FPGA Prototyping Using Synopsys HAPS Final

李育松 111501555

1. Functionality

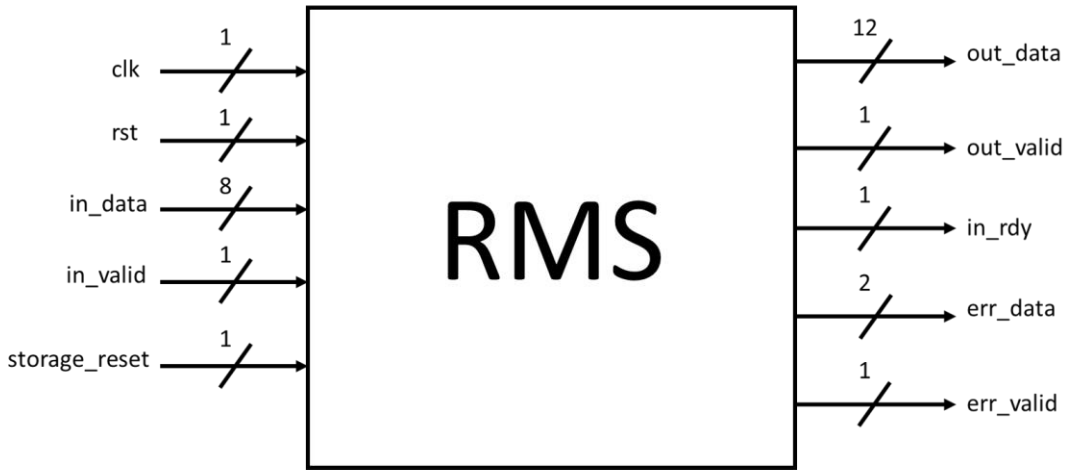
Generate a sequence of 8-bit integer numbers {a1, …, aN}, design a logic circuit that can compute the rootmean-square denoted as RMS, where

$$RMS = \sqrt{\frac{a_1^2 + \cdots + a_N^2}{N}}$$

Note that, the final RMS result can be represented as a fixed-point number with 8 integer bits and 4 fractional bits, say RMS_code[11:0].

2. Specification

- Module required function
  - Do root-mead-square(RMS)
  - Support max 256 pieces of input data
  - Support max 8 bits input data
  - Support max 8 interger bits and 4 fractional bits output data
  - Error detection mechanism
    - overflow
    - max input data length exceeds
    - no error
  - input/ output handshaking mechanism
- Block diagram



  - out_valid and err_valid shouldn't high at the same time.
  - when RMS is ready to get data in_rdy should be high.
- IO lists

  input

  - clk
    - bit
      - 1
    - definition
      - clock singal
  - rst
    - bit

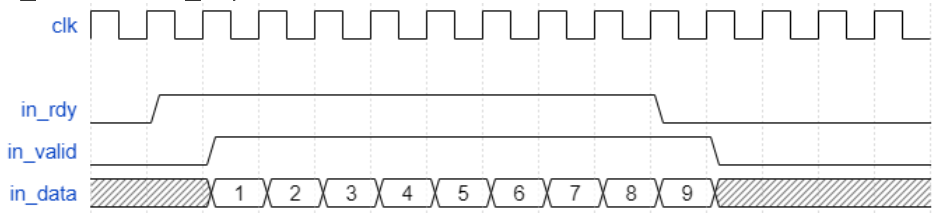
        - 1
    - definition
        - active-high synchronous reset
- in_data
    - bit
        - 8
    - definition
        - input interger data
- in_valid
    - bit
        - 1
    - definition
        - input data valid 1 (high) for valid.
- storage_reset
    - bit
        - 1
    - definition
        - you should have a storage variable for the sum of square input data and the length of all input data. when this singal pull high you should reset those for the next data sequences.

