Ibex 指令集扩展项目总结

1 单个模块的设计和实现

设计了一个 uart 模块并将其挂载在 SoC 的 APB 上。

uart 一共有 5 子个模块，分别为 rx_div、rx、tx_div、tx、uart_top 模块。其中rx 为输入模块，tx 为输出模块，rx_div 和 tx_div 分别产生输入和输出模块的时钟，uart_top 为顶层模块。

该 uart 是一个没有 FIFO 的简化版本的 uart，数据的收发主要通过有限状态机实现。接收模块有 IDLE/START/RECV_DATA/CHECK/FINISH 5 个状态，发送模块有 IDLE/START/DATA/CHECK/STOP/FINISH 5 个状态。

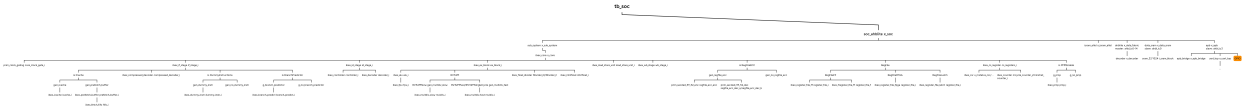
uart 支持的功能为：

1. 波特率可配置，支持常见波特率（9600, 19200等）
2. 数据位可配置（5/6/7/8位）
3. 停止位可配置（1位或2位）
4. 支持发送和接受中断，中断可单独使能或屏蔽
5. 可通过中断获得发送或接收状态，也可通过状态寄存器查询发送或接收状态
6. 支持奇偶校验位

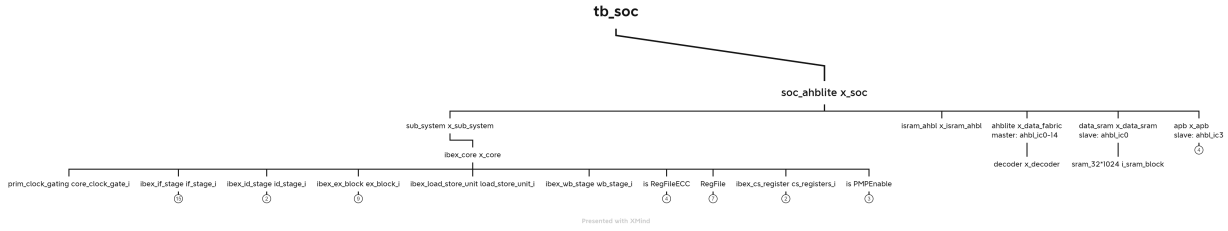
这些配置都可以通过向 uart 的配置寄存器中写相应的配置信息实现。

2 SoC的集成

在做项目的过程中画了一个 SoC 的整体结构框图：



略去子模块的细节，结构框图为：



简单描述一下，总线上挂载着这些模块：子系统（sub_system）模块、指令 sram （isram_ahbl）模块、数据 sram（data_sram）模块、ahb2apb 桥（apb）、总线译码（ahblite）模块。其中指令 sram 自己占用一条总线，数据 sram、ahb2apb 桥共享一条总线，由总线译码模块统一管理。GPIO、uart 等低速模块都挂载在 apb 上，根据地址的特点实现在 core 和不同模块之间数据的传输。处理器核心位于子系统内，指令 sram 根据处理器核心给出的指令地址将指令传送给处理器，处理器通过给出数据地址将数据写入数据 sram、GPIO 寄存器或 uart 寄存器内。

3 总线的理解

本项目中，总线是已经给出的，我只研究了如何使用总线。

本项目中的总线分为指令总线和数据总线，其中数据总线是数据 sram 和 apb 上挂载的低速模块共享的，处理器通过不同的地址将数据写入不同的模块。

总线上有一个译码模块，负责统一管理挂在数据总线上的所有模块（本项目中，挂在数据总线 ahb 上的模块只有数据 sram 和 ahp2apb 桥）。具体方式为：ahblite 模块通过解析处理器传入的地址决定数据传输的目的地是数据 sram 还是 ahp2apb 桥，然后将数据传入指定模块；其中若数据传入 ahp2apb 桥模块，则 ahp2apb 模块根据地址决定将数据传给哪一个挂载在其下的子模块。

