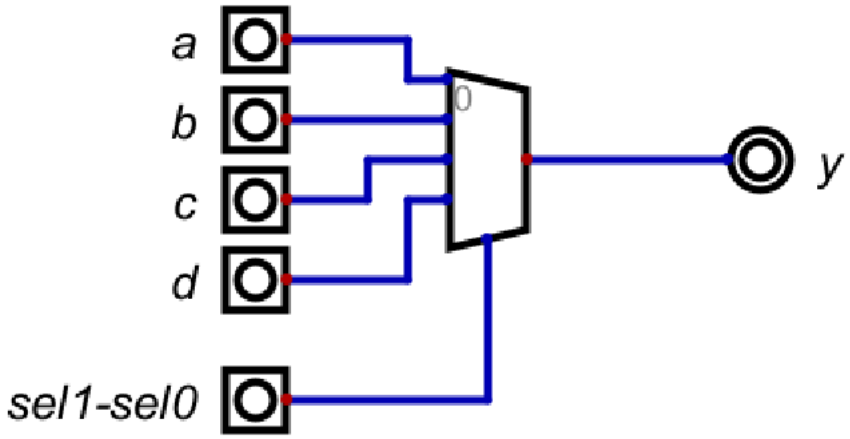
```
a b:sel y
0 1:1 1
0 1:0 0
```

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



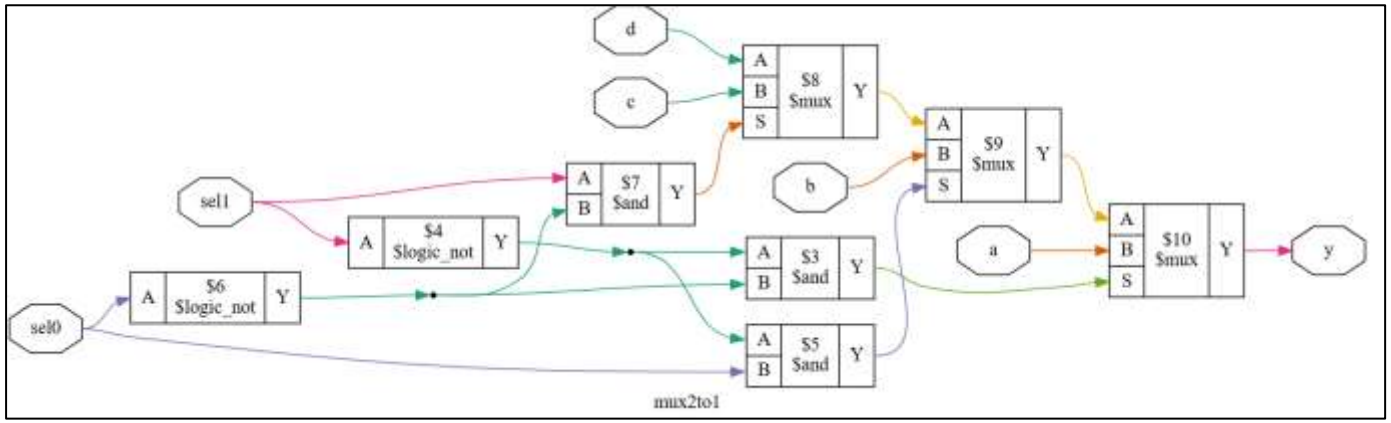
Dataflow Modeling : MUX4to1 with Conditional

- 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



Dataflow Modeling : MUX4to1 with Conditional

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

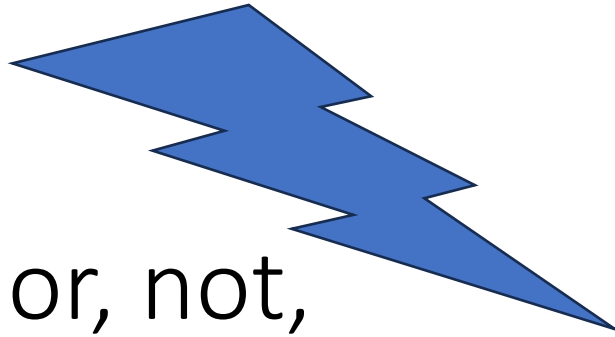
sel1	sel0	y
0	0	1
0	1	1
1	0	0
1	1	0

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

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

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

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

VERILOG MODELING



1. **Gate Level Modeling:** primitives (and, or, not, xor, etc), UDP

2. **Dataflow Modeling** : continuous assignment, logic, arithmetic and conditional operators

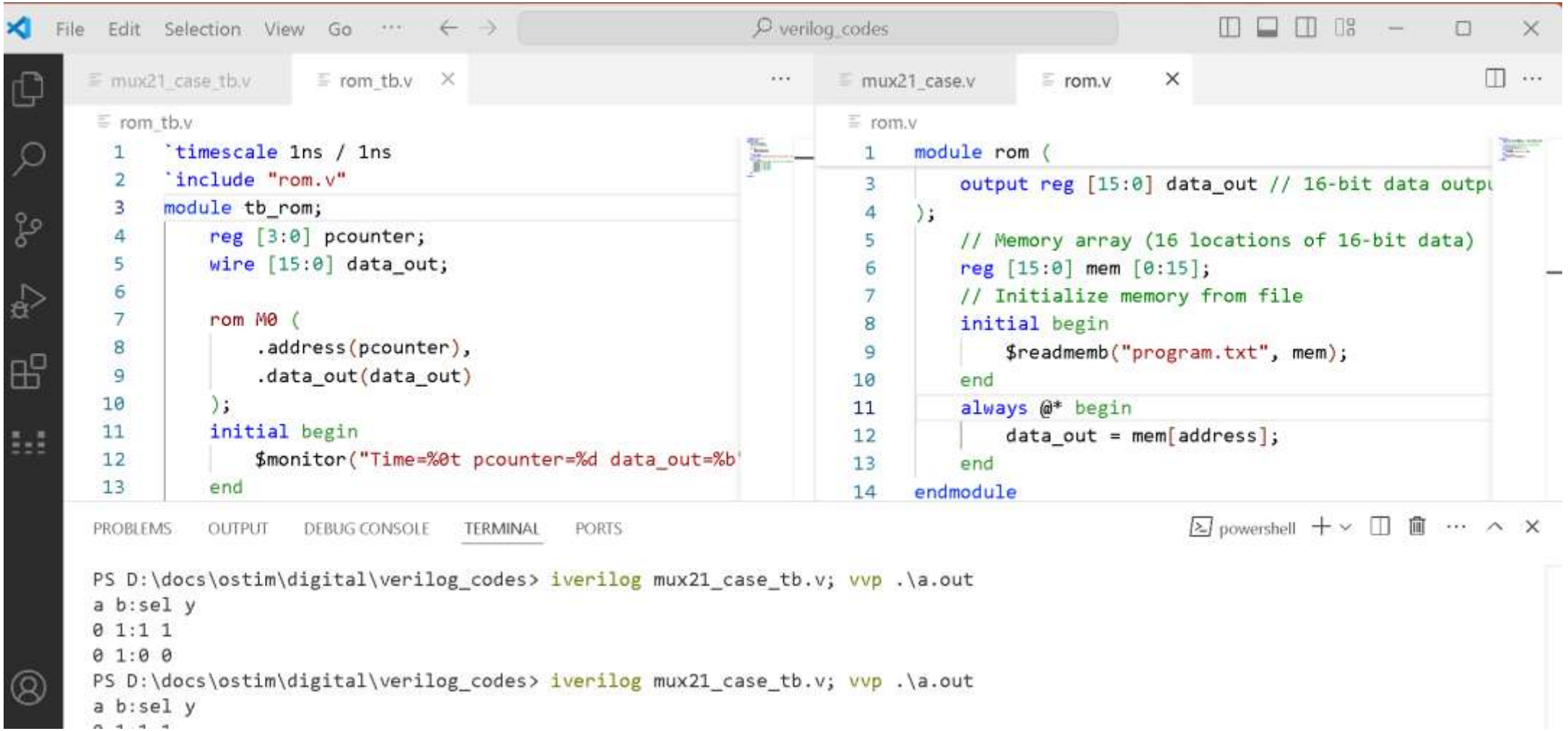
3. **Behavioral Modeling**: always @*, if-else, case,

EDAPLAYGROUND VS ICARUS VERILOG

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

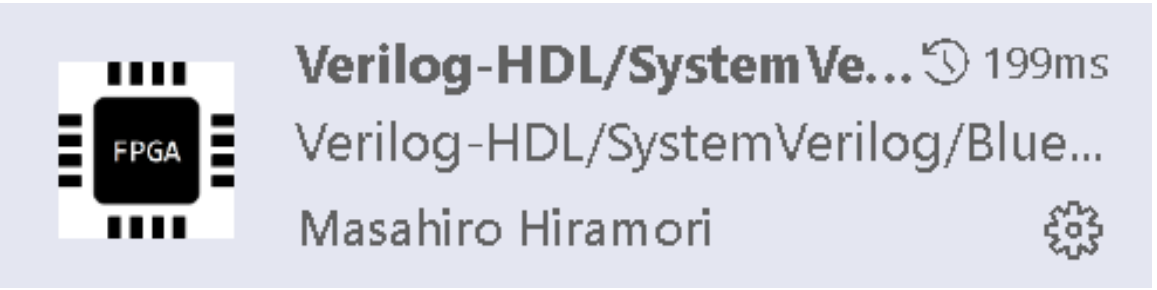
VSCode

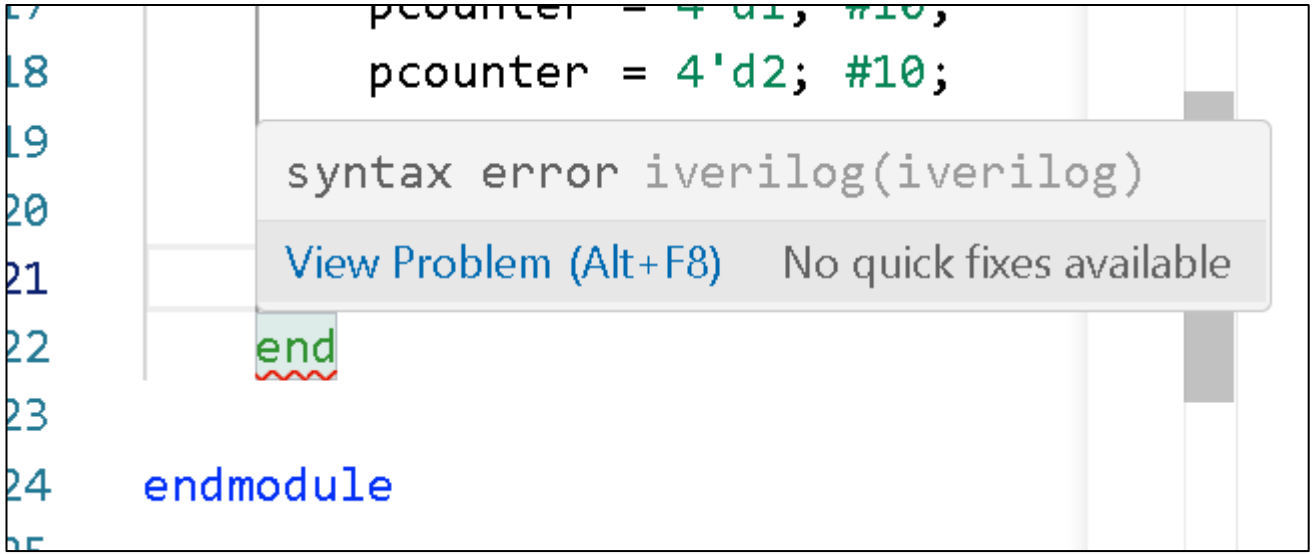
- **We can create an IDE similar to EDAPlayground by using VSCode**
- **Design file is at the right, testbench is at the left and command line at the bottom**



VSCode

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





Icarus Verilog

- **Icarus Verilog developed by Stephen Williams:**
- Icarus Verilog — Icarus Verilog documentation (steveicarus.github.io)

- **Windows compiled version :**
- iverilog-v12-20220611-x64_setup [18.2MB]
- Icarus Verilog for Windows (bleyer.org)

- **Command line**
- **iverilog -o mux21case.out mux21_case.v mux21_case_tb.v will create mux21case.out. default output file is a.out**
- **vvp mux21case.out will simulate the verilog code and print simulation results**
- **We can include design file into testbench using include, and vvp a.out to make or life easy**
- **iverilog mux21_case_tb.v; vvp .\a.out**

GTKWave

- GTKWave (sourceforge.net)

Icarus Verilog Documentation

- [Icarus Verilog — Icarus Verilog documentation (steveicarus.github.io)]

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

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

User Defined Primitives (UDP)

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

```
iverilog .\mux21_udp_tb.v; vvp .\a.out
time absel y
0 xxx x
1 000 0
2 001 0
3 010 0
4 011 1
5 100 1
6 101 0
7 110 1
8 111 1
```

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

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

Behavioral Modeling

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

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

10.0 Procedural Blocks

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

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

Procedural Statements : always @ ()

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

10.2 Sensitivity Lists

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

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

Procedural Statements: if-else

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

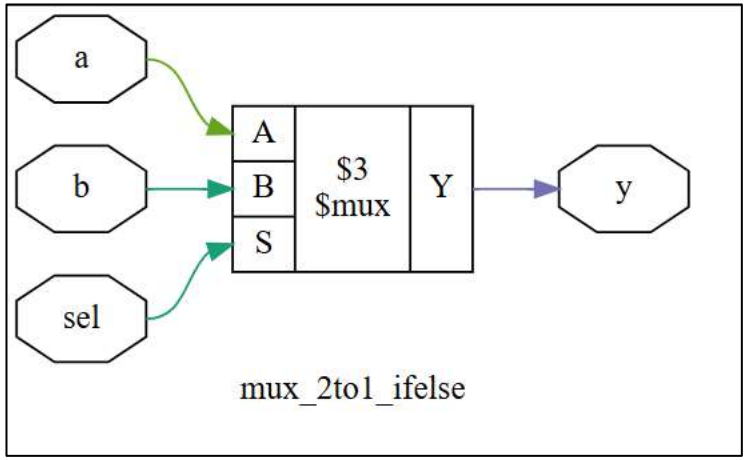
```
module mux_2to1_ifelse (
   input wire a,        // Input 0
   input wire b,        // Input 1
   input wire sel,      // Select line
   output reg y         // Output
);

   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



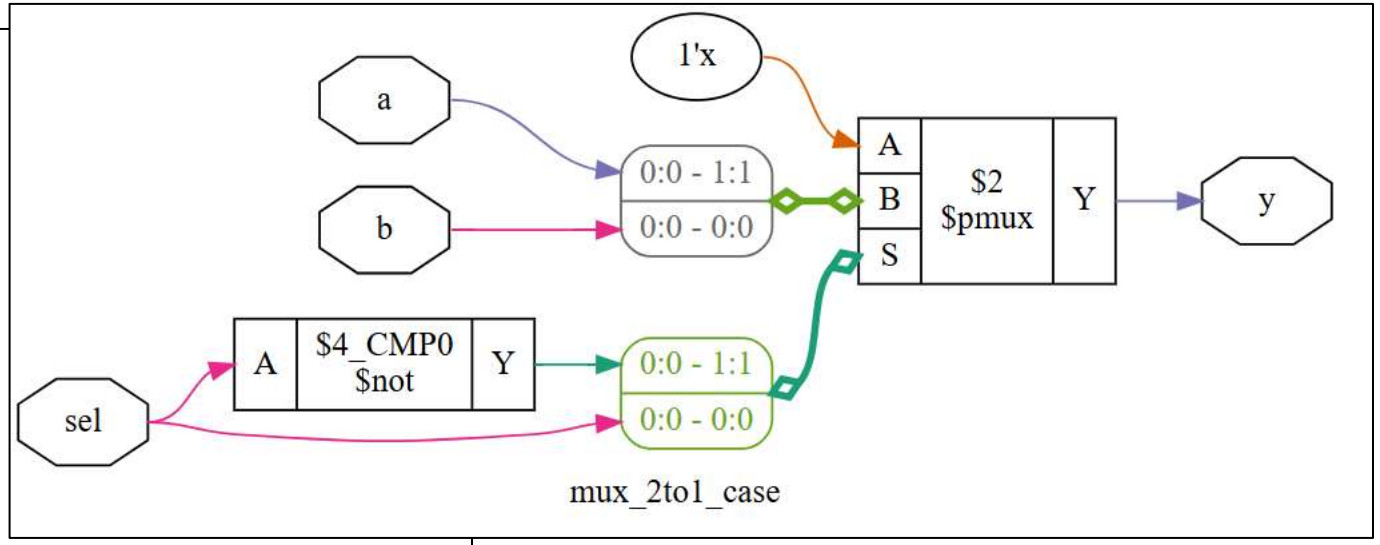
Procedural Statements: if-else

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

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

Procedural Statements: case

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

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

endmodule
```



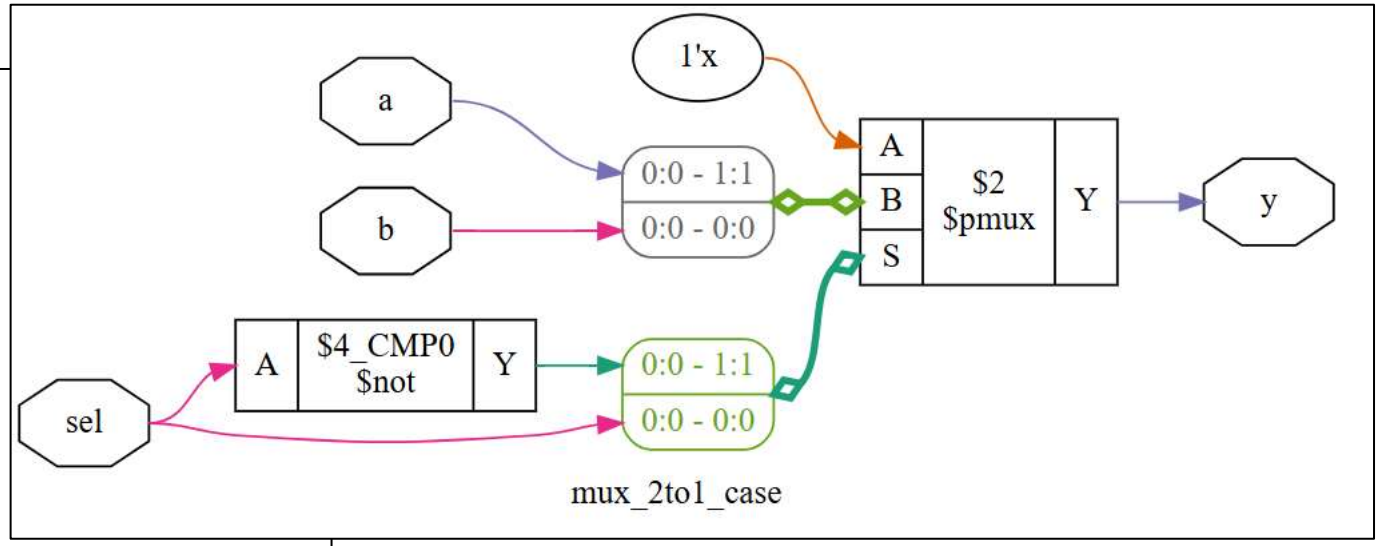
`case` (*expression*)
- *case_item*: — *statement or statement_group*
- *case_item*, *case_item*: — *statement or statement_group*
- `default`: — *statement or statement_group*

`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

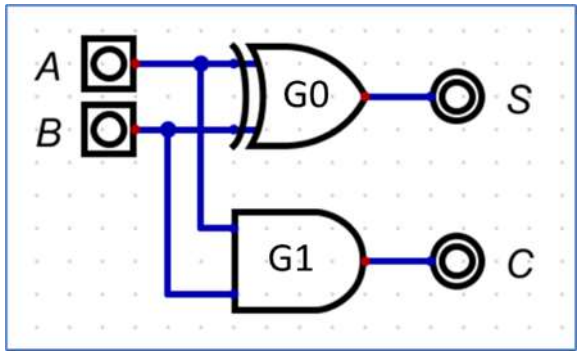
```
module mux_2to1_case (
   input wire a,        // Input 0
   input wire b,        // Input 1
   input wire sel,      // Select line
   output reg y         // Output
);

   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).

Half Adder with Primitives

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

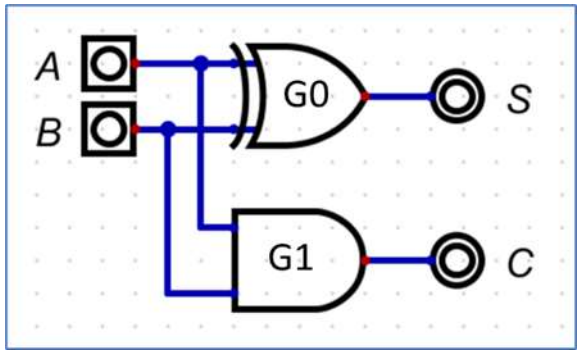



```
//half adder
module halfadder(
  output S,C,
  input A,B
);
  xor G0 (S,A,B);
  and G1 (C,A,B);
endmodule
```

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

Half Adder with User Defined Primitives

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

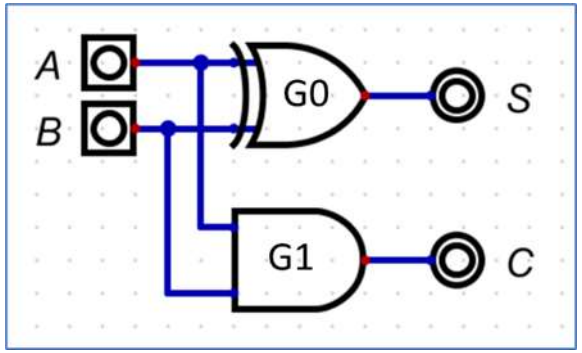



```
//half adder
module halfadder(
  output S,C,
  input A,B
);
  xor G0 (S,A,B);
  and G1 (C,A,B);
endmodule
```

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

Half Adder with Data Flow Modeling: Logic Operators

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

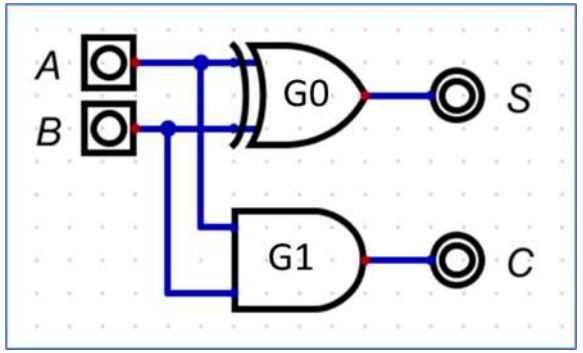



```
module halfadder
(
  output C,S,
  input A,B);

