mp_verif

Part 2

Waveform!

Use waveform when in doubt

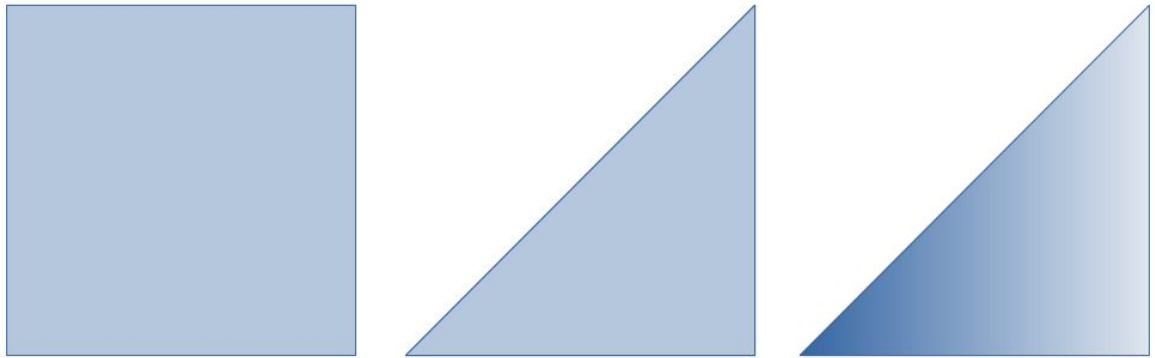
r_type? i_type?

```
83      constraint instr_c {
84          // Reg-imm instructions
85          instr_type[0] -> {
86            instr.i_type.opcode == op_imm;
87
88            // Implies syntax: if funct3 is sr, then funct7 must be
89            // one of two possibilities.
90            instr.r_type.funct3 == sr -> {
91              instr.r_type.funct7 inside {base, variant};
92            }
93
94            // This if syntax is equivalent to the implies syntax above
95            // but also supports an else { ... } clause.
96            if (instr.r_type.funct3 == sll) {
97              instr.r_type.funct7 == base;
98            }
99          }
```

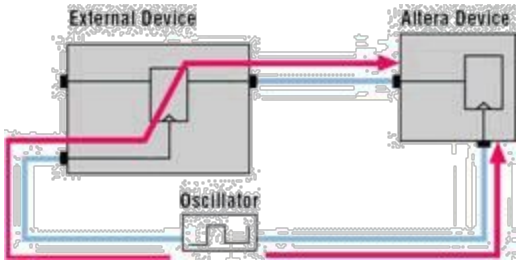
```
typedef union packed {
  bit [31:0] word;

  struct packed {
    bit [11:0] i_imm;
    bit [4:0] rs1;
    bit [2:0] funct3;
    bit [4:0] rd;
    rv32i_opcode opcode;
  } i_type;

  struct packed {
    bit [6:0] funct7;
    bit [4:0] rs2;
    bit [4:0] rs1;
    bit [2:0] funct3;
    bit [4:0] rd;
    rv32i_opcode opcode;
  } r_type;
```

Solve order

```
class RandEx;
  rand bit [31:0] a;
  rand bit [31:0] b;
  constraint ab_c {
    a < b;
    b < 32'h80000000;
    solve b before a;
  }
endclass : RandEx
```



Input delay



How to find critical paths?

- Run `make synth` to synthesize the design
- Look at the timing report located at `/synth/reports/timing.rpt`
- What does all the text mean?

```
Startpoint: rst (input port clocked by my_clk)
Endpoint: clk_gate_rand_bit_reg/latch
          (gating element for clock my_clk)
Path Group: my_clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
lfsr               5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                   Incr       Path
-----------------------------------------------------------------------------
clock my_clk (rise edge)                                0.000000   0.000000
clock network delay (ideal)                             0.000000   0.000000
input external delay                                    0.500000   0.500000 r
rst (in)                                                0.000000   0.500000 r
U29/ZN (INV_X1)                                         0.017293   0.517293 f
U30/ZN (INV_X1)                                         0.087238   0.604531 r
U31/ZN (OR2_X1)                                         0.053453   0.657984 r
clk_gate_rand_bit_reg/EN (SNPS_CLOCK_GATE_HIGH_lfsr)
                                                        0.000000   0.657984 r
clk_gate_rand_bit_reg/latch/E (CLKGATETST_X1)           0.007406   0.665390 r
data arrival time                                                  0.665390

clock my_clk (rise edge)                               10.000000  10.000000
clock network delay (ideal)                             0.000000  10.000000
clk_gate_rand_bit_reg/latch/CK (CLKGATETST_X1)          0.000000  10.000000 r
clock gating setup time                                -0.080193   9.919806
data required time                                                 9.919806
-----------------------------------------------------------------------------
data required time                                                 9.919806
data arrival time                                                 -0.665390
-----------------------------------------------------------------------------
slack (MET)                                                        9.254416
```

