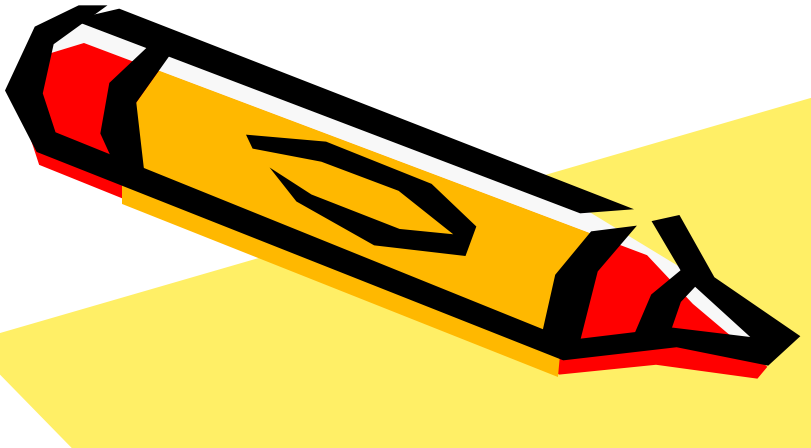
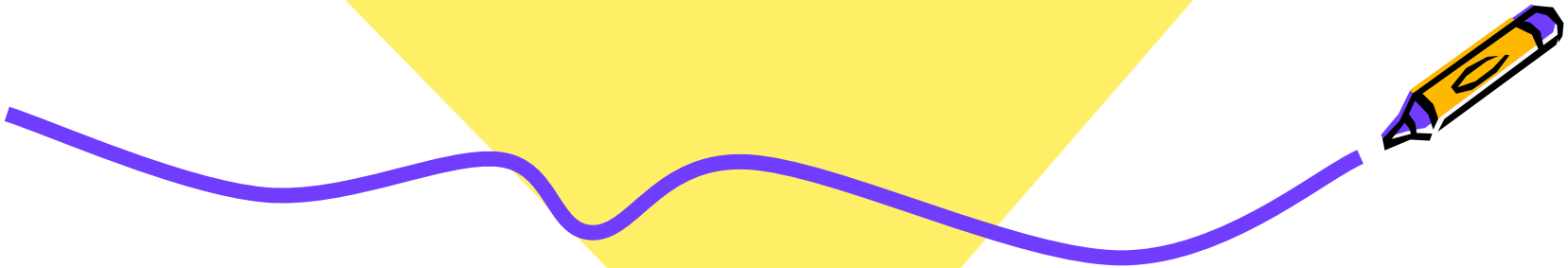


Verilog Basics



台灣IC產業在全球具重要地位

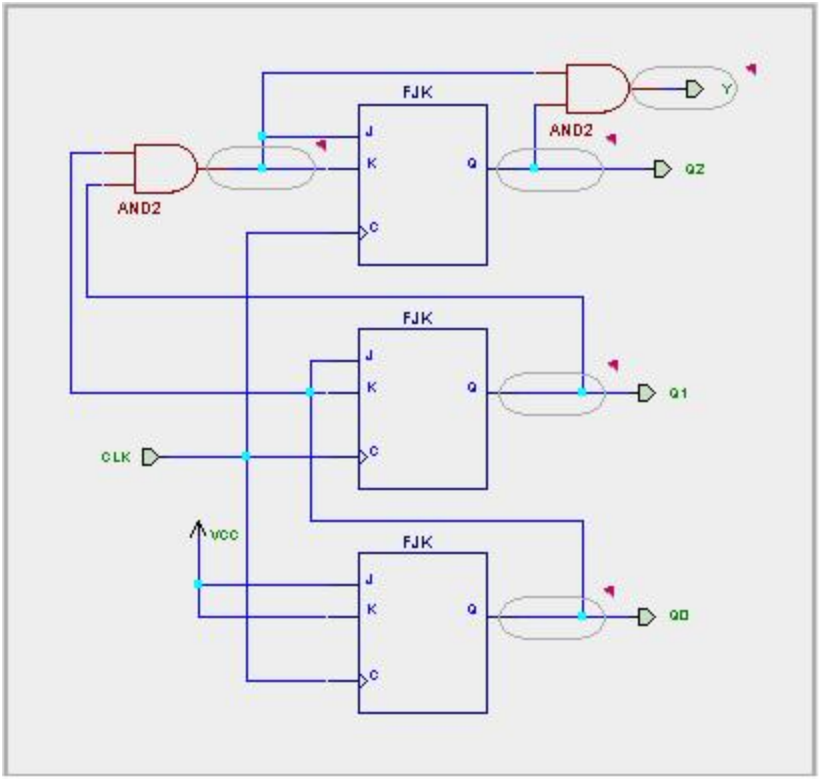
2017年	台灣產值(億美元)	全球產值(億美元)	台灣佔有率(%)	台灣排名	台灣大廠	領先國
IC產業鏈產值=A+B+C+D	810	5,026	16.1%	NO.3	台積電	美國、韓國
A.IC設計	203	976	20.8%	NO.2	聯發科	美國
B.IDM(含記憶體)	53	3,227	1.6%	NO.5	南亞科	美國、韓國、日本、歐洲
C.晶圓代工	397	547	72.5%	NO.1	台積電	台灣
D.IC封測代工	157	281	55.9%	NO.1	日月光	台灣
IC產品產值(IC品牌) =A+B	256	4,203	6.1%	NO.4	聯發科	美國、韓國、日本

- 台灣IC產業上下游產業鏈完整，總IC產值全球排名第3
- 台灣IC設計產值全球排名第2，僅次於美國 (超過中國大陸)。
- 台灣晶圓代工產值全球排名第1，居全球領導地位，先進製程邁入10nm以下。
  台灣IC封測產值全球排名第1，全球前十大專業封測業者中，台灣佔有一半以上。

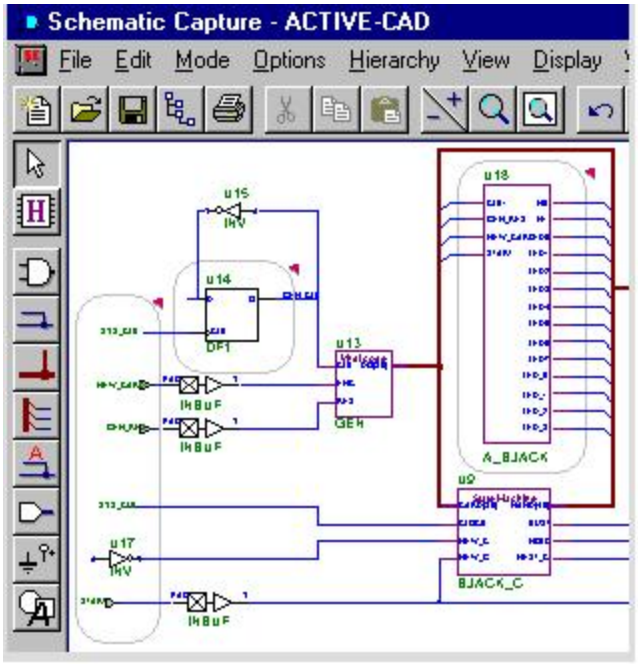
Evolution of Computer-Aided Design

- Earlier digital circuits
  - Vacuum tube→ Transistors -> Integrated Circuits (ICs)
- Integrated Circuits (logic gates were placed on a single chip)
- Integration Level
  - SSI → MSI → LSI →VLSI

Traditional Design approaches

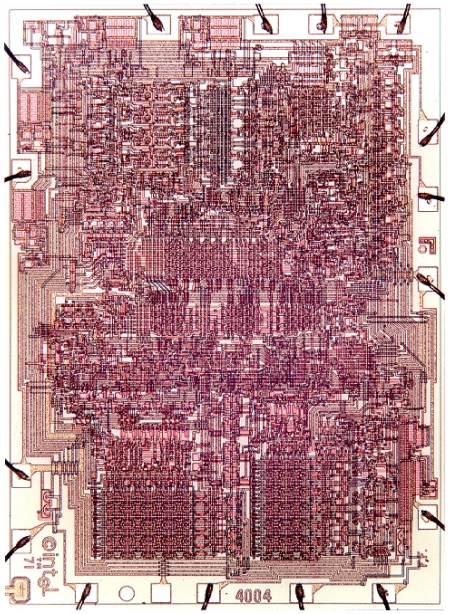


Gate Level Design

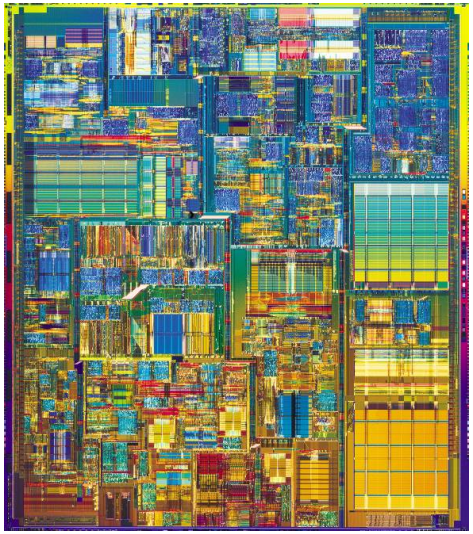


Schematic Design

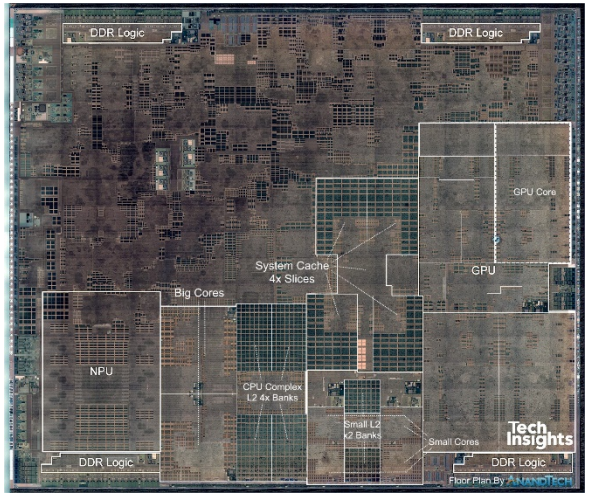
Advancements over the years



- Intel 4004 Processor
- Introduced in 1971
- 2250 Transistors
- 108 KHz Clock



- Intel P4 Processor
- Introduced in 2000
- 40 Million Transistors
- 1.5GHz Clock



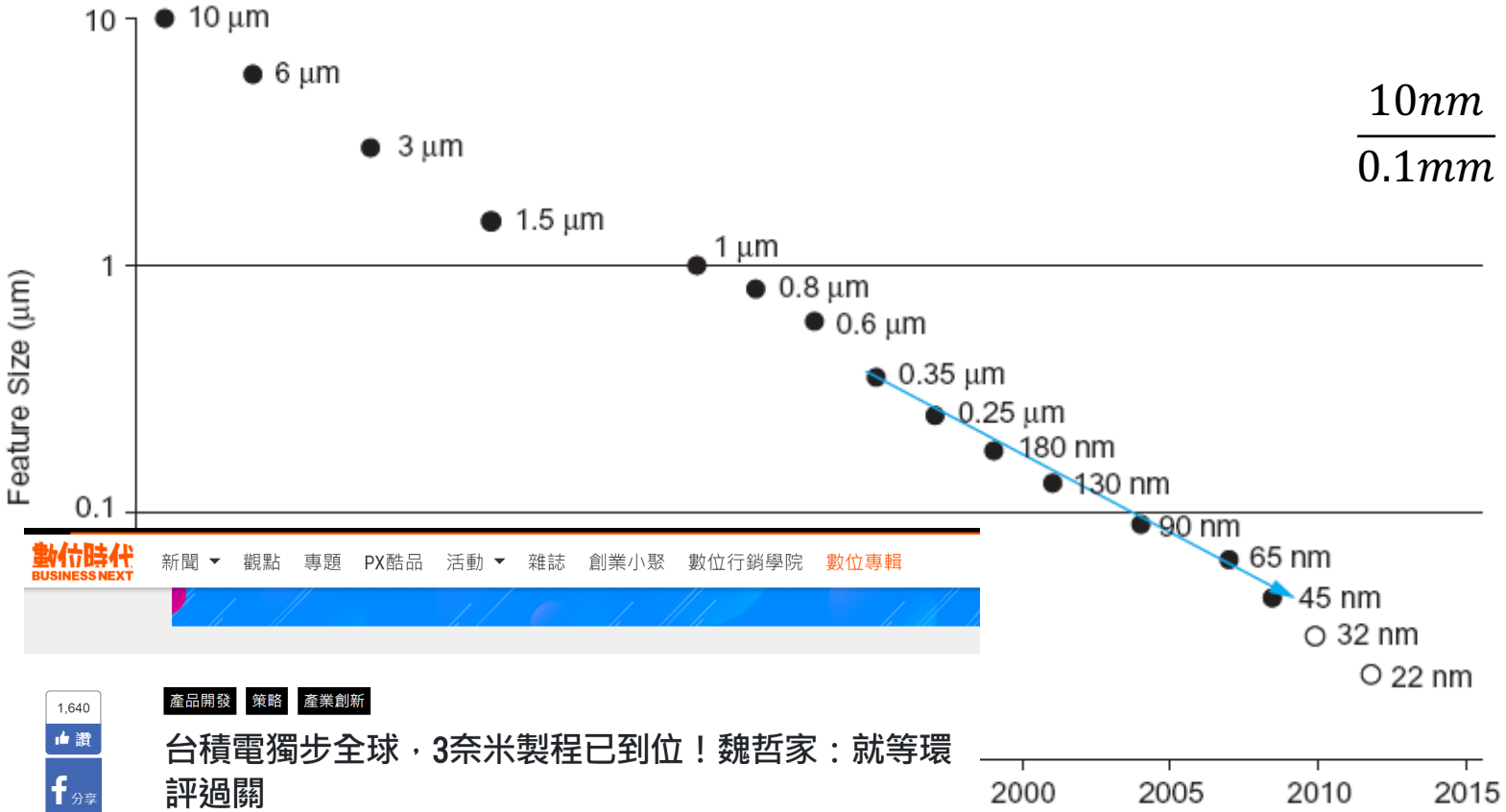
- Apple A12 Processor
- Introduced in 2018
- 6.9 Billion Transistors
- 2.4 GHz Clock