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

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

Half Adder with Data Flow Modeling: Arithmetic Operators

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




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

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

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

Half Adder with Data Flow Modeling: Conditional Assignment

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

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

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

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

Binary Numbers in Verilog

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

Port Range -Vector

5.2 Port Declarations

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

Mux 4 to 1 Tetbench

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

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

4bit adder

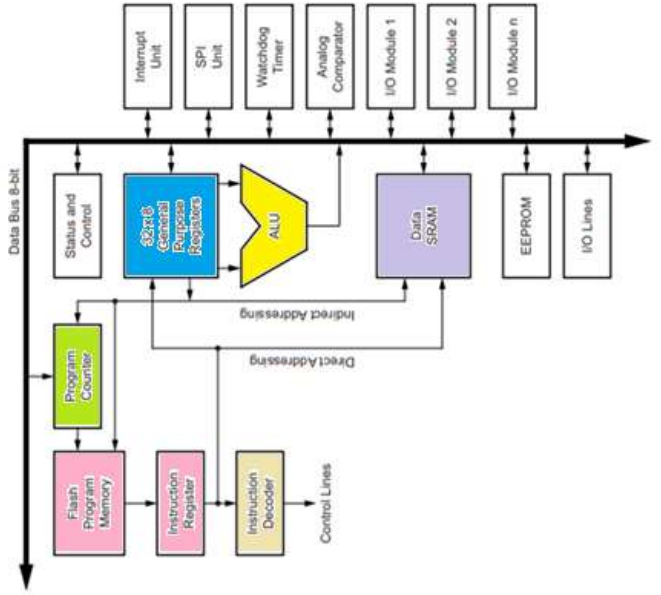
```
module adder4bit (
    input wire [3:0] a,       // 4-bit input a
    input wire [3:0] b,       // 4-bit input b
    input wire cin,           // Carry-in input
    output wire [3:0] sum,    // 4-bit sum output
    output wire cout          // Carry-out output
);
    // Calculate the sum and carry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

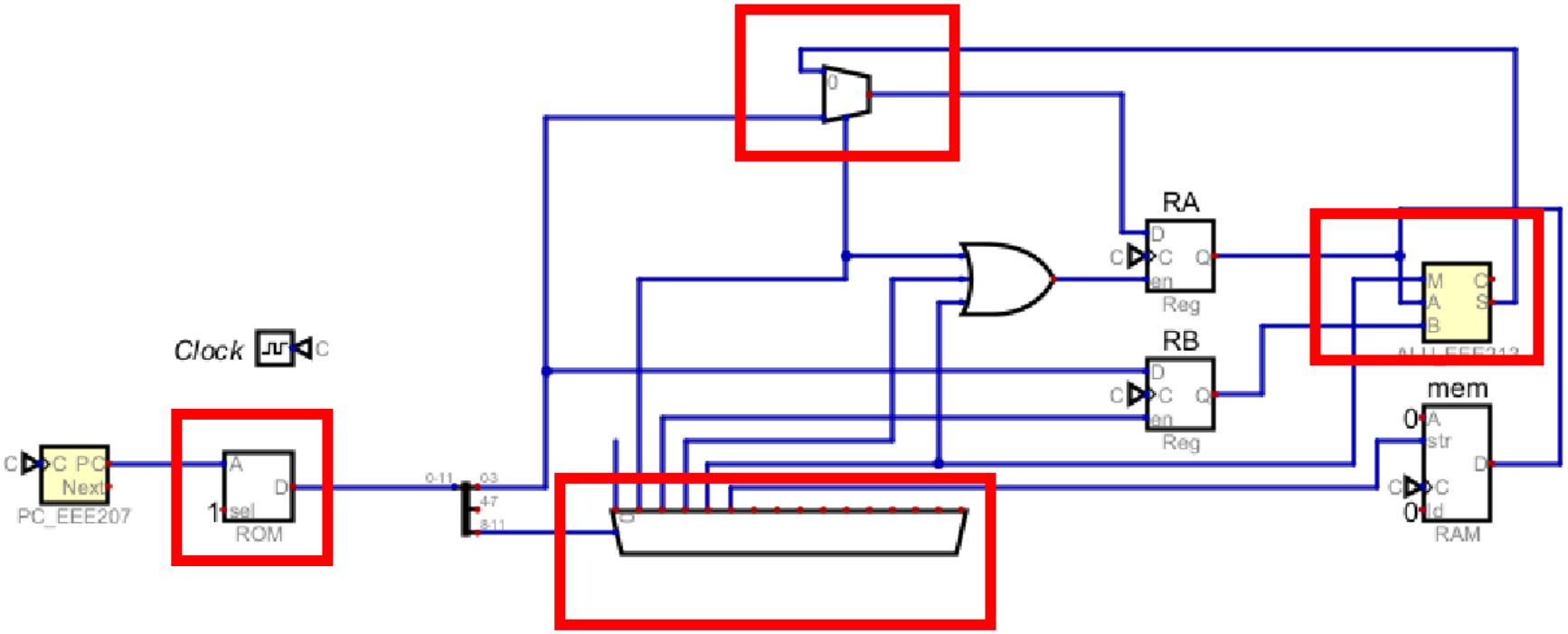
4bit adder testbench

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

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

MCU Blocks





Combinational

- **MUX**
- **Decoder**
- **ROM**
- **Async RAM**
- **ALU**

Sequential

- **Program Counter**
- **Registers**
- **Synch RAM**

2 to 1 MUX

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

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

endmodule
```

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

4 bit decoder

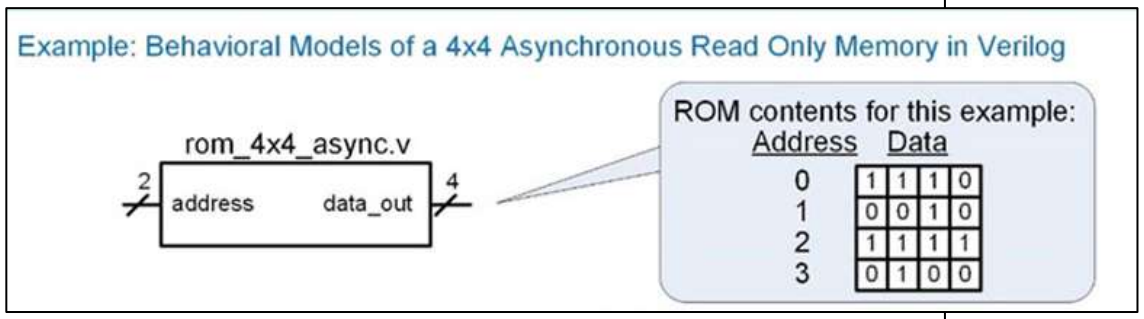
```
module decoder4to16 (
    input wire [3:0] in,     // 4-bit input
    output reg out0,         // 16 individual
1-bit outputs
    output reg out1,
    output reg out2,
    output reg out3,
    output reg out4,
    output reg out5,
    output reg out6,
    output reg out7,
    output reg out8,
  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

4 bit decoder Testbench

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

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

ROM Initialized



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

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

ROM Initialized Testbench

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

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

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

ROM from File

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

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

ROM from File Testbench

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

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

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

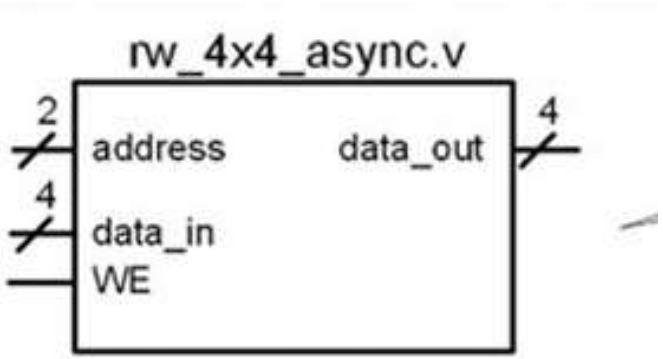
Async RAM

- We can WRITE/READ to RAM
- Address is the input

- Asynchronous DRAM (ADRAM)
- example arduino

- Synchronous DRAM (SDRAM)
- notebooks

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



Contents to be written:

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

Async RAM Test Bench

- READ or WRiTE to RAM by using WE write enable

```
`timescale 1ns / 1ns
`include "memory.v"
module tb_memory;
    reg we;
    reg [4:0] address;
    reg [15:0] data_in;
    wire [15:0] data_out;
    memory M0 (
        .we(we),
        .address(address),
        .data_in(data_in),
        .data_out(data_out)
    );
    initial begin
        $monitor("Time=%0t we=%b address=%d data_in=%b
data_out=%b", $time, we, address, data_in, data_out);
        // Write data to addresses
        we = 1'b1;
        address = 5'd0; data_in = 16'b1110000100000010; #10;
        address = 5'd1; data_in = 16'b1110001100010100; #10;
        address = 5'd2; data_in = 16'b0001111100000001; #10;
        address = 5'd3; data_in = 16'b1001000000000010; #10;
        // Switch to read mode
        we = 1'b0;
```

Async RAM Test Bench

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

```
iverilog memory_tb.v; vvp .\a.out
Time=0 we=1 address= 0 data_in=1110000100000010 data_out=zzzzzzzzzzzzzzzz
Time=10 we=1 address= 1 data_in=1110001100010100 data_out=zzzzzzzzzzzzzzzz
Time=20 we=1 address= 2 data_in=0001111100000001 data_out=zzzzzzzzzzzzzzzz
Time=30 we=1 address= 3 data_in=1001000000000010 data_out=zzzzzzzzzzzzzzzz
Time=40 we=0 address= 0 data_in=1001000000000010 data_out=1110000100000010
Time=50 we=0 address= 1 data_in=1001000000000010 data_out=1110001100010100
Time=60 we=0 address= 2 data_in=1001000000000010 data_out=0001111100000001
Time=70 we=0 address= 3 data_in=1001000000000010 data_out=1001000000000010
```

8-bit ALU for ADD, SUB

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

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

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

NOTICE that 8'd2, 4bit size DECIMAL 2

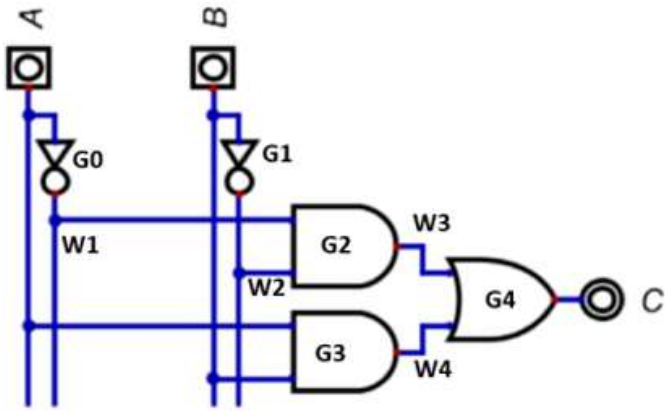
? :	sel?m:n	conditional operator; if sel is true, return m: else return n

4.11 Literal Integer Numbers

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

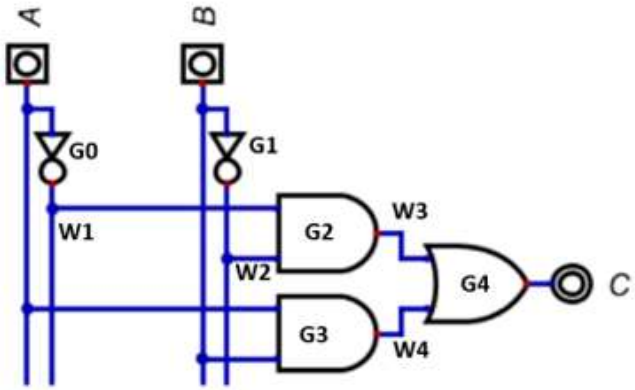
EXERCISES- Primitives

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



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

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



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

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

EXERCISES-testbench

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

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

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

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

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

EXERCISES-testbench

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

```
`timescale 1ns/1ps

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

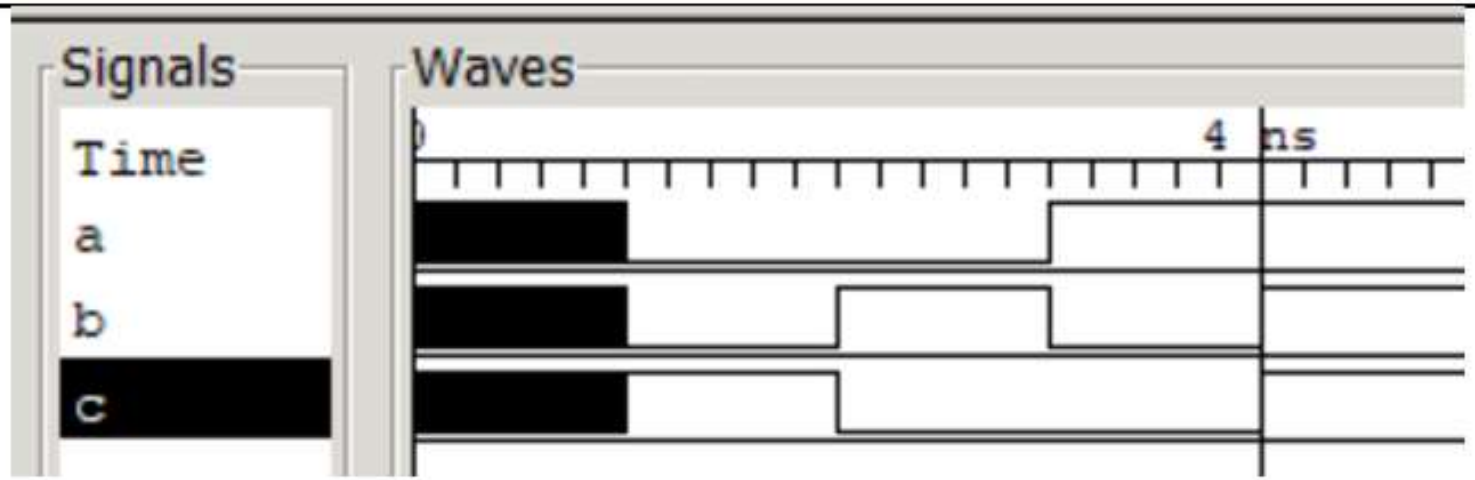
```
`timescale 1ns/1ns

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

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

EXERCISES-testbench

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

EXERCISES-Waveform



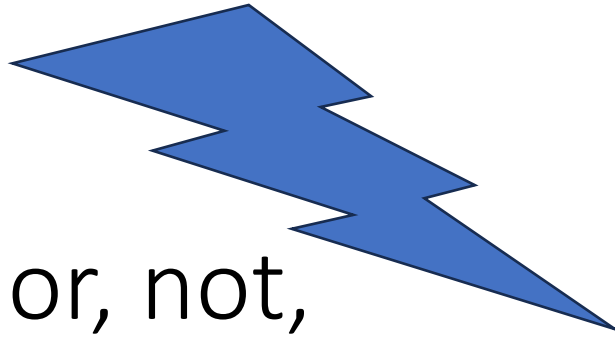
8. (10 points) What are the initial values of a, b and c in simulation

a=_____b=_____c=_____

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

a=_____b=_____c=_____

VERILOG MODELING



1. **Gate Level Modeling:** primitives (and, or, not, xor, etc), UDP

2. **Dataflow Modeling** : continuous assignment, logic, arithmetic and conditional operators

3. **Behavioral Modeling**: always @*, if-else, case,

EXERCISES- Assignment with operators

```
module sample_module(
   input wire a, b,        // 1-bit inputs
   output wire logic_out,   // 1-bit output for logical operations
   output wire [1:0] arith_out // 2-bit output for arithmetic operations
);

   // Logical operations
   assign logic_out = a & b;  // Logical AND

   // Arithmetic operations
   assign arith_out = a + b;  // Arithmetic addition

endmodule
```

EXERCISES-Conditional Assignment

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

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

endmodule
```

EXERCISES: **always @*** and if-else

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

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

endmodule
```

EXERCISES-Case

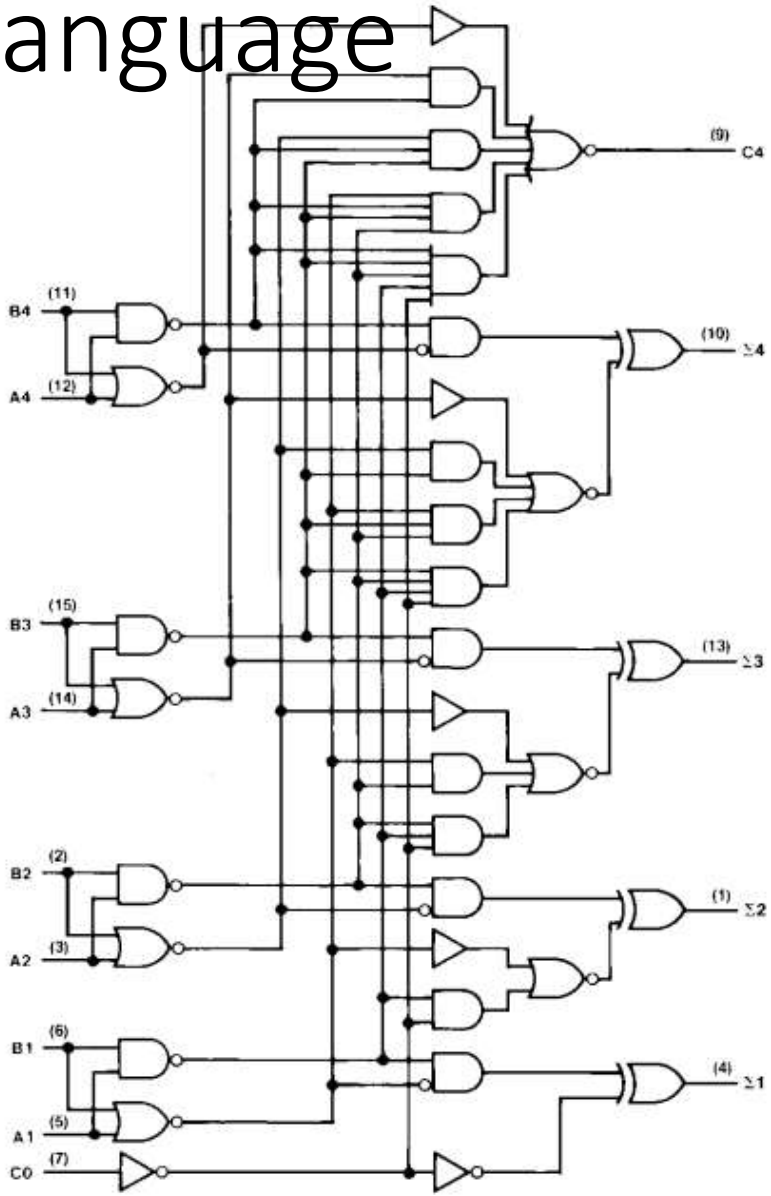
```
module mux_2to1_case (
    input wire a,           // Input 0
    input wire b,           // Input 1
    input wire sel,         // Select line
    output reg y            // Output
);
    always @* begin
        case (sel)
            1'b0: y = a;    // sel is 0
            1'b1: y = b;    // sel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

