**Design Sources**

**(\*** DONT\_TOUCH **=** "TRUE" **\*)**

**module** calc\_hamming

**(**

**input** **[**31**:**0**]** DATA**,**

**output** **[**5**:**0**]** RESULT

**);**

**wire** **[**2**:**0**]** sum\_ham **[**7**:**0**];**

**reg** **[**5**:**0**]** res\_reg**;**

**integer** j**;**

**genvar** i**;**

**generate**

**for(**i **=** 0**;** i**<**8**;** i **=** i**+**1**)** **begin** **:** gen\_part

slice\_adder gen\_slice

**(**

DATA**[**i**\***4 **+:**4**],**

sum\_ham**[**i**]**

**);**

**end**

**endgenerate**

**always** **@(\*)** **begin**

res\_reg **=** 6'b0**;**

**for** **(**j **=** 0**;** j **<** 8**;** j **=** j **+** 1**)** **begin**

res\_reg **=** res\_reg **+** sum\_ham**[**j**];**

**end**

**end**

**assign** RESULT **=** res\_reg**;**

**endmodule**

"calc\_hamming" is our top module. We take a 32-bit binary input and create a 6-bit output. We divide the 32-bit input into eight 4-bit slices. We create eight “sum\_ham” wires. Each wire has 3-bit length. These wires hold the ones in each 4-bit slice. We use always structure to calculate all slice results to the output of "calc\_hamming" and we need reg type. Therefore, we create the “res\_reg” wire that holds the number of ones in all slices

Next, we instantiate the “slice\_adder" module and calculate the ones in each 4-bit slice with generate-for structure. Finally, we add all slice results to the output of "calc\_hamming" with always structure.

**module** slice\_adder

**(**

**input** **[**3**:**0**]** slice**,**

**output** **[**2**:**0**]** sum

**);**

**wire** **[**1**:**0**]** half\_sum1**;**

**wire** **[**1**:**0**]** half\_sum2**;**

HA half1

**(**

**.**x**(**slice**[**0**]),**

**.**y**(**slice**[**1**]),**

**.**sum**(**half\_sum1**[**0**]),**

**.**cout**(**half\_sum1**[**1**])**

**);**

HA half2

**(**

**.**x**(**slice**[**2**]),**

**.**y**(**slice**[**3**]),**

**.**sum**(**half\_sum2**[**0**]),**

**.**cout**(**half\_sum2**[**1**])**

**);**

parametric\_RCA para1

**(**

**.**x**(**half\_sum1**),**

**.**y**(**half\_sum2**),**

**.**cin**(**0**),**

**.**cout**(**sum**[**2**]),**

**.**sum**(**sum**[**1**:**0**])**

**);**

**endmodule**

"slice\_adder" is the module that calculates the number of ones in a slice. We take 4-bit input and 3-bit output. We use two half adder and a 2-bit ripple carry adder. We create two 2-bit wires. These wires hold number of ones two bit of 4-bit slice. Finally, we add two wires with 2-bit ripple carry adder. The inputs to the ripple carry adder are two 2-bit wires, the cin is logical 0 because we calculate each slice independently. The sum of cout and ripple carry adder depends on the total output of "slice\_adder".

