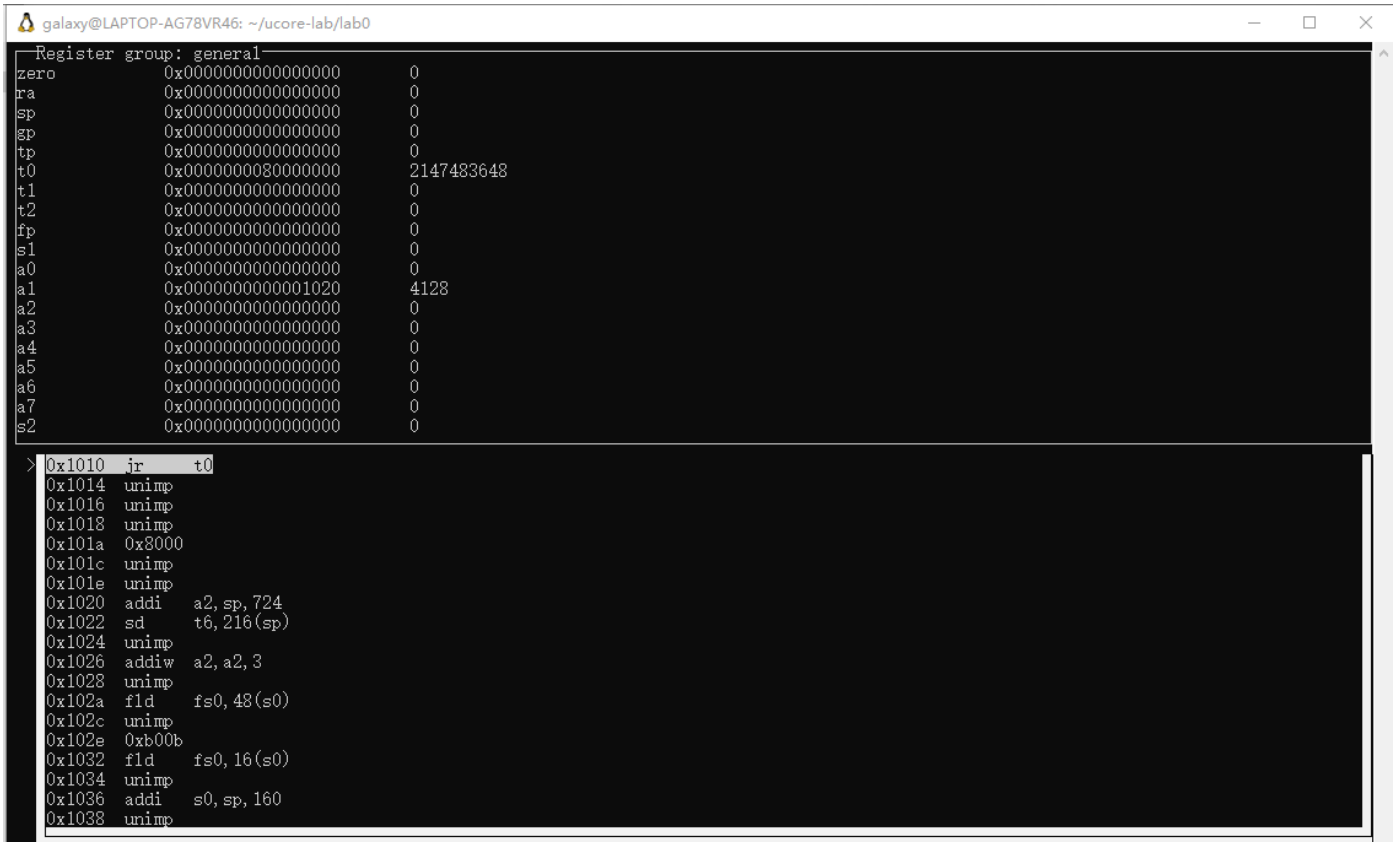
ucore启动过程

1.从复位代码处开始执行

- 主板上电后，首先执行复位代码。在QEMU模拟的这款risc-v处理器中，将复位向量地址初始化为0x1000，再将PC初始化为该复位地址，因此处理器将从此处开始执行复位代码，复位代码主要是将计算机系统的各个组件（包括处理器、内存、设备等）置于初始状态，并且会启动Bootloader，0x80000000是链接的起始位置，Bootloader将加载操作系统内核并启动操作系统的执行（但在我们的lab0中操作系统内核已经被qemu加载到指定地址）。

```
0x1000  auipc   t0,0x0
0x1004  addi    a1,t0,32
0x1008  csrr    a0,mhartid
0x100c  ld      t0,24(t0)
0x1010  jr      t0
0x1014  unimp
0x1016  unimp
0x1018  unimp
0x101a  0x8000
0x101c  unimp
0x101e  unimp
0x1020  addi    a2,sp,724
0x1022  sd      t6,216(sp)
0x1024  unimp
0x1026  addiw   a2,a2,3
0x1028  unimp
0x102a  fld     fs0,48(s0)
0x102c  unimp
0x102e  0xb00b
```

- 完成计算机系统各个组件初始化后，准备跳转到bootloader（OpenSBI固件）处执行，此时t0寄存器存储的即为bootloader入口地址



- 以上复位寄存器处代码存储在 `~/qemu-3.1.0/hw/riscv/spike.c` 中（qemu-3.1.0由命名确定）
- `0x100c:  ld  t0,24(t0)` 在t0初始化为0x1000后，t0+24可以访问到地址0x1018，而这个地址在加载初期就已经初始化为0x80000000，所以即使在0x1018位置显示的是unimp，执行 `0x1010:  jr  t0` 依旧可以跳转到地址0x80000000。

```
0x0000000000001000 in ?? ()
(gdb) x/x 0x1018
0x1018: 0x80000000
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      0x8000
   0x101c:      unimp
```

2.OpenSBI

代码跳转到0x80000000处后控制权转交给OpenSBI，此时OpenSBI要经历以下几个阶段

底层初始化阶段

- **判断hard id**：取启动处理器的 ID，并将其保存到寄存器 a6 中，然后检查该处理器是否为启动处理器，如果不是则跳转到等待重定位完成的循环中，否则执行接下来的步骤。

```
0x80000000      csrr    a6,mhartid
0x80000008      auipc   t0,0x0
```

