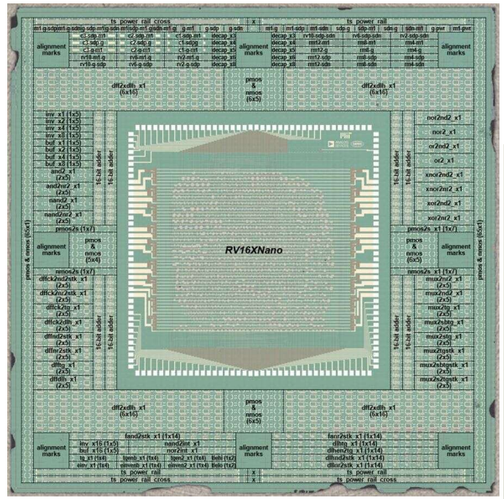
EECS151 : Introduction to Digital Design and ICs

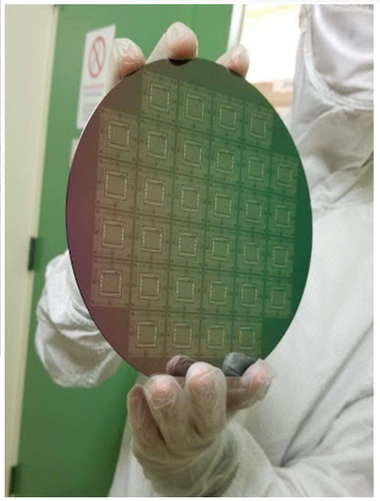
Lecture 4 – Verilog II

Bora Nikolić and Sophia Shao

MIT Builds Carbon Nanotube FET Based RISC-V Microprocessor (EE Times, Sept 9, 2019)



Microscopy image of a full fabricated RV16X-NANO die. The processor core is in the middle of the die, with test circuitry surrounding the perimeter (Image: Nature)



RV16X-NANO 150-mm wafer built from complementary carbon nanotube transistors. Each wafer includes 32 dies

Review

- The design flow involves translating an abstracted RTL design into physical logic gates, and verifying that it has been done correctly
- Verilog is commonly used to describe RTL
  - Structural or
  - Behavioral

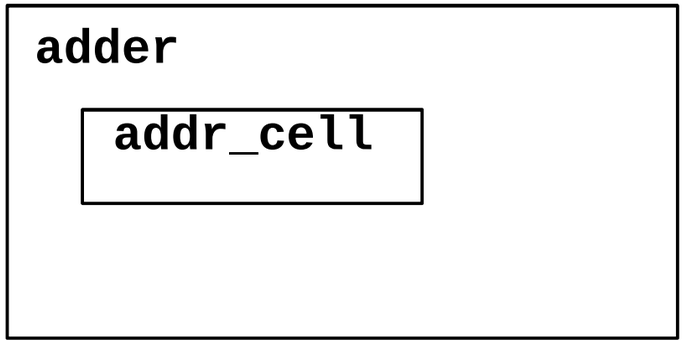
Verilog

Verilog Introduction

- A **`module`** definition describes a component in a circuit
- Two ways to describe module contents:
  - **Structural Verilog**
    - List of sub-components and how they are connected
    - Just like schematics, but using text
    - tedious to write, hard to decode
    - You get precise control over circuit details
    - May be necessary to map to special resources of the FPGA/ASIC
  - **Behavioral Verilog**
    - Describe what a component does, not how it does it
    - Synthesized into a circuit that has this behavior
    - Result is only as good as the tools
- Build up a hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.



name

port list

```
module addr_cell (a, b, cin, s, cout);
   input     a, b, cin;
   output    s, cout;

endmodule

module adder (A, B, S);

addr_cell ac1 (    ... connections ...    );

endmodule
```

keywords

port declarations (input, output, or inout)

module body

Instance of addr_cell

Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure.

Structural Model Example- XOR from AND, OR, NOT

module name

port list

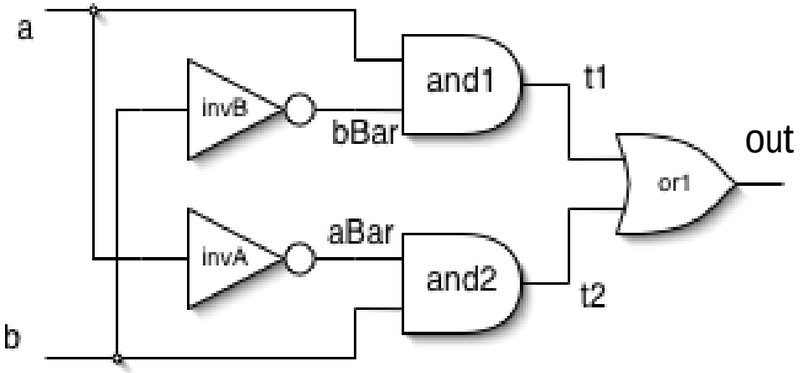
port declarations

internal signal declarations

Built-in gates

instances

```
module xor_gate ( out, a, b );
  input     a, b;
  output    out;
  wire      aBar, bBar, t1, t2;

  not invA (aBar, a);
  not invB (bBar, b);
  and and1 (t1, a, bBar);
  and and2 (t2, b, aBar);
  or  or1 (out, t1, t2);

endmodule
```

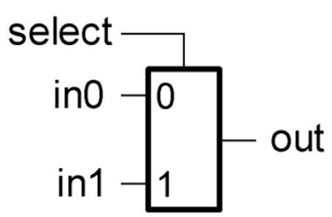


Interconnections (note output is first)

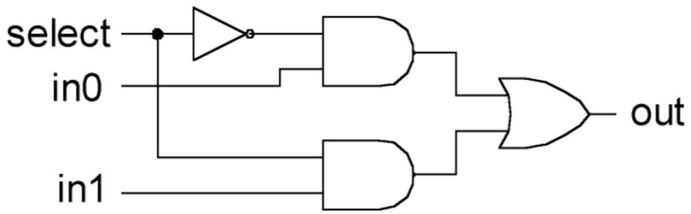
Instance name

- Notes:
  - The instantiated gates are not “executed”. They are active always.
  - xor gate already exists as a built-in (so really no need to define it).
  - Undeclared variables assumed to be wires. Don’t let this happen to you!

