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
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)
<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>
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)
<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>

*RISC-V Calling Convention:*
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)
<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>

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;
  . . .

<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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Mode</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0F12</td>
<td>MRO</td>
<td>marchid</td>
<td>Architecture ID.</td>
</tr>
<tr>
<td>0x0F13</td>
<td>MRO</td>
<td>mimpid</td>
<td>Implementation ID.</td>
</tr>
<tr>
<td>0x0F14</td>
<td>MRO</td>
<td>mhartid</td>
<td>Hardware thread ID.</td>
</tr>
</tbody>
</table>

### Machine Trap Setup

<table>
<thead>
<tr>
<th>Address</th>
<th>Mode</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
<td>MRW</td>
<td>mstatus</td>
<td>Machine status register.</td>
</tr>
<tr>
<td>0x0301</td>
<td>MRW</td>
<td>misa</td>
<td>ISA and extensions</td>
</tr>
<tr>
<td>0x0302</td>
<td>MRW</td>
<td>medeleg</td>
<td>Machine exception delegation register.</td>
</tr>
<tr>
<td>0x0303</td>
<td>MRW</td>
<td>midleg</td>
<td>Machine interrupt delegation register.</td>
</tr>
<tr>
<td>0x0304</td>
<td>MRW</td>
<td>mie</td>
<td>Machine interrupt-enable register.</td>
</tr>
<tr>
<td>0x0305</td>
<td>MRW</td>
<td>mtvec</td>
<td>Machine trap-handler base address.</td>
</tr>
<tr>
<td>0x0306</td>
<td>MRW</td>
<td>mcounteren</td>
<td>Machine counter enable.</td>
</tr>
</tbody>
</table>

### Machine Trap Handling

<table>
<thead>
<tr>
<th>Address</th>
<th>Mode</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0340</td>
<td>MRW</td>
<td>mscratch</td>
<td>Scratch register for machine trap handlers.</td>
</tr>
<tr>
<td>0x0341</td>
<td>MRW</td>
<td>mepc</td>
<td>Machine exception program counter.</td>
</tr>
<tr>
<td>0x0342</td>
<td>MRW</td>
<td>mcause</td>
<td>Machine trap cause.</td>
</tr>
<tr>
<td>0x0343</td>
<td>MRW</td>
<td>mtval</td>
<td>Machine bad address or instruction.</td>
</tr>
<tr>
<td>0x0344</td>
<td>MRW</td>
<td>mip</td>
<td>Machine interrupt pending.</td>
</tr>
</tbody>
</table>

### Machine Protection and Translation

<table>
<thead>
<tr>
<th>Address</th>
<th>Mode</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03A0</td>
<td>MRW</td>
<td>pmpcfg0</td>
<td>Physical memory protection configuration.</td>
</tr>
<tr>
<td>0x03A1</td>
<td>MRW</td>
<td>pmpcfg1</td>
<td>Physical memory protection configuration, RV32 only.</td>
</tr>
<tr>
<td>0x03A2</td>
<td>MRW</td>
<td>pmpcfg2</td>
<td>Physical memory protection configuration.</td>
</tr>
<tr>
<td>0x03A3</td>
<td>MRW</td>
<td>pmpcfg3</td>
<td>Physical memory protection configuration, RV32 only.</td>
</tr>
<tr>
<td>0x03B0</td>
<td>MRW</td>
<td>pmpaddr0</td>
<td>Physical memory protection address register.</td>
</tr>
<tr>
<td>0x03B1</td>
<td>MRW</td>
<td>pmpaddr1</td>
<td>Physical memory protection address register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x03BF</td>
<td>MRW</td>
<td>pmpaddr15</td>
<td>Physical memory protection address register.</td>
</tr>
</tbody>
</table>

Table 2.4: Currently allocated RISC-V machine-level CSR addresses.
## Control and Status Registers (CSRs)

