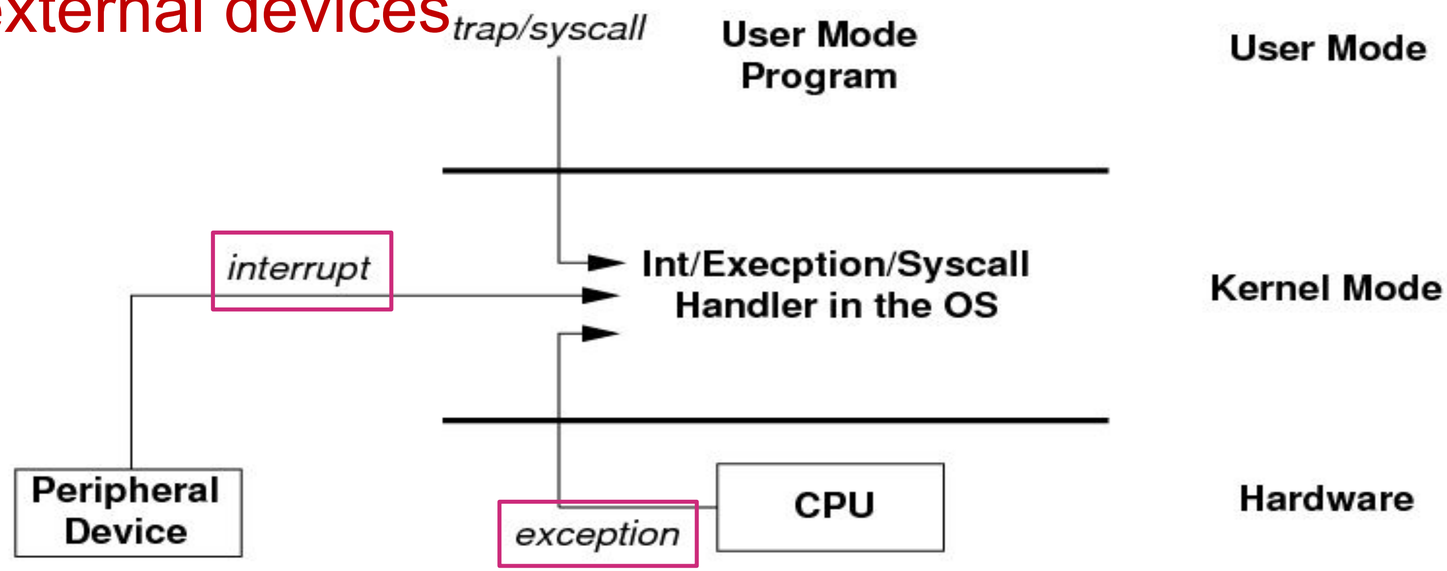
Interrupts, Privilege Levels, and Protection

Kevin A. Negy
Adapted from slides by Yunhao Zhang

Exceptions and Interrupts

- Exceptions are triggered by CPU instructions
  - Invalid memory access, divide by zero, ecall,…
  - Synchronous
- Interrupts are triggered by external devices
  - Timer, I/O, software interrupt (!)
  - Asynchronous



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

Agenda

- Recap normal function call
- Understand interrupt handler call
- RISC-V privilege levels
- RISC-V memory protection

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

Restore caller-saved registers

. . .

<myfunc>:

Store callee-saved registers on the stack

. . .

Restore callee-saved registers

Return to main() (set pc to ra)

Say main() calls myfunc()

<main>:

. . .

PC: Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

Restore caller-saved registers

. . .

<myfunc>:

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

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

PC: Call myfunc (set `ra` to the address of next line)

Restore caller-saved registers

. . .

<myfunc>:

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

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

Restore caller-saved registers

. . .

<myfunc>:

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

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

Restore caller-saved registers

. . .

<myfunc>:

Store callee-saved registers on the stack

. . .

PC: Restore callee-saved registers

Return to main() (set pc to ra)

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

Restore caller-saved registers

. . .

