Digital VLSI chip design: from Verilog to Layout

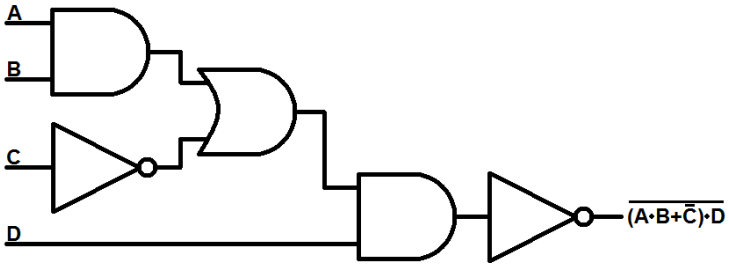
Course Tutorial for ELEC5160 / EESM5020

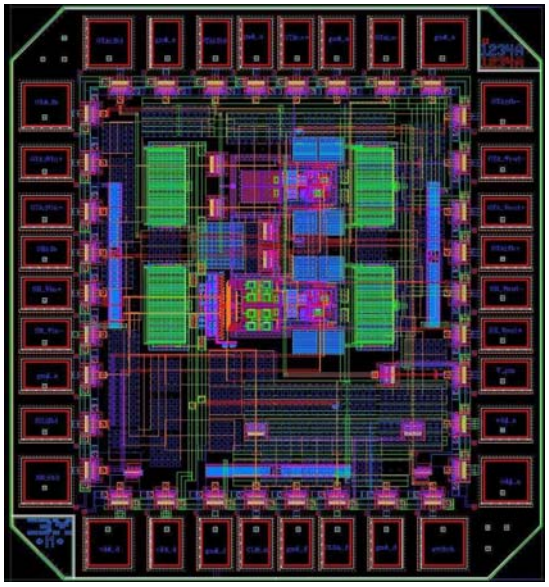
ZHU, Jingyang
jzhuak@connect.ust.hk

Outline

- Register-transfer level (RTL) language
- Synthesize technique
- Placement and routing (P&R)

```
module Top(
  A, B, C, D, OUT
);

…

endmodule
```





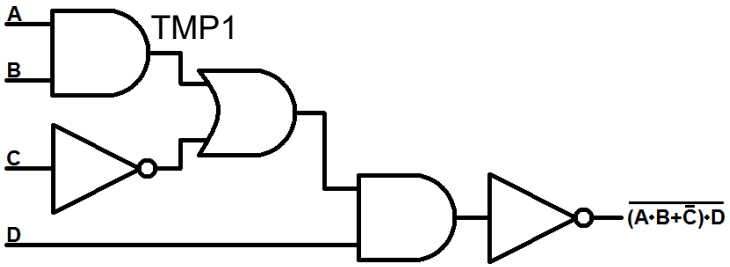
High level description

Gate-level netlist

Physical level layout

Hardware description language (HDL)

- Different types of languages (like C/C++, Java, Python, etc.)
  - VHDL (*.vhd)
  - Verilog (*.v)
  - System Verilog (*.sv)
- Key concepts
  - HDL is simple regarding to the syntax
  - HDL is complex regarding to the thinking pattern
- Keep in mind that HDL is the language of describing a hardware



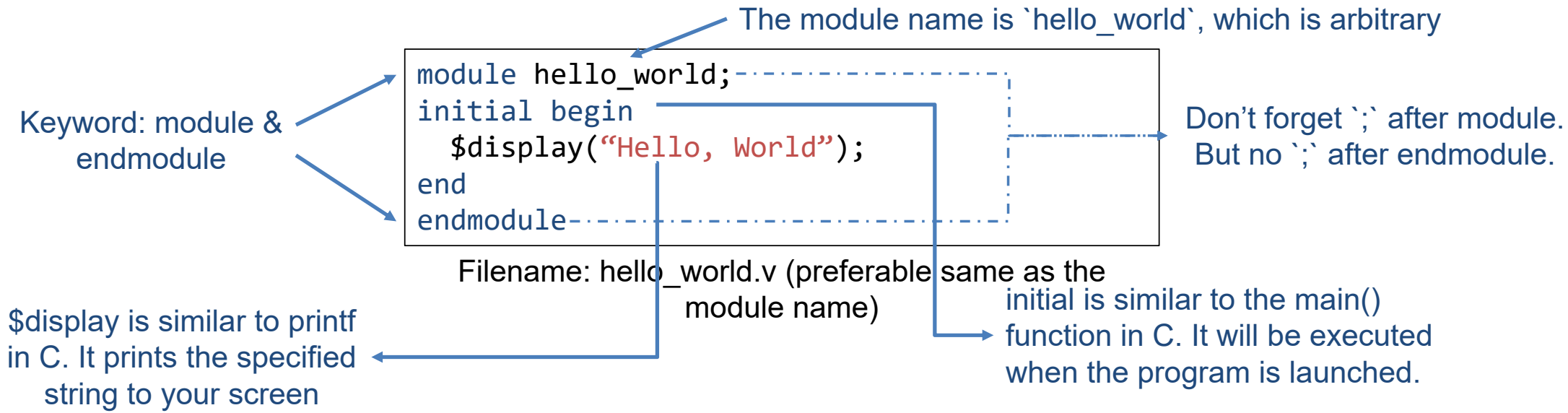
The circuit block in your mind

```
The circuit block contains 5 gates.
First AND gate connects with the primary
input A and B, and the output is connected
to a net called `TMP1`.
…
```

Role of the HDL: description of the circuit block

`Hello World` in HDL

- Verilog is described in the granularity of module
- Each module corresponds to a block in the circuit
  - The small block can be called as a module, like decoder, encoder, 16-bit adder, etc.
  - The large block can be called as a module, like CPU, GPU, DRAM controller, etc.



Verilog is similar to C

- `Hello, World` in C vs. `Hello, World` in Verilog

```c
#include <stdio.h>
int main() {
  printf("Hello, World\n");
  return 0;
}
```

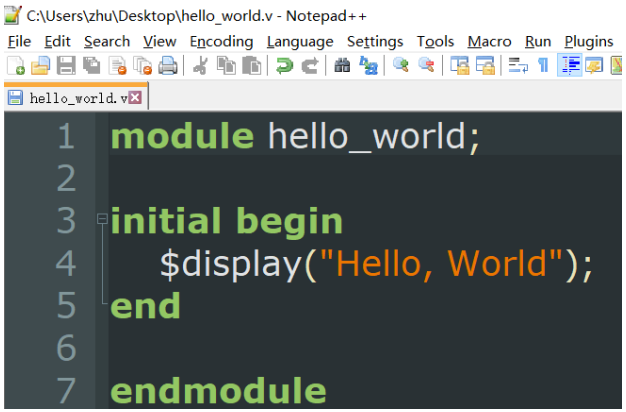
hello_world.c

begin end is the same as {}

```
module hello_world;
initial begin
  $display("Hello, World");
end
endmodule
```

hello_world.v

- Suggested editor tools
  - Notepad++: https://notepad-plus-plus.org/
  - Sublime: https://www.sublimetext.com/
  - Vim: http://www.vim.org/
  - Emacs: https://www.gnu.org/software/emacs/
- Useful feature
  - Column selection: operate over multiple rows & cols



Automatic Syntax Highlight in Notepad++

Run the simulation of Verilog

- Suggested simulation tools
  - Modelsim Student Edition (Windows & Free)
  - Synopsys VCS (Linux & Available on UST server): target platform in this tutorial
- Same as C programming: compilation + execution
  - Compile it use vcs compiler: `vcs source_files [options]`
  - The command to compile the `hello_world.v`

Shell prompt (Don't Type!)

```
>> vcs hello_world.v -o hello_world
```

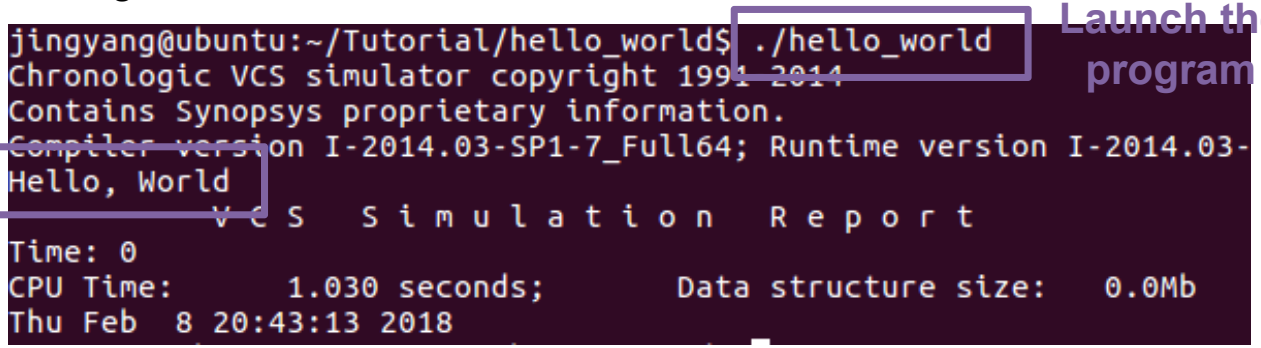
Source file

Target file name: hello_world. By default, the target file name is simv

  - Several intermediate files will be generated
    - csrc, hello_world.daidir
  - Launch the program

```
>> ./hello_world
```

Correct! 'Hello world' are printed on the console

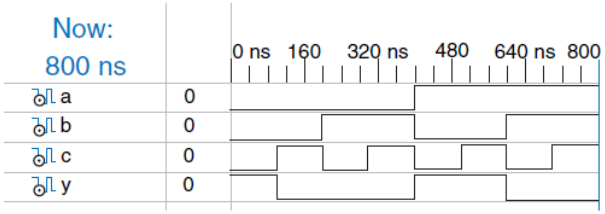


Launch the program

Simulation and Synthesis

- Verilog files can be divided into 2 purposes
  - **Simulation (a.k.a. Testbench)**
    - Generate input stimulus for a circuit block
    - Compare the output of a circuit block with the desired one

You can write any fancy code here, like function, task, while loop



Input stimulus to be generated by Testbench

Apply to inputs

Circuit Block Under Test

Output waveforms

Output waveforms are Correct ???

  - **Synthesis**
    - Describe the behavior/structure of the underlying circuit block
    - Should be **VERY VERY careful** in terms of syntax

Only the synthesizable codes can be written here

Combinational logic

- A circuit block has multiple outputs Y and multiple inputs X
  - The value Y is determined by a Boolean function: Y = f(X)
  - No loop within a combinational logic

```
module inv(
   input wire [3:0] a,
   output wire [3:0] y
);

assign y = ~a;

endmodule
```

The module with port declaration (2 ports)
- Input port a: with bit width 4
- Output port b: with bit width 4

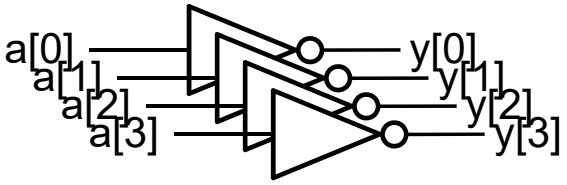
Wire is the default data type (like int, float in C) in Verilog

Inverter in Verilog

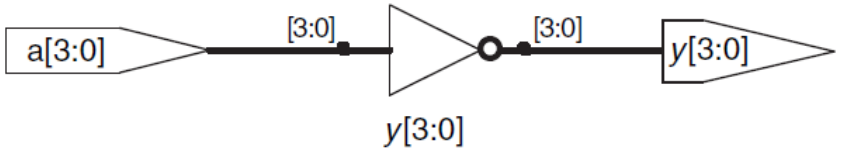
~ is the bitwise inversion

wire [3:0] a	a[3]	a[2]	a[1]	a[0]	Little-endian
wire [0:3] a	a[0]	a[1]	a[2]	a[3]	Big-endian

- Adopt little-endian convention in this tutorial
  - N-bit bus: [N-1:0]



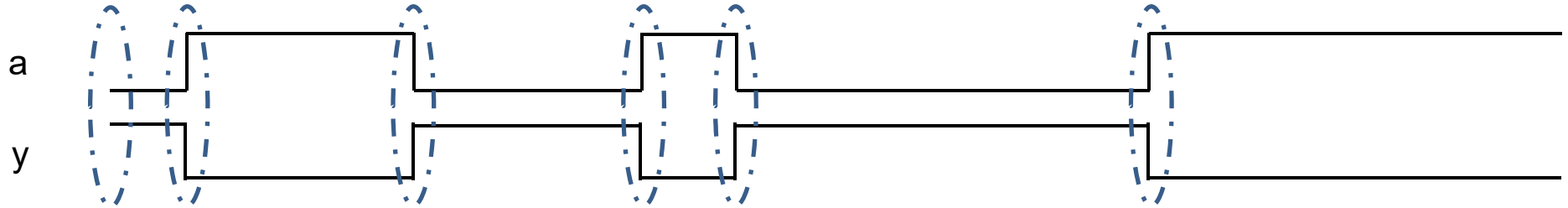
Equivalent



Implied circuit block: 4 inverters

The continuous assignment

- Keyword assign represents
  Anytime, if any input on the right hand side changes, the output on the left hand side needs to be re-evaluated




```
assign y = ~a;
```

Y re-evaluates here. That is the behavior of the combinational logic!

- Assign statement **always** represents the **combinational logic**
- The data type on the left hand side (output) should be **wire ONLY**

Supported operations in Verilog

- Verilog supports a wide range of operations (same as C)
  - You can use all of them in the combinational logic

Logical Operators	
&	bitwise AND
|	bitwise OR
^	bitwise XOR
^~	bitwise XNOR
~	bitwise NOT
&&	boolean AND
||	boolean OR
!	boolean NOT

Arithmetic Operators	
+	addition
-	subtraction

Reduction Operators	
&	reduce via AND
~&	reduce via NAND
|	reduce via OR
~|	reduce via NOR
^	reduce via XOR
^~	reduce via XNOR

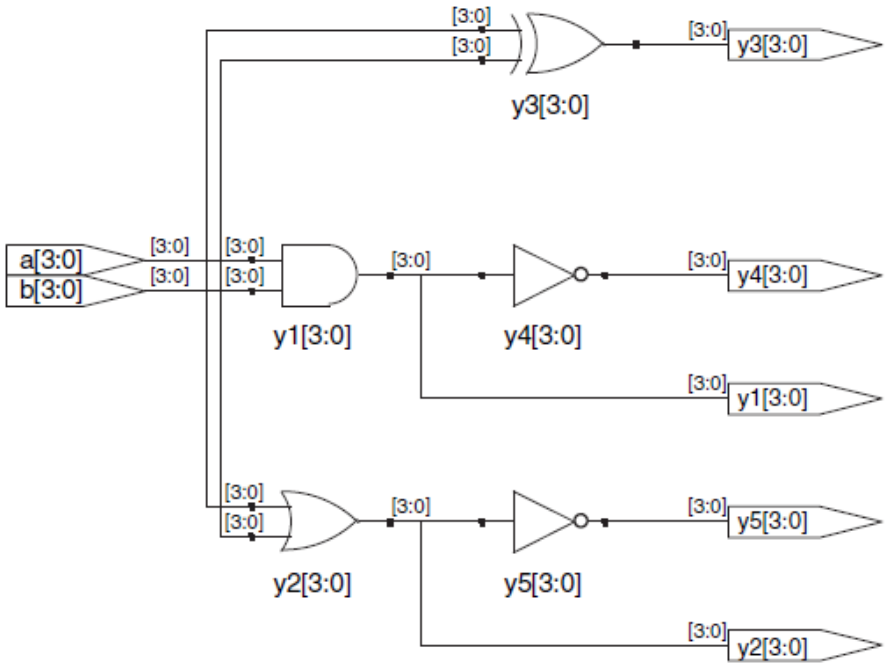
Shift Operators	
>>	shift right
<<	shift left
>>>	arithmetic shift right

Relational Operators	
==	equal
!=	not equal
>	greater than
>=	greater than or equals
<	less than
<=	less than or equals

Other Operators	
{}	concatenate
{N{}}	replicate N times

Simple example for bit-wise operations

- The module `gates` contain 5 gates
  - AND, OR, XOR, NAND, NOR



Implied module `gates`

```
module gates(
   input wire [3:0] a, b,
   output wire [3:0] y1, y2, y3, y4, y5
);
assign y1 = a & b;     // AND gate
assign y2 = a | b;     // OR gate
assign y3 = a ^ b;     // XOR gate
assign y4 = ~(a & b); // NAND gate
assign y5 = ~(a | b); // NOR gate
endmodule
```

Module `gates` in Verilog

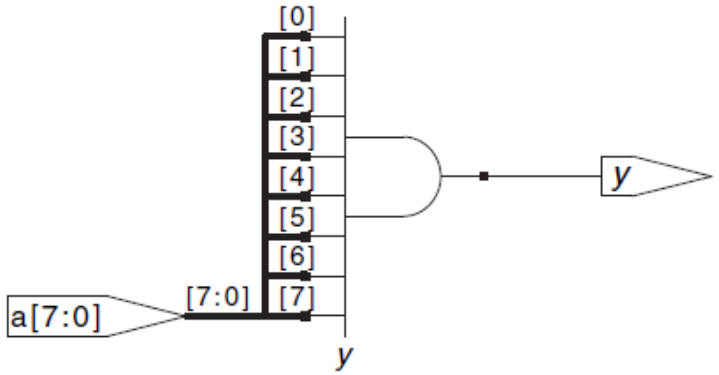
- Comments in Verilog are the same as C

```
// single-line comment
/*
 * multiple-line comment
 */
```

Simple example for reduction operations

- Reduce the input bus into a 1-bit output (it is quite useful under some circumstance)

```
module gates(
   input wire [7:0] a,
   output wire y
);

// AND reduce
assign y = &a;

// It is equivalent as follows
// assign y = a[0] & a[1] & a[2] & a[3]
//            a[4] & a[5] & a[6] & a[7];

endmodule
```



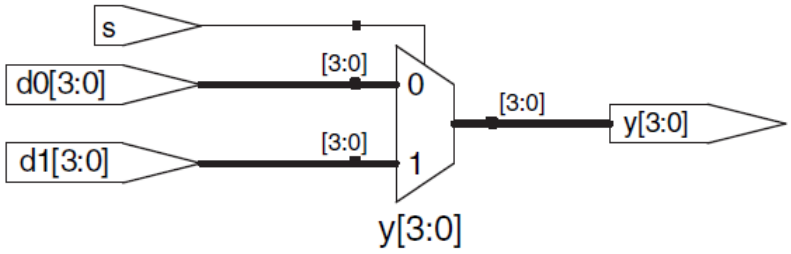
Implied circuit with large fan-in (8-input AND)

Conditional assignment

- The ternary operator (which accepts 3 inputs)
- Form of `condition ? A : B` (C programming has the same operator)
- Implies multiplexer in the digital circuit

```
module mux2(input wire [3:0] d0, d1,
            input wire       s,
            output wire [3:0] y);
assign y = s ? d1 : d0;
endmodule
```

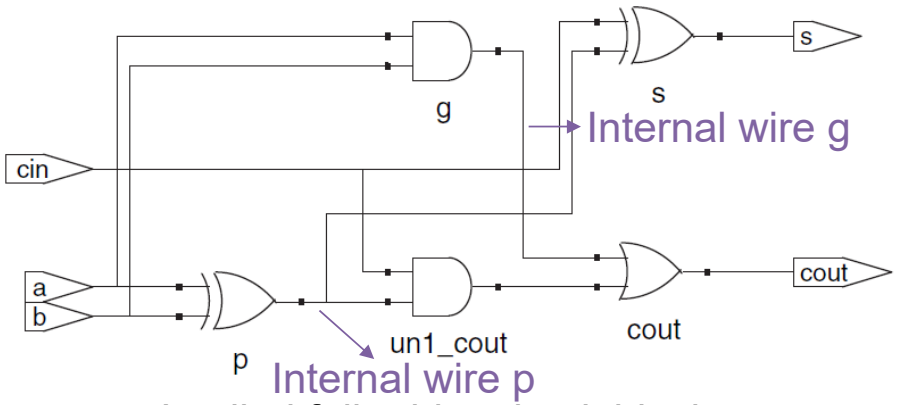
```
module mux4(input wire [3:0] d0, d1, d2, d3
            input wire [1:0] s,
            output wire [3:0] y);
// concatenate 2 conditional assignments
assign y = s[1] ? (s[0] ? d3 : d2) :
                  (s[0] ? d1 : d0);
endmodule
```



Implied Mux2 circuits

More complex circuit block: internal variables

- Sometimes, the output of circuit can not be simply represented by the primary inputs. Some intermediate variables will be introduced
- Recall: full adder with 3 inputs A, B, Cin; generates 2 outputs S, Cout
  - Intermediate signals
    - Generate G: G = A and B
    - Propagate P: P = A xor B
  - Primary outputs
    - Sum S: S = P xor Cin
    - Carry out Cout: Cout = (P and Cin) or G



Implied full adder circuit block

```
module full_adder(input wire a, b, cin,
                  output wire s, cout);
wire p, g; // internal signals
assign p = a ^ b;
assign g = a & b;
assign s = p ^ cin;
assign cout = g | (p & cin);
endmodule
```

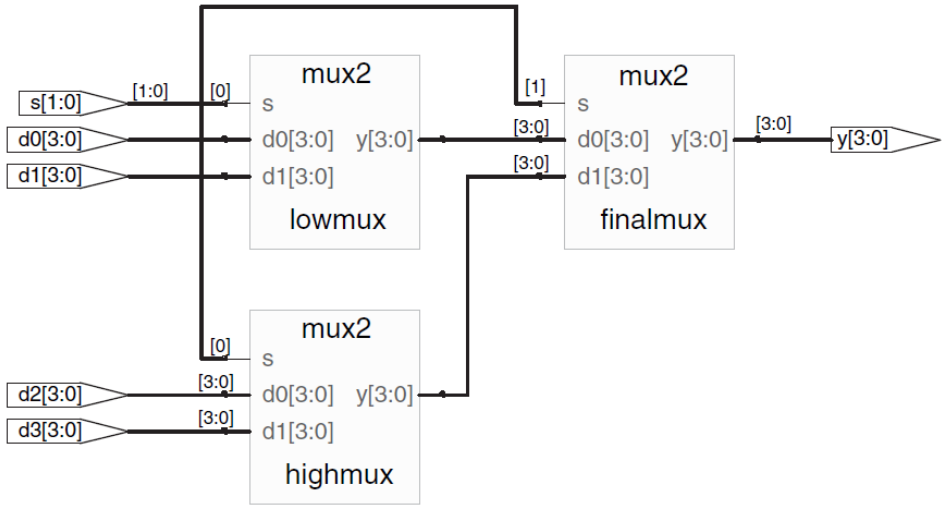
Simply define the internal wires within the module

Structural modeling

- Hierarchical design of a complicated module
- Examples: build 4-to-1 MUX by 2-to-1 MUXes

```
// suppose we have a module `mux2`
// defined in mux2.v
module mux4(input wire [3:0] d0, d1, d2, d3,
            input wire [1:0] s,
            output wire [3:0] y);
wire [3:0] low, high; // internal wires
mux2 low_mux(d0, d1, s[0], low);
mux2 high_mux(d2, d3, s[0], high);
mux2 final_mux(low, high, s[1], y);
endmodule
```

Instantiate 3 2-to-1 MUXes here, here we bind the wires with ports by the order (not preferred)



Hierarchical design of a 4-to-1 MUX

Coding style for port binding in instantiation

- Binding by order: short but unclear and error-prone

```
// `mux2` is defined as (input a, input b, input sel, output y)
mux2 low_mux(d0, d1, s[0], low); // d0 is connected to first port of mux2
                                 // i.e. input a
```

- Binding by name: long but clear

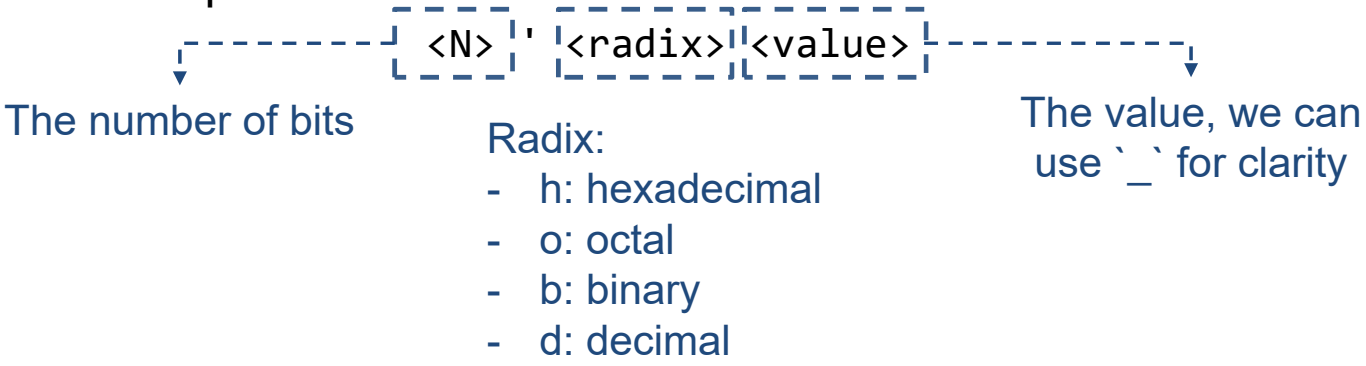
```
mux2 low_mux(.a(d0), .b(d1), .sel(s[0]), .y(low)); // .port_name(wire_name)
```

- Binding by name with comments (preferred): meaningful

```
mux2 low_mux(
    .a    (d0),     // 1st input of 2-to-1 MUX
    .b    (d1),     // 2nd input of 2-to-1 MUX
    .sel  (s[0]),   // select of 2-to-1 MUX
    .y    (low)     // output of 2-to-1 MUX
);
```

Number representation

- Number can be represented as follows:



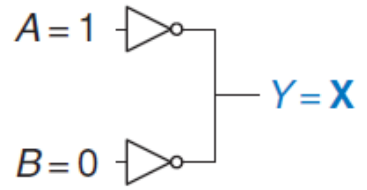
- Examples
  - 3'b101: 3 bit binary number 101 (decimal 5)
  - 8'b11: 8 bit binary number 00000011 (Verilog zero-extends to the specified width)
  - 8'b1110_0110: 8 bit binary number, here `_` for clarity
  - 6'o42: 6 bit octal number (decimal 34)
  - 8'hAF: 8 bit hex number

Four values in Verilog

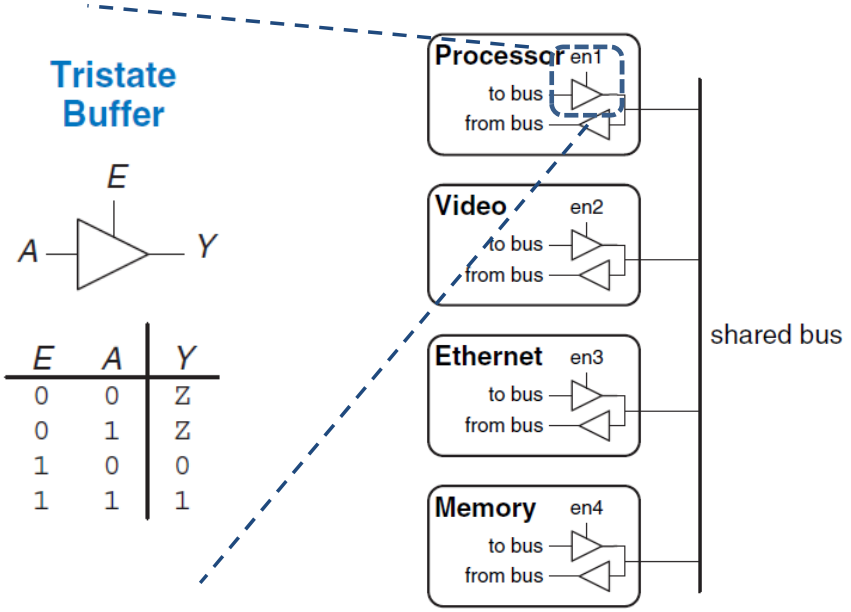
- Verilog supports 4 values
  - 1, 0: logic high (VDD) and logic low (GND)
  - X: unknown value or conflict
  - Z: high impedance
- X: usually implies problems in the circuit
  - The flip-flop is not reset properly
- Z: floating node
  - Tristate buffer for IO interface

```
// simple tristate buffer
assign Y = E ? A : 1'bz;
```

