

一、前言

目前CPU进度可搭建片上soc，已通过Vivado综合，各功能验证无误。
此软核仅用于学习交流，若要进行商用，请务必与作者本人联系！

1.1 资料说明

链接：https://github.com/hihii11/Fly_v_for_riscv

资料总共提供了两份工程，其一用于软核的仿真，可以观察软核内部信号的变化。
另一份用提供了一SOC设计实例，使用者可根据自己的板卡和需求修改工程配置，进行比特流烧录。

RISC-V	2022/7/22 11:12	文件夹
RISC-V-SOC	2022/7/22 11:12	文件夹
在测试例程中给出了CPU和SOC的测试例程，分别用于上述两个工程。		
RISC-V-CPU-test-pro	2022/7/21 17:09	文件夹
RISC-V-SOC-test-pro	2022/7/22 15:29	文件夹

CPU测试例程

1.基本指令集测试	2022/7/20 10:40	文件夹
2.斐波那契数列	2022/7/20 13:59	文件夹
3.冒泡排序	2022/7/20 14:40	文件夹
4.素数	2022/7/20 15:10	文件夹
5.字符串替换	2022/7/20 16:20	文件夹
6.MIO输入输出测试	2022/7/20 21:48	文件夹
7.MIO中断	2022/7/20 22:58	文件夹
8.自定义外部中断	2022/7/20 23:22	文件夹
9.周期计数器	2022/7/21 13:43	文件夹
10.信号脉宽测量	2022/7/21 14:02	文件夹
11.周期计数器延迟	2022/7/21 14:21	文件夹
12.Timer64计数	2022/7/21 16:44	文件夹
13.Timer64定时器中断	2022/7/21 17:09	文件夹
dist	2022/7/22 16:14	文件夹

SOC测试例程

1.MIO输出测试	2022/7/22 11:05	文件夹
2.AHB_GPIO输出测试	2022/7/22 11:16	文件夹
3.I2C测试	2022/7/22 13:14	文件夹
4.SPI_LCD测试	2022/7/22 14:58	文件夹
5.AHB_GPIO中断测试	2022/7/22 15:30	文件夹
dist	2022/7/22 16:13	文件夹

创建日期: 2022/7/22 16:15
大小: 2.29 KB
文件: spi_lcd.asm

在编译器中，提供了RISC-V的RARS编译器，具体程序烧录例程参考第五章。

 rars_27a7c1f.jar

2022/7/8 22:21

Executable Jar File

1,812 KB

二、飞V软核介绍

- 32位 RISC-V架构CPU
- 经典五级流水线架构
- 支持RISC-V IM指令集
- 可拓展AHB主机接口
- 可编程快速MIO口，最多32个
- 内部可编程周期计数器、Timer64计数器（定时器）
- PLIC中断管理器，可编程中断屏蔽、优先级
- 最多支持23个用户自定义中断（不支持中断嵌套）
- 暂无实现JTAG

三、支持指令介绍

3.1 R型指令

31~25	24~20	19~15	14~12	11~7	6~0	
func7	op2	op1	func3	rd	op_code	operation
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

3.2 整数乘法除法指令(R型)

31~25	24~20	19~15	14~12	11~7	6~0	
func7	op2	op1	func3	rd	op_code	operation
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

3.3 I型指令

31~20		19~15	14~12	11~7	6~0	
immediate data		op1	func3	rd	op_code	operation
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

3.4 加载指令(I型)

31~20	19~15	14~12	11~7	6~0	
immediate data	op1	func3	rd	op_code	operation
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

3.5 B型指令

31	30~25	24~20	19~15	14~12	11~8	7	6~0	
imm	imm	op2	op1	func3	imm	imm	op_code	operation
imm[12]	imm[10:5]	rs2	rs1	000	imm[4:1]	imm[11]	1100011	BEQ
imm[12]	imm[10:5]	rs2	rs1	001	imm[4:1]	imm[11]	1100011	BNE
imm[12]	imm[10:5]	rs2	rs1	100	imm[4:1]	imm[11]	1100011	BLT
imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1]	imm[11]	1100011	BGE
imm[12]	imm[10:5]	rs2	rs1	110	imm[4:1]	imm[11]	1100011	BLTU
imm[12]	imm[10:5]	rs2	rs1	111	imm[4:1]	imm[11]	1100011	BGEU

3.6 J型指令

31	30~21	20	19~12	11~7	6~0	
imm	imm	imm	imm	rd	op_code	operation
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	1101111	JAL

31~20	19~15	14~12	11~7	6~0	
imm	rs1	func3	rd	op_code	operation
imm[11:0]	rs1	000	rd	1100111	JALR

3.7 S型指令

31~25	24~20	19~15	14~12	11~7	6~0	
imm	op2	op1	func3	imm	op_code	operation
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

