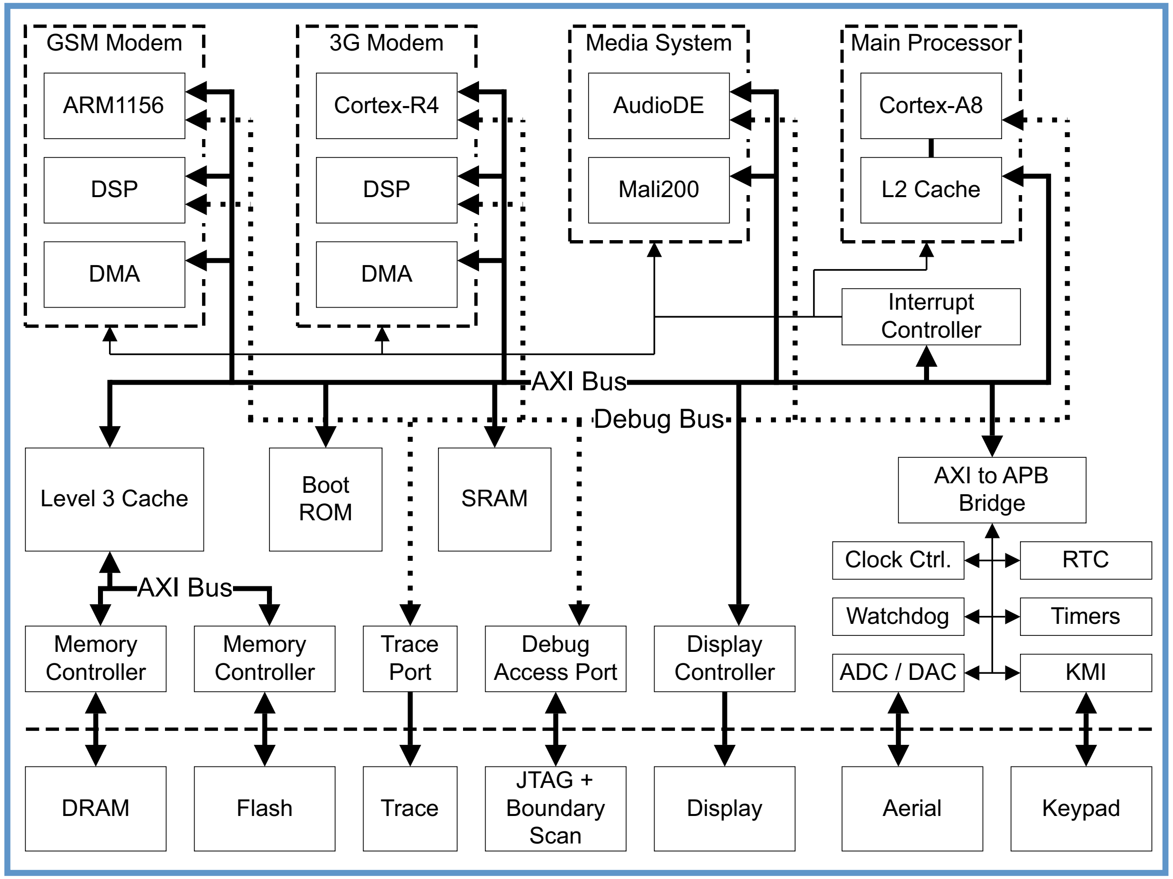
Hardware and Booting

Chester Rebeiro
IIT Madras

System Organization



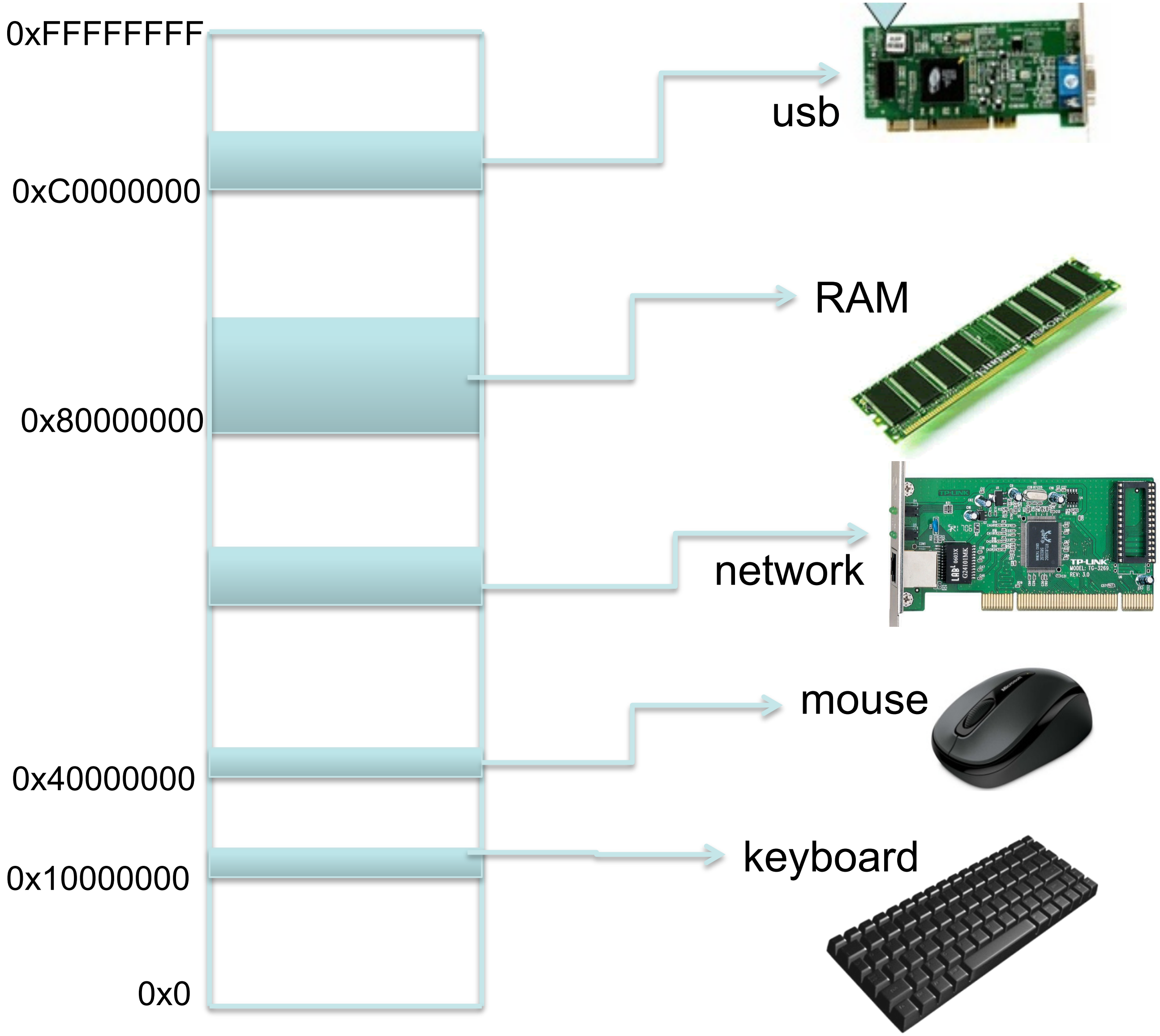
the north and south bridges together are called a CHIPSET module now

