DDR3专题篇

1 DDR3基础知识

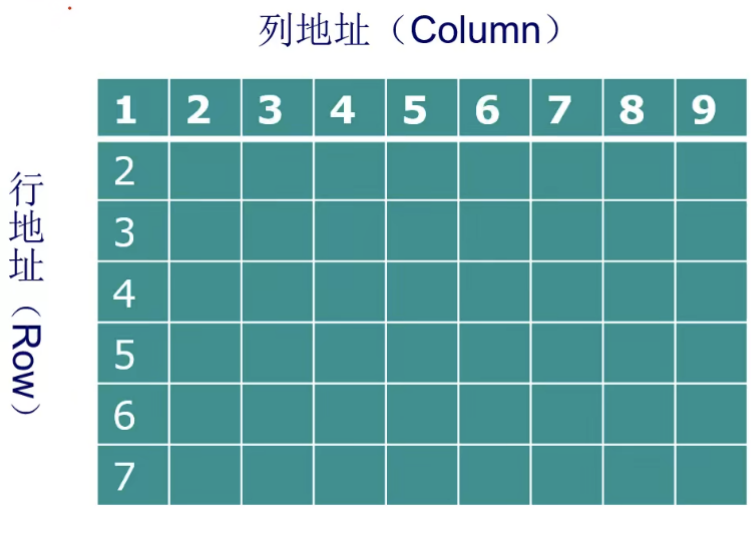
1.1 DDR3概述

DDR3 SDRAM（Double Data Rate 3 Synchronous Dynamic RAM），即第三代双倍速率**同步动态随机存储器。**

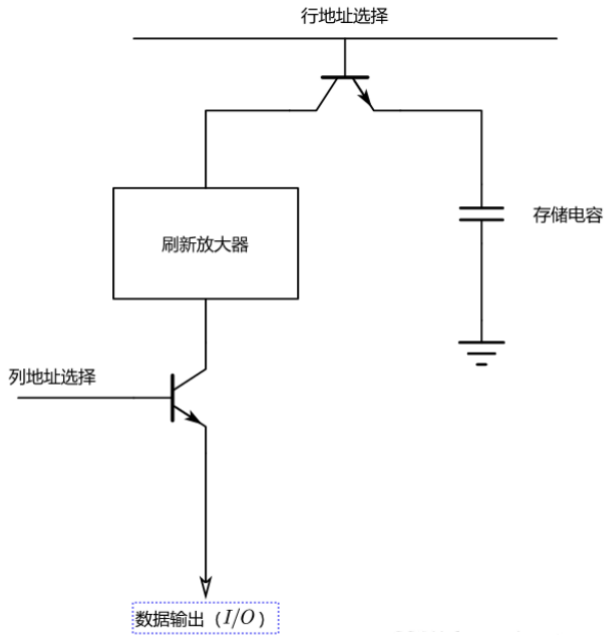
- 双倍速率：DDR3在时钟的上升沿和下降沿都可以读取数据；
- 同步：DDR3读取数据是按时钟同步的，它的时钟频率与CPU前端总线的系统时钟频率相同，并且内部的命令发送与数据的传输都以它为基准；
- 动态：DDR3中的数据无法掉电保存，同时如果想要保存数据，需要对数据进行周期性的刷新；
- 随机存取：可以随机操作任意地址的数据，数据不是线性依次存储的，可以自由指定要读写的地址。

1.2 SDRAM的基础结构

DRAM中最核心的结构为**逻辑Bank**（Logical Bank，L-Bank），它是SDRAM内部空间划分的片区。每一个L-Bank都可以想象成一个巨大的“方格”矩阵，每一个方格中都可以存储和芯片位宽等长的数据，每一个方格都有对应的行编码和对应的列编码。如下图所示：



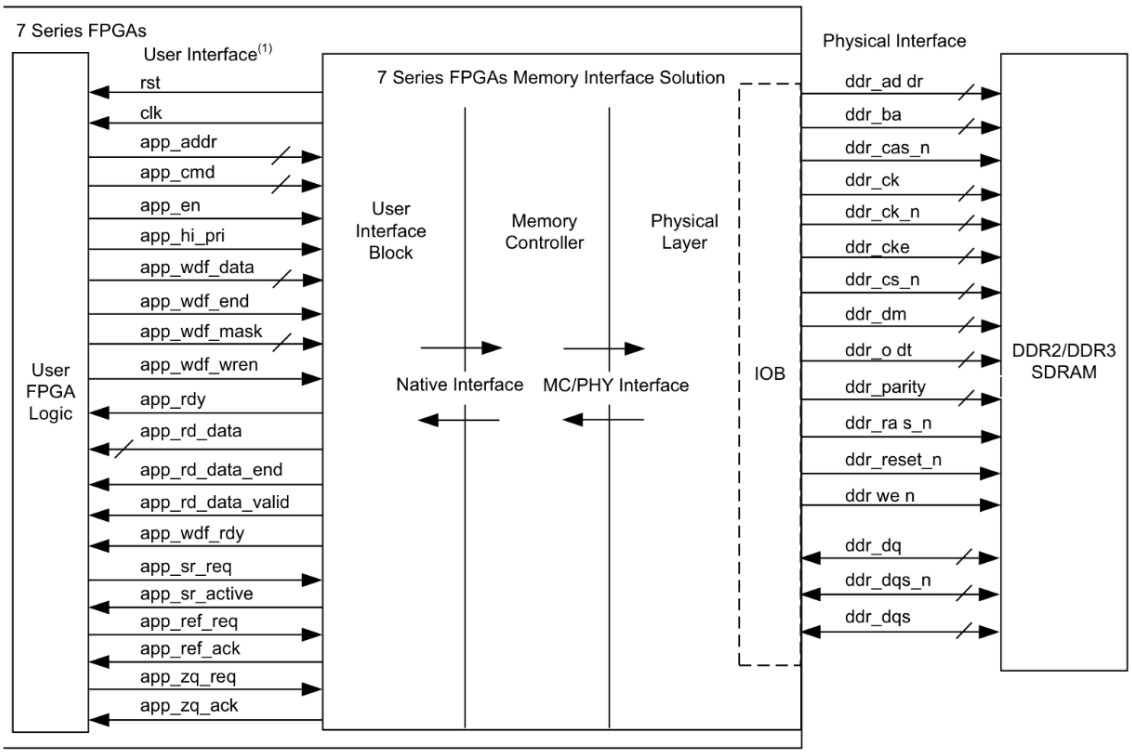
SDRAM中存储数据是通过电容实现的。下面是一个bit的简单存储原理示意图。通过读取电容上的电量来判断该位为0还是1，并且需要不断刷新重复写入才能保证数据不丢失。