Verilog code for a tristate buffer



Contention @ wire Y
will cause X value

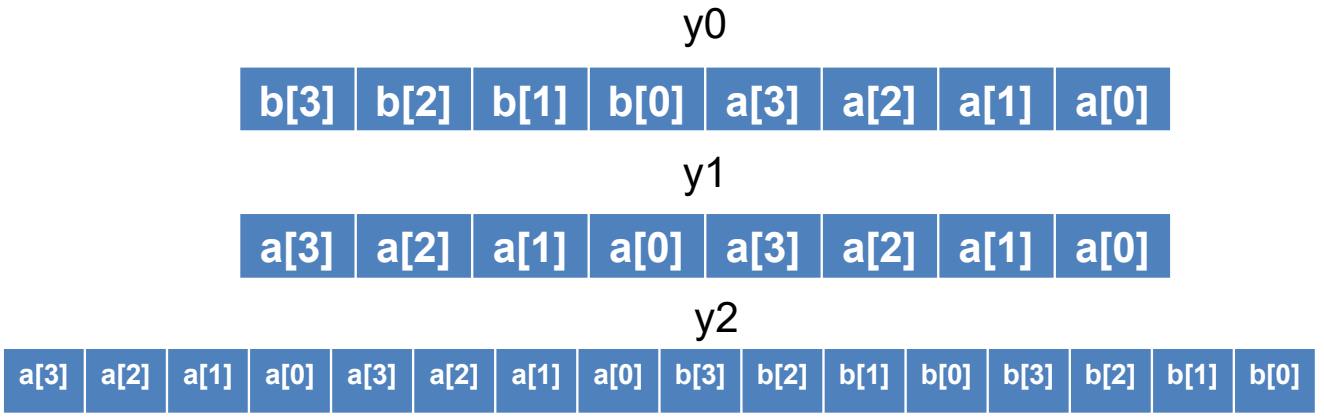


Typical case for high impedance Z

Simple example for bus concatenation

- Concatenate (merge) several signals/buses into a single bus

```
wire [3:0] a, b;
wire [7:0] y0, y1;
wire [15:0] y2;
// concat: {N{variable}}
// N is the repeat times
assign y0 = {b, a};
assign y1 = {2{a}};
assign y2 = {{2{a}}, {2{b}}};
```

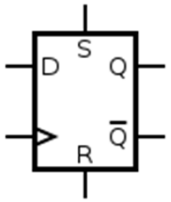


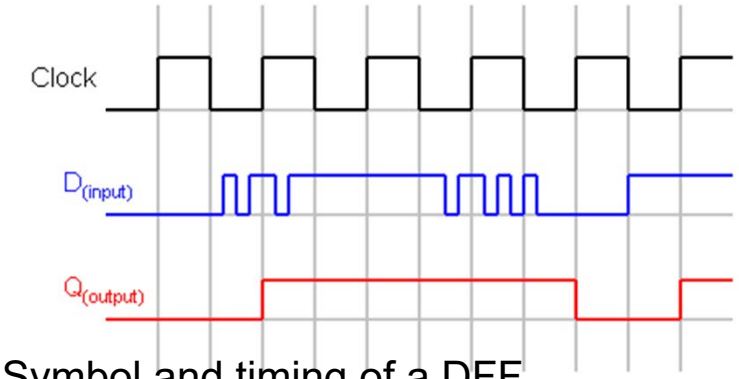
- Commonly used as
  - Initialization: {8{1'b0}} (8-bit initialized to full 0s)
  - Shifting

```
wire [3:0] a;
wire [3:0] y;
assign y = {a[1:0], 2'b00}; // y is a << 2
```

Sequential logic

- The output is not only determined by current inputs but the previous inputs (like human memory)
- Typical sequential logic
  - D flip-flop (DFF)
  - D latch
  - Register





Symbol and timing of a DFF

```
module DFF(input wire clk,
           input wire [3:0] d,
           output reg [3:0] q);
always @(posedge clk) begin
  q <= d;
end
endmodule
```

4-bit DFF in Verilog

All sequential logic has a clock input

reg: a special data type to implement sequential logic, **but it can also be used to represent combinational logic**!

`<=` is called nonblocking assignment. It can be simply regarded as `=` in the sequential logic

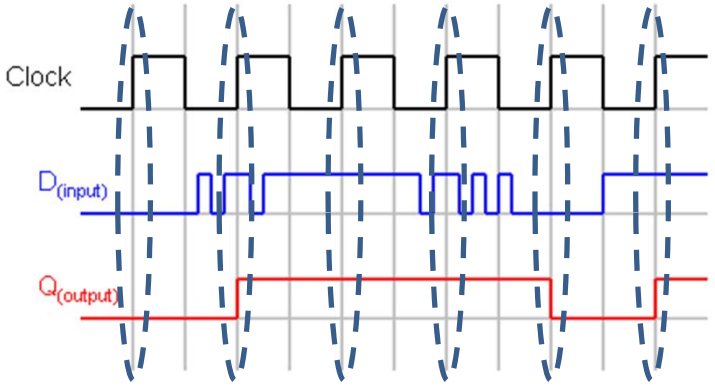
Always statement

- Always statement is used for implementing **combinational logic** and **sequential logic**
- The data type on left hand side (LHS) in always statement MUST be **reg**

```
always @(sensitive list) begin
  statement one; // LHS should be of
  statement two; // type reg
  …
end
```

- The statements within always block will be re-evaluated only when the event in sensitive list occurs
- DFF: output q samples the input d **when the clock is at the rising edge**

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



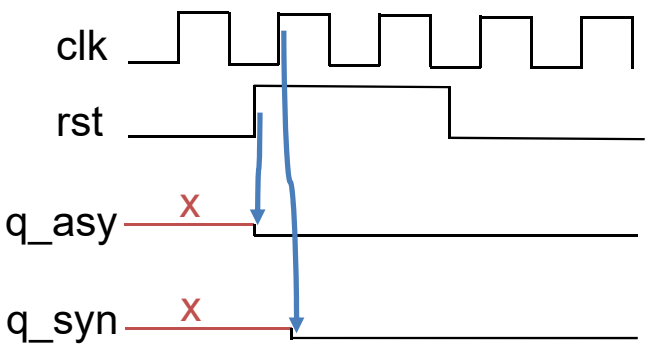
The value of q needs to be re-evaluated here

Various types of DFF

- DFF with reset (preferred, otherwise the value is **x** initially)

```
module dff_sync(
  input wire clk,
  input wire rst,
  input wire [3:0] d,
  output reg [3:0] q
);
always @(posedge clk) begin
  if (rst) begin
    q <= 1'b0;
  end else begin
    q <= d;
  end
end
endmodule
```

```
module dff_async(
  input wire clk,
  input wire rst,
  input wire [3:0] d,
  output reg [3:0] q
);
always @(posedge clk or
  posedge rst) begin
  if (rst) begin
    q <= 1'b0;
  end else begin
    q <= d;
  end
end
endmodule
```



The first q is asynchronous DFF
The second q is synchronous DFF

The reset in the sensitive list will cause the q re-evaluates as soon as reset rises, that is the definition of asynchronous DFF!

Synchronous DFF vs. Asynchronous DFF

DFF with enable and reset

- The enable signal for DFF is **always synchronous**

```
module dff_sync(
  input wire clk,
  input wire rst,
  input wire en,
  input wire [3:0] d,
  output reg [3:0] q
);
always @(posedge clk) begin
  if (rst) begin
    q <= 1'b0;
  end else if (en) begin
    q <= d;
  end
end
endmodule
```

```
module dff_async(
  input wire clk,
  input wire rst,
  input wire en,
  input wire [3:0] d,
  output reg [3:0] q
);
always @(posedge clk or
    posedge rst) begin
  if (rst) begin
    q <= 1'b0;
  end else if (en) begin
    q <= d;
  end
end
endmodule
```

Common Pitfall: Abuse of sensitive list

```
always@ (posedge
clk, negedge clk)
```

```
always@ (posedge
clk, negedge en)
```

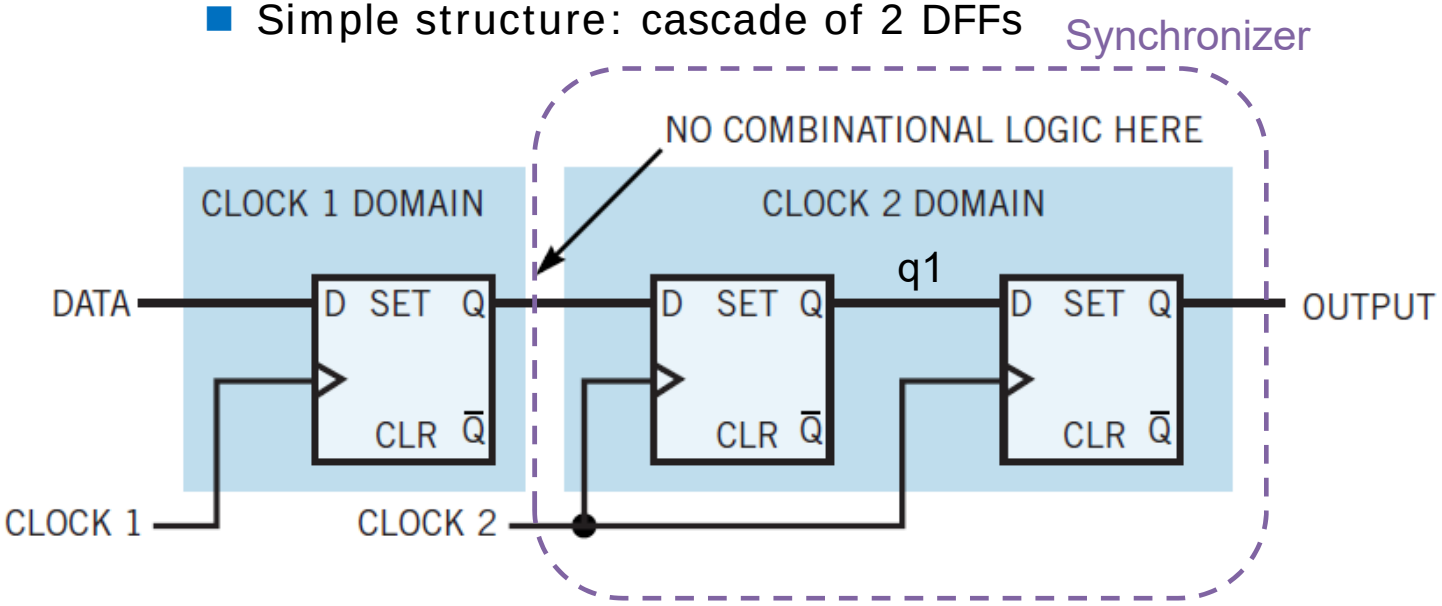
```
always@ (posedge
clk, posedge en)
```

```
always@ (posedge
clk, posedge rst,
posedge en)
```

Rule of thumb: does the circuit really exist?

Synchronizer

- Common circuit block for crossing the clock domain
  - Simple structure: cascade of 2 DFFs

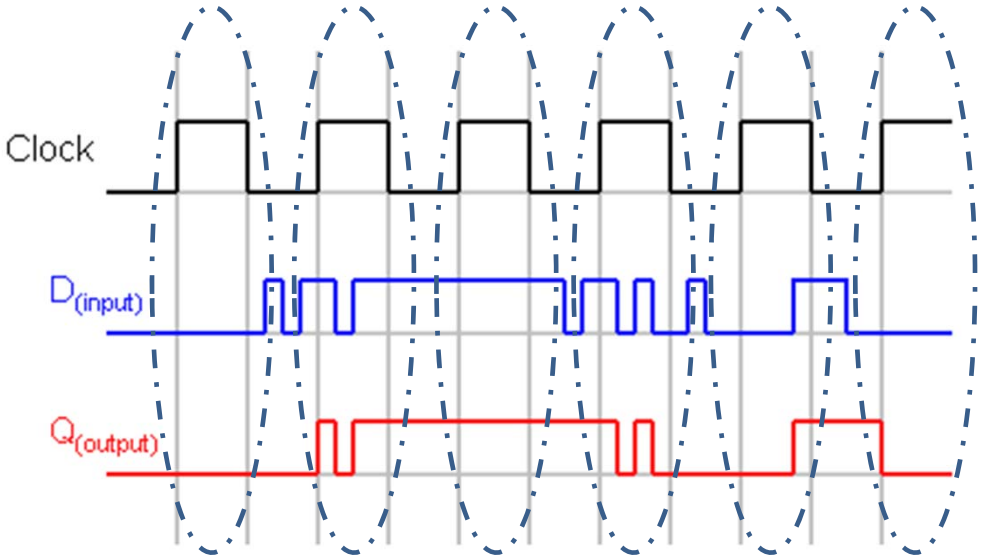



```
module synchronizer(
  input wire clk,
  input wire d,
  output reg q
);
reg q1;//must be of type reg
       // because q1 occurs
       // @ LHS in the
       // always block
always @(posedge clk) begin
  q1 <= d; // q1 @ LHS
  q  <= q1;
end
endmodule
```

D-Latch

- WARNING: rare in the real design except for the clock gating
- If the synthesized circuit has the latch, it usually implies some problem!!!
  - Most of time, **the code is wrong**, which leads to **the unintended latch**!

```
module d_latch(
  input wire clk,
  input wire [3:0] d,
  output reg [3:0] q
);
always @(clk, d) begin
  if (clk) begin
    q <= d;
  end
end
endmodule
```



q needs be re-evaluated when clk and d change.
The if statement ensures that q copies the value of d during clk is high (transparent when clk = 1)

Combinational logic using always and reg

- The combinational logic can be described by the continuous assignment
  - It is very tedious to write **assign**: try to implement a 16-to-1 MUX using assign
- The combinational logic can be described with always block
  - Left hand side in always block should be of **reg**
  - We can use the **case, if, else** statement to describe combinational logic
  - Be careful, otherwise creates unintended latches, multiple drivers

```
module mux4(input wire [3:0] d0, d1, d2, d3
            input wire [1:0] s,
            output wire [3:0] y);
// use continuous assignment
assign y = s[1] ? (s[0] ? d3 : d2) :
                  (s[0] ? d1 : d0);
endmodule
```

Different data types

```
module mux4(input wire [3:0] d0, d1, d2, d3
            input wire [1:0] s,
            output reg [3:0] y);
always@ (d0, d1, d2, d3, s) begin
  if (s == 2'b00) y = d0;
  else if (s == 2'b01) y = d1;
  else if (s == 2'b10) y = d2;
  else y = d3;
end
endmodule
```

Comments on combinational logic use always

- It is convenient to use always instead of assign to implement **a complicated combinational logic**
- If … else or case MUST be within the always block
- Sensitive list of always block MUST be **complete** for the combinational logic

```
module mux4(input wire [3:0] d0, d1, d2, d3
            input wire [1:0] s,
            output reg [3:0] y);
always@ (d0, d1, d2, d3, s) begin
  if (s == 2'b00) y = d0;
  else if (s == 2'b01) y = d1;
  else if (s == 2'b10) y = d2;
  else y = d3;
end
```

For combinational logic, the output depends on the current value of inputs. Therefore, when any of inputs is changed, the output should be re-evaluated.
We must **put all inputs to the sensitive list**.
In Verilog, we can use the shortcut:
always@ (*) to automatically include all inputs!

The assignment to output y should be **complete**! Common pitfall like no else for if statement:
if …. else if …. ~~else~~ … no else clause here
which will cause **unintended latch!!!** Because output will keep its value under some condition, i.e. Latch

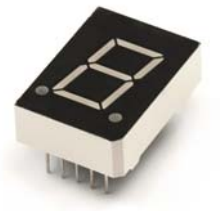
Case study of combinational logic using always

- Use always@(*)
- Case: contain **default** clause; if: contain **else** clause

```
module seven_segment(input wire [3:0] data,
                     output reg [6:0] segments);
always@(*) begin
  case(data)
    4'd0: segments = 7'b111_1110;
    4'd1: segments = 7'b011_0000;
    // omitted cases of 4'd2 ~ 4'd9…
    default: segments = 7'b000_0000;
  endcase
end
endmodule
```

Important to have default in the case statement, otherwise unintended latch!

7-segment decoder



```
module priority(input wire [3:0] a,
                        output reg [3:0] y);
always@(*) begin
  if(a[3])       y = 4'b1000;
  else if (a[2]) y = 4'b0100;
  else if (a[1]) y = 4'b0010;
  else if (a[0]) y = 4'b0001;
  else           y = 4'b0000; // no 1s
end
endmodule
```

Priority encoder

Blocking assignment vs. non-blocking assignment

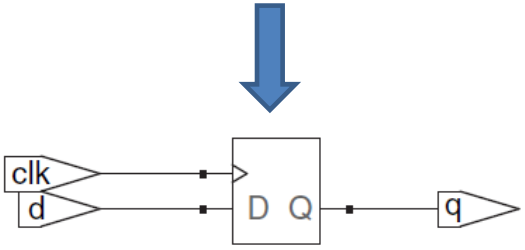
- In always block, there exists 2 types of assignments
  - Blocking (=): happens sequentially
  - Non-blocking (<=): happens concurrently

```
// synchronizer
always @(posedge clk) begin
  q1 <= d;
  q  <= q1;
end
```

≠

```
// synchronizer: use blocking
always @(posedge clk) begin
  q1 = d;
  q  = q1;
end
```

Non-blocking assignment happens concurrently, where q1 get the original value of d (before clk edge). q get the **original** value of q1 (before clk edge). Therefore, it will generate 2 cascaded DFFs.
Blocking assignment happens sequentially, where q1 get the original value of d and get **updated**! q get q1, but q1 has been updated to the value of d. Therefore, only 1 DFF is inferred.



q

After synthesizing, it will generate a single DFF instead of 2 cascaded DFFs

Recommended coding styles

- Always use nonblocking assignment (<=) for sequential logic

```
// sequential logic
always @(posedge clk) begin
  q <= in; // non-blocking
end
```

- Use continuous assignment for simple combinational logic

```
// simple combinational logic: like 2-to-1 MUX
assign y = s ? d0 : d1;
```

- Use always@(*) and blocking assignment (=) for complex combinational logic

```
// complex combinational logic
always@(*) begin
  // if … else if … else if … else
  // or case-endcase with default clause
end
```

```
always@(*) y = a & b;
// later on, y is re-assigned
always@(*) y = c;
```

Error: multiple-driver conflict

- Do NOT make assignment to one signal in multiple always blocks or continuous assignments

Advanced technique: parameterized modules

- The port width is fixed up to now, but it can be parameterized for a better reusability

```
module mux2 #(parameter BIT_WIDTH=4)
(
  input  wire [BIT_WIDTH-1:0] d0, d1,
  input  wire                 s,
  output wire [BIT_WIDTH-1:0] y
);
assign y = s ? d1 : d0;
endmodule
```

2-to-1 MUX with parameterized bit width

Has a parameter `BIT_WIDTH` with the default value of 4

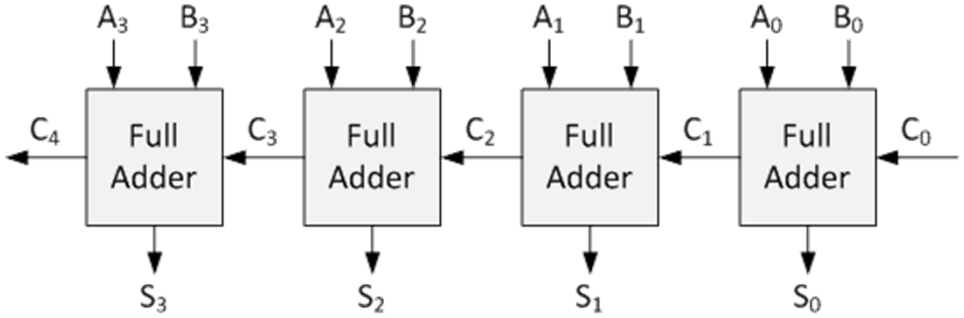
```
// instantiate parameterized module
// with default bit width = 4
mux2 mux_0(.d0(d0),.d1(d1),.s(s),.y(y));
```

```
// instantiate parameterized module
// with bit width = 8
mux2 #(.BIT_WIDTH(8))
mux_0 (
  .d0  (d0),  // d0 is 8-bit
  .d1  (d1),  // d1 is 8-bit
  .s   (s),   // select is always 1-bit
  .y   (y)    // y is 8-bit
);
```

Advanced technique: generate statements

- Generate statement is useful to build repeated modules (generate & for)

```
module ripple_carry_adder #(parameter BIT_WIDTH=4)
(
  input  wire [BIT_WIDTH-1:0] A, B,
  input  wire                 Cin,
  output wire [BIT_WIDTH-1:0] S,
  output wire                 Cout
);
genvar i; // iterator of generate loop, NOT integer
wire [BIT_WIDTH:0] C; // internal carry chain
generate
for(i = 0; i < BITWIDTH; i = i + 1) begin:gen_chain
  full_adder FA(.A(A[i]),.B(B[i]),.Cin(C[i]),
                .S(S[i],.Cout(C[i+1]));
end
endgenerate
assign Cout = C[BIT_WIDTH];
assign C[0] = Cin;
endmodule
```



Ripple carry adder: consisting cascaded full adders

The for loop will be automatically unrolled to generate multiple full adders here:

```
full_adder gen_chain[0].FA(.A(A[0]), …);
full_adder gen_chain[1].FA(.A(A[1]), …);
full_adder gen_chain[2].FA(.A(A[2]), …);
full_adder gen_chain[3].FA(.A(A[3]), …);
```

The name of generate block must be defined for instantiation

Advanced technique: generate statements (cont.)

- Generate statement is useful to generate different types of instantiation (generate & if)

```
module multiplier #(parameter BIT_WIDTH=4)
(
  input  wire [BIT_WIDTH-1:0]   A, B,
  output wire [2*BIT_WIDTH-1:0] Y
);
generate
if(BIT_WIDTH < 8) begin: gen_small_bitwidth
  cla_multiplier #(.BIT_WIDTH(BIT_WIDTH)) mult(
    .A(A), .B(B), .Y(Y));
end else begin: gen_large_bitwidth
  wallace_multiplier #(.BIT_WIDTH(BIT_WIDTH)) mult(
    .A(A), .B(B), .Y(Y));
end
endgenerate
endmodule
```

Generate Carry-lookahead multiplier for small bit-width & Wallace Tree multiplier for large bit-width

Testbench for the combinational logic

- Testbench is used for verifying the functionality of the circuit
- The limitation for the synthesize consideration (free to use while, wait, for)

```
`timescale 1ns/1ps # time unit/precision unit
module testbench;
// declare the IO of Device Under Test (DUT)
reg [3:0] d0, d1;
reg s;
wire [3:0] y;
// Instantiation of DUT
mux2 DUT(.d0(d0), .d1(d1), .s(s), .y(y));
// Apply the input stimulus
initial begin
  d0 = 4'd1; d1 = 4'd10; s = 1'b0;
  #10; d0 = 4'd4; d1 = 4'd8; s = 1'b1;
  #10; d0 = 4'd3; d1 = 4'd10; s = 1'b1;
end
endmodule
```

Declare IO connections in the beginning. Typically, inputs are of type reg; and outputs are of type wire. Because the LHS in initial block should be reg, and the output connections of instances should be wire.

The input stimulus is applied within the initial block, where it is executed from time = 0. # is different from parameterized module, it indicates delay time. E.g. #10 means delay 10 time unit (ns in this case), and the apply the next stimulus.

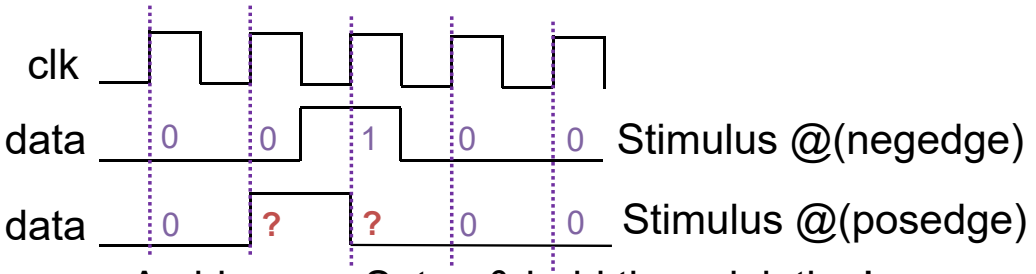
Testbench for the sequential logic

- Clock and reset generation
- Apply the input stimulus at **negedge / posedge clock** (stable during the **posedge / negedge clock**)

```
localparam CLK_PERIOD = 1.0; # 1ns clk (1GHz)
reg clk, rst, s;
reg [3:0] a, b;
wire [3:0] y;
// Device Under Test
mux_with_reg DUT(clk, rst, a, b, s, y);
// clock generation
initial begin
  clk = 1'b0;
  forever #(CLK_PERIOD/2.0) clk = ~clk;
end
// reset generation
initial begin
  rst = 1'b1;
  #10 rst = 1'b0;
end
```

```
initial begin
  wait(rst == 1'b0); // block until reset = 0
  @(negedge clk);    // apply stimulus @ negedge
  a = 4'd10; b = 4'd1; s = 1'b0;
  @(negedge clk);    // apply stimulus @ negedge
  a = 4'd7; b = 4'd1; s = 1'b1;
end
```

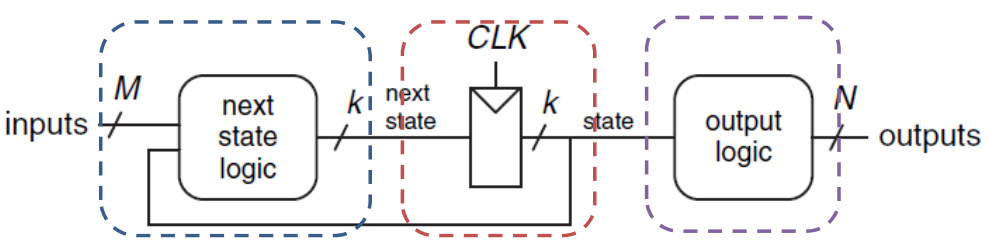
All initial blocks start concurrently from time = 0



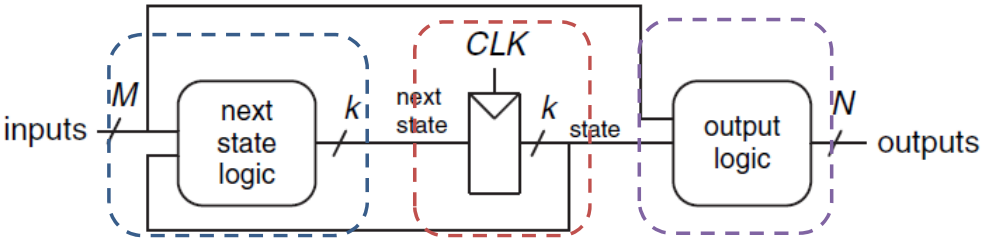
Ambiguous: Setup & hold time violation!

