Put Projects P0-P2 Together

— the first 3 steps of building an operating system

Step1-3 of building an OS

➡️ Step #1: understand computer architecture

- memory and context

- function call and calling convention

- Step #2: understand interrupt and exception

- Step #3: understand context-switch and multi-threading

Before building an OS, you have

Computer

Hardware documents

Two important documents

From the CPU vendor

From the computer vendor

Required reading

- A reading assignment is released on CMSx

  - RISC-V and SiFive documents

  - because they are simpler and shorter than Intel/Dell

Chapter4 of ⬛ siFive is memory map

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	
0x0000_1000	0x0000_1FFF	R XC	Mode Select	
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4: FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

CPU debug @0000_0000
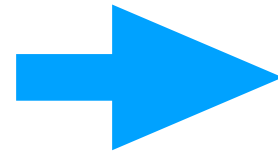(ignore this for building an OS)

Device control @0200_0000

Boot ROM @2000_0000
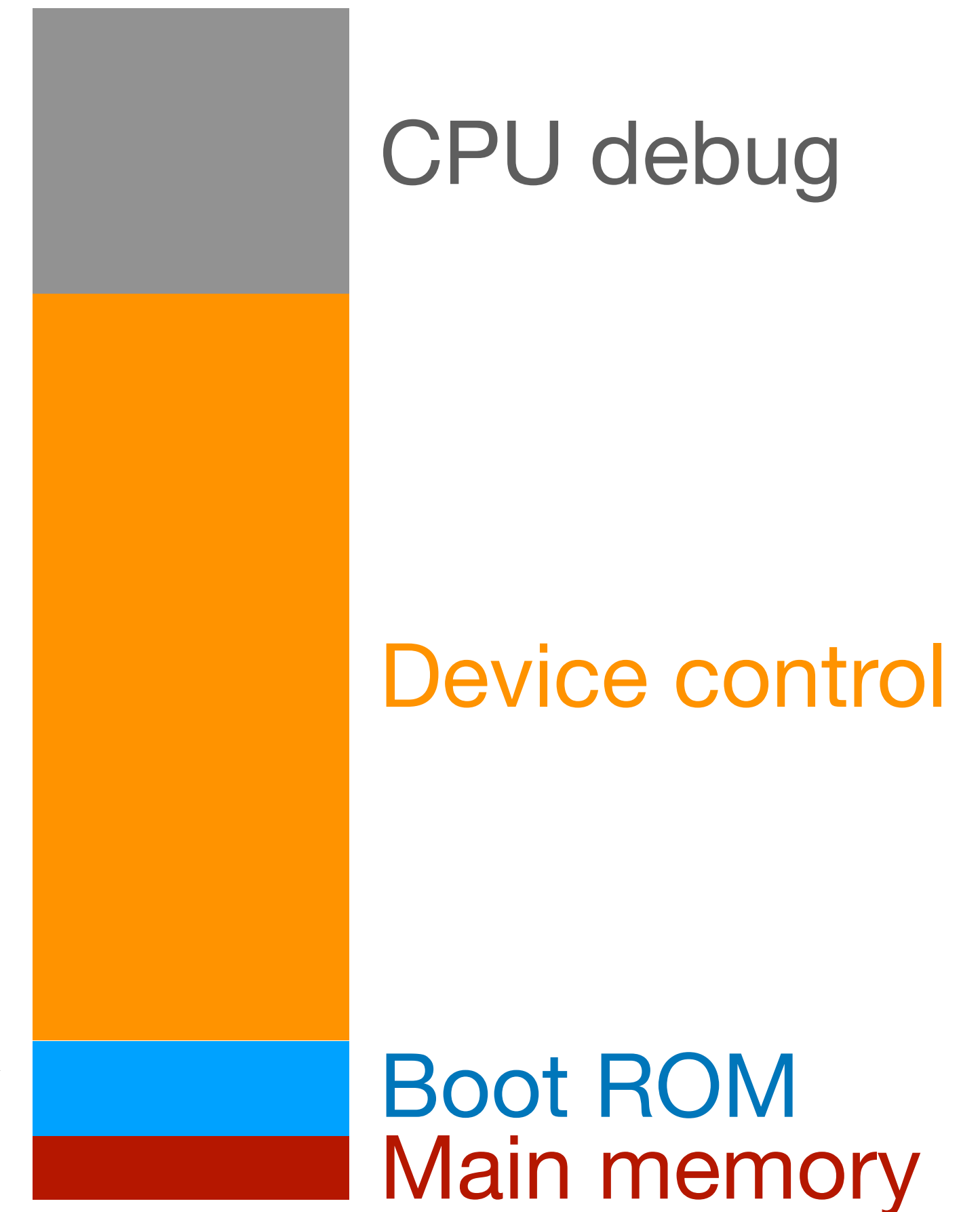Main memory @0x8000_0000