3.8 CSR寄存器操作指令

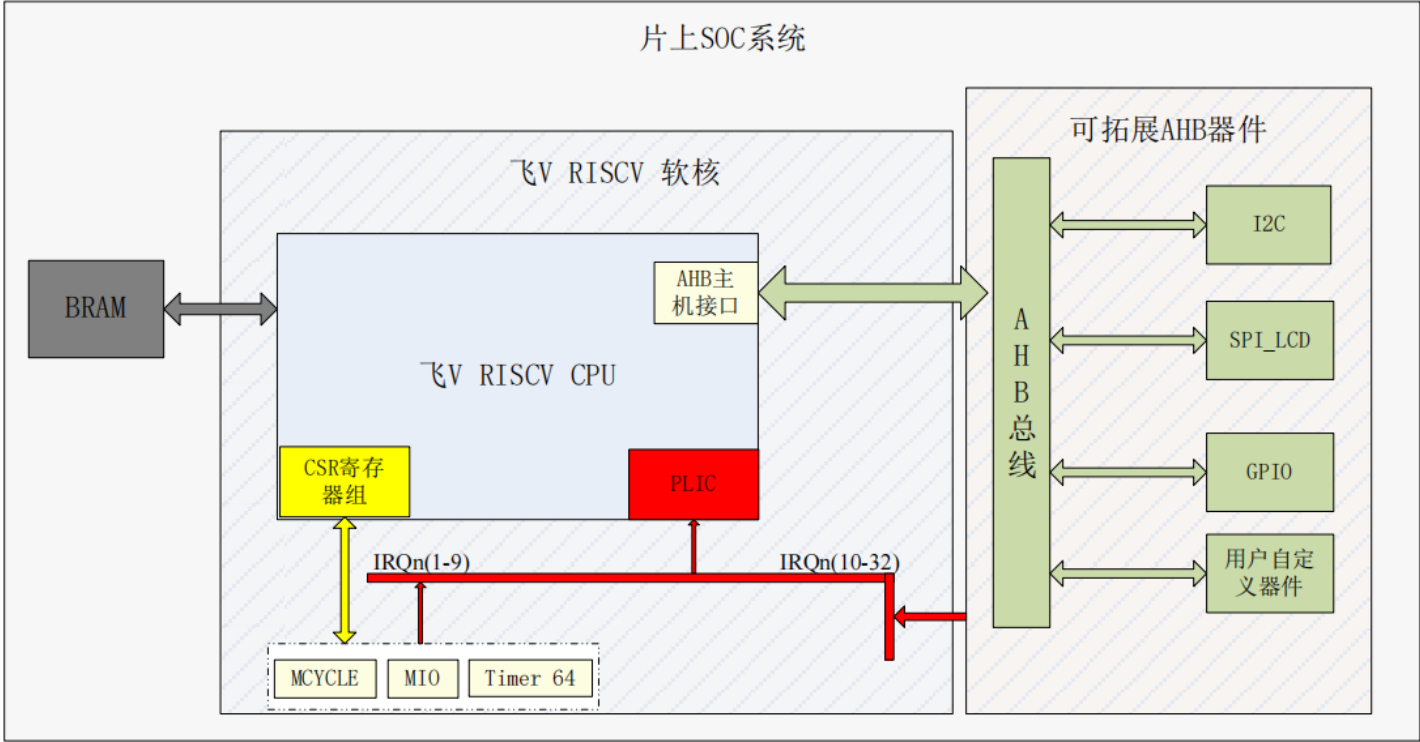
31~20	19~15	14~12	11~7	6~0	
csr	op1	func3	rd	op_code	operation
csr	rs1	001	rd	1110011	CSRRW
csr	rs1	010	rd	1110011	CSRRS
csr	rs1	011	rd	1110011	CSRRS
csr	zimm	101	rd	1110011	CSRRWI
csr	zimm	110	rd	1110011	CSRRSI
csr	zimm	111	rd	1110011	CSRRCI

3.9 中断返回指令

31~20	19~15	14~12	11~7	6~0	
csr	op1	func3	rd	op_code	operation
0000000_00010	00000	000	00000	1110011	URET
0001000_00010	00000	000	00000	1110011	SRET

0011000_00010	00000	000	00000	1110011	MRET
---------------	-------	-----	-------	---------	------

四、飞V核心SOC架构



•软核部分

在软核中设计了能通过CSR直接进行控制的周期计数器（MYCYCLE）、可编程快速IO口（MIO）、64位计数器（Timer64）。

这些器件固定占用了PLIC中断的ID 1~9。

CPU内嵌了PLIC中断管理器，其产生的中断直接作为外部中断进行处理，其配置也可通过CSR寄存器组实现。

CPU内嵌了AHB主机接口，能够通过AHB总线访问挂载在AHB总线上的各种器件，其配置也可通过CSR寄存器组实现。

内存访问部分使用单独的数据总线，ROM使用了Xilinx板载ROM，但RAM使用了LUT RAM。

这是因为使用者可以通过LUT RAM来直观的观察RAM数据的读写过程，这里也可将LUT RAM直接替换为BRAM。

•自定义IP设计

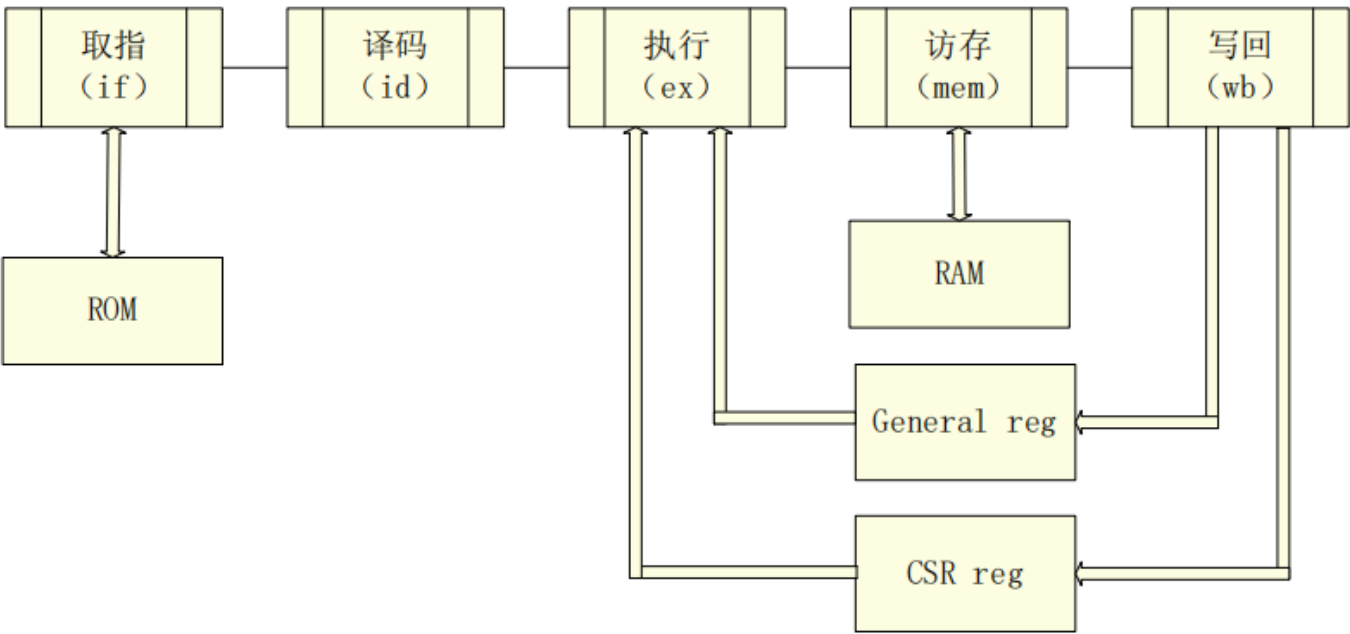
使用者自定义的IP核将直接挂载在AHB总线上，在本设计提供的示例设计中，默认挂载了一AHB读写测试从机、I2C、GPIO、SPI_LCD（LCD专用高速驱动器）。

使用者可根据需要自行增删改查AHB接口的IP，具体方法在资料的AHB文档中有说明。

最多可支持的AHB器件（推荐16）为4096。所有AHB器件若需要中断处理，则将连接至PLIC中断管理器的ID（10~32）（ID：10对应软核上的IRQn0）

4.1 软核设计思路

该RISCV软核采用了经典的五级流水线架构。
即取指 (if) 、译码 (id) 、执行 (ex) 、访存 (mem) 、写回 (wb)