2 MIG基础知识

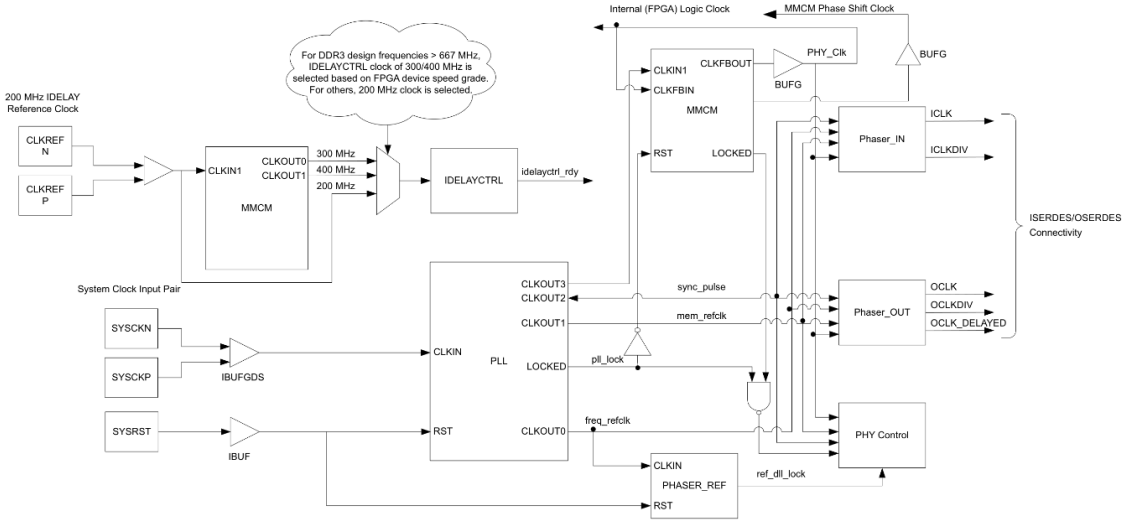
2.1 MIG框架

由于DDR3的控制时序较为复杂，Xilinx厂商专门提供MIG的IP 供用户使用，该IP可以大大简化用户使用DDR的难度，提高开发效率；MIG结构图如下图所示：



可以看出当我们使用MIG IP的时候，用户就不需要直接对DDR3控制，我们只需要对MIG进行读写控制就可以完成对DDR3的读写控制；

2.2 MIG时钟架构



reference clock：供idelayctrl使用，通常需要200MHz，可以外部提供（single-end，differential），也可以内部提供（no buffer，use system clock），对DDR频率>667MHz时，参考时钟需要300/400MHz（也可从内部产生）；

system clock：mig的系统时钟（主时钟），可以外部提供（single-end，differential），也可以内部提供（no buffer）。

ui_clk：供用户接口端使用，对应图中的internal (FPGA) logic clock，具体时钟频率，由PHY to Controller Clock Ratio决定（4:1 or 2:1）。

2.3 MIG配置

内存控制器参数设置：

Clock Period:800M 即数据速度 1600M

PHY to Controller Clock Ration:DDR PHY 的时钟到用户时钟的比率 4：1=800M:200M 即用户接口时钟 200M

Vccaux_io:当 DDR 工作于高性能模式下，必须设置 2.0V

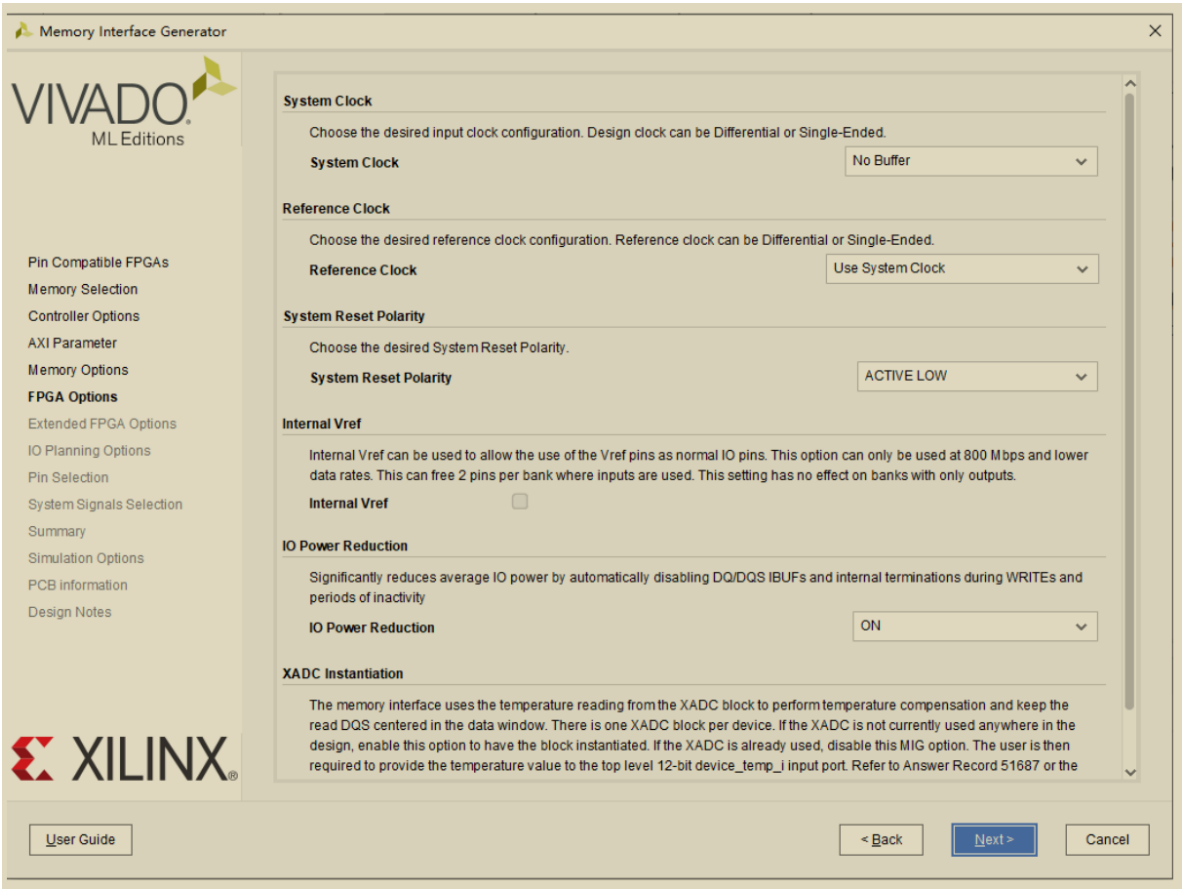
Memory Type:DDR3 SDRAM 的内存类型有：内存颗粒（component）、内存条（RDIMMs、UDIMMs、SODIMMs）

Memory Part:DDR3 SDRAM 的内存型号，如果列表里没有，则需要自己创建新的组件。根据内存颗粒的数据手册填写（tcke、tfaw、tras 等参数）

Memory Voltage：DDR 的工作电压，根据硬件电路而定，DDR3 是 1.5V DDR3L 是 1.35V(DDR3L 也可以兼容工作在 1.5V)

Data Width：一片 DDR3 数据位宽为 16bit，如果你是两片则为 32bit

Number of Bank Machines:bank mechine 管理，DDR3 bank，这个值越大 DDR 的访问效率越高，同时占用的资源也更多，选择的越小占用资源少，但是会影响效率。



System Clock:这里选择 No Buffer,因为我们准备用一个 PLL 倍频到 200M 给 MIG 提供 sytem 时钟,这个时钟必须和Memory Option(内存选项)中 Input Clock Perild 时钟一致。

Reference Clock:参考时钟,必须设置 200M，如果 System Clock 也是 200M 则可以选择 Use system colck 这样外围设计可以少 1 根时钟线(其实就是 MIG IP 源码里面互联在一起了)。

