

Operating Systems Engineering

Lecture 3: From Library to OS

Michael Engel (michael.engel@uni-bamberg.de)

Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung

<https://www.uni-bamberg.de/sysnap>

Licensed under CC BY-SA 4.0
unless noted otherwise



- Tasks of an operating system
- The OS as a library of common functions
- Use of RISC-V privilege modes in OS design
- Configuring RISC-V: configuration and status registers (CSRs)
- Switching privilege modes: system calls
- Short detour: Physical memory protection
- Implementing and calling system calls
- Discussion: system call overhead and alternatives

Why do we want to have an operating system?

- Share a computer among multiple programs
- Provide a more useful set of services than the hardware alone supports
 - manage and abstract the low-level hardware
 - for example, a word processor need not concern itself with which type of disk hardware is being used
- Share the hardware among multiple programs so that they run (or appear to run) at the same time
- Provide controlled ways for programs to interact, so that they can share data, work together and ensure consistency and privacy of their own data

- **How were the first operating systems developed?**
 - Early computers (1950s) were single-program machines
 - load a program → execute it → print results → load next...
- Some functionality had to be reimplemented by every program
 - e.g., disk or printer access
 - this was tedious and error-prone...
- ***Libraries of reusable common functions*** were developed
- Users did not want to reload the library (which took minutes!)
 - Protect library from the programs and keep it in memory
 - ***Privilege modes*** restricting access to the library's memory and functions for regular programs were introduced


```
                                .text
                                .align 1
                                .globl add
                                .type add, @function
                                0x100 add:
                                addw    a0, a0, a1
                                jr      ra
                                .globl foo
                                .type foo, @function
                                0x108 foo:
                                li      a0, #23
                                li      a1, #42
                                auipc   t0, #0
                                jalr    ra, pc, #0x100
                                jr      ra
```

Diagram illustrating function calls in RISC-V assembly:

- The `add` function is located at address `0x100`. It consists of an `addw` instruction followed by a `jr` instruction that jumps to the `ra` register.
- The `foo` function is located at address `0x108`. It consists of several instructions, including `li` for loading constants, `auipc` for loading the PC into a temporary register, and a `jalr` instruction that jumps to the `ra` register with an immediate value of `#0x100`.
- Arrows indicate the flow of control: from the `jalr` instruction in `foo` to the `add` function, and from the `jr` instruction in `add` back to the `foo` function.

"Universal" RISC-V
jump instruction:

JALR `rd, rs, immediate`

- Writes PC+4 to `rd` (return address)
- Sets PC = `rs + immediate` (12 bit, sign extended)
- Uses same immediates as arithmetic and loads

Function calls in RISC-V – *wait a minute!*

```
int add(int a, int b) {  
    return a + b;  
}  
  
0x100 add:  
    .text  
    .align 1  
    .globl add  
    .type add, @function  
    addw a0, a0, a1  
    jr ra ; footnote1
```

```
int foo(void) {  
    return add(23, 42);  
}  
  
0x108 foo:  
    .globl foo  
    .type foo, @function  
    li a0, #23  
    li a1, #42  
    auipc t0, #0  
    jalr ra, t0, #0x100  
    jr ra
```

Register ra is **overwritten** here →

This is no longer the correct return address!

¹ The **jr ra** instruction does not really exist, it's a **pseudo-op** expanded into **jalr x0, ra, 0**

Saving the return address

```
int foo(void) {  
    return add(23, 42);  
}
```

Reserve 8 bytes on stack and temporarily store "old" ra value

Now we can change ra to call the add function

Retrieve stored "old" ra value and fix up stack pointer

```
.globl foo  
.type foo, @function  
0x108 foo:  
    addi sp, sp, -8  
    sd   ra, 8(sp)  
    li   a0, #23  
    li   a1, #42  
    auipc t0, #0  
    jalr ra, t0, #0x100  
    ld   ra, 8(sp)  
    addi sp, sp, 8  
    jr   ra
```