**module** OR

**(**

**input** l1**,**

**input** l2**,**

**output** O

**);**

**assign** O **=** l1 **|** l2**;**

**endmodule**

**module** parametric\_RCA

**(**

**input** **[**1**:**0**]** x**,**

**input** **[**1**:**0**]** y**,**

**input** cin**,**

**output** cout**,**

**output** **[**1**:**0**]** sum

**);**

**wire** **[**2**:**0**]** cout\_gen**;**

**assign** cout\_gen**[**0**]** **=** cin**;**

**genvar** i**;**

**generate**

**for(**i **=** 0**;** i**<**2**;** i **=** i**+**1**)** **begin** **:** gen\_full\_adder

FA gen\_full

**(**

x**[**i**],**

y**[**i**],**

cout\_gen**[**i**],**

cout\_gen**[**i**+**1**],**

sum**[**i**]**

**);**

**end**

**endgenerate**

**assign** cout **=** cout\_gen**[**2**];**

**endmodule**

**module** HA

**(**

**input** x**,**

**input** y**,**

**output** cout**,**

**output** sum

**);**

**assign** sum **=** x **^** y **;**

**assign** cout **=** x **&** y**;**

**endmodule**

**module** FA

**(**

**input** x**,**

**input** y**,**

**input** cin**,**

**output** cout**,**

**output** sum

**);**

**wire** ha1\_sum**,** ha1\_cout**,** ha2\_cout**;**

HA half1

**(**

**.**x**(**x**),**

**.**y**(**y**),**

**.**sum**(**ha1\_sum**),**

**.**cout**(**ha1\_cout**)**

**);**

HA half2

**(**

**.**x**(**ha1\_sum**),**

**.**y**(**cin**),**

**.**sum**(**sum**),**

**.**cout**(**ha2\_cout**)**

**);**

OR or1

**(**

**.**l1**(**ha1\_cout**),**

**.**l2**(**ha2\_cout**),**

**.**O**(**cout**)**

**);**

**endmodule**

**module** HA

**(**

**input** x**,**

**input** y**,**

**output** cout**,**

**output** sum

**);**

**assign** sum **=** x **^** y **;**

**assign** cout **=** x **&** y**;**

**endmodule**

**module** FA

**(**

**input** x**,**

**input** y**,**

**input** cin**,**

**output** cout**,**

**output** sum

**);**

**wire** ha1\_sum**,** ha1\_cout**,** ha2\_cout**;**

HA half1

**(**

**.**x**(**x**),**

**.**y**(**y**),**

**.**sum**(**ha1\_sum**),**

**.**cout**(**ha1\_cout**)**

**);**

HA half2

**(**

**.**x**(**ha1\_sum**),**

**.**y**(**cin**),**

**.**sum**(**sum**),**

**.**cout**(**ha2\_cout**)**

**);**

OR or1

**(**

**.**l1**(**ha1\_cout**),**

**.**l2**(**ha2\_cout**),**

**.**O**(**cout**)**

**);**

**endmodule**

**Block Schema of “slice\_adder” Module Simulation Source**

slice[3]

slice[2]

slice[1]

slice[0]

**Half**

**Adder**

half\_sum2[1]

half\_sum1[1]

**Half**

**Adder**

half\_sum2[0]

half\_sum1[0]

sum[0]

sum[1]

sum[2]

0

**Full**

**Adder**

**Full**

**Adder**

**Ripple Carry Adder**

**Simulation Source**

**module** slice\_adder\_tb**();**

**reg** **[**3**:**0**]** SLICE **=** 4'b000**;**

**wire** **[**2**:**0**]** SUM**;**

slice\_adder uut

**(**

**.**slice**(**SLICE**),**

**.**sum**(**SUM**)**

**);**

**initial**

**begin**

SLICE **=** 4'b0000**;**

**#**10**;**

SLICE **=** 4'b0001**;**

**#**10**;**

SLICE **=** 4'b0010**;**

**#**10**;**

SLICE **=** 4'b0011**;**

**#**10**;**

SLICE **=** 4'b0100**;**

**#**10**;**

SLICE **=** 4'b0101**;**

**#**10**;**

SLICE **=** 4'b0110**;**

**#**10**;**

SLICE **=** 4'b0111**;**

**#**10**;**

SLICE **=** 4'b1000**;**

**#**10**;**

SLICE **=** 4'b1001**;**

**#**10**;**

SLICE **=** 4'b1010**;**

**#**10**;**

SLICE **=** 4'b1011**;**

**#**10**;**

SLICE **=** 4'b1100**;**

**#**10**;**

SLICE **=** 4'b1101**;**

**#**10**;**

SLICE **=** 4'b1110**;**

**#**10**;**

SLICE **=** 4'b1111**;**

**#**10**;**

$finish**;**

**end**

**endmodule**

We created all 4-bit numbers and assigned them to the input of the system. I controlled “slice\_adder” module whether the system is working.