System Reset Polarity:设置 DDR 是高电平复位还是低电平复位。

Internal Vref:设置 DDR 是用 FPGA 内部的参考电平还是外部参考电平，使用 FPGA 内部参考电平仅可以是 800Mbps以下。

IO Power Reduction:支持功耗降低模式。

XADC:设置支持 XADC 可以用于温度补偿。

2.4 MIG读写时序

命令控制端口:

app_cmd[2:0]：当该信号为3'b0时，表示写操作，为3'b001时，表示读操作。

app_addr[27:0]:读写地址信号

app_en:命令的使能信号

app_rdy:命令有效信号

写数据端口:

app_wdf_data[127:0]:写入数据信号

app_wdf_end:突发结束信号

app_wdf_wren:数据的写使能

app_wdf_rdy:数据的有效信号

app_wdf_mask[15:0]:数据掩码，每个bit针对一个字节

读数据端口：

app_rd_data[127:0]:读数据信号

app_rd_data_end:数据突发结束信号

app_rd_data_valid:数据的有效信号

时钟端口：

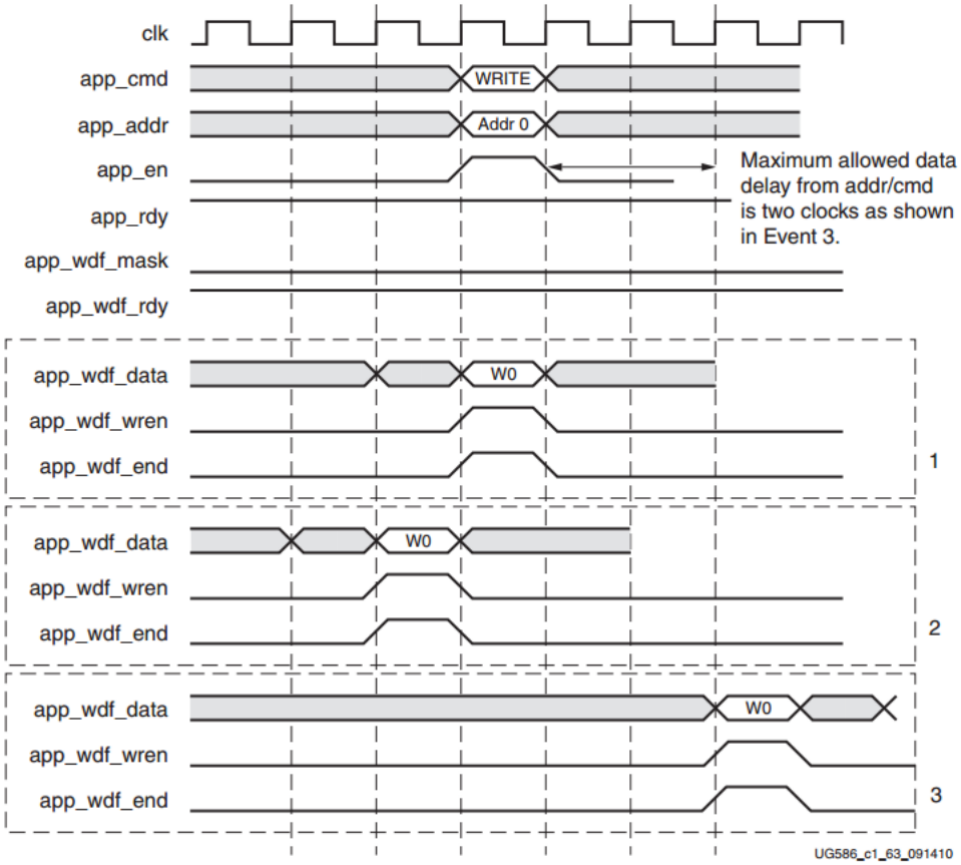
ui_clk:用户时钟，由MIG产生的时钟

ui_clk_sync_rst:用户端的复位信号，高电平复位,MIG产生

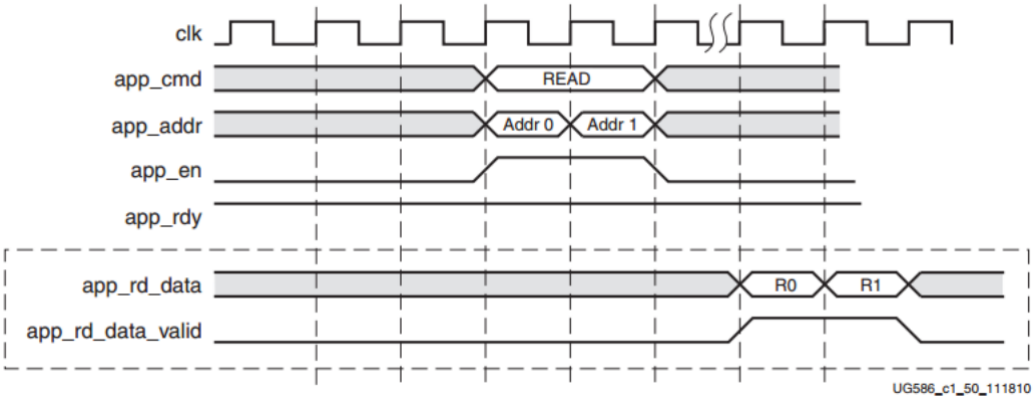
sys_clk:IP核的系统时钟，FPGA产生（200Mhz）

sys_rst:IP核的复位信号，FPGA产生，低电平复位。

写时序



读时序：



3 AXI4总线

3.1 AXI4总线概述

AXI(Advanced eXtensible Interface) 是 ARM 公司推出的一种高性能、低成本、可扩展的高速总线接口。它被广泛应用于数字系统中，尤其是嵌入式系统中。AXI 接口具有高度的灵活性和可扩展性，可以适应不同的应用场景和系统需求。

在 XIINX FPGA 的软件工具 vivado 以及相关 IP 中有支持三种 AXI 总线，拥有三种 AXI 接口，当然用的都是AXI 协议。其中三种 AXI 总线分别为：

AXI4：（For high-performance memory-mapped requirements.）主要面向高性能地址映射通信的需求，是面向地址映射的接口，允许最大 256 次的数据突发传输；

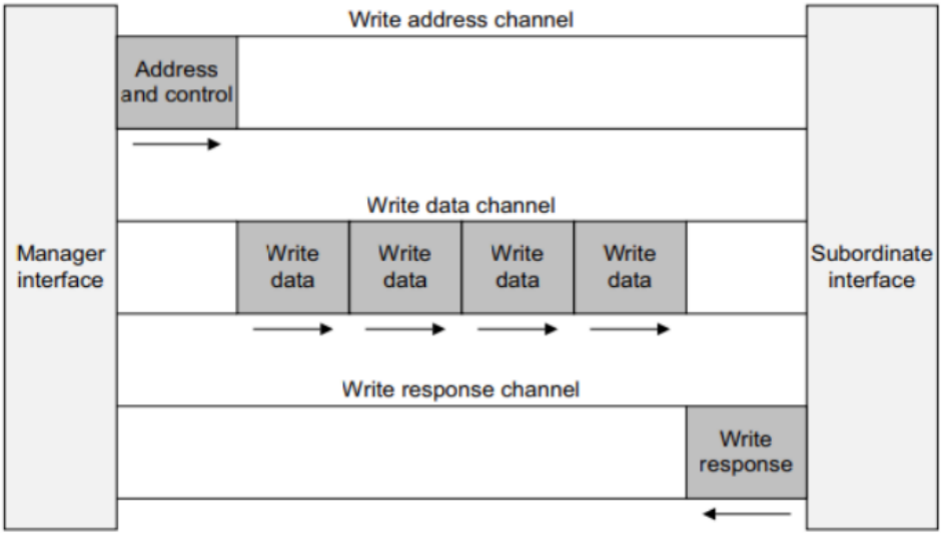
AXI4-Lite：（For simple, low-throughput memory-mapped communication ）是一个轻量级的地址映射单次传输接口，占用很少的逻辑单元。