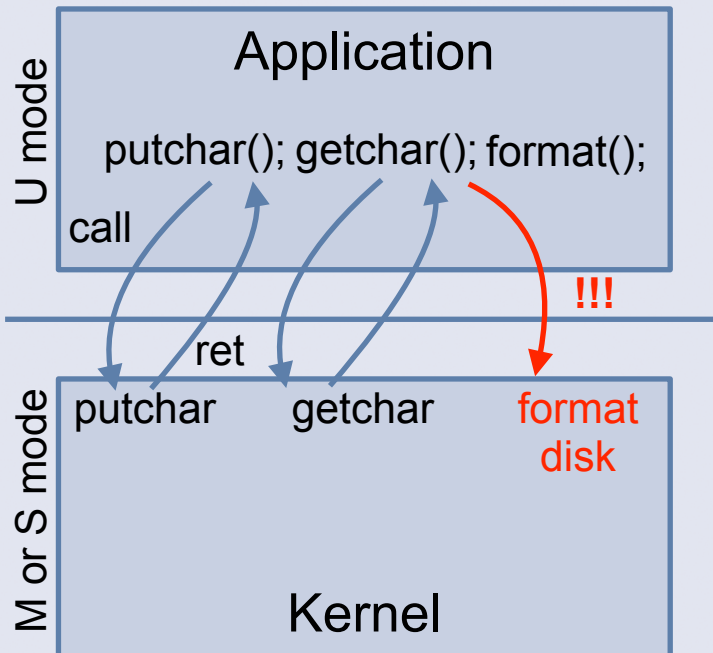
This is now the correct return address

Storing and retrieving the return address register ra is not required for **leaf functions**

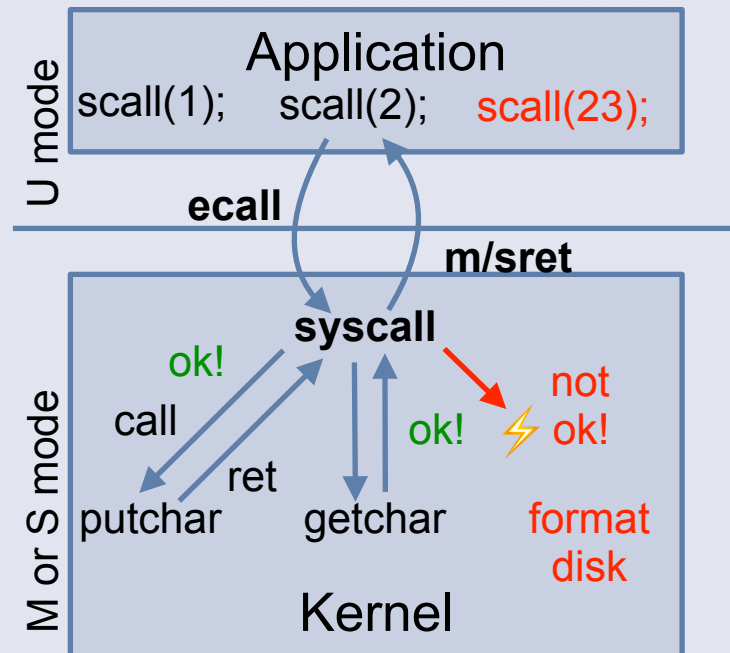
- functions which never call another function (like our "add")
- optimization especially for recursion in functional languages

- Ensure a **controlled** transition from user mode to kernel
 - User mode program may only call permitted functions
 - Kernel should be able to check parameters