Case study: finite state machine (FSM)

- FSM is usually adopted for the controller, e.g. traffic light controller, vending machine controller, sequence detector, CPU controller etc.
- Two types of FSM
  - Moore machine: output signals only depend on the internal states
    - Pro: robust in the timing; con: more states
  - Mealy machine: output signals are determined by the internal states and inputs
    - Pro: less states; con: less robust (output is asynchronous w.r.t. clock)



Moore machine

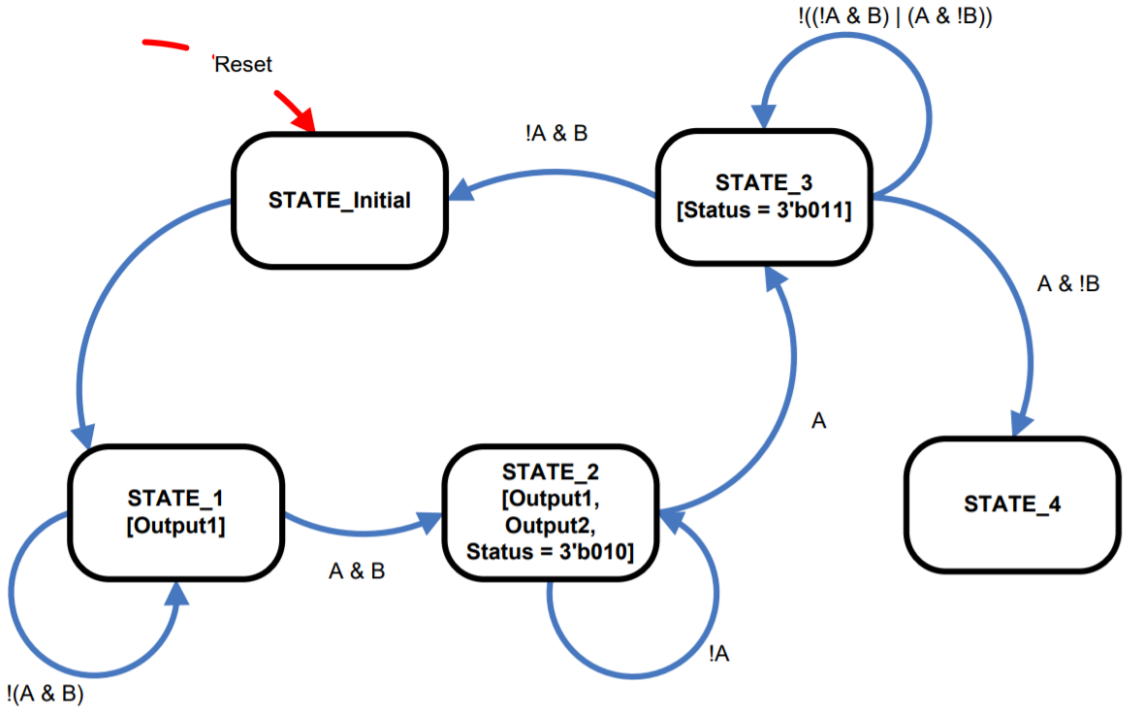


Mealy machine

  - Next state logic: combinational logic; always@(*) or assign
  - State registers: sequential logic; always@ (posedge/negedge clk)
  - Output logic: combinational logic; always@(*) or assign

Case study: FSM – 4 components

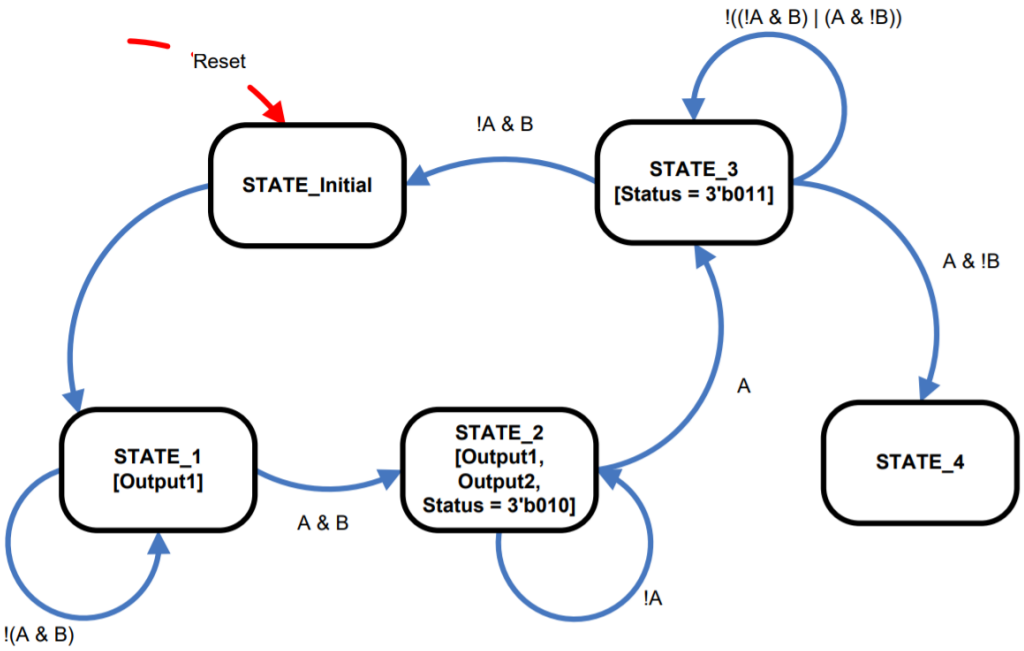
- Key step: derive the state transfer diagram

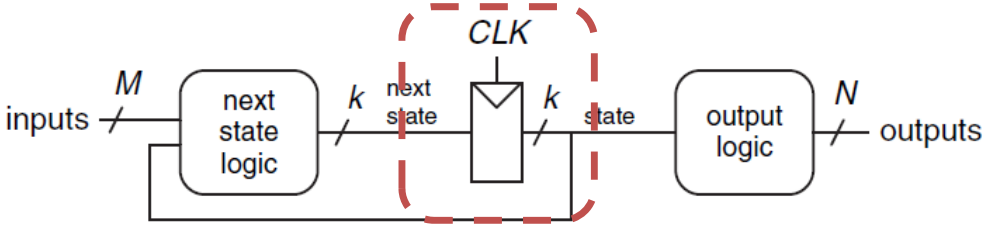


The state transfer diagram of a Moore Machine

```
module simple_fsm(
  input wire clk, rst, // system clock & reset
  input wire A, B,     // 2 input signals
  output wire Output1, // 3 outputs:
  output wire Output2, // two 1-bit & one 3-bit
  output reg [2:0] Status
);
// Part I: state encoding

// Internal variables, such as state register

// Part II: state register (sequential)

// Part III: next state logic (combinational)

// Part IV: output logic (combinational)

endmodule
```

Case study: FSM – state registers

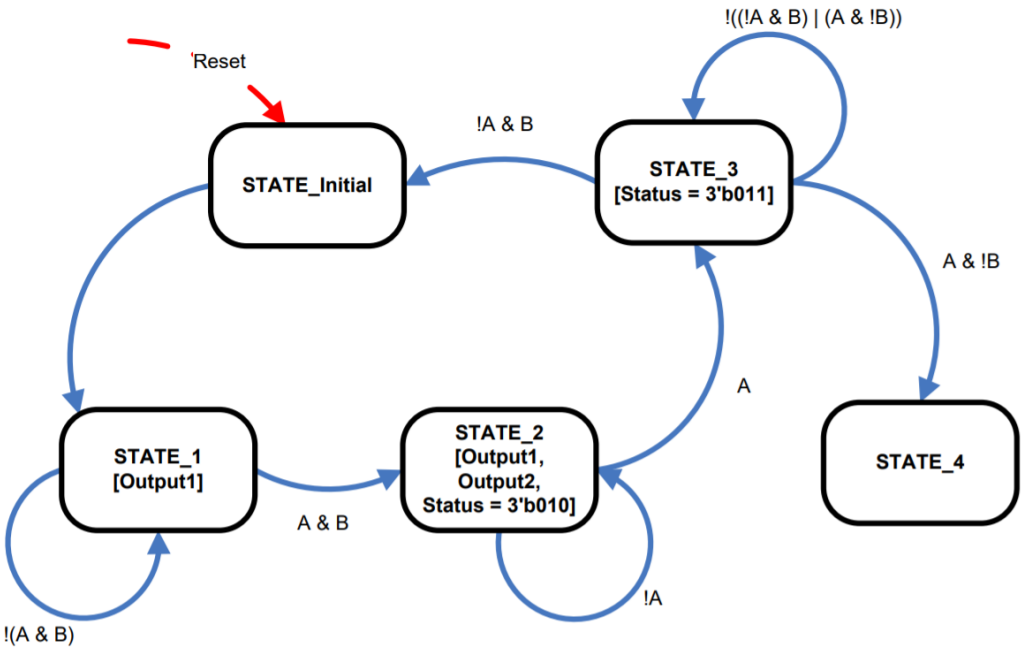


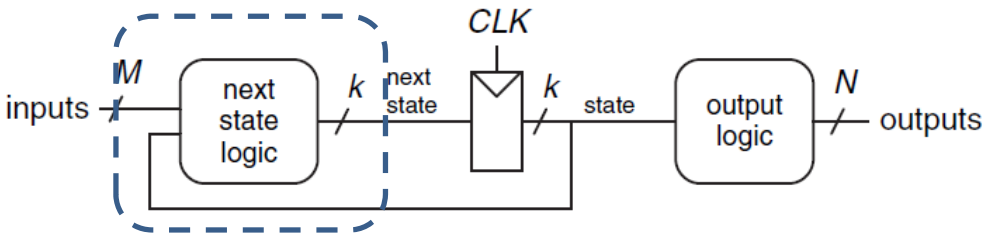



```
// Part I: state encoding
// localparam is like define MACRO in C
localparam STATE_Initial = 3'd0, // use meaningful
           STATE_1       = 3'd1, // name, so it'll
           STATE_2       = 3'd2, // easy to under-
           STATE_3       = 3'd3, // stand
           STATE_4       = 3'd4;
```

```
// Internal variables (state register)
reg [2:0] current_state,          // Q port of DFF
          next_state;             // D port of DFF

// Part II: state register (sequential)
// Synchronous DFF
always @(posedge clk) begin
  if(rst) current_state <= STATE_Initial;
  else    current_state <= next_state;
end
```

Case study: FSM – next state logic

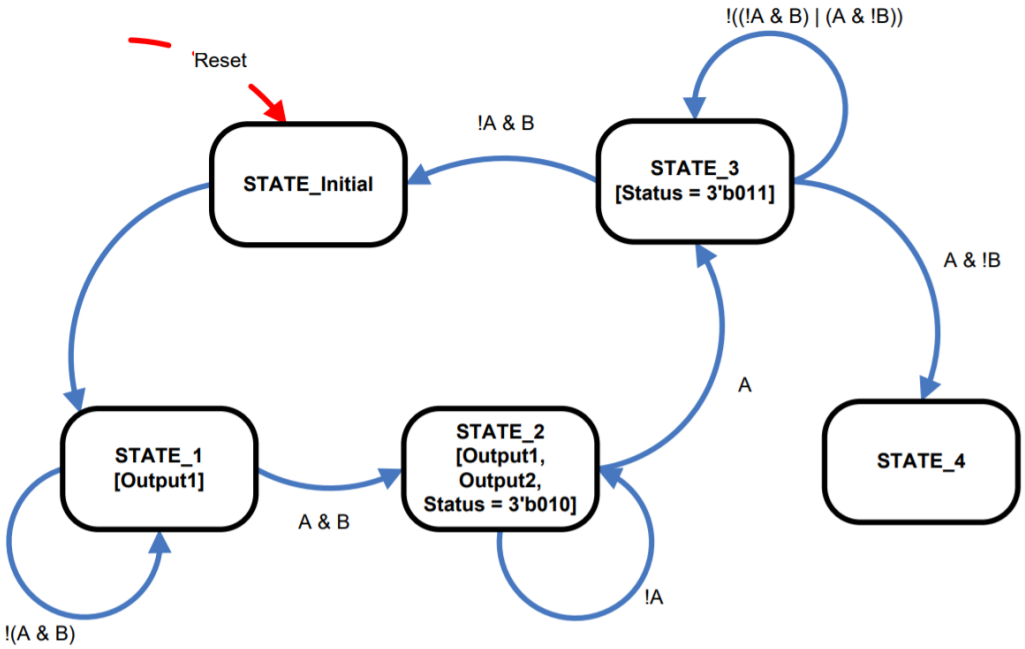


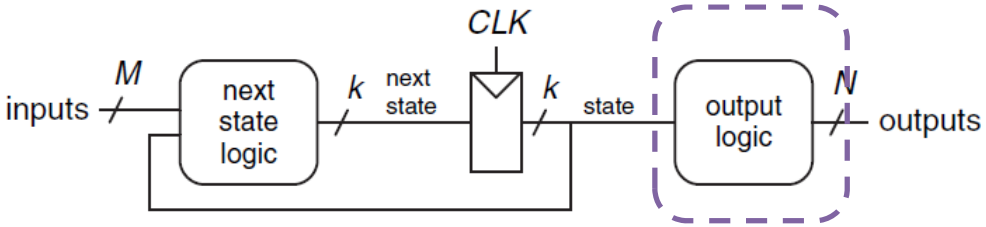



```
// Part III: next state logic (combinational)
always@(*) begin
  // Trick: set the default values of all outputs
  // at the beginning of always block to avoid
  // the problem of unintended latches and
  // simplify the following case codes
  next_state = current_state;
  case(current_state)
    STATE_Initial: next_state = STATE_1;
    STATE_1: if(A & B) next_state = STATE_2;
    STATE_2: if(A) next_state = STATE_3;
    STATE_3: begin
      if(!A & B) next_state = STATE_Initial;
      else if(A & !B) next_state = STATE_4;
    end
  endcase
end
```

The sequential execution of the blocking assignment(=) can help us simplify the code here!!! We do NOT need to worry about the completeness of case & if!!!

Case study: FSM – output logic






```
// Part IV: output logic (combinational)
// 1-bit outputs, Output1 & 2 must be of type wire
assign Output1 = (current_state = STATE_1) |
                 (current_state = STATE_2);
assign Output2 = (current_state == STATE_2);
// 3-bit output, Status must be of type reg
always@(*) begin
  case(current_state)
    STATE_2: Status = 3'b010;
    STATE_3: Status = 3'b011;
    default: Status = 3'b000; // Must have default
                              // otherwise unintended
                              // latches!
  endcase
end
```

Alternatively, we can move it here

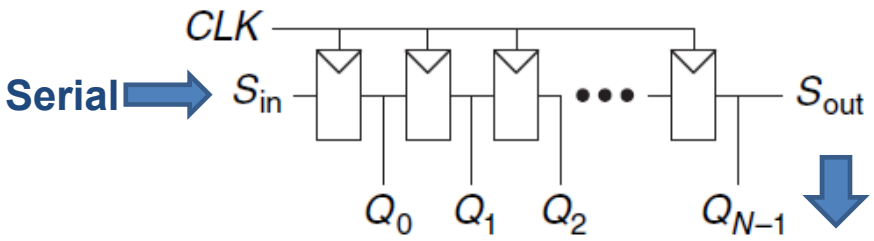
The difference between Moore Machine and Mealy Machine is in the output logic part.

Case study: shift registers

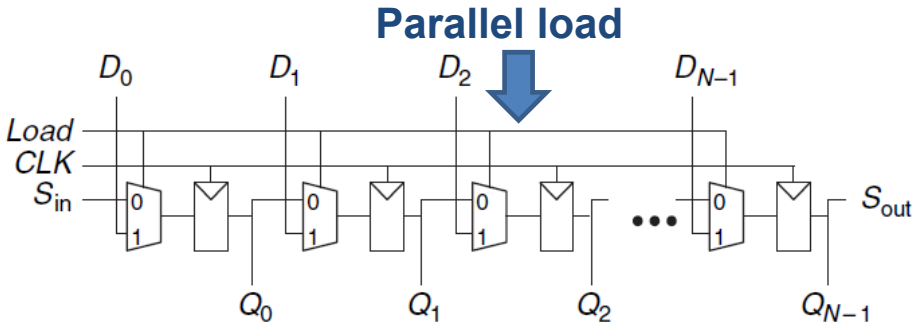
- IO interface
  - Serial in parallel output (SIPO)
  - Parallel in serial output (PISO)

```
// Example of serial in parallel output
module SIPO #(parameter N = 4)
(input wire           clk,  // system clock
 input wire           rst,  // system reset
 input wire           Sin,  // serial input port
 output wire [N-1:0] Q);    // parallel output port
// synchronous DFF
always@(posedge clk) begin
  if(rst) Q <= {N{1'b0}};  // reset N-bit 0s
  else    Q <= {Q[N-2:0], Sin}
end
endmodule
```

Use the trick of concatenation



Schematic of SIPO



Schematic of SIPO with parallel load

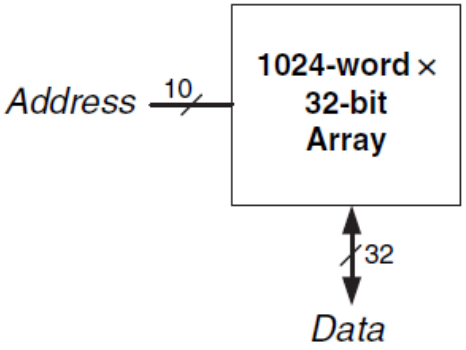
- Exercise: modify the code of SIPO to SIPO_PLOAD (SIPO with parallel load)

Case study: memory

- Memory types

Memory types	# transistors per bit cell	Latency	Synthesizable
Flip-flop	~20	Fast	Yes
SRAM	6	Medium	No, special IP
DRAM	1	Slow	No, special IP of DRAM controller

- Memory is organized into an array
  - Width and Depth: determine the size of memory
  - Multi-port memory: # of read/write accesses can happen
    - Single port: 1 read & write port
    - Dual port: 2 read & write ports



Depth = 1024; **width** = 32;
size = 1024x32 = 32kb

Case study: memory – behavior model

- **WANRNING**: we should **NEVER** synthesize our behavior model of memory
  - Only DFFs are supported in the standard cell, there is no 6T memory cell, sense amplifier, precharger, etc. (memory is more like an analog circuit)
  - Write port: synchronous with clock with write enable
  - Read port: synchronous with clock with read enable

For a memory with N-bit address, the depth is 2 to the power of N

write

read

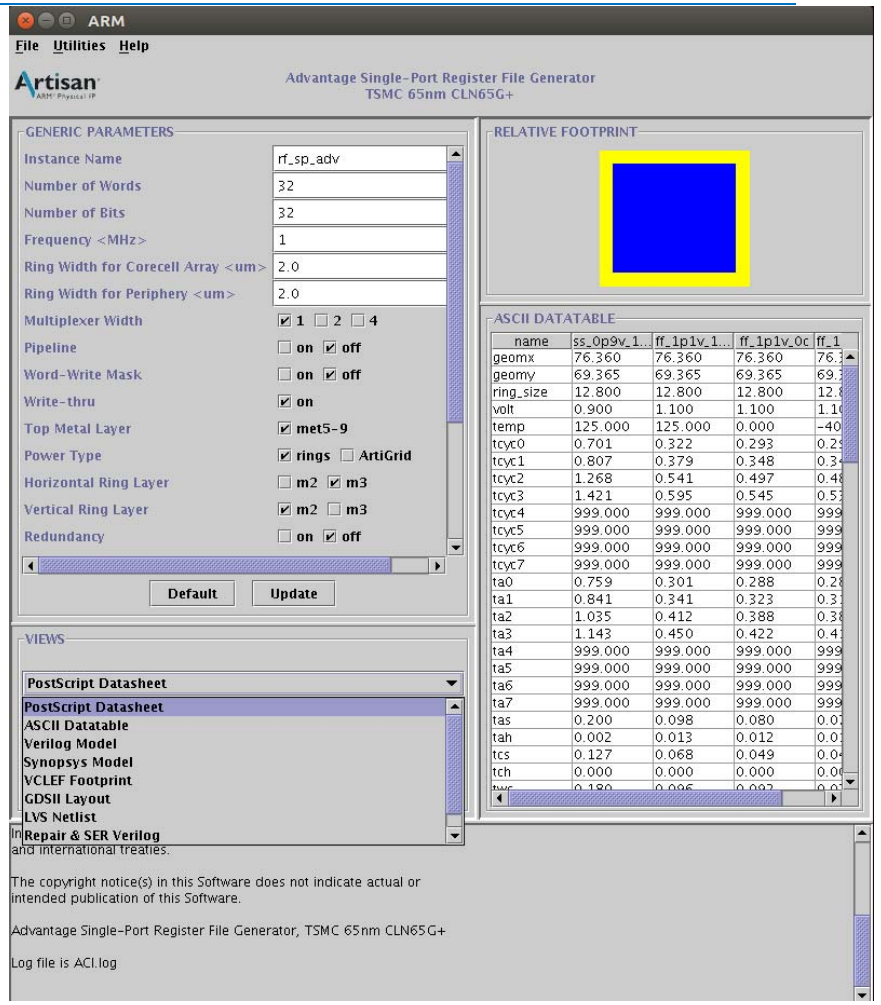
```
module spram #(                 // single-port ram
  parameter   N = 32,           // memory data width
  parameter   M = 10            // memory address width
)(input wire           clk,     // system clock
  input wire  [M-1:0]  addr,    // address
  input wire           we,      // write enable
  input wire  [N-1:0]  wdata,   // write data
  output reg  [N-1:0]  rdata);  // read data
// memory behavior model…
endmodule
```

```
// 2D array: width x depth
reg [N-1:0] mem_array[2**M-1:0];
// synchronous read & write
always@(posedge clk) begin
  // write (write enable == 1)
  if(we) mem_array[addr] <= wdata;
  // read (write enable == 0)
  else   rdata <= mem_array[addr];
end
initial begin
  $readmemb("init_file", mem_array);
end
```

Optional memory initialization: quite common to initialize the content with a file, where each line corresponds 1 word in the memory

Case study: memory – memory compiler

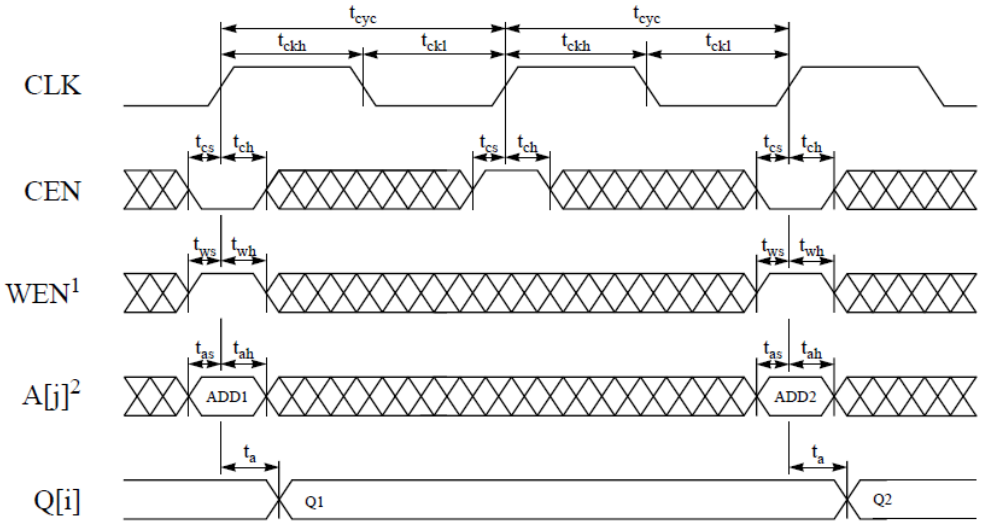
- To build SRAM, we will use the memory compiler from IP vendor (like ARM)
- Specify the basic configuration of a memory
  - Depth; width; pipeline, etc.
- It can generate
  - Datasheet (*.pdf)
  - Verilog behavior model (*.v)
  - Synthesize library (*.lib)
  - Placement and routing library (*.lef)
  - Layout (*.gds)
  - Spice model (*.sp)



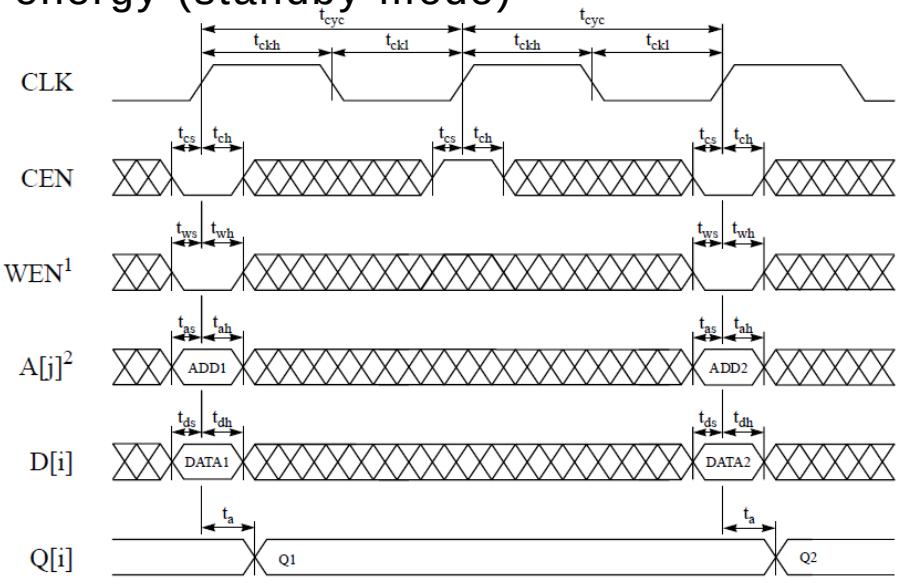
Typical GUI of the memory compiler

Case study: memory – timing diagram

- Timing of a real single port SRAM generated from memory compiler
  - Enable signal is **active low**: 0 means enable
  - Additional chip enable (**CEN**) to save the energy (standby mode)



SRAM read-cycle timing

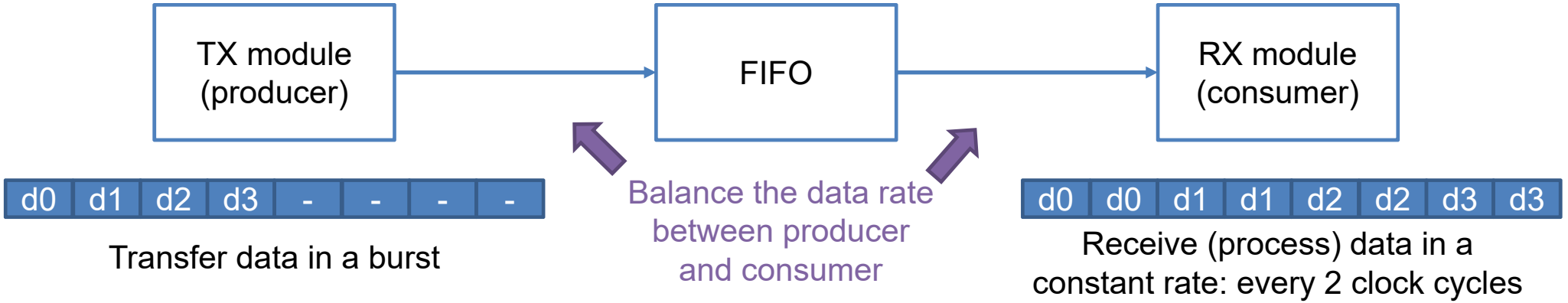


SRAM write-cycle timing

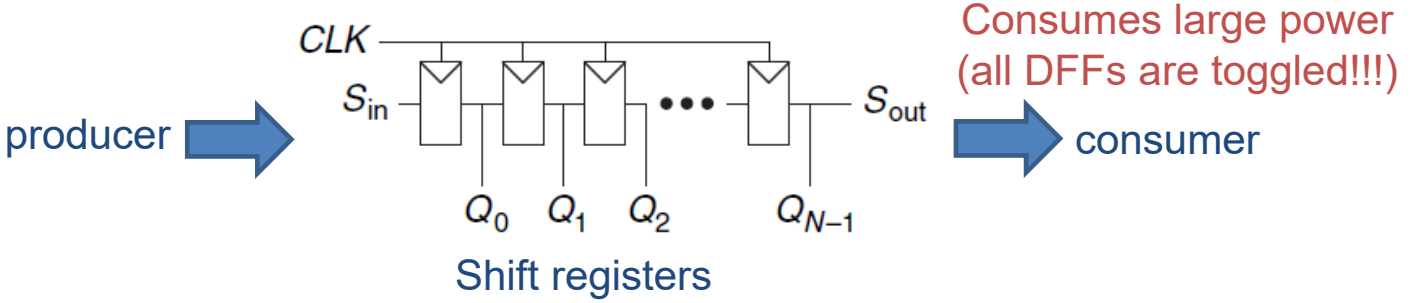
- Exercise: modify the original behavior model to include CEN and adopt the active low enable

Case study: First-In-First-Out (FIFO)

- Act as a buffer to connect two subsystems



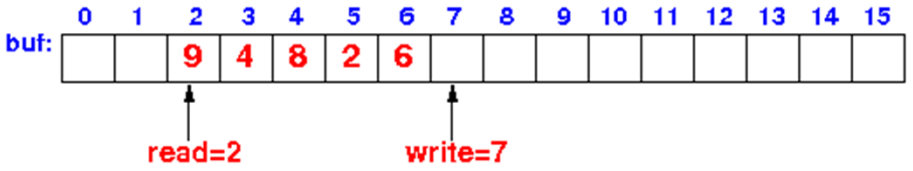
- Behave as the Queue data structure
  - Write operation: push the input data to the tail of FIFO
  - Read operation: remove the head data from the head of FIFO to data port
  - Naïve implementation: shift register

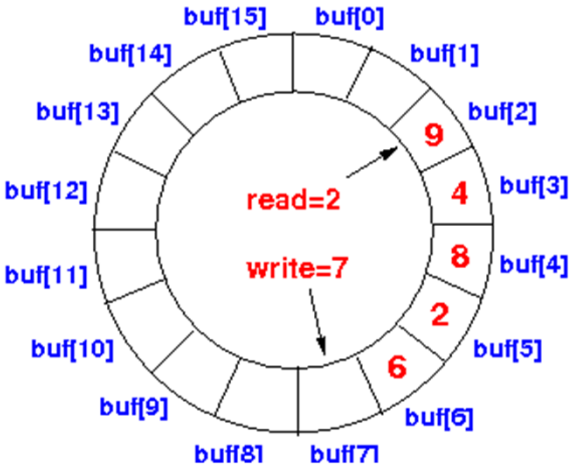


NEVER use shift register!
Consumes large power
(all DFFs are toggled!!!)