endmodule
```

Verilog : Hardware Description Language

1. The initial idea was to create a language that could **simulate digital circuits** efficiently and be easy to use.

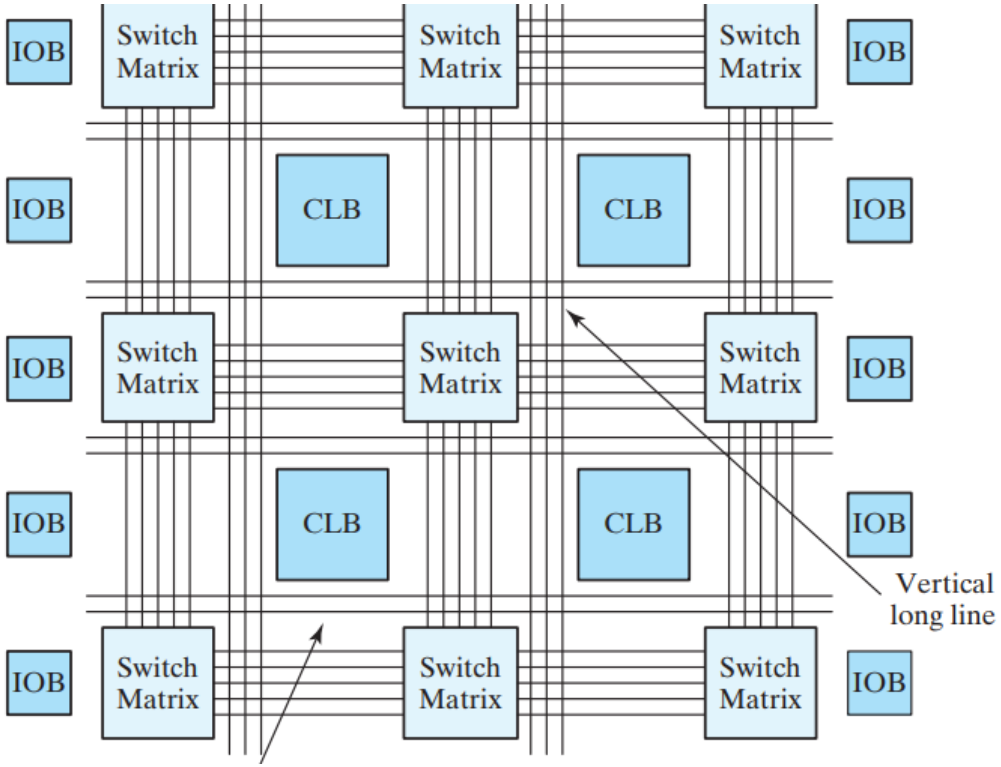
2. Later Verilog used to design Logic Circuits using **Synthesis** of HDL code

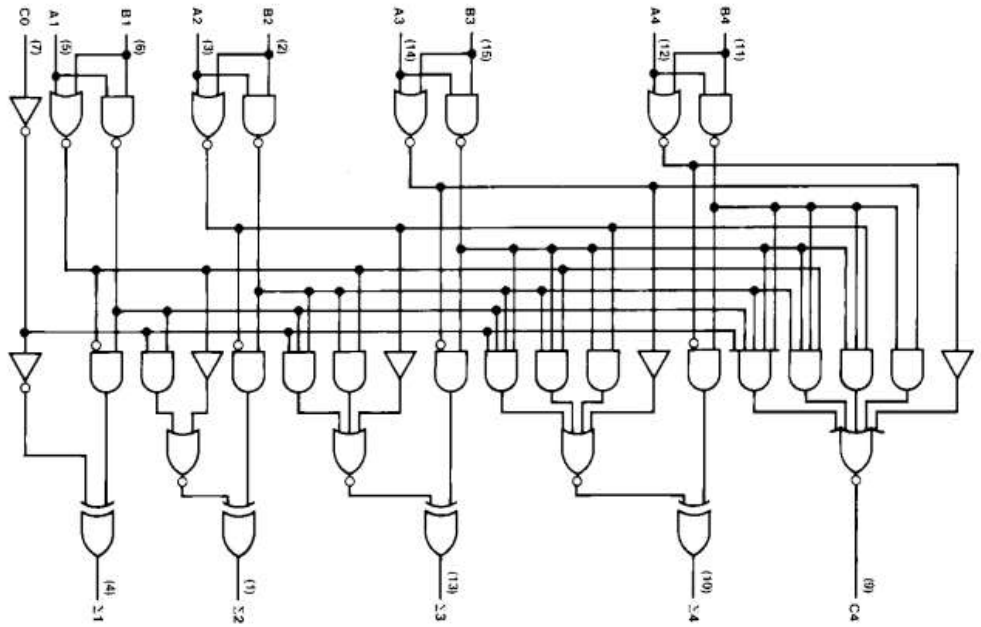
3. Finally Verilog can be used to develop **FPGA Applications**



Logic Circuits vs FPGA

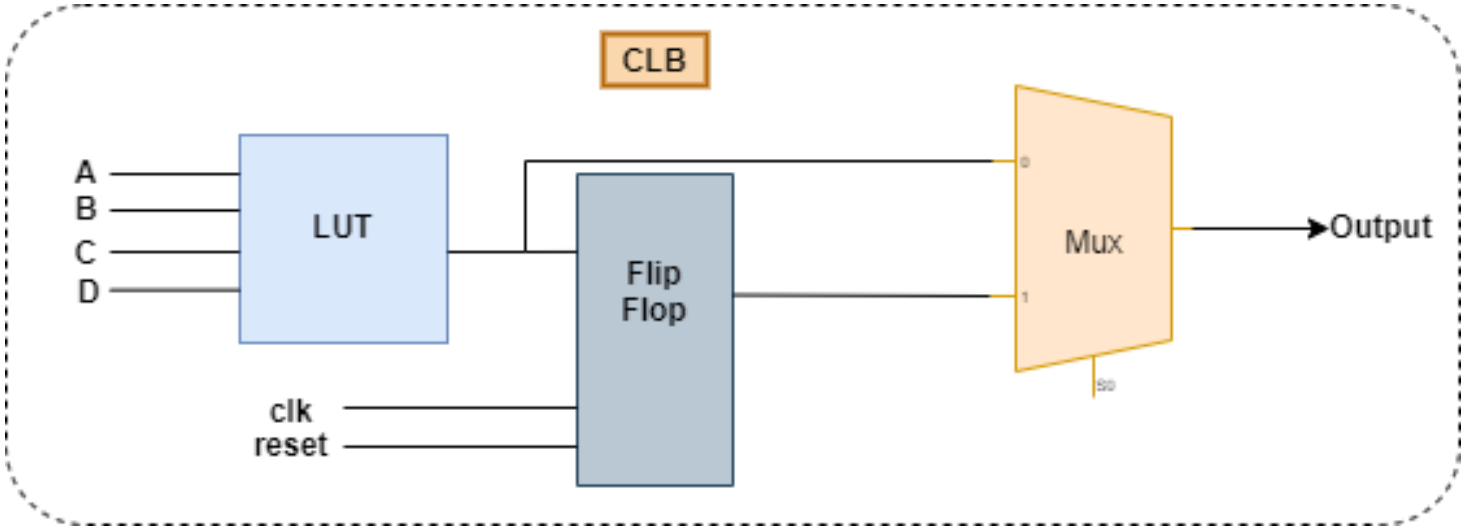
- Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects.
- FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes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

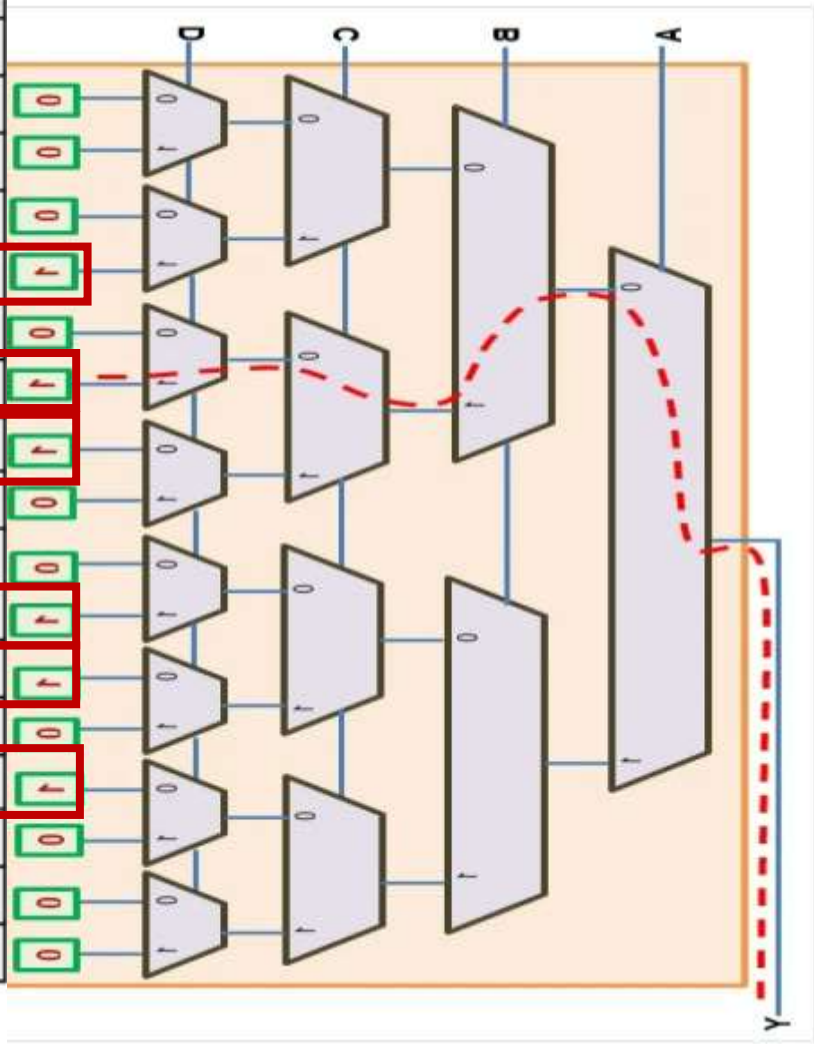


Implementation of Logic Functions using LUT

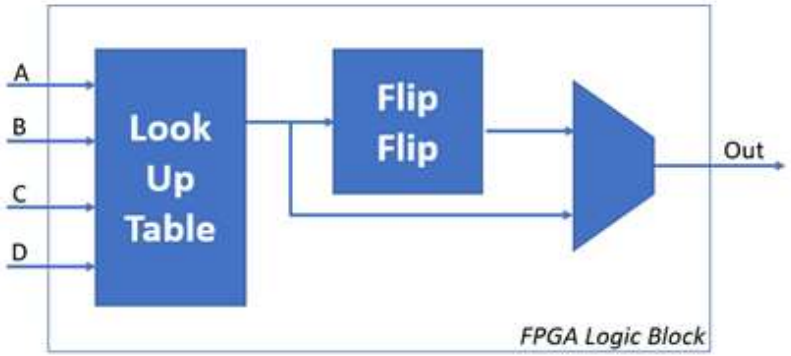
- Suppose we want to realize a Boolean Function of four input variables A, B, C and D using a 4-input LUT
- While realizing this function using an FPGA, A, B, C, and D will be the inputs to LUT.
- Next, the values of the output variable for each of their combination (available in the last column of the 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).

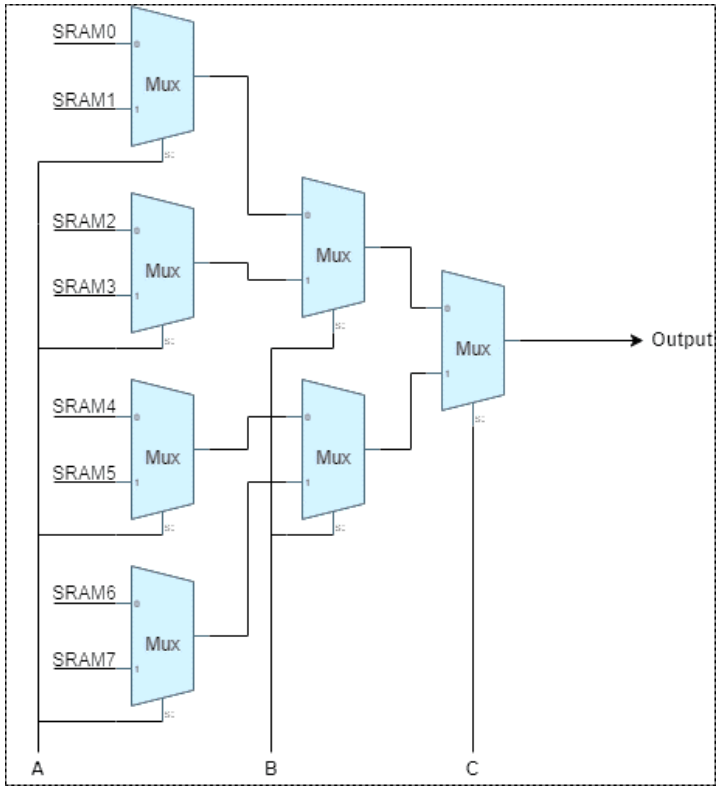
Truth Table

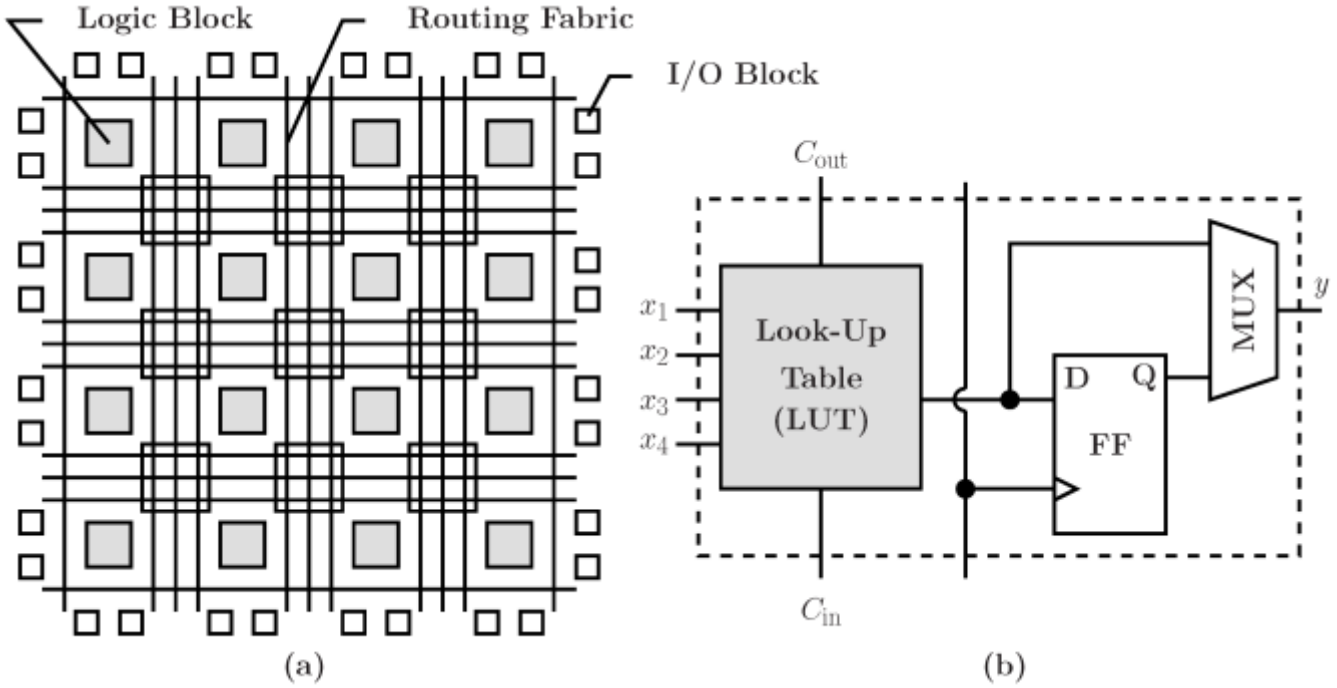
Inputs				Output
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



Tang Nano 9k –FPGA







Tang Nano 9k –FPGA

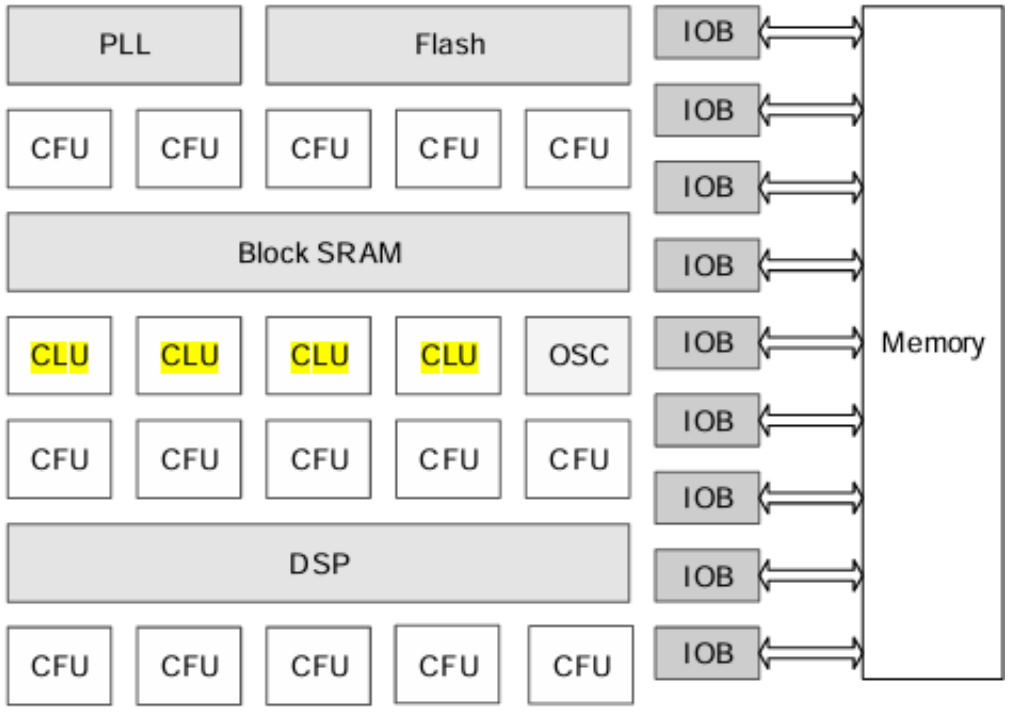


Figure 2-4 CFU Structure View

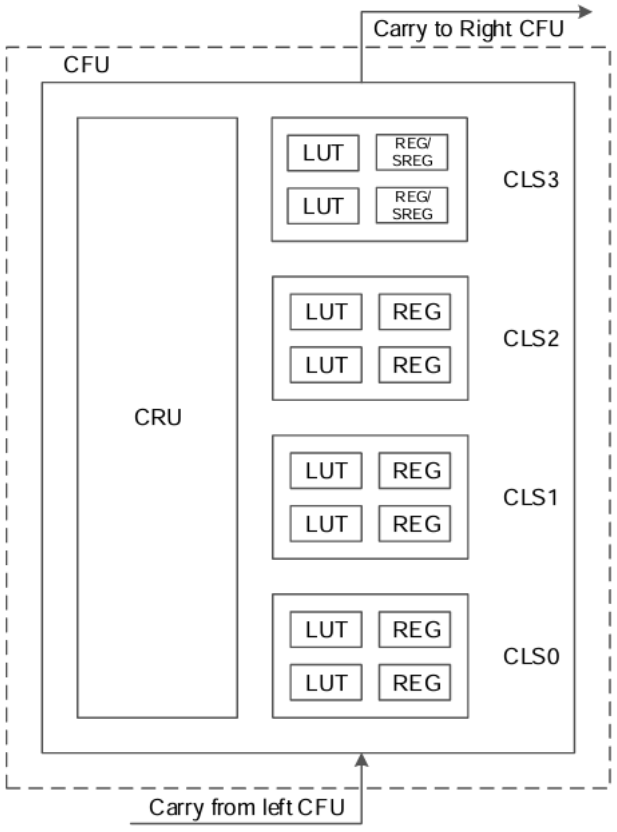
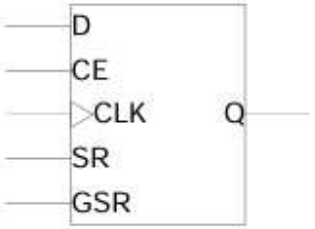
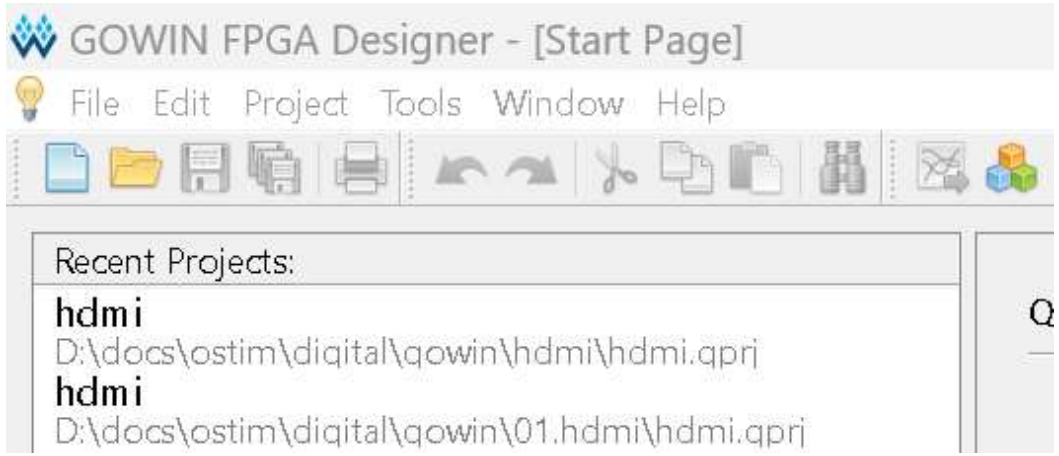


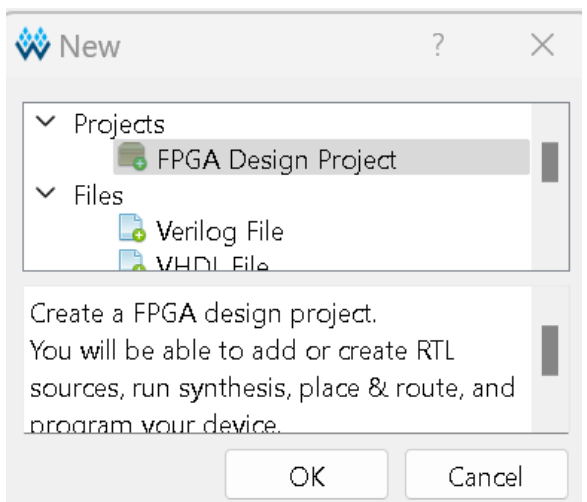
Figure 2-2 Register in CFU

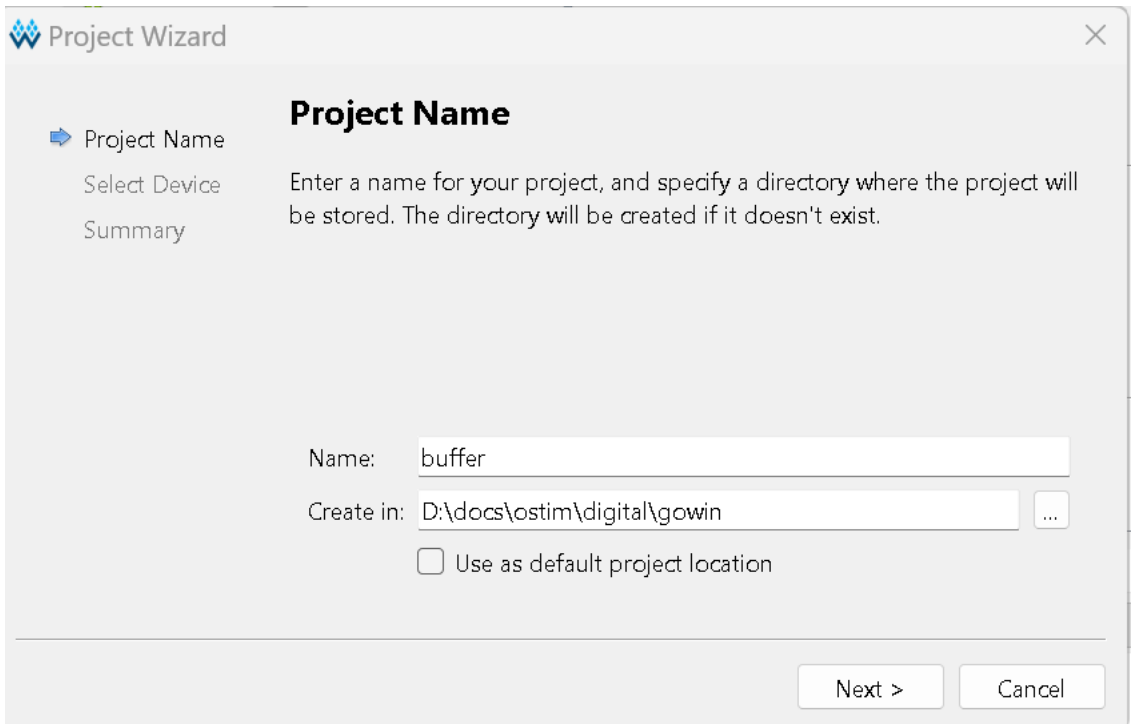


Tang Nano 9k –New Project

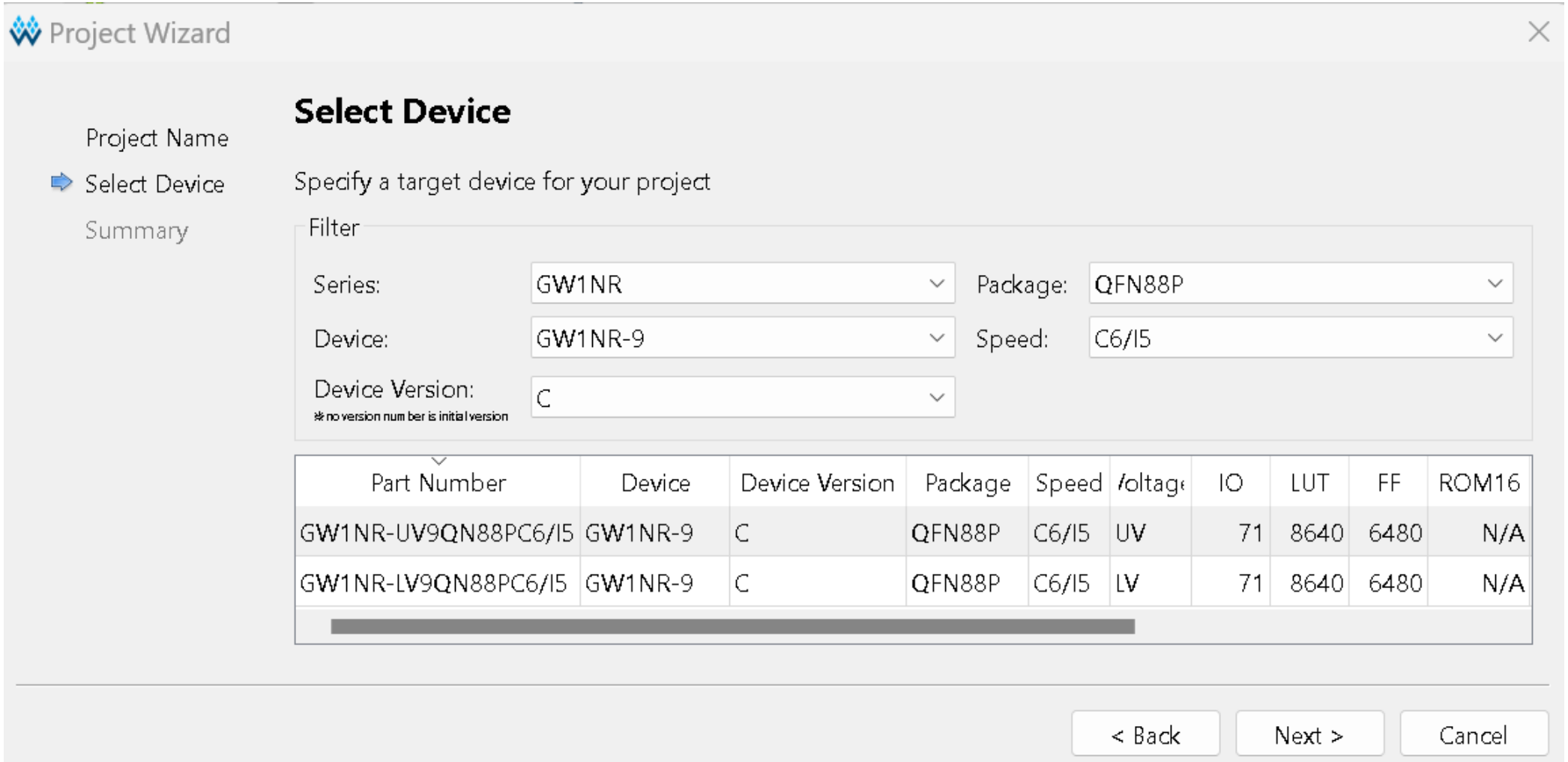
[Light LED - Sipeed Wiki]



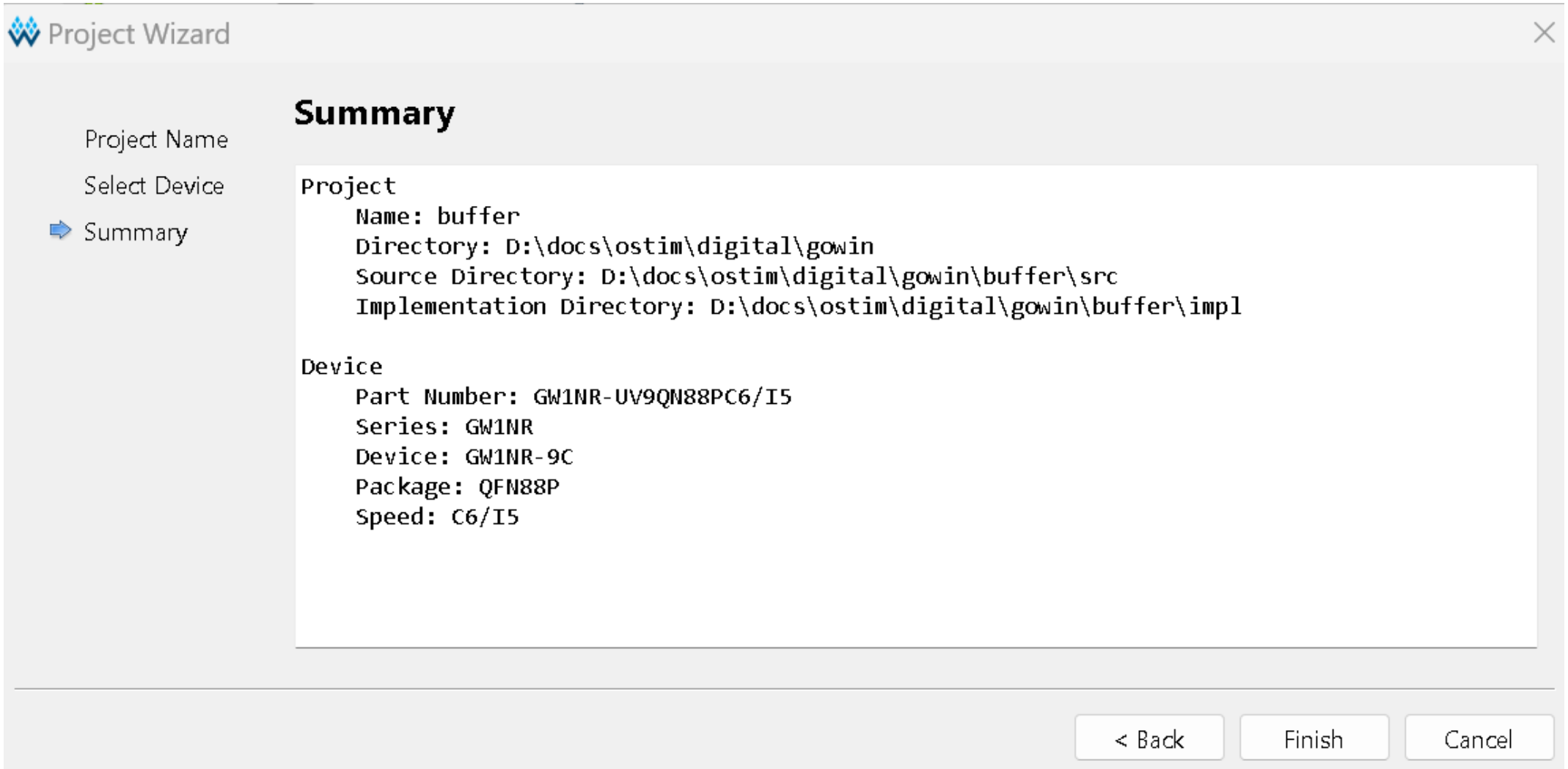




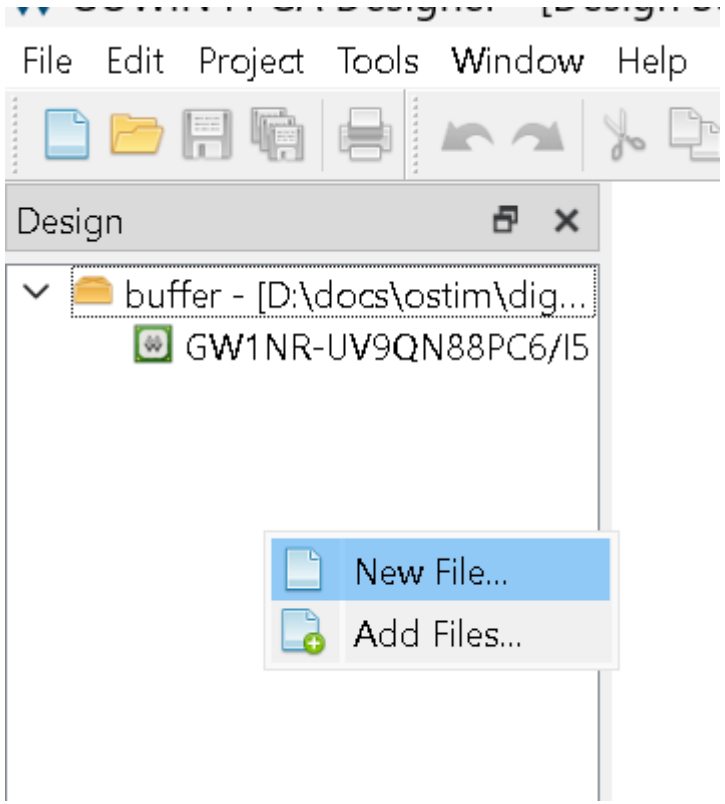
Tang Nano 9k –Select Device

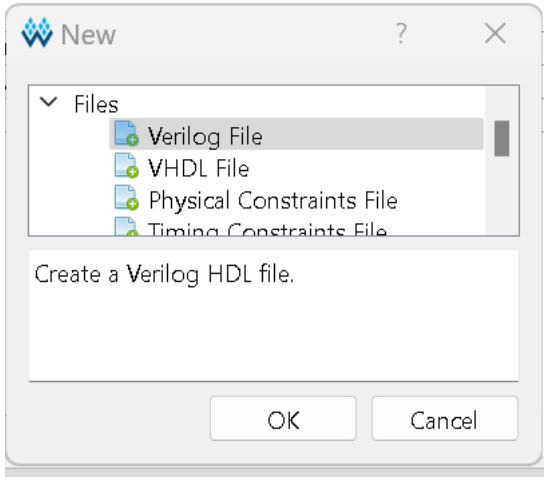


Tang Nano 9k –Summary

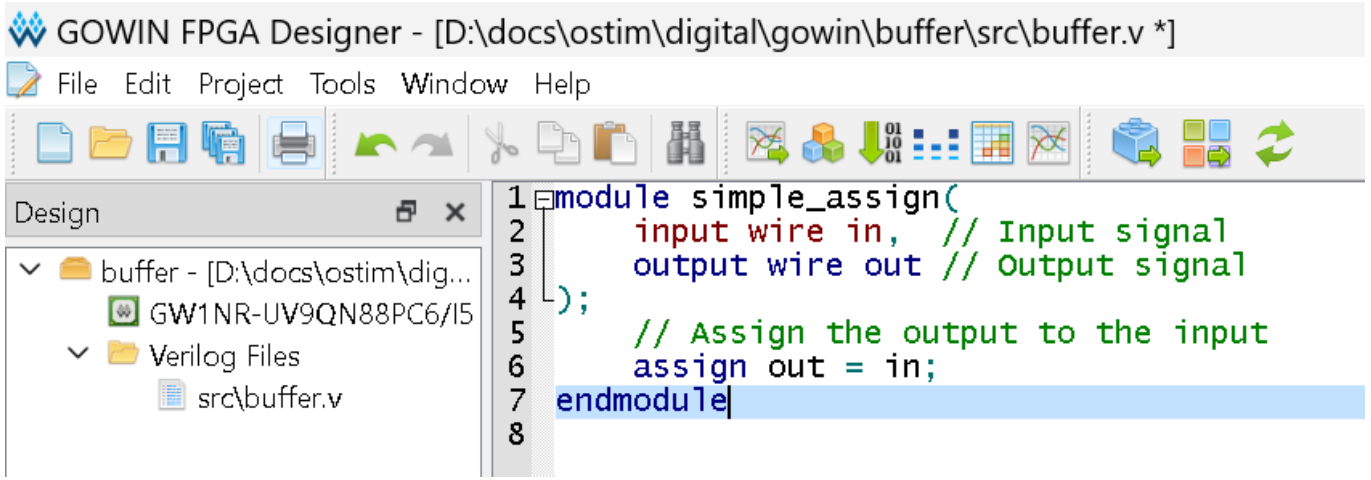


Tang Nano 9k –Verilog File

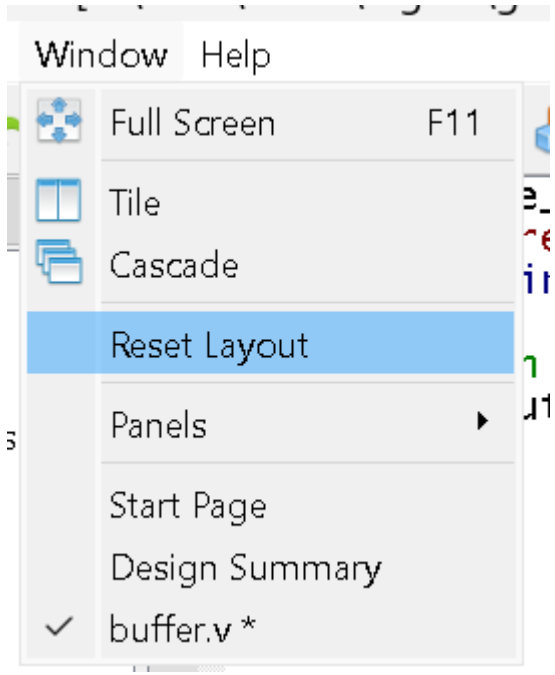


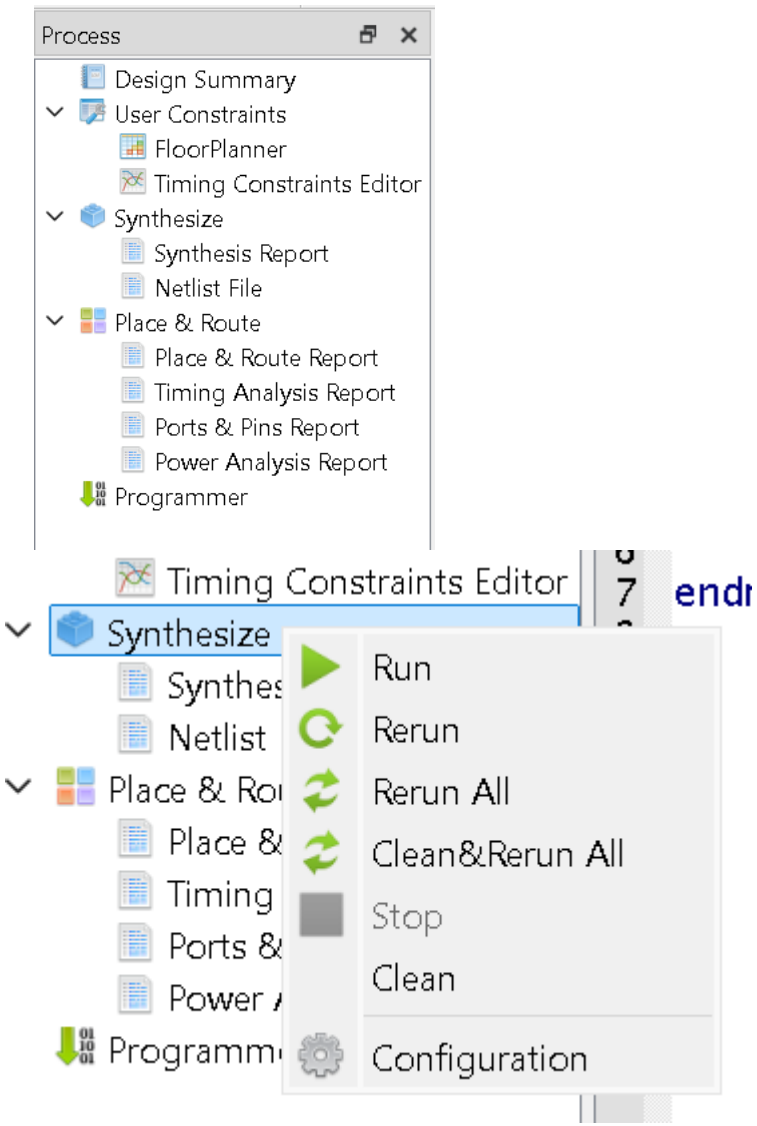


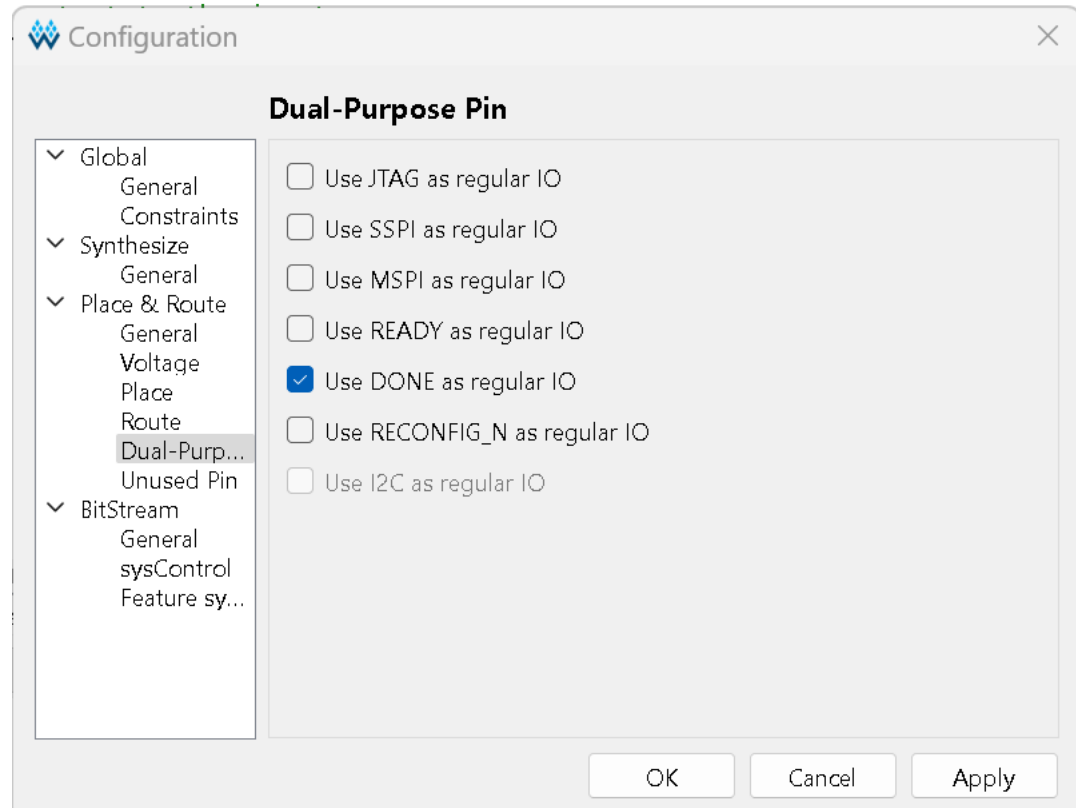

