wBPF - out-of-order coprocessor optimized for eBPF

周鹤洋

2022.11.12

简介

- Work: Cloud Infrastructure@Deno
- 这项工作是我的本科毕业设计
- GitHub @losfair

主要工作

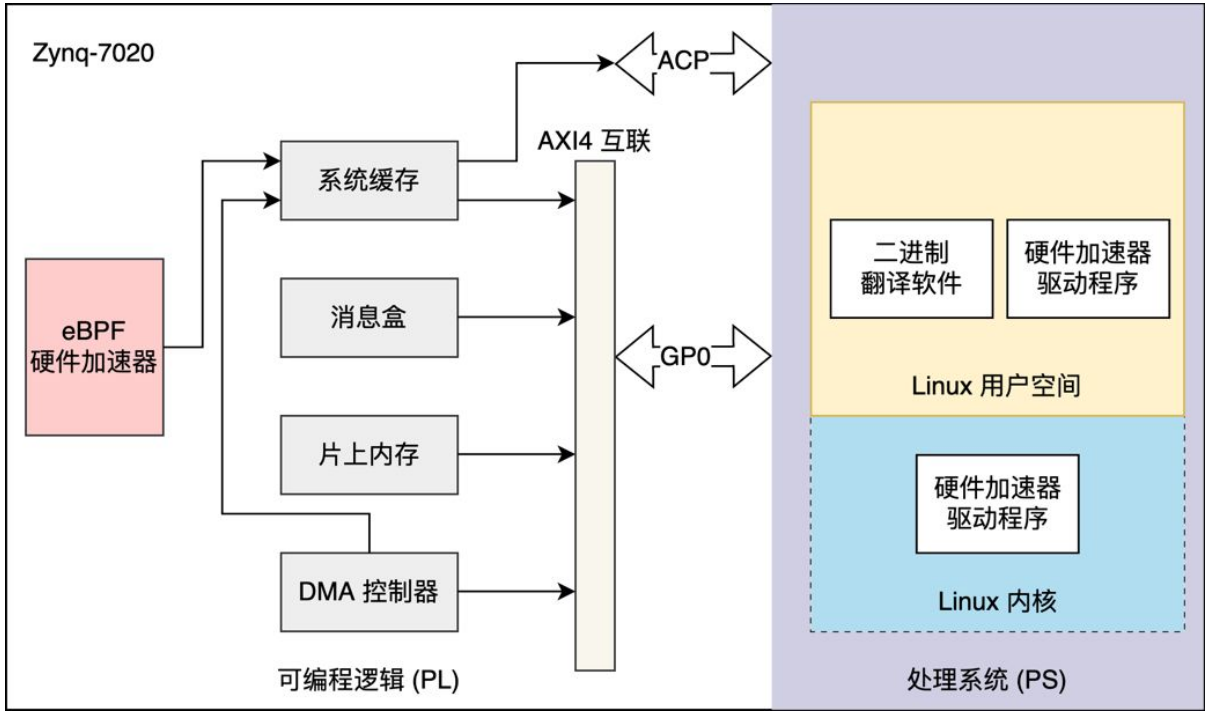
1. **硬件:** 面向FPGA的RISC-V处理器核心
   - 面向网络报文处理场景下的内存访问特征优化
   - 基于受限数据流机制实现指令乱序执行
   - 针对FPGA实现优化了面积用量与最高频率
2. **软件:** eBPF二进制翻译软件与Linux内核驱动程序
3. **方法:** 软硬件协同设计、硬件敏捷开发与敏捷验证、硬件自动化测试

Why

先前的硬件 eBPF 工作:hXDP/FFShark (VLIW/众核架构)

现代高性能应用处理器都采用乱序执行 (OoOE) 架构. 尝试针对 eBPF 应用场景设计专用的 OoO 处理器.

