

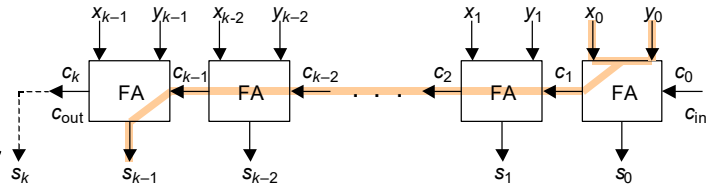
HDL HW1: Adder and Synthesis

HDL HW1 Outlines

- describe a 32-bit adder using 3 different modeling methods
 - dataflow, behavior, structure
 - repeated module instantiation in structural modeling
 - both un-stored output and stored output
- prepare testbench for RTL simulation
- synthesize into gate netlists using Synopsys Design Compiler (DC)
- post-synthesis gate-level simulation
- Lab slides for EDA tools
 - Verilog simulators (vcs, nc-Verilog, modelsim, ...)
 - Design compiler

adders in 3 modeling levels

- structure (*adder_structure*)
 - describe the structure of the adder
 - composed of full-adder (FA) sub-modules
 - ✓ use Verilog built-in gate primitives (**or**, **and**, **xor**, ...) for FA design
 - use duplicated module instantiation
 - ✓ use **generate** loop for repeated instantiation of same modules
- dataflow (*adder_dataflow*) *assign C = a + b*
 - use **assign** with operators to describe the adder
- behavior (*adder_behavior*)
 - describe the adder inside **always** block
- use registers to store the output of the adder
 - use **always @ (posedge clk)** ... to implement the registers
 - might need module instantiation
 - ✓ module instance signal connection by order or by name



RCA with **generate** loop

- use **generate** loop to duplicate module instances
 - e.g. in FA-cascaded RCA
- or simple
 - **array of instances**

```
// structural level modeling
module RCA (sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;

wire c1, c2, c3;
// construct 4-bit adder fulladd4 from
// previously defined module fulladd

fulladd fa0 (sum[0], c1, a[0], b[0], c_in);
fulladd fa1 (sum[1], c2, a[1], b[1], c1);
fulladd fa2 (sum[2], c3, a[2], b[2], c2);
fulladd fa3 (sum[3], c_out, a[3], b[3], c3);

endmodule
```

```
module RCA_generate (sum, c_out, a, b, c_in);
parameter N=4;
output [N-1:0] sum;
output c_out;
input [N-1:0] a, b;
input c_in;
wire [N:0] c;

assign c[0]=c_in;

genvar i;
generate
  for (i=0; i<=N-1; i=i+1)
    begin: FA_loop // block named "FA_loop"
      fulladd fa (sum[i], c[i+1], a[i], b[i], c[i]);
      // fulladd FA_loop[0].fa ( sum[0], c[1], a[0], b[0], c[0] );
      // fulladd FA_loop[1].fa ( sum[1], c[2], a[1], b[1], c[1] );
      // fulladd FA_loop[2].fa ( sum[2], c[3], a[2], b[2], c[2] );
      // fulladd FA_loop[3].fa ( sum[3], c[4], a[3], b[3], c[3] );
    end
endgenerate

// fulladd FA_loop[0:3] (sum[0:3], c[1:4], a[0:3], b[0:3], c[0:3]);

assign c_out = c[N];

endmodule
```

