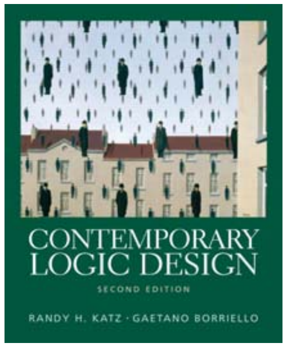
CSE 370 Spring 2006 Introduction to Digital Design

Lecture 15: Sequential Verilog



Last Lecture
- Latches
- Flip-flops

Today
- Timing Methodology
- Sequential Verilog

Administrivia

- Turn in HW#5
- Lab #6 on the web

Quiz #2

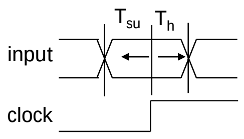
Quiz #2

Timing methodologies

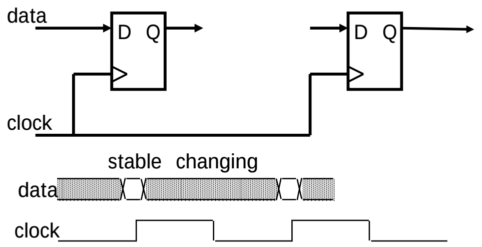
- Rules for interconnecting components and clocks
  - guarantee proper operation of system when strictly followed
- Approach depends on building blocks used for memory elements
  - we'll focus on systems with edge-triggered flip-flops
    - found in programmable logic devices
  - many custom integrated circuits focus on level-sensitive latches
- Basic rules for correct timing:
  - (1) correct inputs, with respect to time, are provided to the flip-flops
  - (2) no flip-flop changes state more than once per clocking event

Timing methodologies (cont'd)

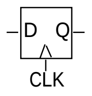
- Definition of terms
  - clock: periodic event, causes state of memory element to change can be rising edge or falling edge or high level or low level
  - setup time: minimum time before the clocking event by which the input must be stable (Tsu)
  - hold time: minimum time after the clocking event until which the input must remain stable (Th)

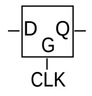


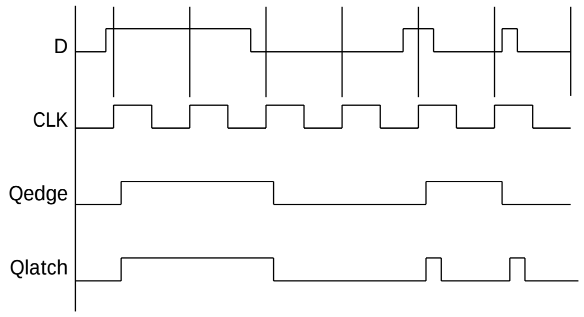
there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized



Comparison of latches and flip-flops


positive edge-triggered flip-flop


transparent (level-sensitive) latch



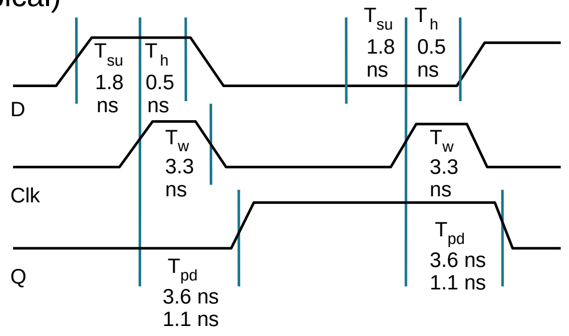
behavior is the same unless input changes while the clock is high

Comparison of latches and flip-flops (cont'd)

Type	When inputs are sampled	When output is valid
unclocked latch	always	propagation delay from input change
level-sensitive latch	clock high (Tsu/Th around falling edge of clock)	propagation delay from input change or clock edge (whichever is later)
master-slave flip-flop	clock high (Tsu/Th around falling edge of clock)	propagation delay from falling edge of clock
negative edge-triggered flip-flop	clock hi-to-lo transition (Tsu/Th around falling edge of clock)	propagation delay from falling edge of clock

Typical timing specifications

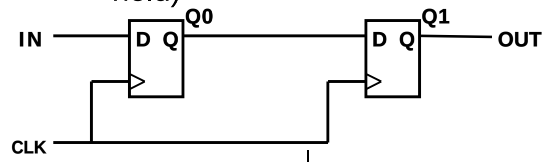
- Positive edge-triggered D flip-flop
  - setup and hold times
  - minimum clock width
  - propagation delays (low to high, high to low, max and typical)

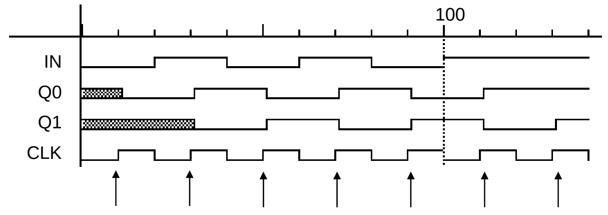


all measurements are made from the clocking event (the rising edge of the clock)

