ASIC/FPGA Chip Design

HDL Coding (Verilog)

Mahdi Shabany

Department of Electrical Engineering

Sharif University of technology

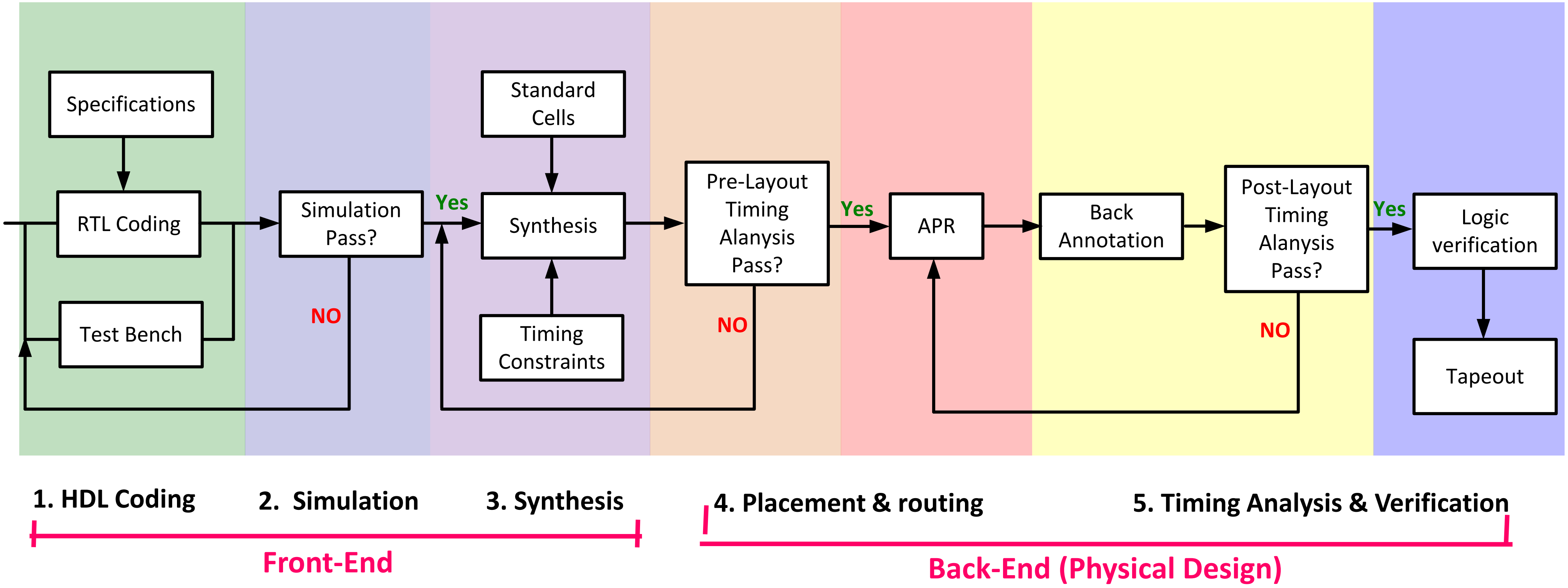
Outline

- ❑ **ASIC/FPGA Design Flow**
- ❑ **Hardware Description Language (HDL)**
  - ➢ **Verilog**
    - o Introduction
    - o Language Fundamentals
    - o Modeling Combinational & Sequential Logic Circuits
    - o Modeling Finite State Machines
    - o Verilog Operations

Outline

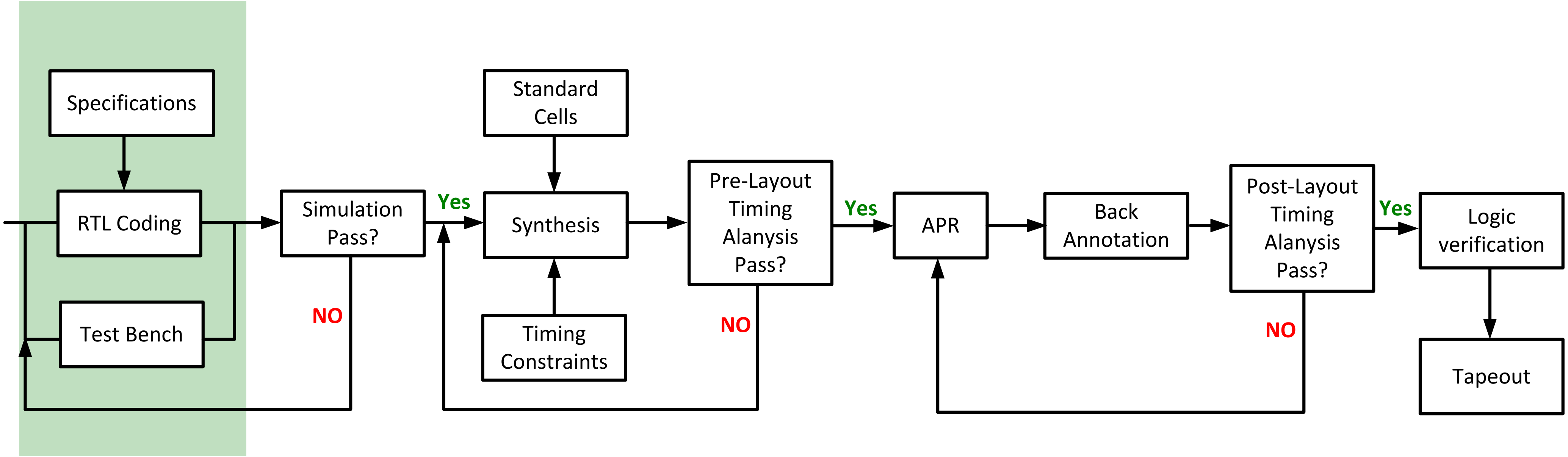
- ❑ **ASIC/FPGA Design Flow**
- ❑ Hardware Description Language (HDL)
  - ➢ Verilog
    - o Introduction
    - o Language Fundamentals
    - o Modeling Combinational & Sequential Logic Circuits
    - o Modeling Finite State Machines
    - o Verilog Operations

ASIC/FPGA Design Flow



❑ In this course we learn all the above steps in detail for ASIC

1. HDL Coding

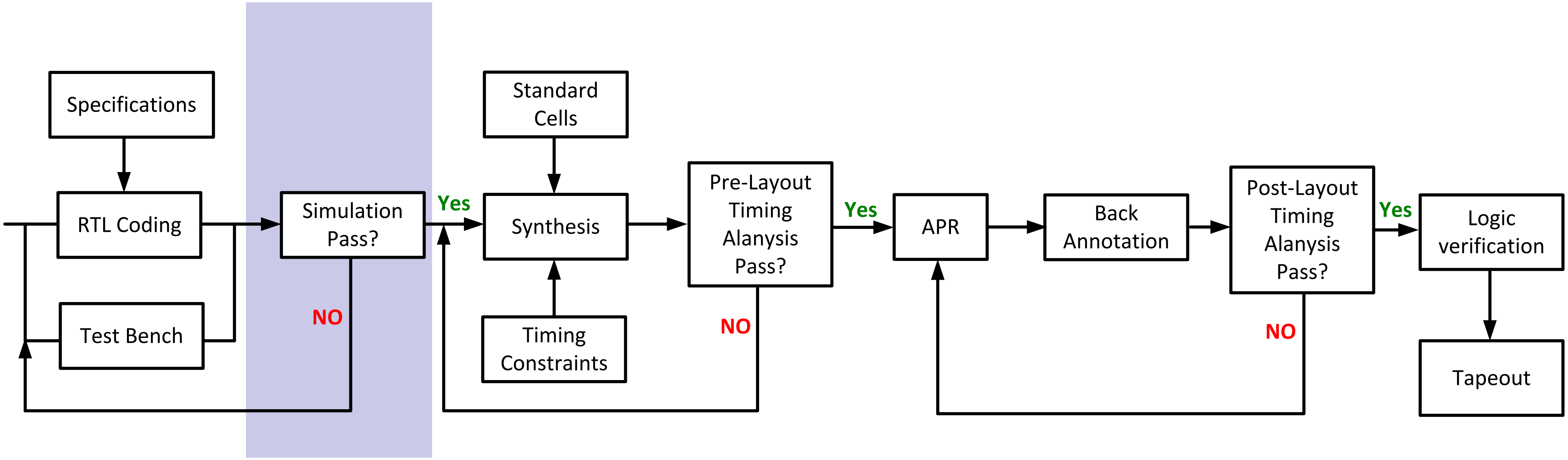


❑ HDL allows us to describe the functionality of a logic circuit in a language that is:

- ➢ Easy to understand
- ➢ Easy to share
- ➢ Hides complicated implementation details

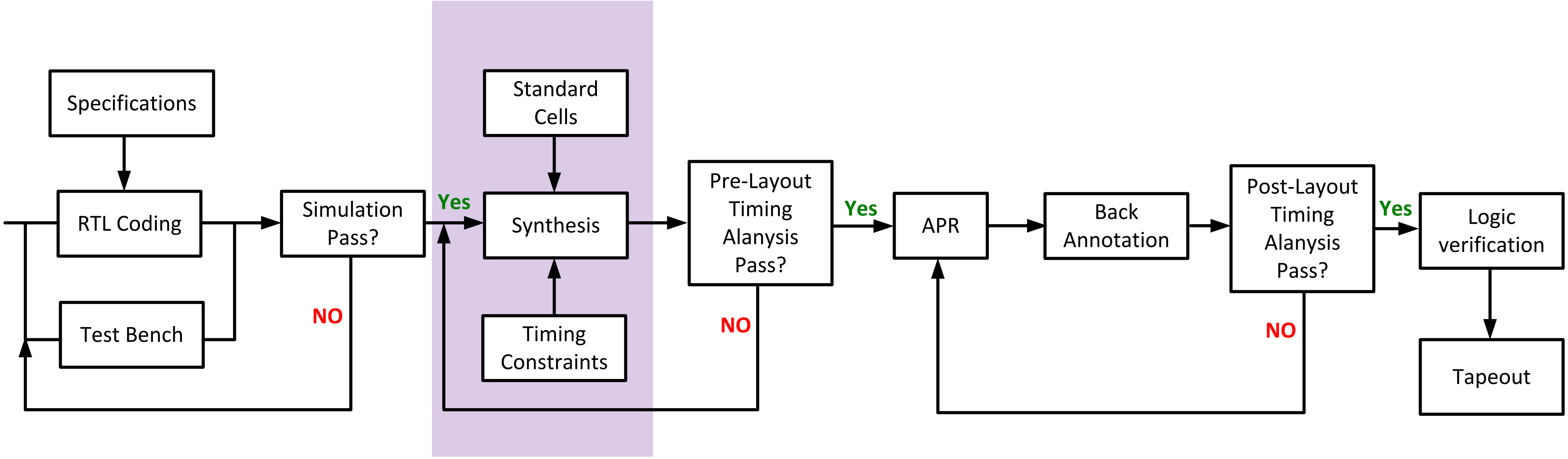
❑ Designer more concerned about the design functionality than the detailed circuit design

2. Simulation by Testbenches



- ❑ After HDL coding, the code has to be tested using “testbenches” (Verification).
- ❑ Simulation tools:
  - ➢ Synopsys VCS (Synopsys)
  - ➢ Modelsim (Mentor Graphics)
  - ➢ NCVerilog (Cadence)

3. Synthesis



- ❑ Synthesis tool:
  - ➢ Analyzes a piece of Verilog code and converts it into optimized logic gates
  - ➢ This conversion is done according to the **"language semantics"**
  - ➡ We have to learn these language semantics, i.e., Verilog code.

3. Synthesis

❑ Why using synthesis tools?

➢ It is an important tool to improve designers' productivity to meet today's design complexity.

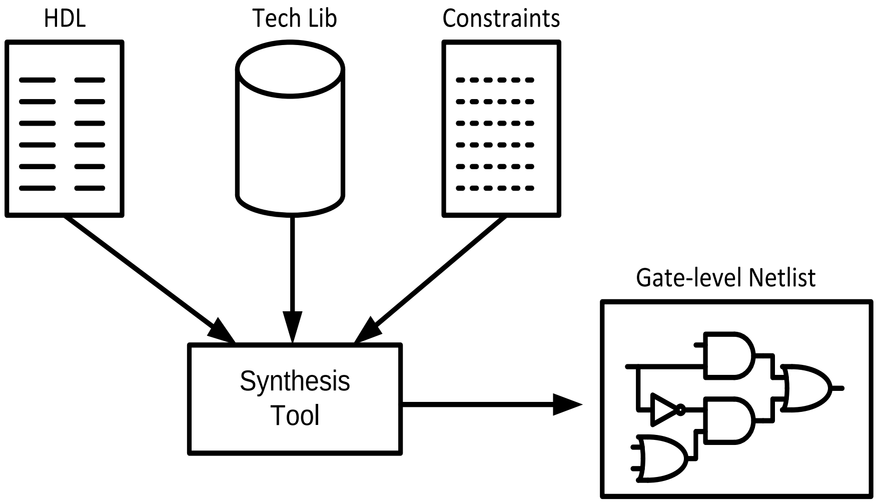
➢ If a designer can design 150 gates a day, it will take 6666 man's day to design a 10-million gate design, or almost 20 years for 10 designers! This is assuming a linear grow of complexity when design gets bigger.

3. Synthesis

- ❑ Synthesis tool:
  - ➢ **Input:**
    - ▪ HDL Code
    - ▪ "Technology library" file ➡ Standard cells (known by transistor size, 90nm)
      - o Basic gates (AND, OR, NOR, …)
      - o Macro cells (Adders, Muxes, Memory, Flip-flops, …)
    - ▪ Constraint file (Timing, area, power, loading requirement, optimization Alg.)
  - ➢ **Output:**
    - ▪ A gate-level "Netlist" of the design
    - ▪ Timing files (.sdf)

3. Synthesis Tools

- ❑ Synthesis tool:

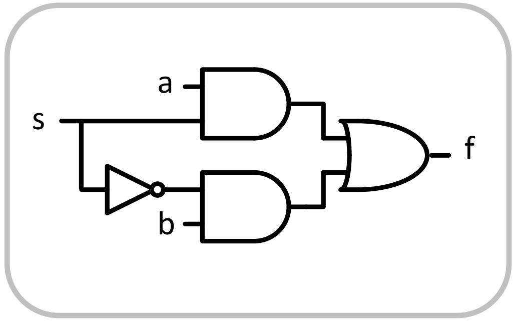


❖ **Example:** A 2-to-1 Multiplexer (2x1-MUX)

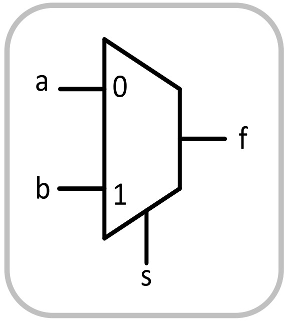
```
If (s==0)
   f = a;
else
   f = b;
```

Verilog code
(has to comply with certain structures)

Synthesis Tool



Synthesized gate-level



Schematic

3. Synthesis

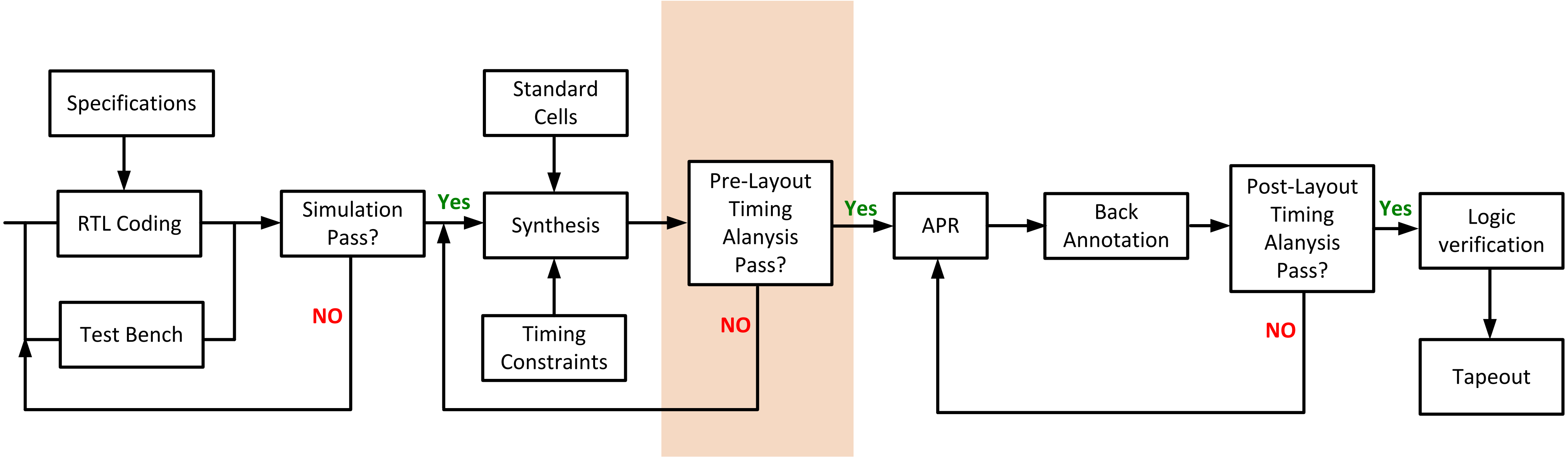
- Synthesis tools:
  - Infer logic and state elements
  - Perform technology-independent optimizations
    - e.g., logic simplification, state assignment
  - Map elements to the target technology
  - Perform technology-dependent optimizations
    - Multi-level logic optimization
    - Choose gate strengths to achieve speed goals

Synthesis Tools

- Commercial Synthesis Tools:

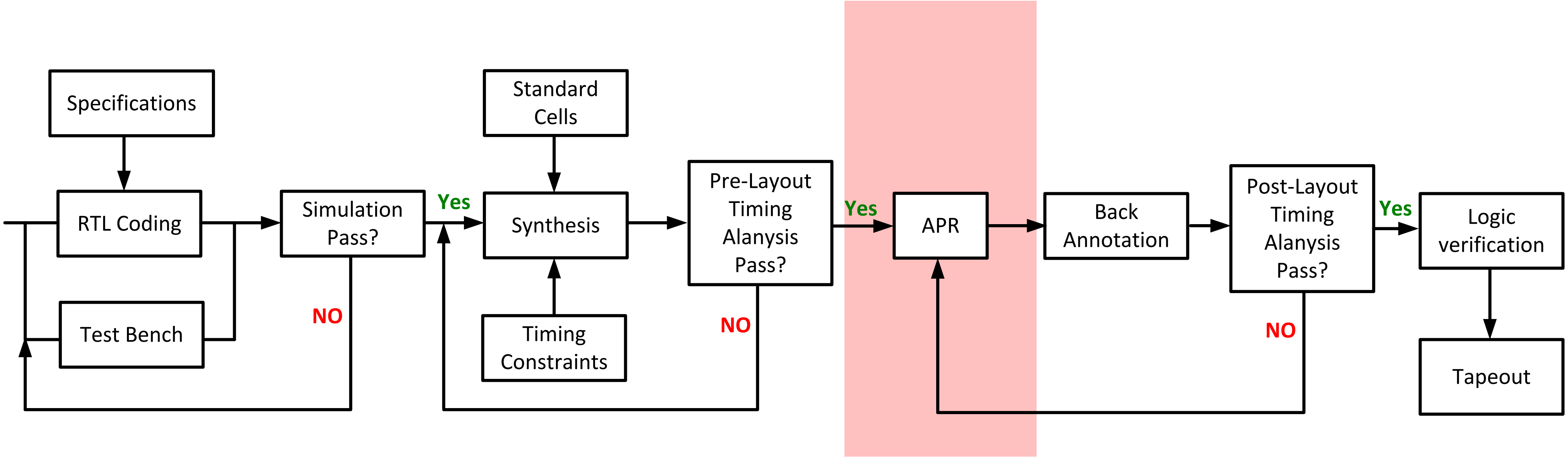
Vendor Name	Product Name	Platform
Altera	Quartus II	FPGA
Xilinx	ISE	FPGA
Mentor Graphics	Modelsim, Precision	FPGA/ASIC
Synopsys	Design Compiler, Galaxy	ASIC
Synplicity	Synplify	ASIC
Cadence	Ambit, BG, RC	ASIC

4. Pre-Layout Timing Analysis



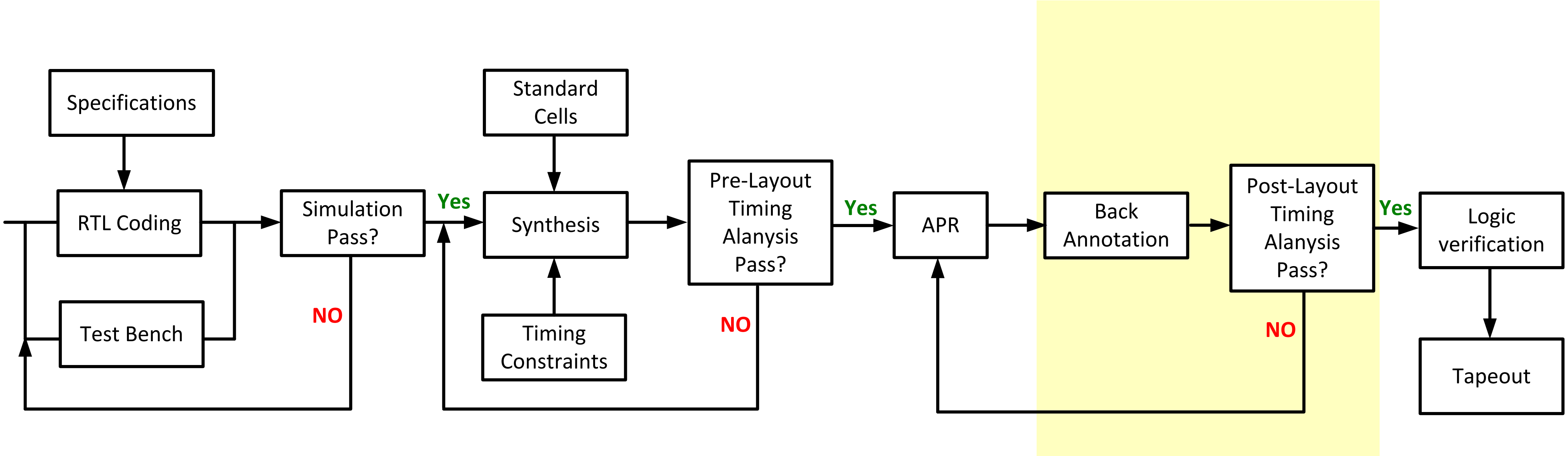
- ❑ Timing analysis across all design corners:
  - ➢ Different voltages and temperatures
  - ➢ Check for setup-time and hold-time violation
- ❑ Rough estimation as wire delays and RC models are not considered

5. APR



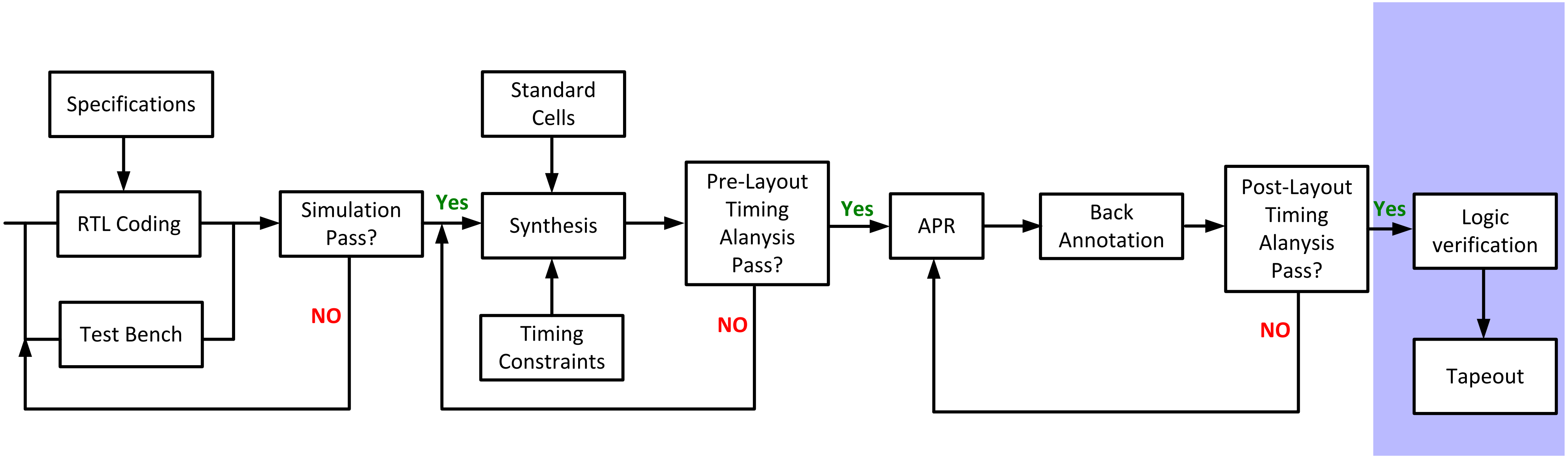
- Automatic Placement and Routing (APR)
  - Floorplan (Die size, Pad configuration, Die-to-pad space)
  - Placement (where each submodule sits in the chip)
  - Routing (metal wiring to connect all instances together according to the netlist)

6. Back Annotation & Timing Analysis



- ❑ Back Annotation (Timing Closure)
  - ➢ To estimate the delay after tapeout
  - ➢ Extraction of RC parasitics in the layout netlist interconnect delay
  - ➢ Some paths might now violate (setup-time and hold-time)
  - ➢ Causes increase in the path delay (specially in deep submicron)

7. Logic Verification & Tapeout



- Logic Verification
  - Simulate and test the very final netlist after APR
  - Timing analysis using testbenches
  - Send the final design (GDS file) for fabrication

Outline

❑ ASIC/FPGA Design Flow

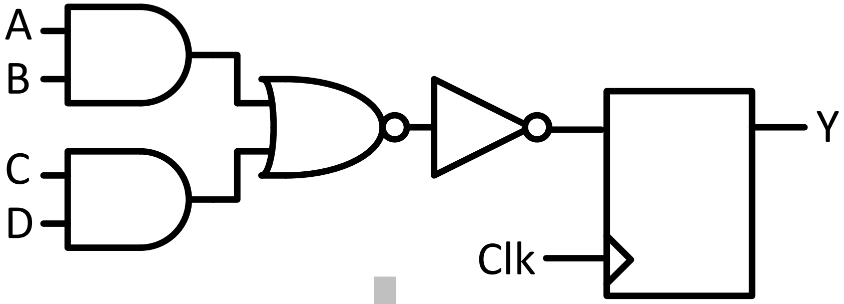
❑ **Hardware Description Language (HDL)**

- ➢ **Verilog**
  - o Introduction
  - o Language Fundamentals
  - o Modeling Combinational & Sequential Logic Circuits
  - o Modeling Finite State Machines
  - o Verilog Operations

Introduction: Digital Logic Design

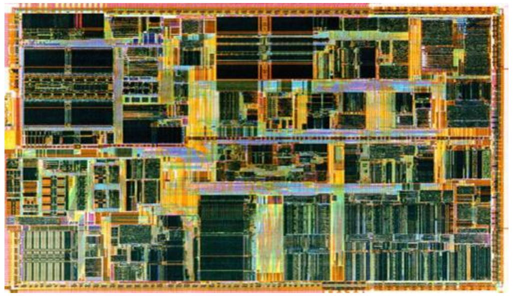
- Conventional Approach:
  - Schematic Entry → good for fairly small designs

    (Draw K-maps, optimize the Boolean logic, draw the schematic)



(10 gates)

  - Possible for large designs?
    - **NO!**



(10,000,000 gates)

Introduction: Why HDL?

- Schematic entry not feasible for large designs:
  - Time consuming to draw the schematic for millions of gates
  - Prone to mistakes
  - Difficult design entry and sharing
  - Different design entry tools to learn
  - Tools not compatible (hard to convert the design from one to another)
  - Not easy to modify

- **Solution:**
  - Describe the design in text → Hardware Description Language (HDL)
  - Just describe the design **"behavior"** not the detailed gate-level logic
  - Gate-level logic is generated automatically by a **"synthesis"** tool

Introduction: Why HDL?

- ❑ Complicated designs can be easily described by HDL
- ❑ Can be used as the input to the synthesis tool
- ❑ Supports behavioral and structural descriptions
- ❑ Supports bit-level descriptions
- ❑ Detailed design cycle-by-cycle timing is supported
- ❑ Concurrent cores can be implemented and simultaneously simulated, which is vital to describe the hardware systems
  - ➢ Software programming languages typically have no concept of time. In hardware, there are delays associated with going from an input to an output.

HDL Coding

This image cannot currently be displayed.

Advantages of HDL Coding

❑ Designer describes what the hardware should do without actually designing the hardware itself

❑ HDL Coding allows designers to separate behavior from implementation

❑ Designers develop an executable functional specification that documents the exact behavior of all the components and their interfaces

❑ Designers can make decisions about cost, performance, power, and area earlier in the design process

Advantages of HDL Coding

There are several benefits to using an HDL to describe your design:

- An HDL facilitates a top-down design methodology using synthesis
  - You can design at a high implementation-independent level
  - You can delay decisions on implementation details
  - You can easily explore design alternatives
  - You can solve architectural problems before implementation
  - You can automate mapping of your high-level description to a technology-specific implementation

- An HDL provides greater flexibility
  - You can re-use earlier design components
  - You can move your design between multiple vendors and tools

HDL Coding Goals

1. To simulate digital designs

2. To synthesize digital designs

- ❑ Some tools can automatically manipulate the design for verification, synthesis, optimization, etc.
  - ➢ Computer Aided Design (CAD) tools

HDL is NOT a Software Programming Languae

- ❑ **Software Programming Language**
  - ➢ Language which can be translated into machine instructions and then executed on a computer
- ❑ **Hardware Description Language**
  - ➢ Language with syntactic and semantic support for modeling the temporal behavior and spatial structure of hardware

HDL Coding

❑ A Hardware Description Language is a high-level programming language that offers special constructs, used to model microelectronic circuits

❑ Two standard HDLs:

- ➢ VHDL (**V**ery high-speed integrated circuit **HDL**)
- ➢ Verilog

❑ Verilog:

- ➢ Developed by Philip Moorby in 1985 as a proprietary language
- ➢ Open to public by Cadence Design Systems in 1990
- ➢ IEEE standard in 1995 and revised in 2001

Verilog is used in this course!

Verilog or VHDL?

VHDL	Verilog
Commissioned in 1981 by Department of Defense	Created by Gateway Design Automation in 1985
An IEEE standard	An IEEE standard
Initially created for ASIC Synthesis	Initially an interpreted language for gate-level simulation
Strong support for package management and large designs	No special extensions for large designs
ADA-like verbose syntax, lots of redundancy	C-like concise syntax
Design is composed of **entities** each of which can have multiple architectures	Design is composed of **modules** which have just one implementation
Gate-level, dataflow, and behavioral modeling. Synthesizable subset.	Gate-level, dataflow, and behavioral modeling. Synthesizable subset.
Harder to learn and use	**Easy to learn and use**

Verilog in Three Flavors

- There are three types of Verilog Coding:
  - **Behavioral:**
    - Describes a system **by the flow of data** between its functional Blocks
    - Defines signal values when they change

Most Descriptive

  - **Structural:**
    - Shows detailed design components, nets, and interconnects
    - Uses technology-specific, low-level components
    - Used to pass netlist information b/w design tools (e.g., from DC to APR)

Least Descriptive

  - **RTL (Register Transfer Level):**
    - Describe how data transfers b/w registers and input/outputs
    - Describes a system by **the flow of data and control signals** between and **within its functional blocks**
    - Defines signal values **with respect to a clock**

Focus of this course

Somehow Descriptive

Verilog Coding Styles

RTL

```
module RTL ( A,  B, C,  D, Out);
  input   A,  B, C, D;
  output Out;
  reg Out;
  always @ (A or B or C or D)
    begin
      if (A & B & ~D)
        Out = C;
      else if (A & D & ~C)
        Out = B;
      else
        Out = 0;
    end
endmodule
```

Our Focus

Synthesizable

Behavioral

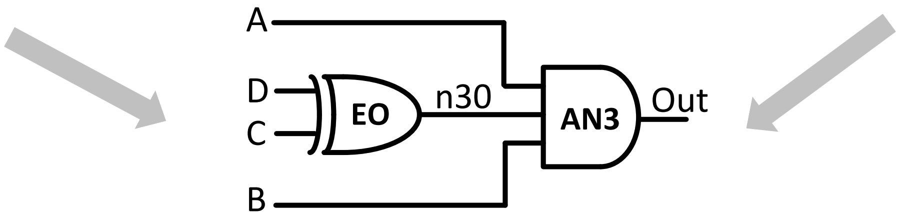
```
module behavior (A,B, C, D, Out);
  input A, B, C, D;
  output Out;
  reg Out;
  always @ (A or B or C or D)
    begin
      if (A & B & ~D)
        Out = #5 C;
      else if (A & D & ~C)
        Out = #3 B;
      else if ((A ==1'bx) | (B ==1'bx) |
                 (C ==1'bx) |(D ==1'bz))
        Out = #7 1'bx;
      else if ((A ==1'bz) | (B ==1'bZ))
        Out = #7 1'bZ;
      else
        Out = #3 0;
    end
endmodule
```

Not synthesizable!

Structural

```
module structural (A,B, C, D, Out);
  input A, B, C, D;
  output Out;
  wire n30;
  EO U9 ( .A(D), .B(C), .Z(n30) );
  AN3 U8 ( .A(A), .B(n30), .C(B), .Z(Out) );
endmodule
```

Synthesizable

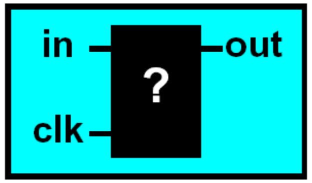


Verilog in Three Flavors : Behavioral

The behavioral level describes the behavior of a design without implying any specific internal architecture:

- You use high level constructs, such as **@**, **case**, **if**, **repeat**, **wait**, **while**
- You can use any behavioral construct of the HDL in your testbench
- Synthesis tools accept only a limited subset of these behavioral constructs

This behavioral model defines the behavior of the design as seen at its ports:

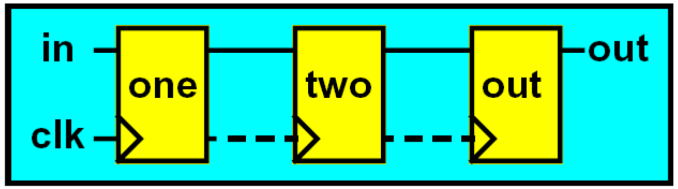


```
module pipe ( out, in, clk );
output out; reg out;
input  in, clk;
  always @(in)
    @(posedge clk)
      out <= repeat(2) @(posedge clk) in;
endmodule
```

Verilog in Three Flavors : RTL

The RTL (functional) level describes the design architecture in sufficient detail that a synthesis tool can construct the circuit.

This functional model defines three storage elements and their assignments:



```
module pipe ( out, in, clk );
output out; reg out;
input  in, clk;
  reg one, two;
  always @(posedge clk) begin
      out <= two;
      two <= one;
      one <= in;
    end
endmodule
```

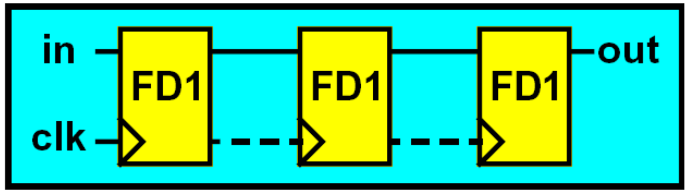
Verilog in Three Flavors : Structural

Synthesis tools produce a purely structural design description.

The structural level is also appropriate for small library components:

- You can use built-in Verilog primitives, such as the **and** gate
- You can describe your own User Defined Primitives (UDPs)

This structural model instantiates predefined library components:

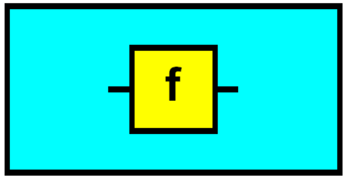


```
module pipe ( out, in, clk );
output out;
input in, clk;
  FD1 one_reg(.Q(one), .D(in ), .CP(clk));
  FD1 two_reg(.Q(two), .D(one), .CP(clk));
  FD1 out_reg(.Q(out), .D(two), .CP(clk));
endmodule
```

Verilog Coding Styles: Levels of Abstraction

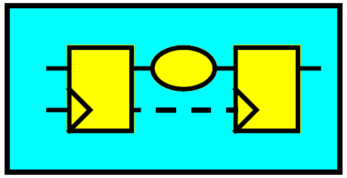
At each level of abstraction, you can describe a system as a group of hierarchical models in varying amount of detail. EDA tools facilitate this process.

Behavioral representation



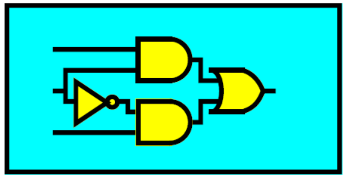
Verilog simulation, behavioral synthesis

Functional representation



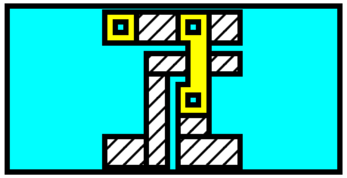
Verilog simulation, logic synthesis

Structural representation



Verilog simulation
static functional analysis
static timing analysis
place & route

Physical representation



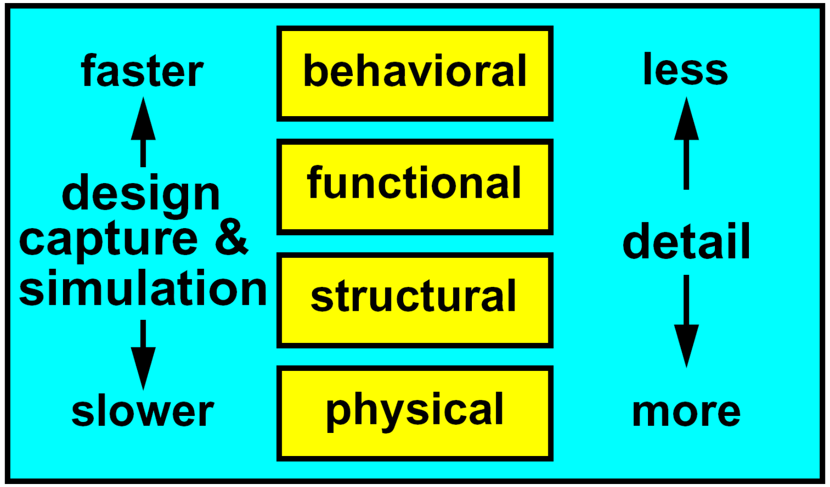
Spice simulation
design rule checking
parasitic analysis

Verilog Coding Styles: Levels of Abstraction

- ❑ **Trade-offs:**

Each level of abstraction permits modeling at a higher or lower level of detail.

More detail means more work for you and the simulator.



faster ↑ design capture & simulation ↓ slower

behavioral

functional

structural

physical

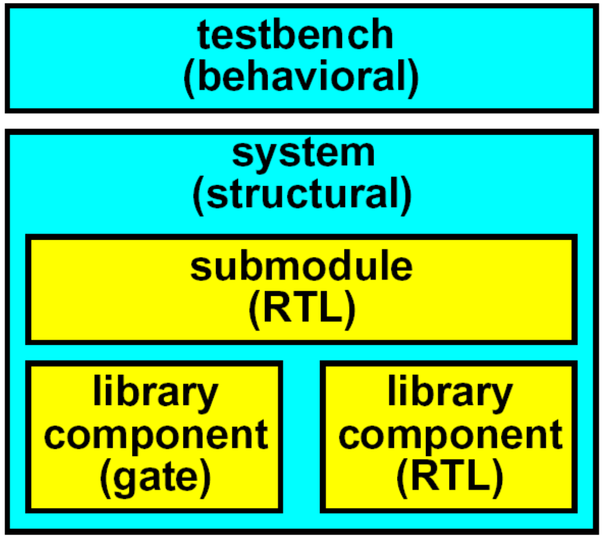
less ↑ detail ↓ more

Verilog Coding Styles: Levels of Abstraction

❑ One language for all levels:

Designers usually mix levels of abstraction within a simulation:

- RTL and gate-level library components
- RTL functional submodule descriptions
- Structural system netlist
- Behavioral system testbench

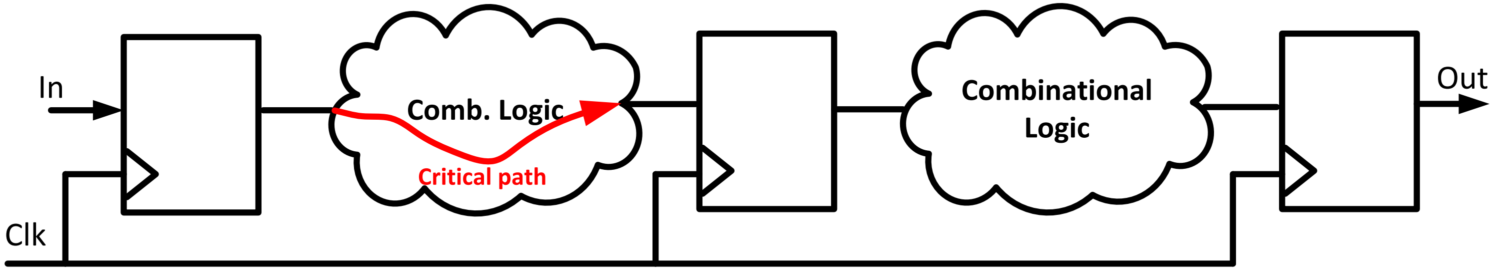


Verilog Coding Styles: Design Style

- Verilog, like any other hardware description language, permits a design in either Bottom-up or Top-down methodology.
  - **Bottom-Up Design**
    - The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standard gates. With the increasing complexity of new designs this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom-up designs have to give way to new structural, hierarchical design methods.
  - **Top-Down Design**
    - A real top-down design allows early testing, easy change of different technologies, a structured system design and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are a mix of both methods, implementing some key elements of both design styles.

Verilog for Synthesis (RTL)

- ❑ In this course we focus on RTL coding
- ❑ RTL coding is the closest one to the actual hardware implementation
- ❑ RTL code includes a subset of all Verilog syntax
  - ➢ Not all Verilog syntax are synthesizable
- ❑ We cover most Verilog coding parts that are needed for logic synthesis
- ❑ Simulation of the RTL code is also covered
- ❑ We learn how to write a “good” Verilog code for synthesis
- ❑ Lots of examples on the synthesized RTL!



Verilog Applications

The Verilog HDL is used by:

- ❑ **System Architects:** doing high level system simulations
- ❑ **Verification Engineers:** writing advanced tests for all levels of simulation
- ❑ **ASIC and FPGA Designers:** writing RTL code for synthesis
- ❑ **Library Developers:** describing ASIC or FPGA cells, or higher level components

Outline

- ❑ ASIC/FPGA Design Flow
- ❑ **Hardware Description Language (HDL)**
  - ➢ **Verilog**
    - o Introduction
    - o Language Fundamentals
    - o Modeling Combinational & Sequential Logic Circuits
    - o Modeling Finite State Machines
    - o Verilog Styles for Synthesis
    - o Testbench and Simulation

Verilog Fundamentals : Comment

- ❑ Comments are used for documentation

- ❑ Comments are in two types:
  - ➢ Short comments (single line)

```
// This is a comment
```

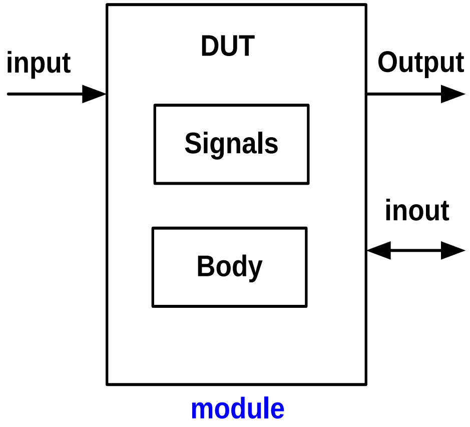
  - ➢ Long comments (Multiple lines)

```
/* This a multiple
   line comment
   in Verilog */
```

- ❑ Space, tab and blank lines are ignored by the compiler

Verilog Fundamentals : Module

- Any circuit or subcircuit is declared as a "module" in Verilog.