Direct function calls to the kernel



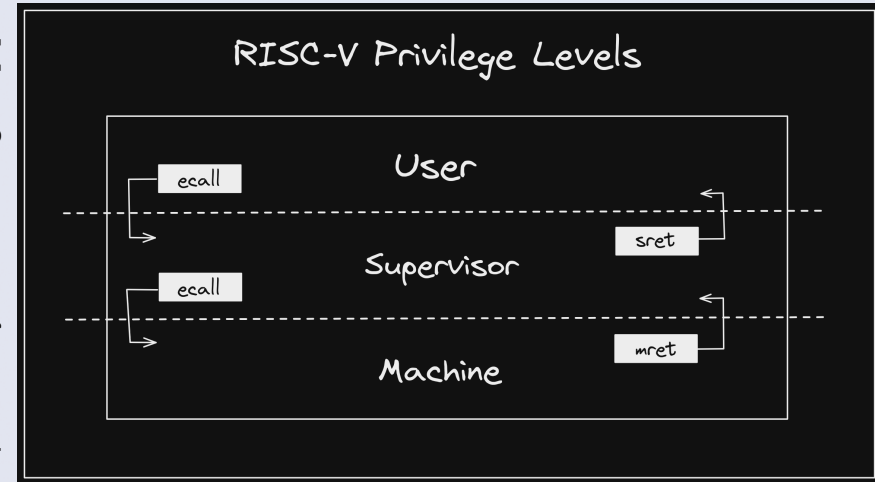
Using system calls



Use of RISC-V privilege modes in OS design

- **RISC-V supports three privilege levels**
 - **User** – least privileged, apps
 - **Supervisor** – for the OS
 - **Machine** – full HW access

picture by Daniel Mangum [3]



- **Switching** between modes can occur **explicitly** or **implicitly**
 - Explicit: initiated by software ("**ecall**" assembler instruction)
 - Implicit: by a hardware interrupt or as side effect of executing an instruction (e.g. illegal instruction or invalid address)
- Returning always requires a special instruction
 - **mret** (return from machine mode) or **sret** (return from supervisor mode)

- **Controlled transition from user mode to kernel:**
 - **ecall** instruction (takes no parameter)
 - switches processor to machine mode¹
 - stores previous processor state (M/S/U, interrupt enables...)
 - jumps to a central address ("trap vector") in the kernel
- **Controlled transition back to previous mode:**
 - **{m,s}ret** instructions (return from machine or supervisor mode)
 - **Why not use a simple jr ra instruction?**
 - The processor needs to restore the previous state in addition to jumping back to user mode
 - **Where to return to, if not to the address in ra?**
 - Special register (CSR) "exception PC" (m/sepc)
 - Automatically used by m/sret instruction

¹ Machine mode can delegate exceptions "down" to supervisor or user mode again – more on this later

- **Parameters to C functions are passed in registers**
 - Which parameter is assigned to which register is specified in the Application Binary Interface (ABI) of a platform ([1] for RISC-V)
- **Code of functions can overwrite registers**
 - Values of the "clobbered" registers must be preserved
 - These (can) hold values which are still needed in the calling function
 - Some registers have to be saved by the caller (calling function), some by the callee (function that is called)
- Usually, the C compiler takes care of inserting code to save and restore register values
 - Exceptions are different – **the C compiler does not know the calling context**
 - It can only know this for functions called directly in the C code, but not for an **ecall**

- **Function call:** uses a subroutine call instruction (**jalr** on RISC-V)
 - Call destination given in the target address
- **System call:** uses **ecall** *for all system calls*
 - **All syscalls jump** to our **exception handler!**
 - *We cannot allow user code to jump directly into kernel code!*
- How do we know which syscall to execute?
- **Syscall numbers** indicate requested syscall
 - Passed as additional parameter (e.g., in register a7 in xv6 for RISC-V)
- Other parameters to syscalls passed in additional registers a0, a1, ..., a6

```
#define SYS_fork      1
#define SYS_exit      2
#define SYS_wait      3
#define SYS_pipe      4
#define SYS_read       5
#define SYS_kill       6
#define SYS_exec       7
#define SYS_fstat      8
#define SYS_chdir      9
#define SYS_dup       10
#define SYS_getpid     11
#define SYS_sbrk       12
#define SYS_sleep      13
#define SYS_uptime     14
#define SYS_open       15
#define SYS_write      16
#define SYS_mknod      17
#define SYS_unlink     18
#define SYS_link       19
#define SYS_mkdir      20
#define SYS_close      21
```

