

CSE 141L: Introduction to Computer Architecture Lab

SystemVerilog II

Pat Pannuto, UC San Diego

ppannuto@ucsd.edu

Milestone 1 is due in 48 hours [before class Wed]

- What to submit?
 - SOMETHING
- M1 is graded for completion, not accuracy
 - The purpose of milestones is to *help you* manage large, long-term project
 - TAs will use gradescope “grades” to help give feedback
 - Recall: Only Milestone 4 (final submission) is actual grade*
 - *With exceptions for things such as skipping milestones altogether

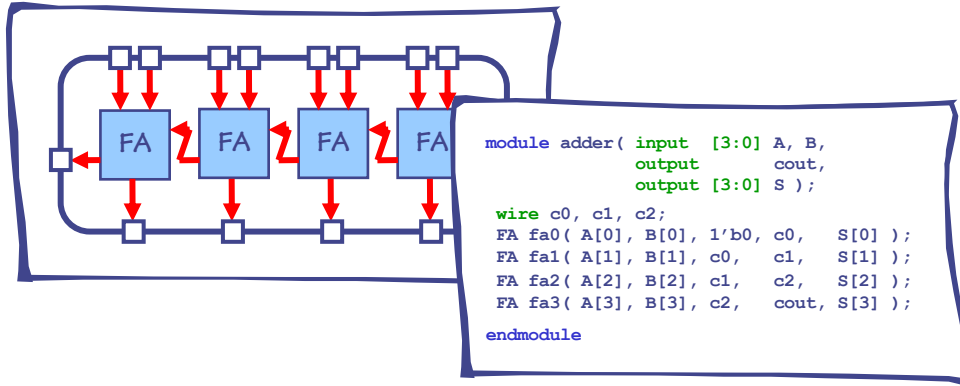
Today's Objectives:

Learning more about SystemVerilog

- We'll get through ~half+ these slides today, and the rest on Wednesday
- Treat these slides as a reference
 - It'll go kind of fast, goal is to expose to you things you can learn more about on your own

Logistics Updates – Course Modality Poll

- **Not matter what**, remote attendance will be an option all quarter
 - Will always stream via Zoom and recordings in Canvas
- **Current guidance is that we *can* resume in-person instruction next week**
 - We are also permitted up to 5 weeks to shift modality
 - And this class does not meet during week 10 [all project hours]
- **POLL: Would you come to CENTR 115 M/W from 12-1 every day?**
 - A: Definitely Yes
 - B: Maybe leaning Yes
 - C: Maybe leaning No
 - D: Definitely No



[Picking this back up from week 1]

SYNTHESIZABLE SYSTEM VERILOG I – FUNDAMENTALS

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) );
```

A couple of combinational trees that connect to each other



```
endmodule
```

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);
```

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 @*
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
```

```
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)
```

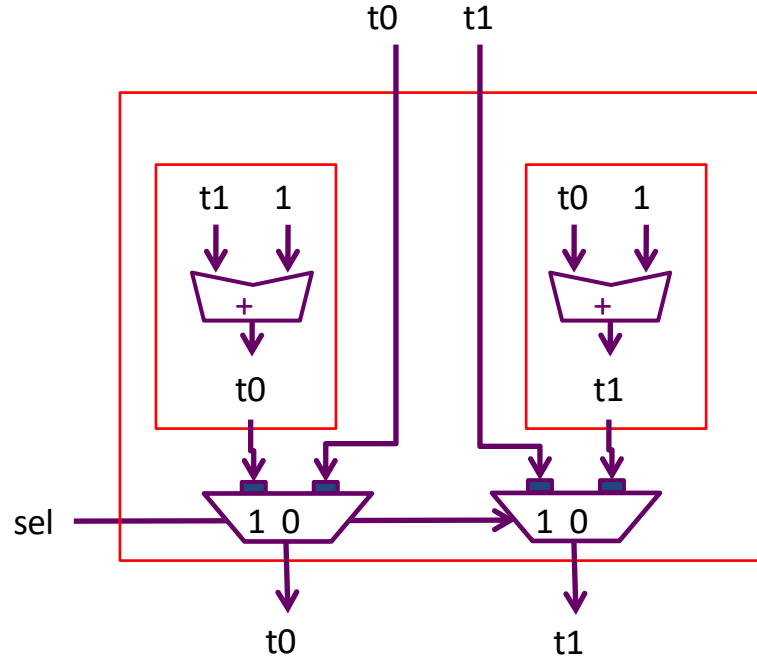
```
    t0 = t1+1;
```

