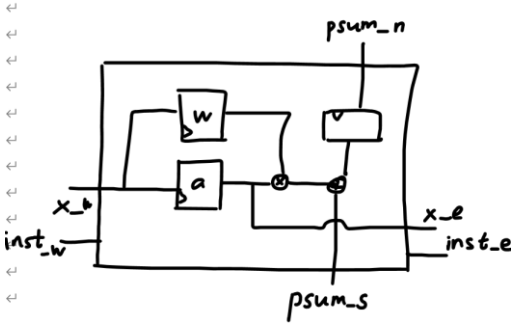
项目一

访存瓶颈：算力单元（MAC）吞吐量远高于片外/片内存带宽，导致运算单元等待数据，利用率下降。
数据重用：weight stationary method - 将权重值预先加载到计算单元上，使得每个权重值可以服务于多个激活值，加强了数据重用率，减少带宽需求。
数据量化: 使用 INT8 / INT4 或稀疏存储，减少带宽需求，QAT、模型剪枝后的推理
并行化数据加载和计算：矩阵计算是流水线式的->一边计算一边加载数据。
“总体上，我会优先提高片上数据重用率、并行化数据加载与计算，结合量化和访存模式优化来缓解瓶颈，这样既能保持算力利用率，又能在功耗和带宽之间取得平衡。”

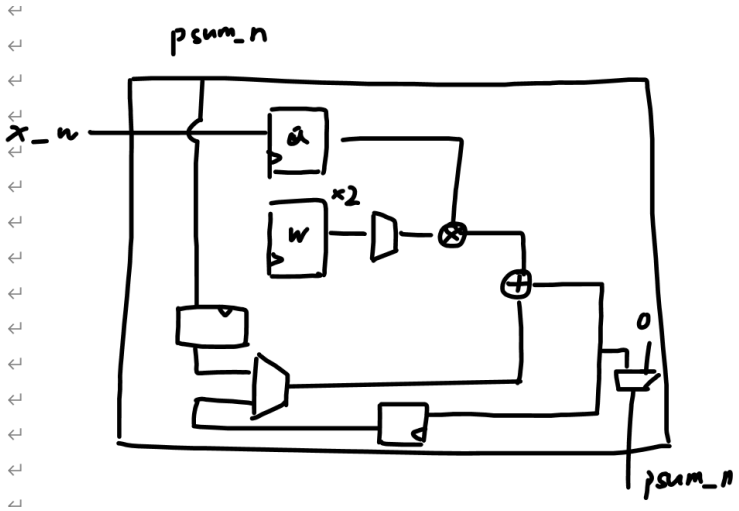
请解释双重缓冲的原理，以及它是如何将内存占用从线性级降到常数级的。
未优化之前：array->ofifo->pmem->sfp->pmem。将所有点积结果存入 pmem 后，sfp 再从中取出数据进行累加和 relu->内存占用随矩阵大小线性增长。
使用 double buffering: 第一次：array -> ofifo->sfp (同时读 pmem2)->写 pmem1
第二次：array -> ofifo->sfp (同时读 pmem1)->写 pmem2 （可覆盖）
如此循环往复。最终需要的内存大小不随矩阵大小而改变。同时这种方式提高了 array 和 sfp 的并行度，减少了延迟。

请解释 QAT 和普通量化的区别，以及为什么选择 QAT
普通量化：训练后直接将权重和激活值从高精度（如 FP32）映射到低精度（如 INT8）。实现简单，但可能出现较大精度下降。
QAT：在训练阶段就模拟低精度计算（插入量化/反量化算子），让前向传播中的权重和激活按照 INT8（或更低）精度进行运算。反向传播时依然用高精度梯度更新。模型在训练中“看到”了量化误差，并通过参数调整来适应它，使得精度损失较小。

在设计 8x8 阵列扩展到 16x8 时，你是如何重新设计 MAC 单元的？
优化之前



优化之后：



原来每个 mac 只能存放一个权重值，优化之后可以存储两个权重值，可以进行两次乘加运算->在同尺寸的计算阵列中支持更大的矩阵。

FPGA 和 ASIC 的主要设计考虑差异是什么

“FPGA 更强调快速实现和可调试性，适合验证架构和功能；ASIC 则要求在固定工艺下最大化 PPA，需要精细的物理实现和功耗优化。在 2D 脉动阵列中，我在 FPGA 阶段主要优化了

数据流和缓冲策略，如果迁移到 ASIC，会重点在存储架构、时钟门控和流水线深度上做针对性优化。”

