數位系統導論實驗
Lab8 RTL Modeling(Adder) & Testbench

負責助教：王偉丞

Email: wmike851223@gmail.com

Outline

- 課程目的
- 浮點數加法運算
- 範例 – 32-bit浮點數加法器
- Testbench
- 範例 – Testbench
- 作業說明
- 評分方式

課程目的

- 在上次實驗中，已向同學介紹過IEEE 754的浮點數表示法和其加法器實作，本次實驗將介紹浮點數加法運算並以Verilog實做其加法器
- 前幾次實驗都是由助教提供Testbench，本次實驗亦將帶大家了解Testbench可以怎麼設計

浮點數加法運算（1/3）

- 我們將以三個步驟介紹IEEE 754單精度浮點數加法運算

1. 對齊 (Alignment)
   - 開始運算之前，我們需要將兩數的指數及尾數對齊
2. 運算 (Calculation)
   - 以正負號決定執行加法或減法
3. 正規化 (Normalization)
   - 運算後的結果我們要轉回IEEE 754表示法，而在運算的過程中可能因為精確度造成bit過多，這時我們需要將這些bit做捨入

浮點數加法運算（2 / 3）

- 我們以浮點數a和b的示範IEEE 754加法運算（s = a + b）

  a: 0_01111000_11000000000000000010001, b: 1_01111101_00000100000000000000000,

  s_a = 0, e_a = 01111000 = 120, and 1.f_a = 1.11000000000000000010001

  s_b = 1, e_b = 01111101 = 125, and 1.f_b = 1.00000100000000000000000

1. |b| > |a|，$e_b - e_a = 125 - 120 = 5$，將1.$f_a$右移 5bit

   $a = 2^{120-127} \times 1.11000000000000000010001 = 2^{125-127} \times 0.00001110000000000000000\_10001$

   多出來的10001，我們將最左邊兩bit保留 (保護、循環位)，其餘我們使用reduction OR簡化成1bit，得到101。

   設grs=101，利用grs參與有效位計算以確保更精確的結果

		grs	
	01.00000100000000000000000	000	(significand of *b*)
−	00.00001110000000000000000	101	(significand of *a*)
	00.11110101111111111111111	011	(significand of *s*)

2. 由於兩個有效位的加法結果可能大於2，因此我們在計算過程中使用了一個附加位到最高有效位位置

   總位數是1 + 24 + 3或28

浮點數加法運算（3 / 3）

3. 運算後結果需要轉成$1.f$，從左邊找到第一個 “1” ，並調整指數

   $s = -2^{125-127} \times 0.11110101111111111111111\_011 = -2^{124-127} \times 1.11101011111111111111110\_110$

   最後處理剩餘的bit，將結果修約為最接近且可以表示的值，但是當存在兩個數一樣接近的時候，則取其中的偶數（Ties To Even），得到結果：

   s: 1_01111100_11101011111111111111111

- 關於捨入方式可以參考： https://en.wikipedia.org/wiki/IEEE_754#Rounding_rules

範例 – 32-bit浮點數加法器（Top Module）

- 利用Top Module整合各個Module，將所有IO接好，藉此有利於我們個別除錯
  1. Alignment：對齊a和b
  2. Calculation：進行a與b的Fraction運算
  3. Normalization：對Fraction運算結果做捨入並轉成IEEE 754格式

```
`include "fp_align.v"
`include "fp_cal.v"
`include "fp_norm.v"
`timescale 1ns / 1ps

module fpadder (          // fp adder
    input [31:0] a, b,    // fp a and b
    input sub,            // 1: sub; 0: add
    output [31:0] s);     // fp output

    wire a_sign;
    wire [7:0] a_exp;
    wire a_op_sub;
    wire [26:0] a_small_frac;
    wire [23:0] a_large_frac;
```

```
    // Alignment
    fadd_align alignment (
        .a(a),
        .b(b),
        .sub(sub),
        .sign(a_sign),       // sign of large one
        .temp_exp(a_exp),    // exponent of large one
        .op_sub(a_op_sub),   // add or sub
        .large_frac24(a_large_frac),    // fraction of large one
        .small_frac27(a_small_frac));   // fraction of small one

    // Calculation
    wire [27:0] c_frac;
    fadd_cal calculation (
        .op_sub(a_op_sub),
        .large_frac24(a_large_frac),
        .small_frac27(a_small_frac),
        .cal_frac(c_frac)); // result fraction

    // Normalization
    fadd_norm normalization (
        .sign(a_sign),       // result sign
        .temp_exp(a_exp),    // result exponent
        .cal_frac(c_frac),   // result fraction
        .s(s));              // result after normalization
endmodule
```