- **代码重定位**：如果未成功获取启动处理器 ID 或者获取到的处理器 ID 不是当前处理器，则执行随机选择重定位目标地址的过程，即尝试从 `_relocate_lottery` 标签处开始循环等待。根据 `_link_start` 和 `_link_end` 标签获取链接地址， `_load_start` 标签获取加载地址，并判断二者是否相同。如果不同，则需要进行重定位操作。在我们的ucore中通过gdb追踪代码如下：

```
0x80000000      csrr    a6,mhartid
0x80000004      bgtz    a6,0x80000108
0x80000008      auipc   t0,0x0
0x8000000c      addi    t0,t0,1032
0x80000010      auipc   t1,0x0
0x80000014      addi    t1,t1,-16
0x80000018      sd      t1,0(t0)
0x8000001c      auipc   t0,0x0
0x80000020      addi    t0,t0,1020
0x80000024      ld      t0,0(t0)
0x80000028      auipc   t1,0x0
0x8000002c      addi    t1,t1,1016
0x80000030      ld      t1,0(t1)
0x80000034      auipc   t2,0x0
0x80000038      addi    t2,t2,988
0x8000003c      ld      t2,0(t2)
0x80000040      sub     t3,t1,t0
0x80000044      add     t3,t3,t2
0x80000046      beq     t0,t2,0x8000014e
```

- **清除寄存器值**：在完成代码重定位后，我们需要清除寄存器值，继续使用gdb追踪代码如下：

```
0x8000014e:  auipc   t0,0x0
0x80000152:  addi    t0,t0,698
0x80000156:  li      t1,1
0x80000158:  sd      t1,0(t0)
0x8000015c:  fence   rw,rw
0x80000160:  li      ra,0
0x80000162:  jal     ra,0x80000550
0x80000166:  add     s0,a0,zero
0x8000016a:  add     s1,a1,zero
0x8000016e:  add     s2,a2,zero
```

> 调用0x80000550处函数进行寄存器初始化

```
0x80000550:  fence.i
0x80000554:  li      sp,0
0x80000556:  li      gp,0
0x80000558:  li      tp,0
0x8000055a:  li      t0,0
0x8000055c:  li      t1,0
0x8000055e:  li      t2,0
0x80000560:  li      s0,0
0x80000562:  li      s1,0
0x80000564:  li      a3,0
```

