CA_Assignment

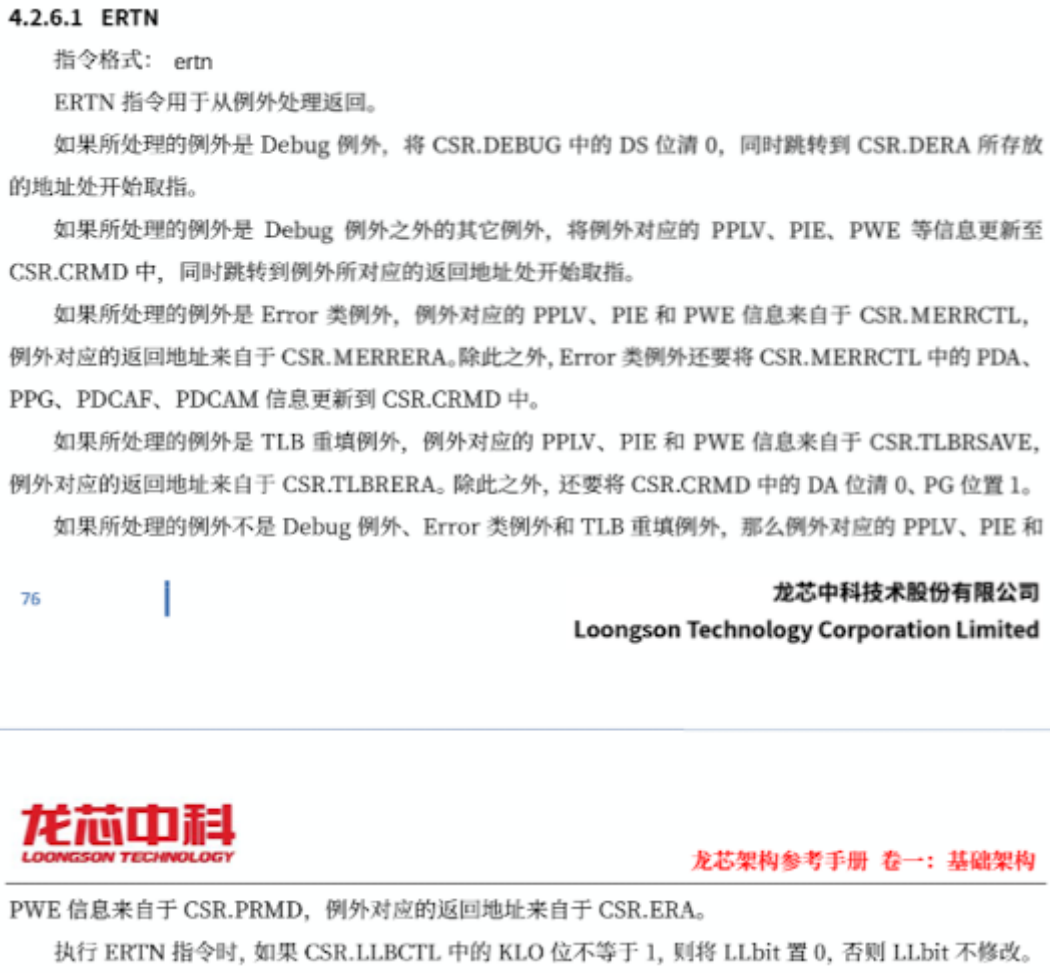
Assign	
tag	homework
姓名	周鹏宇
学号	2019K8009929039

1. 请说明LoongArch指令系统为什么要定义ERTN指令以用于异常处理的返回。

需要该指令**原子地**完成恢复权限等级、恢复中断使能状态、跳转至异常返回目标等多个操作。

X86中的IRET指令有类似的效果。

该指令的具体操作如下图所示：



4.2.6.1 ERTN

指令格式： ertn

ERTN 指令用于从例外处理返回。

如果所处理的例外是 Debug 例外，将 CSR.DEBUG 中的 DS 位清 0，同时跳转到 CSR.DERA 所存放的地址处开始取指。

如果所处理的例外是 Debug 例外之外的其它例外，将例外对应的 PPLV、PIE、PWE 等信息更新至 CSR.CRMD 中，同时跳转到例外所对应的返回地址处开始取指。

如果所处理的例外是 Error 类例外，例外对应的 PPLV、PIE 和 PWE 信息来自于 CSR.MERRCTL，例外对应的返回地址来自于 CSR.MERRERA。除此之外，Error 类例外还要将 CSR.MERRCTL 中的 PDA、PPG、PDCAF、PDCAM 信息更新到 CSR.CRMD 中。

如果所处理的例外是 TLB 重填例外，例外对应的 PPLV、PIE 和 PWE 信息来自于 CSR.TLBRSAVE，例外对应的返回地址来自于 CSR.TLBRERA。除此之外，还要将 CSR.CRMD 中的 DA 位清 0、PG 位置 1。