Address Types

- Memory Addresses

- Memory Mapped IO Addresses

Memory Maps



0xFFFFFFFF

0xC0000000

0x80000000

0x40000000

0x10000000

0x0

usb

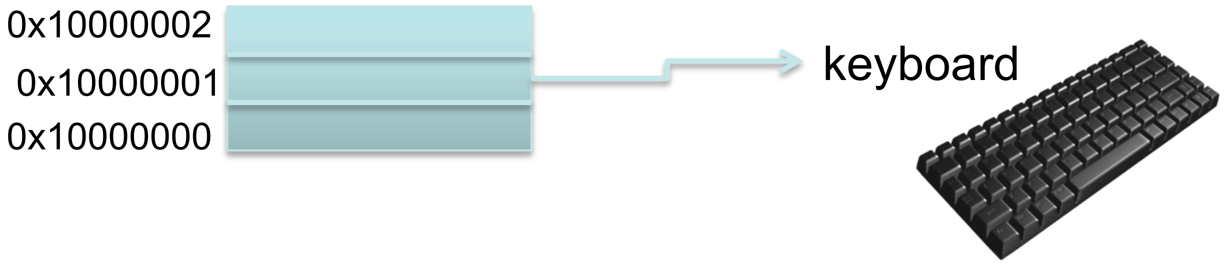
RAM

network

mouse

keyboard

Accessing keyboards



2 input ports and 1 output port

Pointer to the keyboard:

```
char *keyboard_ptr = (char *) 0x10000000;
```

Read from keyboard

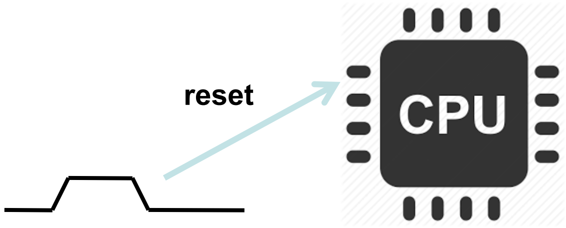
```
char x = *(keyboard_ptr + 1);
```

Write to keyboard

```
*(keyboard) = y;
```

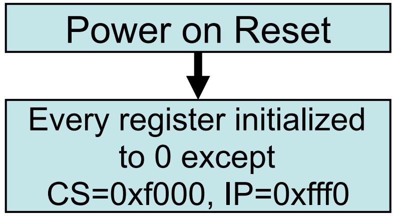
Powering Up

Power on Reset

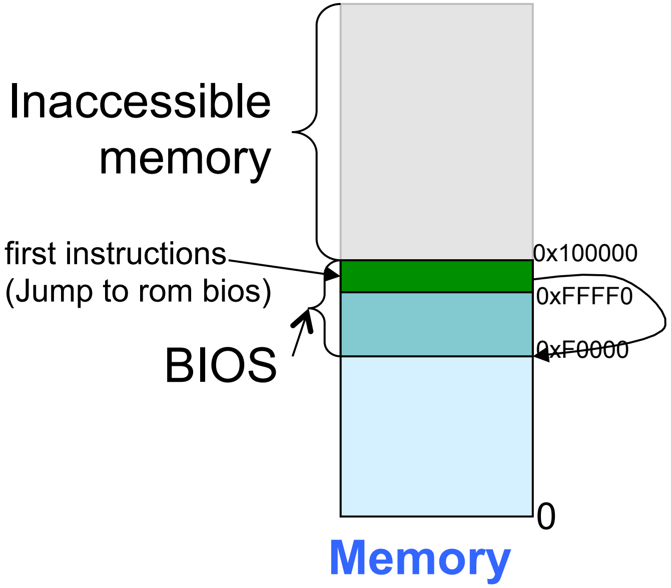


power reset signal and clock are needed by all cpus to start and work

Powering up : Reset

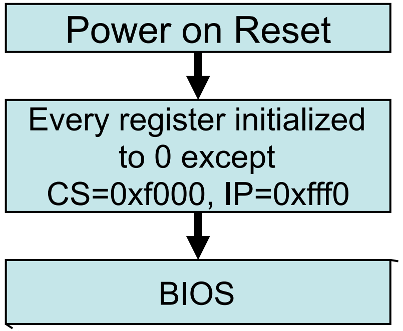


the first instruction after boot is at address 0xfff0 -> the start instruction
this usually points to the start of the Boot ROM -> the bios is present in this in most cases



ami bios is used everywhere meh

Powering up : BIOS



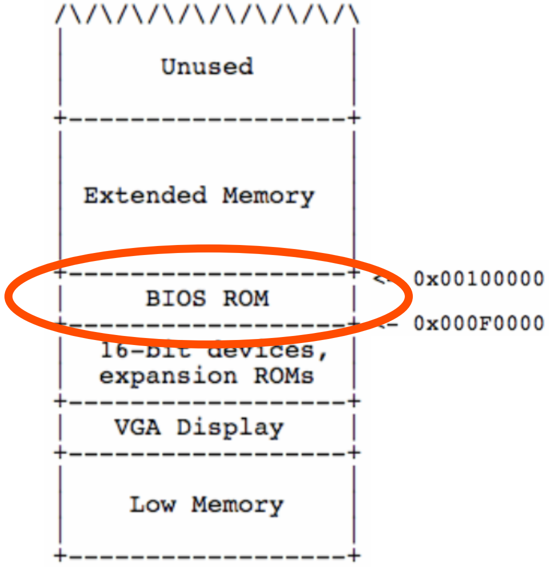
Power on Reset

Every register initialized to 0 except CS=0xf000, IP=0xfff0

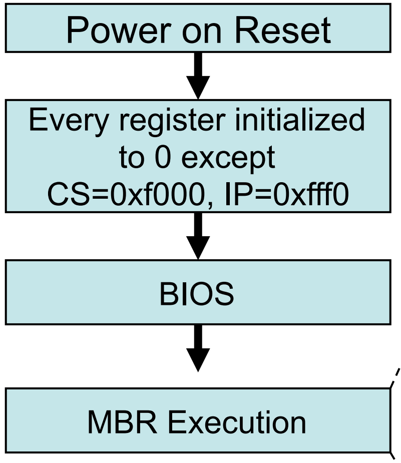
BIOS

- Present in a small chip connected to the processor
  - Flash/EPROM/E^{2}PROM
- Does the following
  - Power on self test
  - Initialize video card and other devices
  - Display BIOS screen
  - Perform brief memory test
  - Set DRAM memory parameters
  - Configure Plug & Play devices
  - Assign resources (DMA channels & IRQs)
  - Identify the boot device
    - Read sector 0 from boot device into memory location **0x7c00**
    - Jumps to 0x7c00

the sector 0 usually has the default
os or the grub to choose which os to run
it is then pushed to this address and exectuted
from that point



Powering up : MBR



Power on Reset

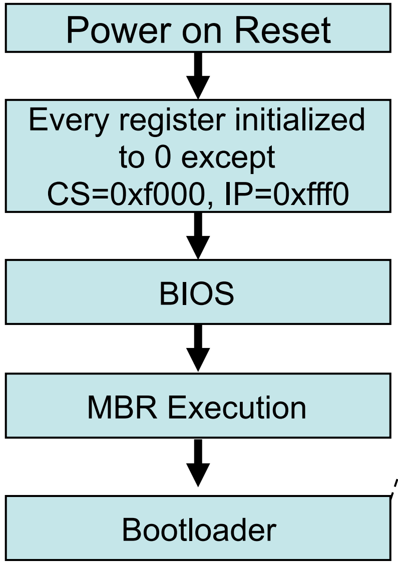
Every register initialized to 0 except CS=0xf000, IP=0xfff0

BIOS

MBR Execution

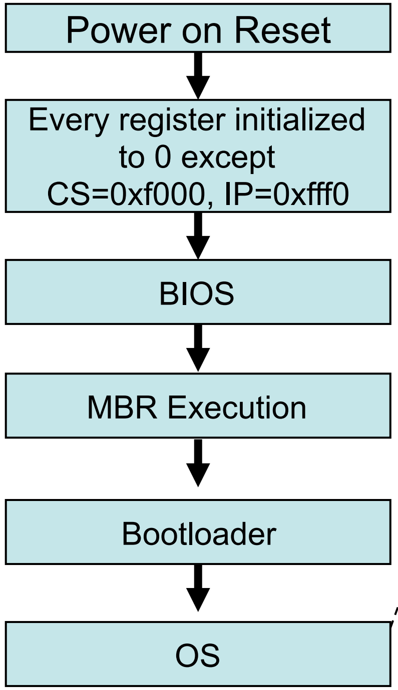
- Sector 0 in the disk called Master Boot Record (MBR)
- Contains code that boots the OS or another boot loader
- Copied from disk to RAM by BIOS and then begins to execute
- Size 512 bytes
  - 446 bytes bootable code
  - 64 bytes disk partition information (16 bytes per partition)
  - 2 bytes signature
- Typically, MBR code looks through partition table and loads the bootloader (such as Linux or Windows)
- or, it may directly load the OS