AXI4-Stream：（For high-speed streaming data.）面向高速流数据传输；去掉了地址项，允许无限制的数据突发传输规模。

3.2 AXI4读写时序

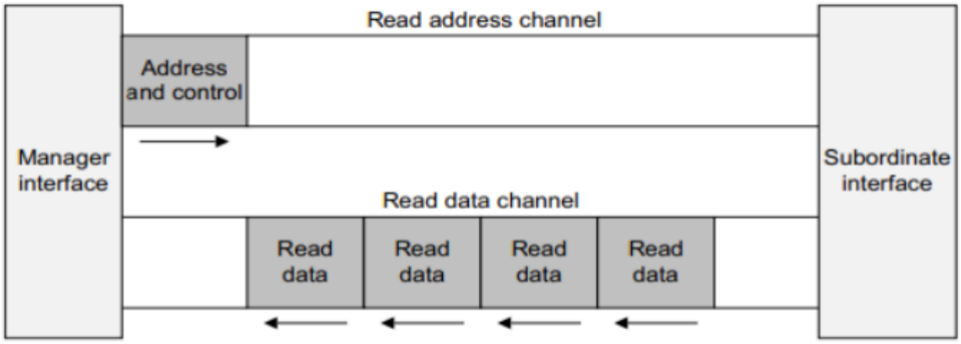
对于AXI4接口，它由五个独立的通道构成，读地址通道、读数据通道、写地址通道、写数据通道、写响应通道。

写操作通道：



主机向从机写入数据：主机在写地址通道中发出了地址以及控制信号，紧接着在写数据通道中，主机向从机写入数据，最后，从机在写响应通道中向主机发出应答信号。

读操作通道：



从机将数据读给主机：主机在读地址通道中发出了地址以及控制信号，紧接着在读数据通道中，从机把数据读出来。

AXI-4 总线信号功能

信号	方向	描述
ACLK	时钟源	全局时钟信号
ARESETn	复位源	全局复位信号，低有效

写地址通道：

AWADDR	主机 to 从机	写地址，给出一次写突发传输的写地址
AWLEN	主机 to 从机	AWLEN[7:0]决定写传输的突发长度。AXI3 只支持 1~16 次的突发传输（Burst_length=AxLEN[3:0]+1），AXI4 扩展突发长度支持 INCR 突发类型为 1~256 次传输，对于其他的传输类型依然保持 1~16 次突发传输（Burst_Length=AxLEN[7:0]+1）。 burst 传输具有如下规则： wraping burst ,burst 长度必须是 2,4,8,16 burst 不能跨 4KB 边界 不支持提前终止 burst 传输
AWSIZE	主机 to 从机	写突发大小，给出每次突发传输的字节数支持 1、2、4、8、16、32、64、128
AWBURST	主机 to 从机	突发类型： 2'b00 FIXED：突发传输过程中地址固定，用于 FIFO 访问 2'b01 INCR ：增量突发，传输过程中，地址递增。增加量取决 AxSIZE 的值。 2'b10 WRAP：回环突发，和增量突发类似，但会在特定高地址的边界处回到低地址处。回环突发的长度只能是 2,4,8,16 次传输，传输首地址和每次传输的大小对齐。最低的地址整个传输的数据大小对齐。回环边界等于（AxSIZE*AxLEN） 2'b11 Reserved
AWLOCK	主机 to 从机	总线锁信号，可提供操作的原子性
AWCACHE	主机 to 从机	内存类型，表明一次传输是怎样通过系统的
AWPROT	主机 to 从机	保护类型，表明一次传输的特权级及安全等级
AWQOS	主机 to 从机	AXI4 增加的质量服务信号，QoS 主要针对不同写/读事务的优先级，如果不使用，建议设置为 default value 4'b0000
AWREGION	主机 to 从机	区域标志，能实现单一物理接口对应的多个逻辑接口
AWUSER	主机 to 从机	AXI4 增加的用户自定义信号，一般不使用
AWVALID	主机 to 从机	有效信号，表明此通道的地址控制信号有效
AWREADY	从机 to 主机	表明"从"可以接收地址和对应的控制信号

写数据通道：

信号名	方向	描述
WID	主机 to 从机	一次写传输的 ID tag,对于 AXI4 总线已经取消了 WID,所以 AXI4 数据必须是 order 有序的
WDATA	主机 to 从机	写数据
WSTRB	主机 to 从机	WSTRB[n:0]对应于对应的写字节，WSTRB[n]对应 WDATA[8n+7:8n]。WVALID 为低时，WSTRB 可以为任意值，WVALID 为高时，WSTRB 为高的字节线必须指示有效的数据。
WLAST	主机 to 从机	表明此次传输是最后一个突发传输
WUSER	主机 to 从机	用户自定义信号
WVALID	主机 to 从机	写有效，表明此次写有效
WREADY	从机 to 主机	表明从机可以接收写数据

写响应通道

信号名	方向	描述
BID	从机 to 主机	写响应 ID tag
BRESP	从机 to 主机	写响应，表明写传输的状态
BUSER	从机 to 主机	用户自定义
BVALID	从机 to 主机	写响应有效
BREADY	主机 to 从机	表明主机能够接收写响应