Feature Size

- Minimum feature size shrinking 30% every 2-3 years

Hair diameter = 0.1mm

$$\frac{10nm}{0.1mm} = \frac{1}{10000}$$



數位時代 BUSINESS NEXT 新聞 觀點 專題 PX酷品 活動 雜誌 創業小聚 數位行銷學院 數位專輯

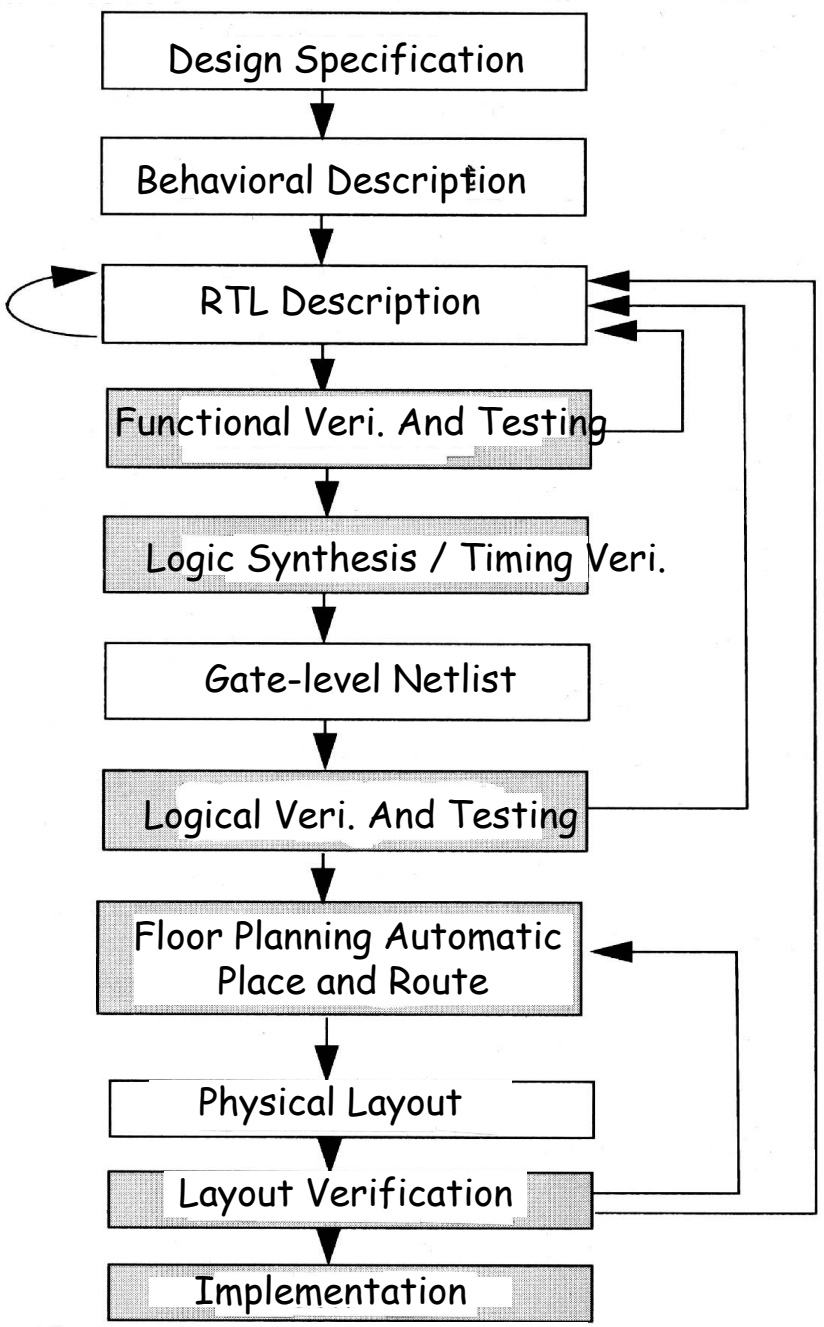
1,640 讚 分享 LINE

產品開發 策略 產業創新

台積電獨步全球，3奈米製程已到位！魏哲家：就等環評過關

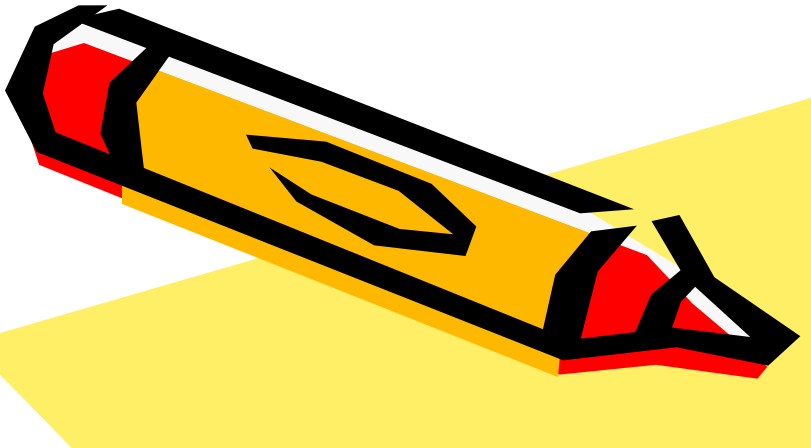
2018.12.06 by 葛翰勳

Typical Design Flow

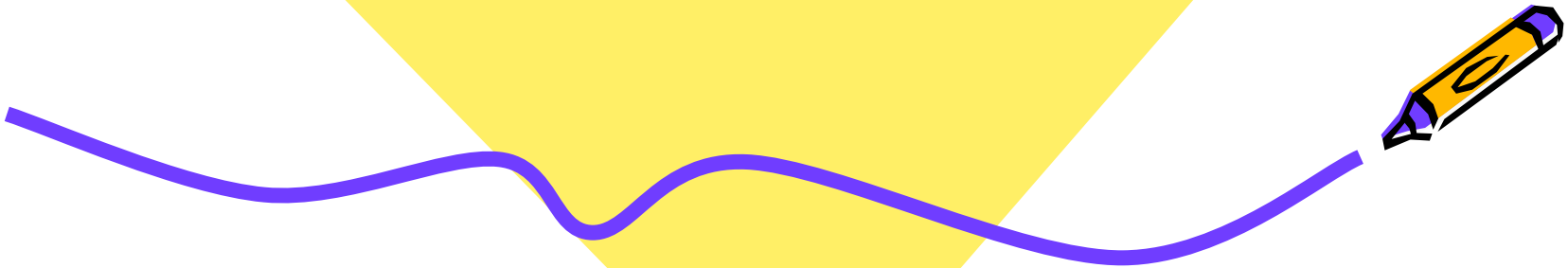


Design Representation

- Different Level of Representation from High Level to Low Level
- Specification
  - Describe the functionality, interface and overall architecture
- RTL(Register-Transfer Level):
  - Describe the data flow that will implement the desired digital circuit
- Gate-level netlist:
  - Describe the circuit in terms of gate and connections between them
- Physical Layout
  - Describe the final representation for manufacturing.

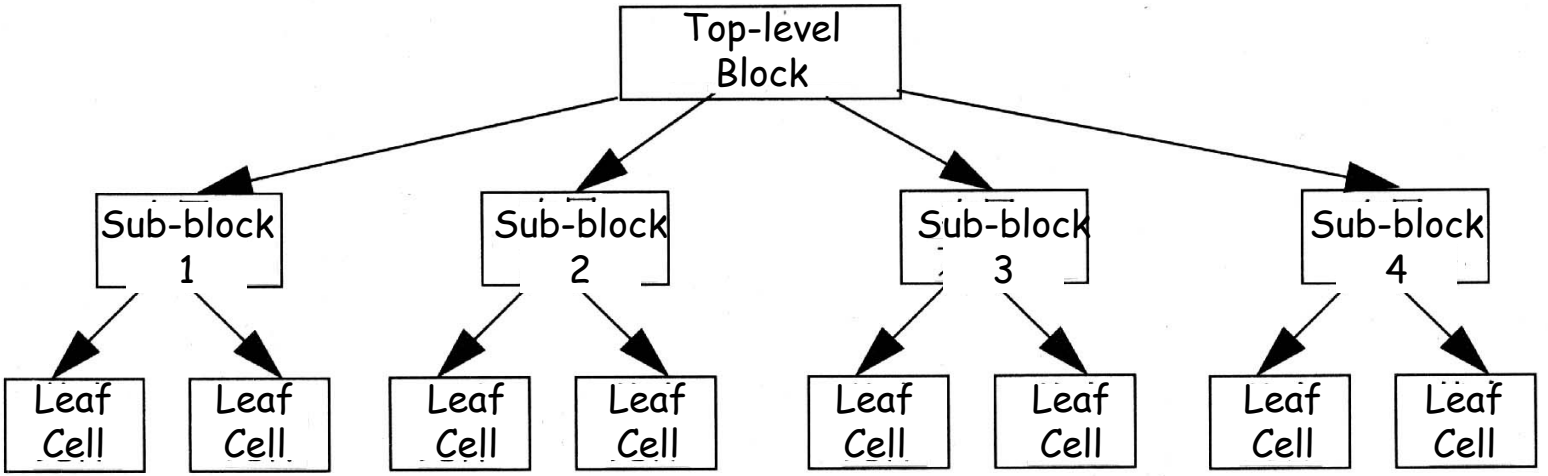


Hierarchical Modeling

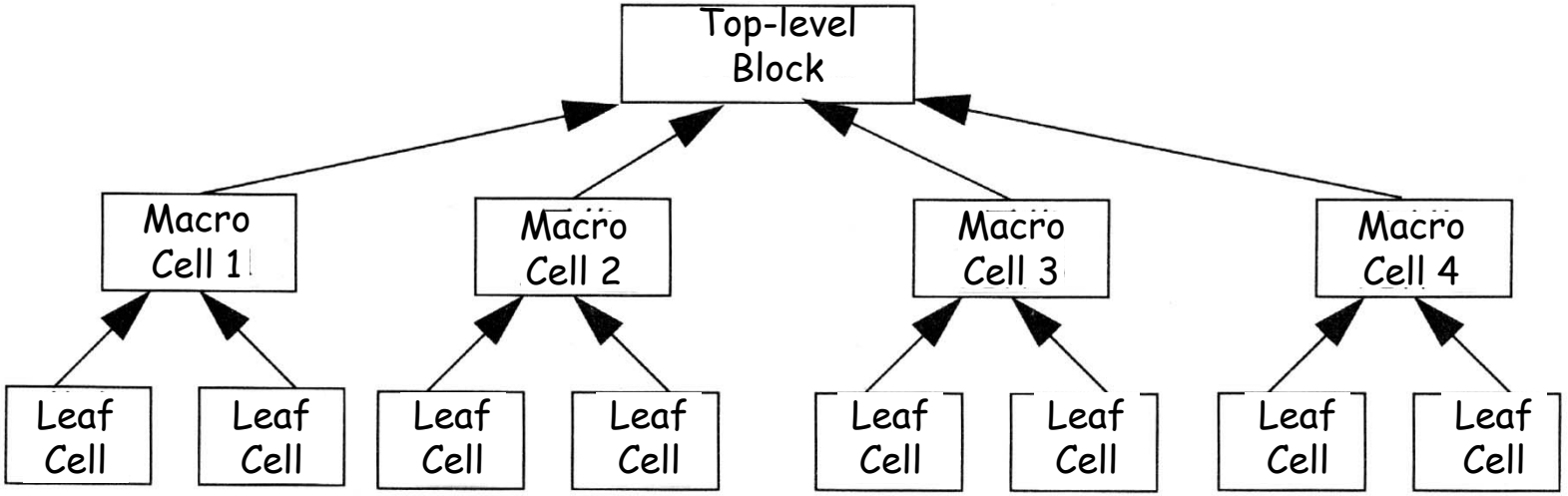


Design Methodology

- Top-down
  - Define the top-level block and identify the sub-blocks
- Bottom-up
  - Identify the building blocks first and combine the blocks to bigger blocks
- A combination of top-down and bottom-up flows is typically used.