如果阵列规模继续扩大，访存带宽成为瓶颈，你会如何优化？

我先用 output-stationary/row-stationary 提高在片重用，再把 tile 做大、用三缓冲+DMA 预取把计算和搬运完全重叠，同时通过 INT4/稀疏压缩把数据量再砍一半。片上用多 bank SRAM 和轻量 NoC 减少冲突，必要时在接口侧加宽位宽或上 HBM。这样通常能把外部带宽需求降 2–4×，阵列利用率接近满载。

项目二

你在项目中完成了从 RTL 到 GDS 的全流程，能否详细描述一下这个流程中的主要步骤和对应的 EDA 工具？

“RTL 到 GDS 的完整流程是：RTL 设计与验证 → 综合（synopsys design compiler） → 等价验证 → 物理设计（平面规划、布局、时钟树、布线）(cadence innovus)→ STA 收敛与功耗优化 → Signoff & Tape-out。

你提到优化了关键路径并确保 0 WNS，请解释什么是 WNS，以及多周期约束和流水线优化是如何影响时序的。

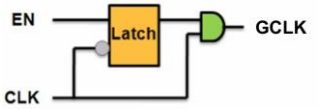
WNS：芯片所有时序路径里最差的（最负的）Slack 值。如果 WNS < 0，说明至少有一个路径无法满足时序要求（setup violation）。

默认 STA 假设信号必须在一个时钟周期内完成传播，但有些路径设计上就是允许多个周期完成（比如跨时钟域接口、流水线中非关键控制路径）。因此用 SDC 命令 set_multicycle_path 修改该路径的 Required Time，让 STA 分析时有更宽松的限制。不过必须确认功能设计上允许多周期，否则会引入功能错误。

流水线：在长的组合逻辑路径中插入寄存器，减少 r2r 路径中的组合逻辑延迟。

在功耗优化中，你提到用时钟门控降低了 50%的功耗，请描述一下你在 RTL 级是如何插入时钟门控的。

首先在 rtl 代码中加入 enable 信号，然后在综合脚本中加入指令自动插入时钟门控单元。也可以在 rtl 代码中手动例化 ICG 单元



你采用了四核环形拓扑，为什么选择环形而不是总线或交叉开关？

在 1D 向量计算加速器中，四核主要做相邻数据交换，所以环形拓扑能在保持较低面积和功耗的前提下提供足够的带宽和可扩展性，比总线性能好、比交叉开关成本低，是更符合我们设计约束的方案。

在后端布局布线中，如果遇到严重的拥塞问题，你会怎么处理？

(a) Floorplan 调整

增加宏单元之间的通道宽度（channel spacing）;

调整 I/O 或宏单元位置，让关键走线更短;

合理预留时钟树和电源网络空间。

(b) Cell Placement 优化

降低热点区域的 cell density（目标 <70%）;

启用工具的 congestion-driven placement;

在热点区放置 blockages（soft/hard）引导工具疏散单元。

(c) Routing 层面

允许使用更高层金属做信号布线;

调整 non-default routing rules (NDR) 提高关键线间距;

你提到在 SHA-256 中通过流水线化减少了计算周期 50%，请具体描述你是如何决定在哪些位置插入流水线级的。

将 word expansion 和 SHA256 运算放到同一个周期完成，使得最终周期数降低一半。

减少冗余寄存器：

原来：w[64] 64 个元素的 array 用来计算 word expansion。实际上计算 word expansion 只需要最近 16 个 word，不用存储 64 个元素。->减小了资源占用，提高性能。

对高 fanout net 进行 buffer insertion 或拆分。

simple dual port memory

```
module dual_port_ram #(
   parameter DATA_WIDTH=32, parameter ADDR_WIDTH=32)
(
   input logic wr_clk,
   input logic wr_en,
   input logic[DATA_WIDTH-1:0] write_data,
   input logic[ADDR_WIDTH-1:0] write_addr,

   input logic rd_en,
```

```
   input logic[ADDR_WIDTH-1:0] read_addr,
   output logic[DATA_WIDTH-1:0] read_data);

   // Two dimensional memory array
   logic[DATA_WIDTH-1:0] mem[2**ADDR_WIDTH-1:0];

   // Synchronous write
   always_ff@(posedge wr_clk) begin

      if(wr_en) mem[write_addr]  <= write_data;
      if (rd_en) read_data <= mem[read_addr];
   end

endmodule:dual_port_ram
```

