Digital System Design

Logic Design at Register-Transfer Level - Continuous Assignments

Lecturer: 王景平

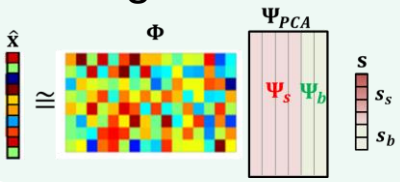
Date: 2025.03.06

Based on: Ch.4 & Ch.7 of the textbook
Review: Logic Design (Concepts of Combinational & Sequential Circuits)

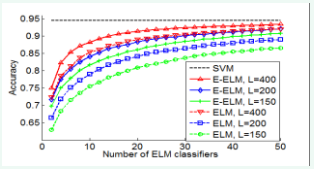
Cell-Based IC Design Flow

Algorithm

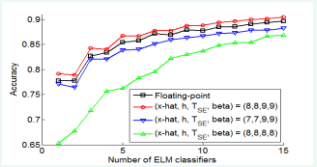
Algorithm



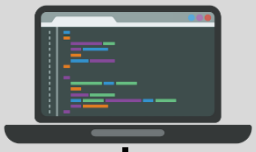
Floating-point Analysis



Fixed-point Analysis



RTL

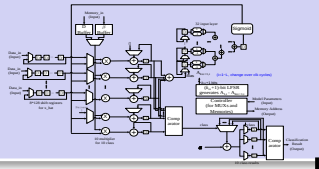


No Delay!

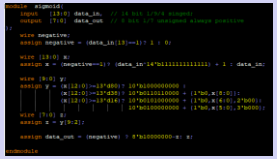
Spec

Parameter	Range	Descript ion
Dimension of input data (D_x)	128, 256, 512, 1024	MNIST Databas e
Number of hidden neurons (h_n)	1-256	

Architecture

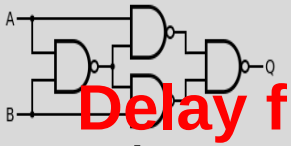


Verilog



RTL Simulation

Synthesis

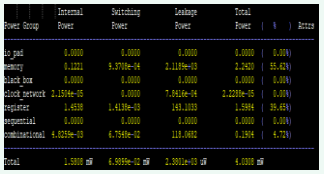


Delay from TSMC file

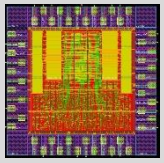
Gate-level Netlist

Gate-level Simulation

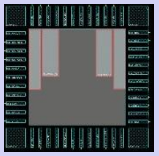
Power Optimization



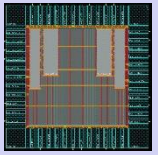
Layout



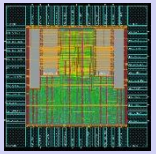
Floorplan



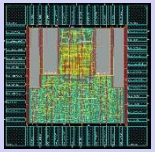
Powerplan



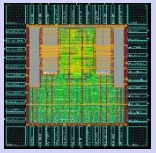
Placement



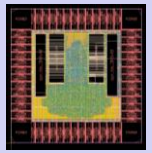
CTS



Route



Post-layout Simulation



Bit Order of Vectors

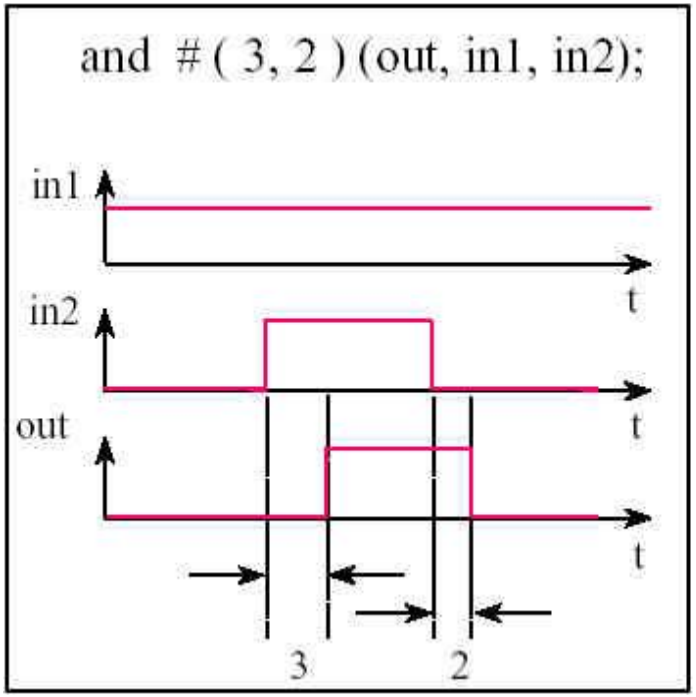
- a[7:0] and b[0:7] has the **same value** but **different bit order**

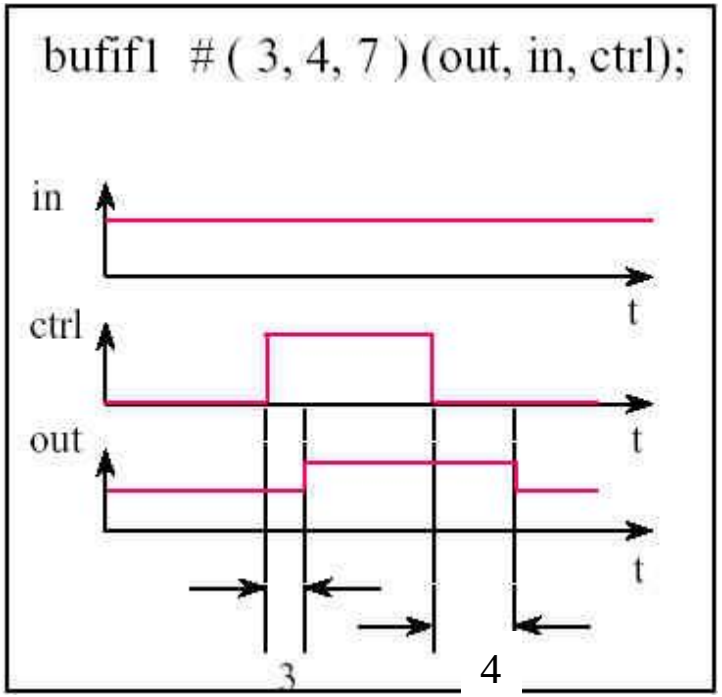
8'd37	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
a[7:0]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
b[0:7]	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]

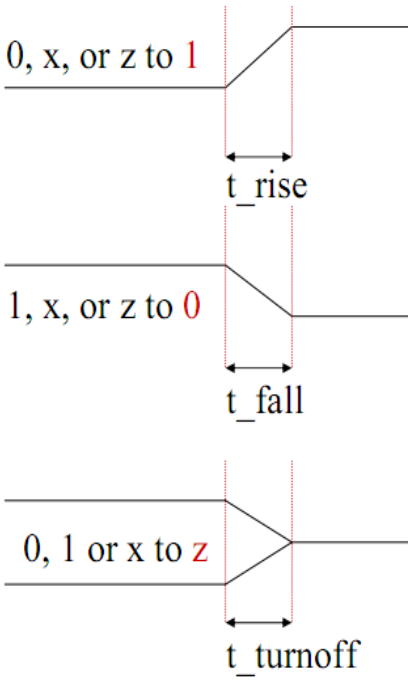
- Any operation between two vectors is on their **values**
  - `$display(b[7], b[6], b[5], b[4], b[3], b[2], b[1], b[0]);`
    > 10100100
  - `$display(a == b);`
    > 1
- Changing bit order of the vector declaration affects the **notation** but the value

Delay Specification in Primitives

- Verilog supports (rise, fall, turn-off) delay specification.