Case study: FIFO (cont.)

- Circular-queue-based implementation
  - Maintain two registers
    - Read pointer: buffer index to read from
    - Write pointer: buffer index to write into
  - The FIFO usually contains status signals
    - `full` & `empty`
    - `almost_full` & `almost_empty`: one position to write/read
    - `half_full`: half of FIFO is occupied





read

write

```
module fifo #(
  parameter   N = 32,          // FIFO data width
  parameter   W = 4            // FIFO address width
) (input wire clk, rst,        // system clock & reset
  input wire           r_en,  // read enable
  output wire [N-1:0] r_data,// read data
  input wire           w_en,  // write enable
  input wire  [N-1:0] w_data,// write data
  output wire full, empty);  // status signals
…
endmodule
```

Case study: FIFO (cont.)

- Use a dedicated counter to track the status of FIFO
  - It can be eliminated: https://github.com/olofk/fifo/blob/master/rtl/verilog/fifo.v

```
// Internal variables
reg [N-1:0] fifo_mem[2**W-1:0]; // fifo memory
reg [W-1:0] read_pointer;       // read pointer
reg [W-1:0] write_pointer;      // write pointer
reg [W:0] counter;              // counter for status

// read of fifo memory (combinational)
assign r_data = r_en ? fifo_mem[read_pointer] : 0;

// write of fifo memory (sequential)
always@(posedge clk) begin
  if (w_en) begin
    fifo_mem[write_pointer] <= w_data;
  end
end
```

```
// pointer & counter (sequential + combinational)
always@(posedge clk) begin
  if(rst) begin
    read_pointer  <= 0;
    write_pointer <= 0;
    counter       <= 0;
  end else begin
    case({w_en, r_en})
      2'b01: begin
        read_pointer  <= read_pointer + 1;
        counter       <= counter - 1;
      end
      2'b10: begin
        write_pointer <= write_pointer + 1;
        counter       <= counter + 1;
      end
      2'b11: begin
        read_pointer  <= read_pointer + 1;
        write_pointer <= write_pointer + 1;
      end
    endcase
  end
end
```

This is by far the first time we combine the sequential & combinational logic together. We only recommend to merge the simple combinational logic into sequential logic

Case study: FIFO (cont.)

- The combinational logic and sequential logic can be merged together

```
always@ (posedge clk) begin
  if (rst) counter <= 0;
  else if (en) counter <= counter + 1;
end
```

```
// synchronous DFF (sequential)
always@ (posedge clk) begin
  if (rst) counter <= 0;
  else counter <= counter_next;
end
// next logic (combinational logic)
always@ (*) begin
  if (en) counter_next = counter + 1;
  else counter_next = counter;
end
```

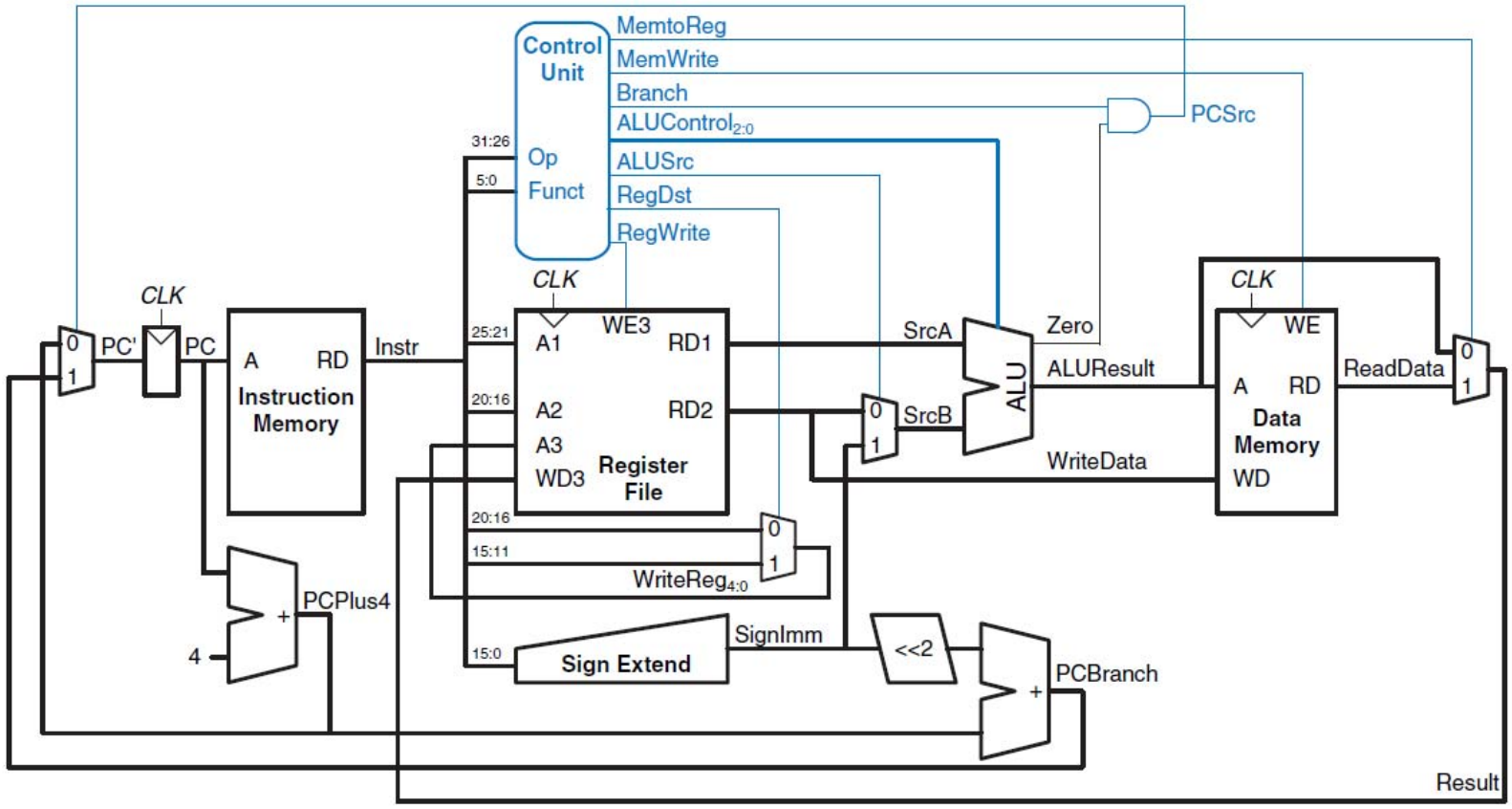
The merge of combinational logic and sequential makes the code more concise, but the circuit-level representation is not as clear as the separate one

```
// status signals
assign full = (counter == 2**W);
assign empty = (counter == 0);
```

- For large FIFO size, we use dual-port RAM (dpram) instead of DFFs
  - `read_pointer` and `write_pointer` are the address port of dpram
- Exercise: write the Last-In-First-Out (LIFO, a.k.a. Stack)

Case study: single-cycle MIPS processor

- A moderately-complex system design: the microarchitecture of **a real CPU**
- Follow top-down design strategy

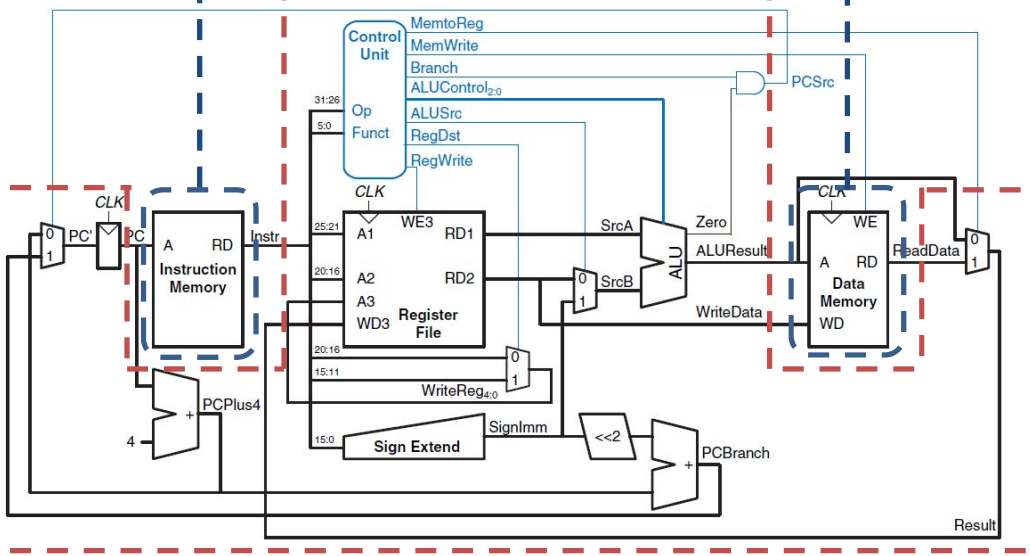


Single-cycle MIPS processor: hierarchy

- Hierarchy of MIPS processor
  - MIPS processor
    - Datapath: ALU, register file, PC, etc.
    - Controller: main decoder, ALU decoder
  - External memory
    - Instruction memory
    - Data memory

Peripheral of CPU processor

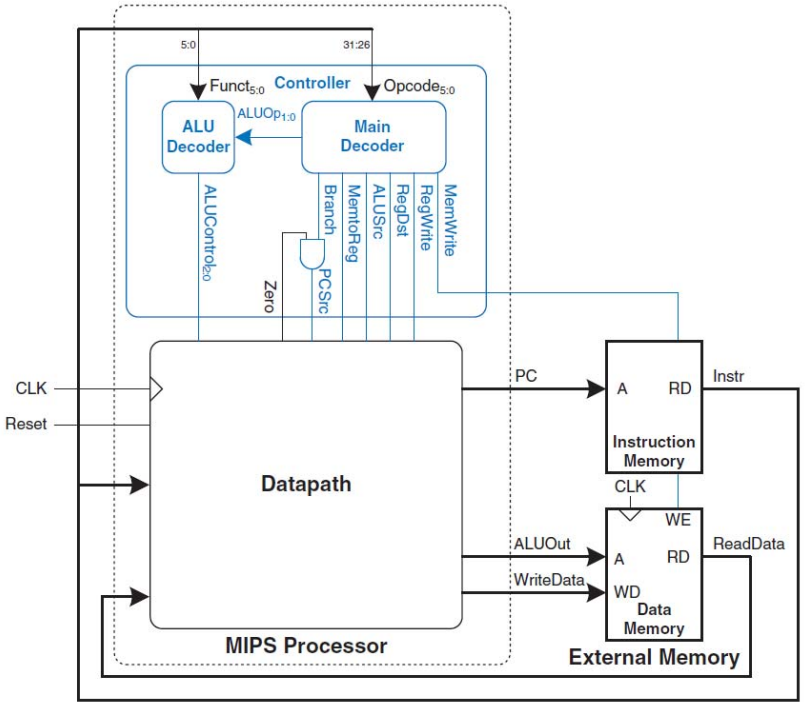
Core of CPU processor



- Only the skeleton of the code is provided for this design
  - Try to fill the missing part as an exercise (**TODO** part)

Single-cycle MIPS processor: MIPS core

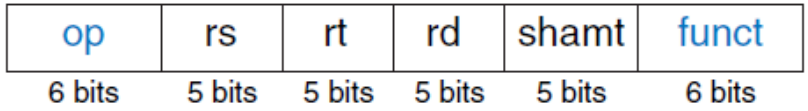
```
module mips_core (
  input  wire        clk, rst,  // system clk & reset
  output wire [31:0] pc,        // instruction ram addr
  input  wire [31:0] inst,      // instruction
  output wire        memwrite,  // data memory write en
  output wire [31:0] aluout,    // data memory address
  output wire [31:0] writedata, // data memory write data
  input  wire [31:0] readdata); // data memory read data
// TODO: internal variables

// Instantiation of sub-modules
// TODO: instantiate controller module
controller c(…);
// TODO: instantiate datapath module
datapath d(…);

endmodule
```



Single-cycle MIPS processor: ISA review

- MIPS has 3 different types of instructions
  - R-type (register-type): 3 register operands, i.e. rs, rt, rd

R-type

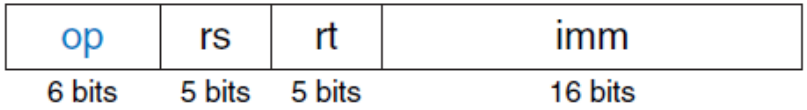


Example

Assembly Code	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
`add $s0, $s1, $s2`	0	17	18	16	0	32
`sub $t0, $t3, $t5`	0	11	13	8	0	34

  - I-type (immediate-type): 2 register operands and 1 immediate operand

I-type

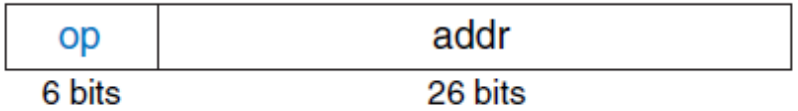


Example

Assembly Code	op	rs	rt	imm
	6 bits	5 bits	5 bits	16 bits
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	–12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4

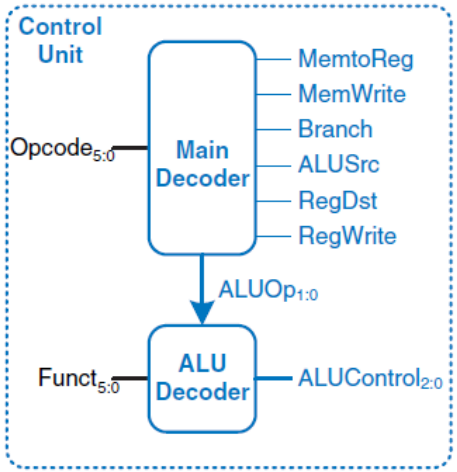
  - J-type (jump-type): only 6-bit opcode and address

J-type



Single-cycle MIPS processor: controller

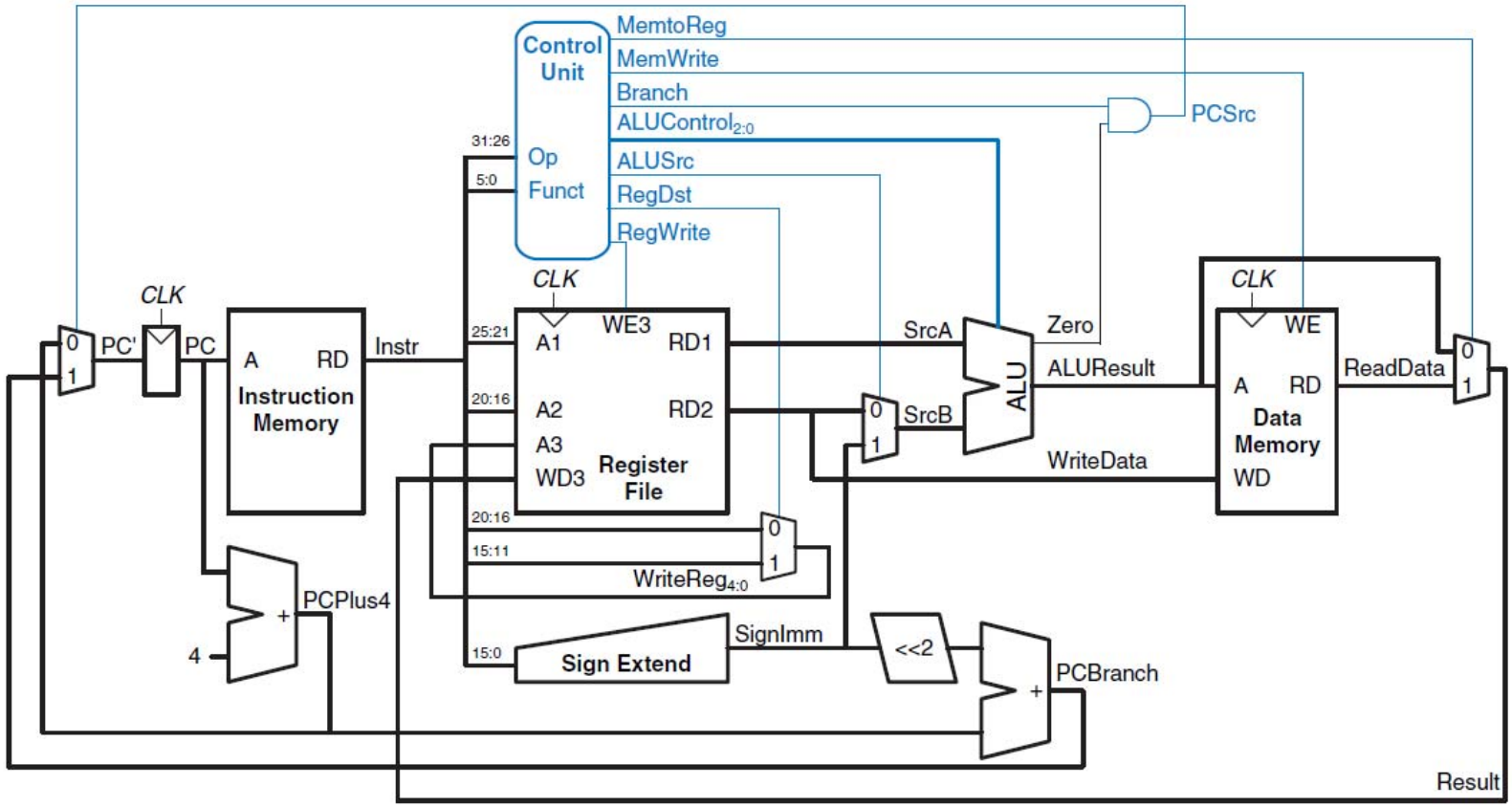
- Based on the types of instruction, we can generate the control signals for the datapath
- The control signals are based on `opcode` & `funct` field
  - `MainDecoder`: determine most control signals from `opcode`
  - `ALUDecoder`: determine `ALUControl` by `funct` and `ALUOp`




```
module controller(
  input wire  [5:0] opcode,     // opcode: inst[31:26]
  input wire  [5:0] funct,      // funct: inst[5:0]
  output wire       memtoreg,   // regfile write data mux select
  output wire       memwrite,   // data memory write enable
  output wire       branch,     // branch instr
  output wire       alusrc,     // ALU operand mux select
  output wire       regdst,     // regfile write addr mux select
  output wire       regwrite,   // regfile write enable
  output wire [2:0] alucontrol // ALU control
);
main_decoder md(…); // TODO: instantiate main decoder module
alu_decoder ad(…);  // TODO: instantiate alu decoder module
endmodule
```

Single-cycle MIPS processor: main decoder

- R-type: MemToReg = 0 (select data from ALU); MemWrite = 0 (not write data memory); Branch = 0 (not branch); ALUSrc = 0 (from 2nd read port of regfile); RegDst = 1 (Rd field); RegWrite = 1 (write to regfile)



Single-cycle MIPS processor: main decoder

- Complete truth table of main decoder (**x** stands for don't care)

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

```
// part of main decoder
always@ (*) begin
  case(opcode)
    6'b000_000: begin  // R-type
      regwrite = 1'b1; regdst = 1'b1; alusrc = 1'b0;
      branch = 1'b0; memwrite = 1'b0; memtoreg = 1'b0;
      aluop = 2'b10;
    end
    // TODO: add more instructions here
  endcase
end
```

ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

Meaning of ALUOp

Single-cycle MIPS processor: ALU decoder

- Generates the control signals for ALU based on `ALUOp` (from main_decoder) and `funct` field (instr[5:0])

```
// part of alu_decoder
always@ (*) begin
  case(aluop)
    alu_control   = 3'b000; // default value
    2'b00: begin
      alu_control = 3'b010; // add (lw/sw/addi)
    end
    2'b01: begin
      alu_control = 3'b110; // sub (beq)
    end
    2'b10: begin            // R-type (check funct)
      case(funct)
        6'b100_000: begin
          alu_control = 3'b010; // add
        end
        // TODO: add more R-type decodings here
      endcase
    end
end
```

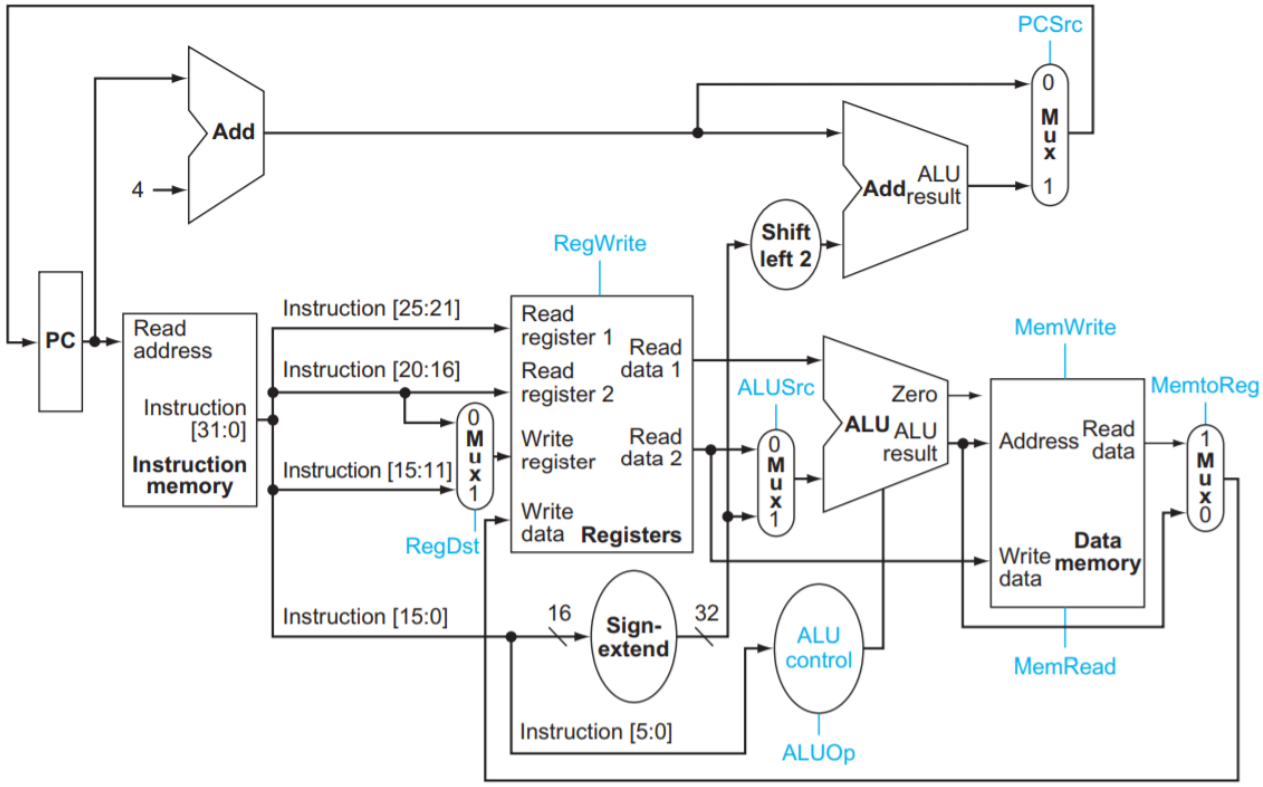
Truth table of ALU decoder

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Single-cycle MIPS processor: datapath

- Main component: ALU, register file (regfile), sign extension, mux

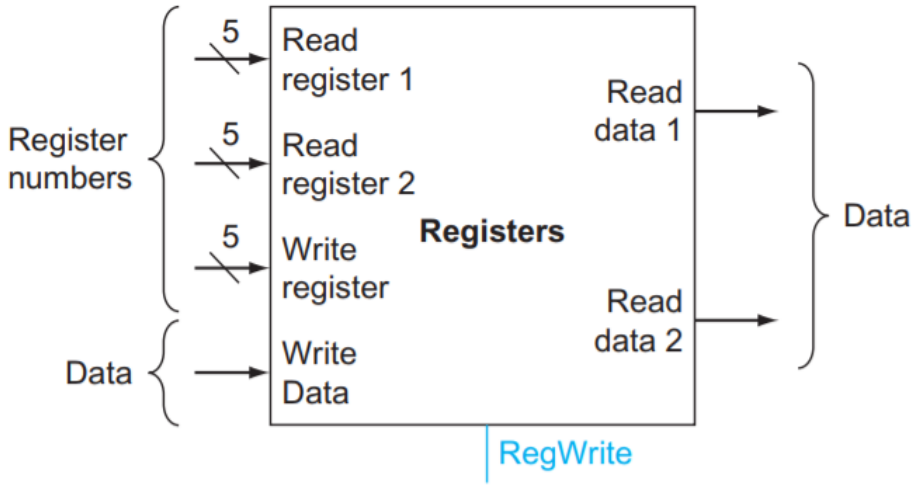
```
// part of datapath
// TODO: PC register & add
dff pc(…);
adder pc_adder(…);
// TODO: register file
regfile reg_file(…);
// TODO: sign extension
signext sign_ext(…);
// TODO: ALU
alu alu(…);
// TODO: Mux
mux pc_mux(…);
mux wr_mux(…);
…
```



Single-cycle MIPS processor: regfile

- 3-ports: 2 read ports (for rs & rt); 1 write port (for rd)

```
// part of regfile
reg [31:0] rf_mem [31:0]; // 32 32-bit registers

// write port (sequential logic)
always @(posedge clk) begin
  if (w_en) rf_mem[w_addr] <= w_data;
end
// read data port 0 ($0 is reserved to be 0)
assign r_data_0 = (r_addr_0 == 0) ? 0 :
                   rf_mem[r_addr_0]
// read data port 1 ($0 is reserved to be 0)
assign r_data_1 = (r_addr_1 == 0) ? 0 :
                   rf_mem[r_addr_1]
```



Single-cycle MIPS processor: ALU

- ALU: arithmetic/logical unit
- Suggestion: use the primitive operator +, -

```
// part of ALU (combinational)
always@(*) begin
  aluresult = 32'b0;              // default value
  zero      = 1'b0;               // flag for zero
  case(alucontrol)
    3'b000: aluresult = a & b;  // bit-wise and
    3'b001: aluresult = a | b;  // bit-wise or
    3'b010: aluresult = a + b;  // add
    3'b100: aluresult = a & ~b; // and bar
    // TODO: remaining cases
  endcase
end
```

Supported ALU operations

$F_{2:0}$ ALUControl	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

Single-cycle MIPS processor: top

- Ideal behavior model of DMEM & IMEM (not for synthesize)

```
// Behavior model of IMEM (like a ROM)
reg [31:0] ROM [127:0]; // max support 128 instr
assign r_data = ROM[addr[8:2]];
initial begin
  $readmemh("memfile.dat", ROM);
end
```

PC provides the byte address, change to word address

```
// Behavior model of DMEM (like a RAM)
reg [31:0] RAM [127:0]; // max support 128 data
// data memory read
assign r_data = RAM[addr[8:2]];
// data memory write
always@(posedge clk) begin
  if (wen) RAM[addr[8:2]] <= w_data;
end
```

```
// Top module
// MIPS processor core + external mem
module top(
  input wire clk,
  input wire rst
);
// TODO: Instantiate MIPS core
mips_core mips(…);
// External memory
imem imem(…); // TODO: IMEM
dmem dmem(…); // TODO: DMEM
endmodule
```

Single-cycle MIPS processor: Testbench

- Run the *ad hoc* code of supported instructions

Binary machine code (stored in the `memfile.dat`)

#	Assembly	Description	Address	Machine
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c
	addi $7, $3, −9	# initialize $7 = 3	8	2067fff7
	or $4, $7, $2	# $4 = (3 OR 5) = 7	c	00e22025
	and $5, $3, $4	# $5 = (12 AND 7) = 4	10	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a
	beq $4, $0, around	# should be taken	20	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820
	sub $7, $7, $2	# $7 = 12 − 5 = 7	30	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050
	j end	# should be taken	3c	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001
end:	sw $2, 84($0)	# write mem[84] = 7	44	ac020054

Single-cycle MIPS processor: Testbench (cont.)