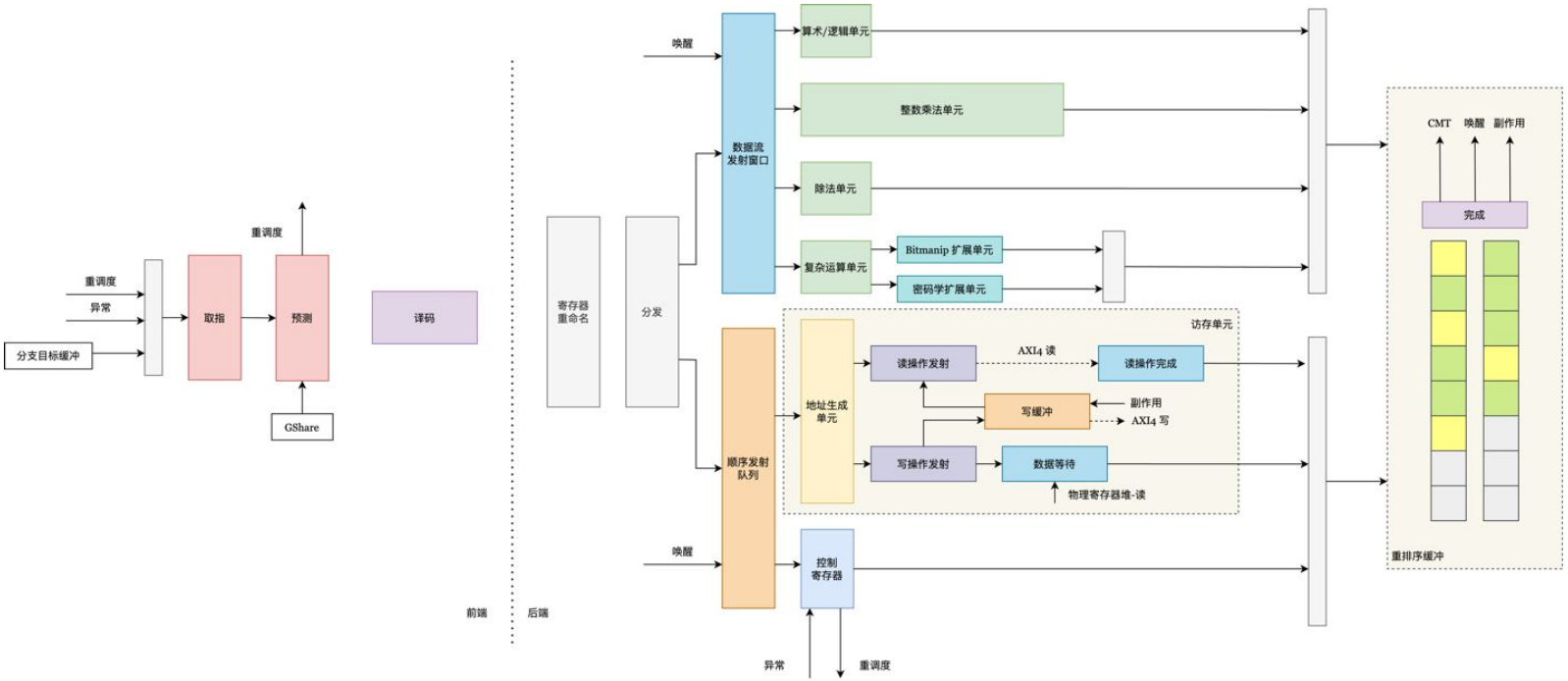
优势:动态指令级并行;无需缓存实时处理 DDR 数据



系统架构

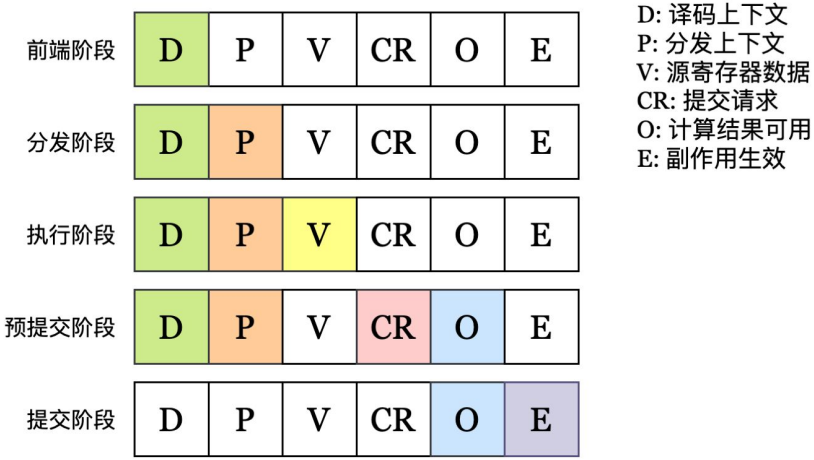
平台： Xilinx FPGA (Zynq-7020)
数字逻辑： 基于 AXI4 互联协议
软件： 基于 Linux

硬件系统：流水线结构



顺序执行	乱序执行	顺序执行

硬件系统：流程视角（指令生命周期）



D: 译码上下文
P: 分发上下文
V: 源寄存器数据
CR: 提交请求
O: 计算结果可用
E: 副作用生效

前端阶段：进入后端前的指令处理不涉及数据，均属于前端阶段，包括分支预测、取指与译码。

分发阶段：指令进入执行后端后的第一个阶段是分发（Dispatch），该阶段亦是动态数据流分析的第一阶段。指令分发逻辑利用前阶段的译码信号与物理寄存器状态表为指令依赖的数据与产生的数据分配物理寄存器，同时分配重排序缓冲槽。