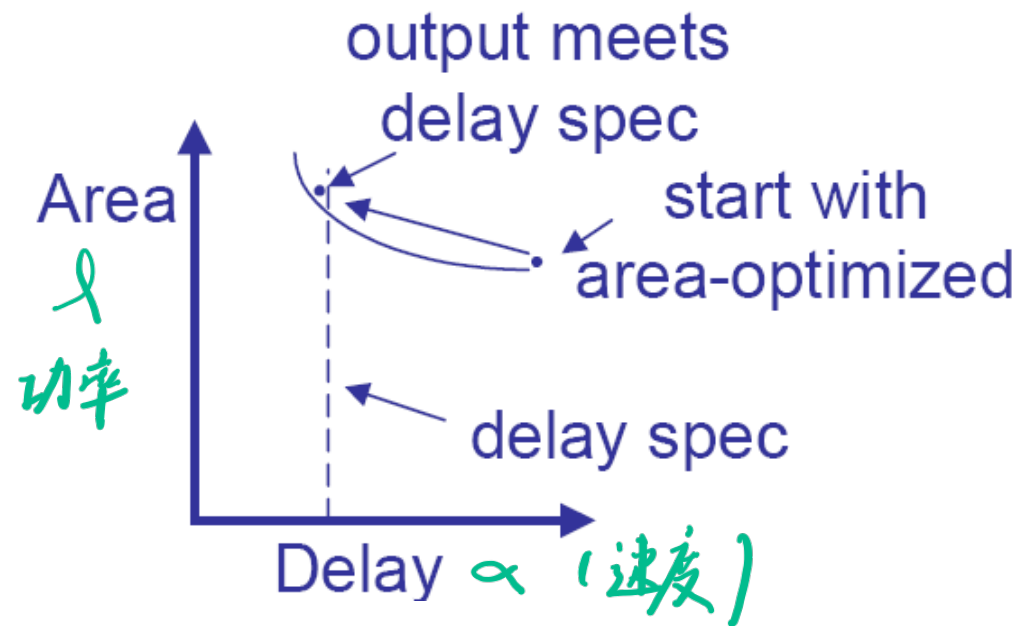
logic synthesis (RTL -> gate netlists)

- ASIC

- use Synopsys Design Compiler (DC) to synthesize the RTL adder description into gate-level
- try different synthesis constraints
 - ✓ area-optimized (面积最小)
 - ✓ delay-optimized (速度最快)
 - ✓ in-between
- trade-off between area and delay

- FPGA

- flexible
- quick prototype
- in HW2



summary table

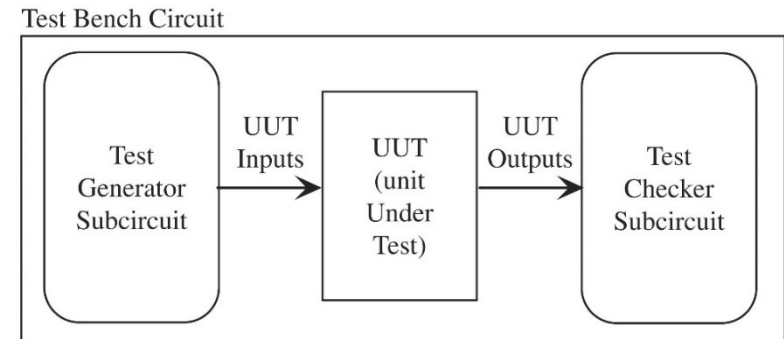
- collect all synthesis results in a table for easy comparision

		Area (um ²)			Delay	Power (W)		
		CL	SL	Total	(ns)	dynamic	leakage	total
adder_structure	delay							
	area							
	between							
adder_structure_reg	delay							
	area							
	between							
adder_dataflow	delay							
	area							
	between							
adder_dataflow_reg	delay							
	area							
	between							
adder_behavior	delay							
	area							
	between							
adder_behavior_reg	delay							
	area							
	between							

Supplement: Testbench

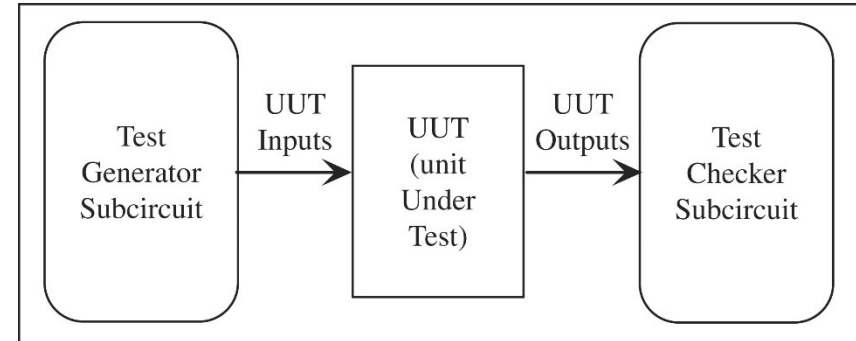
Testbench

- instantiate hardware design block
 - map input/output pins
- specify inputs
 - could directly in testbench, or
 - read from test vector files, or
 - randomly generate inputs
- apply test input test patterns and get output results
 - connect the specified inputs to the instantiated hardware
 - obtain the simulation outputs
- compare simulation outputs with expected results
 - expected could be from test vector file, or
 - generated using non-synthesizable codes in testbench