- For the simple testbench we only need to monitor the waveform

```
// Testbench for MIPS processor
module testbench;
// clock & reset
reg clk, rst;
// TODO: clock generation
// TODO: reset generation
// DUT instantiation
top dut(clk, rst);
endmodule
```

```
vcs +v2k -f mips.flist \
  -timescale=1ns/10ps  \
  -top testbench -debug_all
```

VCS compile option

Dump the waveforms of all variables

```
filepath/testbench.v
filepath/top.v
filepath/mips_core.v
filepath/imem.v
filepath/dmem.v
filepath/controller.v
filepath/main_decoder.v
filepath/alu_decoder.v
filepath/datapath.v
filepath/alu.v
# TODO: put remaining source files
```

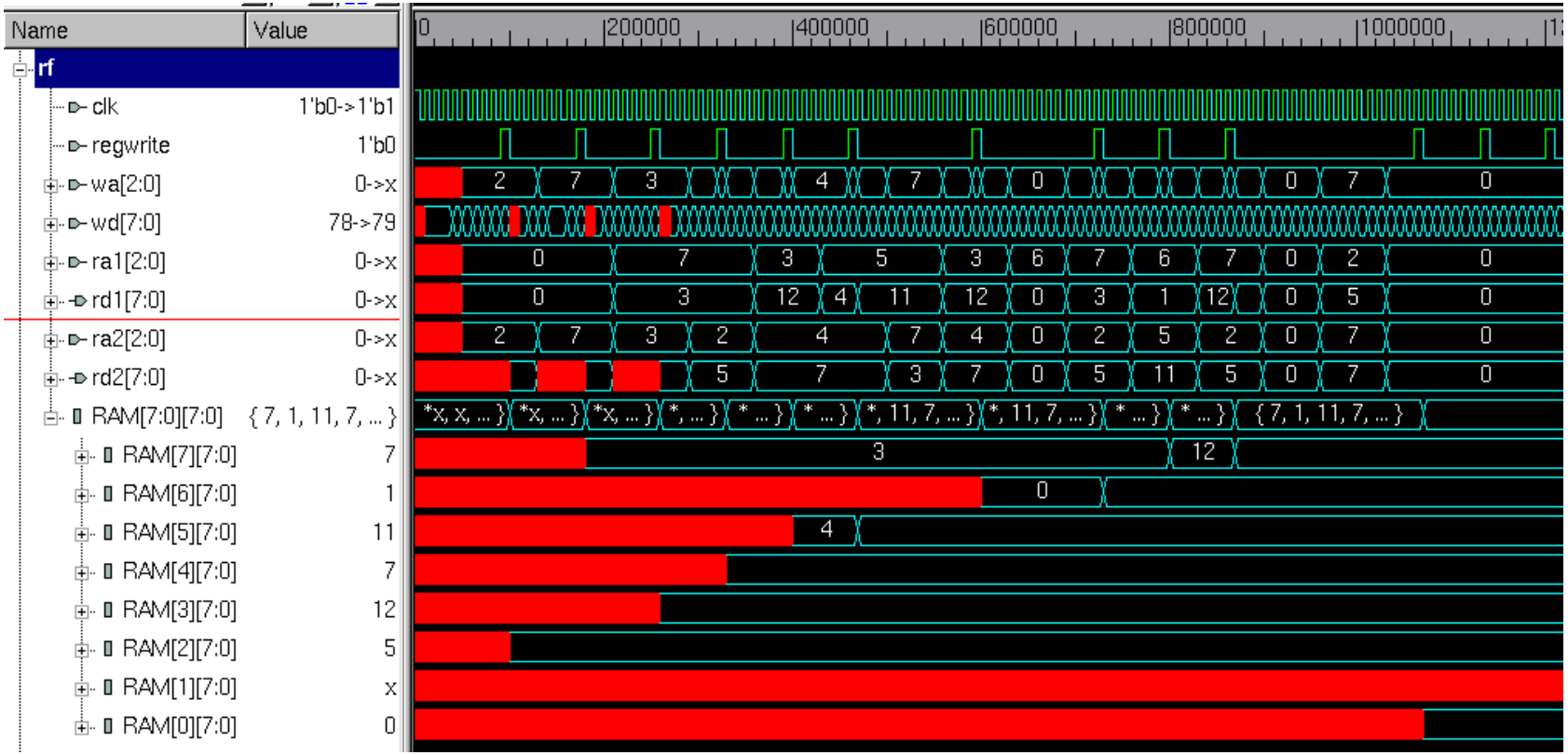
Sampled file list for all Verilog source codes

```
>> ./simv -gui
```

Launch DVE for waveform inspect after compilation

Single-cycle MIPS processor: Testbench (cont.)

- Trace the register content during the instruction execution



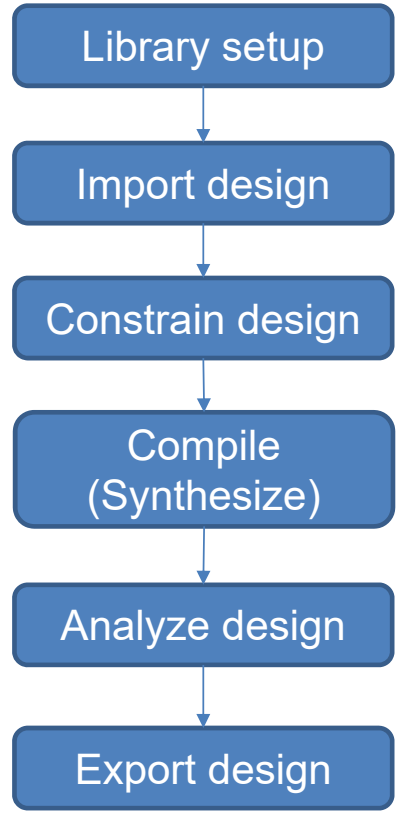
Sampled waveform from DVE of the register file module in MIPS processor

Summary on Verilog

- Hardware description languages (HDL) are different from software programming languages
  - Describe a hardware diagram instead of executing the instructions
- Top-down design flow
  - Sketch a block diagram of your system
  - Identify the combinational logics and the sequential logics
    - Simple combinational logic: continuous assignment & blocking assignment (=)
    - Complicated combinational logic: `always@(*)` block & blocking assignment (=)
      - Outputs are complete: `case-default`, `if-else`, define the default at the beginning
    - Sequential logic: `always @(posedge clk)` & nonblocking assignment (<=)
  - Split a complex design into multiple simple designs (divide and conquer)

Digital synthesis flow

- Load technology data and design
  - Technology data: standard cell library provided by the foundry
  - Design: the HDL file of your design (*.v, *.vhd, *.sv)
- Apply the constraints
  - The target frequency of your design (e.g. 1ns)
  - The input / output latency
  - The environment attributes (transition of inputs, load of outputs)
- Synthesize the design
  - HDL file is compiled into the gate-level netlist
  - Optimization is handled by the software (e.g. Boolean simplification)
- Analyze the results
  - Check the critical path, area, power of the synthesized circuit
- Write out the design data
  - Dump the netlist and design file for placement and routing (P&R)



Synthesis tool

- In this tutorial, we will use Synopsys Design Compiler (DC)
- Three different interfaces to launch the DC
  - Interactive GUI `>> design_vision`
  - Interactive shell

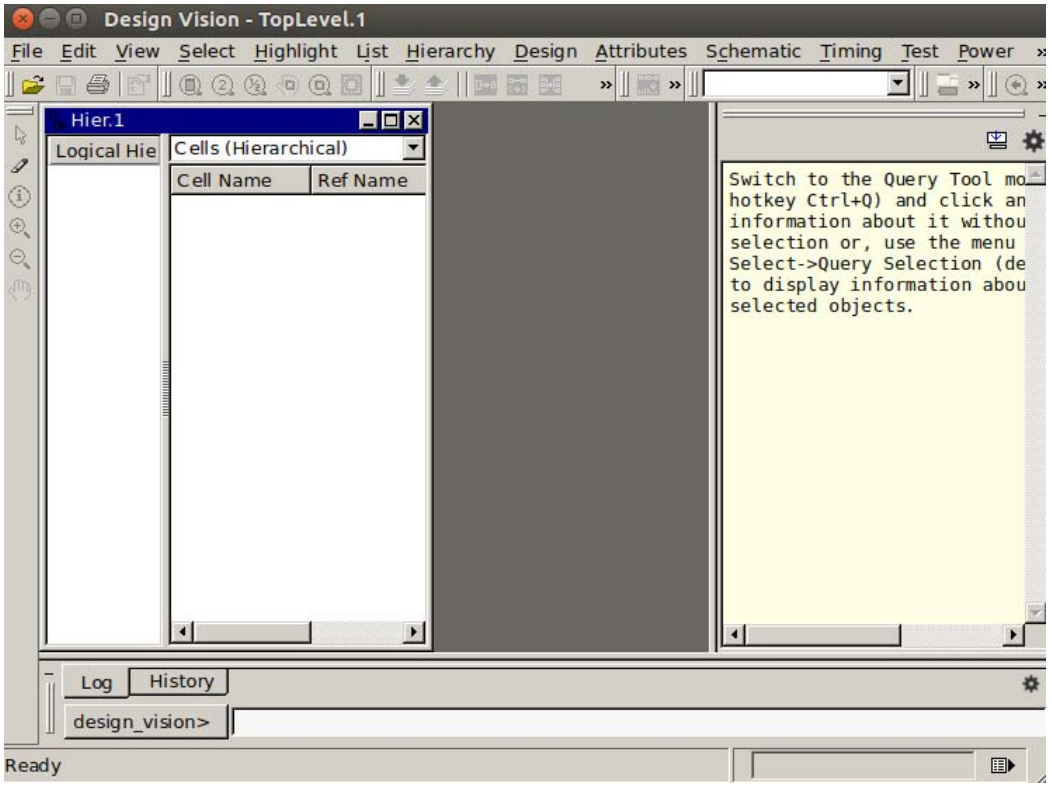
```
>> dc_shell
Initializing…
dc_shell>
```

  - Batch mode (preferred)

```
>> dc_shell –f RUN.tcl | tee –i log
```

Top TCL script

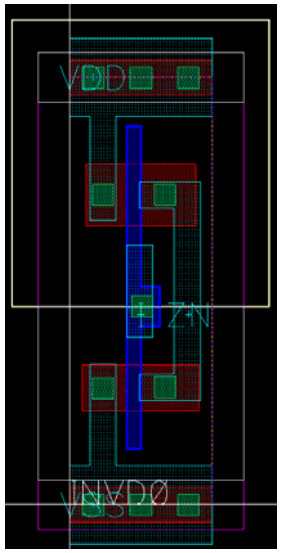
Linux command:
`|` stands for pipe
`tee`: read from standard inputs to output and files
Here we will dump the design compiler output info to the `log` file for further inspection



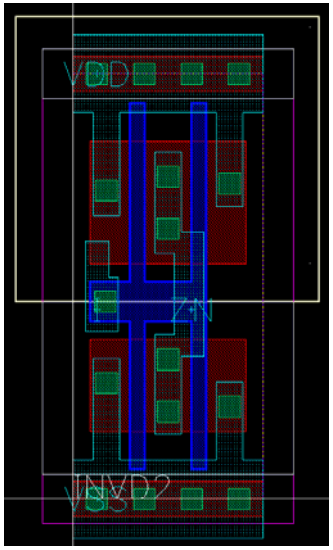
Design Vision: the GUI mode of design compiler

The standard cell library

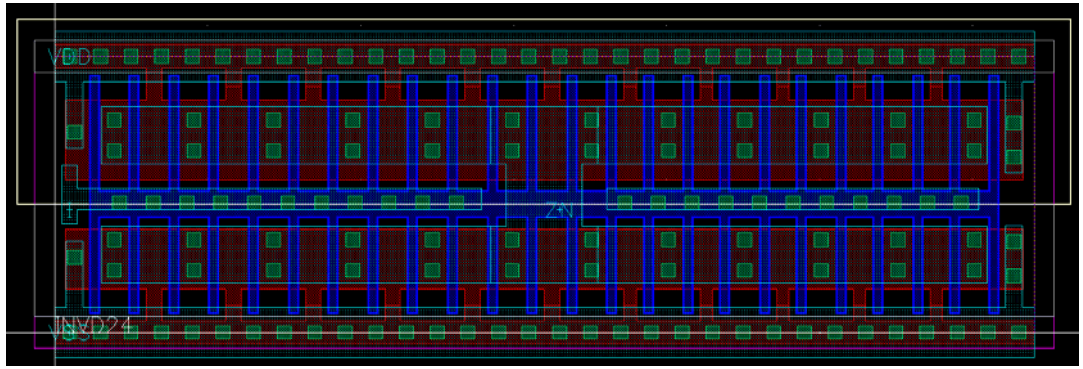
- The standard cell library includes many basic digital gates
  - Simple combinational logic: inverter, buffer, nand, etc.
  - Complex combinational logic: AO (and-or), AOI (and-or-inverter), etc.
  - Sequential logic: DFF, DFF with synchronous set/reset, etc.
  - Datapath logic: booth encoder, full adder, etc.
- Each type of a gate has different sets of driving capabilities



INV0



INV2



INV24

Here we show layouts of 3 inverters with different driving capabilities in TSMC65nm

Logical technology library

- The technology library (ASCII: *.lib; binary: *.db) is the abstraction of each gate in the standard cell

```
cell (INVD0) {
  area : 1.08;
  …
  leakage_power () {
    value : 0.027842;
    related_pg_pin : VDD;
  }
  …
  pin(I) {
    direction : input;
    related_ground_pin : VSS;
    related_power_pin : VDD;
    capacitance : 0.0007247;
    rise_capacitance : 0.0007247;
    fall_capacitance : 0.0007004;
  }
  pin(ZN) {
    direction : output;
    function : "(!I)";
    …
    timing () {
      cell_rise (delay_template_7x7_0) {
        index_1 ("0.0049, 0.0125, 0.0277, 0.0582, 0.1192, 0.2412, 0.4851");
        index_2 ("0.00053, 0.00099, 0.00191, 0.00375, 0.00744, 0.01481, 0.02955");
        values ( \
          "0.01464, 0.01843, 0.02588, 0.04043, 0.06986, 0.1281, 0.2438", \
          "0.01664, 0.02039, 0.02788, 0.04256, 0.07167, 0.1298, 0.246", \
        …
```

- Area
- Leakage power
- Capacitance of input
- Logic of the gate
- Timing of a gate (LUT)
  - index_1: input transition time
  - index_2: total output capacitance load

Import logical technology library

- Target library: the target library that we want the circuit to be synthesized into
  - Set the taget_library to the logic technology library

```
set_app_var target_library filepath/tsmc65_wc.db
```

  - set_app_var: the same as `set`
  - There exists different corners of library, we will use the **worst case**!
  - The import library is in **the format of binary (*.db),** not ASCII (*.lib). The vendor will provide the both db and lib or we can compile the db file from lib

- Link library: resolve the instantiated references in the design

```
// part of design
spram32x1024 mem(…);
```

A memory in the design

This cell is define in the spram32x1024.lib which is generated by the memory compiler. So we should include it into our link library

```
set_app_var link_library “* $target_library filepath/spram32x1024.db”
```

  - * stands for the DC memory, i.e. all designs DC has seen up to now
  - It is a good habit to include the target library into the link library as well. Otherwise, you will encounter **the unresolved error** when you read a synthesized design
  - $ is used to substitute the variable (TCL syntax)

Search path and default setting

- We can set the search_path to simplify our library and design import
  - Include the standard cell path to the search path
  - Include the RTL design path to the search path

```
set_app_var search_path “$search_path filepath_to_standard_cell \
  filepath_to_memory_lib filepath_to_rtl”
```

- Default settings: put the library related settings into a hidden file `.synopsys_dc.setup` in the **current work directory (CWD)**
  - Automatically run the TCL code in it when launch DC there

```
# .synopsys_dc.setup file in the project’s `CWD`
set_app_var search_path “$search_path filepath_to_standard_cell \
  filepath_to_memory_lib filepath_to_rtl”
set_app_var target_library tsmc65n_wc.db
set_app_var link_library “* $target_library sram32x1024.db”
```

Import the design

- We can then import the HDL design into DC
  - Suppose we have 4 Verilog files: A.v, B.v, C.v, Top.v

```
# ensure RTL path is included in search_path
define_design_lib WORK –path ./work
analyze –format verilog {A.v B.v C.v Top.v}
elaborate Top; # Top module name
```

  - Design lib: define a new path to store the intermediate files during design import. By default, it will dump to the CWD, and mess up your CWD
  - Analyze: read Verilog and check syntax of source codes
  - Elaborate: set the specified design to the current design, link the current design, and build the GTECH design (technology-independent) in DC memory
  - Common **pitfall**: import the behavior model or testbench into the design
    - Behavior model of memory can never be imported!!! Otherwise large amounts of DFFs will be synthesized later
    - Sol: set **link_library** to include the library file for resolve the reference

Save the design after importing the design

- A good habit to save the design after analyze & elaborate

```
write -format ddc -hier -output filepath/my_design_unmapped.ddc
```

Combine the hierarchy design into a single file. It is convenient for manipulation during synthesis and place routing.

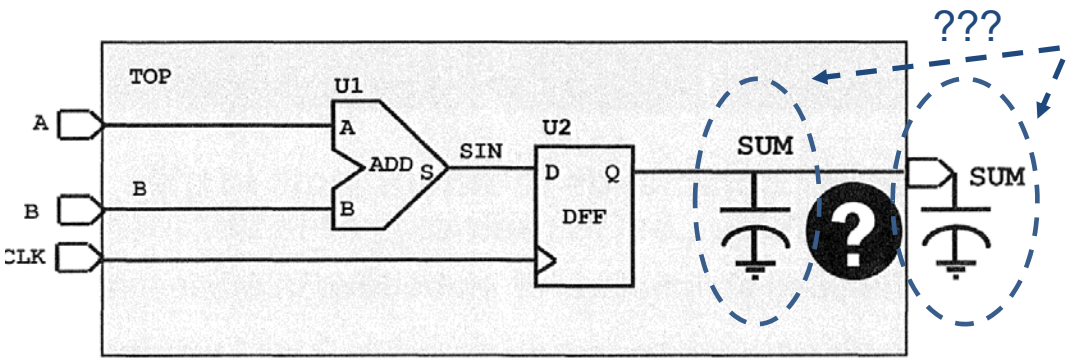
  - DDC is a binary format of DC
    - It stores netlist, constraints, and attributes
    - It is more efficient to directly read in DDC rather than Verilog code
- After synthesizing, we will save our mapped design to DDC

```
# After compilation, we will save the synthesized design
change_names -rule verilog -hier
write -format verilog -hier -output filepath/my_design_mapped.v
write -format ddc -hier -output filepath/my_design_mapped.ddc
```

  - We dump the design into Verilog so that we can run the gate-level simulation and export to the third-party place & route tools
  - `change_names`: replace the special characters with non-special ones to make it compatible with other tools
    - E.g. `\bus[2] -> bus_2_`

Constrain our design

- We can not compile (synthesize) directly after we import the design and technology library
- Tools need to know the optimization object: e.g. frequency
- We should be able to specify the design in DC before constraining the design
  - Design: module name in Verilog
  - Cell: instance name in Verilog
  - Port: primary IO in the current design (top design)
    - Special port: `clock`
  - Pin: IO of any cell instantiated in the current design
    - Pin is always associated with a specific instance: `cell_name/pin_name`



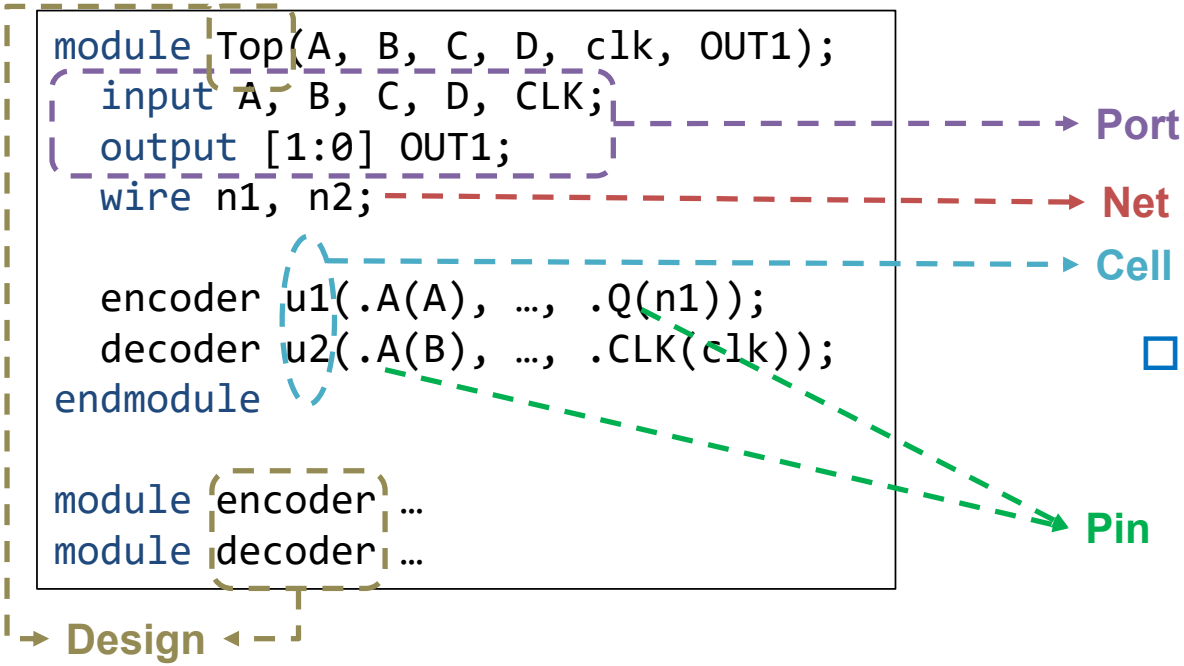

```
set_load 5 SUM
```

It is ambiguous to set the load of `SUM` here:
- Internal wire SUM: **overwrite** the wire load of internal wire SUM with 5
- Output port SUM: **add** 5 to the wire load of internal wire SUM (make it larger)

The reason for specifying the design: otherwise we may not constrain our design properly

Example of design objects in Verilog

- Designs: {Top, encoder, decoder}
- Cells: {u1, u2}
- Pins: {u1/A, u1/Q, u2/A, u2/CLK}



- get_* command to obtain the objects and apply the constraint
  - get_cells
  - get_clocks
  - get_nets
  - get_pins
  - get_ports
  - get_designs
  - get_libs
- We can use ? or * wildcards

```
# set the loading to be 5 unit
# for the ports of the current
# design with name starts with
# addr_bus
set_load 5 [get_ports addr_bus*]
```

Simple example of setting the load

Some handy all_* commands

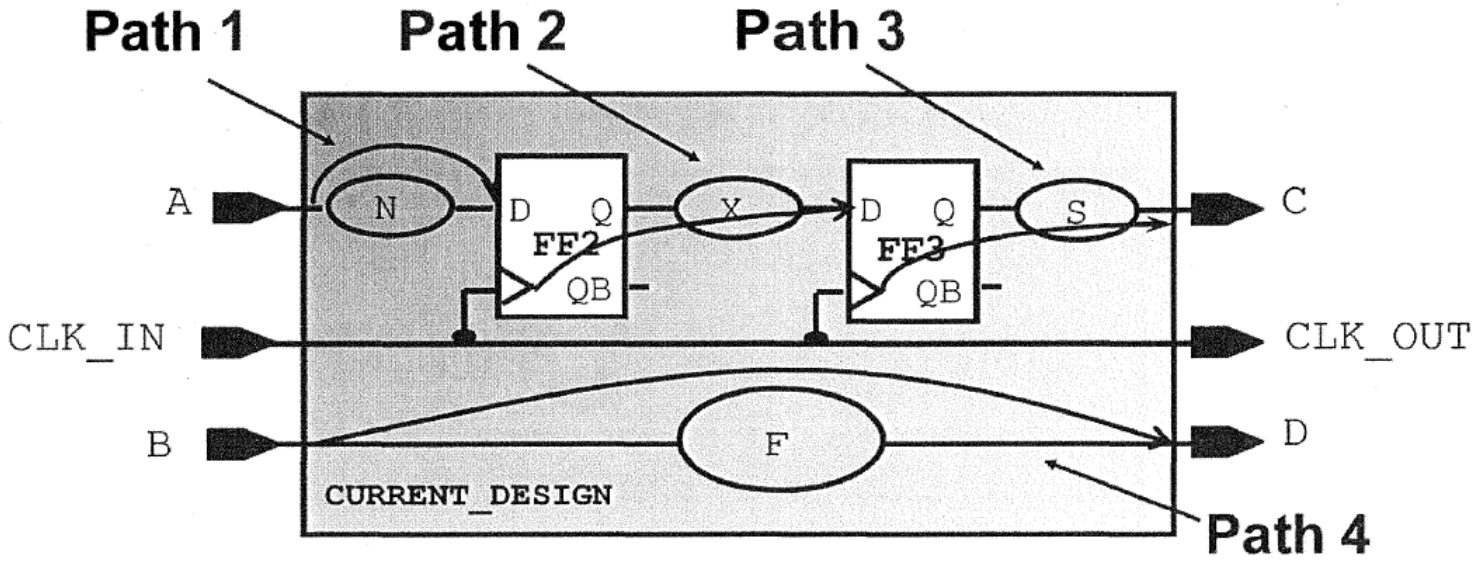
- all_inputs: get all input and inout ports of the current design
- all_outputs: get all output and inout ports of the current design
- remove_from_collection: remove the collection of arg_2 from the collection of arg_1

```
# handy collections includes all input ports except clock
set all_inputs_except_clk [remove_from_collection [all_inputs] \
  [get_ports clk]]
```

The design has a clock port of name `clk`

Constrain the critical path of design

- Critical path: the combinational logic in the design with the longest latency
- 4 different types of critical path
  - Primary input to the internal register (D port): path 1
  - Critical path between internal registers (Q port to D port): path 2
  - Internal register (Q port) to the primary output: path 3
  - Primary input to primary output (pure combinational): path 4

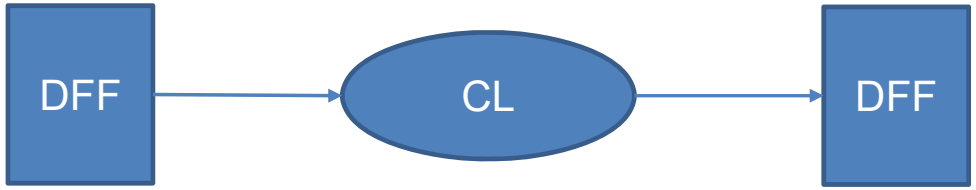


Constrain the critical path in DC

- Constrain the latency between internal registers

```
create_clock -period 2 [get_ports clk]
```

Ensure the clk matches with your design name

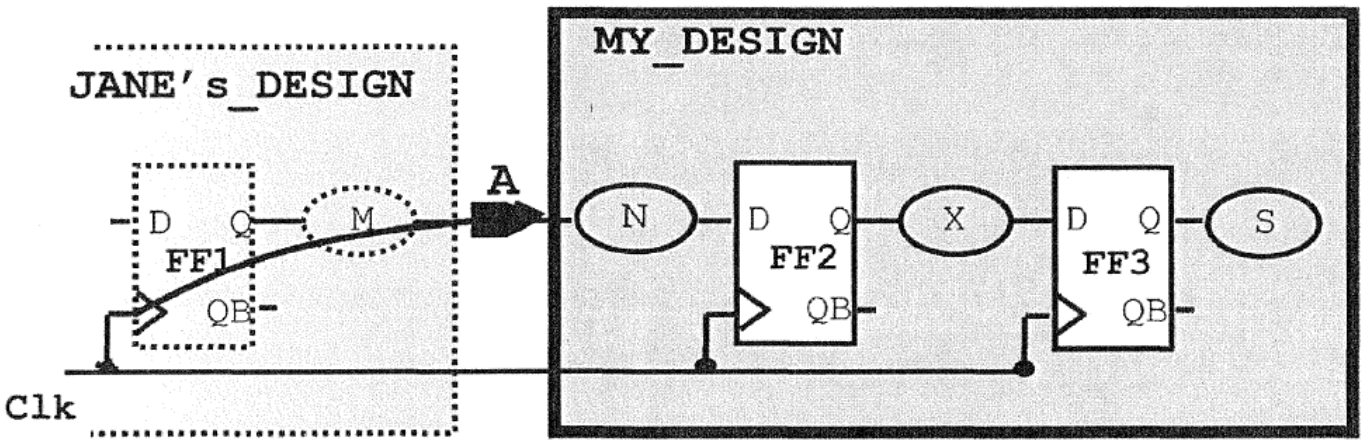


$$T_{clk2q} + T_{CL} + T_{setup} \leq 2$$

  - Advance clock network modeling
    - Clock skew: `set_clock_uncertainty`
    - Clock transition: `set_clock_transition`

Constrain the critical path in DC (cont.)