Cascading edge-triggered flip-flops

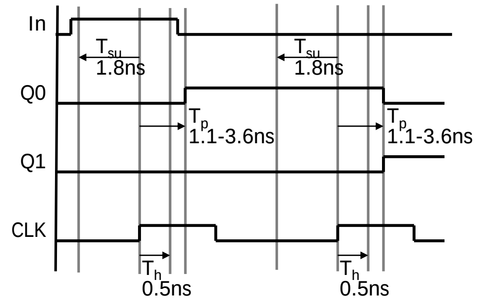
- Shift register
  - new value goes into first stage
  - while previous value of first stage goes into second stage
  - consider setup/hold/propagation delays (prop must be > hold)





Cascading edge-triggered flip-flops (cont'd)

- Why this works
  - propagation delays exceed hold times
  - clock width constraint exceeds setup time
  - this guarantees following stage will latch current value before it changes to new value



timing constraints guarantee proper operation of cascaded components

assumes infinitely fast distribution of the clock

Variables

- wire
  - Connects components together
- reg
  - Saves a value
    - Part of a behavioral description
  - Does *NOT* necessarily become a register when you synthesize
    - May become a wire
- The rule
  - Declare a variable as reg if it is a target of an assignment statement
    - Continuous assign doesn't count

Sequential Verilog

- Sequential circuits: Registers & combinational logic
  - Use positive edge-triggered registers
  - Avoid latches and negative edge-triggered registers
- Register is triggered by “posedge clk”

Example: A D flip-flop

```
module register(Q, D, clock);
  input   D, clock;
  output  Q;
  reg     Q;

  always @(posedge clock) begin
     Q = D;
  end
endmodule
```

A real register. Holds *Q* between clock edges

always block

- A procedure that describes a circuit’s function
  - Can contain multiple statements
  - Can contain *if, for, while, case*
  - Triggers at the specified conditions
  - *begin/end* groups statements within *always* block

```
module register(Q, D, clock);
  input   D, clock;
  output  Q;
  reg     Q;

  always @(posedge clock) begin
     Q = D;
  end
endmodule
```

always example

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

  always @(in1 or in2) begin
     out = in1 & in2;
  end
endmodule
```

Not a real register!! Holds assignment in always block

The compiler will not synthesize this code to a register, because *out* changes whenever *in1* or *in2* change. Can instead simply write

```
wire out, in1, in2;
and (out, in1, in2);
```

specifies when block is executed i.e. triggered by changes in *in1* or *in2*

Incomplete trigger or incomplete assignment

- What if you omit an input trigger (e.g. *in*2)
  - Compiler will insert a register to hold the state
  - Becomes a sequential circuit — *NOT* what you want

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

  always @(in1) begin
     out = in1 & in2;
  end
endmodule
```

A real register!! Holds *out* because *in2* isn’t specified in *always* trigger

2 rules:

1) Include all inputs in the trigger list
2) Use complete assignments
   - ⇒ Every path must lead to an assignment for *out*
   - ⇒ Otherwise *out* needs a state element

Another way: Use functions

- Functions for combinational logic
  - Functions can't have state

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

  assign out = myfunction(in1, in2);
  function myfunction;
    input in1, in2;
    begin
      myfunction = in1 & in2;
    end
  endfunction
endmodule
```

Benefits:
Functions force a result
⇒ Compiler will fail if function does not generate a result
⇒ If you build a function wrong the circuit will not synthesize. If you build an always block wrong you get a register

if

- Same as C *if* statement

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;     // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;               // target of assignment

  always @(sel or A or B or C or D)
    if (sel == 2'b00) Y = A;
    else if (sel == 2'b01) Y = B;
    else if (sel == 2'b10) Y = C;
    else if (sel == 2'b11) Y = D;
endmodule
```

⇒ Single *if* statements synthesize to multiplexers
⇒ Nested *if* / *else* statements usually synthesize to logic

if (another way)

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;     // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;               // target of assignment

  always @(sel or A or B or C or D)
    if (sel[0] == 0)
      if (sel[1] == 0)  Y = A;
      else              Y = B;
    else
      if (sel[1] == 0)  Y = C;
      else              Y = D;
endmodule
```

case

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;     // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;               // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

case executes sequentially
⇒ First match executes
⇒ Don't need to break out of *case*
case statements synthesize to muxes

case (another way)

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;      // 2-bit control signal
input A, B, C, D;
output Y;

 assign out = mymux(sel, A, B, C, D);
 function mymux;
    input [1:0] sel, A, B, C, D;
    begin
        case (sel)
          2'b00: mymux = A;
          2'b01: mymux = B;
          2'b10: mymux = C;
          2'b11: mymux = D;
        endcase
    end
 endfunction
endmodule
```

