CSE 141L: Introduction to Computer Architecture Lab

SystemVerilog

Pat Pannuto, UC San Diego
ppannuto@ucsd.edu

Logistics Updates

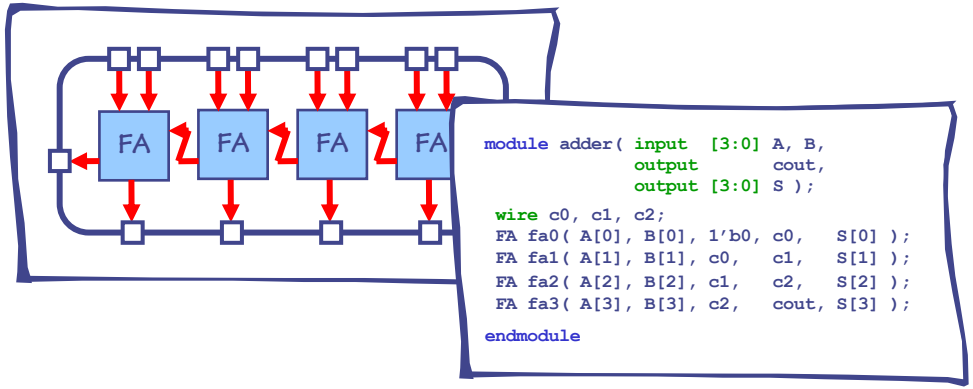
- Lab Hours
  - Internal signups for CSE basement labs end today
  - Expect to post hours beginning of next week
  - Trickling in on Canvas now, but subject to change still…
- Tools
  - CloudLabs is live
  - ModelSim is dead, long live Questa [but ModelSim is fine too]
    - When following tutorials, seems safe to `s/ModelSim/Questa`
- Vocabulary
  - Labs -> Milestones + Final Report

Logistics Update: Waitlists

- 24 people and counting who are in/finished 141 but waitlisted for 141L
  - This is too many to just let everyone in

If you are considering dropping this course, please do so ASAP

- If you are far back on the waitlist for 141, then please make room in 141L
- 141 will be offered next quarter
  - (I'm teaching it)



SYNTHESIZABLE SYSTEM VERILOG 1 – FUNDAMENTALS

What is SystemVerilog (SV)?

- In this class and in the real world, SystemVerilog is a specification language, not a programming language.
  - Draw your schematic and state machines and then transcribe it into SV.
  - When you sit down to write SV you should know exactly what you are implementing.
- We are constraining you to a subset of the language for two reasons
  - These are the parts that people use to design real processors
  - Steer you clear of problematic constructs that lead to bad design.

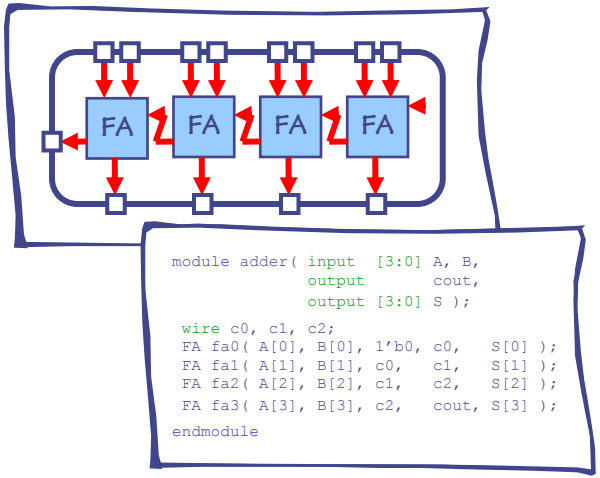
[System]Verilog is a Hardware Description Language (HDL)

- The other popular HDL is VHDL
- ***An HDL is not a programming language — it is an HDL!***

- SystemVerilog is a new-ish improvement over Verilog
  - Technically, it's a backwards-compatible superset
  - This can be troublesome, as Verilog is earlier to make mistakes in :/

SV Fundamentals

- What is System Verilog?
- Data types
- Structural SV
- RTL SV
  - Combinational Logic
  - Sequential Logic



Bit-vectors are the primary data type in Synthesizable SV

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

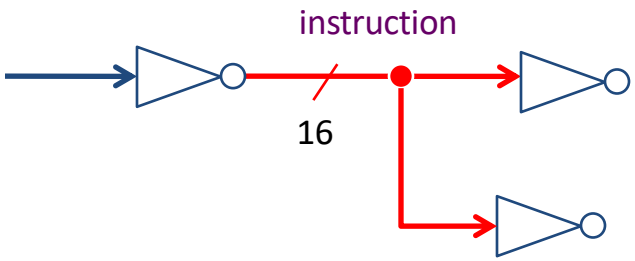
In the simulation waveform viewer, Unknown signals are RED. There should be no red after reset.

