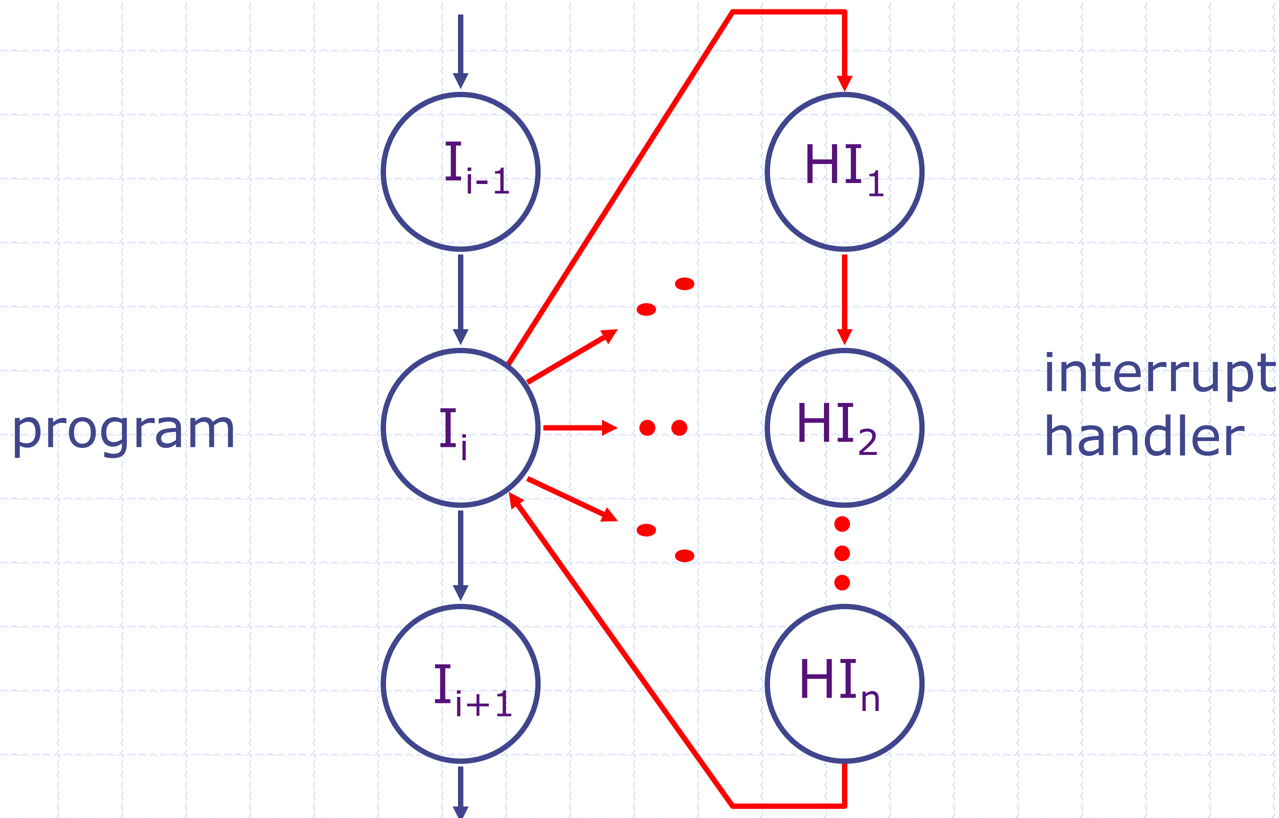
Constructive Computer Architecture

Interrupts/Exceptions/Faults

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

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

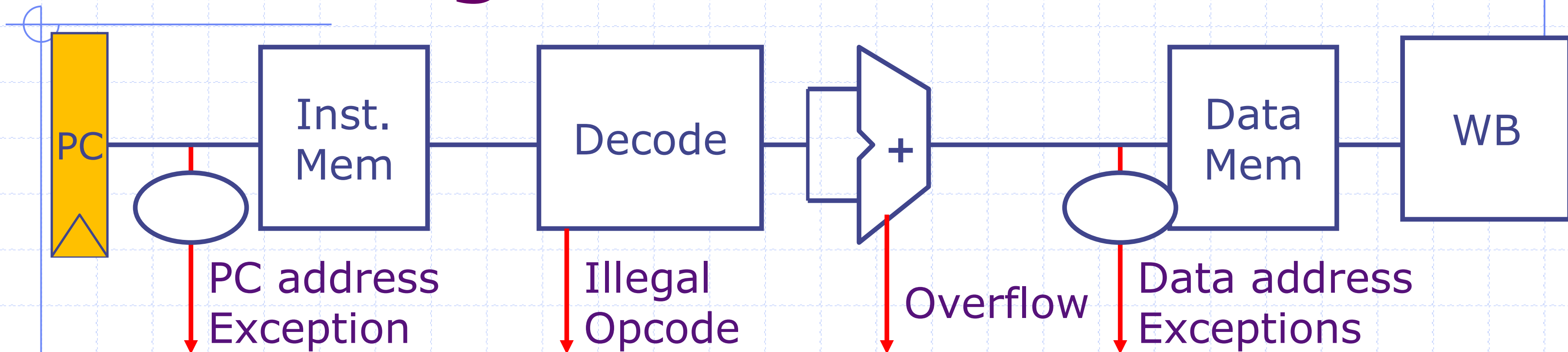
- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- After the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} *(Precise interrupt)*
  - It saves the PC of instruction I_i in a special register
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel 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 typically needs to be *restarted* after the exception has been handled
  - 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 kernel

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

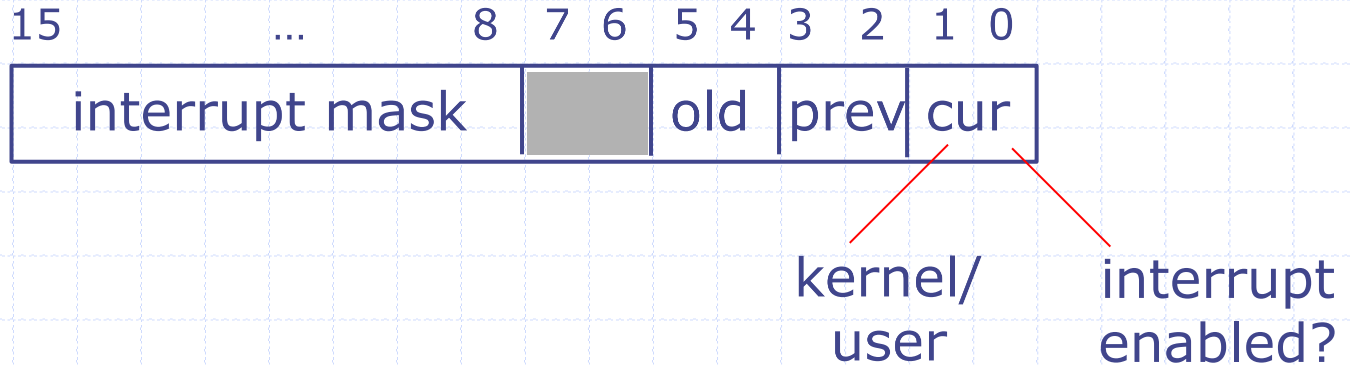
Architectural features for Interrupt Handling

- Special registers
  - `epc` holds `pc` of the instruction that causes the exception/fault
  - Cause Register to indicate the cause of the interrupt
  - Status Register …

In MIPS EPC, Cause and Status Register are Coprocessor registers

- Special instructions
  - `eret` (*return-from-exception*) to return from an exception/fault handler sub-routine using epc. It restores the previous interrupt state, mode, cause register, …
  - Instruction to move EPC etc. into GPRs
  - need a way to mask further interrupts at least until EPC can be saved

Status register



- Keeps the user/kernel, interrupt enabled info and the mask
- It keeps tract of some information about the two previous interrupts to speed up control transfer
  - When an interrupt is taken, (kernel, disabled) is pushed on to stack
  - The stack is popped by the eret instruction

Interrupt Handling

System calls

- A system call instruction causes an interrupt when it reaches the execute stage
  - decoder recognizes a sys call instruction
  - current pc is stored in EPC
  - the processor is switched to kernel mode and disables interrupt
  - Fetch is redirected to the Exception handler
  - Exact behavior depends on the kernel's implementation of the exception handler routine
  - The use of the SYSCALL instruction depends on the convention used by the kernel
    - Suppose: Register $fn contains the desired function,
    - register $arg contains the argument,
    - and the result is stored in register $res