synchronous fifo

```
module sync_fifo#(
  parameter DATA_WIDTH = 32,
  parameter FIFO_DEPTH = 16)
(
  input  logic clk,
  input  logic reset,
  input  logic wr_en,
  input  logic rd_en,
  input  logic [DATA_WIDTH-1:0] data_in,
  output logic [DATA_WIDTH-1:0] data_out,
  output logic fifo_full,
  output logic fifo_empty
);

 // Local parameter to set address width based on FIFO DEPTH
 localparam ADDR_WIDTH = $clog2(FIFO_DEPTH);

 // internal register declaration
 logic [ADDR_WIDTH:0]  wr_ptr;
 logic [ADDR_WIDTH:0]  rd_ptr;

 // Increment write pointer each time wr_en is '1'
 always_ff@(posedge clk,posedge reset) begin
   if(reset) begin
     wr_ptr <= 0;
   end
   else begin
    if(wr_en && !fifo_full) begin
```

```
      wr_ptr <= wr_ptr + 1;
     end
   end
 end


 // Increment read pointer each time rd_en is '1'
 always_ff@(posedge clk,posedge reset) begin
   if(reset) begin
     rd_ptr <= 0;
   end
   else begin
    if(rd_en && !fifo_empty) begin
     rd_ptr <= rd_ptr + 1;
     end
   end
 end

  // The FIFO is empty when both read and write pointers point to the same
location
  assign fifo_empty =  (wr_ptr == rd_ptr) ? 1 : 0;

  // Fifo is full when wr_ptr - rd_ptr = 2^address_width.
  // In that case, the Lower address bits are identical, but the MSB address bit
is different.
  assign fifo_full  =  ((wr_ptr[ADDR_WIDTH] != rd_ptr[ADDR_WIDTH])
               && (wr_ptr[ADDR_WIDTH-1:0] == rd_ptr[ADDR_WIDTH-1:0])) ? 1 : 0;

 // Instantiate FIFO Memory
 dual_port_ram #(
   .DATA_WIDTH(DATA_WIDTH),
    .ADDR_WIDTH(ADDR_WIDTH))
 fifo_memory(
  .write_addr(wr_ptr),
  .read_addr(rd_ptr),
  .write_data(data_in),
  .read_data(data_out),
  .wr_en(wr_en && !fifo_full),
  .rd_en(rd_en && !fifo_empty),
  .wr_clk(clk),
 );
endmodule:sync_fifo
```

simple dual port dual clock memory

```
module dual_port_ram #(
   parameter DATA_WIDTH=32, parameter ADDR_WIDTH=32)
(
   input logic wr_clk,
   input logic wr_en,
   input logic[DATA_WIDTH-1:0] write_data,
   input logic[ADDR_WIDTH-1:0] write_addr,

   input logic rd_en,
   input logic rd_clk;
   input logic[ADDR_WIDTH-1:0] read_addr,
   output logic[DATA_WIDTH-1:0] read_data);

   // Two dimensional memory array
   logic[DATA_WIDTH-1:0] mem[2**ADDR_WIDTH-1:0];

   // Synchronous write
   always_ff@(posedge wr_clk) begin

      if(wr_en) mem[write_addr]  <= write_data;

   end
   always_ff@(posedge rd_clk) begin

      if(rd_en) read_data  <= mem[read_addr];

   end

endmodule:dual_port_ram
```

shift register

```
module shift_register#(
 parameter integer WIDTH=32,
 parameter integer NUM_OF_STAGES=2,
 parameter logic[WIDTH-1:0] RESET_VALUE=0)
(
  input logic clk, reset,
  input logic [WIDTH-1:0] d,
```

```
  output logic [WIDTH-1:0] q
);

 logic[WIDTH-1:0] r[NUM_OF_STAGES-1:0];
 always_ff@(posedge clk, posedge reset) begin
  if(reset == 1) begin
    for(int i=0; i<NUM_OF_STAGES; i++) begin
        r[i] <= RESET_VALUE;
    end
  end
  else begin
    r[0] <= d;
    for(int i=0; i<(NUM_OF_STAGES-1); i++) begin
      r[i+1] <= r[i];
    end
  end
 end
 assign q = (reset==1) ? RESET_VALUE : r[NUM_OF_STAGES-1];
endmodule: shift_register
```