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.

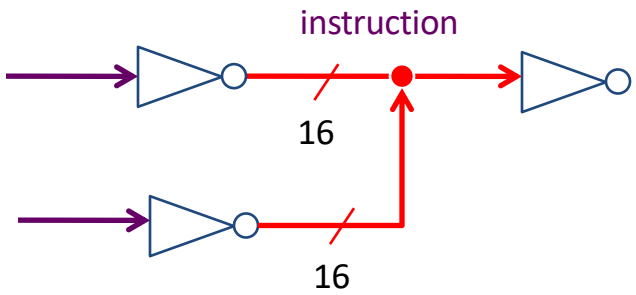
`logic` keyword denotes a hardware net that has a single driver but possibly multiple outputs

- It can be combinational or sequential – other syntax will tell which

```
logic [15:0] instruction;
```



Legal

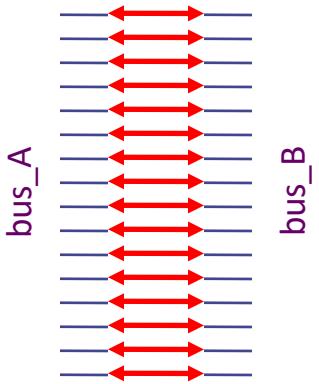


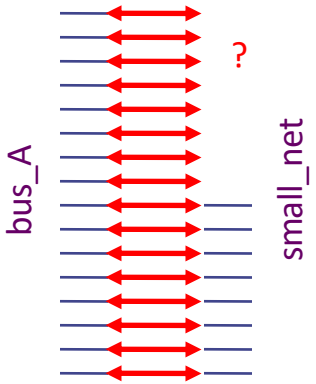
Illegal

`wire` keyword denotes a hardware net that has >=1 drivers, or that has unknown (or bi-) directionality

```
wire [15:0] bus_A;
wire [15:0] bus_B;
wire [ 7:0] small_net;
```

Absolutely no type safety when connecting nets!





Bit literals

```
4'b10_11
```

Underscores are ignored

Base format (d,b,o,h)

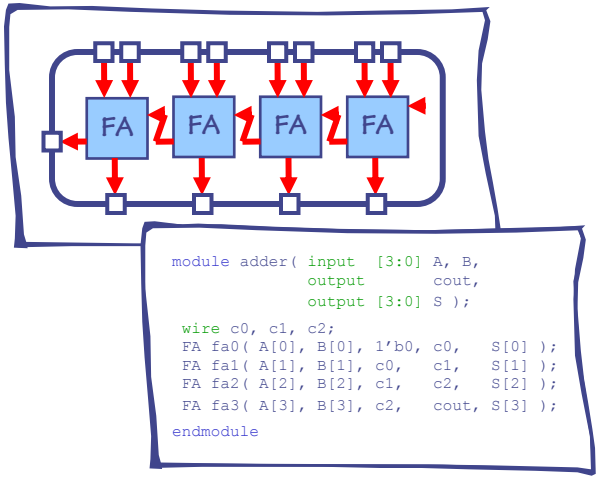
Decimal number representing size in bits

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

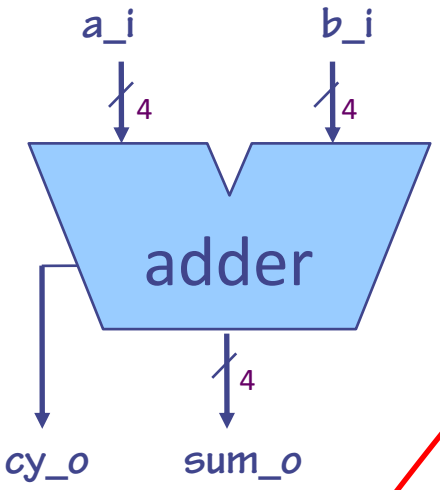
- Binary literals
  - `8'b0000_0000`
  - `8'b0xx0_1xx1`
- Hexadecimal literals
  - `32'h0a34_def1`
  - `16'haxxx`
- Decimal literals
  - `32'd42`

SV Fundamentals

- What is System Verilog?
- Data types
- Structural SV
- RTL SV
  - Combinational Logic
  - Sequential Logic