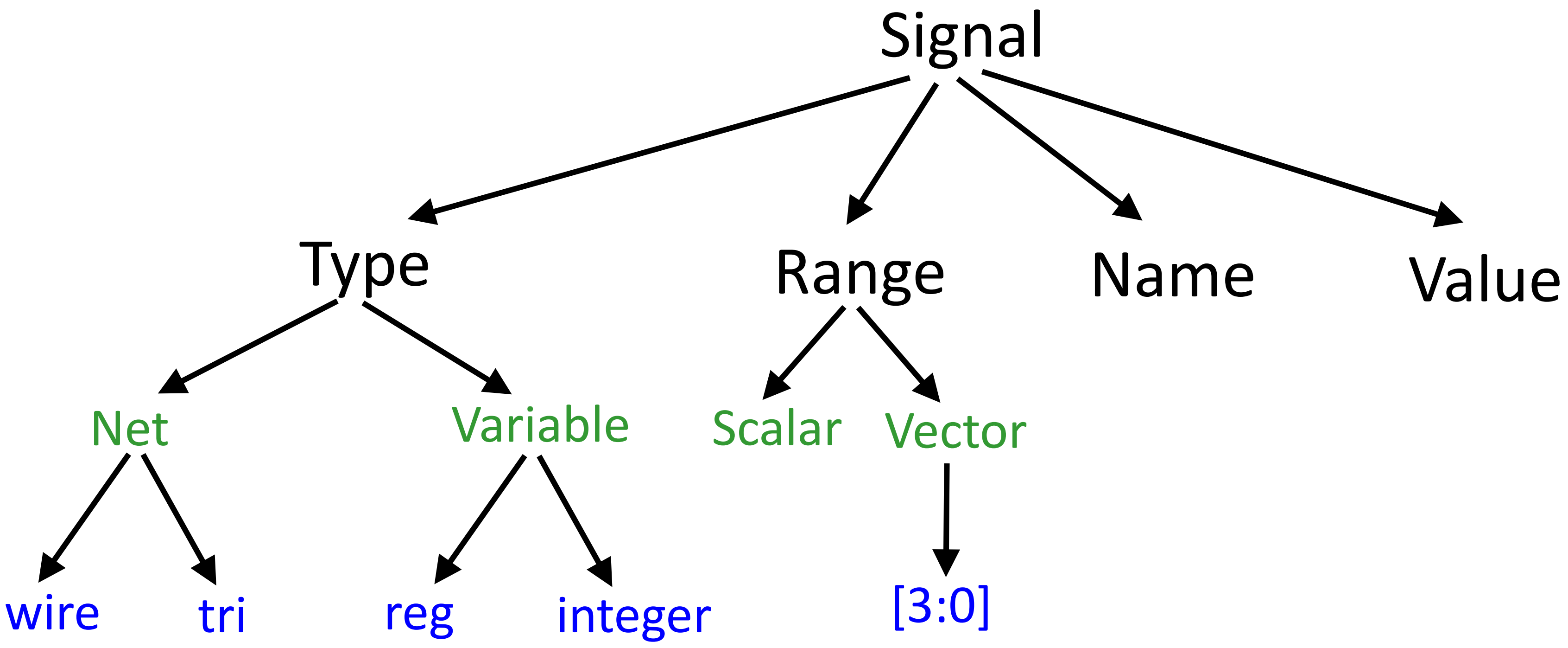


- Each module may have:
  - **Ports** (Three possibilities),
    - input
    - output
    - inout
  - **Signals** (main or intermediate)
  - **Body-code**

    (statements for module description)

```
module  DUT (A, B, C);
   input A;
   output B;
   inout C;
   Signals
   Body-code
endmodule
```

Verilog Fundamentals : Signals



Signal

Type

Range

Name

Value

Net

Variable

Scalar

Vector

wire

tri

reg

integer

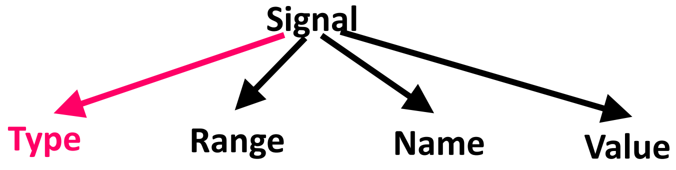
[3:0]

❖ **Example:** wire [2:0] tmp ;
tmp = 3’b001;

tmp[0]=1
tmp[1]=0
tmp[2]=0

Each element of a vector can be accessed

Verilog Fundamentals : Signal Type



Signal

Type

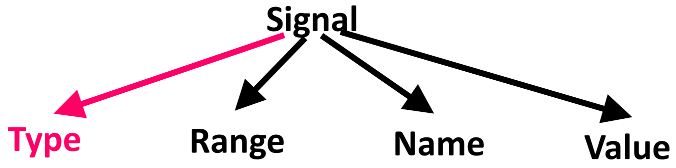
Range

Name

Value

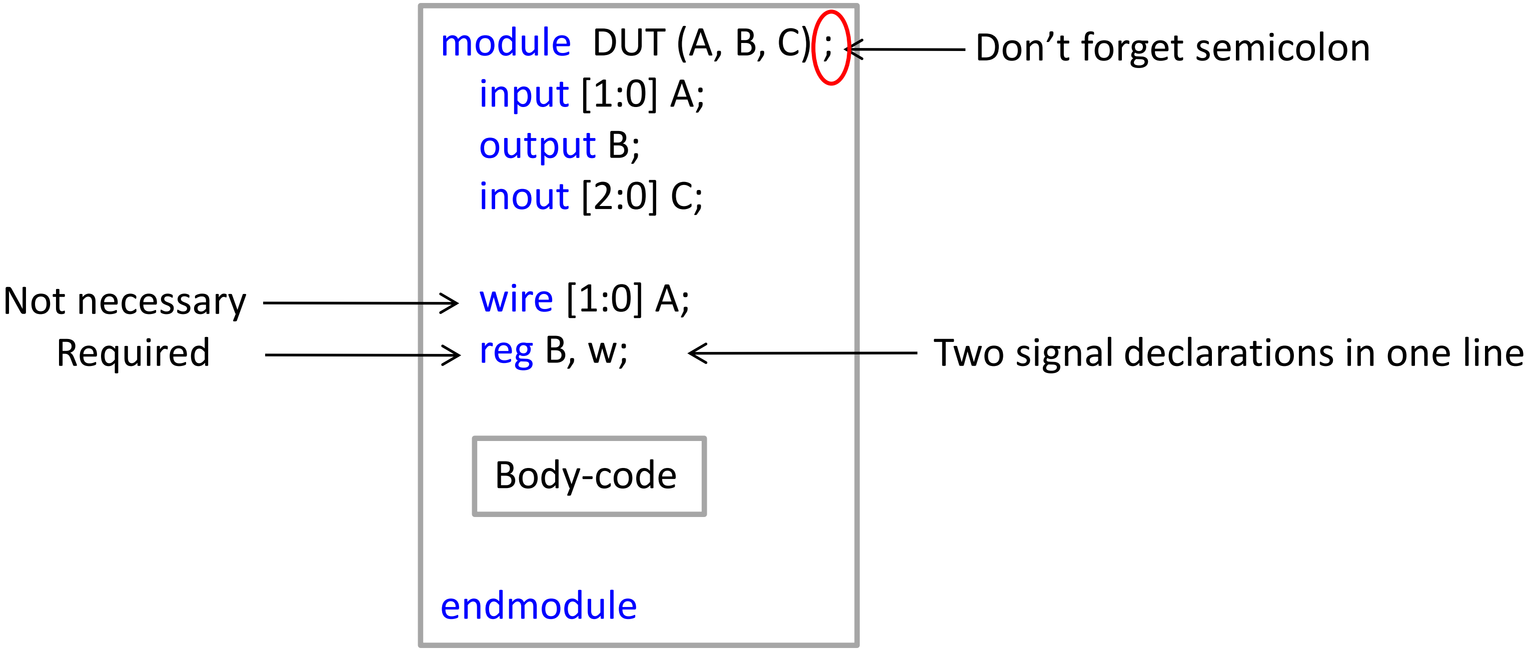
- ❑ **Net**
  - ➢ **wire:**
    - For interconnecting logic elements (LEs)
    - To connect an output of a logic element to the input of another LE
  - ➢ **tri**
    - Circuit nodes that are connected in a tri-state fashion
- ❑ **Variable**
  - ➢ **reg** (unsigned in general)
    - Corresponds to a circuit node (not necessarily a register!)
    - Allow a circuit to be described in terms of its behavior
    - Retains its value until it is overwritten by a subsequent assignment
  - ➢ **integer** (signed in general)
    - Used for loop counters

Verilog Fundamentals : Signal Type



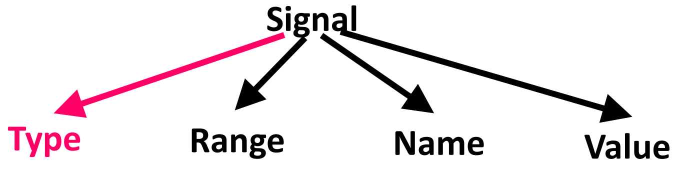
- ❑ The "wire" declarations are not necessary as Verilog assumes that signals are nets by default .
- ❑ The "reg" declaration is required!

❖ Example:

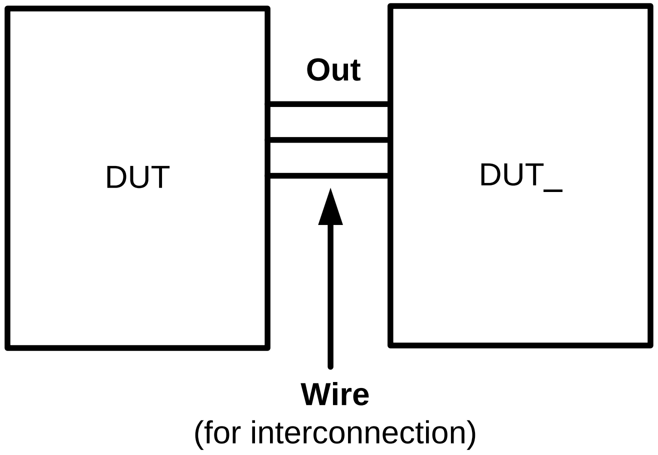


```
module  DUT (A, B, C);
   input [1:0] A;
   output B;
   inout [2:0] C;

   wire [1:0] A;
   reg B, w;

     Body-code

endmodule
```

Verilog Fundamentals : Signal Type



❖ Example:



Ports

Signals

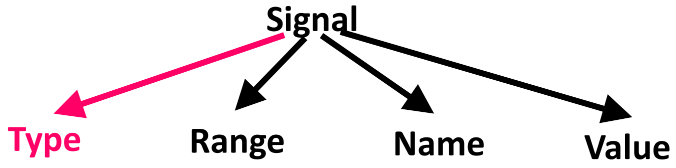
Code Body

```
module  DUT (s, Out);
   input [3:0] s;
   output [2:0] Out;

   wire [2:0] Out;
   reg [2:0] Count;
   integer k;

   Count = 0;
   for (k=0; k<4; k=k+1)
      if (s[k])
         Count = Count + 1;
   assign Out = Count;

endmodule
```

Loop counter

";" at the end of each line

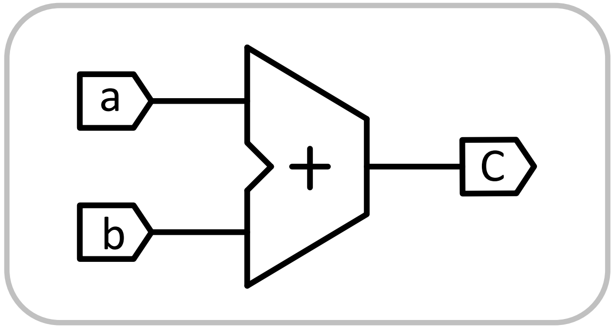
Reg Type



- □ The keyword “reg” does NOT necessarily denote a storage element or register.
- □ “reg” only models the behavior of a circuit.
- □ May or may not be synthesized as a register.

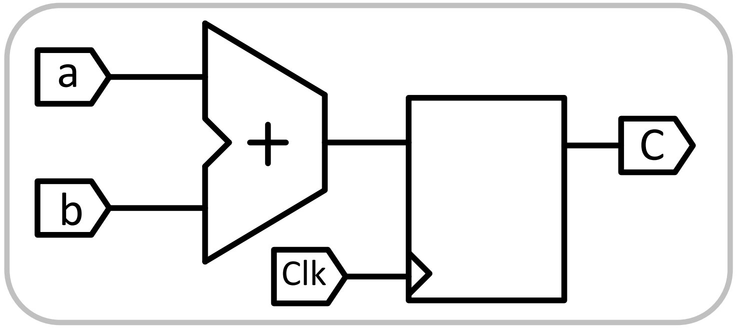
Not Register

```
reg  C;
   always @ (a,b)
        C = a+b;
```



Register

```
reg  C;
   always @ (posedge Clk)
        C <= a+b;
```



Verilog Fundamentals : Signal Range

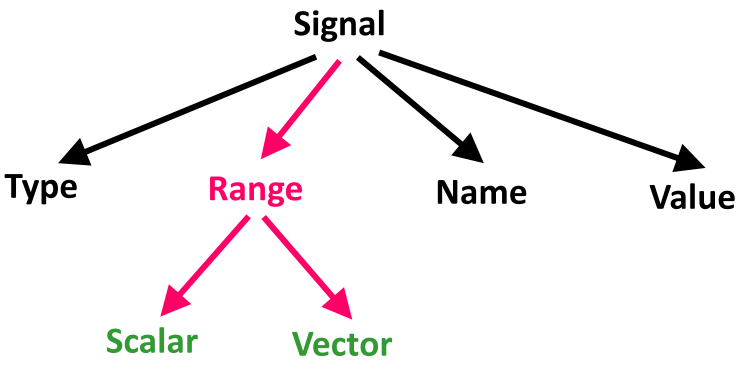
❑ Signals in Verilog can be:

- **Scalar**: representing a node

```
reg  C;
wire B;
```

- **Vector**: representing a bus

```
reg  [10:0] Data;
reg  [0:6] S;
wire [7:4] B;
```

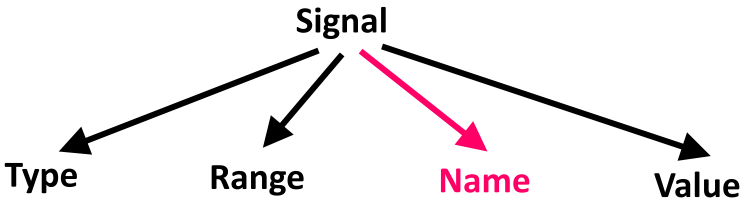


❑ Each element of a bus can be accessed.

```
assign a = Data[8];
```

Verilog Fundamentals : Signal Name

- ❑ Signal name may consists of:
  - ➢ Any letter
  - ➢ Any digit
  - ➢ Underscore (_) and $ sign
- ❑ **DON'Ts:**
  - ➢ Should not start with a digit
  - ➢ Should not be a Verilog keyword



Legal

A_m
B1_signal
My$

Illegal

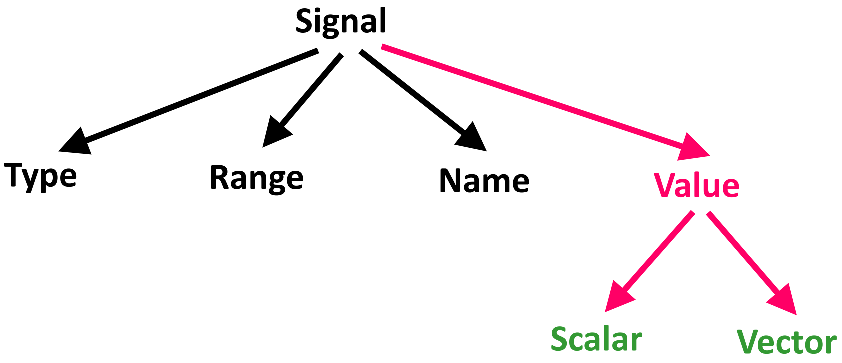
1xb
wire
R&z

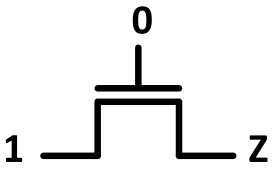
Note: Verilog is case sensitive!

Verilog Fundamentals : Signal Value

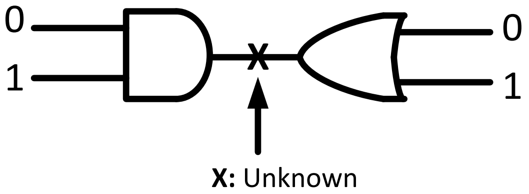
❑ **Scalar**: each scalar signal can have four possible values:

- **0:** Logic value "0"
- **1:** Logic value "1"
- **Z(z):** Tri-state (high impedance)





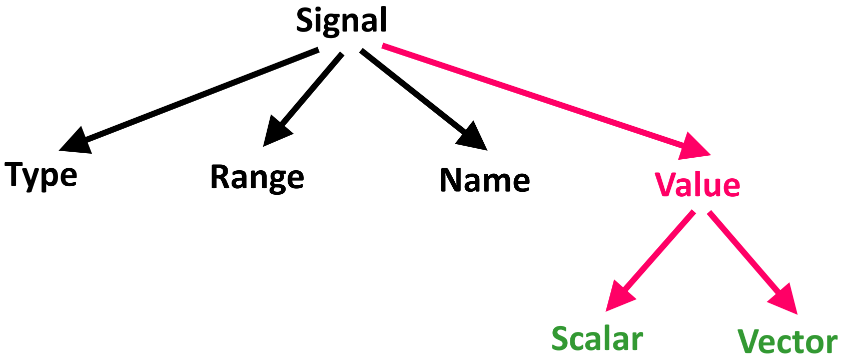
- **X(x):** Unknown value



Verilog Fundamentals : Signal Strength

- ❑ سطوح قدرت به منظور حل اختلافات زمانیکه یک گره به درایورهایی با مقادیر یا قدرتهای متفاوت متصل شده باشد استفاده می شود .بطور مثال حاصل اتصال مقدار strong1 و weak0 مقدار strong1 خواهد بود. اگر دو سیگنال با مقدار مختلف ولی با سطح قدرت یکسان به یکدیگر متصل شوند حاصل نامعلوم (x) خواهد بود.

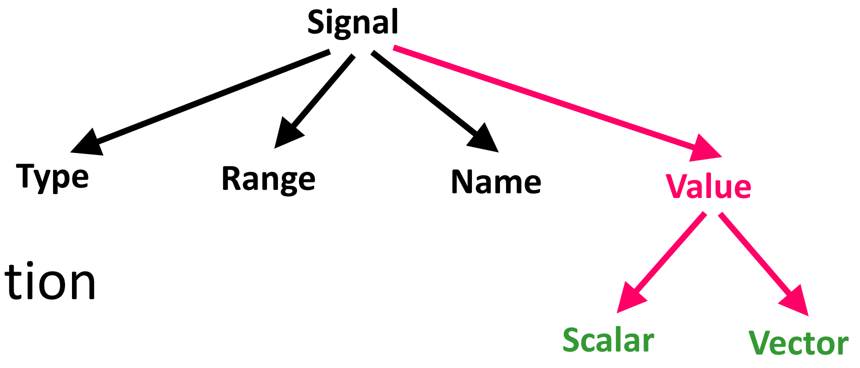
Strength Level	Type	Degree
supply	Driving	Strogest
strong	Driving	
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest



Verilog Fundamentals : Signal Value

- ❑ **Vector:** **<# of bits> <base> <number>**

4 ‘b 0101



- ➢ **<# of bits> :** number of bits for representation
- ➢ **<base> :** (default decimal)
  - **“d” : Decimal**
  - **“b” : Binary**
  - **“h” : Hexadecimal**
  - **“o” : Octal**
- ➢ **<number> :** signal value in base

Used for clarity

❖ **Example:**

```
K = 8’ha9;       // K=1010_1001
C= 4’d3;         // C=0011
D= 4’b100;       // D=0100
F= ‘b10x;        // F=10X
L = -6’b3        // L = 111101
```

Verilog Fundamentals : Parameters

- ❑ A parameter is used as a “constant” to facilitate coding.

❖ Example:

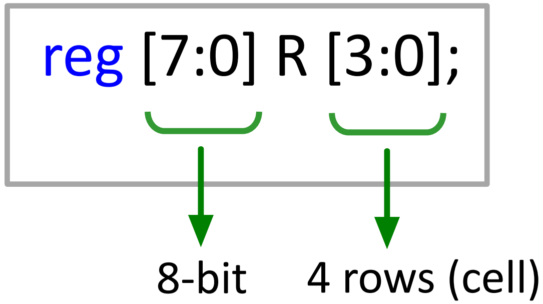
```
module  DUT (s, Out)

   parameter n = 3;
   parameter S0 = 4’b1010;

   input [n-1:0] s;
   output [n:0] Out;

   wire [n:0] Out;
   assign Out = S0;

endmodule
```

Verilog Fundamentals : Memories

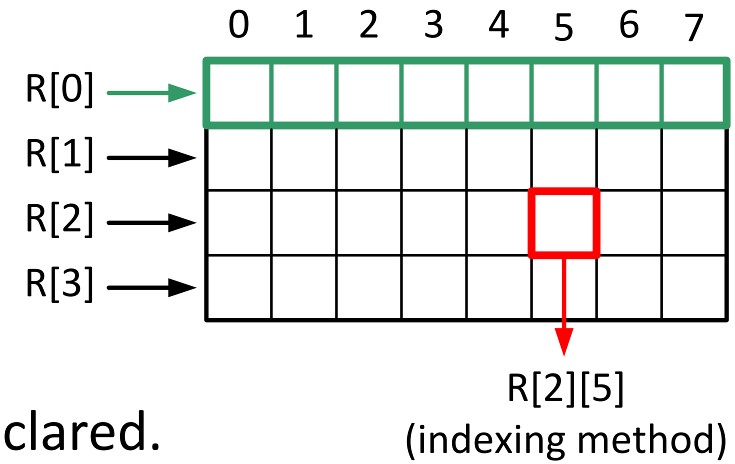
❑ **Memory**:

- ➢ A two-dimensional array of bits
- ➢ Declared in Verilog as a two-dimensional variable (reg)

❖ Example: A 4-byte memory:



```
reg [7:0] R [3:0];
```



❑ A three-dimensional array may also be declared.

❖ Example:

```
reg [7:0] M [3:0][1:0];
```

❑ If an 8-bit A is declared then the legal assignment is:

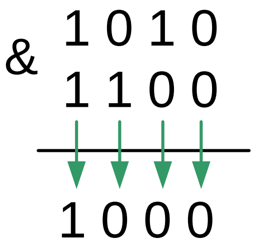
```
reg [7:0] A;   ➡   A = M[3][0];
```

Verilog Fundamentals : Operators

❖ Example:

- **Bitwise:**

Operation	Result
1010 & 1100	1000
1010 | 1100	1110
~1010	0101
1101 ^ 0100	1001



- **Logical:**

X || 1 = 1
X && 0 = 0

Operation	Result
1010 && 1100	1
2'b11 || 2'b00	1
!0010	0
2'b1X && 2'b11	X

Non-zero operand=logical "1"

Any operand X/Z, result is X

Verilog Fundamentals : Operators

❖ Example:

➢ **Reduction:**

Operation	Result
& 1100	0
& 111	1
^ 0100	1

➢ **Relational:**

A=2’b10

Operation	Result
B=(A == 2’b10)	B=1
B=(A == 2’b11)	B=0
B=(A === 2’b1x)	B=0
B=(A <= 2’b11)	B=1

== Used only with 0 and 1

=== Used with x and z

Verilog Fundamentals : Operators

- Example:
  - **Logical Shift:**

A=6’b001100

Operation	Result
C = A >> 1	C = 000110
D = A << 2	D = 110000
F = A >> 3	F = 000001

- **Concatenation:**

A=2’b11
B=3’b010

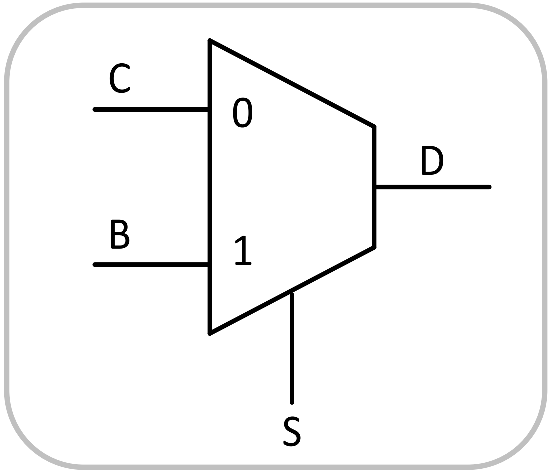
Operation	Result
{A, B}	5’b11010
{3{A}}	6’b111111
{B, B}	6’b010010
{{3{A}}, {2{B}}}	12’b111111010010

Be generous in {}

Verilog Fundamentals : Operators

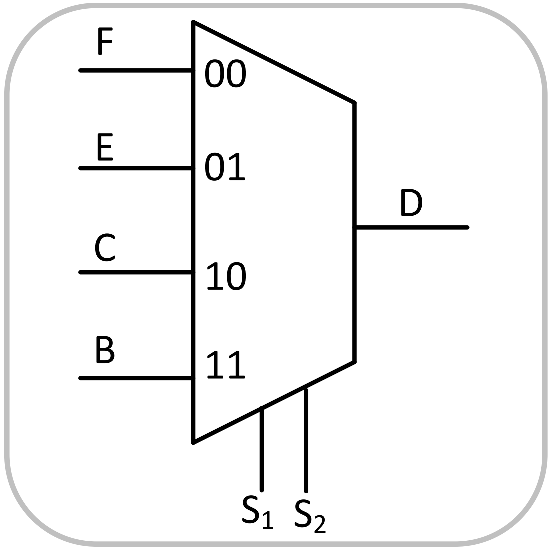
- **Conditional: (? , :)**
  - D = S ? B:C;

$$D = \begin{cases} B & \text{if } S=1; \\ C & \text{if } S=0; \end{cases}$$



  - D = ({S_1,S_2}==2’b00)? F:
    ({S_1,S_2}==2’b01)? E:
    ({S_1,S_2}==2’b10)? C:B;
  - D = ({S_1,S_2}==2’b00)? F:
    ({S_1,S_2}==2’b01)? E:
    ({S_1,S_2}==2’b10)? C:
    ({S_1,S_2}==2’b11)? B:B;

Default



4-input Multiplexer (MUX)

Verilog Fundamentals : Operators (All in One)

<table>
<tr><th>Type of Operators</th><th>Symbols</th><th></th><th></th><th></th><th></th><th></th></tr>
<tr><td>Concatenate & replicate</td><td>{ }</td><td>{{ }}</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Unary</td><td>!</td><td>~</td><td>&</td><td>^</td><td>^~</td><td>|</td></tr>
<tr><td>Arithmetic</td><td>*</td><td>/</td><td>%</td><td></td><td></td><td></td></tr>
<tr><td></td><td>+</td><td>-</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Logical shift</td><td><<</td><td>>></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Relational</td><td><</td><td><=</td><td>></td><td>>=</td><td></td><td></td></tr>
<tr><td>Equality</td><td>==</td><td>!=</td><td>===</td><td>!==</td><td></td><td></td></tr>
<tr><td>Binary bit-wise</td><td>&</td><td>^</td><td>^~</td><td>|</td><td></td><td></td></tr>
<tr><td>Binary logical</td><td>&&</td><td>||</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Conditional</td><td>? :</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

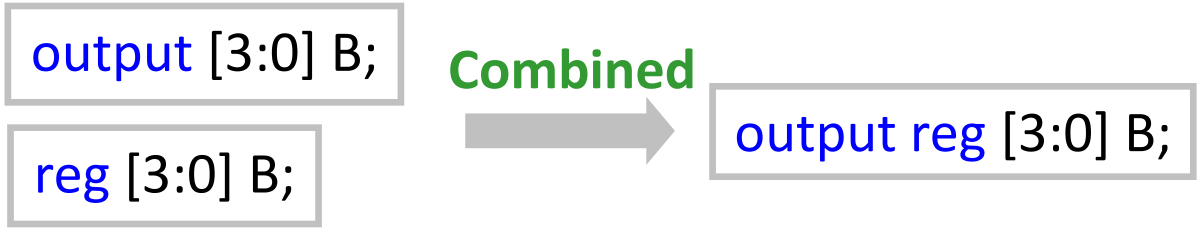
Verilog Fundamentals : Module-Revisited

❑ Any circuit or subcircuit is declared as a “module” in Verilog.

❑ There are three types of ports:

- ➢ input → type “wire”
- ➢ output → type “wire” or “reg”
- ➢ inout → type “wire”

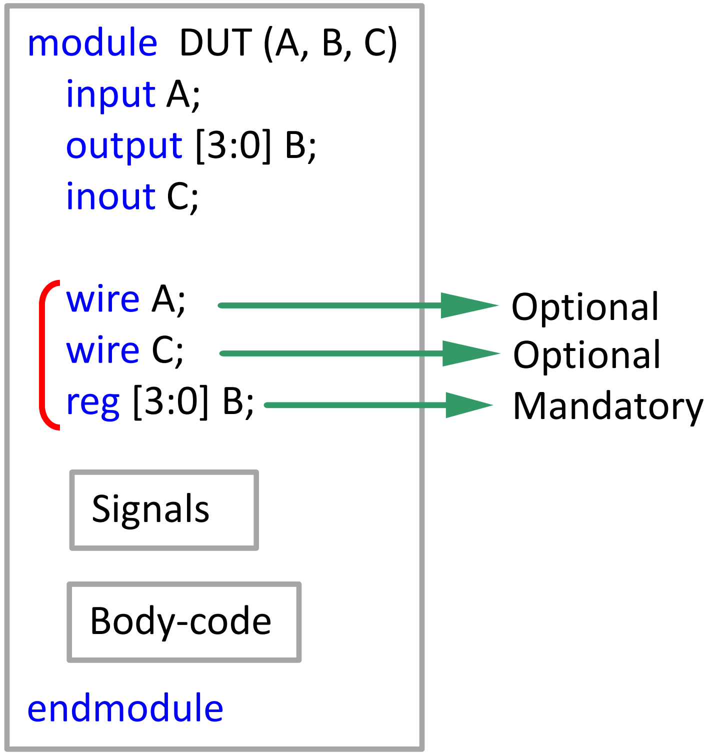
❑ Note:



```
output [3:0] B;
reg [3:0] B;
```

Combined

```
output reg [3:0] B;
```



```
module  DUT (A, B, C)
   input A;
   output [3:0] B;
   inout C;

   wire A;
   wire C;
   reg [3:0] B;

   Signals

   Body-code

endmodule
```

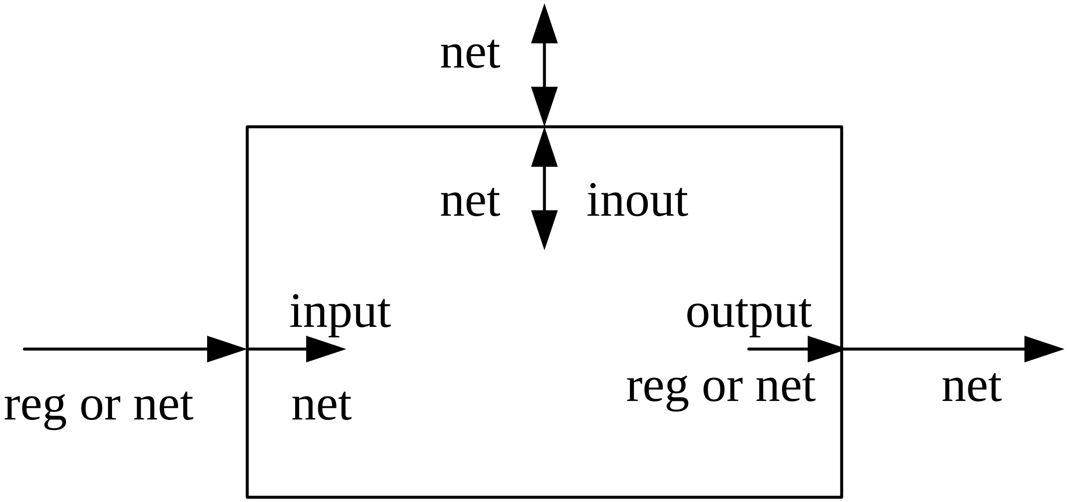
Verilog Fundamentals : Module Ports

❑ Inside view of the module

- ➢ input port: wire
- ➢ output port : wire or reg
- ➢ inout: wire

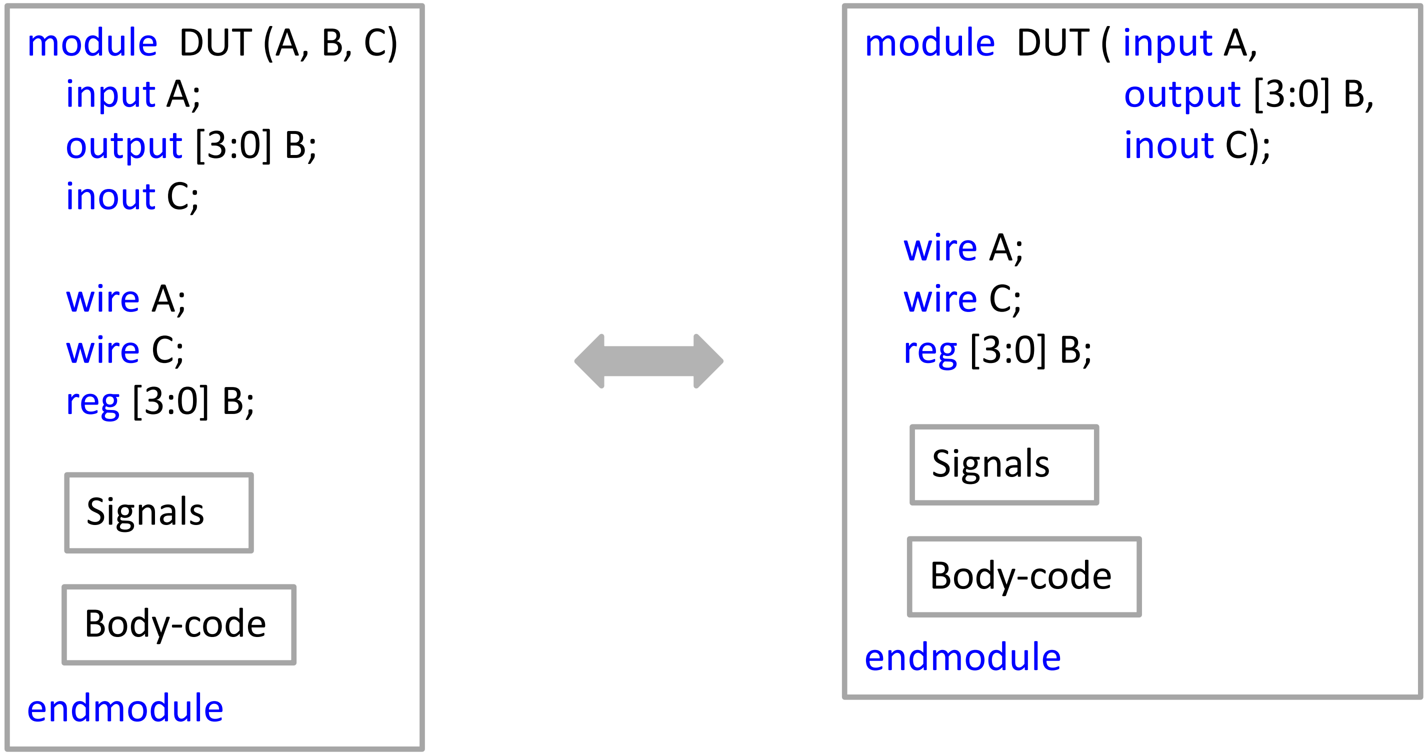
❑ Outside view of the module

- ➢ input port: wire or reg
- ➢ output port : wire
- ➢ inout: wire



Verilog Fundamentals : Module-Revisited

- ❑ In Verilog-2001 the port list can directly follow the module declaration



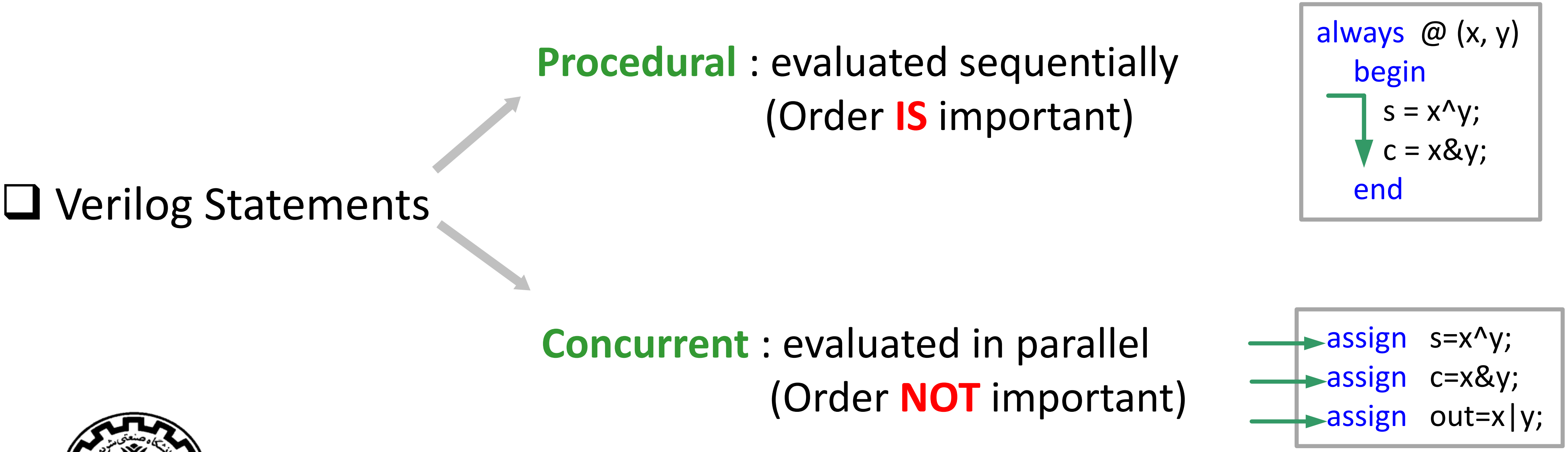
```
module  DUT (A, B, C)
   input A;
   output [3:0] B;
   inout C;

   wire A;
   wire C;
   reg [3:0] B;

    Signals

    Body-code

endmodule
```

```
module  DUT ( input A,
              output [3:0] B,
              inout C);

   wire A;
   wire C;
   reg [3:0] B;

    Signals

    Body-code

endmodule
```

- ❑ Body-code consists of some “statements”
- ❑ Statements describe the circuit/module functionality

Verilog Fundamentals : Statements

- ❑ Programming languages:
  - ➢ **High-Level Language (HLL):** C, Pascal, Matlab
  - ➢ **Hardware Description Language (HDL):** Verilog, VHDL
- ❑ In HLL programming all statements are sequential (procedural)
  - ➢ Statements evaluated in the order and one-bye-one



❑ Verilog Statements

Procedural : evaluated sequentially
(Order **IS** important)

```
always  @ (x, y)
   begin
      s = x^y;
      c = x&y;
   end
```

Concurrent : evaluated in parallel
(Order **NOT** important)

```
assign   s=x^y;
assign   c=x&y;
assign   out=x|y;
```

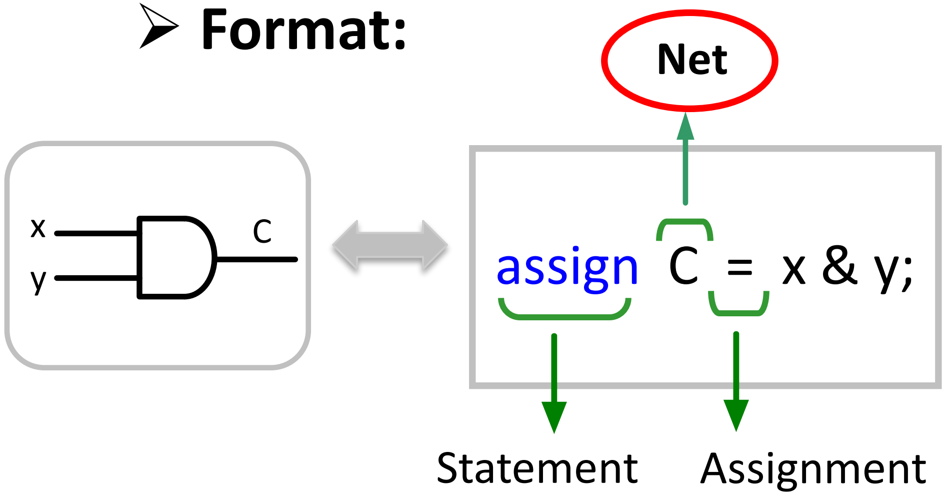
Verilog Fundamentals : Concurrent Statements

❑ Evaluated in parallel

❑ Each statement describes part of the circuit, thus concurrent

❑ Most popular:

- **Continuous statements**: realized as connection or wire in the design
- **Format:**



```
assign C = x & y;
```

❖ Example:

```
wire   [1:3] A, B, C;
assign  C = A&B;
```

Equivalent

```
assign  C[1] = A[1]&B[1];
assign  C[2] = A[2]&B[2];
assign  C[3] = A[3]&B[3];
```

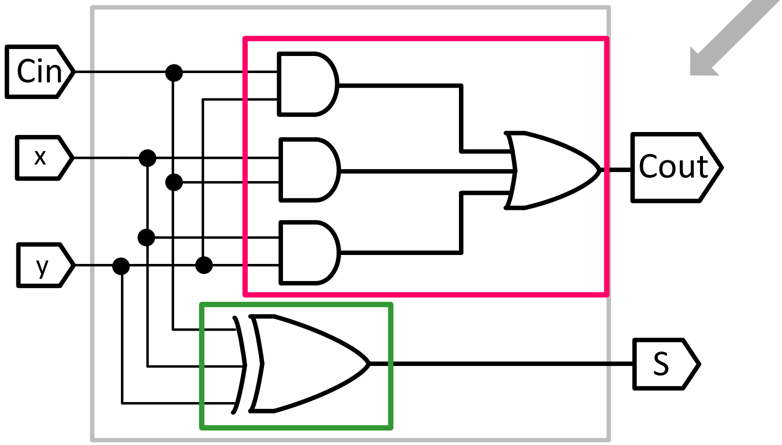
❑ assign used only for nets (to be synthesizable)

Concurrent Statements

❖Example: Full Adder, same circuit, two descriptions:

```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x & y)|(x & Cin)|(y & Cin);

endmodule
```

```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;


   assign {Cout, S} = x + y + Cin;

endmodule
```



x	y	Cin	Cout	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	0	1

Concurrent Statements

❖Example: Signed vs. unsigned addition:

❑ In Verilog “+” declares unsigned addition

❑ Signed addition has to be explicitly specified using the sign extension

```
module Adder_sign (X, Y, S_unsigned, S_signed);
   input  [3:0] X, Y;
   output  [4:0] S_unsigned, S_signed;

   assign S_unsigned = X + Y;

   assign S_signed = {{X[3]},X} + {{Y[3]},Y};

endmodule
```

Sign extension

```
module Adder_sign (X, Y, S_unsigned, S_signed);
   parameter n = 4;
   input  [n-1:0] X, Y;
   output  [n:0] S_unsigned, S_signed;

   assign S_unsigned = X + Y;
   assign S_signed = {{X[n-1]},X} + {{Y[n-1]},Y};

endmodule
```

X = 0011 (unsigned 3) or (signed +3)
Y = 1101 (unsigned 13) or (signed -3)

↓

S_unsigned = 10000 (unsigned 16)
S_signed = 00000 (0 signed)

Concurrent Statements: Sign Extension

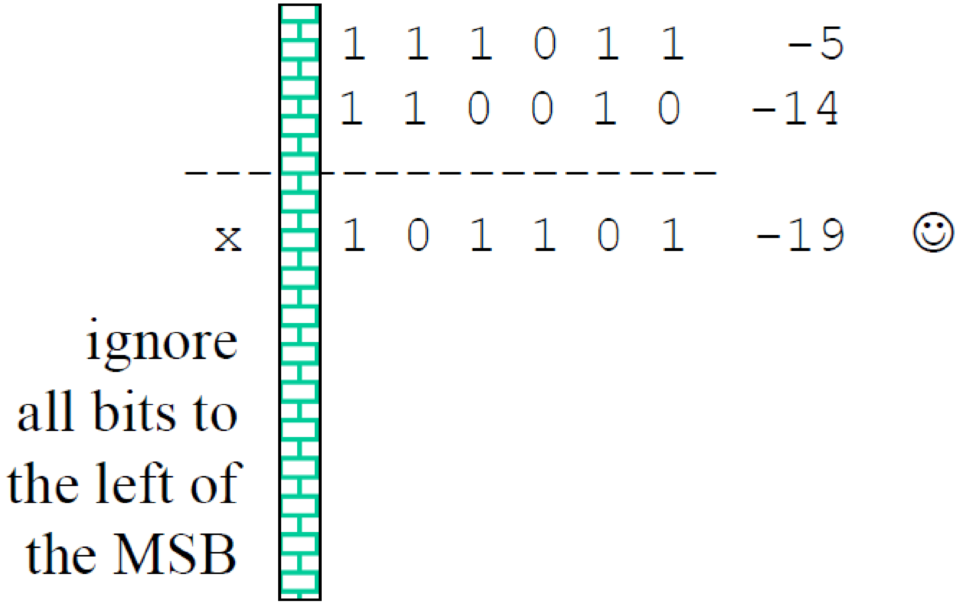
0 0 1 0 1 1   -5
0 1 0 0 1 0   +18
----------------
0 1 1 1 0 1   +29!