```
module simple_assign(
    input wire in,  // Input signal
    output wire out // Output signal
);
    // Assign the output to the input
    assign out = in;
endmodule
```



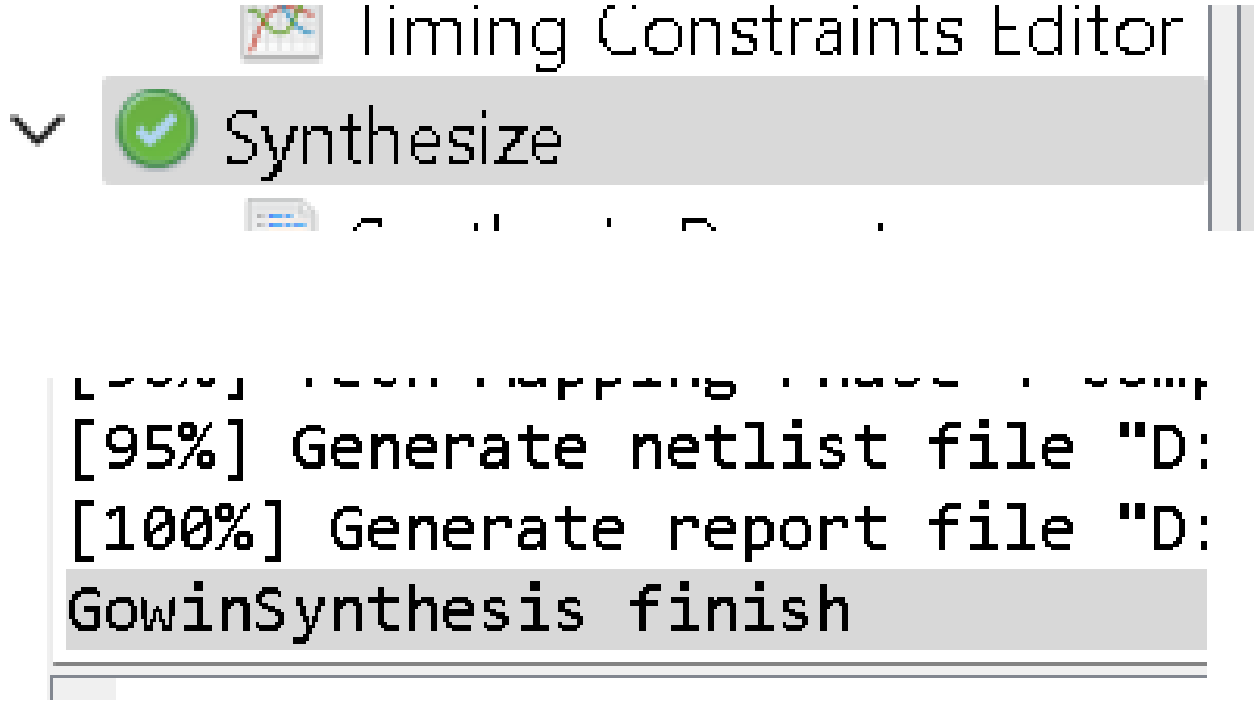
Tang Nano 9k –Synthesize Configuration

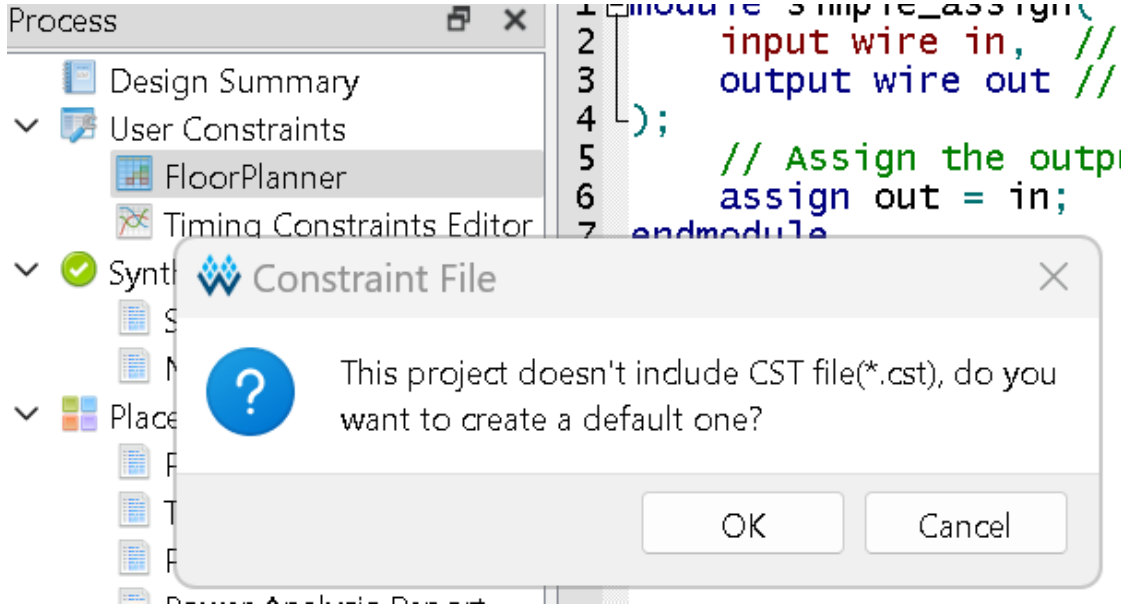




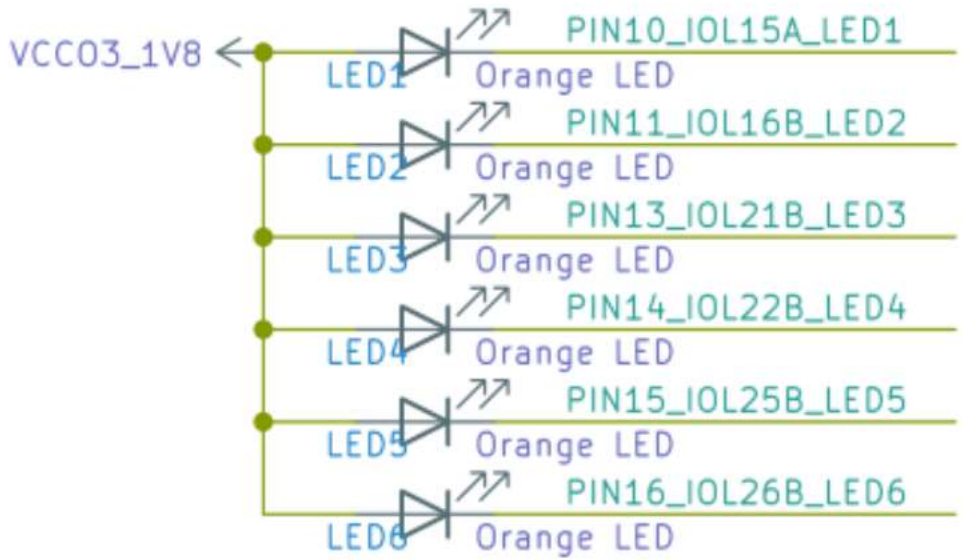


Tang Nano 9k –Synthesize





Tang Nano 9k –Floor Planner

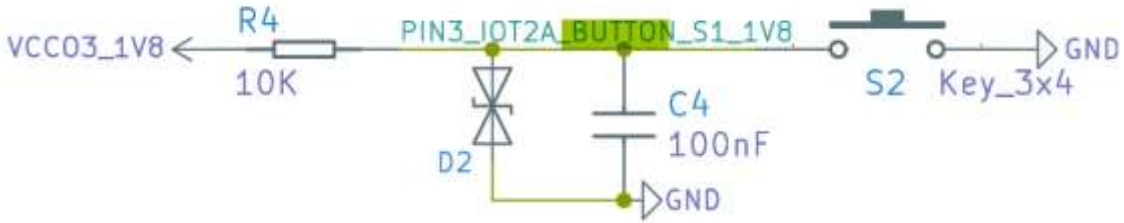


Summary | Netlist

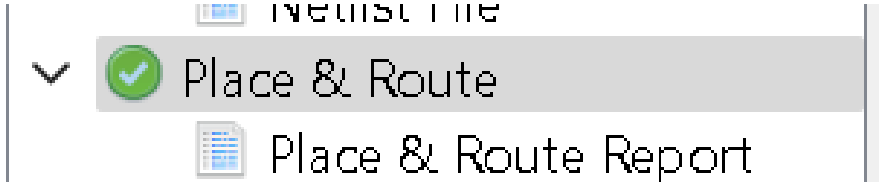
I/O Constraints

	Port	Direction	Diff Pair	Location
1	in	input		3
2	out	output		10

Me... | I/O C... | Primitive C... | Group C... | Resource R... | Clock Net C...

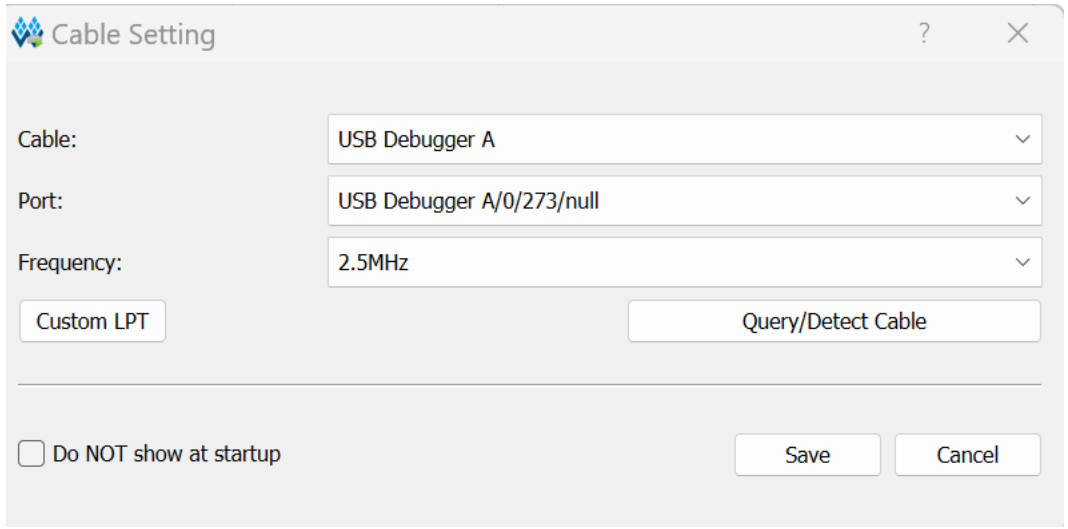


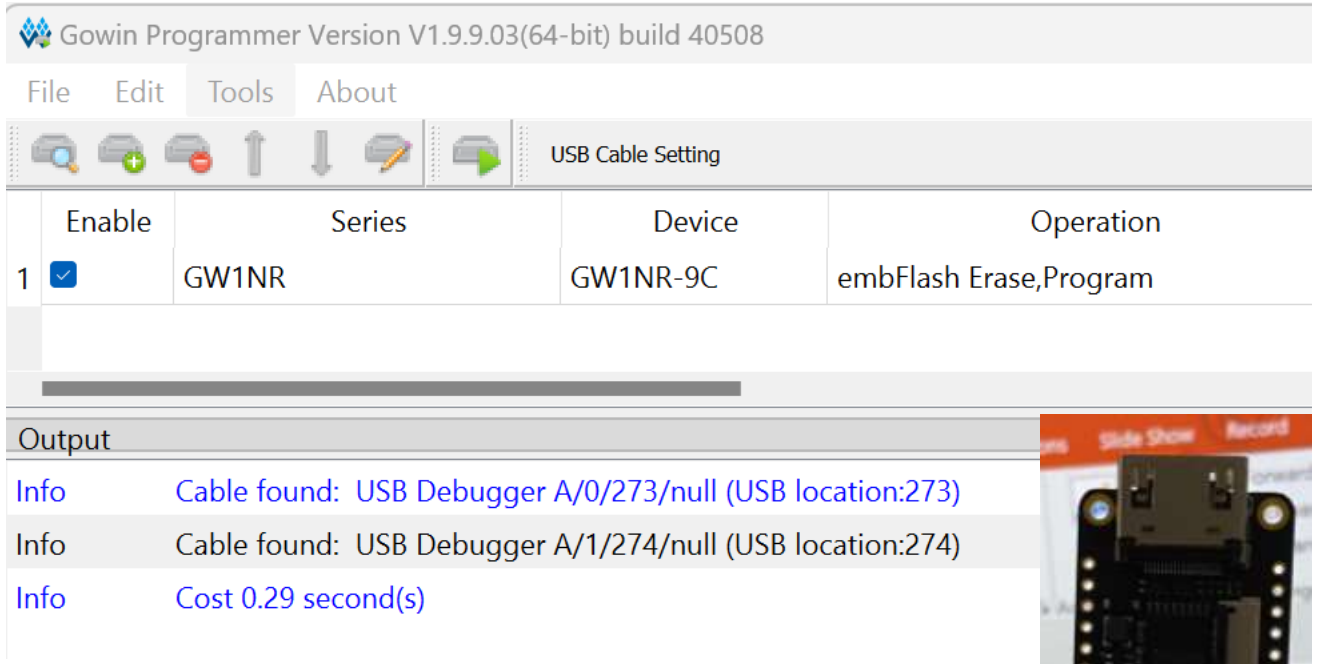
Tang Nano 9k –Place and Route




