Admin

Your system nearing completion -- exciting!

Interrupts

Today

Exceptional control flow

Suspend, jump to different code, then resume

How to do this safely and correctly

Low-level mechanisms

Next lecture

Using interrupts as client

Coordination of activity

Concurrency, multiple handlers, shared data

Blocking I/O

```
while (1) {
  char ch = keyboard_read_next();
  update_screen();
}
```

How long does it take to send a scan code?
    11 bits, clock rate 15kHz
How long does it take to update the screen?
What could go wrong?
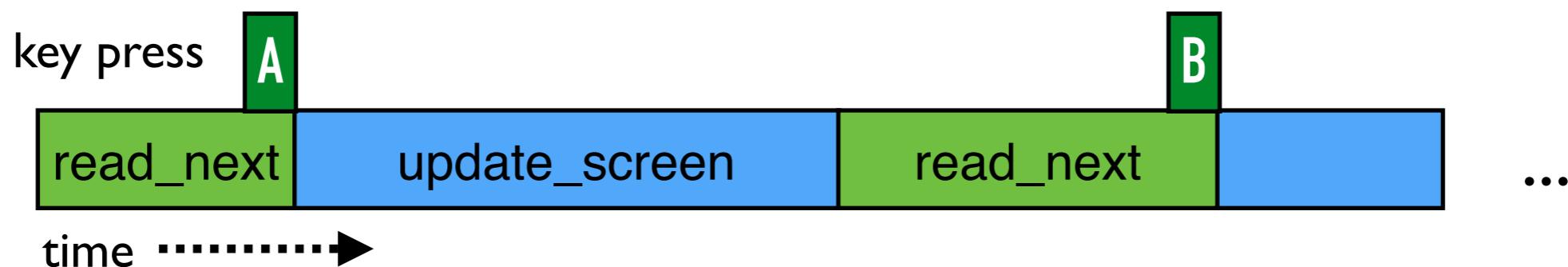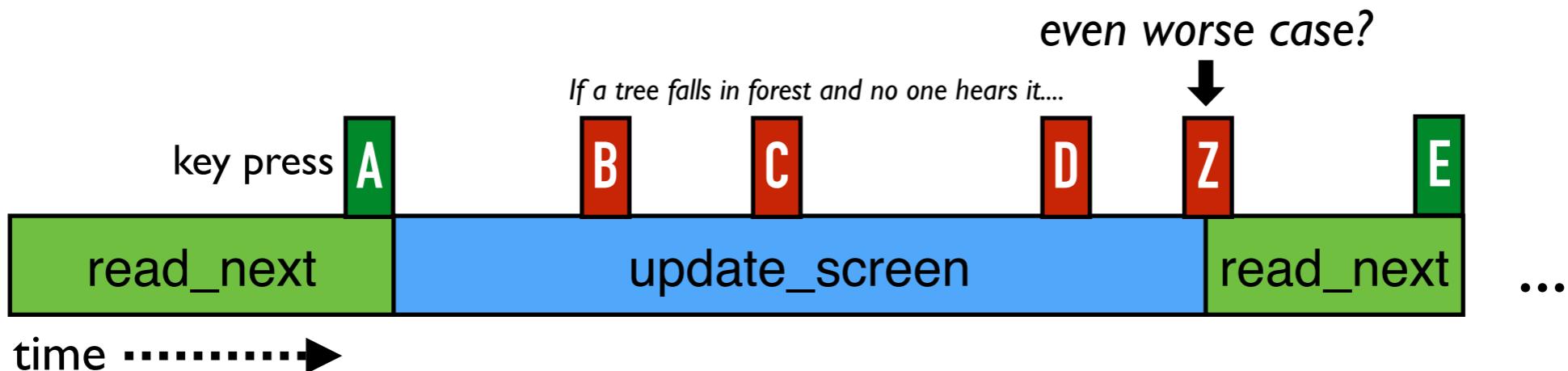
Blocking I/O

```
while (1) {
    char ch = keyboard_read_next();
    update_screen();
}
```

key press   A   B

| read_next | update_screen | read_next | | ...

time ┈┈┈┈►

Blocking I/O

```
while (1) {
    char ch = keyboard_read_next();
    update_screen();
}
```

even worse case?