除数据 sram 和 ahp2apb 桥之外，在做 SoC 扩展时还可以在总线上挂载其它模块，从而使整个 SoC 具有良好的可扩展性。

4 RISCV 指令集架构的理解

RISCV 指令集中的基础整型指令集为 RV32I，其标准扩展指令集主要包括 RV32M（主要负责整数的乘除和取模运算）、RV32A（原子操作）、RV32F/D（单双精度浮点指令）等。其中指令的 [6:2] 位被主要用来区分不同的扩展指令集。ibex 支持 RV32I、RV32C、RV32M、RV32B 等拓展，但是不支持浮点指令。

RISCV 指令集中的每一条指令都较为简单，实现比较容易，也没有很复杂的控制逻辑，和 MIPS 指令集比较像。

5 扩展指令集的设计

5.1 硬件的设计实现

5.1.1 浮点数寄存器堆

为了实现浮点指令集的扩展，需要增加一个浮点寄存器堆。浮点数寄存器堆和整数寄存器堆几乎没有区别，含有 32 个 32 位的寄存器。为了支持三个源操作数的运算指令，我为浮点寄存器堆设计了三个读端口。

5.1.2 浮点数运算单元

新加的浮点运算单元 ibex_fpu 可以在一个周期内实现加减乘的操作，浮点除法单元 ibex_float_divider 可以实现多周期除法，模块 ibex_int2float 可以实现定点数到浮点数的转化。这些单元的代码全部都来自开源社区，我在源码的基础上做了一些修改来让其适合 ibex 其他模块的接口和实现指令需要的功能。

ibex_fpu.sv 源码来自：(github.com)https://github.com/danshanley/FPU/blob/master/fpu.v

ibex_float_divider.sv 源码来自：https://github.com/dawsonjon/fpu/blob/master/divider/divider.v

ibex_int2float.sv 源码来自：https://github.com/dawsonjon/fpu/blob/master/int_to_float/int_to_float.v

5.1.3 SRAM 的修改

ibex 原来的数据 sram 是由四块 $8bit \times 1024$ 的 SRAM 拼接而成的，为了方便，我将其改为一整块的 $32bit \times 1024$ SRAM。

5.1.4 GPIO

为了验证代码是否可以在 FPGA 板上成功运行，我设计了一个 GPIO 模块。使用方法为在汇编代码最后将存储算法结果的内存块中的数据写到 GPIO 的寄存器堆里，然后通过拨码开关选择要看 GPIO 寄存器堆里哪个寄存器的内容。GPIO 挂载在 APB 上。

5.2 软件编译工具链的生成使用及修改

软件编译工具链的生成和修改步骤如下：

1. 下载 rsicv gnu toolchain

   ref：https://blog.csdn.net/ALLap97/article/details/106345045

2. 安装依赖库

```
sudo yum install autoconf automake python3 libmpc-devel mpfr-devel gmp-devel gawk bison flex texinfo patchutils gcc gcc-c++ zlib-devel expa
```

3. 安装 gmp,mpfr,mpc，并执行：

```
cd riscv-gnu-toolchain/riscv-gcc
contrib/download_prerequisites
```

4. 编译工具链

   ref：https://blog.csdn.net/weiqi7777/article/details/88045720

5. 参考/work/AHBL_SOC_IBEX/tools/setup.csh将该目录加入环境变量

6. 添加自定义指令，然后重复步骤4，重新编译工具链，得到添加自定义指令后的工具链
   ref：https://cloud.tencent.com/developer/article/1886469

7. 修改完毕之后直接在 .s 文件里写自定义的指令，之后编译时即可被工具链识别。

注意：这样对编译工具的修改存在局限性：定义的指令必须符合现有的 RISCV 指令的模板才能被识别。例如 RV32I 型指令只有 R/I/S/B/U/J 六种类型（模板），若自定义的 RV32I 指令的格式不符合这六种类型中的任何一种（比如有三个源操作数的整数指令），就无法通过修改编译工具链的方式让工具链识别这种指令。

5.3 浮点指令集扩展

ibex 处理器并不支持 RV32F 浮点指令集中的指令，而 NMS 算法的实现涉及到浮点数运算，因此需要实现浮点指令集。为此我实现了 RV32F 指令集的一个子集， 包括如下15条指令（这些指令均是 RISCV 官方标准的扩展指令集中的指令）：