0, 1, z to x

Module Declaration

- Encapsulate structural and functional details in a module

```
module <Module Name> (<PortName List>);

 // Structural part
      <List of Ports>
      <Lists of Nets and Registers>
      <SubModule List>  <SubModule Connections>

// Behavior part
      <Timing Control Statements>
            <Parameter/Value Assignments>
      <Stimuli>
      <System Task>
endmodule
```

```
module adder(out,in1,in2);
    output  out;
    input   in1,in2    ;

    assign  out=in1 + in2;
endmodule
```

- Encapsulation makes the model available for instantiation in other modules

Port Declaration

- Port directions
  - Input port: **`input a;`**
  - Output port: **`output b;`**
  - Bidirectional port: **`inout c;`**
- Ports are declared as wires
  - **`input w;`** → w is a wire
  - Equivalent: **`input wire w;`**
- Output ports can be regs
  - **`output reg out;`**

```
module FA_co(co, a, b, ci);
    output co;
    input  a, b, ci;
    wire ab, bc, ac;
    and g0(ab, a, b);
    and g1(bc, b, ci);
    and g2(ac, ci, a);
    or  g3(co, ab, bc, ac);
endmodule
```

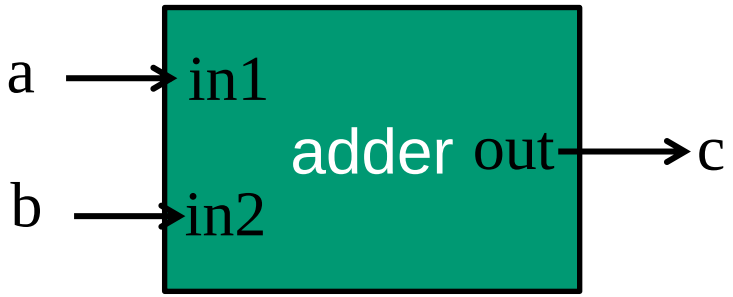
```
module FA_co(co, a, b, ci);
    output co;
    input  a, b, ci;
    …
endmodule
```

=

```
module FA_co(
    output co,
    input  a,
    input  b,
    input  ci);
    …
endmodule
```

Ports Connection

```
module adder (out,in1,in2);
  output out;
  input  in1 , in2;

  assign out = in1 + in2;
endmodule
```



- Connect module ports by *name* **(Recommended!!!!!)**
  - Usage: .PortName (NetName)
  - `adder adder_0 ( .out(C) , .in1(A) , .in2(B) );`
- Connect module ports by order list
  - `adder adder_1 ( C , A , B );    // C = A + B`
- Not fully connected
  - `adder adder_2 ( .out(C) , .in1(A) , .in2() );`
- Output ports can only be connected to **wire**

Vector

- wire and reg can be defined vector, default is 1bit
- vector is multi-bits element
- Format: [High#:Low#] or [Low#:High#]
  MSB LSB MSB LSB
- Using range specify part signals

```
wire        a;      // scalar net variable, default
wire [7:0]  bus;    // 8-bit bus
reg         clock;  // scalar register, default
reg [0:23]  addr;   // Vector register, virtual address 24 bits wide
```

```
bus[7]      // bit #7 of vector bus
bus[2:0]    // Three least significant bits of vector bus
            // using bus[0:2] is illegal because the significant bit should
            // always be on the left of a range specification
addr[0:1]   // Two most significant bits of vector addr
```

Width Width

a[7:0] ⇔ a[0+:8] ⇔ a[7-:8]

LSB MSB

Vector Concatenations

- A easy way to group vectors into a larger vector

Representation	Meanings
`{cout, sum}`	`{cout, sum}`
`{b[7:4],c[3:0]}`	`{b[7],b[6],b[5],b[4],c[3],c[2],c[1],c[0]}`
`{a,b[3:1],c,2'b10}`	`{a, b[3], b[2], b[1], c, 1'b1, 1'b0}`
`{4{2'b01}}`	`8'b01010101`
`wire [7:0] byte;` `{{8{byte[7]}},byte};`	**Sign extension**

Array

- Arrays are allowed in Verilog for reg, integer, time, and vector register data types.
- Format: [High#:Low#] name [Low#:High#]

```
integer     count[0:7];          // An array of 8 count variables
reg         bool[31:0];          // Array of 32 one-bit Boolean register variables
time        chk_ptr[1:100];      // Array of 100 time checkpoint variables
reg  [4:0]  port_id[0:7];        // Array of 8 port_id, each port_id is 5 bits wide
```

```
count[5]        // 5th element of array of count variables
chk_ptr[100]    // 100th time check point value
port_id[3]      // 3rd element of port_id array. This is a 5-bit value
```

Four-Valued Logic System

- Verilog's nets and registers hold four-valued data
  - 0 represent a logic **low** or **false** condition
  - 1 represent a logic **high** or **true** condition
  - z
    - Output of an undriven tri-state driver – high-impedance value
    - Models case where nothing is setting a wire's value
    - No connections!
  - x
    - Models when the simulator can't decide the value – uninitialized or unknown logic value
      - Initial state of registers
      - **When a wire is being driven to 0 and 1 simultaneously, BAD in pre-simulation (no delay in pre-simulation)**

Number Representation

- Format: <size>′<base_format><number>
  - <size> - decimal specification of number of bits
    - default is unsized and machine-dependent but at least 32 bits
  - <base format> - ' followed by arithmetic base of number
    - <d> <D> - decimal - default base if no <base_format> given
    - <h> <H> - hexadecimal
    - <o> <O> - octal
    - <b> <B> - binary
  - <number> - value given in base of <base_format>
    - _ can be used for reading clarity
    - If first character of sized, binary number 0, 1, x or z, will extend 0, 1, x or z (defined later!)

Number Representation

- Examples:
  - 6′b010_111 gives 010111
  - 8′b0110 gives 00000110
  - 4′bx01 gives xx01
  - 5′O36 gives 11_110
  - 6′d13 gives 001101
  - 16′H3AB gives 0000_0011_1010_1011
  - 16′Hx gives xxxxxxxxxxxxxxxx
  - 8′hz gives zzzzzzzz
  - 24 gives 0...0011000 **(default as 32-bit unsigned decimal)**

Compiler Directives

- ***`define***
  - `define RAM_SIZE 16
  - Defining a name and gives a constant value to it.
  - the identifier `RAM_SIZE will be replaced by 16
- ***`include***
  - `include adder.v
  - Including the entire contents of other verilog source file.
  - Can be replaced by specifying files to simulator in console
- ***`timescale***
  - `timescale 1ns/10ps
  - `timescale *<reference_time_unit>* / *<time_precision>*
  - Setting the reference time unit and time precision of your simulation.

