CASS Exercise session 5

Operating Systems

Programming bare-metal

- Assume you have ...
  - A simple RISC-V CPU connected to memory
  - A machine to compile RISC-V assembly to bytecode
  - A device to
    - Load your bytecode in memory
    - initialize the RISC-V processor
    - Read values from memory

- Question: can you now write useful programs?

Programming bare-metal

- Yes! All RISC-V programs from previous sessions
    - Possible to execute!

- But… we lack support for
    - Using a hard disk
        - Writing files
    - I/O devices
        - Screen
        - Mouse
        - Keyboard
    - ...

- And we need a different machine to write code?

OS to the rescue!

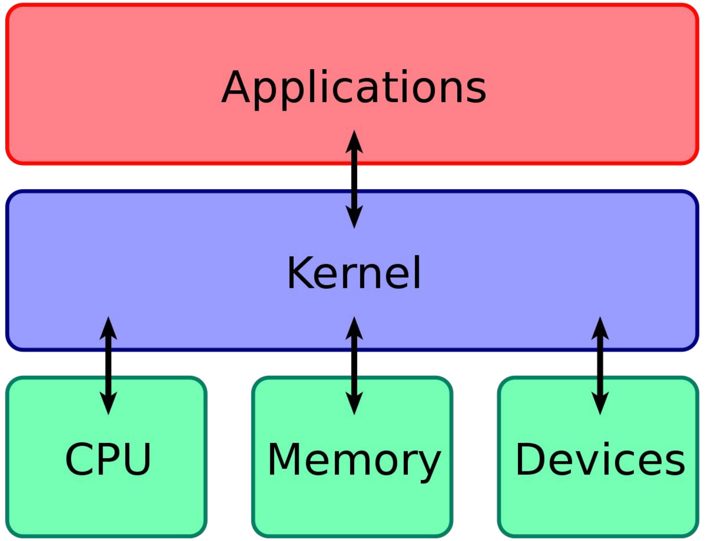
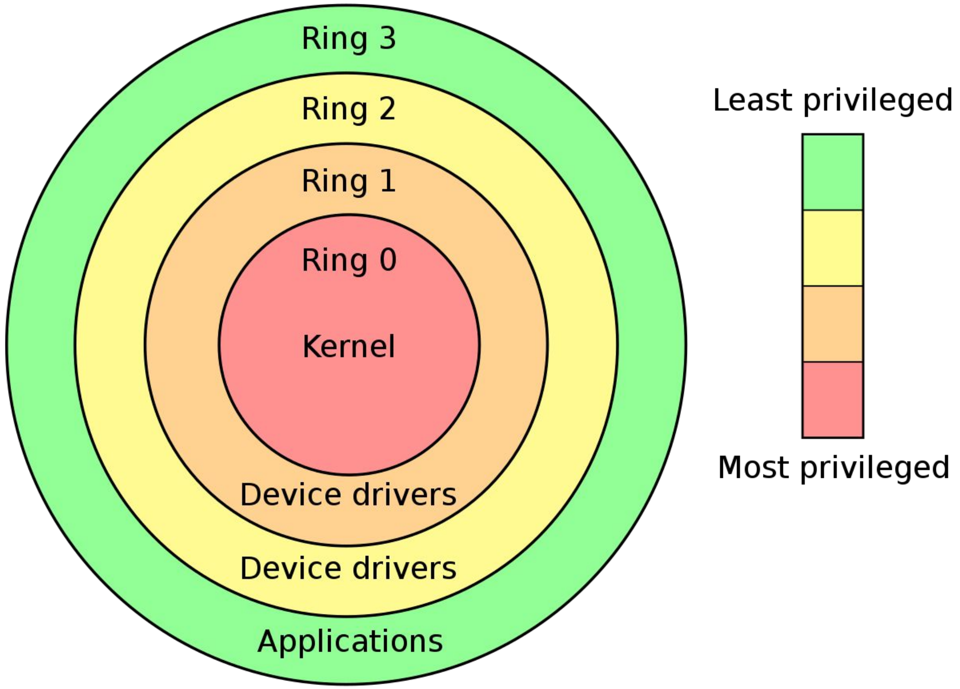


Image source: Wikipedia

How?


Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Device drivers
Device drivers
Applications
Least privileged
Most privileged

Protection Rings on RISC-V

- Three privilege levels
  - Machine mode
    - Allowed execution of any machine instruction
    - Full access to the machine
    - Typically used during boot
    - Only mode that is *required* in RISC-V processor
  - Supervisor mode
    - Allowed execution of *most* instructions
    - Typically used during kernel execution
  - User mode
    - Allowed limited execution of instructions
    - Typically used during user process execution

Operating System

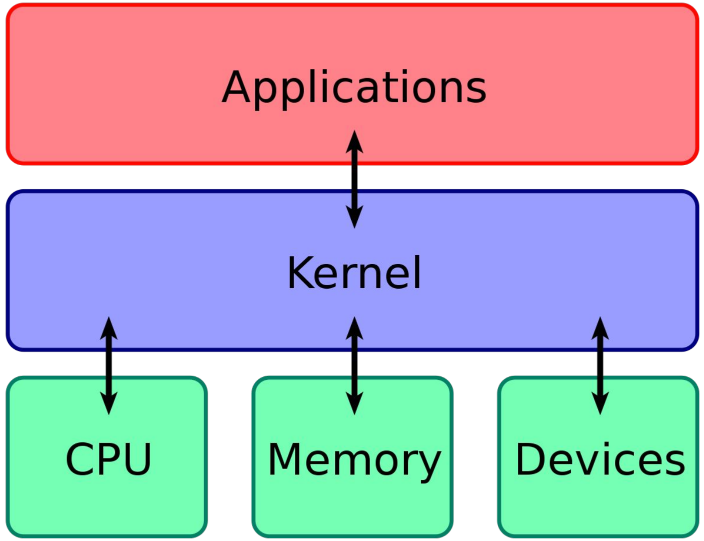


Image source: Wikipedia

- Privileged **kernel space** software offers services to **user space** programs
  - Process abstraction
    - Isolation
    - Scheduling
    - Communication
    - ...
  - Drivers
  - File systems
  - ...

You’ve been using an OS!

- OS of your own machine
  - Windows
  - Ubuntu
  - …

- RARS is not just an emulator
  - Also simulates tiny RISC-V OS
  - Own set of services

Using OS services

- Request a service using a **system call**
  - Also called environment call in RISC-V
  - Similar to a function call
  - Difference? Privilege level change!
  - Full list: https://github.com/TheThirdOne/rars/wiki/Environment-Calls

Name	# (a7)	Description	Inputs	Outputs
PrintInt	1	Prints an integer	a0 = integer to print	N/A
ReadInt	5	Reads an int from input console	N/A	a0 = the int
Sbrk	9	Allocate heap memory	a0 = # bytes	a0 = address of bytes
Exit	10	Exits the program with code 0	N/A	N/A

Example program

system-call.asm

```
.text
.globl main

main:
    li a0, 666
    li a7, 1 #PrintInt
    ecall
```

RARS console

```
666

-- program is finished running (dropped off bottom) --
```