1 0 1 1   -5 = -8 + 2 + 1
1 1 0 1 1   -5 = -16 + 8 + 2 + 1
1 1 1 0 1 1   -5 = -32 + 16 + 8 + 2 + 1

1 1 1 0 1 1   -5
0 1 0 0 1 0   +18
----------------
x | 0 0 1 1 0 1   +13   ☺

Concurrent Statements: Sign Extension

- ❑ **Ignore carry bits:** Do not spend any hardware calculating any bits to the left of the answer's MSB



```
     1 1 1 0 1 1    -5
     1 1 0 0 1 0   -14
---------------------
x    1 0 1 1 0 1   -19   ☺
```

ignore
all bits to
the left of
the MSB

Verilog Fundamentals : Delay

- Delay can be used with continuous assignments by using the “#” sign

```
wire  #2  S;
assign  #5  S = x&y;
```

  - 2 time unit of delay on wire S
  - 5 time units of delay for AND gate
  - Any change in x or y reflects on S after 7 time unit delay

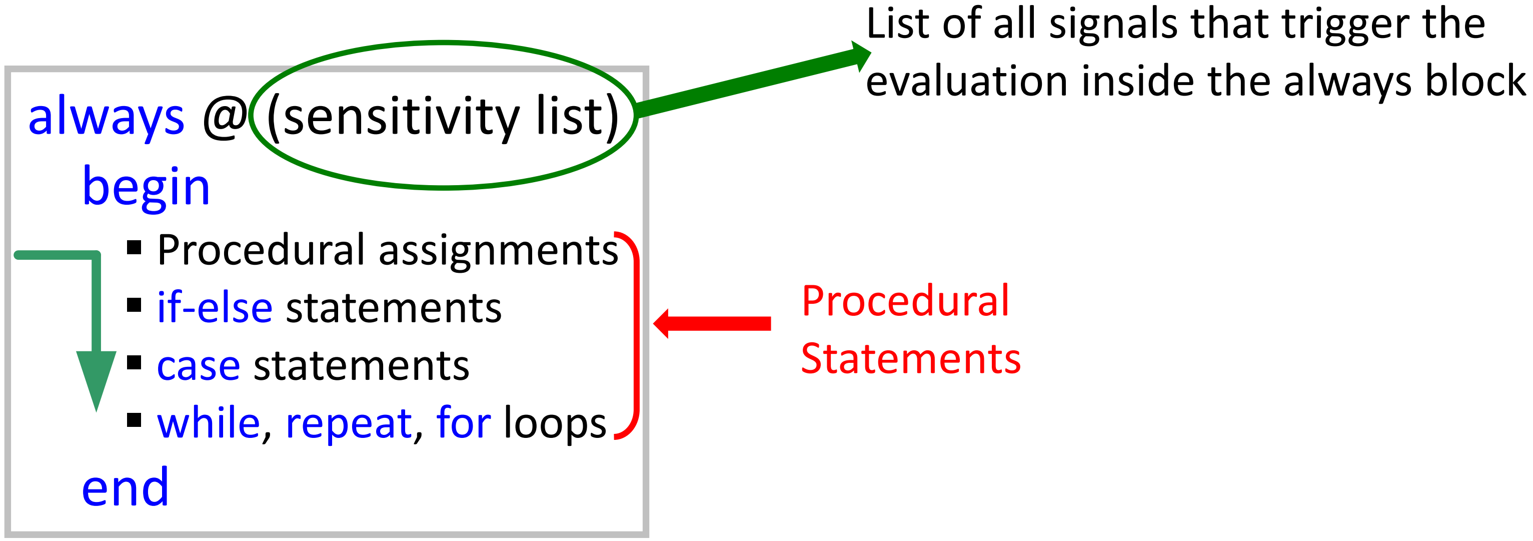
- Used only for simulation purposes
  - No meaning for synthesis
  - **Not** synthesizable

Procedural Statements

❑ Evaluated in the order in which they appear in the code (sequential)

❑ Should be inside an “**always**” block

❑ An “**always**” block contains one or more procedural statements



```
always @ (sensitivity list)
   begin
      ▪ Procedural assignments
      ▪ if-else statements
      ▪ case statements
      ▪ while, repeat, for loops
   end
```

List of all signals that trigger the evaluation inside the always block

Procedural Statements

Procedural Statements: Half-Adder

❖Example:

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

Type: “reg”

```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   reg S, C;
   always @ (x, y)
      begin
         S = x Λ y;
         C = x & y;
      end
endmodule
```

If either x or y changes, the statements inside the always block are evaluated.

```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   wire S, C;
   assign S = x Λ y;
   assign C = x & y;
endmodule
```

Type: “wire”

- ❑ Anything on the RHS should be on the sensitivity list
  - ➢ always @(*) ➡ Automatically considers all signals on the RHS in the sensitivity list
- ❑ Any signal assigned inside an always block has to be a variable of type
  - ➢ **reg**
  - ➢ **integer**

always block: Quick Notes

- The always construct requires begin-end only if there are multiple statements in the block.

❖ Example:

```
always @ (x, y, z)
      begin
            z = x;
            if (x == 1)
                  z = y;
      end
```

Correct

```
always @ (x, y, z)
      z = x;
      if (x == 1)
         z = y;
```

Not part of the always block

Incorrect

- A given variable should never be assigned a value in more than one always block.
  - Because always blocks are concurrent with respect to one another.

Incorrect

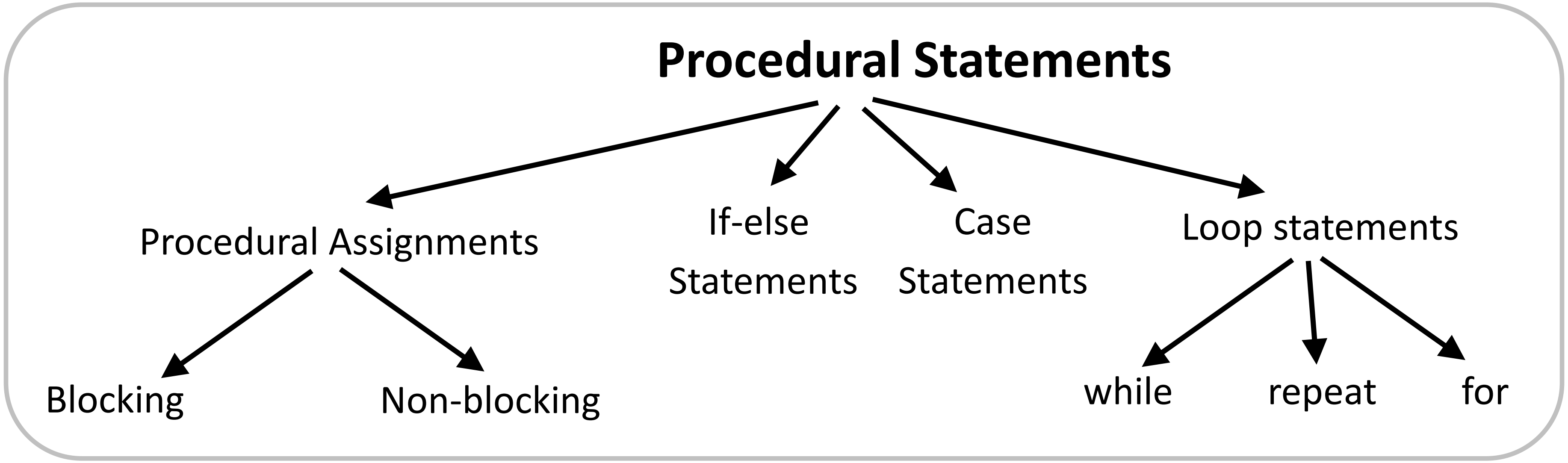
```
always @ (x,y)
      a <= x;
always @ (x,y)
      a <= y;
```

```
always @ (x,y)
begin
      a <= x;
      a <= y;
end
```

```
always @ (x,y)
      a <= y;
```

Correct

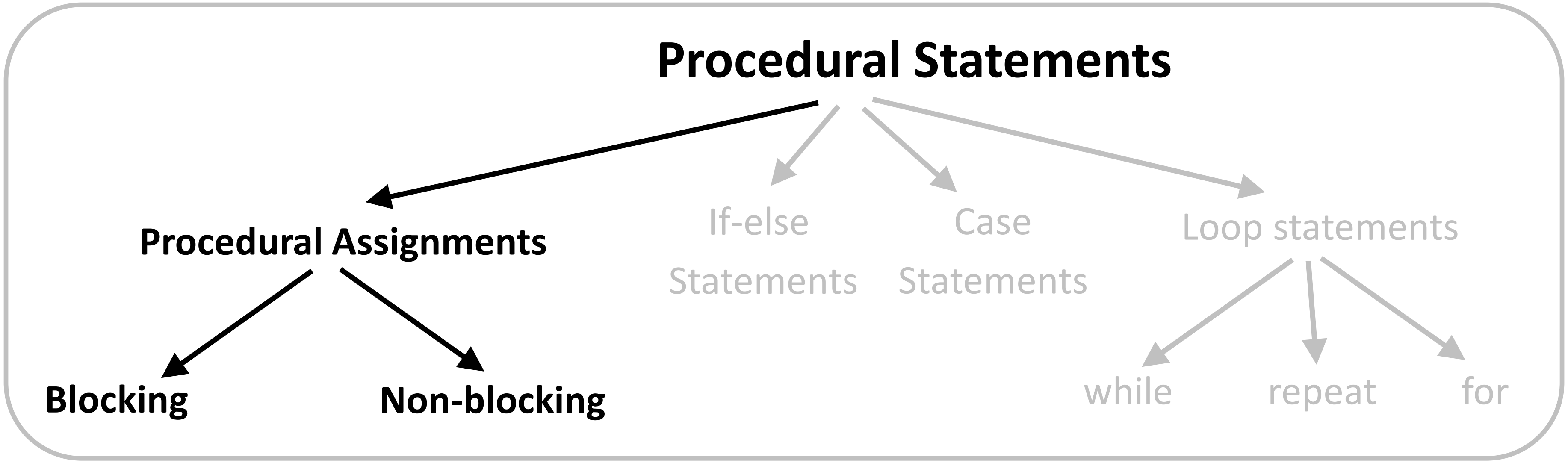
Procedural Statements



Procedural Statements

```
always @ (sensitivity list)
   begin
     ▪ Procedural assignments
     ▪ if-else statements
     ▪ case statements
     ▪ while, repeat, for loops
  end
```

Procedural Statements

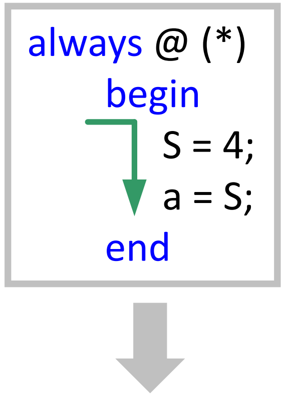


- ❑ Used inside an always block and are of two types:
  - ➢ **Blocking**: denoted by “=“ token
    - ▪ Evaluation within the always block is “blocked” until this assignment is completed
  - ➢ **Non-blocking**: denoted by “<=“ token
    - ▪ Nothing is hold or blocked (parallel evaluation)

Blocking vs. Non-Blocking Assignments

❖ Example: assume S=2 then

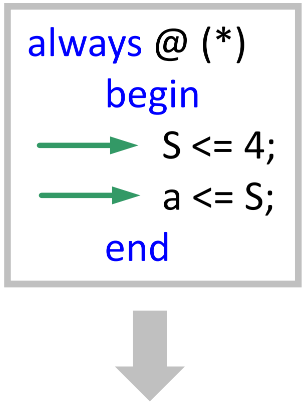
Blocking



```
always @ (*)
     begin
          S = 4;
          a = S;
     end
```

S=4 & a=4
(sequential)

Non-Blocking



```
always @ (*)
     begin
          S <= 4;
          a <= S;
     end
```

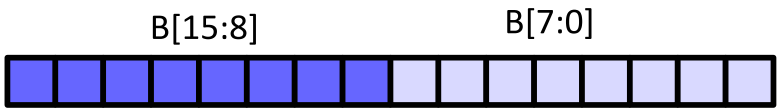
S=4 & a=2
(Parallel)

❑ Evaluated and assigned in a single step
❑ Sequential nature
❑ Assignment ordering **IS** important
❑ S=4 “blocks” a=S to be evaluated
   ➢ a=S has to wait for S=4 to be evaluated first

❑ Evaluated and assigned in two steps
   1. All RHSs are evaluated in parallel
   2. Assignments to LHSs are performed together
❑ They all are evaluated all at once
❑ Assignment ordering is **NOT** important
❑ S<=4 and a<=S evaluated in parallel

Blocking vs. Non-Blocking Assignments

❖ Example: Swap bytes in words

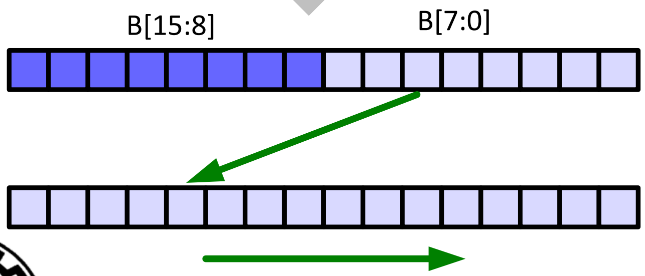


❑ Which one is correct?

Blocking

Incorrect

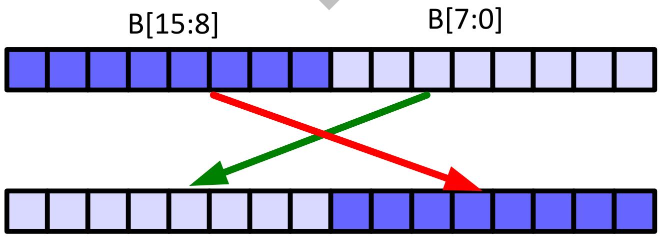
```
always @ (*)
      begin
           B[15:8] = B[7:0];
           B[7:0]   = B[15:8] ;
      end
```



Non-Blocking

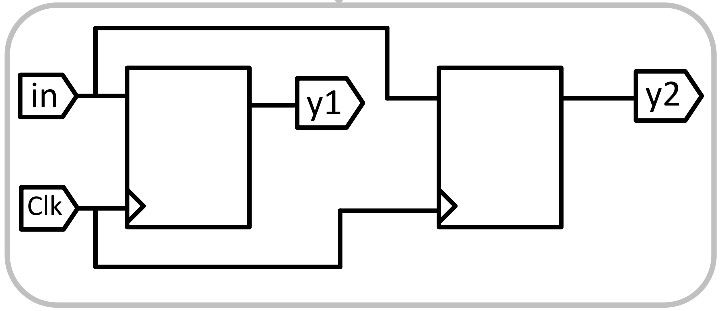
```
always @ (*)
      begin
           B[15:8] <= B[7:0];
            B[7:0]   <= B[15:8] ;
       end
```

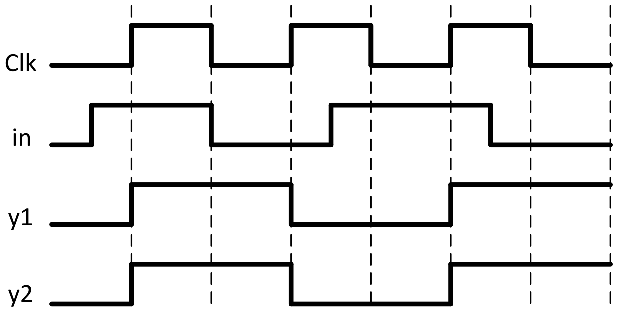
Correct



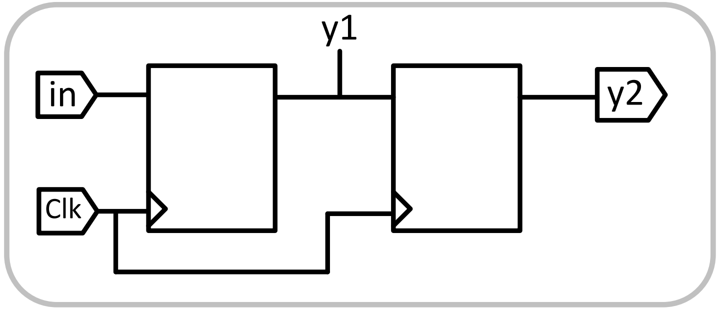
Blocking vs. Non-Blocking after Synthesis:

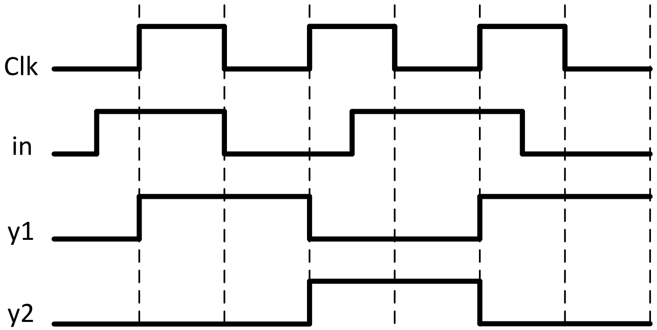
```
always @ (posedge Clk)
      begin
           y1 = in;
           y2 = y1 ;
      end
```



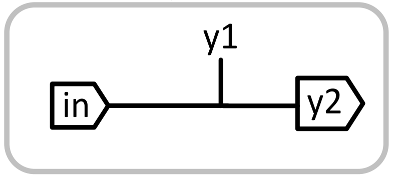


```
always @ (posedge Clk)
      begin
           y1 <= in;
           y2 <= y1 ;
      end
```



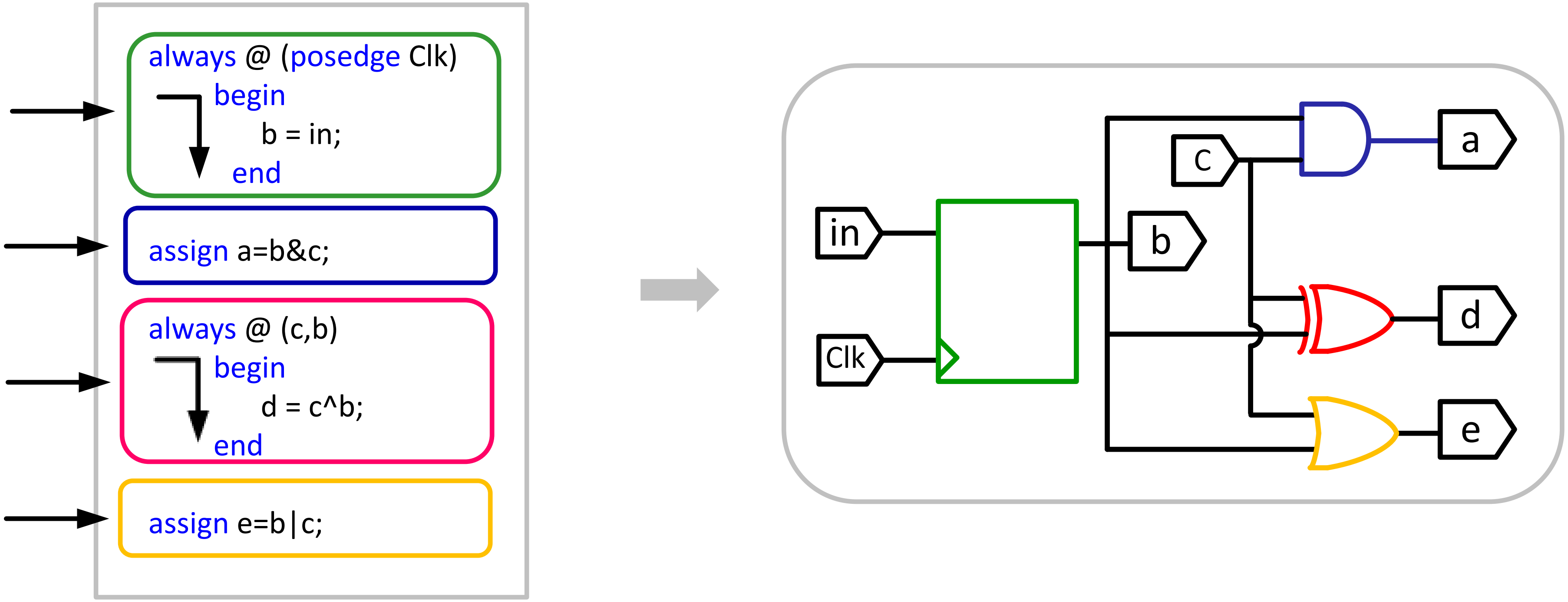


```
always @ (*)
      begin
           y1 = in;
           y2 = y1 ;
      end
```



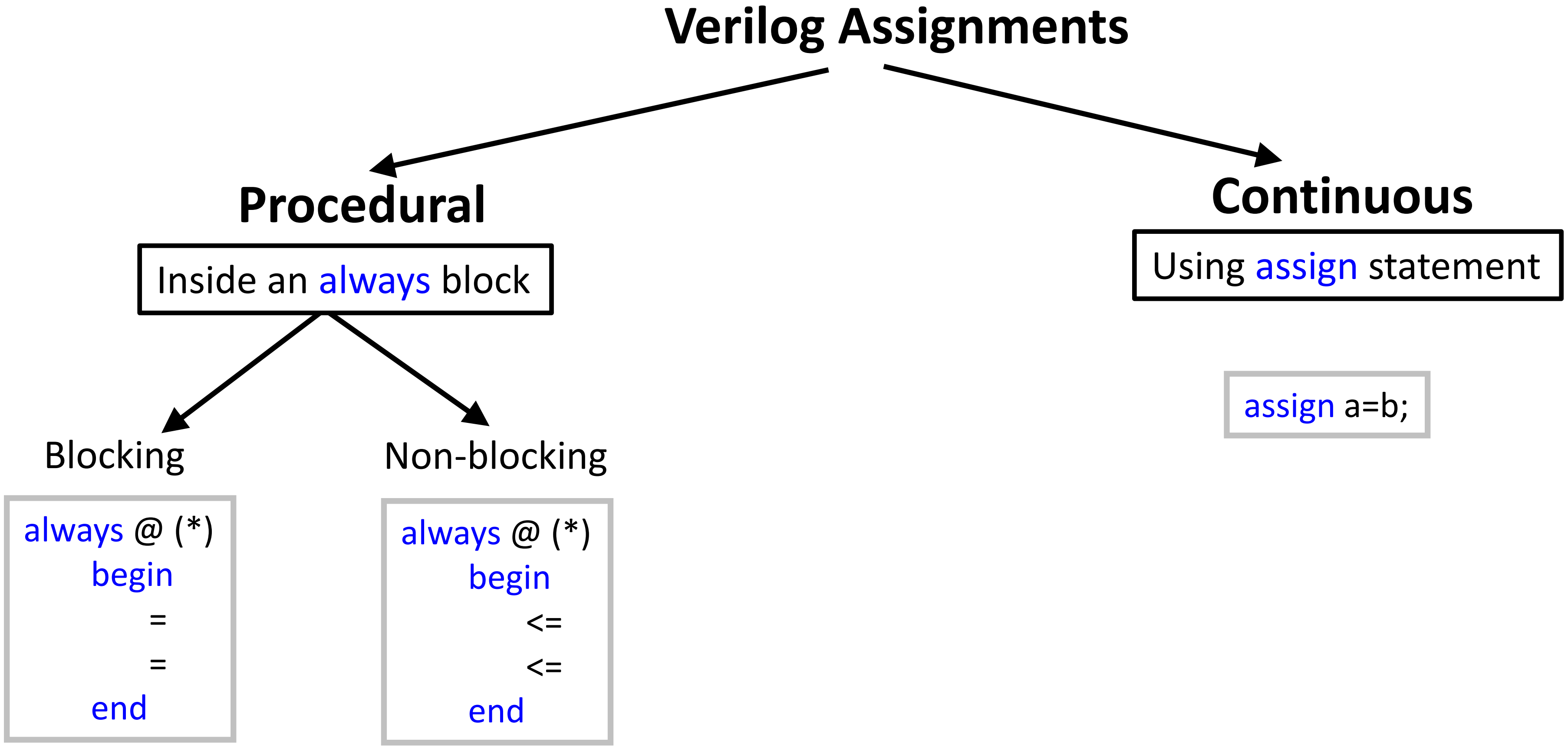
Overall Code Parallelism

- ❑ Statements inside an always block are evaluated sequentially
- ❑ However, all always blocks are evaluated concurrently
- ❑ All continuous assignments are evaluated concurrently too



```
always @ (posedge Clk)
    begin
        b = in;
    end

assign a=b&c;

always @ (c,b)
    begin
        d = c^b;
    end

assign e=b|c;
```

Verilog Assignments in a Glance



Verilog Assignments

Procedural

Inside an always block

Blocking

```
always @ (*)
    begin
        =
        =
    end
```

Non-blocking

```
always @ (*)
    begin
        <=
        <=
    end
```

Continuous

Using assign statement

```
assign a=b;
```

- ❑ assign can not be used inside an always block b/c assign is used for nets.
  - ➢ Nets can not be assigned inside an always blocks (only reg or integer).

Outline

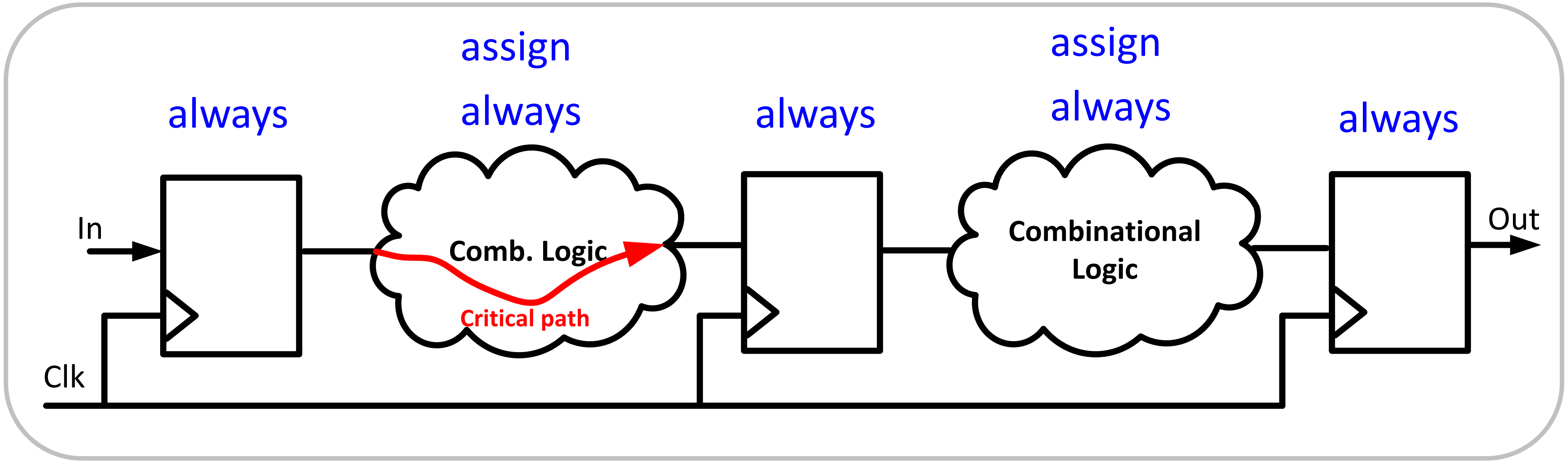
- ❑ ASIC/FPGA Design Flow
- ❑ Hardware Description Language (HDL)
  - ➢ Verilog
    - o Introduction
    - o Language Fundamentals
    - o Modeling Combinational & Sequential Logic Circuits
    - o Modeling Finite State Machines
    - o Verilog Operations

Logic Circuits Category

- ❑ **Logic Circuits:**
  - ➢ **Combinational logic: (realized by assign and always)**
    - Output depends on inputs
    - Inputs propagates to the output through some gates with delay
    - e.g., adders, Mux, multiplier, all logic gates
  - ➢ **Sequential Logic: (realized only by always)**
    - Output depends on inputs and circuit history
    - Circuit history is kept using flip-flops, registers or latches
    - e.g., Finite State Machines (FSM), shift registers, Flip Flops (FF)
- ❑ Sequential logic has two flavors:
  - ➢ **Synchronous:** all registers controlled by a global clock
  - ➢ **Asynchronous:** based on the handshaking process

Logic Circuits Category

❑ A general system consists of both combinational and sequential circuits



❑ Critical path of the Comb. Logic determines the max operating frequency

❑ Combinational logic can be realized using assign and always constructs

❑ Sequential logic can only be realized using always blocks.

Combinational Logic

- ❑ Combinational logic can be realized using assign and always constructs

❖Example: Full Adder:

```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   reg S, C;
   always @ (x, y)
      begin
           S = x Λ y;
           C = x & y;
      end
endmodule
```

⟷

```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   wire S, C;
   assign S = x Λ y;
   assign C = x & y;
endmodule
```

- ❑ When using always block for Com. Logic, “blocking” assignments are used
- ❑ When using an always block, time instant changes when one of the sensitivity list variables changes

Blocking Assignment for Combinational Logic

❑ Use only blocking assignments for combinational logic. Why?

❖Example: Accumulator: (Assume Count == 0)

```
always @ (*)
      begin
       for (k=0; k<4; k=k+1)
              Count = Count + k;
      end
```

```
Count = Count + 0;
Count = 0 + 1;
Count = 0 + 1 + 2;
Count = 0 + 1 + 2 + 3;
Result: Count = 6
```

Correct

```
always @ (*)
      begin
       for (k=0; k<4; k=k+1)
              Count <= Count + k;
      end
```

```
Count <= Count + 0;
Count <= Count + 1;
Count <= Count + 2;
Count <= Count + 3;
Result: Count =3
```

Incorrect

In multiple concurrent non-blocking assignments to a variable, the last one executes

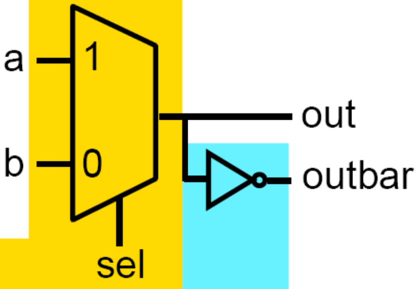
Combinational Logic

- Procedural and continuous assignments can (and often do) co-exist within a module
- Procedural assignments update the value of `reg`. The value will remain unchanged till another procedural assignment updates the variable. This is the main difference with continuous assignments in which the right hand expression is constantly placed on the left-side

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```

procedural description

continuous description



always block vs. assign for Combinational Logic

❑ When do we use the always block to describe a combinational logic?

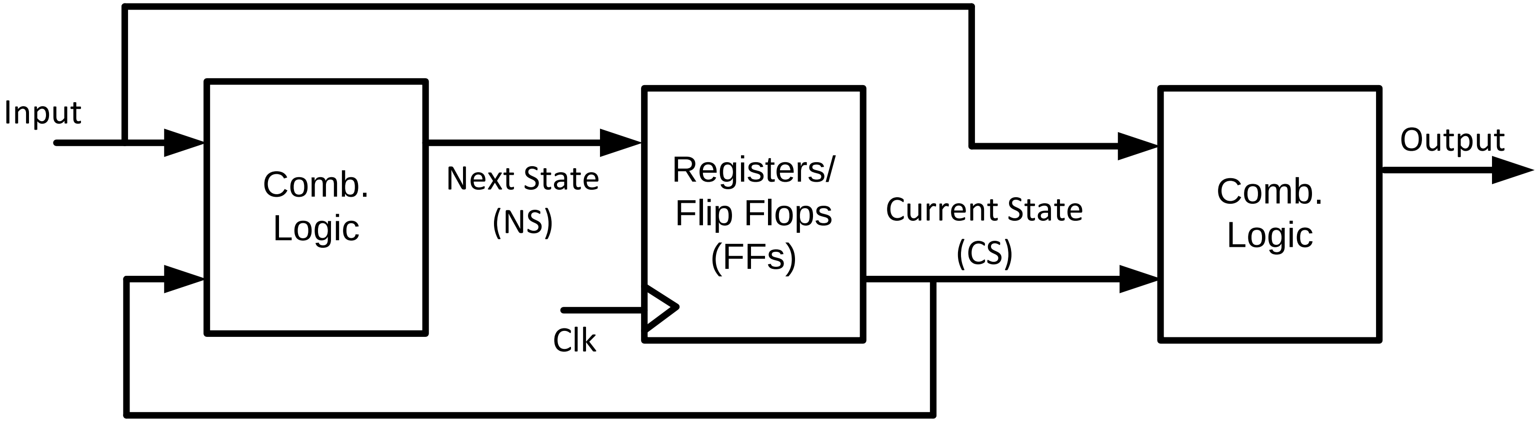
1. Normally for high-complexity Comb. Logic
2. When output depends on several conditions, which requires if-else or case constructs to be fully described

❑ Why?

1. Because powerful statements like if-else and loop constructs can only be used inside an always block
   - ➢ Comes with more clarity and more concise description than assign
2. Multiple outputs can be assigned within a single always block

Sequential Logic

- ❑ Sequential circuits have memory (i.e., remembers the past)
- ❑ The current state is held in memory and the next state is computed through the combinational logic
- ❑ In a synchronous system, a global clock signal orchestrates the flow of the data and the sequence of events



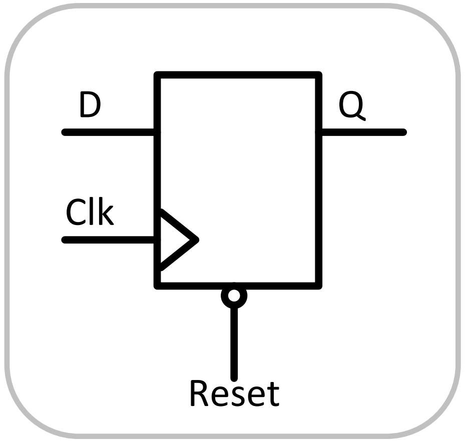
Sequential Logic

- ❑ Sequential logic can only be realized using an always block
- ❑ Consists of :
  - ➢ **Flip flops** that are normally controlled by:
    - ▪ Positive edge of the clock (posedge) ➡ always @ (posedge Clk)
    - ▪ Negative edge of the clock (negedge) ➡ always @ (negedge Clk)
    - ▪ Have posedge or negedge in the sensitivity list
    - ▪ Any variable assigned a value is the output of a flip-flop
  - ➢ **Latches**
    - ▪ Transfers input to output when clock is “1” and stores the value O.W.
  - ➢ **Finite State Machine (FSM)**
- ❑ When using the always block for the sequential Logic, “**Non-blocking**” assignments are used

Sequential Logic: Flip-Flop

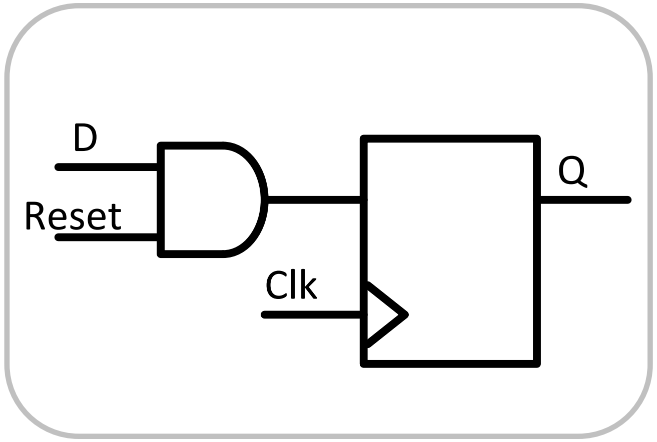
- Example: Flip-flop with asynchronous Reset:

```
always @ (posedge Clk, negedge Reset)
    if (Reset == 0)
            Q<=0;
    else
            Q<=D;
```



- Example: Flip-flop with synchronous Reset:

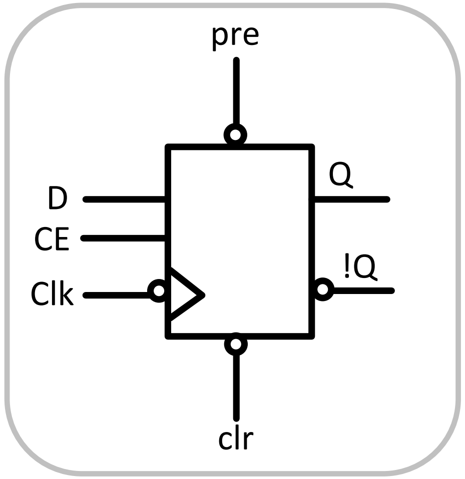
```
always @ (posedge Clk)
    if (Reset == 0)
            Q<=0;
    else
            Q<=D;
```



Sequential Logic: Flip-Flop

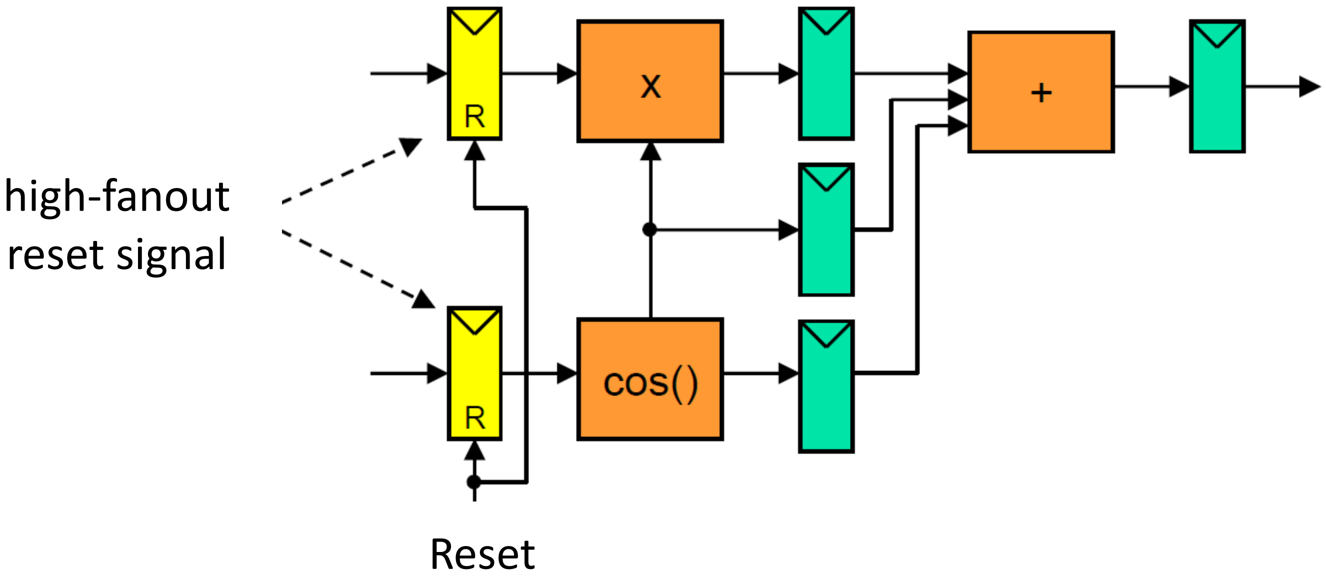
❖Example: Flip-flop with complete features:

```
module flip_flop_n ( output reg Q , output Q_n ,input pre_n, clr_n,
D, input clk_n, CE );

always@ (negedge clk_n or negedge pre_n or negedge clr_n)
        begin
                if (!pre_n) Q <= 1'b1;
                elseif (!clr_n) Q <= 1'b0;
                elseif (CE) Q <= D;
        end
assign Q_n = ~Q;
endmodule
```



Sequential Logic: Flip-Flop

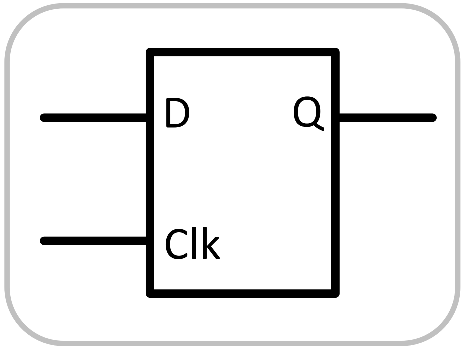
- ❑ Use reset-able FFs only where needed
  - ➢ FFs are a little larger and higher power
  - ➢ Requires the global routing of the high-fanout reset signal



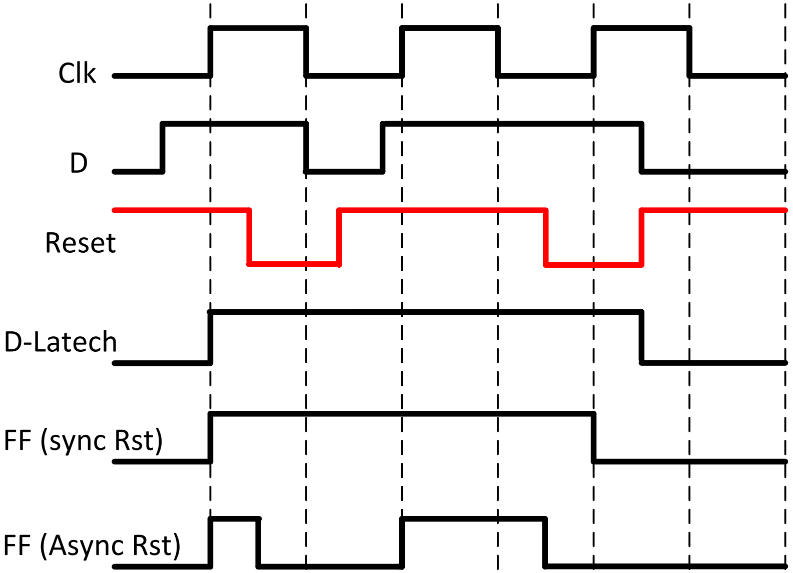
Sequential Logic

❖Example: D-Latch:

```
module Latch(D, Clk, Q);
  input D, Clk;
  output reg Q;
  always @ (D, Clk)
      if (Clk)
                Q<=D;
endmodule
```



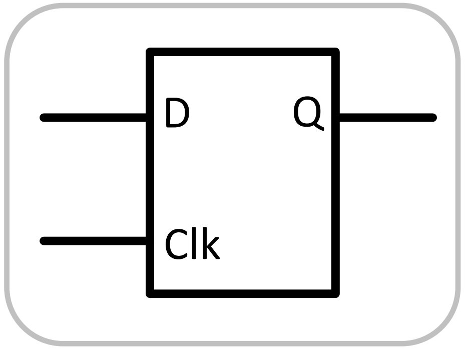
❖Example:



Sequential Logic

❖Example: D-Latch:

```
module Latch(D, Clk, Q);
  input D, Clk;
  output reg Q;
  always @ (D, Clk)
     if (Clk)
              Q<=D;
endmodule
```



Both results in a latch

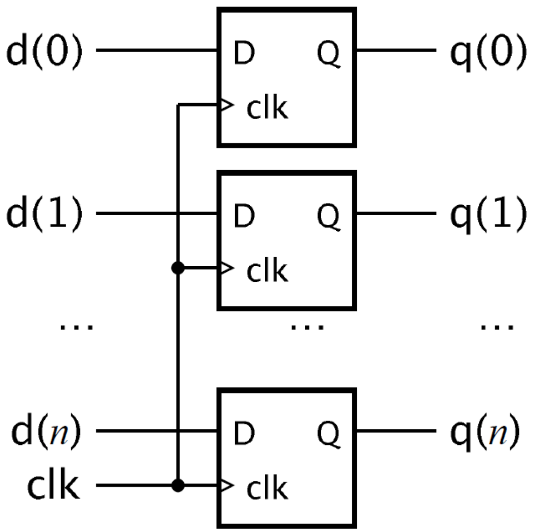
```
module Latch(D, Clk, Q);
  input D, Clk;
  output reg Q;
  always @ (Clk)
     if (Clk)
              Q<=D;
endmodule
```

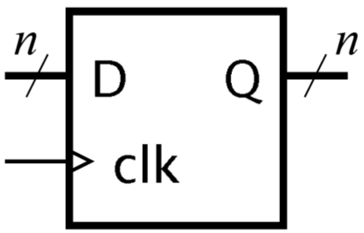
This results in a warning saying D is not in the sensitivity list

Sequential Logic: Registers

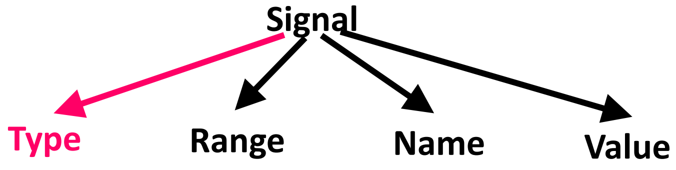
- ❑ Store a multi-bit encoded value
  - ➢ One D-FF per bit
  - ➢ Stores a new value on each clock cycle

```
wire [n:0] d;
reg [n:0] q;
...
always @ (posedge Clk)
               q<=d;
```





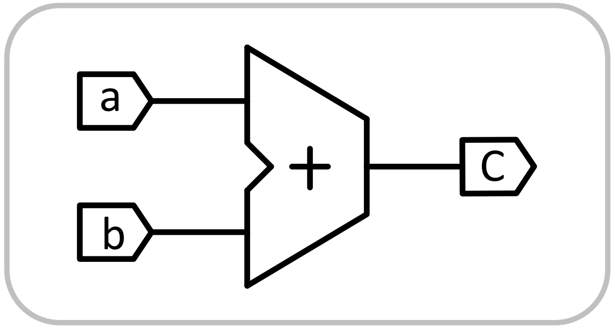
Reg Type (Revisited)



- ❑ The keyword “reg” does NOT necessarily denote a storage element or register.
- ❑ “reg” simply means a <u>variable that can hold a value</u>
- ❑ May or may not be synthesized as a register.