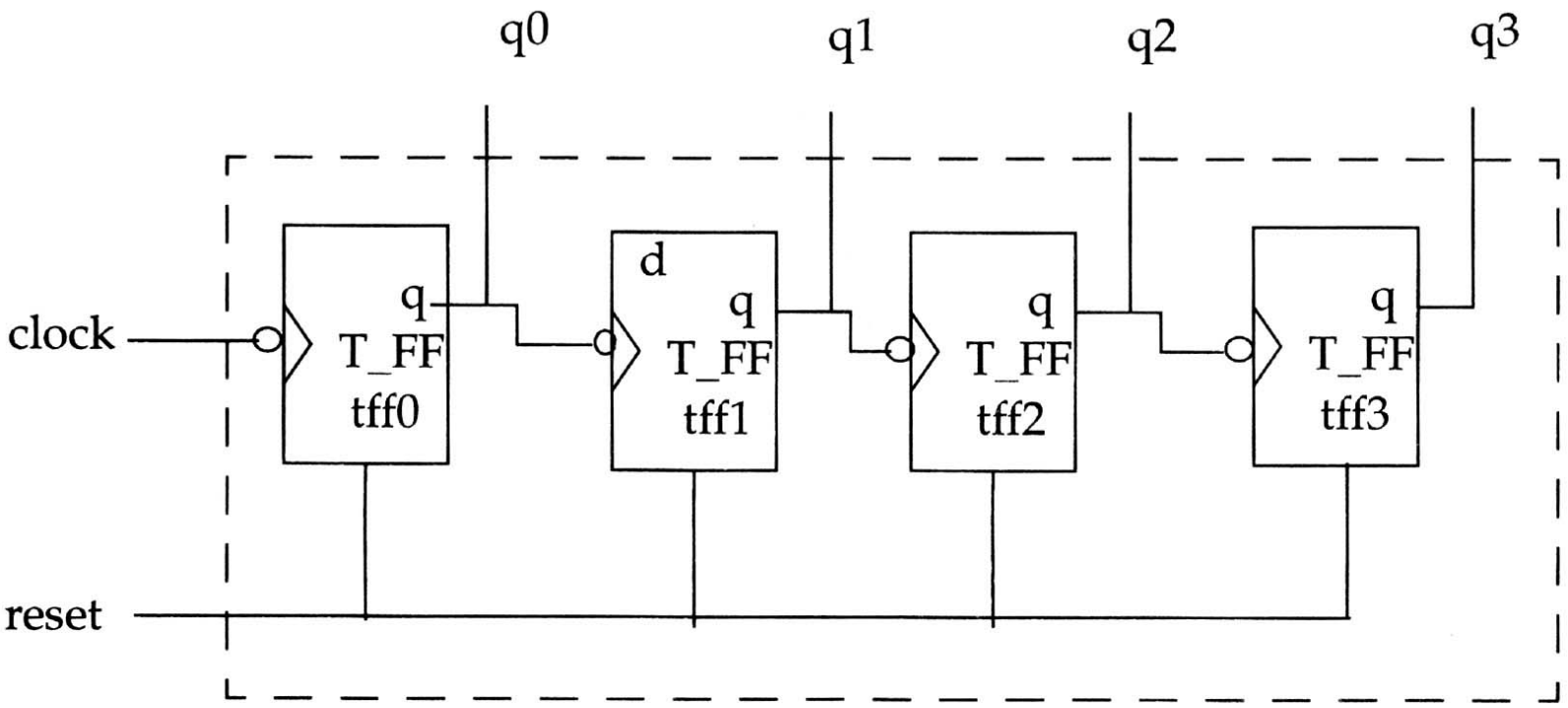


Top-down design Methodology



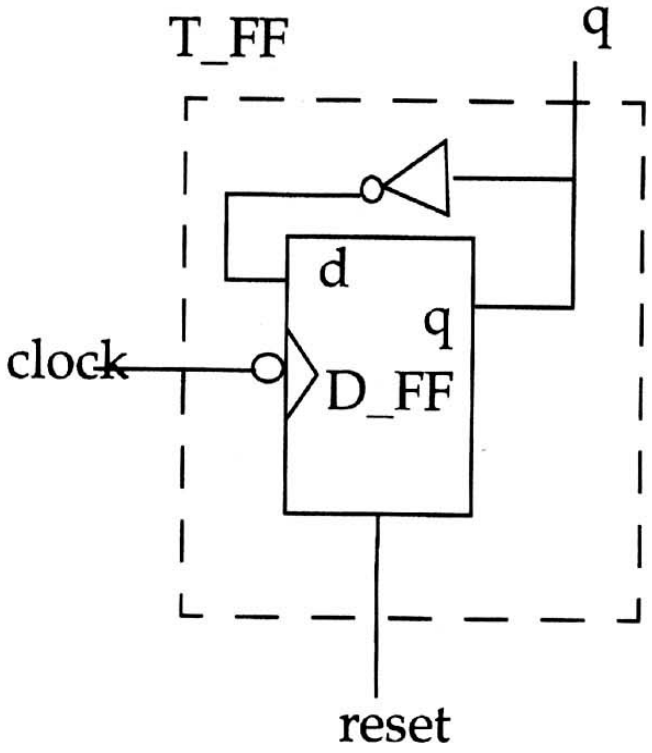
Bottom-up design Methodology

Example: 4-bit Ripple Carry Counter



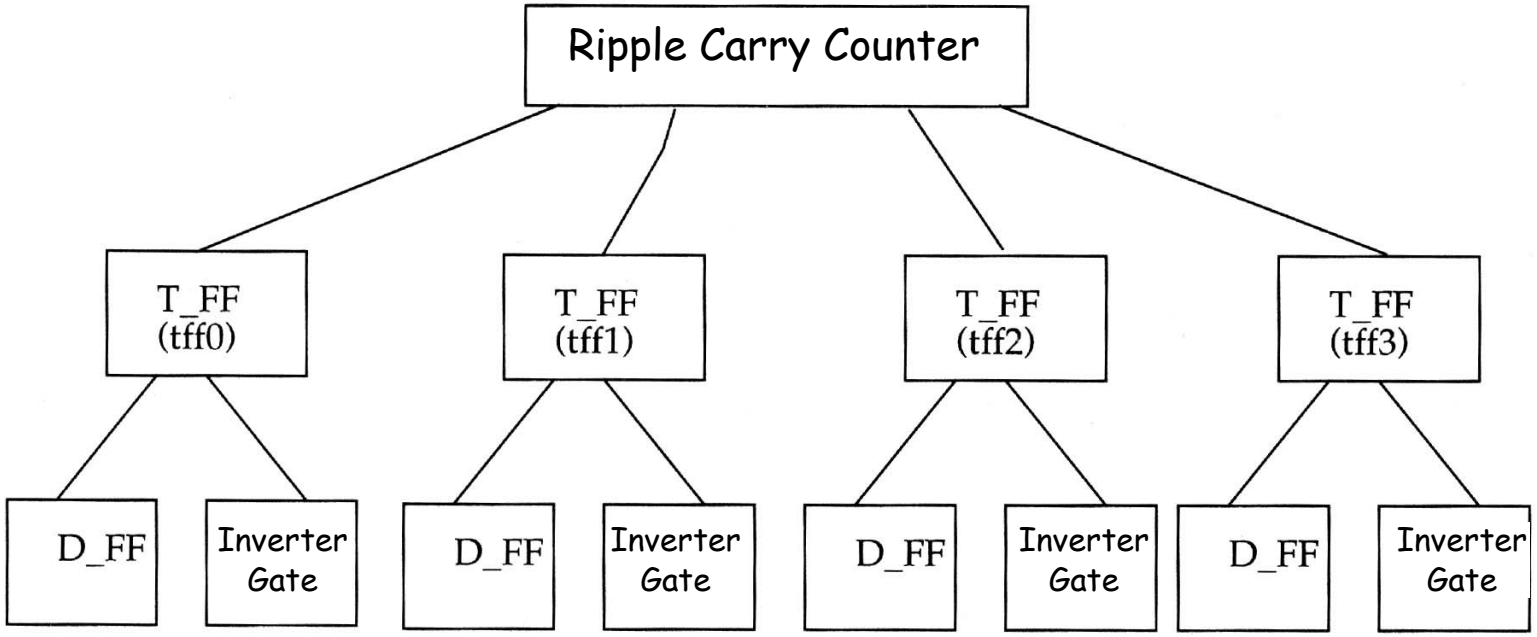
Recall: T Flip-Flop

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



T- Flip Flop

Design Hierarchy



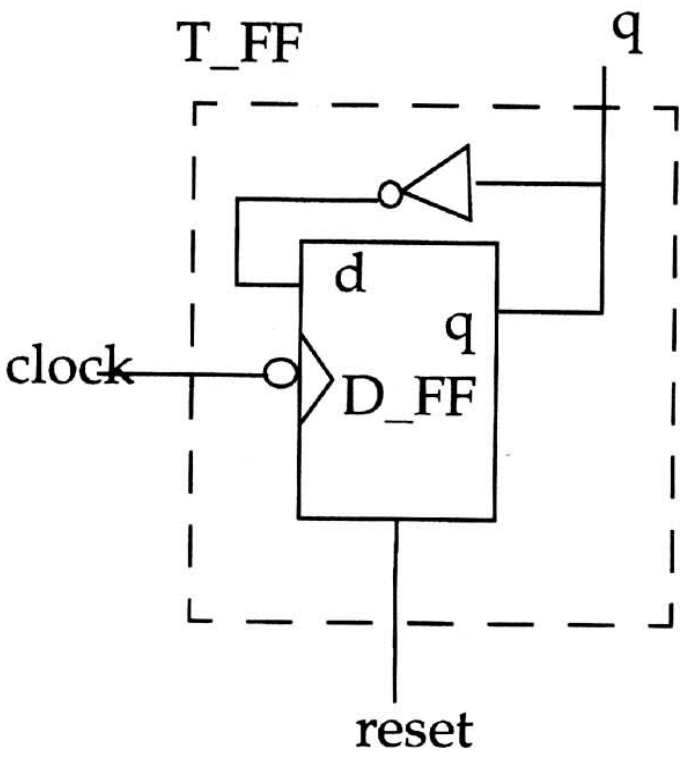
Ripple Carry Counter Design Hierarchy

Module in Verilog

- Design Hierarchy vs. Module in Verilog
- Module is the basic building block
  - An element, or
  - A collection of lower-level blocks
- Input/Output port interface are used for communication
  - Internal implementation is hidden

```
module <module_name>(<module_terminal_list>);
….
<module internals>
…
endmodule
```

```
module T_FF(q, clock, reset);
….
<module internals>
…
endmodule
```



Levels of Abstraction

- Behavioral or algorithmic level
  - Similar to C programming
- Dataflow level
  - Focus on how the data is processed
- Gate level
  - Gate and interconnect
- Switch level
  - Switch, storage node and interconnect

Register Transfer Level (RTL) = mixed description of behavioral and dataflow constructs that is synthesizable

```
module behave;
reg [1:0]a,b;

initial
begin
  a = 'b1;
  b = 'b0;
end

always
begin
  #50 a = ~a;
end

always
begin
  #100 b = ~b;
end
End module
```

```
module orgate(out, a, b, c, d);
input a, b, c, d;
wire x, y;
output out;
or or1(x, a, b);
or or2(y, c, d);
or orfinal(out, x, y);
endmodule
```

```
// Dataflow description of 2-to-4 line decoder with enable input (E)
module decoder_df (A,B,E,D);
    input A,B,E;
    output [3,:0] D;

        assign D[3] =~(~A & ~B & ~E);
        assign D[2] =~(~A &  B & ~E);
        assign D[1] =~( A & ~B & ~E);
        assign D[0] =~( A &  B & ~E);

endmodule
```

Or

```
assign D[3] =~(~A & ~B & ~E),
       D[2] =~(~A &  B & ~E),
       D[1] =~( A & ~B & ~E),
       D[0] =~( A &  B & ~E);
```

Instantiation and Instances

- Instantiation: the process of creating objects from a module
  - The objects are call instances
- No nesting is allowed

```
module ripple_carry_counter(q, clk, reset);

output [3:0] q;
input clk, reset;

//4 instances of the module TFF are created.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

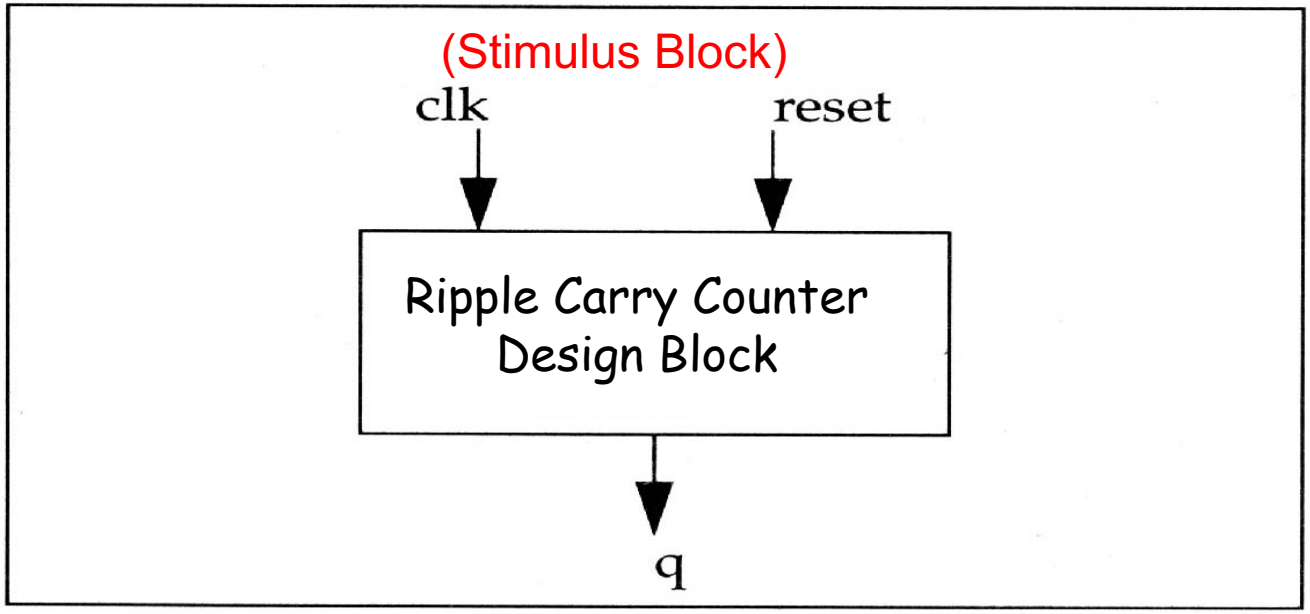
Create four T_FF instances