Structural Example: 2-to1 mux



a) 2-input mux symbol



b) 2-input mux gate-level circuit diagram

```
/* 2-input multiplexer in gates */
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;

   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);

endmodule // mux2
```

C++ style comments

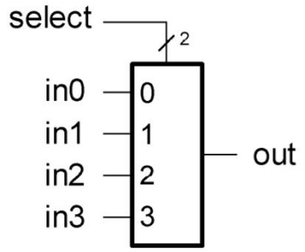
Built-ins don't need instance names

Multiple instances can share the same "master" name.

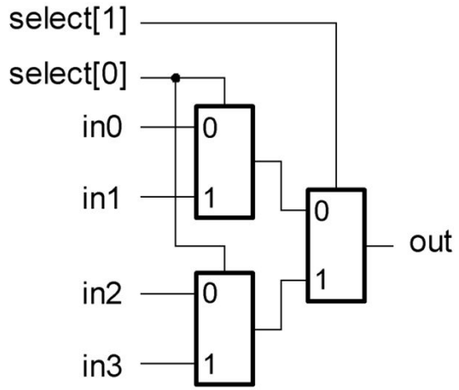
Built-in gates can have > 2 inputs. Ex:

```
and (w0, a, b, c, d);
```

Instantiation, Signal Array, Named ports



a) 4-input mux symbol



b) 4-input mux implemented with 2-input muxes

```
/* 2-input multiplexer in gates */
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;
   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);
endmodule // mux2
```

```
module mux4 (in0, in1, in2, in3, select, out);
input in0,in1,in2,in3;
input [1:0] select;
output out;
wire w0,w1;
  mux2
    m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
    m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
    m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

Signal array. Declares select[1], select[0]

Named ports. Highly recommended.

Simple Behavioral Model

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

    assign out = in1 & in2;

endmodule
```

"continuous assignment"

Connects out to be the logical "and" of in1 and in2.

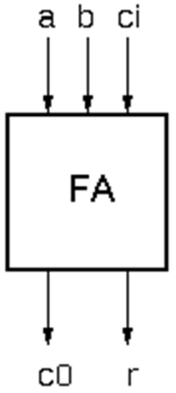
Short-hand for explicit instantiation of bit-wise"and" gate (in this case).

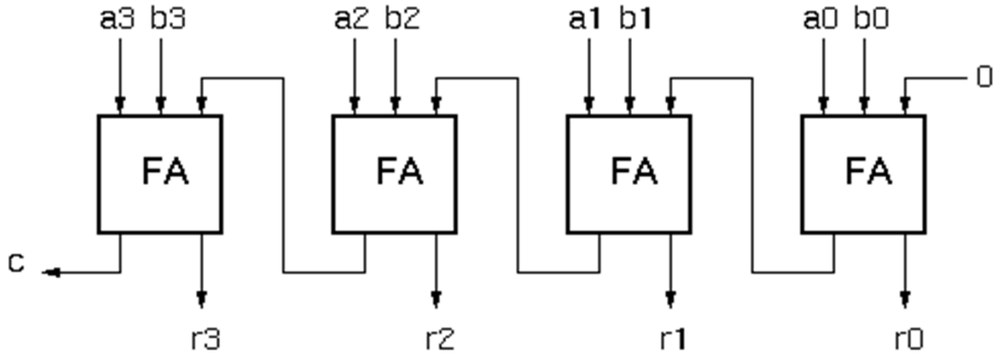
The assignment continuously happens, therefore any change on the rhs is reflected in **`out`** immediately (except for the small delay associated with the implementation of the practical &).

Not like an assignment in C that takes place when the program counter gets to that place in the program.

Example - Ripple Adder

```
module FullAdder(a, b, ci, r, co);
   input a, b, ci;
   output r, co;

   assign r = a ^ b ^ ci;
  assign co = a&ci | a&b | b&cin;

endmodule
```






```
module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;

  wire c1, c2, c3;
  FullAdder
  add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1),   .r(R[0]) ),
  add1(.a(A[1]), .b(B[1]), .ci(c1),   .co(c2),   .r(R[1]) ),
  add2(.a(A[2]), .b(B[2]), .ci(c2),   .co(c3),   .r(R[2]) ),
  add3(.a(A[3]), .b(B[3]), .ci(c3),   .co(R[4]), .r(R[3]) );
endmodule
```

Verilog Operators

Verilog Operator	Name	Functional Group
[]	bit-select or part-select	
()	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

>= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

Verilog Numbers

Constants:

14 ordinary decimal number

-14 2's complement representation

12'b0000_0100_0110 binary number ("_" is ignored)

12'h046 hexadecimal number with 12 bits

Signal Values:

By default, values are unsigned

e.g., **C[4:0] = A[3:0] + B[3:0];**

if A = 0110 (6) and B = 1010(-6)

C = 10000 (not 00000)

i.e., B is zero-padded, not sign-extended

wire signed [31:0] x;

Declares a signed (2's complement) signal array.

Administrivia

- Concurrent enrollments (partially) processed
- Everybody needs to be in one lab!
- Dave Conroy (Apple Fellow) tech talk tonight at 8pm in Sibley
  - And a visit right after the class…

Verilog Assignment Types

Continuous Assignment Examples

```
wire [3:0] A, X,Y,R,Z;
wire [7:0] P;
wire r, a, cout, cin;

assign R = X | (Y & ~Z);

assign r = &X;

assign R = (a == 1'b0) ? X : Y;

assign P = 8'hff;

assign P = X * Y;

assign P[7:0] = {4{X[3]}, X[3:0]};

assign {cout, R} = X + Y + cin;

assign Y = A << 2;

assign Y = {A[1], A[0], 1'b0, 1'b0};
```

use of bit-wise Boolean operators

example reduction operator

conditional operator

example constants

arithmetic operators (use with care!)

(ex: sign-extension)

bit field concatenation

bit shift operator

equivalent bit shift

Non-Continuous Assignments

A bit unusual from a hardware specification point of view.
Shows off Verilog roots as a simulation language.