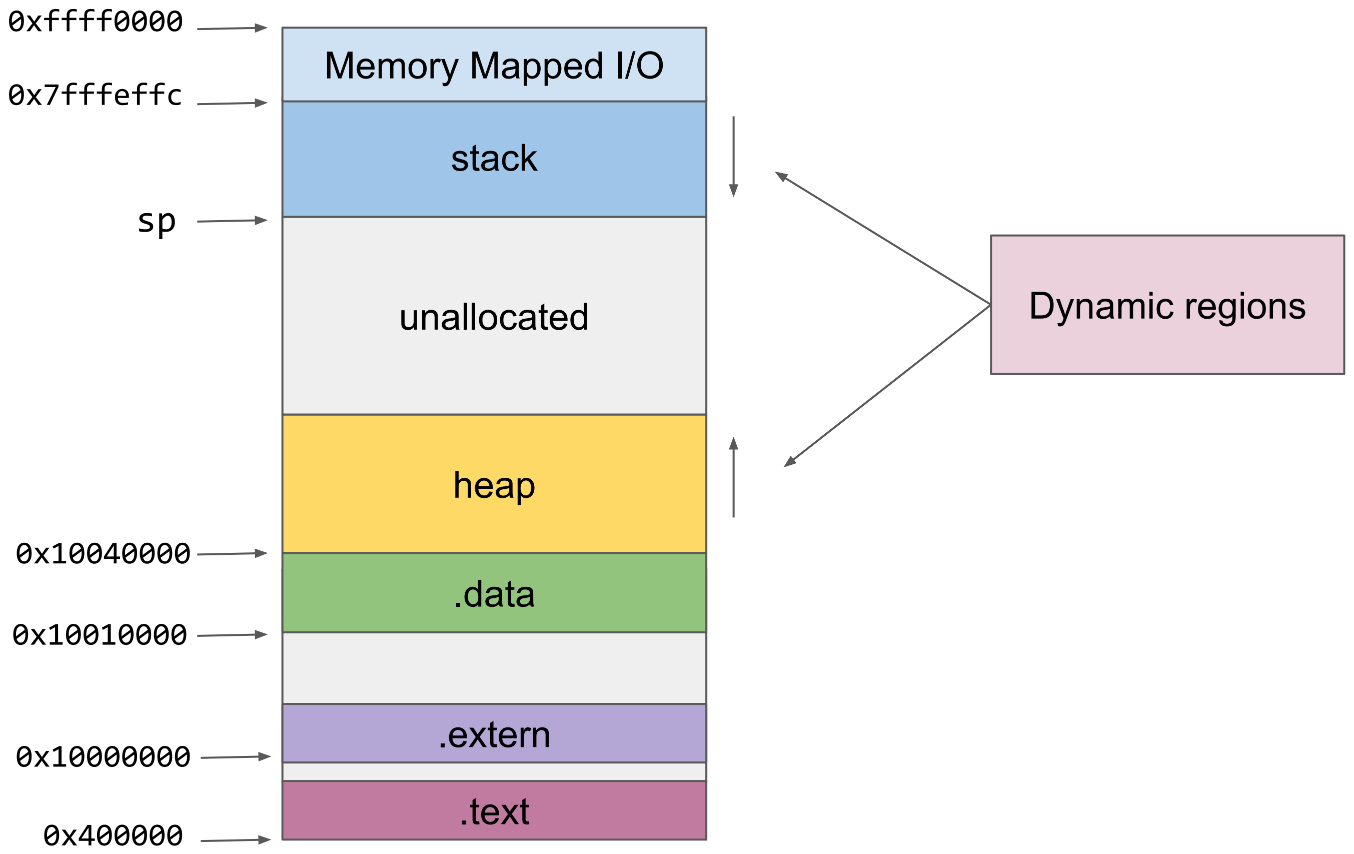
system-call-2.asm

```
.data
str: .string "You entered: "
.text
    li a7, 5  #ReadInt
    ecall
    mv t0, a0
    la a0, str
    li a7, 4  #PrintString
    ecall
    mv a0, t0
    li a7, 1  #PrintInt
    ecall
    li a7, 10 #exit(0)
    ecall
```

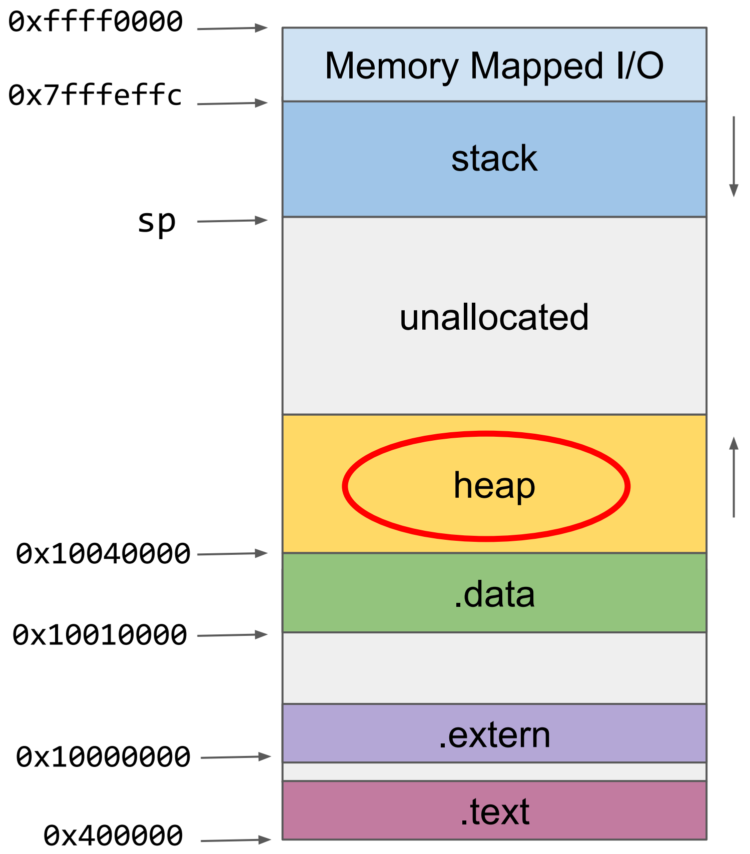
RARS console

```
2
You entered: 2
-- program is finished running (0) --
```