output

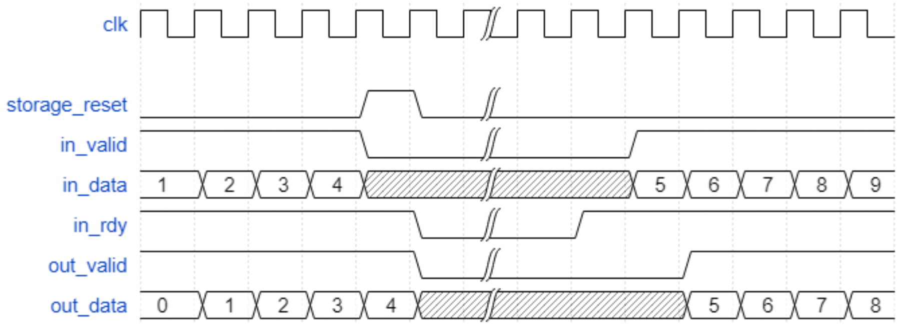
- out_data
    - bit
        - 12
    - definition
        - out data of your RMS. should come at every cycle
- out_valid
    - bit
        - 1
    - definition
        - out data valid
- in_rdy
    - bit
        - 1
    - definition
        - should pull high when your RMS is ready for receiving input data
- err_data
    - bit
        - 2
    - definition
        - 2'b00 is no error
        - 2'b01 is input data length exceeds 256
        - 2'b10 is the sum of square input data exceed your max storage

- err_valid
    - bit
        - 1
    - definition
        - err_data valid. should be high when err_data is 01, or 10

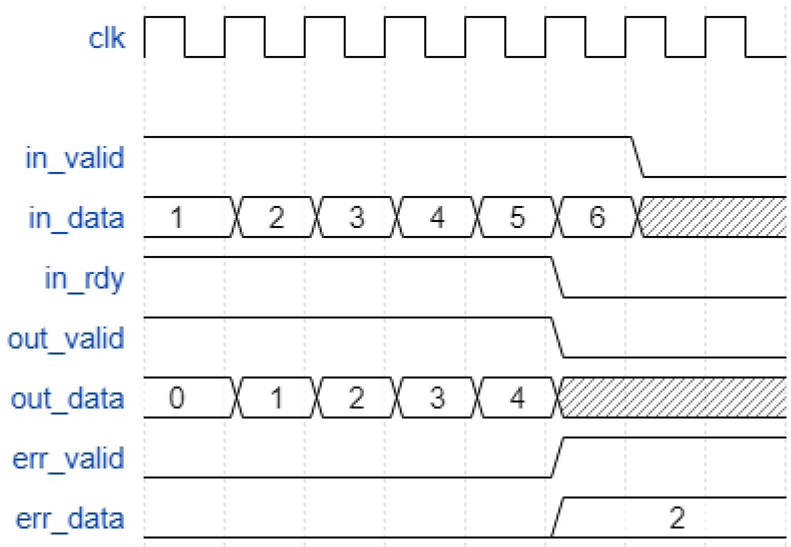
- Example Waveform
    - in_data, in_valid and in_rdy



        -
        - When in_rdy is pulled high, you can start sending data in the next cycle. When in_rdy is pulled low the data at that cycle (data 9 in the example) can't be process by RMS and you should stop sending data.
    - storage reset



        -
        - When storage reset is pulled up, the in_valid won't be pulled up at the same time. After some time your design will pull up in_rdy, meaning your design has reseted all storage and readied for next input data. Later we will send next in_data and get the corrsponding out data after a cycle.
    - error and output example