(main memory ≤2GB in this architecture)

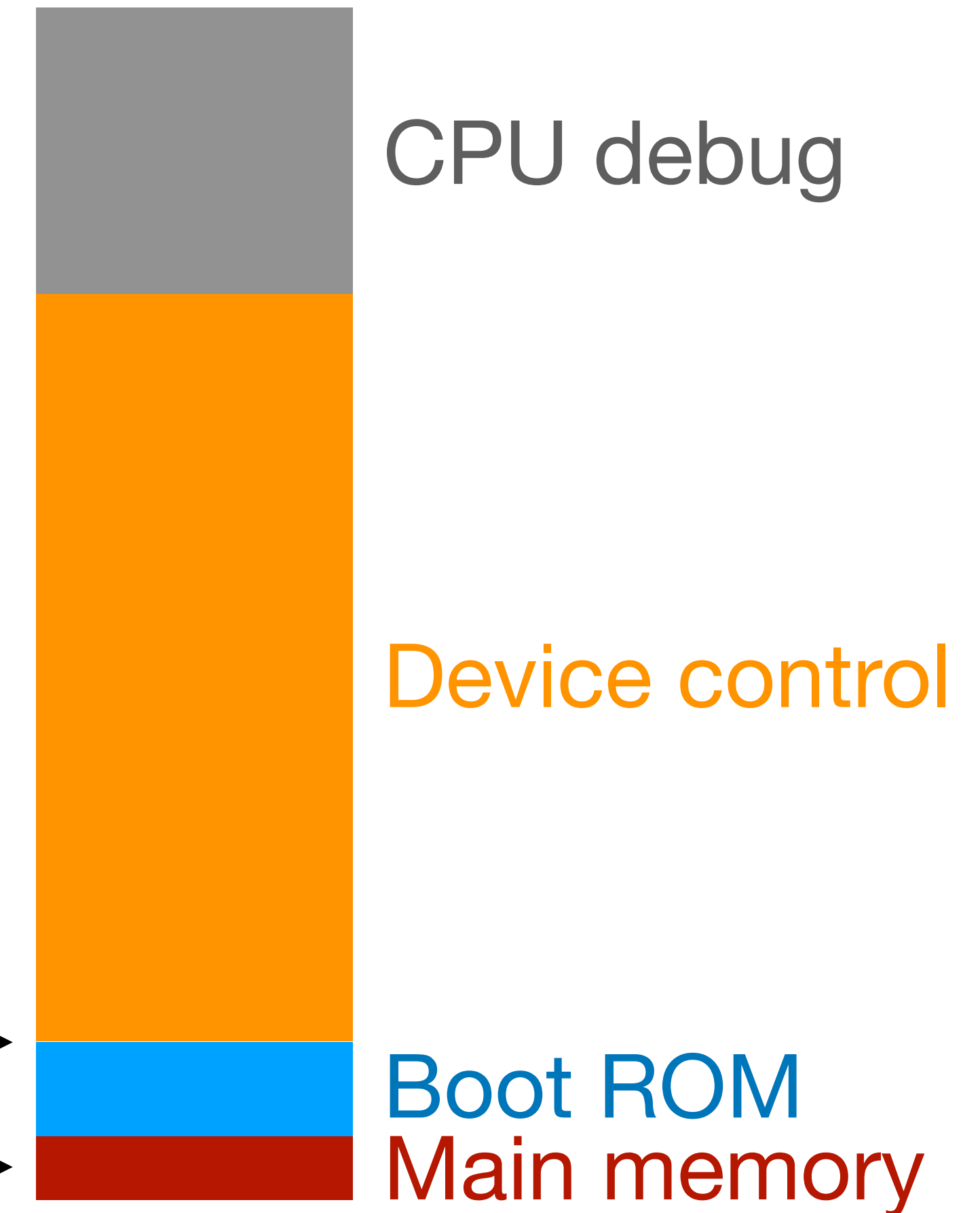# Enter the context of hello world

Step#3: press the boot button on the computer

Step#4: CPU set instruction pointer to the beginning of boot ROM

Step#5: an `li` instruction sets the stack pointer to main memory

CPU debug

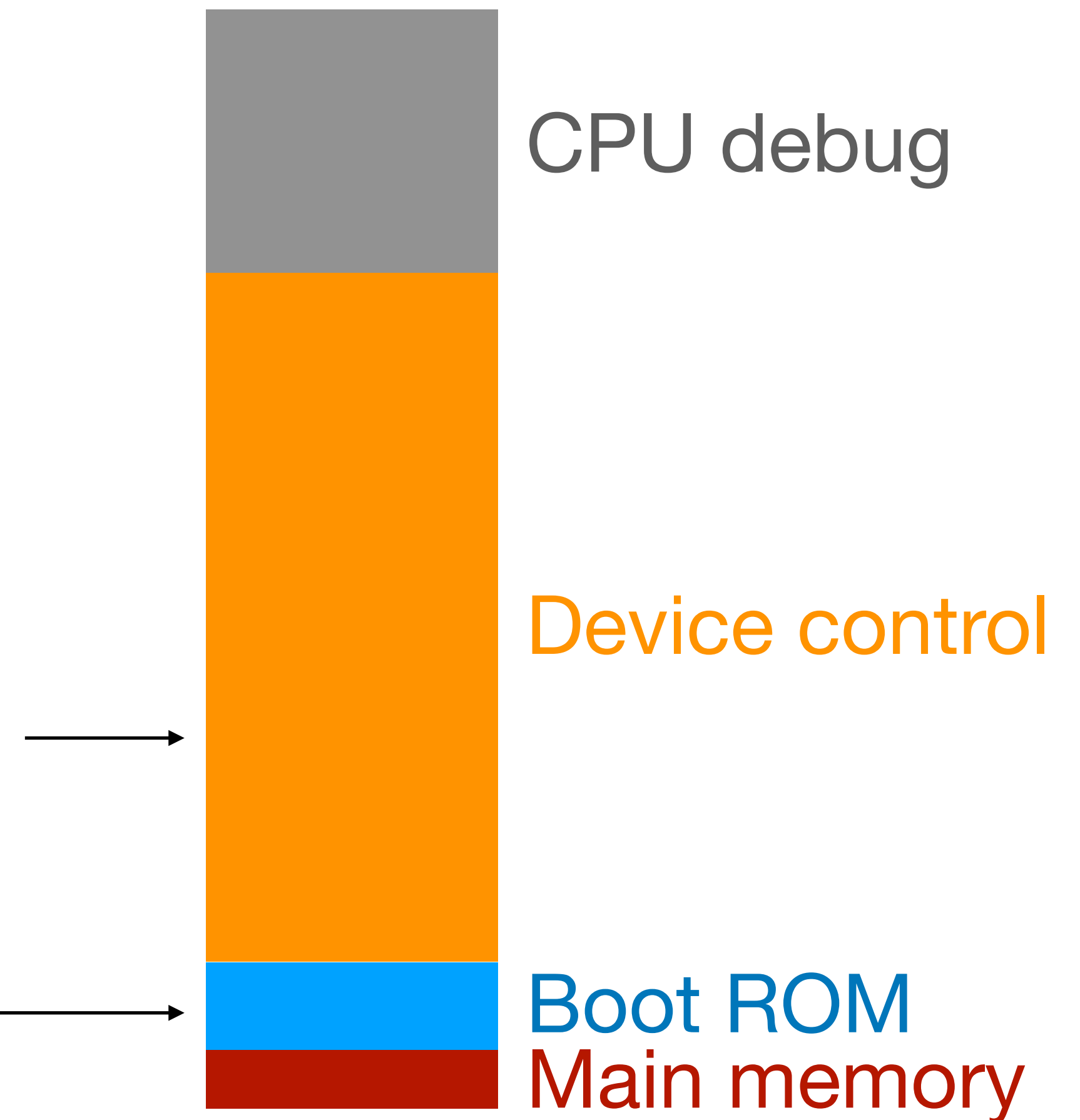Device control

Boot ROM

Main memory

hello-world prints to screen

Step#8: the screen shows "Hello World!"

Step#7: during `printf()`, `store` instructions will send data to the screen

Step#6: a `call` instruction calls `main()` which calls `printf()`

CPU debug

Device control

Boot ROM

Main memory

Step1-3 of building an OS

➡️ Step #1: understand computer architecture

    ✅ memory and context

      • function call and calling convention

• Step #2: understand interrupt and exception

• Step #3: understand context-switch and multi-threading

Calling convention in RISC-V

Register	ABI Name	Description	Saver	
x0	zero	Hard-wired zero	—	
x1	ra	Return address	Caller	main()
x2	sp	Stack pointer	Callee	printf()
x3	gp	Global pointer	—	
x4	tp	Thread pointer	—	
x5	t0	Temporary/alternate link register	Caller	main()