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
FLW	imm[11:0]			rs1	010	rd	0000111
FSW	imm[11:5]		rs2	rs1	010	imm[4:0]	0100111
FMADD.S	rs3	00	rs2	rs1	rm	rd	1000011
FMSUB.S	rs3	00	rs2	rs1	rm	rd	1000111
FNMSUB.S	rs3	00	rs2	rs1	rm	rd	1001011
FNMADD.S	rs3	00	rs2	rs1	rm	rd	1001111
FADD.S	0000000		rs2	rs1	rm	rd	1010011
FSUB.S	0000100		rs2	rs1	rm	rd	1010011
FMUL.S	0001000		rs2	rs1	rm	rd	1010011
FDIV.S	0001100		rs2	rs1	rm	rd	1010011
FMIN.S	0010100		rs2	rs1	000	rd	1010011
FMAX.S	0010100		rs2	rs1	001	rd	1010011
FEQ.S	1010000		rs2	rs1	010	rd	1010011
FLTS	1010000		rs2	rs1	001	rd	1010011
FCVT.S.W	1101000		00000	rs1	rm	rd	1010011

5.4 针对特定算法的指令设计

根据 NMS 算法的汇编代码特点，我做了以下指令扩展以优化汇编代码的性能：

		31 25	24 20	19 15	14 12	11 7	6 0
RV32I							
①	addrtwo	0000000	rs2	rs1	001	rd	0001011
②	addrfive	0000001	rs2	rs1	001	rd	0001011
③	plusonelt	imm[11:0]	rs2	rs1	010	imm[4:0]	0001011
RV32F							
④	FSUBABS.S	0000010	rs2	rs1	rm	rd	1010011

5.4.1 addrtwo 指令

代码示例：

```
addrtwo r3,r1,r2
```

其中 r1 寄存器为内存区基址，r2 寄存器的内容为偏移量（以字为单位），r3 寄存器保存目标数据在内存中的绝对地址。

实现的功能为：rd=rs1+rs2>>2，在上例中即为：r3=r1+r2>>2。

该指令将以下两条指令合二为一：

```
mul r5,r2,r4
add r3,r1,r5
```

其中 r4 寄存器的内容为整数值 4，r2寄存器的内容为偏移量，r5 寄存器保存计算出的内存地址偏移量，r1 寄存器为内存区基址，r3 寄存器保存目标数据在内存中的绝对地址。

扩展这条指令的目的是简化原本在访问内存中的某个数据时乘加（乘法计算偏移量，加法计算绝对地址）的地址计算操作。这条指令的实现只需要一个周期，若 ALU 的乘法操作需要多个周期实现，则这两条新扩展的指令可以在很大程度上减少绝对地址计算所需的时钟周期数。由于我找的乘法计算单元只需要一个周期就可完成计算，故这两条指令的扩展对我的处理器而言并没有太大的效率提升；但是一般的乘法运算都无法在一个周期内完成，所以这条指令的扩展对大多数处理器运行 NMS 算法性能的提升都是有帮助的。

5.4.2 addrfive 指令

代码示例：

```
addrfive r3,r1,r2
```

其中 r1 寄存器为内存区基址，r2 寄存器的内容为偏移量（以字为单位），r3 寄存器保存目标数据在内存中的绝对地址。

实现的功能为：rd=rs1+rs2>>5，在上例中即为：r3=r1+r2>>5。

该指令将以下两条指令合二为一：

```
mul r5,r2,r4
add r3,r1,r5
```

其中 r4 寄存器的内容为整数值 32，r2寄存器的内容为偏移量，r5 寄存器保存计算出的内存地址偏移量，r1 寄存器为内存区基址，r3 寄存器保存目标数据在内存中的绝对地址。

扩展这条指令的原因与扩展 ① 号指令的目的相同。

5.4.3 plusonelt 指令

代码示例：

```
plusonelt r1,r2,DEST
```

实现的功能为：rs1=rs1+1，rs1<rs2？ 跳转：顺序执行。在上例中即为：r1=r1+1，若 r1 寄存器中的数据 +1 后小于 r2 寄存器中的数据，则跳转到 DEST 所指示的指令处，否则顺序执行。