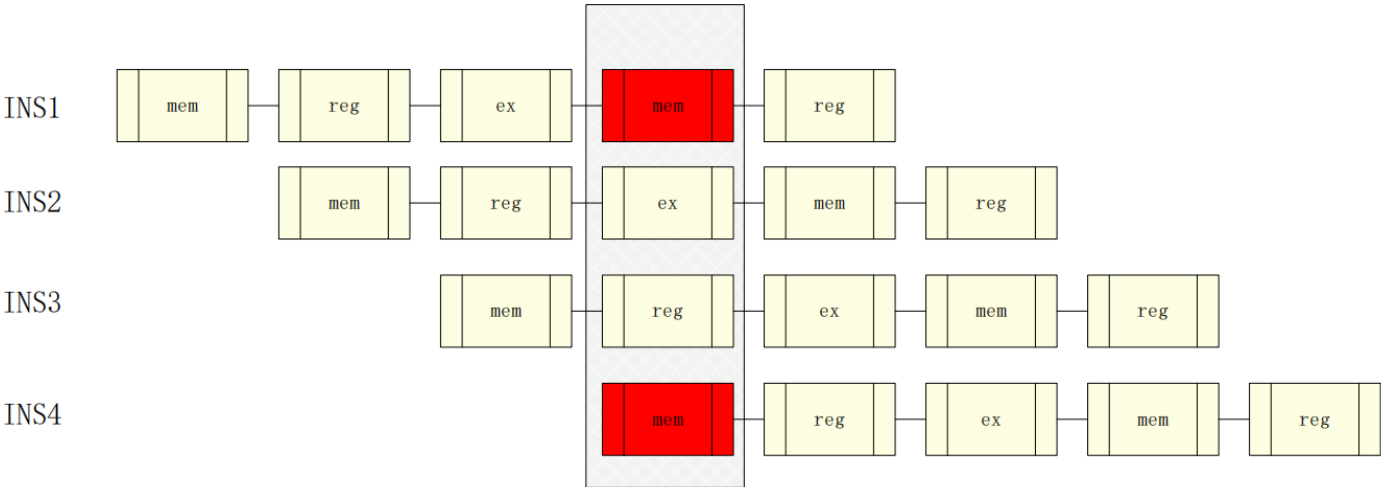
但这种架构的CPU会存在流水线冒险问题，这里主要存在的是RAW（read after write）。

4.2 流水线冒险

冒险会干扰流水线的正常指令流，从而导致CPU的执行效率降低

4.2.1 结构冒险

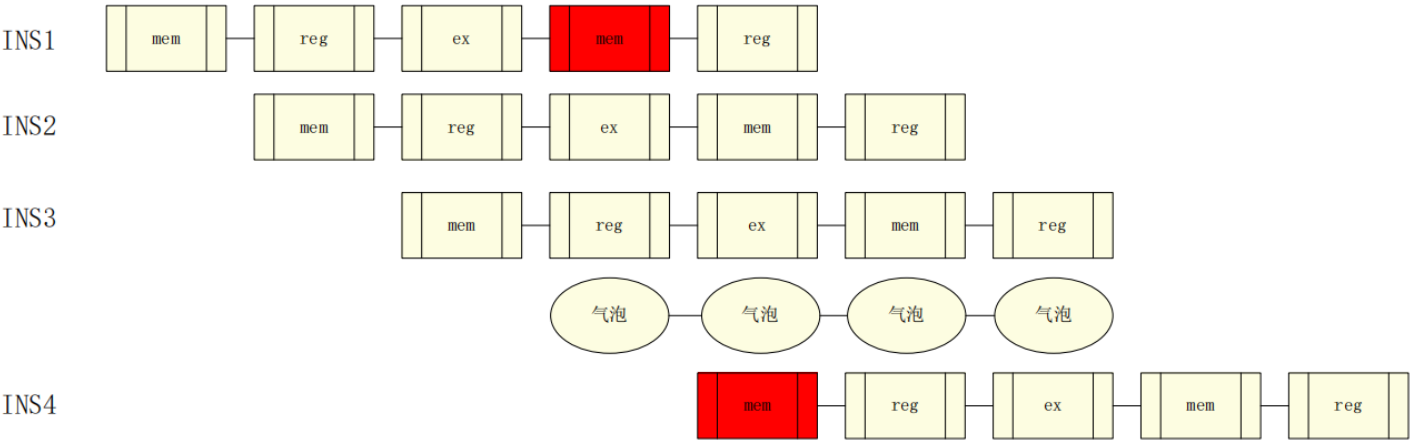
结构冒险主要由硬件的控制产生，如在同一时刻需要对Memory进行写入和读出操作，而Memory读写共用了一组端口，此时就存储器的端口控制，产生了冲突。



如上图中，第一条指令和第四条指令的mem阶段出现冲突。

解决这个问题的方法之一为在发生冲突时，加入一个流水线气泡，这个气泡可以由硬件产生或软件产生。

无论是硬件方案还是软件方案，都会增加一个周期的延迟。



另一种方案则是在if和mem阶段中避免使用同一个存储器。

在飞V的设计中，将RAM的读写操作统一放到了MEM阶段，同一时刻只允许执行写或读操作，因此不会出现控制冒险。

但这样设计会引入加载指令的数据冒险问题。

4.2.2 数据冒险

• 读后写 (RAW) :

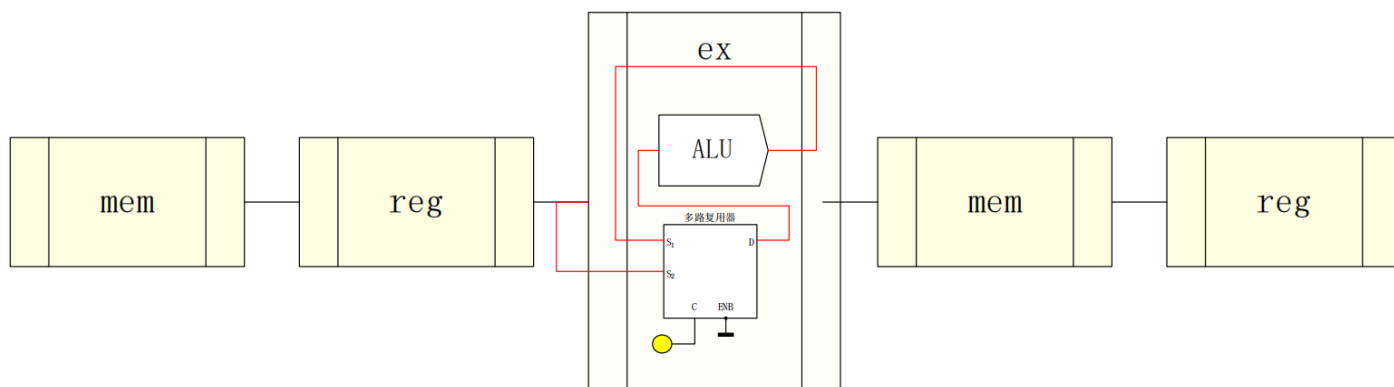
当第一条指令需要对目的寄存器进行写操作时，第二条指令需要读取上一条指令的目的寄存器作为操作数时，就会发生写后读，寄存器数据还未更新的情况。