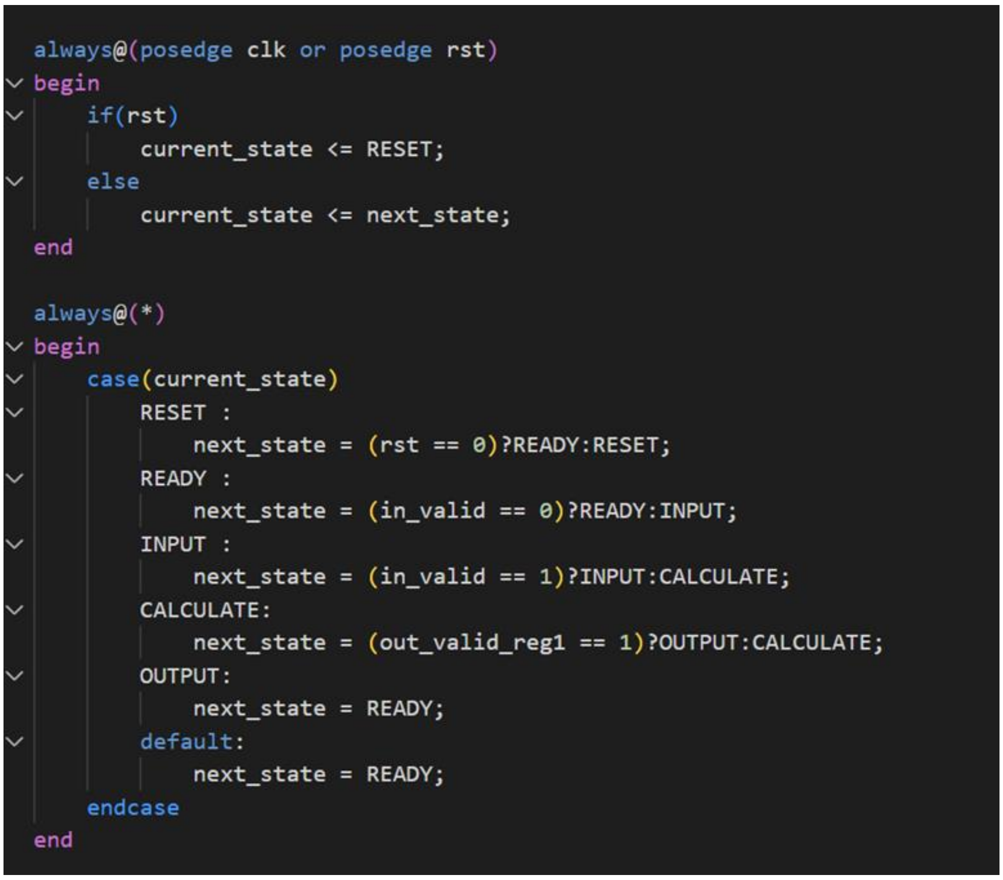
■ When error occures (happened when computing in_data 5), you in_rdy and out_valid should be pulled low. error_valid should be pulled high.

- Simulation
  - go to design folder
  - type "./01_run" in your terminal
  - modify RMS.v for design
  - modify pattern/RMS_tb.v for testbench
  - modify golden/pattern/pattern_example.in for input/output data

3. Verilog Design

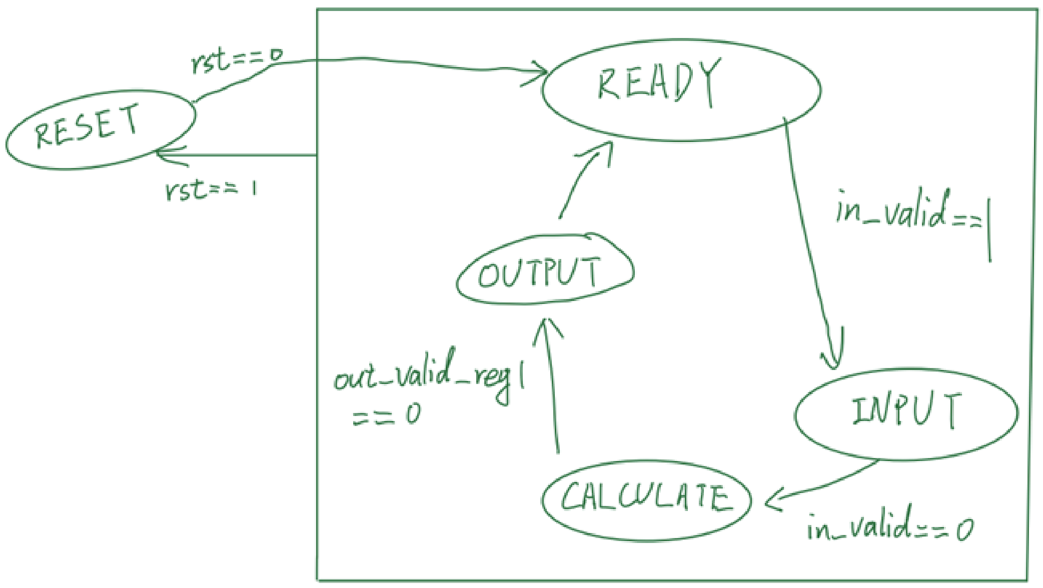
本次設計主要以 RMS.v 爲主 module，計算平方根的 sqrt.v 為子 module