该指令将以下两条指令合二为一：

```
addi r1,r1,1
blt r1,r2,DEST
```

扩展这条指令是基于对一次 for 循环结束后判断是否跳转的条件的观察。由于我写的 NMS 算法中有较多的 for 循环，而每次 for 循环结束都会执行 addi 和 blt 这两条指令检查是否可以跳出 for 循环，因此设计 plusonelt 指令将两条指令合二为一，每次执行时都可

以减少一个周期的时间开销。

5.4.4 FSUBABS.S 指令

代码示例：

```
fsubabs.s r1,r2,r3
```

实现的功能为：rd=|rs1-rs2|，在上例中即为：r1=|r2-r3|。

扩展这条指令是因为 NMS 算法在计算 IoU 时需要计算矩形的边长，而边长的计算涉及到求两点间的距离，若没有扩展这条指令，完成这样的一次计算需要 5 条指令，其中还包括了跳转指令，时间开销较大。由于计算边长是 NMS 算法中出现频率较高的操作，故实现此指令扩展可以减少计算时间开销。

5.5 其他扩展指令设计

我还进行了其他的指令集扩展，这些指令在后续的迭代中又被优化掉，没有被运用到最终的汇编代码中，也列举如下：

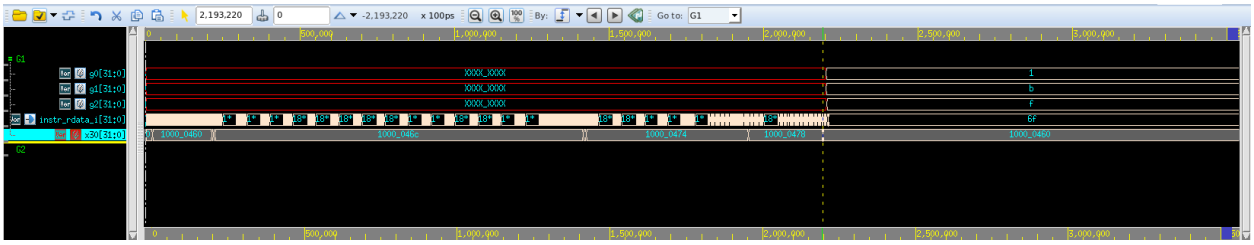
		31 27	26 25	24 20	19 15	14 12	11 7	6 0
RV32I								
⑤	MADD	rs3	00	rs2	rs1	000	rd	0001011
⑥	MUL	0000001		rs2	rs1	000	rd	0011011
RV32F								
⑦	FADDDIV.S	rs3	00	rs2	rs1	rm	rd	0101011

⑤ 号指令的功能是 rd=rs1*rs2+rs3，⑥ 号指令的功能是 rd=rs1*rs2，⑦ 号指令的功能是 rd=(rs1+rs2)/rs3，在此不再展开详细介绍。

其中 ⑤ 号指令因为有三个源操作数，而 RV32I 的指令模板里并没有三操作数的格式，因此该条指令在汇编代码被转化为机器码时无法被识别。使用这条指令的方法是在汇编代码中该条指令的位置用其他可以被识别的指令代替，然后在编译生成的 .vmem 文件中该指令的位置用此指令的二进制编码代替原占位指令。

