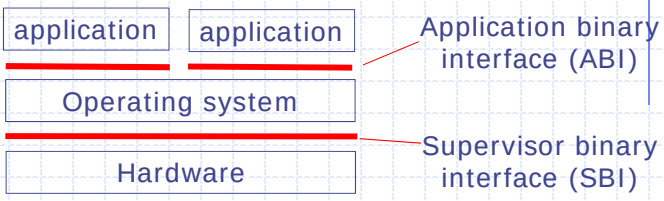
Constructive Computer Architecture

Interrupts/Exceptions/Faults

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

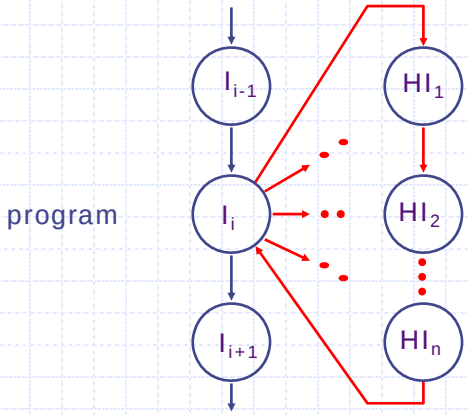
Hardware Software Interactions



- Modern processors cannot function without some resident programs ("services"), which are shared by all users
- Usually such programs need some special registers which are not visible to the users
- Therefore all ISAs have extensions to deal with these special registers
- Furthermore, these special registers cannot be manipulated by user programs; therefore *user/privileged mode* is needed to use these instructions

Interrupts

altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

Asynchronous Interrupts

caused by an external event

- External event
  - input/output device service-request/response
  - timer expiration
  - power disruptions, hardware failure
- After the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} *(Precise interrupt)*
  - It saves the PC of instruction I_i in a special register
  - It disables interrupts and transfers control to a designated interrupt handler running in the privilege mode
    - *Privileged/user mode* to prevent user programs from causing harm to other users or OS

Synchronous Interrupts

caused by the execution of instruction

- The instruction cannot be completed
  - undefined opcode, privileged instructions
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults, TLB misses, protection violations

  } exception
- System call: Deliberately used by the programmer to invoke a kernel service

  } trap
  - Either the faulting condition is fixed, the instruction is emulated by the exception handler, or the program is aborted
  - The pipeline must undo any partial execution and record the cause of the exception

Architectural features for Interrupt Handling

- Special registers
  - `mepc` holds pc of the instruction that causes the interrupt
  - `mcause` indicates the cause of the interrupt
  - `mscratch` holds the pointer to HW-thread local storage for saving context before handling the interrupt
  - `mstatus`

  In RISC-V mepc, mcause and mstatus are some of the Control and Status Registers (CSRs)
- Special instructions
  - ERET (*environment return*) to return from an exception/fault handler program using mepc. It restores the previous interrupt state, mode, cause register, …
  - Instruction to manipulate and move CSRs into GPRs
  - need a way to mask further interrupts at least until mepc can be saved

  RISC-V has four modes; we deal with only user and machine modes

Interrupt Handling

System calls

- A system call instruction causes an interrupt when it reaches the execute stage
  - decoder recognizes a SCALL instruction
  - current pc is stored in `mepc`
  - `mcause` is set to the value defined for system calls
  - the processor is switched to privileged mode and disables interrupts; previous privilege level is saved in `mstatus`
  - PC is redirected to the Exception handler which is available in the `mtvec` for the user mode
  - The effective meaning of the SCALL instruction is defined by the kernel; a convention
    - Register a7 contains the desired function,
    - register a0,a1,a2 contain the arguments,
    - result is returned in register a0

Processor can't function without the cooperation of the software

Single-cycle implementation: next few slides

Single-Cycle RISC-V

```
rule doExecute;
  let inst = iMem.req(pc);
  let dInst = decode(inst, csrf.getStatus);
  let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
  let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
  let eInst = exec(dInst, rVal1, rVal2, pc, ?);
  if(eInst.iType == Ld)
     eInst.data <- dMem.req(MemReq{op: Ld, addr:
           eInst.addr, data: ?});
  else if(eInst.iType == St)
     let d <- dMem.req(MemReq{op: St, addr:
           eInst.addr, data: eInst.data});
  if (isValid(eInst.dst))
     rf.wr(fromMaybe(?, eInst.dst), eInst.data);
  … setting special registers …
  … next address calculation …
endrule endmodule
```