FSM：



```
always@(posedge clk or posedge rst)
begin
    if(rst)
        current_state <= RESET;
    else
        current_state <= next_state;
end

always@(*)
begin
    case(current_state)
        RESET :
            next_state = (rst == 0)?READY:RESET;
        READY :
            next_state = (in_valid == 0)?READY:INPUT;
        INPUT :
            next_state = (in_valid == 1)?INPUT:CALCULATE;
        CALCULATE:
            next_state = (out_valid_reg1 == 1)?OUTPUT:CALCULATE;
        OUTPUT:
            next_state = READY;
        default:
            next_state = READY;
    endcase
end
```

圖六、FSM control

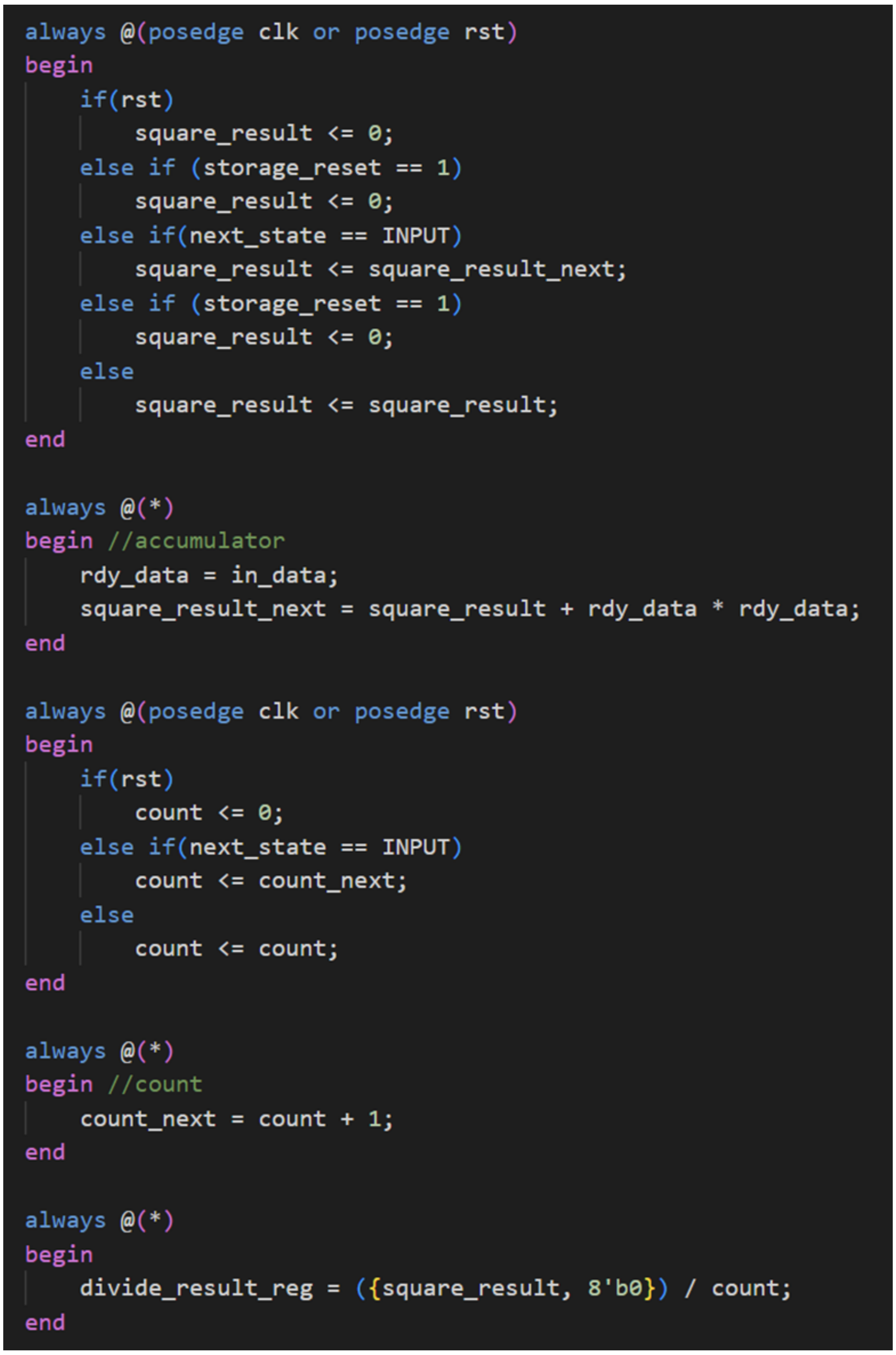


圖七、state control

RMS 中，通過 FSM 來分割控制 state，分別爲 RESET，等待輸入的 READY，

輸入資料並計算平方和以及平均的 INPUT，計算方根結果的 CALCULATE，輸出資料的 OUTPUT，圖六為 FSM 詳細 code，圖七爲 state control 示意圖。

