Interrupt handler and privilege mode

Yu-Ju Huang, Slides adapted from Yunhao Zhang

Recap: Manage Hardware

CSR: control and status registers

- **Setting up timer**
  - **mtvec**: Machine Trap-Vector Base-Address Register
  - **mstatus**: Machine Status Register
    - bit#3: enable interrupt
  - **mie**: Machine Interrupt Enable Register
    - bit#7: enable timer interrupt

A timer handler program

```
int quantum = 50000;

void handler() {
    printf("Got timer interrupt.");
    mtimecmp_set(mtime_get() + quantum);
}

int main() {
    asm("csrw mtvec, %0" ::"r"(handler));

    int mstatus, mie;
    asm("csrr %0, mstatus" : "=r"(mstatus));
    asm("csrw mstatus, %0" ::"r"(mstatus | 0x8));
    asm("csrr %0, mie" : "=r"(mie));
    asm("csrw mie, %0" ::"r"(mie | 0x80));

    mtimecmp_set(mtime_get() + quantum);
    while(1);
}
```

Start another timer

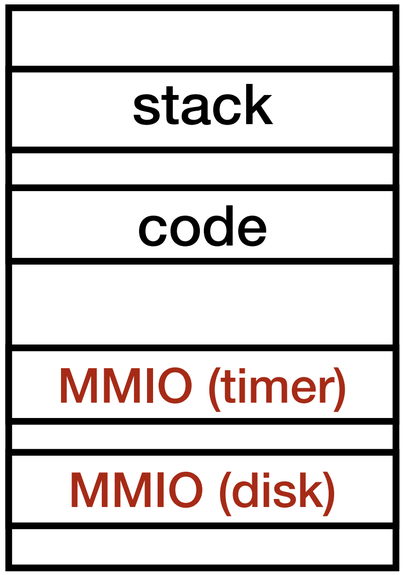
Register handler

Enable timer interrupt

Start a timer

Recap: Manage Hardware

MMIO



`mtimecmp_set()` writes 8 bytes to

`mtime_get()` reads 8 bytes from

Address	Width	Attr.	Description
0x2000000	4B	RW	msip for hart 0
0x2004008			Reserved
…			
0x200bff7			
0x2004000	8B	RW	mtimecmp for hart 0
0x2004008			Reserved
…			
0x200bff7			
0x200bff8	8B	RW	mtime
0x200c000			Reserved

Sifive FE310 manual, v19p04
https://egos.fun/book/assets/sifive-fe310-v19p04-1.pdf

```
int quantum = 500000;

void handler() {
    ...
    mtimecmp_set(mtime_get() + quantum);
}

int main() {
    ...
    mtimecmp_set(mtime_get() + quantum);
    ...
}
```

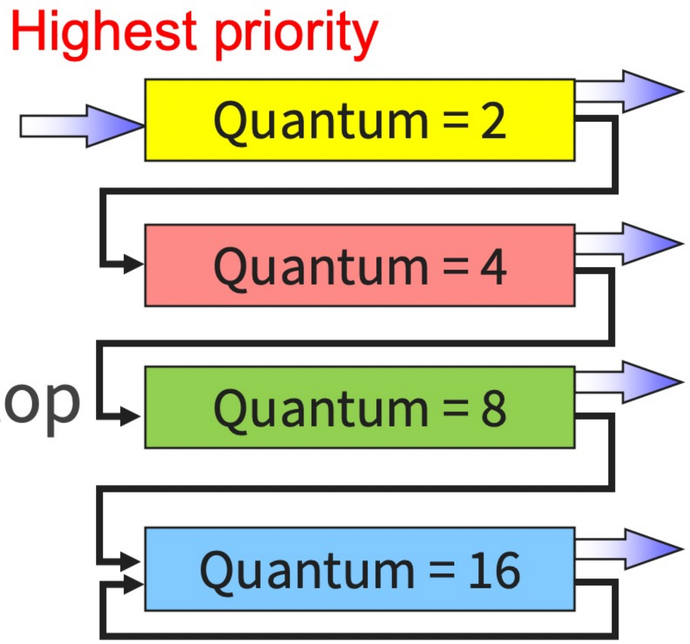
Read current time

Set timer

5411 Project: MLFQ Optional for 4411

Multi-Level Feedback Queue (MLFQ)

- Multiple levels of RR queue
- Jobs start at the top
  - Use quantum? move down
  - Don't? Stay where you are
- Periodically all jobs back to top
- *Approximates SRTF*



Need parameters for:

- Number of queues
- Quantum length per queue
- Time to move jobs back up

Used by MacOSX, Windows, some versions of Linux, …

Agenda

- Recap normal function call
- Understand interrupt handler call
- Put it all together timer & scheduler flow
- Understand privilege levels

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>:
    . . .
PC ➡ 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
PC  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>:
PC  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)
```

PC

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)
```

PC

Function call step#6

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

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

Agenda

- Recap normal function call
- Understand interrupt handler call
- Put it all together timer & scheduler flow
- Understand privilege levels

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.