System Tasks

- Displaying information
  - *$display*(“ID of the port is %b”, port_id);
    - ID of the port is 00101
- Monitoring information
  - *$monitor*($time, “Value of signals clk = %b rst = %b”, clk, rst);

```
0 Value of signals clk = 0 rst = 1
5 Value of signals clk = 1 rst = 1
10 Value of signals clk = 0 rst = 0
```

- Stopping and finishing in a simulation
  - *$stop*; // stop during a simulation
  - *$finish*; // terminates the simulator

System Tasks

- Open file
  - *f_in = $fopen("filepath", "mode");*
    - Integer f_in: file pointer
    - Mode: w/wb/r/rb/a/ab/…
- Read from file
  - *ret = $fscanf(f_in, "fmt", …);*
    - Integer ret: Normal(1)/Error(0)/EOF(-1)
    - Similar to **fscanf** in C
- Write to file
  - *$fwrite(f_in, "fmt", …);*
    - Similar to **fprintf** in C
- Read memory data
  - $readmem[b/h]("filepath", mem, [start], [end]);
    - Load data to mem[start:end]

System Tasks

- Dump waveform
  - $fsdbDumpfile("waveform_filepath", size_limit(in MB));
    - Specify waveform file to dump
  - $fsdbDumpvars(depth, instance, "option");
    - depth: all signal(0)/signal in current scope(1)/n-1 levels below(n)
    - instance: instance name(e.g., DUT)
    - Option:
      - "+all"
      - "+mda"
      - …

Outline

- Operators
- Definition of Register-Transfer Level (RTL)
- Continuous Assignments for Combinational Circuits
- Building Blocks of Sequential Circuits

Operators

Concatenation and replications	{,}
Bitwise	~, &, ~^, ^, |
Reduction	&, |, ^, ^~
Arithmetic	+ , - , * , / , %, **
Shift	>> , <<, >>>, <<<
Relational	< , <= , > , >=
Equality	== , != , === , !==
Logical	!, && , ||
Conditional	? :

Operator Precedence

	Operators
High	+, -, !, ~, &, |, ^~ (Unary)
	**
	*, /, %
	+, -
	<<, >>, <<<, >>>
	==, !=, ===, !==
	&, ^, |
	&&, ||
Low	?:

Concatenation & Replication

- Concatenation and Replication Operator ({})

Concatenation operator in LHS

```
module add_32 (co, sum, a, b, ci);
  output co;
  output [31:0] sum;
  input  [31:0] a, b;
  input  ci;
   assign #100 {co, sum} = a + b + ci;
endmodule
```

Bit replication to produce *01010101*

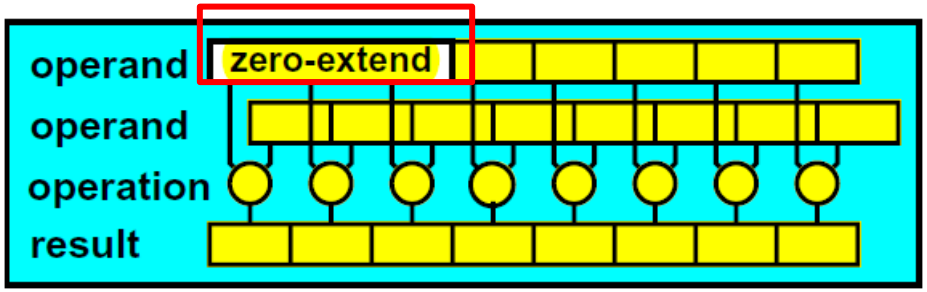
```
assign byte = {4{2'b01}};
```

Sign Extension

```
assign word = {{8{byte[7]}}, byte};
```

Bit-Wise Operators

- Bit-wise operators (`~`, `&`, `^`, `~^`, `|`) operate on each bit of a vector



Verilog fills in smaller-width operands by default using ***zero extension***!

```
$displayb(~4'b01zx );                              // 10xx
$displayb( 8'b01zx_01zx  & 8'b00001111 );   // 000001xx
$displayb( 8'b01zx_01zx  ^ 8'b00001111 );   // 01xx10xx
$displayb( 8'b01zx_01zx ~^ 8'b00001111 );   // 10xx01xx
$displayb( 8'b01zx_01zx  | 8'b00001111 );   // 01xx1111
```

Unary Reduction Operators

- The unary reduction operators (`&`, `|`, `^`) produce a 0, 1, or X scalar value.
  - `&a[3:0]` is the syntax sugar of `a[3] & a[2] & a[1] & a[0]`

```
$displayb( &4'b1110 );  // 0
$displayb( &4'b1111 );  // 1
$displayb( &4'b111z );  // x
$displayb( &4'b111x );  // x

$displayb( |4'b0000 );  // 0
$displayb( |4'b0001 );  // 1
$displayb( |4'b000z );  // x
$displayb( |4'b000x );  // x

$displayb( ^4'b1111 );  // 0
$displayb( ^4'b1110 );  // 1
$displayb( ^4'b111z );  // x
$displayb( ^4'b111x );  // x
```

```
$displayb( ^~4'b1111 );  // 1
$displayb( ^~4'b1110 );  // 0
$displayb( ^~4'b111z );  // x
$displayb( ^~4'b111x );  // x
```

Arithmetic Operators

- The arithmetic operators (*, /, %, +, -, **)
  - Produce numerical or unknown results
  - Integer division discards any remainder
  - If any operand bit has a value ”x”, the result of the expression is all ”x”

```
$display ( -3 * 5 );          // -15
$display ( -3 / 2 );          // -1
$display ( -3 % 2 );          // -1
$display ( -3 + 2 );          // -1
$display (  2 – 3 );          // -1
$displayh( 8'hfd / 8'd2 ); // 7e
$displayb( 2 * 2'b1x );     // xxxxxxxx
```

Singed for Arithmetic Operators

```
module test;
reg  [3:0] A,B;
wire [4:0] sum;

assign sum = A+B;

initial begin
  #5 A=5;  B=-2;
  $display(" A  B  sum");
  $display("%d %d %d",A,B,sum);
end

endmodule
```

A	B	Sum
5	14	19
00101	01110	10011
	↑ unsigned	↑ unsigned

```
reg signed [3:0] A,B;
assign sum= A+B;
```
←Recommended

```
reg [3:0] A,B;
assign sum = $signed(A)+$signed(B);
```

```
module test;
reg  signed [3:0] A,B;
wire signed [4:0] sum;

assign sum = A+B;

initial begin
  #5 A=5;  B=-2;
  $display(" A  B  sum");
  $display("%d %d %d",A,B,sum);
end

endmodule
```

A	B	Sum
5	-2	3
00101	11110	00011

Bit Length of Arithmetic Operations

- (Signed or Unsigned) addition bit length
  - A(8 bits) + B(8 bits) → C(8+1 bits)
  - A(M bits) + B (N bits) → C(max(M, N)+1 bits)

```
wire [7:0] A, B;
wire [8:0] C;
assign C = {A[7], A} + {B[7], B};
assign C = $signed(A) + $ signed(B);
```

```
wire signed [7:0] A, B;
wire signed [8:0] C;
assign C = A+B;
```

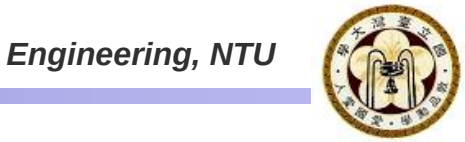
- (Signed or Unsigned) multiplication bit length
  - A(3 bits) x B(5 bits) → C(3 + 5 bits)

```
wire signed [2:0];
wire signed [4:0];
wire signed [7:0];
assign C = A * B;
```