```
module TFF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q); // not gate is a Verilog provided primitive.
endmodule
```

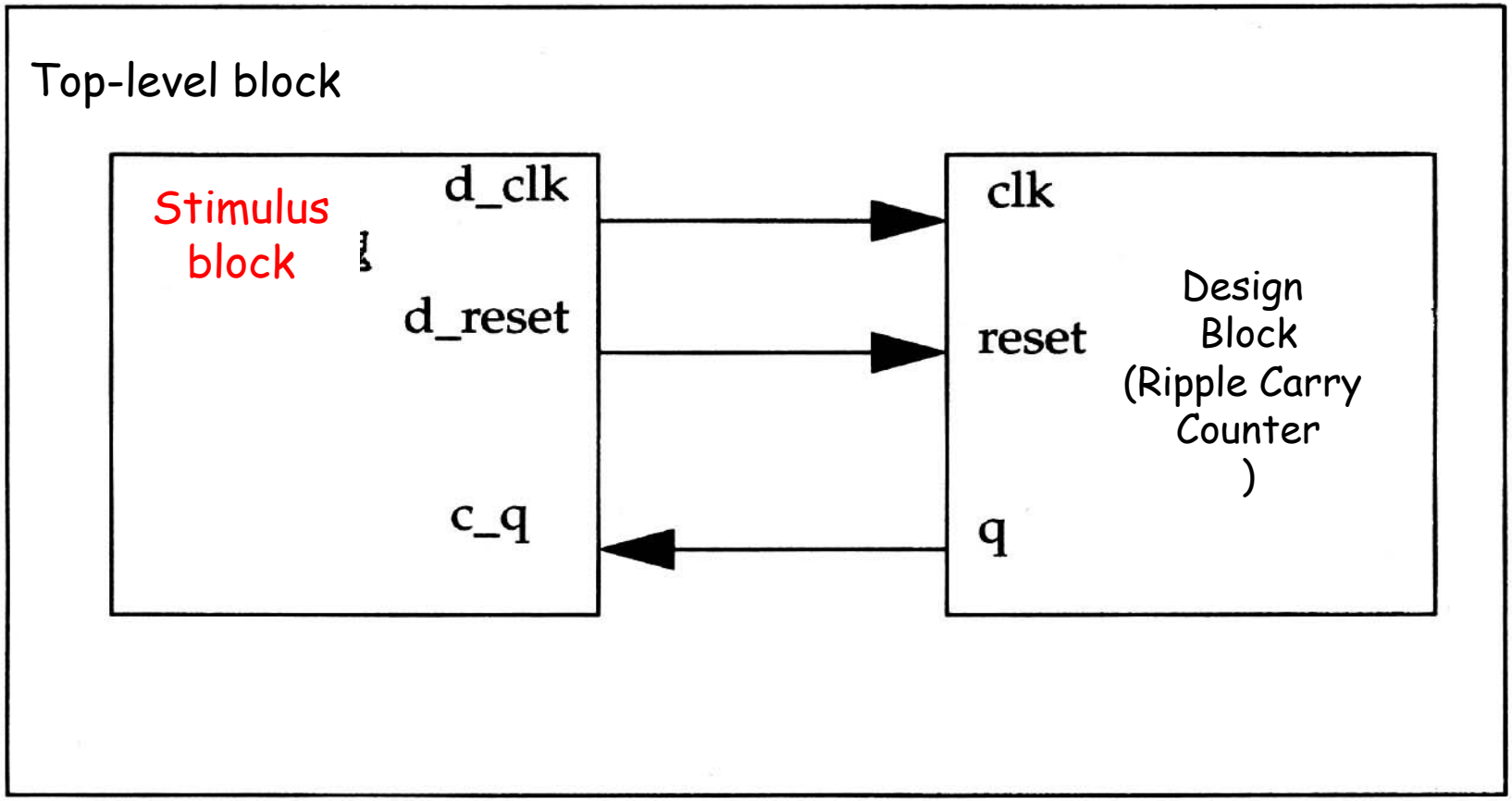
Create D_FF instances

Components of a Simulation

- Use stimulus block to test design block (design under test, DUT)
- Separate stimulus block (test bench) and design block

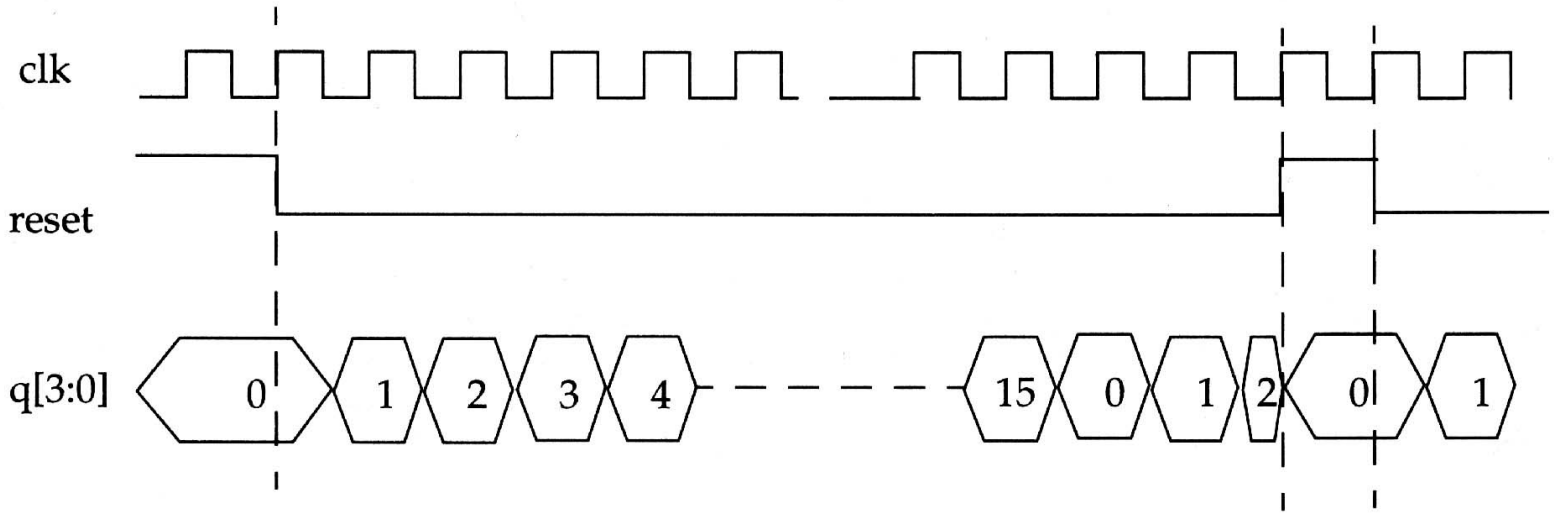


Stimulus and Design block in Dummy top-level module



塊

Ripple Carry Adder Waveform



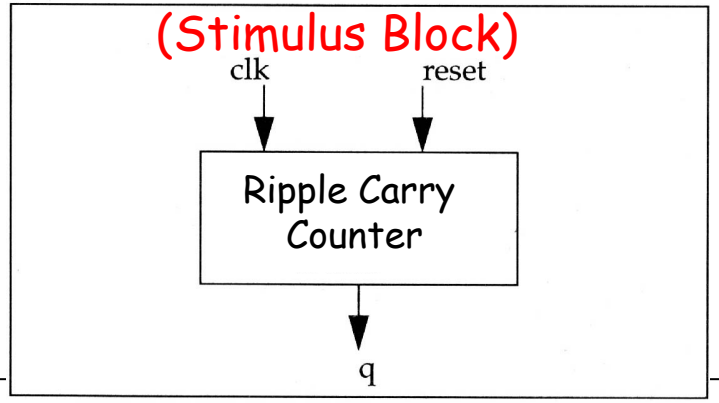
Ripple Carry Counter Top Block

```
module ripple_carry_counter(q, clk, reset);
 output [3:0] q;
 input  clk, reset;
 T_FF tff0 (q[0] , clk , reset);
 T_FF tff1 (q[1] , q[0] , reset);
 T_FF tff2 (q[2] , q[1] , reset);
 T_FF tff3 (q[3] , q[2] , reset);
endmodule
```

```
module T_FF (q, clk, reset);
output q;
input clk, reset;

wire d;

D_FF dff0 (q, d, clk, reset);
not  n1 (d, q);
  //not is a Verilog primitive
endmodule
```

```
// module DFF with synchronous reset
module D_FF(q, d, clk, reset);

output q;
input d, clk, reset;
reg q;

always @(posedge reset or negedge clk)
if (reset)
                        q = 1'b0;
else
                        q = d;

endmodule
```

```
module stimulus;
reg clk;
reg reset;
wire[3:0] q;

// instantiate the design block
ripple_carry_counter r1(q, clk, reset);

// Control the clk signal that drives the design block.
initial clk = 1'b0;
always #5 clk = ~clk;

// Control the reset signal that drives the design block
initial
begin
reset = 1'b1;
#15 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $stop;
end
```



```
// Monitor the outputs
initial
$monitor($time, " Output q = %d",  q);

//Waveforms

initial
$monitor($time, “ Output q = %d”, q);

endmodule
```

output

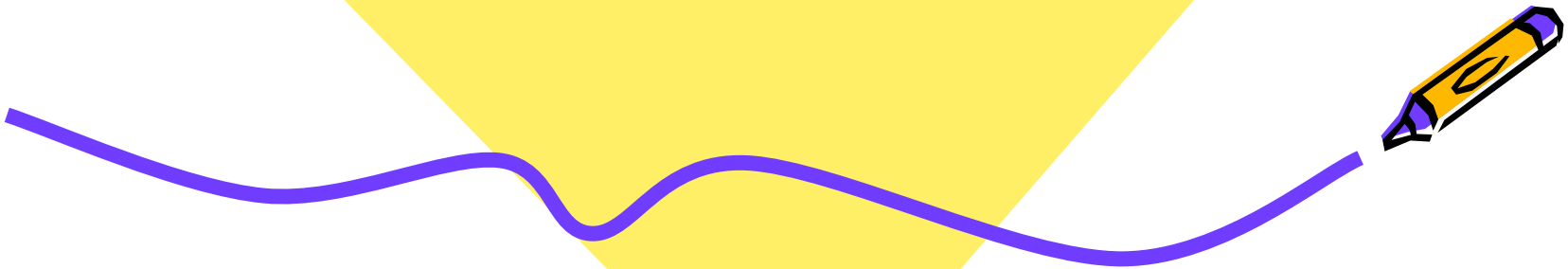
```
0 Output q = 0
20 Output q = 1
30 Output q = 2
40 Output q = 3
50 Output q = 4
60 Output q = 5
70 Output q = 6
80 Output q = 7
90 Output q = 8
100 Output q = 9
110 Output q = 10
120 Output q = 11
130 Output q = 12
140 Output q = 13
150 Output q = 14
160 Output q = 15
170 Output q = 0
```

Summary

- Two kinds of design methodologies
  - Top-down and bottom-up
- Basic building block: Module
- Two components in simulation
  - Design block
  - stimulus

Verilog Basics (Chapter 3)

Partial from By Thanasis Oikonomou



User Identifiers

- Formed from {[A-Z], [a-z], [0-9], _, $}, but can't begin with $ or [0-9]
  - `myidentifier` OK
  - `m_y_identifier` OK
  - `3my_identifier` Not OK
  - `$my_identifier` Not OK
  - `_myidentifier$` OK
- Case sensitivity
  - `myid` ≠ `Myid`

Keywords are special words used to describe the language construct. i.e., module, assign are keywords.

All keywords are in lower case: module, input, …

```
module comparator(
    input x,
    input y,
    output z
    );

assign z = (~x & ~y) |(x & y);

endmodule
```

Identifiers

Keywords

Comments

```
//    The rest of the line is a comment

/*    Multiple line
      comment    */

/*    Nesting /* comments */ do NOT work   */
```

No nested comments

White Space

- White is ignored except when it separates tokens
  - \b (blank space)
  - \t (tab)
  - \n (newlines)

```
module comparator(input x input y, output z );
assign z = (~x & ~y) |(x & y); endmodule
```

```
module comparator(
    input x,
    input y,
    output z
    );

assign z = (~x & ~y) |(x & y);

endmodule
```

Easier to read

Verilog Value Set

0 represents low logic level or false condition

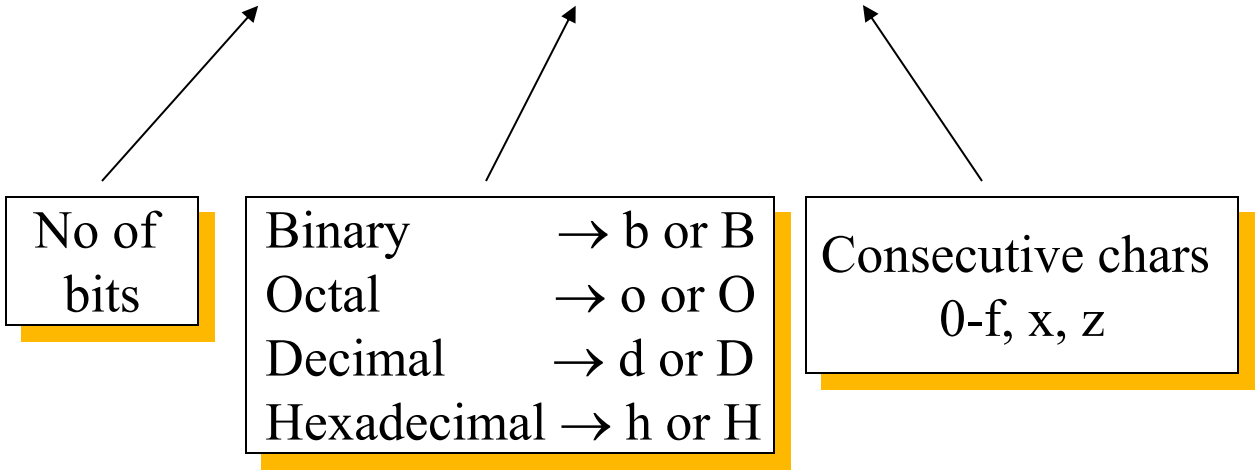
1 represents high logic level or true condition

x represents unknown logic level

z represents high impedance logic level

Numbers in Verilog (i)

<size>'<radix> <value>



– 8'h ax = 1010xxxx

hexadecimal

– 12'o 3zx7 = 011zzzxxx111

octal

Numbers in Verilog (ii)

- You can insert "_" for readability
  - 12'b 000_111_010_100
  - 12'b 000111010100
  - 12'o 07_24

  } Represent the same number

- Bit extension
  - MS bit = 0, x or z $\Rightarrow$ extend this
    - 4'b x1 = 4'b xx_x1
  - MS bit = 1 $\Rightarrow$ zero extension
    - 4'b 1x = 4'b 00_1x

Numbers in Verilog (iii)

- If *size* is ommitted
  - it is inferred from the *value* or
  - it takes the simulation specific number of bits or
  - It takes the machine specific number of bits
- If *radix* is ommitted too .. decimal is assumed

  15 = <size>'d 15

String

- String are enclosed by double quotes
- Example

“Hello Verilog World”
“a / b”

Escaped Identifiers

- Escaped Identifiers begins with backslash(\) and end with whitespace(space, tab, or newline)
- All characters between backslash and whitespace are processed literally
- Ex:

  \a+b-c

  my_name

Escaped Identifiers-2

- Basically you add a backward slash at the beginning of a keyword and you can use any character you want.

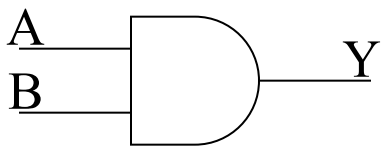
```
module \#differential (
input \clk1~ ,
\rst#
);
```

Nets (i)

- Can be thought as hardware wires driven by logic
- Equal to *z* when unconnected
- Various types of nets
  - `wire`
  - `wand` (wired-AND)
  - `wor` (wired-OR)
  - `tri` (tri-state)
- Synthesizable: wire, wor, wand, tri,

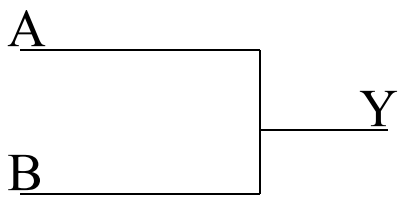
Nets (ii)

Y is evaluated, ***automatically***, every time A or B changes




```
wire Y;  // declaration
assign Y = A & B;
```

Multiple driver on the net

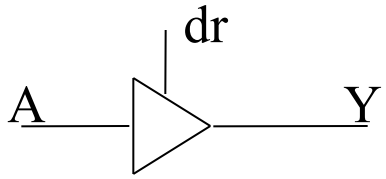



```
wand Y;  // declaration
assign Y = A & B;
```

Y	A=0	A=1
B=0	0	0
B=1	0	1

```
wor Y;  // declaration
assign Y = A or B;
```

Y	A=0	A=1
B=0	0	1
B=1	1	1




```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

Nets (iii)

```
module test( a, b, c, d, e );
   input a, b;
   output c, d, e;
   wire c;
   wand d;
   wor e;

  // wire are connected  to source-→ Error
  assign c = a;
  assign c = b;

  // wire-and → d = a&b
   assign d = a;
   assign d = b;
  // wire-or → e = a|b
   assign e = a;
   assign e = b;
endmodule
```

Registers

- Variables that store values
- Do not represent real hardware but .... real hardware can be implemented with registers
- Do not need a driver or clock
- Only one type: **reg**

```
reg A, C; // declaration
// assignments are always done inside a procedure
A = 1;
C = A; // C gets the logical value 1
A = 0; // C is still 1
C = 0; // C is now 0
```

- Register values are updated explicitly!!