INPUT：



```
always @(posedge clk or posedge rst)
begin
    if(rst)
        square_result <= 0;
    else if (storage_reset == 1)
        square_result <= 0;
    else if(next_state == INPUT)
        square_result <= square_result_next;
    else if (storage_reset == 1)
        square_result <= 0;
    else
        square_result <= square_result;
end

always @(*)
begin //accumulator
    rdy_data = in_data;
    square_result_next = square_result + rdy_data * rdy_data;
end

always @(posedge clk or posedge rst)
begin
    if(rst)
        count <= 0;
    else if(next_state == INPUT)
        count <= count_next;
    else
        count <= count;
end

always @(*)
begin //count
    count_next = count + 1;
end

always @(*)
begin
    divide_result_reg = ({square_result, 8'b0}) / count;
end
```

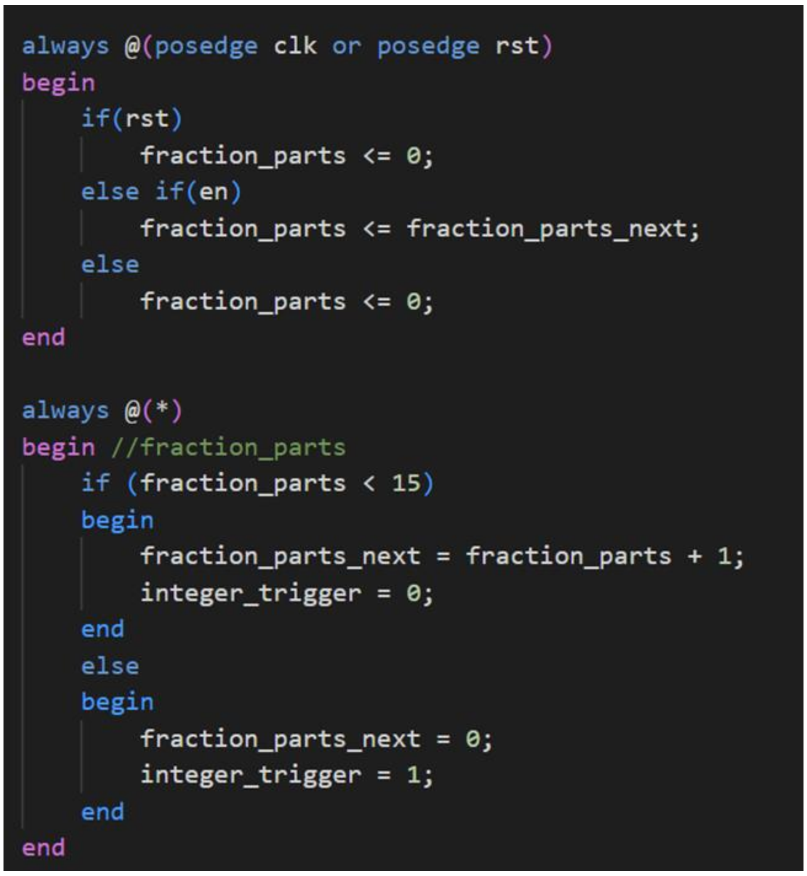
圖八、INPUT control

圖八為 INPUT state 詳細 code，在 input state，我們將 input data 存在 rdy_data 中，並用 flipflop 存取每一次平方和的結果以及 input data 的數量。在

最後做除法的過程中，我將平方和向左移 8 位再進行除法，給後續進行 sqrt 時提供有 8bit fraction parts 以及 16 bit integer parts 的數據。（雖然後來決定直接把總 bit 數拉到 32bit，靠後續合成來撇除未用到的 bit 數）