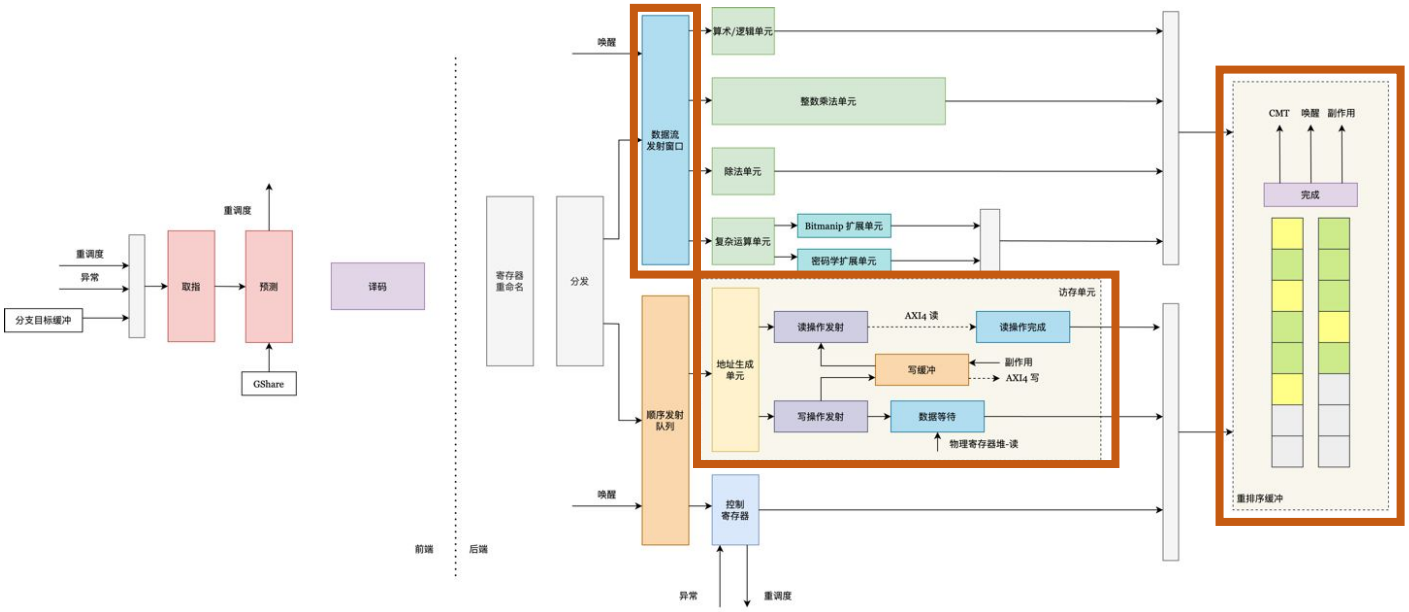
执行阶段：指令的所有数据依赖就绪后即会被发射，进入执行阶段。发射逻辑从物理寄存器堆中取得源寄存器数据，为指令生成发射上下文。

预提交阶段：指令执行完毕后进入预提交阶段，生成提交请求写入重排序缓冲，将指令计算结果写入目的寄存器，并生成唤醒信号。

提交阶段：预提交阶段指令的提交请求到达重排序缓冲头部后进入提交阶段，输出物理寄存器状态被更新为已提交，寄存器提交映射表（CMT）中与输出架构寄存器对应的项被更新为输出物理寄存器编号，副作用被广播到副作用总线上，指令生命周期结束。

硬件系统：关键结构

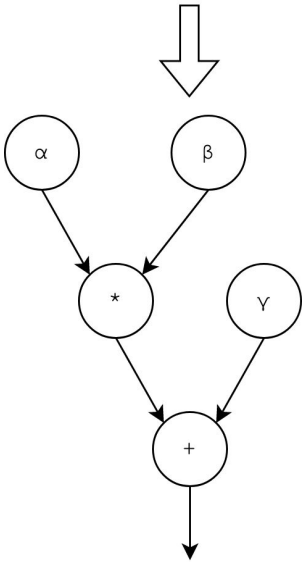
乱序发射单元 OoO-IQ：对指令进行数据等待，监听唤醒信号，将指令发射到功能单元
数据唤醒通道 Wake：向 OoO-IQ 通知数据就绪状态
重排序缓冲 ROB：将乱序完成的指令重排序为指令流顺序
访存子系统 LSU：执行指令的内存访问操作，并保证副作用顺序