```
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;
```

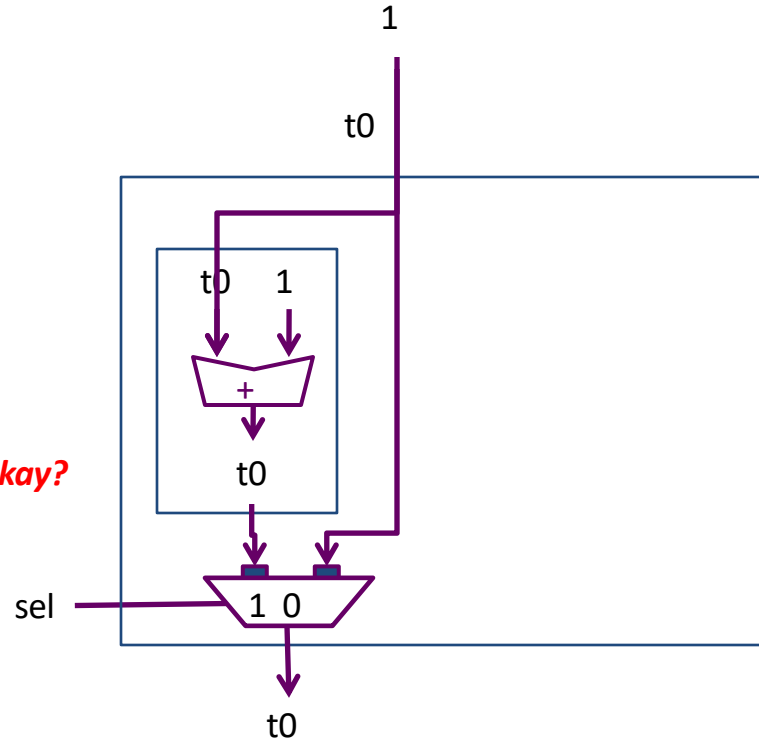
```
  else
```

```
    ;
```

```
  ...
```

```
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

```
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 );
```

default value

```
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
```

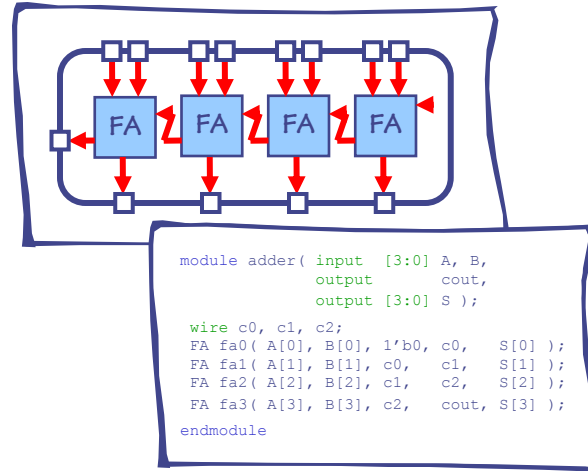
Parameterization is a good practice for reusable modules
Writing a mux n is challenging, but can be done with
“idiomatic” verilog.

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`.

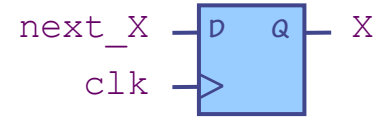
2) `always_ff` keyword indicates our intent to create registers; you could use the `always` keyword instead, but this makes it clear what you want!

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

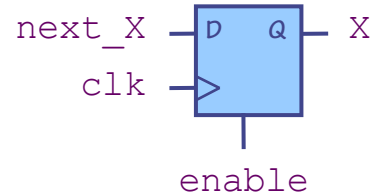
4) Combined with 2) and 3), the `<=` creates a register whose input is wired to `q_n` and whose output is wired to `q_r`. Use `_r` to indicate a wire that comes directly out of a register, and `_n` (i.e., next) to indicate a wire that goes directly into one, and becomes the new output on the next cycle.