```
Generate file "D:\docs\ostim\digital\gowin\buffer\impl\pnr\buffer.tr.html" completed
Fri Oct 25 22:31:22 2024
```

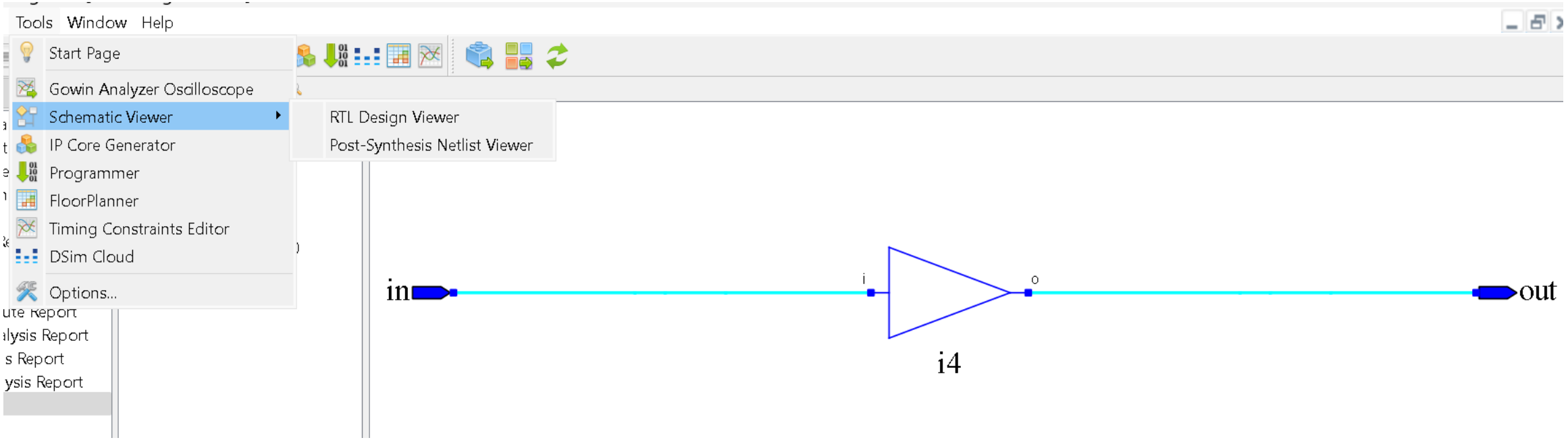
Tang Nano 9k –Programmer





Info Operation "embFlash Erase,Progra
Info Status Code is: 0x0003F020
Info User Code: 0x0000AA94
Info Program Finished!
Info Finished.
Info Cost 8.74 second(s)

Tang Nano 9k –Schematic Viewer

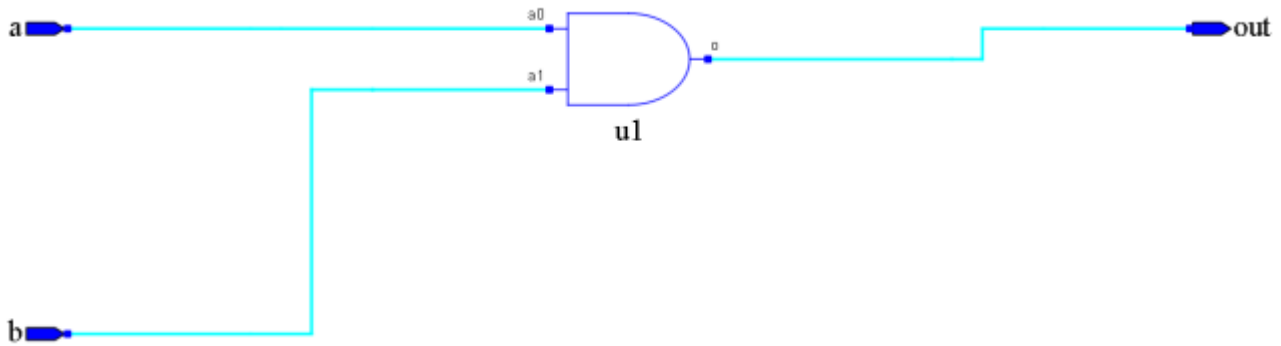


Tang Nano 9k –Menus

- New project
- Select FPGA
- Design TAB : New file – Verilog – write your code inside
- Process TAB (Reset Layout if needed)
- Synthesis- Configuration – Dual Purpose – Use DONE –Click Synthesis
- Floor Planner – CST file YES – Package view and IO Constraint
- Select pins (Butoon 3,4, LEDS 10-11-13-14-15-16)- SAVE
- Place & Route
- Programmer – USB cable detected – Embedded Flah Mode
- Upload Flash

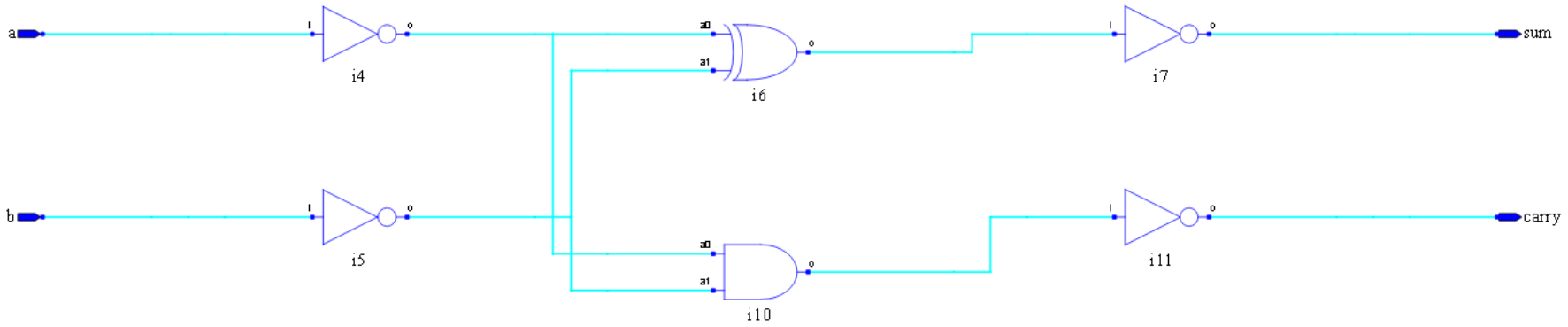
Tang Nano 9k –And Gate

```
1  module and_gate_primitive(
2      input wire a, b,   // 1-bit
3      output wire out    // 1-bit
4  );
5
6      // Using the AND gate prim
7      and u1 (out, a, b);
8
9  endmodule
10
```



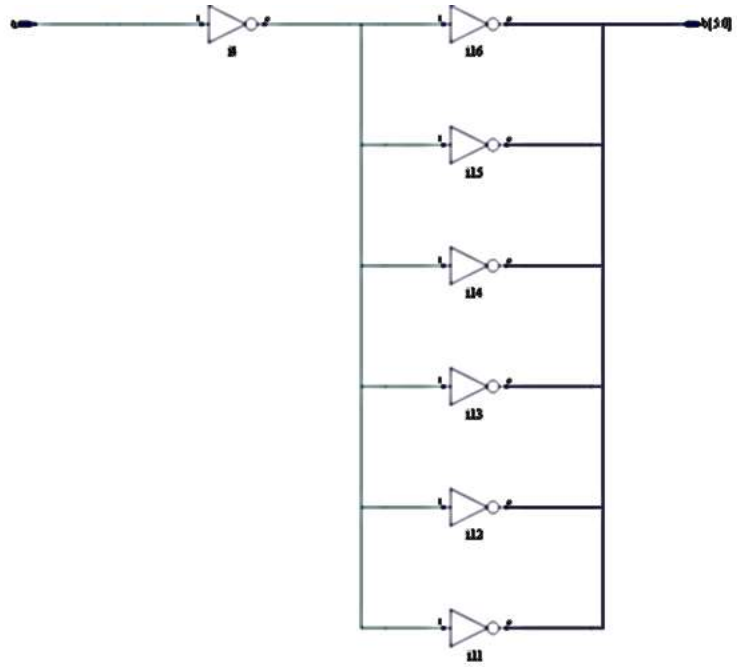
Tang Nano 9k –Half Adder

```
module half_adder(
    input wire a, b,        // 1-bit inputs
    output wire sum, carry  // 1-bit outputs
);

    // Sum is the XOR of the inputs
    assign sum = ~(~a ^ ~b);

    // Carry is the AND of the inputs
    assign carry = ~(~a & ~b);

endmodule
```



Tang Nano 9k –if else

```
module ifelseexample(
   input wire a,         // 1-bit input
   output reg [5:0] b     // 6-bit output
);

   always @* begin
      if (a == 0)
         b = 6'b000000;
      else
         b = 6'b111111;
   end

endmodule
```

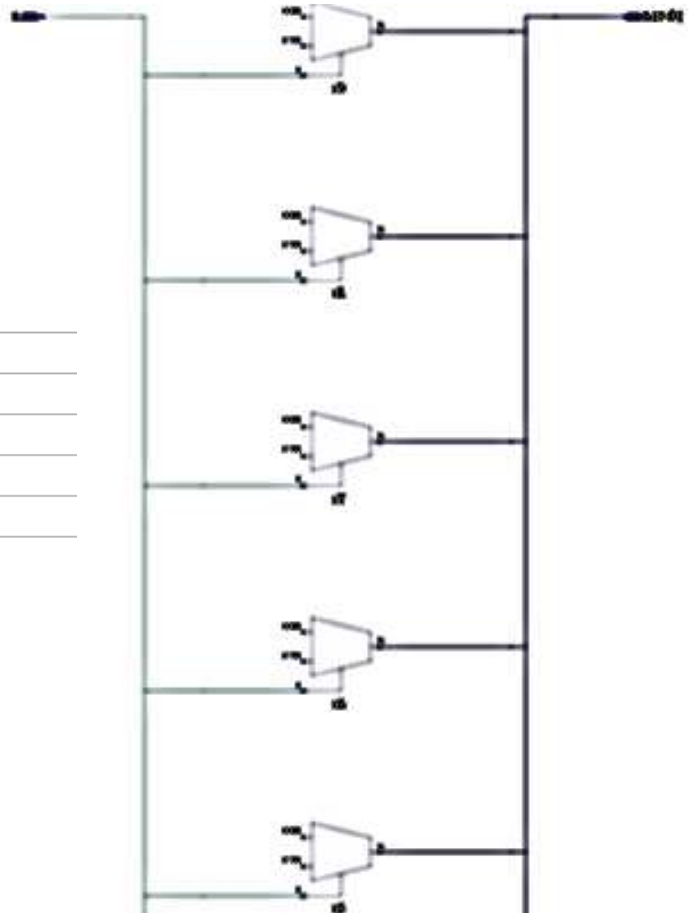


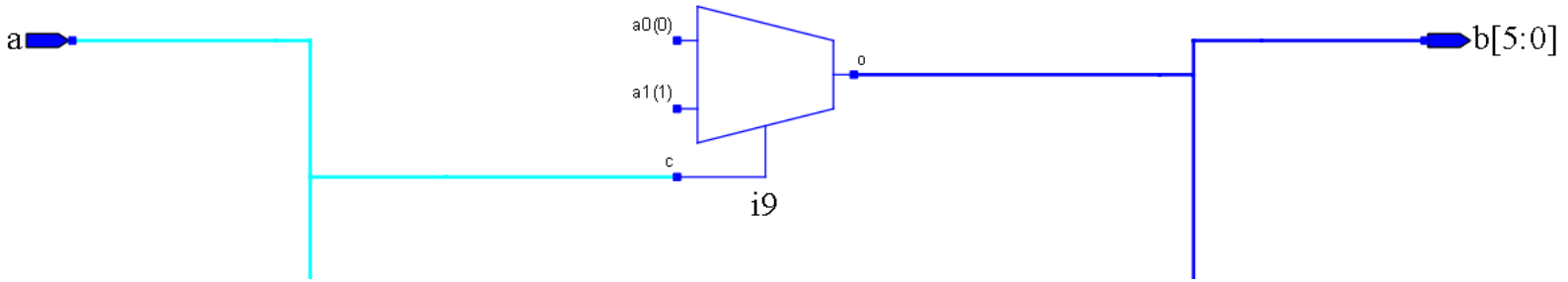
Tang Nano 9k –Case

```
module conditional_assign_case(
   input wire a,         // 1-bit input
   output reg [5:0] b     // 6-bit output
);

   always @* begin
      case (a)
         1'b0: b = 6'b000000;
         1'b1: b = 6'b111111;
         default: b = 6'b000000;  // Optional
default case for completeness
      endcase
   end

endmodule
```

Resource Usage Summary

Resource	Usage
I/O Port	7
I/O Buf	7
IBUF	1
OBUF	6





Equality Check

- MIDTERM question

I/O Constraints

	Port	Direction	Diff Pair	Location
1	button3	input		3
2	button4	input		4
3	led	output		10

```
module button_led(
   input wire button3,
   input wire button4,
   output reg led
);
always @(*) begin
   case (button3)
      button4: led = 0; // button3 and button4 are equal
      default: led = 1;     // button3 and button4 are not equal
   endcase
end
endmodule
```

2 to 4 Decoder

- 2 to 4 decoder using tang nano

I/O Constraints

	Port	Direction	Diff Pair	Location
1	button3	input		3
2	button4	input		4
3	led[0]	output		10
4	led[1]	output		11
5	led[2]	output		13
6	led[3]	output		14

```
module decoder_2to4 (
    input wire button3,
    input wire button4,
    output reg [3:0] led
);

always @(*) begin
    case ({~button4, ~button3}) // Invert the inputs since they
are active low
        2'b00: led = 4'b1110; // Active low output
        2'b01: led = 4'b1101;
        2'b10: led = 4'b1011;
        2'b11: led = 4'b0111;
        default: led = 4'b1111; // All LEDs off
    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 of the CLOCK**

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

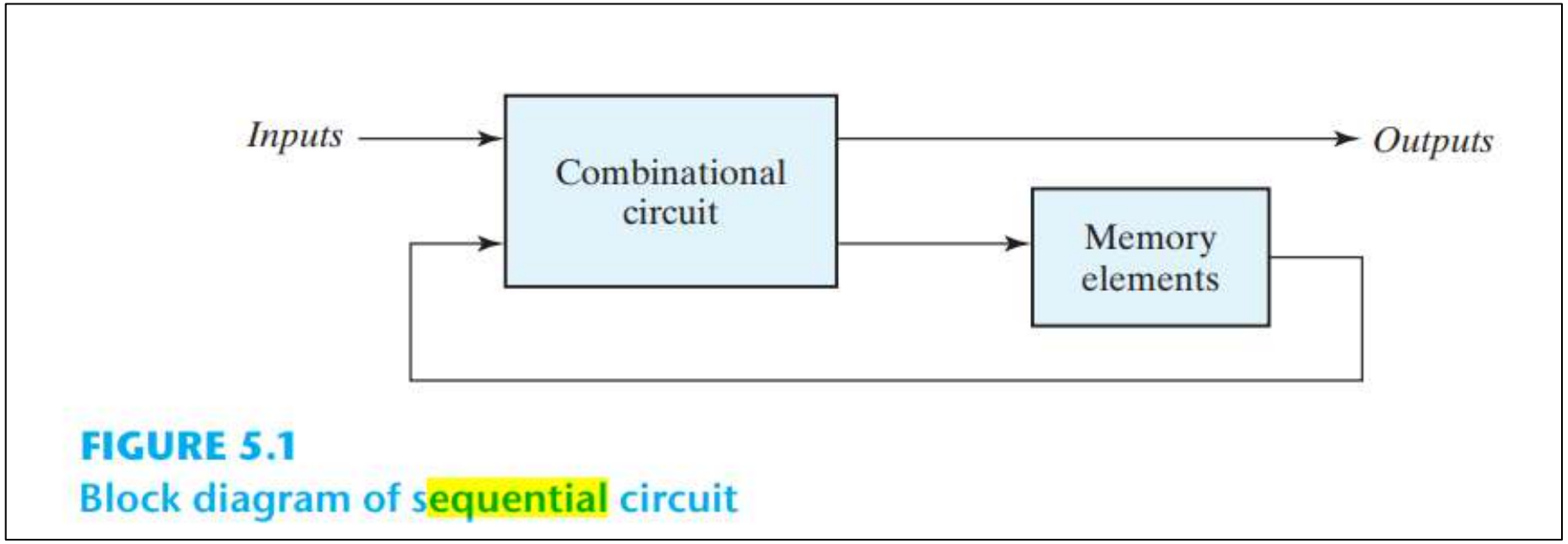


FIGURE 5.1
Block diagram of sequential circuit

Verilog Sequential Circuit Design: D-FlipFlop

- inputs are data, clock and reset. output is Q

- Sequential circuits performs actions **at clock signals**.

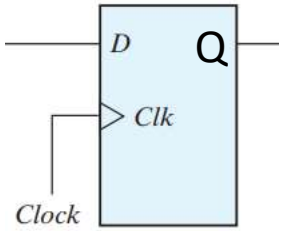
always @ (sensitivity list) begin
statement;
end

- 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 #5 toggle, means 10ns 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 or manually change**

- **$dumpfile("dump.vcd"); dump to file for gtkwave**

- **finish the simulation at #100 ticks, otherwise dump file can become BIG due to forever statement**

```
// Testbench
`timescale 1 ns / 1 ps
`include "dflipflop.v"
module testbench;
  reg d,clk; //inputs
  wire q; //outputs
  // Instantiate
  DFF D0 (.CLK(clk),.D(d), .Q(q));
   initial begin
     clk=0;
      forever #5 clk=~clk;
   end
  initial begin
  d=0;
#20 d=1;
#20 d=0;
#20 d=1;
 //    forever #10 d=~d;
  end
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    end
  initial #100 $finish;
endmodule
```

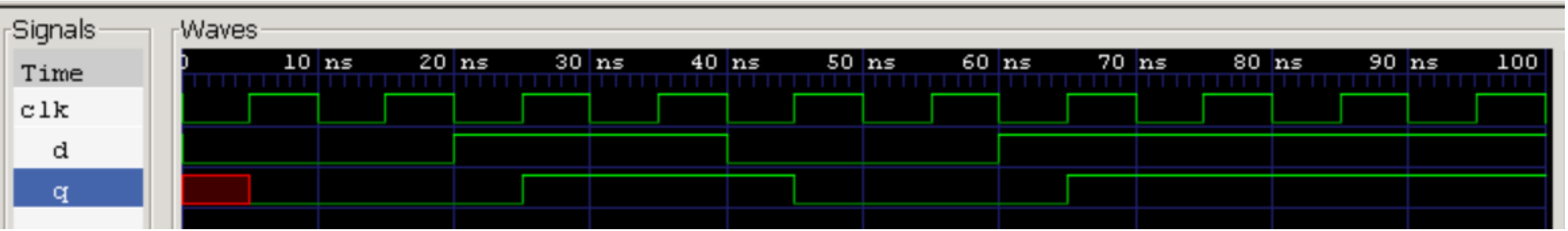
- **forever** loop - executes continually

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

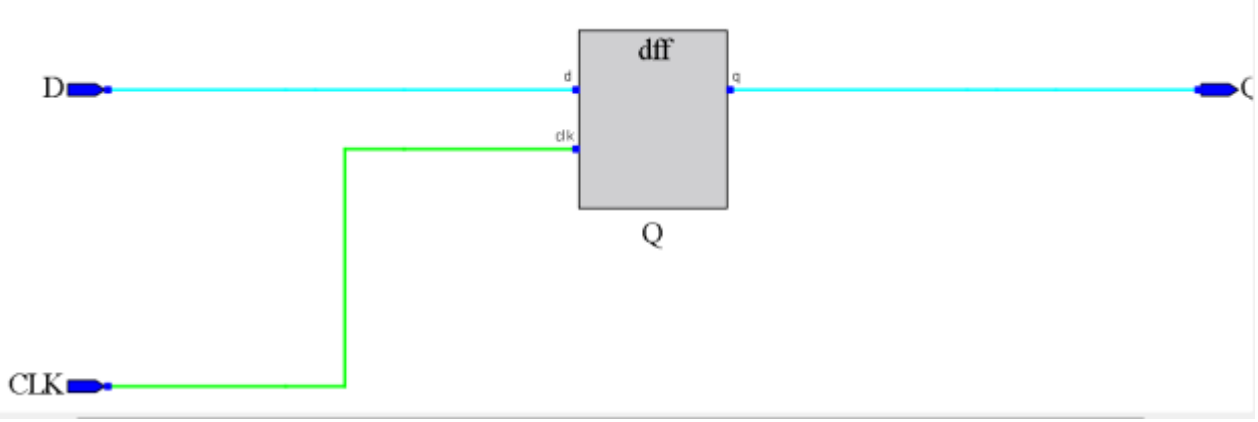
Verilog timescale

- 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.

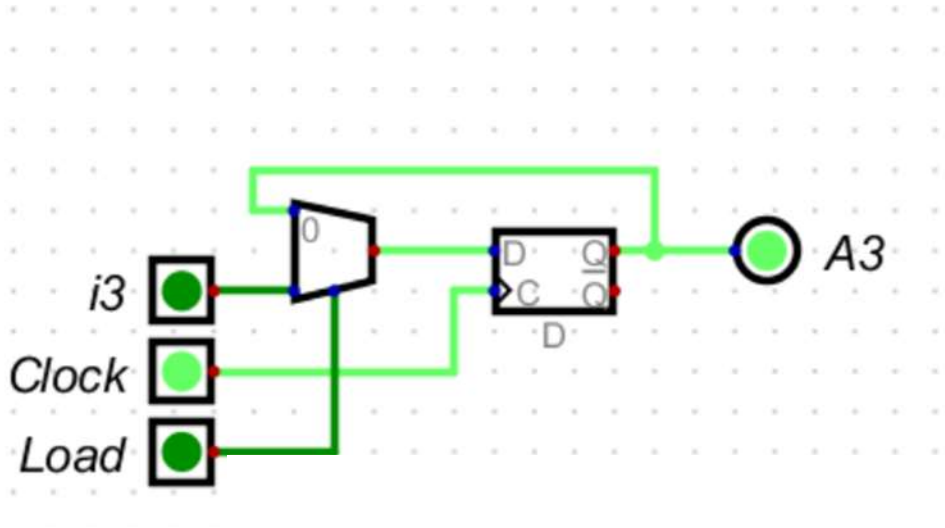


D-FlipFlop: Synthesis

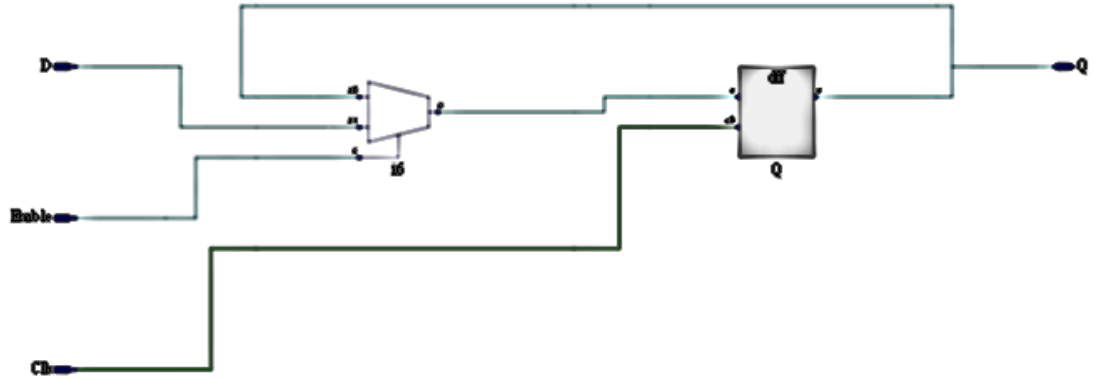



```
// D flip-flop without reset
`timescale 1 ns / 1 ps
module DFF (Q, D, CLK);
output Q;
input D, CLK;
reg Q;
always @ (posedge CLK)
 Q <= D;
endmodule
```

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

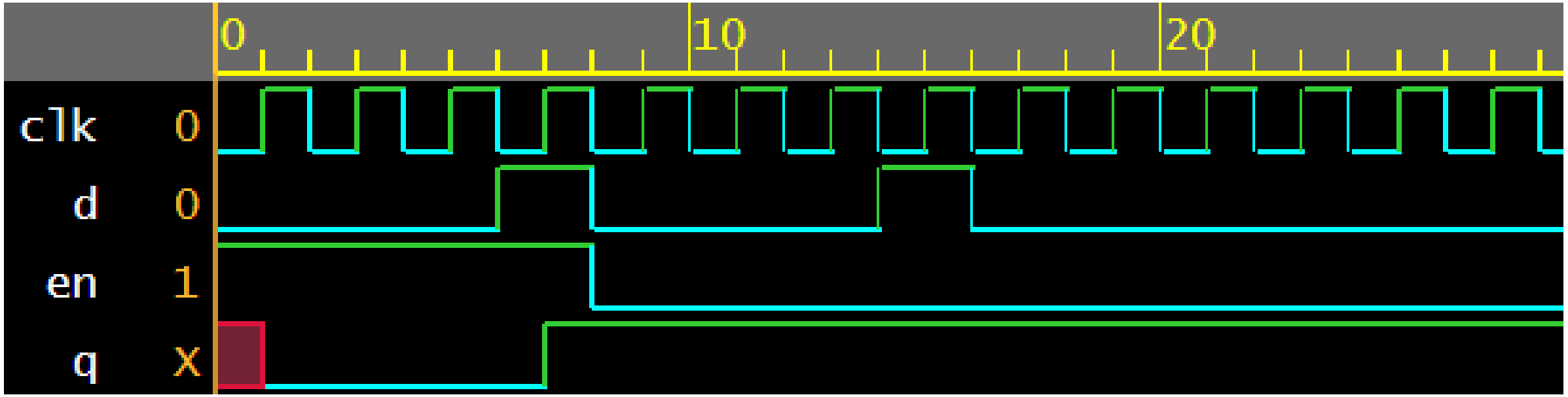
```
`timescale 1 ns / 1 ns
module onebitreg (
output reg Q,
input D,Enable,Clk);

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

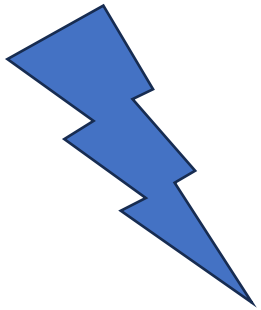
```
// Testbench
`timescale 1 ns / 1 ns
module testbench;
  reg d,en,clk;  //inputs
  wire q; //outputs

  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



Blocking and Nonblocking Assignments



Blocking Statements (=)

- **Syntax**: variable = expression;
- **Execution**: Blocking statements are executed sequentially, one after the other, within the same time step.
- **Usage**: Typically used in combinational logic where the order of execution matters.

```
always @(posedge clk) begin
   a = b;  // Blocking assignment
   c = a;  // 'c' gets the value of 'a' after 'a' is assigned the
value of 'b'
end
```

Nonblocking Statements (<=)

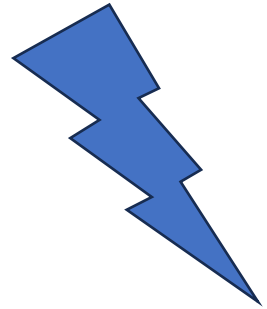
Syntax: variable <= expression;

Execution: Nonblocking statements are scheduled to execute concurrently at the end of the time step, allowing all right-hand side expressions to be evaluated before any left-hand side assignments are made.

