Report-Lab3

author: Yunxin Yang

studentID: 2023233204

Design and Implementation

Design of the handler kernel code

the asm code is below

```
# a1: address
# a2: number
# a0：max return number
# a2 must > 1
# addi    t1,     x0,     1
# blt     a2,     t1,     end # if t0 <= t1 then target

lw      a0,     0(a1)       # initialize max number : a0
addi    t1,     x0,     0   # counter : t1

loop:
    lw      t2,     0(a1)   # get new number
    blt     t2,     a0,     gotonextloop    # if new number < a0
    add     a0,     t2,     x0 # update max value
gotonextloop:
    addi    a1,     a1,     4   # address +4
    addi    t1,     t1,     1   # counter +1
    blt     t1,     a2,     loop
    sret    #   00010   00  00010   00000   000 00000   11100
```

I implement this handler to transfer a1(the array address) and a2 (the number of array elements) to return the max value of the array.

After design, I use venus to translate the above asm code into binary machine code below:

```
1w x10 0(x11)     0x0005A503
addi x6 x0 0      0x00000313
1w x7 0(x11)      0x0005A383
blt x7 x10 8      0x00A3C463
add x10 x7 x0     0x00038533
addi x11 x11 4    0x00458593
addi x6 x6 1      0x00130313
blt x6 x12 -20    0xFEC346E3
sret              0x10200073
```

store the handler code

1. At the begining of the simulater’s simulate function, I designate the kernel code address at `0x80000000` , and store the binary code above into the address for later acquiring the handler kernel code.

```
    const int handlerCodesNumber = 9;
  const int handlerCodesAddress = 0x80000000;
  const u_int32_t handlerCodes[9] = {
      0x0005A503,
      0x00000313,
      0x0005A383,
      0x00A3C463,
      0x00038533,
      0x00458593,
      0x00130313,
      0xFEC346E3,
      0x10200073
  };
void Simulator::storeHandlerCodes() {
  uint32_t cycles;
  for (size_t i = 0; i < handlerCodesNumber; i++)
```

```
    this->memory->setInt(handlerCodesAddress+i*4, handlerCodes[
}
```

the handler ecall design

1. I design the call is triggered when a7’s value is 7 as the type of the ecall, and use global flag `ifExecutingHandler` to mark incurring handler to wait to switch to kernel, at the same time mark the nextpc to `pc_preserve` as `dReg.pc` to save the pc
2. before simulate executes next cycle, it will judge if `ifExecutingHandler` is set to 1; if set to 1, judge if the Mem stage and Writeback stage is done to ensure instructions before committed before the handler executes
3. when the Mem stage and Writeback stage is done , save the all registers to reg_preserve array for recover, and switch the `this->pc = handlerhandlerCodesAddress` for next cycle to extract the handler code from memory.

implement code is below:

```
// incur handler
    if (ifExecutingHandler){
      this->fReg.bubble = true;
      this->dReg.bubble = true;
      printf("ifExecutingHandler 1, waiting to switch to kernel
      // if done previous instructions
      if ((true == this->eReg.bubble || !this->eReg.pc) && (true
        if (!pc_preserve){
          panic("pc_preserve == 0\n");
        }
        printf("check pc_preserve: %x", pc_preserve);
        // switch to kernel code
        this->pc = handlerCodesAddress;
        // preserve registers
        for (size_t i = 0; i < RISCV::REGNUM; i++)
          reg_preserve[i] = reg[i];
        // clear user mode waiting for kerner mode
        ifExecutingHandler = false;
      }else{
      // not done previous instructions
        if (this->dReg.pc)
          pc_preserve = this->dReg.pc;
      }
    }
```

Handler design

1. I add decoding the sret

   ```
   } else if (funct3 == 0x0 && funct7 == 0x8){
           // handler
           printf("incur handler sret\n");
           // wait for next stage to judge sret using type 8
           instname = "ecall";
           op2 = 8;
           insttype = ECALL;
   ```

   and when it comes to execute stage, it will regonise it and set global mask `ifExecutingSret`

