一、RoCC部分

YOLO加速器的顶层模块在top.scala。它仅仅是继承自Module，接收隐式参数Parameters。集成时，隐式的clock端口要与RISC-V核的核心时钟相连。RoCC接口只需要读取指令，因此只使用了RoCCCoreIO。其中cmd用于接收自定义指令，rs1用于接收数据。busy表示忙碌状态，高有效。interrupt表示中断，算完一幅图像并写回DDR后高有效一个周期。

二、AXI4部分

YOLO模块对外的输出接口为AXI总线，用于读写DDR。相关参数如下：

object AXI4Parameters {

val C\_S\_AXI\_ID\_WIDTH = 4

val C\_S\_AXI\_ADDR\_WIDTH = 32

val C\_S\_AXI\_DATA\_WIDTH = 512

val C\_S\_AXI\_NBURST\_SUPPORT = 0

val C\_S\_CTL\_SIG\_WIDTH = 3

val DBG\_WR\_STS\_WIDTH = 40

val DBG\_RD\_STS\_WIDTH = 40

val C\_S\_EN\_UPSIZER = 0

val C\_S\_WDG\_TIMER\_WIDTH = 11

}

即ID位宽是4bit，地址32bit，数据与DDR一致，是512bit。DBG\_WR\_STS\_WIDTH和DBG\_RD\_STS\_WIDTH是读、写状态的位宽，Vivado默认是40bit，看门狗定时器默认是11bit。

使用的AXI端口如下：

class HasAXI4IO extends Bundle {

//slave interface write address ports

val awid = Input(UInt(C\_S\_AXI\_ID\_WIDTH.W))

val awaddr = Input(UInt(C\_S\_AXI\_ADDR\_WIDTH.W))

val awlen = Input(UInt(8.W))

val awsize = Input(UInt(3.W))

val awburst = Input(UInt(2.W))

val awlock = Input(Bool())

val awcache = Input(UInt(4.W))

val awprot = Input(UInt(3.W))

val awqos = Input(UInt(4.W))

val awvalid = Input(Bool())

val awready = Output(Bool())

//slave interface write data ports

val wdata = Input(UInt(C\_S\_AXI\_DATA\_WIDTH.W))

val wstrb = Input(UInt((C\_S\_AXI\_DATA\_WIDTH/8).W))

val wlast = Input(Bool())

val wvalid = Input(Bool())

val wready = Output(Bool())

//slave interface write response ports

val bid = Output(UInt(C\_S\_AXI\_ID\_WIDTH.W))

val bresp = Output(UInt(2.W))

val bvalid = Output(Bool())

val bready = Input(Bool())

//slave interface read address ports

val arid = Input(UInt(C\_S\_AXI\_ID\_WIDTH.W))

val araddr = Input(UInt(C\_S\_AXI\_ADDR\_WIDTH.W))

val arlen = Input(UInt(8.W))

val arsize = Input(UInt(3.W))

val arburst = Input(UInt(2.W))

val arlock = Input(Bool())

val arcache = Input(UInt(4.W))

val arprot = Input(UInt(3.W))

val arqos = Input(UInt(4.W))

val arvalid = Input(Bool())

val arready = Output(Bool())

//slave interface read data ports

val rid = Output(UInt(C\_S\_AXI\_ID\_WIDTH.W))

val rdata = Output(UInt(C\_S\_AXI\_DATA\_WIDTH.W))

val rresp = Output(UInt(2.W))

val rlast = Output(Bool())

val rvalid = Output(Bool())

val rready = Input(Bool())

}

声明为具体IO时，都会带上统一的前缀“axi\_”。

这部分接口定义来自于Vivado的MIG IP核，可以连接到AXI Crossbar，与RISC-V核共享DDR的读写。

三、指令部分

控制YOLO的指令如下：

object Instructions {

def YoloOutputAddr = BitPat("b0000000??????????010000000001011")

def YoloInputAddr = BitPat("b0000001??????????010000010001011")

def YoloWeightAddr = BitPat("b0000010??????????010000100001011")

def YoloBiasAddr = BitPat("b0000011??????????010000110001011")

def YoloBiasInfo = BitPat("b0000100??????????010001000001011")

def YoloInputInfo = BitPat("b0000101??????????010001010001011")

def YoloOutputInfo = BitPat("b0000110??????????010001100001011")

def YoloRegReset = BitPat("b0000000??????????000?????0001011")

def YoloFSMStart = BitPat("b0000001??????????000?????0001011")

def YoloNOP = BitPat("b0000010??????????000?????0001011")

}

前七条指令用到了rs1寄存器的数据，后三条指令不需要任何寄存器。其中，YoloOutputAddr表明rs1发送了输出结果存储在DDR的首地址，YoloInputAddr 、YoloWeightAddr和YoloBiasAddr则分别是rs1发送了输入图像在DDR的首地址、权值的首地址和偏置的首地址。

YoloBiasInfo表明rs1发送了偏置的个数，也就是整个YOLO v2的所有层的输出通道数之和。因为偏置容量小，所以可以一次取完整个网络的偏置。

YoloInputInfo表明rs1发送了当前层的输入通道信息，格式如下：



即低11位是输入通道的个数，再高8位是输入的尺寸大小。第一层的输入图像的尺寸最大，为224；最后几层的通道数最多，为1024。

YoloOutputInfo表明rsS1发送了当前层的输出通道信息，格式如下：



与输入信息类似，但是多了卷积核尺寸、当前层是否需要池化、是不是第一层、是不是最后一层。如果相应的比特位为1，表明需要池化、是第一层、是最后一层。

YoloRegReset是通过指令来复位YOLO加速器。每次发送指令前，应该先用复位指令进行复位，让寄存器都归零。复位期间，busy信号拉高，持续五个周期。然后才能发送其他指令。除了指令复位，AXI总线的复位信号aresetn也可以复位加速器。

YoloFSMStart是启动指令。在前七条配置指令发送完毕后，通过这条指令启动加速器工作。YoloNOP是空指令，让YOLO内部的解码器失效，空指令一定要紧跟在启动指令的后面，并且后面不能再有其它指令。