RISC-V Arch Test

Task 5

Test Description:

This test validates the SV32 virtual memory mechanism in RISC-V by verifying that virtual addresses are correctly translated to physical addresses and that page-level permissions (Read, Write, Execute) are enforced.

The test starts in Machine mode (M-mode), sets up the L1 (root) and L0 (leaf) page tables, and configures page table entries (PTEs) with different permissions for instruction and data pages. A trap handler is installed to capture instruction, load, and store page faults via the mcause CSR.

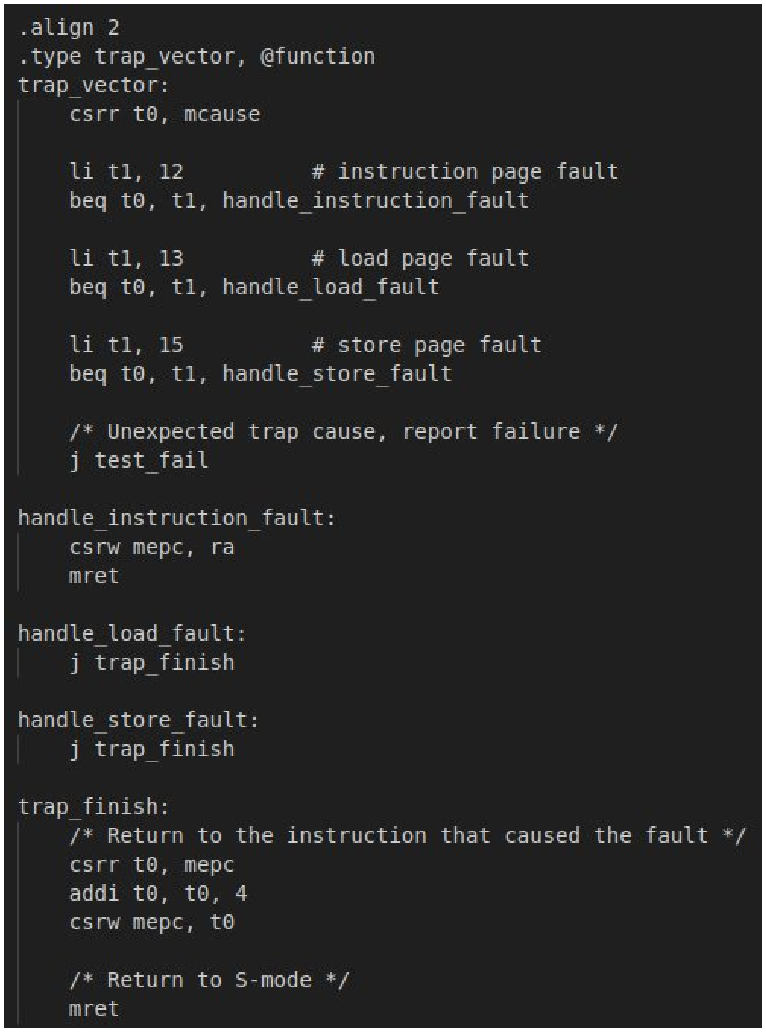
After configuring the satp register to enable SV32 virtual memory, the CPU transitions to Supervisor mode (S-mode) using mret. In S-mode, the test performs the following:

- Executes instructions and performs load/store on pages with RWX permissions (should succeed).
- Changes a page’s permissions to RW only and verifies that instruction execution traps while read/write succeed.
- Changes a page’s permissions to Read-only (R) and verifies that write traps occur, while reads succeed.
- Check test results via the tohost register, indicating pass (0x1) or fail (0x3).

This ensures that SV32 translation, page table hierarchy, and permission enforcement are functioning correctly.

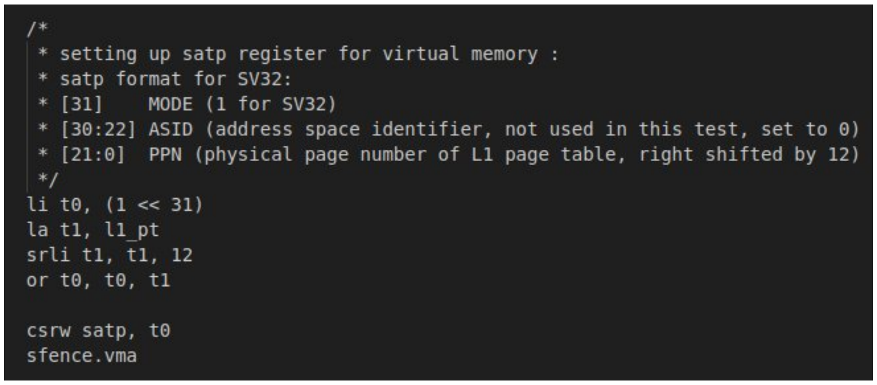
Trap Handler:

Trap handler is defined to handle the instruction page fault, load page fault, and store page fault correctly.



```
.align 2
.type trap_vector, @function
trap_vector:
    csrr t0, mcause

    li t1, 12          # instruction page fault
    beq t0, t1, handle_instruction_fault

    li t1, 13          # load page fault
    beq t0, t1, handle_load_fault

    li t1, 15          # store page fault
    beq t0, t1, handle_store_fault

    /* Unexpected trap cause, report failure */
    j test_fail

handle_instruction_fault:
    csrw mepc, ra
    mret

handle_load_fault:
    j trap_finish

handle_store_fault:
    j trap_finish

trap_finish:
    /* Return to the instruction that caused the fault */
    csrr t0, mepc
    addi t0, t0, 4
    csrw mepc, t0

    /* Return to S-mode */
    mret
```

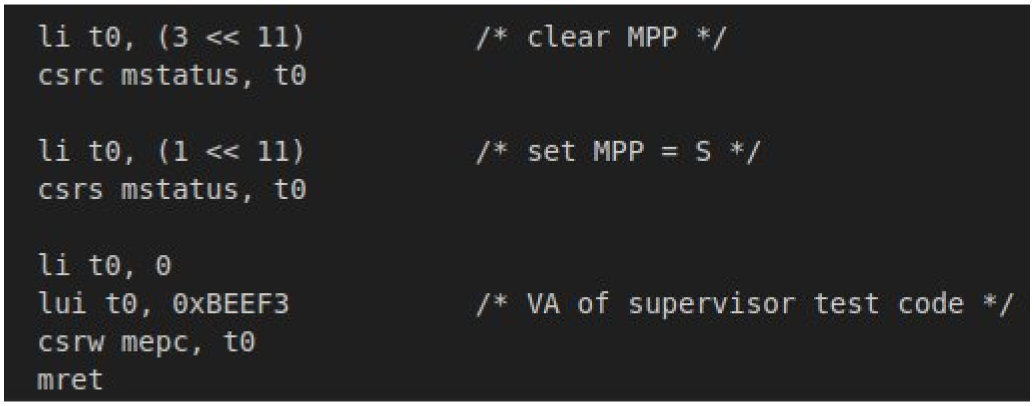
SATP register for SV32:



```
/*
 * setting up satp register for virtual memory :
 * satp format for SV32:
 * [31]    MODE (1 for SV32)
 * [30:22] ASID (address space identifier, not used in this test, set to 0)
 * [21:0]  PPN (physical page number of L1 page table, right shifted by 12)
 */
li t0, (1 << 31)
la t1, l1_pt
srli t1, t1, 12
or t0, t0, t1

csrw satp, t0
sfence.vma
```

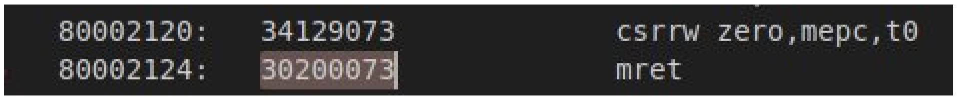
This shows that after the PTEs and trap handler setup, satp is set appropriately to enable virtualization with sv32 translation scheme.

After Satp setup, privilege mode is change to supervisor mode and mret is executed to jump to virtual address of supervisor test code.