Test bench example

Test Bench Circuit



Name	Value	Signal
R= a	1	
R= b	1	
R= borrow_in	1	
res	1	
borrow_out	1	
R= expected_res	1	
R= expected_bo	1	
R= error_count	00000000	

Console integer error_count

- Simulation has been initialized
- Selected Top-Level: test_sub (test_sub)
- : Starting the test.
- : Done with test.
- : There were 0 errors.
- RUNTIME: RUNTIME_0068 tb_full_sub.v (31): \$finish called
- KERNEL: stopped at time: 90 ns

```
`timescale 1ns/100ps // time unit = 1ns, precision = 100ps
```

```
module test_sub();
```

```
  reg a, b, borrow_in; // inputs to full_sub_struct
```

```
  wire res, borrow_out; // outputs of full_sub_struct
```

```
  reg expected_res, expected_bo; // expected results
```

```
  integer error_count; // number of errors
```

```
// make connections to the unit under test (UUT)
```

```
full_sub_struct U1 (res, borrow_out, a, b, borrow_in);
```

```
initial begin // one-time execution block
```

```
  $display("Starting the test.");
```

```
  error_count = 0;
```

```
  {a, b, borrow_in} = 3'b000;
```

```
end
```

```
always begin // repeated execution block
```

```
  #10; // wait for 10 * 1ns = 10ns
```

```
  {expected_bo, expected_res} = a - b - borrow_in;
```

```
  if ({expected_bo, expected_res} != {borrow_out, res}) begin
```

```
    $display("Expected (%b, %b) != actual (%b, %b) at time %0t.",
```

```
        expected_bo, expected_res, borrow_out, res, $time);
```

```
    error_count = error_count + 1;
```

```
  end
```

```
  if ({a, b, borrow_in} == 3'b111) begin
```

```
    #10 $display("Done with test.");
```

```
    $display("There were %0d errors.", error_count);
```

```
    $finish;
```

```
  end
```

```
  else // compute next test vector
```

```
    {a, b, borrow_in} = {a, b, borrow_in} + 1;
```

```
end // of main always block
```

```
endmodule
```

instantiate design, read test patterns, apply test patterns, and get results

```
module sillyfunction (input a, b, c, output y);  
wire y = ~a & ~b & ~c | a ^ ~b ^ ~c | a & ~b & c;  
endmodule  
  
module testbench ();  
...  
reg [3:0] testvectors[10000:0];  
...  
sillyfunction dut (a, b, c, y); // instantiate device under test  
...  
initial begin  
    // system task $readmemb reads binary data from file  
    $readmemb ("example.tv", testvectors); ...  
end  
...  
always @(posedge clk) begin  
    #1; {a, b, c, yexpected} = testvectors[vectornum];  
    // assign inputs a, b, c; get output y ...  
end ...
```

```
// testvector  
// fine namej  
// example.tv
```

```
000_1  
001_0  
010_0  
011_0  
100_1  
101_1  
110_0  
111_0
```

Random Number Generation (**\$random**)

- Seed can be either **reg**, **integer**, or **time** variable
- \$random** returns a 32-bit *signed* integer

```
module test;
integer r_seed;
reg [31:0] addr; // input to ROM
wire [31:0] data; // output from ROM
reg [23:0] rand1, rand2;
...
ROM rom1 (data, addr);
...
initial r_seed = 2; // arbitrarily define seed as 2, better use different seed values
always @(posedge clock)
    addr = $random (r_seed); // generate random numbers of signed integer
    rand1 = $random % 60; // generate random number between -59 and 59
    rand2 = {$random} % 60; // random number between 0 and 59
...
// check output of ROM against expected results
...
endmodule
```