Powering Up : bootloader



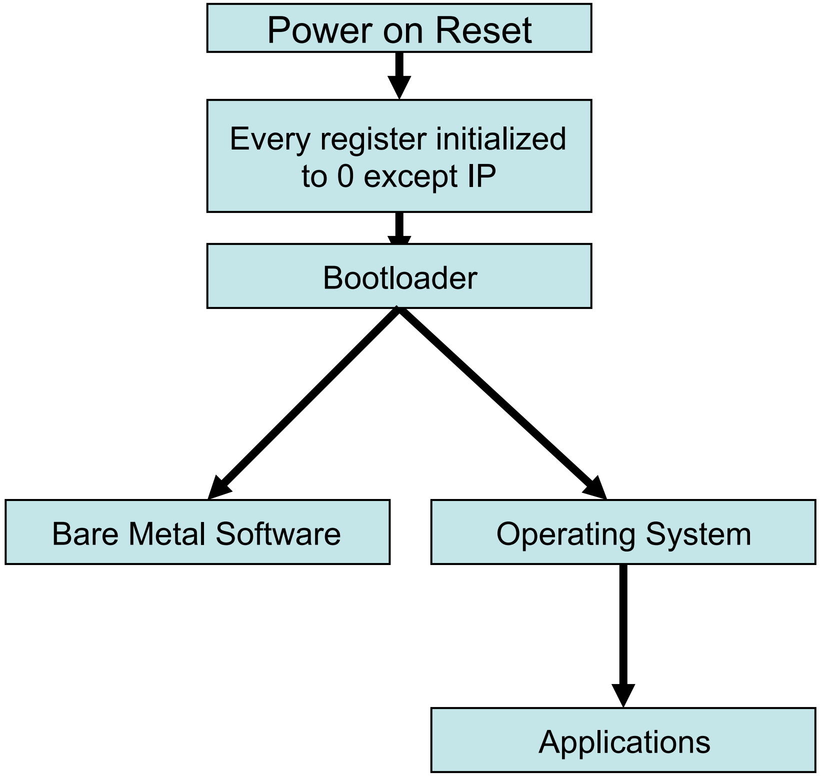
- Loads the operating system
  - May also allow the user to select which OS to load (eg. Windows or Linux)
- Other jobs done
  - Disable interrupts :
    - Don’t want to bother with interrupts at this stage
    - Interrupts re-enabled by xv6 when ready
  - Setup GDT
  - Switch modes (eg. Machine mode to Supervisor mode)

Powering Up : OS



- Set up virtual memory
- Initialize interrupt vectors
- Initilize
  - timers,
  - monitors,
  - hard disks,
  - consoles,
  - filesystems,
- Initialized other processors (if any)
- Startup user process

Powering up an Embedded Device



Power on Reset

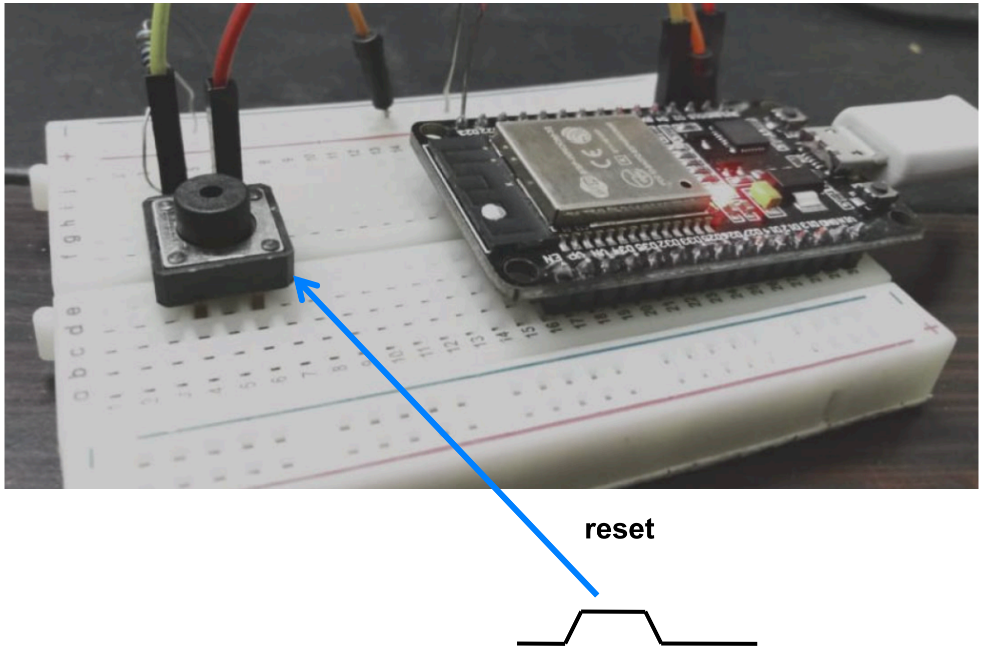
Every register initialized to 0 except IP