asynchronous fifo

```
//`include "dual_port_ram.sv"
//`include "shift_register.sv"
module async_fifo#(
  parameter DATA_WIDTH = 32,    // width of each data element in FIFO Memory
  parameter FIFO_DEPTH = 16)    // Number of locations in FIFO Memory
(
  input  logic wr_clk, rd_clk, // Write and Read Clocks
  input  logic reset, // Common reset
  input  logic wr_en, // write enable, if wr_en == 1, data gets written to FIFO
Memory
  input  logic rd_en, // read_enable, if rd_en == 1, data gets read out from
FIFO Memory
  input  logic [DATA_WIDTH-1:0] data_in,  // Input data to be written to FIFO
Memory
  output logic [DATA_WIDTH-1:0] data_out, // Data read out from FIFO Memory
  output logic fifo_full, // Indicates FIFO is full and there are no locations
inside FIFO memory for further writes
  output logic fifo_empty, // Indicates FIFO is empty and there are no data
available inside FIFO memory for reading
  output logic fifo_almost_full, // One cycle early indication of FIFO_FULL (fifo
```

```
is not full yet, it will be next cycle)
  output logic fifo_almost_empty // One cycle early indication of FIFO_EMPTY
(fifo is not empty yet, it will be next cycle)
);

 // Local parameter to set address width based on FIFO DEPTH
 localparam ADDR_WIDTH = $clog2(FIFO_DEPTH);

 // Internal register declaration
 // Note:
 // wr_ptr is binary counting write address pointer
 // wr_ptr_gray is a gray counting write address pointer convrted from wr_ptr
binary counting write address pointer
 // wr_ptr_gray2 is output of 2-FF synchronizer (which is output of 2 stage shift
register). Input to this synchronizer is wr_ptr_gray
 // wr_ptr_binary2 is binary countng write address pointer generated after
converting from wr_ptr_gray2 gray counting write address pointer
 // rd_ptr is binary counting read address pointer
 // rd_ptr_gray is a gray counting read address pointer convrted from wr_ptr
binary counting read address pointer
 // rd_ptr_gray2 is output of 2-FF synchronizer (which is output of 2 stage shift
register). Input to this synchronizer is rd_ptr_gray
 // rd_ptr_binary2 is binary countng read address pointer generated after
converting from rd_ptr_gray2 gray counting read address pointer
 logic [ADDR_WIDTH:0] wr_ptr, wr_ptr_gray, wr_ptr_gray2, wr_ptr_binary2;
 logic [ADDR_WIDTH:0] rd_ptr, rd_ptr_gray, rd_ptr_gray2, rd_ptr_binary2;
 logic t_fifo_empty, t_fifo_full;




 // Step-1 : Increment write pointer each time wr_en is '1' (binary counter)
 // If reset == 1 then assign 0 to wr_ptr
 always_ff@(posedge wr_clk,posedge reset) begin
   // Student to add code
    if(reset) begin
     wr_ptr <= 0;
   end
   else begin
    if(wr_en && !fifo_full) begin
      wr_ptr <= wr_ptr + 1;
     end
   end
 end
```

```
 // Step-2 : Convert write pointer to gray value from binary write pointer value
before
 // sending write pointer to rd_clk domain through 2-FlipFlip synchronizer
 // use binary_to_gray function
 assign wr_ptr_gray = binary_to_gray(wr_ptr); // Student to add code


 // Step-3 : Increment read pointer each time rd_en is '1' (binary counter)
 // If reset == 1then assign 0 to rd_ptr
 always_ff@(posedge rd_clk,posedge reset) begin
     // Student to add code
    if(reset) begin
     rd_ptr <= 0;
   end
   else begin
    if(rd_en && !fifo_empty) begin
     rd_ptr <= rd_ptr + 1;
     end
   end

 end


 // Step-4 : Convert read pointer to gray value from binary read pointer value
before
 // sending read pointer to wr_clk domain through 2-FlipFlip synchronizer
 // use binary_to_gray function
 assign rd_ptr_gray = binary_to_gray(rd_ptr); // Student to add code


 // Step-5 : Geneate fifo empty flag
 // The FIFO is empty when both read and write pointers point to the same location
 assign t_fifo_empty = (rd_ptr == wr_ptr_binary2) ? 1 : 0; // Student to add
code
