Privilege Levels and Protection

Agenda

- ➔ Recap normal function call
- Understand interrupt handler call
- Understand privilege levels and protection

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
    . . .
PC ➡ 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
PC ➔ 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 ➔)
PC ➔ 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 at PC

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

<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
- Understand privilege levels and protection

Problem #1 (out of 2)

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
```

Problem #2 (out of 2)

How to restore the return address?

Cannot use `ra` after solving problem #1.

Recall that ra was restored at PC

```
<main>:
    . . .
    Store caller-saved registers (ra was saved here)
    Call handler (set ra to the address of → )
PC → Restore the ra register.
    . . . // But the code above doesn't exist now!

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

Address problem #2

<main>:

. . .

~~Store caller-saved registers on the stack~~

CPU inserts a call to handler

(set the mepc CSR to the address of)

~~Restore caller-saved registers~~

. . .

<handler>:

Store ALL registers on the handler stack

. . .

Restore ALL registers

Return to main() with mepc, which holds

Line23 of earth/cpu_intr.c

```
void trap_entry()  __attribute__((interrupt ("machine"), aligned(128)));
void trap_entry() {
```

Compiler: solution of problem 1 is [blue box] and of problem 2 is [red box]

```
20400280 <trap_entry>:
trap_entry():
20400280: fa010113          addi  sp,sp,-96
20400284: 04112e23          sw ra,92(sp)
. . . . . .                 // save other registers
. . . . . .                 // do the work of trap_entry()
20400360: 05c12083          lw ra,92(sp)
. . . . . .                 // restore other registers
204003a4: 06010113          addi  sp,sp,96
204003a8: 30200073          mret
```

Recap: ecall in P2

```
<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
```

Agenda

- Recap normal function call
- Understand interrupt handler call
- ➔ Understand privilege levels and protection

Privilege levels explained in CPU manuals

8.2.1 Interrupt Entry and Exit

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.

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 `mstatus.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 `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)

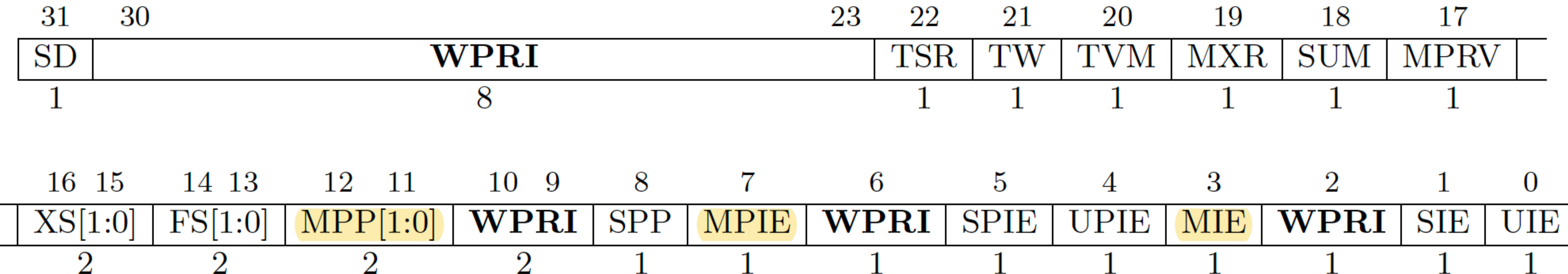


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

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`.