Analyzing the timing report

Input delay caused by input capacitance

Start of the critical path (probably a register)

Combinational logic gates passed through

End of the critical path (probably also a register)

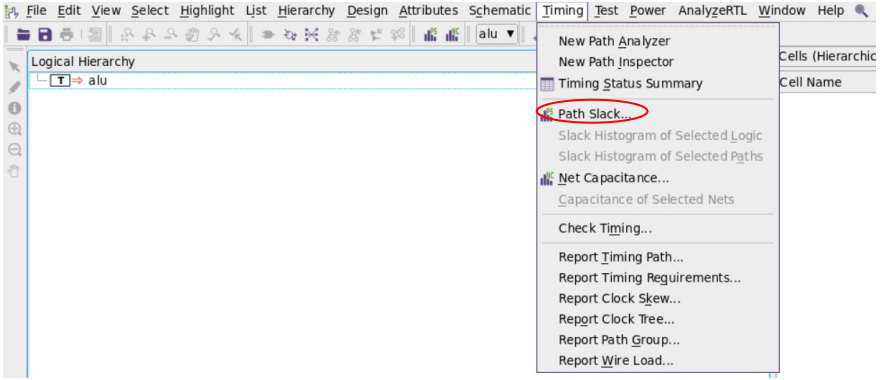
But you may be asking, how do I know what this corresponds to in HDL?

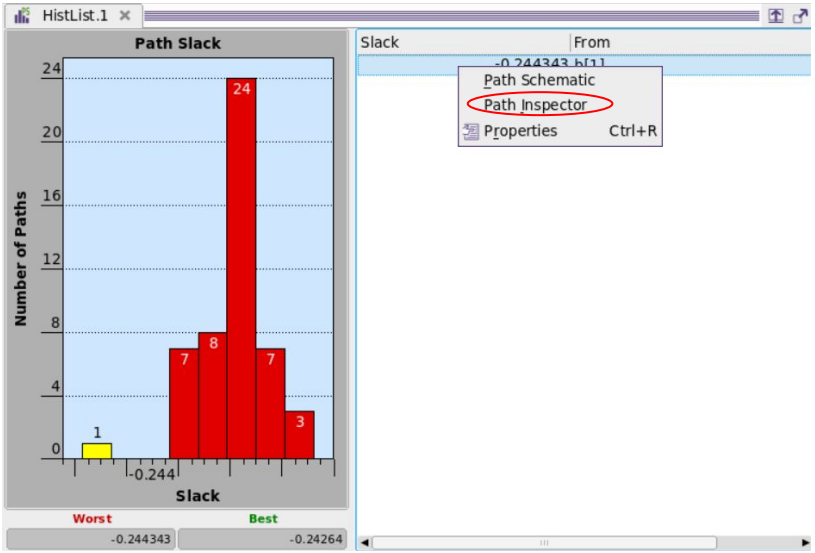
- Use DesignVision!!!!

```
Startpoint: b[1] (input port clocked by my_clk)
Endpoint: mux_reg[3][9]
          (rising edge-triggered flip-flop clocked by my_clk)
Path Group: my_clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
alu                5K_hvratio_1_1        NangateOpenCellLibrary

Point                                    Incr       Path
-----------------------------------------------------------
clock my_clk (rise edge)                 0.000000   0.000000
clock network delay (ideal)              0.000000   0.000000
input external delay                     1.000000   1.000000 f
b[1] (in)                                0.000000   1.000000 f
U1953/ZN (INV_X1)                        0.031841   1.031841 r
U1783/ZN (AND2_X1)                       0.075015   1.106856 r
U2066/Z (BUF_X2)                         0.073130   1.179987 r
U1520/ZN (AND2_X2)                       0.055272   1.235259 r
U2238/ZN (NOR3_X1)                       0.023008   1.258267 f
U2889/ZN (OR2_X1)                        0.050738   1.309005 f
U2890/ZN (OAI21_X1)                      0.032366   1.341371 r
U2891/ZN (INV_X1)                        0.026303   1.367674 f
U2892/ZN (NAND3_X1)                      0.029454   1.397128 r
U2893/ZN (AOI22_X1)                      0.034926   1.432054 f
U2894/ZN (NAND2_X1)                      0.031205   1.463259 r
U1697/ZN (NAND3_X1)                      0.027980   1.491239 f
U1910/ZN (AND2_X1)                       0.035448   1.526688 f
mux_reg[3][9]/D (DFF_X1)                 0.008556   1.535244 f
data arrival time                                   1.535244

clock my_clk (rise edge)                 1.430000   1.430000
clock network delay (ideal)              0.000000   1.430000
clock uncertainty                       -0.100000   1.330000
mux_reg[3][9]/CK (DFF_X1)                0.000000   1.330000 r
library setup time                      -0.039099   1.290901
data required time                                  1.290901
-----------------------------------------------------------
data required time                                  1.290901
data arrival time                                  -1.535244
-----------------------------------------------------------
slack (VIOLATED)                                   -0.244343
```