Vectors

- Represent buses

```
wire [3:0] busA;
reg [1:4] busB;
reg [1:0] busC;
```

- Left number is most significant bit
- Slice management

```
busC = busA[2:1];   ⇔   { busC[1] = busA[2];
                         busC[0] = busA[1];
```

- Vector assignment (***by position!!***)

```
busB = busA;   ⇔   { busB[1] = busA[3];
                    busB[2] = busA[2];
                    busB[3] = busA[1];
                    busB[4] = busA[0];
```

Integer & Real Data Types

- Declaration

```
integer i, k;
real r;
```

- Use as registers (inside procedures)

```
i = 1; // assignments occur inside procedure
r = 2.9;
k = r; // k is rounded to 3
```

- Integers are not initialized!!
- Reals are initialized to *0.0*

Time Data Type

- Special data type for simulation time measuring
- Declaration

```
time my_time;
```

- Use inside procedure

```
my_time = $time; // get current sim time
```

- Simulation runs at simulation time, not real time

Arrays (i)

- Syntax

```
integer count[1:5]; // 5 integers
reg var[-15:16]; // 32 1-bit regs
reg [7:0] mem[0:1023]; // 1024 8-bit regs
```

- Accessing array elements
  - Entire element: `mem[10] = 8'b 10101010;`
  - Element subfield (needs temp storage):

```
reg [7:0] temp;
..
temp = mem[10];
var[6] = temp[2];
```

Arrays (ii)

- Vector vs. Array
  - reg [4:0] port [0:7] : array of vector
- Limitation: Cannot access array subfield or entire array at once

```
reg var[-15:16]; // 32 1-bit regs
var[2:9] = ???; // WRONG!!
  var = ???; // WRONG!!
```

- Arrays don't work for the "real" data type

```
real r[1:10]; // WRONG !!
```

Strings

- Implemented with regs:

```
reg [8*13:1] string_val; // can hold up to 13 chars
..
string_val = “Hello Verilog”;
string_val = “hello”; // MS Bytes are filled with 0
string_val = “I am overflowed”; // “I ” is truncated
```

- Escaped chars:
  - \n newline
  - \t tab
  - %% %
  - \\ \
  - \“ “

System Tasks

Always written inside procedures

- $display("..", arg2, arg3, ..); → much like printf(), displays formatted string in std output when encountered
- $monitor("..", arg2, arg3, ..); → like $display(), but .. displays string each time any of arg2, arg3, .. Changes
- $stop; → suspends sim when encountered
- $finish; → finishes sim when encountered
- $fopen("filename"); → returns file descriptor (integer); then, you can use $fdisplay(fd, "..", arg2, arg3, ..); or $fmonitor(fd, "..", arg2, arg3, ..); to write to file
- $fclose(fd); → closes file
- $random(seed); → returns random integer; give an integer as a seed

$display & $monitor string format

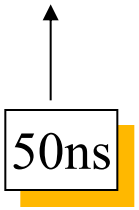
Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

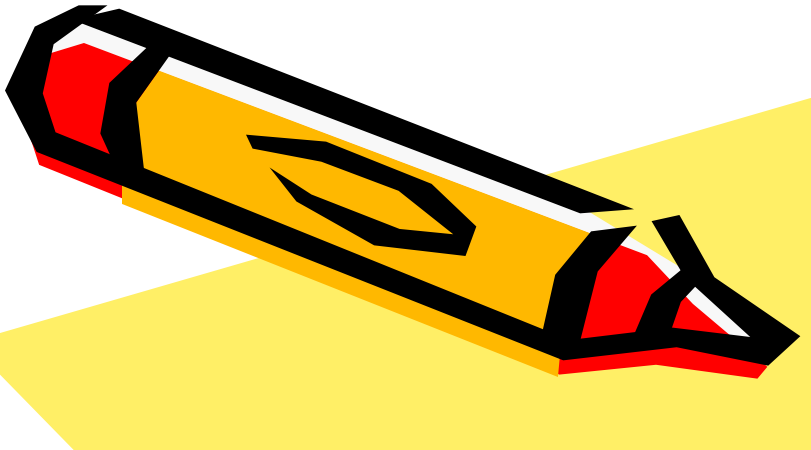
Compiler Directives

- `include "filename" → inserts contents of file into current file; write it anywhere in code ..

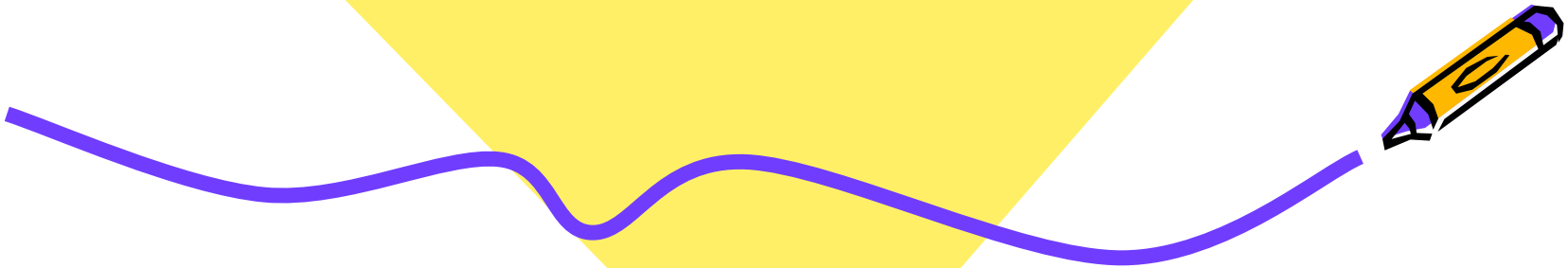
- `define <text1> <text2> → text1 substitutes text2;
  - e.g. `define BUS reg [31:0]` in declaration part:
    `BUS data;`

- `timescale <time unit>/<precision>
  - e.g. `timescale 10ns/1ns` later: `#5 a = b;`





Verilog Basics
Modules and Ports



Components of a Verilog Module

Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),

Declarations of wires, regs and other variables

Data flow statements (assign)

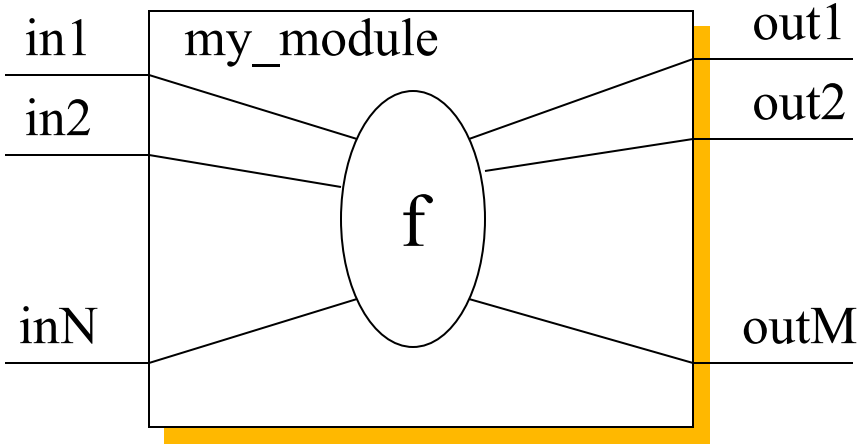
Instantiation of lower level modules

always and initial blocks. All behavioral statements go in these blocks

Tasks and functions

Endmodule statement

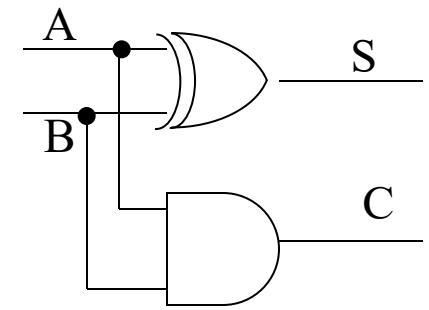
Module

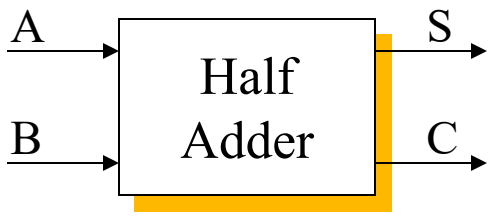



```
module my_module(out1, .., inN);
output out1, .., outM;
input in1, .., inN;

.. // declarations
.. // description of f (maybe
.. // sequential)

endmodule
```

Everything you write in Verilog must be inside a module
exception: compiler directives

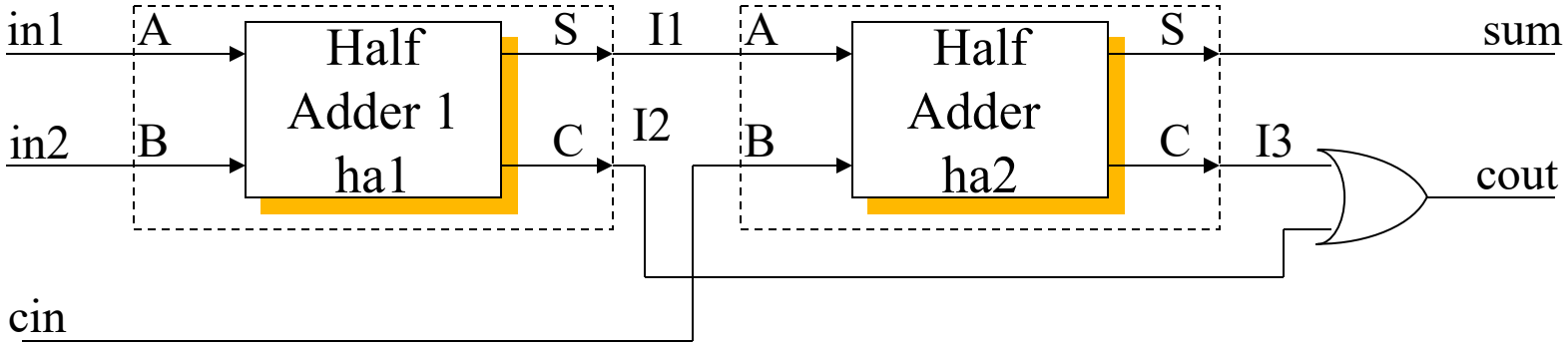
Example: Half Adder






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

wire S, C, A, B;

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

endmodule
```

Example: Full Adder



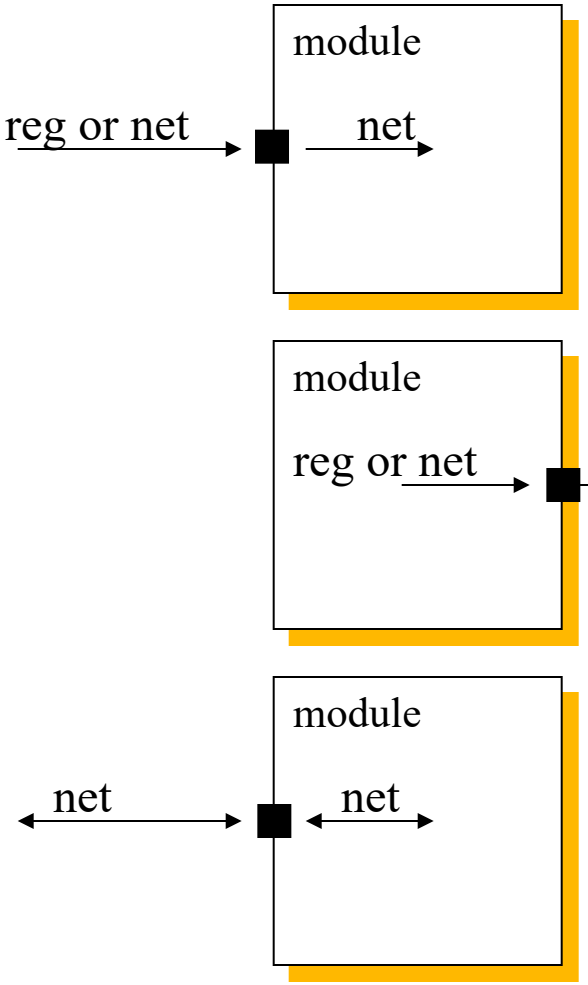

```
module full_adder(sum, cout, in1, in2, cin);
output sum, cout;
input in1, in2, cin;

wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(I1, I2, in1, in2);
half_adder ha2(sum, I3, I1, cin);

assign cout = I2 || I3;