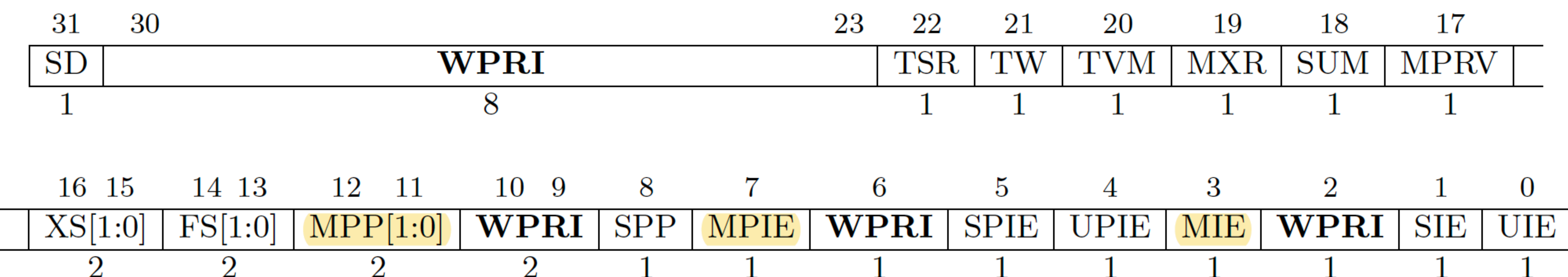
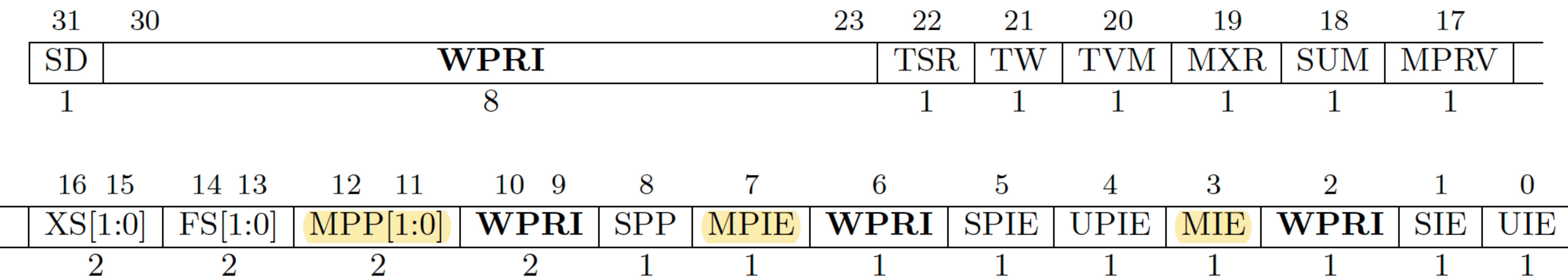


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

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`.



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.

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 */
        . . .
    }
. . .
}
```

Memory protection

- Machine mode can access all memory regions.
- OS specifies which regions can be accessed by user mode.
- In P2, you will specify 4 PMP regions for user mode
  - PMP stands for Physical Memory Protection
  - Read section 3.6 of the RISC-V reference manual

How to read CPU manuals?

An example of reading Figure 3.25

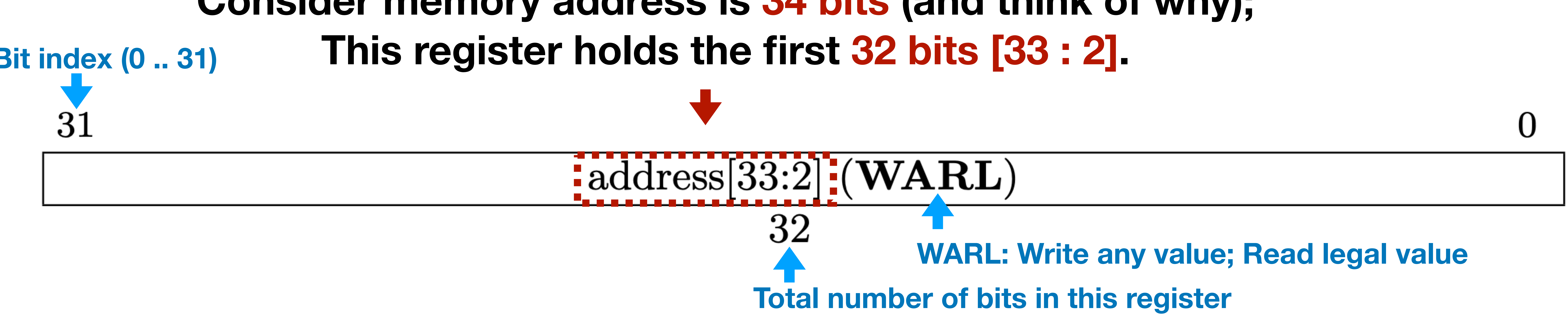


Figure 3.25: PMP address register format, RV32.

Understanding the current PMP setup

```
/* Setup a PMP region for the lowest 4GB address space */
```

The address encoded here is 0x4000_0000 << 2 == 0x1_0000_0000 (4GB)

```
asm("csrw pmpaddr0, %0" : : "r" (0x40000000));
asm("csrw pmpcfg0, %0" : : "r" (0xF));
```

PMP region0 is TOR (Top of Region), i.e., pmpaddr0 is the region top;
And it this region is enabled, readable, writable, executable.
Learn more in Figure 3.27 of the CPU manual.

Homework

- P2 is due on Oct 20
- Handle system calls using `ecall`
- Handle memory exceptions in kernel
- Setup memory protection using PMP
- Please remember to fill up the mid-term evaluation!