Kernel Stack!

```
<main>:
    . . .

    Store caller-saved registers on the stack
    Call handler (set ?? to the address of )
    Restore caller-saved registers
    . . .

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

Problem #2

How to restore the return address? (Cannot use `ra`)

mepc!
Machine Exception Program Counter

```
<main>:
    . . .

    Store caller-saved registers on the stack
    Call handler (set mepc to the address of )
    Restore caller-saved registers
    . . .

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

Agenda

- Recap normal function call
- Understand interrupt handler call
- Put it all together timer & scheduler flow
- Understand privilege levels

```c
void intr_init(uint core_id) {
    /* Initialize the timer */
    earth->timer_reset = timer_reset;
    mtimecmp_set(0x0FFFFFFFFFFFFFFFUL, core_id);

    /* Setup the interrupt/exception handling entry */
    void trap_entry(); /* (see grass/kernel.s) */
    asm("csrw mtvec, %0" ::"r"(trap_entry));
    INFO("Use direct mode and put the address of the trap_entry into mtvec");

    /* Enable timer interrupt */
    asm("csrw mip, %0" ::"r"(0));
    asm("csrs mie, %0" ::"r"(0x80));
    asm("csrs mstatus, %0" ::"r"(0x88));
}
```

```c
static void intr_entry(uint id) {
    if (id == INTR_ID_TIMER) {
        proc_yield();
        return;
    }

    ...

    FATAL("excp_entry: kernel got interrupt %d", id);
}

static void proc_yield() {
    /* Student's code goes here (Preemptive Scheduler). */
    if (!CORE_IDLE && curr_status == PROC_RUNNING) proc_set_runnable(curr_pid);
    int next_idx = MAX_NPROCESS;
    for (uint i = 1; i <= MAX_NPROCESS; i++) {
        struct process* p = &proc_set[(curr_proc_idx + i) % MAX_NPROCESS];
        if (p->status == PROC_PENDING_SYSCALL) proc_try_syscall(p);

        if (p->status == PROC_READY || p->status == PROC_RUNNABLE) {
            next_idx = (curr_proc_idx + i) % MAX_NPROCESS;
            break;
        }
    }

    ...

    curr_proc_idx = next_idx;
    proc_set_running(curr_pid);
}
```

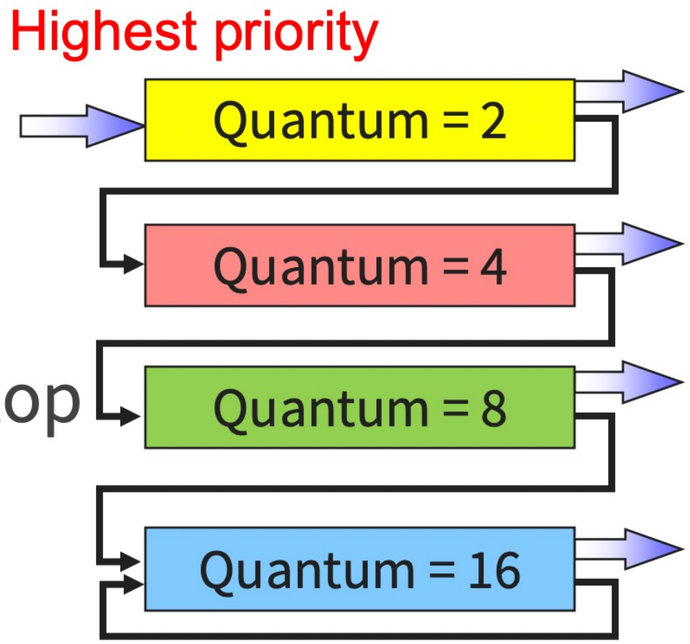
```asm
trap_entry:
    /* Step1: acquire the kernel lock (only for P8)
     * Step2: switch to the kernel stack
     * Step3: save all the registers on the kernel stack
     * Step4: call kernel_entry()
     * Step5: restore all the registers
     * Step6: switch back to the process stack
     * Step7: release the kernel lock (only for P8)
     * Step8: invoke mret and return to the process context */

    /* Step1 */
    /* Student's code goes here (Multicore & Locks). */
    /* Acquire the kernel lock and make sure not to modify any registers, */
    /* so you may need to use sscratch just like how Step2 uses mscratch. */

    /* Student's code ends here. */

    /* Step2 */
    csrw mscratch, sp
    li sp, 0x80400000

    /* Step3 */
    addi sp, sp, -128 /* sp == SAVED_REGISTER_ADDR */
    sw a0,  0(sp)
    ...
    sw t0,  120(sp)   /* t0 holds the value of the old sp before trap_entry */

    /* Step4 */
    csrr a0, mcause
    call kernel_entry

    /* Step5 */
    lw a0,  0(sp)
    ...
    lw tp,  116(sp)

    /* Step6 */
    lw sp,  120(sp)

    mret
```

```c
void kernel_entry(uint mcause) {
    /* With the kernel lock, only one core can enter this point at any time */
    asm("csrr %0, mhartid" : "=r"(core_in_kernel));

    /* Save process context */
    asm("csrr %0, mepc" : "=r"(proc_set[curr_proc_idx].mepc));
    memcpy(proc_set[curr_proc_idx].saved_register, SAVED_REGISTER_ADDR,
           SAVED_REGISTER_SIZE);

    (mcause & (1 << 31)) ? intr_entry(mcause & 0x3FF) : excp_entry(mcause);

    /* Restore process context */
    asm("csrw mepc, %0" ::"r"(proc_set[curr_proc_idx].mepc));
    memcpy(SAVED_REGISTER_ADDR, proc_set[curr_proc_idx].saved_register,
           SAVED_REGISTER_SIZE);
}
```