这是由于通用寄存器的写回操作在wb阶段，而读取操作在ex阶段。

下列代码段将用于无法得到正确的结果

```
1  0x14:      addi  rs1,rs1,10;    000000000101000001000000010010011
2  0x18:      addi  rs2,rs1,2;     000000000001000001000000010010011
3  0x22:      sub   rs3,rs1,rs2;   010000000001000100000000110110011
4  0x26:      slt   rs4,rs1,rs3;   00000000000100011010001000110011
```

在飞V中，使用了数据转移的方法解决该问题。



其主要方法是在ex模块中增加一MUX，其选择信号由ex产生，当判断当前操作数存储地址与上一次的目
的操作数寄存器地址相同时，则直接将ALU的输出作为本次运算ALU的输入。

• 写后写 (WAW) :

两条指令连续写同一个寄存器，但写的次序颠倒。这种情况在飞V的架构中不会出现

• 写后读 (WAR) :

前一条指令在读出寄存器的值后，下一条指令才写入该寄存器。

如下面代码段：

```
1 lw x1, 0(x0)
2 addi x4, x1, x5
```

第一条加载指令发lw在mem阶段才将在存储器中存储的数据取出，再经过WB写回到寄存器组。
而第二条指令在ex阶段执行。

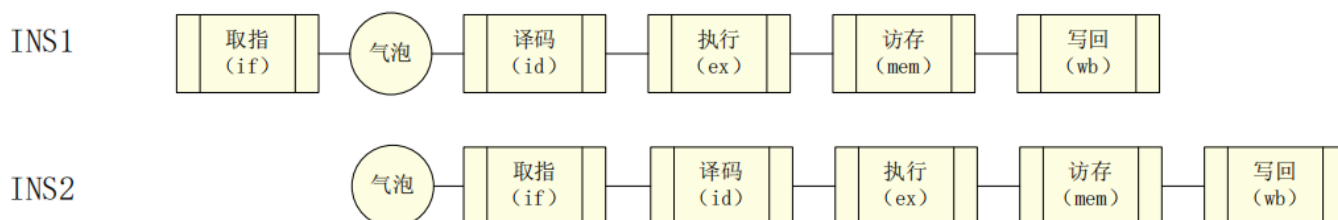
这也就造成了，addi在执行阶段时，lw在访存阶段。

这种情况下可以引入流水线纵向气泡解决。

纵向气泡的引入可以通过软件或硬件来实现。软件方法可参考RAW，硬件方法则是在addi执行到
EX阶段时，将流水线停止一个时钟周期。

在飞V的设计方法中，为了使得程序流获得更高的执行效率，将这种情况的判断提前到了if取指阶
段。

在if模块中，采用了双口ROM来对下一条指令进行预测，若本次执行的加载指令的目的寄存器为下
次执行指令的源操作数，则将PC暂停一个时钟周期。



4.3 流水线冲刷

在执行分支跳转指令时，由于分支跳转指令也是在ex模块执行，所以会出现以下情况。

- 1

INS0
- 2

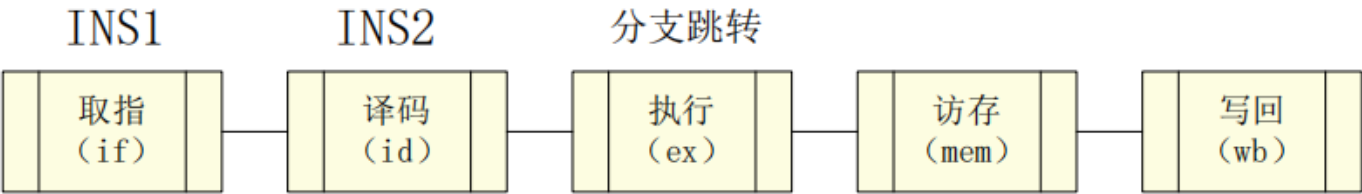
分支跳转
- 3

INS1
- 4

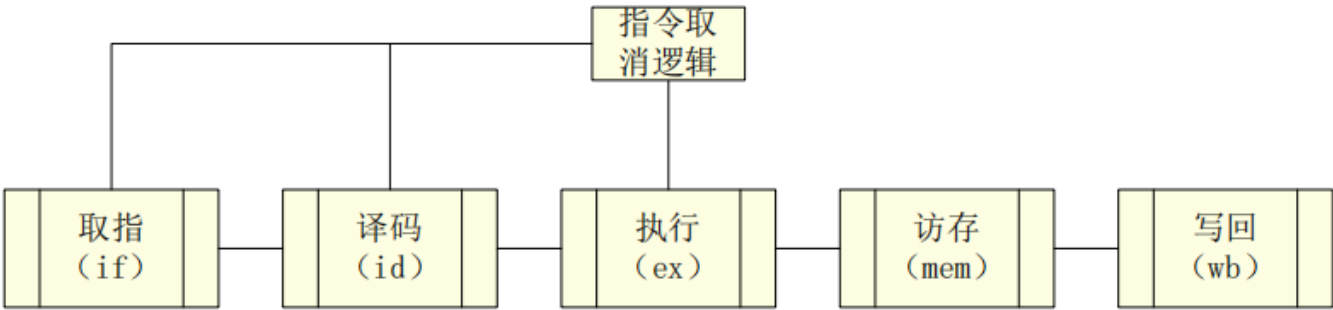
INS2

在ex执行的分支跳转指令，若条件成立，则会进行PC的置数操作，而此时，分支跳转指令后的INS1和INS2分别处于取指和译码阶段。

最终也会被执行。但实际情况是，分支跳转后的指令不应该被执行。



因此需要在硬件上设置电路，将进入程序流的INS1和INS2取消掉。



其方法是再增加一指令取消逻辑的判断，当需要置数PC时，拉高取指和译码的复位端一个周期，将if和id的当前指令替换为无效指令或nop。

上述方法也被称为流水线冲刷技术。

其造成当需要跳转时，会在程序流中引入两个周期延迟。

4.4 跳转冲突

跳转冲突也属于控制冒险的一种，在飞V中主要为中断跳转与跳转指令的冲突。

1) 中断跳转与分支跳转指令同时发生。

由于PC中设置的是分支跳转指令优先，所以中断跳转会被忽略，但中断处理器默认CPU正在处理中断，但实际上CPU正在处理正常程序流，因此不会运行到MRET(URET)指令，因此中断信号将被锁死。