How to use DesignVision

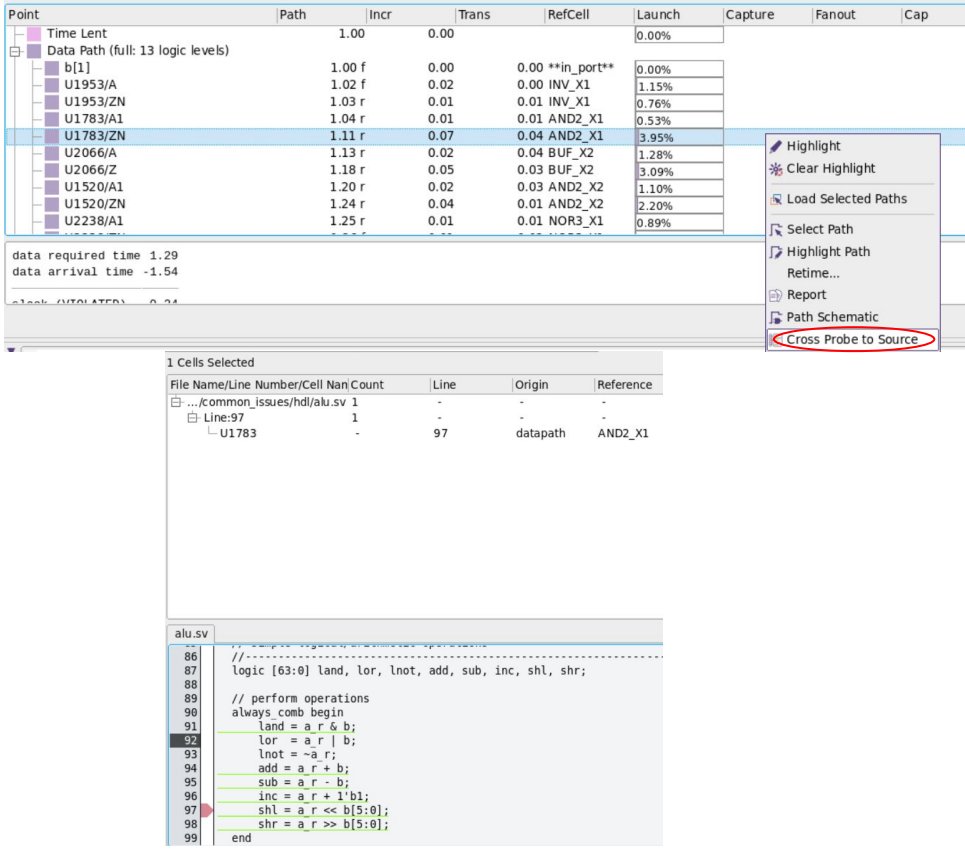
- After running `make synth`, run `make dv` to open up the DesignVision GUI
- From the bar on the top, navigate to Timing and then Path Slack, and click OK on the window that opens
- Choose the rightmost bar of the bar graph, right-click on the slack that pops up on the right, and select Path Inspector





How to use DesignVision (cont.)