x6–7	t1–2	Temporaries	Caller	main()
x8	s0/fp	Saved register/frame pointer	Callee	printf()
x9	s1	Saved register	Callee	printf()
x10–11	a0–1	Function arguments/return values	Caller	main()
x12–17	a2–7	Function arguments	Caller	main()
x18–27	s2–11	Saved registers	Callee	printf()
x28–31	t3–6	Temporaries	Caller	main()

Page 137, table 25.1 of RISC-V manual, volume1
https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

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)
```

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)
```

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)

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)

Step1-3 of building an OS

➡️ Step #1: understand computer architecture

✅ memory and context

✅ function call and calling convention

- Step #2: understand interrupt and exception

- Step #3: understand context-switch and multi-threading

Can we do more than hello world?

Yes, add a simple timer handler.

Step1-3 of building an OS

- Step #1: understand computer architecture

➡ Step #2: understand interrupt and exception

  - control and status registers (CSR)

  - inserting a call to the handler function

- Step #3: understand context-switch and multi-threading

Control and status registers (CSR)

- There are **many** registers other than the 32 user-level ones:

  - `misa`: 32-bit or 64-bit?

  - `mhartid`: the core ID number

  - `mstatus`: the machine status

  - `mtvec`, `mie`, `mtime`, `mtimecmp`: interrupt handling

Recap: timer interrupt

- How to register an interrupt handler?

  - write the address of `handler` function to `mtvec`

- How to set a timer?

  - write (`mtime` + `quantum`) to `mtimecmp`

- How to enable timer interrupt?

  - set certain bit of `mstatus` and `mie` to 1

Recap: a timer handler program

```c
int quantum = 50000;

void handler() {
    earth->tty_info("Got timer interrupt.");
    mtimecmp_set(mtime_get() + quantum);
}

int main() {
    earth->tty_success("A timer interrupt example.");

    asm("csrw mtvec, %0" ::"r"(handler));
    mtimecmp_set(mtime_get() + quantum);

    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));

    while(1);
}
```

Set a timer
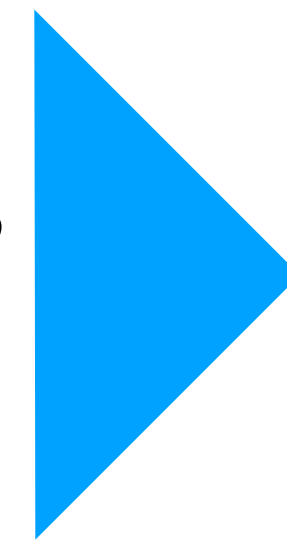
Register handler
Set a timer

Enable timer interrupt

How does the RISC-V processor insert a call to the handler function?