解决方法：

在PC中设置中断跳转优先，此时当前的分支跳转指令将被忽略，因此MEPC需要更新至当前未执行的分支跳转指令。

PC设计

```
1 if(IRQ2PC_LOAD && PC_LOAD) pc_addr <= IRQ2PC_PC_ADDR;
```

中断管理器MEPC逻辑

```
1 if(EX2PC_LOAD) irq2csr_mepc <= PC_ADDR + 32'h4 - 32'h0C;
```

由于中断跳转发出的流水线冲刷与分支跳转指令发出的流水线冲刷相重合，所以PC需要在减去0x08的基础上再减去0x0c以指向未处理的分支跳转指令

2) 中断跳转在分支跳转流水线冲刷后发生

此时分支跳转指令已经执行，但由于中断跳转发生在流水线冲刷后，中断跳转的流水线冲刷将冲刷掉一条跳转后的指令，所以只需要减去0x04，来指向这条被冲刷的指令。

```
1 else if(ex2pc_load_dff0) irq2csr_mepc <= PC_ADDR + 32'h4 - 32'h04;
```


五、开发环境

本RV软核设计的所有开发文件均在Vivado 2018.3中完成设计和调试。

5.1 程序烧录

软核程序的下载通过配置ROM的向量文件来实现。

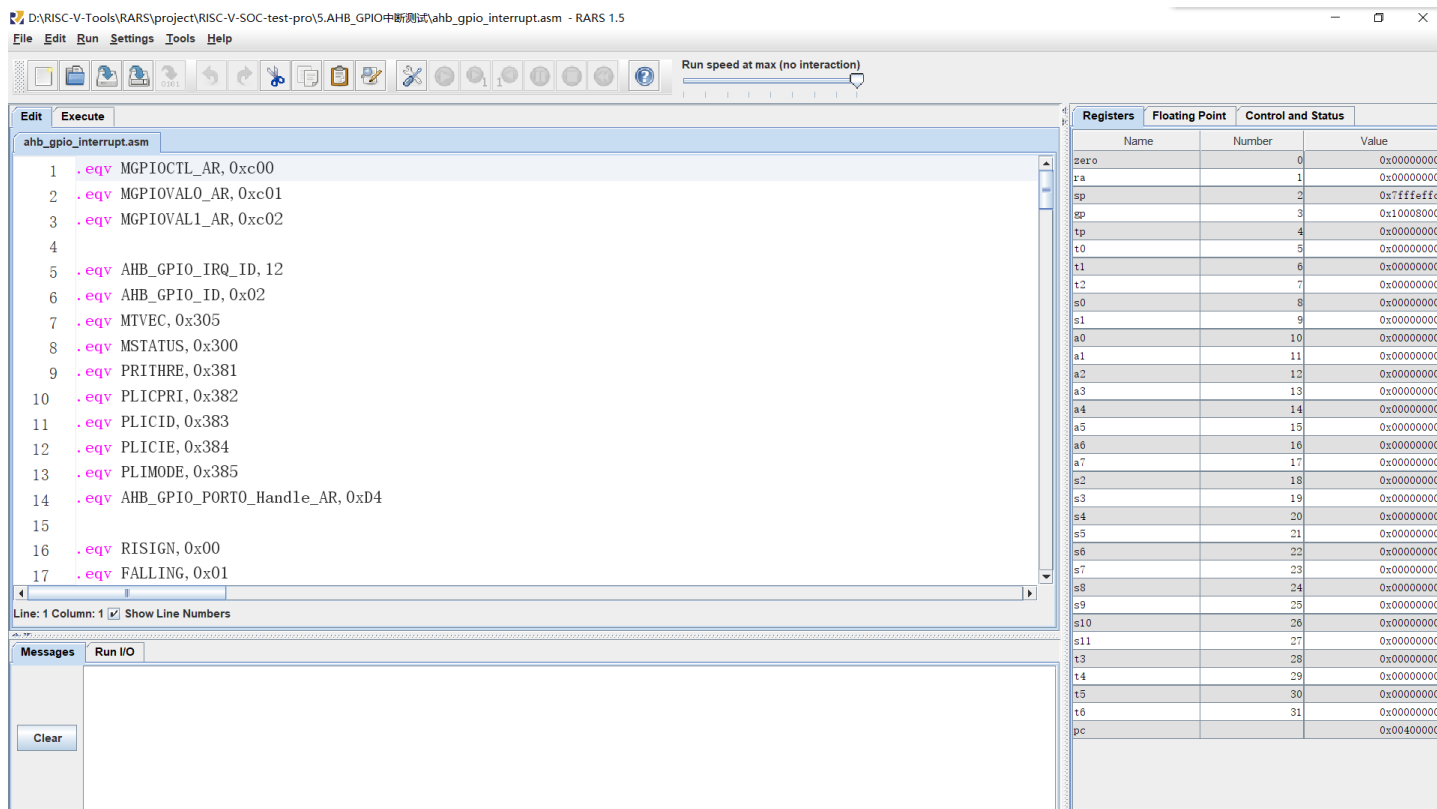
首先通过RARS完成汇编的编写

 rars_27a7c1f.jar

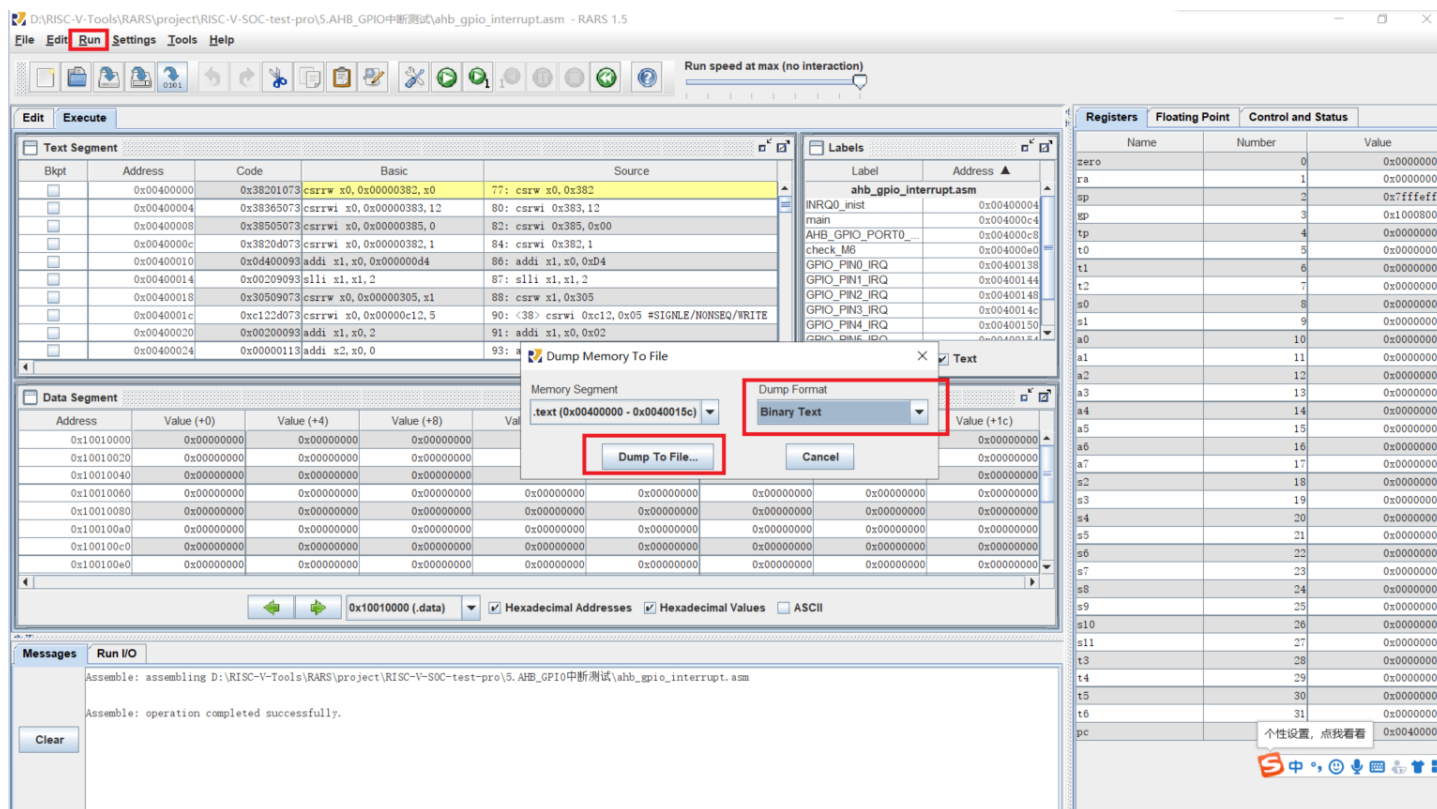
2022/7/8 22:21