"always" block example:

```
module and_or_gate (out, in1, in2, in3);
  input     in1, in2, in3;
  output    out;
  reg       out;

  always @(in1 or in2 or in3) begin
     out = (in1 & in2) | in3;
  end

endmodule
```

"reg" type declaration. Not really a register in this case. Just a Verilog idiosyncrasy.

keyword

"sensitivity" list, triggers the action in the body.

brackets multiple statements (not necessary in this example.

Isn't this just: `assign out = (in1 & in2) | in3;`?

Why bother?

Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

case statement example:

```
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
   case (select)
      2'b00: out=in0;
      2'b01: out=in1;
      2'b10: out=in2;
      2'b11: out=in3;
   endcase
endmodule // mux4
```

keyword

The statement(s) corresponding to whichever constant matches "select" get applied.

Couldn't we just do this with nested "if"s?

Well yes and no!

Always Blocks

Nested if-else example:

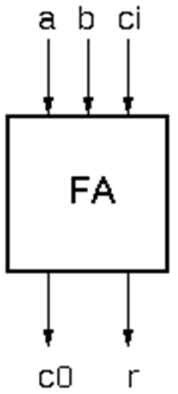
```
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
   if (select == 2'b00) out=in0;
      else if (select == 2'b01) out=in1;
           else if (select == 2'b10) out=in2;
                else out=in3;
endmodule // mux4
```

Nested if structure leads to “priority logic” structure, with different delays for different inputs (in3 to out delay > than in0 to out delay).

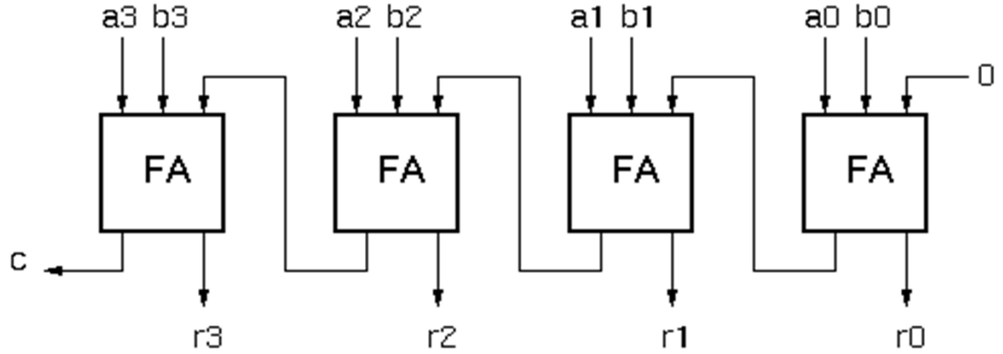
Case version treats all inputs the same.

Review – Ripple-Carry Adder Example

```
module FullAdder(a, b, ci, r, co);
   input a, b, ci;
   output r, co;

   assign r = a ^ b ^ ci;
  assign co = a&ci + a&b + b&cin;

endmodule
```




```
module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;

  wire c1, c2, c3;
  FullAdder
  add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1),    .r(R[0]) ),
  add1(.a(A[1]), .b(B[1]), .ci(c1),   .co(c2),    .r(R[1]) ),
  add2(.a(A[2]), .b(B[2]), .ci(c2),   .co(c3),    .r(R[2]) ),
  add3(.a(A[3]), .b(B[3]), .ci(c3),   .co(R[4]), .r(R[3]) );
endmodule
```



Example - Ripple Adder Generator

Parameters give us a way to generalize our designs. A module becomes a "generator" for different variations. Enables design/module reuse. Can simplify testing.

```
module Adder(A, B, R);
  parameter N = 4;
  input [N-1:0] A;
  input [N-1:0] B;
  output [N:0] R;
  wire [N:0] C;

  genvar i;

  generate
    for (i=0; i<N; i=i+1) begin:bit
      FullAdder add(.a(A[i], .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
    end
  endgenerate

  assign C[0] = 1'b0;
  assign R[N] = C[N];
endmodule
```

Declare a parameter with default value.

Note: this is not a port. Acts like a "synthesis-time" constant.

Replace all occurrences of "4" with "N".

variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

For-loop creates instances (with unique names)

```
Adder adder4 ( ... );

Adder #(.N(64))
adder64 ( ... );
```

Overwrite parameter N at instantiation.

More on Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
    parameter SIZE = 8;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;

    genvar i;

    generate for (i=0; i<SIZE; i=i+1) begin:bit
        assign bin[i] = ^gray[SIZE-1:i];
    end endgenerate
endmodule
```

variable exists only in the specification - not in the final circuit.

Keywords that denotes synthesis-time operations

For-loop creates instances of assignments

Loop must have constant bounds

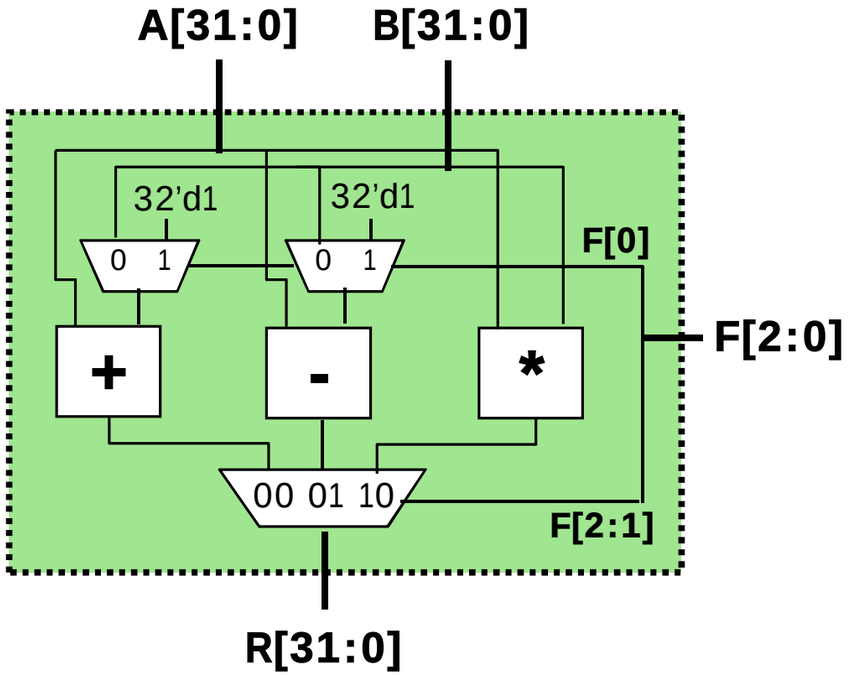
generate if-else-if based on an expression that is deterministic at the time the design is synthesized.

generate case : selecting case expression must be deterministic at the time the design is synthesized.

Defining Processor ALU in 5 mins

- Modularity is essential to the success of large designs
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Example: A 32-bit ALU



Function Table

F2	F1	F0	Function
0	0	0	A + B
0	0	1	A + 1
0	1	0	A - B
0	1	1	A - 1
1	0	X	A * B

Module Definitions

2-to-1 MUX