If a tree falls in forest and no one hears it....

key press | A | | B | C | | D | Z | | E

read_next | update_screen | read_next | ...

time ➔

The Problem

Ongoing, long-running tasks (graphics, simulations, applications) keep CPU occupied, but …

When an external event arises, need to respond quickly.

It's like baking a chicken …

- Polling: Checking the thermometer every few minutes

- Interrupts: Wireless thermometer notifying you
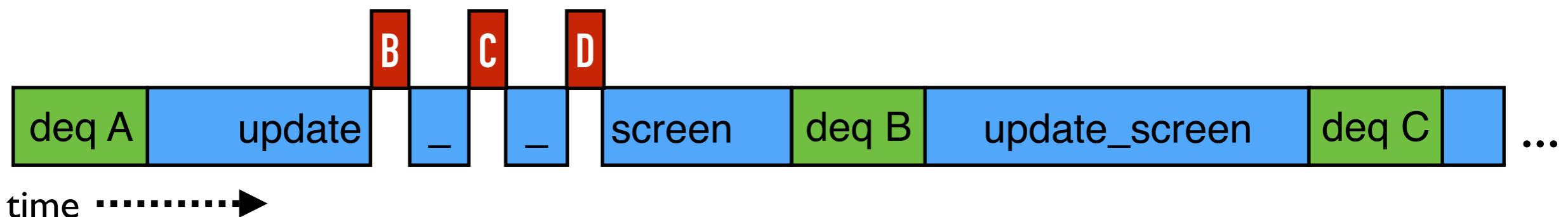
Concurrency

How to juggle doing more than one thing at once?

```
while (1) {
    dequeue scancode from queue
    update_screen();
}
```

whenever scancode arrives, enqueue it

scancodes buffered for later processing

| B | C | D |

| deq A | update | _ | _ | screen | deq B | update_screen | deq C | | ... |

time ·········▶

Interrupts to the rescue!

Processor pauses current execution, switch to code that handles interrupt, switch back and resume execution

Asynchronous external event (peripherals, timer)

Synchronous exception (invalid address, illegal instruction)

Critical for responsive systems, hosted OS

Interrupts are essential and powerful, but getting them right requires using everything you've learned:

Architecture, assembly, linking, memory, C, peripherals, …

code/button-blocking
code/button-interrupt

Interrupt mechanics

Somewhat analogous to function call

- Suspend currently executing code, save state

- Jump to handler code, process interrupt

- When finished, restore state and resume

Must adhere to conventions to avoid stepping on each other

- Consider: processor state, register use, memory

- Hardware support helps out

C abstraction ...
Asm understanding!

```
update_screen:
  add sp,sp,-16
  sd  ra,8(sp)
  sd  s0,0(sp)
  add s0,sp,16
  jal 0x400006c4 <console_clear>
Interrupt! lui a0,0x40008
  add a0,a0,496
  jal 0x40000224 <co
  li  s1,0
  j   0x400002cc <up
  lui a5,0x40022
  lw  a1,-760(a5)
  lui a0,0x40008
  add a0,a0,528 # 0x40008210
  jal 0x40000224 <console_printf>
```

How to go from **update_screen** to **button_pressed** and back again?

```
button_pressed:
  add sp,sp,-16
  sd  ra,8(sp)
```

1) pause, preserve state
2) jump to handler, execute
3) restore state, resume

```
  li  a0,1293
  jal 0x400022d8 <gpio_interrupt_clear>
  ld  ra,8(sp)
  ld  s0,0(sp)
  add sp,sp,16
  ret
```

The RISC-V Instruction Set Manual
Volume II: Privileged Architecture
Document Version 20190608-Priv-MSU-Ratified

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
June 8, 2019

https://cs107e.github.io/readings/riscv-privileged-20190608-1.pdf

Terminology

Exception

- Problem arises when executing an instruction (invalid memory address, illegal instruction)