<table>
<thead>
<tr>
<th>Address</th>
<th>Access</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>MRW</td>
<td>mstatus</td>
<td>Machine status register.</td>
</tr>
<tr>
<td>0x01</td>
<td>MRW</td>
<td>misa</td>
<td>ISA and extensions.</td>
</tr>
<tr>
<td>0x02</td>
<td>MRW</td>
<td>mdeleg</td>
<td>Machine exception delegation register.</td>
</tr>
<tr>
<td>0x03</td>
<td>MRW</td>
<td>mideleg</td>
<td>Machine interrupt delegation register.</td>
</tr>
<tr>
<td>0x04</td>
<td>MRW</td>
<td>mie</td>
<td>Machine interrupt-enable register.</td>
</tr>
<tr>
<td>0x05</td>
<td>MRW</td>
<td>mtvec</td>
<td>Machine trap-handler base address.</td>
</tr>
<tr>
<td>0x06</td>
<td>MRW</td>
<td>mcounteren</td>
<td>Machine counter enable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Access</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x30</td>
<td>MRW</td>
<td>mscratch</td>
<td>Scratch register for machine trap handlers.</td>
</tr>
<tr>
<td>0x31</td>
<td>MRW</td>
<td>mepc</td>
<td>Machine exception program counter.</td>
</tr>
<tr>
<td>0x32</td>
<td>MRW</td>
<td>mcause</td>
<td>Machine trap cause.</td>
</tr>
<tr>
<td>0x33</td>
<td>MRW</td>
<td>mtval</td>
<td>Machine bad address or instruction.</td>
</tr>
<tr>
<td>0x34</td>
<td>MRW</td>
<td>mip</td>
<td>Machine interrupt pending.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Access</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3A</td>
<td>MRW</td>
<td>pmpcfg0</td>
<td>Physical memory protection configuration.</td>
</tr>
<tr>
<td>0x3A</td>
<td>MRW</td>
<td>pmpcfg1</td>
<td>Physical memory protection configuration, RV32 only.</td>
</tr>
<tr>
<td>0x3A</td>
<td>MRW</td>
<td>pmpcfg2</td>
<td>Physical memory protection configuration.</td>
</tr>
<tr>
<td>0x3A</td>
<td>MRW</td>
<td>pmpcfg3</td>
<td>Physical memory protection configuration, RV32 only.</td>
</tr>
<tr>
<td>0x3B</td>
<td>MRW</td>
<td>pmpaddr0</td>
<td>Physical memory protection address register.</td>
</tr>
<tr>
<td>0x3B</td>
<td>MRW</td>
<td>pmpaddr1</td>
<td>Physical memory protection address register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3F</td>
<td>MRW</td>
<td>pmpaddr15</td>
<td>Physical memory protection address register.</td>
</tr>
</tbody>
</table>

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()

<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;
    . . .
</main>

<handler>
    Store ALL registers on the handler stack
    . . .
    Restore ALL registers
    mret
</handler>
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; Store current instruction address in mepc
  ...

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Encoding</th>
<th>Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>User/Application</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Supervisor</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Machine</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 1.1: RISC-V privilege levels.
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.

![Machine-mode status register (mstatus) for RV32.](image)

Figure 3.6: Machine-mode status register (mstatus) for RV32.
### 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.
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)](image)

Figure 3.6: Machine-mode status register (mstatus) for RV32.
Returning from interrupt with \texttt{mret}

Machine Previous Privilege (MPP)

- The privilege mode is set to the value encoded in \texttt{mstatus.MPP}.
- The global interrupt enable, \texttt{mstatus.MIE}, is set to the value of \texttt{mstatus.MPIE}.
- The \texttt{pc} is set to the value of \texttt{mepc}.

![Machine-Mode Status Register](image)

Figure 3.6: Machine-mode status register (\texttt{mstatus}) for RV32.
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-\texttt{pmpaddr0}
- Convert physical address to TOR address
  - \(0x1\_0000\_0000\) (4GB) >> 2 == \(0x4000\_0000\)
- Config \texttt{0xF} means TOR, readable, writable, executable.

\[
\text{asm("csrw pmpaddr0, %r0" : : "r" (0x40000000));}
\text{asm("csrw pmpcfg0, %r0" : : "r" (0xF));}
\]
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