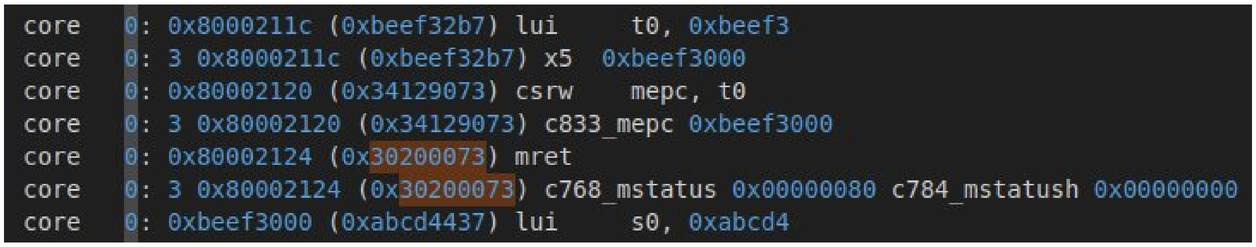


```
li t0, (3 << 11)          /* clear MPP */
csrc mstatus, t0

li t0, (1 << 11)          /* set MPP = S */
csrs mstatus, t0

li t0, 0
lui t0, 0xBEEF3           /* VA of supervisor test code */
csrw mepc, t0
mret
```

Virtual address for supervisor mode test is 0xbeef3000 so it successfully jump to that address.



```
80002120:   34129073            csrrw zero,mepc,t0
80002124:   30200073            mret
```



```
core   0: 0x8000211c (0xbeef32b7) lui     t0, 0xbeef3
core   0: 3 0x8000211c (0xbeef32b7) x5  0xbeef3000
core   0: 0x80002120 (0x34129073) csrw    mepc, t0
core   0: 3 0x80002120 (0x34129073) c833_mepc 0xbeef3000
core   0: 0x80002124 (0x30200073) mret
core   0: 3 0x80002124 (0x30200073) c768_mstatus 0x00000080 c784_mstatush 0x00000000
core   0: 0xbeef3000 (0xabcd4437) lui     s0, 0xabcd4
```

Set_PTE function:

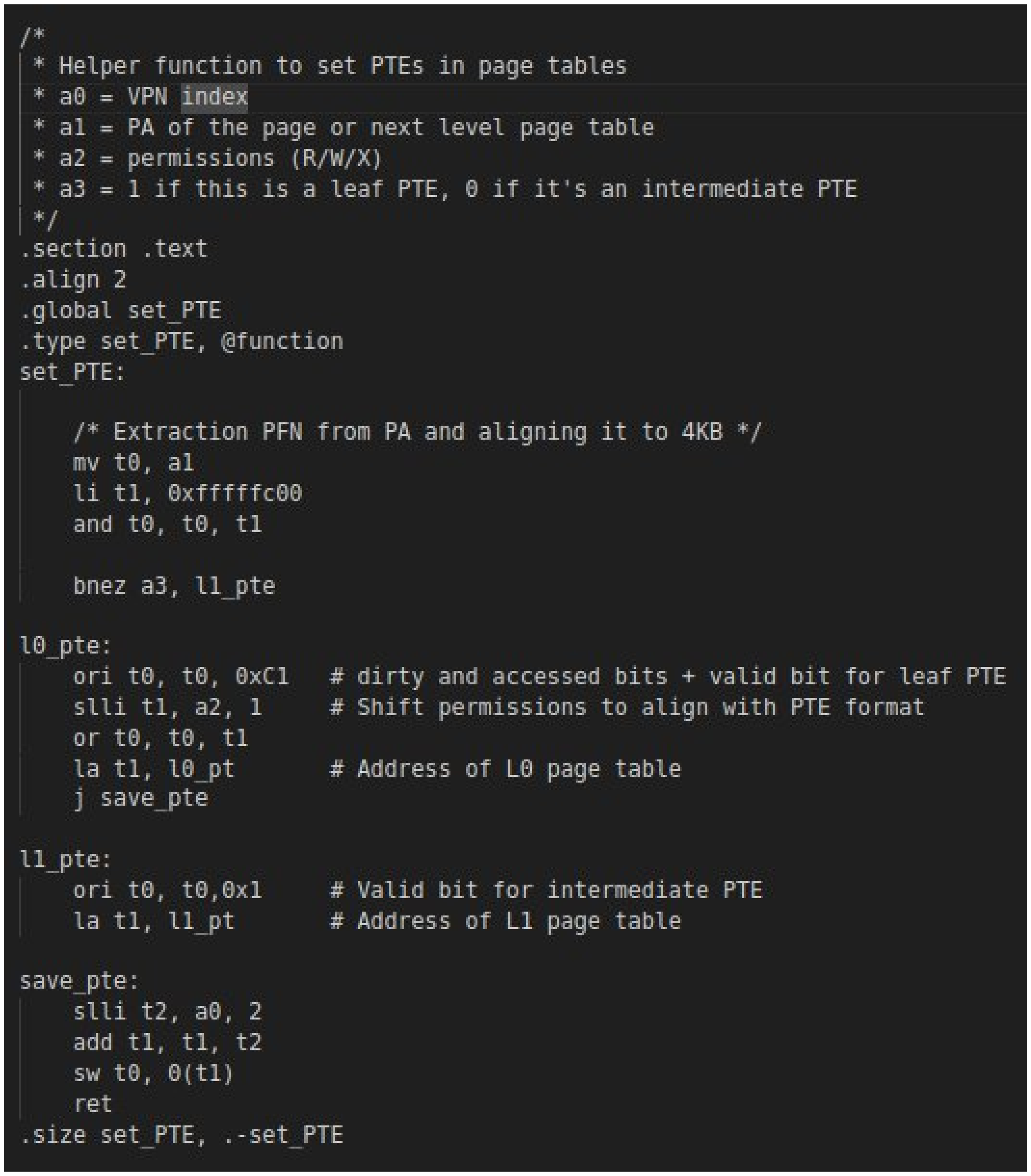
This function set the PTE values to the respective page tables using the inputs provided as:

a0 = virtual page number index

a1 = physical address of the page or next page table

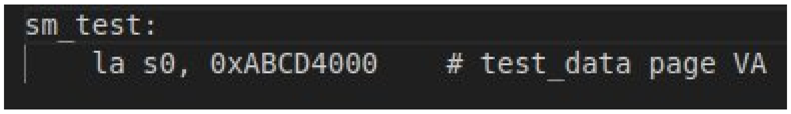
a2 = value for permission bits of PTE (R/W/X)

a3 = If 1 it is a leaf pte, if 0 it is not a leaf pte



```
/*
 * Helper function to set PTEs in page tables
 * a0 = VPN index
 * a1 = PA of the page or next level page table
 * a2 = permissions (R/W/X)
 * a3 = 1 if this is a leaf PTE, 0 if it's an intermediate PTE
 */
.section .text
.align 2
.global set_PTE
.type set_PTE, @function
set_PTE:

    /* Extraction PFN from PA and aligning it to 4KB */
    mv t0, a1
    li t1, 0xfffffc00
    and t0, t0, t1

    bnez a3, l1_pte

l0_pte:
    ori t0, t0, 0xC1   # dirty and accessed bits + valid bit for leaf PTE
    slli t1, a2, 1     # Shift permissions to align with PTE format
    or t0, t0, t1
    la t1, l0_pt       # Address of L0 page table
    j save_pte

l1_pte:
    ori t0, t0,0x1     # Valid bit for intermediate PTE
    la t1, l1_pt       # Address of L1 page table

save_pte:
    slli t2, a0, 2
    add t1, t1, t2
    sw t0, 0(t1)
    ret
.size set_PTE, .-set_PTE
```

Virtual address of test_data page is defined as 0xabcd4000

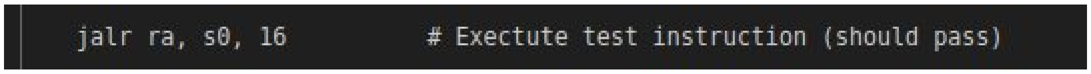


```
sm_test:
    la s0, 0xABCD4000    # test_data page VA
```

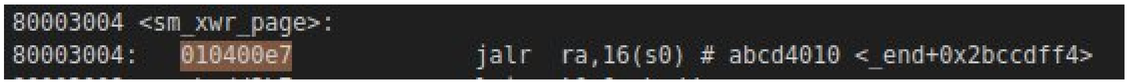
RWX Test:

In this test, all the permission bits are checked as read, write execute. Whether they works and pass correctly or not.

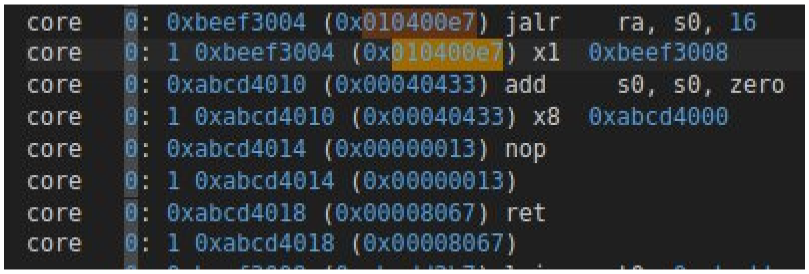
- Execute Test



```
    jalr ra, s0, 16          # Exectute test instruction (should pass)
```