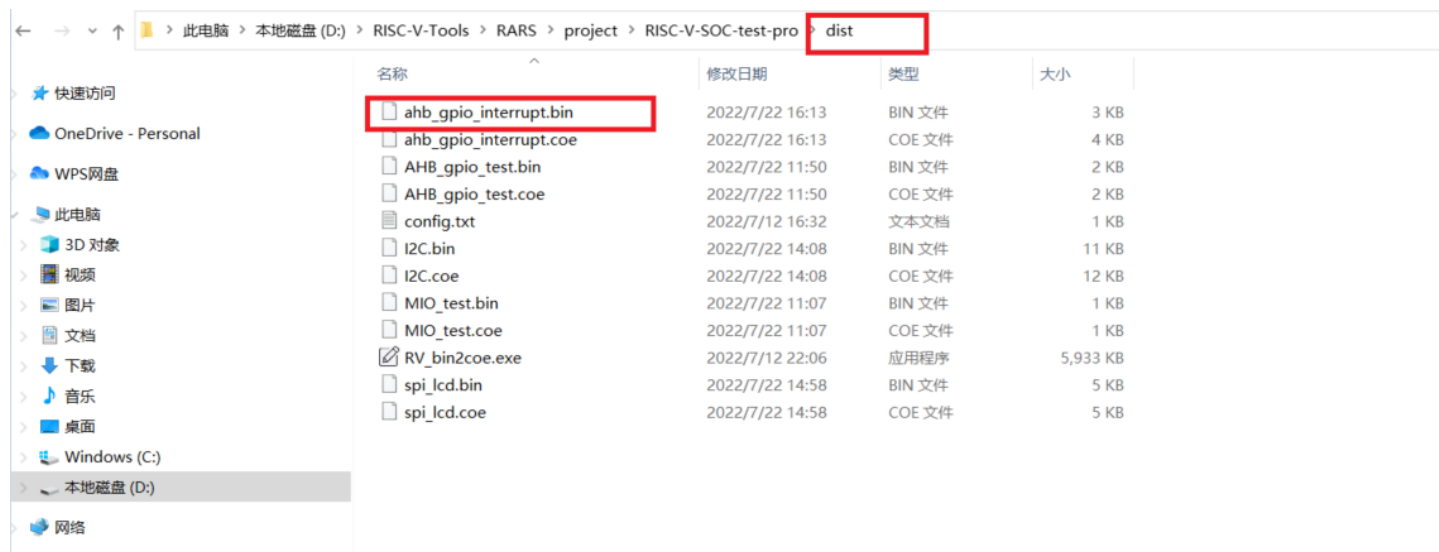
Executable Jar File

1,812 KB



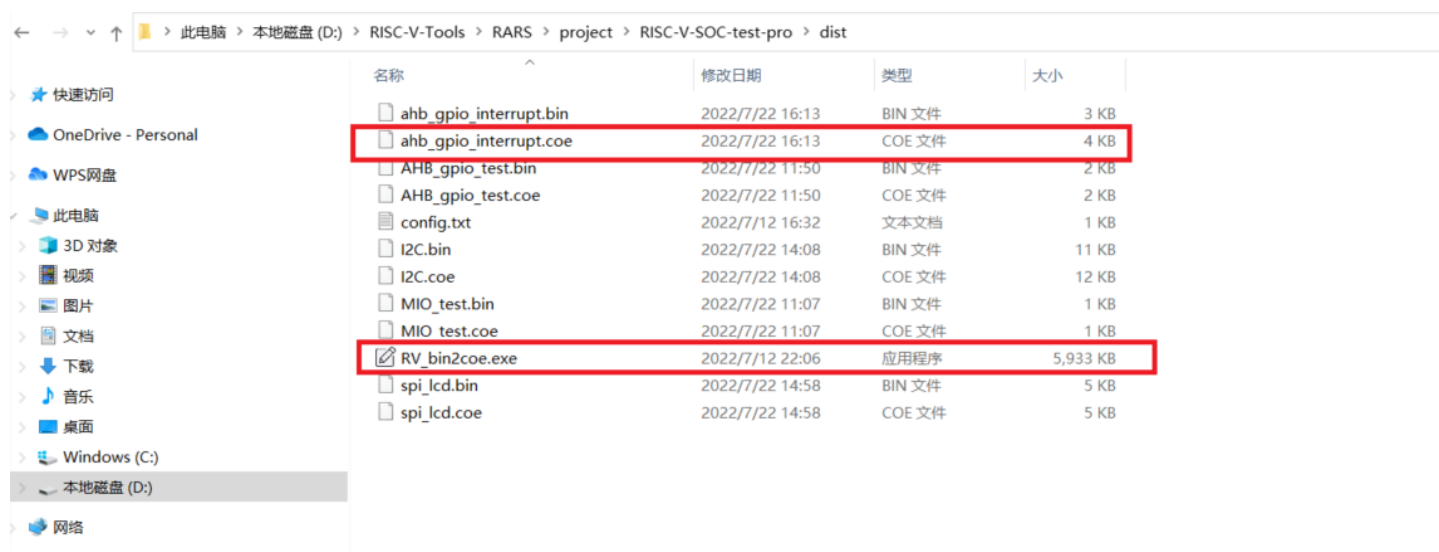
然后通过RARS生成机器码文件。文件路径选择为dist，后缀选择为bin。



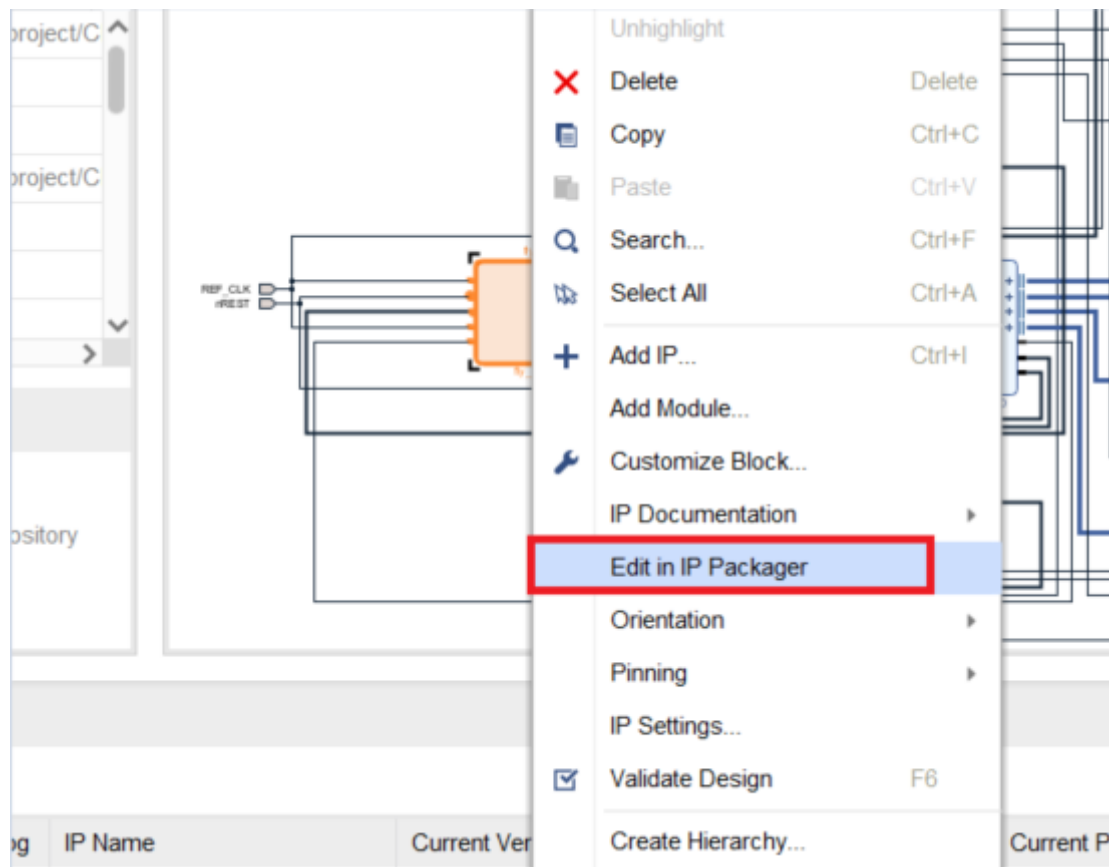


将生成的bin文件拖拽至RV_bin2coe.exe脚本文件上，制作可供Vivado ROM使用的coe文件。

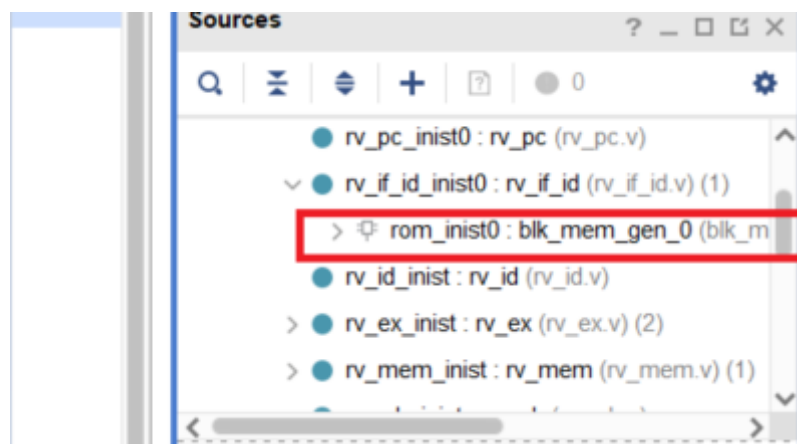
(注意：请务必保留config.txt文件，其中包含了coe文件的基本配置。如使用者需要也可自行修改)