Usage: Typically used in sequential logic, such as flip-flops, where you want to capture values simultaneously

```
always @(posedge clk) begin
   a <= b;  // Nonblocking assignment
   c <= a;  // 'c' gets the old value of 'a' before 'a'
is assigned the value of 'b'
end
```

Blocking and Nonblocking Assignments



Summary:

- Blocking (=): Sequential execution, used in **combinational** logic.
- Nonblocking (<=): Concurrent execution, used in **sequential** logic.

Clock Usage: Blink LED

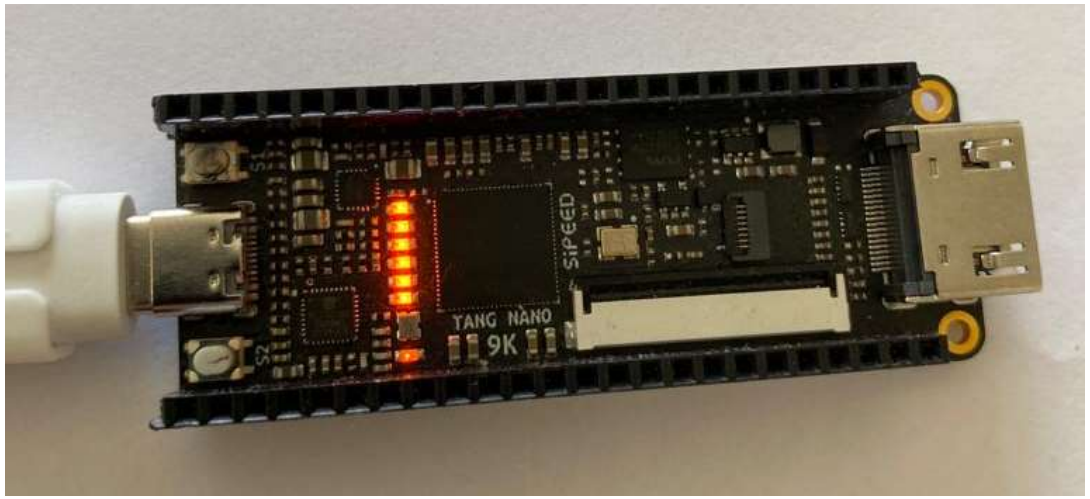
- Sequential circuits performs actions **at clock signals**. Thus we have clock input
- Output leds defined as 6-bit [5:0]
- localparam is local to the module, number of clock cycles that we we will wait for changing leds
- **localparam**: a local constant for storing integers, real numbers, time, delays,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

```
module blinkled
(
   input clk,
   output reg [5:0]  leds
);
localparam WAIT_TIME = 27000000;
reg [31:0] clockCounter = 0;
always @(posedge clk) begin
   clockCounter <= clockCounter + 1;
   if (clockCounter == WAIT_TIME) begin
      clockCounter <= 0;
      leds=~leds;
   end
end
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,



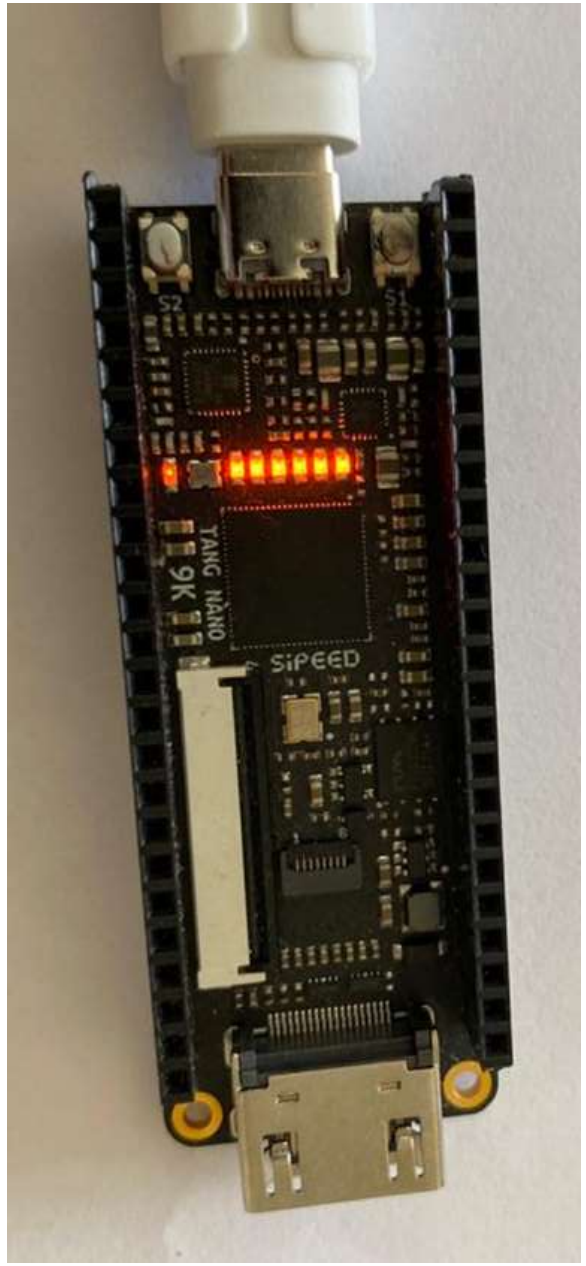

```
module blinkled
(
   input clk,
   output reg [5:0]  leds
);
localparam WAIT_TIME = 15000000;
reg [23:0] clockCounter = 0;
always @(posedge clk) begin
   clockCounter <= clockCounter + 1;
   if (clockCounter == WAIT_TIME) begin
      clockCounter <= 0;
      leds=~leds;
   end
end
endmodule
```

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**

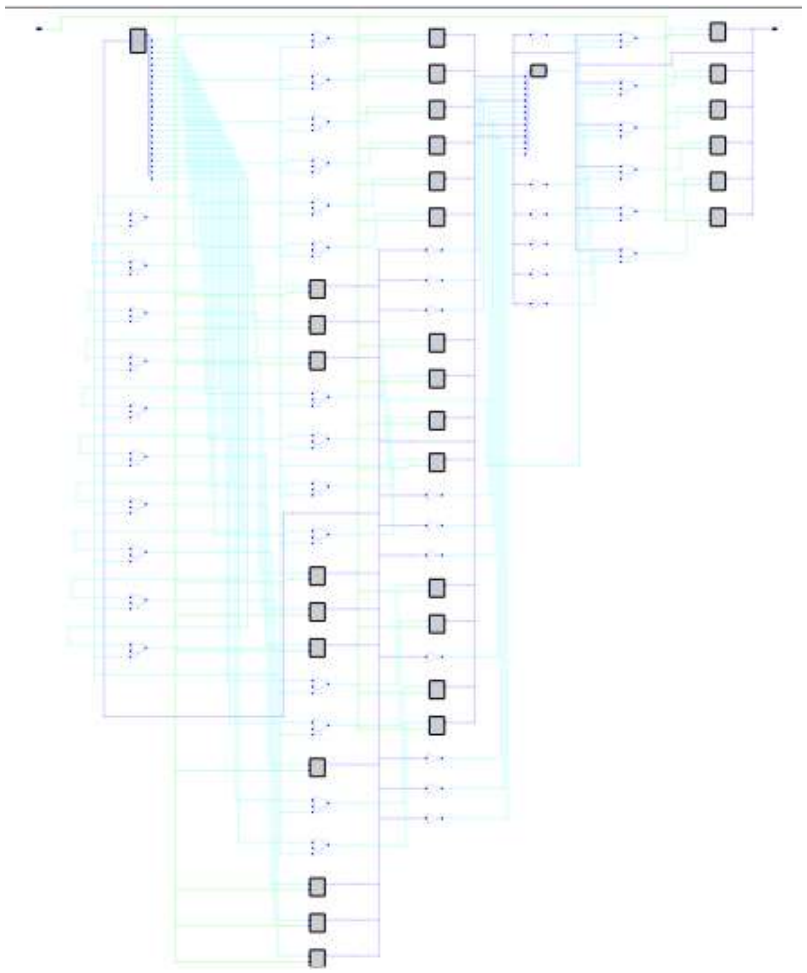
I/O Constraints

	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



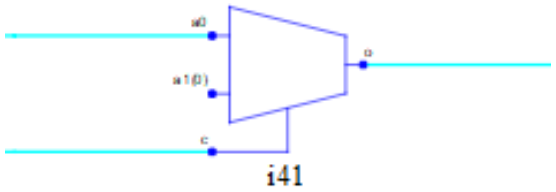
Tang Nano 9K: Blink LED RTL Shematic

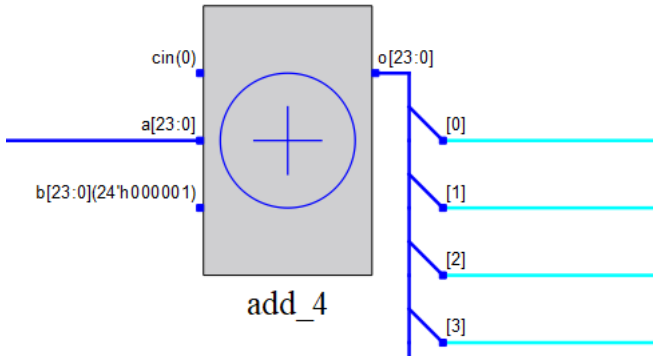
- a little complicated ☺
- lets zoom in

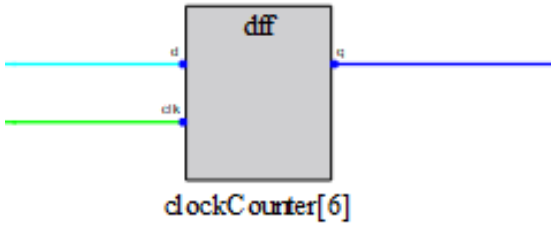


Tang Nano 9K: Blink LED RTL Shematic

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



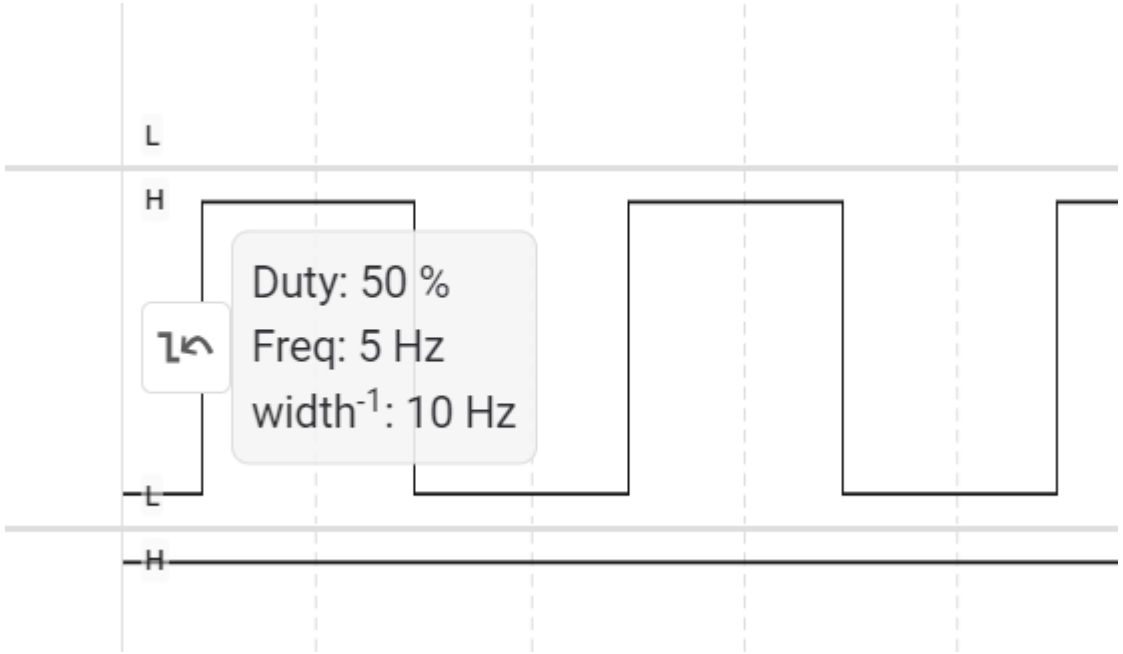





```
module blinkled
(
   input clk,
   output reg [5:0]  leds
);
localparam WAIT_TIME = 27000000;
reg [31:0] clockCounter = 0;

always @(posedge clk) begin

   clockCounter <= clockCounter + 1;

   if (clockCounter == WAIT_TIME) begin
      clockCounter <= 0;
      leds=~leds;
   end //if
end  //always

endmodule
```

Blink LED w pin32

- pin32 toggling




```
module blinkled
(
   input clk,
   output reg [5:0]  leds,
output reg pin32
);
localparam WAIT_TIME = 2700000;
reg [31:0] clockCounter = 0;

always @(posedge clk) begin

   clockCounter <= clockCounter + 1;

   if (clockCounter == WAIT_TIME) begin
      clockCounter <= 0;
      leds=~leds;
      pin32 <= ~pin32;
   end //if
end  //always

endmodule
```

Tang Nano 9K: Blink LED RTL Shematic

- clk 27MHz, at pin52
- buttons are 3,4
- led 10,11,13,14,15,16

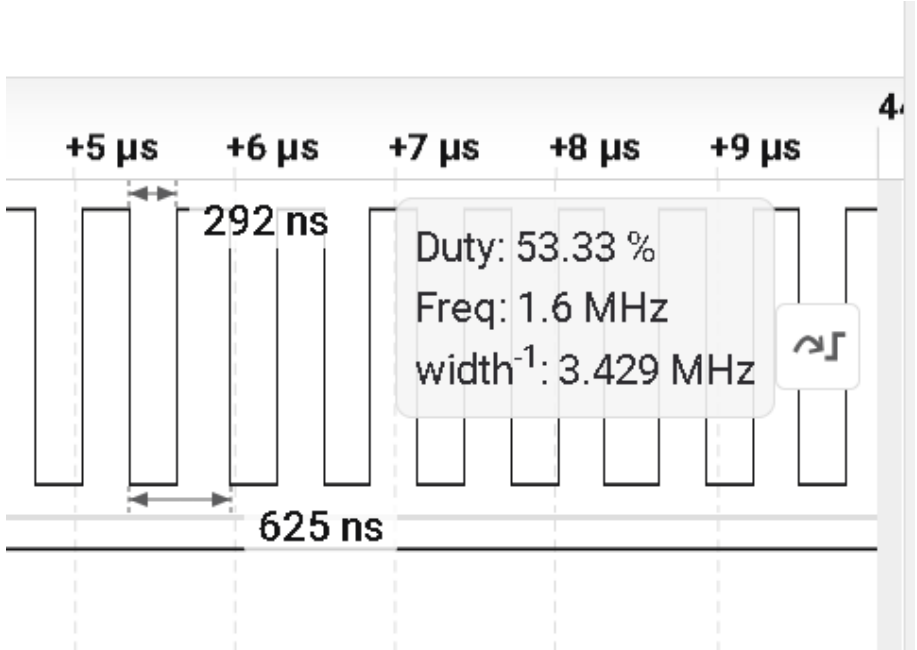
Clock Divider

```
module clockdivider (
    input wire clk,
    output wire out1
);
reg [3:0] c1 = 4'b0;
always @(posedge clk) begin
        c1 <= c1 + 1'b1;
end
assign out1 = c1[3];  // Simplified output assignment 27MHz/16
endmodule
```

I/O Constraints

	Port	Direction	Diff Pair	Location
1	clk	input		52
2	out1	output		32

- Tang Nano 9k has 27MHz clok
- Saleae Logic analyzer can show max 25MHz
- Thus we divide clock to 16 = 1.7MHz



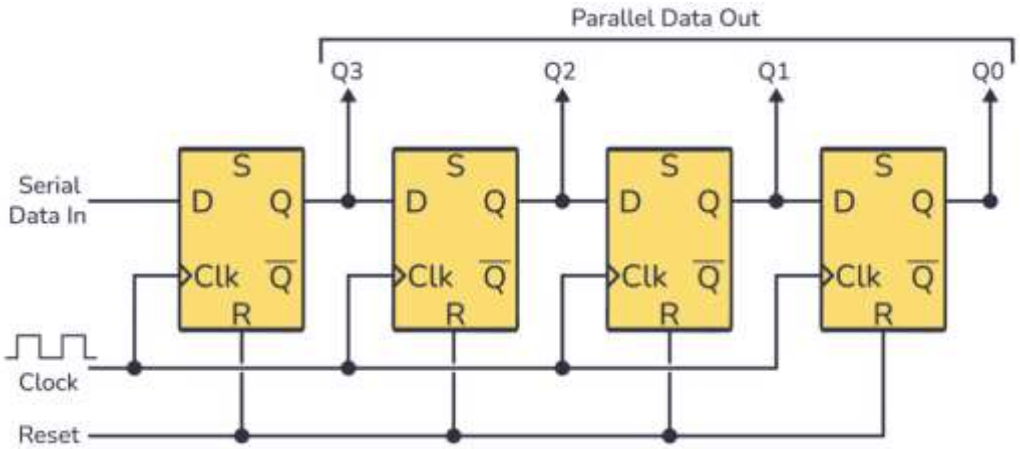
Counter 6-bit with Reset

I/O Constraints

	Port	Direction	Diff Pair	Location
1	clk	input		52
2	led[0]	output		10
3	led[1]	output		11
4	led[2]	output		13
5	led[3]	output		14
6	led[4]	output		15
7	led[5]	output		16
8	reset	input		3

```
module counter
(
   input clk,
   input reset,  // Active low reset
   output [5:0] led
);
localparam WAIT_TIME = 27000000;
reg [5:0] ledCounter = 0;
reg [31:0] clockCounter = 0;
always @(posedge clk or negedge reset) begin
   if (!reset) begin  // Reset is active low
      clockCounter <= 0;
      ledCounter <= 0;
   end else begin
      clockCounter <= clockCounter + 1;
      if (clockCounter == WAIT_TIME) begin
         clockCounter <= 0;
         ledCounter <= ledCounter + 1;
      end
   end
end
assign led = ~ledCounter;
endmodule
```

6-bit Shift Register with Reset




```
module counter
(
   input clk,
   input reset,  // Active low reset
   input data,   // Active low data
   output reg [5:0] led  // Active low led
);
localparam WAIT_TIME = 27000000;
reg [31:0] clockCounter = 0;
always @(posedge clk or negedge reset) begin
   if (!reset) begin  // Reset is active low
      clockCounter <= 0;
      led <= 6'b111111;  // Active low reset value
   end else begin
      clockCounter <= clockCounter + 1;
      if (clockCounter == WAIT_TIME) begin
         clockCounter <= 0;
         led <= {led[4:0], ~data};  // Shift in the inverted
value of data
      end
   end
end
endmodule
```

Tang Nano:Memory

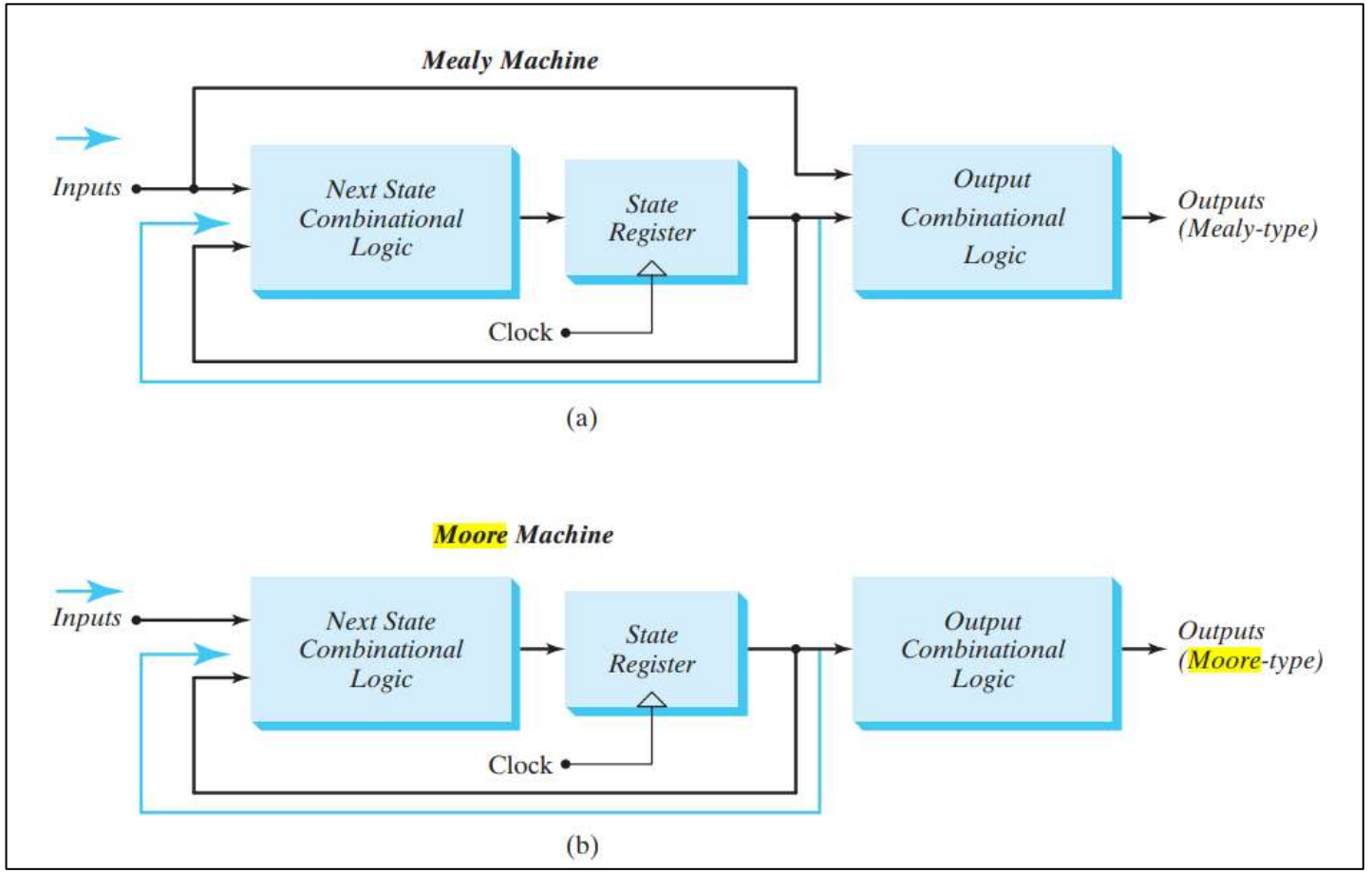
- lets write an example which reads led[n] values from memory and outputs to leds
- 64x6 bits memory defined
- Fill the memory first 6 address
- in always @() block increments Address counter
-
- flash into Tang Nano 9k
- Leds will turn on as described in memory