- Right-click on one of the gates that appears from the critical path, and click Cross Probe to Source to see where it originates from
- The red icon indicates the line corresponding to the gate
- Use DesignVision to analyze critical path and see what parts can be optimized!



Verilog (Style) Guidelines

Slides by Scott Smith

Writing Verilog HDL

HDL - Hardware Description Language

- Remember at all times: **we are describing circuits**
  - This is not a programming language!
  - Always be thinking about what circuit will be created to realize your design.
- We write **behavioral** designs in this class.
  - A behavioral design describes the *behavior* of a circuit, not the specific gates/devices used to implement the circuit.
  - We let the synthesis tools take our (relatively) high-level design and map it to specific hardware gates/flops/etc
  - Example behavioral Verilog operators: ***, +, -,** and **/**
  - Example behavioral Verilog constructs: **if, case**
  - This saves us time by letting us focus on the big picture.

Tips for Writing HDL

- Draw a block design for how the different parts of your system will interact before you write any code.
- Write clean Finite State Machines (FSMs) for all control logic.
  - Tips on this in a few slides
- Avoid latches (see lab1)
- Use modules for large chunks of hardware.
- Remember that tons of stuff is happening concurrently. You aren't writing a serial program.
- **If you can't roughly guess the hardware that will be generated by your design, it is a sign you need to rethink/rewrite/replan your code.**

Verilog Procedures

All Verilog Procedures run concurrently!

Procedures syntax:

```
always @(sensitivity list) begin
        …
end
```

Example Procedure:

```
always @* begin
        x = 5;
end
```

SystemVerilog Procedures

Sequential Logic Procedures

```
always_ff @(posedge clk) begin
      if (rst) x <= 0;
      else x  <= x_next;
end
```

The above will produce a positive-edge triggered flop with a synchronous reset.

Combinational Logic Procedures

```
always_comb begin
      x_next = 5;
end
```

always_comb is the same as always @*

Variable Types

- reg - used in procedural blocks
- wire - used in assign statements
- logic - can be used anywhere

We recommend using logic. It can be used anywhere and leads to cleaner code.

Although logic can be used anywhere, a particular variable can only be assigned in either an assign statement **or** inside of **one** always block.

```
logic x;
assign x = 5;
always_comb begin
        x=4;
end
```

Illegal

OK

```
logic x;
assign x = 5;
```

```
logic x;
always_comb begin
        x=4;
end
```

Synthesizable vs Non-Synthesizable Verilog

We write two types of Verilog in this class.

Synthesizable:

- Can be realized in hardware (the behavioral specification can be mapped to specific gates/flops/etc)
- Only a subset of Verilog's features are synthesizable. For example, you can't synthesize a $display() or a delay (#5)

Non-Synthesizable:

- Used to control the simulator/compiler
- We write a lot of non-synthesizable code in our testbenches