Type: Decoded Instruction

```
typedef struct {
  IType            iType;
  AluFunc          aluFunc;
  BrFunc           brFunc;
  Maybe#(RIndx)    dst;
  Maybe#(RIndx)    src1;
  Maybe#(RIndx)    src2;
  Maybe#(Data)     imm;
  Maybe#(CsrIndx)  csr;
} DecodedInst deriving(Bits, Eq);

typedef enum {Unsupported, NoPermit, Alu, Ld, St, J,
Jr, Br,..., Syscall, ERet} IType deriving(Bits, Eq);
```


Decode

```
function DecodedInst decode(Data inst, Status status);
  DecodedInst dInst = ?; ...
  opSystem: begin
    case (funct3) ...
      fnPRIV: begin // privilege inst
        dInst.iType = (case(inst[31:20])
          fn12SCALL: Syscall; // sys call
          // ERET can only be executed under privilege mode
          fn12ERET: isPrivMode(status) ? Eret : NoPermit;
          ...
          default: Unsupported;
        endcase);
        dInst.dst  = Invalid; dInst.src1 = Invalid;
        dInst.src2 = Invalid; dInst.imm  = Invalid;
        dInst.aluFunc = ?; dInst.brFunc = NT;
      end ... endcase   end
  ...
  return dInst;
endfunction
```

Set special registers

```
if (eInst.iType==Syscall)
begin
    csrf.setStatus(statusPushKU(csrf.getStatus));
    csrf.setCause(32'h08); // cause for System call
    csrf.setEpc(pc);
end else
if (eInst.iType==ERet) begin
    cop.setStatus(statusPopKU(cop.getStatus));
end
```

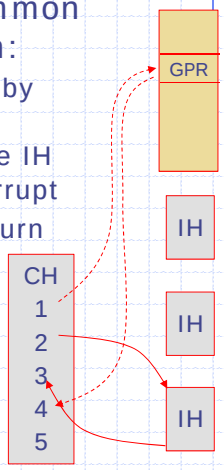

Redirecting PC

```
if (eInst.iType==Syscall)
    pc <= csrf.getTvec;
else if (eInst.iType==ERet)
    pc <= cop.getEpc;
else
    pc <= eInst.brTaken ? eInst.addr : pc + 4;
```

Speed is usually not the paramount concern in hardware handling of interrupts

Software for interrupt handling

- Hardware transfers control to the common software interrupt handler (CH) which:
  1. Saves all GPRs into the memory pointed by `mscratch`
  2. Passes `mcause`, `mepc`, stack pointer to the IH (a C function) to handle the specific interrupt
  3. On the return from the IH, writes the return value to `mepc`
  4. Loads all GPRs from the memory
  5. Execute ERET, which does:
     - set pc to `mepc`
     - pop `mstatus` (mode, enable) stack



Common Interrupt Handler- SW

Not BSV code

```
tvec: # fixed entry point for each mode
                # for user mode the address is 0x100
  j common_handler # One level of indirection

common_handler: # Common wrapper for all IH
  # get the pointer to HW-thread local stack
  csrrw sp, mscratch, sp # swap sp and mscratch
  # save x1, x3 ~ x31 to stack (x2 is sp, save later)
  addi sp, sp, -128
  sw x1, 4(sp)
  sw x3, 12(sp)
  ...
  sw x31, 124(sp)
  # save original sp (now in mscratch) to stack
  csrr s0, mscratch # store mscratch to s0
  sw s0, 8(sp)
```

