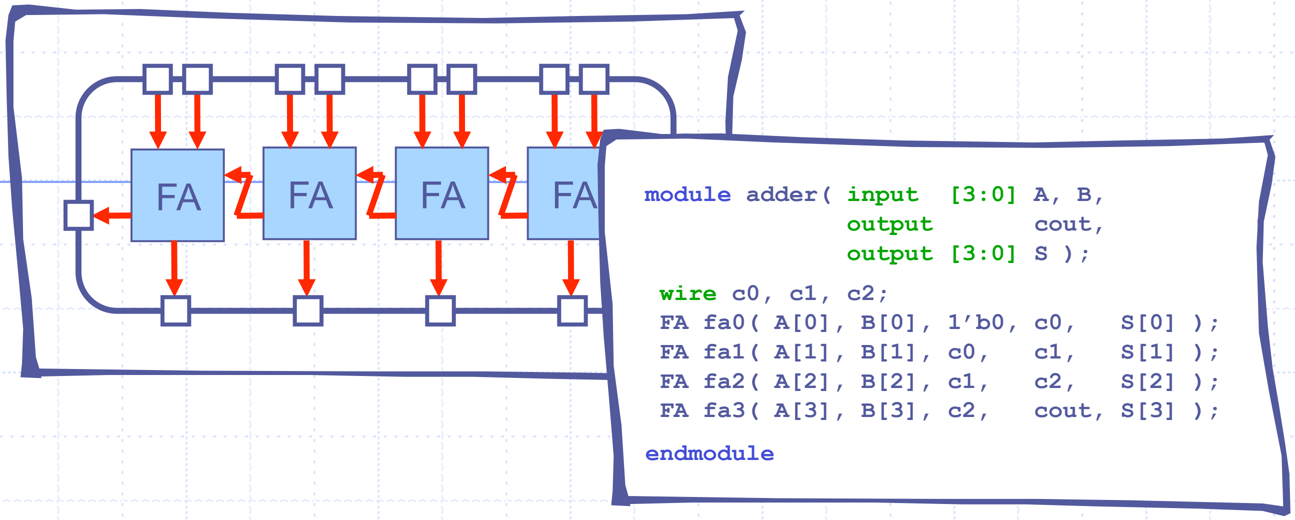
Verilog 1 - Fundamentals



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
 FA fa1( A[1], B[1], c0,   c1,   S[1] );
 FA fa2( A[2], B[2], c1,   c2,   S[2] );
 FA fa3( A[3], B[3], c2,   cout, S[3] );

endmodule
```

6.375 Complex Digital Systems

Arvind

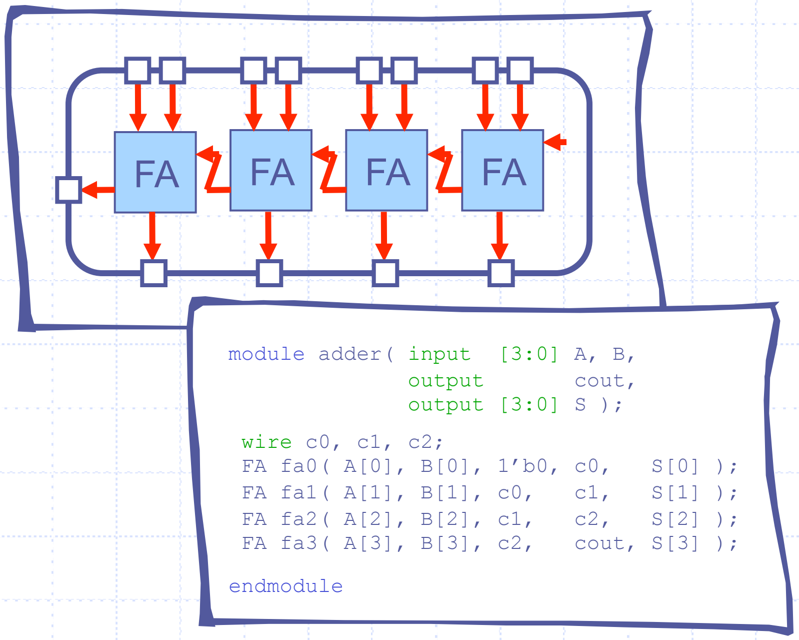
Courtesy of Arvind http://csg.csail.mit.edu/6.375/

What is Verilog?

- Verilog is a hardware description language
  - There is more to it than that, but this is what we will use it for.
- In this class, Verilog is an implementation language, not a design language
  - When you sit down to write verilog you should know *exactly* what you are implementing.
  - Draw your schematic and state machines and *transcribe* it into Verilog.
- We are constraining you to a subset of the language for two reasons
  - Reduce complexity for the course
  - These are parts that people use to design real processors

Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
 FA fa1( A[1], B[1], c0,   c1,   S[1] );
 FA fa2( A[2], B[2], c1,   c2,   S[2] );
 FA fa3( A[3], B[3], c2,   cout, S[3] );

endmodule
```