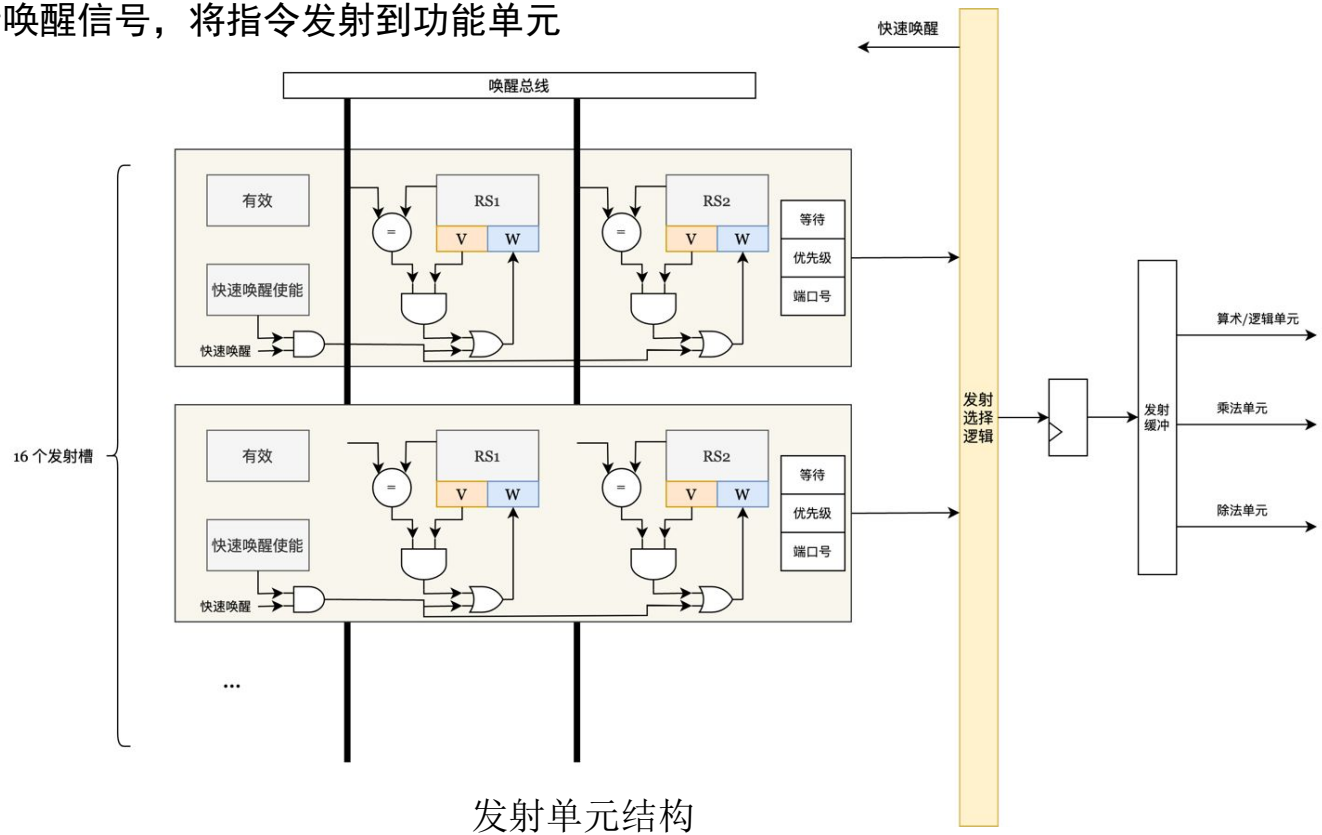
硬件系统：乱序发射单元 OoO-IQ

对指令进行数据等待，监听唤醒信号，将指令发射到功能单元

```
mul x10, x10, x11
add x10, x10, x12
```



指令流 -> 数据流转换示例



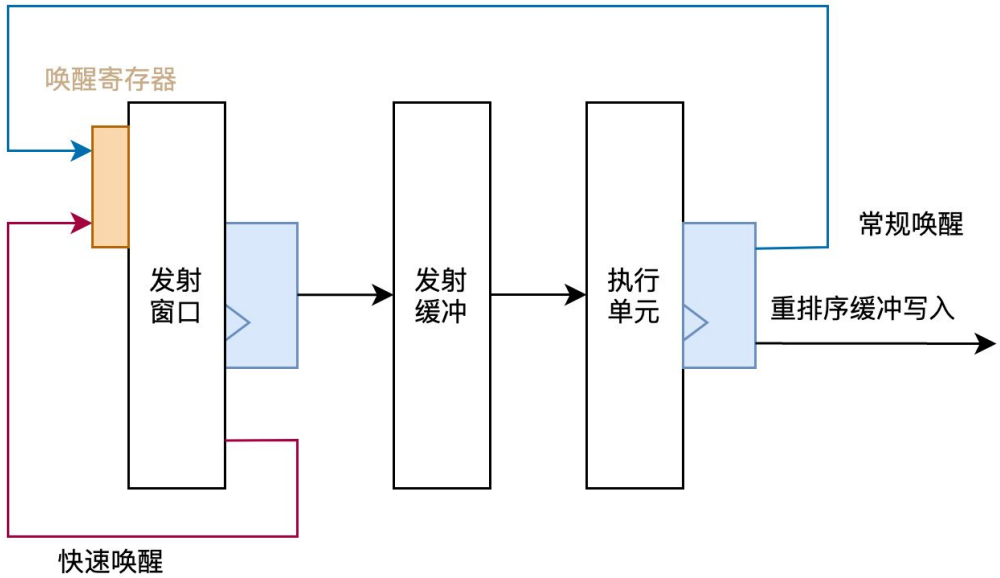
发射单元结构

硬件系统：数据唤醒通道 Wake

向 OoO-IQ 通知数据就绪状态

常规唤醒：
在预提交阶段触发。