A SV module has a name and a port list

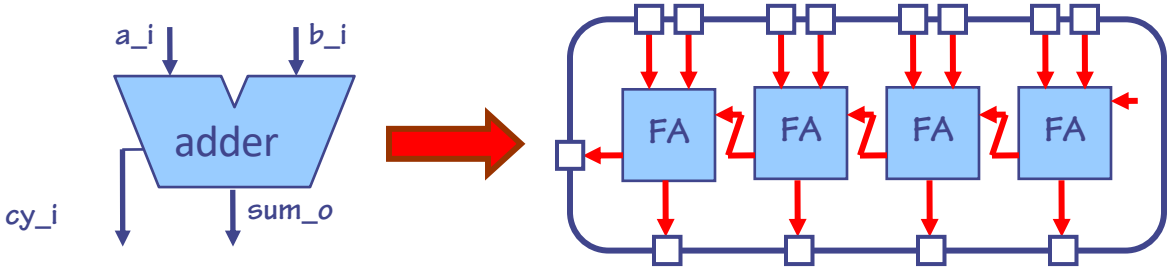


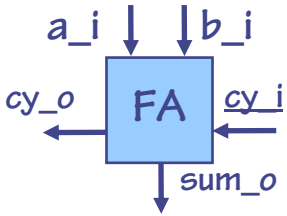

```
// HDL modeling of
// adder functionality
module adder( input  [3:0] a_i,
              input  [3:0] b_i,
              output       cy_o,
              output [3:0] sum_o );

endmodule
```

Ports must have a direction and a bitwidth.
In this class we use `_i` to denote in port variables and `_o` to denote out port variables.

Note the semicolon at the end of the port list!

A module can instantiate other modules

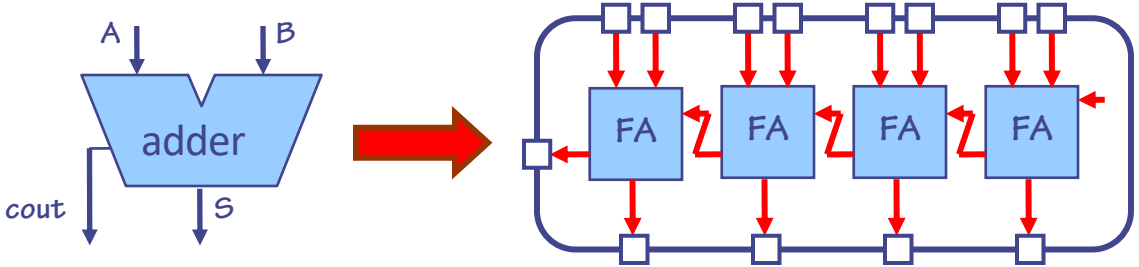





```
module adder( input  [3:0] a_i, b_i,
              output       cy_o,
              output [3:0] sum_o );

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

endmodule
```

```
module FA( input a_i, b_i, cy_i
           output cy_o, sum_o);
  // HDL modeling of 1 bit
  // full adder functionality
endmodule
```

Connecting modules




```
module adder( input  [3:0] a_i, b_i,
              output       cy_o,
              output [3:0] sum_o );

 logic c0, c1, c2;
 FA fa0( a_i[0], b_i[0], 1'b0, c0,    sum_o[0] );
 FA fa1( a_i[1], b_i[1], c0,   c1,    sum_o[1] );
 FA fa2( a_i[2], b_i[2], c1,   c2,    sum_o[2] );
 FA fa3( a_i[3], b_i[3], c2,   cy_o,  sum_o[3] );

endmodule
```

Carry Chain

Only connect ports by name and not by position.

Connecting ports by ordered list is compact but bug prone:

```
FA fa0( a_i[0], b_i[0], 1'b0, c0, sum_o[0] );
```

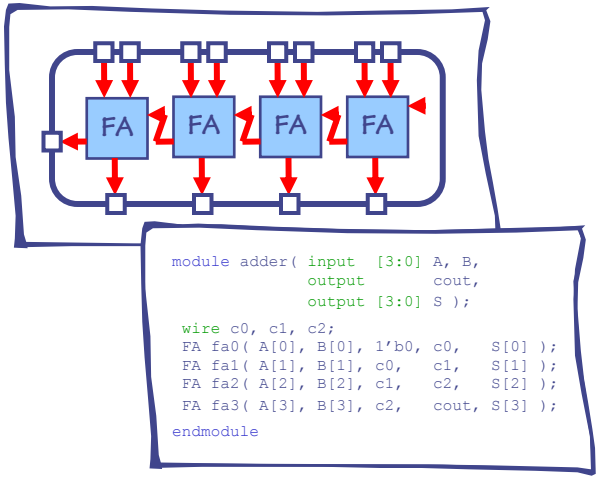
Connecting by name is less compact but leads to fewer bugs. This is how you should do it in this class. You should also line up like parameters so it is easy to check correctness.

```
FA fa0( .a_i(a_i[0])
       ,.b_i(b_i[0])
       ,.cy_i(1'b0)
       ,.cy_o(c0)
       ,.sum_o(sum_o[0])
      );
```

Connecting ports by name yields clearer and less buggy code. In the slides, we may do it by position for space. But you should do it by name and not position.