**import** random

**def** gen\_binary**():**

**with** **open(**"stimulus\_file.txt"**,**"w"**)** **as** f**:**

**for** k **in** **range(**0**,**100**):**

number **=** random**.***getrandbits***(**32**)**

bin\_num **=** **format(**number**,** "0b"**)**

**print(len(**bin\_num**))**

**if** **len(**bin\_num**)<**32**:**

**for** i **in** **range(**0**,**32**-len(**bin\_num**)):**

bin\_num **=** "0" **+** **str(**bin\_num**)**

ones **=** bin\_num**.***count***(**'1'**)**

**print(**"number of 1s"**)**

**print(**ones**)**

f**.***write***(**bin\_num **+** " " **+** **str(**ones**)** **+**"\n"**)**

**print(len(**bin\_num**))**

**print(**"----------------"**)**

We created a Python code to generate 32-bit random data. We used random library. "random.getrandbits(32)" produces numbers in the range 0 to 2^(32) -1. "format function" converts decimal type to binary. If the numbers are less than 32 bits long, we do zero extension. For example, if the number is 5, it has a binary representation of 101. After zero extension, it has a 32-bit representation. "ones" function calculates a number of ones in the string. To compare the ones count with our Verilog results, we added the 32-bit data and the ones count of the data to the same line in the file.

**Example input file line**

10101010011010001111111100010000 16

32-bit data Number of ones in DATA

"calc\_hamming\_tb” is our main testbench module. We use the module for simulation. Firstly, we open the input file(stimulus\_file) and output file(outputs) with read and write modes respectively.  If it cannot open the files, it gives an error message. Next, while loop scans the line-by-line input file, it assigns the first value in the file (32-bit input) to “DATA” and, the second to “ones\_count”(number of ones in data). After calculating “RESULT,” it compares “RESULT” with “ones\_count”. If it has the same results, it gives the true message. Otherwise, it provides a false message. It provides message TCL Console and out output file. Finally, it closes the input and output files and completes the module.

**module** calc\_hamming\_tb**();**

**reg** **[**31**:**0**]** DATA **=** 32'b0**;**

**wire** **[**5**:**0**]** RESULT**;**

**integer** file**,** out\_file**,**ones\_count**;**

calc\_hamming uut

**(**

**.**DATA**(**DATA**),**

**.**RESULT**(**RESULT**)**

**);**

**initial** **begin**

file **=** $fopen**(**"stimulus\_file.txt"**,** "r"**);**

**if** **(**file **==** 0**)** **begin**

$display**(**"Cannot open stimulus file."**);**

$finish**;**

**end**

out\_file **=** $fopen**(**"outputs.txt"**,** "w"**);**

**if** **(**out\_file **==** 0**)** **begin**

$display**(**"Cannot open outputs file."**);**

$finish**;**

**end**

**while** **(!**$feof**(**file**))** **begin**

$fscanf**(**file**,** "%b %d\n"**,** DATA**,** ones\_count**);**

**#**10**;**

**if(** ones\_count **==** RESULT**)** **begin**

$display**(**"DATA: %b, Ones Count:%d ,RESULT: %d, TRUE"**,**DATA**,** ones\_count**,**RESULT**);**

$fdisplay**(**out\_file**,** "DATA: %b, Ones Count: %d, RESULT: %d, TRUE"**,** DATA**,** ones\_count**,** RESULT**);**

**end**

**else** **begin**

$display**(**"DATA: %b, Ones Count:%d ,RESULT: %d, FALSE"**,**DATA**,** ones\_count**,**RESULT**);**

$fdisplay**(**out\_file**,** "DATA: %b, Ones Count: %d, RESULT: %d, TRUE"**,** DATA**,** ones\_count**,** RESULT**);**

**end**

**end**

$fclose**(**file**);**

$fclose**(**out\_file**);**

$finish**;**

**end**

**endmodule**

**Example output file line**

DATA: 10101010011010001111111100010000, Ones Count: 16, RESULT: 16, TRUE