<myfunc>:

Store callee-saved registers on the stack

. . .

Restore callee-saved registers

PC: Return to main() (set pc to ra)

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

RA: Restore caller-saved registers

. . .

<myfunc>:

Store callee-saved registers on the stack

. . .

Restore callee-saved registers

PC: Return to main() (set pc to ra)

Say main() calls myfunc()

<main>:

. . .

Store caller-saved registers on the stack

Call myfunc (set ra to the address of next line)

PC: Restore caller-saved registers

. . .

<myfunc>:

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
- RISC-V privilege levels
- RISC-V memory protection

Problem #1

If an interrupt happens during main(),
the CPU will call interrupt_handler, but the compiler
can't predict this to store registers on main() stack. What should
happen with registers?

Address problem #1

<main>:

. . .

some code;

. . .

Timer interrupt calls handler()

<handler>:

Store ALL registers on the handler stack

. . .

Restore ALL registers

Return to main()

Problem #2

How do you get back to main? Can you use the `ra` register?

Control and Status Registers (CSRs)

0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Protection and Translation			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

Table 2.4: Currently allocated RISC-V machine-level CSR addresses.

Control and Status Registers (CSRs)

0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Protection and Translation			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

Table 2.4: Currently allocated RISC-V machine-level CSR addresses.

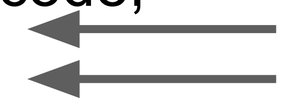
Address problem #2

<main>:

. . .

some code;

. . .



Timer interrupt calls handler()

Store next instruction address in `mepc`

<handler>:

Store ALL registers on the handler stack

. . .

Restore ALL registers

Return to main() with mepc

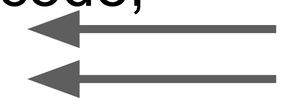
Address problem #2

<main>:

. . .

some code;

. . .



Timer interrupt calls handler()

Store next instruction address in `mepc`

<handler>:

Store ALL registers on the handler stack

. . .

Restore ALL registers

Return to main() with mepc

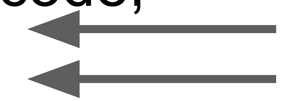
Address problem #2

<main>:

. . .

some code;

. . .



Timer interrupt calls handler()

Store next instruction address in `mepc`

<handler>:

Store ALL registers on the handler stack

. . .

Restore ALL registers

Return to main() with mepc => mret

What about exceptions?

```
<main>:
    . . .
    exception causing code;
    . . .

<handler>:
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    mret
```

What about exceptions?

<main>:

. . .

exception causing code; ← Store current instruction address in `mepc`

. . .

<handler>:

Store ALL registers on the handler stack

. . .

Restore ALL registers

mret

What about exceptions?

```
<main>:
    . . .
    exception causing code;
    . . .

<handler>:
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    if exception == syscall
        mepc += 4
    mret
```

Store current instruction address in `mepc`

Line23 of
earth/cpu_intr.c

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

```
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
```

Agenda

- Recap normal function call
- Understand interrupt handler call
- RISC-V privilege levels
- RISC-V memory protection

Privilege levels in RISC-V

1.3 Privilege Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Three RISC-V privilege levels are currently defined as shown in Table 1.1.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table 1.1: RISC-V privilege levels.

Page 3 (13 in pdf): https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

CSR mstatus register

3.1.6 Machine Status Register (mstatus)

The **mstatus** register is an XLEN-bit read/write register formatted as shown in Figure 3.6 for RV32 and Figure 3.7 for RV64 and RV128. The **mstatus** register keeps track of and controls the hart's current operating state. Restricted views of the **mstatus** register appear as the **sstatus** and **ustatus** registers in the S-level and U-level ISAs respectively.