SV Fundamentals

- What is System Verilog?
- Data types
- Structural SV
- RTL SV
  - Combinational Logic
  - Sequential Logic

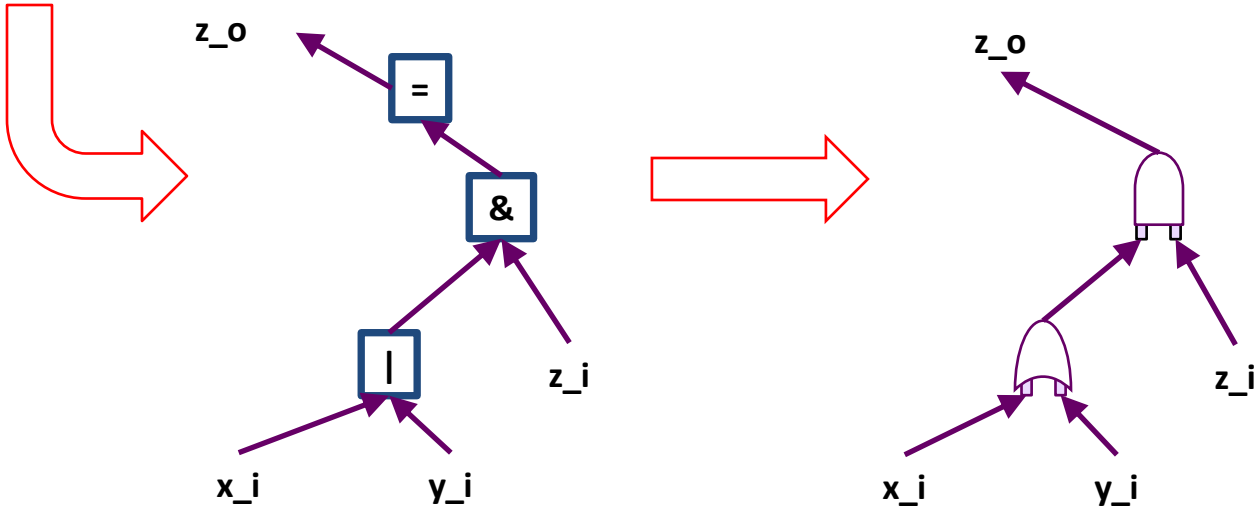


Combinational Verilog: `assign`

very straightforward mapping to hardware
variables are names of wires; operators are gates

```
assign z_o = (x_i | y_i) & z_i;
```

Language-defined operators:
| is 'OR'
& is 'AND'



parse tree

tree of gates

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

Example: mux4

mux4:
Using continuous assignments to generate combinational logic

```
module mux4( input  a_i, b_i, c_i, d_i,
             input [1:0] sel_i,
             output z_o );

logic t0, t1;

assign z_o = ~((t0 | sel_i[0]) & (t1 | ~sel_i[0]));
assign t1  = ~((sel_i[1] & d_i) | (~sel_i[1] & b_i));
assign t0  = ~((sel_i[1] & c_i) | (~sel_i[1] & a_i));

endmodule
```

A couple of combinational trees that connect to each other

The order of these continuous **`assign`** statements in the source code does not affect functionality – they are just specifying a bunch of gates – *a combinational cloud*.

Any time an input to the combinational cloud changes, it propagates through the cloud of gates and the outputs are updated. (Be careful not to create combinational cycles!)

mux4:
Using ? :

```
// Four input multiplexer
module mux4( input  a_i, b_i, c_i, d_i,
             input [1:0] sel_i,
             output z_o);

  assign z_o = ( sel_i == 0 ) ? a_i :
               ( sel_i == 1 ) ? b_i :
               ( sel_i == 2 ) ? c_i :
               ( sel_i == 3 ) ? d_i : 1'bx;

endmodule
```

*If **sel_i** is X or Z, without the 1'bX condition, it would get d_i in behavioural simulation but maybe not in timing. Bad!*

Having the 1'bx will help make sure your timing simulation looks the same as your behavioural.

mux4:
Using combinational always_comb or always @(*) block

```
module mux4( input  a_i, b_i, c_i, d_i,
             input [1:0] sel_i,
             output logic z_o );
  logic t0, t1;

  always_comb  // system verilog; replaces always @(*)
  begin
    t0  =   (sel_i[1] & c_i) | (~sel_i[1] & a_i);
    t1  = ~((sel_i[1] & d_i) | (~sel_i[1] & b_i));
    t0  = ~t0;
    z_o = ~( (t0 | sel_i[0]) & (t1 | ~sel_i[0]) );
  end

endmodule
```

Within the always_comb block, the synthesis tool synthesizes the lines in order.
Each L-value (variable to the left of =) creates a name for the wire that is at the top of a logic tree.
If a variable is assigned again (like t0), then the mapping is updated – no cycles are created.