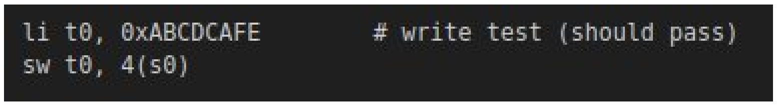


```
80003004 <sm_xwr_page>:
80003004:   010400e7            jalr  ra,16(s0) # abcd4010 <_end+0x2bccdff4>
```

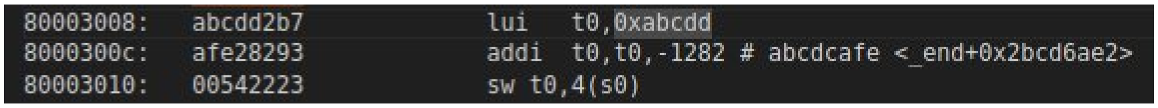


```
core   0: 0xbeef3004 (0x010400e7) jalr    ra, s0, 16
core   0: 1 0xbeef3004 (0x010400e7) x1  0xbeef3008
core   0: 0xabcd4010 (0x00040433) add     s0, s0, zero
core   0: 1 0xabcd4010 (0x00040433) x8  0xabcd4000
core   0: 0xabcd4014 (0x00000013) nop
core   0: 1 0xabcd4014 (0x00000013)
core   0: 0xabcd4018 (0x00008067) ret
core   0: 1 0xabcd4018 (0x00008067)
```

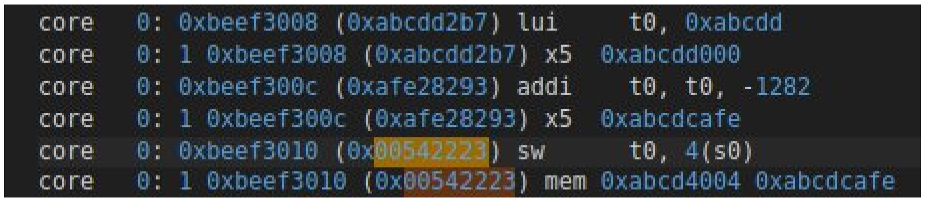
- Write Test



```
li t0, 0xABCDCAFE        # write test (should pass)
sw t0, 4(s0)
```

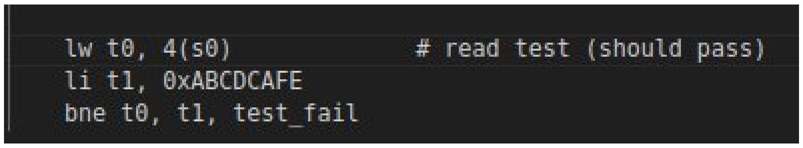


```
80003008:   abcdd2b7            lui   t0,0xabcdd
8000300c:   afe28293            addi  t0,t0,-1282 # abcdcafe <_end+0x2bcd6ae2>
80003010:   00542223            sw t0,4(s0)
```

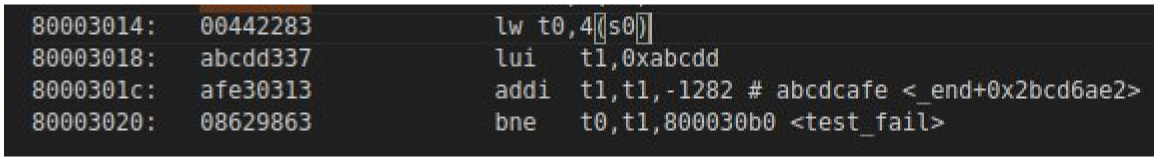


```
core   0: 0xbeef3008 (0xabcdd2b7) lui     t0, 0xabcdd
core   0: 1 0xbeef3008 (0xabcdd2b7) x5  0xabcdd000
core   0: 0xbeef300c (0xafe28293) addi    t0, t0, -1282
core   0: 1 0xbeef300c (0xafe28293) x5  0xabcdcafe
core   0: 0xbeef3010 (0x00542223) sw      t0, 4(s0)
core   0: 1 0xbeef3010 (0x00542223) mem 0xabcd4004 0xabcdcafe
```