Note: You can define a function in a file
Then *include* it into your Verilog module

default case

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;              // 8-bit input vector
output [2:0] Y;              // 3-bit encoded output
reg    [2:0] Y;              // target of assignment

  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:  Y = 3'bx; // Don't care about other cases
    endcase
endmodule
```

If you omit the *default*, the compiler will create a latch for Y
⇒ Either list all 256 cases
⇒ Or use a function (compiler will warn you of missing cases)

case executes sequentially

```
// Priority encoder
module encode (A, Y);
input  [7:0] A;              // 8-bit input vector
output [2:0] Y;              // 3-bit encoded output
reg    [2:0] Y;              // target of assignment

  always @(A)
    case (1'b1)
      A[0]: Y = 0;
      A[1]: Y = 1;
      A[2]: Y = 2;
      A[3]: Y = 3;
      A[4]: Y = 4;
      A[5]: Y = 5;
      A[6]: Y = 6;
      A[7]: Y = 7;
      default:  Y = 3'bx;   // Don't care when input is all 0's
    endcase
endmodule
```

Case statements execute sequentially
⇒ Take the first alternative that matches

for

```
// simple encoder
module encode (A, Y);
input  [7:0] A;       // 8-bit input vector
output [2:0] Y;       // 3-bit encoded output
reg    [2:0] Y;       // target of assignment
integer i;            // Temporary variables for program
reg    [7:0] test;

  always @(A) begin
    test = 8b'00000001;
    Y = 3'bx;
    for (i = 0; i < 8; i = i + 1) begin
       if (A == test) Y = i;
       test = test << 1;  // Shift left, pad with 0s
    end
  end
endmodule
```

for statements synthesize as cascaded combinational logic
⇒ Verilog unrolls the loop

Verilog *while/repeat/forever*

- *while* (expression) statement
  - execute statement while expression is true
- *repeat* (expression) statement
  - execute statement a fixed number of times
- *forever* statement
  - execute statement forever

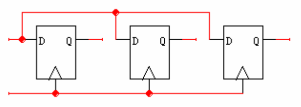
Blocking and non-blocking assignments

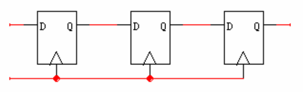
- Blocking assignments (Q = A)
  - Variable is assigned immediately
  - New value is used by subsequent statements
- Non-blocking assignments (Q <= A)
  - Variable is assigned after all scheduled statements are executed
  - Value to be assigned is computed but saved for later
- Example: Swap

```
always @(posedge CLK)          always @(posedge CLK)
   begin                          begin
      temp = B;                      A <= B;
      B = A;                         B <= A;
      A = temp;                   end
   end
```

Blocking and non-blocking assignments

```
reg B, C, D;                   reg B, C, D;
always @(posedge clk)          always @(posedge clk)
   begin                          begin
      B = A;                         B <= A;
      C = B;                         C <= B;
      D = C;                         D <= C;
   end                            end
```





Swap

- The following code executes incorrectly
  - One block executes first
  - Loses previous value of variable

```
always @(posedge CLK)          always @(posedge CLK)
   begin                          begin
      A = B;                         B = A;
   end                            end
```

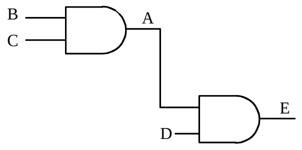
- Non-blocking assignment fixes this
  - Both blocks are scheduled by posedge CLK

```
always @(posedge CLK)          always @(posedge CLK)
   begin                          begin
      A <= B;                        B <= A;
   end                            end
```

Parallel versus serial execution

- **`assign`** statements are implicitly parallel
  - "=" means continuous assignment
  - Example

```
        assign E = A & D;
        assign A = B & C;
```

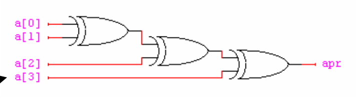


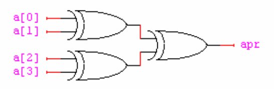
  - **A** and **E** change if **B** changes
- **`always`** blocks execute in parallel
  - **`always @(posedge clock)`**
- Procedural block internals not necessarily parallel
  - "=" is a blocking assignment (sequential)
  - "<=" is a nonblocking assignment (parallel)
  - Examples of procedures: **`always`**, **`function`**, etc.

Synthesis examples

```
wire [3:0] x, y, a, b, c, d;

assign apr = ^a;
assign y = a & ~b;
assign x = (a == b) ?
            a + c : d + a;
```


a[0]
a[1]
a[2]
a[3]
apr


a[0]
a[1]
a[2]
a[3]
apr