always_comb permits more advanced combinational idioms

```
module mux4( input  a_i,b_i,c_i,d_i
             input [1:0] sel_i,
             output logic z_o);
```

```
always @*
  begin
    if (sel_i == 2'd0 )
      z_o = a_i;
    else if (sel_i == 2'd1)
      z_o = b_i;
    else if (sel_i == 2'd2)
      z_o = c_i;
    else if (sel_i == 2'd3)
      z_o = d_i;
    else
      z_o = 1'bx;
  end
endmodule
```

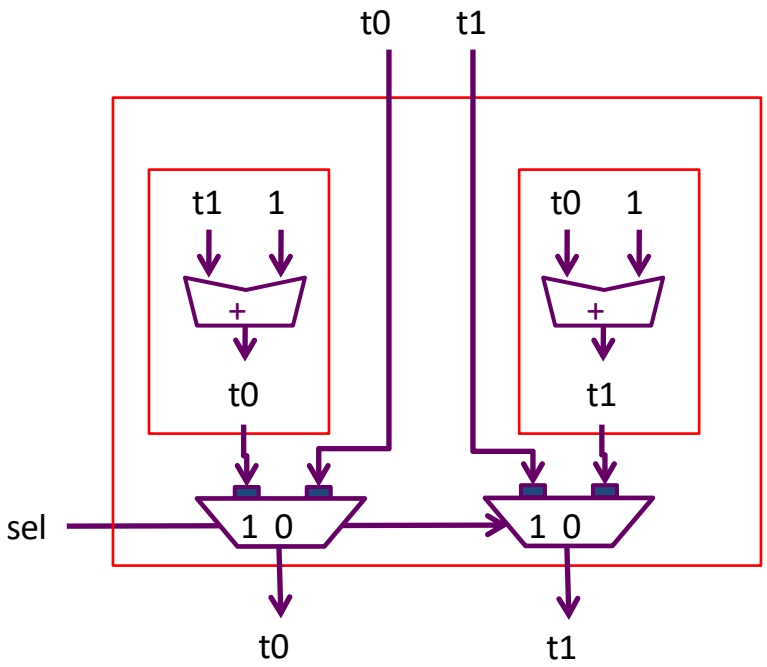
Good idea for avoiding behavioral versus timing mismatches.

If none of these match, behavioral will just use last value. Timing will give you an X probably.

```
always_comb
   begin
    case ( sel_i )
      2'd0 : z_o = a_i;
      2'd1 : z_o = b_i;
      2'd2 : z_o = c_i;
      2'd3 : z_o = d_i;
      default: z_o = 1'bX;
     endcase
  end
endmodule
```

Synthesis of if/else

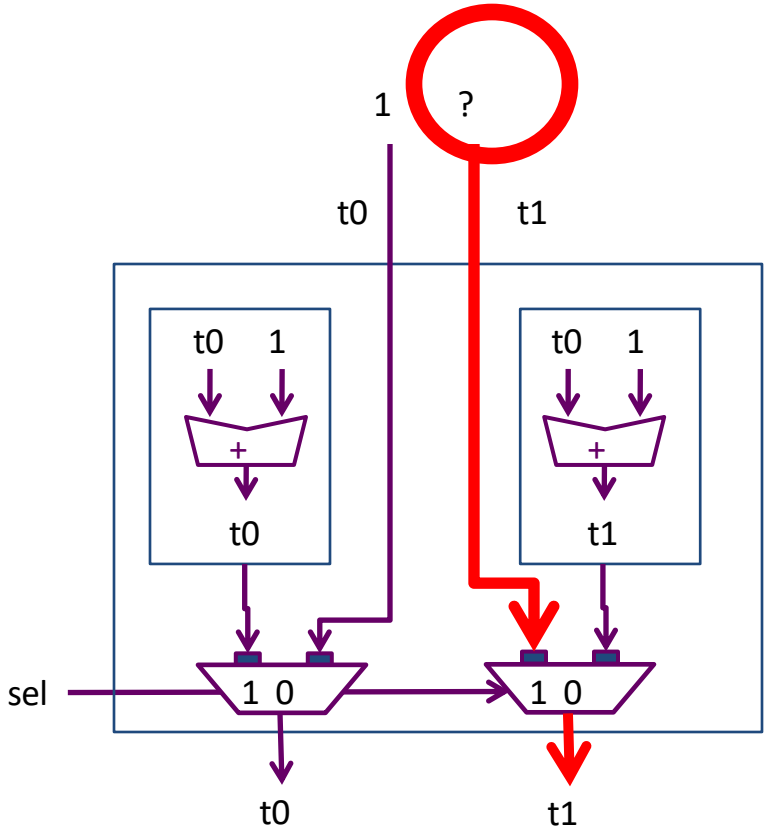
```
logic t0, t1, sel;

always_comb
 begin
 …
    if (sel)
       t0 = t1+1;
    else
       t1 = t0+1;

 …
 end
```



Note: a mux is created for every L-value written by all branches of the if/else or case statement.