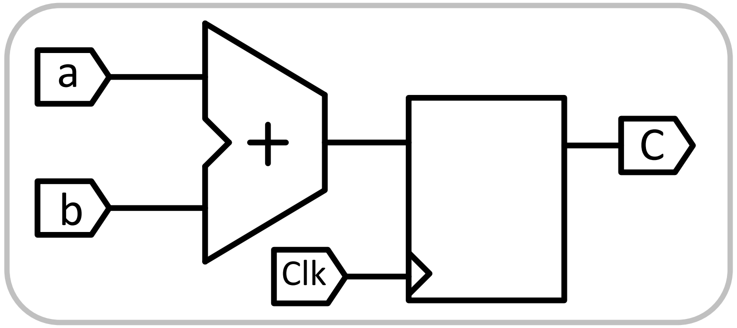
Not Register

```
reg  C;
   always @ (a,b)
        C = a+b;
```



Register

```
reg  C;
   always @ (posedge Clk)
        C <= a+b;
```



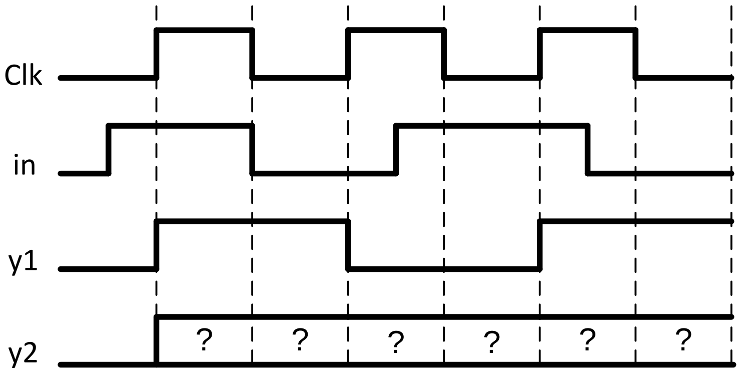
Sequential Logic

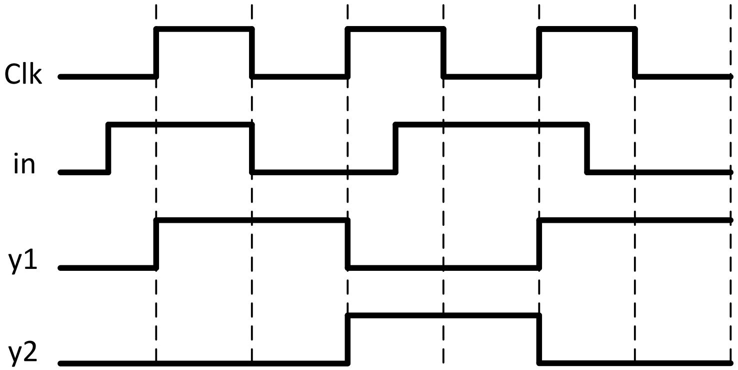
- ❑ When using always block for sequential Logic, "Non-blocking" assignments are used. Why?

```
always @ (posedge Clk)
        y1=in;

always @ (posedge Clk)
        y2=y1;
```

Race Condition

```
always @ (posedge Clk)
        y1<=in;

always @ (posedge Clk)
        y2<=y1;
```





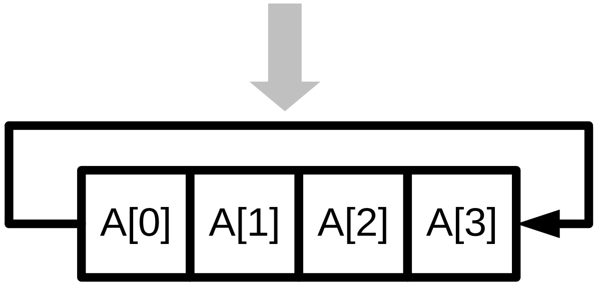
Sequential Logic

- When using always blocks for sequential Logic, “Non-blocking” assignments are used. Why?
- Example: Shift register

```
always @ (A)
     begin
        for (k=0; k<4;k=k+1)
            A[k]=A[k+1];
        A[3] = A[0];
     end
```

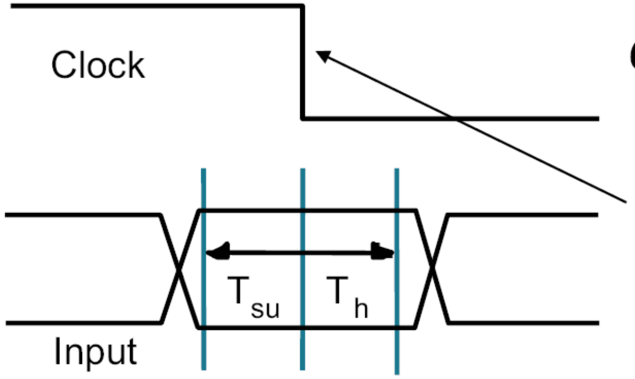
Incorrect!

```
always @ (A)
     begin
        for (k=0; k<4;k=k+1)
            A[k]<=A[k+1];
        A[3] <= A[0];
     end
```



- Do **NOT** use blocking assignments for sequential logic

Important Timing Parameters



Clock:
Periodic Event, causes state of memory element to change

memory element can be updated on the: rising *edge*, falling *edge*, high *level*, low *level*

There is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized

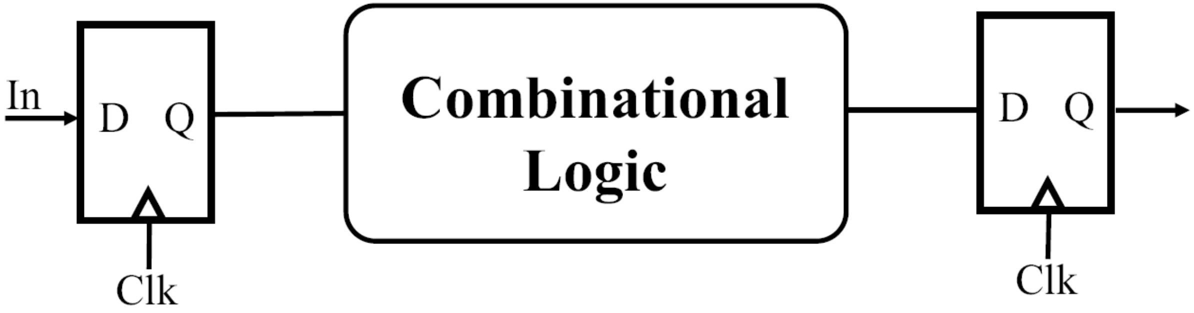
Setup Time (T_{su})
Minimum time before the clocking event by which the input must be stable

Hold Time (T_h)
Minimum time after the clocking event during which the input must remain stable

Propagation Delay (T_{cq} for an edge-triggered register and T_{dq} for a latch)

Delay overhead of the memory element

System Timing Parameters



Register Timing Parameters

T_{cq} : worst case rising edge clock to q delay

$T_{cq,\ cd}$: contamination or minimum delay from clock to q

T_{su}: setup time

T_h: hold time

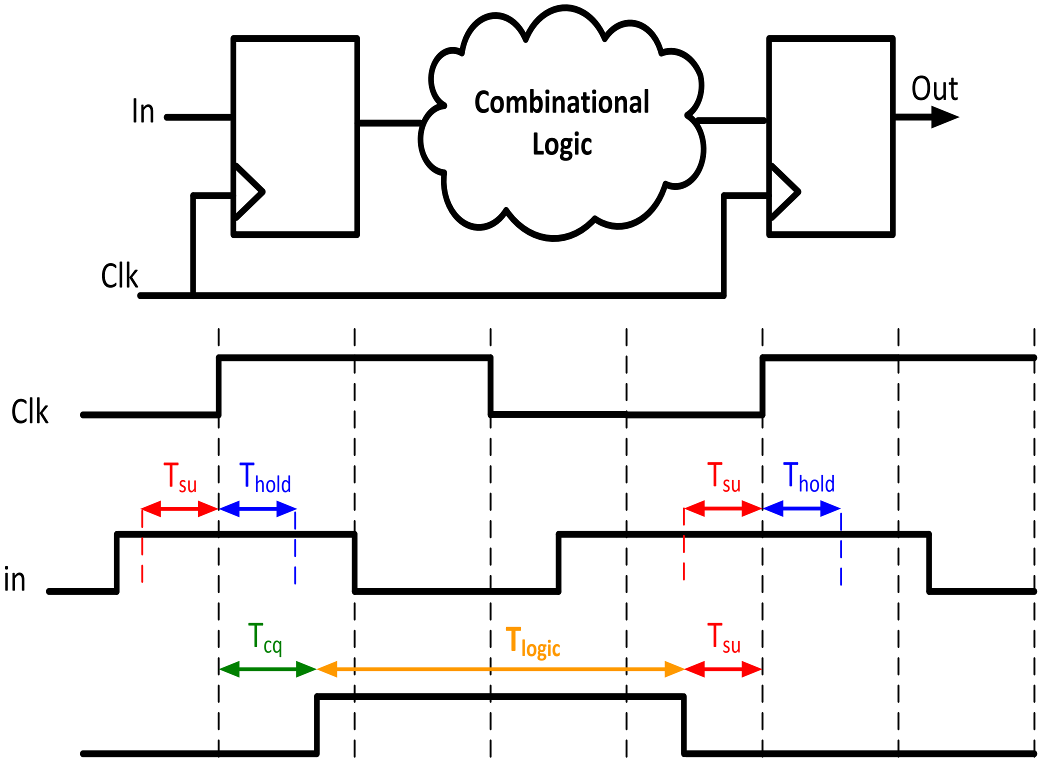
Logic Timing Parameters

T_{logic} : worst case delay through the combinational logic network

$T_{logic,cd}$: contamination or minimum delay through logic network

System Timing Parameters : Minimum Period

- **Setup-time Condition:**
  - If violates circuit works at lower frequency (why?)

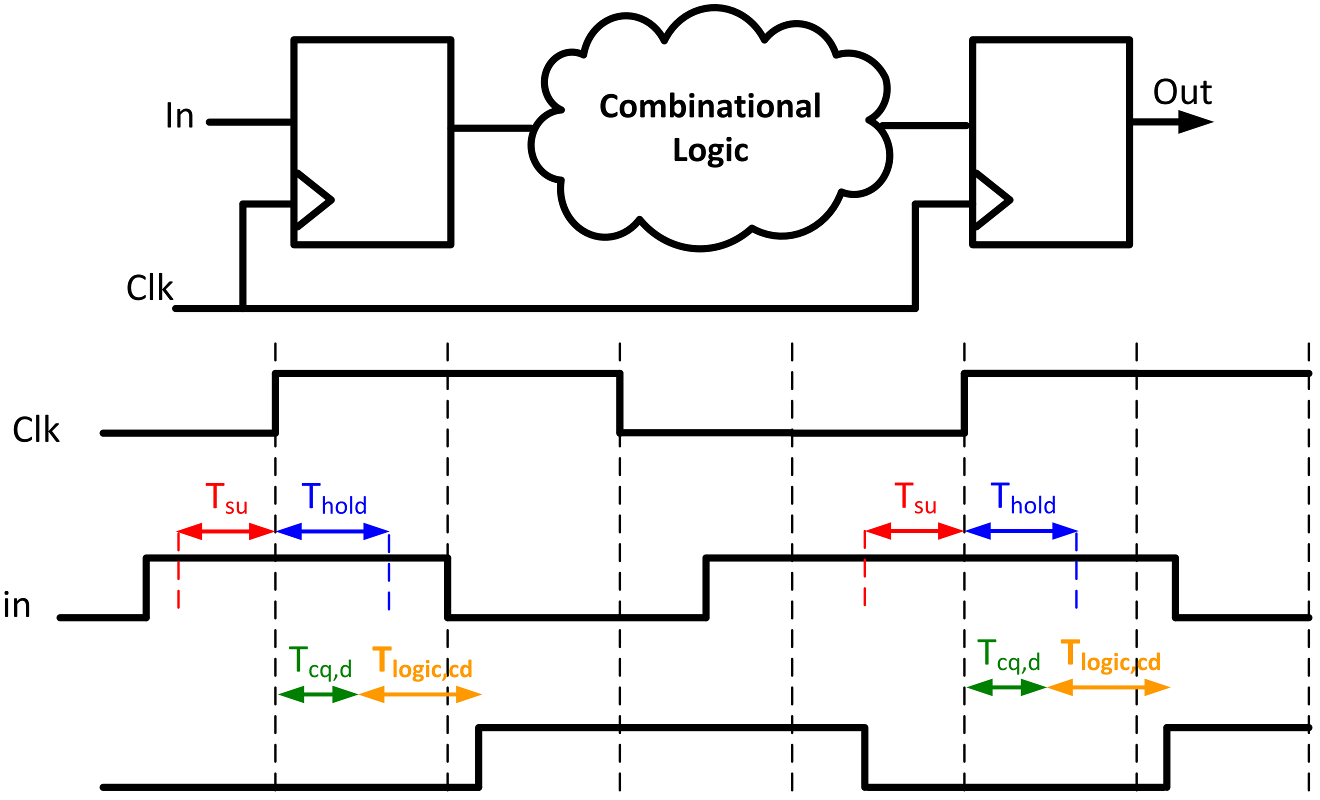


$$T_{Clk} > T_{cq} + T_{logic} + T_{su}$$

$$T_{logic} < T_{Clk} - T_{su} - T_{cq}$$

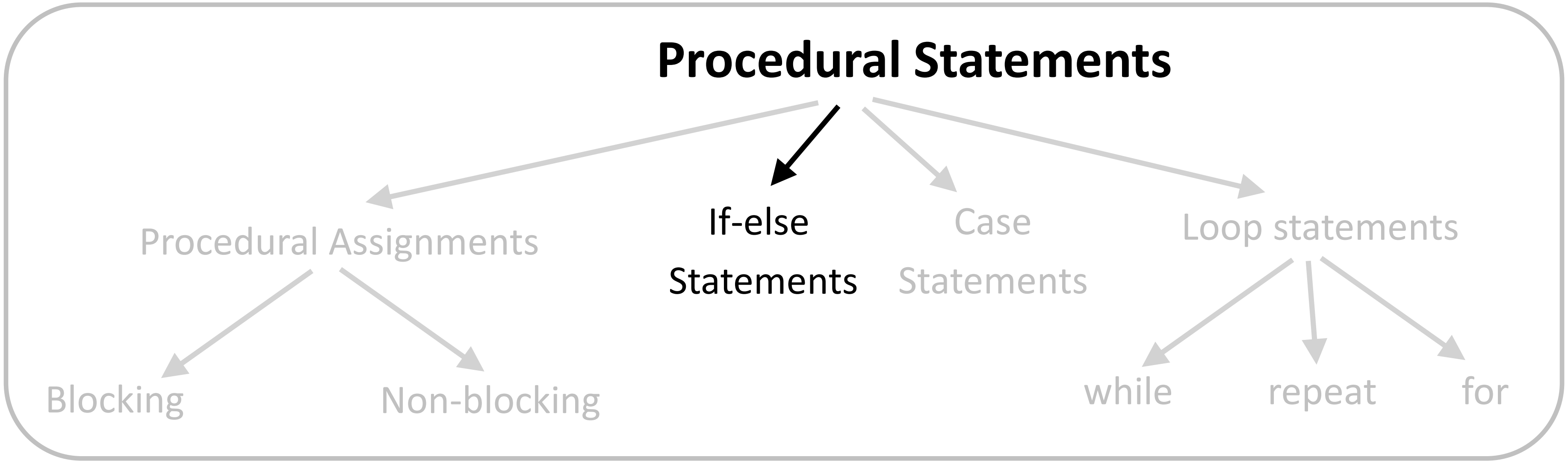
System Timing Parameters : Minimum Delay

- **Hold-time Condition:**
  - If violates circuit does not work (even at lower frequencies) (why?)



$$T_{cq,cd}+T_{logic,cd}>T_{hold}$$

Procedural Statements



Procedural Statements

```
always @ (sensitivity list)
   begin
     ▪ Procedural assignments
     ▪ if-else statements
     ▪ case statements
     ▪ while, repeat, for loops
   end
```

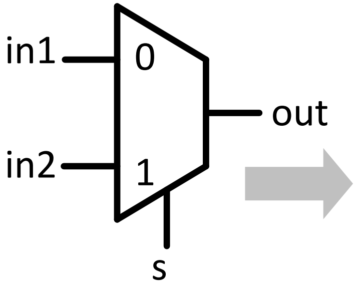
If-else statements

- ❑ Used only inside an always block
- ❑ Format:

```
If (expression1)
   statement1;
else if (expression2)
   statement2;
else
   statement3;
```

Single statement no need for begin-end
Multiple statements, begin-end is needed

❖ Example:



```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;

   always @ (in1, in2, s)
      if (s==0)
         out = in1;
      else
         out = in2;
endmodule
```

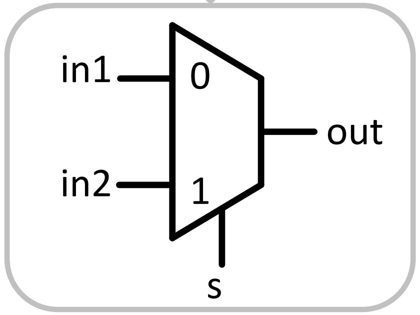
```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;
   always @ (in1, in2, s)
      begin
         out = in1;
         if (s==1)
            out = in2;
      end
endmodule
```

If-else statements

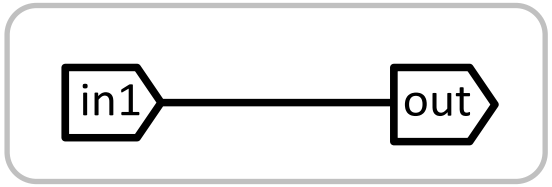
- ❑ If-else construct inside an always block have a sequential nature when used by blocking assignments. Sequential means direct effect on synthesis not necessarily sequential in actual hardware implementation
- ❑ This means ordering is important

❖ Example:

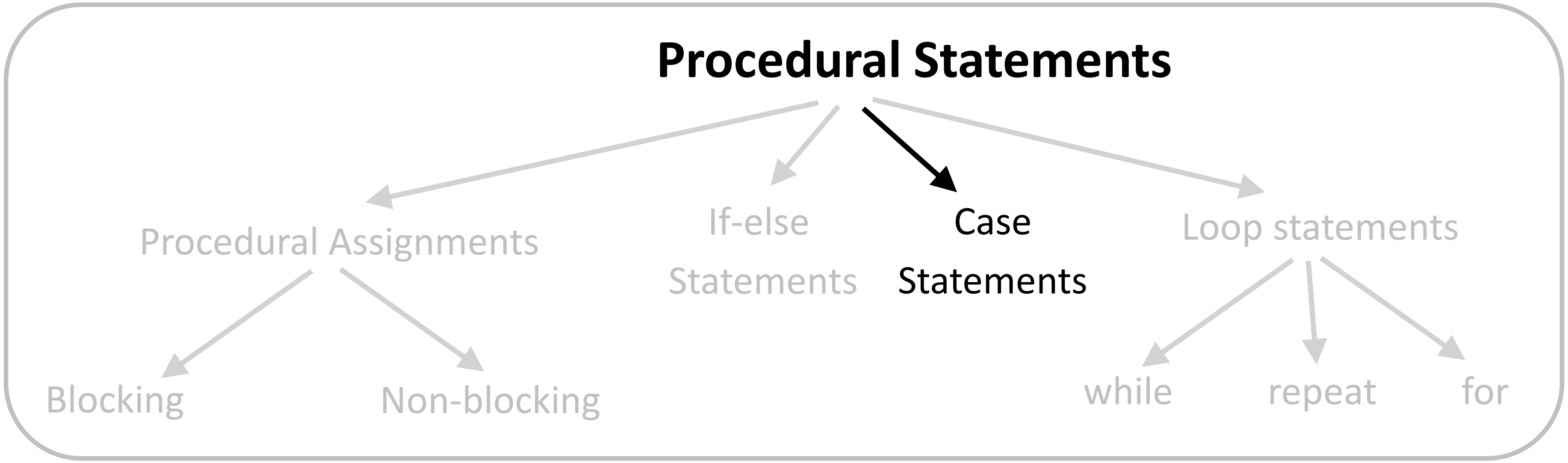
```
always @ (*)
      begin
         out = in1;
         if (s==1)
            out = in2;
      end
```



```
always @ (*)
      begin
         if (s==1)
            out = in2;
         out = in1;
      end
```



Procedural Statements



Procedural Statements

Procedural Assignments

Blocking

Non-blocking

If-else Statements

Case Statements

Loop statements

while

repeat

for

Procedural Statements

```
always @ (sensitivity list)
   begin
     ▪ Procedural assignments
     ▪ if-else statements
     ▪ case statements
     ▪ while, repeat, for loops
   end
```

Case statements

- ❑ Used only inside an always block
- ❑ Format:

```
case (expression)
   alternative1: statement1;
   alternative2: begin
                    statement2;
                 end
   default: statementn;
endcase
```

Single statement no need for begin-end

Multiple statements, begin-end is needed

❖ Example:

```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;

   always @ (in1, in2, s)
       case (s)
           1'b0: out = in1;
           1'b1: out = in2;
       endcase
endmodule
```

⟷

```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;

   always @ (in1, in2, s)
       case (s)
           1'b0: out = in1;
           default: out = in2;
       endcase
endmodule
```

Case statements

- Example: Combinational logic using both assign and always block

```
module FullAdder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x & y)|(x & Cin)|(y & Cin);

endmodule
```

```
module FullAdder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output reg  S, Cout;
   always @ (Cin, x, y)
        begin
            case ({Cin, x, y})
                3’b000: {Cout, S} = ‘b00;
                3’b001: {Cout, S} = ‘b01;
                3’b010: {Cout, S} = ‘b01;
                3’b011: {Cout, S} = ‘b10;
                3’b100: {Cout, S} = ‘b01;
                3’b101: {Cout, S} = ‘b10;
                3’b110: {Cout, S} = ‘b10;
                3’b111: {Cout, S} = ‘b11;
            endcase
        end
endmodule
```

concatenation

x	y	Cin	Cout	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	0	1

Case statements

- ❑ In case statements, each alternative is compared for an exact match
- ❑ Synthesis tools are only concerned about matching of “0” and “1” while “Z” and “X” are not important
- ❑ If “X” or “Z” are needed to be added, casex is used (casex is synthesizable).
- ❑ In fact casex treats them as don’t care.

❖ Example: 4-to-2 priority encoder

w3	w2	w1	w0	y1	y0	f
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

```
module Priority (W, Y, f)
   input [3:0] W;
   output reg [1:0] Y;
   output f;
   assign f = (W!=0)
   always @ (W)
     begin
       casex (W)
           ’b1xxx:  Y = 3;
           ‘b01xx:  Y = 2;
           ‘b001x:  Y = 1;
           default: Y = 0;
        endcase
     end
endmodule
```

Case statements

- ❑ **casez** allows use of wildcard “?” character for don’t

w3	w2	w1	w0	y1	y0	f
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

```
module Priority (W, Y, f)
   input [3:0] W;
   output reg [1:0] Y;
   output f;
   assign f = (W!=0)
   always @ (W)
      begin
         casez (W)
              ’b1???:  Y = 3;
              ‘b01??:  Y = 2;
              ‘b001?:  Y = 1;
              default: Y = 0;
         endcase
      end
endmodule
```

Latch Inference in Combinational Logic

❑ When realizing **combinational logic** with always block using if-else or case constructs care has to be taken to avoid latch inference after synthesis

❑ The latch is inferred when "incomplete" if-else or case statements are declared

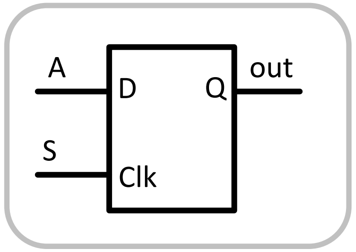
❑ This latch is "unwanted" as the logic is combinational not sequential

❑ If there is some logic path through the always block that does not assign a value to the output, a latch is inferred

Latch Inference in If-else or Case Statements

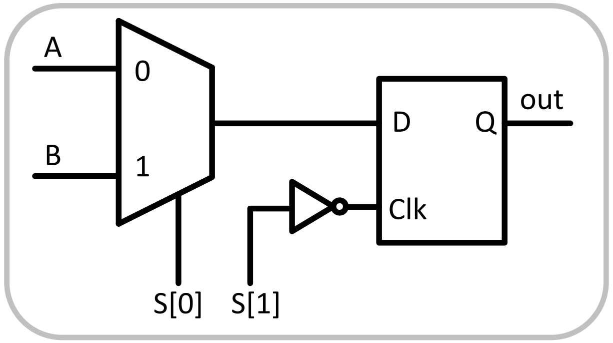
❖Example:

```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
begin
   if (S==1)
       out = A;
end
endmodule
```



Latch Inference

```
module DUT (A, B, S, out);
input A, B;
Input [1:0] S;
output reg out;
always @(A, B, S)
begin
   case (S)
       2’b00: out = A;
       2’b01: out = B;
   endcase
end
endmodule
```



Latch Inference

Latch Inference in Combinational Logic

- To avoid latch inference make sure to specify all possible cases “explicitly”

- Two practical approaches to avoid latch inference:

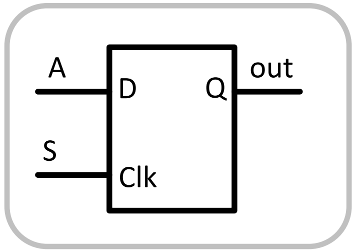
  - **For if-else construct:**
    1. Initialize the variable before the if-else construct
    2. Use else to explicitly list all possible cases
  - **For case constructs:**
    1. Use default to make sure no case is missed!

- Do NOT let it up to the synthesis tool to act in unspecified cases and do specify all cases explicitly.

Avoid Latch Inference in If-else Statements

❖Example:

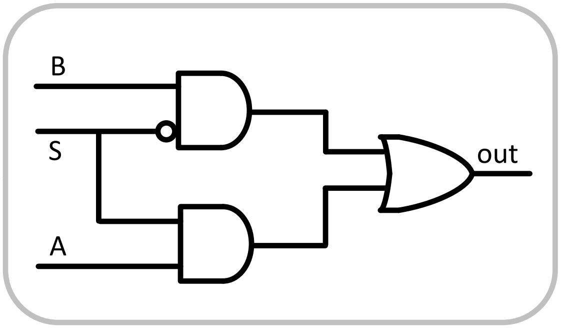
```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
begin
   if (S==1)
       out = A;
end
endmodule
```



Latch Inference

1

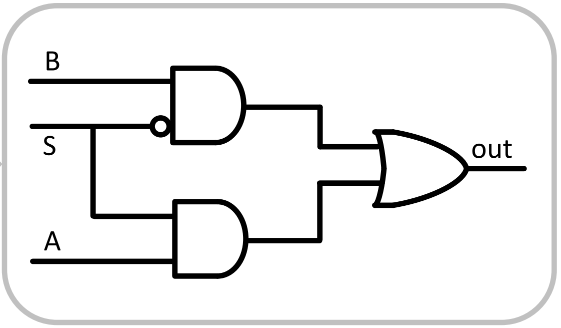
```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
Begin
   out = B;
   if (S==1)
       out = A;
end
endmodule
```



No Latch

2

```
module DUT (A, B, S, out);
input A, B, S;
output reg out;
always @(*)
begin
   if (S==1)
       out = A;
   else
       out =B;
end
endmodule
```



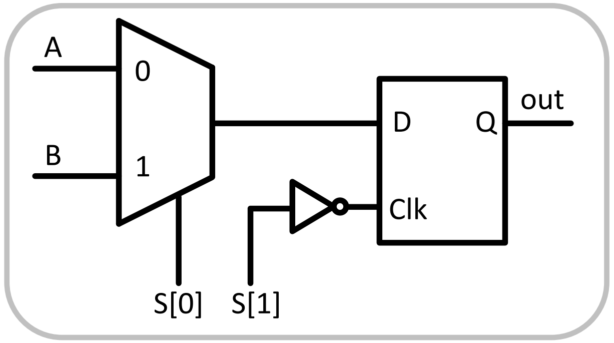
No Latch

Avoid Latch Inference in Case Statements

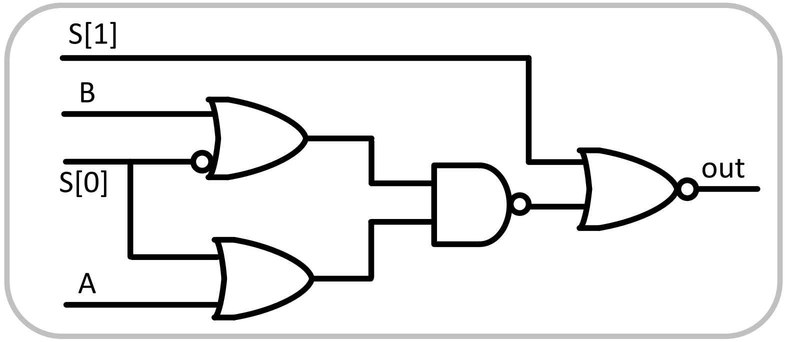
❖Example:

```
module DUT (A, B, S, out);
input A, B;
Input [1:0] S;
output reg out;

always @(A, B, S)
begin
   case (S)
      2'b00: out = A;
      2'b01: out = B;
   endcase
end
endmodule
```

```
module DUT (A, B, S, out);
input A, B;
Input [1:0] S;
output reg out;
always @(A, B, S)
begin
   case (S)
      2'b00: out = A;
      2'b01: out = B;
      default: out = 1'b0;
   endcase
end
endmodule
```

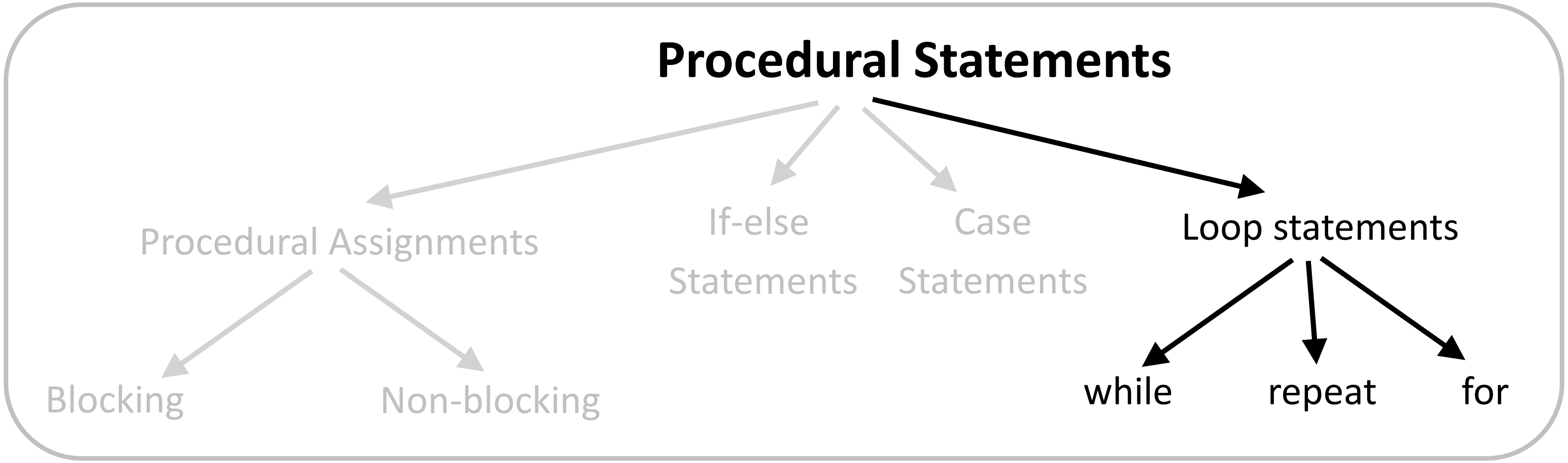


Latch Inference



No Latch

Procedural Statements



Procedural Statements

Procedural Assignments

Blocking

Non-blocking

If-else Statements

Case Statements

Loop statements

while

repeat

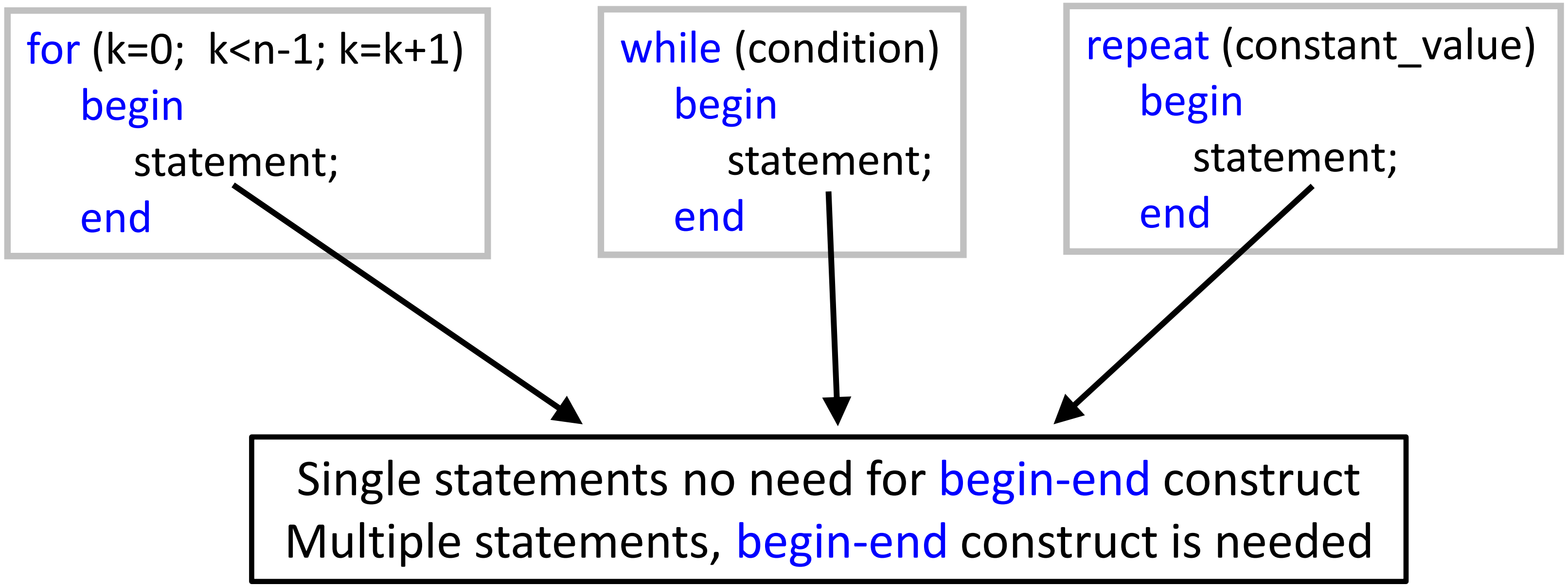
for

Procedural Statements

```
always @ (sensitivity list)
   begin
     ▪ Procedural assignments
     ▪ if-else statements
     ▪ case statements
     ▪ while, repeat, for loops
  end
```

Loop Statements

- ❑ To be used for efficient coding style
- ❑ **All being used inside an always block**
- ❑ Make sure to use blocking statements for combinational logic



```
for (k=0;  k<n-1; k=k+1)
    begin
        statement;
    end
```

```
while (condition)
    begin
        statement;
    end
```

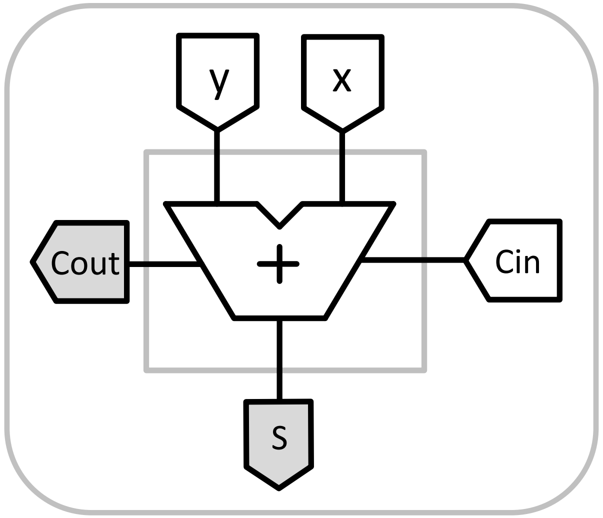
```
repeat (constant_value)
    begin
        statement;
    end
```

Single statements no need for begin-end construct
Multiple statements, begin-end construct is needed

Loop Statements

❖Example: Remember: 1-bit full Adder

```
module Full_Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x & y)|(x & Cin)|(y & Cin);

endmodule
```

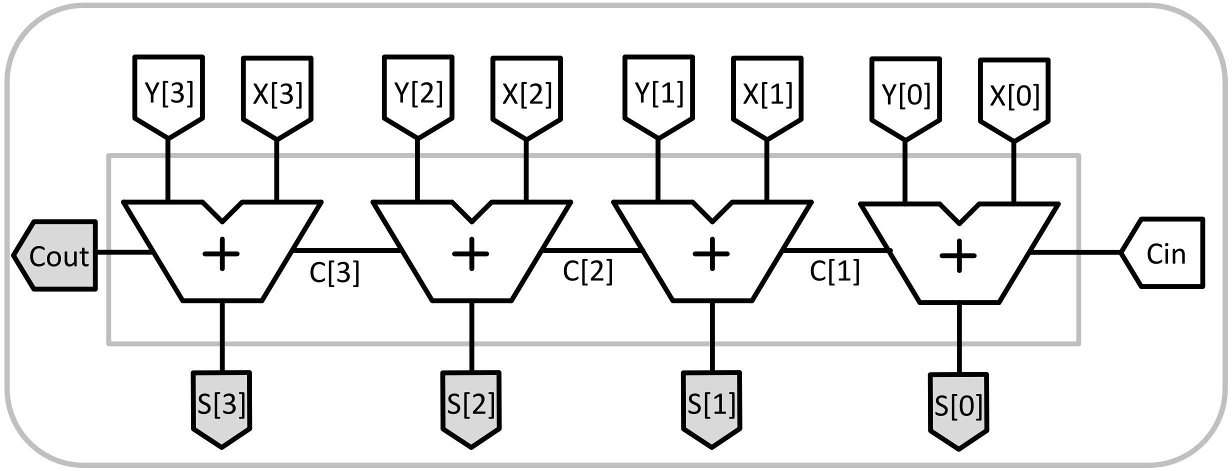


Loop Statements

❖Example: n-Bit ripple carry adder

```
module RippleCarryAdderI (Cin, X, Y, S, Cout)
   parameter n = 4;
   input  Cin;
   input [n-1:0] X, Y;
   output  reg [n-1:0] S;
   output reg  Cout;
   reg [n:0] C;
   integer k;
   always  @(X, Y, Cin)
      begin
         C[0] = Cin;
         for (k=0;k<=n-1;k=k+1)
            begin
               S[k] = X[k] ^ Y[k] ^ C[k];
               C[k+1] = (X[k] & Y[k])
                        |(X[k] & C[k])|(Y[k] & C[k]);
            end
         Cout = C[n];
      end
endmodule
```

Sequential Structure



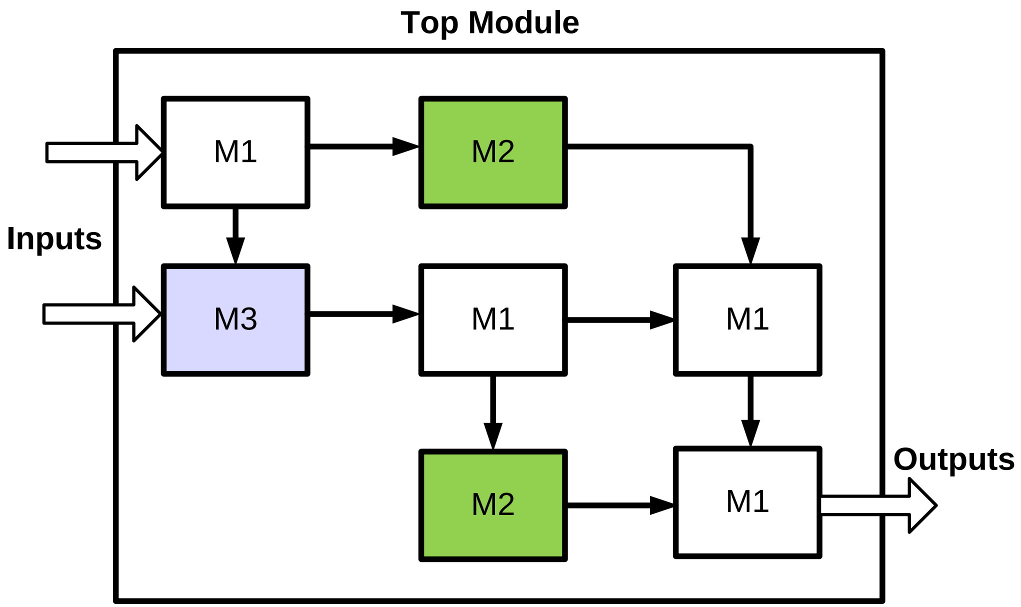
Breaking one statement in two lines is allowed!