Bootloader

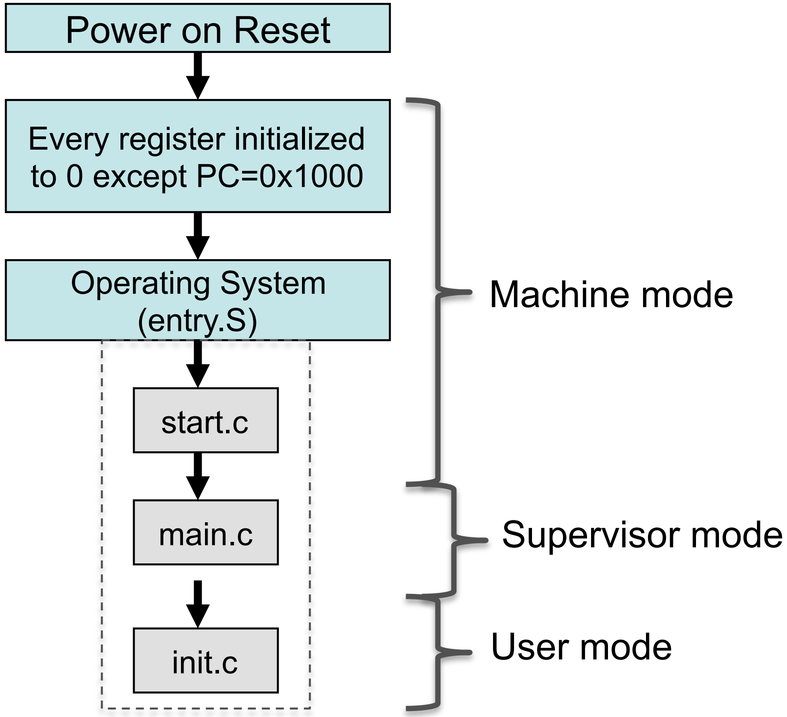
Bare Metal Software

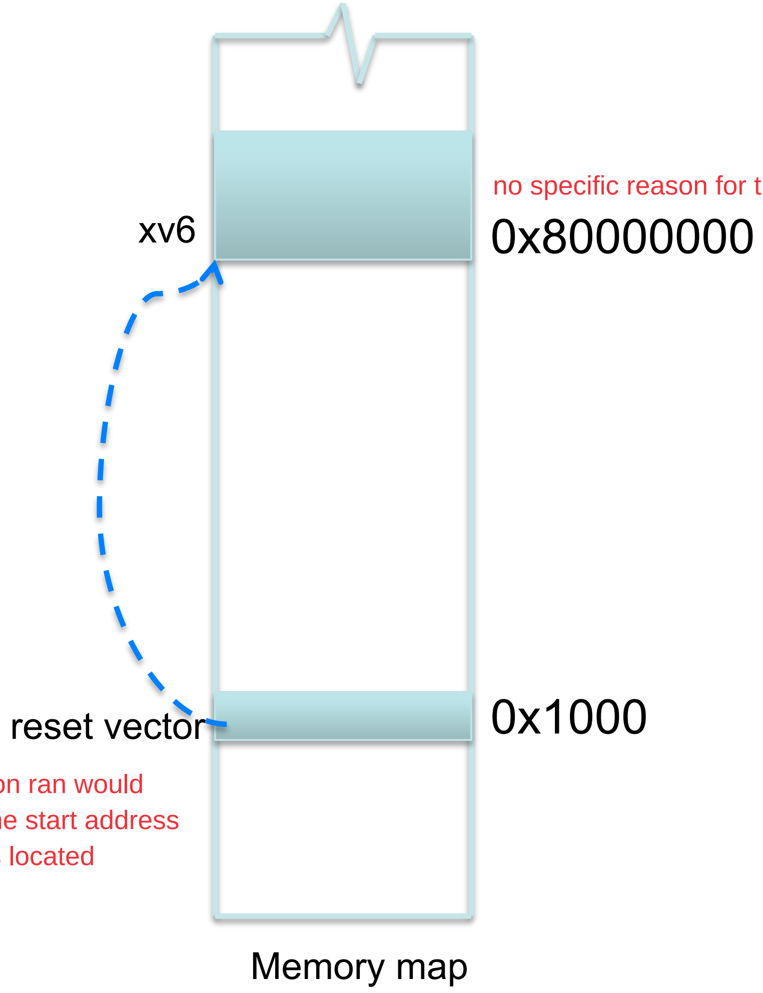
Operating System

Applications



Powering up xv6 on qemu


Power on Reset
Every register initialized to 0 except PC=0x1000
Operating System (entry.S)
start.c
main.c
init.c
Machine mode
Supervisor mode
User mode

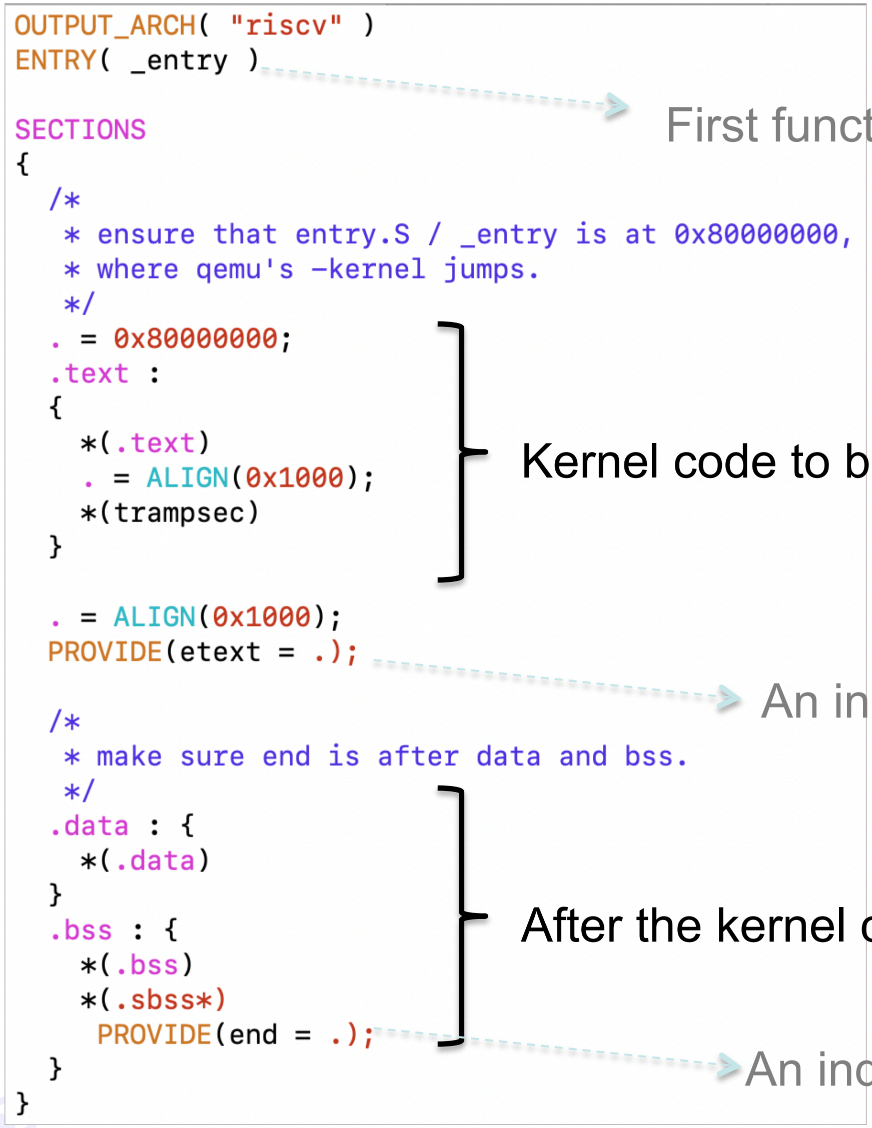


no specific reason for this location

the first execution ran would be an jump to the start address where the OS is located

Linker Descriptor

kernel.ld



```
OUTPUT_ARCH( "riscv" )
ENTRY( _entry )

SECTIONS
{
  /*
   * ensure that entry.S / _entry is at 0x80000000,
   * where qemu's -kernel jumps.
   */
  . = 0x80000000;
  .text :
  {
    *(.text)
    . = ALIGN(0x1000);
    *(trampsec)
  }

  . = ALIGN(0x1000);
  PROVIDE(etext = .);

  /*
   * make sure end is after data and bss.
   */
  .data : {
    *(.data)
  }
  .bss : {
    *(.bss)
    *(.sbss*)
     PROVIDE(end = .);
  }
}
```

First function to be executed (before main)

find out how the linker ensures the entry instructions are put at that instruction