```
module memory_example(
    input clk,
    output reg [5:0] leds
);
reg [5:0] Address = 6'b0;
localparam WAIT_TIME = 27000000;
reg [31:0] clockCounter = 0;
reg [5:0] Mem [0:11]; // 12 x 6 memory
initial begin
    Mem[0] = 6'b000001;
    Mem[1] = 6'b000010;
    Mem[2] = 6'b000100;
    Mem[3] = 6'b001000;
    Mem[4] = 6'b010000;
    Mem[5] = 6'b100000;
    Mem[6] = 6'b100000;
    Mem[7] = 6'b010000;
    Mem[8] = 6'b001000;
    Mem[9] = 6'b000100;
    Mem[10] = 6'b000010;
    Mem[11] = 6'b000001;
end
always @(posedge clk) begin
    clockCounter <= clockCounter + 1'b1;
    if (clockCounter == WAIT_TIME) begin
```

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**

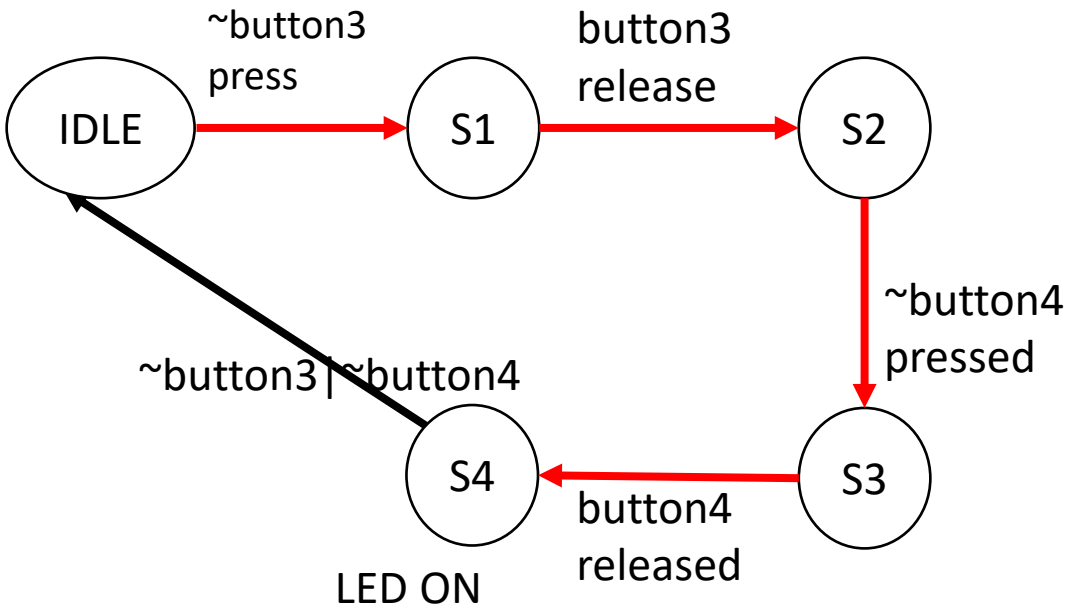


Finite State Machines: Pattern Detection

- In the **Mealy model**, the output is a function of input and **present state**.

- state transition conditions defined inside the state
- lets catch button3-button4 sequence in Tang Nano 9k

- State machines allows to design sequential flow of execution similar to programming



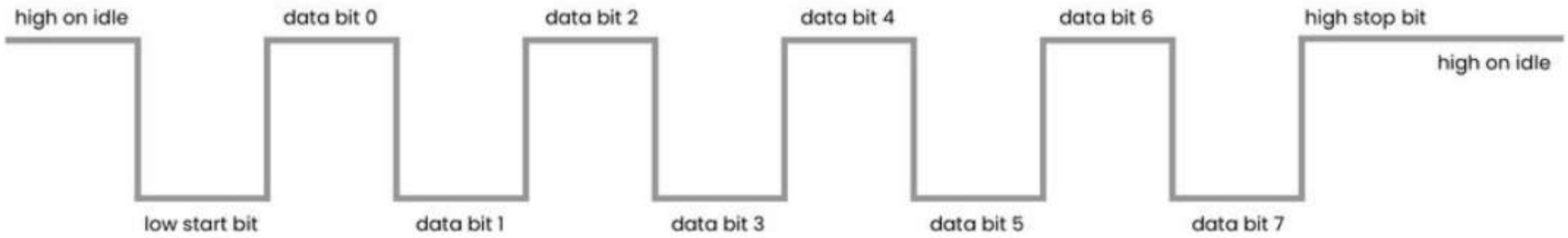
Pattern Detection

- In the **Moore model**, the output is a function of **only the present state**.
- state transition conditions defined inside the state
- lets catch button3-button4 sequence in Tang Nano 9k

```
parameter S3 = 3'b011;
parameter S4 = 3'b100;
reg [2:0] state = IDLE;
always @(posedge clk) begin
  case (state)
    IDLE: begin
      led = 1'b1; // Ensure LED is OFF in IDLE state
      if (~button3) // Check if button3 is pressed (active low)
        state = S1;
    end
    S1: begin
      if (button3) // Check if button3 is released (active high)
        state = S2;
    end
    S2: begin
      if (~button4) // Check if button4 is pressed (active low)
        state = S3;
    end
    S3: begin
      if (button4) // Check if button4 is released (active high)
        state = S4;
    end
    S4: begin
      led = 1'b0; // LED ON (active low)
      if (~button3 || ~button4) // Return to IDLE if any button is pressed
        state = IDLE;
    end
  endcase
end
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: UART-TX

- lets flash UART transmit example

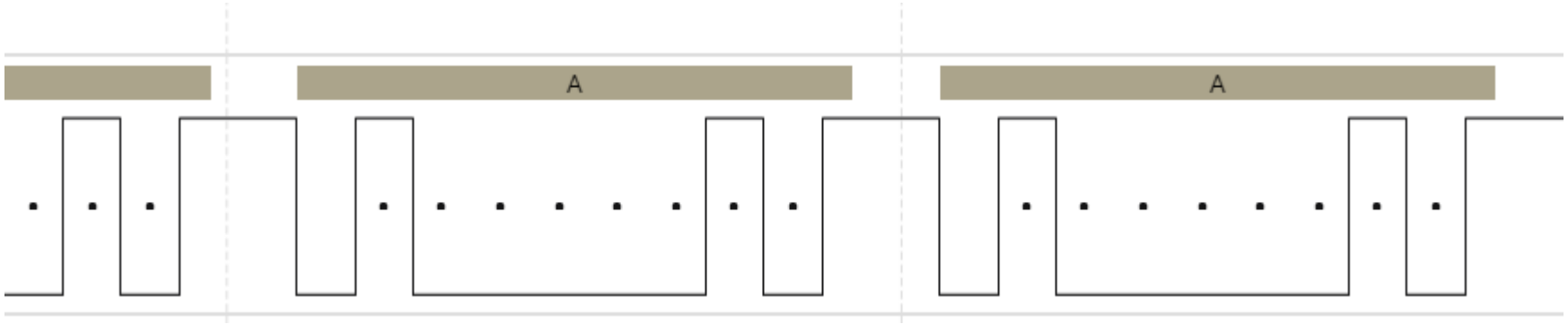
I/O Constraints

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


```
module uart_tx (
   input wire clk,
   output reg tx
);
   parameter CLK_FREQ = 27000000; // 27 MHz
   parameter BAUD_RATE = 115200;
   parameter BIT_PERIOD = CLK_FREQ / BAUD_RATE;
   reg [15:0] clk_count = 0;
   reg [3:0] bit_index = 0;
   reg [9:0] tx_data = 10'b1_01000001_0; // Start bit, 'A',
Stop bit
   always @(posedge clk) begin
      if (clk_count < BIT_PERIOD - 1) begin
         clk_count <= clk_count + 1;
      end else begin
         clk_count <= 0;
         bit_index <= bit_index + 1;
         if (bit_index < 10) begin
            tx <= tx_data[bit_index];
         end else begin
            bit_index <= 0;
            tx <= 1; // Idle state
         end
      end
   end
endmodule
```

Tang Nano: UART-TX

- Saleae Logic Analyzer can decode UART



Async Serial

Input Channel *	03. Channel 3
Bit Rate (Bits/s)	115200
Bits per Frame	8 Bits per Transfer (Standard)
Stop Bits	1 Stop Bit (Standard)
Parity Bit	No Parity Bit (Standard)

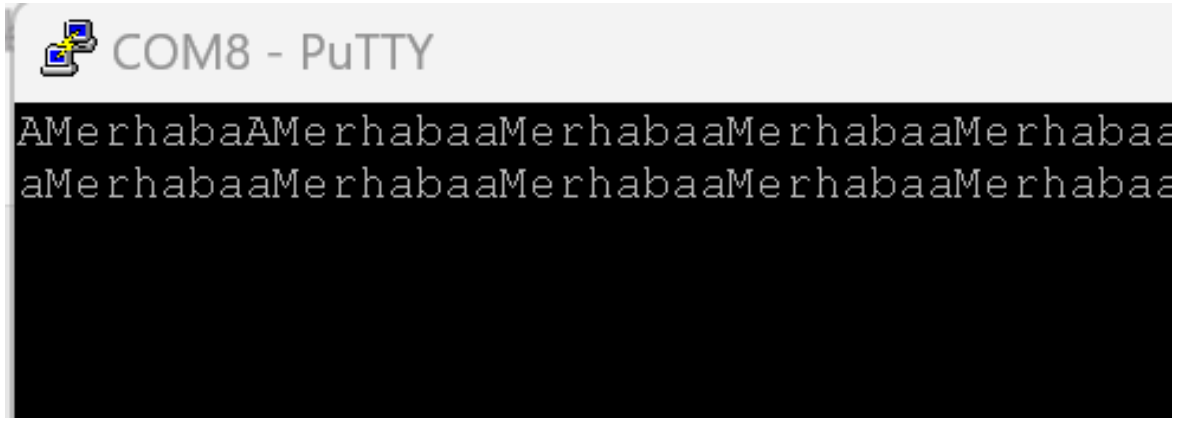
```
reg [9:0] tx_data = 10'b1_01000001_0; // Start bit, 'A', Stop bit
```

Tang Nano: UART-TX

- lets flash UART transmit example

I/O Constraints

	Port	Direction	Diff Pair	Location
1	clk	input		52
2	reset_n	input		3
3	tx	output		17



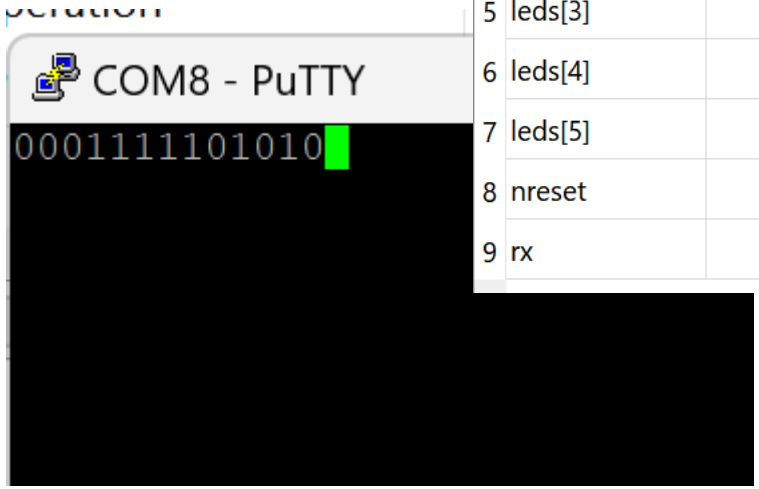

```
module uart_tx (
   input wire clk,
   input wire reset_n, // Active low reset button
   output reg tx
);

   parameter CLK_FREQ = 27000000; // 27 MHz
   parameter BAUD_RATE = 115200;
   parameter BIT_PERIOD = CLK_FREQ / BAUD_RATE;

   reg [15:0] clk_count = 0;
   reg [3:0] bit_index = 0;
   reg [3:0] char_index = 0;
   reg [9:0] tx_data = 10'b1_01000001_0; // Start bit, 'A',
Stop bit
   reg [7:0] message [0:6]; // Array to hold "Merhaba"
   reg sending = 0;

   initial begin
      message[0] = "M";
      message[1] = "e";
      message[2] = "r";
      message[3] = "h";
      message[4] = "a";
      message[5] = "b";
      message[6] = "a";
   end

   always @(posedge clk or negedge reset_n) begin
      if (!reset_n) begin
         // Reset condition
```

Tang Nano: UART-RX

- lets flash UART transmit example

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
8	nreset	input		3
9	rx	input		18

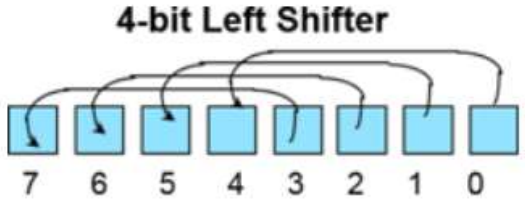



```
module uart_rx_leds (
    input wire clk,
    input wire nreset,
    input wire rx, // UART receive pin
    output reg [5:0] leds // Active low LEDs
);
    parameter CLK_FREQ = 27000000; // 27 MHz
    parameter BAUD_RATE = 115200;
    parameter integer BAUD_DIV = CLK_FREQ / BAUD_RATE;

    reg [15:0] baud_counter;
    reg [3:0] bit_counter;
    reg [9:0] shift_reg;
    reg receiving;
    reg [7:0] rx_data;
    reg rx_prev =1; // Register to store the previous state of
rx

    always @(posedge clk or negedge nreset) begin
        if (!nreset) begin
            baud_counter <= 0;
            bit_counter <= 0;
            shift_reg <= 0;
            receiving <= 0;
            rx_data <= 8'b0;
            leds <= 6'b000000; // Turn on all LEDs (active low)
            rx_prev <= 1; // Initialize to idle state (high)
        end else begin
            rx_prev <= rx; // Update previous state of rx
            if (!receiving && rx_prev && !rx) begin // Detect
negative edge of rx (start bit)
```

For Loop

For Loop

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

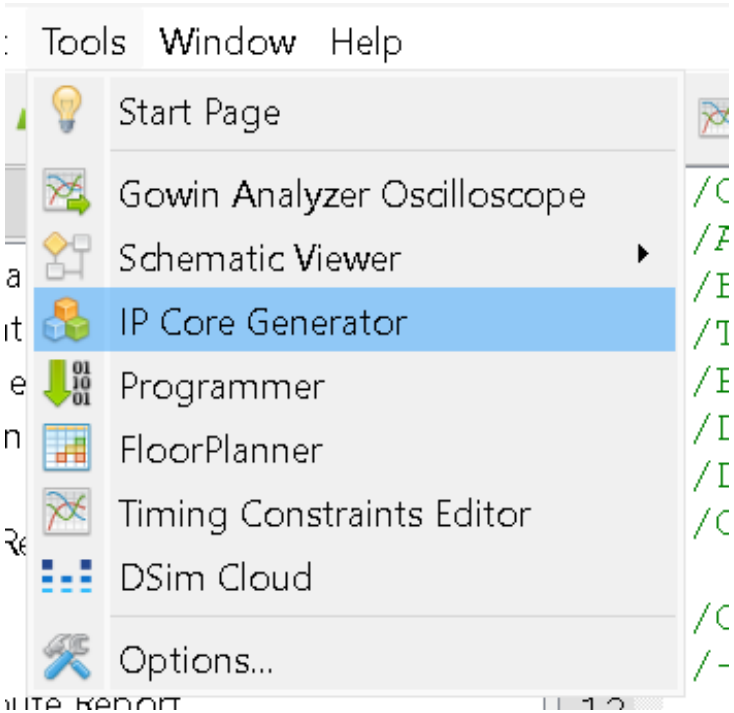


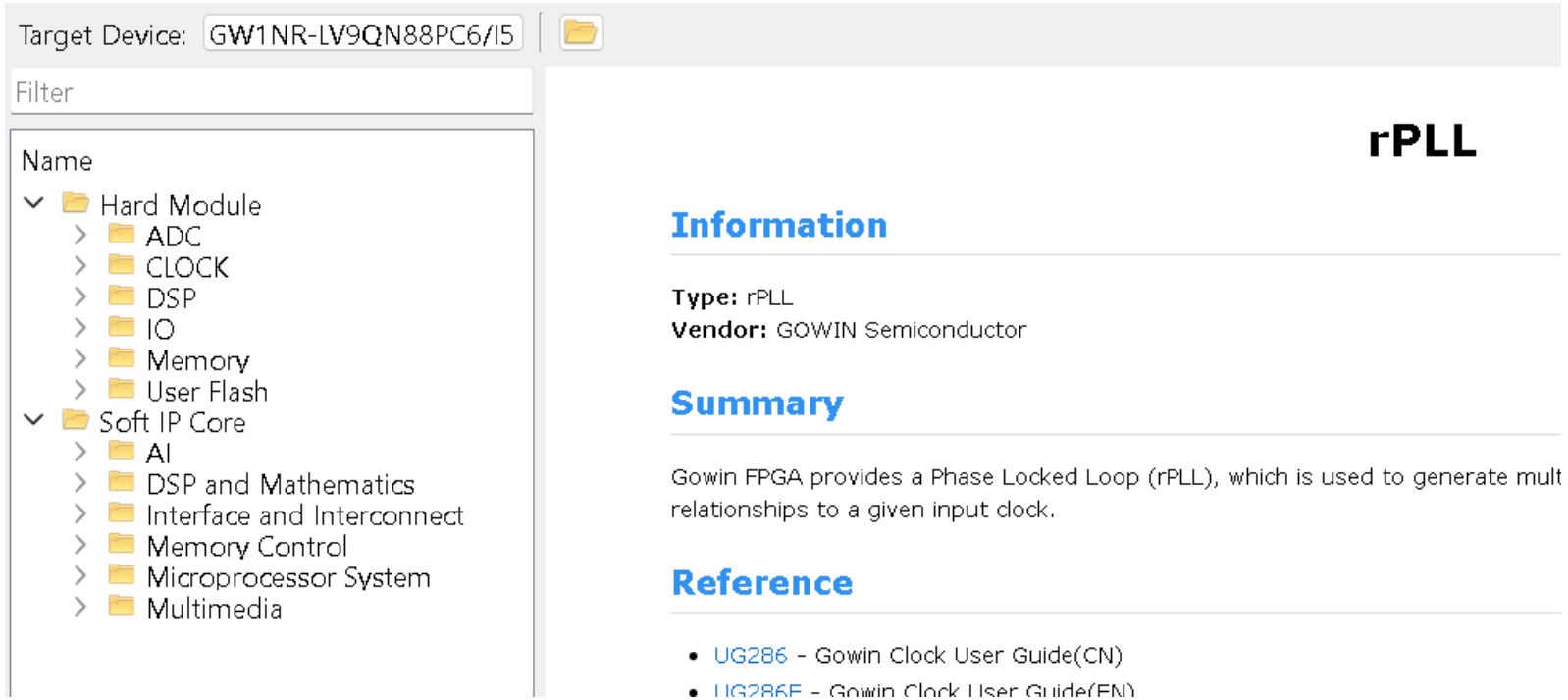

```
// declare the index for the FOR loop
integer  i;

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™

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.

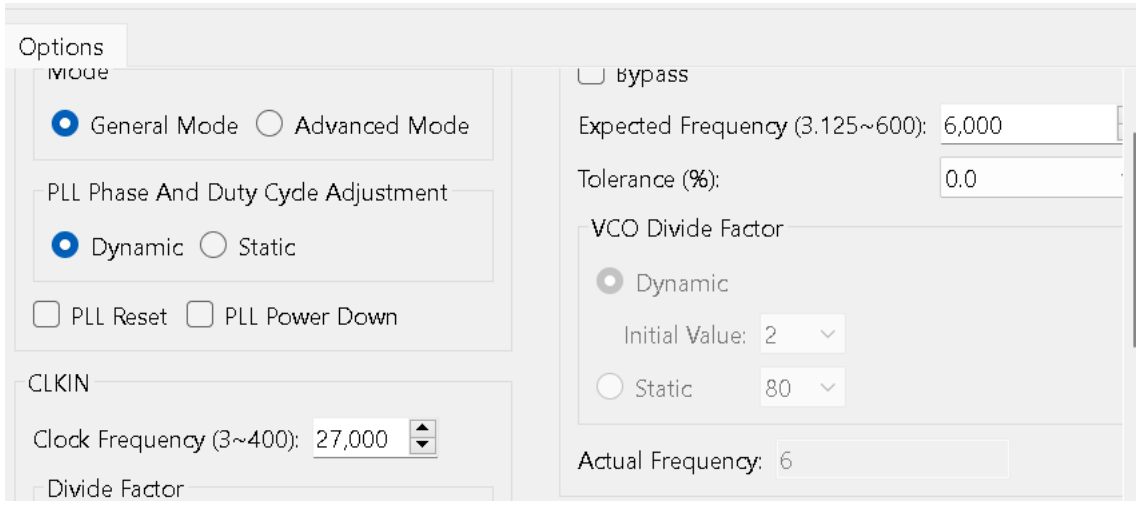




Tang Nao IP Core Example rPLL

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

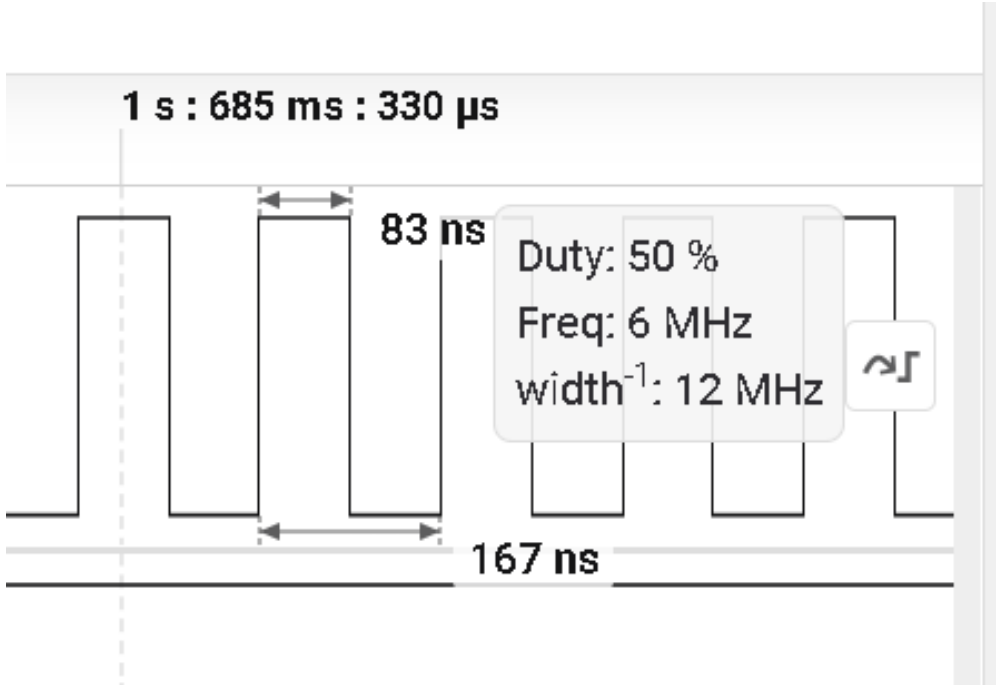
	Port	Direction	Diff Pair	Location
1	clkin	input		52
2	clkout	output		32

Tang Nao IP Core Example rPLL

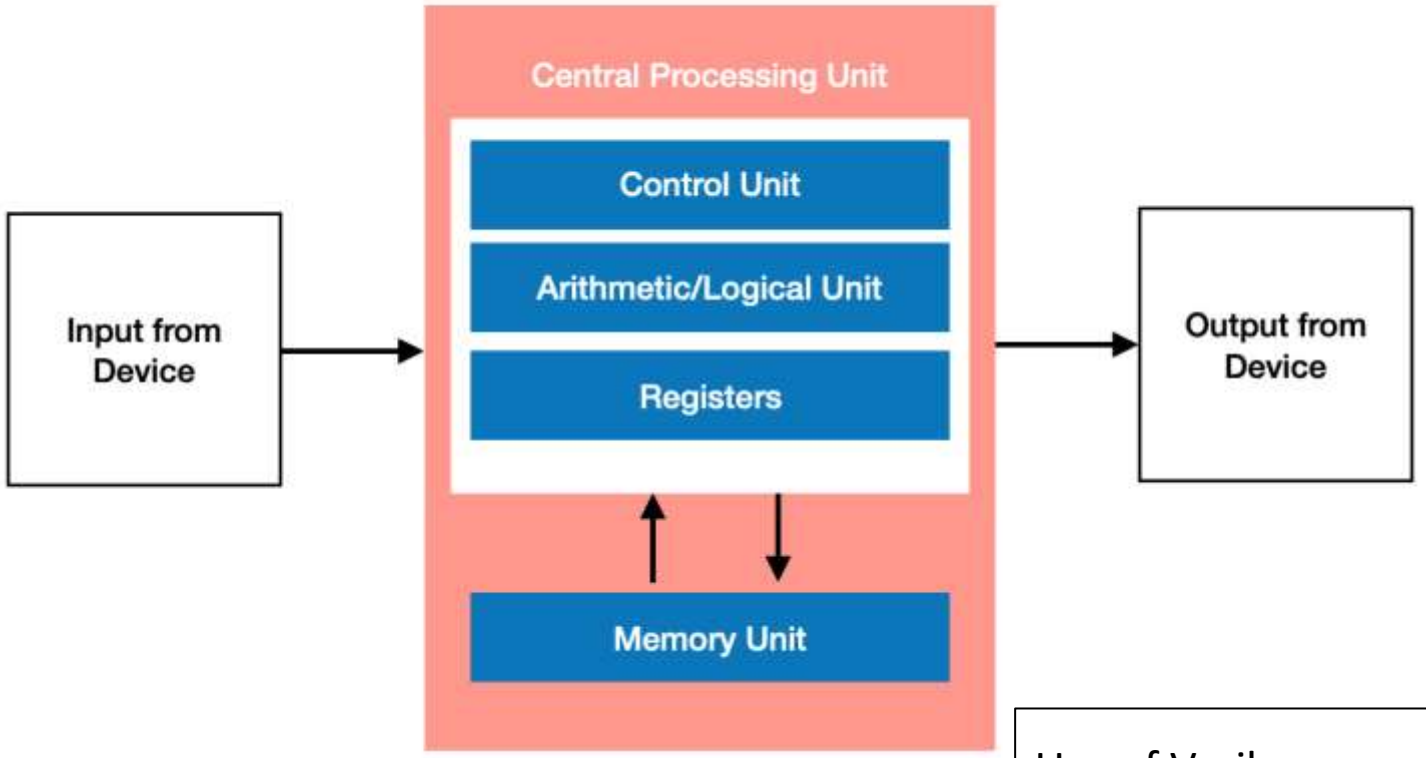
We will generate 6MHz clock at pin 32 from internal 27MHz vlock at pin 52

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)



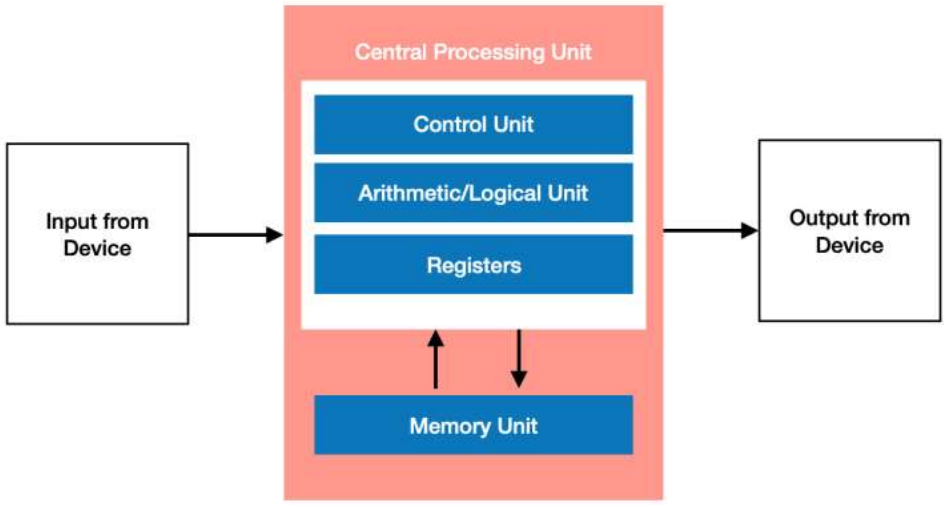
The **von Neumann** architecture—also known as the von Neumann model or Princeton architecture—is a computer architecture based on a 1945 description by **John von Neumann**, and by others, in the First Draft of a Report on the EDVAC.

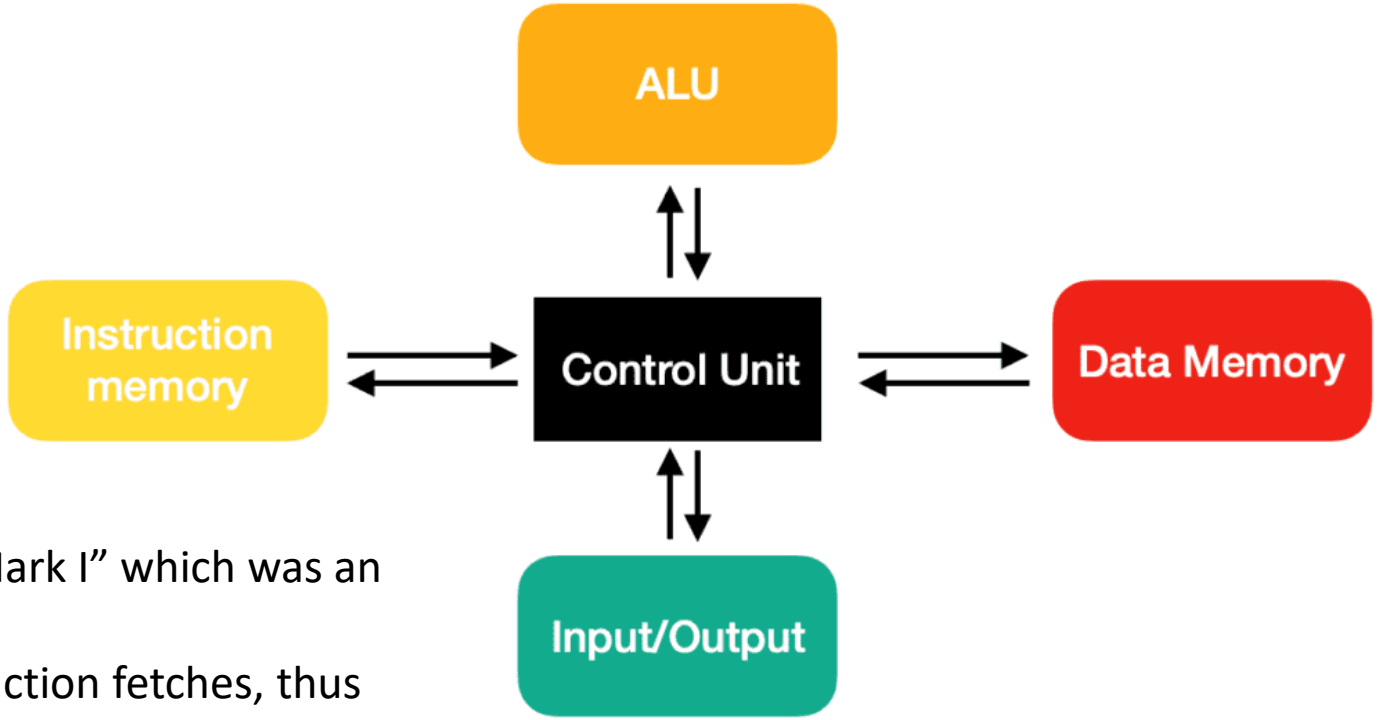
The document describes a design architecture for an electronic digital computer with these components:

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) 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

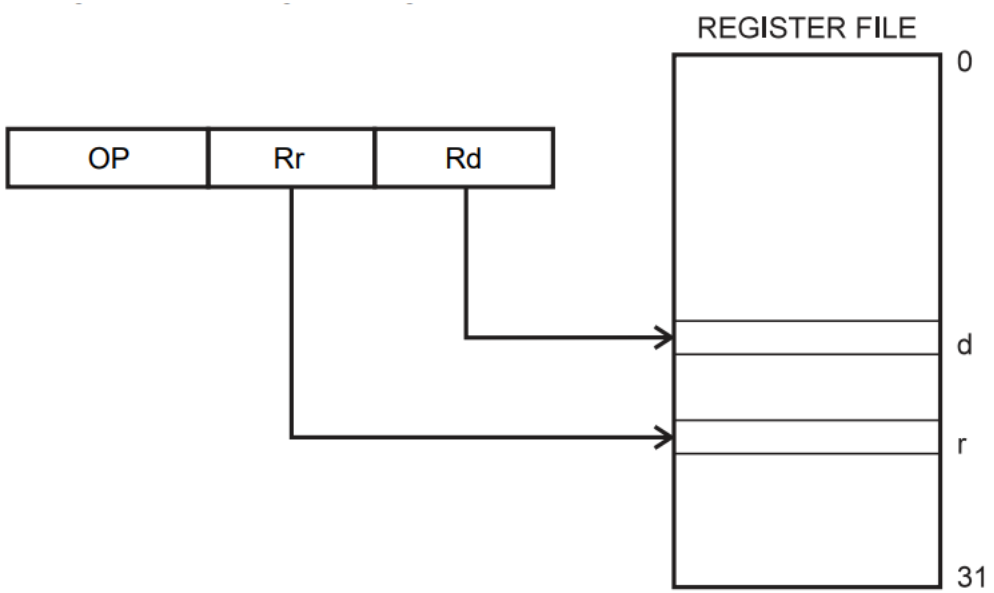
32-bit Opcode:

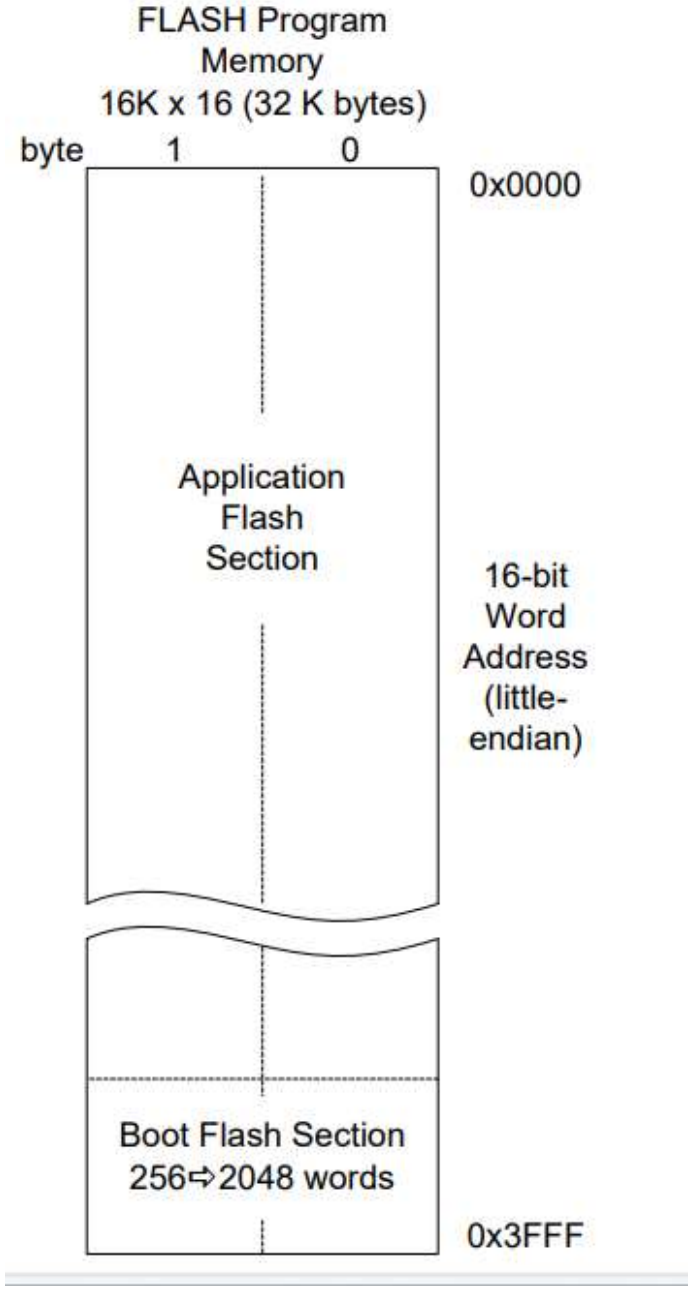
1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

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

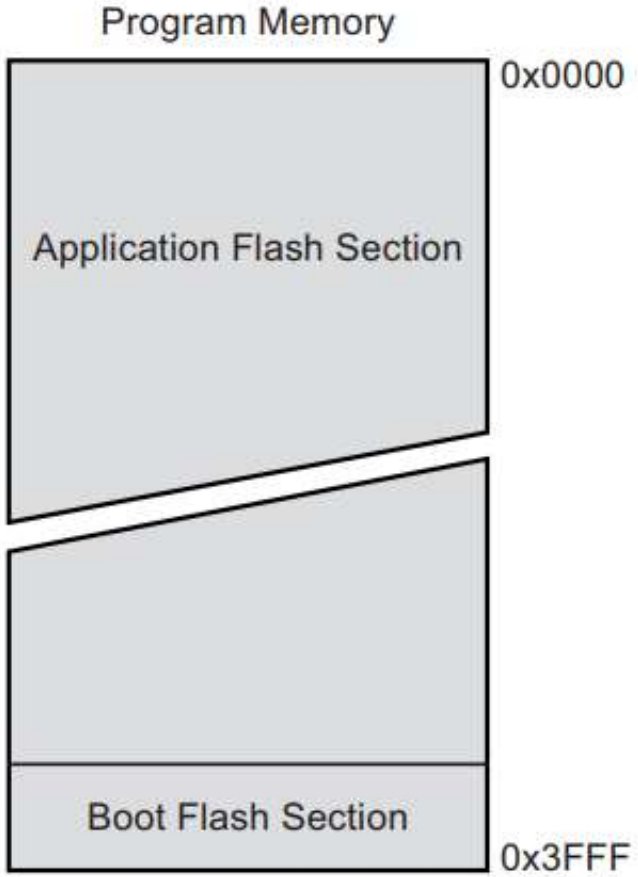




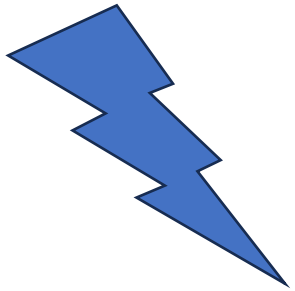
MCU: Program Memory

- example code for 7 + 2 - 6 = 3
- LOADA 7
- LOADB 2
- ADD
- LOADB 6
- SUB