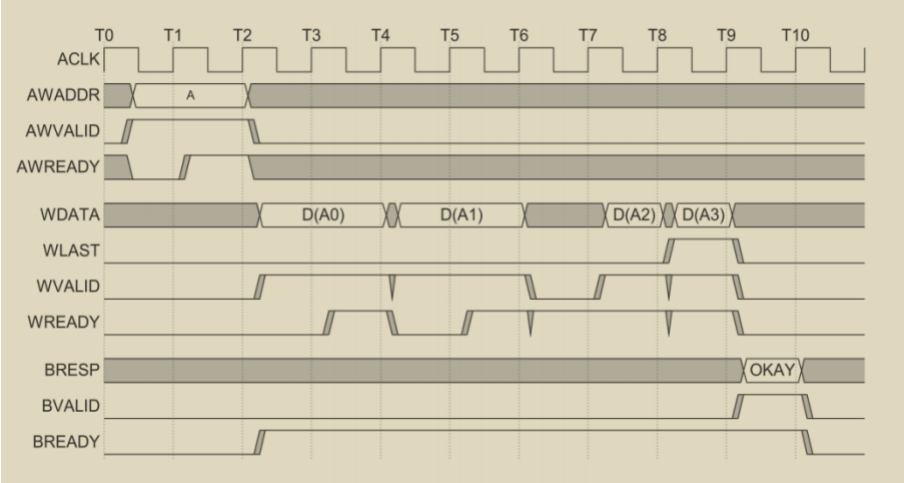
读地址通道信号

信号	方向	描述
ARID	主机 to 从机	读地址 ID，用来标志一组读信号
ARADDR	主机 to 从机	读地址，给出一次读突发传输的读地址
ARLEN	主机 to 从机	ARLEN[7:0]决定读传输的突发长度。AXI3 只支持 1~16 次的突发传输（Burst_length=AxLEN[3:0]+1），AXI4 扩展突发长度支持 INCR 突发类型为 1~256 次传输，对于其他的传输类型依然保持 1~16 次突发传输（Burst_Length=AxLEN[7:0]+1）。 burst 传输具有如下规则： wraping burst ,burst 长度必须是 2,4,8,16 burst 不能跨 4KB 边界 不支持提前终止 burst 传输
ARSIZE	主机 to 从机	读突发大小，给出每次突发传输的字节数支持 1、2、4、8、16、32、64、128
ARBURST	主机 to 从机	突发类型： 2’b00 FIXED：突发传输过程中地址固定，用于 FIFO 访问 2’b01 INCR ：增量突发，传输过程中，地址递增。增加量取决 AxSIZE 的值。 2’b10 WRAP：回环突发，和增量突发类似，但会在特定高地址的边界处回到低地址处。回环突发的长度只能是 2,4,8,16 次传输，传输首地址和每次传输的大小对齐。最低的地址整个传输的数据大小对齐。回环边界等于（AxSIZE*AxLEN） 2’b11 Reserved
ARLOCK	主机 to 从机	总线锁信号，可提供操作的原子性
ARCACHE	主机 to 从机	内存类型，表明一次传输是怎样通过系统
ARPROT	主机 to 从机	保护类型，表明一次传输的特权级及安全等级
ARQOS	主机 to 从机	AXI4 增加的质量服务信号，QoS 主要针对不同写/读事务的优先级，如果不使用，建议设置为 default value 4‘b0000
ARREGION	主机 to 从机	区域标志，能实现单一物理接口对应的多个逻辑接口
ARUSER	主机 to 从机	用户自定义信号，一般不使用
ARVALID	主机 to 从机	有效信号，表明此通道的地址控制信号有效
ARREADY	从机 to 主机	表明“从”可以接收地址和对应的控制信号

读数据通道信号

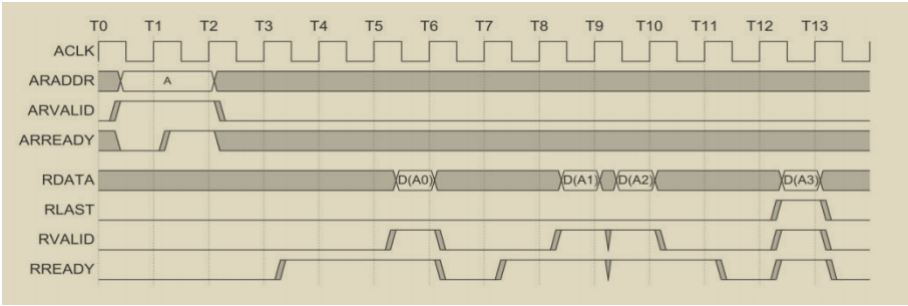
信号名	方向	描述
RID	从机 to 主机	一次读传输的 ID tag
RDATA	从机 to 主机	读数据
RRESP	从机 to 主机	读响应，表明读传输的状态
RLAST	从机 to 主机	表明此次传输是最后一个突发传输
RUSER	从机 to 主机	用户自定义信号
RVALID	从机 to 主机	读有效，表明数据总线上数据有效
RREADY	主机 to 从机	表明主机准备好可以接收数据

写操作时序：



这一过程的开始时，主机发送地址和控制信息到写地址通道中，然后主机发送每一个写数据到写数据通道中。当主机发送最后一个数据时，WLAST 信号就变为高。当设备接收完所有数据之后他将一个写响应发送回主机来表明写事务完成。

读操作时序：

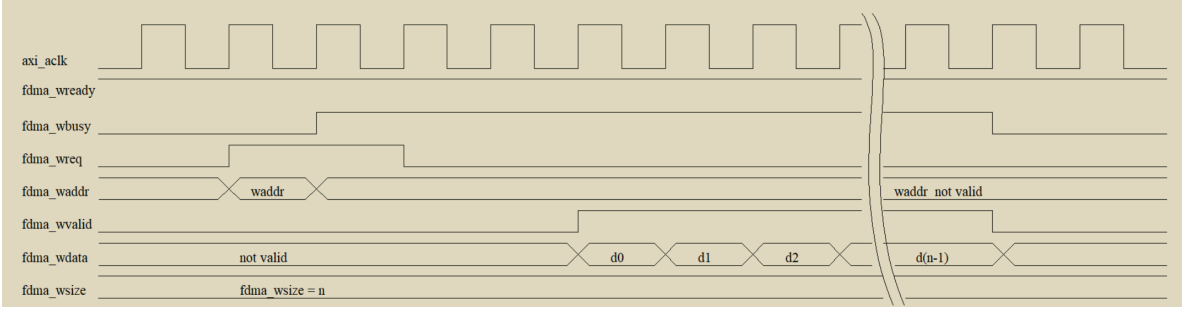


当地址出现在地址总线后，传输的数据将出现在读数据通道上。设备保持 VALID 为低直到读数据有效。为了表明一次突发式读写的完成，设备用 RLAST 信号来表示最后一个被传输的数据。

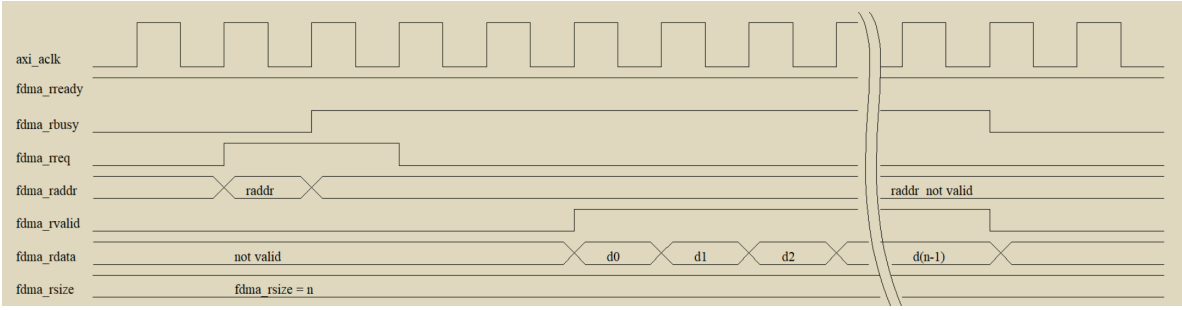
3.3 FMDA读写控制器

FMDA是一个有米联客公司所设计的AXI4_FUL接口控制器，该控制器可以方便我们用户使用AXI4总线来进行读写控制，下面是FMDA读写控制时序图：

FDMA 的写时序



fdma_wready 设置为 1，当 fdma_wbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_wreq=1，同时设置 fdma burst 的起始地址和 fdma_wsize 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_wvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_wvalid 和fdma_wbusy 变为 0。



fdma_rready 设置为 1，当 fdma_rbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_rreq=1，同时设置 fdma burst 的起始地址和 fdma_rsize 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_rvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_rvalid 和 fdma_rbusy变为 0。