Single-cycle implementation follows

One-Cycle SMIPS

```
rule doExecute;
  let inst = iMem.req(pc);
  let dInst = decode(inst, cop.getStatus);
  let rVal1 = rf.rd1(validRegValue(dInst.src1));
  let rVal2 = rf.rd2(validRegValue(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(validRegValue(eInst.dst), eInst.data);
   … setting special registers …
   … next address calculation …
endrule endmodule
```

Decoded Instruction

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

```
Bit#(6) fcSYSCALL  = 6'b001100;
Bit#(5) rsERET   = 5'b10000;
```

```
typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br,
Syscall, ERet} IType deriving(Bits, Eq);
typedef enum {Add, Sub, And, Or, Xor, Nor, Slt, Sltu,
LShift, RShift, Sra} AluFunc deriving(Bits, Eq);
typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT} BrFunc
deriving(Bits, Eq);
```

Decode

```
function DecodedInst decode(Data Inst, Status status);
  DecodedInst dInst = ?; ...
  opFUNC:                              begin
    case (funct) ...
      fcSYSCALL:
      begin
        dInst.iType = Syscall; dInst.dst  = Invalid;
        dInst.src1 = Invalid; dInst.src2 = Invalid;
        dInst.imm  = Invalid; dInst.brFunc = NT;
      end                              end

  opRS:
   if (status.kuc == 0) // eret is a Kernel Mode instruction
     if (rs==rsERET)                     begin
        dInst.iType = ERet; dInst.brFunc  = AT;
        dInst.rDst    = Invalid; dInst.rSrc1   = Invalid;
        dInst.rSrc2   = Invalid; end
  return dInst;
endfunction
```

Set special registers

```
if (eInst.iType==Syscall)
begin
    Status status=cop.getStatus;
    status=statusPushKU(status);
    cop.setStatus(status);
    Cause cause=cop.getCause;
    cause.excCode=causeExcCode(eInst.iType);
    cop.setCause(cause);
    cop.setEpc(pc);
end else
if (eInst.iType==ERet) begin
    Status status=cop.getStatus;
    status=statusPopKU(status);
    cop.setStatus(status);
end
```

Redirecting PC

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

Interrupt handler- SW

```
exception_handler:
  mfc0 $26, $cause        # get cause register
  srl $26, 2              # shift to get cause in LSB
  andi $26, $26, 0x1f     # apply mask to cause
  li $27, 0x0008          # syscall cause
  beq $26, $27, syscallhandler
  …
syscallhandler:
  li $26, 0xA
  beq $fn, $26, testDone
  li $26, 0xE
  beq $fn, $26, getInsts
  …
testdone: …
getInsts: …
retfrmsyscall:…
```

0xA is the code for syscall testDone

0xE is the code for syscall getInsts

Interrupt handler *cont*

```
testdone:
  mtc0 $arg, $21        # end simulation with $arg
  nop …

getInsts:
  mfc0 $res, $insts
  j retfrmsyscall

retfrmsyscall:
  mfc0 $26, $epc        # get epc
  addiu $26, $26, 4     # add 4 to epc to skip SYSCALL
  mtc0 $26, $epc        # store new epc
  eret                  # return to main at new epc
```

Another Example: SW emulation of MULT instruction

```
mult ra, rb
```

- Suppose there is no hardware multiplier. With proper exception handlers we can implement unsupported instructions in SW
- Multiply returns a 64-bit result stored in two registers: hi and lo
  - These registers are accessed using special instructions (mfhi, mflo, mthi, mtlo)
- Mult is decoded as an unsupported instruction and will throw an RI (reserved instruction) exception
  - The opcode (i.e. Mult or Multu) is checked in software to jump to the emulated multiply function
  - The results are moved to hi and lo using mthi and mtlo
- Control is resumed after the multiply instruction (ERET)

Interrupt handler

```
exception_handler:
  mfc0 $26, $cause              # get cause register
  srl $26, 2                    # shift to get cause in LSB
  andi $26, $26, 0x1f           # apply mask to cause
  li $27, 0x0008                # syscall cause
  beq $26, $27, syscallhandler
  li $27, 0x000a                # ri cause
  beq $26, $27, rihandler
  …

rihandler:
  mfc0 $26, $epc                 # get EPC
  lw $26, 0($26)                 # fetch EPC instruction
  li $27, 0xfc00ffff             # opcode mask for MULT
  and $26, $26, $27
  li $27, 0x00000018             # opcode pattern for MULT
  beq $26, $27, emumult
emumult: …
```

Emulating multiply in SW

- Need to load the contents of ra and rb
- We have the register numbers for ra and rb encoded in Mem[EPC]
- How do we do this?
  - Self-modifying code: construct mov instruction whose rs field is set to ra, etc.
  - Without self-modifying code?