RARS 32bit: process layout



RARS 32bit: process layout



sbrk(n) expands heap region by n bytes

Simple allocator vs sbrk

- Previous session
  - Simple allocator that allocates *n* bytes
  - Predefined space in data section
    - Memory always allocated!
- Sbrk
  - Simple allocator that allocates *n* bytes
  - OS support for a dynamic region
    - OS can allocate more process memory as it grows

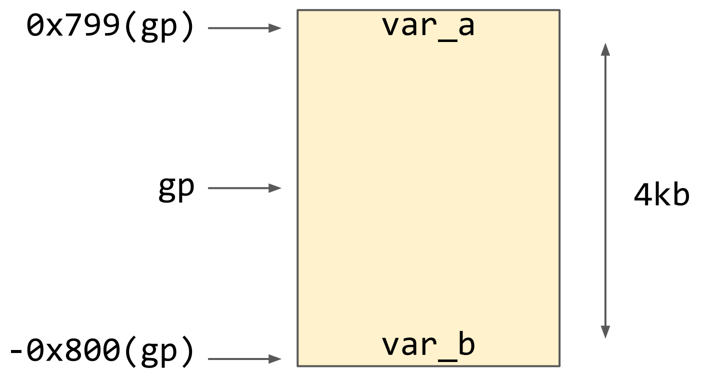
tmp

```
.data
    memory: .space 1000000
```

Intermezzo: gp

- Use of *gp* in previous session
  - Stored heap pointer for our simple allocator
  - But in reality *gp* has a different purpose
  - What to use instead?
    - **Use sbrk syscall**
    - OR save heap pointer in memory

- *gp*
  - Points to the middle of a 4 kb region
  - Used for single-instruction loading
    - Note: lw a0, var_a is pseudo, translated to 2 instructions
  - Relative offsets in range [-0x800, 0x799]
  - Code size optimization!



tmp

```
lw a0,  0x799(gp) #load var a
lw a1, -0x800(gp) #load var b
lw a2,  0x800(gp) #operand out of range
lw a3, -0x801(gp) #operand out of range
```