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
```

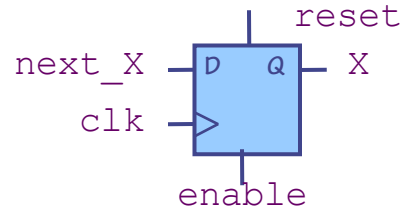


```
module FF (input clk, input d_i,  
            input en_i, output logic q_r_o);  
always_ff @( posedge clk )  
begin  
    if ( en_i )  
        q_r_o <= d_i;  
end  
endmodule
```



idiom: flip-flops with reset

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



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
```

Do they behave the same?

yes

```
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
```

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

more clear

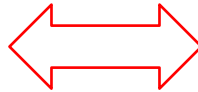
```
module accum #(parameter width_p=1)
  ( input  clk,
    input  data_i,
    input  en_i,
    output logic [width_p-1:0] sum_o;
  );

  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
```

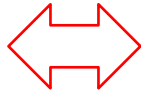

Syntactic Sugar: You can always convert an `always_ff` that combines combinational and sequential logic into two separate `always_ff` and `always_comb` blocks.

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;
  );

  reg [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;
```

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

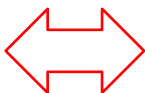
beginning of block for default case

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;
```



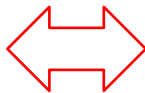
more clear

```
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
```

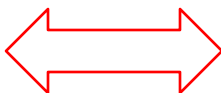
```
always_ff @(posedge clk)
  sum_r[wr_i] <= sum_cond_next;
```

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;
```

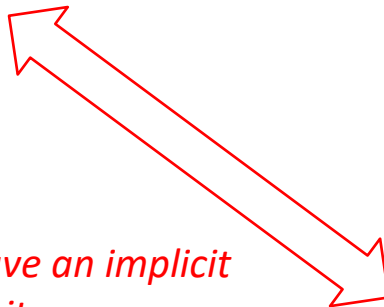


```
logic [15:0] x = y;
```



“initialization”
non-synthesizable

“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.”



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



“continuous assignment”
synthesizable

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**.

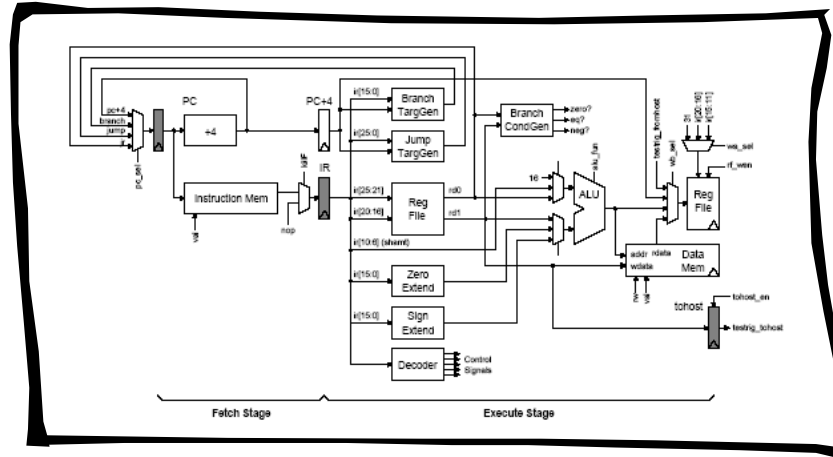
Synopsys VCS: Does not generate X output, just says:

```
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.

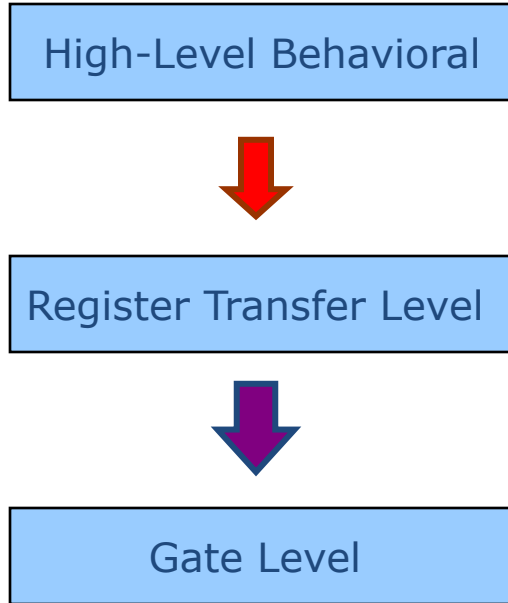
Note: Our SV Subset

- 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.