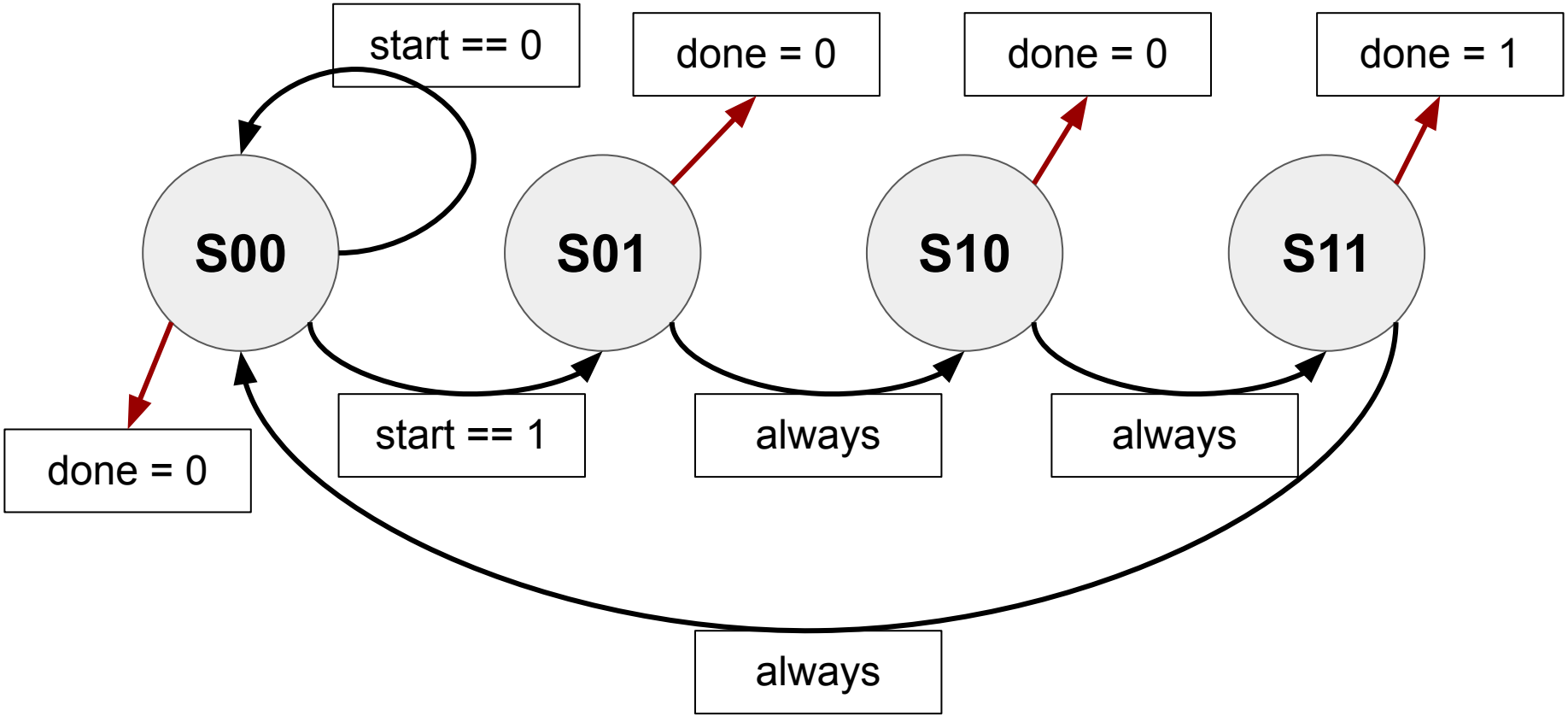
Non-Synthesizable

```
initial begin
  @(posedge clk);
   #10 a = 8'd45; b = 1'b1;
   $disply("tacos are yummy");
 end
always_comb begin
        for (i = 0; i < a; i++) begin
                $display("I am hungry");
        end
end
```

Synthesizable

```
assign x = a + b;
always_comb begin
   y = !x;
end
for (i = 0; i < 3; i++) a[i] = 0;
```

Writing a State Machine



Writing a State Machine

Split your code into 4 pieces:

1. **your variable & I/O declarations,**
2. your next-state logic,
3. your output logic,
4. your flip-flop logic

```
module example(
        input logic clk,
        input logic rst,
        input logic start,
        output logic done,
);
enum logic [1:0] {
        S_1, S_2, S_3, S_4
}state, next_state;
```

Writing a State Machine

Split your code into 4 pieces:

1. your variable & I/O declarations,
2. **your next-state logic,**
3. your output logic,
4. your flip-flop logic

```
always_comb begin
        next_state = state; // default value
        case (state) begin
                S_0: begin
                        if (valid) next_state =S_1;
                end
                …
        end
end
```

Writing a State Machine

Split your code into 4 pieces:

1. your variable & I/O declarations,
2. your next-state logic,
3. **your output logic,**
4. your flip-flop logic

```
assign done = state == S_4;
```

Writing a State Machine

Split your code into 4 pieces:

1. your variable & I/O declarations,
2. your next-state logic,
3. your output logic,
4. **your flip-flop logic**

```
always_ff @(posedge clk) begin
        if (rst) state <= 0;
        else state <= next_state;
end
```

Other tips

Always use <= for assignment inside of always_ff

Always use = for assignment inside of always_comb

Keep your always_ff blocks as simple as possible. Keep your complex logic in assign statements and always_comb blocks.

Use synchronous resets (don’t add reset to always_ff sensitivity list)

Keep your code simple. No need to create a module for simple functionality that can be expressed using higher-level system verilog syntax. For example, use the ternary operator or if statements instead of designing a MUX module.