syscall numbers in xv6

- **The ABI describes registers to be saved by caller/callee**
 - This works only if C functions are called in a regular way
 - ...and not as exception handler
- **We cannot simply use a C function as exception handler**
 - *all registers need to be in the same state as before the exception was called*
 - for **ecalls**, the only register that may have a different value is the **return value in register a0**

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

- **System calls provide functions commonly used by applications that require direct hardware access or need to be shared by different applications**
- For our example:
 - `syscall #1: int putstring(char *s);`
 - `syscall #2: int putchar(char c);`
 - `syscall #3: char getachar(void);`
- Of course, we can have more parameters in the future
 - However, the register convention (parameters in registers a0–a6) constrain us to a max. of 7 parameters (+ syscall number in a7)
 - ...this is usually not a problem
- Why is there no syscall #0?
 - Possible, but having no valid syscall when a zero in a7 might catch some bugs in the user code program (e.g. jumping into a data region which accidentally decodes to the "ecall" instruction)

- **How can we map a function call to a syscall?**
- **Write C stub functions that can be called directly**
 - We can write a C function for every syscall
 - This is what the C library (libc) also does
- Stub functions have to put parameters in the right registers
 - a7 = syscall number, a0–a6 = function parameters
 - *all values are register values (64 bit int) → cast as required!*
 - ...and then perform an **ecall** instruction!
- After return from ecall, a0 contains a possible return value
 - ...or an error code (a0 < 0)

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

- **Save/restore all registers before/after calling the C except. handler**
 - We have to do this in assembler again: file **ex.S**

```
.globl ex
.globl exception
.align 4
ex:
    // make room to save registers.
    addi sp, sp, -256

    // save the registers.
    sd ra, 0(sp)
    sd sp, 8(sp)
    sd gp, 16(sp)
    sd tp, 24(sp)
    sd t0, 32(sp)
    sd t1, 40(sp)
    sd t2, 48(sp)
    sd s0, 56(sp)
    sd s1, 64(sp)
    sd a0, 72(sp)
    ...
    // a2 to a6
    sd a7, 128(sp)
    sd s2, 136(sp)
    ...
    // s3 to s11
    sd s11, 208(sp)
    sd t3, 216(sp)
    sd t4, 224(sp)
    sd t5, 232(sp)
    sd t6, 240(sp)

    // call the C trap handler in kernel.c
    call exception // see slide 30!
```

```
    // restore registers.
    ld ra, 0(sp)
    ld sp, 8(sp)
    ld gp, 16(sp)
    ld tp, 24(sp)
    ld t0, 32(sp)
    ld t1, 40(sp)
    ld t2, 48(sp)
    ld s0, 56(sp)
    ld s1, 64(sp)
    // ld a0, 72(sp) // nope, it's the return value...
    ld a1, 80(sp)
    ...
    // a2 to a6
    ld a7, 128(sp)
    ld s2, 136(sp)
    ...
    // s3 to s11
    ld s11, 208(sp)
    ld t3, 216(sp)
    ld t4, 224(sp)
    ld t5, 232(sp)
    ld t6, 240(sp)

    addi sp, sp, 256

    // return to whatever we were doing in the kernel.
    mret
```


- **How to determine where to return to?**
 - Some architectures store information on the interrupted program's stack...
- **RISC-V has special *configuration and status registers* (CSRs) for this: `mepc` and `sepc`**
 - **machine/supervisor exception program counter**
 - contains the address of the instruction that was executed when the mode switch occurred
 - e.g. the address of the **ecall** or the faulting instruction
 - ...or the address of the instruction executed when an interrupt occurred (we will discuss interrupts later)
- The **mret** and **sret** instructions return to the address stored in the `mepc` or `sepc` CSR, respectively