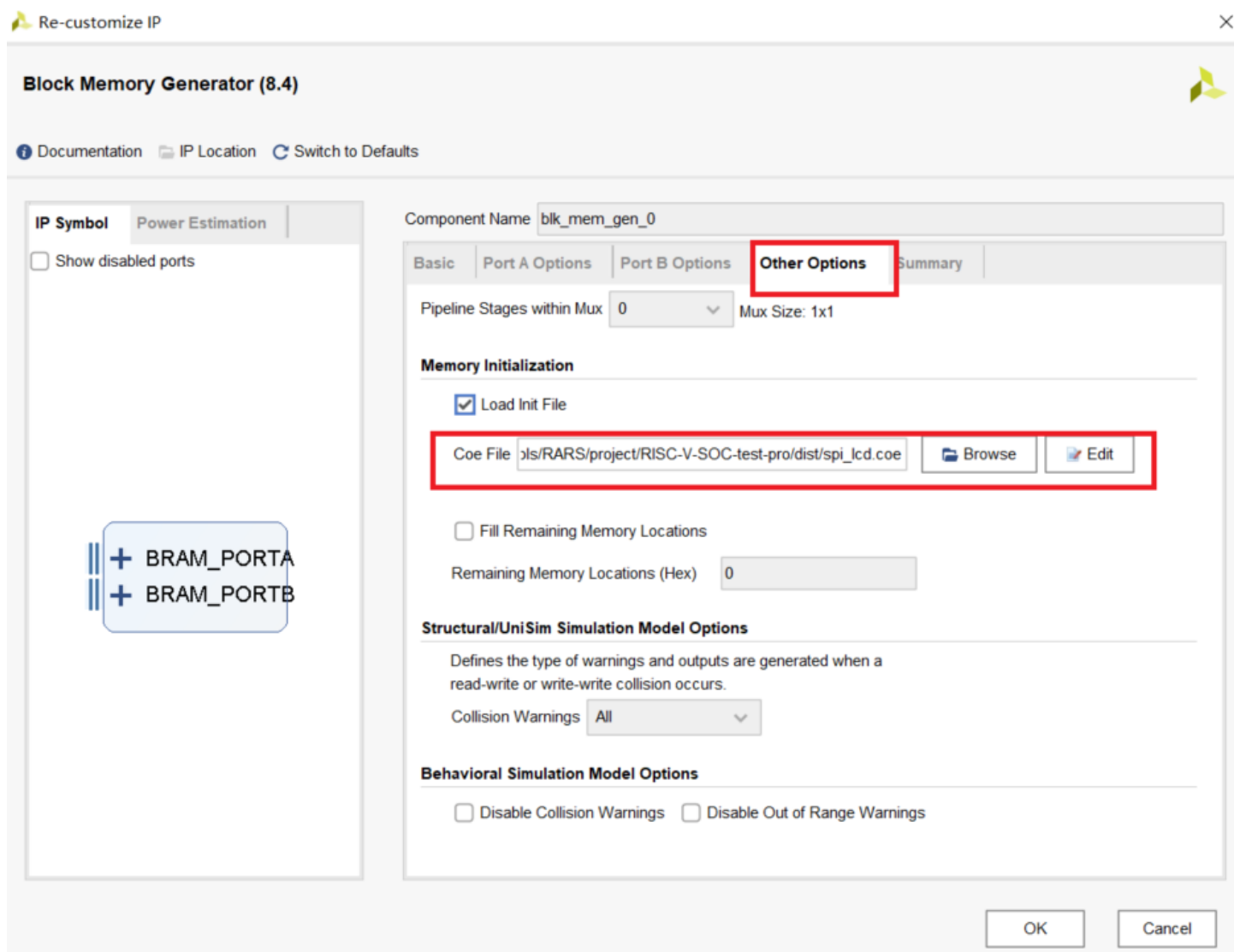


打开Vivado工程，如果是SOC工程，则右击fly_v_top IP核，选择edit in ip packager。

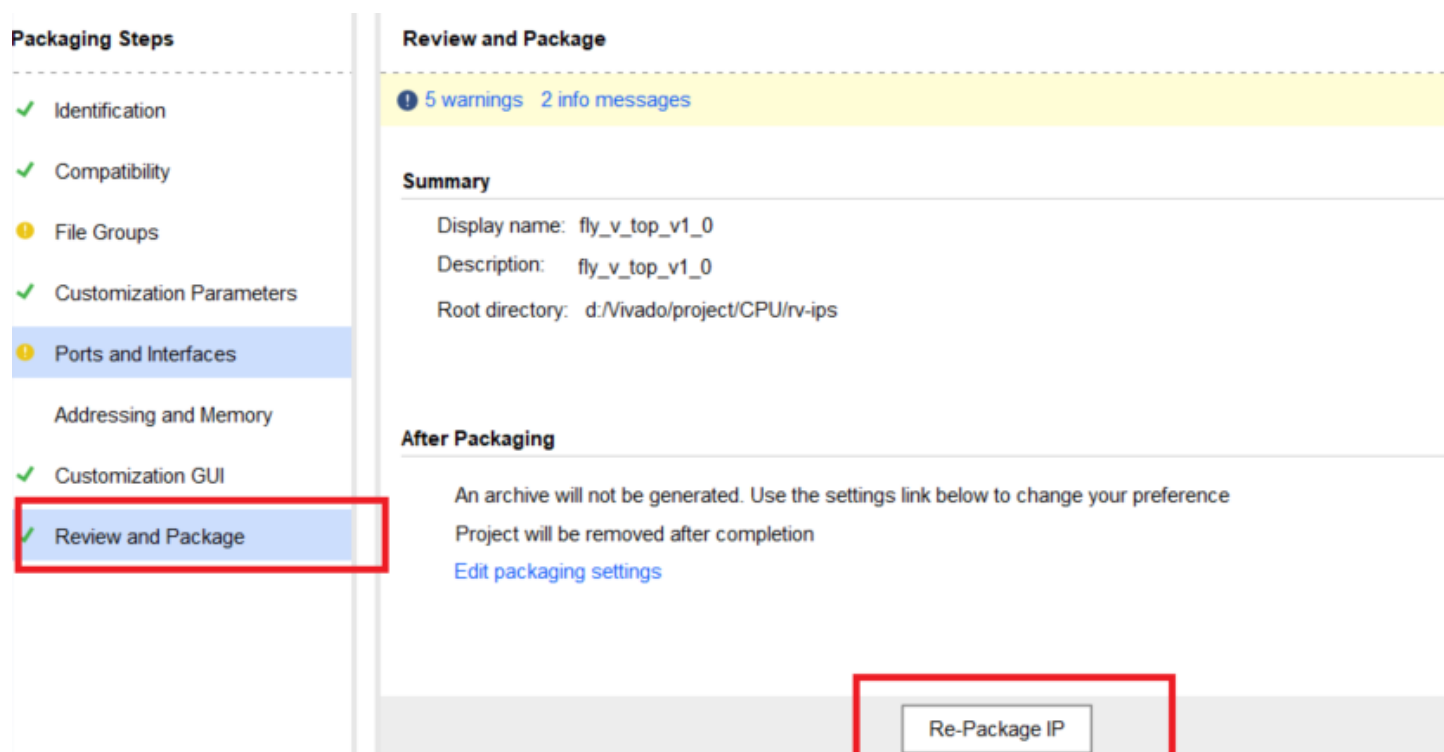


找到if模块下的ROM文件，双击打开后，将其中的coe文件选择为需要烧录的文件。





点击ok并生成ROM实例后，在打包器中重新打包。



最后返回soc工程，更新IP核即可。

如果是CPU测试文件，则只要找到if-id下的rom，按照上述步骤生成实例ROM即可。