- **清除bss段**：如果要想c语言执行起来，必须要做的事情有两个，一个是设置sp栈地址，另外就是清除bss段。

```
0x8000017e      add     a0, s0, zero
0x80000182      add     a1, s1, zero
0x80000186      add     a2, s2, zero
0x8000018a      add     a3, s3, zero
0x8000018e      add     a4, s4, zero
0x80000192      auipc   a4, 0x9
0x80000196      addi    a4, a4, 1070
0x8000019a      lwu     s7, 80(a4)
0x8000019e      lwu     s8, 84(a4)
0x800001a2      auipc   tp, 0xc
0x800001a6      addi    tp, tp, -418 # 0xfffffffffffffe5e
0x800001aa      mul     a5, s7, s8
0x800001ae      add     tp, tp, a5
0x800001b0      add     t3, tp, zero
0x800001b4      li      t2, 1
0x800001b6      li      t1, 0
0x800001b8      add     tp, t3, zero
0x800001bc      mul     a5, s8, t1
0x800001c0      sub     tp, tp, a5
```

- **准备 scratch 空间**：从地址0x800001b8开始我们进行初始化 `struct sbi_scratch` 结构体，这个结构体将会传递给 `sbi_init()` 函数。（篇幅原因仅列出部分代码）

```
 _scratch_init:
/*

+ The following registers hold values that are computed before
+ entering this block, and should remain unchanged.
*
+ t3 -> the firmware end address
+ s7 -> HART count
+ s8 -> HART stack size
+ s9 -> Heap Size
+ s10 -> Heap Offset
*/
add tp, t3, zero
sub tp, tp, s9
mul a5, s8, t1
sub tp, tp, a5
li a5, SBI_SCRATCH_SIZE
sub tp, tp, a5

/*Initialize scratch space*/
/*Store fw_start and fw_size in scratch space*/
lla a4, _fw_start
sub a5, t3, a4
REG_S a4, SBI_SCRATCH_FW_START_OFFSET(tp)
REG_S a5, SBI_SCRATCH_FW_SIZE_OFFSET(tp)

/*Store R/W section's offset in scratch space*/
lla a5, _fw_rw_start
sub a5, a5, a4
REG_S a5, SBI_SCRATCH_FW_RW_OFFSET(tp)

/*Store fw_heap_offset and fw_heap_size in scratch space*/
REG_S s10, SBI_SCRATCH_FW_HEAP_OFFSET(tp)
REG_S s9, SBI_SCRATCH_FW_HEAP_SIZE_OFFSET(tp)

/*Store next arg1 in scratch space*/
MOV_3R s0, a0, s1, a1, s2, a2
call fw_next_arg1
REG_S a0, SBI_SCRATCH_NEXT_ARG1_OFFSET(tp)
MOV_3R a0, s0, a1, s1, a2, s2
/*Store next address in scratch space*/
MOV_3R s0, a0, s1, a1, s2, a2
call fw_next_addr
REG_S a0, SBI_SCRATCH_NEXT_ADDR_OFFSET(tp)
MOV_3R a0, s0, a1, s1, a2, s2
/*Store next mode in scratch space*/
MOV_3R s0, a0, s1, a1, s2, a2
call fw_next_mode
REG_S a0, SBI_SCRATCH_NEXT_MODE_OFFSET(tp)
MOV_3R a0, s0, a1, s1, a2, s2
```

```
/*Store warm_boot address in scratch space*/
lla a4, _start_warm
REG_S a4, SBI_SCRATCH_WARMBOOT_ADDR_OFFSET(tp)
/* Store platform address in scratch space */
lla a4, platform
REG_S a4, SBI_SCRATCH_PLATFORM_ADDR_OFFSET(tp)
/* Store hartid-to-scratch function address in scratch space */
lla a4,_hartid_to_scratch
REG_S a4, SBI_SCRATCH_HARTID_TO_SCRATCH_OFFSET(tp)
/*Store trap-exit function address in scratch space*/
lla a4, _trap_exit
REG_S a4, SBI_SCRATCH_TRAP_EXIT_OFFSET(tp)
/* Clear tmp0 in scratch space */
REG_S zero, SBI_SCRATCH_TMP0_OFFSET(tp)
/* Store firmware options in scratch space */
MOV_3R s0, a0, s1, a1, s2, a2
```