FMDA源码：

```
`timescale 1ns / 1ns
module uiFDMA#
(
parameter  integer          M_AXI_ID_WIDTH                = 3     ,
parameter  integer          M_AXI_ID                   = 0         ,
parameter  integer          M_AXI_ADDR_WIDTH           = 32        ,
parameter  integer          M_AXI_DATA_WIDTH           = 128       ,
parameter  integer         M_AXI_MAX_BURST_LEN         = 64
)
(
input   wire [M_AXI_ADDR_WIDTH-1 : 0]      fdma_waddr          ,
input                                       fdma_wareq          ,
input   wire [15 : 0]                      fdma_wsize          ,

output                                      fdma_wbusy          ,

input   wire [M_AXI_DATA_WIDTH-1 :0]       fdma_wdata          ,
output  wire                               fdma_wvalid         ,
input   wire                               fdma_wready         ,

input   wire [M_AXI_ADDR_WIDTH-1 : 0]      fdma_raddr          ,
input                                       fdma_rareq          ,
input   wire [15 : 0]                      fdma_rsize          ,

output                                      fdma_rbusy          ,

output  wire [M_AXI_DATA_WIDTH-1 :0]       fdma_rdata          ,
output  wire                               fdma_rvalid         ,
input   wire                               fdma_rready         ,

input   wire                                M_AXI_ACLK          ,
input   wire                                M_AXI_ARESETN       ,
output  wire [M_AXI_ID_WIDTH-1 : 0]         M_AXI_AWID          ,
output  wire [M_AXI_ADDR_WIDTH-1 : 0]       M_AXI_AWADDR        ,
output  wire [7 : 0]                        M_AXI_AWLEN         ,
output  wire [2 : 0]                        M_AXI_AWSIZE        ,
output  wire [1 : 0]                        M_AXI_AWBURST       ,
output  wire                                M_AXI_AWLOCK        ,
output  wire [3 : 0]                        M_AXI_AWCACHE       ,
output  wire [2 : 0]                        M_AXI_AWPROT        ,
output  wire [3 : 0]                        M_AXI_AWQOS         ,
```

```
output  wire                                M_AXI_AWVALID       ,
input   wire                                M_AXI_AWREADY       ,
output  wire [M_AXI_ID_WIDTH-1 : 0]         M_AXI_WID           ,
output  wire [M_AXI_DATA_WIDTH-1 : 0]       M_AXI_WDATA         ,
output  wire [M_AXI_DATA_WIDTH/8-1 : 0]     M_AXI_WSTRB         ,
output  wire                                M_AXI_WLAST         ,
output  wire                                M_AXI_WVALID        ,
input   wire                                M_AXI_WREADY        ,
input   wire [M_AXI_ID_WIDTH-1 : 0]         M_AXI_BID           ,
input   wire [1 : 0]                        M_AXI_BRESP         ,
input   wire                                M_AXI_BVALID        ,
output  wire                                M_AXI_BREADY        ,
output  wire [M_AXI_ID_WIDTH-1 : 0]         M_AXI_ARID          ,

output  wire [M_AXI_ADDR_WIDTH-1 : 0]       M_AXI_ARADDR        ,
output  wire [7 : 0]                        M_AXI_ARLEN         ,
output  wire [2 : 0]                        M_AXI_ARSIZE        ,
output  wire [1 : 0]                        M_AXI_ARBURST       ,
output  wire                                M_AXI_ARLOCK        ,
output  wire [3 : 0]                        M_AXI_ARCACHE       ,
output  wire [2 : 0]                        M_AXI_ARPROT        ,
output  wire [3 : 0]                        M_AXI_ARQOS         ,
output  wire                                M_AXI_ARVALID       ,
input   wire                                M_AXI_ARREADY       ,
input   wire [M_AXI_ID_WIDTH-1 : 0]         M_AXI_RID           ,
input   wire [M_AXI_DATA_WIDTH-1 : 0]       M_AXI_RDATA         ,
input   wire [1 : 0]                        M_AXI_RRESP         ,
input   wire                                M_AXI_RLAST         ,
input   wire                                M_AXI_RVALID        ,
output  wire                                M_AXI_RREADY
    );

function integer clogb2 (input integer bit_depth);
begin
     for(clogb2=0; bit_depth>0; clogb2=clogb2+1)
     bit_depth = bit_depth >> 1;
end
endfunction

localparam AXI_BYTES =  M_AXI_DATA_WIDTH/8;
localparam [3:0] MAX_BURST_LEN_SIZE = clogb2(M_AXI_MAX_BURST_LEN -1);

//fdma axi write-------------------------------------------
reg     [M_AXI_ADDR_WIDTH-1 : 0]    axi_awaddr  =0;
reg                                 axi_awvalid = 1'b0;
wire    [M_AXI_DATA_WIDTH-1 : 0]    axi_wdata   ;
wire                                axi_wlast   ;
reg                                 axi_wvalid  = 1'b0;
wire                               w_next      = (M_AXI_WVALID & M_AXI_WREADY);
reg   [8 :0]                       wburst_len  = 1  ;
reg   [8 :0]                       wburst_cnt  = 0  ;
reg   [15:0]                       wfdma_cnt   = 0  ;
reg                                axi_wstart_locked  =0;
wire  [15:0] axi_wburst_size   =   wburst_len * AXI_BYTES;

assign M_AXI_WID        = 0;
assign M_AXI_AWID       = M_AXI_ID;
assign M_AXI_AWADDR     = axi_awaddr;
```

```
assign M_AXI_AWLEN      = wburst_len - 1;
assign M_AXI_AWSIZE     = clogb2(AXI_BYTES-1);
assign M_AXI_AWBURST    = 2'b01;
assign M_AXI_AWLOCK     = 1'b0;
assign M_AXI_AWCACHE    = 4'b0010;
assign M_AXI_AWPROT     = 3'h0;
assign M_AXI_AWQOS      = 4'h0;
assign M_AXI_AWVALID    = axi_awvalid;
assign M_AXI_WDATA      = axi_wdata;
assign M_AXI_WSTRB      = {(AXI_BYTES){1'b1}};
assign M_AXI_WLAST      = axi_wlast;
assign M_AXI_WVALID     = axi_wvalid & fdma_wready;
assign M_AXI_BREADY     = 1'b1;


//------------------------------------------------------------------------
//AXI4 FULL Write
assign  axi_wdata        = fdma_wdata;
assign  fdma_wvalid      = w_next;
reg     fdma_wstart_locked = 1'b0;
wire    fdma_wend;
wire    fdma_wstart;
assign   fdma_wbusy = fdma_wstart_locked ;

always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0 || fdma_wend == 1'b1 )
        fdma_wstart_locked <= 1'b0;
    else if(fdma_wstart)
        fdma_wstart_locked <= 1'b1;

assign fdma_wstart = (fdma_wstart_locked == 1'b0 && fdma_wareq == 1'b1);
//AXI4 write burst lenth busrt addr ------------------------------
always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)begin
        axi_awaddr <= 0;
    end
    else if(fdma_wstart)
        axi_awaddr <= fdma_waddr;
    else if(axi_wlast == 1'b1)
        axi_awaddr <= axi_awaddr + axi_wburst_size ;
//AXI4 write cycle ------------------------------------------------
reg axi_wstart_locked_r1 = 1'b0, axi_wstart_locked_r2 = 1'b0;
always @(posedge M_AXI_ACLK)begin
    if(M_AXI_ARESETN == 1'b0)begin
        axi_wstart_locked_r1 <= 1'b0;
        axi_wstart_locked_r2 <= 1'b0;
    end
    else begin
        axi_wstart_locked_r1 <= axi_wstart_locked;
        axi_wstart_locked_r2 <= axi_wstart_locked_r1;
    end
end

always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)begin
        axi_wstart_locked <= 1'b0;
    end
    else if((fdma_wstart_locked == 1'b1) &&  axi_wstart_locked == 1'b0)
```