Courtesy of Arvind http://csg.csail.mit.edu/6.375/

Bit-vector is the only data type in Verilog

A bit can take on one of four values

Value	Meaning
0	Logic zero
1	Logic one
X	Unknown logic value
Z	High impedance, floating

In the simulation waveform viewer, Unknown signals are RED. There should be no red after reset

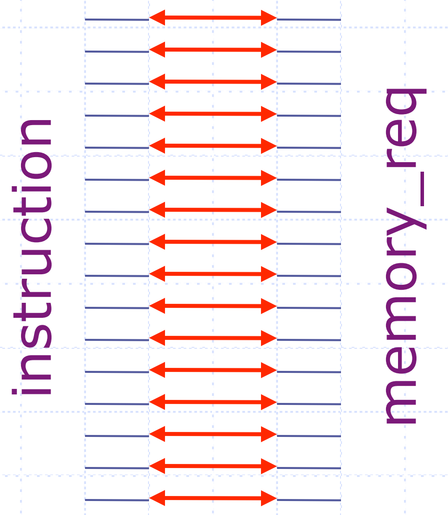
An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don't care what the value is. This can help catch bugs and improve synthesis quality.

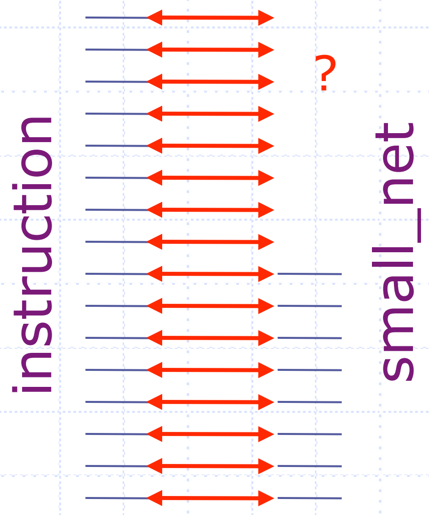
Courtesy of Arvind http://csg.csail.mit.edu/6.375/

“wire” is used to denote a hardware net

```
wire [15:0] instruction;
wire [15:0] memory_req;
wire [  7:0] small_net;
```

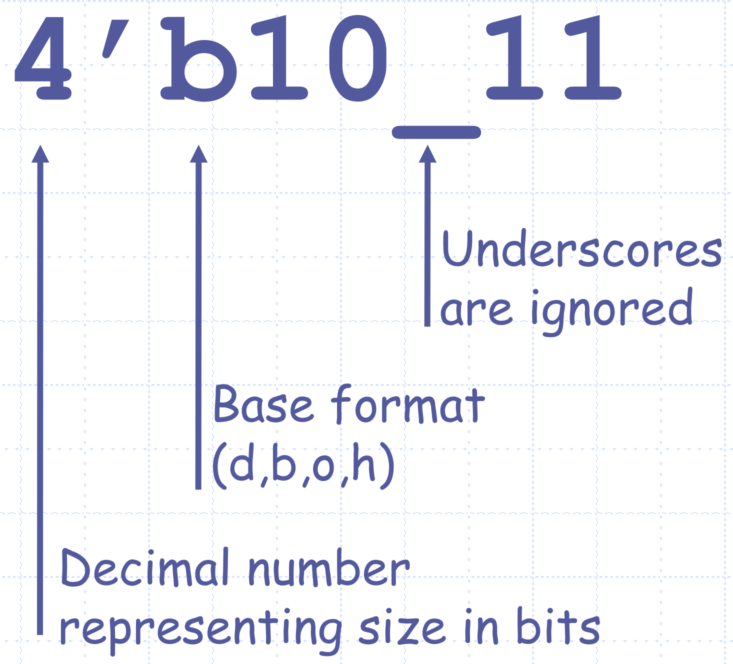
Absolutely no type safety when connecting nets!