5411 Project: MLFQ Optional for 4411

Multi-Level Feedback Queue (MLFQ)

- Multiple levels of RR queue
- Jobs start at the top
  - Use quantum? move down
  - Don't? Stay where you are
- Periodically all jobs back to top
- *Approximates SRTF*



Need parameters for:

- Number of queues
- Quantum length per queue
- Time to move jobs back up

Used by MacOSX, Windows, some versions of Linux, …

Design choices

- Refresh the quantum every time when the process is scheduled?
- When to move process up or down?
- How many levels?
- Time quantum for each level?

- For this project, it’s all up to you!
  - As long as the interactive program won’t be blocked by cpu-bound process for too long.

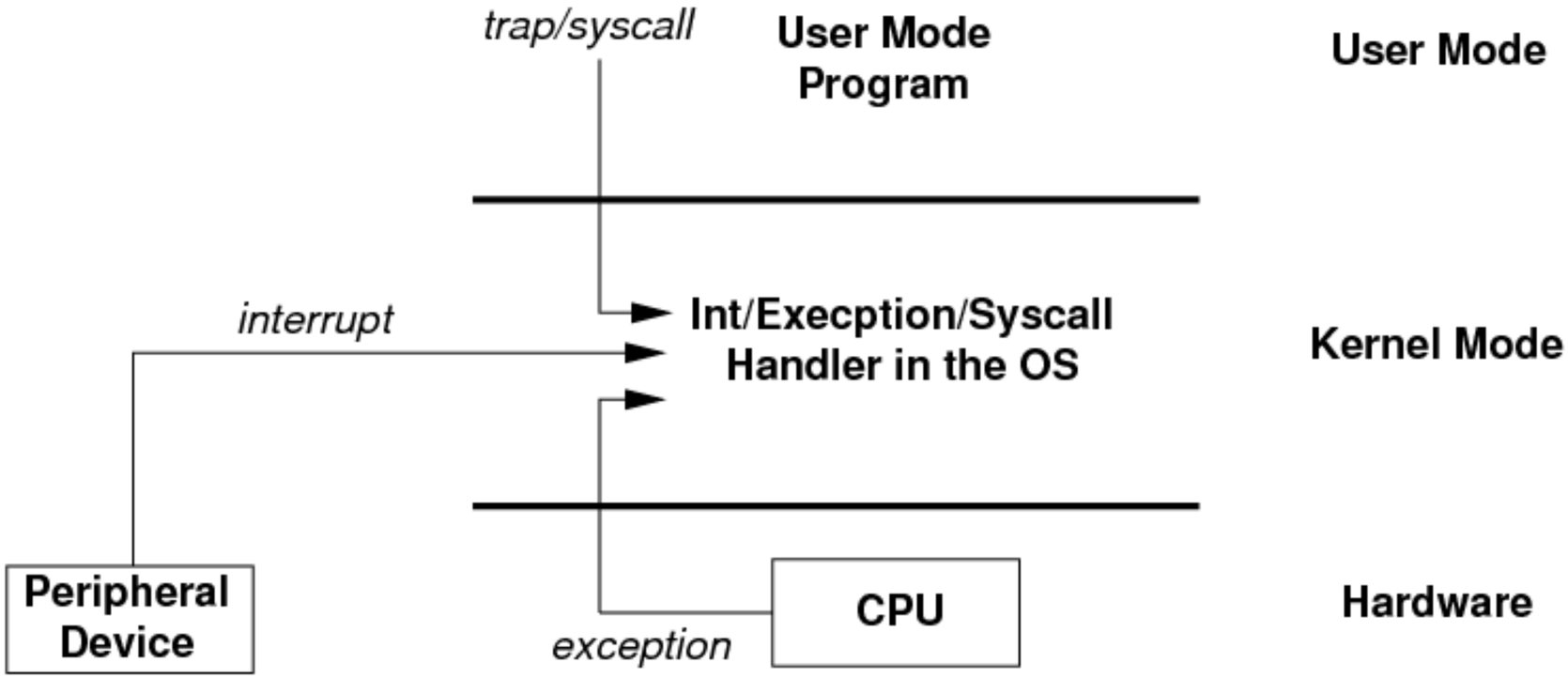
Navigate Ambiguity!

