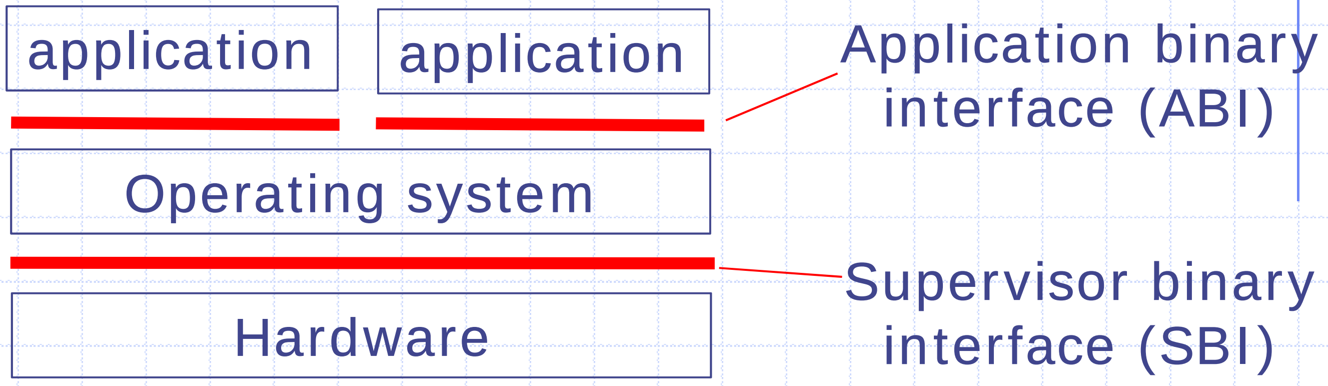
Constructive Computer Architecture

Interrupts/Exceptions/Faults

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

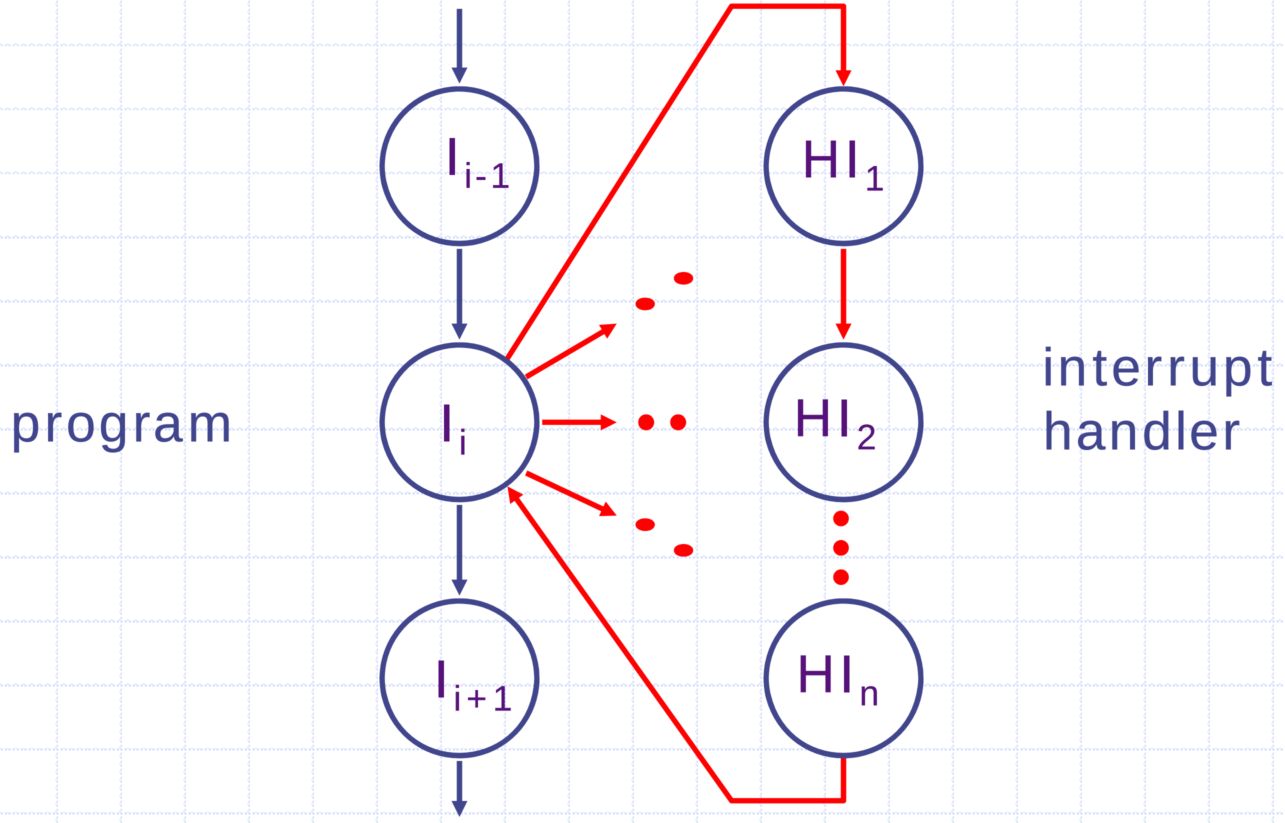
Hardware Software Interactions


application
application
Operating system
Hardware
Application binary interface (ABI)
Supervisor binary interface (SBI)

- 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.

Causes of Interrupts

events that request the attention of the processor

- Asynchronous: an *external event*
  - input/output device service-request/response
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event* caused by the execution of an instruction
  - *exceptions:* 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
  - *traps:* Deliberately used by the programmer for a purpose, e.g., a system call to jump into kernel

Asynchronous Interrupts:
invoking the interrupt handler