Ref: https://zipcpu.com/dsp/2017/07/21/bit-growth.html

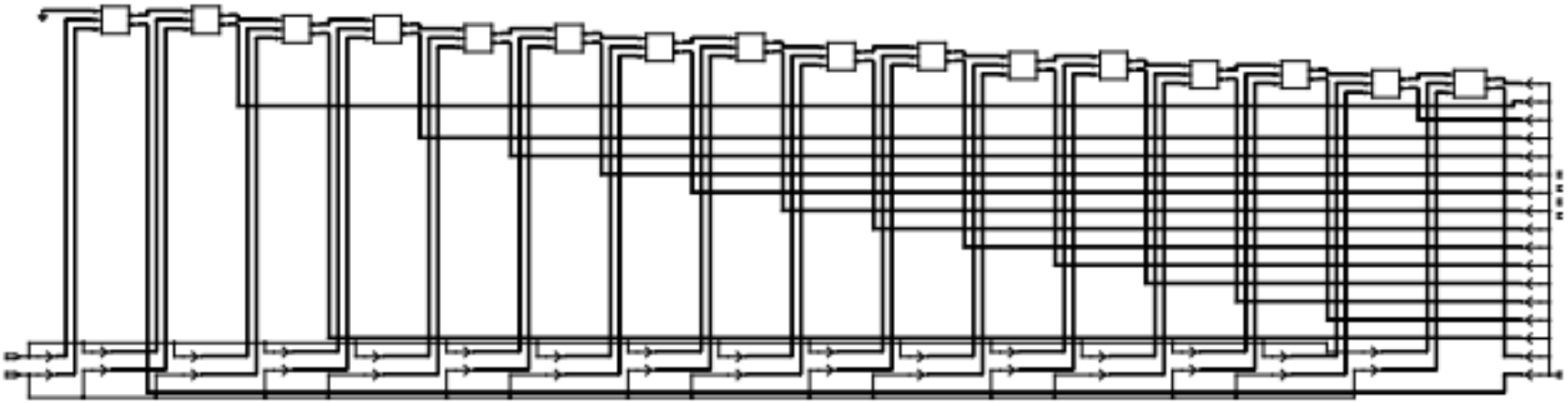
Arithmetic Operators (cont.)

- The logic gate realization depends on several variables
  - coding style
  - synthesis tool used
  - synthesis constraints (more later on this)
- So, when we say ”+”, is it a...
  - ripple-carry adder
  - look-ahead-carry adder (how many bits of lookahead to be used?)
  - carry-save adder
- When writing RTL code, keep in mind what will eventually be needed. Continually thinking about structure, timing, size, power



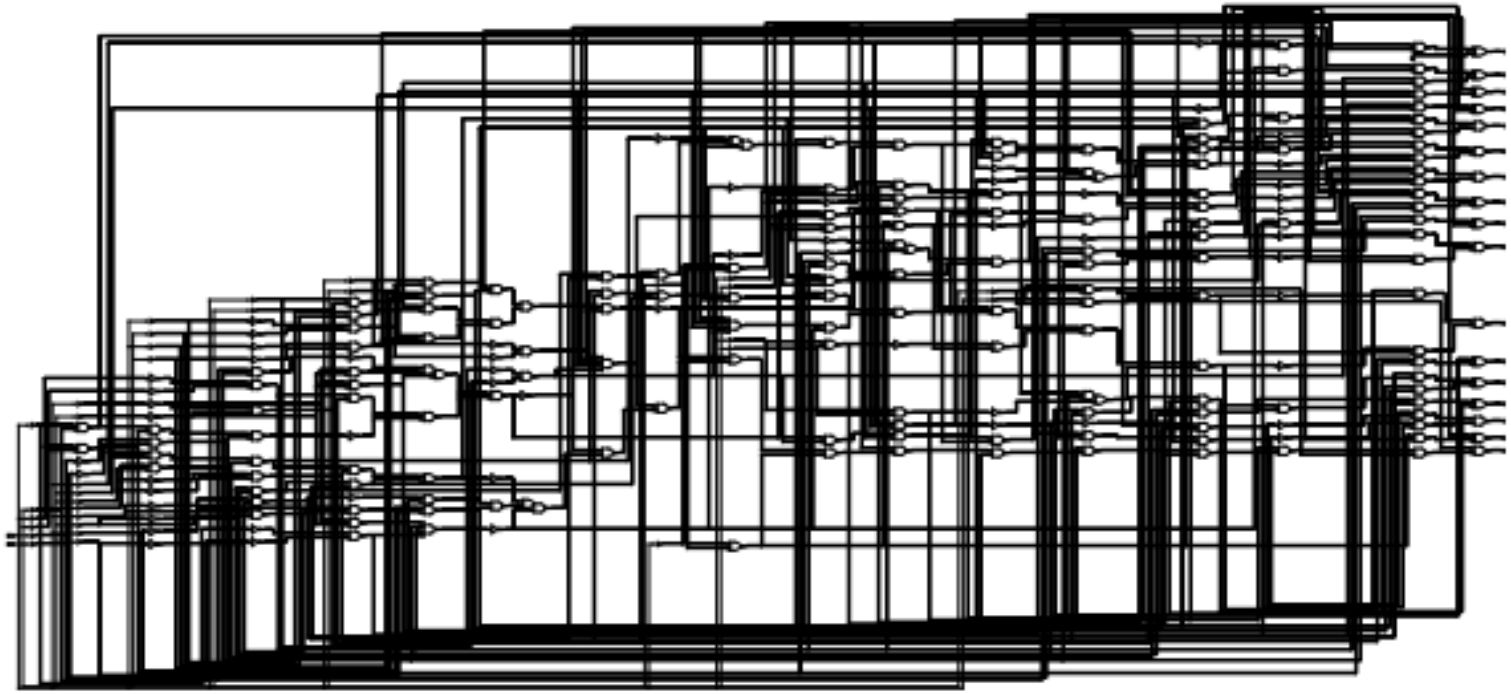
Arithmetic Operators (cont.)

- 16-bit adder with **loose** constraints:
- set_max_delay 2 [get_ports sum*]
- max delay = 0.8ns, area = 472 = **85** gates



Arithmetic Operators (cont.)

- 16-bit adder with **tighter** constraints:
- set_max_delay 0.5 [get_ports sum*]
- max delay = 0.5ns, area = 2038 = **368** gates



Shift Operators

- Shift operator
  - **“>>”** logical shift right, **“<<“** logical shift left
  - **“>>>”** arithmetic shift right, **“<<<“** arithmetic shift left
- Treat right operand as **unsigned**

```
$displayb ( 8'b00010011 << 2    );   // 01001100
$displayb ( 8'b00010011 >> 2    );   // 00000100
$displayb ( 8'b00010011 >> -2   );   // 00000000
$displayb ( 8'b00010011 >> 1'bx );   // xxxxxxxx

$displayb ( 8'b00010011 << -2'd3);   // 00100110
```

Negation Operators

- The logical negation operator (**!**)
  - produces a 0, 1, or X scalar value.
  - An operand is logically false if **all** of its bits are 0
  - An operand is logically true if **any** of its bits are 1

```
$displayb ( !4'b0100 );  // 0
$displayb ( !4'b0000 );  // 1
$displayb ( !4'b00z0 );  // x
$displayb ( !4'b000x );  // x
```

Relational Operators

- Relational operators (<, <=, >=, >)
  - Outputs unknown x if any of the operands has unknown x bits

```
$displayb ( 4'b1010 <  4'b0110 );  // 0
$displayb ( 4'b0010 <= 4'b0010 );  // 1
$displayb ( 4'b1010 <  4'b0x10 );  // x
$displayb ( 4'b0010 <= 4'b0x10 );  // x
$displayb ( 4'b1010 >= 4'b1x10 );  // x
$displayb ( 4'b1x10 >  4'b1x10 );  // x
$displayb ( 4'b1z10 >  4'b1z10 );  // x
```

Equality Operators

- Equality operators
  - "==" , "!="does not perform a definitive match for Z or X
    - synthesizable
  - "===" , "!=="does perform a definitive match for Z or X
    - not synthesizable

```
$displayb ( 4'b0011 == 4'b1010 );  // 0
$displayb ( 4'b0011 != 4'b1x10 );  // 1
$displayb ( 4'b1010 == 4'b1x10 );  // x
$displayb ( 4'b1x10 == 4'b1x10 );  // x
$displayb ( 4'b1z10 == 4'b1z10 );  // x

$displayb ( 4'b01zx === 4'b01zx );  // 1
$displayb ( 4'b01zx !== 4'b01zx );  // 0
$displayb ( 4'b01zx === 4'b00zx );  // 0
$displayb ( 4'b01zx !== 4'b11zx );  // 1
```