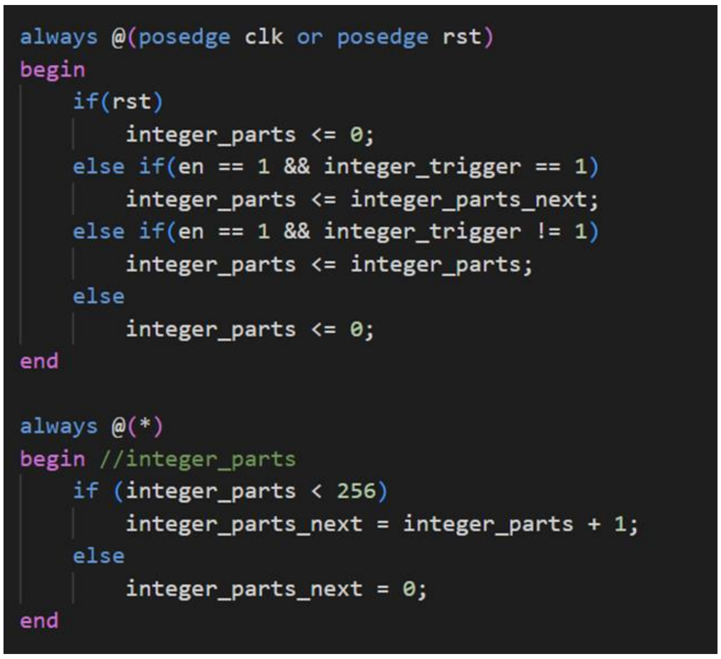
此進程需要 2 個 cycle

SQRT：



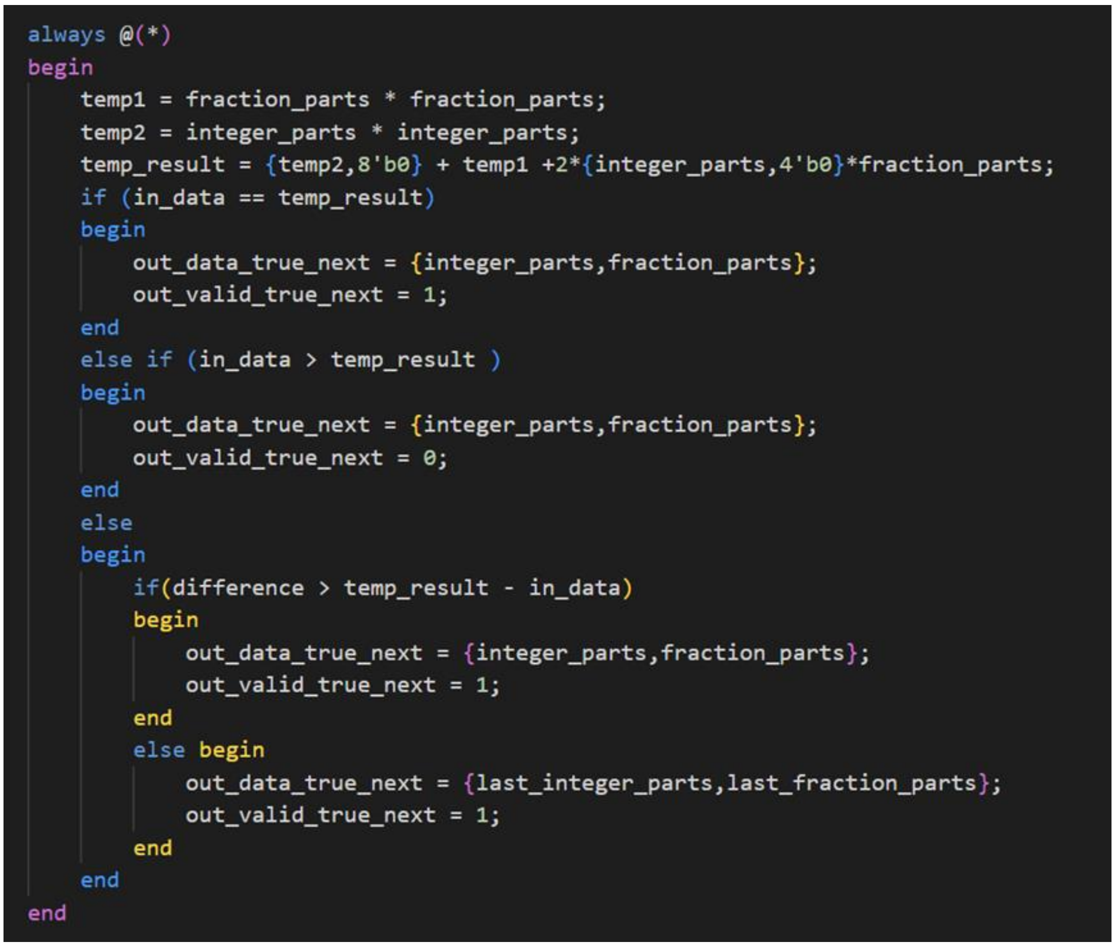
```
always @(posedge clk or posedge rst)
begin
    if(rst)
        fraction_parts <= 0;
    else if(en)
        fraction_parts <= fraction_parts_next;
    else
        fraction_parts <= 0;
end

always @(*)
begin //fraction_parts
    if (fraction_parts < 15)
    begin
        fraction_parts_next = fraction_parts + 1;
        integer_trigger = 0;
    end
    else
    begin
        fraction_parts_next = 0;
        integer_trigger = 1;
    end
end
```