Trap

- Synchronous transfer of control to trap handler

Interrupt

- External event that occurs asynchronously

- Executing instruction is interrupted, will experience a trap

Processor modes

Machine (most privileged), Supervisor, User (least privileged)
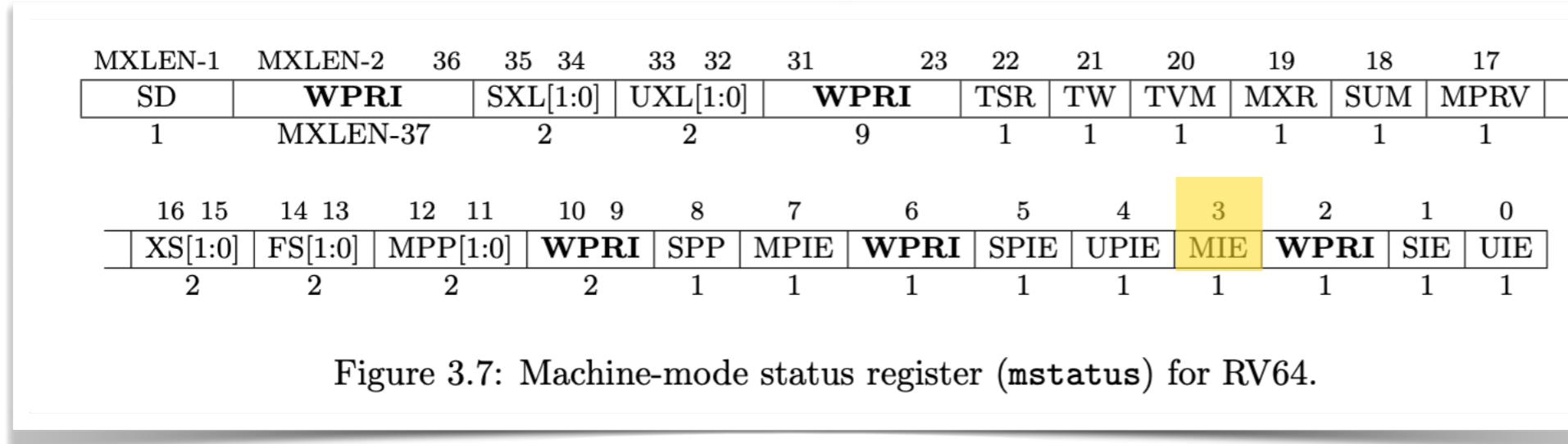
Reset starts in **machine** (M) mode
      (all our code executes in M mode)

Trap usually executes with more privilege
      (hardware support to change mode on enter/exit trap handler)
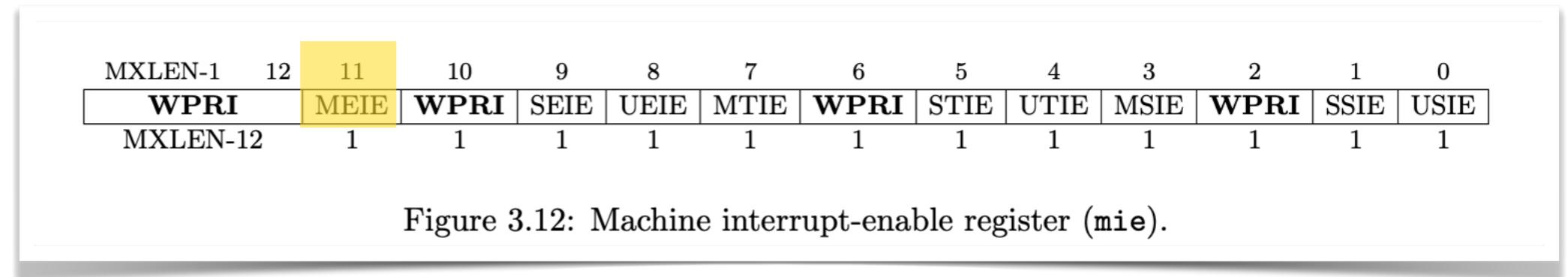
What code is executed on trap?

  - **mtvec** CSR stores address of code to jump to

CSRs

MXLEN-1	MXLEN-2	36	35 34	33 32	31	23	22	21	20	19	18	17
SD	**WPRI**		SXL[1:0]	UXL[1:0]	**WPRI**		TSR	TW	TVM	MXR	SUM	MPRV
1	MXLEN-37		2	2	9		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 3.7: Machine-mode status register (`mstatus`) for RV64.

https://cs107e.github.io/readings/riscv-privileged-20190608-1.pdf

MXLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI		MEIE	**WPRI**	SEIE	UEIE	MTIE	**WPRI**	STIE	UTIE	MSIE	**WPRI**	SSIE	USIE
MXLEN-12		1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.12: Machine interrupt-enable register (`mie`).