Synthesis of if/else

```
logic t0, t1, sel;

always_comb
 begin
  t0 = 1'b1;

   if (sel)
      t0 = t0+1;
   else
      t1 = t0+1;

  ...
 end
```

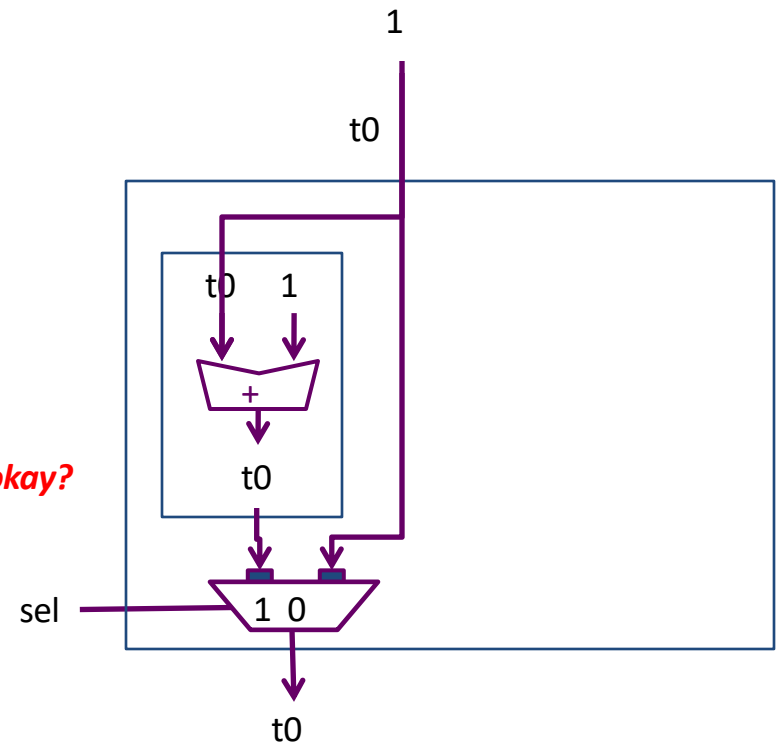


Note: no L-value should be undefined on any path; behavior is undefined; Verilog will create a latch (ugh)!

Synthesis of if/else

```
logic t0, t1, sel;

always_comb
 begin
  t0 = 1'b1;

  if (sel)
    t0 = t0+1;
  else
    ;

  …
 end
```

Is this example okay?



What happens if the case statement is not complete?

```
module mux3( input  a_i, b_i, c_i,
             input [1:0] sel_i,
             output logic z_o );

always @( * )
  begin
    case ( sel_i )
      2'd0 : z_o = a_i;
      2'd1 : z_o = b_i;
      2'd2 : z_o = c_i;
    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_i, b_i, c_i
             input [1:0] sel_i,
             output logic z_o );

always @( * )

  begin
    case ( sel_i )
      2'd0 : z_o = a_i;
      2'd1 : z_o = b_i;
      2'd2 : z_o = c_i;
      default : z_o = 1'bx;
    endcase
  end

endmodule
```

We CAN prevent creating a latch with a default statement

What happens if the case statement is not complete?

```
module mux3( input  a_i, b_i, c_i
             input [1:0] sel_i,
             output logic z_o );

always @( * )

always_comb

  begin
    case ( sel_i )
      2'd0 : z_o = a_i;
      2'd1 : z_o = b_i;
      2'd2 : z_o = c_i;
      default : z_o = 1'bx;
    endcase
  end

endmodule
```

SystemVerilog will protect you!

Be wary, many examples online are still plain ol' Verilog, and will work fine … until they don't ☹

Parameterized mux4

default value

```
module mux4 #( parameter width_p = 1 )
            ( input[width_p-1:0]  a_i, b_i, c_i, d_i,
              input [1:0] sel_i,
              output[width_p-1:0] z_o );

 wire [width_p-1:0] t0, t1;

 assign t0  = (sel_i[1]? c_i : a_i);
 assign t1  = (sel_i[1]? d_i : b_i);
 assign z_o = (sel_i[0]? t0 : t1);
endmodule
```

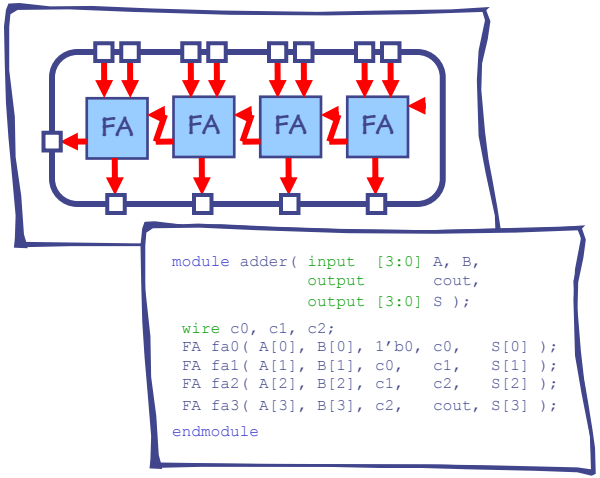
Parameterization is a good practice for reusable modules
Writing a mux*n* is challenging, but can be done with "idiomatic" verilog.

