Kernel and Privilege Levels

Review: kernel ≈ 3 handlers

```
void kernel() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 3) { syscall(); }
        if (id == 7) { yield(); }
    } else {
        fault();
    }
}
```

First step of P2, system call

```
void kernel() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 3) { syscall(); }
        if (id == 7) { yield(); }
    } else {
        fault();
        // Exceptions 8-11 are triggered by ecall
        if (id >= 8 && id <= 11) { syscall(); }
    }
}
```

Understanding `ecall`

- Review RISC-V function call
- Understand interrupt handler call
- Understand the RISC-V instruction `ecall`

Say `main()` calls `printf()`

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡)
 ➡  Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#1

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡)
 ➡  Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

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

RISC-V Calling Convention: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

Function call step#2

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of )
    Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

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

Modified by the `call` instruction

RISC-V Calling Convention: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

Function call step#3

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of )
    Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

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

RISC-V Calling Convention: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

Function call step#4

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➔)
 ➔  Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#5

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of )
    Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#6

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of )
    Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

In particular, ra is restored here

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of )
    Restore the ra register
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Understanding `ecall`

- Review RISC-V function call
- Understand interrupt handler call
- Understand the RISC-V instruction `ecall`

Problem #1

If an interrupt happens during `main()`,
the CPU will call `handler()`, but the compiler
can't predict it and store registers on `main()` stack.

Address problem #1

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call handler (set ra to the address of )
    Restore caller-saved registers
    . . .

<handler>:
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    Return to main() with ra
```

If main() calls handler() directly

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call handler (set ra to the address of )
    Restore caller-saved registers
    . . .

<handler>:
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    Return to main() with ra
```

But handler() is inserted by CPU

```
<main>:
    . . .
    Store caller-saved registers on the stack
    CPU inserts a call to handler
        (set ra to the address of )
    Restore caller-saved registers
    . . . ra will be a different value at this point!

<handler>:
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    Return to main() with ra
```

Problem #2
How to keep the value of `ra`?

Introducing the mepc CSR

```
<main>:
    . . .
    Store caller-saved registers on the stack
    CPU inserts a call to handler
        (set mepc to the address of ➔)
    Restore caller-saved registers
➔   . . .  ra is still the same value at this point!

<handler>:
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    Return to main() with mepc
```

Understanding `ecall`

- Review RISC-V function call
- Understand interrupt handler call
- Understand the RISC-V instruction `ecall`

Call handler() by triggering an exception

```
<some user function>:
    . . .
    ecall // Triggers exception 8 or 11
    . . . // CPU inserts a call to handler

<handler>:
    . . .
    // handle the system call
    // read  value   of mepc (the value of )
    // write value+4 to mepc (the next instruction)
    mret // similar to ret but uses mepc instead of ra
```

Summary of `ecall`

- Review the RISC-V calling convention
- Understand interrupt handler call
  - Solve problem #1: save all registers on the `handler` stack
  - Solve problem #2: use `mepc/mret` instead of `ra/ret`
- Understand the `ecall` instruction
  - triggering exception 8 or 11 for system call

Adding privilege levels

8.2.1 Interrupt Entry and Exit

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into `mcause.MPIE`, and then `mstatus.MIE` is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in `mstatus.MPP`.
- The current `pc` is copied into the `mepc` register, and then `pc` is set to the value specified by `mtvec` as defined by the `mtvec.MODE` described in Table 19.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. Interrupts can be re-enabled by explicitly setting `mstatus.MIE` or by executing an `MRET` instruction to exit the handler. When an MRET instruction is executed, the following occurs:

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mcause.MPIE`.
- The `pc` is set to the value of `mepc`.

SiFive FE310 CPU manual: https://github.com/yhzhang0128/egos-2000/blob/main/references/sifive-fe310-v19p04.pdf

When an interrupt occurs

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into ~~`mcause.MPIE`~~ `mstatus.MPIE`, and then `mstatus.MIE` is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in `mstatus.MPP` (Machine Previous Privilege (MPP))
- The current `pc` is copied into the `mepc` register, and then `pc` is set to the value specified by `mtvec` as defined by the `mtvec.MODE` described in Table 19.