```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

32-bit Adder

```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```

32-bit Subtractor

```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

3-to-1 MUX

```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;

always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    default: out = 32'bx;
  endcase
end
endmodule
```

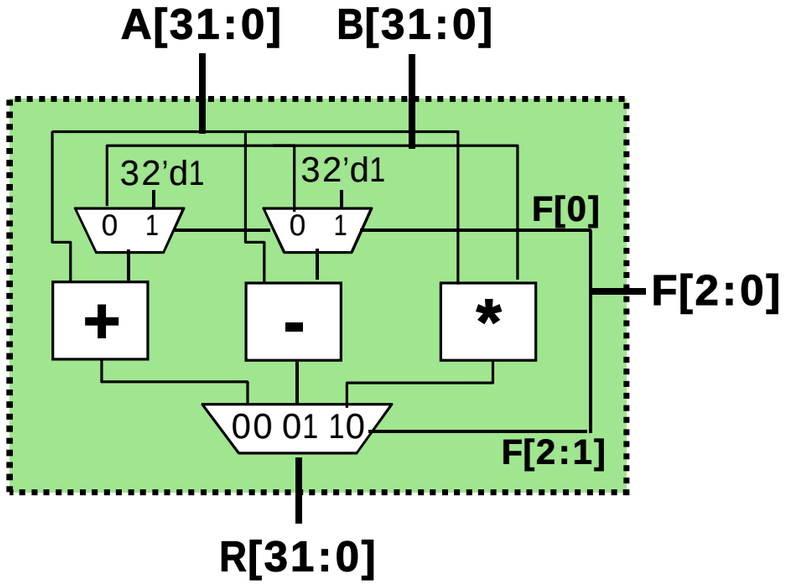
16-bit Multiplier

```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

Top-Level ALU Declaration

- Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```



- Declaration of the ALU Module:

```
module alu(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;
  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;
  mux32two   adder_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(addmux_out));
  mux32two   sub_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(submux_out));
  add32      our_adder(.i0(a), .i1(addmux_out), .sum(add_out));
  sub32      our_subtracter(.i0(a), .i1(submux_out), .diff(sub_out));
  mul16      our_multiplier(.i0(a[15:0]), .i1(b[15:0]), .prod(mul_out));
  mux32three output_mux(.i0(add_out), .i1(sub_out), .i2(mul_out), .sel(f[2:1]), .out(r));
endmodule
```

intermediate output nodes

module names

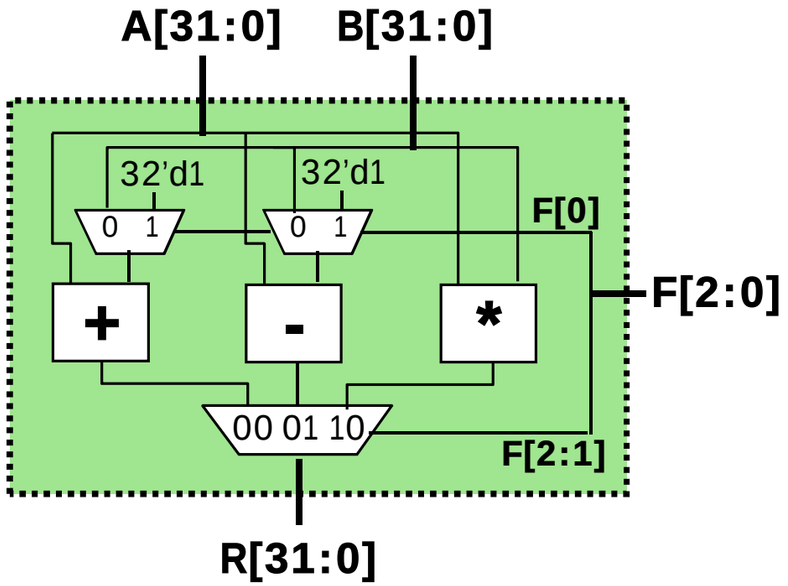
(unique) instance names

corresponding wires/regs in module alu

Top-Level ALU Declaration, take 2

- No Hierarchy:
- Declaration of the ALU Module:

```
module alu(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;
  always @ (a or b or f)
    case (f)
      3'b000:  r = a + b;
      3'b001:  r = a + 1'b1;
      3'b010:  r = a – b;
      3'b011:  r = a – 1'b1;
      3'b100:  r = a * b;
      default: r = 32'bx;
    endcase
endmodule
```



Will this synthesize into 2 adders and 2 subtractors or 1 of each?

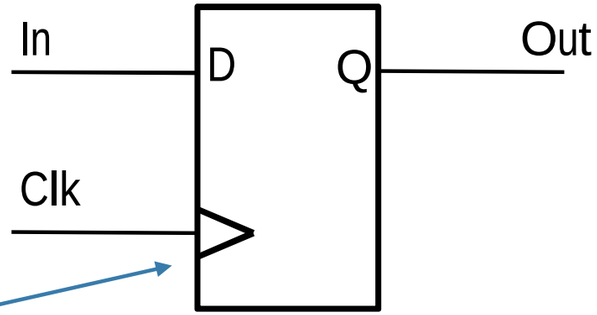
Sequential Logic, Take 2

Latches and Flip-Flops

- Flip-flop is edge-triggered, latch is level-sensitive

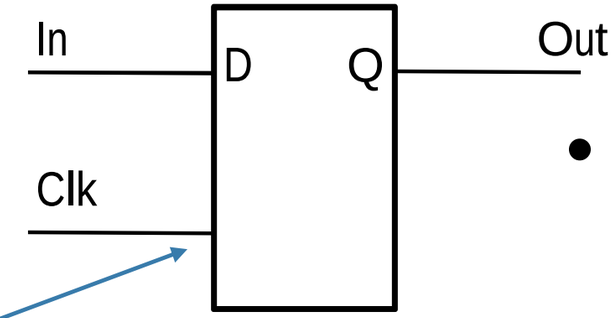
- D Flip-flop
  - Rising edge

Signifies ‘edge triggered’

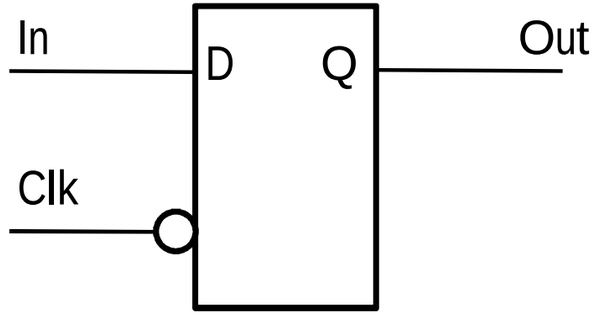


- D Latch
  - Transparent HIGH

Level sensitive if there is no ‘edge’

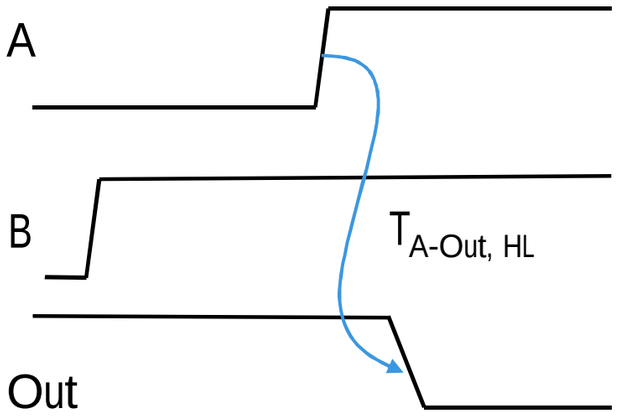


- Transparent LOW

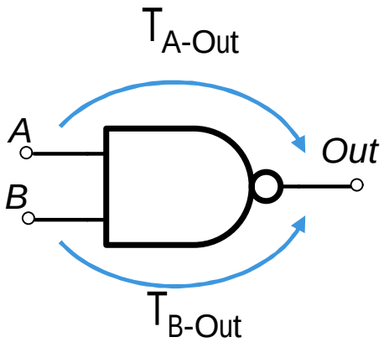


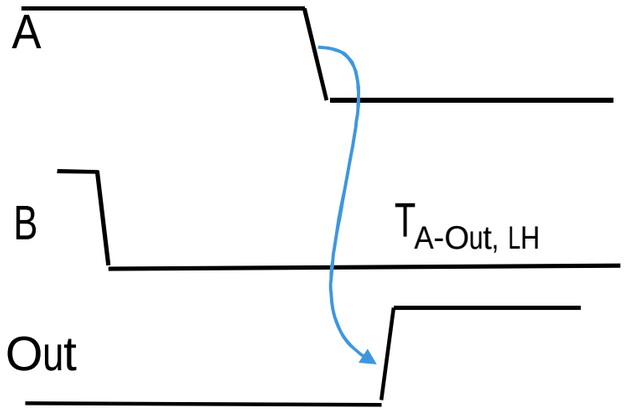
Timing

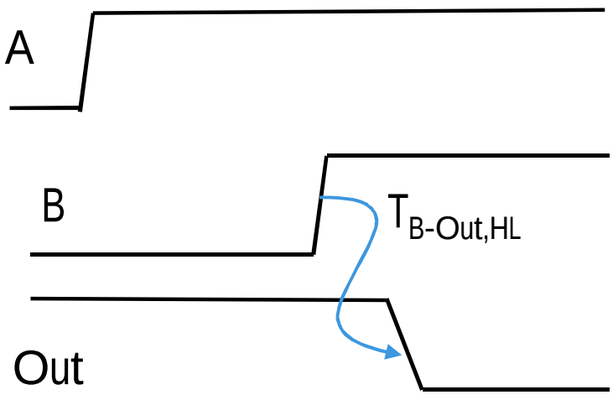
- Combinational logic timing



A is arriving late