2. before simulate executes next cycle, it will judge if `ifExecutingSret`is set to 1; if set to 1, judge if the Mem stage and Writeback stage is done to ensure instructions before committed before the handler executes

3. when the Mem stage and Writeback stage is done , recoverto reg_preserve array to the all registers , and switch the `this->pc = pc_preserve` for recovery

implement code is below:

```
if (ifExecutingSret){
    this->fReg.bubble = true;
    this->dReg.bubble = true;
    printf("ifExecutingSret 1, waiting to switch to user code\n
    // if done previous instructions
    if ((true == this->eReg.bubble || !this->eReg.pc) && (true
```

```
      printf("go back to pc_preserve: %x\n", pc_preserve);
      // switch to user code, recover registers
      this->pc = pc_preserve;
      // preserve registers
      for (size_t i = 0; i < RISCV::REGNUM; i++){
        // but overwrite with a0
        if (i != REG_A0)
          reg[i] = reg_preserve[i];
      }
      // clear kerner mode waiting for user mode
      ifExecutingSret = false;
    }else{
    // not done previous instructions
    }
  }
```

Test

test code

first store array {0,1,2,3,4,5,6,7} at 0x80000300, and then use the syscall to find the max value

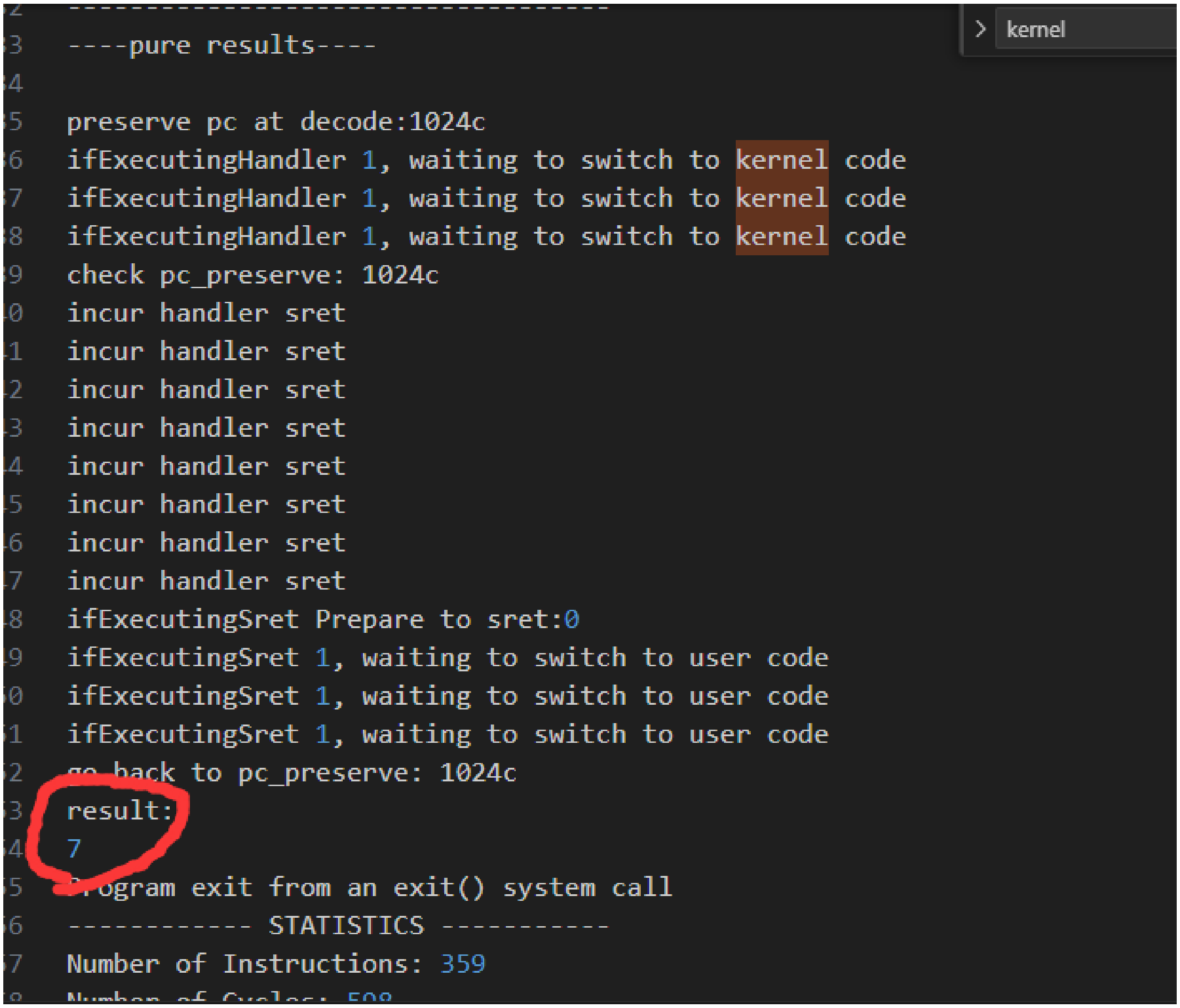
```
#include "/home/yangyx/desktop/CA2/cs211_23f_lab_sim_framework/
#include <stdint.h>
#include <stdio.h>

int result[10]={1,2,3,4,5,6,7,8,9,10};

//result:11 12 13 14 15 1 2 3 4 5

int main()
{
    int maxArrayValue;
  asm volatile(
    "addi   t0,     x0,     0;"
    "lui    t0,     0x80000;"
    // "lui t0,     0x7FFFF;"
    "addi   t0,     t0,     768;" // address : 0x80000300
        "addi   t1,     x0,     0;"
        "sw     t1,     0(t0);"
        "addi   t1,     x0,     1;"
        "sw     t1,     4(t0);"
        "addi   t1,     x0,     2;"