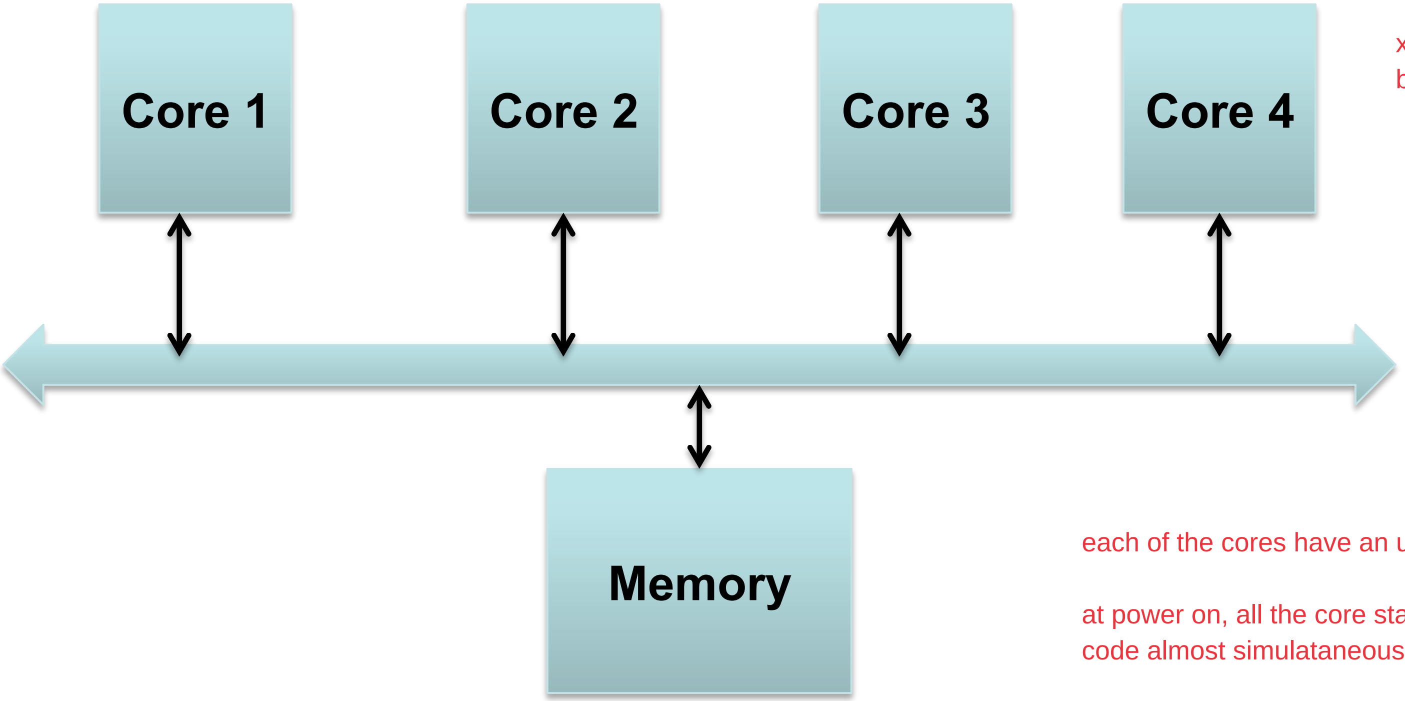
Kernel code to be placed in 0x80000000

An indicator of the end of kernel code

After the kernel code, place data and then bss

An indicator of the end of kernel

Powering Up : xv6 on qemu



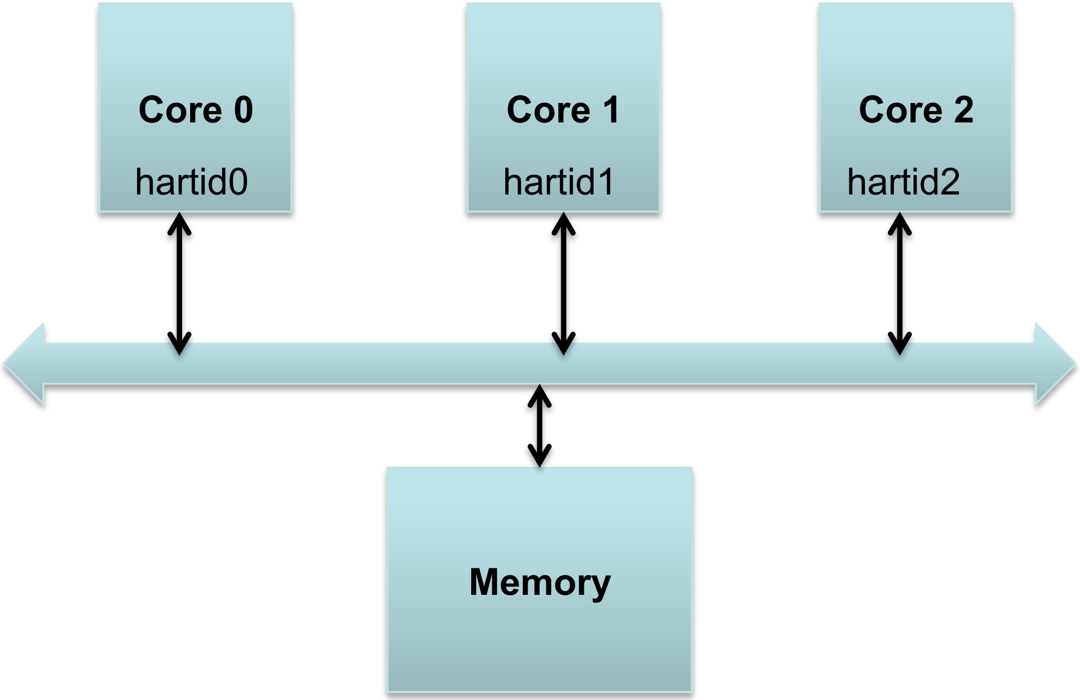
xv6 is made for a multicore arch, but typically is init for 3 cores

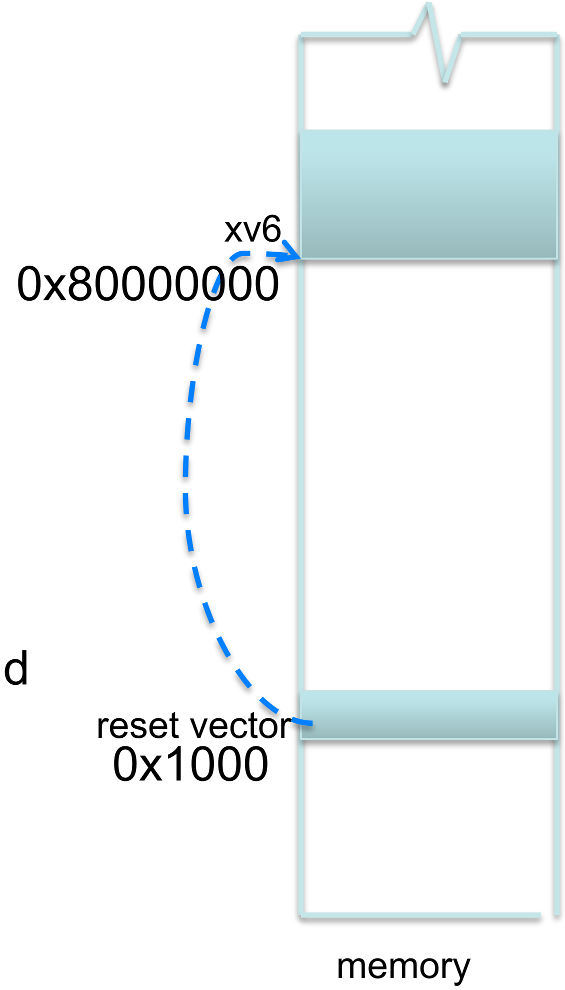
each of the cores have an unique hart id

at power on, all the core start executing the exact same code almost simulataneously

- Memory Symmetry
  - All processors in the system share the same memory space
  - Advantage : Common operating system code
- I/O Symmetry
  - All processors share the same I/O subsystem
  - Every processor can receive interrupt from any I/O device