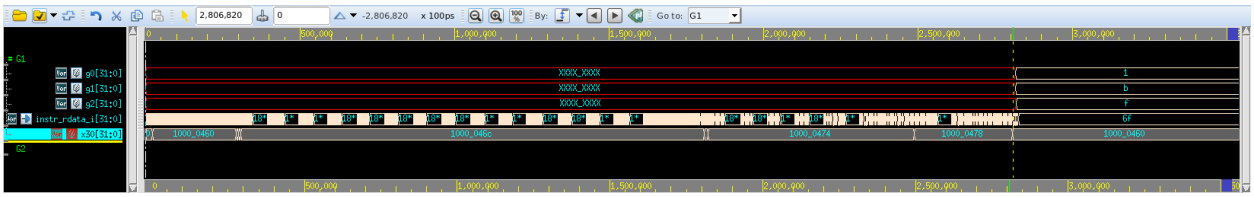
5.6 实验结果

优化前：



$t_1 = 280682ns$

优化后：



$t_2 = 219322ns$

加速比：$speedup = \frac{t_1}{t_2} = 1.28\text{x}$

注意：优化的代码相比优化前的代码多扩展了以下指令：

FMIN.S　FMAX.S　FSUBABS.S　addrtwo　addrfive　plusonelt

6 收获与经验教训

在一开始看到 ibex 处理器的源码的时候我是非常没有头绪的，因为其代码量较大，结构复杂，涉及到的信号非常多。我一开始看代码的方式是跟踪某一个信号，但是这样做的结果是容易迷失在与指令执行的关键步骤无关的细节里。后来根据师兄的建议，我先跟踪一条最简单的指令（add 指令）的数据通路，发现源代码中的很多模块在指令执行中并没有起到功能性的作用（比如异常处理单元），于是我就先忽略这些单元，把注意力放在关键的模块上，思路就清晰了很多。

整个项目的推进路径是：了解一条件单指令的数据通路 - 实现一条自定义运算指令的数据通路 - 实现浮点指令的扩展 - 根据 NMS 算法的汇编代码做指令集层面的优化，实现指令集扩展。这是一个循序渐进的过程，在将一个大的任务拆解成一个个阶段性的工作并逐步完成每个工作的过程中加深对 RISCV 指令集以及处理器的理解。这样的学习曲线对我而言是非常合适的，之后在做指令层面优化时我也是进行了很多次迭代，每次迭代提升一点性能，到最后就实现了性能上比较大的提升。

和同伴的交流非常重要，可以为工作带来新的思路。我和王文迅都优化了计算数据在内存中绝对地址的指令。我一开始的优化方法是把乘法指令（计算地址偏移量）和加法指令（计算绝对地址）合并成一条乘加指令，以减少运算的周期。在之后的讨论中我发现王文迅的思路也是将两条指令合为一条，只是他把乘法操作换成了移位。我觉得这是一个非常好的想法，虽然我的乘法单元在一个周期内就能完成运算，将乘法换成移位对我的处理器来说并没有性能（指计算用时）上的提升，但是鉴于很多处理器的乘法都是无法在一个周期内完成的，而使用移位操作后可以让计算在一个周期内完成，因此这样的指令设计在大多数处理器上都可以获得一定的性能提升。后续上 FPGA 板子验证的时候，我的行为级仿真正确，但是在 FPGA 板子上总是跑不出正确结果，和王文迅交流之后他建议把复位键绑定在拨码开关而不是按键上，我修改之后就在板子上跑通了。自行分析原因可能是因为按键有抖动等因素，导致复位结果不正确。可见项目推进中的很多问题都可以在交流中获得解决。

另外有两个提升工作效率的经验。第一个是关于写汇编代码。我先写了 NMS 算法的 C 代码，因为写 C 代码的最终目的是将其转化为汇编，而我可以利用的指令集只是 RISCV 指令集的一个子集，功能并不十分完全，因此我在写 C 代码的时候就尽量将其写得像汇编代码，尽量全部采用基础的加减乘除和分支判断，不采用封装得很好的函数，这样在转化为汇编代码时会较为容易。第二个是关于画电路结构的模块图。因为最后有上板子的需要，我设计了一个 GPIO 模块。在处理 core 与 GPIO 模块通过总线与转接桥传输数据的过程中，为了理清思路，我画了整个 SoC 的结构模块图，这个模块图帮我明确了各个信号的源宿和传输关系，也让我对 SoC 的整体结构有了更深入的认识。

将以上的收获与经验教训总结如下：

1. 看源代码时主要看对于功能实现起关键作用的代码片段，忽略不起决定性作用的细节；
2. 将大的任务拆解成由易到难的阶段性工作，步步深入；
3. 优化是一个一次次迭代的过程，不需要一蹴而就；
4. 同伴交流可以为工作带来高的效率和新的思路。

关于这个项目可能还可以有的更进阶的工作：在现有的基础上实现一些更先进的分支预测策略、增加运算单元实现数据的并行处理、实现指令的多发射和乱序执行和其他一些在体系结构上的改进（或许可以把体系结构课程上讲到的一些理论在硬件上做一个实现）。