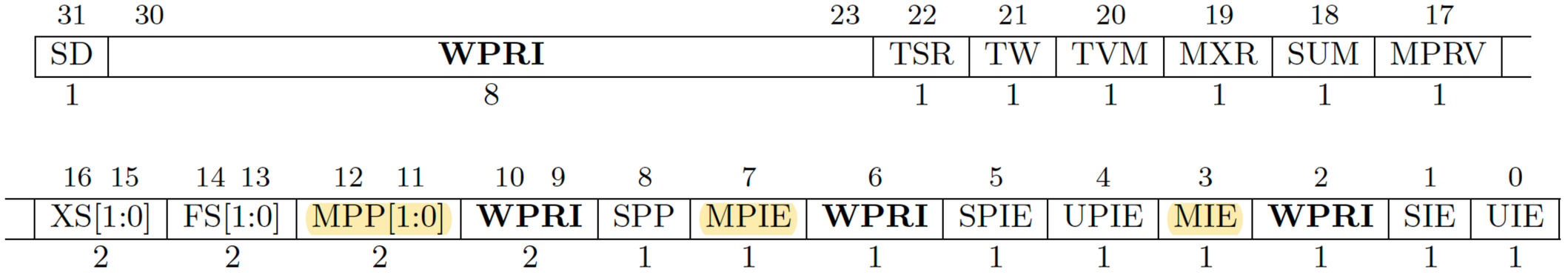

Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

When invoking `mret`

tion to exit the handler. When an MRET instruction is executed, the following occurs:

Machine Previous Privilege (MPP)

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mcause.MPIE`.
- The pc is set to the value of `mepc`.

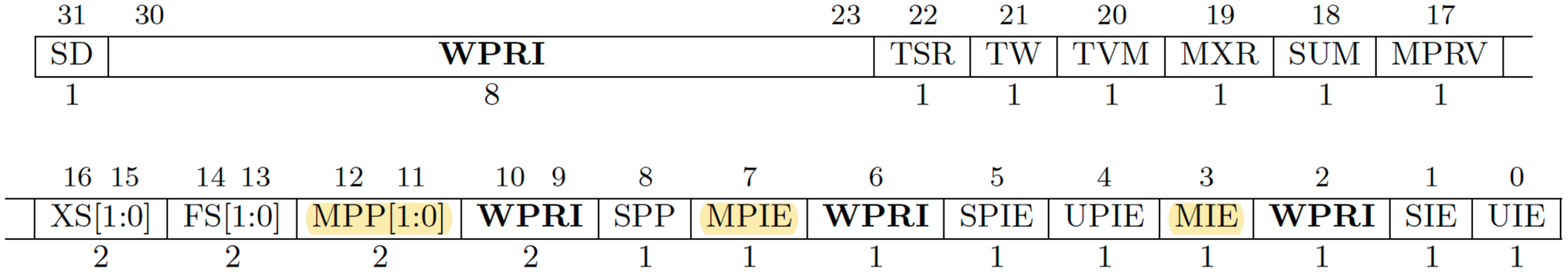

Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

Switching privilege level

Kernel can modify these 2 bits

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mcause.MPIE`.
- The `pc` is set to the value of `mepc`.

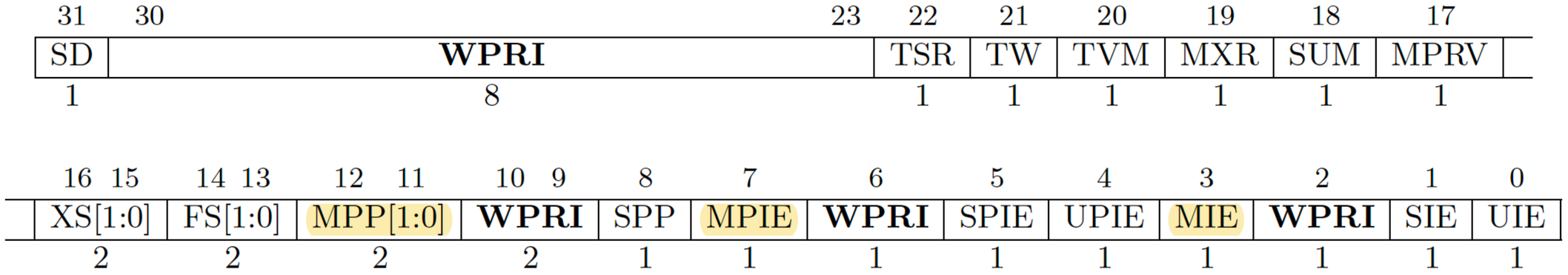

Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

In proc_yield()

```
static void proc_yield() {
    . . .
    if (curr_pid >= GPID_USER_START) {
        /* Modify mstatus.MPP to user mode */
        . . .
    } else {
        /* Modify mstatus.MPP to machine mode */
        . . .
   }
. . .
}
```

Homework

- Handle system call with the `ecall` instruction
  - i.e., `asm("ecall")`
- P2 will be due on Mar 24, but likely extended next week