Malloc

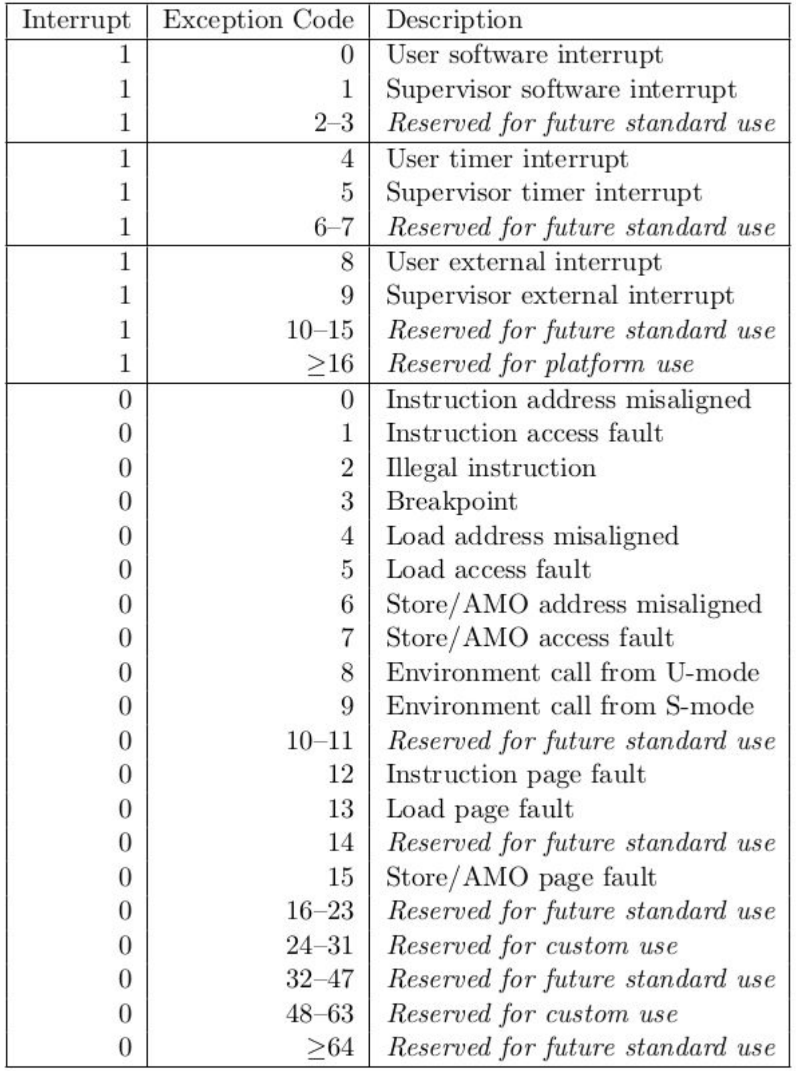
- Calling sbrk for every heap allocation is inefficient
    - System call requires kernel to handle it
    - No free function to reuse memory

- In C: malloc and free
    - First allocation: sbrk(*n*) to create initial heap region
        - Make *n* not too small
    - Malloc allocates bytes from this region to callers
    - Free deallocates bytes
    - Call sbrk(n) again when heap is too small
    - **Not** required on exam
        - High complexity!

Interrupts and exceptions

- Interrupt
  - Request OS to handle special event
    - key press on keyboard
    - timer tick
  - Handle *as soon as possible*
    - Raise **trap**

- Exception
  - Something went “wrong”
  - **Trap** immediately
  - Ecall is a special example of an exception
    - See code 8 and 9



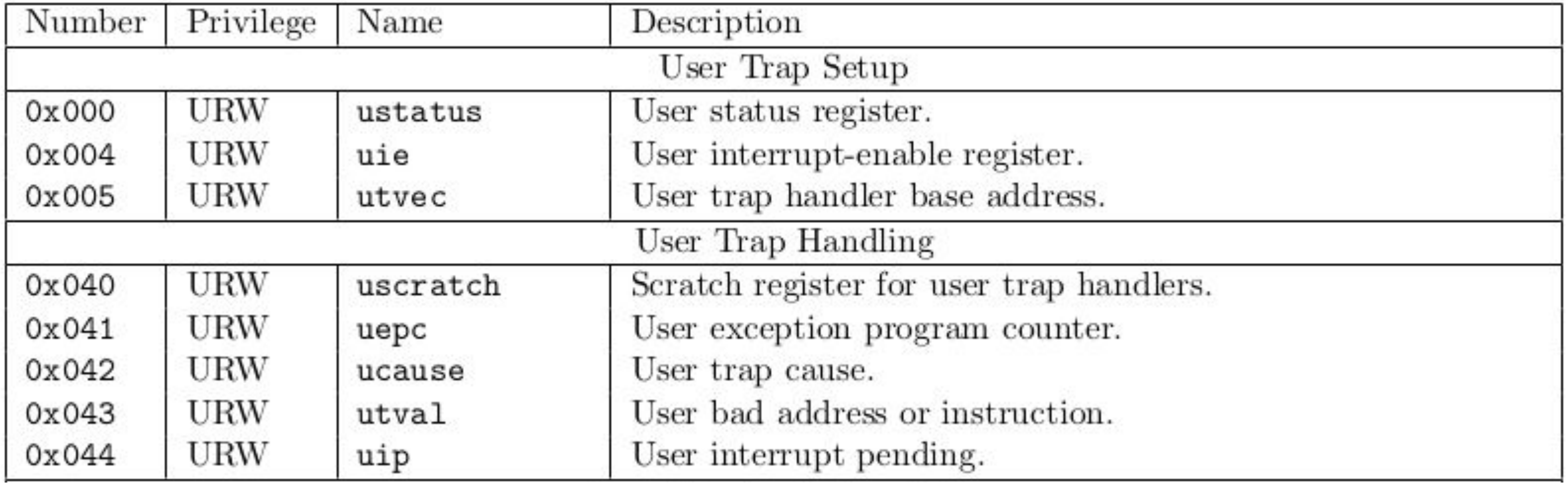
Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved for future standard use*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved for future standard use*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved for future standard use*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Traps in RISC-V