interrupts_global_enable:
```
    li a0,1<<11         # set MEIE bit (m-mode external interrupts)
    csrs mie,a0         # update mie register
    li a0,1<<3          # set MIE bit (global enable m-mode interrupts)
    csrs mstatus,a0     # update mstatus register
    ret
```

Interrupt, hardware-side

External event triggers interrupt line. Processor response:

    Complete current instruction

    Pause, transfer control

        Save interrupted **pc** in **mepc** CSR, elevate privilege

        Disable further interrupts

        pc = mtvec (address of handler function)

Software takes over

    Trap handler begins executing

    Must save/restore any registers it uses

    Process interrupt and clear/mark handled
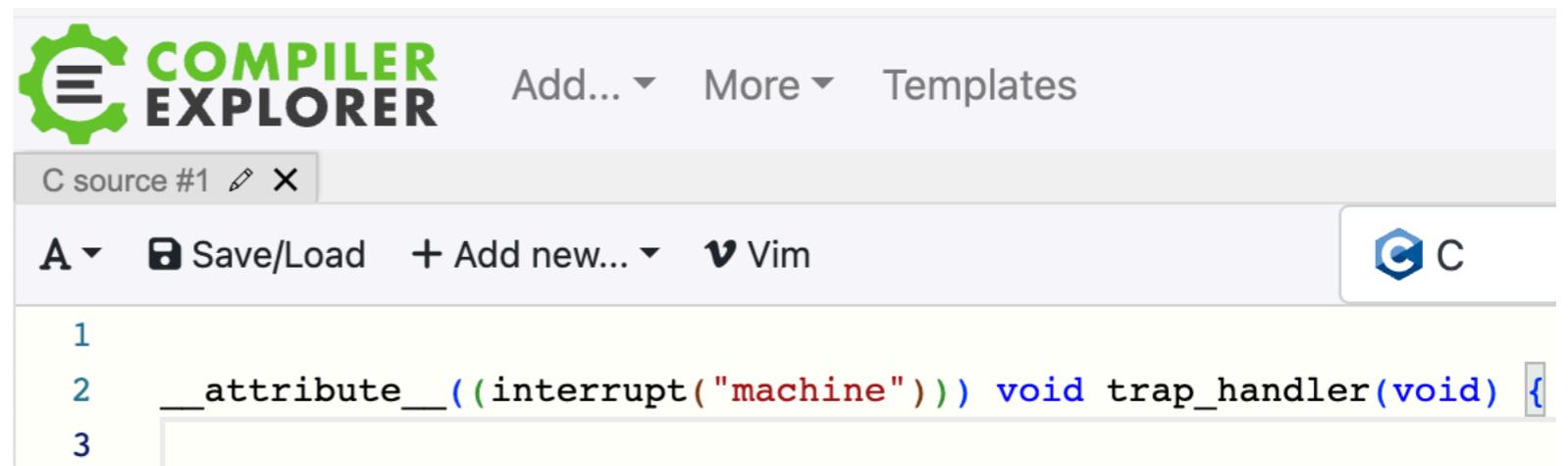
    mret instruction to restore state and resume

        Restores privilege level, reenables interrupts, **pc = mepc**

Interrupt, software-side

```c
__attribute__((interrupt("machine")))
void trap_handler(void)
{
  ....
}
```

How does this attribute change the generated code? Let's find out!

Install trap handler

```c
__attribute__((interrupt("machine"))) void trap_handler(void) {
  // code to respond to interrupt
}


void main(void) {
    set_mtvec(trap_handler); // store function pointer
}
```

```asm
interrupts_set_mtvec:     # assembly
  csrw mtvec,a0          # mtvec holds address of first insn
  ret
```

Interrupts (so far)

Hardware support
   Processor modes, exceptional control flow

Software config
   Install handler, enable interrupts

Exception received, handler must:
   Init stack, preserve registers, handle event
   Restore and resume

Interrupts (so far)

Top-level interrupts system configuration

- External interrupts enabled mstatus, mie CSRs
- Trap handler address installed in mtvec CSR

Transfer of control to enter/exit interrupt code

- Assembly to preserve registers
- Call into C code
- Assembly to restore registers, resume interrupted code

Next Lecture

Which interrupts are supported and how to configure (events from gpio, hstimer, uart, ...)

How to dispatch to specific handler per event source

What steps needed to init/enable interrupts

Writing safe interrupt handlers
  How to share state without step on each other