Powering Up : xv6 on qemu



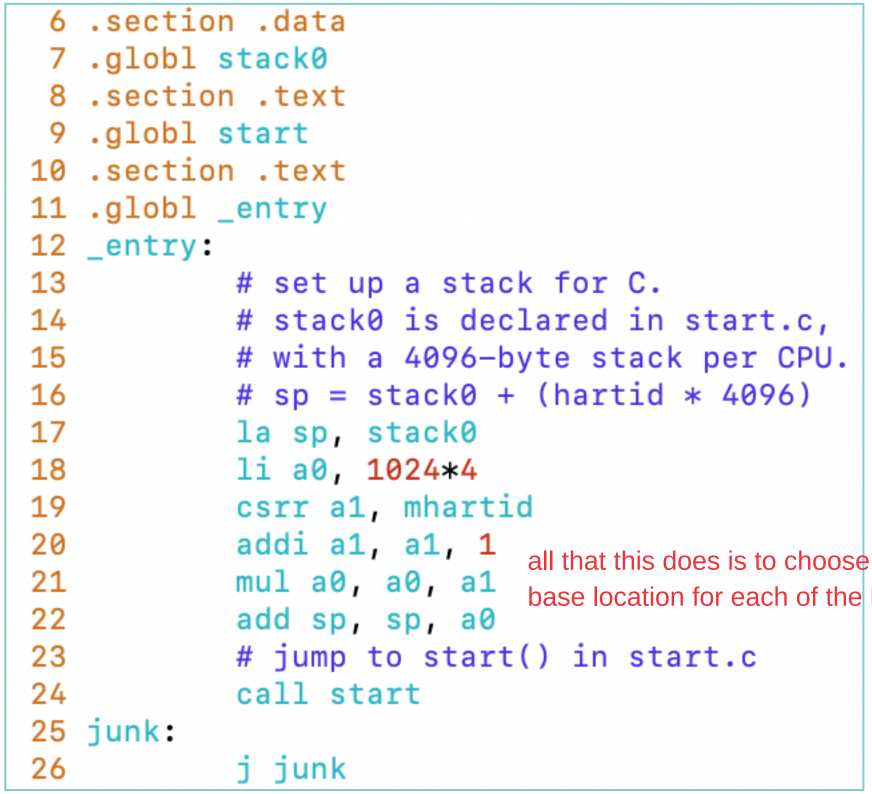


`csrr a1, mhartid` ; instruction that reads the hartid

All cores start the same way with IP=0x1000, and then jumping to 0x80000000

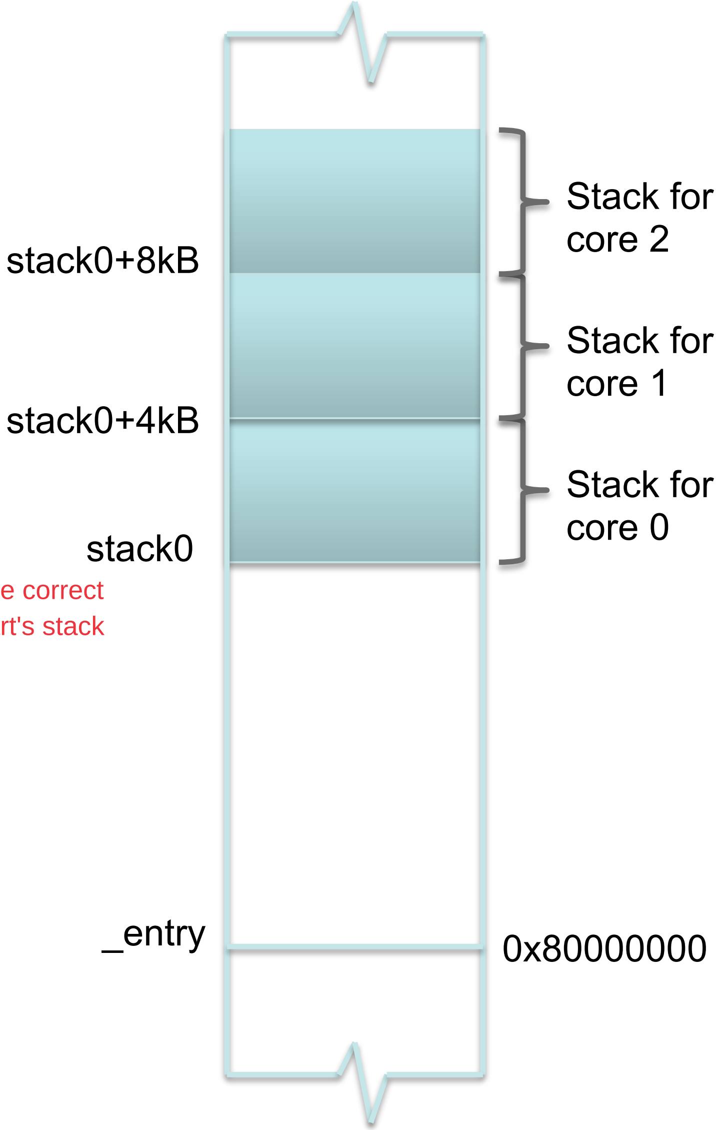
xv6-boot

entry.S



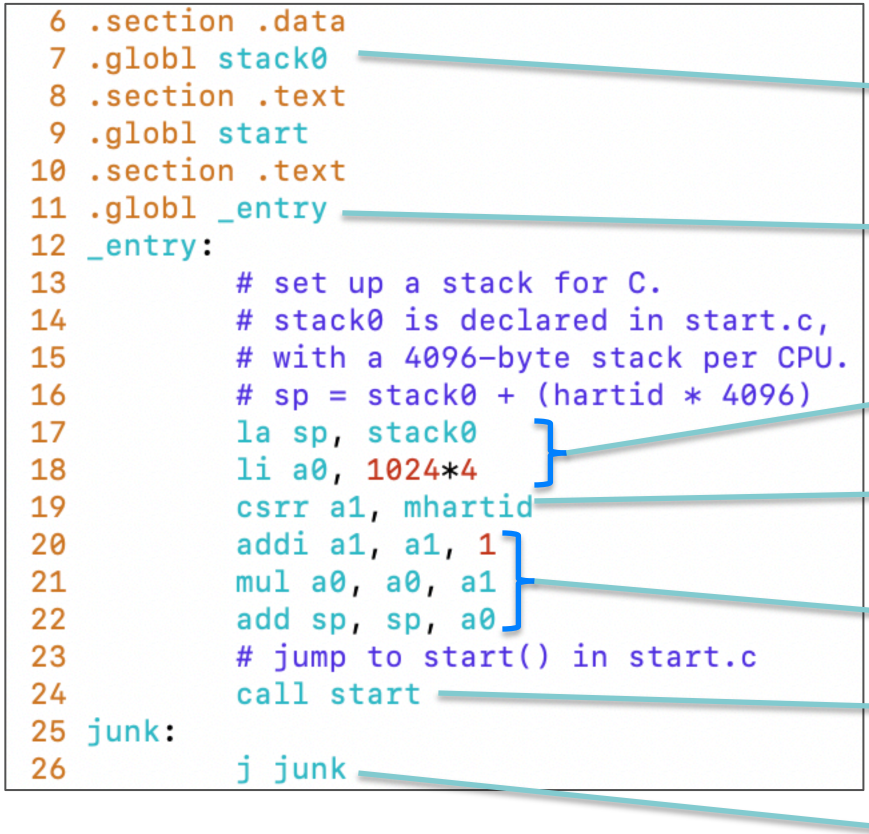
```
.section .data
.globl stack0
.section .text
.globl start
.section .text
.globl _entry
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0
        li a0, 1024*4
        csrr a1, mhartid
        addi a1, a1, 1
        mul a0, a0, a1
        add sp, sp, a0
        # jump to start() in start.c
        call start
junk:
        j junk
```

all that this does is to choose the correct base location for each of the hart's stack



xv6-boot

entry.S



```
.section .data
.globl stack0
.section .text
.globl start
.section .text
.globl _entry
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0
        li a0, 1024*4
        csrr a1, mhartid
        addi a1, a1, 1
        mul a0, a0, a1
        add sp, sp, a0
        # jump to start() in start.c
        call start
junk:
        j junk
```