- Constrain the latency from the primary input to the internal register
- Define the input delay
  - E.g. there might be some potential logic M in the driving module

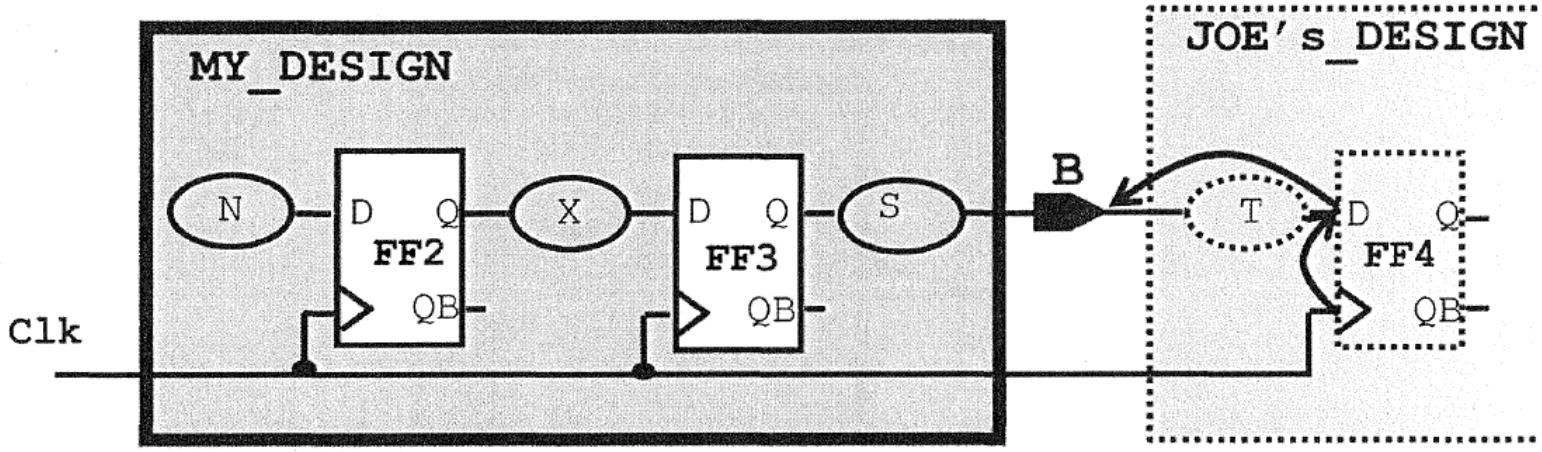



```
create_clock -period 2 [get_ports clk]
set_input_delay -max 0.6 -clock clk [get_ports A]
```

$$0.6 + T_N + T_{setup} \leq 2$$

Constrain the critical path in DC (cont.)

- Constrain the latency from the internal register to the primary output
- Define the output delay
  - E.g. there might be some potential logic T before we reach to the DFF input

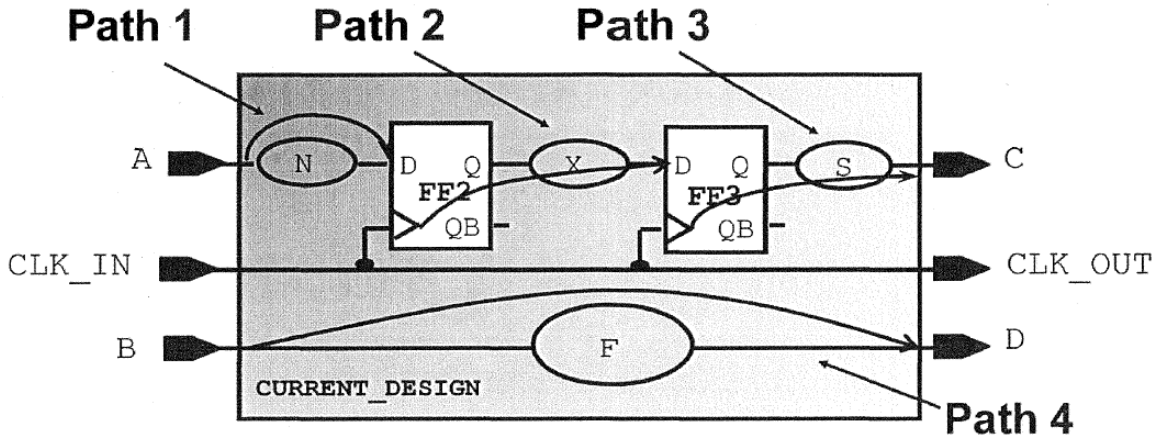



```
create_clock -period 2 [get_ports clk]
set_output_delay -max 0.8 -clock clk [get_ports B]
```

$$T_{clk2q} + T_S + 0.8 \leq 2$$

Constrain the critical path in DC (cont.)

- Constrain the latency between primary input to primary output
- Automatically constrain by the clock period, input delay and output delay




```
create_clock -period 2 [get_ports clk]
set_input_delay -max 0.6 -clock clk [get_ports B]
set_output_delay -max 0.8 -clock clk [get_ports D]
```

$$0.6 + T_F + 0.8 \leq 2$$

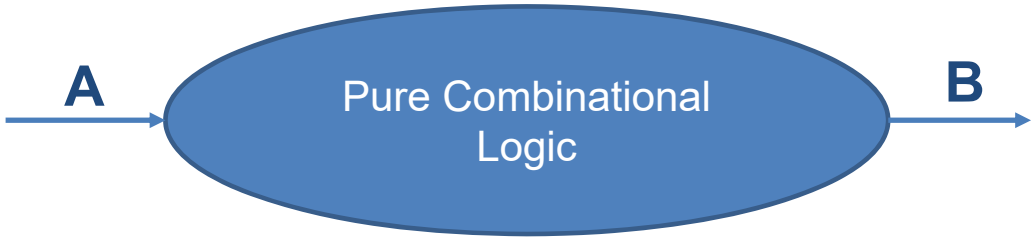
Constrain a pure combinational design

- There is no clock port in this design
- Solution: create a virtual clock (**VCLK**)
  - A clock that is not connected to any port or pin within the current design

```
create_clock -period 2 -name VCLK
```

Give a name: VCLK here

No associated port or pin name



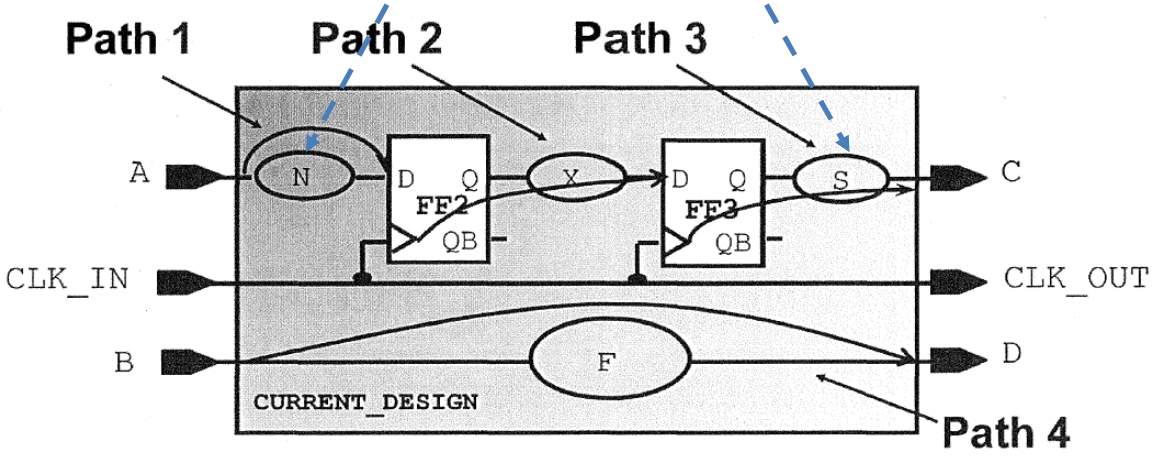

```
create_clock -period 2 -name VCLK
set_input_delay -max 0.6 -clock VCLK \
  [get_ports A]
set_output_delay -max 0.8 -clock VCLK \
  [get_ports B]
```

$$0.6 + T_{CL} + 0.8 \leq 2$$

Time budgeting

- If we don't know the input delay and the output delay in advance
- Time budgeting: conservative on input & output delay
  - 40% of the clock period is reserved for the input logic
  - 40% of the clock period is reserved for the output logic

Logic N & S: 40% clock period



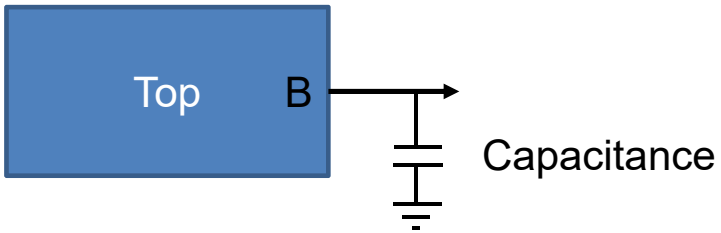
The input & output delay is referred to the **external** latency (i.e. 60% clock period)

```
# A generic timing budgeting constraint
create_clock -period 1.0 [get_ports clk]
set_input_delay -max 0.6 -clock clk \
  [all_inputs]
remove_input_delay [get_ports clk]
set_output_delay -max 0.6 -clock clk \
  [all_outputs]
```

Remove the input delay of the clock port

Environment attribute

- The latency of a gate depends on the input signal transition time and output loading
- Output load
  - Absolute value: set a 10pF at the output port B



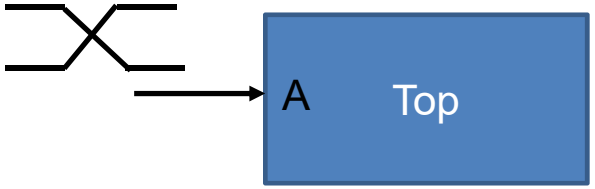

```
set_load 10 [get_ports B]; # cap unit: pF
```

  - Use an input pin of a cell in the library: 3x INV2 I input

```
set_load [expr {[load_of tcbn64lpwc/INV2/I]*3}] [get_ports B]
```

Suppose there is an INVD2 in the library `tcbn64lpwc`, modify it based on your own library

- Input transition
  - Absolute value: set 0.1ns transition at the input port A



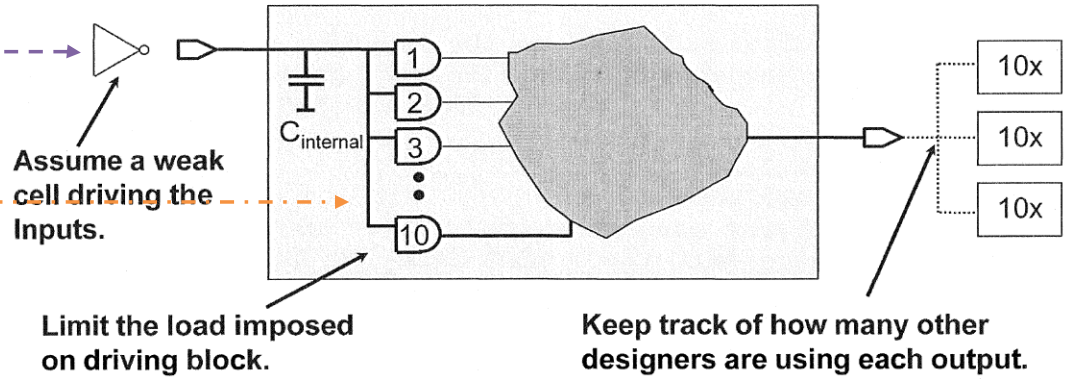

```
set_input_transition 0.1 [get_ports A]
```

  - Use an output pin of a cell in the library: Q port of DFF

```
set_driving_cell –lib_cell DFCN0 –pin Q [get_ports A]
```

Load budgeting

- Assume a weak cell driving the inputs (to be **conservative**): e.g. INV1
- Limit the input capacitance of each input port: `set_max_capacitance`
- Estimate the number of other major blocks at outputs




```
set all_inputs_except_clk [remove_from_collection [all_inputs] [get_ports clk]]
# Assume a weak device driving inputs
set_driving_cell -no_design_rule -lib_cell INV1 -pin Z $all_inputs_except_clk
# Limit the input load: 10x AND2 gate, assume AND2 gate with pin A exists
set max_input_load [expr {[load_of tcbn64lpwc/AND2/A] * 10}]
set_max_capacitance $max_input_load $all_inputs_except_clk
# Limit the output load: each output can drive up to 3*10 ANDs gates
set_load [expr {$max_input_load * 3}] [all_outputs]
```

Constraint file summary

- Store the timing constraint & environment attribute into a single constraint file

```
# sampled constraint file: sample.constraints.tcl
# timing constraint
create_clock –period 1.0 [get_ports clk]
set_clock_uncertainty –setup 0.1 [get_ports clk]; # clock skew
set_input_delay –max …
set_output_delay –max …
# environment attribute
set_load …
set_input_transition …
# or
set_driving_cell …
```

- In dc_shell>> source sample.constraints.tcl

Compile the design

- >> compile
  - Invoke the synthesis procedure to compile the design
- >> compile_ultra
  - Apply multiple optimizations to synthesize the design, including
  - Boundary optimization
  - Auto ungrouping (turn off -no_autoungropu)
    - Break the hierarchy of design, which is not preferred for the purpose of debug
  - Adaptive retiming
  - Register replication
  - …

Check manual of compile_ultra for more details:
dc_shell>> man compile_ultra

Analyze the synthesized results

- report_area: report the area breakdown of the synthesized design
- report_power: report the power consumption of the synthesized design
  - Assume an ideal toggle rate 10% for each net
- report_timing: report the critical path
  - Slack: positive means meeting your time specification
  - Observe the critical path report
    - Starts from Q port of a DFF
    - Ends at D port of a DFF

The timing report is meaningful. It automatically identifies the critical path. Here the data arrives at D port @ 1.49ns, but the required arrival time is 1 – 0.08 (setup) = 0.92ns. The slack is 0.92-1.49 = **-0.57**. Negative means **violating** the spec!!!

```
Point                                                   Fanout    Trans  Incr  Path
-------------------------------------------------------------------------------
clock ideal_clock1 (rise edge)                                    0.00  0.00
clock network delay (ideal)                                       0.00  0.00
elm_S0S1$002/out_reg[1]/CLK (DFFX1)                               0.00  0.00  0.00 r
elm_S0S1$002/out_reg[1]/Q (DFFX1)                                 0.03  0.18  0.18 f
elm_S0S1$002/out[1] (net)                               2         0.00  0.18 f
elm_S0S1$002/out[1] (Reg_0x45f1552f10c5f05d_6)                    0.00  0.18 f
elm_S0S1$002$out[1] (net)                                         0.00  0.18 f
...
minmax1_S1/out_max[7] (net)                             1         0.00  1.45 r
minmax1_S1/out_max[7] (MinMaxUnit_0x152ab97dfd22b898_0)           0.00  1.45 r
minmax1_S1$out_max[7] (net)                                       0.00  1.45 r
elm_S1S2$003/in_[7] (Reg_0x45f1552f10c5f05d_9)                    0.00  1.45 r
elm_S1S2$003/in_[7] (net)                                         0.00  1.45 r
elm_S1S2$003/out_reg[7]/D (DFFX2)                                 0.04  0.04  1.49 r
data arrival time                                                       1.49

clock ideal_clock1 (rise edge)                                    1.00  1.00
clock network delay (ideal)                                       0.00  1.00
elm_S1S2$003/out_reg[7]/CLK (DFFX2)                               0.00  1.00 r
library setup time                                               -0.08  0.92
data required time                                                      0.92
-------------------------------------------------------------------------------
data required time                                                      0.92
data arrival time                                                      -1.49
-------------------------------------------------------------------------------
slack (VIOLATED)                                                       -0.57
```

start

end

Target freq: 1ns

Sampled timing report

Summary on synthesis flow

- Specify your design file (HDL) and the standard cell library
- Correctly constrain your design
  - Target frequency, loading, input driving capability
  - Be conservative
- Compile the design
- Save the synthesized result
  - DDC file: contain all info of the design
  - Netlist (*.v): for the place & route
- Analyze the critical path, area, and power
- A sampled script will be provided to help you

Back annotation and gate-level simulation

- After synthesizing or P&R*, we will get a gate-level netlist (in Verilog)

```
module top(a, b, c, …);
input [31:0] a;
…
wire n1, n2, n3, …;
INV_X1 U210(.A(n123), .ZN(n90));
NAND2_X4 U46(.A(n1), .B(n89), ZN(a[4]));
…
endmodule
```

Module name and interface are unchanged, but the parameterized design would be inferred and fixed

The internal wires

Logic gates from the target library

Sampled gate-level netlist: topology of the synthesized circuit

- We need to verify the functionality of the synthesized results
- 2 more files are required
  - Verilog behavior model of each gates (*.v shipped with standard cell)
  - Delay file containing the gate latency and interconnection (*.sdf from synthesis tool or P&R)

* P&R stands for place and route, which we will talk it later. Therefore, there are 2 phases of gate-level simulation: after synthesizing (post-synthesis) and after P&R (post-layout).

Back annotation and gate-level simulation (cont.)

- Testbench needs to be slightly modified for back-annotation of SDF file

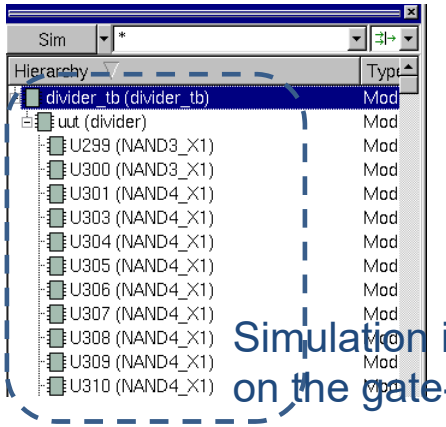
```
// testbench is same as before, except the following sdf_annotate is added
// to the testbench module
initial begin
  $sdf_annotate("sdf_filename.sdf", uut); // where uut is the instance name of our
                                          // design being synthesized and back-annotated
end
```

- VCS compilation should include behavior model of Verilog

```
vcs testbench_with_sdf.v design_netlist.v -debug_all -v path_to_lib.v
```

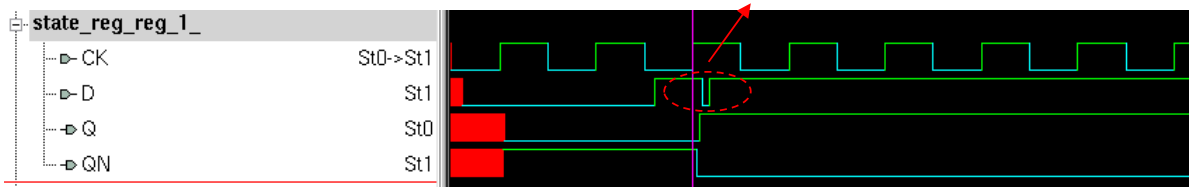
Filepath of Verilog behavior model in the standard cell

- The simulation will contain gate delay



Simulation is conducted on the gate-level

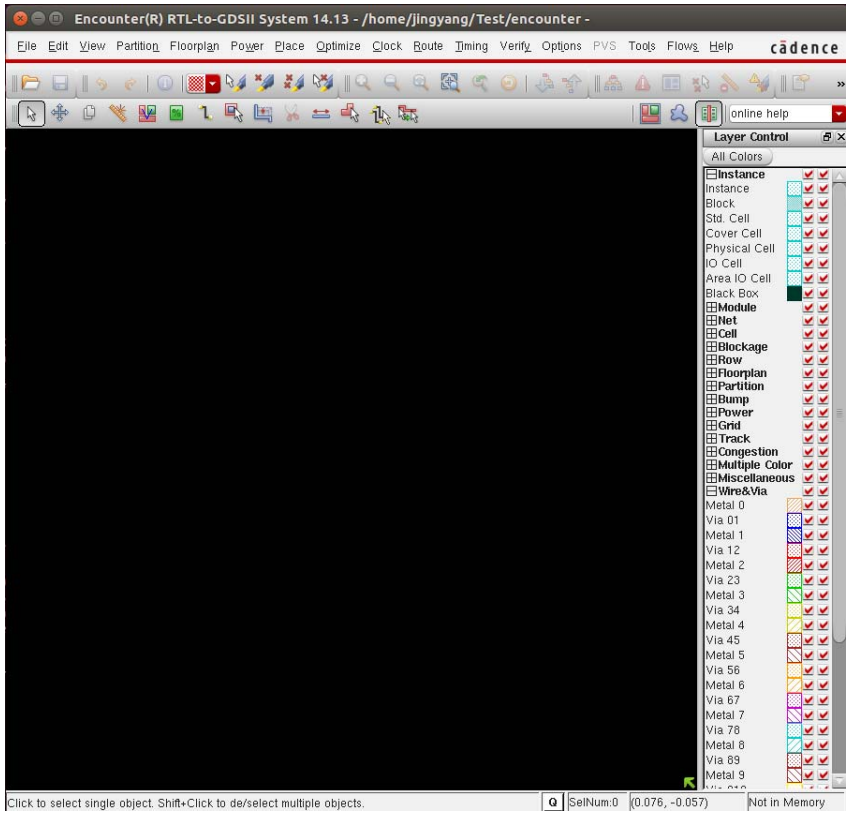
Glitch happens in D



A sampled waveform from DFF in gate-level simulation (there exists some latency from D to Q, i.e. clk2q delay)

Place and route (P&R)

- Use Cadence Encounter Digital Implementation (EDI)
- Unlike the synthesis flow, P&R usually works in the GUI
  - Interactive with the layout
- Working flow
  - Import design (*.v, *.sdc from DC)
  - Import library (*.lib, *.lef)
  - Floorplan
  - Create power ring and stripe
  - Run placement
  - Run trial routing
  - Pre-CTS timing optimization
  - Clock tree synthesis (CTS)
  - Post-CTS timing optimization
  - Run detail routing
  - Post-route timing optimization
- Example: P&R for the LEON processor



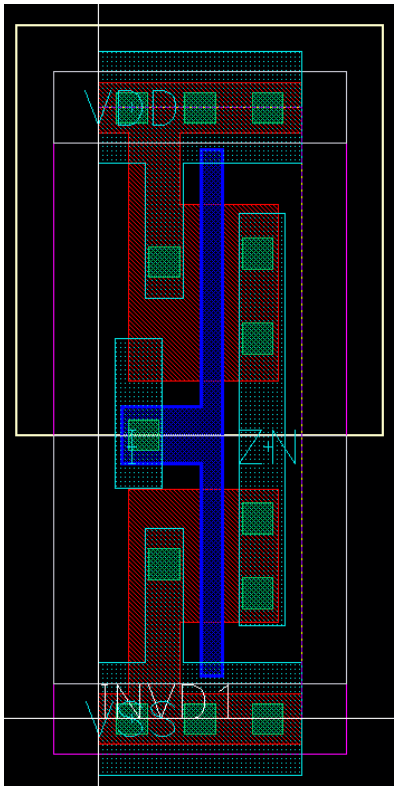
Cadence EDI GUI

Library for P&R

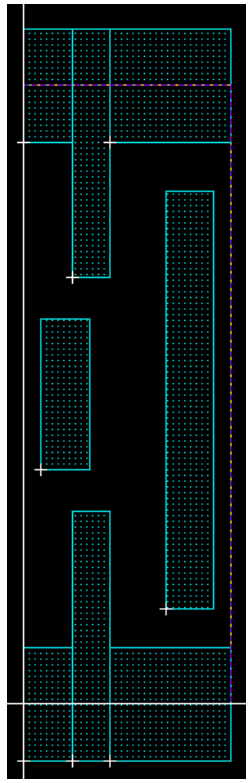
- LEF file: the abstract view of layout
  - Contains the boundary, locations of IO pins, metal info
- Cap table: the capacitance of each metal layer
  - Model the coupling, fringe, area capacitance of each layer