SYSTEM VERILOG II - DESIGN EXAMPLES

Verilog can be used at several levels



A common approach is to use C/C++ for initial behavioral modeling, and for building test rigs

automatic tools to synthesize a low-level gate-level model

Recap: Combinational logic

- Use continuous assignments (**assign**)

```
assign c_i = b_o + 1;
```

- Use **always_comb** blocks with blocking assignments (=)

```
always_comb
begin
    out = 2'd0;
    if (in1 == 1)
        out = 2'd1;
    else if (in2 == 1)
        out = 2'd2;
end
```

default value

always blocks allow more expressive control structures, though not all will synthesize

- Every variable should have a *default value* to avoid inadvertent introduction of latches
- Don't assign to same variable from more than one **always** block. Race conditions in behavioral sim, synthesizes incorrectly.

Recap: Sequential Logic

- Use `always_ff @(posedge clk)` only with non-blocking assignment operator (`<=`)

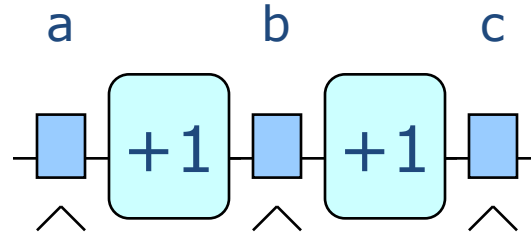
```
always_ff @(posedge clk )  
    c_o <= c_i;
```
- Careful when mixing pos-edge & neg-edge triggered flip-flops
- Do not assign the same variable from more than one `always` block. Race condition in behavioral simulation; synthesizes incorrectly.
- Do not mix blocking and non-blocking assignments
 - only use non-blocking assignments (`<=`) for sequential logic.
 - only use block assignments (`=`) for combinational logic.
- Like in software engineering, express your design as a module hierarchy that corresponds to logical boundaries in the design. Also, separate datapath and control (more later).

An Example: Good Style

```
logic a_n, b_n, c_n;  
logic a_r, b_r, c_r;
```

```
always_ff @( posedge clk )  
begin  
    a_r <= a_n;  
    b_r <= b_n;  
    c_r <= c_n;  
end
```

```
assign b_n = a_r + 1;  
assign c_n = b_r + 1;
```

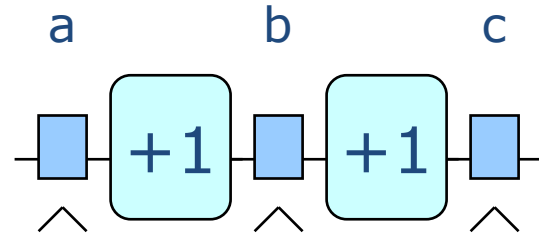


Readable, combinational and sequential logic are separated.

Consistent naming: A_r is the output of the register and A_n (or A_n) is the input.

An Example: Good Style

```
logic a_n;  
logic b_n, c_n;  
logic a_r, b_r, c_r;  
  
always_ff @( posedge clk )  
begin  
    a_r <= a_n;    // list in any order  
    b_r <= b_n;  
    c_r <= c_n;  
end  
  
always_comb  
begin  
    b_n = a_r + 1; // triggers when a_r or b_r changes  
    c_n = b_r + 1;  
end
```

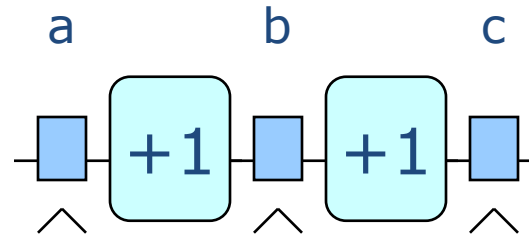


Readable,
combinational and
sequential logic are
separated.

Alternate implementation?

```
logic a_n, b_n, c_n;
logic a_r, b_r, c_r;

always_ff @( posedge clk )
begin
    a_r <= a_n;
    b_r <= b_n;
    c_r <= c_n;
    assign b_n = a_r + 1;
    assign c_n = b_r + 1;
end
```



Syntactically
Incorrect.

