Operating Systems Design (CS 423)

Elsa L Gunter
2112 SC, UIUC
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Code from Sam King

```
typedef uint32_t u32;
typedef int32_t i32;
// Visible state of CPU
struct CPU {
    u32 regs[NUM_REGS];
    u32 pc;
    bool icc_z;
    bool icc_n;
};
```


Main setup

```
int main(int argc, char *argv[]) {  ...
    // initialize our state
    ram = new uint8_t[RAM_SIZE];
    cpu = new struct CPU;
    for(int idx = 0; idx < NUM_REGS; idx++) {
        cpu->regs[idx] = 0;}
    cpu->pc = 0;

    // setup the stack pointer
    cpu->regs[14] = RAM_SIZE-4-120;
    // setup the program
    cpu->regs[8] = RAM_SIZE-4;
```


Main setup

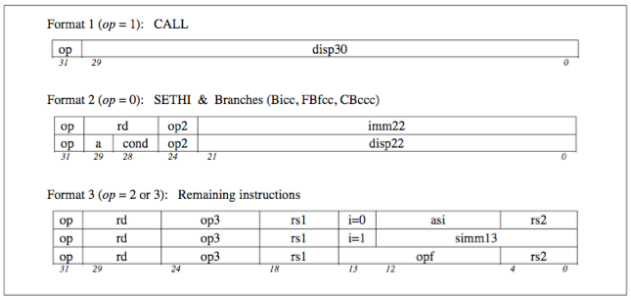
```
    // fetch our memory image and cpu state
  (if set)
    fillState(argv[1], ram, RAM_SIZE);
    if(argc >= 3) {
       fillState(argv[2], cpu, sizeof(struct
  CPU));
    }
    startTime = time(NULL);

    cpu_exec(cpu);

    return 0;
}
```


Task 2: Implement exec

- Decode instructions
- Update register state
- Update pc state

- Note: r0 is always 0!
- Note: advancing the pc in nonbranch
  - pc+=4

Sparc instruction format



Code from Sam King

```
struct control_signals {
   u32 op;
   u32 rd;
   u32 cond;
   u32 op2;
   u32 op3;
   u32 rs1;
   u32 i;
   u32 asi;
   i32 simm;
   u32 imm;
   u32 disp22;
   u32 disp30;
   u32 rs2;
   u32 raw;
};
```


Decode

```
void decode_control_signals(u32 opcode, struct
  control_signals *control) {
    control->raw = opcode;
    control->op = (opcode >> 30) & 0x3;
    control->rd = (opcode >> 25) & 0x1f;
    control->op2 = (opcode >> 22) & 0x7;
    control->op3 = (opcode >> 19) & 0x3f;
    control->rs1 = (opcode >> 14) & 0x1f;
    control->i = (opcode >> 13) & 0x1;
    control->asi = (opcode >> 5) & 0xff;
    control->simm = sign_extend_13(opcode & 0x1fff);
    control->imm = opcode & 0x3fffff;
    control->rs2 = opcode & 0x1f;
    control->disp22 = opcode & 0x3fffff;
    control->disp30 = opcode & 0x3fffffff;
    control->cond = (opcode >> 25) & 0xf;
}
```


Task 2: Implement exec

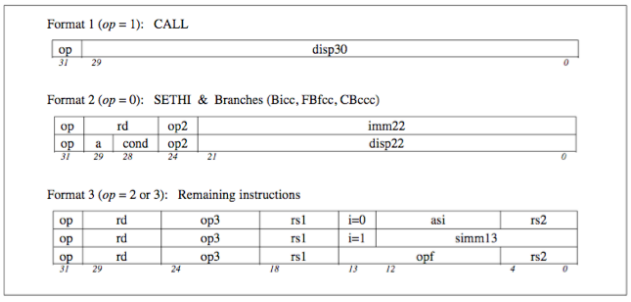
- Decode instructions
- Update register state
- Update pc state

- Note: r0 is always 0!
- Note: advancing the pc in nonbranch
  - pc+=4

cpu_exec

```
void cpu_exec(struct CPU *cpu) {
   struct control_signals *control = new struct control_signals;
   u32 opcode;
   bool runSimulation = true;

   while(runSimulation && !userQuit) {
      opcode = fetch(cpu);
      decode_control_signals(opcode, control);
      // second part of decode and write back also
      runSimulation = execute(cpu, control);
   }
```


Sparc instruction format



execute

```
bool execute(struct CPU *cpu, struct control_signals *control) {
   u32 savedPc = cpu->pc;
   // writes to reg 0 are ignored and reads should always be 0
   cpu->regs[0] = 0;
   switch (control->op) {
      case OP_FORMAT_1:
         process_format_1(cpu, control);
         break;
      ...
      case OP_FORMAT_3_ALU:     //  0x2
         process_format_3_alu(cpu, control);
         break;    . . .    }
```

Execute – pc state

```
// the pc is modified in the loop after the branch instruction
if(cpu->pc == savedPc) {
    cpu->pc += sizeof(cpu->pc);
    if(cpu->pc < savedPc) {
        return false;
    }
}
return true;
}
```


process_format_3_alu

```
void process_format_3_alu(struct CPU *cpu,
                          struct control_signals *control) {
  switch(control->op3)
  {
    case OP3_ADD:




        break;
```