Courtesy of Arvind http://csg.csail.mit.edu/6.375/

Bit literals

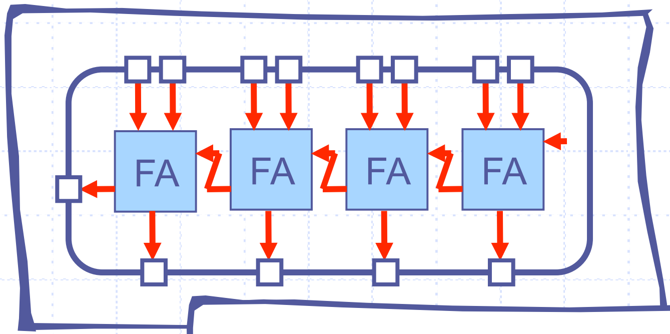


- Binary literals
  - `8'b0000_0000`
  - `8'b0xx0_1xx1`
- Hexadecimal literals
  - `32'h0a34_def1`
  - `16'haxxx`
- Decimal literals
  - `32'd42`

We'll learn how to actually assign literals to nets a little later

Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors



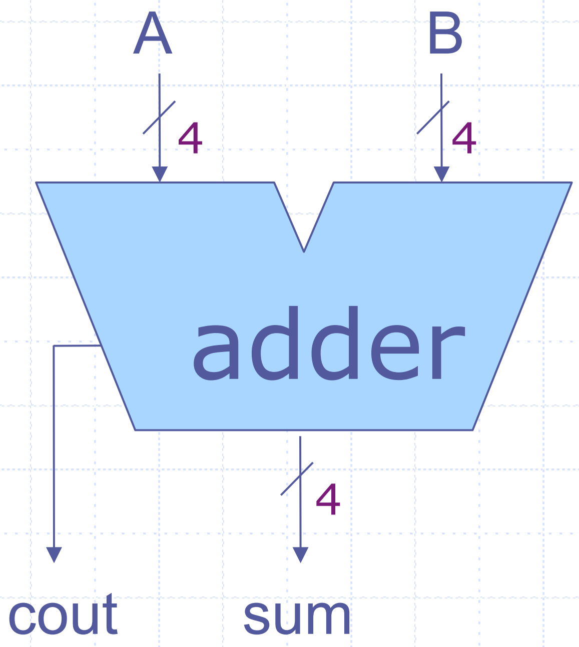
```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
 FA fa1( A[1], B[1], c0,   c1,   S[1] );
 FA fa2( A[2], B[2], c1,   c2,   S[2] );
 FA fa3( A[3], B[3], c2,   cout, S[3] );

endmodule
```

Courtesy of Arvind http://csg.csail.mit.edu/6.375/

Our Verilog Subset

- Verilog is a big language with many features not concerned with synthesizing hardware.
- The code you write for your processor should only contain the languages structures discussed in these slides.
- Anything else is not synthesizable, although it will simulate fine.
- You *MUST* follow the course coding standard