(is in the critical path)







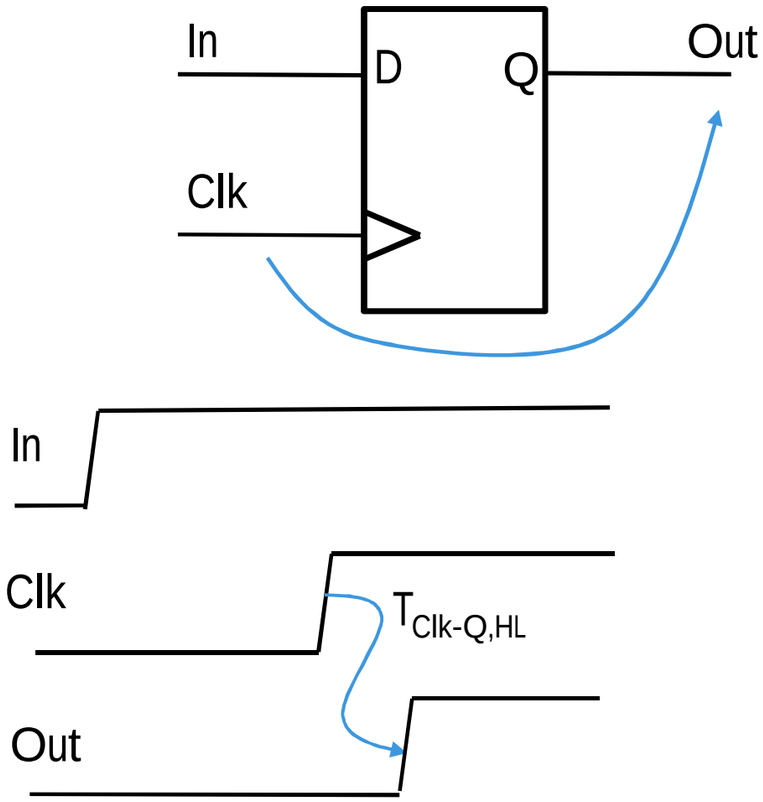
B is arriving late
(is in the critical path)

HL and LH transition differ
$t_{A\text{-}Out}$ and $t_{B\text{-}Out}$ differ
In CMOS, propagation delay depends on:
- Gate type, size (output resistance)
- Capacitive loading
- Input slope

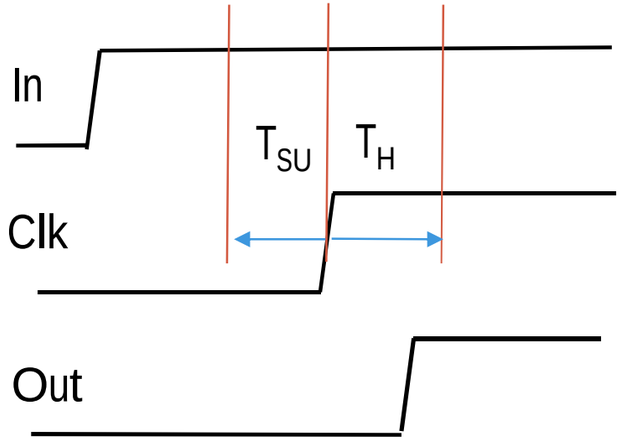
Timing

- Flip-flop timing

  (latch timing will be covered later)

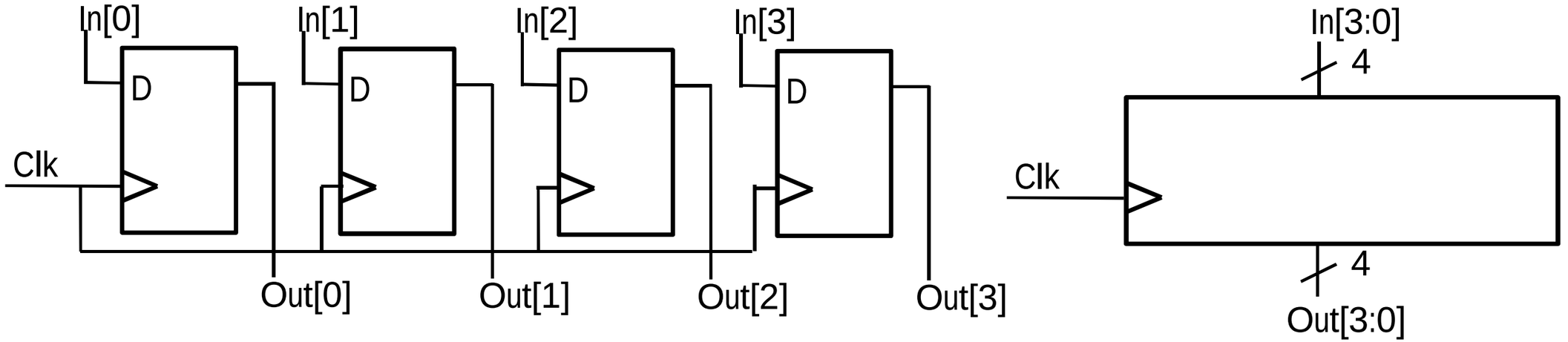


- Setup and hold times
  - Data cannot change in the interval of setup time **before** the clock edge to hold time **after** the clock edge

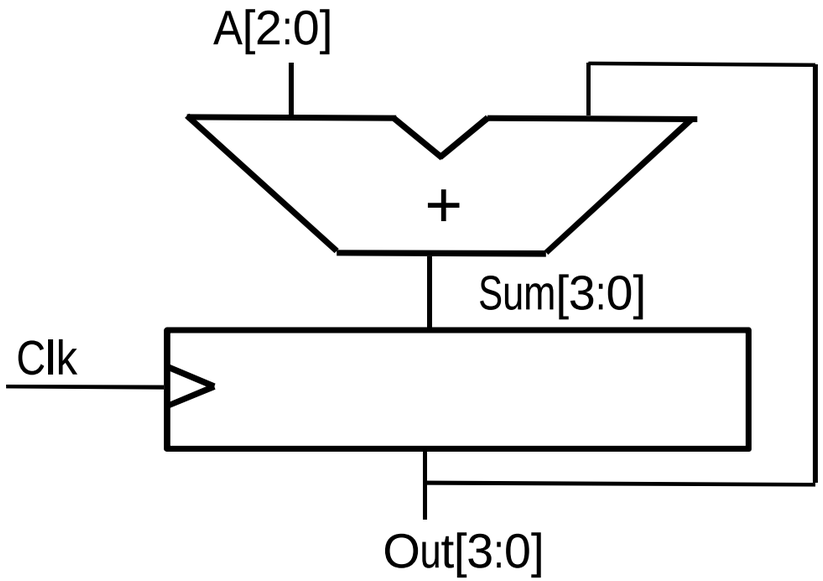


Register

- 4-bit register



- Accumulator



State Elements in Verilog

Always blocks are the only way to specify the “behavior” of state elements.
Synthesis tools will turn state element behaviors into state element instances.

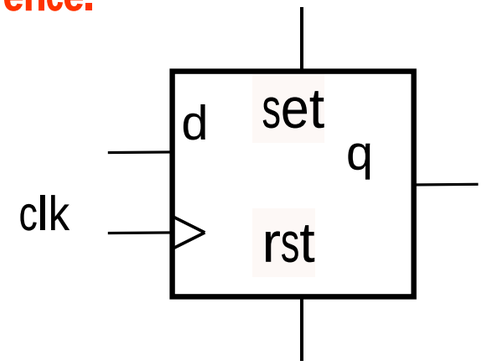
D-flip-flop with synchronous set and reset example:

```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;

  always @(posedge clk)
    if (rst)
      q <= 1’b0;
    else if (set)
      q <= 1’b1;
    else
      q <= d;
endmodule
```

keyword

“always @ (posedge clk)” is key to flip-flop inference.

This gives priority to reset over set and set over d.



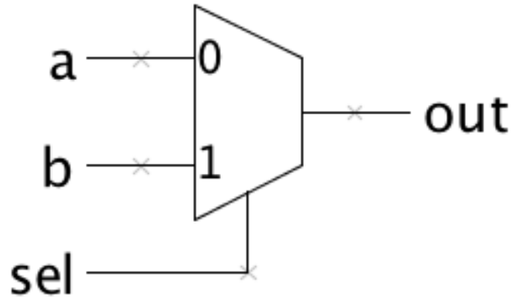
On FPGAs, maps to native flip-flop.

Unlike logic gates, their are no primitive flip-flops in Verilog. Although, it is possible to instantiate FPGA or standard-cell specific flip-flops.

The Sequential always Block

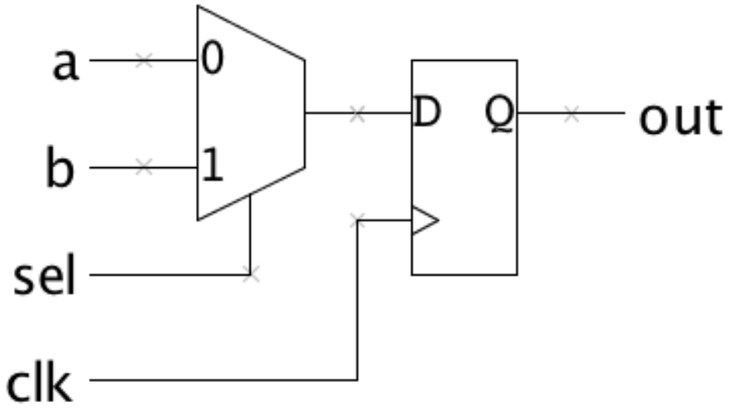
Combinational

```
module comb(input a, b, sel,
            output reg out);
  always @(*) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```



Sequential

```
module seq(input a, b, sel, clk,
           output reg out);
  always @(posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  end
endmodule
```



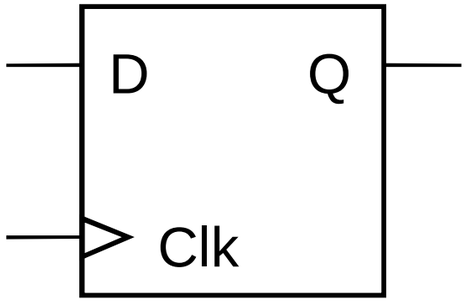
Latches vs. Flip-Flops

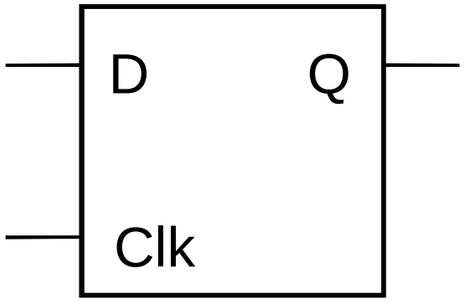
Flip-Flop

```
module flipflop
(
   input clk,
   input d,
   output reg q
);

  always @(posedge clk)
  begin
    q <= d;
  end
endmodule
```

Latch

```
module latch
(
   input clk,
   input d,
   output reg q
);

  always @(clk or d)
  begin
    if ( clk )
      q <= d;
  end
endmodule
```





Importance of the Sensitivity List

- The use of posedge and negedge makes an always block sequential (edge-triggered)

D-Register with synchronous clear

```
module dff_sync_clear(
  input d, clearb, clock,
  output reg q);

  always @(posedge clock)
    begin
      if (!clearb) q <= 1'b0;
      else q <= d;
    end
endmodule
```

`always` block entered only at each positive clock edge

D-Register with asynchronous clear

```
module dff_async_clear(
  input d, clearb, clock,
  output reg q);

  always @(negedge clearb or posedge clock)
    begin
      if (!clearb) q <= 1'b0;
      else q <= d;
    end
endmodule
```

`always` block entered immediately when (active-low) clearb is asserted

Note: The following is incorrect syntax: *always @(clear or negedge clock)*

If one signal in the sensitivity list uses *posedge*/*negedge*, then all signals must.

- *Assign any signal or variable from only one always block.*

Be wary of race conditions: always blocks with same trigger execute concurrently…

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within always blocks, with subtly different behaviors.
  - Blocking assignment (=): evaluation and assignment are immediate

```
always @(*) begin
  x = a | b;      // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;  // 2. evaluate a^b^c, assign result to y
  z = b & ~c;     // 3. evaluate b&(~c), assign result to z
end
```

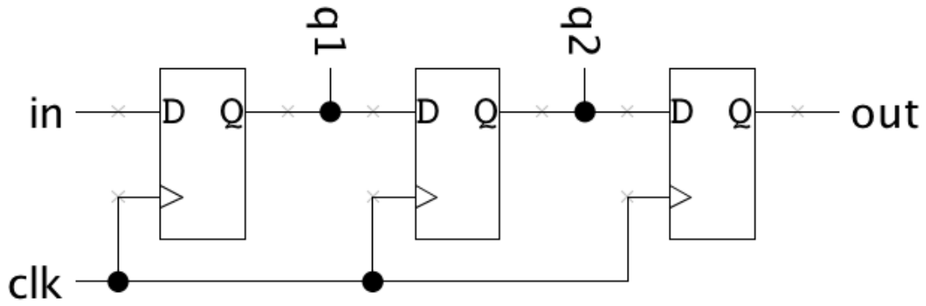
  - Nonblocking assignment (<=): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active always blocks)

```
always @(*) begin
  x <= a | b;     // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c;    // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

Sometimes, as above, both produce the same result. Sometimes, not!

Assignment Styles for Sequential Logic

What we want:
Register-based digital delay line
(a.k.a. shift-register)



Will non-blocking and blocking assignments both produce the desired result?

