YOLO 卷积函数的 SMART 仿真

一、实验目的

通过将 YOLO 源码的卷积函数放在 SMART 上进行仿真和性能监测，进一步介绍嵌入式 C 裸机程序在 C910 上的开发过程和仿真原理。通过观测分支预测开关对程序性能的影响，理解分支预测在现代处理器中的重要性。通过指令分布统计，了解内存墙的概念，并进一步理解缓存在现代处理器中的重要性。

二、实验步骤（包括实验结果，数据记录、截图等）

1. 文件核对和执行仿真

1) 进入 smart9_release 工作目录，输入以下命令完成文件替换

```
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[57]cp -rf /ho
me/ECDesign/ECDesign_share/lab9/ ~
cp: overwrite '/home/ECDesign/ecd09/lab9/Makefile'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/conv_test/data_in'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/conv_test/conv_test.c'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/conv_test/conv_test.h'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/conv_test/work.h'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/conv_test/inst_proc'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/crt0.s'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/soc.v'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/tb.v'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/conv_test.h'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/axi_slave128_copy.v'? y
cp: overwrite '/home/ECDesign/ecd09/lab9/pmu.h'? y
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[58]y
y: Command not found.
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[59]cp -f ~/la
b9/axi slave128 copy.v ./rtl/platform/amba/axi/
cp: cannot stat '/home/ECDesign/ecd09/lab9/axi': No such file or directory
cp: cannot stat 'slave128': No such file or directory
cp: cannot stat 'copy.v': No such file or directory
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[60]cp -f ~/la
b9/axi_slave128_copy.v ./rtl/platform/amba/axi/
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[61]ls
./   case/        lib/       readme  run_smart*  setup.csh  tmp/    workdir/
../  debug_test/  mem_intf/  rtl/    script/     tb/        tools/
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[62]cp -f ~/la
b9/soc.v ./rtl/platform/common/
cp: overwrite './rtl/platform/common/soc.v'? y
```

```
cp: overwrite './rtl/platform/common/soc.v'? y
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[63]cp -f ~/la
b9/Makefile ./lib/
cp: overwrite './lib/Makefile'? y
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[64]cp -f ~/la
b9/tb.v tb/
cp: overwrite 'tb/tb.v'? y
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[65]cp -f ~/la
b9/pmu.h ./lib/clib/
cp: overwrite './lib/clib/pmu.h'? y
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[66]cp -rf ~/l
ab9/con
conv_test.h  conv_test/
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[66]cp -rf ~/l
ab9/conv_test/ case/
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[67]
```

2) 在 workdir 目录下输入以下命令进行仿真

```
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release>[67]cd workdir/
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/smart9_release/workdir>[68]../tools/r
un_case ../case/conv_test/conv_test.c
```

（这是第一次仿真失败的截图，之后为了节省时间，创建了 lab9_hdw 文件夹，将 smart9

文件夹分别复制到 1，2，3 目录进行仿真）

提交仿真后在目录 case/conv_test/ 下对生成的二进制文件执行 ../../tools/toolchain/RV64GC/bin/riscv64-unknown-elf-objdump -S conv_test.elf > conv_test.s 可以进行反汇编

gvim 查看 conv_test.s,找到其中的 guard_start 和 guard_end 的重定向入口分别为 1b50 和 1c10

```
void guard_start()
{
    1b50:       1141
    1b52:       e406
```

```
void guard_end()
{
    1c10:       1141
    1c12:       e406
```

核对 tb/tb.v 中开头部分的定义：

```
`define GUARD_START_PC      40'h1850
`define GUARD_END_PC        40'h1910
```

发现两者并不相符，应当将 tb.v 中的宏定义进行修改，使之相匹配

```
`define GUARD_START_PC      40'h1b50
`define GUARD_END_PC        40'h1c10
```

终止之前的仿真任务，bkill -r [任务号]删除 conv_test.s 并重新提交任务

之后再次查看 conv_test.s 和 tb.v 中的宏定义，确认两者一致后即可等待结果。

2) 打开 lib 目录下的 crt0.s

```
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/lab9_hdw/3/smart9_release/workdir
>[133]
```