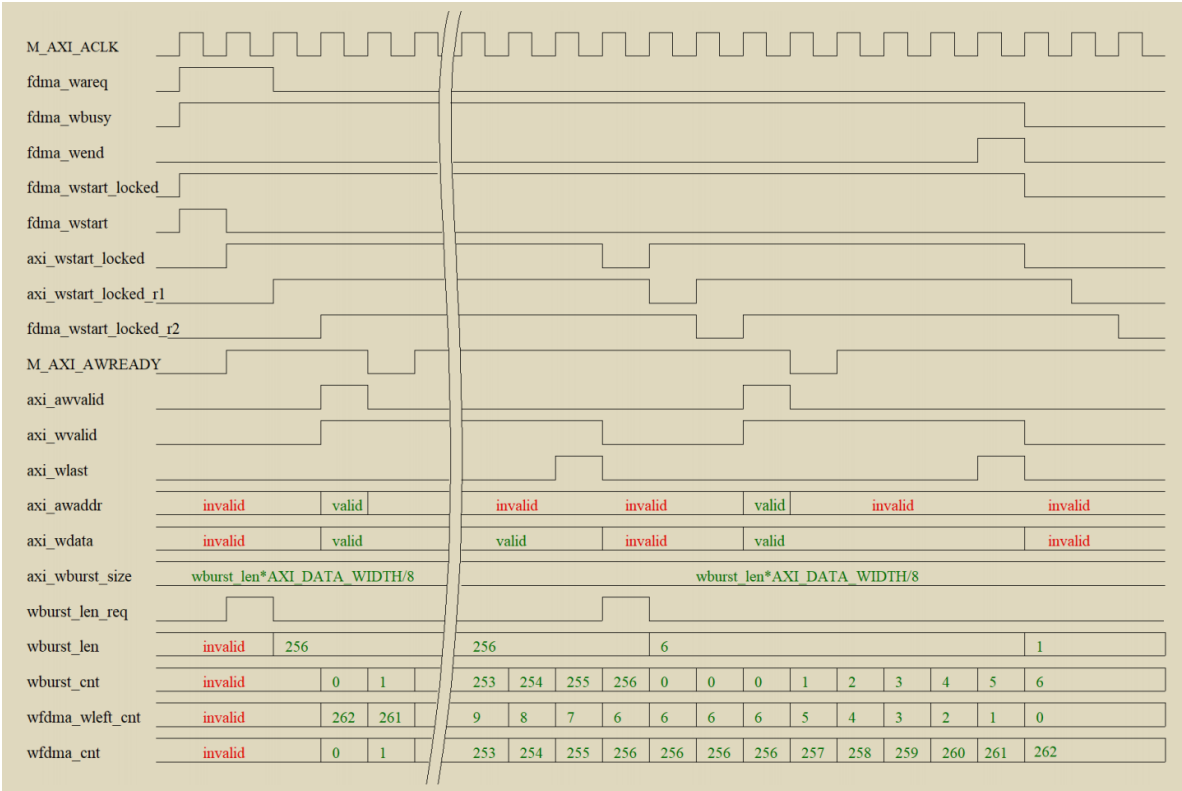
```
        axi_wstart_locked <= 1'b1;
    else if(axi_wlast == 1'b1 || fdma_wstart == 1'b1)
        axi_wstart_locked <= 1'b0;

//AXI4 addr valid and write addr-----------------------------------
always @(posedge M_AXI_ACLK)
     if(M_AXI_ARESETN == 1'b0)begin
         axi_awvalid <= 1'b0;
     end
     else if((axi_wstart_locked_r1 == 1'b1) &&  axi_wstart_locked_r2 == 1'b0)
         axi_awvalid <= 1'b1;
     else if((axi_wstart_locked == 1'b1 && M_AXI_AWREADY == 1'b1)||
axi_wstart_locked == 1'b0)
         axi_awvalid <= 1'b0;
//AXI4 write data--------------------------------------------------
always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)begin
         axi_wvalid <= 1'b0;
    end
    else if((axi_wstart_locked_r1 == 1'b1) &&  axi_wstart_locked_r2 == 1'b0)
        axi_wvalid <= 1'b1;
    else if(axi_wlast == 1'b1 || axi_wstart_locked == 1'b0)
        axi_wvalid <= 1'b0;//
//AXI4 write data burst len counter---------------------------------
always @(posedge M_AXI_ACLK)
    if(axi_wstart_locked == 1'b0)
        wburst_cnt <= 0;
    else if(w_next)
        wburst_cnt <= wburst_cnt + 1'b1;

assign axi_wlast = (w_next == 1'b1) && (wburst_cnt == M_AXI_AWLEN);
//fdma write data burst len counter---------------------------------
reg wburst_len_req = 1'b0;
reg [15:0] fdma_wleft_cnt =16'd0;

always @(posedge M_AXI_ACLK)
        wburst_len_req <= fdma_wstart|axi_wlast;

always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)begin
        wfdma_cnt <= 0;
        fdma_wleft_cnt <= 0;
    end
    else if( fdma_wstart )begin
        wfdma_cnt <= 0;
        fdma_wleft_cnt <= fdma_wsize;
    end
    else if(w_next)begin
        wfdma_cnt <= wfdma_cnt + 1'b1;
        fdma_wleft_cnt <= (fdma_wsize - 1'b1) - wfdma_cnt;
    end

assign  fdma_wend = w_next && (fdma_wleft_cnt == 1 );

always @(posedge M_AXI_ACLK)begin
    if(M_AXI_ARESETN == 1'b0)begin
        wburst_len <= 1;
    end
```

```
    else if(wburst_len_req)begin
        if(fdma_wleft_cnt[15:MAX_BURST_LEN_SIZE] >0)
            wburst_len <= M_AXI_MAX_BURST_LEN;
        else
            wburst_len <= fdma_wleft_cnt[MAX_BURST_LEN_SIZE-1:0];
    end
    else wburst_len <= wburst_len;
end

//fdma axi read-------------------------------------------
reg     [M_AXI_ADDR_WIDTH-1 : 0]    axi_araddr =0   ;
reg                                 axi_arvalid  =1'b0;
wire                                axi_rlast   ;
reg                                 axi_rready  = 1'b0;
wire                               r_next      = (M_AXI_RVALID && M_AXI_RREADY);
reg   [8 :0]                       rburst_len  = 1  ;
reg   [8 :0]                       rburst_cnt  = 0  ;
reg   [15:0]                       rfdma_cnt   = 0  ;
reg                                axi_rstart_locked =0;
wire  [15:0] axi_rburst_size   =   rburst_len * AXI_BYTES;

assign M_AXI_ARID       = M_AXI_ID;
assign M_AXI_ARADDR     = axi_araddr;
assign M_AXI_ARLEN      = rburst_len - 1;
assign M_AXI_ARSIZE     = clogb2((AXI_BYTES)-1);
assign M_AXI_ARBURST    = 2'b01;
assign M_AXI_ARLOCK     = 1'b0;
assign M_AXI_ARCACHE    = 4'b0010;
assign M_AXI_ARPROT     = 3'h0;
assign M_AXI_ARQOS      = 4'h0;
assign M_AXI_ARVALID    = axi_arvalid;
assign M_AXI_RREADY     = axi_rready&&fdma_rready;
assign fdma_rdata       = M_AXI_RDATA;
assign fdma_rvalid      = r_next;

//AXI4 FULL Read-----------------------------------------

reg     fdma_rstart_locked = 1'b0;
wire    fdma_rend;
wire    fdma_rstart;
assign   fdma_rbusy = fdma_rstart_locked ;

always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0 || fdma_rend == 1'b1)
        fdma_rstart_locked <= 1'b0;
    else if(fdma_rstart)
        fdma_rstart_locked <= 1'b1;

assign fdma_rstart = (fdma_rstart_locked == 1'b0 && fdma_rareq == 1'b1);
//AXI4 read burst lenth busrt addr ------------------------------
always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)
        axi_araddr <= 0;
    else if(fdma_rstart == 1'b1)
        axi_araddr <= fdma_raddr;
    else if(axi_rlast == 1'b1)
        axi_araddr <= axi_araddr + axi_rburst_size ;
```