Instantiation

```
mux4#(.width_p(32))
alu_mux
( .a_i (op1),
  .b_i (op2),
  .c_i (op3),
  .d_i (op4),
  .sel_i(alu_mux_sel),
  .z_o(alu_mux_out) );
```

SV Fundamentals

- What is System Verilog?
- Data types
- Structural SV
- RTL SV
  - Combinational Logic
  - Sequential Logic



Sequential Logic: Creating a flip flop

note: always use <= with ***always_ff*** *and = with* ***always_comb***

```
1 logic q_r, q_n;
2 always_ff 3 @( posedge clk )
      q_r 4 <= q_n;
```

1) This line simply creates two signals, one called **q_r** and the other called **q_n**.

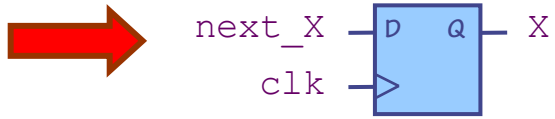
2) **always_ff** keyword indicates our intent to create registers; you could use the **always** keyword instead, but this makes it clear what you want!

3) **@(posedge clk)** indicates that we want these registers to be triggered on the positive edge of the **clk** clock signal.

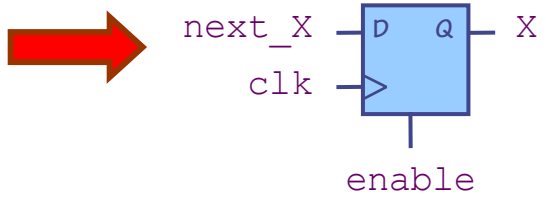
4) Combined with 2) and 3), the **<=** creates a register whose input is wired to **q_n** and whose output is wired to **q_r**. Use **_r** to indicate a wire that comes directly out of a register, and **_n** (i.e., next) to indicate a wire that goes directly into one, and becomes the new output on the next cycle.

Sequential Logic: flip-flop idioms

```
module FF0 (input clk, input  d_i,
            output logic q_r_o);
always_ff @( posedge clk )
  begin
     q_r_o <= d_i;
  end
endmodule
```



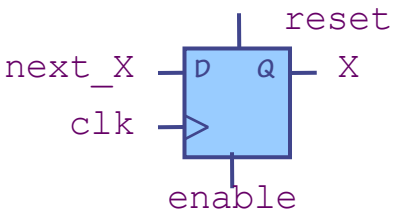

```
module FF (input clk, input  d_i,
           input  en_i, output logic q_r_o);
always_ff @( posedge clk )
  begin
    if ( en_i )
     q_r_o <= d_i;
  end
endmodule
```



idiom: flip-flops with reset

```
always_ff @( posedge clk)
begin
  if (reset)
    Q <= 0;
  else if ( enable )
    Q <= D;
end
```

synchronous reset



Register (i.e. a vector of parallel flip-flops)

```
module register#(parameter width_p = 1)
(
  input  clk,
  input  [width_p-1:0] d_i,
  input  en_i,
  output logic [width_p-1:0] q_r_o
);

  always_ff @( posedge clk )
  begin
    if (en_i)
      q_r_o <= d_i;
  end

endmodule
```

Implementing Wider Registers

```
module register2
(input  clk,
 input  [1:0] d_i,
 input  en_i,
 output logic [1:0] q_r_o
);

 always_ff @(posedge clk)
  begin
    if (en_i)
     q_r_o <= d_i;
  end

endmodule
```

```
module register2
( input  clk,
  input [1:0] d_i,
  input  en_i,
  output logic [1:0] q_r_o
);
 FF ff0 (.clk(clk),
  .d_i(d_i[0]),
          .en_i(en_i),
          .q_r_o(q_r_o[0]));

 FF ff1 (.clk(clk),
          .d_i(d_i[1]),
          .en_i(en_i),
          .q_r_o(q_r_o[1]));

endmodule
```

Do they behave the same?

yes

Syntactic Sugar: `always_ff` allows you to combine combinational and sequential logic; but this can be confusing.