A Verilog module has a name and a port list

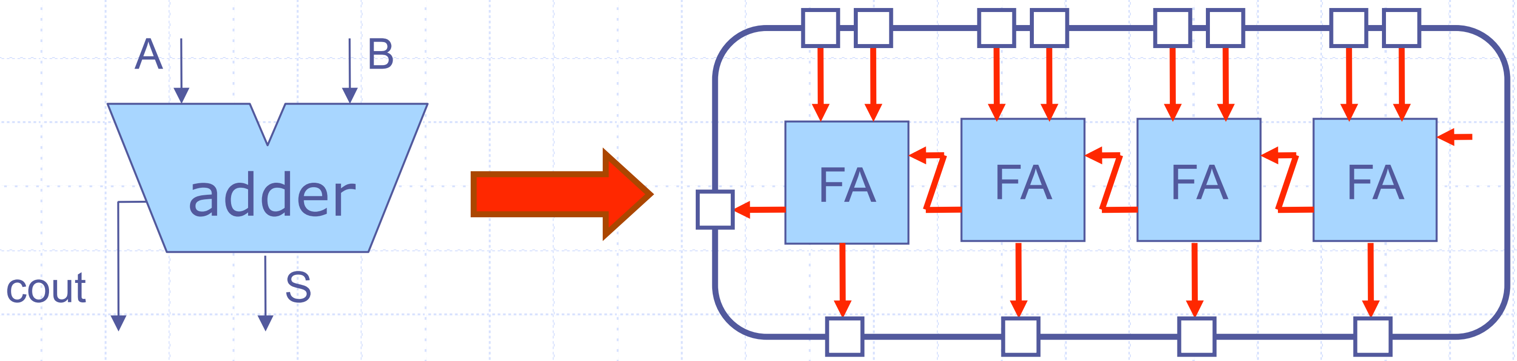


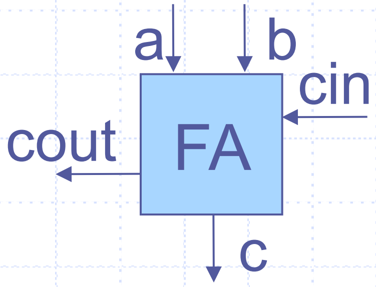
```
module adder( input   [3:0] A,
              input   [3:0] B,
              output        cout,
              output  [3:0] sum );

  // HDL modeling of
  // adder functionality
endmodule
```

Ports must have a direction (or be bidirectional) and a bitwidth

Note the semicolon at the end of the port list!

A module can instantiate other modules

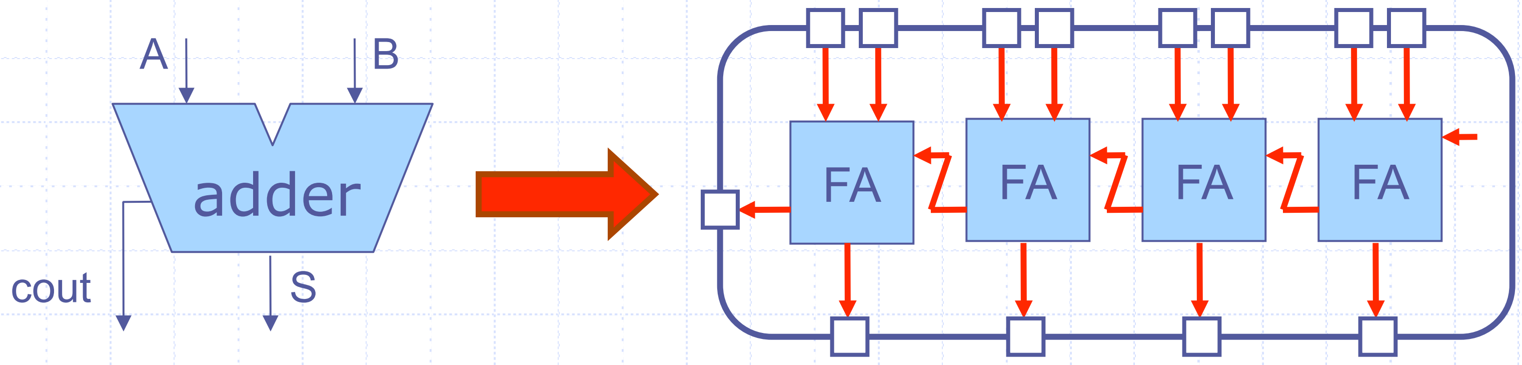




```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( ... );
 FA fa1( ... );
 FA fa2( ... );
 FA fa3( ... );

endmodule
```

```
module FA( input  a, b, cin
           output cout, sum );
  // HDL modeling of 1 bit
  // full adder functionality
endmodule
```