// can instead use
// assign t_fifo_empty = rd_ptr_gray == wr_ptr_gray2;

 // Step-6 : Assert output signal fifo_almost_empty
 // FIFO Almost empty is generated simulataneously when the very last data
available in fifo is read
 // hence it is named as fifo almost empty. In another words, one cycle before
fifo is actually empty
```

```
 //assign fifo_almost_empty = (((rd_ptr + 1'b1) == wr_ptr_binary2) && rd_en ==
1'b1) ? 1 : 0;// Student to add code
 assign fifo_almost_empty = t_fifo_empty;

 // Step-7 : Generate fifo full flag
 // FIFO is full when wr_ptr - rd_ptr = 2^address_width.
 // In that case, the Lower address bits are identical, but the MSB address bit
is different.
 assign t_fifo_full  = ((wr_ptr[ADDR_WIDTH] != rd_ptr_binary2[ADDR_WIDTH])
                && (wr_ptr[ADDR_WIDTH-1:0] == rd_ptr_binary2[ADDR_WIDTH-1:0])) ?
1 : 0;  // Student to add code

//can instead use
//assign t_fifo_full = (wr_ptr_gray[ADDR_WIDTH] != rd_ptr_gray2[ADDR_WIDTH]) &&
(wr_ptr_gray[ADDR_WIDTH-1]       !=       rd_ptr_gray2[ADDR_WIDTH-1])       &&
(wr_ptr_gray[ADDR_WIDTH-2:0] == rd_ptr_gray2[ADDR_WIDTH-2:0]);


 // Step-8 : Assert almost full flag
 // FIFO Almost full is generated simulataneously when the very last location in
fifo is written with data
 // hence it is named as fifo almost full. In another words, one cycle before
the fifo is actually full
 //assign fifo_almost_full = (((rd_ptr_binary2 + {(ADDR_WIDTH){1'b1}}) == wr_ptr)
&& wr_en == 1'b1) ? 1 : 0; // Student to add code
 assign fifo_almost_full = t_fifo_full;

 // Step-9 : Instantiate FIFO Memory (Upon synthesis this will result in dual
port distributed RAM since read from memory is asynchronous)
 // Note : when connecting below wr_en and rd_end remember to do logical-and
with !fifo_full and !fifo_empty respectively
 // if !fifo_full anding with wr_en is added, to prevent writing of datain to
dual_port_ram (fifo memory) when fifo is full.
 // if !fifo_empty anding with rd_en is added, to prevent reading of data from
dual_port_ram (fifo memory) when fifo is empty.
 //.wr_en(wr_en && !fifo_full),
 //.rd_en(rd_en && !fifo_empty),
 // dual port ram is synchronous write and asynchronous read. It can perform
simultaneous write and read operations.
 // Simulatneous write and read memory is required, since data can be pushed
(write) into FIFO and POPPED (read) out of FIFO at same time
 dual_port_ram #(
   .DATA_WIDTH(DATA_WIDTH),
   .ADDR_WIDTH(ADDR_WIDTH))
```

```
 fifo_memory_inst(
     // Student to add code
  .write_addr(wr_ptr),
  .read_addr(rd_ptr),
  .write_data(data_in),
  .read_data(data_out),
  .wr_en(wr_en && !fifo_full),
  .rd_en(rd_en && !fifo_empty),
  .wr_clk(wr_clk),
  .reset(reset)

 );


 // Step-10 : Synchronize wr_ptr to rd_clk domain
 // This is done to synchronize wr_ptr to rd_clk domain, and in rd_clk domain, output of this synchronizer will be used to
 // compute fifo empty flag.
 shift_register #(
  .WIDTH(ADDR_WIDTH+1),
  .NUM_OF_STAGES(2))
 wr_ptr_synchronizer_inst(
  .clk(rd_clk),
  .reset(reset),
  .d(wr_ptr_gray),
  .q(wr_ptr_gray2)
 );


 // Step-11 : Convert synchronized write pointer gray value available from Step-10, back to binary value
 // Prior to generation for fifo empty flag, synchronized gray write pointer value is converted first to binary write pointer value
 // use gray_to_binary function
 assign wr_ptr_binary2 = gray_to_binary(wr_ptr_gray2); // Student to add code


 // Step-12 : Synchronize rd_ptr to wr_clk domain
 // Instantiate shift register to perform 2 stage FlipFlop synchronization (also known as 2-FF synchronizer)
 // This is done to synchronize rd_ptr to wr_clk domain, and in wr_clk domain, output of this synchronizer will be used to
 // compute fifo full flag.
 shift_register #(
```

```
  // Student to add code here similar to wr_ptr to rd_clk domain synchronizatio