Supported

```
module Adder (Cin, X, Y, S, Cout)
   input  Cin;
   input [3:0] X, Y;
   output [3:0] S;
   output Cout;
   assign {Cout, S} = {1’b0, X} + {1’b0, Y} + {4’b0, Cin}
endmodule
```

Using Sub-Circuits (Sub-modules)

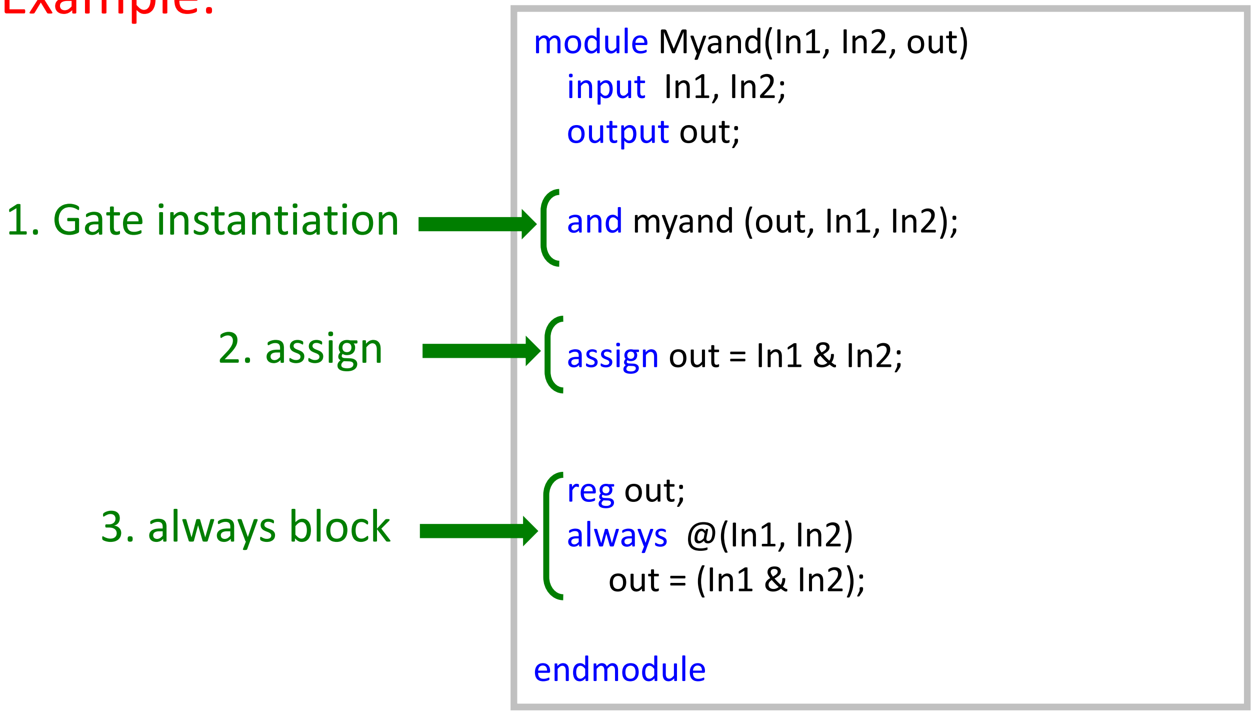
- ❑ A design can use multiple submodules or a module multiple times
- ❑ Using a module in another is called “instantiation”
- ❑ Top-level module: the module that has not been instantiated
- ❑ To use a module inside another, it should be explicitly instantiated



Using Sub-modules

- ❑ There are some built-in **primitive** logic gates in Verilog that can be instantiated
  - ➢ Built-in primitives means there is no need to define a module for these gates
  - ➢ and, or, nor, ….

❖ Example:



1. Gate instantiation

2. assign

3. always block

```
module Myand(In1, In2, out)
   input  In1, In2;
   output out;

   and myand (out, In1, In2);

   assign out = In1 & In2;

   reg out;
   always  @(In1, In2)
       out = (In1 & In2);

endmodule
```

Using Sub-modules : Gate-level Primitives

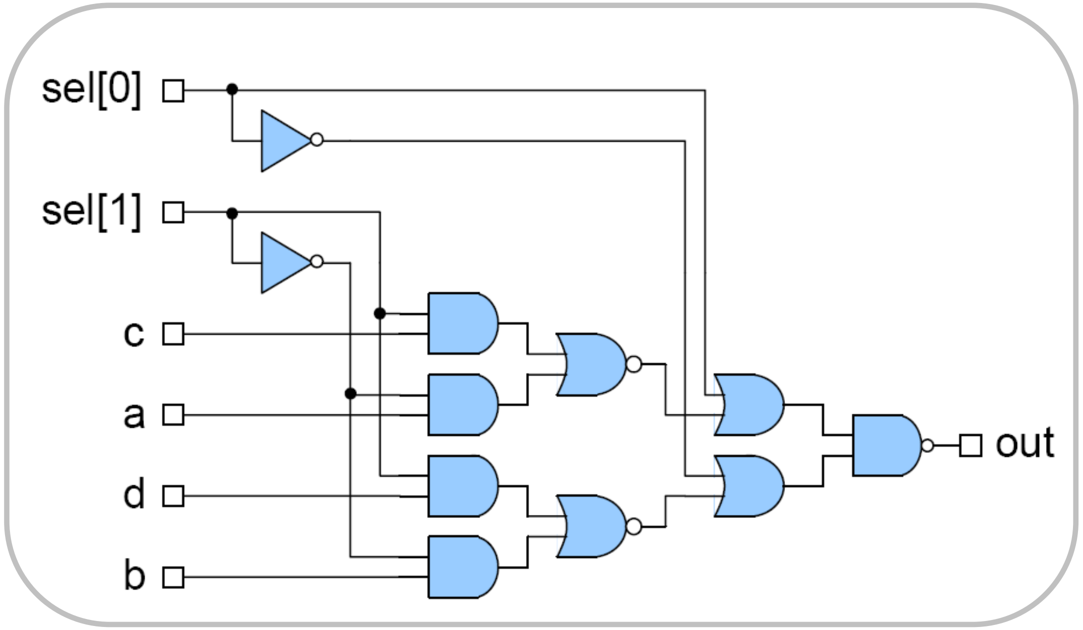
❑ **Gate-Level primitives:**

and	buf	nmos	tran
nand	not	pmos	tranif0
nor	bufif0	cmos	tranif1
or	bufif1	rnmos	rtran
xor	notif0	rpmos	rtranif0
xnor	notif1	rcmos	rtranif1

Using Sub-modules

❖ Example: (4-input MUX using primitives)

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire [1:0] sel_b;
  not not0( sel_b[0], sel[0] );
  not not1( sel_b[1], sel[1] );

  wire n0, n1, n2, n3;
  and and0( n0, c, sel[1]   );
  and and1( n1, a, sel_b[1] );
  and and2( n2, d, sel[1]   );
  and and3( n3, b, sel_b[1] );

  wire x0, x1;
  nor nor0( x0, n0, n1 );
  nor nor1( x1, n2, n3 );

  wire y0, y1;
  or or0( y0, x0, sel[0]   );
  or or1( y1, x1, sel_b[0] );

  nand nand0( out, y0, y1 );

endmodule
```



Sub-modules Instantiation

- ❑ To instantiate a module, two things have to be clearly specified
  - ➢ module’s ports
  - ➢ module’s parameters (considered as default if not specified)
- ❑ Format:

```
Module_name  #(parameter_value)  instance_name (.port_name(port-connection), .port_name(port-connection),….)
```

⇕

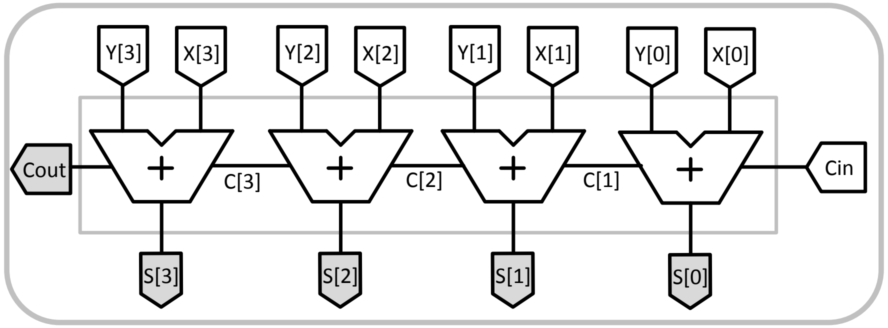
```
Module_name   instance_name (.port_name(port-connection), .port_name(port-connection),….)
defparam  instance_name.parameter_name = parameter_value
```

- ❑ If port connections are in the same order as the original module “.port_name” is not needed in the port list.

Sub-modules Instantiation

❖ Example:

```
module Full_Adder (Cin, x, y, S, Cout);
   input  x, y, Cin;
   output  S, Cout;
   wire S, C;
   assign S = x ^ y ^ Cin;
   assign Cout = (x & y)|(x & Cin)|(y & Cin);
endmodule
```



4-bit Ripple Carry Adder

```
module RippleCarryAdderII (Cin, X, Y, S, Cout);
   parameter n = 4;
   input  Cin;
   input [n-1:0] X, Y;
   output  [n-1:0] S;
   output  Cout;
   wire [n-1:0] C;
```

Cin x y S Cout

```
   Full_Adder stage0  (Cin, X[0], Y[0], S[0], C[1]);
   Full_Adder stage1  (C[1], X[1], Y[1], S[1], C[2]);
   Full_Adder stage2  (C[2], X[2], Y[2], S[2], C[3]);
   Full_Adder stage3  (.Cout(Cout), .Cin(C[3]), .x(X[3]), .y(Y[3]), .S(S[3]));

endmodule
```

Can **NOT** be of type "reg"
(output of a submodule)

Implicit list
(Order **IS** important)

Explicit list
(Order **NOT** important)

Sub-modules Instantiation

❖ Example: 5-bit Ripple Carry Adder:

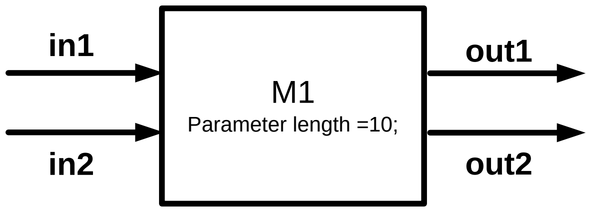
```
module 5-BitRippleCarryAdder (Cin, X, Y, S, Cout)
   parameter n = 5;
   input  Cin;
   input [n-1:0] X, Y;
   output  [n-1:0] S;
   output  Cout;
   wire  C;


   RippleCarryAdderI  #(3)  stage0   (.Cin(Cin), .X(X[2:0]), .Y(Y[2:0]), .S(S[2:0]), .Cout(C));

   defparam  stage1.n = 2;
   RippleCarryAdderI stage1  (.Cin(C), .X(X[4:3]), .Y(Y[4:3]), .S(S[4:3]), .Cout(Cout));

endmodule
```

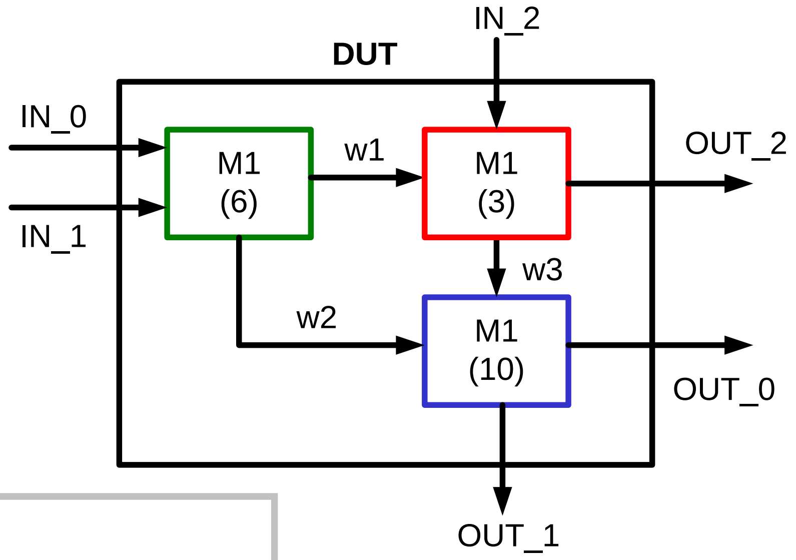
```
defparam  stage0.n = 3;
```

If two parameters : # (3,8)

Sub-modules Instantiation

❖ Example:





```
module DUT (IN, OUT)
   input [2:0] IN;
   output [2:0] OUT;
   wire  w1, w2, w3;

 defparam  stage0.length = 6;
  M1 stage0  (IN[0], IN[1], w1, w2);

  defparam  stage1.length = 3;
  M1 stage1  (.in1(w1), .in2(IN[2]),  .out2(w3), .out1(OUT[2]));

  M1 stage2  (.in1(w2), .in2(w3), .out1(OUT[0]), .out2(OUT[1]));

endmodule
```

Function Construct

❑ function may be used to have a modular code without defining separate modules

❑ A function is defined inside a module

❑ Not crucial for Verilog but might facilitate modular coding

❑ A function can be called both in continuous and procedural assignments

❑ A function can have multiple inputs but does not have any output

❑ Function name serves as the output

Function Construct

- Example: 16-to-1 multiplexer:

```
module my16-to-1MUX (W, S, Out);
   input  [0:15] W;
   input [3:0] S;
   output  reg Out;
   reg  [0:3] M;

   function my4-to-1MUX;
       input [0:3] W;
       input [1:0] s;
       if (s==0) my4-to-1MUX = W[0];
       else if (s==1) my4-to-1MUX = W[1];
       else if (s==2) my4-to-1MUX = W[2];
       else if (s==3) my4-to-1MUX = W[3];
   endfunction

   always@ (W, S)
      begin
         M[0] = my4-to-1MUX(W[0:3],S[1:0]);
         M[1] = my4-to-1MUX(W[4:7],S[1:0]);
         M[2] = my4-to-1MUX(W[8:11],S[1:0]);
         M[3] = my4-to-1MUX(W[12:15],S[1:0]);

         Out   = my4-to-1MUX(M[0:3], S[3:2]);

      end
endmodule
```

```
if       (S[3:2]==0) Out= M[0];
else if (S[3:2]==1) Out= M[1];
else if (S[3:2]==2) Out= M[2];
else if (S[3:2]==3) Out= M[3];
```

Function Construct with multiple-bit output

- Example:

```
module test_fcn (a, b, c, Out);
   input  a, b, c;
   output reg [2:0] Out;

function [2:0] myfcn;
   input a, b, c;
   begin
      myfcn[0] = a^b;
      myfcn[1] = b^c;
      myfcn[2] = c^a;
   end
endfunction

always @(*)
     Out = myfcn(a,b,c);

endmodule
```

With always

```
module test_fcn (a, b, c, Out);
   input  a, b, c;
   output [2:0] Out;

function [2:0] myfcn;
   input a, b, c;
   begin
      myfcn[0] = a^b;
      myfcn[1] = b^c;
      myfcn[2] = c^a;
   end
endfunction

assign   Out = myfcn(a,b,c);

endmodule
```

With assign

Task Construct

❑ task may be used to have a modular code without defining separate modules

❑ A task is defined inside a module

❑ A task can only be called from inside and always (or initial) block

❑ A task can have multiple inputs and outputs

Task Construct

- Example: 16-to-1 multiplexer:

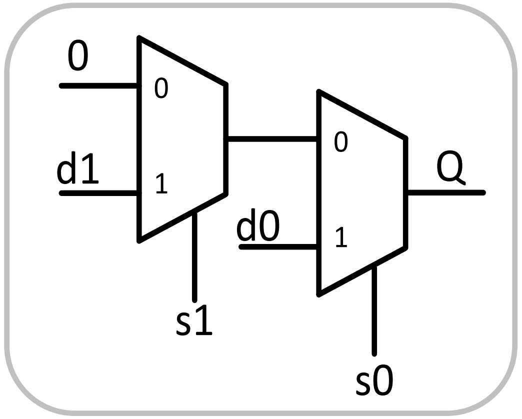
```
module 16-to-1MUX (W, S, Out)
   input  [0:15] W;
   input [3:0] S;
   output  reg Out;
   reg  [0:3] M;

    task 4-to-1MUX;
      input [0:3] W;
      input [1:0] s;
      output Result;
      begin
           if (s==0) Result= W[0];
           elseif (s==1) Result = W[1];
           elseif (s==2) Result = W[2];
           elseif (s==3) Result = W[3];
       end
   endtask
   always@ (W, S)
      begin
         4-to-1MUX(W[0:3],S[1:0], M[0]);
         4-to-1MUX(W[4:7],S[1:0] , M[1]);
         4-to-1MUX(W[8:11],S[1:0] , M[2]);
         4-to-1MUX(W[12:15],S[1:0] , M[3]);
         4-to-1MUX(M[0:3],S[3:2] , Out);
      end
endmodule
```

HDL for Synthesis (Priority logic)

❑ The order in which assignments are written in an always block may affect the logic that is synthesized. **(both conditions in if and else if can be true!)**

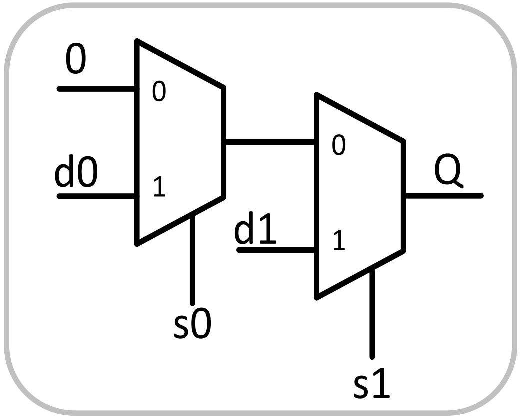
❖ Example:

```
always @ (s0,s1, d0, d1)
   begin
      Q = 0;
      if (s0)   Q = d0;
      else if (s1) Q = d1;
   end
```



Different

```
always @ (s0,s1, d0, d1)
   begin
      Q = 0;
      if (s1)   Q = d1;
      else if (s0) Q = d0;
   end
```



Non of the above infer latch, why?

Example: Up & Down Counters

4-Bit unsigned down-counter with synchronous set

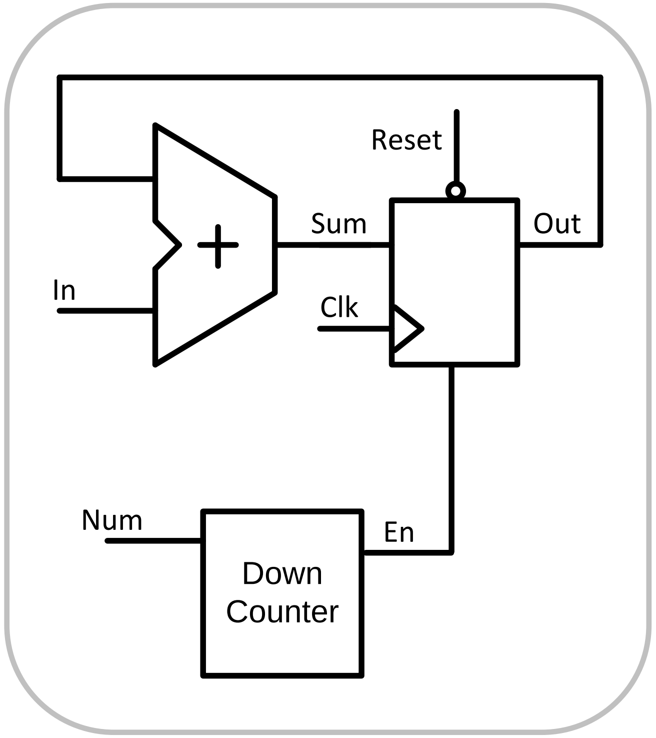
```
module D_counter (C, S, Q);

    input C, S;
    output [3:0] Q;
    reg [3:0] tmp;
    always @(posedge C)
        begin
         if (S)
           tmp <= 4’b1111;
         else
           tmp <= tmp - 1’b1;
        end
    assign Q = tmp;

endmodule
```

4-Bit up-counter with asynchronous reset and modulo maximum

```
module U_counter (C, CLR, Q);

    parameter
      MAX_SQRT = 4,
      MAX = (MAX_SQRT*MAX_SQRT);
     input C, CLR;
     output [MAX_SQRT-1:0] Q;
     reg [MAX_SQRT-1:0] cnt;
    always @ (posedge C or posedge CLR)
         begin
           if (CLR)
              cnt <= 0;
           else
              cnt <= (cnt + 1) %MAX;
           end
     assign Q = cnt;

endmodule
```

Accumulator

- ❑ Accumulates multiple successive k-bit values and stores them into a k-bit register
- ❑ The number of successive numbers (Num) as an input



```
module Accumulator (In, Num, Clk, Rst, Out);
   parameter k = 8;
   parameter m = 4;
   input  [k-1:0] In;
   input [m-1:0] Num;
   input Clk, Rst;
   output  reg  [k-1:0] Out;
   wire [k-1:0] Sum;
   reg  [m-1:0] C;
   wire En, Cout;
   defparam stage0.n = k;
   RippleCarryAdderI  stage0   (.Cin(0), .X(In), .Y(Out), .S(Sum), .Cout(Cout));
   always@ (posedge Clk, negedge Rst)
         if (Rst == 0)
           begin
             C <= Num;
             Out <= {k{1‘b0}};
           end
          else if (En)
           begin
             C  <= C-1;
             Out <= Sum;
           end
assign En = |C;
endmodule
```

Outline

❑ ASIC/FPGA Design Flow

❑ Hardware Description Language (HDL)

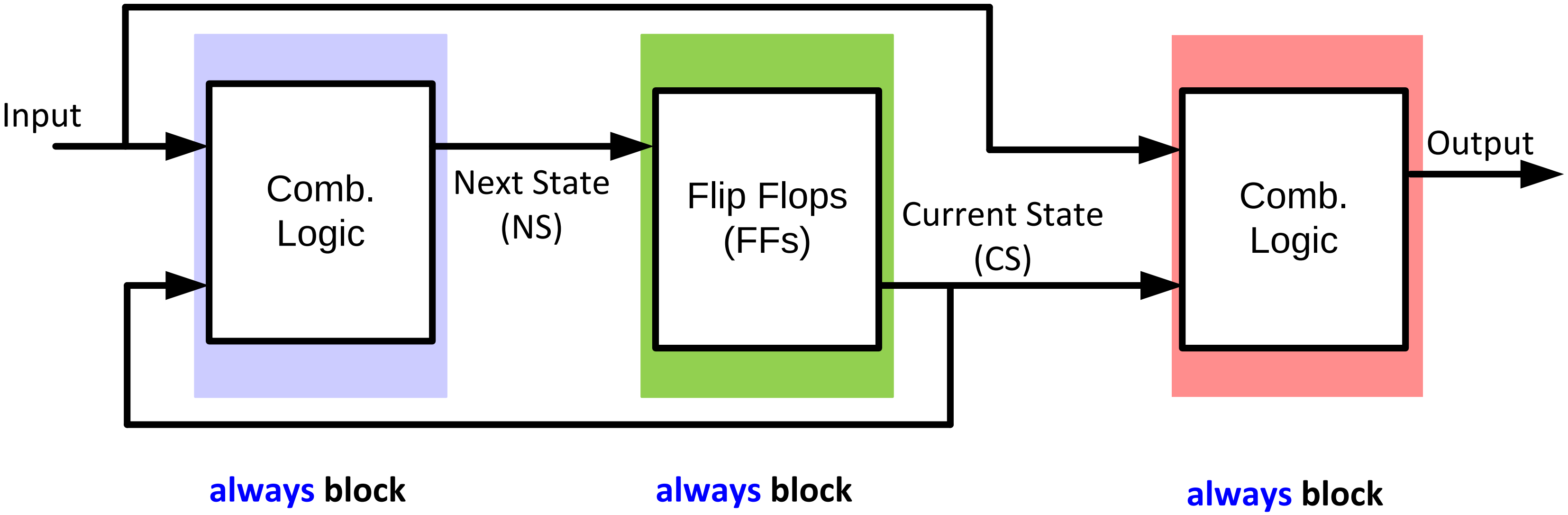
- ➢ Verilog
  - o Introduction
  - o Language Fundamentals
  - o Modeling Combinational & Sequential Logic Circuits
  - o Modeling Finite State Machines
  - o Verilog Operations

Finite State Machine (FSM)

- Used to implement control sequencing
- An FSM is defined by
  - set of inputs
  - set of outputs
  - set of states
  - initial state
  - transition function
  - output function
- States are steps in a sequence of transitions
- There are “Finite”   number of states.

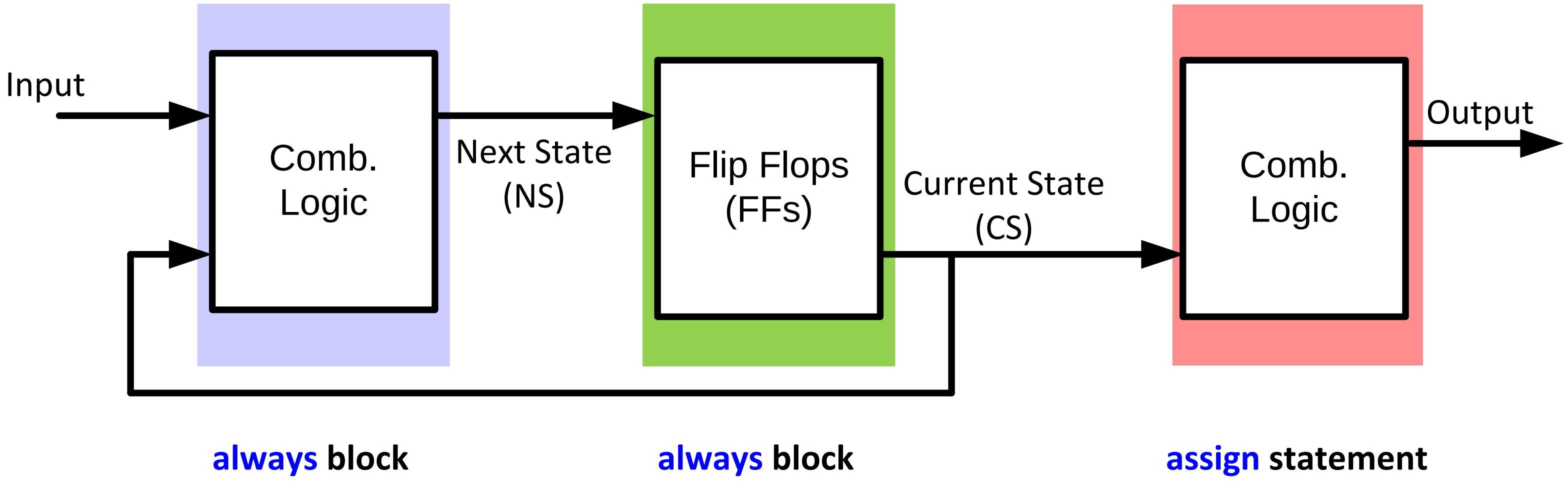
Finite State Machine (FSM)

- ❑ The behavior of the circuit can be represented using a finite number of states
- ❑ Two types:
  - **Mealy:**
    - Output depends on the "current state" and the "input"



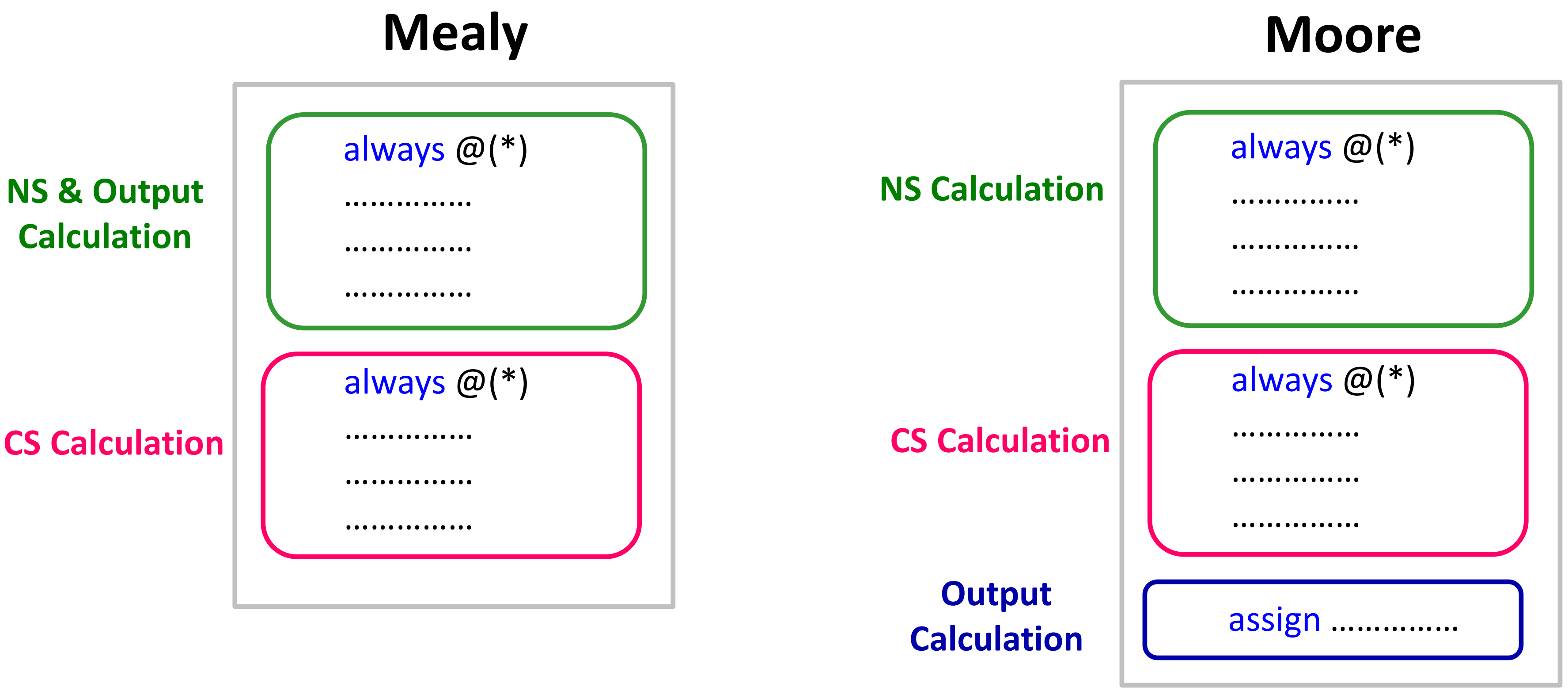
Finite State Machine (FSM)

- **Moore:**
  - Output depends only on the “current state”



- Therefore, to describe an FSM in Verilog we have to show how to derive:
  - Next State (NC)
  - Current State (CS)
  - Output

FSM Code Structure



Mealy

NS & Output Calculation

```
always @(*)
...............
...............
...............
```

CS Calculation

```
always @(*)
...............
...............
...............
```

Moore

NS Calculation

```
always @(*)
...............
...............
...............
```

CS Calculation

```
always @(*)
...............
...............
...............
```

Output Calculation

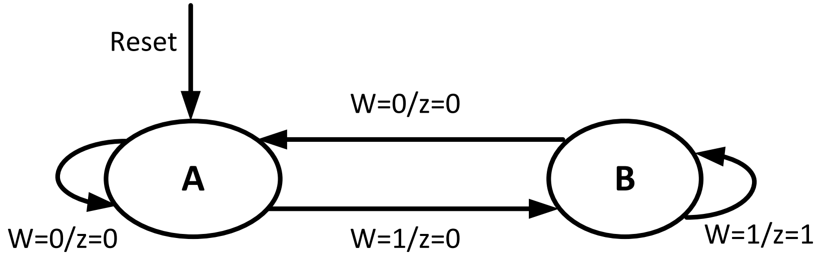
```
assign ...............
```

❑ Output depends on input
❑ Output declared as reg

❑ Output does not depend on input
❑ Output declared as wire

FSM

❖ Example: Mealy Machine



Output: reg

```
module mealy (Clock, w, Resetn, z);
 input Clock, w, Resetn ;
 output reg z ;
 reg CS, NS;
 parameter A = 1'b0, B = 1'b1;
 always @(w, CS)
   case (CS)
      A:  if (w == 0)
             begin
               NS = A;   z = 0;
             end
          else
             begin
               NS = B;    z = 0;
             end
    B:  if (w == 0)
             begin
               NS = A;    z = 0;
             end
         else
             begin
               NS = B;    z = 1;
             end
  endcase
 always @(posedge Clock, negedge Resetn)
  if (Resetn == 0)
    CS <= A;
  else
    CS <= NS;
endmodule
```

NS & Output Calculation

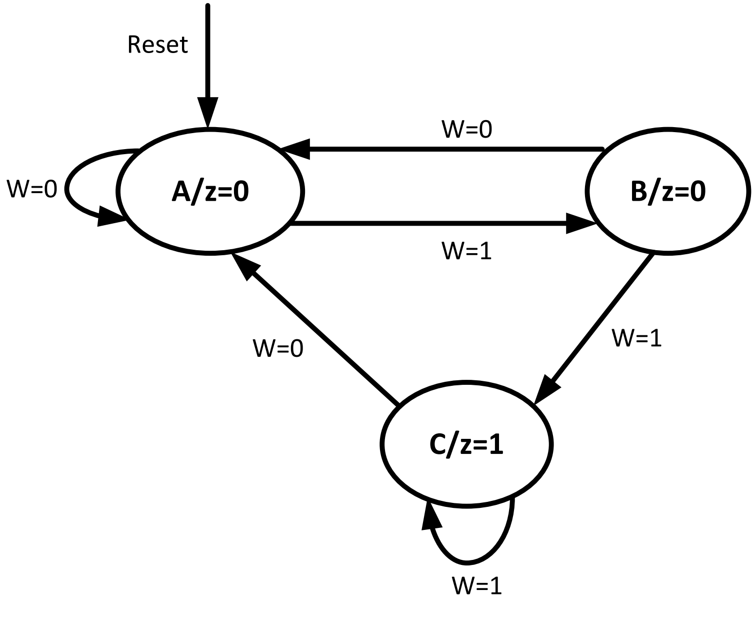
Combinational
(Blocking)

CS Calculation

Sequential
(Non-Blocking)

FSM

- Example: Moore Machine



Output: wire

```
module moore (Clk, w, Resetn, z);
 input Clk, w, Resetn;
 output z;
 reg [1:0] CS, NS;
 parameter A = 2'b00, B = 2'b01, C = 2'b10;

 always @(w, CS)
 begin
  case (CS)
   A:  if (w == 0)  NS = A;
    else  NS = B;
   B:  if (w == 0)  NS = A;
    else  NS = C;
   C:  if (w == 0)  NS = A;
    else  NS = C;
   default:    NS = 2'bxx;
  endcase
 end
 always @(posedge Clk, negedge Resetn)
 begin
  if (Resetn == 0)
   CS <= A;
  else
   CS <= NS;
 end

 assign z = (CS == C);
endmodule
```

NS Calculation

Combinational (Blocking)

CS Calculation

Sequential (Non-Blocking)

Output Calculation

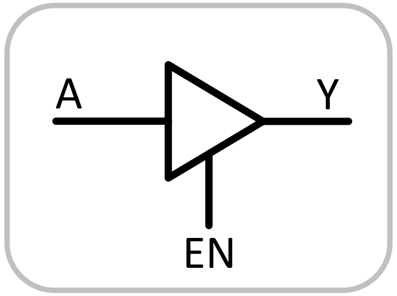
Outline

- ❑ ASIC/FPGA Design Flow
- ❑ Hardware Description Language (HDL)
  - ➢ Verilog
    - o Introduction
    - o Language Fundamentals
    - o Modeling Combinational & Sequential Logic Circuits
    - o Modeling Finite State Machines
    - o Verilog Operations

Tri-State Logic in Verilog

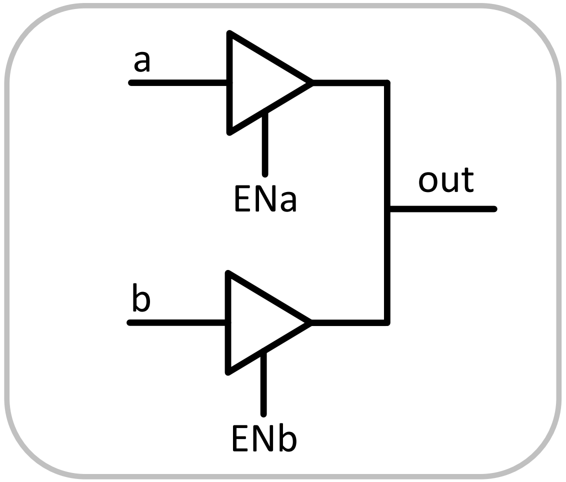
- Tri-state buffer:

$$Y = \begin{cases} A & EN = 1 \\ Z & EN = 0 \end{cases}$$



```
module tri-buffer (A, y, EN)
   input A, EN;
   output Y;

   assign Y = (EN) ? A : 1’bZ;

endmodule
```

- Tri-state driver inference:

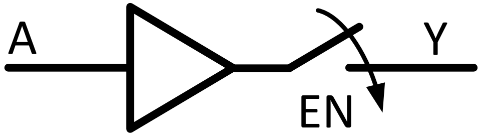


```
always @ (ENa, a)
   begin
     if (ENa)
        out = a;
     else
        out = 1’bz;
   end
always @ (ENb, b)
   begin
     if (ENb)
        out = b;
     else
        out = 1’bz;
   end
```

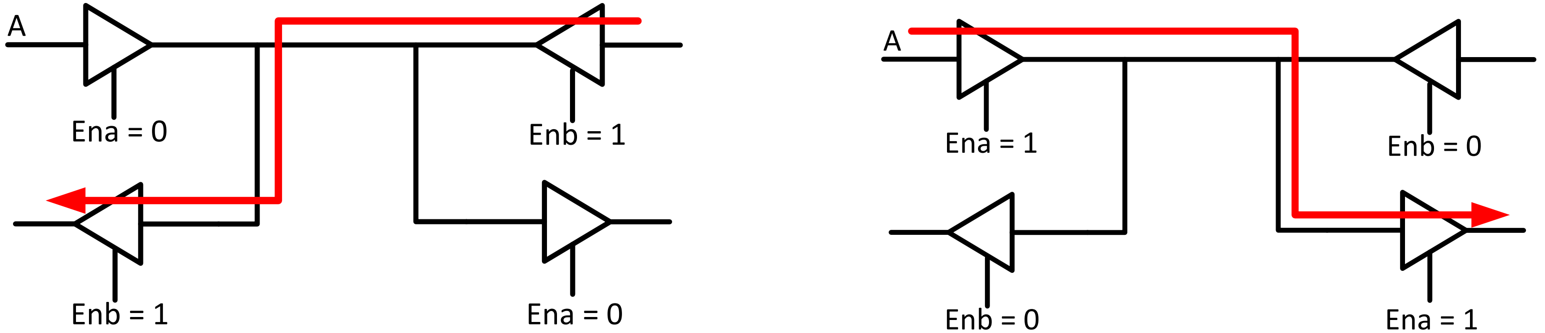
```
     assign out = (ENa) ? a : 1’bz;
     assign out = (ENb) ? b : 1’bz;
```

Tri-State Applications

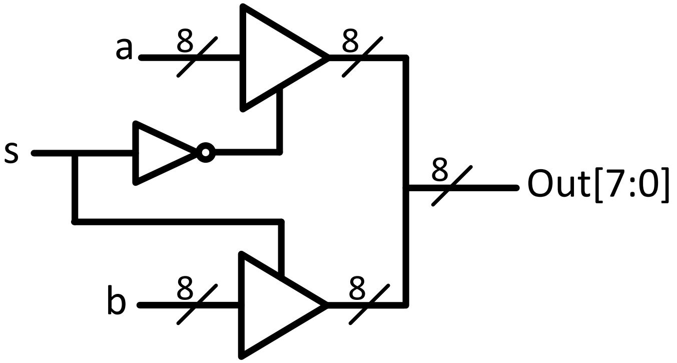
1. **Buffering:**



2. **Half-duplex communication:**

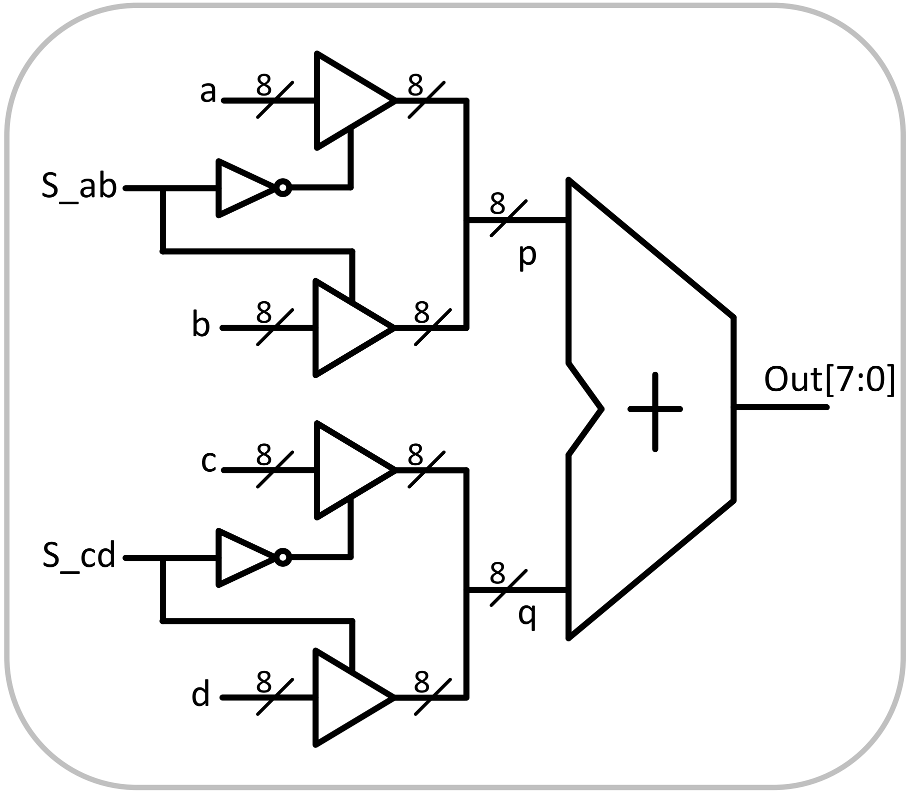


3. **Bus multiplexing:**



Tri-State Applications

- Example: Adder with four options



```
module tri-adder (a, b, c, d, S_ab, S_cd, Out);

  input S_ab, S_cd;
  input [7:0] a, b, c, d;
  output [8:0] Out;
  wire [7:0] p, q;

  assign p = ~S_ab ? a : 8’bzzzzzzzz;
  assign p =   S_ab ? b : 8’bzzzzzzzz;
  assign q = ~S_cd ? c : 8’bzzzzzzzz;
  assign q =   S_cd ? d : 8’bzzzzzzzz;

  assign Out = p + q;

endmodule
```

Z is an allowed logic value and implies a tri-state driver for synthesis

Verilog Operations: Right/Left Shift

- ❑ Verilog supports << for left and >> for right shift. (Only one position)
- ❑ Both of these operators use a zero for the shift input bit.
- ❑ We can also control the shift input