所有指令都可由常规唤醒机制唤醒，且所有指令的完成都会触发常规唤醒机制。
延迟：3周期

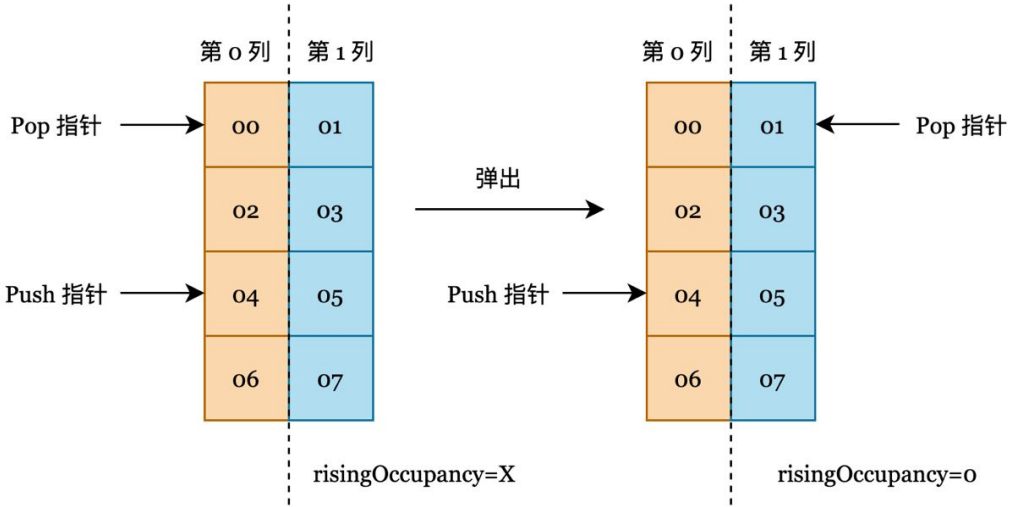
快速唤醒：
由数据流发射单元直接触发。
ALU 指令发射时，立即唤醒发射窗口中所有其他 ALU 指令对该指令目的寄存器的依赖，在下一周期即可发射。
延迟：1周期



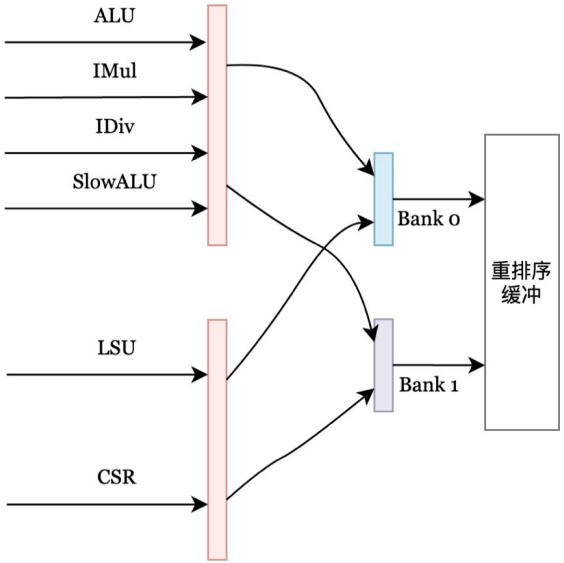
指令类型	延迟（周期数）
ALU（到 ALU）	1
ALU（到其他单元）	4
乘法单元	5
除法单元	36/68
复杂运算单元	≥5