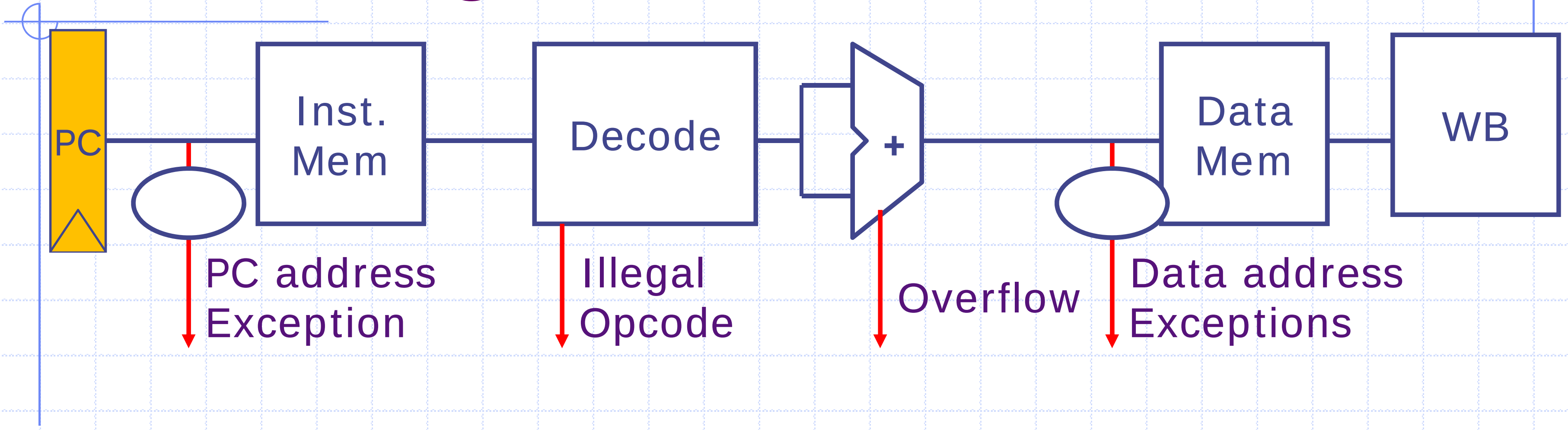
- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- After the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} *(Precise interrupt)*
  - It saves the PC of instruction 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

Usually speed is not the paramount concern in handling interrupts

Synchronous Interrupts

- Requires undoing the effect of one or more partially executed instructions
- Exception: Since the instruction cannot be completed, it is *restarted* if the exception can be handled successfully
  - information about the exception has to be recorded and conveyed to the exception handler
- Trap: After a the kernel has processed a trap, the instruction is typically considered to have completed
  - system calls require changing the mode from user to privilege

Synchronous Interrupt Handling



- Overflow
- Illegal Opcode
- PC address Exception
- Data address Exceptions
- ...