Common handler- SW *cont.*

Setting up and calling IH_Dispacher

```
Common_handler:
  ... # we have saved all GPRs to stack
  # call C function to handle interrupt
  csrr a0, mcause # arg 0: cause
  csrr a1, mepc # arg 1: epc
  mv a2, sp # arg 2: sp – pointer to all saved GPRs
  jal ih_dispatcher # call C function
  # return value is the PC to resume
  csrw mepc, a0
  # restore mscratch and all GPRs
  addi s0, sp, 128; csrw mscratch, s0
  lw x1, 4(sp); lw x3, 12(sp); ...; lw x31, 124(sp)
  lw x2, 8(sp) # restore sp at last
  eret # finish handling interrupt
```


IH Dispatcher

```
long ih_dispatcher(long cause, long epc, long *regs) {
  // regs[i] refers to GPR xi stored in stack
  if(cause == 0x08)
    return syscall_ih(cause, epc, regs);
  else if(cause == 0x02)
    return illegal_ih(cause, epc, regs);
  else ... // other causes
}
```

Syscall Interrupt Handler

```
long syscall_ih(long cause, long epc, long *regs) {
    // figure out the type of SCALL (stored in a7/x17)
    // args are in a0/x10, a1/x11, a2/x12
    long type = regs[17]; long arg0 = regs[10];
    long arg1 = regs[11]; long arg2 = regs[12];
    if(type == SysPrintChar) { ... }
    else if(type == SysPrintInt) { ... }
    else if(type == SysExit) { ... }
    else ...
    // SCALL finshes, we need to resume to epc + 4
    return epc + 4;
}
```


Another Example: SW emulation of MULT instruction

`mul rd, rs1, rs2`

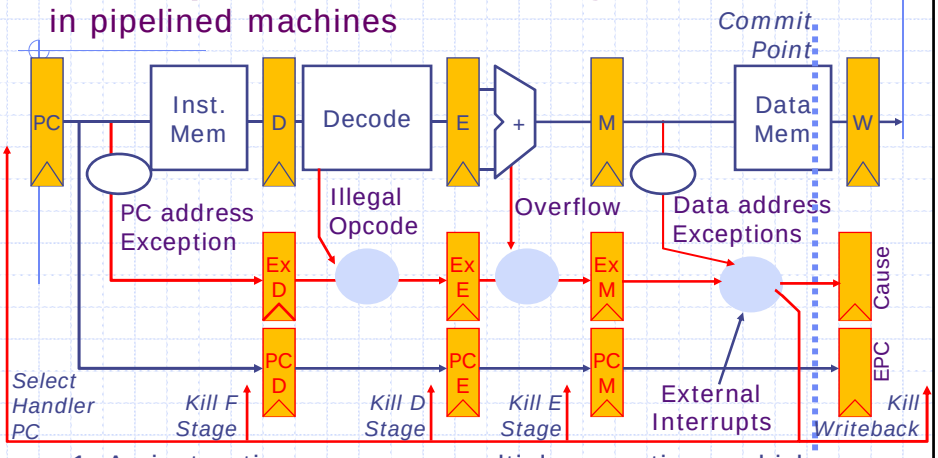
- Suppose there is no hardware multiplier. With proper exception handlers we can implement unsupported instructions in SW
- MUL returns the low 32-bit result of rs1*rs2 into rd
- MUL is decoded as an unsupported instruction and will throw an Illegal Instruction exception
- HW Jump to the same CH as in handling SCALL
- SW handles the exception in illegal_inst_ih() function
  - illegal_inst_ih() checks the opcode and function code of MUL to call the emulated multiply function
- Control is resumed to epc + 4 after emulation is done (ERET)

Illegal Instruction IH

```
long illegal_inst_ih(long cause, long epc, long *regs)
{   uint32_t inst = *((uint32_t*)epc); // fetch inst
    // check opcode & function codes
    if((inst & MASK_MUL) == MATCH_MUL) {
      // is MUL, extract rd, rs1, rs2 fields
      int rd = (inst >> 7) & 0x01F;
      int rs1 = ...; int rs2 = ...;
      // emulate regs[rd] = regs[rs1] * regs[rs2]
      emulate_multiply(rd, rs1, rs2, regs);
      return epc + 4; // done, resume at epc+4
    } else abort();
}
```


Exception Handling

in pipelined machines



1. An instruction may cause multiple exceptions; which one should we process? from the earliest stage
2. When multiple instructions are causing exceptions; which one should we process first? from the oldest instruction