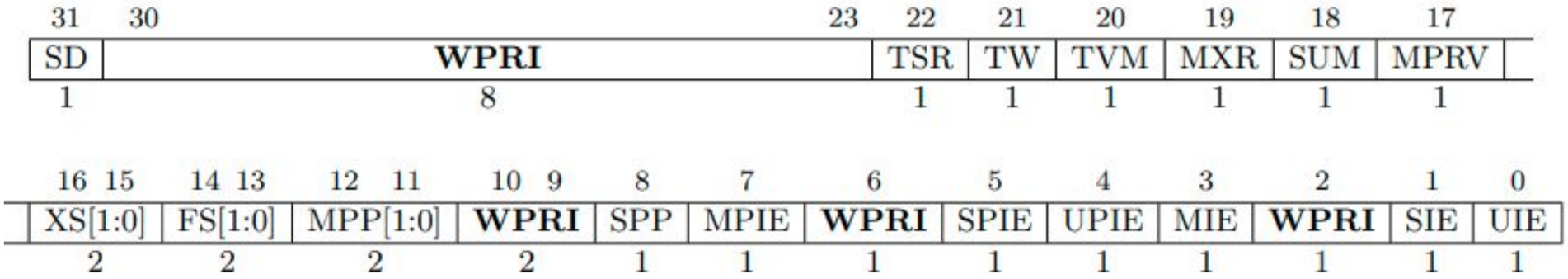


Figure 3.6: Machine-mode status register (**mstatus**) for RV32.

Page 19 (29 in pdf): https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

Interrupt handling 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

Starting an interrupt

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)

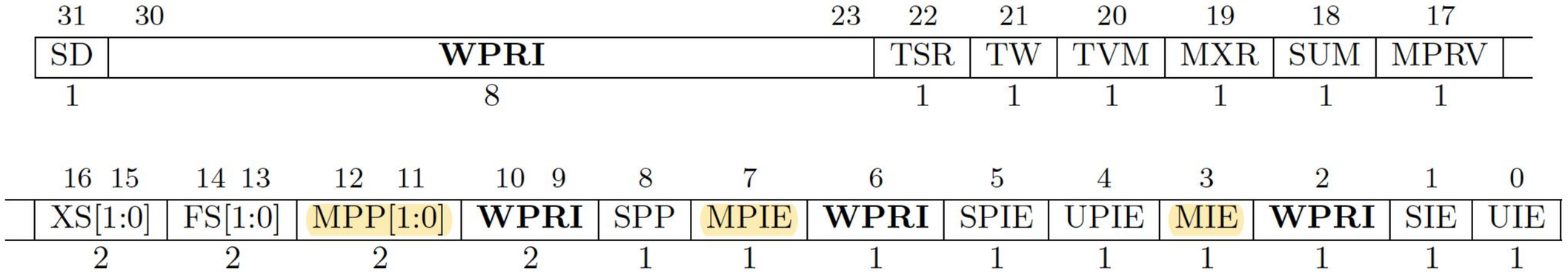


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

Returning from interrupt 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`.

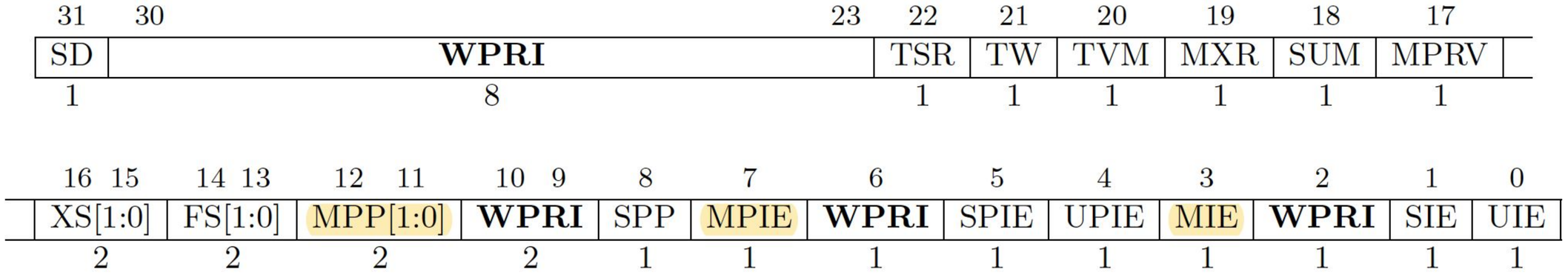


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

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

Agenda

- Recap normal function call
- Understand interrupt handler call
- RISC-V privilege levels
- RISC-V memory protection