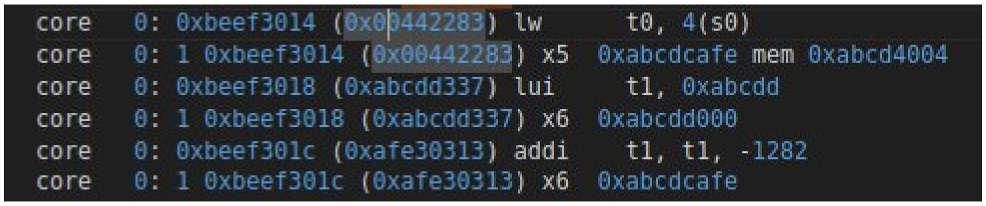
- Read Test



```
    lw t0, 4(s0)            # read test (should pass)
    li t1, 0xABCDCAFE
    bne t0, t1, test_fail
```



```
80003014:   00442283            lw t0,4(s0)
80003018:   abcdd337            lui   t1,0xabcdd
8000301c:   afe30313            addi  t1,t1,-1282 # abcdcafe <_end+0x2bcd6ae2>
80003020:   08629863            bne   t0,t1,800030b0 <test_fail>
```

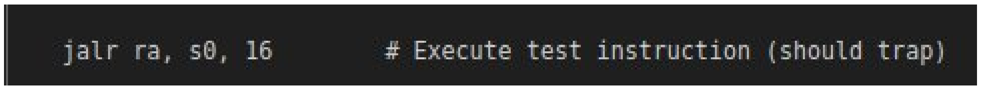


```
core   0: 0xbeef3014 (0x00442283) lw      t0, 4(s0)
core   0: 1 0xbeef3014 (0x00442283) x5  0xabcdcafe mem 0xabcd4004
core   0: 0xbeef3018 (0xabcdd337) lui     t1, 0xabcdd
core   0: 1 0xbeef3018 (0xabcdd337) x6  0xabcdd000
core   0: 0xbeef301c (0xafe30313) addi    t1, t1, -1282
core   0: 1 0xbeef301c (0xafe30313) x6  0xabcdcafe
```

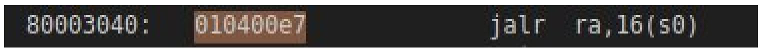
RW Test:

In this test, read/write permission bits are set and will pass. The execute permission bit is not set so it will fault.

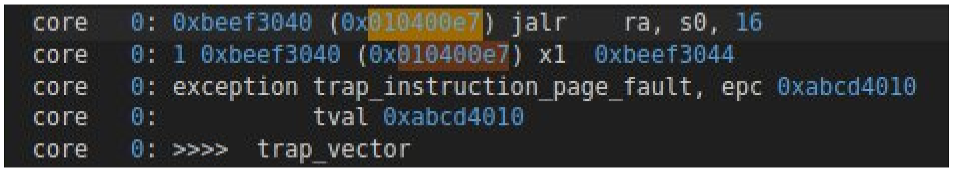
- Execute Test:



```
  jalr ra, s0, 16       # Execute test instruction (should trap)
```



```
80003040:   010400e7            jalr  ra,16(s0)
```

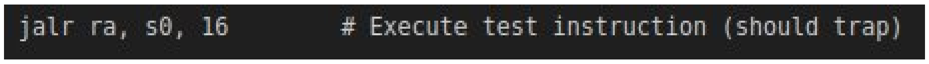


```
core   0: 0xbeef3040 (0x010400e7) jalr    ra, s0, 16
core   0: 1 0xbeef3040 (0x010400e7) x1  0xbeef3044
core   0: exception trap_instruction_page_fault, epc 0xabcd4010
core   0:           tval 0xabcd4010
core   0: >>>>  trap_vector
```

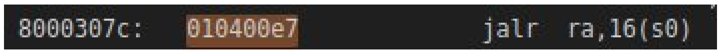
R Test:

In this test, Only read permission bit is set and will pass. The execute/write permission bits are not set so both will give respective page fault.