硬件系统：重排序缓冲 ROB

将乱序完成的指令重排序为指令流顺序



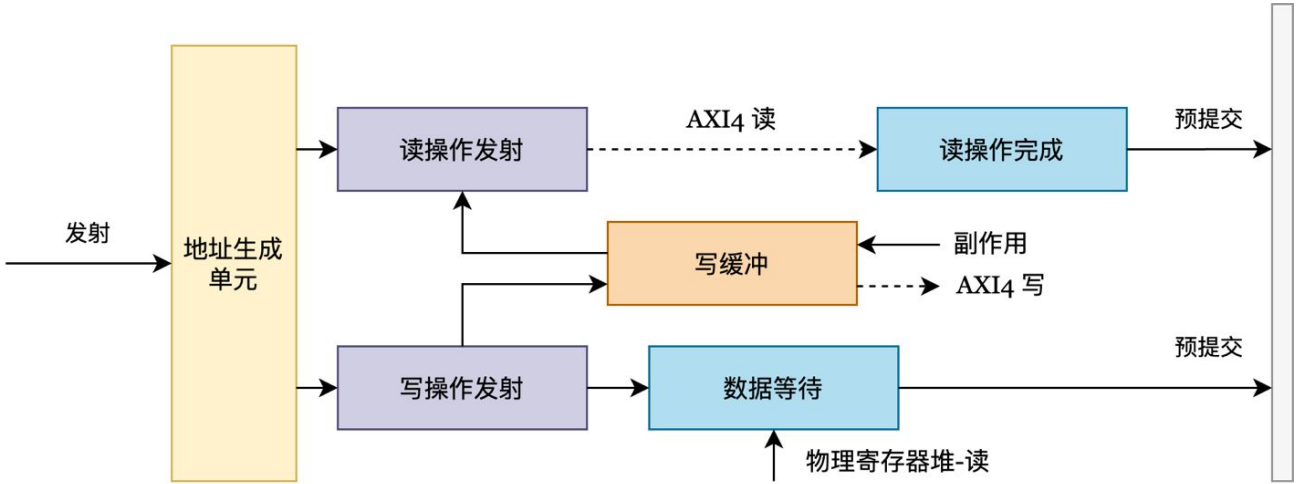
ROB 结构与逻辑



预提交仲裁结构

硬件系统：访存子系统 LSU

执行指令的内存访问操作，并保证副作用顺序



默认内存序保证：

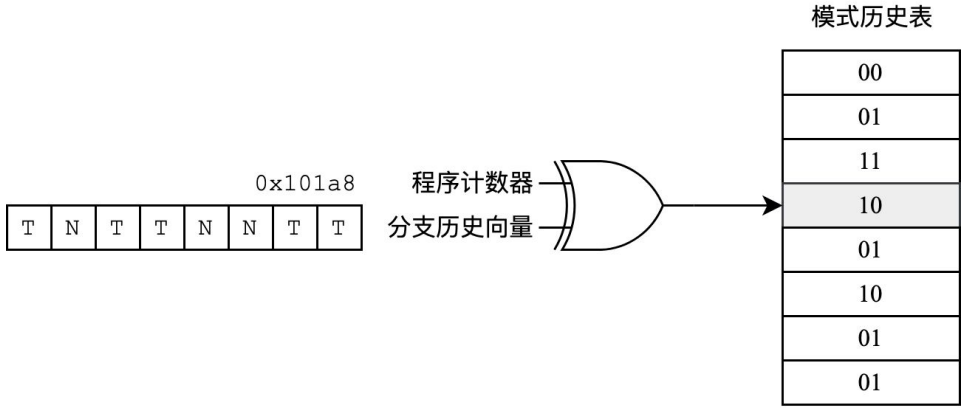
全序 Store (TSO) + 对于非 I/O 设备的推测读

内存屏障：

1. 写-读写屏障 (fence w, rw)
2. 顺序化屏障 (SERIALIZE)

硬件系统：分支预测

低延迟预测器：BTB
高准确率预测器：GShare



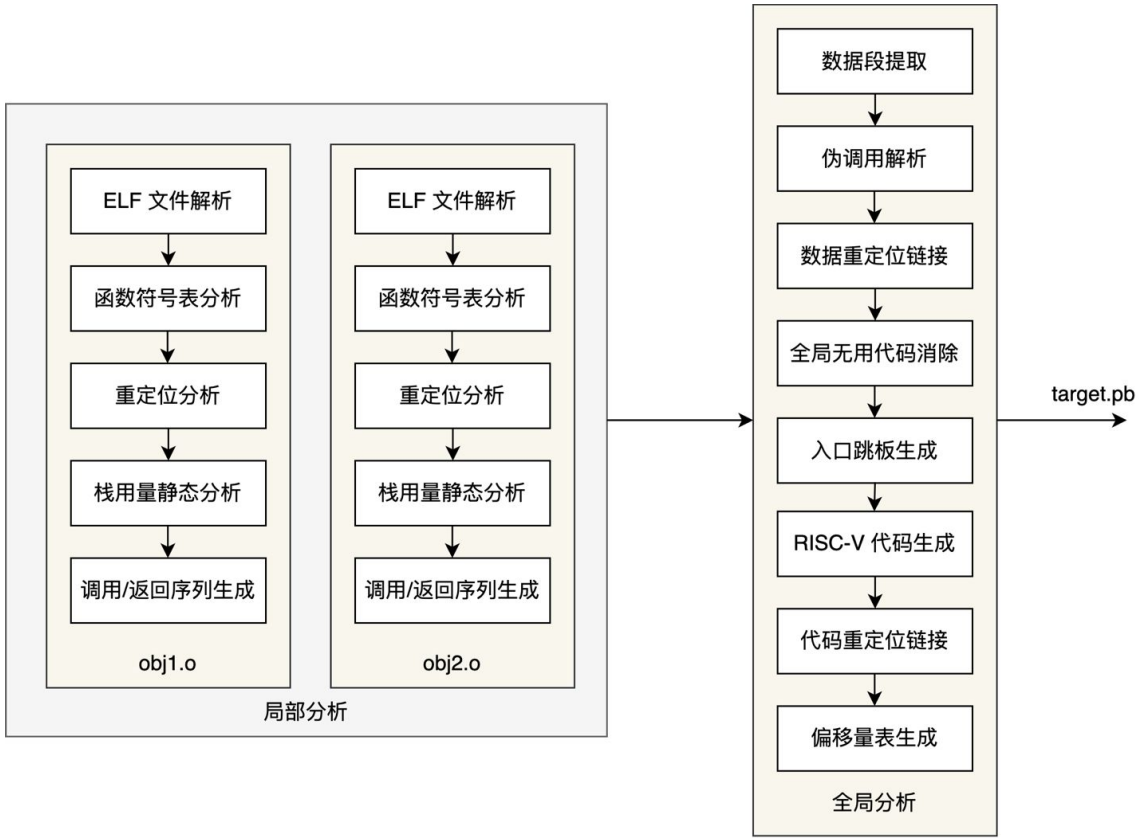
在 CoreMark 测试上的准确率为 89.4%：

```
Iterations: 2000
#brmiss: 19876082
#brhit: 167532846
```

软件系统：二进制翻译

端到端流程：

输入： 一组 eBPF ELF 对象文件
输出： 可加载的 Protobuf 镜像



软件系统：Linux设备驱动程序

```
wbpf0@43c00000 {
  #address-cells = <1>;
  #size-cells = <1>;
  compatible = "bluelogic,wbpf1";
  clocks = <&clkc 15>;
    reg = <0x43c00000 0x10000 0x7aa00000
0x20000>;
  interrupt-parent = <&intc>;
  interrupts = <0 29 4>;
};
```

设备树（Device Tree）结点定义

(1) 获取设备中断请求 (IRQ) 资源。

(2) 获取设备总线地址资源。

(3) 分配等待队列 (waitqueue).