圖九、fraction parts iteration



```verilog
always @(posedge clk or posedge rst)
begin
    if(rst)
        integer_parts <= 0;
    else if(en == 1 && integer_trigger == 1)
        integer_parts <= integer_parts_next;
    else if(en == 1 && integer_trigger != 1)
        integer_parts <= integer_parts;
    else
        integer_parts <= 0;
end

always @(*)
begin //integer_parts
    if (integer_parts < 256)
        integer_parts_next = integer_parts + 1;
    else
        integer_parts_next = 0;
end
```

圖十、integer parts iteration



```verilog
always @(*)
begin
    temp1 = fraction_parts * fraction_parts;
    temp2 = integer_parts * integer_parts;
    temp_result = {temp2,8'b0} + temp1 +2*{integer_parts,4'b0}*fraction_parts;
    if (in_data == temp_result)
    begin
        out_data_true_next = {integer_parts,fraction_parts};
        out_valid_true_next = 1;
    end
    else if (in_data > temp_result )
    begin
        out_data_true_next = {integer_parts,fraction_parts};
        out_valid_true_next = 0;
    end
    else
    begin
        if(difference > temp_result - in_data)
        begin
            out_data_true_next = {integer_parts,fraction_parts};
            out_valid_true_next = 1;
        end
        else begin
            out_data_true_next = {last_integer_parts,last_fraction_parts};
            out_valid_true_next = 1;
        end
    end
end
```

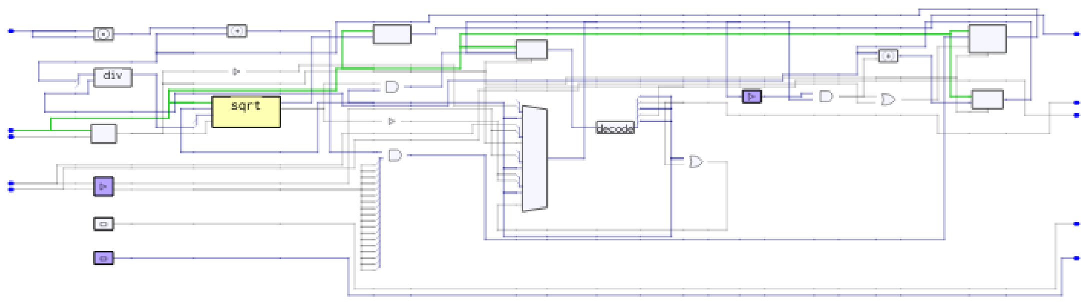
圖十一、SQRT important operation

計算方根的演算法參考 software，使用逼近法來尋找，從 0 開始，每次+1

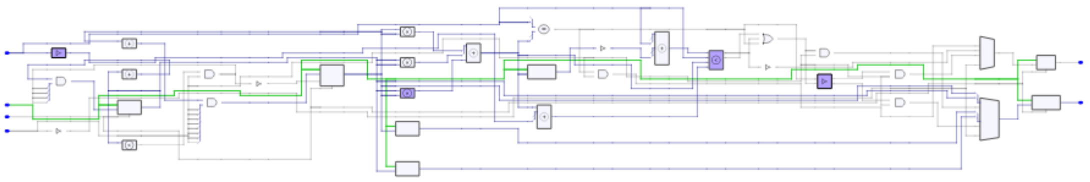
一路往上找，以找到最接近的答案。實際應用時，是分別以 fraction parts 和 integer parts 進行 iteration，再以平方和公式加總算出結果 temp_result，並與 data 進行比較。當上一筆 temp_result 小於 data，而下一筆 temp_result 大於 data，就比較其與 data 的差值（difference）大小，小者就是 root value，並拉起 out_valid 提醒主 module 進入 output state。