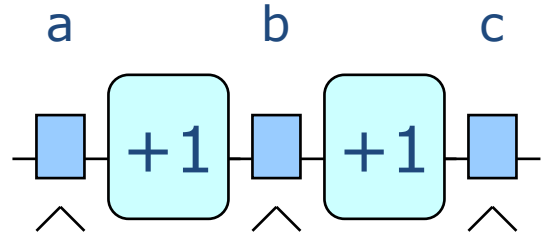
An Example: Okay, but less readable?

```
logic a_n;  
logic a_r, b_r, c_r;  
  
always_ff @( posedge clk )  
begin  
    a_r <= a_n;  
    b_r <= a_r + 1;  
    c_r <= b_r + 1;  
end
```

} — Is $(b_r == a_n + 1)$?

Nope - Why?

$a_r <= a_n$ creates
a register between
 a_n and a_r , not a wire.



Another style – multiple always blocks

```
logic a_n, b_n, c_n;
logic a_r, b_r, c_r;

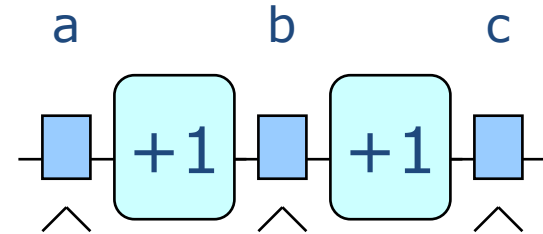
always_ff @( posedge clk )
    a_r <= a_n;

assign b_n = a_r + 1;

always_ff @( posedge clk )
    b_r <= b_n;

assign c_n = b_r + 1;

always_ff @( posedge clk )
    c_r <= c_n;
```



Does it have the
same functionality?

Yes. But why?

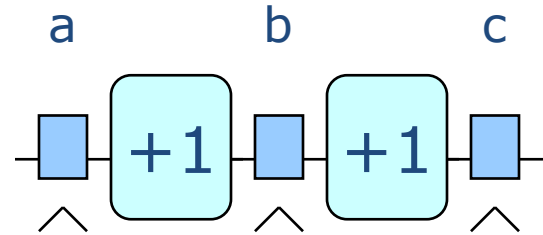
It generates the
same
underlying circuit.

How about this one?

```
logic a_n, b_n, c_n;
logic a_r, b_r, c_r;

always @(posedge clk)
begin
    a_r = a_n;
    b_r = b_n;
    c_r = c_n;
end

assign b_n = a_r + 1;
assign c_n = b_r + 1;
```



Will this synthesize?

→ Maybe

Is it correct?

→ No; Don't use "blocking assignments" in @posedge clk blocks. It creates race conditions. Also, use always_ff instead.

What does this do? (This is correct but bad code.)

```
logic b_i, c_i;
logic a_r;
logic sel;

always @( posedge clk )
begin
    a_r <= 1'b0;
    a_r <= b_i;

    if (sel)
        a_r <= c_i;
end
```

Desugar into separate comb. and seq. logic.

```
logic b_i, c_i;
logic a_r;
logic sel;

always @( posedge clk )
begin
    a_r <= 1'b0; // redundant!
    a_r <= b_i;

    if (sel)
        a_r <= c_i;
end
```

```
logic a_n, b_i, c_i;
logic a_r;
logic sel;

always_comb
begin
    a_n = a_r; // default;
                // rdt. but safe
    a_n = b_i;

    if (sel)
        a_n = c_i;
end

always_ff @( posedge clk )
    a_r <= a_n;
```

What does this do?

For each `always_comb`, `assign`, `always_ff` statement, draw the gates and wires.

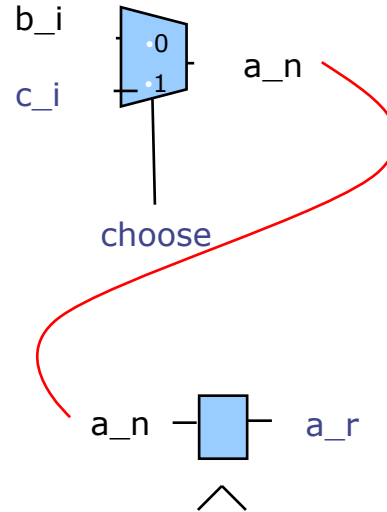
```
logic a_n, b_i, c_i;
logic a_r;
logic choose;

always_comb
begin
    a_n = a_r; // default

    a_n = b_i;

    if (choose)
        a_n = c_i;
end

always_ff @( posedge clk )
    a_r <= a_n;
```