```
module LRShift (Si, L, R, In, Out);

 input Si, L, R;
 input [7:0] In;
 output [7:0] Out;

 always @ (L, R, In, Si)
        begin
                case({R,L})
                   2'b01  :  Out = {In[6:0], Si};   // Left shift
                   2'b10  :  Out = {Si, In[7:1]};   // Right shift
                   default:  Out = In;
                endcase
       end
endmodule
```

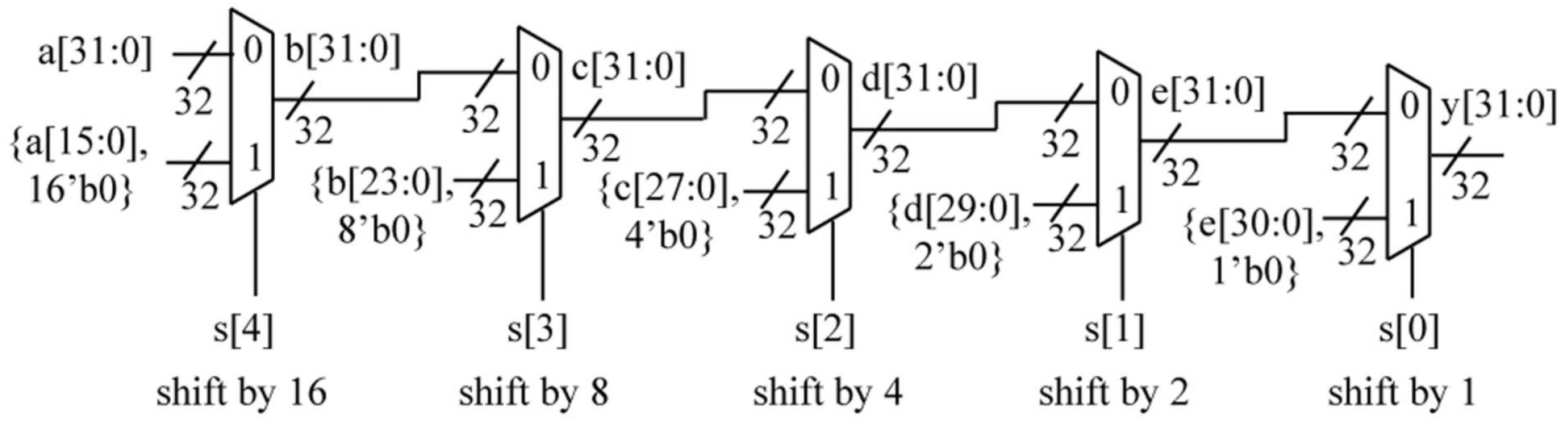
Verilog Operations: Barrel Shifter

- ❑ **Barrel shifter** shifts a signal by multiple positions

```
module BarrelShifter (s, a, y);
  input [4:0] s;
  input [31:0] a;
  output [31:0] y;
  assign y = a<<s;
 endmodule
```

❖ **Example:**

- ➢ 32-bit left shift barrel shifter
- ➢ Left shifts by 0 to 31 positions based on the 5-bit s input
- ➢ Each of its stages corresponds to a fixed shift by a power of 2 (16, 8, 4, 2, 1)
- ➢ Simple HDL implementation, which illustrates the power of HDL to hide implementation details from a designer



Counters

- ❑ Stores an unsigned integer value
  - ➢ Increments or decrements the value
- ❑ Used to count occurrences of
  - ➢ Events
  - ➢ Repetitions of a processing step
- ❑ Used as timers
  - ➢ Count elapsed time intervals by incrementing periodically

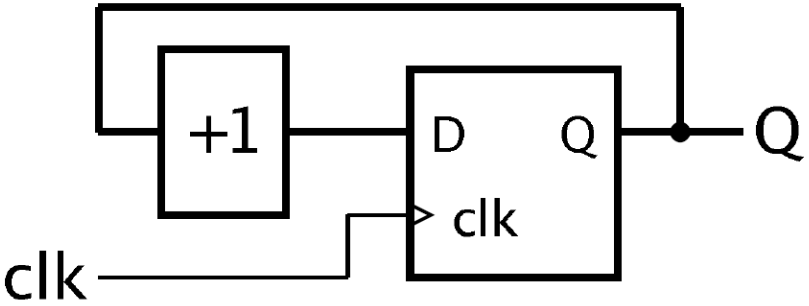
Free-running Counter:

❑ Increments every rising edge of clock

- Up to 2^n–1, then wraps back to 0
- Counts modulo 2^n

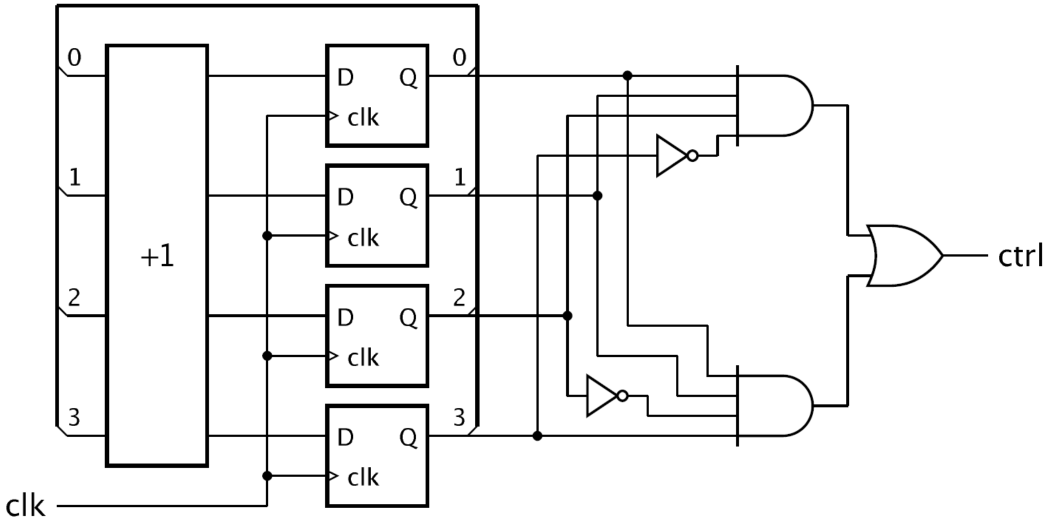
❑ This counter is synchronous

❑ All outputs governed by clock edge



Example: Periodic Control Signal

- ❑ Count modulo 16 clock cycles
- ❑ Control output = 1 every 8th and 12th cycle
- ❑ Decode count values 0111 and 1011



```
module decoded_counter ( output ctrl,
input clk );
reg [3:0] count_value;
always@(posedge clk)
        count_value <= count_value + 1;
        assign ctrl = count_value == 4'b0111
        || count_value == 4'b1011;
endmodule
```

Fixed-point vs. Floating-point

- Fixed-point means allocating a fixed number of bits with a fixed pointer position to represent numbers.
  - Simpler for implementation
  - Less accuracy

- Floating-point representation is provide a much more extensive means for providing real number representations and tend to be used extensively in scientific computation applications.
  - More flexible/accuracy
  - More complexity on implementation side (some times 10 times larger hardware than fixed-point counterpart!)

Verilog Operations: Fixed-Point Simulation

- ❑ For realization of DSP algorithms all variables should be converted to the fixed-point representation

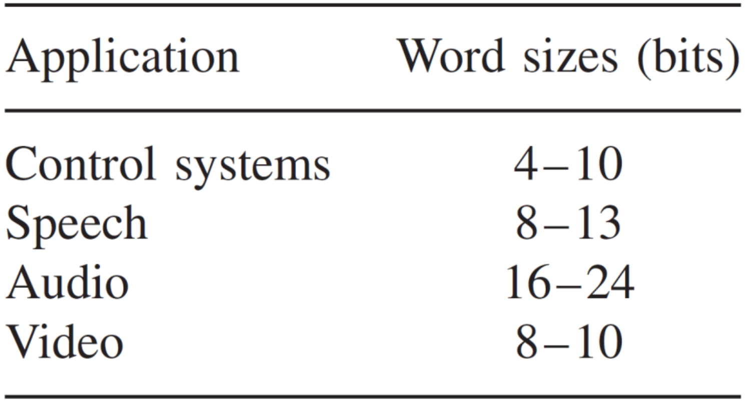
- ❑ Normally 2’s complement representation is used to represent signed numbers

- ❑ A fixed-point 2’s complement representation of a number has two parts:
  - ➢ Integer part (W_I bits)
  - ➢ Fractional part (W_F bits)

- ❑ The length of W_I and W_F are calculated based on the dynamic range of variables
  - ➢ Total length: $W_I + W_F$

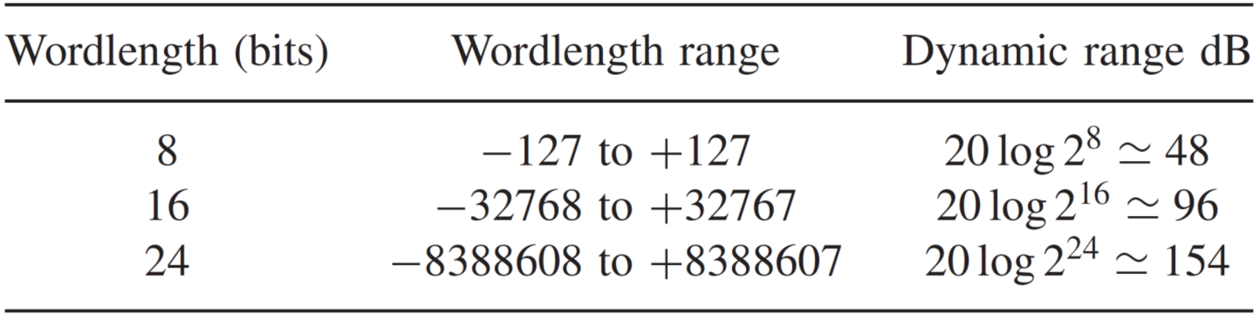
Verilog Operations: Fixed-Point Simulation

❑ Typical word lengths:



Application	Word sizes (bits)
Control systems	4–10
Speech	8–13
Audio	16–24
Video	8–10

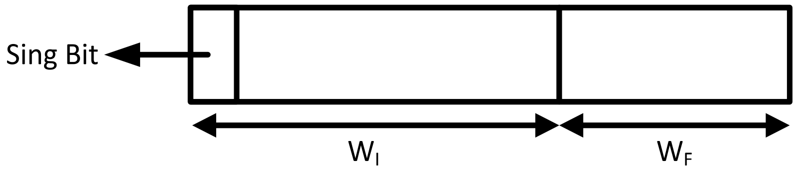
❑ Fixed word-length dynamic range:



Wordlength (bits)	Wordlength range	Dynamic range dB
8	−127 to +127	$20 \log 2^{8} \simeq 48$
16	−32768 to +32767	$20 \log 2^{16} \simeq 96$
24	−8388608 to +8388607	$20 \log 2^{24} \simeq 154$

Verilog Operations: Fixed-Point Simulation

❑ 2's complement Representation: (W_I , W_F) format



$$\text{Sign Bit} = \left[\begin{array}{ll} 0: & \text{positive} \\ 1: & \text{negative} \end{array}\right.$$

❑ Good to represent quantized numbers in the range: $\left[2^{-W_I}, \left(2^{W_I-1} - \left(\frac{1}{2}\right)^{W_F}\right)\right]$

- Resolution : $\left(\frac{1}{2}\right)^{W_F}$

❖ Example:

- in (3,3) 011101 represents 3.625 (smallest number: 0.125)
- in (3,5) 10111000 represents -2.25 (smallest number: 0.03125)

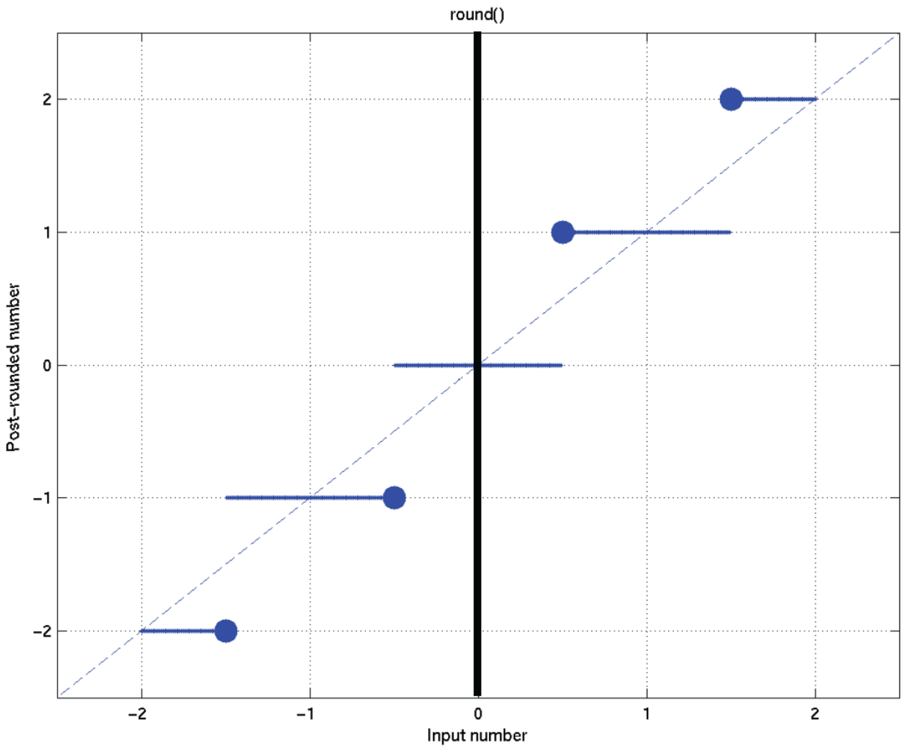
Fixed-Point Simulation: Rounding

- ❑ Eliminates LSB bits
- ❑ Need to reduce the number of bits due to word growth
  - ➢ For example, if we multiply two 5-bit words, the product will have 10 bits, i.e., xxxxx × yyyyy = zzzzzzzzzz and we likely don’t want or need all that precision

- ❑ Matlab rounding:
  - ➢ **round**(·): towards nearest integer
    - ▪ Pos. and neg. numbers are rounded symmetrically about zero
    - ▪ Generally the best possible rounding algorithm
  - ➢ **fix**(·): truncates towards zero
    - ▪ Pos. and neg. numbers are rounded symmetrically about zero
  - ➢ **floor**(·): rounds towards negative infinity
  - ➢ **ceil**(·): rounds towards positive infinity

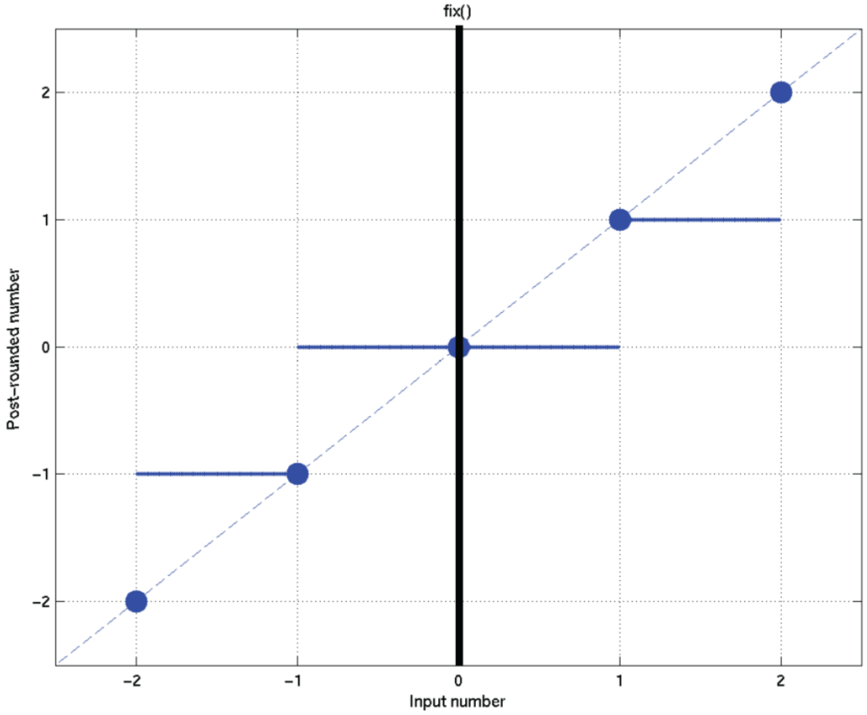
Fixed-Point Simulation: Matlab round(.)

- ❑ One of the best rounding modes
- ❑ "Unbiased" rounding
- ❑ Symmetric rounding for positive and negative numbers
- ❑ Max error ½ LSB



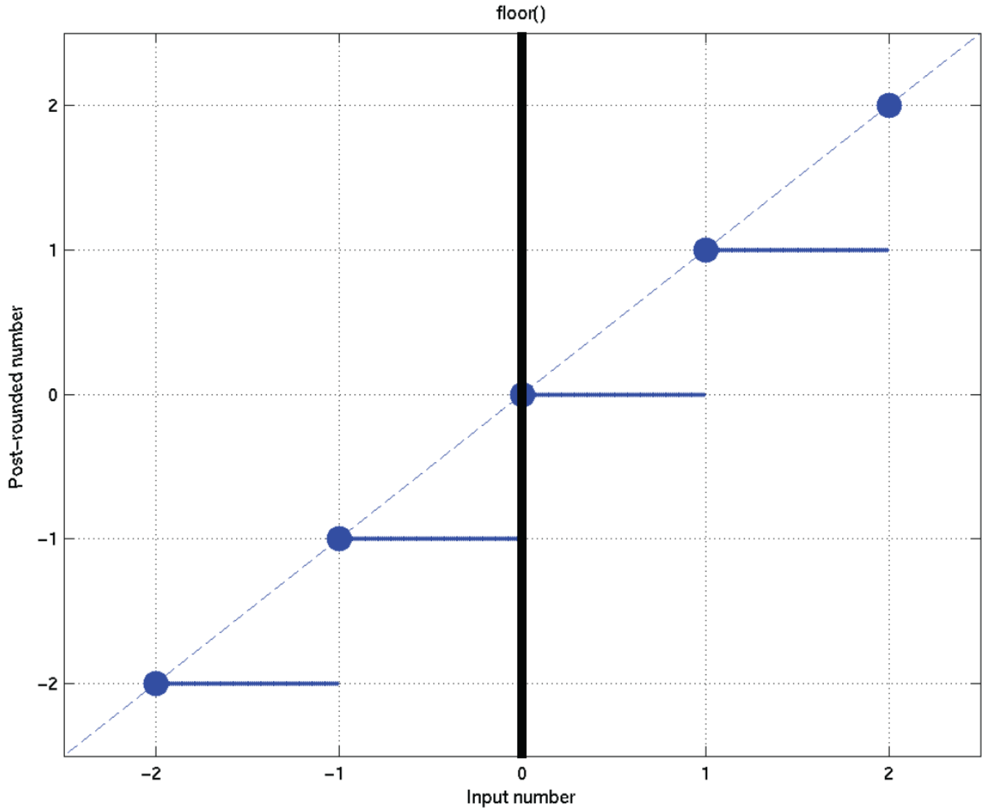
Fixed-Point Simulation: Matlab fix(.)

- ❑ Truncates toward zero
- ❑ Numerical performance poor
- ❑ Symmetric rounding for positive and negative numbers
- ❑ Max error 1 LSB



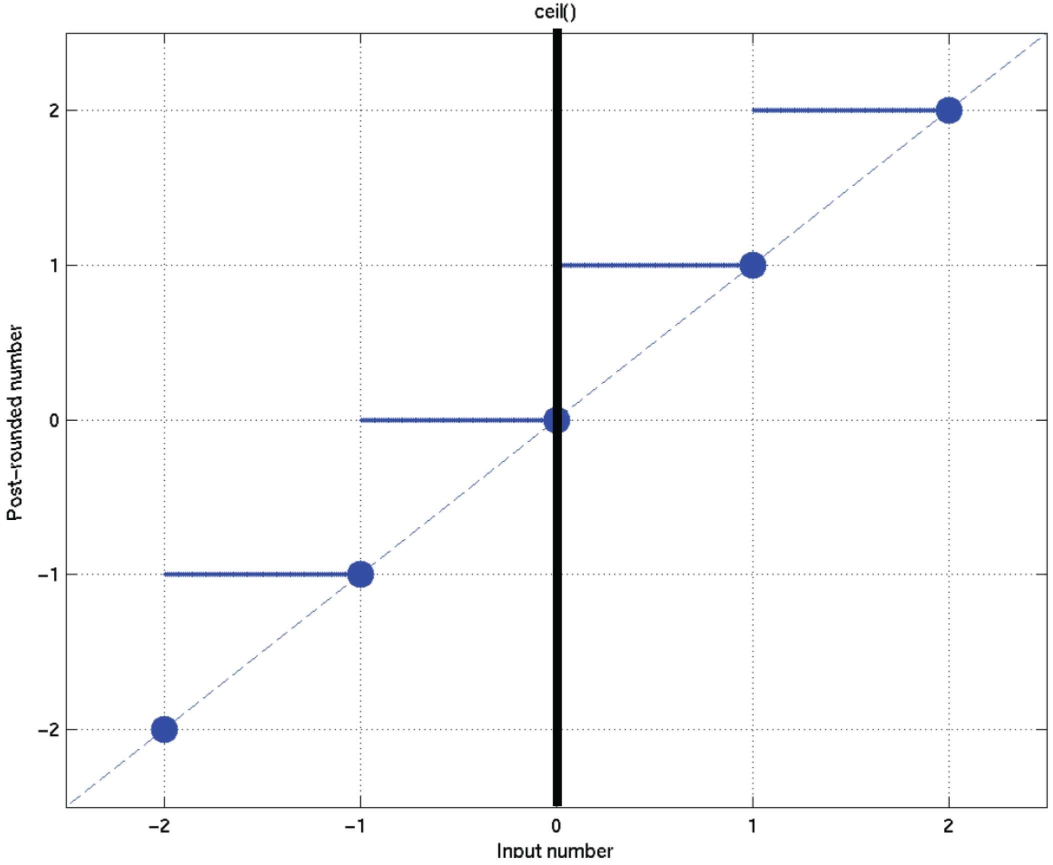
Fixed-Point Simulation: Matlab floor(.) or truncation

- ❑ Numbers rounded down towards –∞ (-infinity)
- ❑ Numerical performance poor
- ❑ Very simple hardware
- ❑ In:xxxxxx -> Out: xxxx--
- ❑ Max error 1 LSB



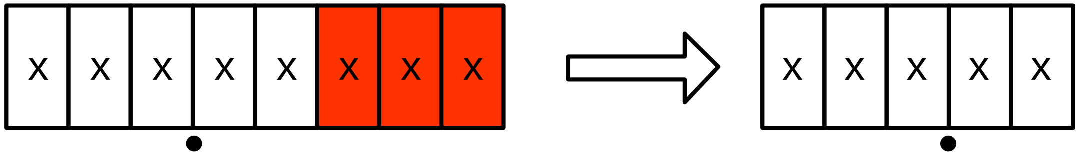
Fixed-Point Simulation: Matlab ceil(.)

- ❑ Numbers rounded up towards + ∞ (+infinity)
- ❑ Numerical performance poor
- ❑ Max error 1 LSB



Hardware Rounding

- Easiest is truncation



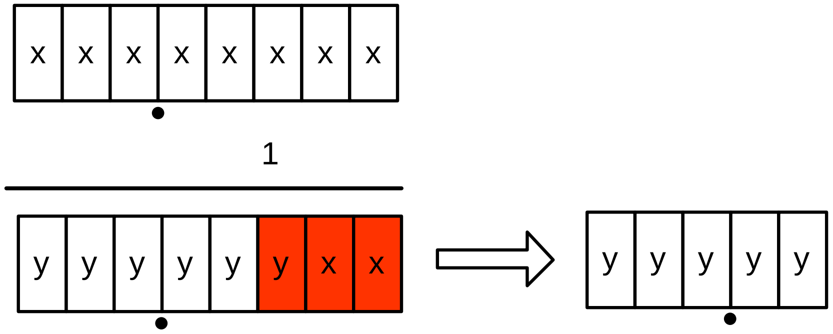
- Maximum rounding error ~1 post-rounded LSB

- Signed magnitude
  - Positive and negative numbers both truncate towards zero
  - Matlab fix(·)

- 2's complement and unsigned
  - All numbers truncate towards negative infinity
  - Matlab floor(·)

Hardware Rounding

- ❑ Better rounding numerically is to add ½ lSB and then truncate



- ❑ Maximum rounding error ½ post-rounded LSB
- ❑ Two cases:

  a. When the input is xxxx.5000 (base 10) (or xxx.xx100 (base 2) in the example above)

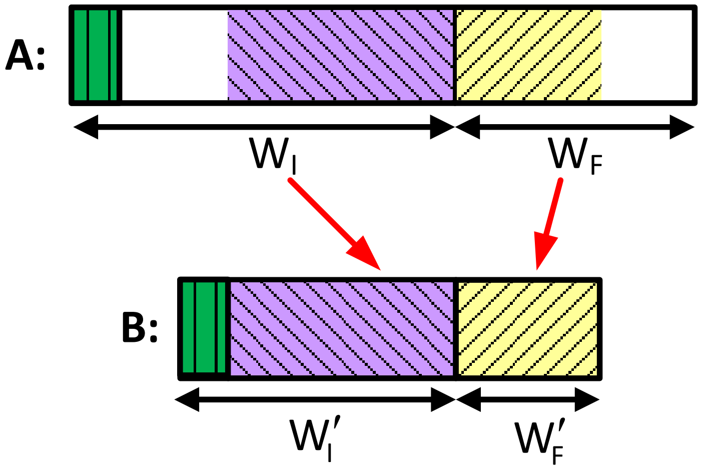
  - Rounding is towards +∞ (for both positive and negative numbers)
  - matlab ceil(·)

  b. Otherwise

  - Performs best rounding: matlab round(·)

Fixed-Point Modeling: Casting

- ❑ Care must be taken when dealing with fixed-point numbers
- ❑ **Casting:** To convert a number with a larger bit length to a smaller one



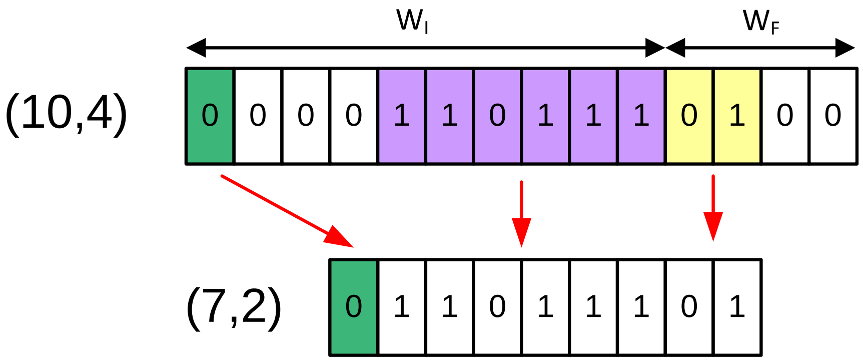
$W_I' < W_I$

$W_F' < W_F$

- ❑ $B = \{A[W_I + W_F - 1], A[W_I' + W_F - 2 : W_F - W_F']\}$
- ❑ Saturation happens if:
  - ➢ "A" is positive and $A[W_I + W_F - 2 : W_I' + W_F - 1] \neq 0$
  - ➢ "A" is negative and $A[W_I + W_F - 2 : W_I' + W_F - 1] \neq 111...111 (\text{all one})$

Fixed-Point Modeling: Casting

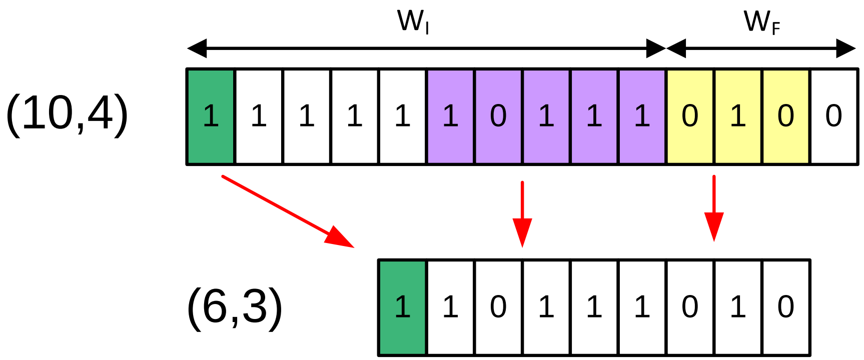
- Example:



W_I W_F

(10,4) 0 0 0 0 1 1 0 1 1 1 0 1 0 0

(7,2) 0 1 1 0 1 1 1 0 1

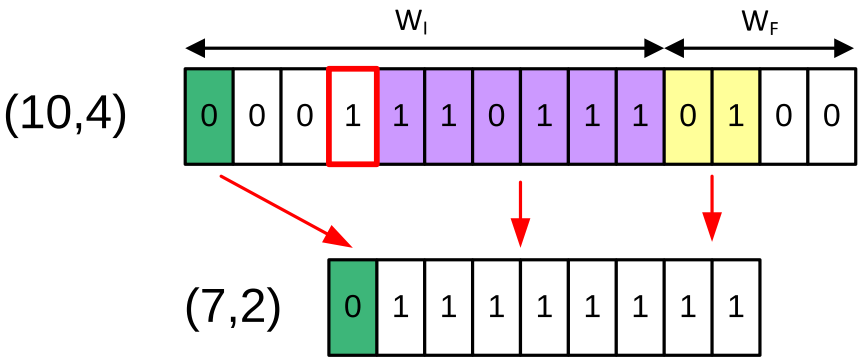


W_I W_F

(10,4) 1 1 1 1 1 1 0 1 1 1 0 1 0 0

(6,3) 1 1 0 1 1 1 0 1 0

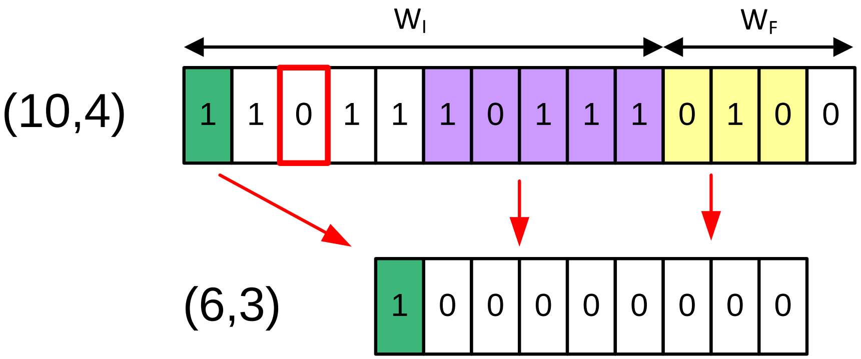
Saturation



W_I W_F

(10,4) 0 0 0 1 1 1 0 1 1 1 0 1 0 0

(7,2) 0 1 1 1 1 1 1 1 1



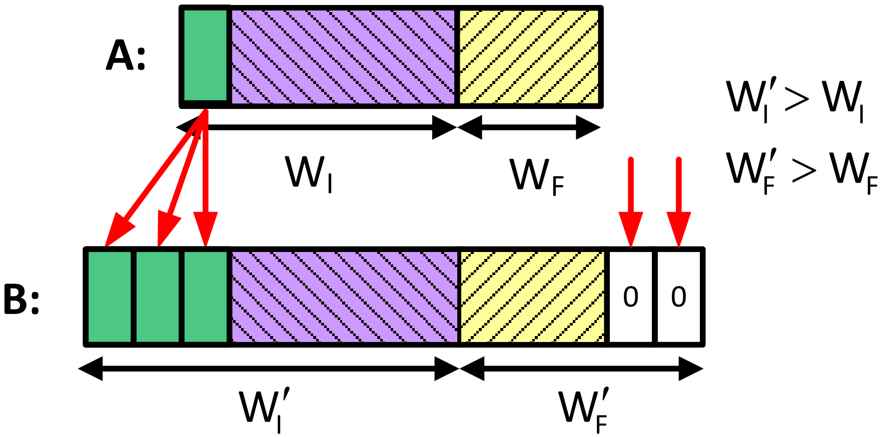
W_I W_F

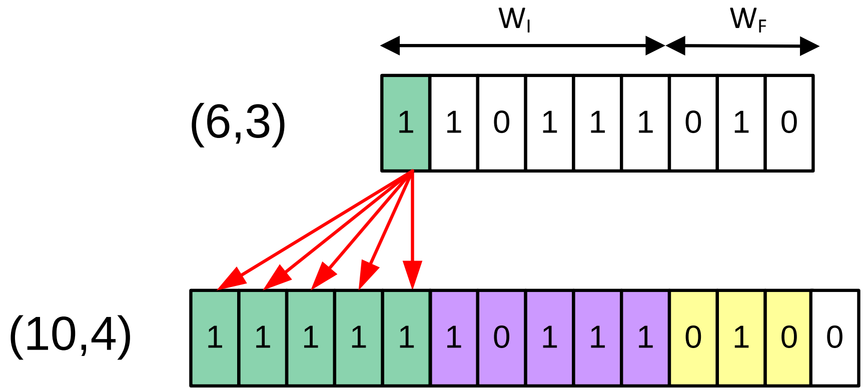
(10,4) 1 1 0 1 1 1 0 1 1 1 0 1 0 0

(6,3) 1 0 0 0 0 0 0 0 0

Fixed-Point Modeling: Sign Extension

- ❑ To convert a number with a smaller bit length to a larger one sign extension is required.





(6,3) 1 1 0 1 1 1 0 1 0

(10,4) 1 1 1 1 1 1 0 1 1 1 0 1 0 0

- ❑ $\text{assign}\ B = \{\{n\{A[W_I + W_F - 1]\}\},A,2'b0\}$
- ❑ Examples: Adding two numbers with different lengths:

```
wire [2:0] A;
wire [5:0] B;
wire [6:0] C;
assign C = {B[5],B} + {{4{A[2]}},A};
```

Verilog Operations: Addition with Same Length

- ❑ Adding two signed n-bit numbers and save it in a signed n-bit number:
  - ➢ Might not be safe if two number are large
  - ➢ Overflow condition should be checked

$$\begin{array}{rr} & A[n-1:0] \\ + & B[n-1:0] \\ \hline & C[n-1:0] \end{array}$$

- ❑ Overflow may happen if:
  - ➢ A[n-1]==1 and B[n-1]==1 and C[n-1]==0
  - ➢ A[n-1]==0 and B[n-1]==0 and C[n-1]==1

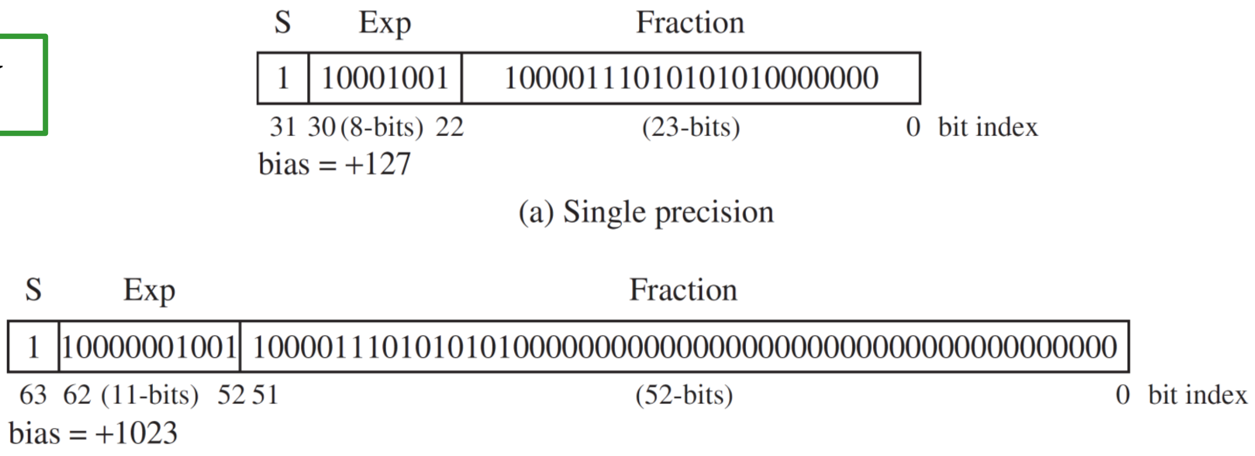
$$\begin{array}{rr} & 0110 \\ + & 0111 \\ \hline & 1101 \end{array} \qquad \begin{array}{rr} & 1010 \\ + & 1001 \\ \hline & 10011 \end{array}$$

```
assign SUM = B + A;
assign OV = (A[n-1]==1 && B[n-1]==1 && C[n-1]==0)||
            (A[n-1]==0 && B[n-1]==0 && C[n-1]==1);

assign C = (OV && A[n-1] == 1) ? MIN_NEG_n : SUM;
assign C = (OV && A[n-1] == 0) ? MAX_POS_n : SUM;
```

Floating-point

- In floating-point, the aim is to represent the real number using a *sign (S)*, *exponent (Exp)* and *mantissa* (or fraction).

$$\boxed{N = 2^{Exp-127} \times M}$$



(a) Single precision

(b) Double precision

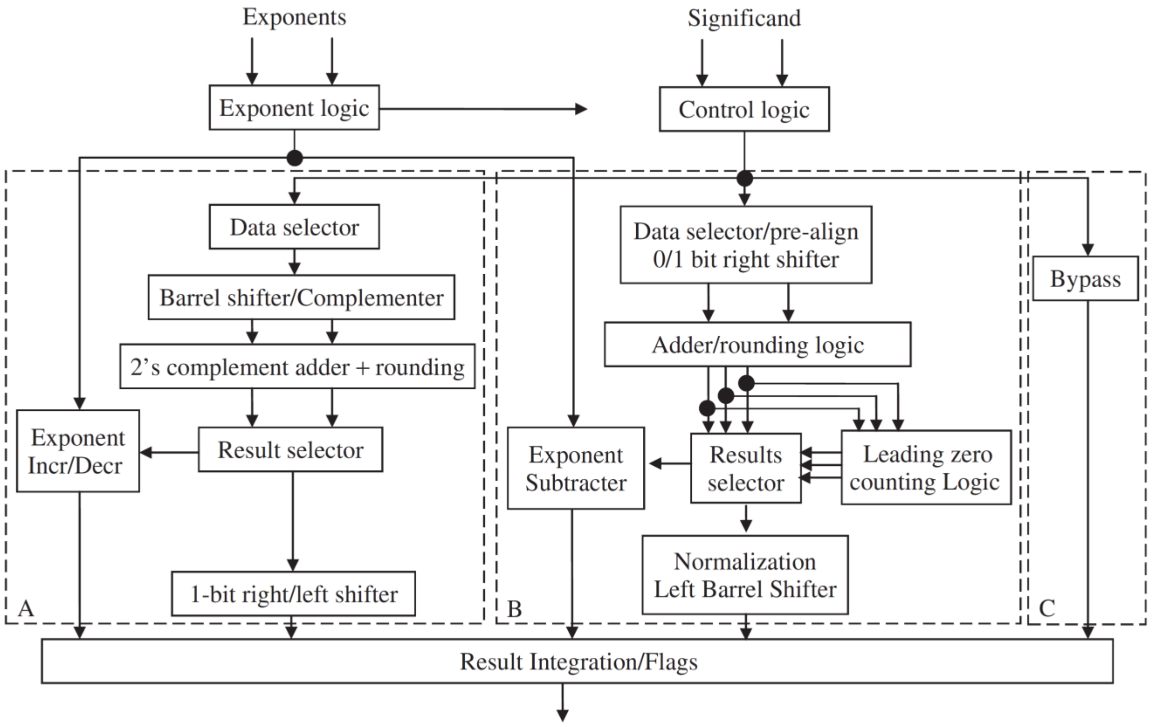
- The most widely used form of floating-point is IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) with two major formats:
  - Single-precision (32-bit)
  - Double-precision (64-bit)

Floating-point: Example

- Converting a real number, −1082.5674 IEEE 754 floating-point representation:
  - It can be determined that *S* = 1 as the number is negative.
  - The number (1082) is converted to binary by successive division, 10000111010.
  - The fractional part (0.65625) is computed in the same way as above, giving 10101.
  - The parts are combined to give the value 10000111010.10101.
  - The radix point is moved left, to leave a single 1 on the left, $1.000011101010101 \times 2^{10}$.
  - Filling with 0s to get the 23-bit mantissa gives the value 10000111010101010000000.
  - The exponent is 10 and with the 32-bit IEEE 754 format bias of 127, giving 137 which is given as 10001001 in binary.

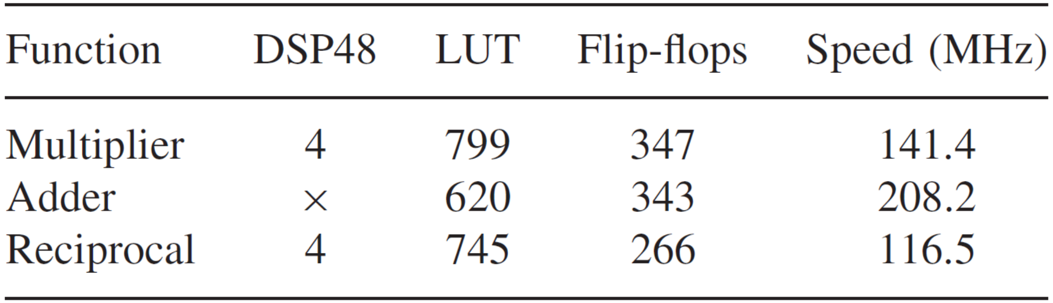
Floating-point Implementation

- The floating-point implementation concurs a complicated hardware compared to the fixed-point counterpart.
- Take into account as an example a floating-point adder!
  - This additional logic is needed to perform the various normalization steps for the adder



Fixed-point vs. Floating-point

- ❑ The area comparison for floating-point is additionally complicated as the relationship between multiplier and adder area is now changed.
- ❑ In fixed-point, multipliers are generally viewed to be *N* times bigger than adders where *N* is the word length.
- ❑ However, in floating-point, the area of floating-point adders is comparable to that of floating-point multipliers which corrupts the assumption at the algorithmic stages to reduce number of multiplications in favor of additions.
- ❑ Table below gives some figures on area and speed figures for floating-point addition and multiplication implemented in a Xilinx Virtex 4 FPGA technology.



Function	DSP48	LUT	Flip-flops	Speed (MHz)
Multiplier	4	799	347	141.4
Adder	×	620	343	208.2
Reciprocal	4	745	266	116.5

Verilog Operations: $signed and $unsigned

- □ A = $signed(B)
  - ➢ Sign extends B and assigns it to A
  - ➢ bit width(B) < bit width (A)
  - ❖ Example

```
wire [5:0] A;
assign A = $signed (3b’110);
```

→ **A = 111110**

- □ A = $unsigned (B)
  - ➢ Zero fill B and assign it to A
  - ➢ bit width(B) < bit width (A)
  - ❖ Example

```
wire [5:0] A;
assign A = $unsigned (3b’110);
```

→ **A = 000110**

Verilog Operations: Signed Addition

- ❑ There are two ways to perform signed addition:

  1. Sing Extension:

```
wire [2:0] A, B;
wire [3:0] SUM;
assign SUM = {B[2],B} + {A[2],A};
```

  2. Using signed signals

```
wire signed [2:0] A, B;
wire signed [3:0] SUM;
assign SUM = B + A;
```

Same result

```
  1110  (-2)
  0010  (+2)
 ------
 10000  (0)
```

Discard Overflow

- ❑ Wrong otherwise:

```
wire [2:0] A, B;
wire [3:0] SUM;
assign SUM = B + A;
```

```
  110  (-2)
  010  (+2)
 -----
 1000  (-8)
```

(Wrong)

Verilog Operations: Signed Addition with Carry-in

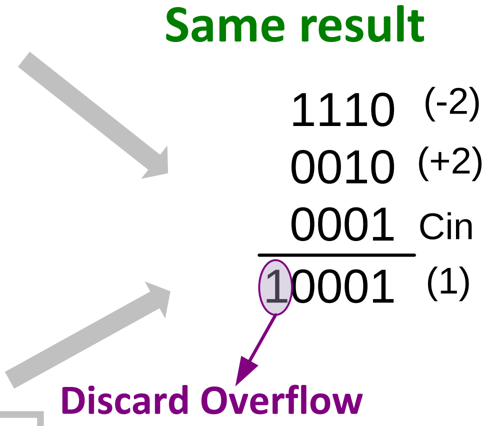
❑ There are two correct ways to perform signed addition with carry-in:

1. Sing Extension:

```
wire [2:0] A, B;
wire Cin;
wire [3:0] SUM;
assign SUM = {B[2],B} + {A[2],A} + Cin;
```

2. Using signed signals

```
wire signed [2:0] A, B;
wire Cin;
wire signed [3:0] SUM;
assign SUM = B + A + $signed({1’b0},Cin);
```



Same result

```
  1110  (-2)
  0010  (+2)
  0001  Cin
 10001  (1)
```

Discard Overflow

Verilog Operations: Signed Addition with Carry-in

- **Incorrect** Codes:

```
wire signed [2:0] A, B;
wire Cin;
wire signed [3:0] SUM;
assign SUM = B + A + Cin;
```

```
 110  (-2)
 010  (+2)
   1  Cin
----
1001  (9)
```

If any operand of an operation is unsigned, the entire operation is performed unsigned

```
wire signed [2:0] A, B;
wire signed Cin;
wire signed [3:0] SUM;
assign SUM = B + A + Cin;
```

```
1110  (-2)
0010  (+2)
1111  Cin
----
1111  (-1)
```

When Cin=1, it sign extends it, to match the size of A and B, which is incorrect!