endmodule
```

Module name

Instance name

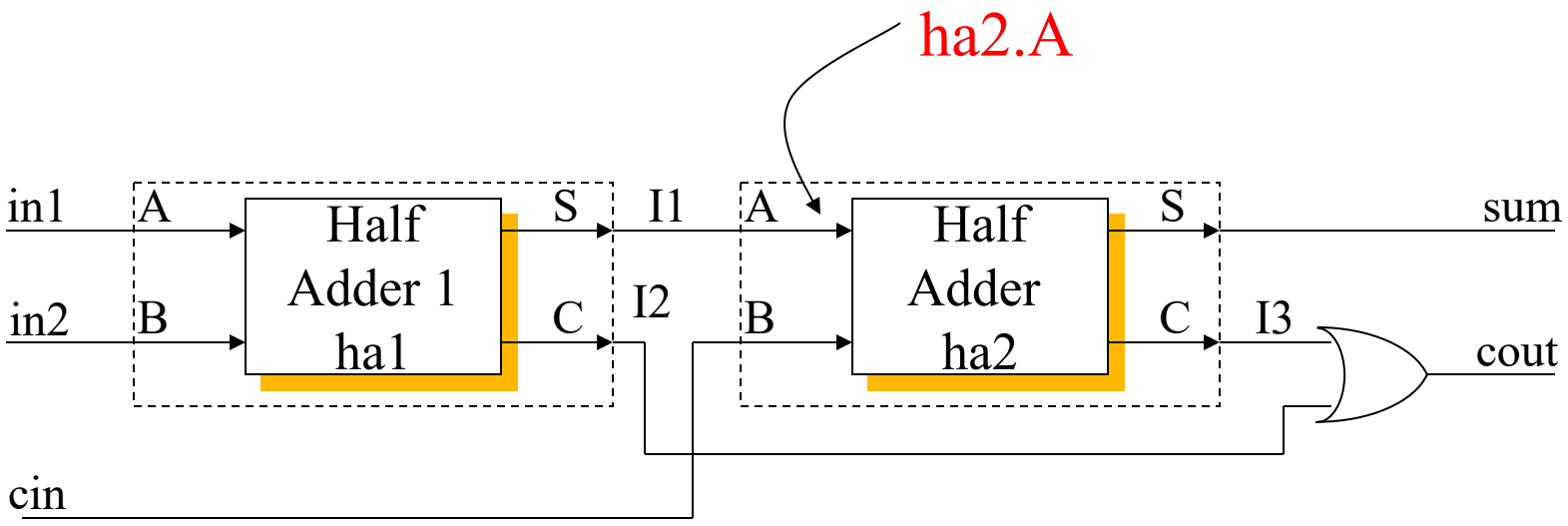
Port Assignments

- Inputs
- Outputs
- Inouts

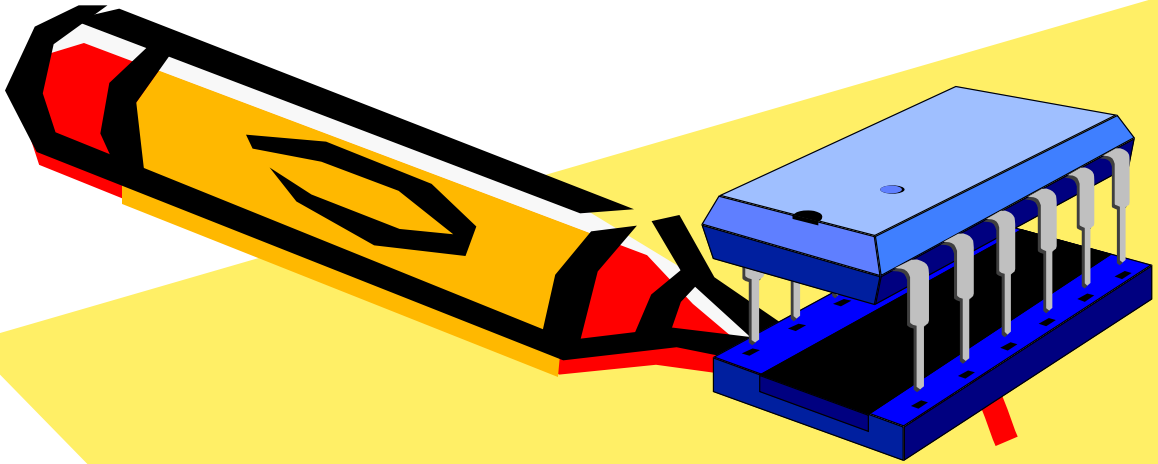


Output port only connects to net

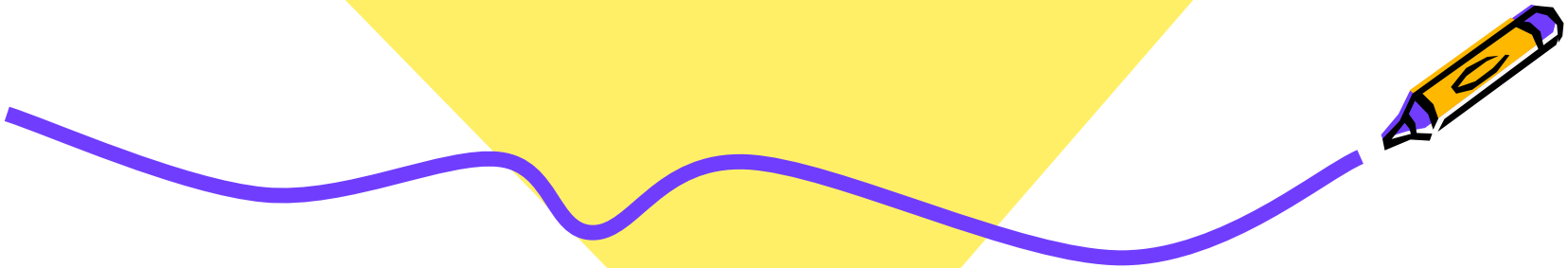
Hierarchical Names



Remember to use instance names,
not module names



Verilog Basics
- Structure Model



Structural Model (Gate Level)

- Built-in gate primitives:

```
and, nand, nor, or, xor, xnor, buf, not, bufif0,
 bufif1, notif0, notif1
```

- Usage:

```
nand (out, in1, in2);
```
2-input NAND without delay

```
and #2 (out, in1, in2, in3);
```
3-input AND with 2 t.u. delay

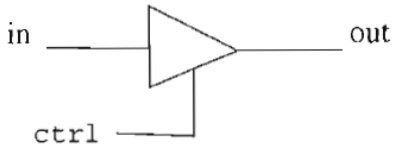
```
not #1 N1(out, in);
```
NOT with 1 t.u. delay and instance name

```
xor X1(out, in1, in2);
```
2-input XOR with instance name

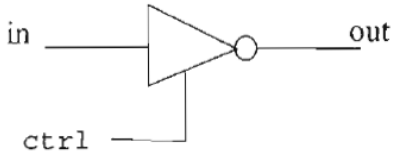
- Write them inside module, outside procedures

- Special gates
  - bufif1
  - bufif0
  - notif1
  - notif0

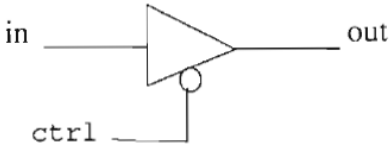
Note: they are synthesizable



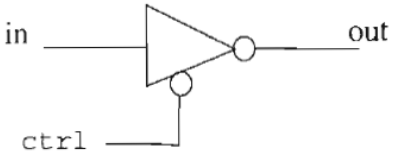
bufif1



notif1



bufif0



notif0

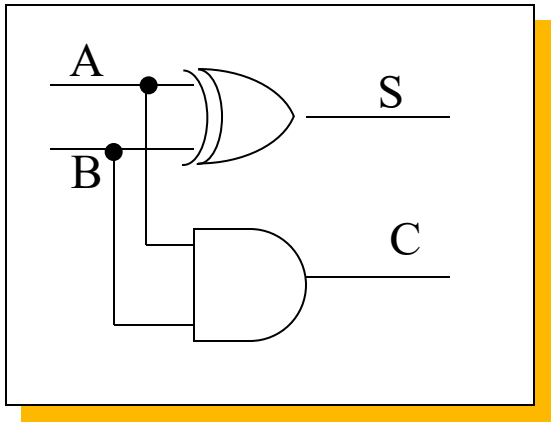
in \ bufif1	ctrl 0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	x	x	x

in \ bufif0	ctrl 0	1	x	z
0	0	z	L	L
1	1	z	H	H
x	x	z	x	x
z	x	z	x	x

in \ notif1	ctrl 0	1	x	z
0	z	1	H	H
1	z	0	L	L
x	z	x	x	x
z	z	x	x	x

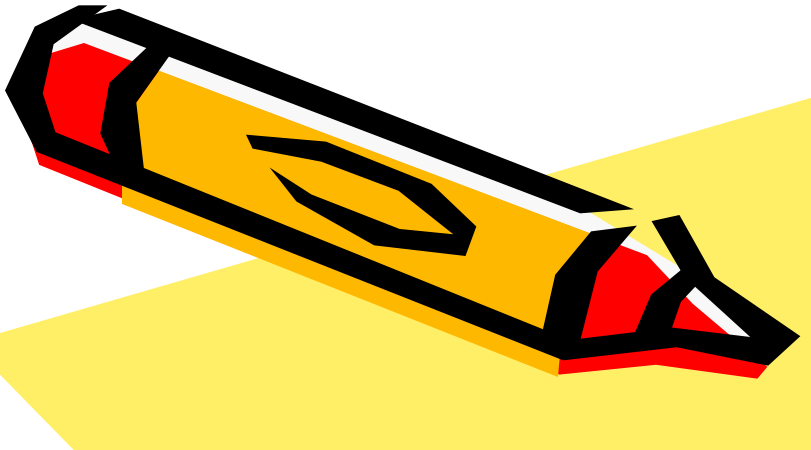
in \ notif0	ctrl 0	1	x	z
0	1	z	H	H
1	0	z	L	L
x	x	z	x	x
z	x	z	x	x

Example: Half Adder, 2nd Implementation

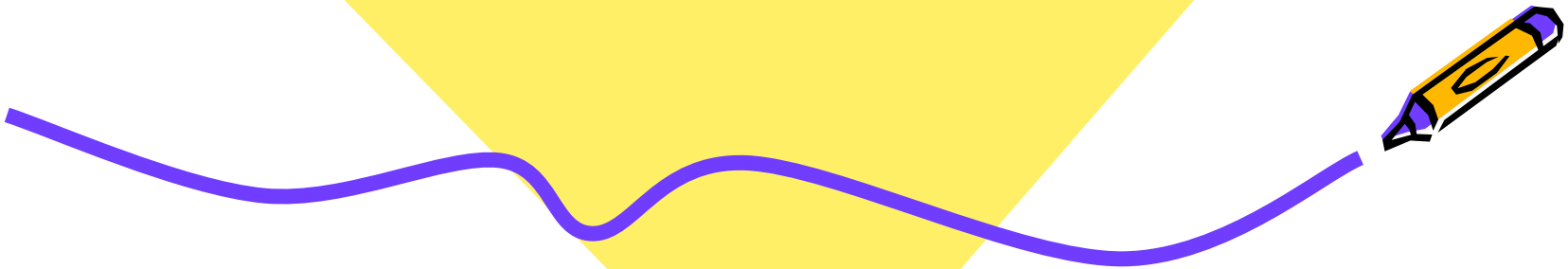


Assuming:
- XOR: 2 t.u. delay
- AND: 1 t.u. delay

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

wire S, C, A, B;

xor #2 (S, A, B);
and #1 (C, A, B);

endmodule
```



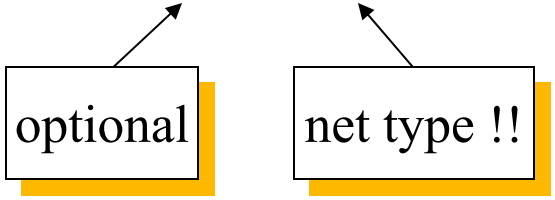
Verilog Basics
- Dataflow Modeling



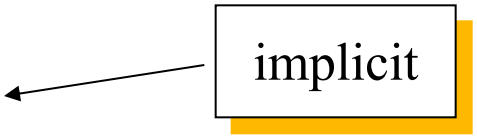
Continuous Assignements
a closer look

- Syntax:

```
assign #delay <id> = <expr>;
```




```
assign out = in & in2;
```



- Where to write them:
  - inside a module or outside procedures
- Properties:
  - they all execute in parallel
  - are order independent
  - are continuously active

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	? :	Conditional	Three

Arithmetic Operators (i)

- +, -, *, /, %, **
  - 16 %4 => 0
- If any operand is *x (unknown)* the result is *x*

  4'b101x+ 4'b1011 => 4'bx
- Negative registers:
  - regs can be assigned negative but are treated as unsigned

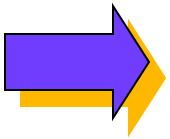
```
reg [15:0] regA;
..
regA = -4'd12;     // stored as 2^16-12 = 65524
regA/3             evaluates to 21861
```

Wrong

Logical Operators

- `&&` → logical AND
- `||` → logical OR
- `!` → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

```
A = 6;
B = 0;
C = x;
```



```
A && B → 1 && 0 → 0
A || !B → 1 || 1 → 1
C || B → x || 0 → x
```

but C&&B=0

Relational Operators

- `>` → greater than
- `<` → less than
- `>=` → greater or equal than
- `<=` → less or equal than

- Result is one bit value: *0*, *1* or *x*

```
1 > 0          → 1
'b1x1 <= 0     → x   // because of x in 'b1x1
10 < z         → x
```

Equality Operators

- `==` → logical equality
- `!=` → logical inequality

Return *0*, *1* or *x*

- `===` → case equality
  - *Including x and z*
- `!==` → case inequality
  - *Including x and z*

Return *0* or *1*

- `4'b 1z0x == 4'b 1z0x` → *x*
- `4'b 1z0x != 4'b 1z0x` → *x*
- `4'b 1z0x === 4'b 1z0x` → *1*
- `4'b 1z0x !== 4'b 1z0x` → *0*

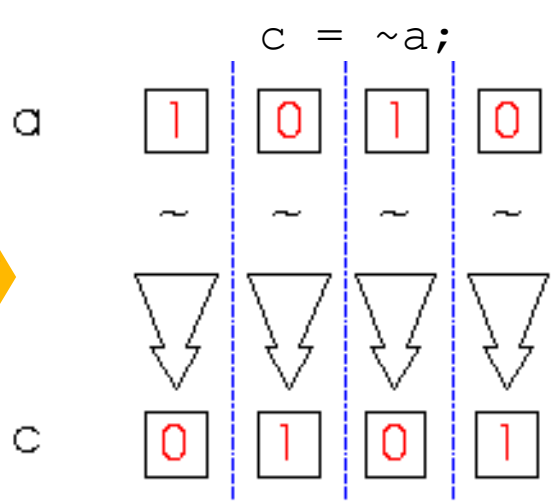
Bitwise Operators (i)

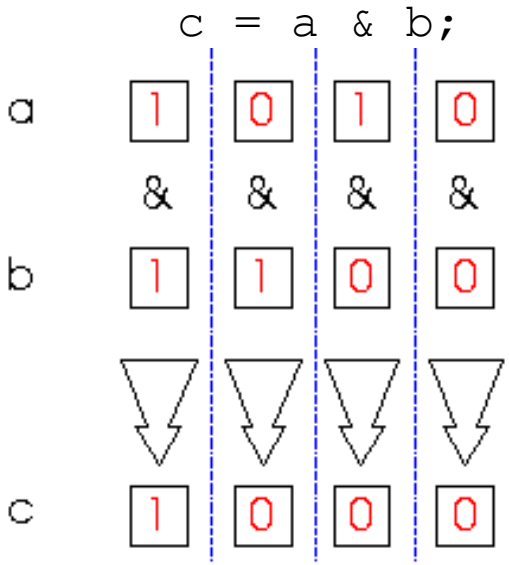
- & → bitwise AND
- | → bitwise OR
- ~ → bitwise NOT
- ^ → bitwise XOR
- ~^ or ^~ → bitwise XNOR

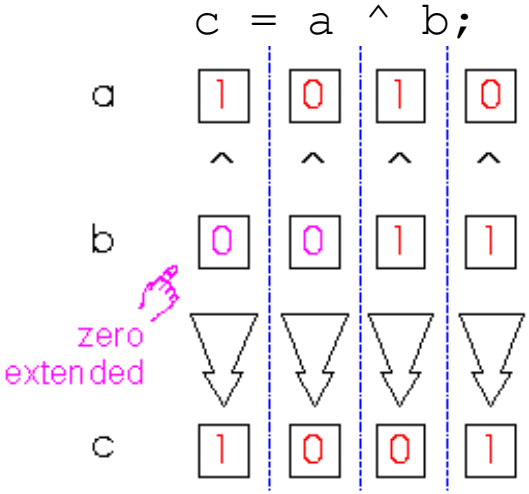
- Operation is done bit by bit

Bitwise Operators (ii)