- **读取设备树（Flattend Device Tree, FDT）**：首先，通过前一个启动阶段传递过来的参数 a0、a1 和 a2，保存了当前 FDT 的源地址指针。接着，通过调用函数 fw_next_arg1()获取下一个启动阶段传递过来的参数 a1，即将被重定位到的 FDT 的目标地址指针。如果 a1 为 0 或者 a1 等于当前 FDT 的源地址指针，则说明不需要进行重定位，直接跳转到_fdt_reloc_done 标签处。如果需要进行重定位，则需要计算出源 FDT 的大小，并将其从源地址拷贝到目标地址，完成重定位。具体操作如下：
  1. 首先，将目标地址按照指针大小对齐，并保存为 t1。
  2. 然后，从源地址中读取 FDT 大小，该大小为大端格式，需要将其拆分为四个字节：bit[31:24]、bit[23:16]、bit[15:8]和 bit[7:0]，并组合成小端格式，保存在 t2 寄存器中。
  3. 接着，将 t1 加上 t2，得到目标 FDT 的结束地址，保存在 t2 寄存器中。这样就确定了拷贝数据的范围。
  4. 最后，循环拷贝数据，将源 FDT 中的数据拷贝到目标 FDT 中。循环次数为源 FDT 大小除以指针大小，即源 FDT 中包含的指针数量。

```
_fdt_reloc_again:
REG_L t3, 0(t0)
REG_S t3, 0(t1)
add t0, t0, **SIZEOF_POINTER**
add t1, t1, **SIZEOF_POINTER**
blt t1, t2,_fdt_reloc_again
_fdt_reloc_done:

/*mark boot hart done*/
li t0, BOOT_STATUS_BOOT_HART_DONE
lla t1, _boot_status
REG_S t0, 0(t1)
fence rw, rw
j _start_warm
```

完成拷贝后，将 BOOT_STATUS_BOOT_HART_DONE 保存到_boot_status 寄存器中，表示当前处理器已经完成启动。最后，通过调用_start_warm 跳转到下一步操作。

- **_start_warm**：在初始化过程中，需要禁用和清除所有中断，并设置当前处理器的栈指针和 `trap handler` （异常处理函数）。使用gdb具体追踪，代码如下：

```
0x8000039e:  li      ra,0
0x800003a0:  jal     ra,0x80000550
0x800003a4:  csrw    mie,zero
0x800003a8:  csrw    mip,zero
0x800003ac:  auipc   a4,0x9
0x800003b0:  addi    a4,a4,532
0x800003b4:  lwu     s7,80(a4)
0x800003b8:  lwu     s8,84(a4)
0x800003bc:  csrr    s6,mhartid
0x800003c0:  ble     s7,s6,0x80000468
```

具体执行过程如下：

1. 首先，调用 `_reset_regs` 函数，将寄存器状态重置为 0，以保证非引导处理器使用前的状态干净、一致。
2. 接着，禁用和清空所有中断，即将 CSR_MIE 和 CSR_MIP 寄存器都设置为 0。
3. 获取 platform 变量的地址，并读取平台配置信息，包括处理器数量（s7）和每个处理器的栈大小（s8）。
4. 获取当前处理器的 ID（s6），并判断其是否超出了处理器数量的范围。如果超出，则跳转到 _start_hang 标签，表示出现了错误。
5. 计算当前处理器对应的 scratch space 的地址，并将其保存到 CSR_MSCRATCH 寄存器中，作为 SBI 运行时的全局变量。
6. 将 scratch space 地址保存到 SP 寄存器中，作为当前处理器的栈指针。
7. 设置 trap handler 为 `_trap_handler` 函数，即当发生异常时会跳转到该函数进行处理。同时，读取 MTVEC 寄存器的值确保 trap handler 已经设置成功。
8. 调用 `sbi_init` 函数进行 SBI 运行时的初始化。 `sbi_init` 函数将会初始化各种全局变量、锁、Hart Table 等

```
0x800003e6      add     sp, tp, zero
0x800003ea      auipc   a4, 0x0
0x800003ee      addi    a4, a4, 134
0x800003f2      csrw    mtvec, a4
0x800003f6      csrr    a5, mtvec
0x800003fa      bne     a4, a5, 0x800003f6
0x800003fe      csrr    a0, mscratch
0x80000402      jal     ra, 0x800005e2
0x80000406      j       0x80000468
0x80000408      c.slli  zero, 0x0
0x8000040a      unimp
0x8000040c      unimp
0x8000040e      unimp
0x80000410      unimp
0x80000412      0x8000
0x80000414      unimp
0x80000416      unimp
0x80000418      unimp
0x8000041a      0x8000
```