範例 – 32-bit浮點數加法器（ Alignment ）

1. Exchange：依照a或b大，分別儲存其Fraction與較大數的Exponent
2. Control logic：判斷結果值正負號與Fraction運算該用加法或減法
3. Shift：將較小數的Fraction位移並加上保護、循環位

```
/* Alignment */
module fadd_align (
    input [31:0] a, b,  // fp a and b
    input sub,          // add or sub
    output sign,            // result sign
    output [7:0] temp_exp,  // result exponent
    output op_sub,          // fraction operation
    output [23:0] large_frac24,     // fraction of large one
    output [26:0] small_frac27);    // fraction of small one

    // 不看sign bit,判斷a or b大
    wire exchange = (b[30:0] > a[30:0]);
    wire [31:0] fp_large = exchange? b : a;
    wire [31:0] fp_small = exchange? a : b;

    // 轉成1.f格式
    wire fp_large_hidden_bit = |fp_large[30:23];
    wire fp_small_hidden_bit = |fp_small[30:23];
    wire [23:0] large_frac24 = {fp_large_hidden_bit,fp_large[22:0]};
    wire [23:0] small_frac24 = {fp_small_hidden_bit,fp_small[22:0]};
```

```
    // 儲存較大數的exponent
    assign temp_exp = fp_large[30:23];

    // 判斷結果值正負
    assign sign = exchange? sub^b[31] : a[31];

    // fraction運算使用加法或減法
    assign op_sub = sub ^ fp_large[31] ^ fp_small[31];

    // ea-eb
    wire [7:0] exp_diff = fp_large[30:23] - fp_small[30:23];

    // 計算需要位移幾位，並且位移
    wire [49:0] small_frac50 = (exp_diff >= 26) ?
                               {26'h0, small_frac24} :
                               {small_frac24, 26'h0} >> exp_diff;

    // 留下24bit以及保護.循環位，計算多出來的位數，使用reduction OR簡化成1bit
    assign small_frac27 = {small_frac50[49:24], |small_frac50[23:0]};
endmodule
```

範例 – 32-bit浮點數加法器（Calculation）

- 在此ALU中，我們將要運算的兩數以及運算符號傳入，加上grs以及考慮溢位的狀況，將兩者bit對齊，再做運算

```
/* Calculation */
module fadd_cal (
    input op_sub,
    input [23:0] large_frac24,
    input [26:0] small_frac27,
    output [27:0] cal_frac);

    //補上grs, 以及避免溢位狀況, 共28bit
    wire [27:0] aligned_large_frac = {1'b0, large_frac24, 3'b000};
    wire [27:0] aligned_small_frac = {1'b0, small_frac27};

    //相減或相加
    assign cal_frac = op_sub?
        aligned_large_frac - aligned_small_frac :
        aligned_large_frac + aligned_small_frac;
endmodule
```

範例 – 32-bit浮點數加法器（Normalization）

1. Shift：依照由左數起有幾個0來決定要向左移動幾位
2. Rounding：以循環、保護位決定捨入

```
/* Normalization */
module fadd_norm (
    input sign,         // result sign
    input [7:0] temp_exp,   // temp exponent
    input [27:0] cal_frac,  // fraction before normalization
    output [31:0] s);       // result

    wire [26:0] f4, f3, f2, f1, f0;

    // 從左邊找到第一個1
    wire [4:0] zeros;   // 左邊數起有幾個0
    assign zeros[4] = ~|cal_frac[26:11];    // 16-bit 0
    assign f4 = zeros[4]? {cal_frac[10:0],16'b0} : cal_frac[26:0];
    assign zeros[3] = ~|f4[26:19];          // 8-bit 0
    assign f3 = zeros[3]? {f4[18:0], 8'b0} : f4;
    assign zeros[2] = ~|f3[26:23];          // 4-bit 0
    assign f2 = zeros[2]? {f3[22:0], 4'b0} : f3;
    assign zeros[1] = ~|f2[26:25];          // 2-bit 0
    assign f1 = zeros[1]? {f2[24:0], 2'b0} : f2;
    assign zeros[0] = ~f1[26];              // 1-bit 0
    assign f0 = zeros[0]? {f1[25:0], 1'b0} : f1;
    reg [26:0] frac0;
    reg [7:0] exp0;
```

```
    // 位移以及修改exponent
    always @ * begin
        if (cal_frac==0) begin  // Result is 0
            frac0 = 0;
            exp0 = 0;
        end
        else if (cal_frac[27]) begin
            frac0 = cal_frac[27:1]; // 1x.xxxxxxxxxxxxxxxxxxxxxxx xx
            exp0 = temp_exp + 8'h1; // ->1.xxxxxxxxxxxxxxxxxxxxxxx xxx
        end
        else begin
            exp0 = temp_exp - zeros;
            frac0 = f0; // 01.xxxxxxxxxxxxxxxxxxxxxxx xxx

        end
    end

    // Rounding
    wire frac_plus_1 =
    frac0[2] & (frac0[1] | frac0[0]) |
    frac0[2] & ~frac0[1] & ~frac0[0] & frac0[3];

    wire [24:0] frac_round = {1'b0, frac0[26:3]} + frac_plus_1;
    wire [7:0] exponent = frac_round[24] ?
                          exp0 + 8'h1 :
                          exp0;

    assign s = {sign, exponent, frac_round[22:0]};
endmodule
```