When an instruction causes multiple exceptions the first one has to be processed

Architectural features for Interrupt Handling

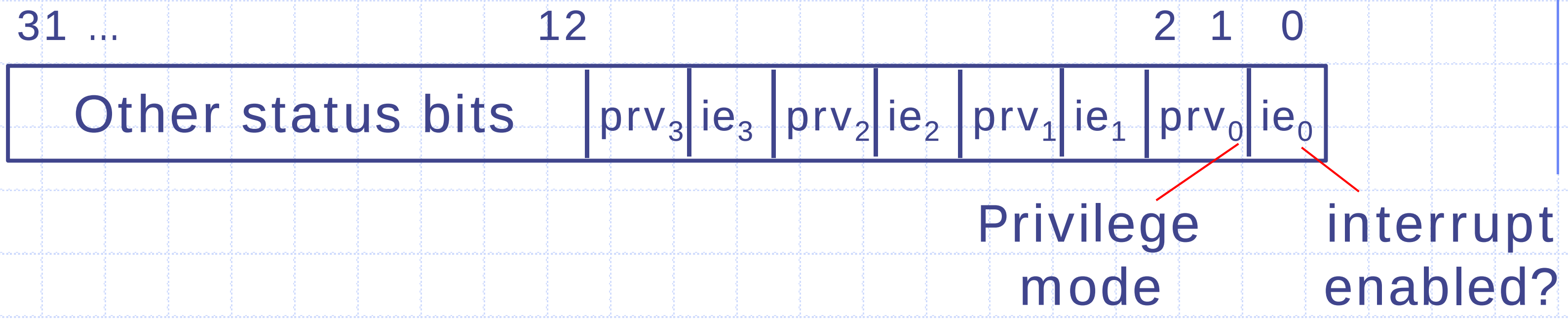
- Special registers
  - mepc holds pc of the instruction that causes the interrupt
  - mcause indicates the cause of the interrupt
  - mscratch holds the pointer to HW-thread local storage for saving context before handling the interrupt
  - mstatus

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

Status register



- Keeps track of three previous interrupts to speed up control transfer
  - When an interrupt is taken, (privilege, disabled) is pushed on to stack
  - The stack is popped by the ERET instruction
    - The default value is (user, enabled)

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 interrupt by pushing (Priv, disable) into `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 is given at the end

Software for interrupt handling

- The hardware transfers the control to the software interrupt handler (IH)
- IH takes following steps:
  - Save all GPRs into the memory pointed by `mscratch`
    - The memory serves as a local stack for interrupt handling
  - Pass `mcause`, `mepc`, stack pointer to a C function, which is supposed to handle this interrupt
  - On the return from the C handler, write the return value to `mepc`
  - Load all GPRs from the memory
  - Execute ERET, which does:
    - set pc to `mepc`
    - pop `mstatus` (mode, enable) stack

Interrupt handler- SW

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

interrupt_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)
```

Interrupt handler- SW *cont.*

```
interrupt_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 c_handler # 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
```

Interrupt handler- SW *cont.*

C function to handle interrupt

```
long c_handler(long cause, long epc, long *regs) {
  // regs[i] refers to GPR xi stored in stack
  if(cause == 0x08) { // cause code for SCALL is 8
    // 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;
  } else if(cause == ...) { ... /* other causes */ }
  else ...
}
```

Another Example: SW emulation of MULT instruction