```
// Initialize instruction memory (example instructions)
  initial begin
     instruction_memory[0] = {LOADA, 12'b000000000111}; // LOADA 7
     instruction_memory[1] = {LOADB, 12'b000000000010}; // LOADB 2
     instruction_memory[2] = {ADD, 12'b000000000000};   // ADD
     instruction_memory[3] = {LOADB, 12'b000000000110}; // LOADB 6
     instruction_memory[4] = {SUB, 12'b000000000000};   // SUB
     // Fill the rest of the memory with NOPs
     for (i = 5; i < 32; i = i + 1) begin
        instruction_memory[i] = {NOP, 12'b000000000000};
     end
  end
```



MYMCU: FETCH→DECODE→EXECUTE



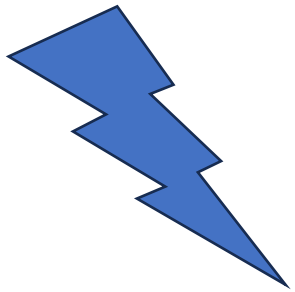
- Fetch:
- In the fetch state, the CPU retrieves the next instruction from memory.
- The address of the instruction is held in the program counter (PC).
- After fetching the instruction, the program counter is incremented to point to the next instruction in memory.

- Decode:
- In the decode state, the CPU interprets the fetched instruction.
- The control unit decodes the instruction to determine what actions are required. This involves identifying the opcode (operation code) and any operands (data or addresses) that are part of the instruction.
- Based on the decoded information, the control unit sets up the necessary control signals for the execution phase.

- Execute:
- In the execute state, the CPU performs the operation specified by the decoded instruction.
- This could involve arithmetic or logical operations, data transfer between registers, memory access, or control operations like jumps or branches.
- The results of the operation are stored in the appropriate registers or memory locations.

MYMCU:

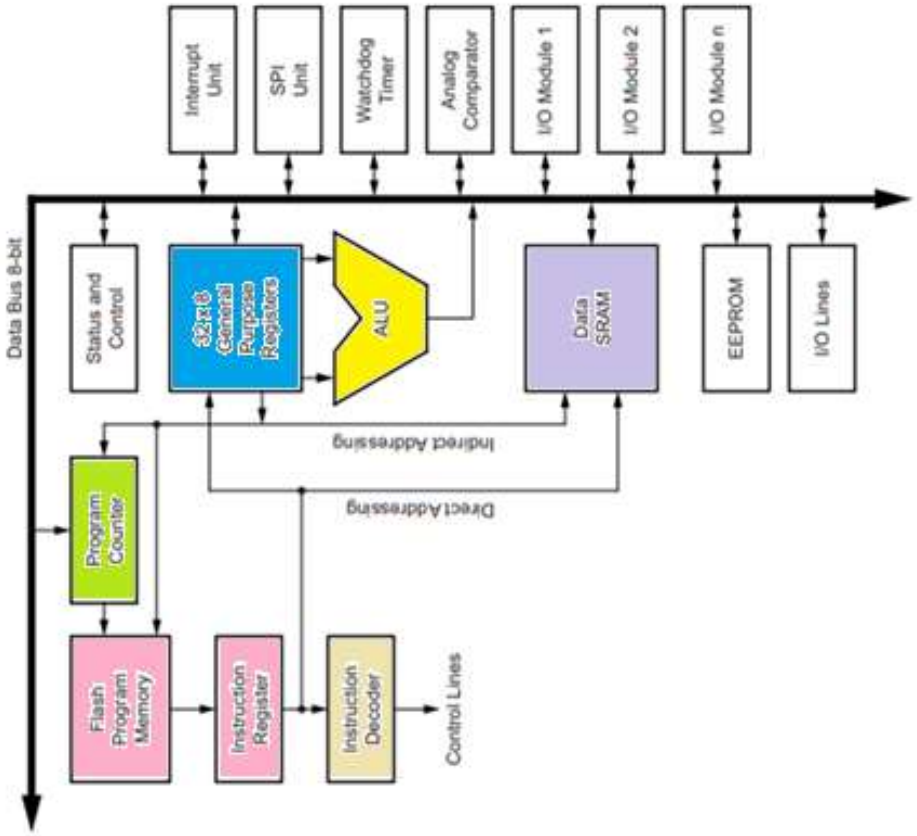
will peform 7 + 2 – 6 = 3 and show on leds

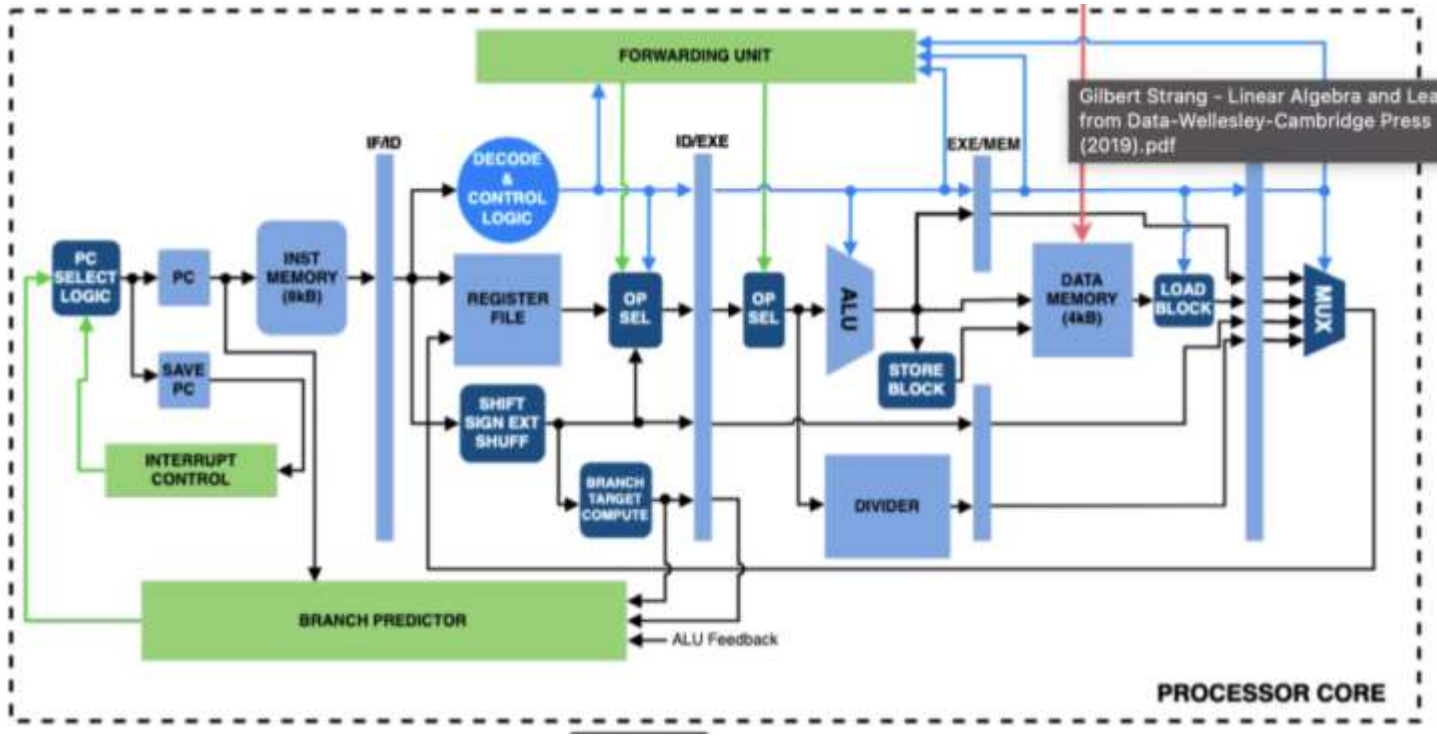


```
module simple_cpu (
    input clk,
    input reset_n, // Active low reset
    output reg [3:0] leds // Active low LEDs
);

    reg [5:0] pc; // Program counter
    reg [3:0] regA; // Register A
    reg [3:0] regB; // Register B
    reg [7:0] instruction_memory [0:31]; // Memory to store
instructions
    reg [7:0] instruction; // Current instruction
    reg [2:0] state; // FSM state
    reg [3:0] opcode; // Decoded opcode
    reg [3:0] operand; // Decoded operand
    integer i; // Loop variable

    // State encoding
    localparam FETCH = 3'b000,
               DECODE = 3'b001,
               EXECUTE = 3'b010;

    // Instruction encoding
    localparam NOP = 4'b0000,
```

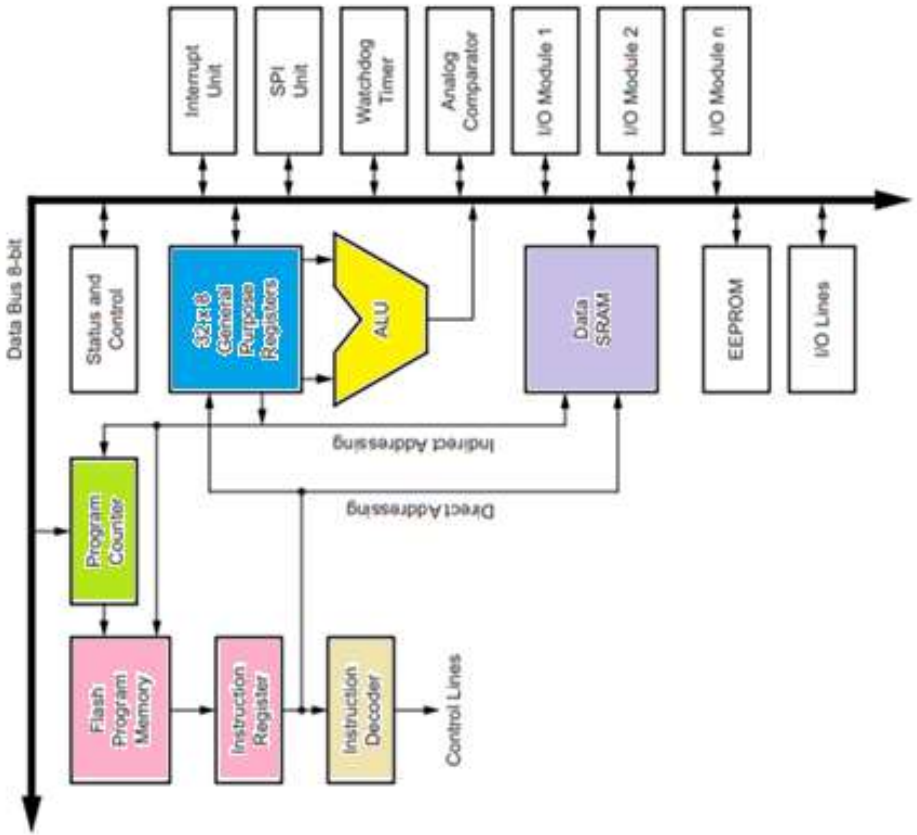
AVR vs RISCV

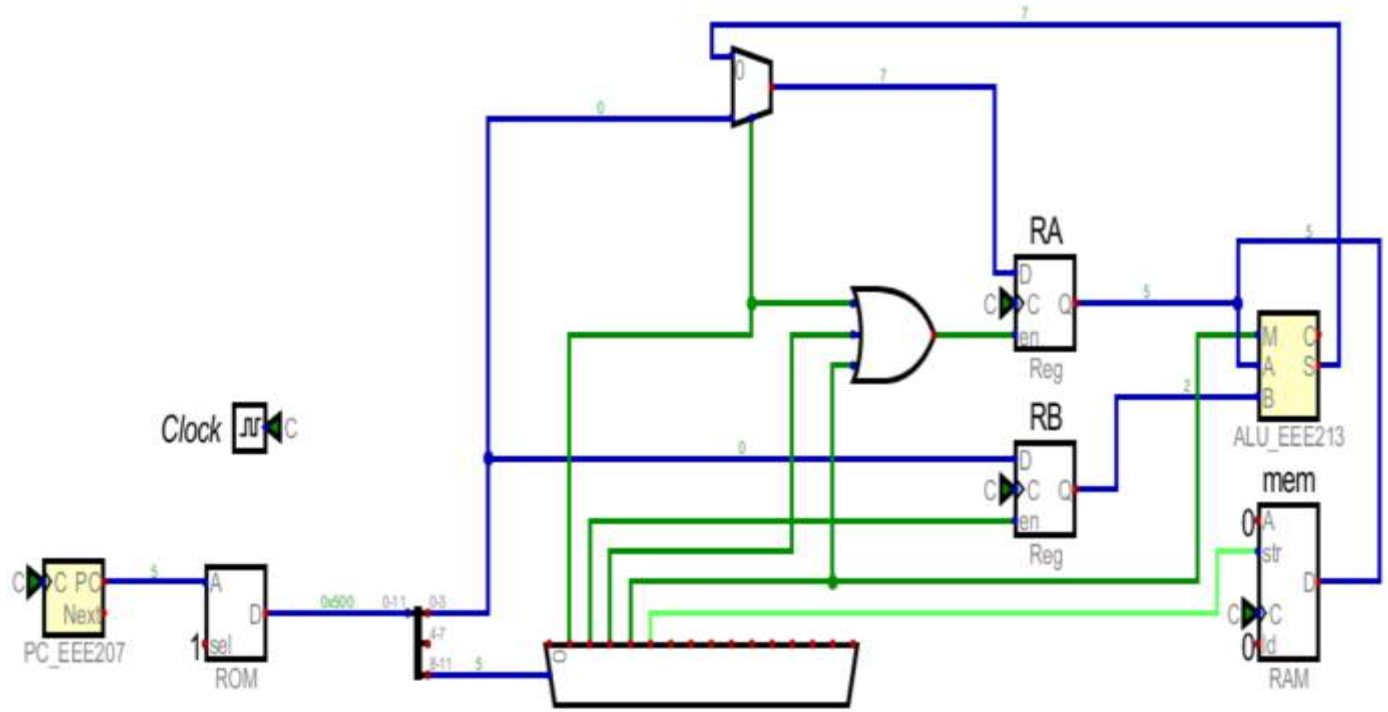
RISC-V (pronounced "risk-five") is an open-source instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC)





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
```