        "sw     t1,     8(t0);"
        "addi   t1,     x0,     3;"
        "sw     t1,     12(t0);"
        "addi   t1,     x0,     4;"
        "sw     t1,     16(t0);"
        "addi   t1,     x0,     5;"
        "sw     t1,     20(t0);"
        "addi   t1,     x0,     6;"
        "sw     t1,     24(t0);"
        "addi   t1,     x0,     7;"
        "sw     t1,     28(t0);"

        "addi   a7,     x0,     7;"
        "addi   a1,     t0,     0;"
        "addi   a2,     x0,     8;"
        "scall;"
    );
    asm volatile(
        "addi %[result], a0,     0"
        : [result] "=r" (maxArrayValue)
    );
    print_s("result:\n");
    print_d(maxArrayValue);
    print_s("\n");
    return 0;
}
```

I write a test code first store the array at 0x80000300, and use below code

```
        "addi   a7,     x0,     7;"
        "addi   a1,     t0,     0;"
        "addi   a2,     x0,     8;"
        "scall;"
```

to trigger the ecall I write by myself

result



```
---------------------------------
----pure results----

preserve pc at decode:1024c
ifExecutingHandler 1, waiting to switch to kernel code
ifExecutingHandler 1, waiting to switch to kernel code
ifExecutingHandler 1, waiting to switch to kernel code
check pc_preserve: 1024c
incur handler sret
incur handler sret
incur handler sret
incur handler sret
incur handler sret
incur handler sret
incur handler sret
incur handler sret
ifExecutingSret Prepare to sret:0
ifExecutingSret 1, waiting to switch to user code
ifExecutingSret 1, waiting to switch to user code
ifExecutingSret 1, waiting to switch to user code
go back to pc_preserve: 1024c
result:
7
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 359
Number of Cycles: 508
```

the result turns out fine. the description line above the “result” is the sign that I switch between kernel code and user code