Types of Testbench

- access files for inputs and outputs
 - \$fscanf, \$readmemb, \$fdisplay, ...
 - need other programs (e.g., C, Python) to generate the input files of test patterns
 - need other programs to compare the simulation results in the output file
 - the previous example read test patterns and expected results from a file, and make comparison in the testbench
- generate inputs in testbench module
 - randomly generate input test patterns (e.g., \$random, ...)
 - could use non-synthesizable codes to generate expected results (e.g., real x, y, z ' assign z=x+y; // floating-pt. add)
 - compare simulation results with expected ones

```

`timescale 1ns/10ps
`define CYCLE 50.0
`define DATA_NUM 100
`define FILE_A ".a.txt"
`define FILE_B ".b.txt"
`define FILE_D ".d.txt"

```

```

module testbench();
    integer file_a, file_b, file_d;

```

```

    reg CLK = 0;
    reg RST = 0;
    reg [31:0] data_a [0:`DATA_NUM - 1];
    reg [31:0] data_b [0:`DATA_NUM - 1];
    reg [31:0] input_a;
    reg [31:0] input_b;
    wire [31:0] outcome;
    reg [31:0] answer;

```

```

    FXP_adder test_module( .a(input_a), .b(input_b), .d(outcome));

```

//Post_sim 使用 · 在nc_post_syn.f 中define SDF_FILE與SDF_FILE路徑

```

`ifdef SDF_FILE

```

```

    initial $sdf_annotate(`SDF_FILE, test_module);

```

```

`endif

```

// 設定clock訊號

```

always begin #(`CYCLE/2) CLK = ~CLK; end

```

```

integer i, flag=0, error=0, garbage;

```

//將file_a與file_b中資料存入data_a與data_b陣列中

```

initial begin

```

```

    file_a = $fopen( `FILE_A , "r"); // file_a=$fopen("a.txt", "r");
    file_b = $fopen( `FILE_B , "r"); // file_b=$fopen("b.txt", "r");
    file_d = $fopen( `FILE_D , "w"); // file_d=$fopen("d.txt", "w");
    for (i = 0; i < `DATA_NUM; i=i+1)
        begin

```

```

            garbage = $fscanf(file_a, "%X", data_a[i]);

```

```

            garbage = $fscanf(file_b, "%X", data_b[i]);

```

```

        end

```

```

    end

```

testbench.v (1/2)

```

module FXP_adder(
    input [31:0] a,
    input [31:0] b,
    output [31:0] d );
    assign d = a + b;
endmodule

```

```

initial begin

```

//File approach

```

    $display("-----\n");

```

```

    $display("-      FXP_ADDER_USE_FILE      -\n");

```

```

    $display("-----\n");

```

```

    CLK = 0;

```

```

    RST = 1;

```

```

    #(`CYCLE*2);

```

```

    RST = 0;

```

```

    for(i=0; i<`DATA_NUM; i=i+1)

```

```

    begin

```

```

        input_a = data_a[i];

```

```

        input_b = data_b[i];

```

```

        answer = $signed(input_a) + $signed(input_b);

```

```

        #(`CYCLE);

```

```

        $fwrite(file_d, "%X\n", outcome);

```

```

        if(answer != outcome)

```

```

        begin

```

```

            error = error+1;

```

```

            if(1 || flag==0)

```

```

            begin

```

```

                $display("-----\n");

```

```

                $display("Output error at #%d\n", i+1);

```

```

                $display("The input A is   : %X\n", input_a);

```

```

                $display("The input B is   : %X\n", input_b);

```

```

                $display("The answer is    : %X\n", answer);

```

```

                $display("Your module output: %X\n", outcome);

```

```

                $display("-----\n");

```

```

                flag = 1;

```

```

            end //if flag

```

```

        end //if

```

```

    end //for

```

```

    if(flag==1)//if wrong

```

```

    begin

```

```

        $display("Total %4d error in %4d testdata.\n", error, i);

```

```

        $display("-----\n");

```

```

    end//if

```

```

    else

```

```

begin//if right
    $display("-----\n");
    $display("All testdata correct!\n");
    $display("-----\n");
end//else

```

testbench.v (2/2)

```

//random approach
$display("-----\n");
$display("-      FXP_ADDER_USE_RANDOM      -\n");
$display("-----\n");