可看到刚才是对 all prediction on 的情况进行仿真，接下来分别取消对应的注释，分别对三种情况进行仿真。

```
li x3, 0x10f7     #all prediction on
#li x3,0x00b7      #BTB,L0BTB off
#li x3,0x0007      #all prediction off
```

仿真结束后打开 run.log 对仿真结果进行查看，并填入下表

```
num_cycle is 5202663
num_instret is 9449136
CPI is 0.550597
num_conditional_branch_mis is 4439
num_L1_Dcache_read_access is 3780647
num_L1_Dcache_read_miss is 14250
num_L1_Dcache_write_access is 1897811
num_L1_Dcache_write_miss is 1858
num_L2_Dcache_read_access is 469435
num_L2_Dcache_read_miss is 4164
num_L2_Dcache_write_access is 121027
num_L2_Dcache_write_miss is 399
guard_cycle_count is    5203315
guard_inst_count is    8681109
**********************************************
*    simulation finished successfully        *
**********************************************
```

```
num_cycle is 5131578
num_instret is 9449136
CPI is 0.543074
num_conditional_branch_mis is 5006
num_L1_Dcache_read_access is 3795212
num_L1_Dcache_read_miss is 14227
num_L1_Dcache_write_access is 1898553
num_L1_Dcache_write_miss is 1827
num_L2_Dcache_read_access is 470737
num_L2_Dcache_read_miss is 4164
num_L2_Dcache_write_access is 121008
num_L2_Dcache_write_miss is 399
guard_cycle_count is    5132230
guard_inst_count is    8724951
**********************************************
*    simulation finished successfully        *
**********************************************
```

```
num_cycle is 8936645

num_instret is 9449136

CPI is 0.945763

num_conditional_branch_mis is 253598

num_L1_Dcache_read_access is 4579889

num_L1_Dcache_read_miss is 12291

num_L1_Dcache_write_access is 1920568

num_L1_Dcache_write_miss is 1805

num_L2_Dcache_read_access is 481926

num_L2_Dcache_read_miss is 4183

num_L2_Dcache_write_access is 122750

num_L2_Dcache_write_miss is 400

guard_cycle_count is    8937286

guard_inst_count is    8718437
***********************************************
*    simulation finished successfully       *
***********************************************
```

2. CPI、Cache 缺失率和分支预测统计

configure	all prediction off	BPE on, BTB off	all prediction on
cycle	8936645	5202663	5131578
insts	9449136	9449136	9449136
CPI	0.945763	0.550597	0.543074
conditional branch miss	253598	4439	5006
L1_Dread access	4579889	3780647	3795212
L1_Dread miss	12291	14250	14227
L1_Dread miss_rate	0.002683689	0.003769196	0.003748671
L1_Dwrite access	1920568	1897811	1898553
L1_Dwrite miss	1805	1858	1827
L1_Dwrite miss_rate	0.000939826	0.000979023	0.000962312
L2_Dread access	481926	469435	470737
L2_Dread miss	4183	4164	4164
L2_Dread miss_rate	0.008679756	0.008870238	0.008845704
L2_Dwrite access	122758	121027	121008
L2_Dwrite miss	400	400	399
L2_Dwrite miss_rate	0.003258443	0.003305048	0.003297303

3. 指令分布统计绘图

将 inst_proc 复制到 workdir 目录下执行，并对各类型数据进行统计得到下图

```
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/lab9_hdw/1/smart9_release/workdir
>[183]cp -f ../case/conv_test/inst_proc ./
cp: overwrite './inst_proc'? y
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/lab9_hdw/1/smart9_release/workdir
>[184]chmod +x inst_proc
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/lab9_hdw/1/smart9_release/workdir
>[185]./inst_proc
inst_class created.
admin:/home/ECDesign/ecd09/huangdaiwei_21307140008/lab9_hdw/1/smart9_release/workdir
>[186]gvim inst_class
```