Verilog execution semantics

- Confusing
- Best solution is to write synthesizable Verilog that corresponds exactly to logic you have *already* designed on paper. Separate combinational from sequential logic.
- Debugging is difficult for Verilog. Don't write code and "see if it works." Test each "unknown" thing individually until you know what it does; then combine into larger entities.
- Before you try to simulate, manually check every wire to make sure that it is correctly (1) defined, connected to (2) source and (3) destination, and that (4) the logic driving it appears to be correct.
 - This is *faster* than finding the same bugs in the waveform viewer!

Useful operators: reduction

```
(| c) // all of the bits of C or'd together (“or reduce”)  
(& c) // all of the bits of C and'd together (“and reduce”)  
(^ c) // all of the bits of C xor'd together (“xor reduce”)
```

SystemVerilog struct example

```
typedef struct packed {
    logic [17-1:0] instr;
    logic [10-1:0] addr;
} instr_packet_s;

instr_packet_s ip_n, ip_A_r, ip_B_r, ip_C_r;

assign ip_n = `{addr: addr_i
               , instr: instr_i};

assign { addr_o, instr_o }
       = { ip_C_r.addr, ip_C_r.instr };

always_ff @( posedge clk )
    { ip_A_r, ip_B_r, ip_C_r } <=
      { ip_n, ip_A_r, ip_B_r };
```

Helpful SV Primitives

`$bits()` - number of bits required to hold value:

```
logic[31:0] foo; // $bits(foo) = 32
```

```
struct happy foo; // $bits(struct happy) =
```

`$clog2()` - number of address bits for a memory of size

- ceiling of log base 2 of X (eg for RF impl.)

```
// $clog2(2) = 1, $clog2(3,4) = 2,
```

```
// $clog2(5,6,7,8) = 3, ...
```

`'1,'0,'X,'Z` - all 1's, all 0's, all X's, etc.

```
logic [63:0] Word; logic [3:0] Byte; // byte = keyword
```

```
Word[Byte*8 +: 8] ; start at Byte*8; grab 8 bits upwards
```

```
Word[Byte*8 -: 8]; start at Byte*8; grab 8 bits downwards
```

Helpful SV Primitives, Cont.

```
typedef enum [2:0] { eFetch, eDecode, eExecute, eMemory, eWriteback } state_e;
```

```
state_e substate_r, substate_next;
```

```
always_ff @(posedge clk)
    substate_r <= reset ? eFetch : substate_next;
```

```
always_comb
```

```
unique case (substate_r)
```

```
    eFetch:    substate_next = eDecode;
    eDecode:  substate_next = eExecute;
    eExecute:
```

```
        unique casez (instr_reg)
            `SW: `LW: substate_next = eMemory;
            default: substate_next = eWriteback;
        endcase
```

```
    eMemory:
```

```
        unique casez(instruction)
            `LW:    substate_next = eWriteback;
            default: substate_next = eFetch;
        endcase
```

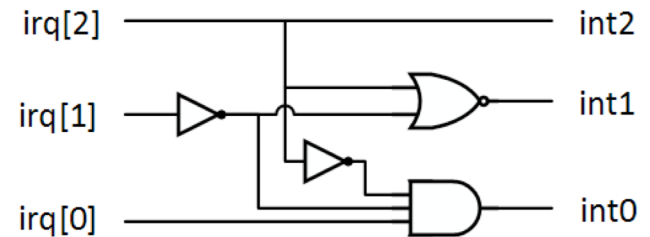
```
    eWriteback: substate_next = eFetch;
    default:    substate_next = eFetch;
```

```
endcase
```

Breakdown of Verilog case types:

<http://www.verilogpro.com/verilog-case-casez-casex/>

```
always @(irq) begin
    {int2, int1, int0} = 3'b000;
    casez (irq)
        3'b1?? : int2 = 1'b1;
        3'b?1? : int1 = 1'b1;
        3'b??1 : int0 = 1'b1;
        default: {int2, int1, int0} = 3'b000;
    endcase end
```



That was a lot.

Remember, these slides are meant as a reference.

You are not expected to have internalized all of this in real time in lecture, but to have 'aha' moments when implementing.