範例 – 32-bit浮點數加法器

- 輸入指令執行程式，檢視設計之 FPADDE 功能是否有錯誤：
  - iverilog –o testbench32 testbench32.v
  - vvp testbench32
  - gtkwave adder32.vcd

Testbench

- 當我們設計完自己的Module後，一定需要驗證其輸出結果是否和預期的一樣，此時就會用另一支程式產生資料做為該Module的輸入，並比較Module輸出與正確輸出是否一致。該產生輸入資料並驗證Module行為的即稱為Testbench（也是一個Module）

範例 – Testbench（1 / 2）

- 宣告好要傳輸入資料與接收輸出的變數並將其與待驗證Module連接（如左圖）
- *initial*內的指令會在程式一跑起來後即開始執行（如右圖）
  - *$dumpfile*內的字串即為產生的波型檔名字
  - *$dumpvars*會紀錄輸入的變數在整個模擬過程的變化，如無輸入則紀錄所有變數
  - #後接的數字代表會在經過多久時間後才繼續往下執行

```
module test_bench;

    /* Variables Declaration */
    reg[31:0] testInput_a;
    reg[31:0] testInput_b;
    reg [31:0] correct_ans;
    wire [31:0] adder_ans;
    /* Variables Declaration */

    /* 待驗證的Module */
    fpadder test(
        .a(testInput_a),
        .b(testInput_b),
        .sub(1'b0),
        .s(adder_ans));
```

```
reg [4:0] cnt_test, cnt_right;
initial begin
    $dumpfile("adder32.vcd");
    $dumpvars;

    #0
        cnt_test = 0;
        cnt_right = 0;

    #1  /********* Test Pattern *********/
        testInput_a = 32'hBA9DBB67;     // -0.0012034
        testInput_b = 32'h4148F5CB;     // 12.560008
        correct_ans = 32'h4148F0DD;     // 12.5588045
        cnt_test = cnt_test + 1;
```

範例 – Testbench（2 / 2）

- 依據Module輸出與預期輸出是否一樣而選擇執行不同task
  - task有點類似c語言的函式，只是它必須被定義在Module內，且內部不能使用always或initial
- 在執行到*$finish*時結束模擬

```
#10
    if (adder_ans == correct_ans) begin
        success_case (testInput_a, testInput_b, correct_ans);
    end else begin
        failure_case (testInput_a, testInput_b, correct_ans, adder_ans);
    end
```

```
/*--------------- Standard Output for Success Case ---------------*/
task success_case;
input [31:0] testInput_a, testInput_b, correct_ans;
begin
    cnt_right = cnt_right + 1;

    $display ("Test %d ", cnt_test);
    $display ("////////////////////////");
    $display ("//// Successful %d ////", cnt_right);
    $display ("////////////////////////");
    $display ("%h + %h = ?", testInput_a, testInput_b);
    $display ("Answer = %h\n", correct_ans);
end endtask
/*--------------- Standard Output for Success Case ---------------*/
```

```
/*--------------- Standard Output for Failure Case ---------------*/
task failure_case;
input [31:0] testInput_a, testInput_b, correct_ans, result;
begin
    $display ("Test %d ", cnt_test);
    $display ("////////////////////////");
    $display ("///////// Fail ////////");
    $display ("////////////////////////");
    $display ("%h + %h = ?", testInput_a, testInput_b);
    $display ("your answer = %h", result);
    $display ("correct answer = %h\n", correct_ans);
end endtask
/*--------------- Standard Output for Failure Case ---------------*/
```

```
    $finish;
end
```

作業

1. 操作範例中單精確度 (32 bit) 浮點數加法器，使用GTKWave觀察其運算結果
2. 參考範例完成半精度 (16 bit) 浮點數加法器

作業說明及課程評分

- Demo 時間：5/6(一)、5/8(三)，測驗時間分別為 19:30、19:50、 20:10 與 20:30
- Demo 地點：工一館105
- 評分方式：
  - Part1：40%
  - Part2：40%
  - 隨堂測驗：20%