Part of the captable

```
 PROCESS_VARIATION ...
LAYER M1
  MinWidth          0.07000
  MinSpace          0.06500
#  Height           0.37000
  Thickness         0.13000
  TopWidth          0.07000
  BottomWidth       0.07000
  WidthDev          0.00000
  Resistance        0.38000
END
...
BASIC_CAP_TABLE ...
M1
width(um)  space(um) Ctot(Ff/um)  Cc(Ff/um)    Carea(Ff/um) Cfrg(Ff/um)
0.070      0.052     0.1986       0.0723       0.0311       0.0115
0.070      0.065     0.1705       0.0509       0.0311       0.0143
0.070      0.200     0.1179       0.0115       0.0311       0.0319
```



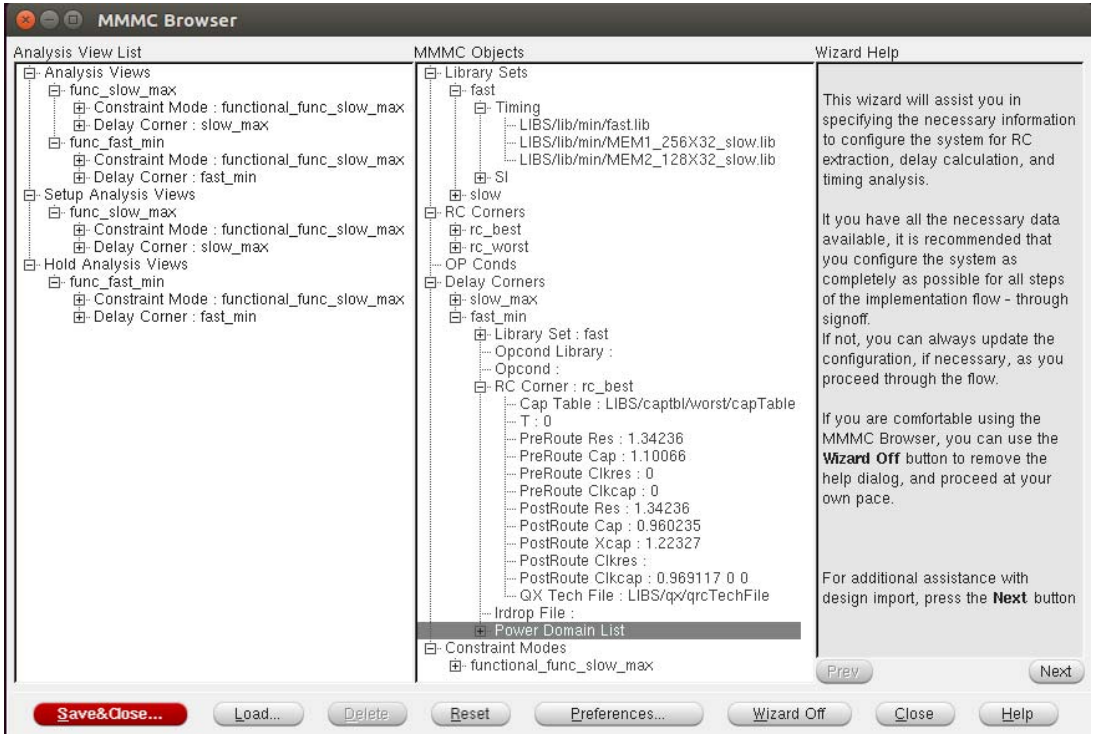
Layout of an inverter



Lef of an inverter

Import the design

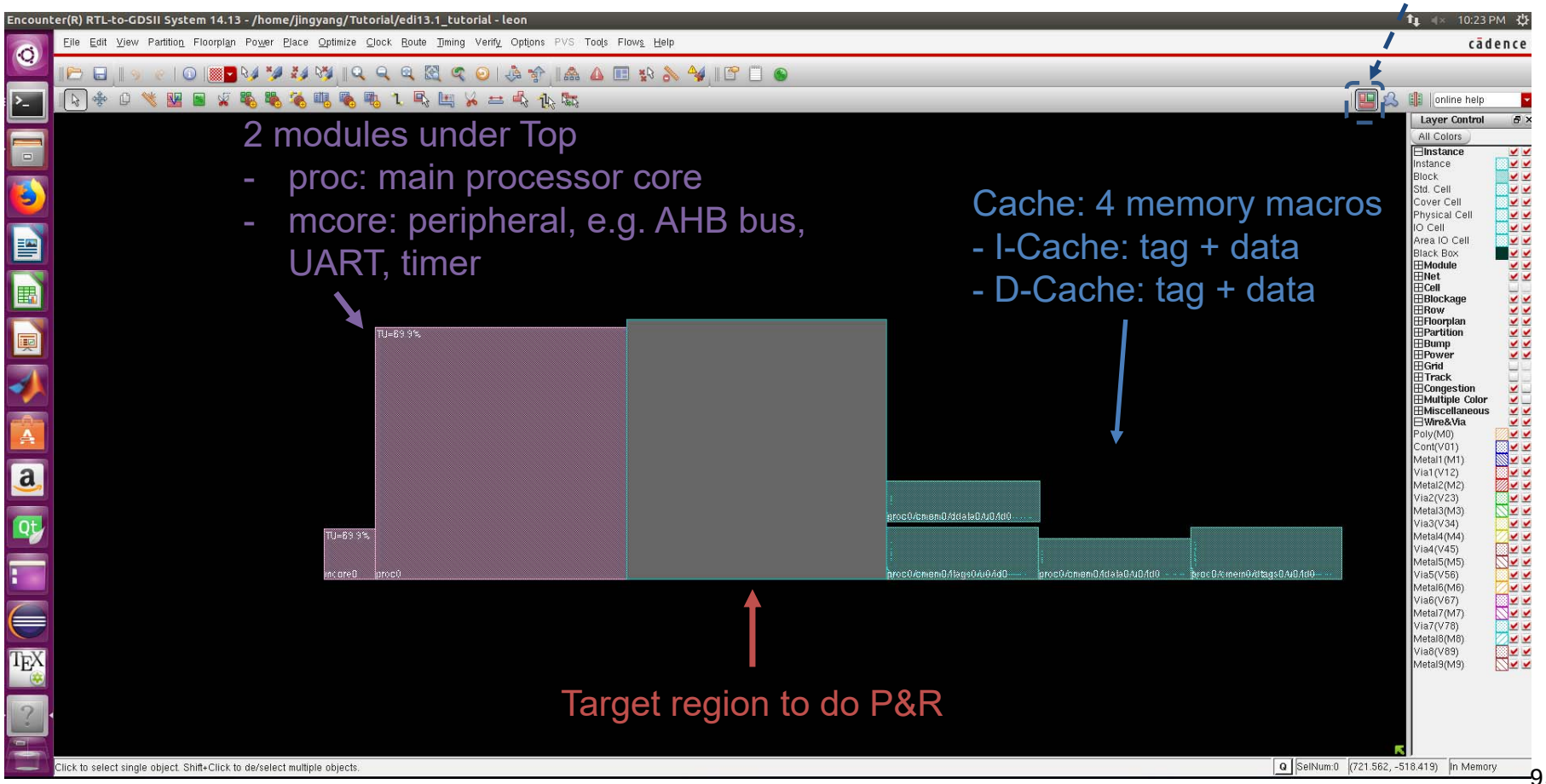
- Use the synthesized Verilog & DC-export constraint file
- MMMC for corner analysis
  - Setup time: slow corner
  - Hold time: fast corner
  - E.g.
    - `tsmc65n_wc.lib` for slow corner
    - `tsmc65n_bc.lib` for fast corner
    - `worst.captbl` for slow corner
    - `best.captble` for fast corner



Mult-Mode Multi-Corner(MMMC) configuration

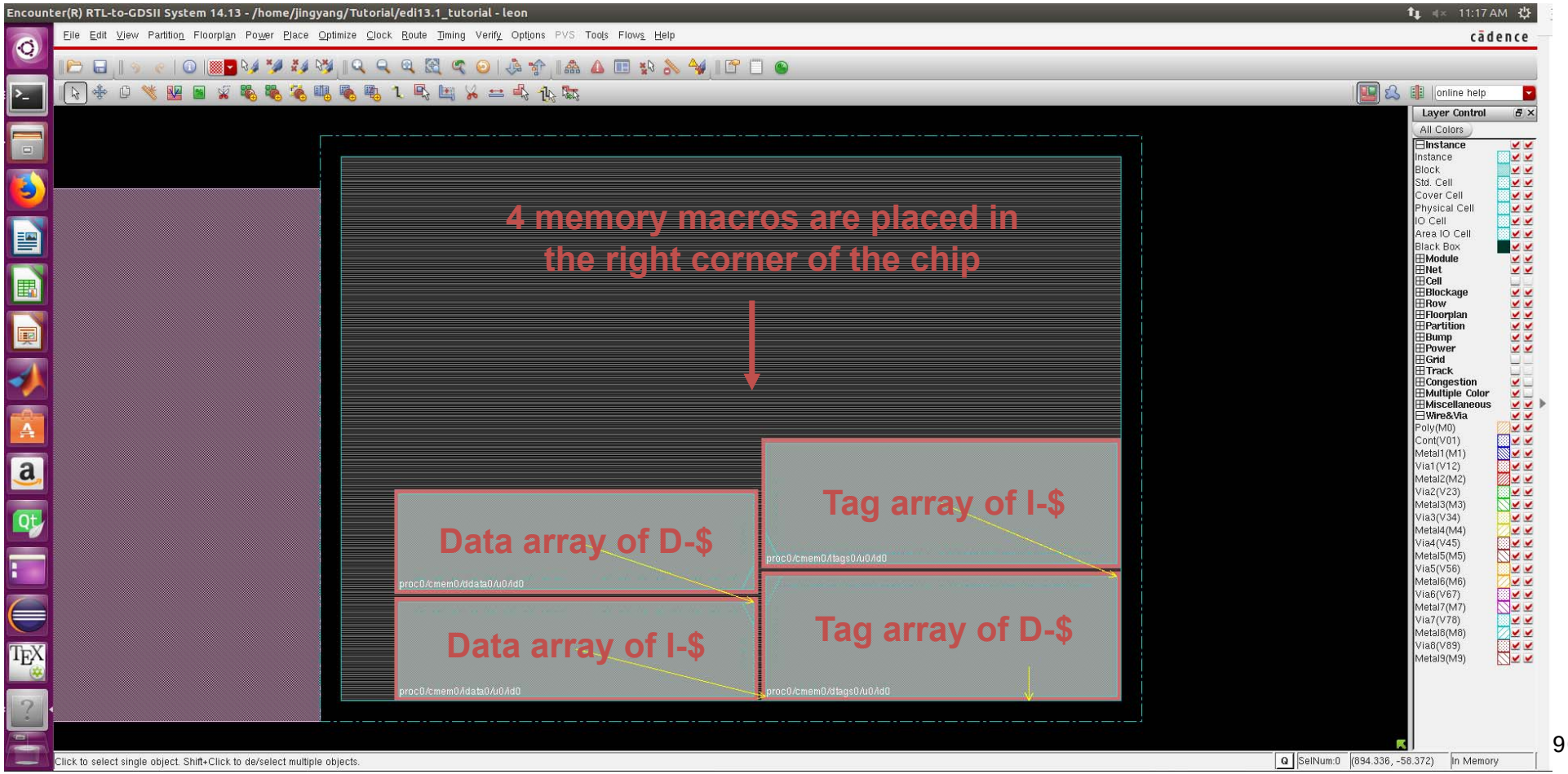
After design import

- Floorplan view of LEON processor



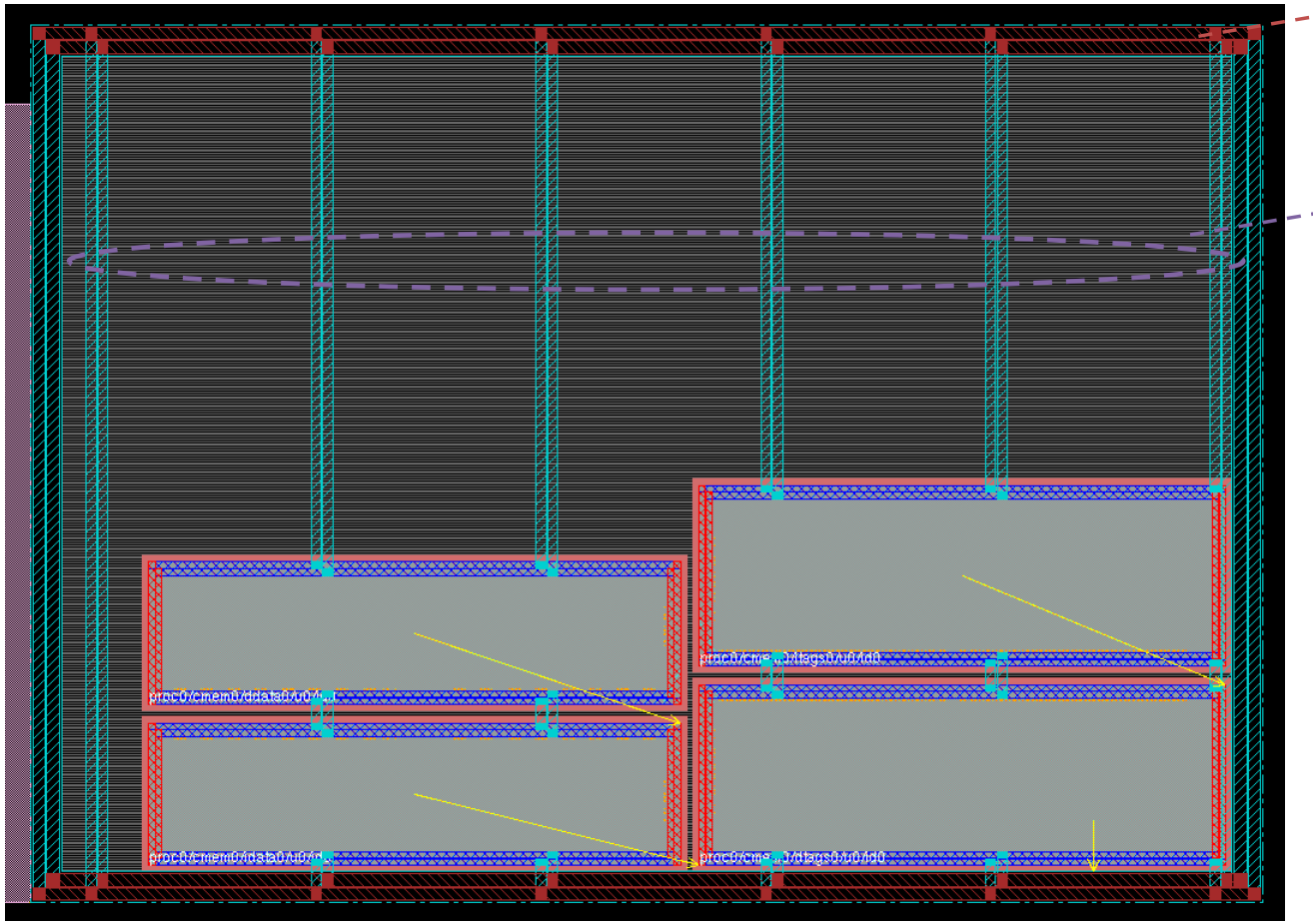
Floorplan

- Specify core ratio, utilization, and place blocks



Power plan: power ring and power stripe

- Create power ring and power stripe around the chip boundary
  - Use top metal for power line