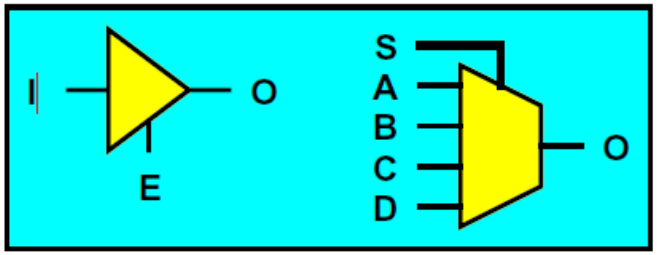
Logical Operators

- Logical operators (&&, ||)
  - An operand is logically false if **all** of its bits are 0
  - An operand is logically true if **any** of its bits are 1

```
$displayb ( 2'b00 && 2'b10 );  // 0
$displayb ( 2'b01 && 2'b10 );  // 1
$displayb ( 2'b0z && 2'b10 );  // x
$displayb ( 2'b0x && 2'b10 );  // x
$displayb ( 2'b1x && 2'b1z );  // 1

$displayb ( 2'b00 || 2'b00 );  // 0
$displayb ( 2'b01 || 2'b00 );  // 1
$displayb ( 2'b0z || 2'b00 );  // x
$displayb ( 2'b0x || 2'b00 );  // x
$displayb ( 2'b0x || 2'b0z );  // x
```

Conditional Operators

- Conditional Operator
  - Usage: ***conditional_expression ? true_expression: false_expression;***



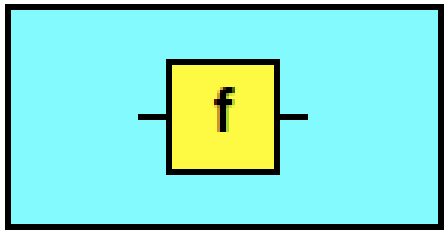

```
module driver(O,I,E);
output O; input I,E;
  assign O = E ? I : 'bz;
endmodule

module mux41(O,S,A,B,C,D);
output O;
input A,B,C,D; input [1:0] S;
  assign O = (S == 2'h0) ? A :
             (S == 2'h1) ? B :
             (S == 2'h2) ? C : D;
endmodule
```

Outline

- Operators
- Definition of Register-Transfer Level (RTL)
- Continuous Assignments for Combinational Circuits
- Building Blocks of Sequential Circuits

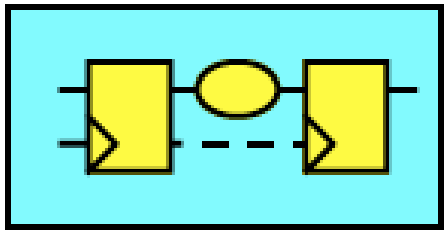
Levels of Modeling

Behavioral Level



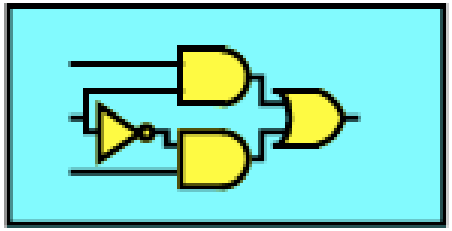

```
initial begin
    #(`CYCLE * `End_CYCLE );
    $display( "\n" );
    $display("-------------------------------------------------------\n");
    $display("Error!!! Something is wrong with your code ...!\n");
    $display("------------------------FAIL-------------------------\n");
    $display( "Terminated at: ", $time, " ns" );
    $display( "\n" );
    $stop;
end
```

Register Transfer Level (RTL)



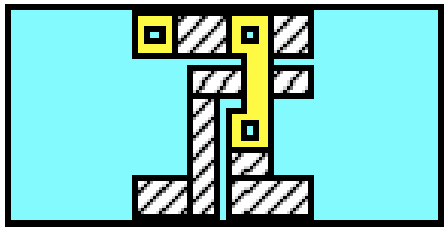
```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    assign  out=sel?in1:in2;
endmodule
```

Structural/Gate Level



```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or  o1(out,a1_o,a2_o);
endmodule
```

Transistor/Physical Level



```
module inv(out, in);
// port declaration
  output  out;
  input   in;
// declare power and ground
  supply1  pwr;
  supply0  gnd;
// Switch level description
  pmos  S0(out, pwr, in);
  nmos  S1(out, gnd, in);
endmodule
```

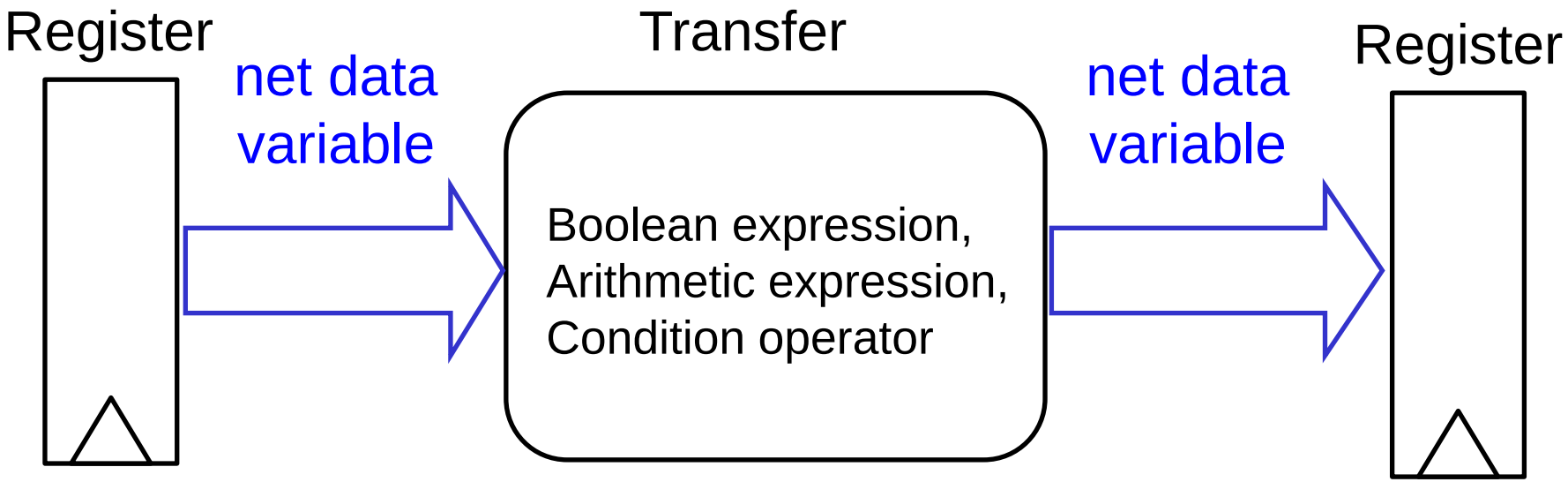
What is Register Transfer Level?

- In integrated circuit design, Register Transfer Level (RTL) description is a way of describing the operation of a synchronous digital circuit.

- In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between synchronous registers, and the logical operations performed on those signals.