- `a = 4'b1010;`
  `b = 4'b1100;`







- `a = 4'b1010;`
  `b = 2'b11;`

Reduction Operators

- `&` → AND
- `|` → OR
- `^` → XOR
- `~&` → NAND
- `~|` → NOR
- `~^` or `^~` → XNOR

- One multi-bit operand → One single-bit result

```
a = 4'b1001;
..
c = |a; // c = 1|0|0|1 = 1
```

Shift Operators

- >> → shift right
- << → shift left

- Result is same size as first operand, **always zero filled**

```
a = 4'b1010;
...
d = a >> 2;     // d = 0010
c = a << 1;     // c = 0100
```

Concatenation Operator

- {op1, op2, ..} → concatenates op1, op2, .. to single number
- Operands must be sized !!

```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catx = {a, b, c};              // catx = 1_010_101
caty = {b, 2'b11, a};          // caty = 010_11_1
catz = {b, 1};          // WRONG !!, do not know size of "1"
```

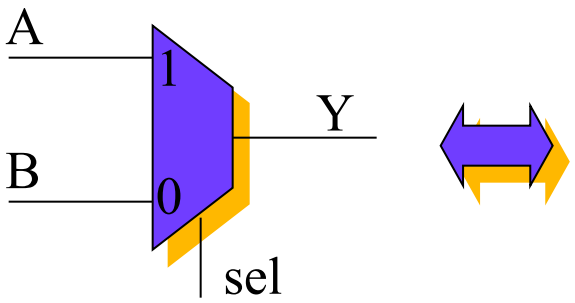
- Replication ..

```
catr = {4{a}, b, 2{c}};      // catr = 1111_010_101101
```

Conditional Operator

- `cond_expr ? true_expr : false_expr`

- Like a 2-to-1 mux ..




```
Y = (sel)? A : B;
```

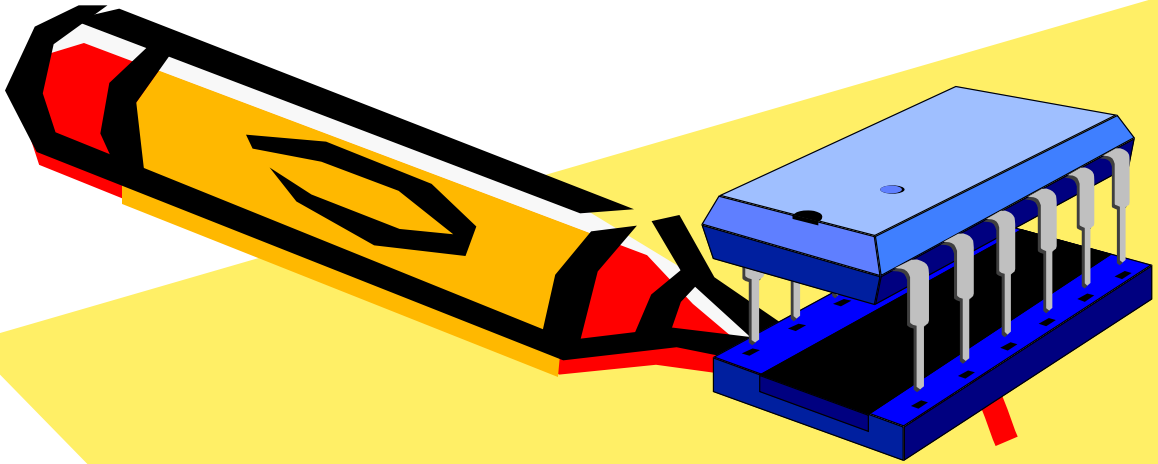
Operator Precedence

+-!~unary	highest precedence
*/%	
+-(binary)	
<< >>	
< <= => >	
== != === !==	
& ~&	
^ ^~ ~^	
| ~|	
&&	
||	
?: conditional	lowest precedence

Use parentheses to enforce your priority

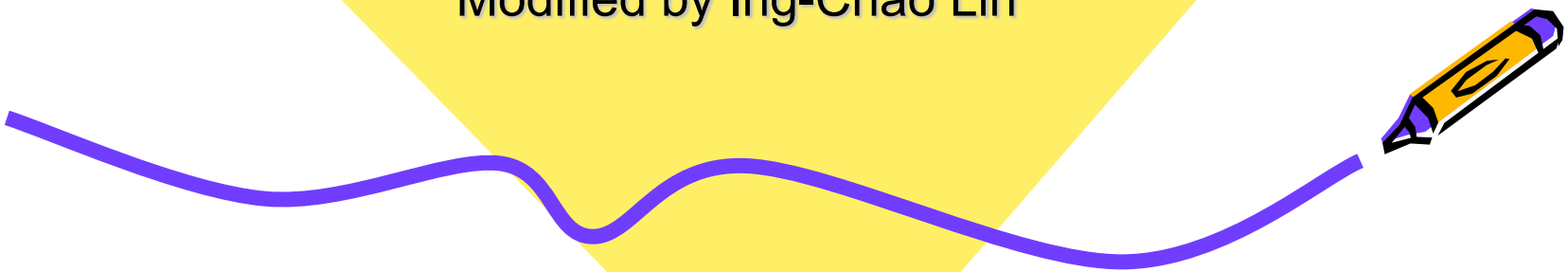
Example

```
// 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

assign out = (~s1 & ~s0 & i0) |
                    (~s1 & s0 & i1) |
                    (s1 & ~s0 & i2) |
                    (s1 & s0 & i3) ;


endmodule
```



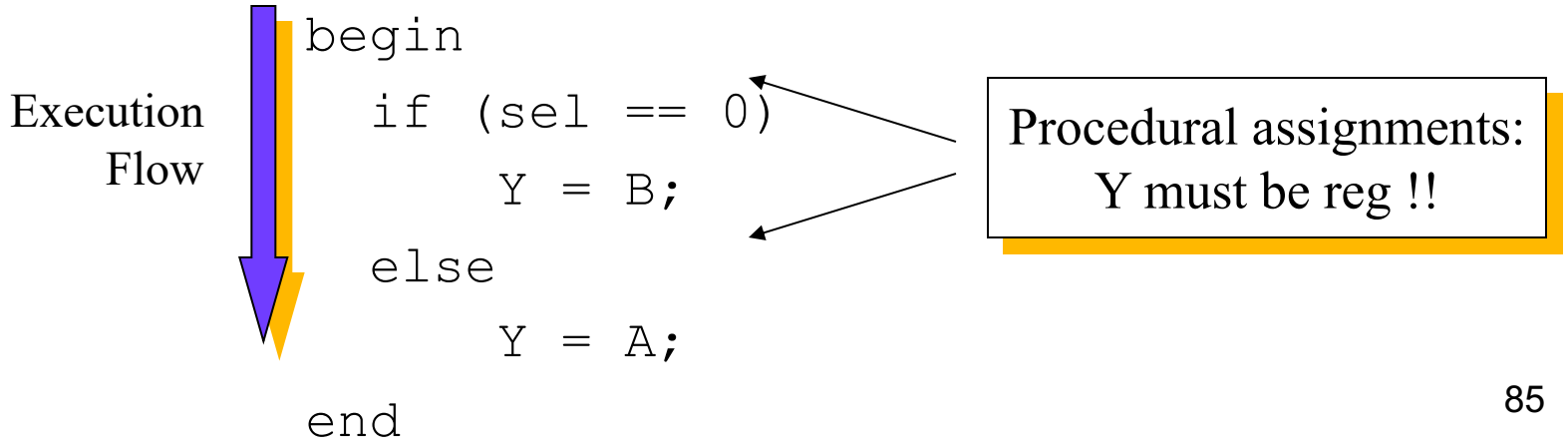
Verilog Basics
- Behavioral Model

Original by Thanasis Oikonomou
Modified by Ing-Chao Lin



Behavioral Model - Procedures (i)

- always and initial
- Procedures = sections of code that we know they execute sequentially
- Procedural statements = statements inside a procedure (they execute sequentially)
- e.g. another 2-to-1 mux implem:



Behavioral Model - Procedures (ii)

- Modules can contain any number of procedures
- Procedures execute in parallel (in respect to each other) and ..
- .. can be expressed in two types of blocks:
  - initial → they execute only once
  - always → they execute for ever (until simulation finishes)

“Initial” Blocks

- Start execution at sim time zero and finish when their last statement executes

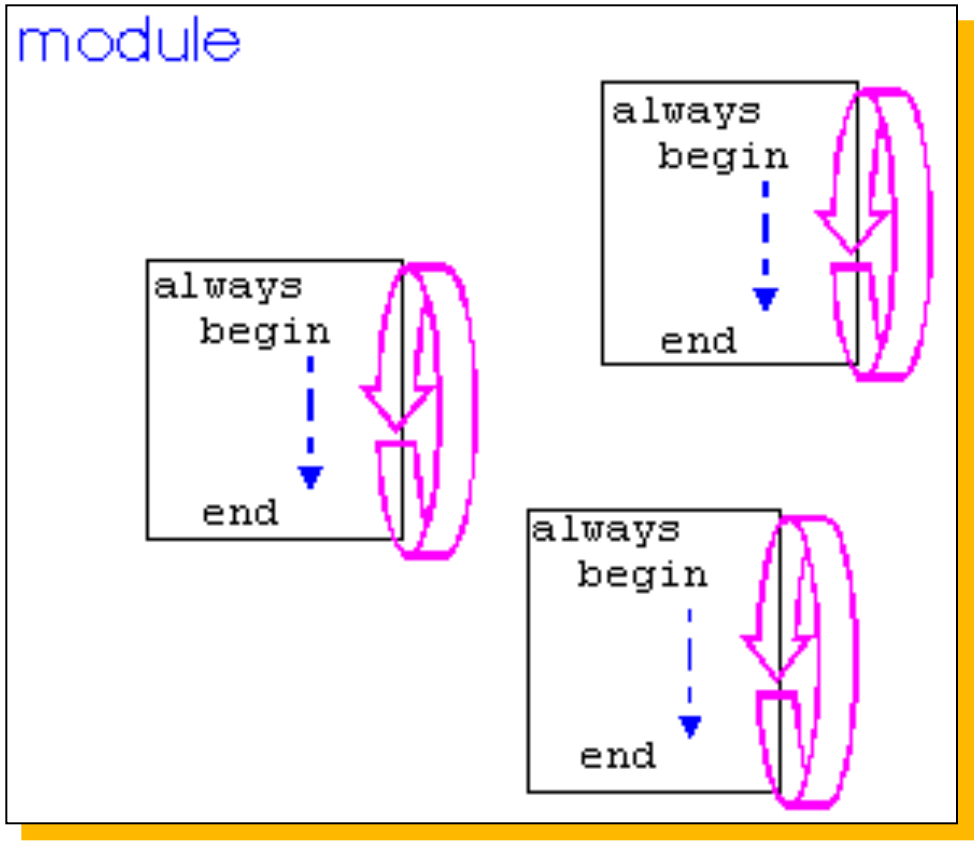
```
module nothing;

initial
  $display(“I’m first”);

initial begin
  #50;
  $display(“Really?”);
  end

endmodule
```

Will be displayed at sim time 0

Will be displayed at sim time 50

“Always” Blocks

- Start execution at sim time zero and continue until sim finishes



Events (i)

- @

```
always @(signal1 or signal2 or ..) begin
    ..
    end
```

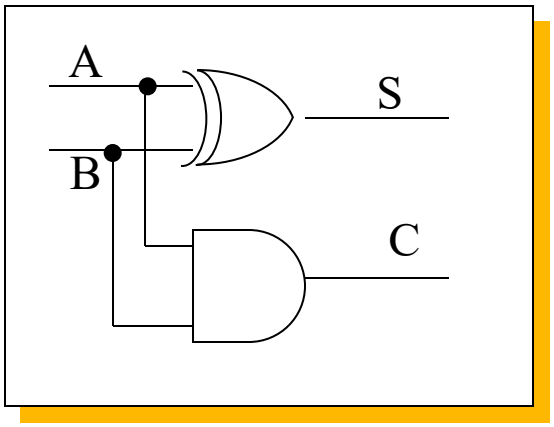
execution triggers every time any signal changes

```
always @(posedge clk) begin
    ..
    end
```

execution triggers every time clk changes from *0* to *1*

```
always @(negedge clk) begin
    ..
    end
```

execution triggers every time clk changes from *1* to *0*



Examples

- 3rd half adder implementation

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

reg S,C;
wire A, B;

always @(A or B) begin
   S = A ^ B;
   C = A && B;
   end
endmodule
```

- Behavioral edge-triggered DFF

```
module dff(Q, D, Clk);
output Q;
input D, Clk;

reg Q;
wire D, Clk;

always @(posedge Clk)
   Q = D;

endmodule
```

Events (ii)

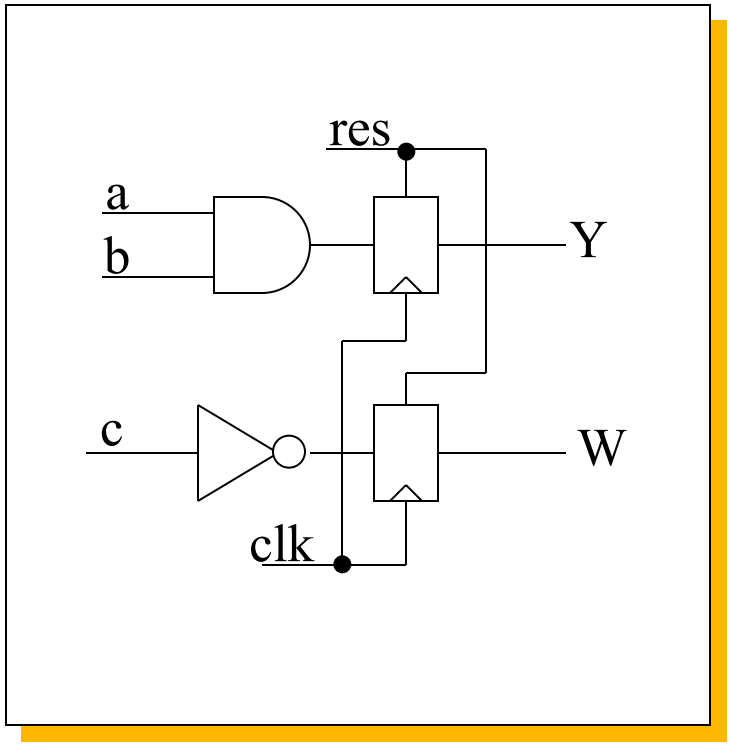
- wait (expr)