Recall the main-printf example

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

    . . .

<handler>:                    // instead of printf()
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to some_user_function() with ra
```

Intuition: CPU inserts these code

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

    . . .


<handler>:
        Store callee-saved registers on the stack

    . . .

        Restore callee-saved registers
        Return to some_user_function() with ra
```
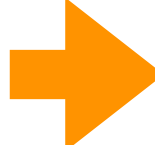
Cleanup these code

```
<some user function>:
    . . .
    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 stack
    . . .
    Restore all registers
    Return to some_user_function() with ra
```
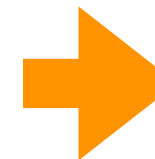
Handler returns to the same context

```
<some user function>:
    . . .
    Call handler (set ra to the address of ➡ )
➡  . . .


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

Question

How does the handler function switch to the context to a different thread?

First, replacing *ra* with CSR mepc

```
<some user function>:

    . . .
    // mepc: machine exception program counter
    Call handler (set mepc to the address of ➡ )
➡  . . .


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

Then, switch context with mepc

```
<some user function>:
    . . .
    Call handler (set mepc to the address of ➡ )
➡  . . .


<handler>:
    Store all registers on the stack
    . . .
    Set mepc to the code section of another thread
    Restore all registers
    Switch to another thread with mepc
```

Brief summary

- The interrupt handler function

  - Stores all register on stack, instead of callee-saved

  - Uses `mret` and `mepc` instead of `ret` and `ra`

- This is why, in the demo code, there is one line:

  - `void handler() __attribute__((interrupt));`

  - telling the compiler this function is an interrupt hander

A demo using mepc and mret

```
void thread0() { while(1) { // print something green } }
void thread1() { while(1) { // print something yellow } }

int next_thread = 0;
void handler() {
    next_thread = 1 - next_thread;
    asm("csrw mepc, %0" ::"r"((next_thread == 0)? thread0 : thread1));

    mtimecmp_set(mtime_get() + quantum);              // reset timer
    asm("li sp, 0x80002000");                         // set stack pointer
    asm("mret");           // forget previous thread and start a new thread
}
```