```
mul rd, rs1, rs2
```

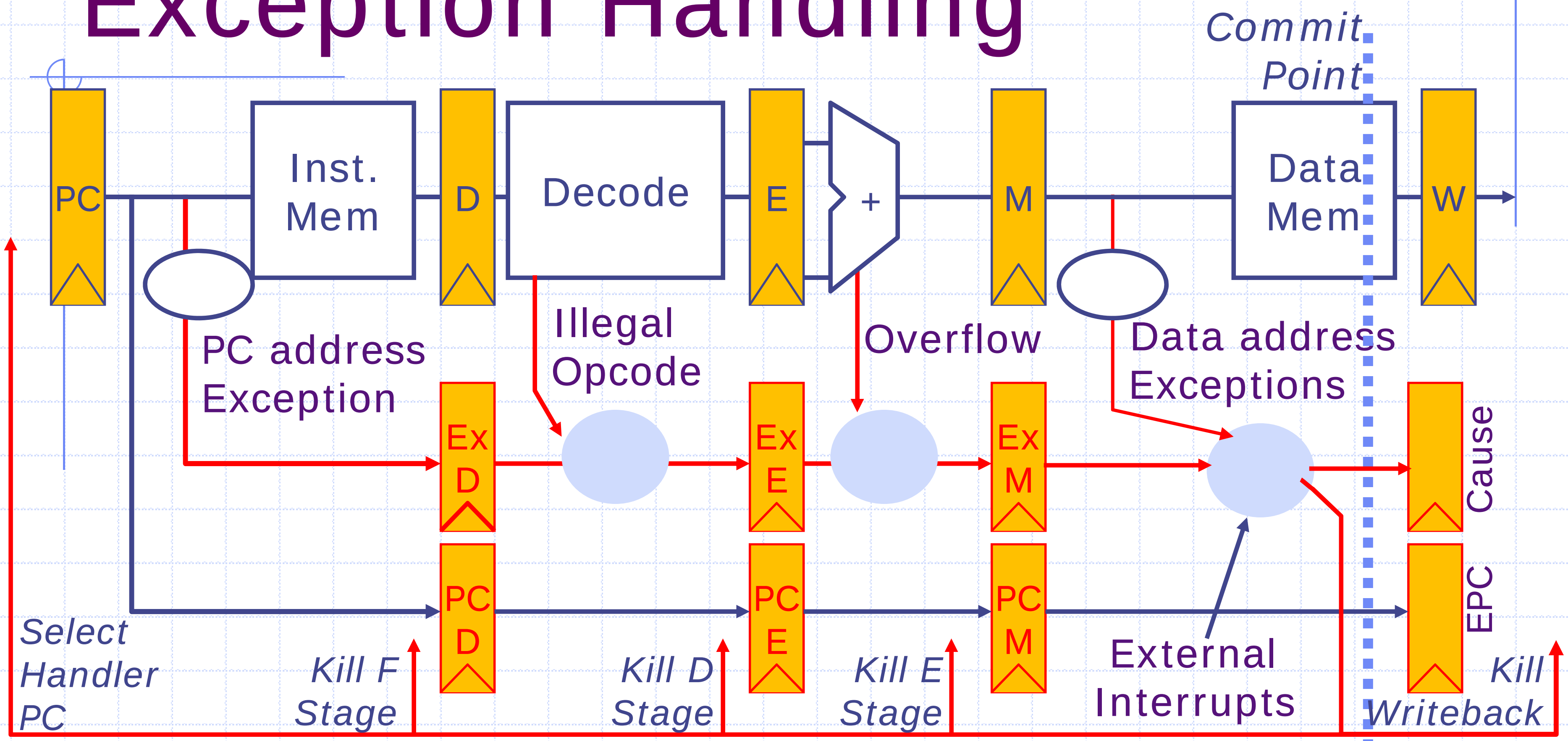
- Suppose there is no hardware multiplier. With proper exception handlers we can implement unsupported instructions in SW
- MUL returns the low 32-bit result of rs1*rs2 into rd
- MUL is decoded as an unsupported instruction and will throw an Illegal Instruction exception
- HW Jump to the same IH as in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

Interrupt handler- SW *cont.*

```
long c_handler(long cause, long epc, long *regs) {
  if(cause == 0x08) { ... /* handle SCALL */ }
  else if(cause == 0x02) {
    // cause code for Illegal Instruction is 2
    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 ... // try to match other inst
  } else ... // other causes
}
```

Exception handling in pipeline machines

Exception Handling



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