Configuring RISC-V: configuration and status registers (CSRs)

- **How can we determine which mode to return to?**
 - mepc/sepc only store the previous address, not the mode
 - e.g. an exception to M mode could come from S or U mode
- **mstatus** CSR provides a privilege mode stack (see [2], sec. 3.1.6)
 - each privilege mode x that can respond to interrupts has a two-level stack of interrupt-enable bits and privilege modes
 - xPIE holds the value of the interrupt-enable bit active prior to the trap, and xPP holds the previous privilege mode
 - The xPP fields can only hold privilege modes up to x, so MPP is two bits wide (M, S or U) and SPP is one bit wide (S or U)

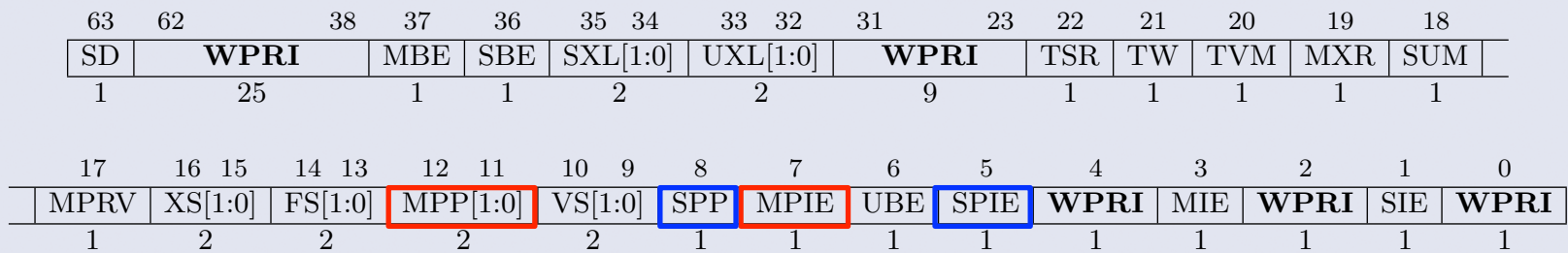


Figure 3.7: Machine-mode status register (mstatus) for RV64.

Configuring RISC-V: configuration and status registers (CSRs)

- **mstatus** CSR provides a privilege mode stack (see [2], sec. 3.1.6)
 - An **mret** or **sret** instruction is used to return from a trap in M-mode or S-mode respectively
 - When executing an **xret** instruction, supposing xPP holds the value y, xIE is set to xPIE; the privilege mode is changed to y; xPIE is set to 1; and xPP is set to the least-privileged supported mode (U if U-mode is implemented, else M)
- **If you think this is complicated... yes, it is (a bit)**
 - Interrupt enables are also important – don't worry, we provide the correct setup code. Try to figure out what it does!

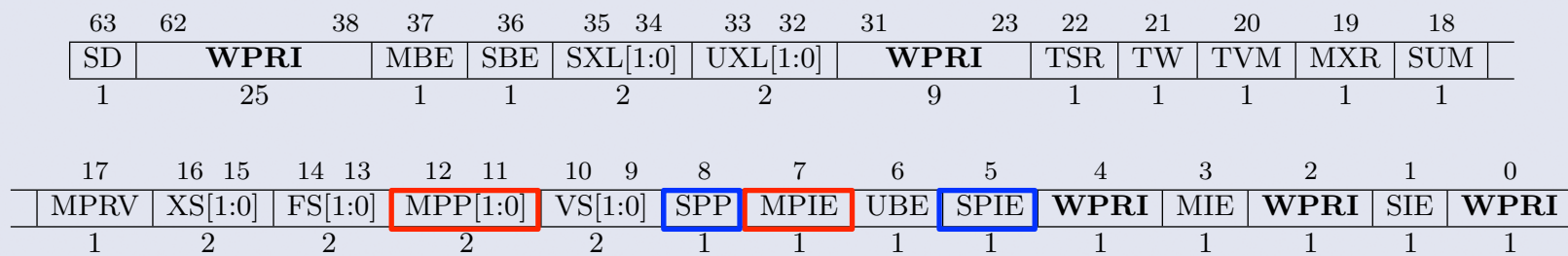


Figure 3.7: Machine-mode status register (**mstatus**) for RV64.