此進程的計算時間在 16*256 cycle 內。

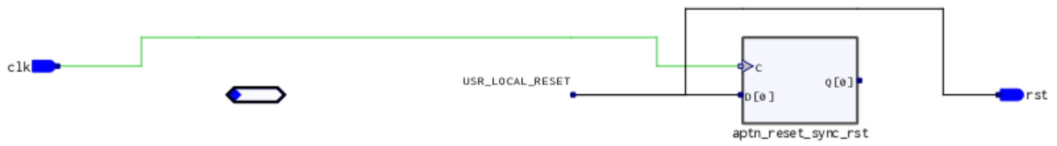
4. Result



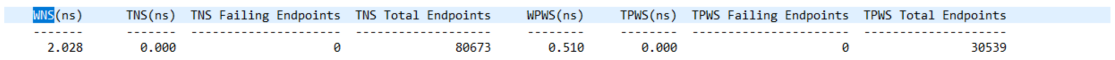
FB1 schematic　RMS



FB1 schematic sqrt

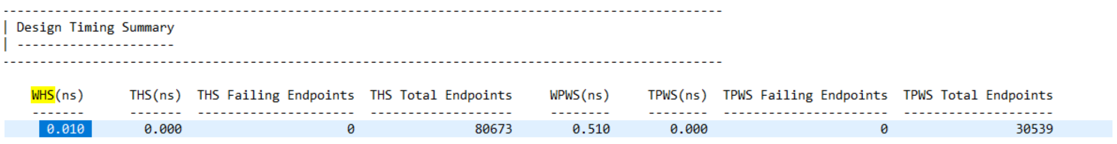


FA1 schematic reset



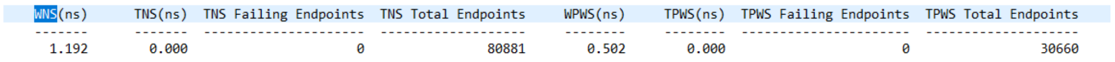
```
    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      --------     --------  ----------------------  --------------------
      2.028        0.000                      0                80673         0.510        0.000                       0                 30539
```

FA1 timing summary



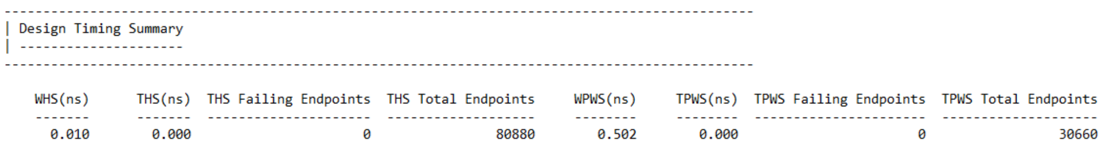
```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints      WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      --------     --------  ----------------------  --------------------
      0.010        0.000                      0                80673         0.510        0.000                       0                 30539
```

FA1 timing summary min



```
    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      --------     --------  ----------------------  --------------------
      1.192        0.000                      0                80881         0.502        0.000                       0                 30660
```

FB1 timing summary



```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints      WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      --------     --------  ----------------------  --------------------
      0.010        0.000                      0                80880         0.502        0.000                       0                 30660
```

FB1 timing summary min

5. Reflection

Through the process of partitioning my Verilog code for the Haps-100 FPGA board using protocompiler, I gained a deeper understanding of how to optimize and adapt code for specific hardware resources. The protocompiler tool allowed me to partition my code into smaller modules that were better suited to the FPGA board's memory and resource limitations. In doing so, I was able to improve the overall efficiency and performance of my design.

One of the most significant benefits of using protocompiler was the flexibility it provided. By breaking down my code into smaller modules, I was able to compile and test each module individually, which saved time and minimized errors. Additionally, the tool's code optimization capabilities allowed me to fine-tune my design for maximum efficiency, without sacrificing functionality.

Overall, the experience of using protocompiler to partition my Verilog code for the Haps-100 FPGA board was a valuable learning opportunity. Through this process, I developed a deeper understanding of FPGA design and optimization, and gained hands-on experience using a powerful tool for code partitioning and optimization. I look forward to applying these skills in future FPGA projects, and continuing to explore the many possibilities that this exciting technology has to offer.