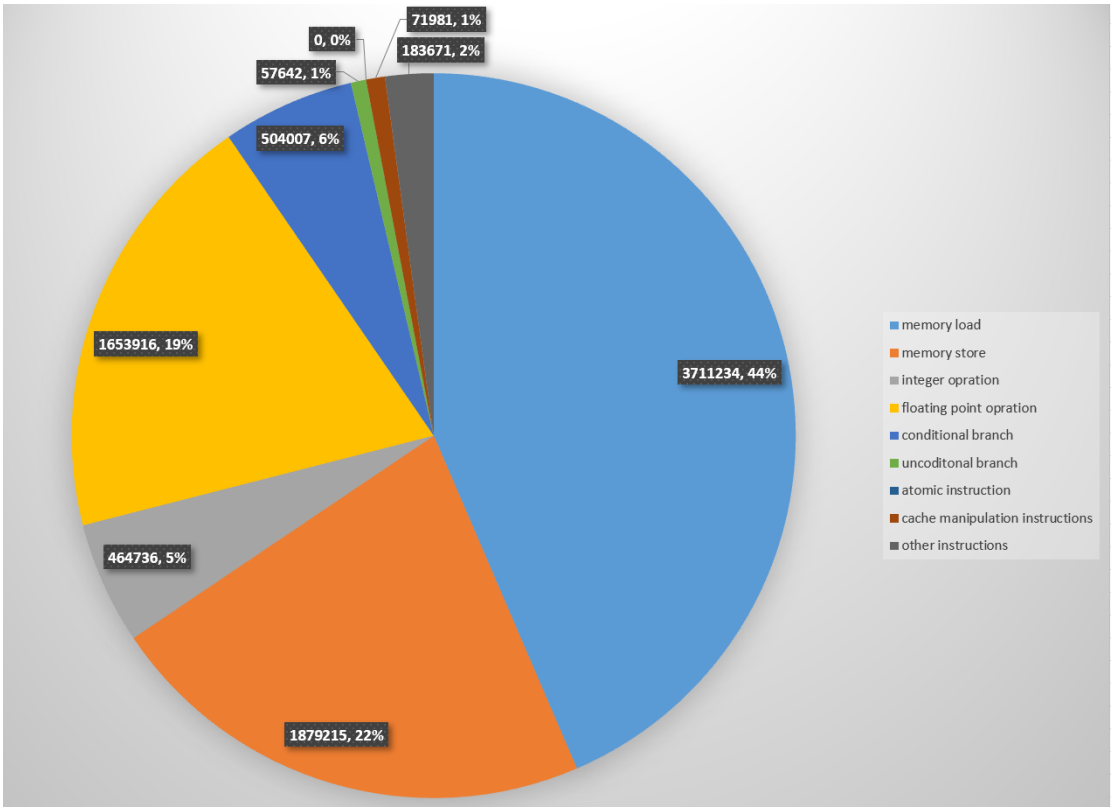
```
memory access instructions:
        memory load:    3711234
        memory store:   1879215

arithmetic  instructions:
        integer operation:      464726
        floating ponit operation:       1653916

branch instructions:
        conditional branch:     504007
        unconditional branch:   57642

atomic instructions:    0
cache manipulation instructions (C910 customized extension):    71981
other instructions:     183617

total:  8526338
~
```



4. Optional

   inst_count 截图

```
op[1:0]=11:
00:         1
01:   3711233
02:     71981
03:         0
04:    317720
05:         0
06:     18347
07:         0
08:        15
09:   1879200
0a:         0
0b:         0
0c:     25348
0d:        17
0e:     99751
0f:         0
10:   1622932
11:         0
12:         0
13:         0
14:     30984
15:         0
16:         0
17:         0
18:    501125
19:         0
1a:         0
1b:     57642
1c:        15
1d:         0
1e:         0
1f:         0
```

```
op[1:0]=00:
0:          0
1:          0
2:          0
3:          0
4:          0
5:          0
6:          0
7:          0

op[1:0]=01:
0:     103486
1:       3554
2:      80099
3:          6
4:          0
5:          0
6:       2882
7:          0

op[1:0]=10:
0:          0
1:          0
2:          0
3:          0
4:          0
5:          0
6:          0
7:          0
```