Connecting modules



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
 FA fa1( A[1], B[1], c0,   c1,   S[1] );
 FA fa2( A[2], B[2], c1,   c2,   S[2] );
 FA fa3( A[3], B[3], c2,   cout, S[3] );

endmodule
```

Carry Chain

Alternative syntax

Connecting ports by ordered list

```
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
```

Connecting ports by name (compact)

```
FA fa0( .a(A[0]), .b(B[0]),
        .cin(1'b0), .cout(c0), .sum(S[0]) );
```

Argument order does not matter when ports are connected by name

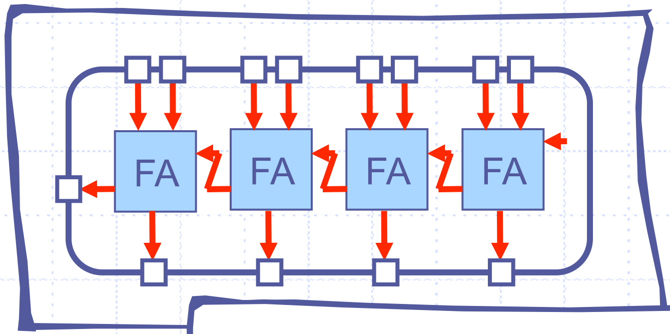
```
FA fa0
( .a    (A[0]),
  .cin  (1'b0),
  .b    (B[0]),
  .cout (c0),
  .sum  (S[0]) );
```

Connecting ports by name yields clearer and less buggy code.

Courtesy of Arvind http://csg.csail.mit.edu/6.375/

Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

 wire c0, c1, c2;
 FA fa0( A[0], B[0], 1’b0, c0,   S[0] );
 FA fa1( A[1], B[1], c0,   c1,   S[1] );
 FA fa2( A[2], B[2], c1,   c2,   S[2] );
 FA fa3( A[3], B[3], c2,   cout, S[3] );

endmodule
```

A module’s behavior can be described in many different ways but it should not matter from outside

Example: mux4

mux4: Using continuous assignments

Language defined operators

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

  wire out, t0, t1;
  assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );

endmodule
```

The order of these continuous assignment statements does not matter.
They essentially happen in parallel!

mux4: Behavioral style

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

  assign out = ( sel == 0 ) ? a :
               ( sel == 1 ) ? b :
               ( sel == 2 ) ? c :
               ( sel == 3 ) ? d : 1'bx;

endmodule
```

If input is undefined we want to propagate that information.

mux4: Using “always block”

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

  reg out, t0, t1;

  always @( * )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end

endmodule
```

The order of these procedural assignment statements DOES matter. They essentially happen sequentially!

“Always blocks” permit more advanced sequential idioms

```
module mux4( input  a,b,c,d
             input [1:0]
sel,
             output out );
  reg out;
  always @( * )
  begin
    if ( sel == 2’d0 )
      out = a;
    else if ( sel == 2’d1 )
      out = b
    else if ( sel == 2’d2 )
      out = c
    else if ( sel == 2’d3 )
      out = d
    else
      out = 1’bx;
  end
endmodule
```

```
module mux4( input  a,b,c,d
             input [1:0]
sel,
             output out );
  reg out;
  always @( * )
   begin
    case ( sel )
      2’d0 : out = a;
      2’d1 : out = b;
      2’d2 : out = c;
      2’d3 : out = d;
      default : out = 1’bx;
    endcase
  end
endmodule
```

Typically we will use always blocks only to describe sequential circuits

What happens if the case statement is not complete?

```
module mux3( input  a, b, c
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2’d0 : out = a;
      2’d1 : out = b;
      2’d2 : out = c;
    endcase
  end

endmodule
```

If sel = 3, mux will output the previous value!

What have we created?

What happens if the case statement is not complete?