Description at RT-Level

- Register (usually D Flip-Flops, **sequential**)
- Transfer (operation, **combinational**)



Procedures to Design at RT-Level

- Design partition
  - Sequential circuit (register part)
  - Combinational circuit (transfer part)
- Net declaration
  - I/O ports
  - Net variable
- Transfer Circuit design
  - Logical operator
  - Arithmetical operator
  - Conditional operator

```
1  /*=======================================
2  Author: Yu Chuan, Chuang
3  Module: Counter
4  Description:
5  When getting start_i signal, counter starts
6  to count from 0 to 15.
7  =======================================*/
8  module counter (
9      input            clk,
10     input            rst,
11     input            start_i,
12     output  [3:0]    count_o
13 );
14
15 //============  Parameter ===============
16     localparam  STATE_IDLE  = 1'b0;
17     localparam  STATE_CNT   = 1'b1;
18
19 //=========  Reg/Wire Declaration ======
20     reg         state, nxt_state;
21     reg [3:0]   cnt, nxt_cnt;
22
23 //========= Finite State Machine =======
24     always@(posedge clk or posedge rst) begin
25         if(rst)
26             state <= STATE_IDLE;
27         else
28             state <= nxt_state;
29     end
30
31     always@(*) begin
32         case(state)
33             STATE_IDLE: begin
34                 if(start_i)
35                     nxt_state = STATE_CNT;
36                 else
37                     nxt_state = STATE_IDLE;
38             end
39             STATE_CNT: begin
40                 if(cnt == 4'd15)
41                     nxt_state = STATE_IDLE;
42                 else
43                     nxt_state = STATE_CNT;
44             end
45             default: nxt_state = STATE_IDLE;
46         endcase
47     end
48
49 //============ Combinational ============
50     assign count_o = cnt;
51
52     always@(*) begin
53         if(state == STATE_CNT) begin
54             nxt_cnt = cnt + 1;
55         end
56         else begin
57             nxt_cnt = 0;
58         end
59     end
60
61 //============== Sequential =============
62     always@(posedge clk or posedge rst) begin
63         if(rst)
64             cnt <= 0;
65         else
66             cnt <= nxt_cnt;
67     end
68
69 endmodule
```

Header/Comment

Module instantiation

Input/output declaration

Parameter

Number Representation

Reg/WIre

If statement

Procedure Block

case statement

FSM

Combinational

Continuous Assignment

Sensitivity list

Operator

Procedure Assignment

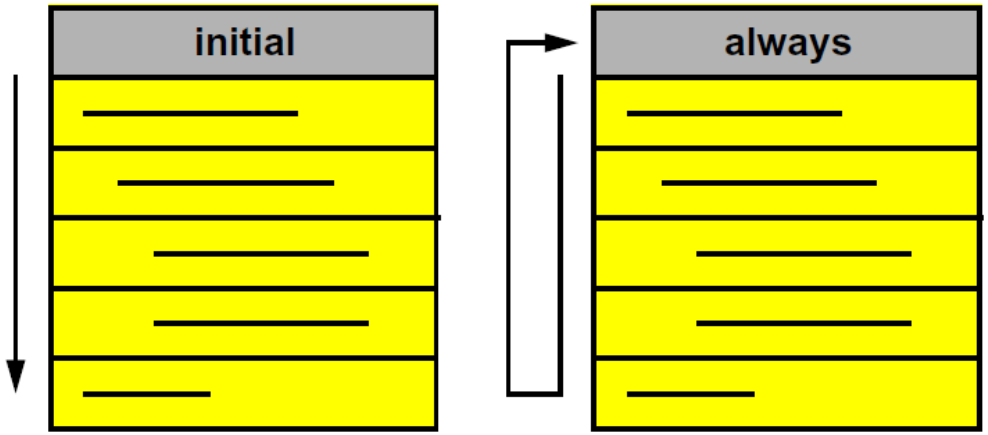
Sequential

Outline

- Definition of Register-Transfer Level (RTL)
- Verilog Syntax of RTL
  - Operators
- Continuous Assignments for Combinational Circuits
- Building Blocks of Sequential Circuits

Procedures and Assignments

- Procedure block: a group of statements that appear between a begin and an end
  - **`initial, always, task, function`**
  - considered as a statement
  - Procedures execute concurrently with other procedures



- The simulator starts executing all procedure blocks at time 0
- The simulator executes all procedural blocks **concurrently !**
- The simulator executes **initial** blocks once and **always** blocks continually

Assignments (1/2)

- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to wire type
  - procedural assignment, which assigns values to reg type
- Basic form

Assignments	Description	Left Hand Side	Example
Continuous Assignment	appear outside procedures	wire	wire a; assign a = 1'b1;
Procedural Assignment	appear inside procedures	reg	reg a; always@(*) a = 1'b1;

P.S. Left hand side (LHS) = Right hand side (RHS)

✓ ***Synthesizable!***

Assignments (2/2)

- Continuous assignment

```
module holiday_1(sat, sun, weekend);
      input sat, sun; output weekend;
      assign weekend = sat | sun;      // outside a procedure
endmodule
```

- Procedural assignment

```
module holiday_2(sat, sun, weekend);
      input sat, sun; output weekend; reg weekend;
      always @(*) weekend = sat | sun;      // inside a procedure
endmodule
```

```
module assignments
        // continuous assignments go here
always begin
        // procedural assignments go here
end
endmodule
```

Continuous Assignments (1/4)

- The LHS of an assignment must always be a scalar or vector **wire** or a concatenation of scalar and vector **wire**. It cannot be a scalar or vector **reg**.
- Continuous assignments are always active. Any changes in RHS of the continuous assignment are evaluated and the LHS is updated.
- The operands on the RHS can be reg or wire.
- Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. (*Not Synthesizable*)

Continuous Assignments (2/4)

- Convenient for logical or datapath specifications

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Define bus widths

Continuous assignment: permanently sets the value of sum to be a+b+carryin

Recomputed when a, b, or carryin changes

Continuous Assignments (3/4)

- Continuous assignments provide a way to model combinational logic

continuous assignment

```
module inv_array(out,in);
  output [31:0] out;
  input [31:0] in;
  assign out=~in;
endmodule
```

gate-level modeling

```
module inv_array(out,in);
  output [31:0] out;
  input [31:0] in;
  not U1(out[0],in[0]);
  not U2(out[1],in[1]);
  ..
  not U31(out[31],in[31]);
endmodule
```

Continuous Assignments (4/4)

- Implicit Continuous Assignments
  - Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.

```
// Regular
wire out;
assign out = in1 & in2;

// implicit continuous assignment
wire out = in1 & in2
```

```
// implicit net declaration
wire in1, in2;
assign out = in1 & in2;
```

Not Recommended!!!!!

- Implicit Net Declaration
  - If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name

Assignment Delays

- Regular Assignment Delay

  wire out;

  assign #10 out = in1 & in2; // delay in a continuous assign

- Implicit Continuous Assignment Delay

  wire #10 out = in1 & in2; **Not Recommended!!!!!**

- Net Declaration Delay

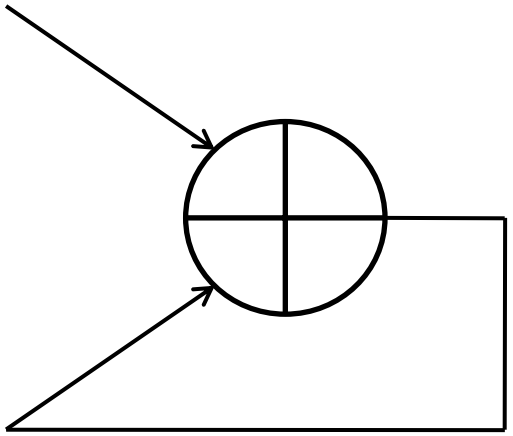
  wire # 10 out; // net delays

  assign out = in1 & in2;

Avoiding Combinational Loops

- Avoid combinational loops (or logic loops)
  - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
  - Example

```
wire [3:0] a;
wire [3:0] b;

assign a = b + a;
```



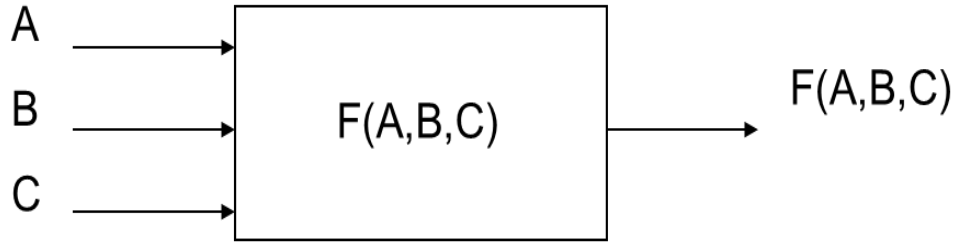
SW language is ok, BUT not in HW language !!!!!