insc_record 截图

```
retire0:0000000b72
retire1:0000000b76
retire0:0000000b7a
retire1:0000000b7e
retire0:0000000b84
retire0:0000000b46
retire0:0000000b4a
retire0:0000000b4e
retire1:0000000b52
retire0:0000000b56
retire1:0000000b5a
retire0:0000000b62
retire1:0000000b66
retire0:0000000b6a
retire1:0000000b6e
retire0:0000000b72
retire1:0000000b76
retire0:0000000b7a
retire1:0000000b7e
```

```
retire1:0000000b5a
retire0:0000000b62
retire1:0000000b66
retire0:0000000b6a
retire1:0000000b6e
retire0:0000000b72
retire1:0000000b76
retire0:0000000b7a
retire1:0000000b7e
retire2:0000000b84
retire0:0000000b46
retire0:0000000b4a
retire0:0000000b4e
retire1:0000000b52
retire0:0000000b56
retire1:0000000b5a
retire0:0000000b62
retire1:0000000b66
retire0:0000000b6a
retire1:0000000b6e
retire0:0000000b72
retire1:0000000b76
retire0:0000000b7a
retire1:0000000b7e
retire2:0000000b84
```

三、实验分析和总结

(一)实验分析

1. 对 Cache 缺失率的统计

(1) 条件分支语句的缺失数量，当 BPE 打开而 BTB 关闭的时候缺失数量最少，而在所有预测关闭的时候缺失数量最大；

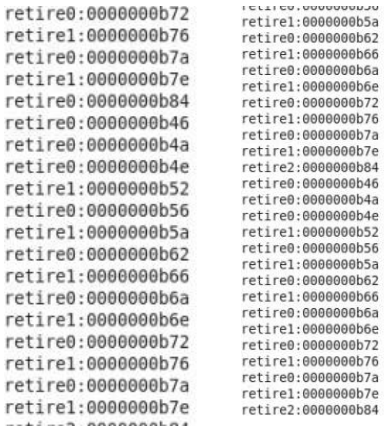
(2) L1 L2 的读缺失，当 BPE 打开而 BTB 关闭的时候缺失率最高，当分支预测全部关闭时缺失率最低；

(3) L1 L2 的写缺失，当 BPE 打开而 BTB 关闭的时候缺失率最高，当分支预测全部关闭时缺失率最低；

分析：

- 由于本实验中的读写缺失率都非常低且相差很小，所以其实分支预测和 cache 关系并不明显；
- 在其他分支预测器都开启的情况下，打开 BTB 反而会提升缺失率：分支预测导致进入了某些本来不会执行的控制流，虽然最后会被取消，但仍然会造成额外的 cache 访问；

- Cache 大小对实验结果影响不大的原因是：由 inst_record 可以看出，该函数循环重复多次。由 cache 的工作原理可知，它会将程序高频的代码和数据存放其中，便于寻找。而循环的重复率很高，其他非循环内部的指令与数据相对较少，因此能够非常轻易的将所需要的绝大部分数据与指令存放在 cache 中，且所占存储并不会非常高，故 miss rate 很低且随着 cache 大小变化不大。



2. 对 CPI 统计

随着分支预测开关的逐步打开，CPI 递减，CPU 性能递增；

正如实验 8 中分析过的，分支预测可以预先知晓需要跳转的目标地址，使得程序进行到分支语句时所需要的周期数更少，从而使得 CPI 更低，CPI 的下降也意味着 CPU 性能的提升。

3. C 程序在 smart 的仿真流程

要将 C 程序放入 smart 平台进行仿真，主要要经过三个前期准备阶段，分别是：源码编译和链接、生成便于 RTL 仿真模型加载的文件以及加载指令和数据到仿真模型。

(1) 源码编译和链接：smart 平台使用哈佛架构，将数据和指令分离，分别存放在 MEM1 和 MEM2 中。在程序开始仿真时，CPU 从起始地址（0x0）开始读取，即从 crt0.o 开始读取。在对各种寄存器、栈指针等初始化后，跳转到用户提交的 C 程序主函数入口开始执行，并生成完全链接的二进制文件（.elf）。