define label for stack0 (something like extern)
actual stack0 defined in start.c

_entry → will reside at 0x80000000

set sp to stack0 and a0 to 4KB

read the hartid for the processor core using
the corresponding CSR

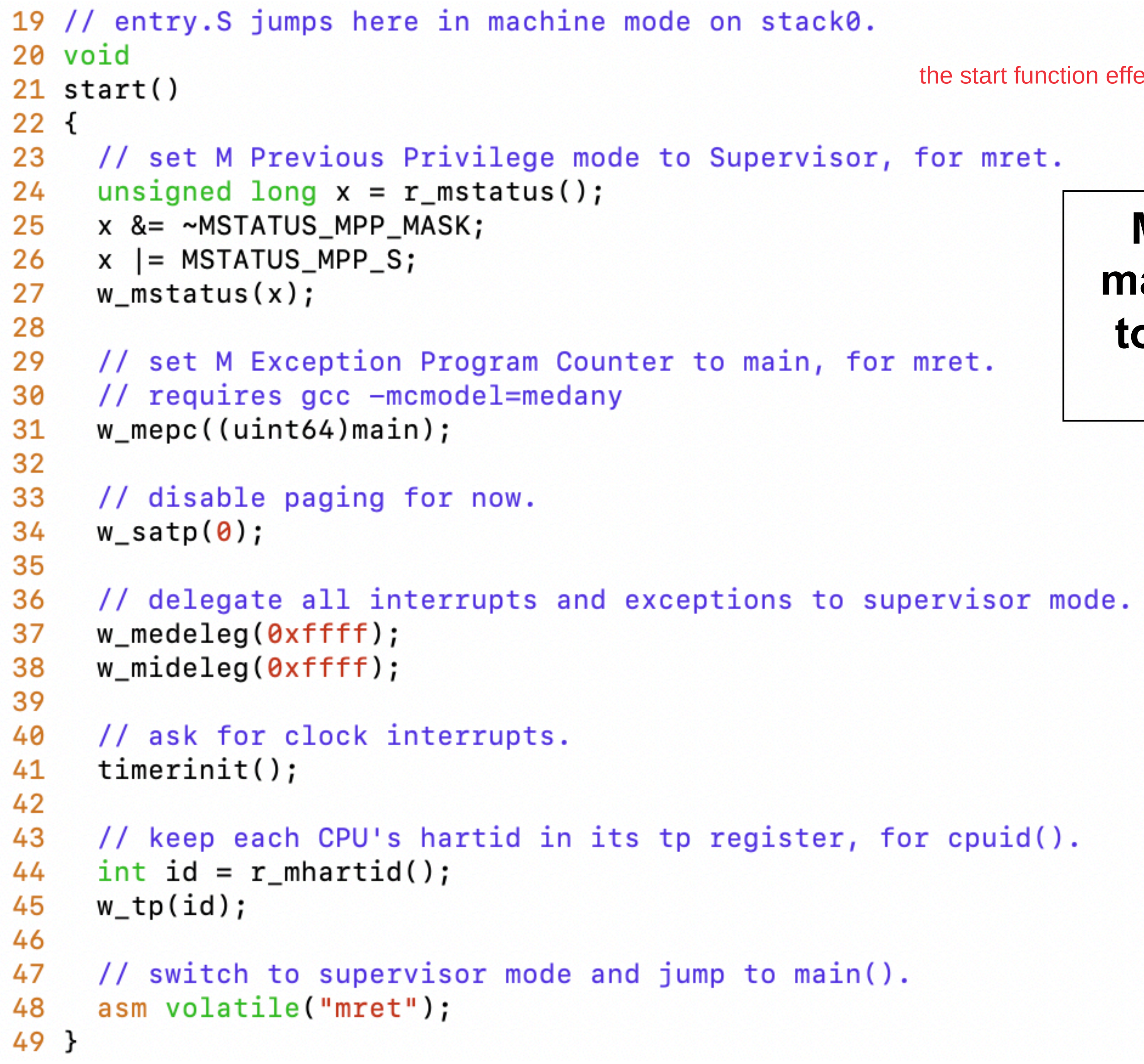
set sp appropriately for the core
why increment a1?

call the C code (present in start.c)

Unreachable code

xv6-boot

start.c



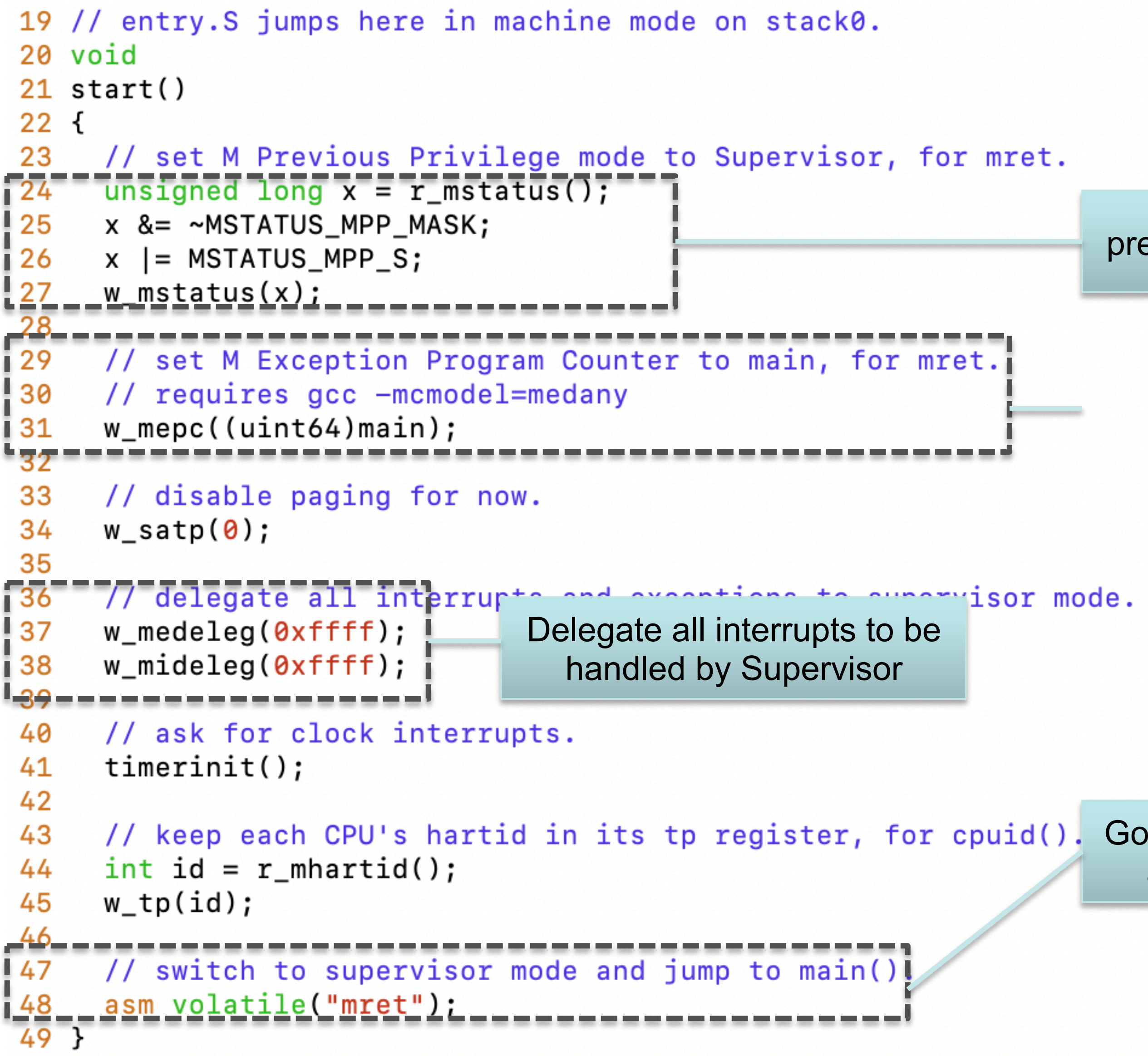
```c
// entry.S jumps here in machine mode on stack0.
void
start()
{
  // set M Previous Privilege mode to Supervisor, for mret.
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK;
  x |= MSTATUS_MPP_S;
  w_mstatus(x);

  // set M Exception Program Counter to main, for mret.
  // requires gcc -mcmodel=medany
  w_mepc((uint64)main);

  // disable paging for now.
  w_satp(0);

  // delegate all interrupts and exceptions to supervisor mode.
  w_medeleg(0xffff);
  w_mideleg(0xffff);

  // ask for clock interrupts.
  timerinit();

  // keep each CPU's hartid in its tp register, for cpuid().
  int id = r_mhartid();
  w_tp(id);

  // switch to supervisor mode and jump to main().
  asm volatile("mret");
}
```

the start function effectively changes from machine mode to sv mode

Moves from machine mode to supervisor mode.