add code here for rd_ptr to wr_clk synchronization
  // Note : This is a 2 Flip flip stage synchronization. So NUM_OF_STAGES
parameter for shift register should be 2
  // Remember to connect clk of shift register to "wr_clk", since rd_ptr_gray is
sent to wr_clk domain through this synchronizer
  .WIDTH(ADDR_WIDTH+1),
  .NUM_OF_STAGES(2))
 rd_ptr_synchronizer_inst(
  .clk(wr_clk),
  .reset(reset),
  .d(rd_ptr_gray),
  .q(rd_ptr_gray2)
 );


 // Step-13 : Convert synchronized read pointer gray value available from Step-
12, back to binary value
 // Prior to generation for fifo full flag, synchronized gray read pointer value
is converted first to binary read pointer value
 //  use gray to binary function
 assign rd_ptr_binary2 = gray_to_binary(rd_ptr_gray2); // Student to add code


 // Step-14 : Delay fifo almost empty (t_fifo_empty) by 1 clock cycle to generate
fifo_empty output signal
 // This is done to generate fifo_empty output signal after the last available
data in FIFO is read out
 shift_register #(
  .WIDTH(1),
  .NUM_OF_STAGES(1),   // Note : Here 2-FF synchronizer is not the intent. Only
1 cycle delayed version of t_fifo_empty is created. Hence NUM_OF_STAGES is '1'
  .RESET_VALUE(1))  // Note : RESET_VALUE is set to '1', since by default out of
reset, FIFO is in empty state.
 fifo_empty_inst(
  .clk(rd_clk),
  .reset(reset),
  .d(t_fifo_empty),
  .q(fifo_empty)
 );

 // Step-15 : Delay fifo almost full (t_fifo_full) by 1 clock cycle to generate
fifo_full output signal
```

```
 // This is done to generate fifo_full output signal after the last available location in FIFO is written with a data
 // Note : Here 2-FF synchronizer is not the intent. Only 1 cycle delayed version of t_fifo_full is created. Hence NUM_OF_STAGES should be set to '1'
 // RESET_VALUE should be set to '0', since by default out of reset, FIFO is not in full state.
 shift_register #(
    // Student to add code here. Similar to fifo_empty code above, add code for fifo_full
    .WIDTH(1),
  .NUM_OF_STAGES(1),   // Note : Here 2-FF synchronizer is not the intent. Only 1 cycle delayed version of t_fifo_empty is created. Hence NUM_OF_STAGES is '1'
  .RESET_VALUE(1))  // Note : RESET_VALUE is set to '1', since by default out of reset, FIFO is in empty state.
 fifo_full_inst(
  .clk(wr_clk),
  .reset(reset),
  .d(t_fifo_full),
  .q(fifo_full)
 );

 // finction to convert binary to gray function
 function automatic [ADDR_WIDTH:0] binary_to_gray(logic [ADDR_WIDTH:0] value);
   begin
     binary_to_gray[ADDR_WIDTH] = value[ADDR_WIDTH];
     for(int i=ADDR_WIDTH; i>0; i = i - 1)
       binary_to_gray[i-1] = value[i] ^ value[i - 1];
    end
 endfunction

 // function to convert gray to binary
 function logic[ADDR_WIDTH:0] gray_to_binary(logic[ADDR_WIDTH:0] value);
  begin
     logic[ADDR_WIDTH:0] l_binary_value;
     l_binary_value[ADDR_WIDTH] = value[ADDR_WIDTH];
     for(int i=ADDR_WIDTH; i>0; i = i - 1) begin
      l_binary_value[i-1] = value[i-1] ^ l_binary_value[i];
     end
     gray_to_binary = l_binary_value;
  end
 endfunction

endmodule:async_fifo
```

三分频

```
Module div3 (
Input logic clk,
Input logic rstn,
Output logic clk_3
);
Logic [1:0] cnt;
Logic clk1;
Logic clk2;

always_ff @(posedge clk) begin
    if (!rstn) begin
        cnt <= ‘0;
    end
    else if (cnt == 2) begin
        cnt <= ‘0;
    end
    else begin
        cnt <= cnt + 1;
    end
end


Always_ff @(posedge clk) begin
    If (!rstn) begin
        Cnt <= 0;
        Clk1 <= 1’b0;
    End
    Else begin
        if (cnt == 0) begin
            Clk1 <= ~clk1;
        end
    End
end

always_ff @(negedge clk) begin
    if (!rstn)
        clk2 <= 1’b0;
    else begin
        if (cnt==2)
            clk2 <= ~clk2;
    end
end
```

```
assign clk3 = clk2 ^ clk1;

endmodule
```