Outline

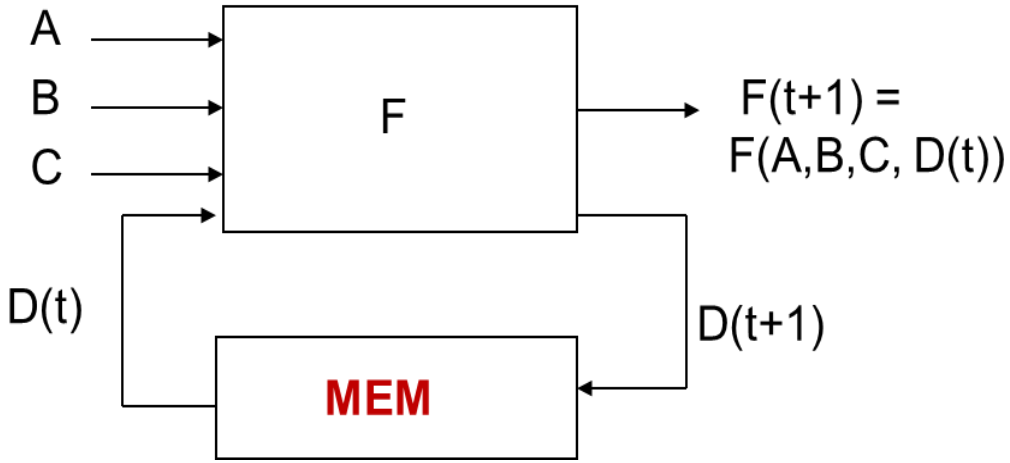
- Definition of Register-Transfer Level (RTL)
- Verilog Syntax of RTL
  - Operators
- Continuous Assignments for Combinational Circuits
- Building Blocks of Sequential Circuits

Combinational & Sequential Circuits

- Combinational Circuits
  - Without memory
  - output only depends on current input

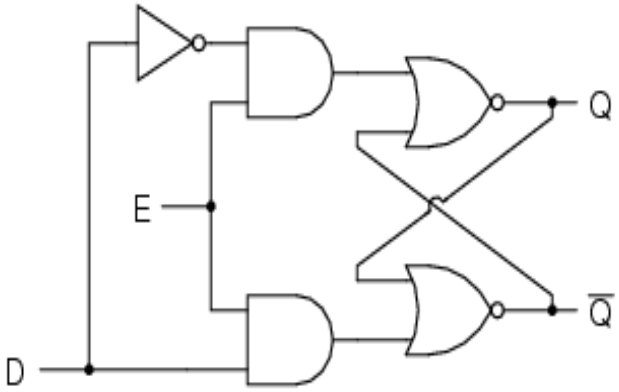


- Sequential Circuits
  - With memory
  - output depends on current input and previous stored value



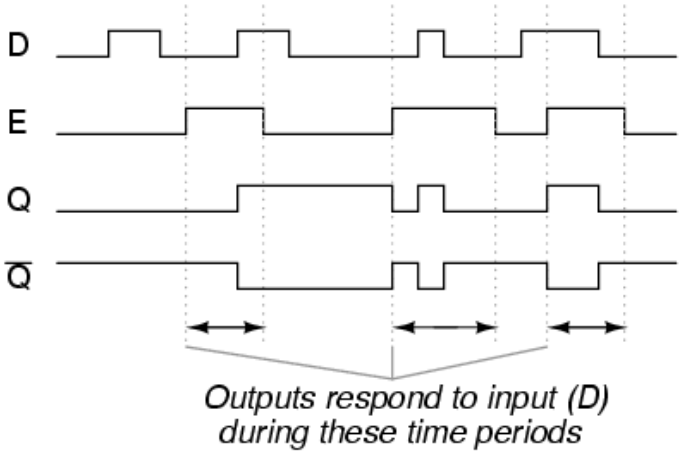
Latch

- Level-sensitive register
- Most of designs should not infer latches
  - Most industry chip designs follow a “synchronous” design methodology
  - Latch-based designs are susceptible to timing problems
  - Glitches in the enable pin of the latches that can cause unrecoverable failure (transparent)
  - **Inferred latches after synthesis due to bad coding styles!**