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

PMP entries

- Comprised of (at least) two parts:
  - a PMP address (one of pmpaddr0 - pmpaddr15)
  - a PMP configuration (one of pmpcfg0 - pmpcfg15)

PMP entries

- Comprised of (at least) two parts:
  - a PMP address (one of pmpaddr0 - pmpaddr15)
  - a PMP configuration (one of pmpcfg0 - pmpcfg15)
- Smallest PMP region you can protect is 4 bytes
  - RISC-V32 has 34 bit physical address, 32 bit registers (bottom two bits not stored in PMP)

PMP entries

- Comprised of (at least) two parts:
  - a PMP address (one of pmpaddr0 - pmpaddr15)
  - a PMP configuration (one of pmpcfg0 - pmpcfg15)
- Smallest PMP region you can protect is 4 bytes
  - RISC-V32 has 34 bit physical address, 32 bit registers (bottom two bits not stored in PMP)
- Different types of PMP configurations
  - e.g. TOR, NA4, NAPOT

How to read RISC-V PMP figures?

RISC-V32 physical memory address is 34 bits;
This register holds the first 32 bits [33 : 2].

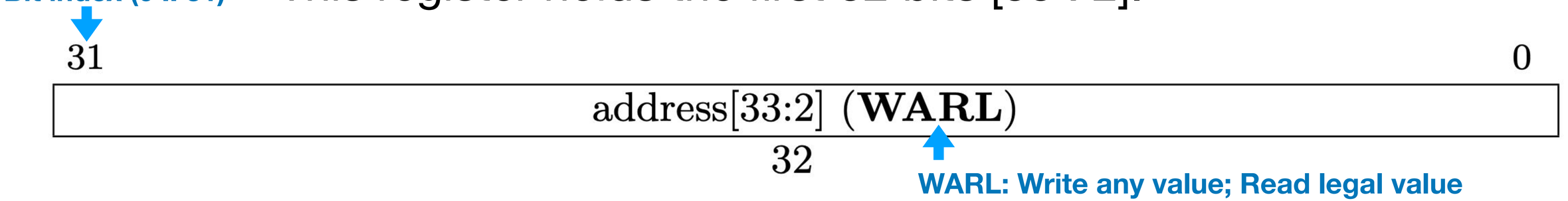


Figure 3.25: PMP address register format, RV32.

Simple PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space

Simple PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space
- A TOR (top of range) entry in `pmpaddr0` has special meaning
  - Protect range `0x0-pmpaddr0`

Simple PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space
- A TOR (top of range) entry in `pmpaddr0` has special meaning
  - Protect range `0x0-pmpaddr0`
- Convert physical address to TOR address
  - 0x1_0000_0000 (4GB) >> 2 == 0x4000_0000

Simple PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space
- A TOR (top of range) entry in `pmpaddr0` has special meaning
  - Protect range `0x0-pmpaddr0`
- Convert physical address to TOR address
  - 0x1_0000_0000 (4GB) >> 2 == 0x4000_0000
- Config 0xF means TOR, readable, writable, executable.

Simple PMP example

- Goal: Set up a PMP region for the lowest 4 GB address space
- A TOR (top of range) entry in `pmpaddr0` has special meaning
  - Protect range `0x0-pmpaddr0`
- Convert physical address to TOR address
  - 0x1_0000_0000 (4GB) >> 2 == 0x4000_0000
- Config 0xF means TOR, readable, writable, executable.

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

Homework

- P2 is due on March 15
- Handle system calls using ecall
- Handle memory exceptions in kernel
- Setup memory protection using PMP
- Please remember to fill up the mid-term evaluation!

Homework Tips

- Start early
- Read instructions and manuals very carefully
- Don’t be afraid to grep (search) and investigate egos-2000
- Test often
- Come to office hours for help