Demo

Agenda

- Recap normal function call
- Understand interrupt handler call
- Put it all together timer & scheduler flow
- Understand privilege levels

Privilege mode



https://minnie.tuhs.org/CompArch/Lectures/week05.html

When an interrupt occurs

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into `mstatus.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.

Machine Previous Privilege (MPP)

31	30 — 23	22	21	20	19	18	17
SD	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV
1	8	1	1	1	1	1	1

16 — 15	14 — 13	12 — 11	10 — 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

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

When interrupt handler returns with `mret`

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 `mstatus.MPIE`.
- The `pc` is set to the value of `mepc`.

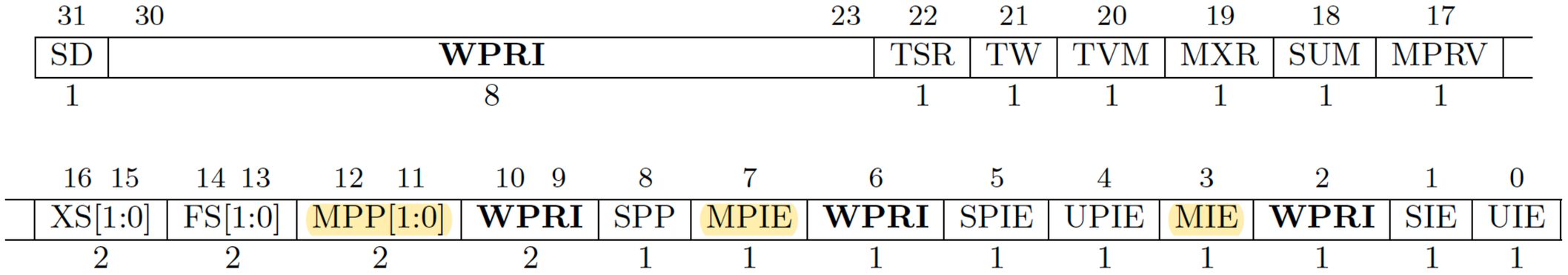


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

Switching privilege level

Kernel, as an interrupt handler,
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 `mstatus.MPIE`.
- The `pc` is set to the value of `mepc`.

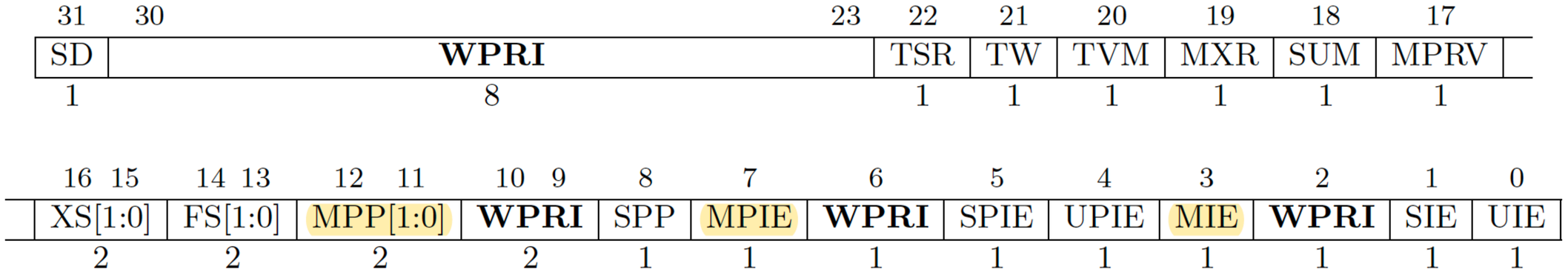


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

Today

- Interrupt handler call
- Privilege mode

- P0b due today
- P2 due next week
- P3 (5411 required, 4411 optional) release next week
- P4 release next week
  - cv_init(struct cv* condition)
  - cv_release(struct cv* condition)