```
module mux3( input  a, b, c
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2'd0 : out = a;
      2'd1 : out = b;
      2'd2 : out = c;
      default : out = 1'bx;
    endcase
  end

endmodule
```

We CAN prevent creating state with a default statement

Parameterized mux4

default value

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

  wire [WIDTH-1:0] out, t0, t1;

  assign t0  = (sel[1]? c : a);
  assign t1  = (sel[1]? d : b);
  assign out = (sel[0]? t0: t1);
endmodule
```

Parameterization is a good practice for reusable modules

Writing a mux*n* is challenging

Instantiation Syntax

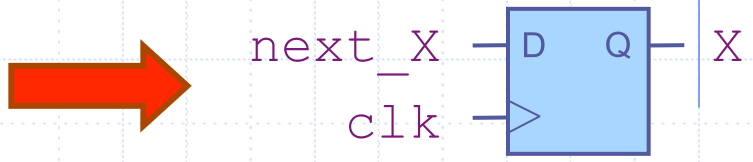
```
mux4#(32) alu_mux
( .a (op1),
  .b (op2),
  .c (op3),
  .d (op4),
  .sel (alu_mux_sel),
  .out (alu_mux_out) );
```

Verilog Registers “reg”

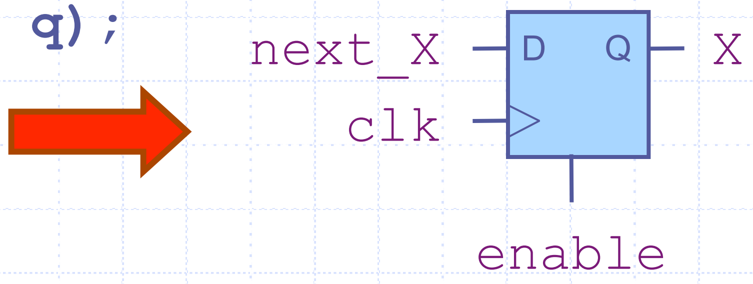
- Wires are line names – they do not represent storage and can be assigned only once
- Regs are imperative variables (as in C):
  - “nonblocking” assignment `r <= v`
  - can be assigned multiple times and holds values between assignments

flip-flops

```
module FF0 (input clk, input  d,
            output reg q);
always @( posedge clk )
  begin
     q <= d;
  end
endmodule
```



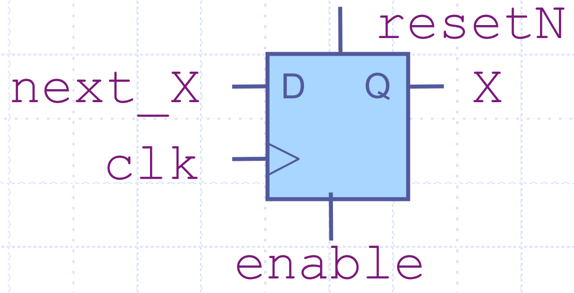
```
module FF (input clk, input  d,
           input  en, output reg q);
always @( posedge clk )
  begin
    if ( en )
     q <= d;
  end
endmodule
```



flip-flops with reset

```
always @( posedge clk)
begin
  if (~resetN)
    Q <= 0;
  else if ( enable )
    Q <= D;
end
```

synchronous reset



Register

```
module register#(parameter WIDTH = 1)
(
  input  clk,
  input  [WIDTH-1:0] d,
  input  en,
  output [WIDTH-1:0] q
);

  always @( posedge clk )
  begin
    if (en)
      q <= d;
  end

endmodule
```

Register in terms of Flipflops

```
module register2
( input  clk,
  input  [1:0] d,
  input  en,
  output [1:0] q );

  always @(posedge clk)
  begin
    if (en)
     q <= d;
  end

endmodule
```

```
module register2
( input  clk,
  input  [1:0] d,
  input  en,
  output [1:0] q
);
 FF ff0 (.clk(clk), .d(d[0]),
         .en(en),   .q(q[0]));

 FF ff1 (.clk(clk), .d(d[1]),
         .en(en),   .q(q[1]));

endmodule
```

Do they behave the same?

yes