Demo on a RISC-V board

https://github.com/yhzhang0128/egos-2000/tree/timer_example/grass
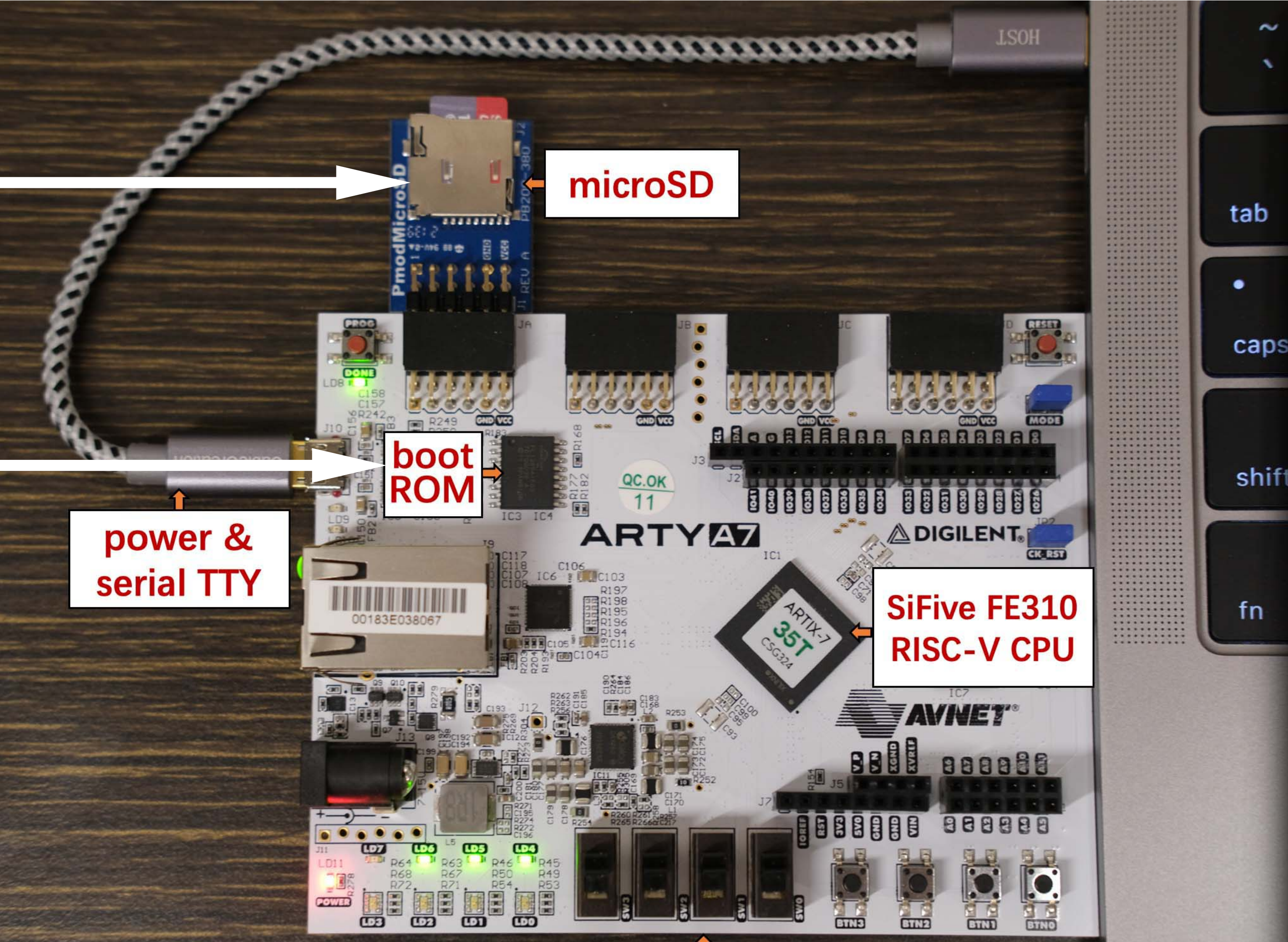

demo code in microSD card

earth layer code in boot ROM
1. Load demo from microSD
2. Print strings to the screen

Step1-3 of building an OS

- Step #1: understand computer architecture

➡️ Step #2: understand interrupt and exception

   ✅control and status registers (CSR)

   ✅inserting a function call to the handler

- Step #3: understand context-switch and multi-threading

Can we do more than timer handler?

Yes, add thread init/create/switch/exit just like P1.

Step1-3 of building an OS

- Step #1: understand computer architecture

- Step #2: understand interrupt and exception

➡️ Step #3: understand context-switch and multi-threading

  - skipped in this lecture since you just finished P1

  - read the grass kernel (358 lines of code)

    - https://github.com/yhzhang0128/egos-2000/tree/main/grass

Homework

- Read

  - the two CPU documents posted on CMSx

  - the 358 lines of code mentioned in the previous slide

- P2 is due on Oct 12.

- No class on Oct. 7 — enjoy your fall break.

After the fall break

- CS4411 has 12 lectures and we have finished half.

  - Step #1: understand computer architecture

  - Step #2: understand interrupt and exception

  - Step #3: understand context-switch and multi-threading

  - Step #4: understand privilege levels

  - Step #5: understand I/O devices

  - Step #6: understand file systems