```

```

flag=0;
error=0;
CLK = 0;
RST = 1;
#(`CYCLE*2);
RST = 0;
for(i=0; i<`DATA_NUM; i=i+1)
begin
    input_a = $random % 2**31 ; //產生介於 -(2^31)-1到(2^31)-1的數
    input_b = $random % 2**31 ;
    answer = $signed(input_a) + $signed(input_b);
    #(`CYCLE);
    if(answer != outcome)
    begin
        error = error+1;
        if(1 || flag==0)
        begin
            $display("-----\n");
            $display("Output error at #%d\n", i+1);
            $display("The input A is   : %X\n", input_a);
            $display("The input B is   : %X\n", input_b);
            $display("The answer is    : %X\n", answer);
            $display("Your module output: %X\n", outcome);
            $display("-----\n");
            flag = 1;
        end //if flag
    end //if
end //for

```

```

if(flag==1)//if wrong
begin
    $display("Total %4d error in %4d testdata.\n", error, i);
    $display("-----\n");
end//if
else
begin//if right
    $display("-----\n");
    $display("All testdata correct!\n");
    $display("-----\n");
end//else

$fclose(file_a);
$fclose(file_b);
$fclose(file_d);
$finish;
end

endmodule

```

Input a.txt

Input b.txt

output d.txt

```

1 558b4567
2 51bc9869
3 5a30dc51
4 3568944a
5 318e1f29
6 3e9b58ba
7 373141f2
8 5ce2a9e3
9 485f007c
10 29200854
11 2116231b
12 2890cde7

```

```

1 397b23c6
2 52b34873
3 2cc95cff
4 50d558ec
5 43687ccd
6 47fed7ab
7 40b71efb
8 5a45e146
9 4dd062c2
10 46b127f8
11 2f96e9e8
12 536f438d

```

```

1 08f06692d
2 0a46fe0dc
3 086fa3950
4 0863ded36
5 074f69bf6
6 0869a3065
7 077e860ed
8 0b7288b29
9 0962f633e
10 06fd1304c
11 050ad0d03
12 07c001174

```

```

... // instantiate hardware
FXP_adder test_module ( .a(input_a), .b(input_b), .d(outcome));
... // read test patterns from files
file_a = $fopen( "a.txt" , "r");
file_b = $fopen( "b.txt" , "r");
file_d = $fopen("c.txt" , "w");
for (i = 0; i < 100; i=i+1) begin ...
    garbage = $fscanf (file_a, "%X", data_a[i]); // hexa-decimal format
    garbage = $fscanf (file_b, "%X", data_b[i]); ... end
... // compute expected answer, get hardware outputs and write to a file
for (i=0; i< 100; i=i+1) begin ...
    input_a = data_a[i]; // apply test patterns to hardware inputs
    input_b = data_b[i];
    answer = $signed (input_a) + $signed(input_b);
    $fwrite (file_d, "%X\n", outcome); ... end
... // an alternative of generating input test patterns randomly
for (i=0; i< 100; i=i+1) begin ...
    input_a = $random % 2**31 ; // between  $-2^{31} + 1$  and  $2^{31} - 1$ 
    input_b = $random % 2**31 ;
    answer = $signed (input_a) + $signed (input_b); ... end

```

...

System Task \$

- read from input
 - \$readmemb, \$readmemh
 - \$fscanf, \$fget. \$fread
- write to output
 - \$fopen, \$fclose
 - \$display, \$strobe, \$monitor, \$write
 - \$fdisplay, \$fstrobe, \$fmonitor, \$fwrite
- random generator
 - \$random
- suspend (\$stop) or finish (\$finish) simulation
- conversion functions
 - \$signed, \$unsigned

Compiler Directives ` (back quote)

- **`define**
 - define text macro for later text macro substitution

```
`define WORD_SIZE 32  
// used as `WORD_SIZE in the code such as  
// wire [`WORD_SIZE-1:0] a, b, c;
```
- **`include**
 - include another Verilog file, e.g.,

```
`include header.v //include the file header.v
```
- **`ifdef**
 - conditional compilation

```
`ifdef TEST module test;  
// compile module test only if text macro TEST  
// is defined using `define TEST
```
- **`timescale**
 - **`timescale** 100ns/1ns

simulate 時最精準到 1ns