more clear

```
module accum #(parameter width_p=1)
( input  clk,
  input  data_i,
  input  en_i,
  output logic [width_p-1:0] sum_o;
);

logic [width_p-1:0] sum_r, sum_next;
assign sum_o = sum_r;

always_comb
  begin
    sum_next = sum_r;

    if (en_i)
     sum_next = sum_r + data_i;
  end

always_ff @(posedge clk)
  sum_r <= sum_next;
```

shorter

```
module accum #(parameter width_p=1)
( input  clk,
  input  data_i,
  input  en_i,
  output logic [width_p-1:0] sum_o;
);

logic [width_p-1:0] sum_r;
assign sum_o = sum_r;

always_ff @(posedge clk)
  begin
    if (en_i)
     sum_r <= sum_r + data_i;
  end
```

Syntactic Sugar: You can always convert an `always_ff` that combines combinational and sequential logic into two separate `always_ff` and `always_comb` blocks.

shorter

```
module accum #(parameter width_p=1)
( input  clk,
  input  data_i,
  input  en_i,
  output logic [width_p-1:0] sum_o;
);

logic [width_p-1:0] sum_r;
assign sum_o = sum_r;

always_ff @(posedge clk)
  begin
    if (en_i)
     sum_r <= sum_r + data_i;
  end
```

more clear

```
module accum #(parameter width_p=1)
( input  clk,
  input  data_i,
  input  en_i,
  output logic [width_p-1:0] sum_o;
);

                                        1
reg [width_p-1:0] sum_r, sum_next;
assign sum_o = sum_r;

                                  2
always_comb
  begin
                                  2b
    sum_next = sum_r;

    if (en_i)                     2a
     sum_next = sum_r + data_i;
  end

                                        3
always_ff @(posedge clk)
  sum_r <= sum_next;
```

When in doubt, use the version on the right.

To go from the left-hand version to the right one:

1. For each register **`xxx_r`**, *introduce a temporary variable that holds the input to each register (e.g.* **`xxx_next`***)*

2. Extract the combinational part of the **`always_ff`** *block into an* **`always_comb`** *block:*

a. change **`xxx_r <=`** *to* **`xxx_next =`**

b. add **`xxx_next = xxx_r;`** *to beginning of block for default case*

3. Extract the sequential part of the **`always_ff`** *by creating a separate* **`always_ff`** *that does* **`xxx_r <= xxx_next;`**

Register array: we recommend you retain the `en_i` idiom in the `always_ff` block – could reduce # of ports.

shorter

```
always_ff @(posedge clk)
 if (en_i)
  sum_r[wr_i] <= foo + far;
```

more clear

```
always_comb
begin
    sum_cond_next = foo + far;
end

always_ff @(posedge clk)
 if (en_i)
  sum_r[wr_i] <= sum_cond_next;
```

shorter

```
always_ff @(posedge clk)
 if (en_i)
  sum_r[wr_i] <= foo + far;
```

extra ports? not so good.

```
always_comb
begin
    sum_cond_next =
        en_i ? (foo + far) : sum_r[wr_i];
end

always_ff @(posedge clk)
   sum_r[wr_i] <= sum_cond_next;
```

Bit Manipulations

```
logic [15:0] x;
logic [31:0] x_sext;
logic [31:0] hi, lo;
logic [63:0] hilo;

// concatenation
assign hilo = { hi, lo};
assign { hi, lo } = { 32'b0, 32'b1 };

// duplicate bits (16 copies of x[15] + bits 15..0 of x)
assign  x_sext =  {{16 { x[15] }}, x[15:0]};

// select top_p bits starting at 0 (same as [top_p-1:0])
assign  foo = x[0+:top_p];
```

Beware of assignment shortcuts

```
logic [15:0] x;
assign x = y;
```

```
logic [15:0] x = y;
```

"initialization"
non-synthesizable

```
wire [15:0] x = y;
```

"continuous assignment"
synthesizable

"Unlike nets, a variable cannot have an implicit continuous assignment as part of its declaration. An assignment as part of the declaration of a variable is a variable initialization, not a continuous assignment."

IEEE 1800-2009 (SystemVerilog Standard) p. 50

`unique` and `priority` for `case` and `if`

`unique` ***exactly one*** *branch or case item* ***must execute****; otherwise it is an error.*

`priority` *choices* ***must*** *be* ***evaluated in order****, and that* ***one branch must execute****.*

Synopsys VCS: Does not generate X output, just says:

RT Warning: No condition matches in 'unique case' statement.
"system.v", line 20, for testbench.dut.cu, at time 100.

So, using 1'bX as the default condition still has some purpose, since it shows up in the waveform viewer. On the other hand, this tells you when the issue happens.

Note: Our SV Subset

- SV is a big language with many features not concerned with synthesizing hardware.
- The code you write for your processor should contain only the language structures discussed in these slides.
- Anything else is not synthesizable, although it will simulate fine.
- We will be mixing in some more synthesizable SystemVerilog later in the course to improve maintainability of your code.