```
module nonblocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

```
module blocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

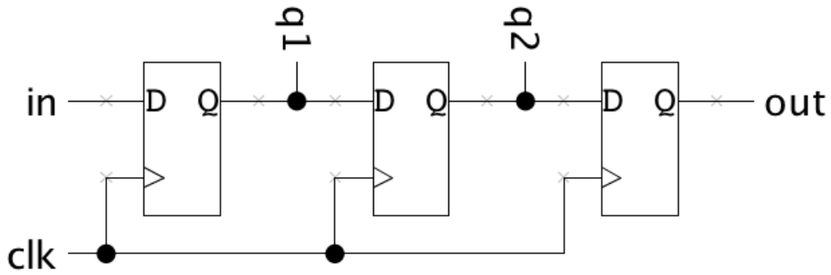
Use Nonblocking for Sequential Logic

```
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;   // uses old q1
    out <= q2;  // uses old q2
  end
```

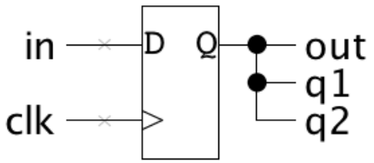
```
always @(posedge clk) begin
    q1 = in;
    q2 = q1;   // uses new q1
    out = q2;  // uses new q2
  end
```

("old" means value before clock edge, "new" means the value after most recent assignment)

"At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2."

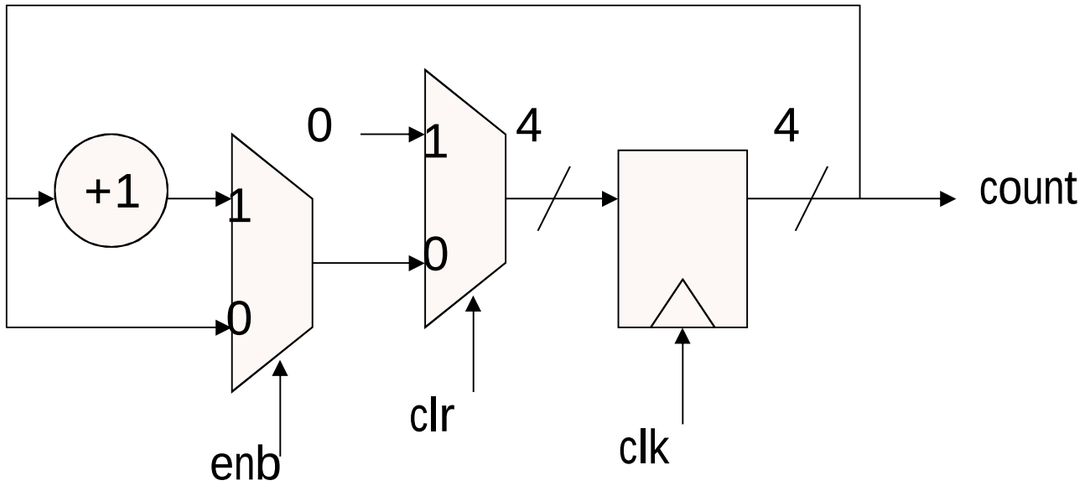


"At each rising clock edge, q1 = in. After that, q2 = q1. After that, out = q2. Therefore out = in."



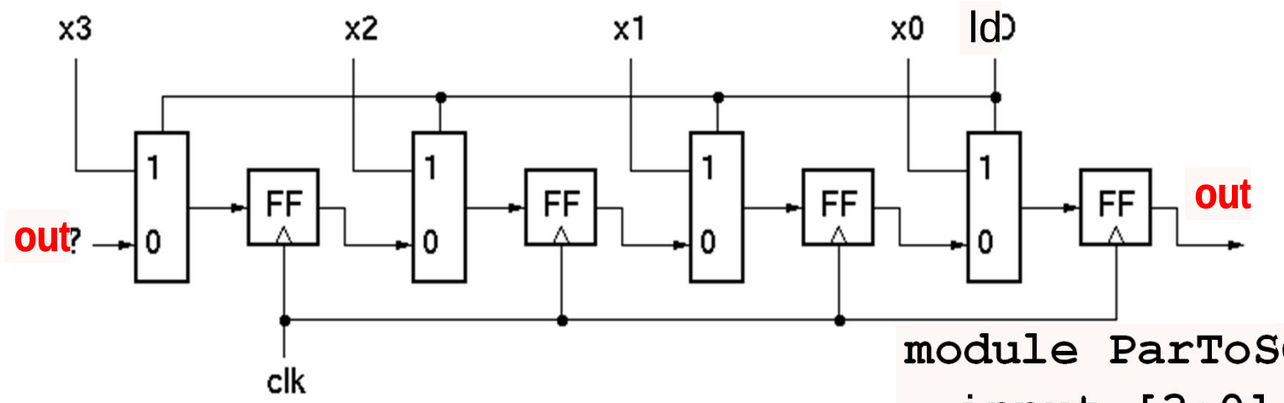
- Blocking assignments **do not** reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use **nonblocking** assignments for sequential always blocks

Example: A Simple Counter




```
// 4-bit counter with enable and synchronous clear
module counter(input clk,enb,clr,
               output reg [3:0] count);
  always @(posedge clk) begin
    count <= clr ? 4'b0 : (enb ? count+1 : count);
  end
endmodule
```

Example - Parallel to Serial Converter




```
module ParToSer(ld, X, out, clk);
  input [3:0] X;
  input ld, clk;
  output out;

  reg [3:0] Q;
  wire [3:0] NS;

  assign NS =
    (ld) ? X : {Q[0], Q[3:1]};

  always @ (posedge clk)
     Q <= NS;

  assign out = Q[0];
endmodule
```

Specifies the muxing with "rotation"

forces Q register (flip-flops) to be rewritten every cycle

connect output

Verilog in EECS 151/251A

- We use behavioral modeling at the bottom of the hierarchy
- Use instantiation to 1) build hierarchy and,
  2) map to FPGA and ASIC resources not supported by synthesis.
- Favor continuous assign and avoid always blocks unless:
  - No other alternative: ex: state elements, case
  - Helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Harris & Harris book chapter 4 is a good source.
  - ***Be careful of what you read on the web***. Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final Thoughts on Verilog Examples

Verilog looks like C, but it describes hardware:

Entirely different semantics: multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. First understand the circuit you want then figure out how to code it in Verilog. If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

As you get more practice, you will know how to best write Verilog for a desired result.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.

Summary

- Verilog is the most commonly used HDL
- We have seen combinatorial and sequential constructs
- Practice is the best way to learn a new language…