如果所处理的例外不是 Debug 例外、Error 类例外和 TLB 重填例外，那么例外对应的 PPLV、PIE 和

76 龙芯中科技术股份有限公司 Loongson Technology Corporation Limited

龙芯中科 LOONGSON TECHNOLOGY 龙芯架构参考手册 卷一：基础架构

PWE 信息来自于 CSR.PRMD，例外对应的返回地址来自于 CSR.ERA。

执行 ERTN 指令时，如果 CSR.LLBCTL 中的 KLO 位不等于 1，则将 LLbit 置 0，否则 LLbit 不修改。

2. 简述 LoongArch 与 X86 在异常处理过程中的区别

- 异常处理准备
  - LoognArch中，EPTR（被异常打断的指令的地址）存于CSR.ERA中
  - x86中，被异常打断的指令的地址（CS和EIP组合）被存放在栈中
- 确定异常来源
  - LoongArch将不同的异常进行编号，其异常处理程序入口地址采用“入口页号与页内偏移进行按位逻辑或”的计算方式
    - 入口页号通过CSR.EENTRY配置
    - 每个普通异常处理程序入口的页内偏移是其异常编号乘以一个可配置间隔(通过CSR.ECFG的VS域配置)
  - X86由硬件确定异常编号，并查询存放在内存中的中断描述符 表(IDT)，得到异常处理地址
- 保存执行状态和处理异常部分基本相似
- 恢复执行状态并返回
  - LoongArch 中，使用 ERET 指令从异常中返回；X86 中，使用 IRET 指令从异常中返回。两者功能相差不大。

---

3. 简述精确异常与非精确异常的区别。找一个使用非精确异常实例

- 精确异常（Precise Exception），要求被异常打断的指令前的指令都执行完毕，被异常打断前的指令及以后的所有指令都如同没执行。在精确异常中，EPTR 必须恰好指向被异常打断的指令。
- 非精确异常（Imprecise Exception），对被异常打断前后的指令执行情况没有要求。在非精确异常中，EPTR 不一定恰好指向被异常打断的指令，通常指向被异常打断的指令后的某条指令。
- 示例：PowerPC架构的浮点异常

> The PowerPC architecture supports four types of exceptions:
> ……
> -Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point
> exception modes, recoverable and nonrecoverable. The 604

> implements only the imprecise nonrecoverable mode. The imprecise, recoverable mode is treated as the precise mode ithe 604.
> *—Advance Information PowerPC 604 RISC Microprocessor Technical Summary*

---

4. 在一台 MIPS-Linux 机器（页大小为 4KB）上执行下列程序片段，会发生多少次异常？

```
void cycle(double *a) {
    int i;
    double b[65536];
    for (i = 0; i < 3 ; i++) {
        for(int j = 0; j < 65536; j++){
            a[j] = b[j];
        }
    }
}
//本处课本提供的程序并没有声明j，故自行添加
```

`double`是8个字节，则数组`b`的大小为`512KB`

故共$\frac{512}{4} = 128$页

则在$i = 0$的循环中，

- 访问b：由于 TLB 内没有 b 所在内存的相应表项，会造成 64 次 TLB 充填异常。同时，b还没有分配物理空间，还会造成 128 次 TLB 无效异常
- 访问a：由于 TLB 内没有 b 所在内存的相应表项，会造成 64 次 TLB 充填异常。同时，若a还没有分配物理空间，还会造成 128 次 TLB 无效异常

由于 TLB 最多同时记录 64 页，且采用 LRU 替换算法，故而在每次遍历数组时将覆盖此前存储的部分数组内存地址。故而在第二次遍历内层循环时，数组起始部分(前1/4)仍不在 TLB中，遍历到后续部分时同样由于起始部分覆盖了 TLB，所以遍历内层循环都会引发共 128 次 TLB 重填异常。

故共640次异常

---

5. 用 C 语言描述包含 TLB 的页式存储管理过程（包含 TLB 操作）

- 页表项内容
  • PPN:物理页号

- Flags:RPLV、NX、NR、PPN、W、P、G、MAT、PLV、D、V
- exts:软件扩展位，用于维护一些硬件没有实现的功能

```
typedef struct {
    ppn_t PPN;
    plv_t PLV;
    rplv_t RPLV;
    mat_t MAT;
    bool NX;
    bool NR;
    bool D;
    bool V;
} page_t;
```

```
typedef struct {
    bool E;
    asid_t ASID;
    bool G;
    ps_t PS;
    vppn_t VPPN;
    ppn_t PPN[2];
    plv_t PLV[2];
    rplv_t RPLV[2];
    mat_t MAT[2];
    bool NX[2];
    bool NR[2];
    bool D[2];
    bool V[2];
} TLB_entry_t;
```

```
add_t address_translation(asid_t ASID, vppn_t VPPN) {
    search_result_t search_result;

    if (!(search_result = TLB_probe(ASID, VPN))) {
        raise_exception(EXCEPTION_TLB_REFILL);
    }
    if (!search_result.V) {
        raise_exception(EXCEPTION_TLB_INVALID);
    }
    if (!search_result.D) {
        raise_exception(EXCEPTION_TLB_MODIFY);
    }
    return search_result.PFN;
}
```