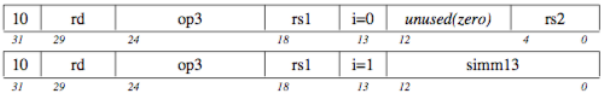

process_format_3_alu

```
void process_format_3_alu(struct CPU *cpu,
                          struct control_signals *control) {
  switch(control->op3)
  {
    case OP3_ADD:
      if(control->i == 0) {
        add(cpu, control->rd, control->rs1, control->rs2);
      }
      else {
        addi(cpu, control->rd, control->rs1, control->simm);
      }
      break;
```


Add format

opcode	*op3*	*operation*
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

Format (3):



Addi

- r[rd]=r[rs1]+sign_ext(simm13)

Addi

- r[rd]=r[rs1]+sign_ext(simm13)

```
void addi(struct CPU *cpu, u32 rd, u32 rs1, i32
  signedImm) {
    cpu->regs[rd] = cpu->regs[rs1] +
  signedImm;
```

Task3: Implement Checkpoint

- Prompt user and let them choose file if they quit
- Save memory state in a file
- Save cpu state in a file

cpu_exec

```
if(userQuit) {
      if(prompt("Would you like to save your system state (y/
  n)?: ") == "y") {
          string memFileName = prompt("mem file name: ");
          string cpuFileName = prompt("cpu file name: ");
          saveState(memFileName.c_str(), ram, RAM_SIZE);
          saveState(cpuFileName.c_str(), cpu, sizeof(struct
  CPU));
      }
  }
}
```


Simulator

- CPU state --- data structure within your sim.
  - Includes registers, IDT, etc.
- Memory --- malloc
  - Guest physical address 0 == beginning of malloc
- Disk --- file
  - Disk block 1 = file offset 1*(size of disk block)
- Display --- window
- Within simulator, does the software "know" it is being simulated?

VMM environment

- Duplicate
  - Virtual machine == physical machine
- Efficient
  - Runs almost as fast as real machine
- Isolated
  - VMM has control over resources
  - What are the resources the VMM controls?
- Which properties does Simulator have?

Key observation

- If the virtual ISA == physical ISA
  - CPU can perform simulation loop
- Can we set host PC to address we want to start at and let it run?
- What property are we violating?
- What else goes wrong?

Main issues

- Privileged instructions
- Instructions operate on physical state, not virtual state

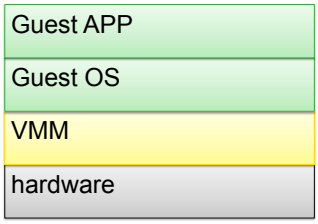
Privileged instructions

- CPU divided into supervisor and user modes
- Part of the CPU ISA only accessible by "supervisor" code
- Allowing guest OS execute these would violate isolation
  - E.g., I/O instructions to write disk blocks
- Solution: run guest OS in user mode
  - CPU user mode: only non-privileged instr
  - CPU supervisor mode: all instr

Example: interrupt descriptor table (IDT)

- x86 processor have an IDT register
  - CPU uses to find interrupt service routines
- Set the IDT register using the lidt instruction
- VMM must handle all interrupts to maintain control
- Goal: allow the guest OS to set the virtual IDT register without affecting the physical CPU

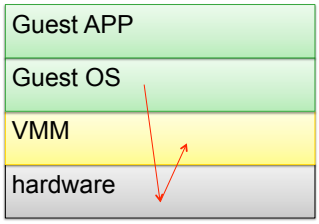
Guest OS priv. inst.



User mode
Supervisor mode

- lidt 0x1234
- vcpu->idtr = 0x4321
- idtr = 0x5555

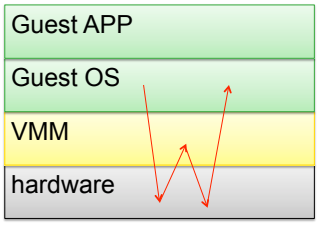
Guest OS priv. inst.



User mode
Supervisor mode

- lidt 0x1234
- vcpu->idtr = 0x4321
- idtr = 0x5555

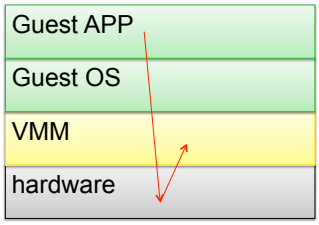
Guest OS priv. inst.



User mode
Supervisor mode

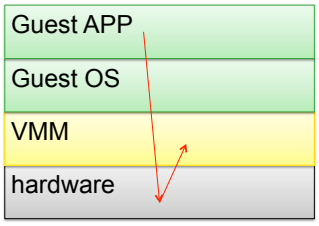
- lidt 0x1234
- vcpu->idtr = 0x1234
- idtr = 0x5555

Guest OS priv. inst.



User mode
Supervisor mode

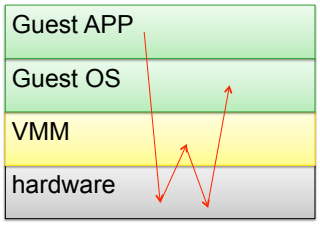
Guest OS priv. inst.



User mode
Supervisor mode

- vcpu->supervisor = false

Guest OS priv. inst.



User mode
Supervisor mode

- vcpu->supervisor = true