- **Inline assembler is a convenient way to directly include assembler instructions in your C code** (gcc extension)
 - code becomes ***non-portable*** – assembler instructions are specific to a given processor architecture
 - *alternative*: call an assembler function for every required assembler instruction → overhead, hard to read
- Interface between data in C (variables) and assembler (register names) has to be specified: *output, input and clobbered values*

```
asm ( assembler template
      : output operands          /* optional */
      : input operands           /* optional */
      : list of clobbered registers /* optional */
    );
```

- *"clobbered": register values changed by the assembler code*


```
asm ( assembler template
      : output operands          /* optional */
      : input operands          /* optional */
      : list of clobbered registers /* optional */
    );
```

- Operand specifiers are format string (a bit similar to printf)
- Example: implement an "add" function ($z=x+y$) in RISC-V assembler:

```
uint64_t x=23, y=43, z;
asm ( "add %0, %1, %2"
      : "=r"(z)           // write register result in %0 to z
      : "r"(x), "r"(y)    // put x and y in registers %1,%2
      :                   // no clobbered registers here
    );
```

- You can add the "volatile" keyword after "asm" to keep the compiler from optimizing your assembler code (e.g. move it out of a loop)

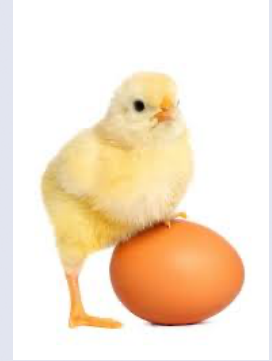
- **Not possible from C!**
 - CSRs are not memory mapped
- **Assembler instructions required** to access CSRs [2]:
 - **csrr**: read a CSR – copy value in the given CSR into CPU register
 - **csrw**: write a CSR – copy value in CPU register into given CSR
- Up to 4096 CSRs per execution mode (M,S,U) are defined
 - *Symbolic names* make their use from assembler a bit easier
- Example: read the current processor core number ("hartid") into a0

```
csrr a0, mhartid
```
- Use from C via **inline assembler**:

```
uint64 x; asm volatile("csrr %0, mhart" : "=r" (x) );
```
- "Stupid" – number of CSR encoded directly in instruction, cannot be passed as register value → cannot write generic csrr/csrw C wrapper

Switching privilege modes: For the very first time...

- **Problem: RISC-V CPU always starts in M mode**
 - How can we start our user mode code
 - Get to user mode: use the **mret** instruction!
- **But where to return to?**
 - There was no previous syscall from U \rightarrow M (or S) mode
 - thus there is no valid return address in **mepc** register
 - Classical chicken and egg problem...
- The first switch to user mode is manually
- *Fake as good as you can*: set required **mepc** and **mstatus** values
 - **mepc**: first instruction in user mode code (e.g. main)
 - **mstatus**: set previous mode bits to U, enable U interrupts