- Trap occurs: jump to a *trap handler*
    - Address of handlers in *vec* registers
        - utvec (user mode)
        - stvec (supervisor mode)
        - mtvec (machine mode)
    - *CSR registers* give info about trap

- Which mode should handle the trap?
    - By default: machine mode BUT...
    - Forward (immediately) using *deleg* registers
        - medeleg -> forward (delegate) exceptions
            - From machine mode to next privilege level
        - sedeleg -> similar (supervisor to user)
        - ...

CSR: Control status register



Number	Privilege	Name	Description
			User Trap Setup
0x000	URW	`ustatus`	User status register.
0x004	URW	`uie`	User interrupt-enable register.
0x005	URW	`utvec`	User trap handler base address.
			User Trap Handling
0x040	URW	`uscratch`	Scratch register for user trap handlers.
0x041	URW	`uepc`	User exception program counter.
0x042	URW	`ucause`	User trap cause.
0x043	URW	`utval`	User bad address or instruction.
0x044	URW	`uip`	User interrupt pending.

Traps in RISC-V

- Summary
  - Interrupts and exceptions raise *traps*
  - *Trap handler* is code at specific location (*vec* register per mode)
    - Needs to “solve” the problem
      - Uses info in CSR registers
    - Are executed in different privilege modes depending on *deleg* regs
  - Ecall causes an exception

- In RARS
  - No access to supervisor exception handler
    - ecalls from user mode are handled there
      - hard to add custom syscall
  - But - we have access to user exception handling!

Trap handler: example

exception.asm

```
.globl main
.text
handler:
      csrrw a0, ucause, zero  # Move ucause to a0, zero to ucause
      la t0, end
      csrrw zero, uepc, t0    # move epc to success and return
      uret
main:
       la t0, handler
       csrrw zero, utvec, t0     # set utvec
       csrrsi zero, ustatus, 1   # set interrupt enable
       lw t0, 1                  # trigger trap
       li a7, 10
       ecall                    # exit (0)
end:
       li a7, 93
       ecall                    # exit (42)
```

RARS console

```
-- program is finished running (4) --
```

Trap handler: example

exception.asm

```
.globl main
.text
handler:
      csrrw a0, ucause, zero  # Move ucause to a0, zero to ucause
      la t0, end
      csrrw zero, uepc, t0    # move epc to success and return
      uret
main:
       la t0, handler
       csrrw zero, utvec, t0      # set utvec
       csrrsi zero, ustatus, 1    # set interrupt enable
       lw t0, 1                   # trigger trap
       li a7, 10
       ecall                     # exit (0)
end:
       li a7, 93
       ecall                     # exit (42)
```

RARS console

```
-- program is finished running (4) --
```

CSR Register usage

csrrw reg1, csr, reg2:
1. Move value in csr to reg1
2. Move value in reg2 to csr

csrrsi reg, csr, imm
1. Move value in csr to reg
2. Store imm in csr

Trap handler: example

exception.asm

```
.globl main
.text
handler:
      csrrw a0, ucause, zero  # Move ucause to a0, zero to ucause
      la t0, end
      csrrw zero, uepc, t0    # move epc to success and return
      uret
main:
      la t0, handler
      csrrw zero, utvec, t0     # set utvec
      csrrsi zero, ustatus, 1   # set interrupt enable
      lw t0, 1                  # trigger trap
      li a7, 10
      ecall                    # exit (0)
end:
      li a7, 93
      ecall                    # exit (42)
```

RARS console

```
-- program is finished running (4) --
```

uret usage

jump to uepc value
(uepc is set to addess where trap occured)