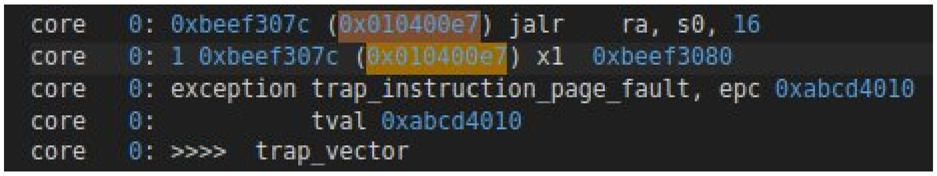
- Execute Test:



```
jalr ra, s0, 16       # Execute test instruction (should trap)
```

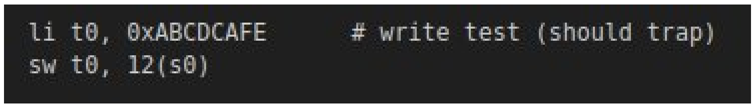


```
8000307c:   010400e7            jalr  ra,16(s0)
```

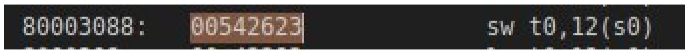


```
core   0: 0xbeef307c (0x010400e7) jalr    ra, s0, 16
core   0: 1 0xbeef307c (0x010400e7) x1  0xbeef3080
core   0: exception trap_instruction_page_fault, epc 0xabcd4010
core   0:           tval 0xabcd4010
core   0: >>>>  trap_vector
```

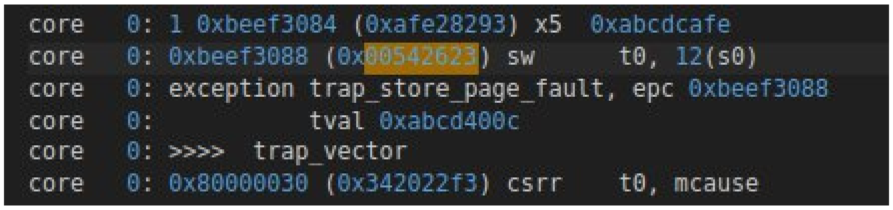
- Write Test:



```
li t0, 0xABCDCAFE      # write test (should trap)
sw t0, 12(s0)
```



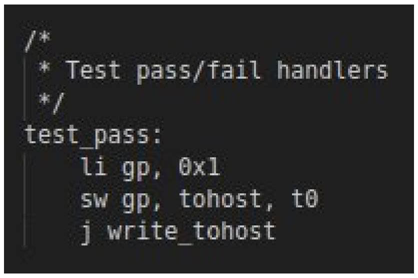
```
80003088:   00542623            sw t0,12(s0)
```



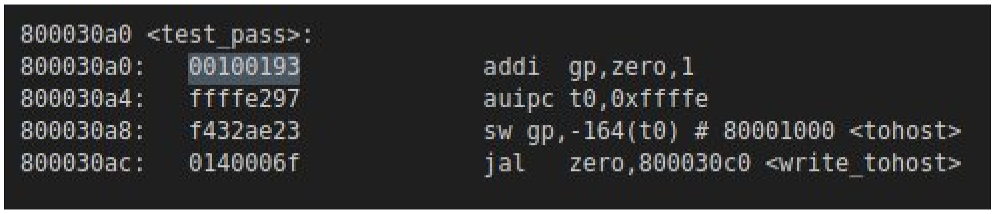
```
core   0: 1 0xbeef3084 (0xafe28293) x5  0xabcdcafe
core   0: 0xbeef3088 (0x00542623) sw      t0, 12(s0)
core   0: exception trap_store_page_fault, epc 0xbeef3088
core   0:           tval 0xabcd400c
core   0: >>>>  trap_vector
core   0: 0x80000030 (0x342022f3) csrr    t0, mcause
```

Test Pass:

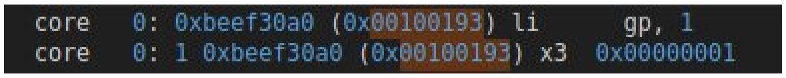
After successful test running, it will jump to test_pass label and write 1 to gp which shows test is successful.



```
/*
 * Test pass/fail handlers
 */
test_pass:
    li gp, 0x1
    sw gp, tohost, t0
    j write_tohost
```



```
800030a0 <test_pass>:
800030a0:   00100193            addi  gp,zero,1
800030a4:   ffffe297            auipc t0,0xffffe
800030a8:   f432ae23            sw gp,-164(t0) # 80001000 <tohost>
800030ac:   0140006f            jal   zero,800030c0 <write_tohost>
```



```
core   0: 0xbeef30a0 (0x00100193) li      gp, 1
core   0: 1 0xbeef30a0 (0x00100193) x3  0x00000001
```