9. 最后，通过跳转到_start_hang 标签等待处理器发生异常或被重置。

- **跳转至设备初始化**

  在一系列的初始化之后，需要进行跳转进行内核的加载。此时，可以分析一下程序是在哪里进行跳转的。

  首先猜想是直接通过地址跳转，所以将0x8000000开始的代码进行检查。以下为部分代码：

```
0x80008762:  ld      ra,40(sp)
0x80008764:  ld      s0,32(sp)
0x80008766:  mv      a0,s2
0x80008768:  ld      s1,24(sp)
0x8000876a:  ld      s2,16(sp)
0x8000876c:  ld      s3,8(sp)
0x8000876e:  ld      s4,0(sp)
0x80008770:  addi    sp,sp,48
0x80008772:  ret
0x80008774:  li      s2,-3
```

  在文档内直接搜索，可以发现代码中并没有直接跳转至0x80200000，于是推测是通过寄存器进行跳转。

```
469
470   0x800005aa: nop
471   0x800005ac: nop
472   0x800005b0: auipc a0,0x0
473   0x800005b4: addi a0,a0,32
474   0x800005b8: ld a0,0(a0)
475   0x800005ba: ret
476
477   0x800005bc: nop
478   0x800005c0: li a0,1
479   0x800005c2: ret
```

80200000 Aa ab .* 无结果

在刚进入gdb后，在0x80200000地址处进行断点，然后直接continue

```
(gdb) break *0x80200000
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) continue
Continuing.
```

在跳转后可以发现，下一步就是<kern_entry>

```
(gdb) x/10i $pc
=> 0x80200000 <kern_entry>:     auipc   sp,0x3
   0x80200004 <kern_entry+4>:   mv      sp,sp
   0x80200008 <kern_entry+8>:   j       0x8020000c <kern_init>
   0x8020000c <kern_init>:      auipc   a0,0x3
   0x80200010 <kern_init+4>:    addi    a0,a0,-4
   0x80200014 <kern_init+8>:    auipc   a2,0x3
   0x80200018 <kern_init+12>:   addi    a2,a2,-12
   0x8020001c <kern_init+16>:   addi    sp,sp,-16
   0x8020001e <kern_init+18>:   li      a1,0
   0x80200020 <kern_init+20>:   sub     a2,a2,a0
```

通过 `info r` ，查看此时的寄存器存储情况

```
(gdb) info r
ra             0x0000000080000a02       2147486210
sp             0x000000008001bd80       2147597696
gp             0x0000000000000000       0
tp             0x000000008001be00       2147597824
t0             0x0000000080200000       2149580800
t1             0x0000000000000001       1
t2             0x0000000000000001       1
fp             0x000000008001bd90       2147597712
s1             0x000000008001be00       2147597824
a0             0x0000000000000000       0
a1             0x0000000082200000       2183135232
a2             0x0000000080200000       2149580800
a3             0x0000000000000001       1
a4             0x0000000000000800       2048
a5             0x0000000000000001       1
a6             0x0000000082200000       2183135232
a7             0x0000000080200000       2149580800
s2             0x00000000800095c0       2147521984
s3             0x0000000000000000       0
s4             0x0000000000000000       0
s5             0x0000000000000000       0
```