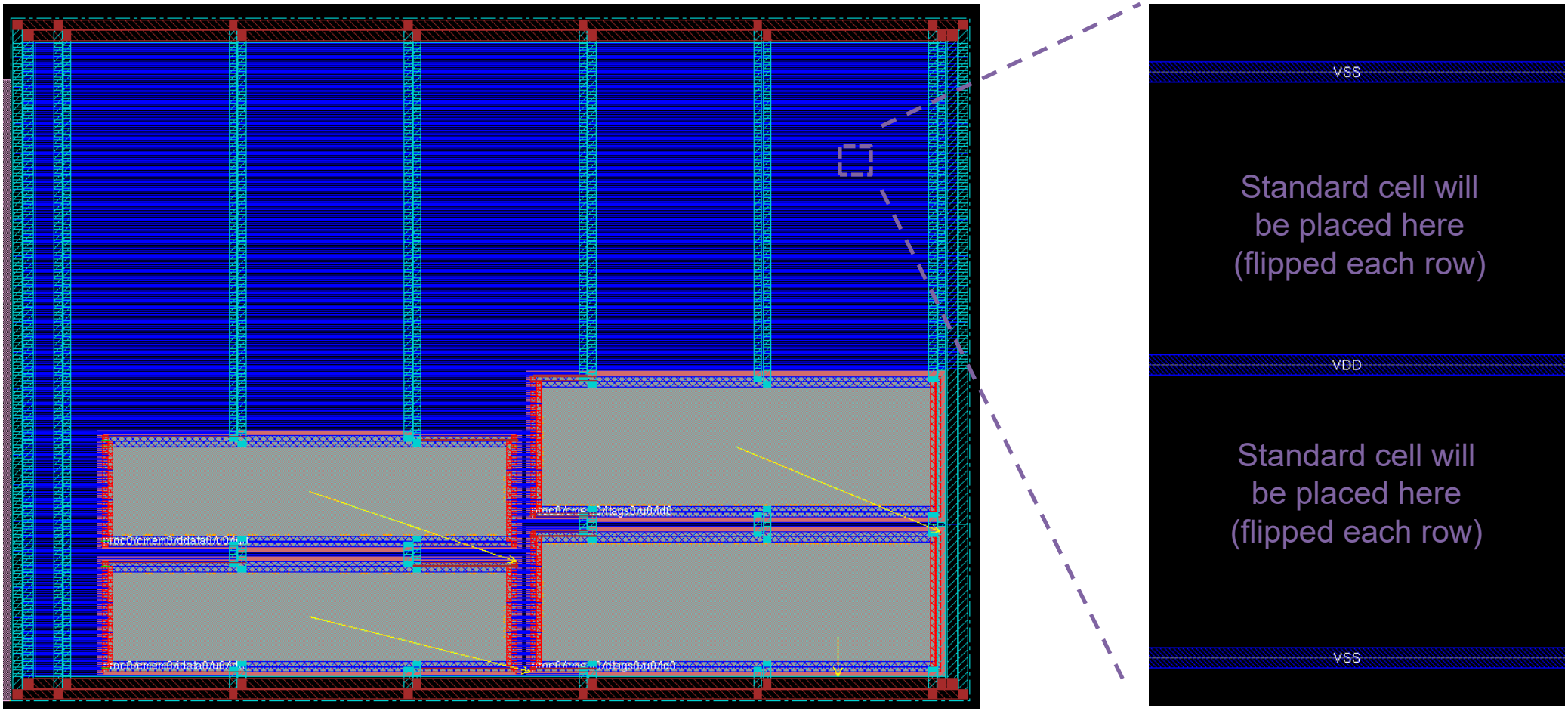


Power ring: VDD & VSS

6 sets of power stripes for a better power delivery

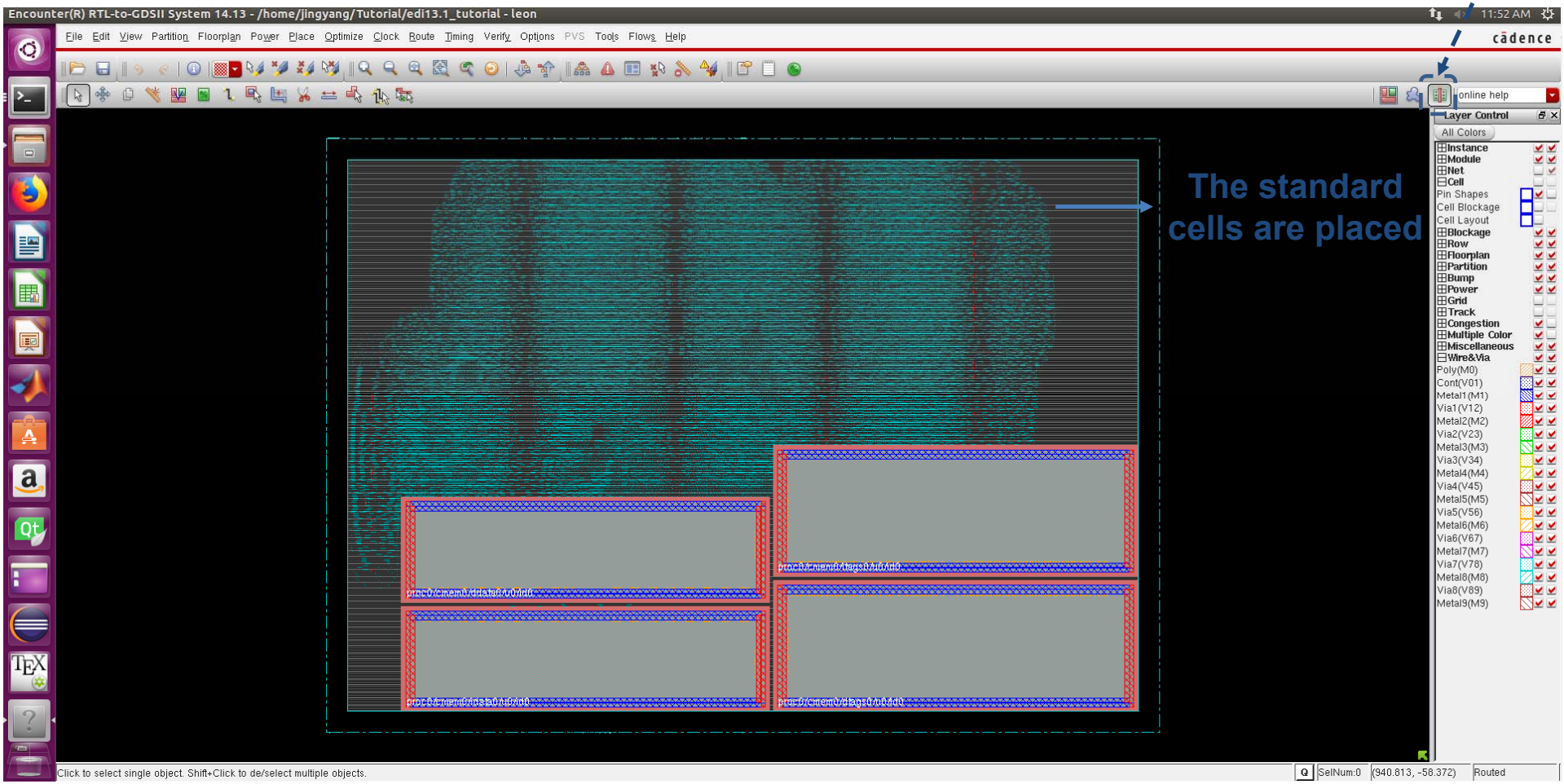
Power plan: special routing

- Add the power rail (Metal1) for the standard cell



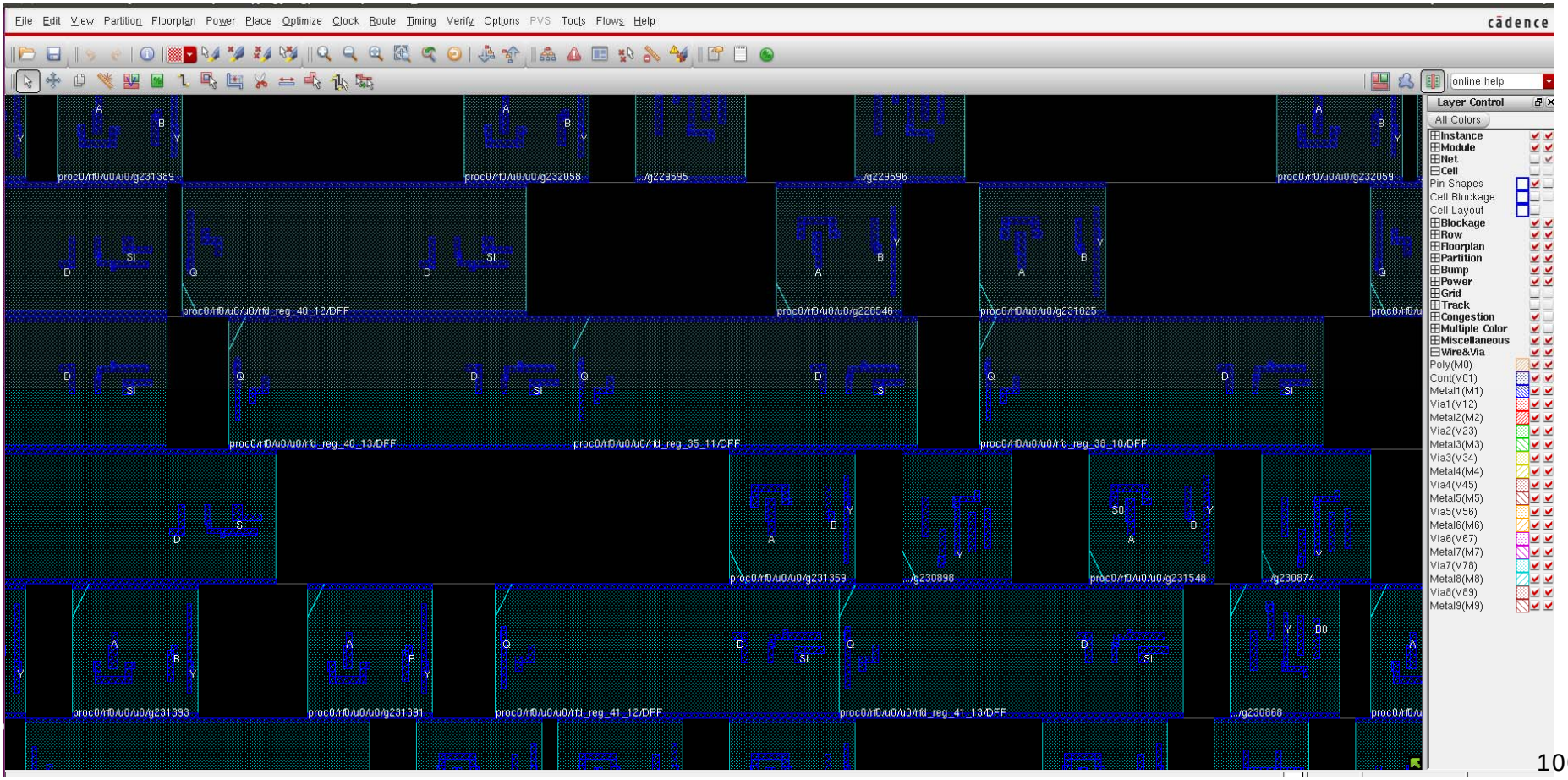
Placement

- Place the standard cell (e.g. inverter, and2) to the design



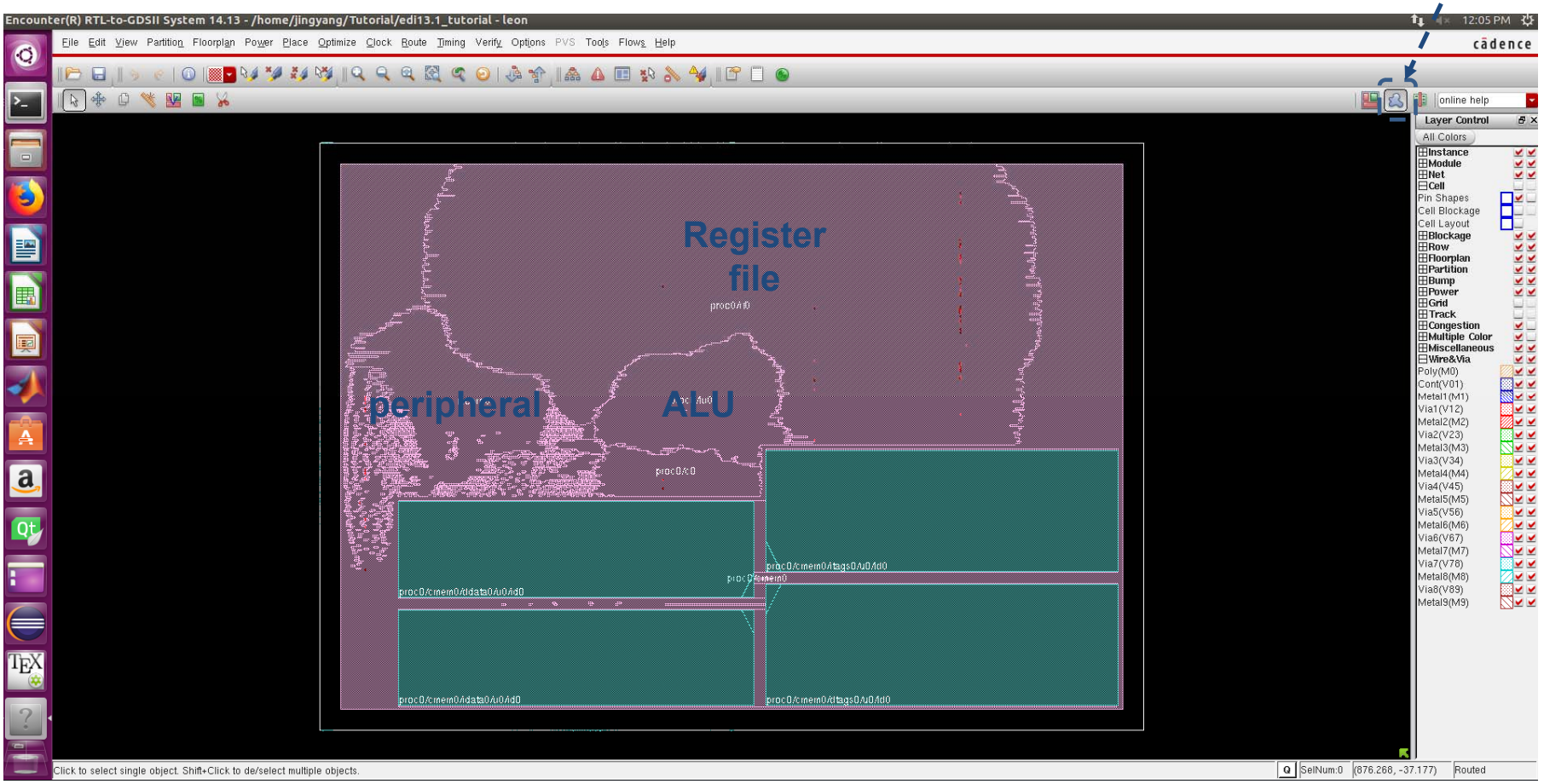
Zoom in of placement results

- The standard cells (in lef view) have been placed between the power rail



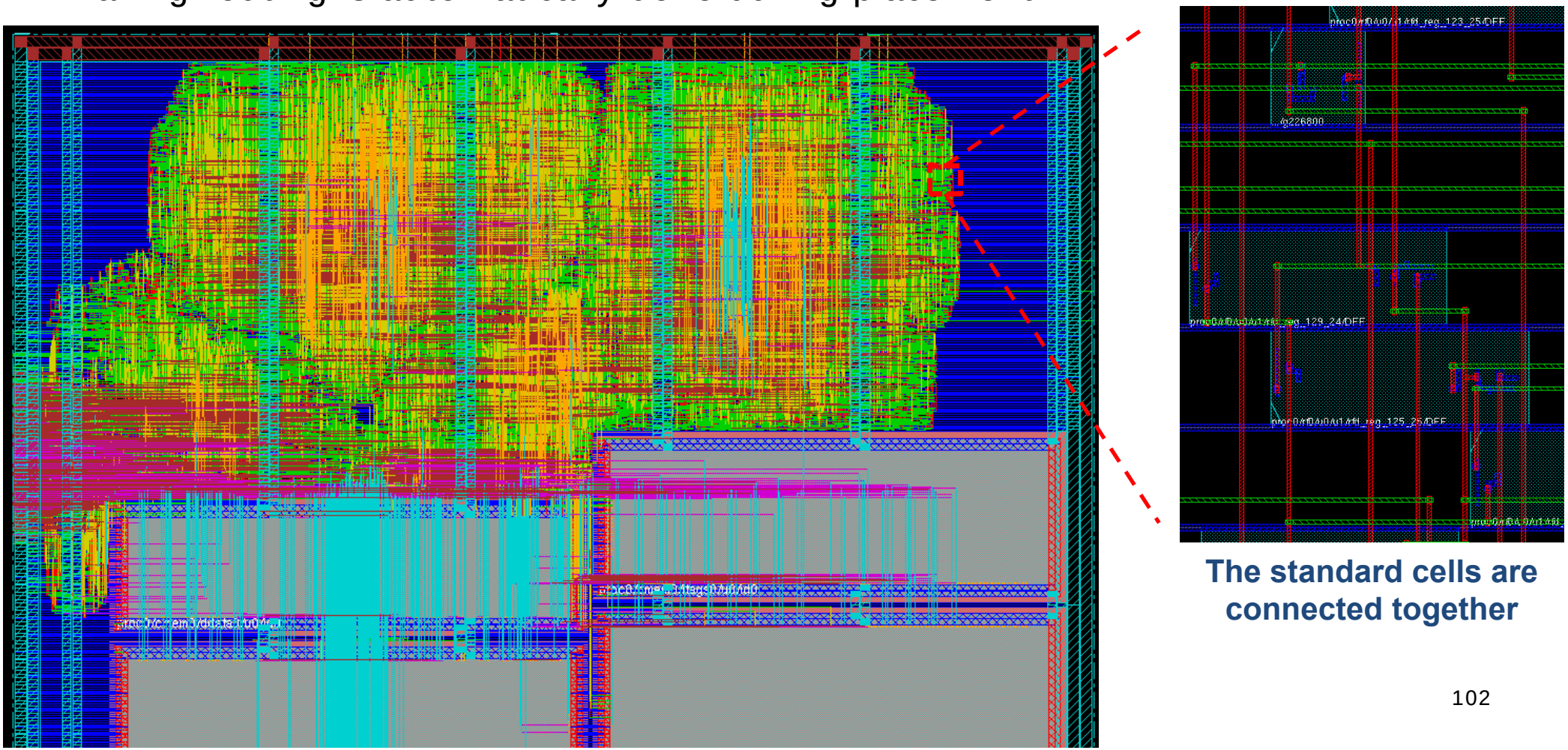
View the design breakdown

- Amoeba View



Run the trail routing

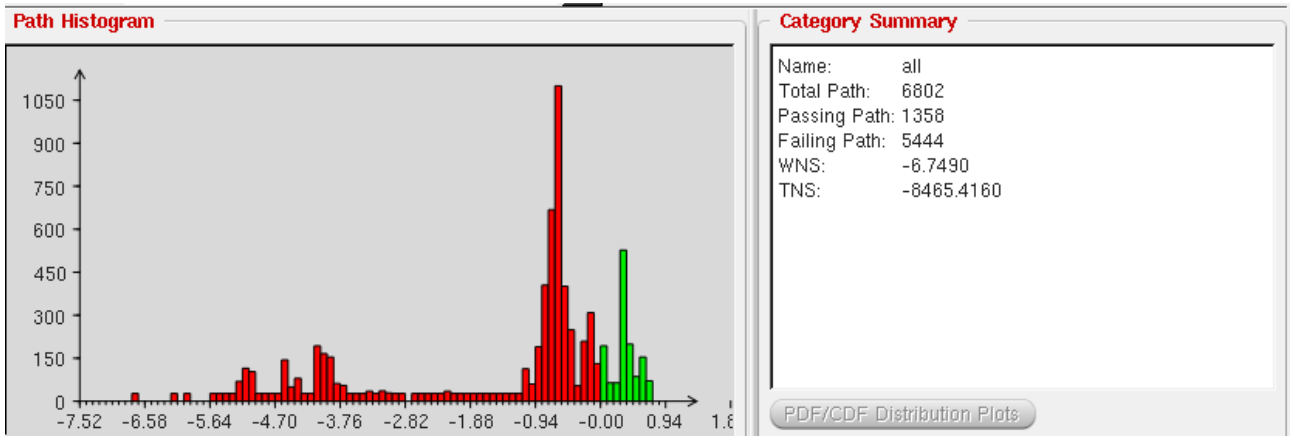
- Trailing routing is automatically done during placement



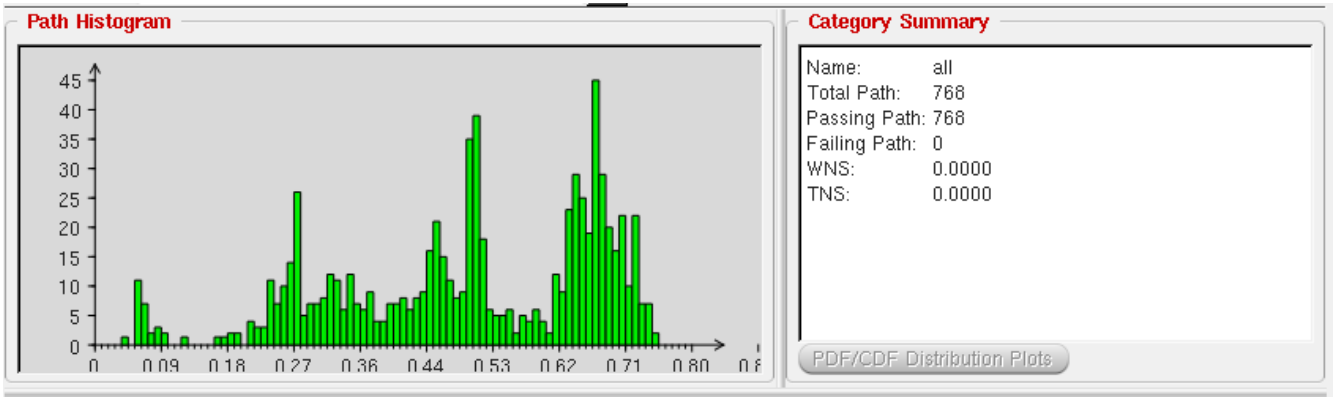
The standard cells are connected together

Pre-CTS timing optimization

- Timing optimization will be conducted at different stages during P&R
- Fix timing violations (setup and hold)



Timing violation before Pre-CTS timing optimization (violation)

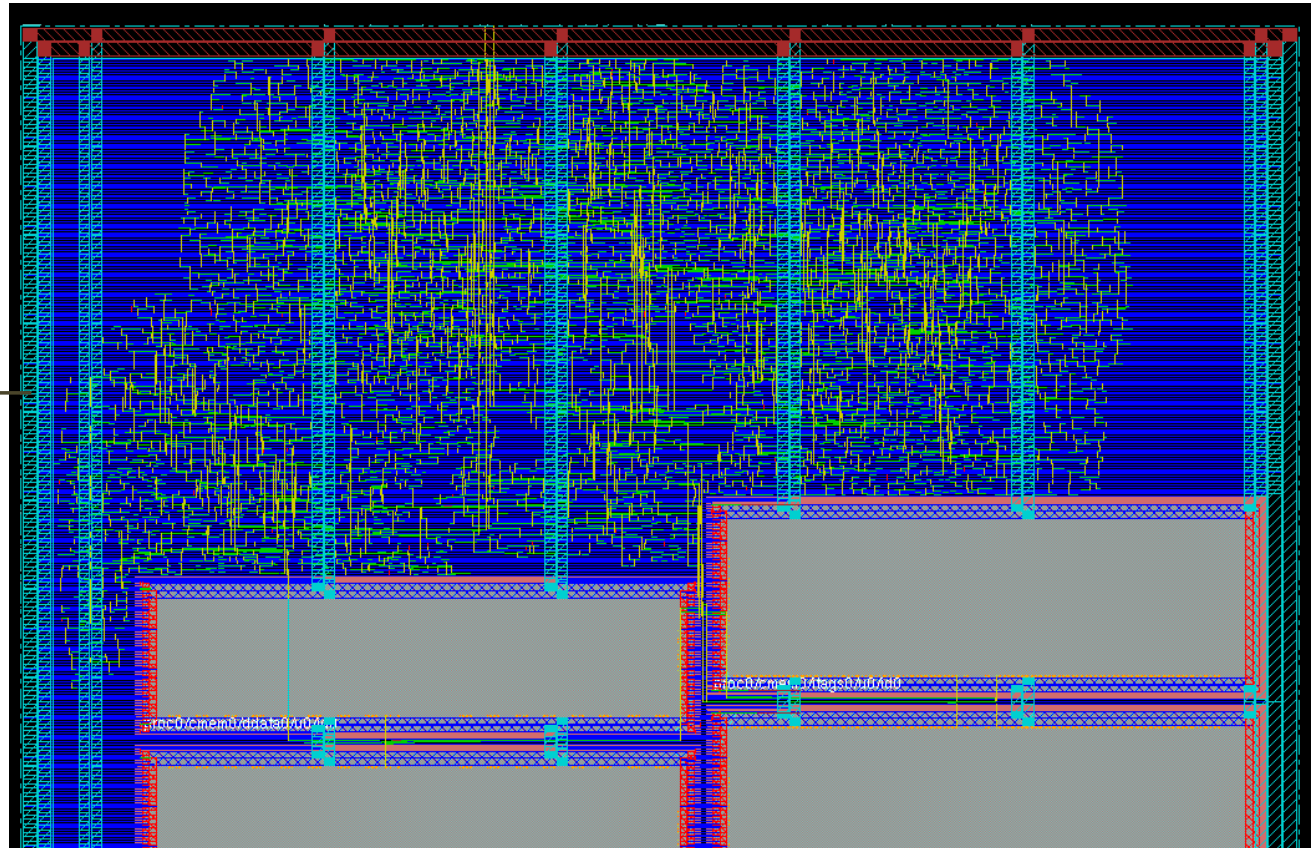


Timing violation after Pre-CTS timing optimization (meet)

Clock tree synthesis (CTS)

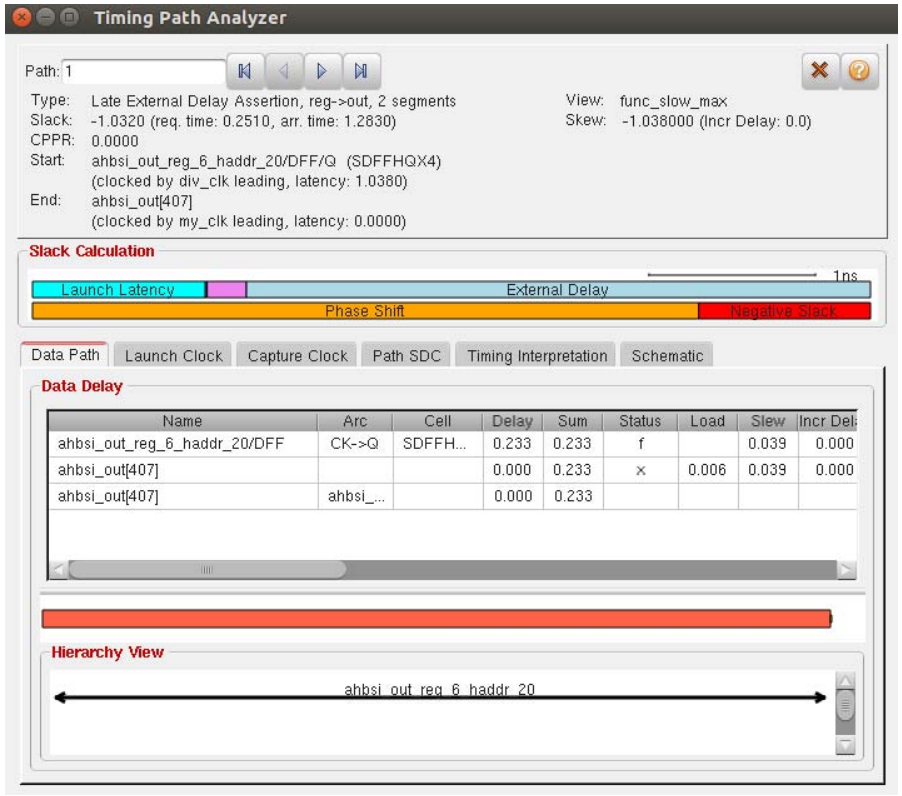
- Design the clock signal distribution network and insert the clock buffers to balance the time delay for DFFs in the design (reduce clock skew)
- Select the desired clock buffer for clock tree

Clock tree is in yellow & green

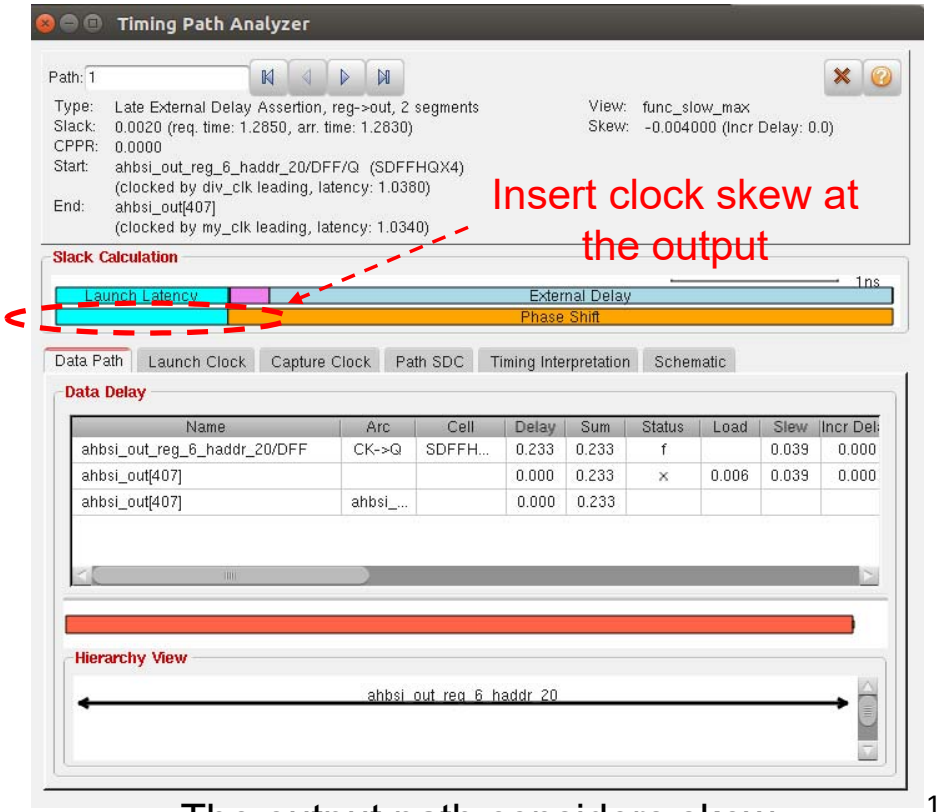


Post-CTS timing optimization

- Fix the time violations after CTS (similar to pre-CTS timing optimization)
  - The clock skew in the output delay calculation needs to be taken into account `update_io_latency`



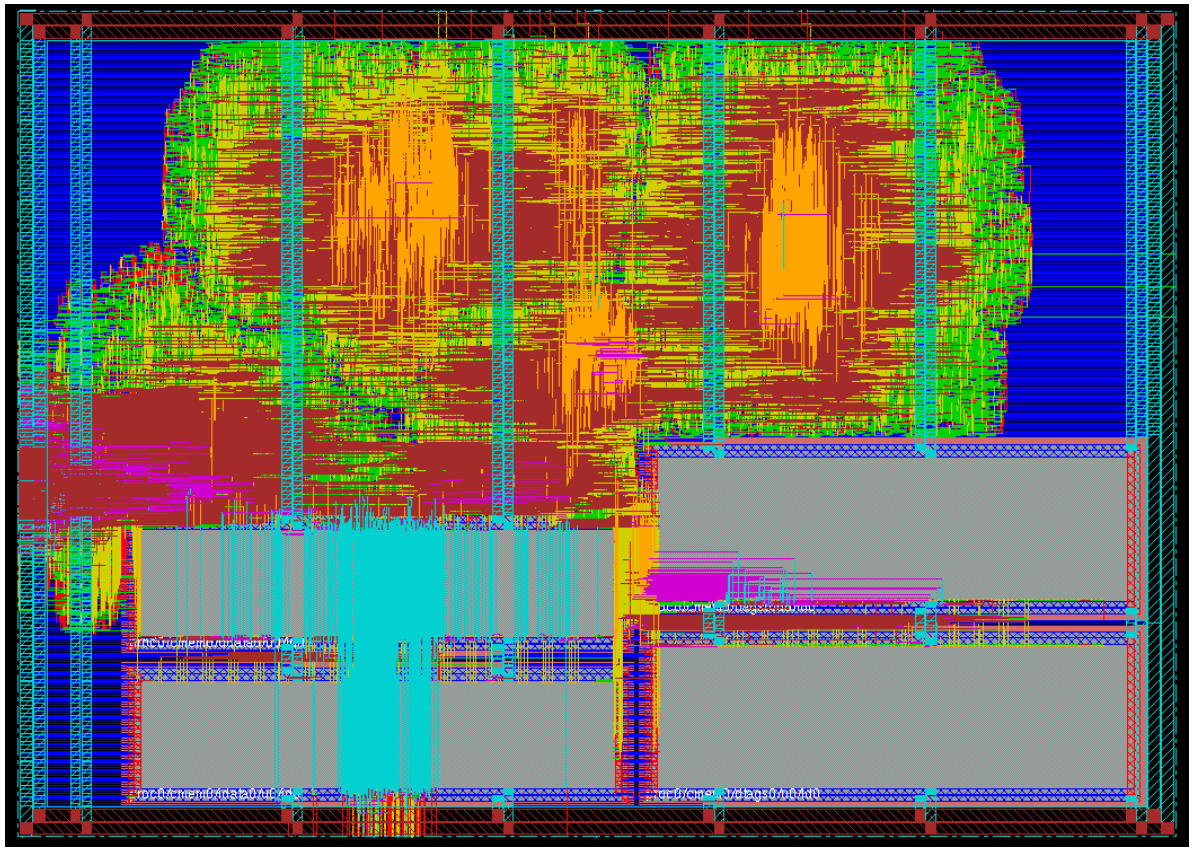
The output path does NOT consider skew



The output path considers skew

Detail routing (NanoRoute)

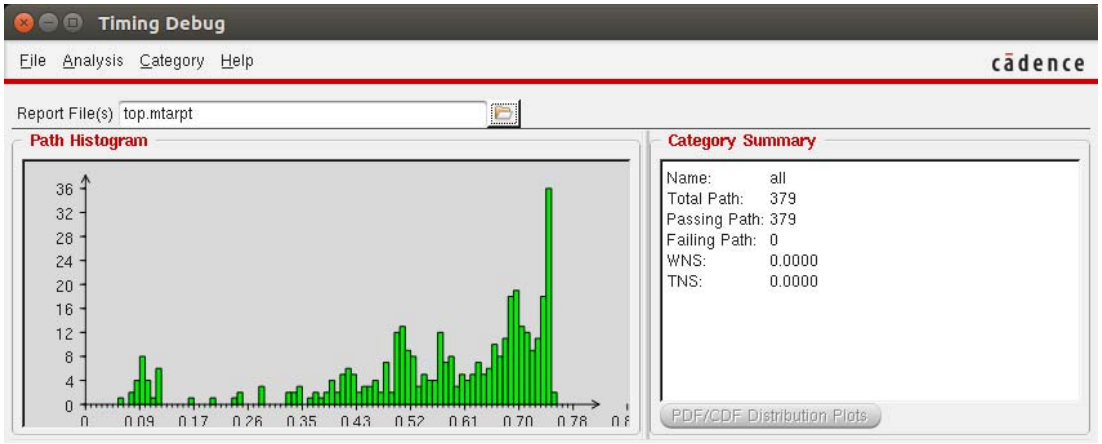
- NanoRoute with timing driven and SI (signal integrity) driven
  - Time driven: meet timing requirement
  - SI driven: prevent the crosstalk



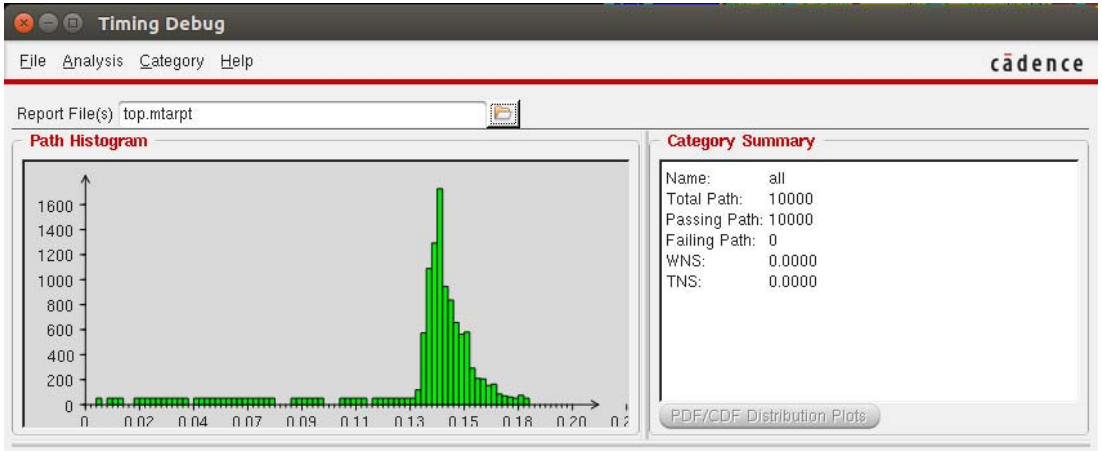
Final LEON processor layout

Post-route timing optimization

- Ensure that setup and hold meet the timing requirements



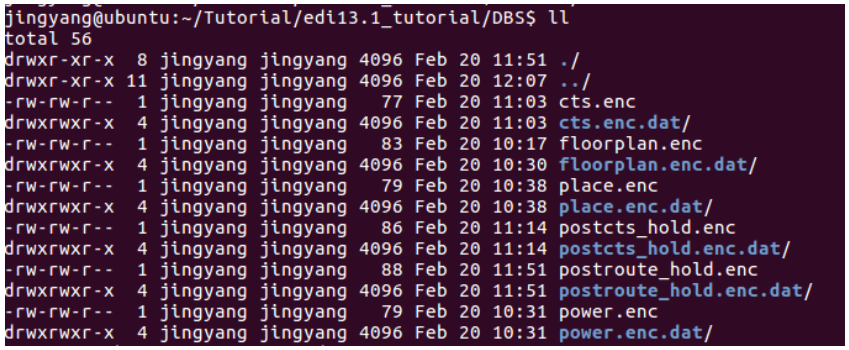
Setup timing report (slow corner)



Hold timing report (fast corner)

Summary on P&R

- Follow the standard flow to run the P&R
  - Import design & library
  - Floorplan
  - Power plan
  - Placement
  - Pre-CTS timing optimization
  - CTS
  - Post-CTS timing optimization
  - NanoRoute
  - Post-route timing optimization



Save the checkpoint after each step

- Make sure there is no timing violation in setup time and hold time

Thank you

The path for the tools

- On UST server, the tools are installed under `/usr/eelocal/`
  - For synopsys tools, under `synopsys` folder
  - For cadence tools, under `cadence` folder
- There are multiple versions of tools (recommend to use latest version)
  - VCS2014: `/usr/eelocal/synopsys/vcs_mx-vi2014.03-2`
  - DC2013: `/usr/eelocal/synopsys/syn-vi2013.12-sp5-5`
  - EDI14: `/usr/eelocal/cadence/edi142/`
- Set the running environment of the software
  - Source the corresponding `.cshrc` under each folder
    - E.g. VCS2014: `source /usr/eelocal/synopsys/vcs_mx-vi2014.03-2/.cshrc`
- Tools to connect the server
  - SSH
  - MobaXterm: https://mobaxterm.mobatek.net/download.html
  - Use VNC for GUI