```
wire signed [2:0] A, B;
wire Cin;
wire signed [3:0] SUM;
assign SUM = B + A + $signed(Cin);
```

```
1110  (-2)
0010  (+2)
1111  Cin
----
1111  (-1)
```

When Cin=1, it sign extends it, to match the size of A and B, which is incorrect!

Verilog Operations: Signed Multiplication

- Use signed construct as we used for signed addition:

  1. Use Verilog constructs:

```
wire signed [16:0] A, B;
wire signed [31:0] MULT;
assign MULT = A*B;
```

  2. Write it manually as a module

Complicated!

Verilog Operations: Signed Multiplication

- ❑ Multiplication of a signed number and an unsigned number:
  - ❑ **Correct:**

```
wire signed [2:0] A;
wire [2:0] B;
wire signed [5:0] PROD;
assign PROD = A*$signed({1'b0,B});
```

$$\begin{array}{r l} 110 & (-2) \\ 111 & (7) \\ \hline 110010 & (-14) \end{array}$$

  - ❑ **Incorrect:**

```
wire signed [2:0] A;
wire [2:0] B;
wire signed [5:0] PROD;
assign PROD = A*$signed(B);
```

When B[2]==1, treats it as a negative number!

$$\begin{array}{r l} 110 & (-2) \\ 111 & \text{(7) treat it as (-1)} \\ \hline 000010 & (+2) \end{array}$$

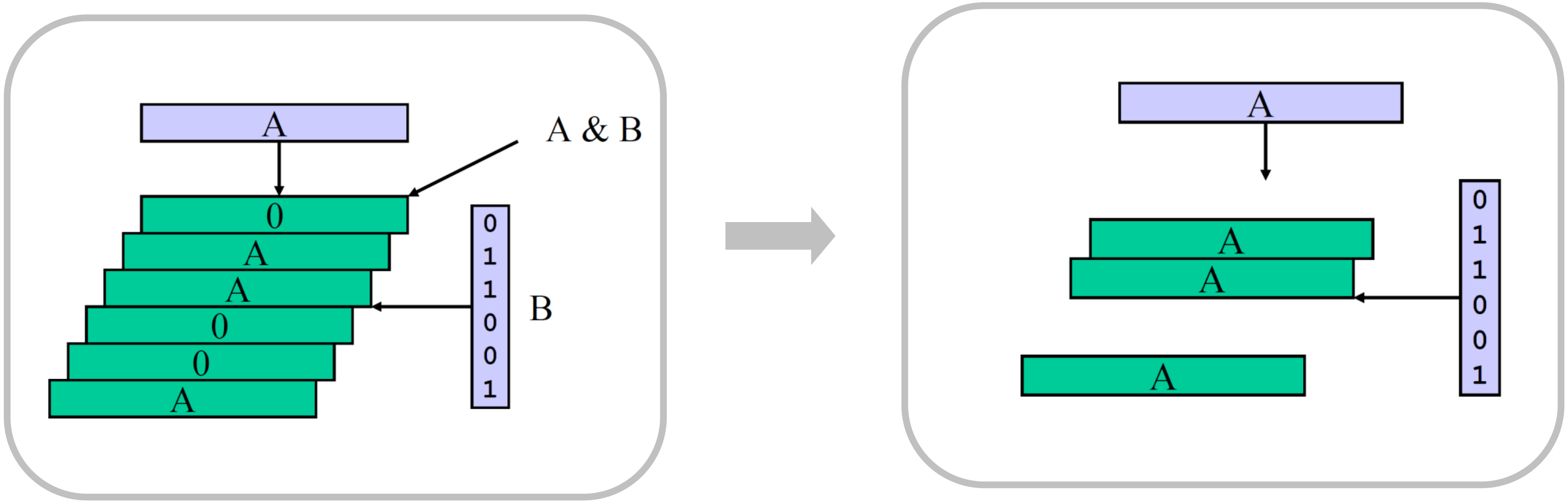
```
wire signed [2:0] A;
wire [2:0] B;
wire signed [5:0] PROD;
assign PROD = A*B;
```

Entire operation is performed unsigned

$$\begin{array}{r l} 110 & (-2) \\ 111 & (7) \\ \hline 101010 & (42) \end{array}$$

Verilog Operations: Fixed Multiplication

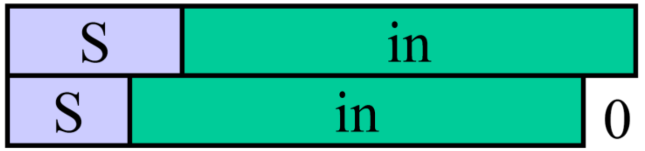
- Sometimes one input is fixed so remove partial products that are always zero
- We have to try to find the minimum number of power-of-2 numbers to add together to equal the fixed multiplier input



Verilog Operations: Fixed Multiplication

- ❑ Example: Multiply by 3:

```
input [7:0] in;
wire [9:0] product;
assign product = {in[7], in, 1’b0}
+ {in[7], in[7], in};
```



- ❑ Example: Multiply by 56:

56=32+16+8

```
input [7:0] in;
wire [13:0] product;
assign product =
{in[7], in, 5’b00000}
+ {in[7], in[7], in, 4’b0000}
+ {in[7], in[7], in[7], in, 3’b000};
```

56=64-8

```
input [7:0] in;
wire [13:0] product;
assign product =
{in, 6’b00000}
- {in[7], in[7], in[7], in, 3’b000};
```

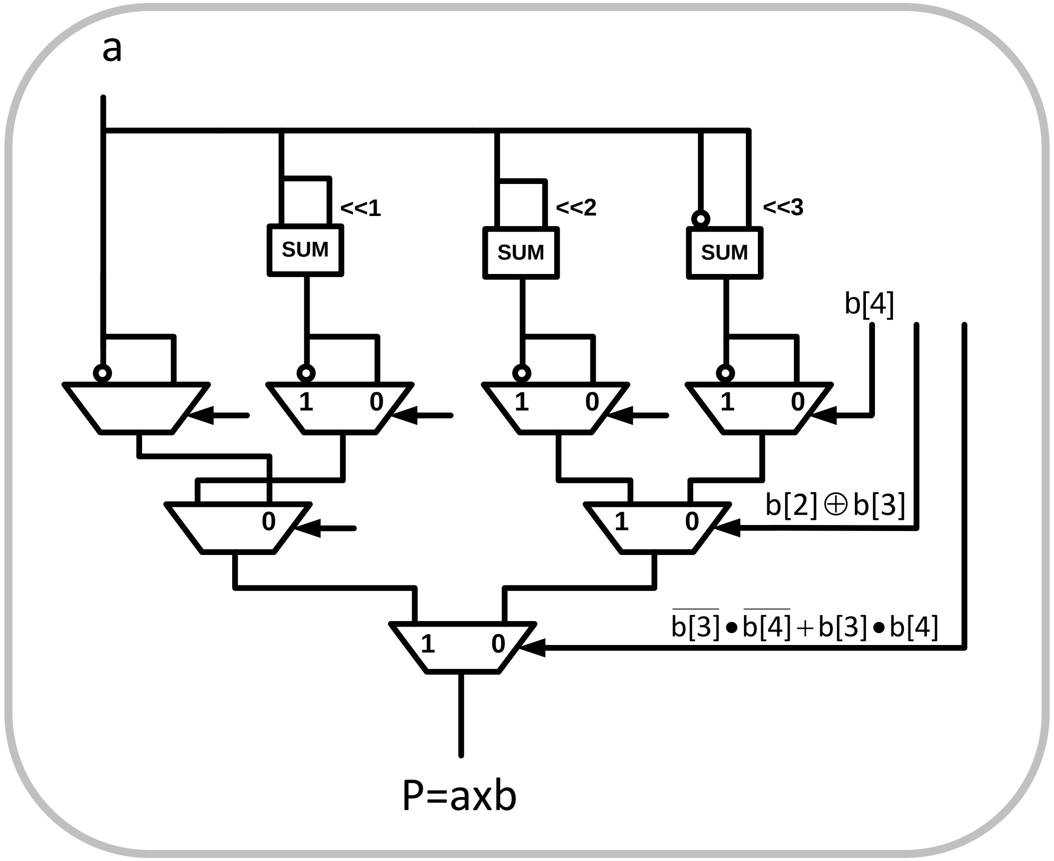
Verilog Operations: Constant Multiplication

- ❑ Multiplication with a set of constant numbers may be implemented more efficiently: $P = a \times b \qquad b \in \{-7,-5,-3,-1,1,3,5,7\}$

b

MSB 4 3 2 1 LSB

0111
0101
0011
0001
1111
1101
1011
1001

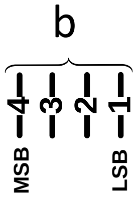


a
<<1
<<2
<<3
SUM
SUM
SUM
b[4]
1 0
1 0
1 0
0
b[2]⊕b[3]
1 0
$\overline{b[3]} \bullet \overline{b[4]} + b[3] \bullet b[4]$
1 0
P=axb

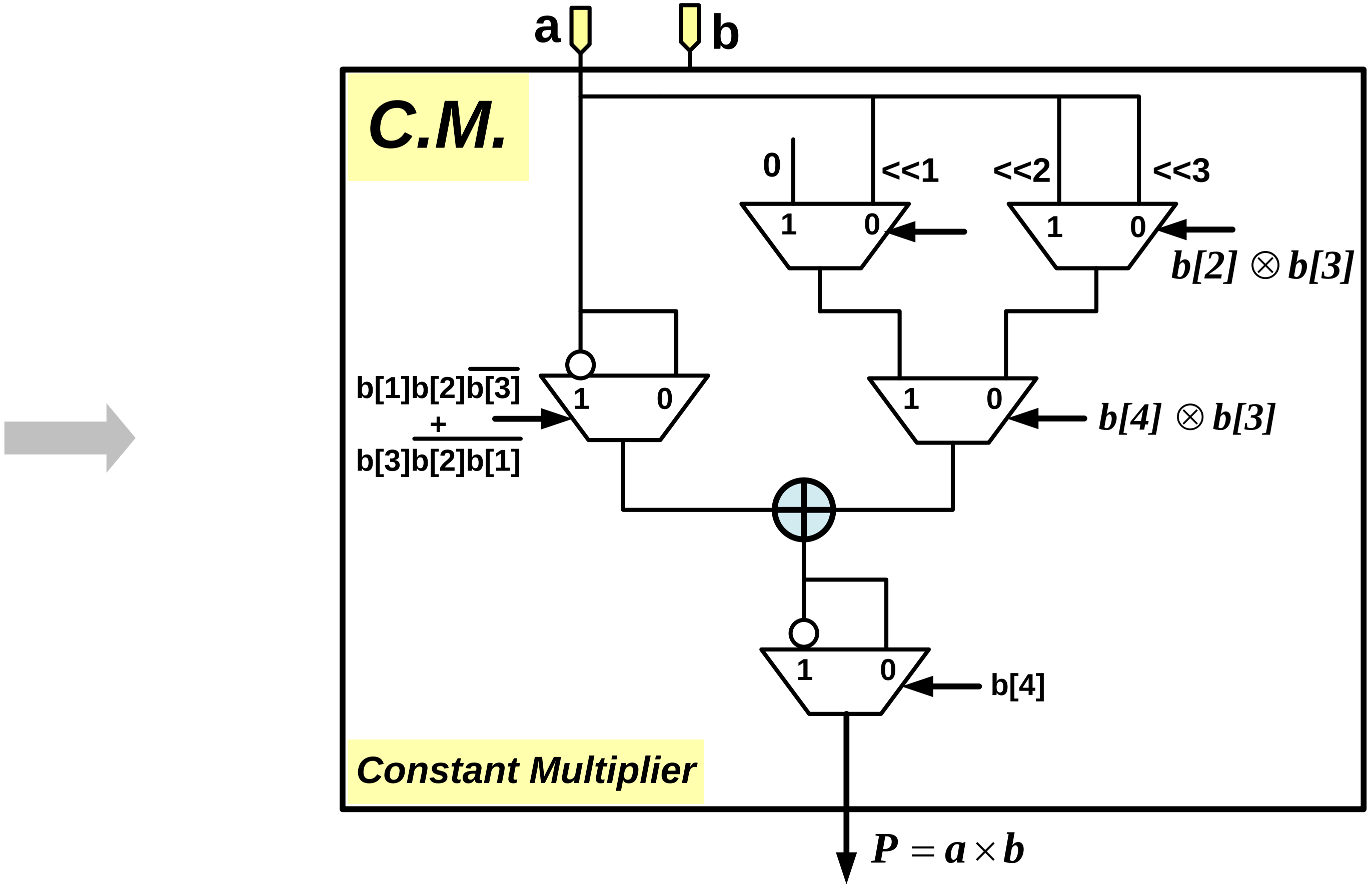
Verilog Operations: Constant Multiplication

❑ Simpler way for implementation: $P = a \times b$ $\quad b \in \{-7,-5,-3,-1,1,3,5,7\}$

Area (um^2)	Critical Path	Multiplier
1800	3.5	Constant MUL
12000	5.1	Normal MUL



0111
0101
0011
0001
1111
1101
1011
1001



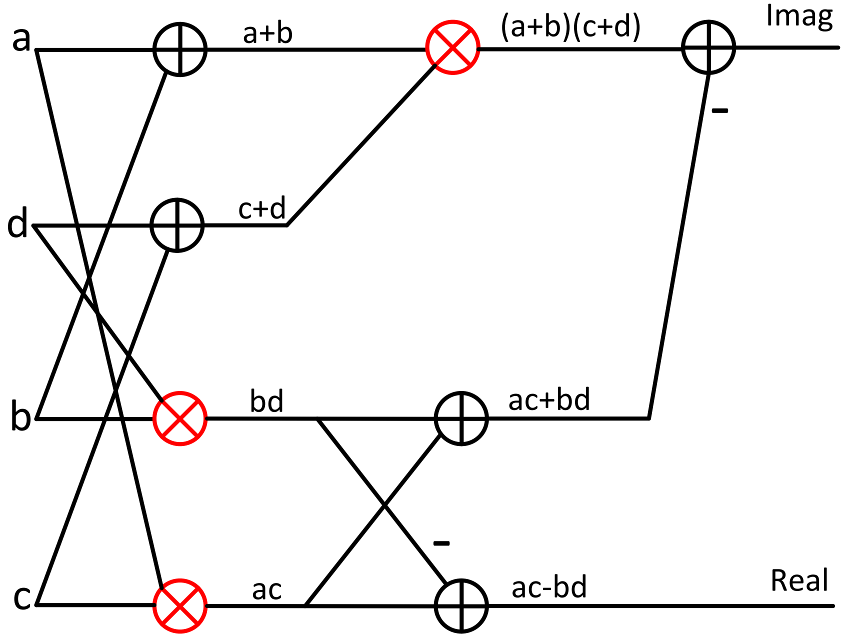
Verilog Operations: Complex Multiplication

- ❑ A complex multiplication is equivalent to four real multiplications

$$(a+jb)(c+jd)=(\underbrace{ac}-\underbrace{bd})+j(\underbrace{ad}+\underbrace{bc})$$

- ❑ However, it can be efficiently realized using only three real multiplications:

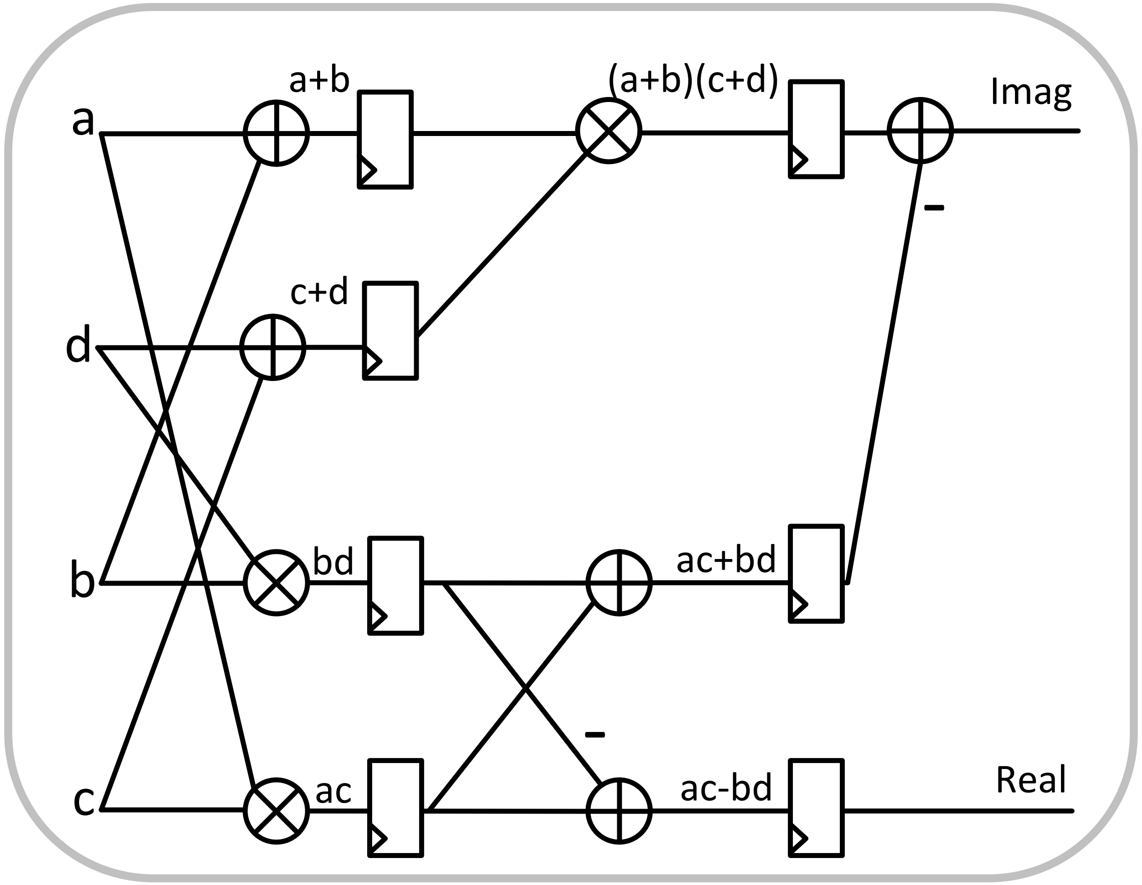
$$(a+jb)(c+jd)=(\underbrace{ac}-\underbrace{bd})+j\left[\underbrace{(a+b)(c+d)}-(ac+bd)\right]$$



Pipelined Complex Multiplication

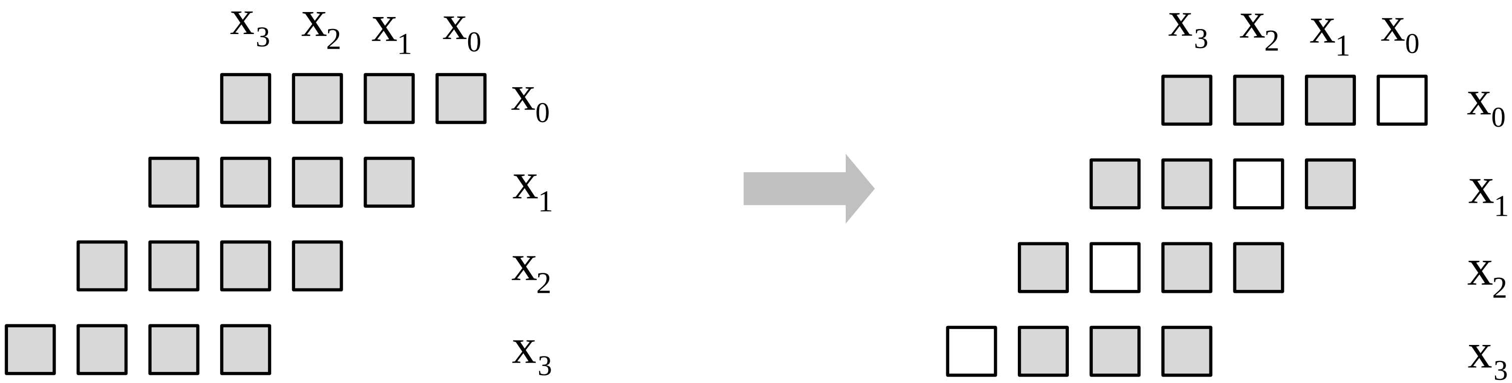
- ❑ Pipelined Implementation:

$$(a+jb)(c+jd)=(\underbrace{ac}-\underbrace{bd})+j\left[\underbrace{(a+b)(c+d)}-(ac+bd)\right]$$



Squaring

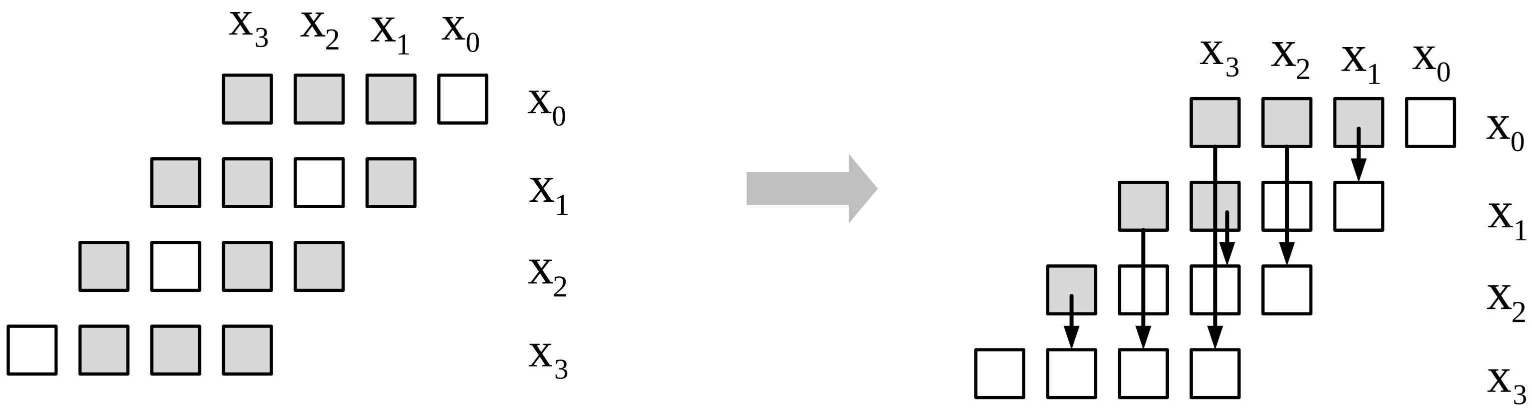
- ❑ x^2 can be done with about half the hardware of a full multiply (for a *dedicated* squaring block, of course)



- ❑ Diagonals ($x_0\ x_0$, $x_1\ x_1$, …) can be replaced by the single input bit with no computation for that bit b/c we have x_0 AND $x_{0=}\ x_0$

Squaring

- ❑ Pairs of equivalent bit products (x_1 x_0 and x_0 x_1, …) can be replaced by one bit product shifted over one column



Resource-Shared Complex Multiplication

- ❑ Operands: 4 integer, 12 fraction bits
- ❑ Result: 8 pre-, 24 post-binary-point bits
- ❑ Subject to tight area constraints

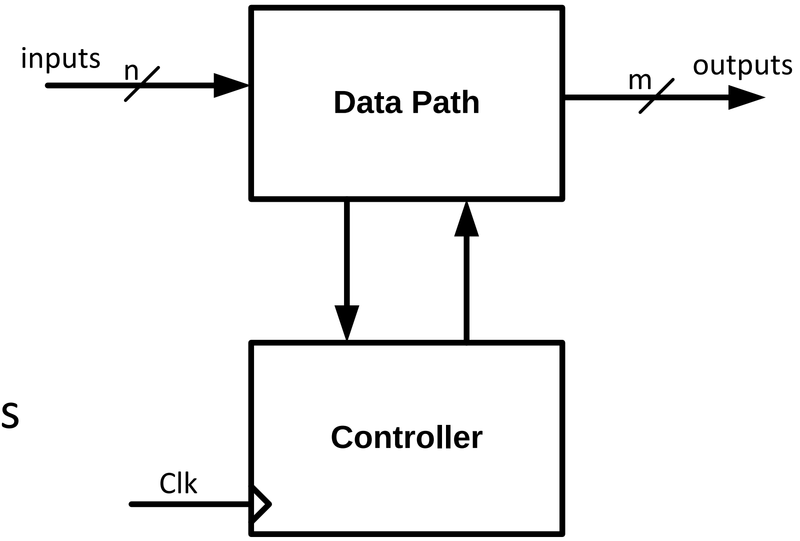
$$a = a_r + ja_i \qquad b = b_r + jb_i$$

$$p = ab = p_r + jp_i = (a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r)$$

- ❑ 4 multiplies, 1 add, 1 subtract
- ❑ Perform sequentially using 1 multiplier, 1 adder/subtracter

Digital Systems

General Architecture:

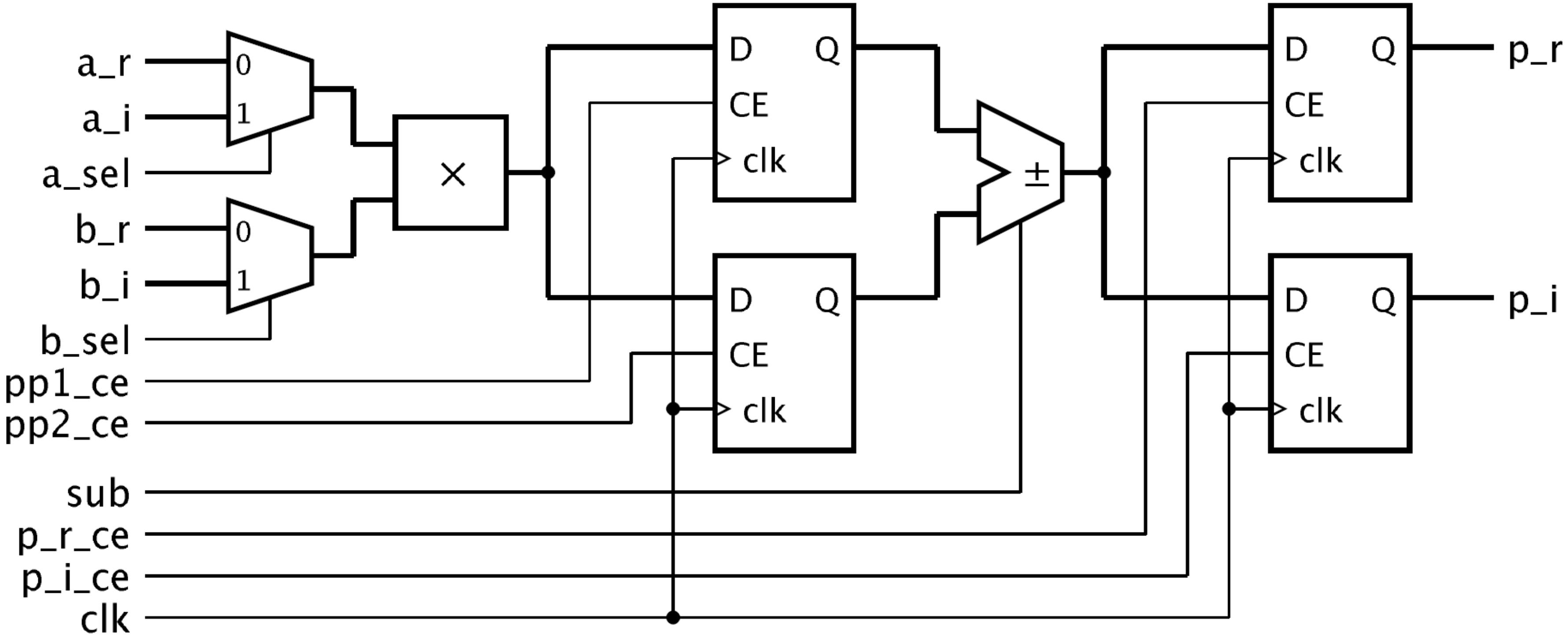


- ❑ **Data Path:**
  - ➢ Transfer input data signals into outputs
  - ➢ Normally combinational logic or counters

- ❑ **Controller:**
  - ➢ Provides any control signal to determine the direction of data flow
  - ➢ Examples: Reset, set, MUX select signals, …
  - ➢ Sequential logic

Resource-Shared Complex Multiplication

❑ Data Path VLSI Architecture:

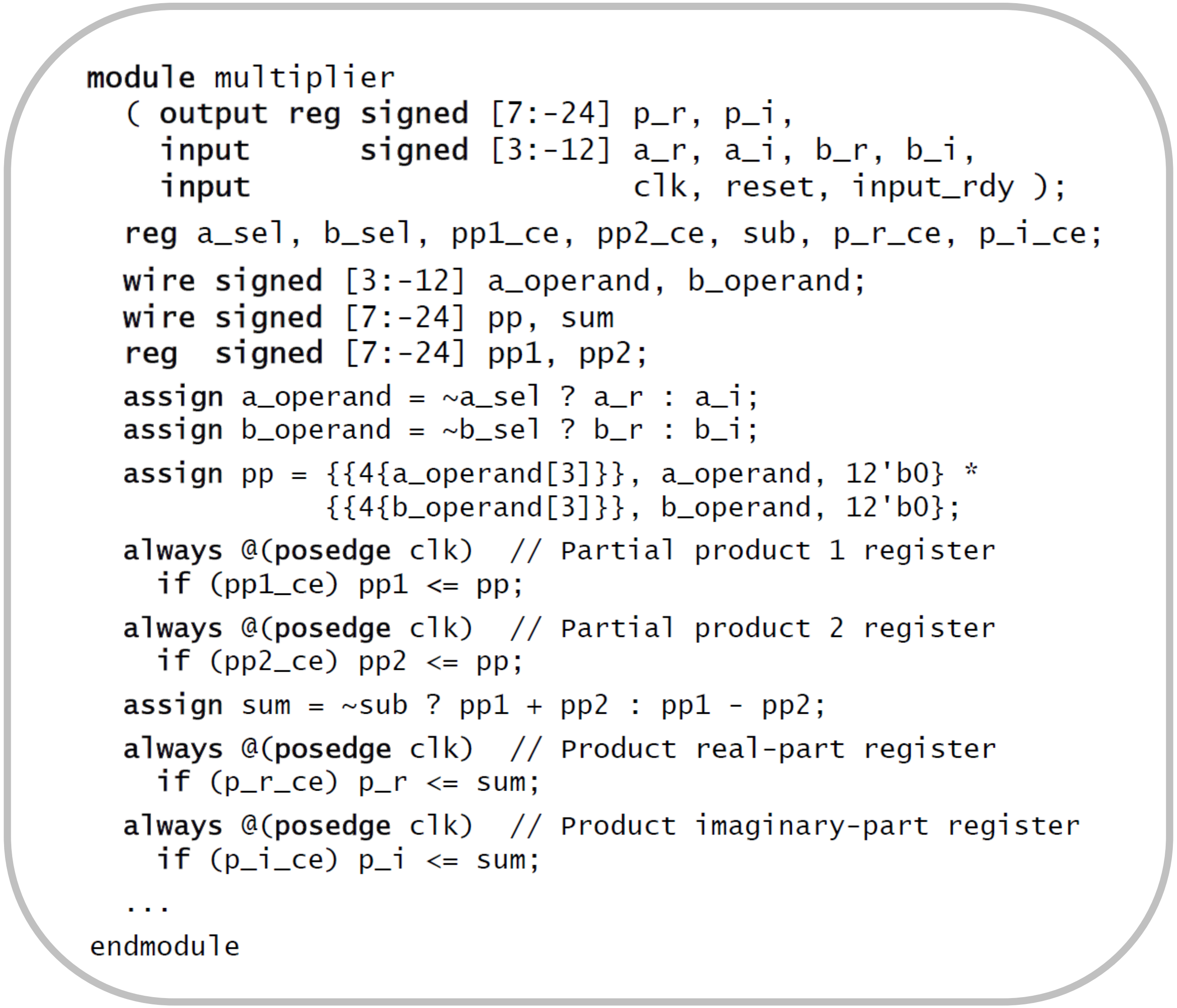


Resource-Shared Complex Multiplication

- ❑ HDL Code:

- ❑ Control Path:
  1. a_r * b_r →pp1_reg
  2. a_i * b_i →pp2_reg
  3. pp1 –pp2 →p_r_reg
     a_r * b_i →pp1_reg
  4. a_i * b_r →pp2_reg
  5. pp1 + pp2 →p_i_reg

- ❑ Takes 5 clock cycles



```
module multiplier
  ( output reg signed [7:-24] p_r, p_i,
    input      signed [3:-12] a_r, a_i, b_r, b_i,
    input                     clk, reset, input_rdy );
  reg a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce;
  wire signed [3:-12] a_operand, b_operand;
  wire signed [7:-24] pp, sum
  reg  signed [7:-24] pp1, pp2;
  assign a_operand = ~a_sel ? a_r : a_i;
  assign b_operand = ~b_sel ? b_r : b_i;
  assign pp = {{4{a_operand[3]}}, a_operand, 12'b0} *
              {{4{b_operand[3]}}, b_operand, 12'b0};
  always @(posedge clk)  // Partial product 1 register
    if (pp1_ce) pp1 <= pp;
  always @(posedge clk)  // Partial product 2 register
    if (pp2_ce) pp2 <= pp;
  assign sum = ~sub ? pp1 + pp2 : pp1 - pp2;
  always @(posedge clk)  // Product real-part register
    if (p_r_ce) p_r <= sum;
  always @(posedge clk)  // Product imaginary-part register
    if (p_i_ce) p_i <= sum;
  ...
endmodule
```

Resource-Shared Complex Multiplication

❑ Control Logic (Timing Schedule):

Step	a_sel	b_sel	pp1_ce	pp2_ce	sub	p_r_ce	p_i_ce
1	0	0	1	0	–	0	0
2	1	1	0	1	–	0	0
3	0	1	1	0	1	1	0
4	1	0	0	1	–	0	0
5	–	–	0	0	0	0	1

Resource-Shared Complex Multiplication

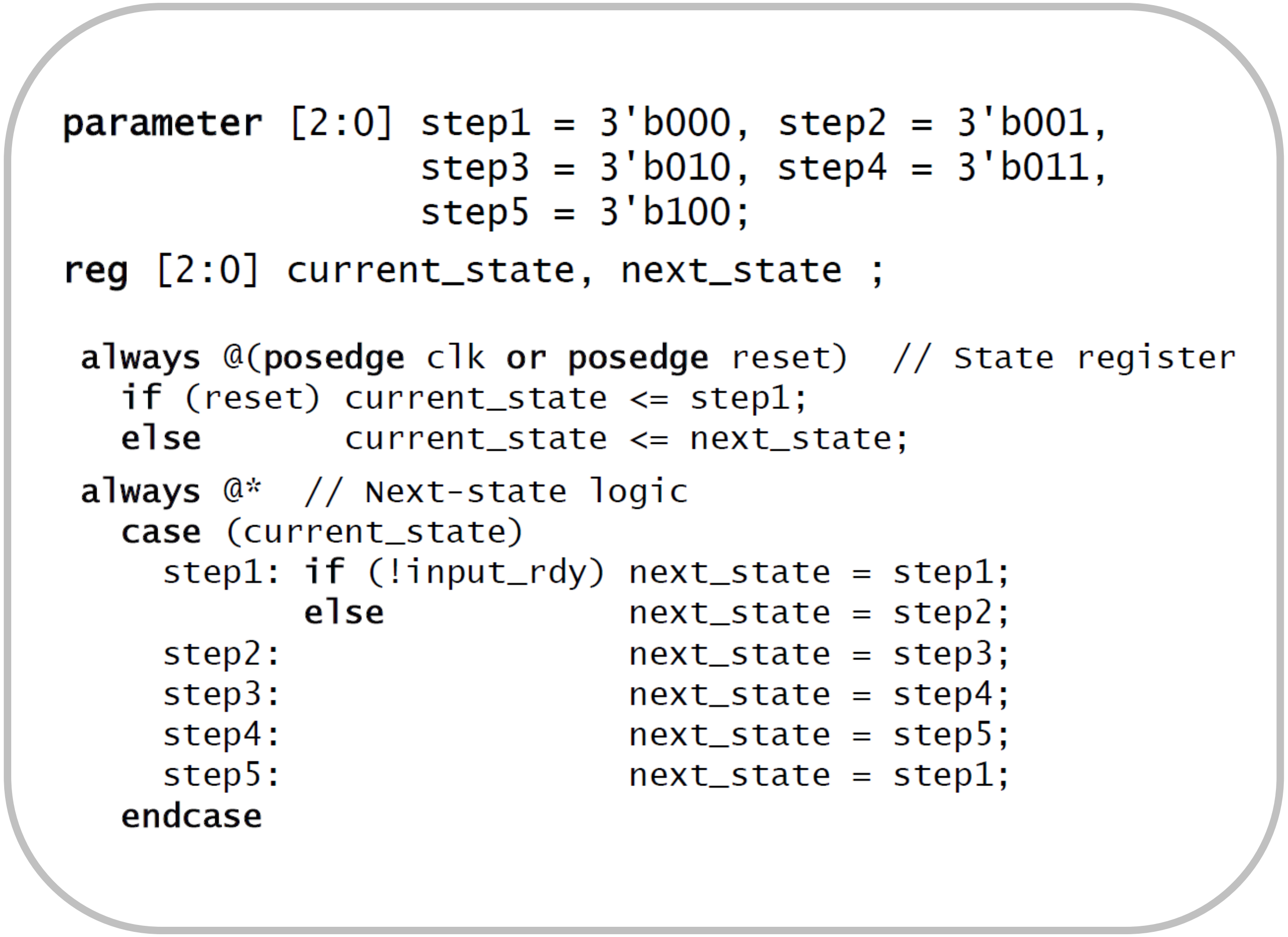
- Control Logic Design:

- One state per step
- Separate idle state?
  - Wait for input_rdy = 1
  - Then proceed to steps 1, 2, ...
  - But this wastes a cycle!
- Use step 1 as idle state
  - Repeat step 1 if input_rdy ≠ 1
  - Proceed to step 2 otherwise

current_ state	input_ rdy	next_ state
step1	0	step1
step1	1	step2
step2	–	step3
step3	–	step4
step4	–	step5
step5	–	step1

Resource-Shared Complex Multiplication

❑ Control Logic Design:



```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
reg [2:0] current_state, next_state ;

 always @(posedge clk or posedge reset)  // State register
   if (reset) current_state <= step1;
   else       current_state <= next_state;
 always @*  // Next-state logic
   case (current_state)
     step1: if (!input_rdy) next_state = step1;
            else            next_state = step2;
     step2:                 next_state = step3;
     step3:                 next_state = step4;
     step4:                 next_state = step5;
     step5:                 next_state = step1;
   endcase
```

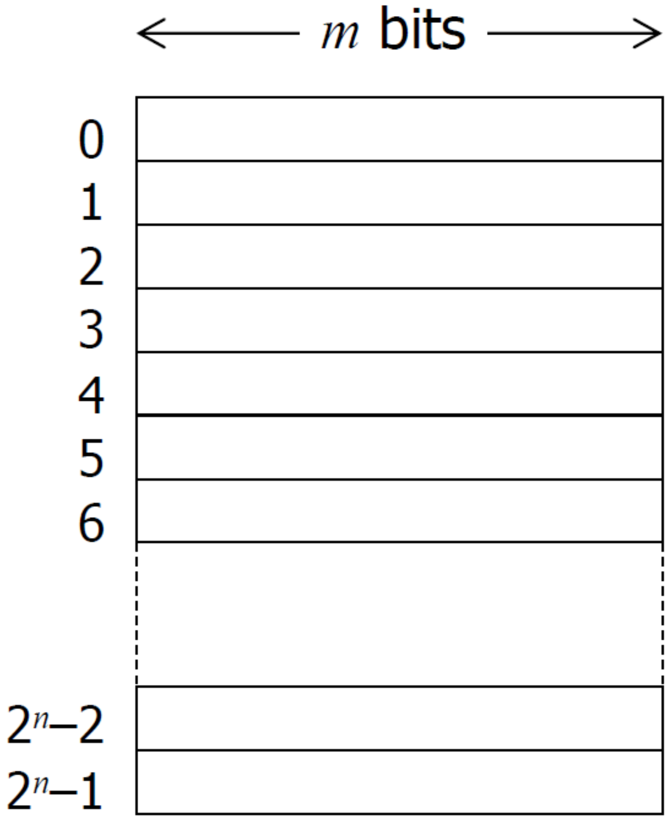
Resource-Shared Complex Multiplication

❑ Control Logic Design:

```
always @* begin  // Output_logic
  a_sel = 1'b0; b_sel = 1'b0;  pp1_ce = 1'b0; pp2_ce = 1'b0;
  sub = 1'b0;   p_r_ce = 1'b0; p_i_ce = 1'b0;
  case (current_state)
    step1: begin
             pp1_ce = 1'b1;
           end
    step2: begin
             a_sel = 1'b1; b_sel = 1'b1; pp2_ce = 1'b1;
           end
    step3: begin
             b_sel = 1'b1; pp1_ce = 1'b1;
             sub = 1'b1;   p_r_ce = 1'b1;
           end
    step4: begin
             a_sel = 1'b1; pp2_ce = 1'b1;
           end
    step5: begin
             p_i_ce = 1'b1;
           end
  endcase
end
```

Memories

- A memory is an array of storage locations
  - Each with a unique address
  - Like a register bank, but with optimized implementation
- Address is unsigned-binary encoded
- n address bits ⇒ 2^n locations
- All locations the same size
- $2^n \times m$ bit memory

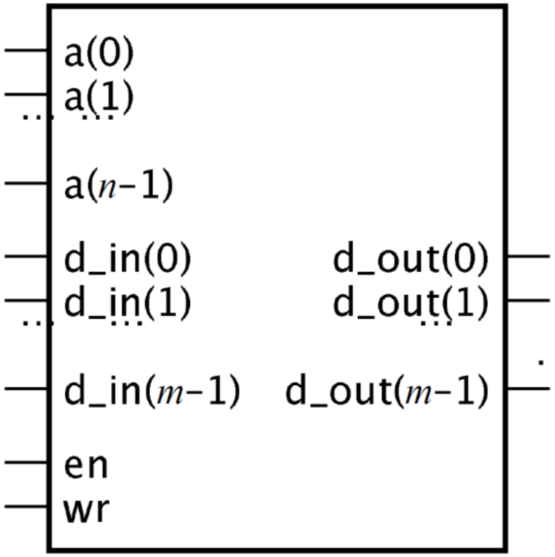


Memory Sizes

- Use power-of-2 multipliers
- Kilo (K): 2^{10}= 1,024 ≈ 10^3
- Mega (M): 2^{20}= 1,048,576 ≈ 10^6
- Giga (G): 2^{30}= 1,073,741,824 ≈ 10^9
- Example:
    - 32K ×32-bit memory
    - Capacity = 1,024K = 1Mbit
    - Requires 15 address bits
- Size is determined by application requirements

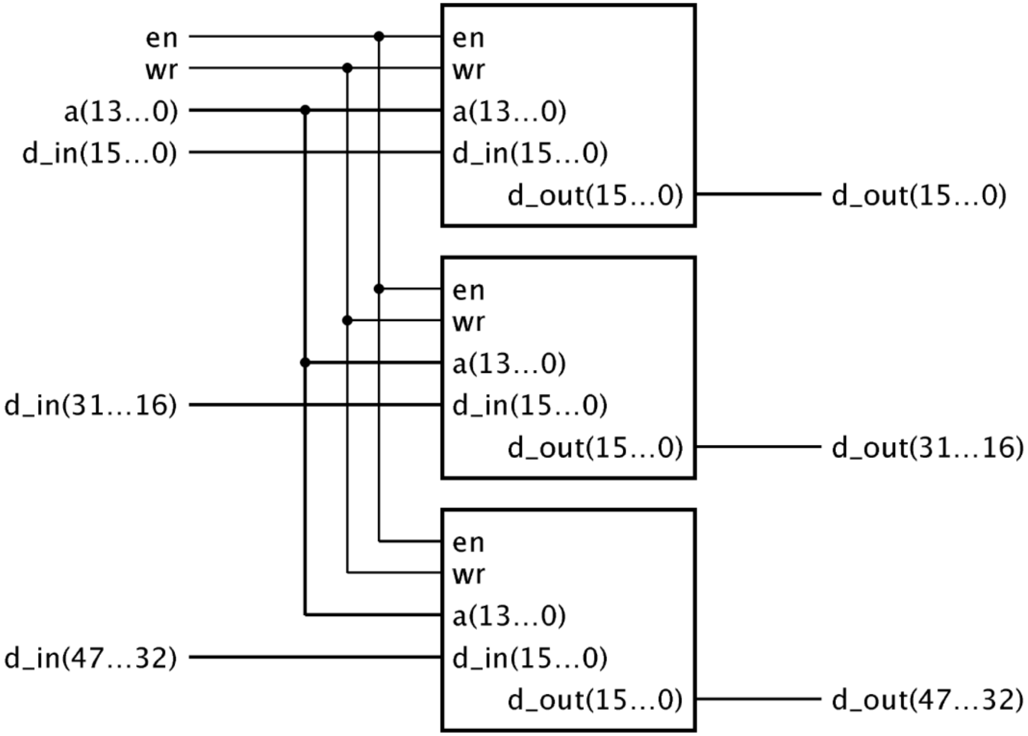
Basic Memory Operations

- ❑ a inputs: unsigned address
- ❑ d_in and d_out
- ❑ Type depends on application
- ❑ Write operation
  - ➢ en = 1, wr = 1
  - ➢ d_in value stored in location given by address inputs
- ❑ Read operation
  - ➢ en = 1, wr = 0
  - ➢ d_out driven with value of location given by address inputs
- ❑ Idle: en = 0



Wider Memories

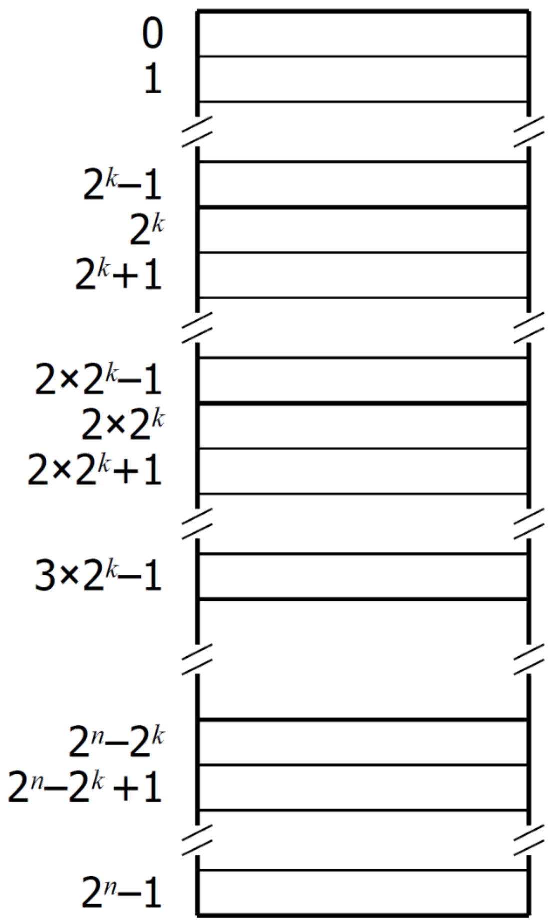
- ❑ Memory components have a fixed width
- ❑ E.g., ×1, ×4, ×8, ×16, ...
- ❑ Use memory components in parallel to make a wider memory
- ❑ E.g, three 16K×16 components for a 16K×48 memory



Larger Memories

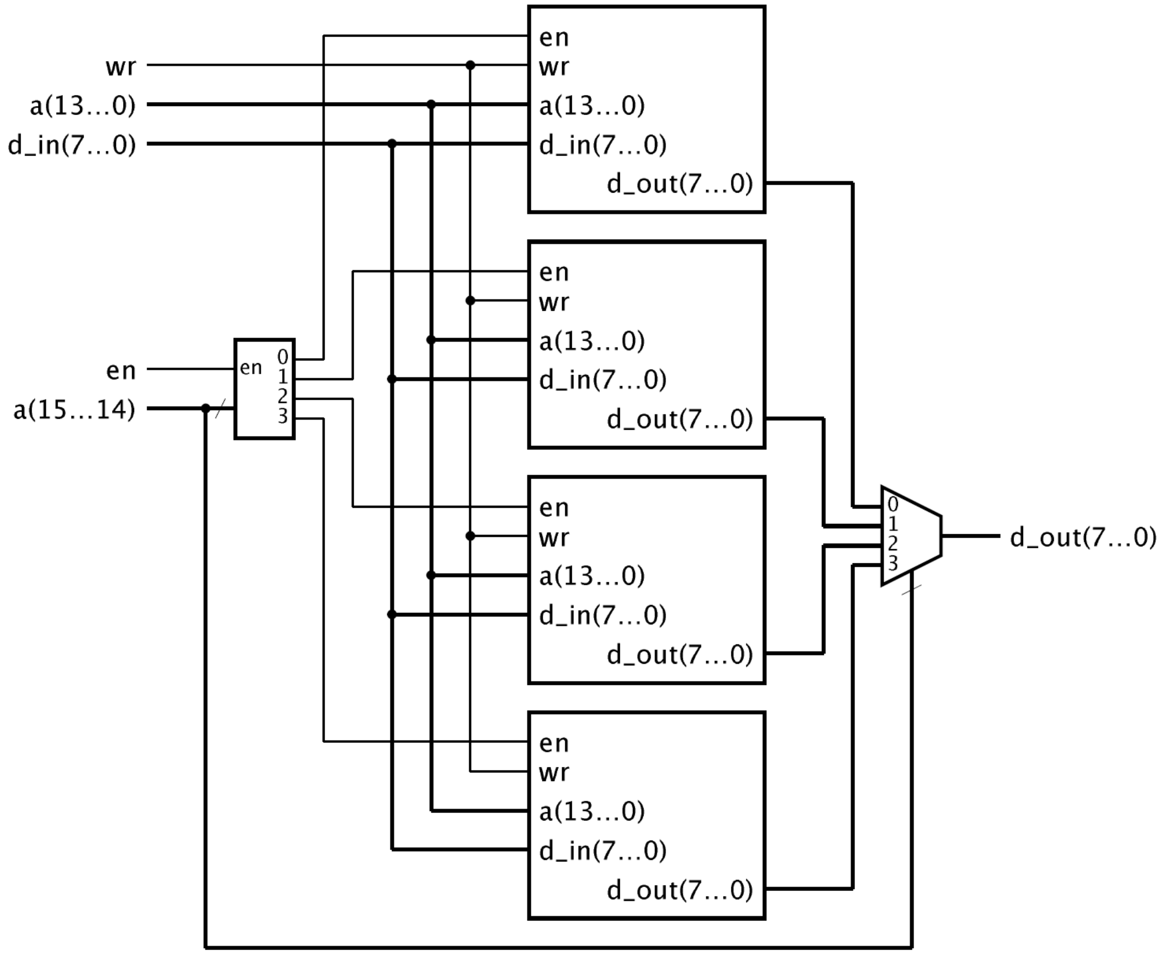
- ❑ To provide 2^n locations with 2^k-location components

- ❑ Use $2^n/2^k$ components



Larger Memories

- ❑ Example: 64K×8 memory composed of 16K×8 components

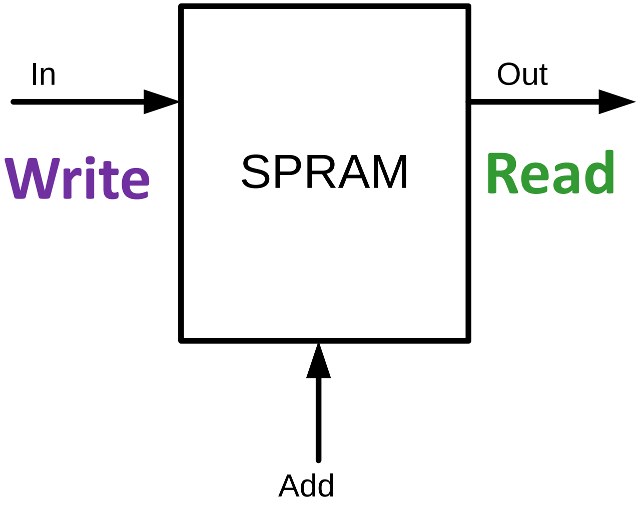


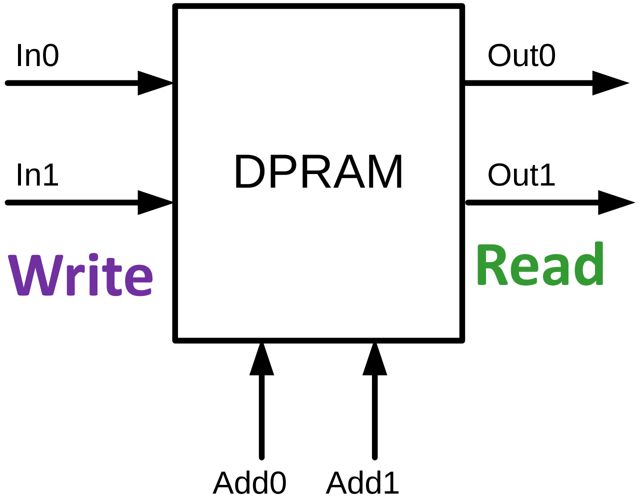
Memory Types

- ❑ Random-Access Memory (RAM)
  - ➢ Can read and write
  - ➢ Static RAM (SRAM)
    - ▪ Stores data so long as power is supplied
    - ▪ Asynchronous SRAM: not clocked
    - ▪ Synchronous SRAM (SSRAM): clocked
  - ➢ Dynamic RAM (DRAM)
    - ▪ Needs to be periodically refreshed
- ❑ Read-Only Memory (ROM)
  - ➢ Combinational
  - ➢ Programmable and Flash rewritable
- ❑ Volatile and non-volatile

Verilog Memories: Single-Port vs. Dual-Port RAM

- ❑ **Single-port RAM (SPRAM):**
  - ➢ Can only be accessed at one address at one time
  - ➢ Read or Write (not both) one memory cell at a time in each clock cycle
- ❑ **Dual-port RAM (DPRAM):**
  - ➢ Can be accessed at two addresses at one time
  - ➢ Read & Write different memory cells at different addresses simultaneously





Verilog Memories: Single-Port RAM

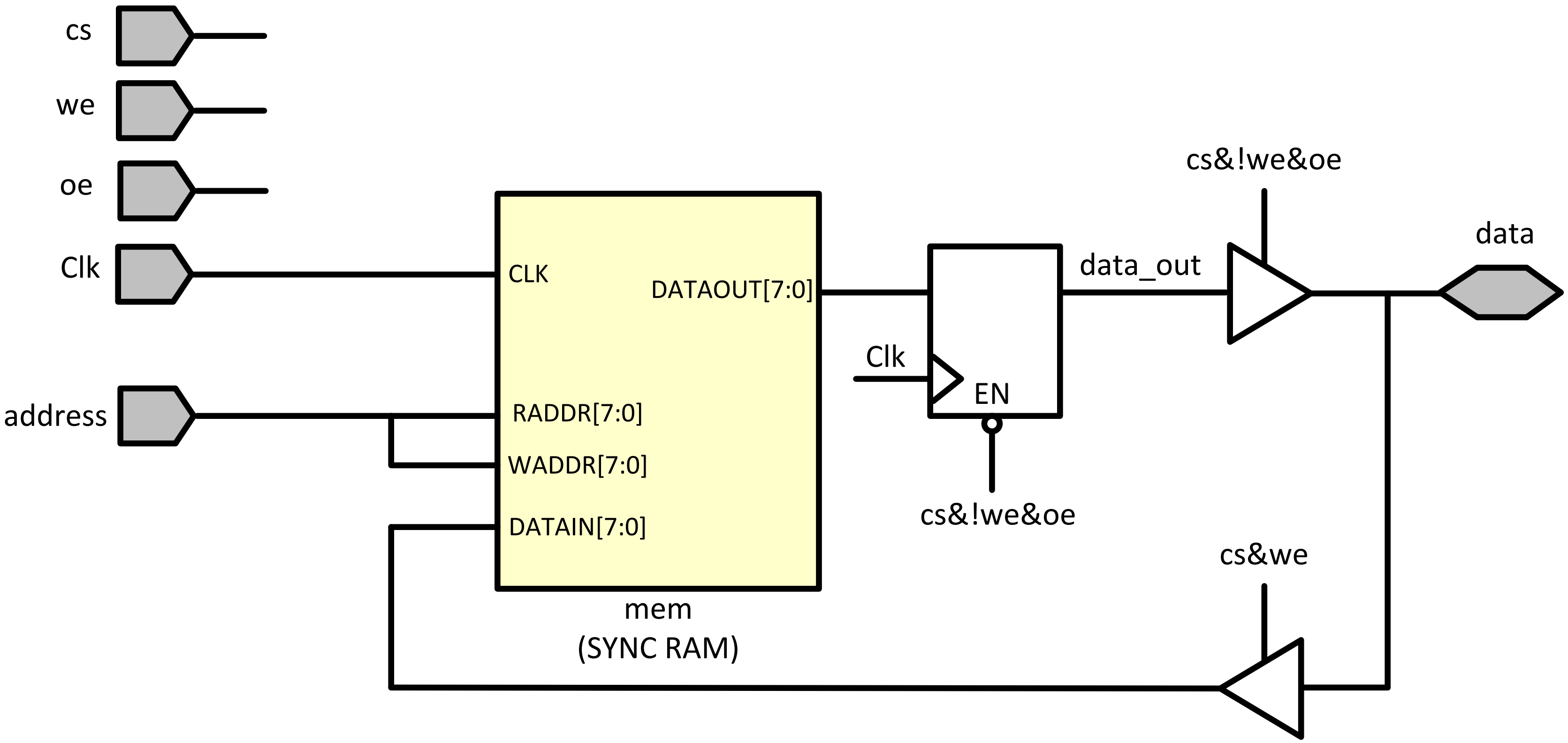
- ❑ 256-Byte SPRAM:
  - ❑ With chip select and read/write enable

```verilog
module SPRAM ( clk , address , data, cs, we , oe);
// cs:chip select, we:Write/Read Enable, oe: Output Enable
parameter DATA_WIDTH = 8 ;
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
input [ADDR_WIDTH-1:0] address ;
input clk, cs, we, oe ;
inout [DATA_WIDTH-1:0] data ;
reg [DATA_WIDTH-1:0] data_out ;
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
//Tri-state buffer, Output: When we = 0 (read), oe = 1, cs = 1
assign data = (cs && oe && !we) ? data_out : 8'bz;
```

```verilog
// CONTINUED
// Memory Write : when we = 1, cs = 1
always @ (posedge clk)
  begin  : MEM_WRITE_READ
    if ( cs && we )
       mem[address] <= data;
// Memory Read: when we = 0, oe = 1, cs = 1
    else if (cs && !we && oe)
       data_out <= mem[address];
  end
endmodule
```

Verilog Memories: Single-Port RAM

- ❑ 256-Byte SPRAM:
  - ❑ With chip select and read/write enable



Verilog Memories: Dual-Port RAM

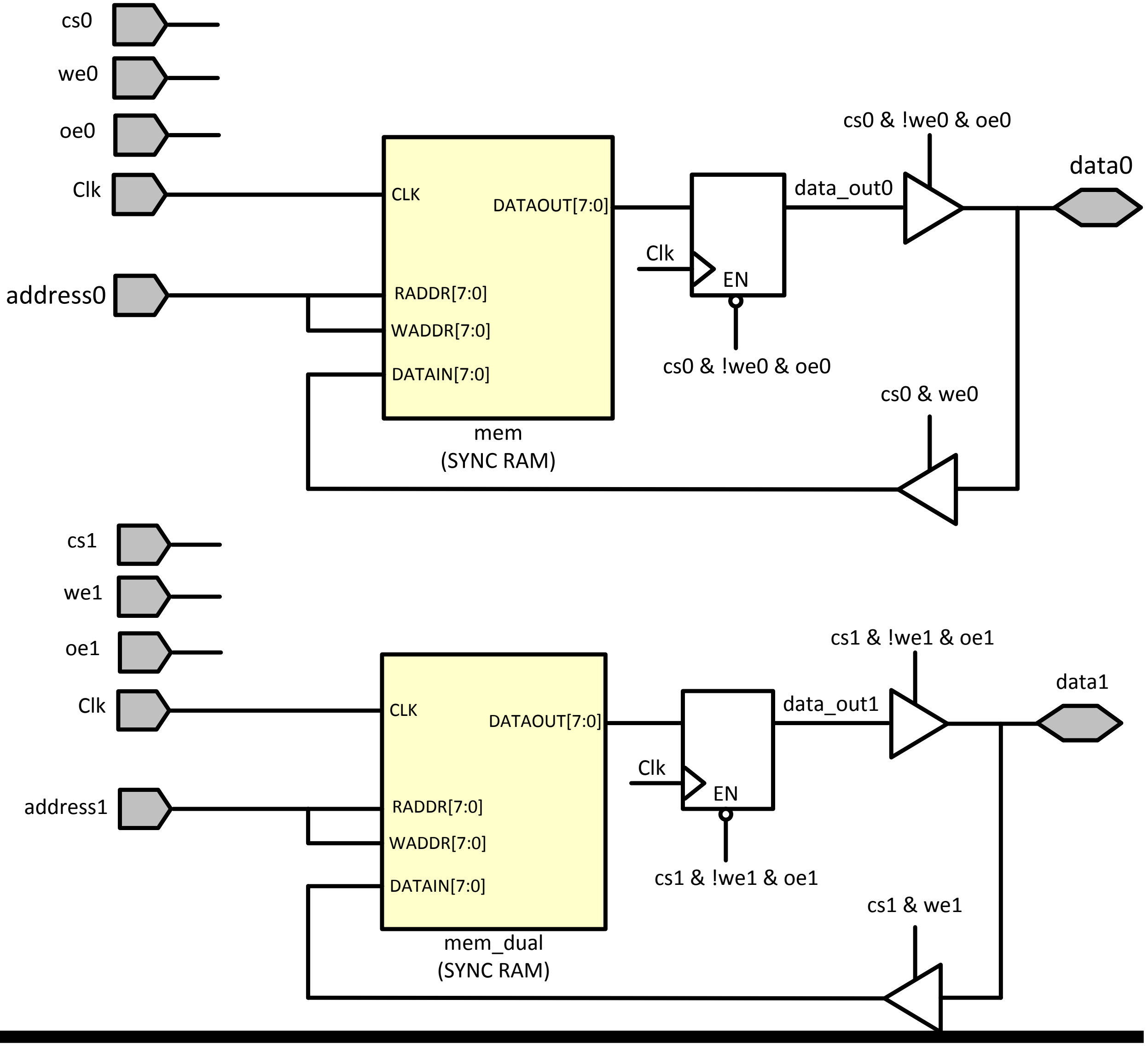
- ❑ 256-Byte DPRAM:
  - ❑ Two separate read/write operations

```
module SPRAM ( clk , address_0 , data_0, cs_0, we_0 , oe_0,
 address_1 , data_1, cs_1, we_1 , oe_1);
// cs:chip select, we:Write/Read Enable, oe: Output Enable
parameter DATA_WIDTH = 8 ;
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
input [ADDR_WIDTH-1:0] address_0, address_1 ;
input clk, cs_0, we_0 , oe_0, cs_1, we_1 , oe_1 ;
inout [DATA_WIDTH-1:0] data_1, data_2 ;
reg [DATA_WIDTH-1:0] data_out _0, data_out _1;
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
//Tri-state buffer, Output: When we = 0, oe = 1, cs = 1
assign data _0= (cs_0 && oe_0 && !we_0) ? data_out _0: 8'bz;
assign data _1= (cs_1 && oe_1&& !we_1) ? data_out _1: 8'bz;
```

```
// CONTINUED
// Memory Write : when we = 1, cs = 1
always @ (posedge clk)
  begin  : MEM_WRITE_READ
   if ( cs_0 && we_0 )
      mem[address_0] <= data_0;
  else if (cs_0 && !we_0 && oe_0)
      data_out_0 <= mem[address_0];
   end
  if ( cs_1 && we_1 )
      mem[address_1] <= data_1;
   else if (cs_1 && !we_1 && oe_1)
      data_out_1 <= mem[address_1];
  end
endmodule
```

Verilog Memories: Dual-Port RAM

- ❑ 256-Byte DPRAM:



Verilog Memories: Dual-Port RAM

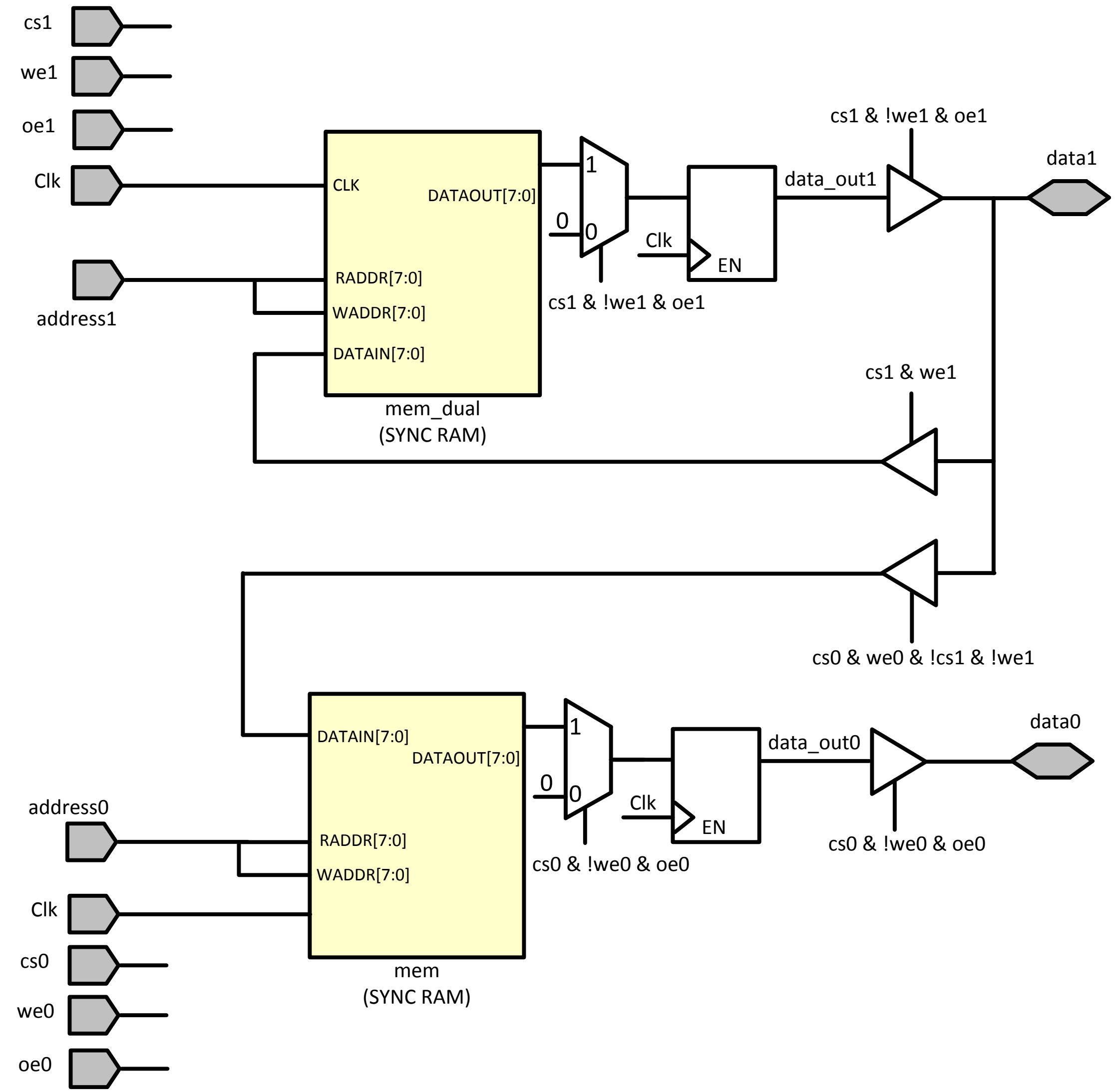
- ❑ 256-Byte DPRAM:
  - ❑ One write two reads simultaneously

```
module SPRAM ( clk , address_0 , data_0, cs_0, we_0 , oe_0,
 address_1 , data_1, cs_1, we_1 , oe_1);
// cs:chip select, we:Write/Read Enable, oe: Output Enable
parameter DATA_WIDTH = 8 ;
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
input [ADDR_WIDTH-1:0] address_0, address_1 ;
input clk, cs_0, we_0 , oe_0, cs_1, we_1 , oe_1 ;
inout [DATA_WIDTH-1:0] data_1, data_2 ;
reg [DATA_WIDTH-1:0] data_out _0, data_out _1;
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
//Tri-state buffer, Output: When we = 0, oe = 1, cs = 1
assign data _0= (cs_0 && oe_0 && !we_0) ? data_out _0: 8'bz;
assign data _1= (cs_1 && oe_1&& !we_1) ? data_out _1: 8'bz;
```

```
// CONTINUED
// Memory Write : when we = 1, cs = 1
always @ (posedge clk)
  begin  : MEM_WRITE_READ
   if ( cs_0 && we_0 )
       mem[address_0] <= data_0;
  else if ( cs_1 && we_1 )
       mem[address_1] <= data_1;
   end
end
always @ (posedge clk)
 if (cs_0 && !we_0 && oe_0)
       data_out_0 <= mem[address_0];
   else
        data_out_0 <= 0;
always @ (posedge clk)
 if (cs_1 && !we_1 && oe_1)
       data_out_1 <= mem[address_1];
   else
        data_out_1 <= 0;
endmodule
```

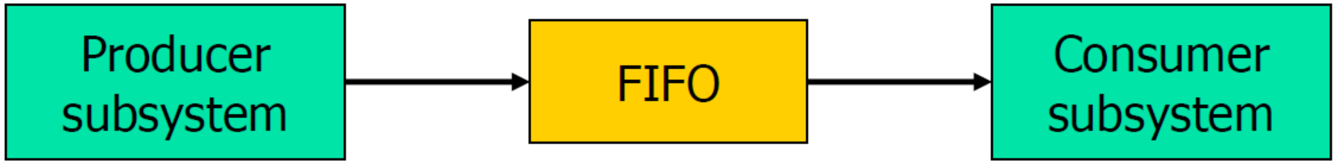
Verilog Memories: Dual-Port RAM

❑ 256-Byte DPRAM:

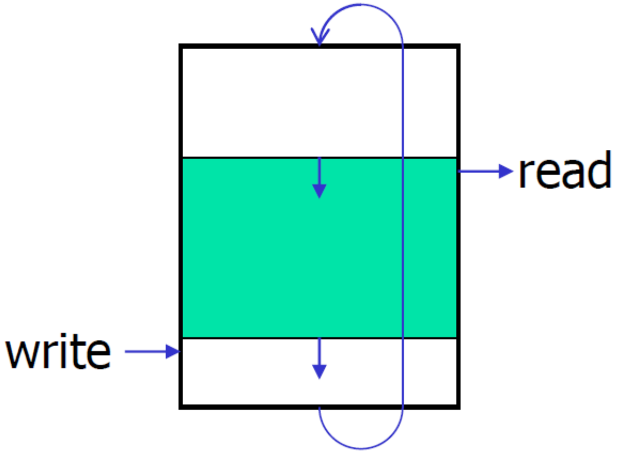


FIFO

- ❑ First-In/First-Out buffer
- ❑ Connecting producer and consumer
- ❑ Decouples rates of production/consumption

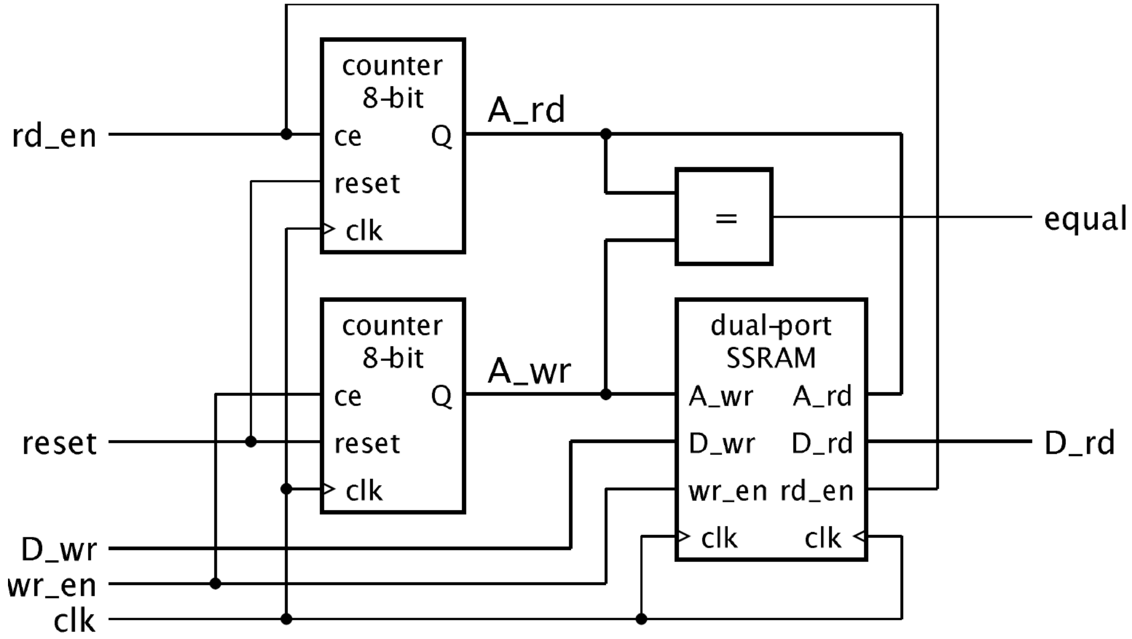


- ❑ Implementation using dual-port RAM
- ❑ Circular buffer
- ❑ Full: write-addr = read-addr
- ❑ Empty: write-addr = read-addr



FIFO Example

- Design a FIFO to store up to 256 data items of 16-bits each, using 256x 16-bit dual-port SSRAM for the data storage. Assume the FIFO will not be read when it is empty, not to be written when it is full, and that the write and read ports share a common clock.



Verilog Memories: ROM

- For constant data, or CPU programs

- Masked ROM
  - Data manufactured into the ROM

- Programmable ROM (PROM)
  - Use a PROM programmer

- Erasable PROM (EPROM)
  - UV erasable
  - Electrically erasable (EEPROM)
  - Flash RAM

Verilog Memories: ROM

- ROM can be realized using two methods:
  - Initialized using a file
  - Initialized explicitly using case statement

```
module ROMFile( address , data , read_en , ce );
 input [7:0] address;
output [7:0] data;
input read_en,  ce;
reg [7:0] mem [0:255] ;
assign data = (ce && read_en) ? mem[address] : 8'b0;
initial
    begin
          $readmemb("memory.list", mem);
          // memory.list is the memory file
     end
endmodule
```

```
module ROMCase( address , data , read_en , ce );
 input [2:0] address;
output reg [7:0] data;
input read_en,  ce;

always @ (ce or read_en or address) begin
          case (address)
             0 : data = 10;
             1 : data = 55;
             2 : data = 244;
             3 : data = 0;
             4 : data = 1;
             5 : data = 8'hff;
             6 : data = 8'h11;
             7 : data = 8'h1;
           endcase
 end
endmodule
```

Function Implementation using look-up tables

- Complex or arbitrary functions are not uncommon
- Example:

$$S(x,y) = \int\int F(\mathcal{X},\mathcal{Y},\mathcal{Z})e^{-j2k(r_i+r_r)}\,d\mathcal{X}\,d\mathcal{Y}$$

- One way is to implement it using Look-up Tables
- Often best to write a Matlab program to write the Verilog table as plain text
- Easy to adapt to other specifications
- Not efficient for very large tables
- Tables with data that is less random will have smaller synthesized area

```
always @ (input) begin
case (input)
  4’b0000: begin real=3’b100; imag=3’b001; end;
  4’b0001: begin real=3’b000; imag=3’b101; end;
  4’b0010: begin real=3’b110; imag=3’b011; end;
                ……
   default: begin real=3’bxxx; imag=3’bxxx; end;
endcase
 end
```

Reviews and Notes

- ❑ Every Verilog statement must end with a semicolon “;”
- ❑ For comparison “==” has to be used not “=”

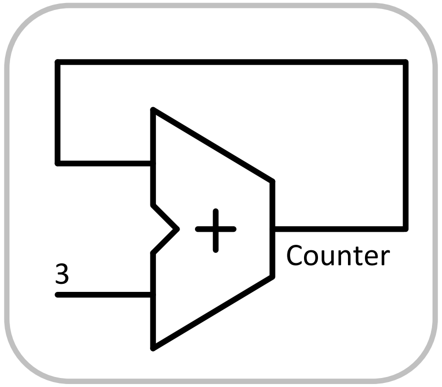
- ❑ When there are multiple assignments to the same variable in an always block, the last statement is evaluated
  - ❖ Example:

```
module DUT(Count );
output reg [2:0] Count;
integer k;

always @ (*)
      begin
            Count <= 0;
            for (k=0; k<4; k=k+1)
                Count <= Count + k;
      end
endmodule
```

⟷

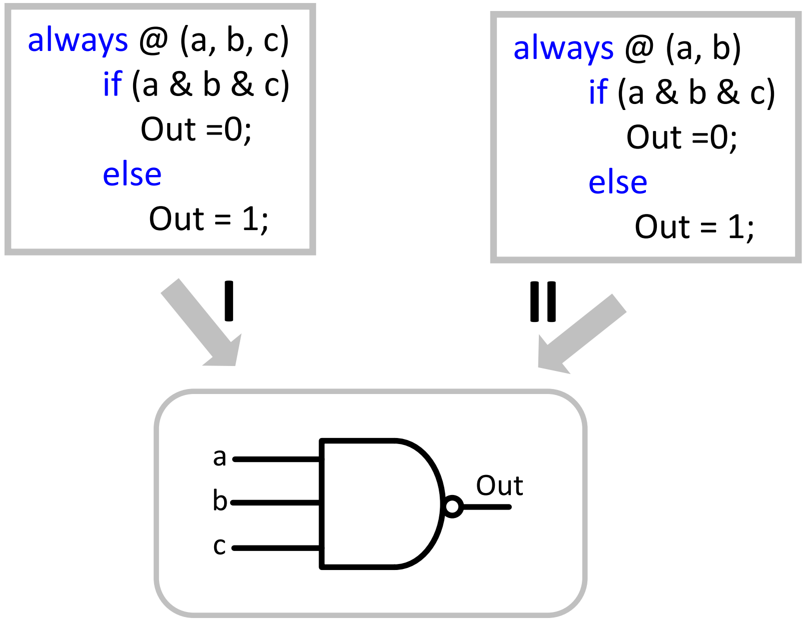
```
module DUT(Count );
output reg [2:0] Count;
integer k;

always @ (*)
        Count <= Count + 3;
endmodule
```

⇒



Reviews and Notes

- ❑ Two codes with different simulation results might have the same synthesized circuit



```
always @ (a, b, c)
        if (a & b & c)
            Out =0;
        else
             Out = 1;
```

```
always @ (a, b)
        if (a & b & c)
            Out =0;
        else
             Out = 1;
```

Same Synthesized Circuit

a	b	c	I	II
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	1

Different Simulation

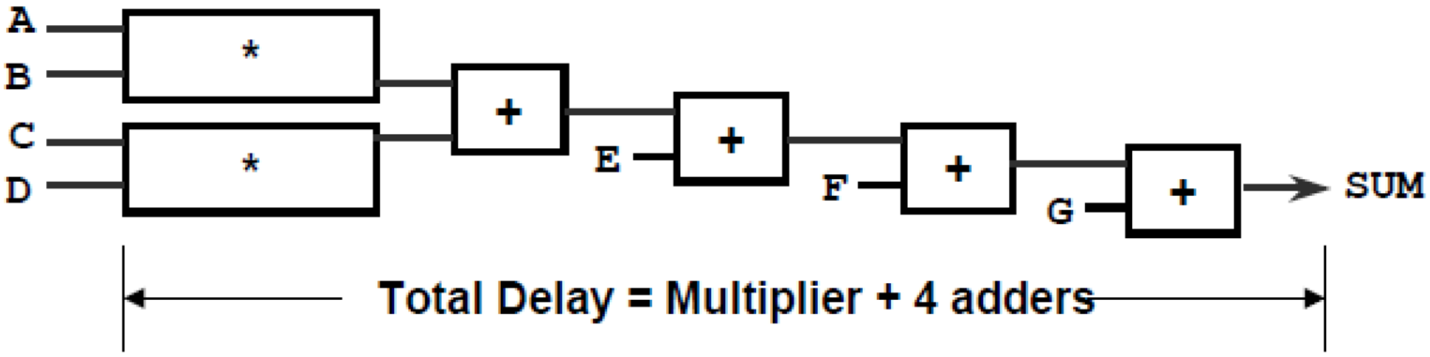
- ❑ Therefore, to avoid mismatch b/w simulation and synthesized version, the sensivity list of always block should include all the signals on the RHS

Coding Styles

- ❑ Do not mix blocking and non-blocking assignments in an always block
- ❑ Use parentheses to optimize logic structure
- ❑ Use meaningful names for signals, variables, and modules
- ❑ Define if-else and case statements explicitly to avoid latch inference
- ❑ Multiple procedural assignments (inside an always block) to a single variable is allowed.
  - ➢ The last assignment is evaluated.
- ❑ Multiple continuous assignments (assign) to a single net in **NOT** allowed.
- ❑ Do not mix edge and level sensitive elements together
- ❑ Use assign statements for simple comb. logic and always block for complex comb. logic
- ❑ Avoid mixing positive-edge and negative-edge triggered flip-flops in one design
  - ➢ Confuses the timing closure

Coding Styles : Parentheses

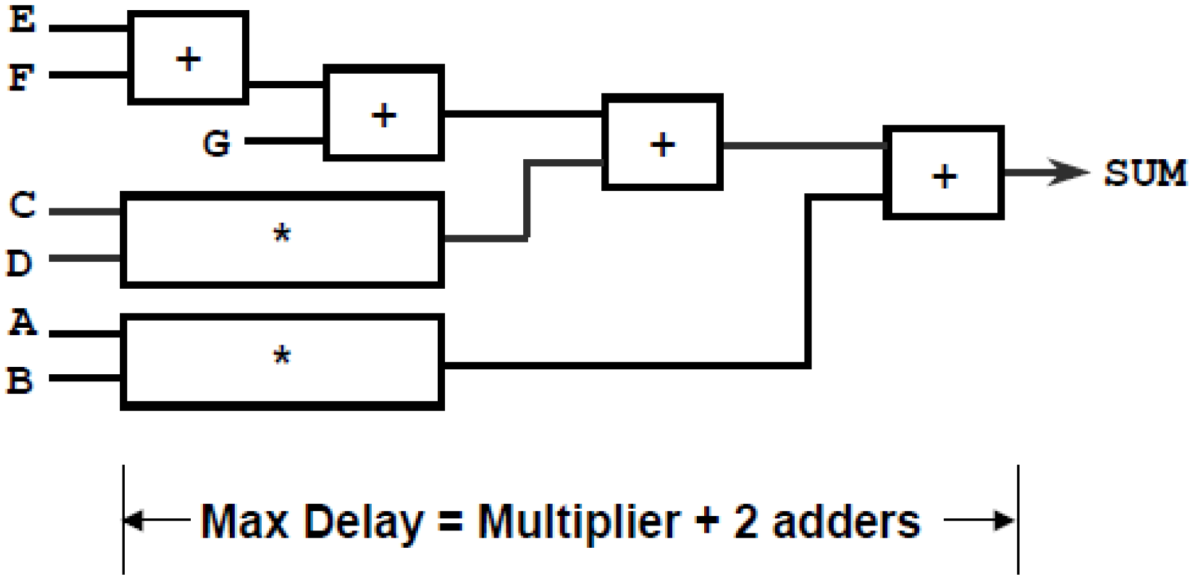
```
SUM <= A*B + C*D + E + F + G
```



Coding Styles : Parentheses

```
SUM <= E + F + G + C*D + A*B
```

```
SUM <= (A*B) + ( (C*D) + ((E+F) + G) )
```



Max Delay = Multiplier + 2 adders

Difference b/w HDL and HLL (1)

- In HLL (high-level language) assignment order is important
- In HDL for "assign" and "non-blocking" assignments, order is NOT important

❖ Example:

HLL: ⇨

```
a = 1; b=0; s=0; na=0; nb=0;
y = na|nb;
nb = b&s;
na = a&~s;
k = a&b;
```

Result: y=0;

```
a = 1; b=0; s=0; na=0; nb=0;
nb = b&s;
na = a&~s;
k = a&b;
y = na|nb;
```

Result: y=1;

HDL: ⇨

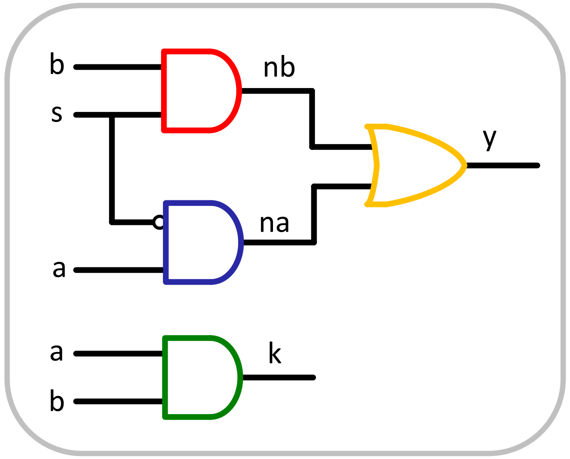
```
wire na, nb;

assign y = na|nb;
assign nb = b&s;
assign na = a&~s;
assign k = a&b;
```

⇔

```
wire na, nb;

assign na = a&~s;
assign k = a&b;
assign nb = b&s;
assign y = na|nb;
```

The same!

⇨



Difference b/w HDL and HLL (2)

- In HLL, multiple assignments to a single signal is allowed
- In HDL , multiple continuous assignments to a signal is NOT allowed.

❖ Example:

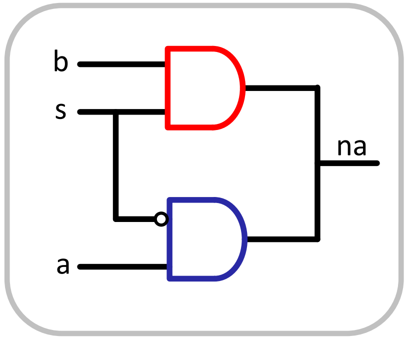
HLL:

```
a = 1; b=0; s=0; na=0; nb=0;
y = na|nb;
na = b&s;
na = a&~s;
```

Result: na = a&~s;

HDL:

```
wire na;

assign y = na|nb;
assign na = b&s;
assign na = a&~s;
```



Illegal

(only used for tri-state implementation)