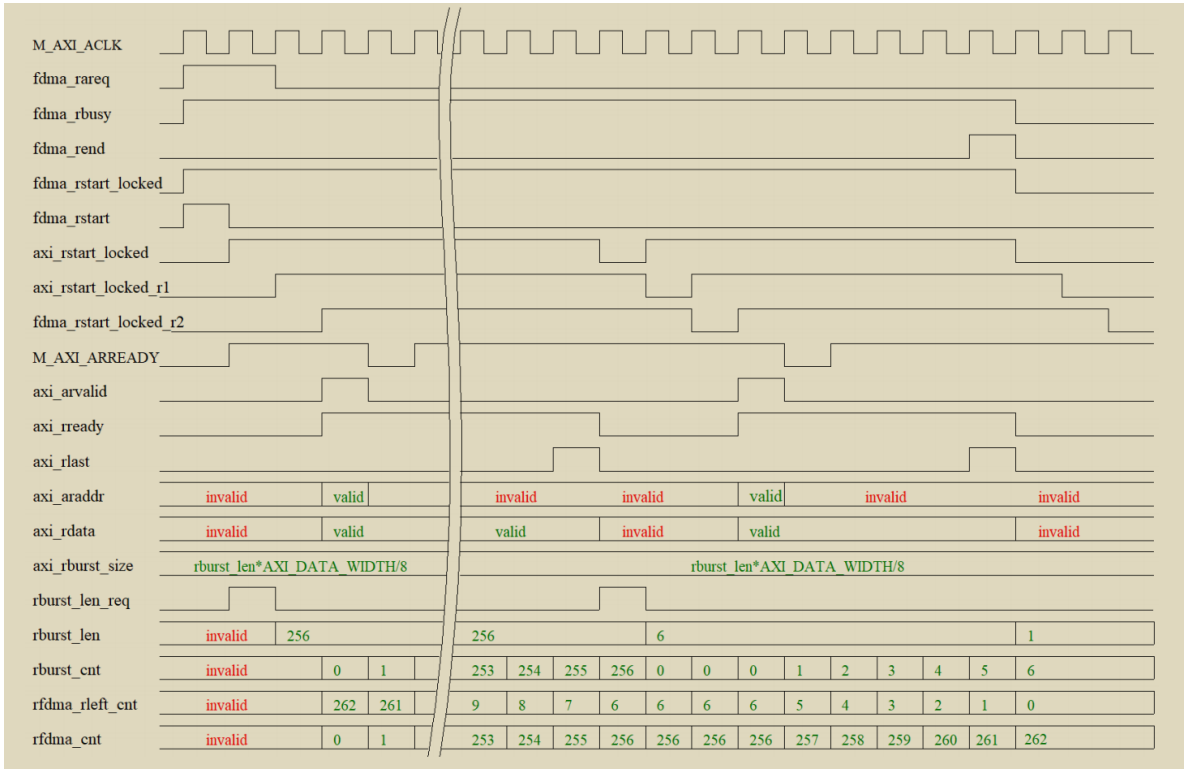
```
//AXI4 r_cycle_flag-------------------------------------
reg axi_rstart_locked_r1 = 1'b0, axi_rstart_locked_r2 = 1'b0;
always @(posedge M_AXI_ACLK)begin
    if(M_AXI_ARESETN == 1'b0)begin
        axi_rstart_locked_r1 <= 1'b0;
        axi_rstart_locked_r2 <= 1'b0;
    end
    else begin
        axi_rstart_locked_r1 <= axi_rstart_locked;
        axi_rstart_locked_r2 <= axi_rstart_locked_r1;
    end
end
always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)
        axi_rstart_locked <= 1'b0;
    else if((fdma_rstart_locked == 1'b1) &&  axi_rstart_locked == 1'b0)
        axi_rstart_locked <= 1'b1;
    else if(axi_rlast == 1'b1 || fdma_rstart == 1'b1)
        axi_rstart_locked <= 1'b0;

//AXI4 addr valid and read addr-----------------------------------
always @(posedge M_AXI_ACLK)
     if(M_AXI_ARESETN == 1'b0)
         axi_arvalid <= 1'b0;
     else if((axi_rstart_locked_r1 == 1'b1) &&  axi_rstart_locked_r2 == 1'b0)
         axi_arvalid <= 1'b1;
     else if((axi_rstart_locked == 1'b1 && M_AXI_ARREADY == 1'b1)||
axi_rstart_locked == 1'b0)
         axi_arvalid <= 1'b0;
//AXI4 read data-------------------------------------------------
always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)
         axi_rready <= 1'b0;
    else if((axi_rstart_locked_r1 == 1'b1) &&  axi_rstart_locked_r2 == 1'b0)
        axi_rready <= 1'b1;
    else if(axi_rlast == 1'b1 || axi_rstart_locked == 1'b0)
        axi_rready <= 1'b0;//

//AXI4 read data burst len counter---------------------------------
always @(posedge M_AXI_ACLK)
    if(axi_rstart_locked == 1'b0)
        rburst_cnt <= 0;
    else if(r_next)
        rburst_cnt <= rburst_cnt + 1'b1;
assign axi_rlast = (r_next == 1'b1) && (rburst_cnt == M_AXI_ARLEN);
//fdma read data burst len counter---------------------------------
reg rburst_len_req = 1'b0;
reg [15:0] fdma_rleft_cnt =16'd0;

always @(posedge M_AXI_ACLK)
        rburst_len_req <= fdma_rstart | axi_rlast;

always @(posedge M_AXI_ACLK)
    if(M_AXI_ARESETN == 1'b0)begin
        rfdma_cnt <= 0;
        fdma_rleft_cnt <=0;
    end
    else if(fdma_rstart )begin
```

```
            rfdma_cnt <= 0;
            fdma_rleft_cnt <= fdma_rsize;
        end
        else if(r_next)begin
            rfdma_cnt <= rfdma_cnt + 1'b1;
            fdma_rleft_cnt <= (fdma_rsize - 1'b1) - rfdma_cnt;
        end

assign  fdma_rend = r_next && (fdma_rleft_cnt == 1 );
//axi auto burst len caculate----------------------------------------

always @(posedge M_AXI_ACLK)begin
     if(M_AXI_ARESETN == 1'b0)begin
        rburst_len <= 1;
     end
     else if(rburst_len_req)begin
        if(fdma_rleft_cnt[15:MAX_BURST_LEN_SIZE] >0)
            rburst_len <= M_AXI_MAX_BURST_LEN;
        else
            rburst_len <= fdma_rleft_cnt[MAX_BURST_LEN_SIZE-1:0];
     end
     else rburst_len <= rburst_len;
end

endmodule
```

写操作时序：



读操作时序：


M_AXI_ACLK
fdma_rareq
fdma_rbusy
fdma_rend
fdma_rstart_locked
fdma_rstart
axi_rstart_locked
axi_rstart_locked_r1
fdma_rstart_locked_r2
M_AXI_ARREADY
axi_arvalid
axi_rready
axi_rlast
axi_araddr
invalid valid invalid invalid valid invalid invalid
axi_rdata
invalid valid valid invalid valid invalid
axi_rburst_size
rburst_len*AXI_DATA_WIDTH/8 rburst_len*AXI_DATA_WIDTH/8
rburst_len_req
rburst_len
invalid 256 256 6 1
rburst_cnt
invalid 0 1 253 254 255 256 0 0 0 1 2 3 4 5 6
rfdma_rleft_cnt
invalid 262 261 9 8 7 6 6 6 6 5 4 3 2 1 0
rfdma_cnt
invalid 0 1 253 254 255 256 256 256 256 257 258 259 260 261 262