(4) 初始化设备时钟，根据设备硬件版本设置对应的时钟频率。

(5) 分配直接内存访问（DMA）缓冲区，向内核请求DMA通道。

(6) 读取设备信息。

(7) 注册IRQ处理回调函数。

(8) 创建 /dev下的字符设备，向用户态提供接口。

设备初始化流程

系统实现：硬件设计与验证

设计语言：SpinalHDL

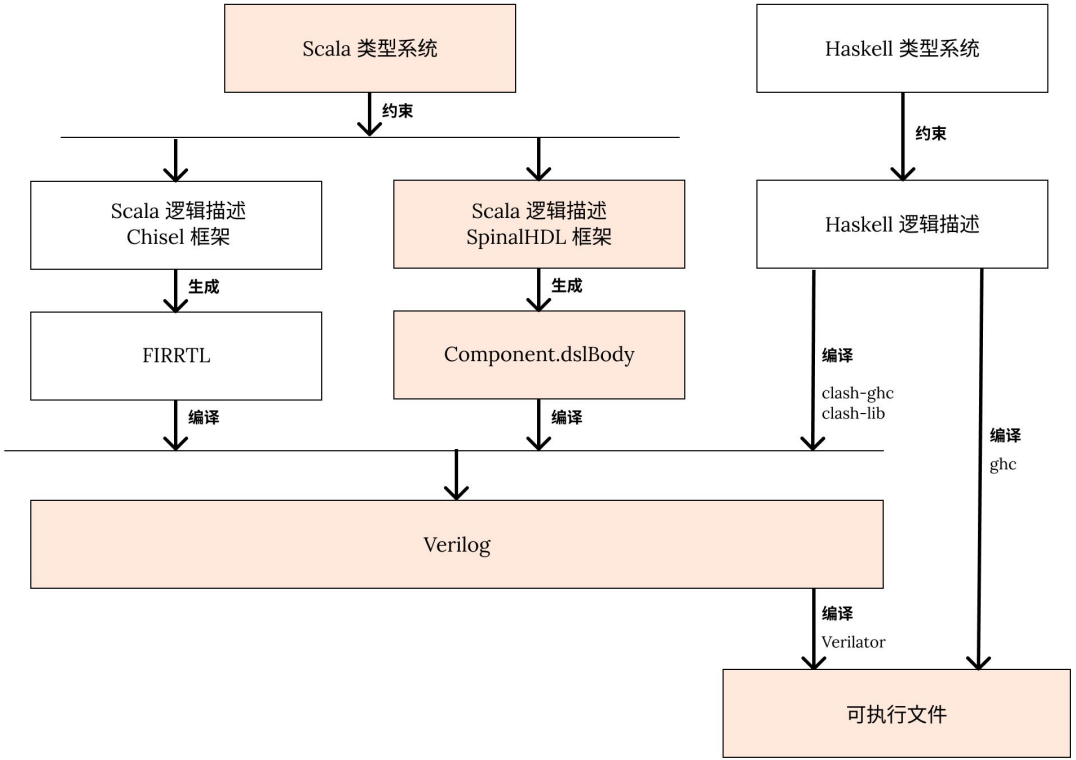
验证框架：Cocotb + Verilator

模块	LUT	FF	BRAM	DSP	最高频率
顶层设计（总用量）	38246	35861	93.5	16	93.9MHz
eBPF 硬件加速器子模块	14977	7860	49.0	16	-

系统实现：硬件设计与验证 – 敏捷硬件设计方法

SpinalHDL：基于 Scala 的浅嵌入型数字硬件描述语言

充分利用 Scala 强大的逻辑表达力与高阶类型系统提供的正确性约束，提高设计效率



系统实现：硬件设计与验证 - CoreMark 测试

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 1405033618
Total time (secs): 18
Iterations/Sec   : 111
Iterations       : 2000
Compiler version : GCC9.3.0
Compiler flags   : -O3 -march=rv64im -mabi=lp64 -mcmodel=medany -ffreestanding -nostdlib
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x4983
Correct operation validated. See README.md for run and reporting rules.
```

1.42 CoreMark/MHz（直接访问 DDR，无数据缓存）

U-Boot + CoreMark 完整输出

```
U-Boot 2022.04-00001-gb7dd17b8e0-dirty (May 01 2022 - 07:05:39 +0000)