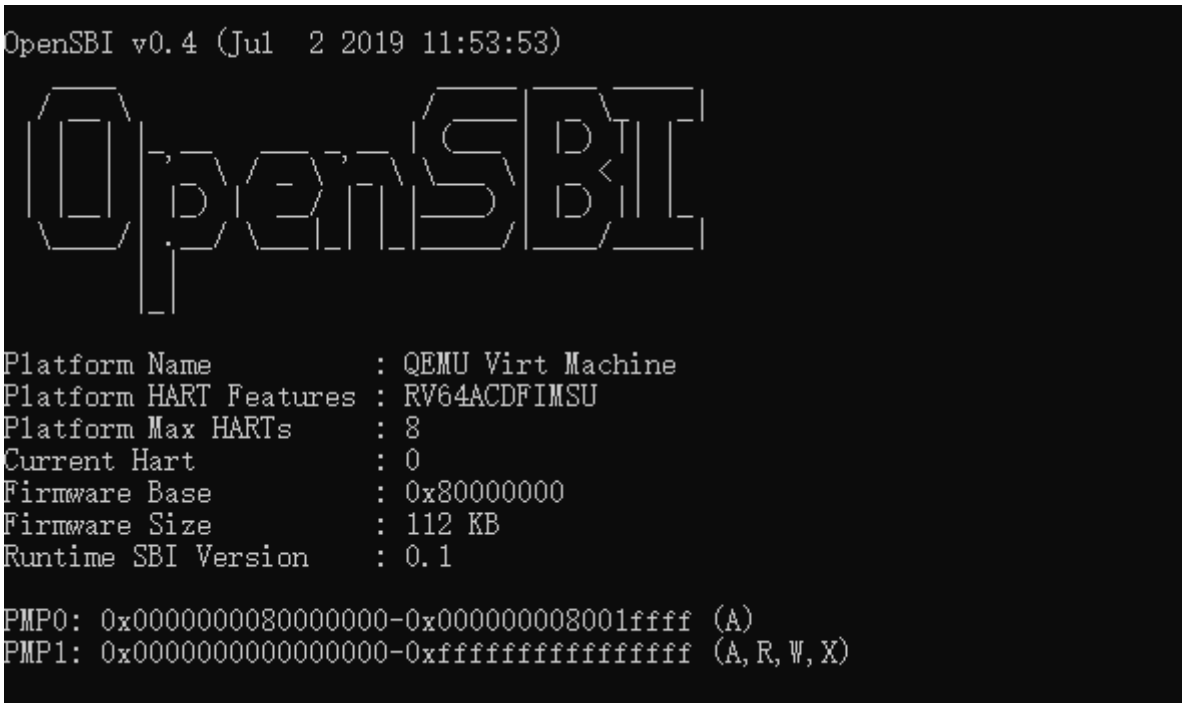
观察可以发现，寄存器t0、a2、a7存储的值为0x80200000

返回代码处进行搜索回溯，再配合在gdb的多次调试，最终发现是在运行完0x80005036后跳转到了0x80200000

```
(gdb) x/10i $pc
=> 0x80005036:  mret
   0x8000503a:  addi    sp,sp,-32
   0x8000503c:  sd      ra,24(sp)
   0x8000503e:  sd      s0,16(sp)
   0x80005040:  sd      s1,8(sp)
   0x80005042:  addi    s0,sp,32
   0x80005044:  mv      s1,a0
   0x80005046:  auipc   a0,0x6
   0x8000504a:  addi    a0,a0,10
   0x8000504e:  jal     ra,0x80003244
(gdb) si
kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
(gdb) x/10i $pc
=> 0x80200000 <kern_entry>:     auipc   sp,0x3
   0x80200004 <kern_entry+4>:   mv      sp,sp
   0x80200008 <kern_entry+8>:   j       0x8020000c <kern_init>
   0x8020000c <kern_init>:      auipc   a0,0x3
   0x80200010 <kern_init+4>:    addi    a0,a0,-4
   0x80200014 <kern_init+8>:    auipc   a2,0x3
   0x80200018 <kern_init+12>:   addi    a2,a2,-12
   0x8020001c <kern_init+16>:   addi    sp,sp,-16
   0x8020001e <kern_init+18>:   li      a1,0
   0x80200020 <kern_init+20>:   sub     a2,a2,a0
```

OpenSBI设备初始化

在前面的过程中opensbi已经基本启动完成，接下来主要进行设备初始化，然后将控制权交给操作系统内核


OpenSBI v0.4 (Jul 2 2019 11:53:53)
Platform Name   : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs   : 8
Current Hart   : 0
Firmware Base   : 0x80000000
Firmware Size   : 112 KB
Runtime SBI Version   : 0.1
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)