(2) 生成便于 RTL 仿真模型加载的文件：文件每行小端存放 4 组 4 个字节的数据，便于 testbench 读入；

(3) 加载指令和数据到仿真模型：模型每次传出 16 字节的数据，而 pc 使用其最低 4 位对模块内的数据进行寻址，从而确定具体对应的字节。接着 tb.v 将数据读入中间数组，并使用 initial 过程对内存模型初始化。其中链接脚本 linker.lcf 所划分的 MEM1、MEM2 和栈空间，其地址全部落在 SRAM_START 和 SRAM_END 之间，因此总是操作 x_axi_slave128 这一模块内的内存模型。

4. 内存墙与指令统计

(1) 定义：内存墙指内存性能严重限制 CPU 性能发挥的现象。在过去的二十多年中，处理器的发展速度与内存的发展速度严重失衡。处理器以每年 55%左右的速度进行发展，而内存发展速度仅为 10%左右。所以如今，DRAM 传输速度不断增加，但访问延迟较大的问题始终无法解决。

DRAM 延迟使得 CPU 需要花时间等待内存访问完成，带来限制 CPU 性能、增加 CPU 发热量等问题。因此，科学家将这种现象称为内存墙。

(2) 解决方法：内存的主要衡量标准是带宽和等待时间。因此要提高内存的能力，就需要从这两个方面入手。

- 提高内存带宽：内存带宽=（传输倍率×总线位宽×工作频率）÷8，因此，可以通过提高总线位宽、提高工作频率和提高内存传输倍率（如将 DDR 内存改为 DDR2 或 DDR3）
- 降低等待时间：运用 cache，根据时间局域性原理，将处理器可能访问的数据和代码预先存储到高速缓存中，使得程序对于高频数据的访问时间大大缩小，从而降低等待时间。

(3) 在本次试验中，可以看出在指令统计中加载与存储（访存类指令）总共占了指令总数的 66%，远高于算术运算指令的 24%与分支指令的 7%。因此可以看出，在现代处理器中，由于内存墙的存在，访存类指令占据了 CPU 性能的绝大部分，对 CPU 的性能造成了极大的限制。也进一步说明了缓存的重要性。

4. Optional

- **insc_count 解析**

通过实验指导文档可知，在 risc-V 指令集中可通过判断最低的两位来判断指令属于 32 位(G,2'b11)或 16 位(C, 2'b00、2'b01、2'b10),同时结合高三位编码来判断具体指令类型

查阅 risc-V 可知部分指令对应的编码如下，由于篇幅过长，只截取部分

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
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
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
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
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
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV64I Base Instruction Set (in addition to RV32I)

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV32/RV64 *Zifencei* Standard Extension

imm[11:0]	rs1	001	rd	0001111	FENCE.I

RV32/RV64 *Zicsr* Standard Extension

csr	rs1	001	rd	1110011	CSRRW
csr	rs1	010	rd	1110011	CSRRS
csr	rs1	011	rd	1110011	CSRRC
csr	uimm	101	rd	1110011	CSRRWI
csr	uimm	110	rd	1110011	CSRRSI
csr	uimm	111	rd	1110011	CSRRCI

RV32M Standard Extension

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

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

对应 riscv_spec 和 insc_count 的结果可知：

32 位指令：

LB:1　　LH:3711233

SB:15　　SH:1879220

ADDI3:17720　　SLTI:18374　ADD/SUB:25348　SLL:17　ADDW/SUBW:99751

FMADD:1922932　　FDD:30984

BEQ:501125　JAL:57642　ECALL/EBREAK:15

C910 customized extension:71981

16 位指令

Others:183617

以上某一分类的总和基本上与 insc_count 中的结果一致

- **insc_record 解析**

  图片和分析见上

四、实验收获、存在问题、改进措施或建议等

通过本次实验，具体操作了将 YOLO 源码放到 smart 平台上进行仿真的过程，了解了 C 语言在 smart 平台上仿真的步骤与原理，理解了内存墙的概念，并且进一步深入了解了分支预测与 cache 在现代处理器中的重要性。