CPU:   rv64im
Model: bluelogic,magicore-dev
DRAM:  64 MiB
Board init
Core:  3 devices, 3 uclasses, devicetree: separate
Loading Environment from nowhere... OK
In:    serial@ff010000
Out:   serial@ff010000
Err:   serial@ff010000
Net:   EMACLITE: 40e00000, phyaddr 7, 1/1
eth0: ethernet@40e00000
=> env set autostart yes; bootelf 0x10000000
## Starting application at 0x11000000 ...
CoreMark @ MagiCore RV64
Testing DCache. size = 1000000
D$ enabled     write: 5000034
D$ disabled    write: 5000915
D$ enabled     read:  3000077
D$ disabled    read:  3000099
Enabled data cache.
#cycle: 893140769
#instret: 107021456
#brmiss: 2857112
#brhit: 23420979
icache flush took 292 cycles.
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 1405033618
Total time (secs): 18
Iterations/Sec   : 111
Iterations       : 2000
Compiler version : GCC9.3.0
Compiler flags   : -O3 -march=rv64im -mabi=lp64 -mcmodel=medany -ffreestanding -nostdlib -fno-common -funroll-loops -finline-functions -finline-limit=1000 -fno-if-conversion2 -fselective-scheduling -fno-d
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x4983
Correct operation validated. See README.md for run and reporting rules.
#cycle: 3082066598
#instret: 1213357391
#brmiss: 22733194
#brhit: 190953825

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.8 | VT102 | Online 0:0 | ttyUL0
[0] 0:minicom* 1:ash-                                                    "zynq-dev" 06:49 01-Jan-70
```

系统实现：软件开发

采用 Rust 与 C 语言混合实现

Linux内核驱动程序：C
二进制翻译软件 + 硬件 FSBL：Rust

```
[   20.846128] wbpf: loading out-of-tree module taints kernel.
[   20.852669] wbpf: lvl shifter: f -> f
[   20.856701] wbpf_driver 43c00000.wbpf0: clk rate: 99999999 -> 124999999
[   20.863972] wbpf_driver 43c00000.wbpf0: hardware revision: 1.1, number of processing elements: 2, instruction buffer size: 2048 words
[   20.876083] wbpf_driver 43c00000.wbpf0: reset all processing elements in 1772 ns
[   20.883976] wbpf_driver 43c00000.wbpf0: cleared 131072 bytes of data memory in 411587 ns
[   20.892534] wbpf_driver 43c00000.wbpf0: device registered. irq 54, mmio 43c00000-43c0ffff, dm 7aa00000-7aa1ffff
[   20.903049] wbpf_driver 43c10000.wbpf1: clk rate: 124999999 -> 124999999
[   20.910378] wbpf_driver 43c10000.wbpf1: hardware revision: 1.1, number of processing elements: 2, instruction buffer size: 2048 words
[   20.922432] wbpf_driver 43c10000.wbpf1: reset all processing elements in 1683 ns
[   20.930327] wbpf_driver 43c10000.wbpf1: cleared 131072 bytes of data memory in 405584 ns
[   20.938820] wbpf_driver 43c10000.wbpf1: device registered. irq 55, mmio 43c10000-43c1ffff, dm 7aa40000-7aa5ffff
[   20.949087] wbpf: driver loaded
zynq-dev:~$
```

系统实现：自动化测试

本设计的自动化测试 (CI/CD) 基于 GitHub Actions实现。流程如下：

1. 软件自动化构建

本项CI流程自动构建Zynq平台的BOOT.BIN启动文件，分为u-boot、Linux内核、启动镜像构建三个阶段。

2. 硬件单元测试 (test)

本项CI流程基于Verilator运行硬件单元测试，验证硬件正确性。

3. 硬件设计RTL生成 (build-socs)

本项CI流程由以SpinalHDL描述的硬件设计构建Verilog RTL，供其他相关流程使用。

4. 系统测试 (system-test)

本项CI流程依赖流程（三），构建FSBL，并基于cocotb与Verilator运行系统测试.

5. 版本发布 (create-release)

本项CI流程依赖流程（三），提交RTL到GitHub Release版本发布系统，作为预发布版本。

总结与展望

本工作实现了以下目标：

1. 实现了一个基于**受限数据流架构、RISC-V指令集与二进制翻译**的eBPF硬件加速器，可作为独立的IP核集成进智能网卡等须在硬件上执行eBPF程序的设计中；
2. 实现了与上述硬件结合的**eBPF二进制翻译软件与Linux内核驱动程序**，提供从Linux用户态调用硬件运行eBPF程序的端到端功能。

本工作下一步的改进方向：

1. **性能优化：**超标量取指与译码；
2. **精确内存权限控制：**独立于内核的硬件执行引擎和复杂的内存拓扑为安全带来了新的挑战，为实现对不可信eBPF程序的的隔离，须实现精确的内存权限控制；
3. **PCIe支持：**兼容当前主流的服务器系统；
4. **与Linux eBPF子系统的集成：**面向网络场景提供完整的硬件eBPF解决方案。

开源地址

硬件设计：https://github.com/losfair/MagiCore

二进制翻译软件：https://github.com/datenlord/wbpf-userspace

Linux设备驱动程序：https://github.com/datenlord/wbpf-driver