start.c

xv6-boot



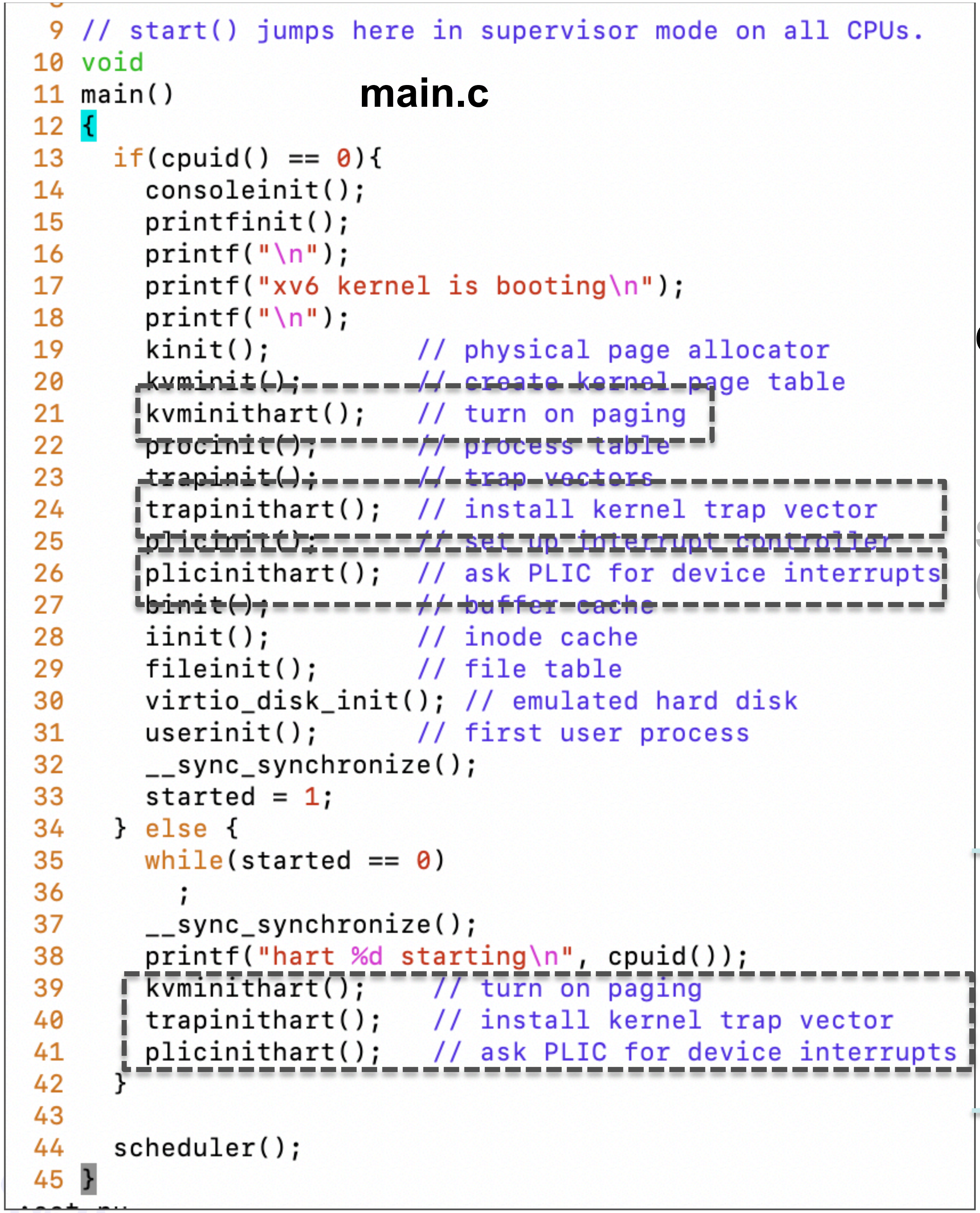
```c
// entry.S jumps here in machine mode on stack0.
void
start()
{
  // set M Previous Privilege mode to Supervisor, for mret.
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK;
  x |= MSTATUS_MPP_S;
  w_mstatus(x);

  // set M Exception Program Counter to main, for mret.
  // requires gcc -mcmodel=medany
  w_mepc((uint64)main);

  // disable paging for now.
  w_satp(0);

  // delegate all interrupts and exceptions to supervisor mode.
  w_medeleg(0xffff);
  w_mideleg(0xffff);

  // ask for clock interrupts.
  timerinit();

  // keep each CPU's hartid in its tp register, for cpuid().
  int id = r_mhartid();
  w_tp(id);

  // switch to supervisor mode and jump to main().
  asm volatile("mret");
}
```

previous mode = supervisor

Delegate all interrupts to be handled by Supervisor

Goto “previous mode” at the address “previous PC”

xv6-boot



main.c

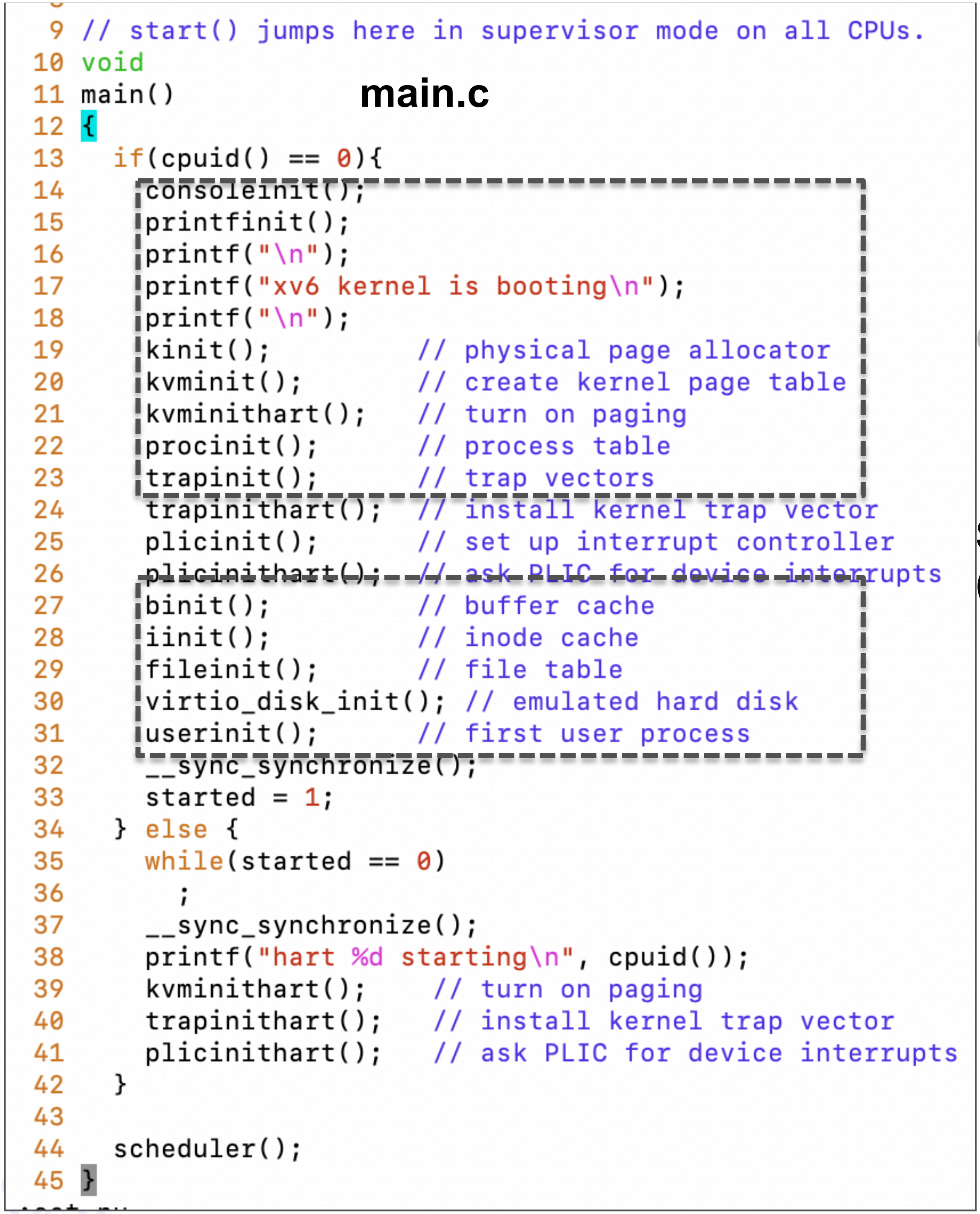
```
// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode cache
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

Core specific initialization (done by all cores)

System specific initialization (done only by core 0)

Executed by all cores except core0 after started=1

xv6-boot



main.c

```
// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode cache
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

Core specific initialization
(done by all cores)

**System specific initialization
(done only by core 0)**