Setup code for our OS

```
void setup(void) {
    // set M Previous Privilege mode to User so mret returns to user mode.
    unsigned long x = r_mstatus();
    x &= ~MSTATUS_MPP_MASK;
    x |= MSTATUS_MPP_U;
    w_mstatus(x);

    // enable machine-mode interrupts.
    w_mstatus(r_mstatus() | MSTATUS_MIE);

    // enable software interrupts (ecall) in M mode.
    w_mie(r_mie() | MIE_MSIE);

    // set the machine-mode trap handler to jump to function "ex" when a trap occurs.
    w_mtvec((uint64)ex);

    // disable paging for now.
    w_satp(0);

    // configure Physical Memory Protection to give user mode access to all of physical memory.
    w_pmpaddr0(0x3fffffffffffffffll);
    w_pmpcfg0(0xf);

    // set M Exception Program Counter to main, for mret, requires gcc -mmodel=medany
    w_mepc((uint64)main);

    // switch to user mode (configured in mstatus) and jump to address in mepc CSR -> main().
    asm volatile("mret");
}
```


Quick detour: setting/resetting single bits

```
#define MSTATUS_MPP_MASK (3L << 11) // previous mode
#define MSTATUS_MPP_M (3L << 11)
#define MSTATUS_MPP_S (1L << 11)
#define MSTATUS_MPP_U (0L << 11)
```

Defining single bits:

1L << 11 means "1" left shifted 11 times

0000_0000_0000_0001 << 11
= 0000_1000_0000_0000



```
void setup(void) {
    unsigned long x = r_mstatus();
    x &= ~MSTATUS_MPP_MASK;
    x |= MSTATUS_MPP_U;
    w_mstatus(x);
    ...
}
```

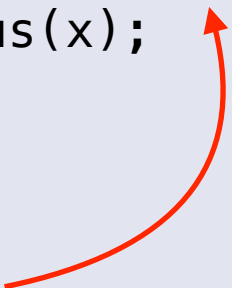
x "and not" value:

invert bits in MSTATUS_MPP_MASK

..._0001_1000_0000_0000
=> ..._1110_0111_1111_1111

and uses this to clear bits 11 and 12 in x

x "or" value:



set bits in x to "1" which are "1" in MSTATUS_MPP_U

- **Memory protection is important**
 - and necessary to run any code in user mode (since ...)
- Two approaches to memory protection
 - ***Virtual memory (paging)*** – mapping + permissions [2] ch. 4.3 ff.
 - ***Segmentation (PMP)*** – segments + permissions [2] ch. 3.7

Details on paging later in the course

→ slides+video on paging and segmentation in NTNU OS course [7]

- Paging and PMP can be combined
 - ***PMP is always active*** in S and U mode (off in M), by default all memory access is prohibited (*enforced in qemu 7 and in real hardware*)
 - ***Paging has to be explicitly enabled*** by configuring the satp CSR
 - off by default (and in M mode) → 1:1 logical ↔ physical translation

- | | | | |
|----------|----------------------|----|---|
| 63 | 54 | 53 | 0 |
| 0 (WARL) | address[55:2] (WARL) | | |
| 10 | 54 | | |

[illegible]

27

- PMP configuration register **pmcfgx** encode access permissions for segment x: eight bits per segment
 - Permissions for the segment: **R**ead, **W**rite, **eX**ecute
 - **A**: address matching
 - **L**: indicates that the PMP entry is locked, i.e., writes to the configuration register and associated address registers are ignored
 - *Locked PMP entries remain locked until reset*

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥ 8 bytes

Table 3.10: Encoding of A field in PMP configuration registers.

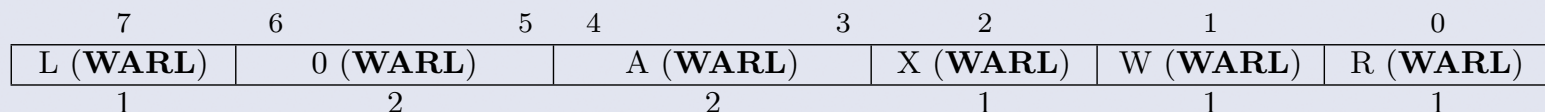


Figure 3.35: PMP configuration register format.