```
always begin
    wait (ctrl)
    #10 cnt = cnt + 1;
    #10 cnt2 = cnt2 + 2;

    end
```

Not synthesizable

execution loops every time ctrl = 1 (level sensitive timing control)

Example




```
always @(res or posedge clk) begin
        if (res) begin
                Y = 0;
                W = 0;
                end
        else begin
                Y = a & b;
                W = ~c;
                end
        end
```

Procedural Assignments

- Blocking Assignment (use =)
- Nonblocking Assignment (use <=)

```
// Blocking assignments
initial begin
a = #10 1'b1;// The simulator assigns 1 to a at time 10
b = #20 1'b0;// The simulator assigns 0 to b at time 30
c = #40 1'b1;// The simulator assigns 1 to c at time 70
end

// Nonblocking assignments
initial begin
d <= #10 1'b1;// The simulator assigns 1 to d at time 10
e <= #20 1'b0;// The simulator assigns 0 to e at time 20
f <= #40 1'b1;// The simulator assigns 1 to f at time 40
end
```

Procedural Statements: if

```
if (expr1)
        true_stmt1;

else if (expr2)
        true_stmt2;
..
else
        def_stmt;
```

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        if (sel == 0)
                out = in[0];
        else if (sel == 1)
                out = in[1];
        else if (sel == 2)
                out = in[2];
        else
                out = in[3];
endmodule
```

"if" is synthesizable

Procedural Statements: case

```
case (expr)

item_1, .., item_n:          stmt1;
item_n+1, .., item_m:  stmt2;
..
default:              def_stmt;

endcase
```

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        case (sel)
        0: out = in[0];
        1: out = in[1];
        2: out = in[2];
        3: out = in[3];
        endcase
endmodule
```

Procedural Statements: for

for (init_assignment; cond; step_assignment)
    stmt;

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start)
        for (i = 0; i < 3; i = i + 1)
                #10 Y = Y + 1;
endmodule
```

"for" is synthesizable

Procedural Statements: while

while (expr) stmt;

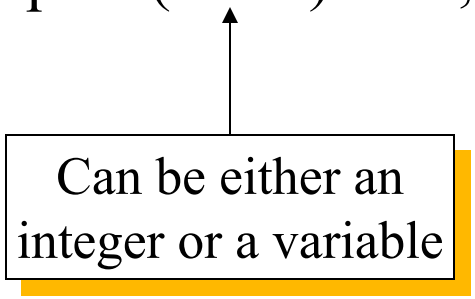
"while" is synthesizable in some tools (not recommended to be used in synthesizable codes)

E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start) begin
        i = 0;
        while (i < 3) begin
                #10 Y = Y + 1;
                i = i + 1;
                end
        end
endmodule
```

Procedural Statements: repeat

repeat (times) stmt;



E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;

initial
        Y = 0;

always @(posedge start)
        repeat (4) #10 Y = Y + 1;
endmodule
```

Procedural Statements: forever

forever stmt;

Executes until sim finishes

Typical example:

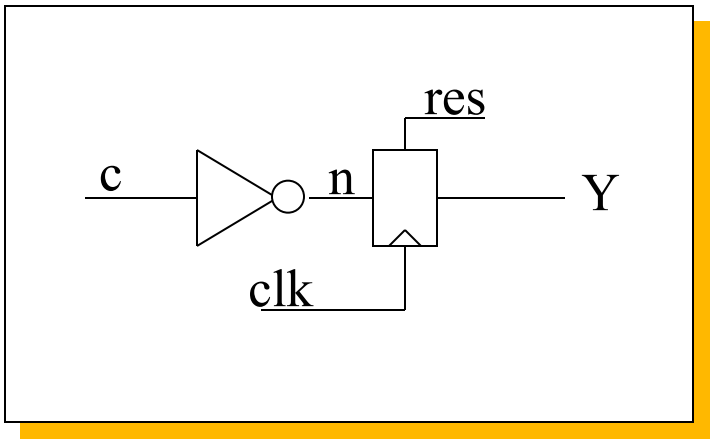
clock generation in test modules

```
module test;

reg clk;

initial begin
        clk = 0;
        forever #10 clk = ~clk;
        end

other_module1 o1(clk, ..);
other_module2 o2(.., clk, ..);

endmodule
```

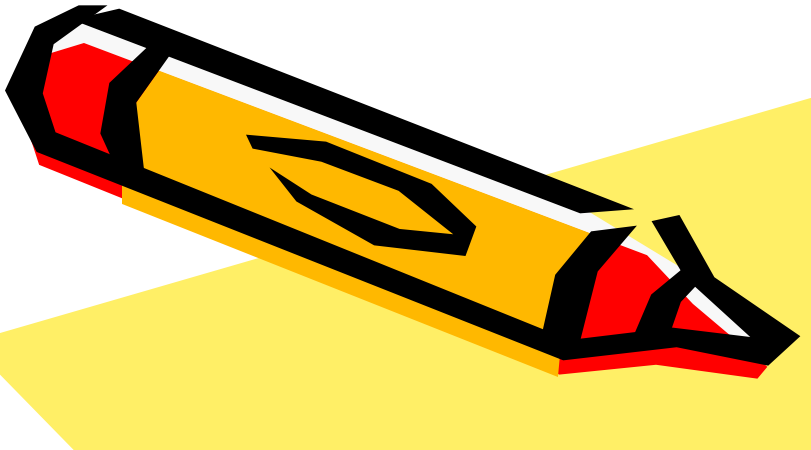
T_{clk} = 20 time units

Mixed Model

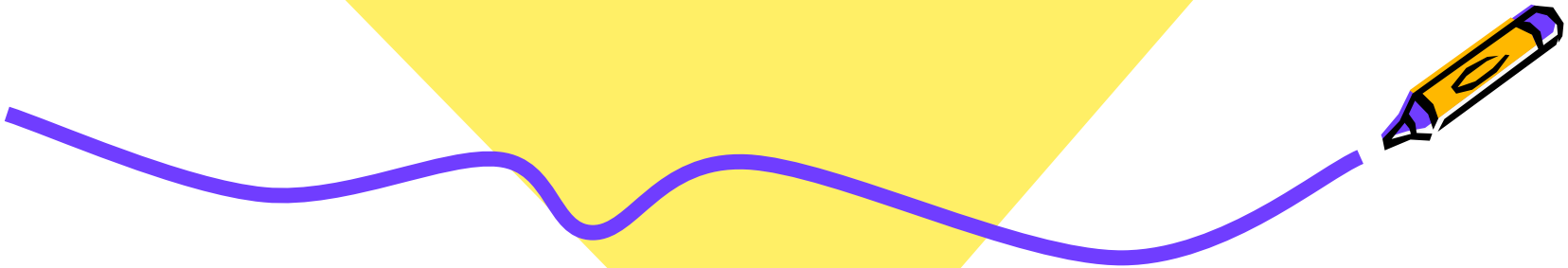
Code that contains various both structure and behavioral styles



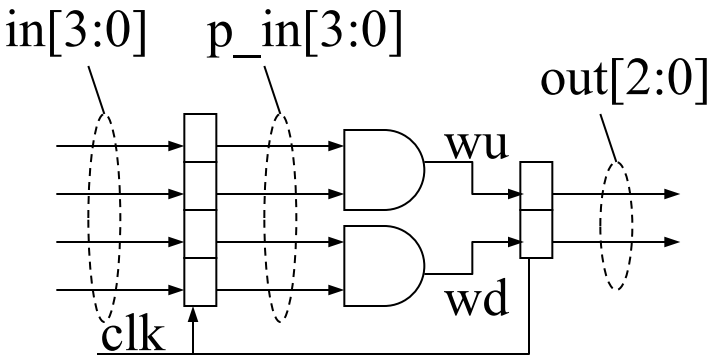

```
module simple(Y, c, clk, res);
output Y;
input c, clk, res;

reg Y;
wire c, clk, res;
wire n;

not(n, c); // gate-level

always @(res or posedge clk)
       if (res)
              Y = 0;
       else
              Y = n;
endmodule
```



Verilog Basics
- Parameter



Parameters



A. Implelementation without parameters

```
module dff4bit(Q, D, clk);
output [3:0] Q;
input [3:0] D;
input clk;

reg [3:0] Q;
wire [3:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

```
module dff2bit(Q, D, clk);
output [1:0] Q;
input [1:0] D;
input clk;

reg [1:0] Q;
wire [1:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

Parameters (ii)

A. Implelementation without parameters (cont.)

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;    // internal nets
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff4bit instA(p_in, in, clk);
dff2bit instB(out, {wu, wd}, clk);
// notice the concatenation!!

endmodule
```

Parameters (iii)

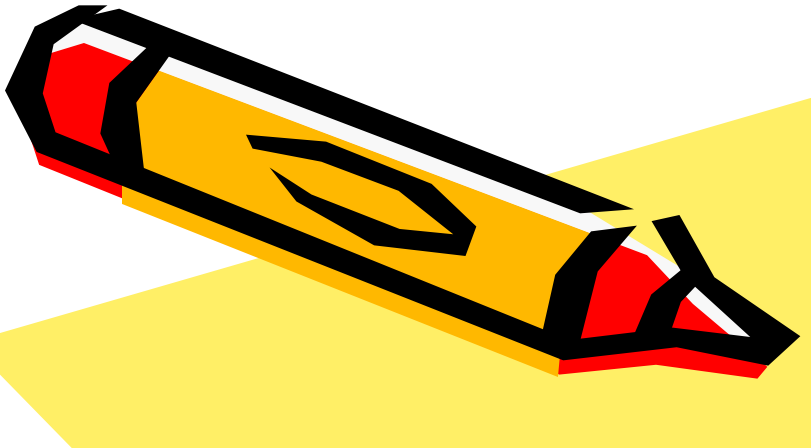
B. Implelementation with parameters

```
module dff(Q, D, clk);
parameter WIDTH = 4;
output [WIDTH-1:0] Q;
input [WIDTH-1:0] D;
input clk;

reg [WIDTH-1:0] Q;
wire [WIDTH-1:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

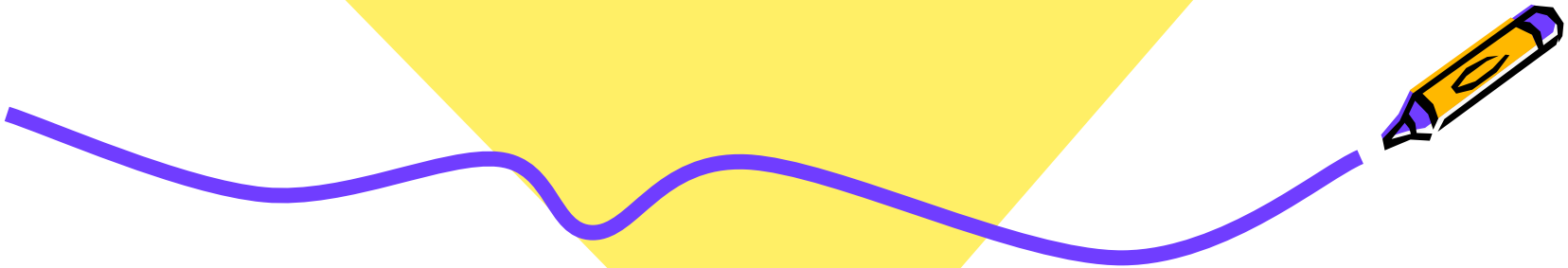
```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff instA(p_in, in, clk);
// WIDTH = 4, from declaration

dff instB(out, {wu, wd}, clk);
       defparam instB.WIDTH = 2;
// We changed WIDTH for instB only

endmodule
```

Testing Your Modules

```
module top_test;
wire [1:0] t_out;    // Top's signals
reg [3:0] t_in;
reg clk;

top inst(t_out, t_in, clk); // Top's instance

initial begin        // Generate clock
       clk = 0;
       forever #10 clk = ~clk;
end

initial begin        // Generate remaining inputs
       $monitor($time, " %b -> %b", t_in, t_out);
       #5 t_in = 4'b0101;
       #20 t_in = 4'b1110;
       #20 t_in[0] = 1;
       #300 $finish;
end

endmodule
```



Synthesizable Verilog Code



Synthesizable Verilog Code

- Not all kinds of Verilog constructs can be synthesized.
- Only a subset of Verilog constructs can be synthesized and the code containing only this subset is synthesizable.

HDL Compiler Unsupported

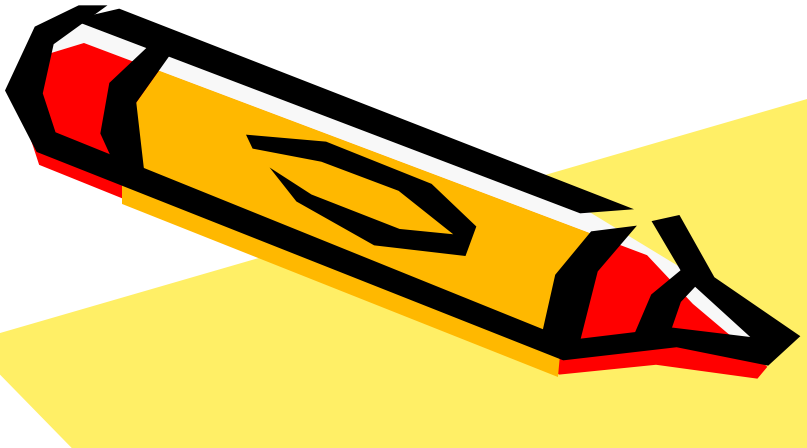
- delay
- initial
- repeat
- wait
- fork … join
- event
- deassign
- force
- release
- primitive -- User defined primitive
- time
- triand, trior, tri1, tri0, trireg
- nmos, pmos, cmos, rnmos, rpmos, rcmos
- pullup, pulldown
- rtran, tranif0, tranif1, rtranif0, rtranif1
- case identity and not identity operators
- Division and modulus operators
  - division can be done using DesignWare instantiation

Verilog Basis & Primitive Cell Supported

- Verilog basis
  - Parameter declarations
  - Wire, wand, wor declarations
  - Reg declarations
  - Input, output, inout declarations
  - Continuous assignments
  - Module instantiations
  - Gate instantiations
  - Always blocks
  - Task statements
  - Function definitions
  - For, while loop
- Synthesizable Verilog primitive cells
  - And, or, not, nand, nor, xor, xnor
  - Bufif0, bufif1, notif0, notif1

Verilog Operators Supported

- . Binary bit-wise (~,&,|,^,~^)
- . Unary reduction (&,~&,|,~|,^,~^)
- . Logical (!,&&,||)
- . 2‘s complement arithmetic (+,-,*,/,%)
- . Relational (>,<,>=,<=)
- . Equality (==,!=)
- . Logical shift (>>,<<)
- . Conditional (?:)



Backup Slides