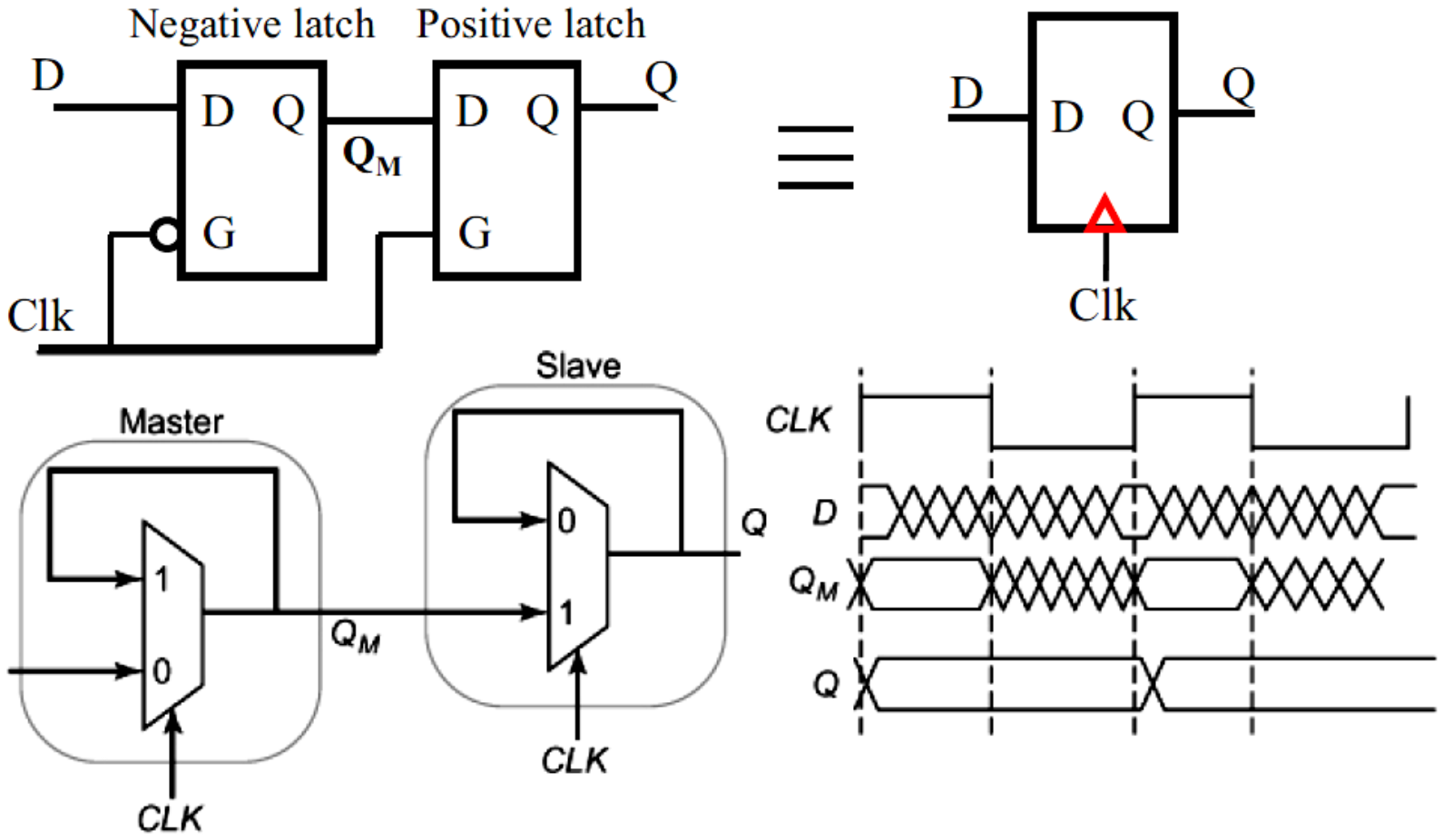


E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

Regular D-latch response



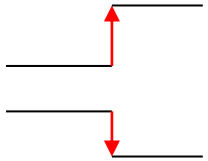
Master-Slave D Flip-Flop

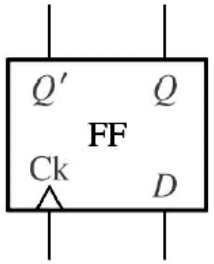


Edge-triggered D Flip-Flop

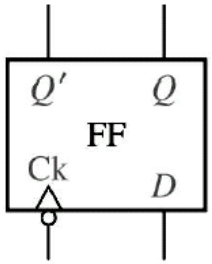
- Positive (Rising edge) trigger
- Negative (Falling edge) trigger

→to align with clock edges





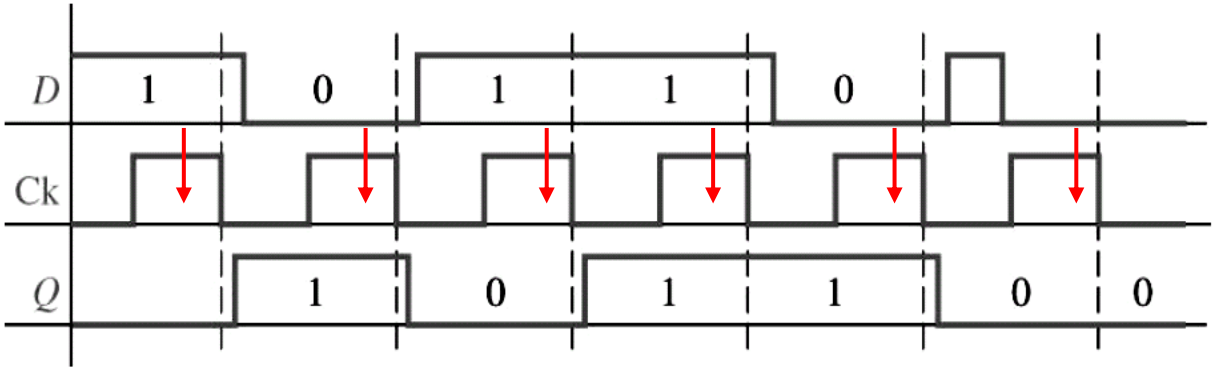
(a) Rising-edge trigger



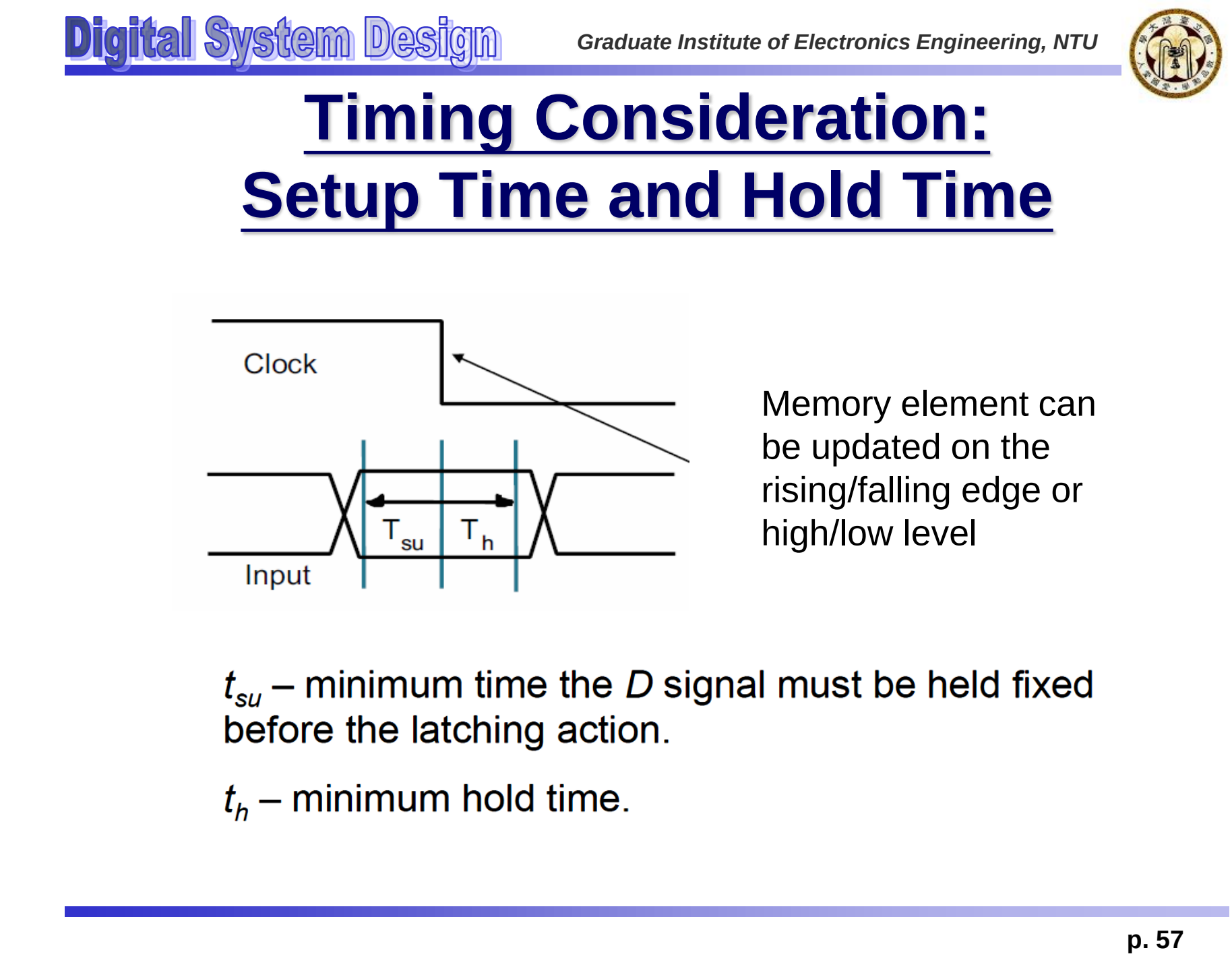
(b) Falling-edge trigger

D Q	Q^+
0 0	0
0 1	0
1 0	1
1 1	1

$Q^+ = D$



Timing for D Flip-Flop ***(Falling-Edge Trigger)***



Timing Consideration: Setup Time and Hold Time

Memory element can be updated on the rising/falling edge or high/low level

t_{su} – minimum time the *D* signal must be held fixed before the latching action.

t_h – minimum hold time.