- **Address matching example**
 - **"TOR": pmpaddrx indicates end of the address range to be protected**
 - Starts at 0 for segment 0, at end of segment 0 for segment 1 etc.
- For now, we use:
 - `w_pmpaddr0(0x3F_FFFF_FFFF_FFFFull); // 54 "1" bits`
 - `w_pmpcfg0(0xF); // TOR (top of range) with R+W+X permission`
- Enables access to all physical memory (addresses $0 \dots 2^{56}-1$)
- Other modes configurable (NA4, NAPOT)
 - pmpaddr gives start address
 - See [2] Section 3.7

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥ 8 bytes

Table 3.10: Encoding of A field in PMP configuration registers.

pmpaddr	pmpcfg.A	Match type and size
yyyy...yyyy	NA4	4-byte NAPOT range
yyyy...yyy0	NAPOT	8-byte NAPOT range
yyyy...yy01	NAPOT	16-byte NAPOT range
yyyy...y011	NAPOT	32-byte NAPOT range
...
yy01...1111	NAPOT	2^{XLEN} -byte NAPOT range
y011...1111	NAPOT	2^{XLEN+1} -byte NAPOT range
0111...1111	NAPOT	2^{XLEN+2} -byte NAPOT range
1111...1111	NAPOT	2^{XLEN+3} -byte NAPOT range

Table 3.11: NAPOT range encoding in PMP address and configuration registers.

```
void exception(void) {
    uint64 syscall_number;
    uint64 parameter;
    uint64 retval = 0;

    // copy registers a7 and a0 to variables nr and param
    ...

    // decode syscall number
    ...

    // adjust return value - return to instr. _after_ ecall! (at address mepc+4)
    uint64 pc = r_mepc(); // read value of exception program counter
    w_mepc(pc+4);          // adjust mepc to skip the causing "ecall" instruction

    // pass the return value back in a0
    asm volatile("add a0, %0, x0" : : "r" (retval));

    // this function returns to ex.S
}
```

Adjusting mepc is critical!

- without this, we would jump right back to the ecall instruction
 - ...and immediately end up back here!
- +4 because a regular RISC-V instruction takes four bytes

- **Separate kernel and user code**
 - Add an exception handler for all syscalls
- **Files:**
 - hardware.h – defines for I/O addresses and structs (UART)
 - riscv.h – all support assembler functions to access CSRs
 - boot.S – assembler startup code, jumps to setup function
 - ex.S – assembler code for saving and restoring registers, called when an exception (ecall) occurs
 - setup.c – setup function, all code only required once to set up the processor and jump to user mode for the first time
 - kernel.c – all kernel code:
 - exception handler and code to implement system calls
 - hello.c – your user mode code containing the "main" function and the system call stub function(s)

- **We built a bit of a 1960s-style OS...**
 - or maybe a *very primitive* version of 1980's MS-DOS? 🤔
 - single-tasking, polling only (no interrupts so far – DOS used them)
- System calls are **defined interfaces** from user to kernel mode
 - Applications can be forced to use the defined system calls only
 - Kernel is able to check parameters for validity
- Next lecture: **protecting the OS**
 - Use PMP for basic protection from user mode accesses
 - separate **address spaces** for OS and user mode
 - Executables as separate binaries
 - Executable formats and loading executables

1. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, Document Version 20191213, chapter 25
2. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, Document Version 20211203
3. Daniel Mangum, *RISC-V Bytes*,
<https://danielmangum.com/categories/risc-v-bytes/>
4. Stephen Marz, *RISC-V OS using Rust*,
<https://osblog.stephenmarz.com/index.html>
5. Russ Cox, Frans Kaashoek and Robert Morris,
xv6: a simple, Unix-like teaching operating system,
<https://github.com/mit-pdos/xv6-riscv-book>
6. Using the GNU Compiler Collection, section 6.47.2
Extended Asm - Assembler Instructions with C Expression Operands
<https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>
7. TDT4186 Operating Systems 2022 Lecture 9: Memory management (slide 24 ff.)
https://folk.ntnu.no/michaeng/tdt4186_22/slides/os09.pdf