Store all registers in memory sequentially and store the base address into r26.

Bring the index of ra into say r27

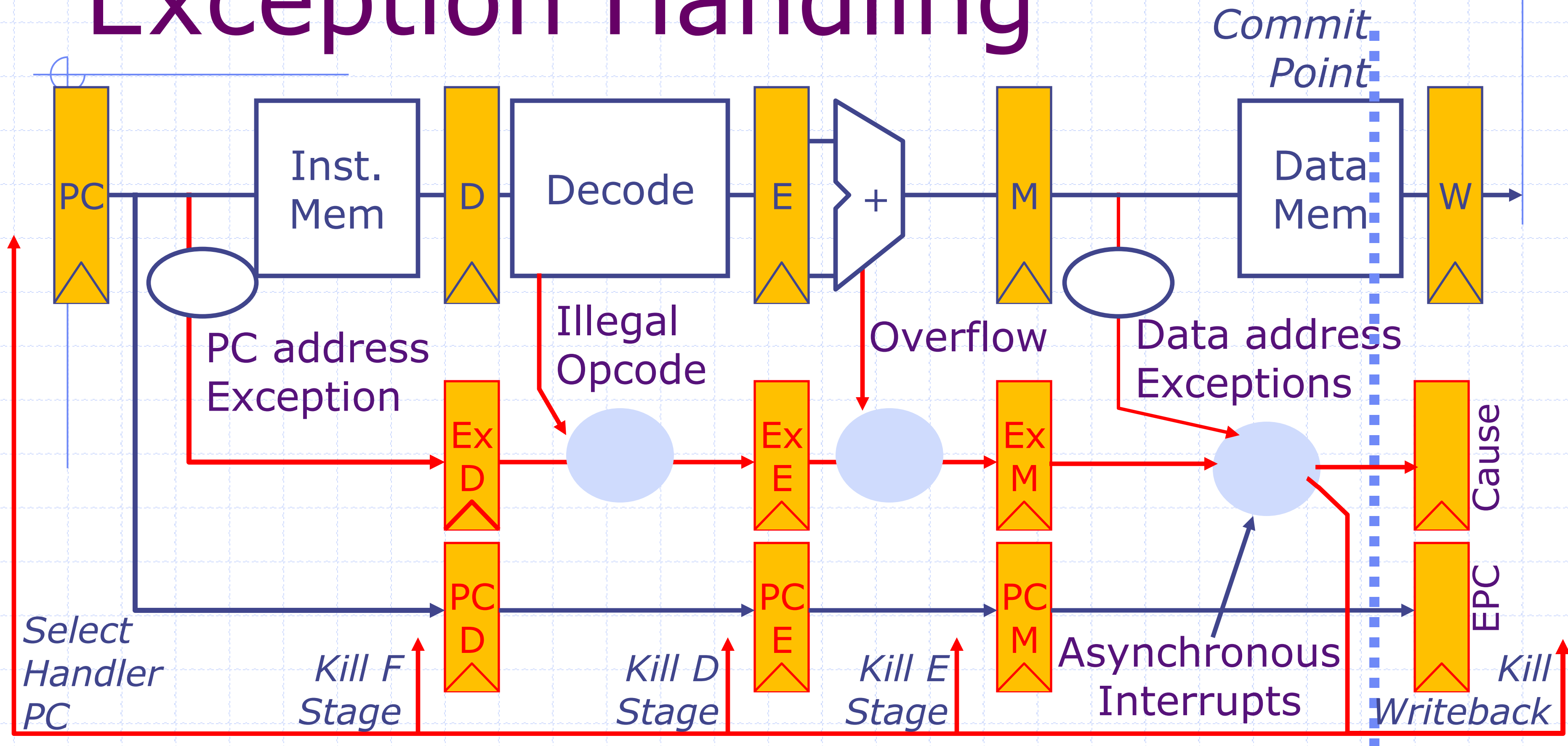
```
r27 = r26 + (r27 << 2);
```

load from address in r27

- The rest of the emulation is straight forward

Exception handling in pipeline machines

Exception Handling



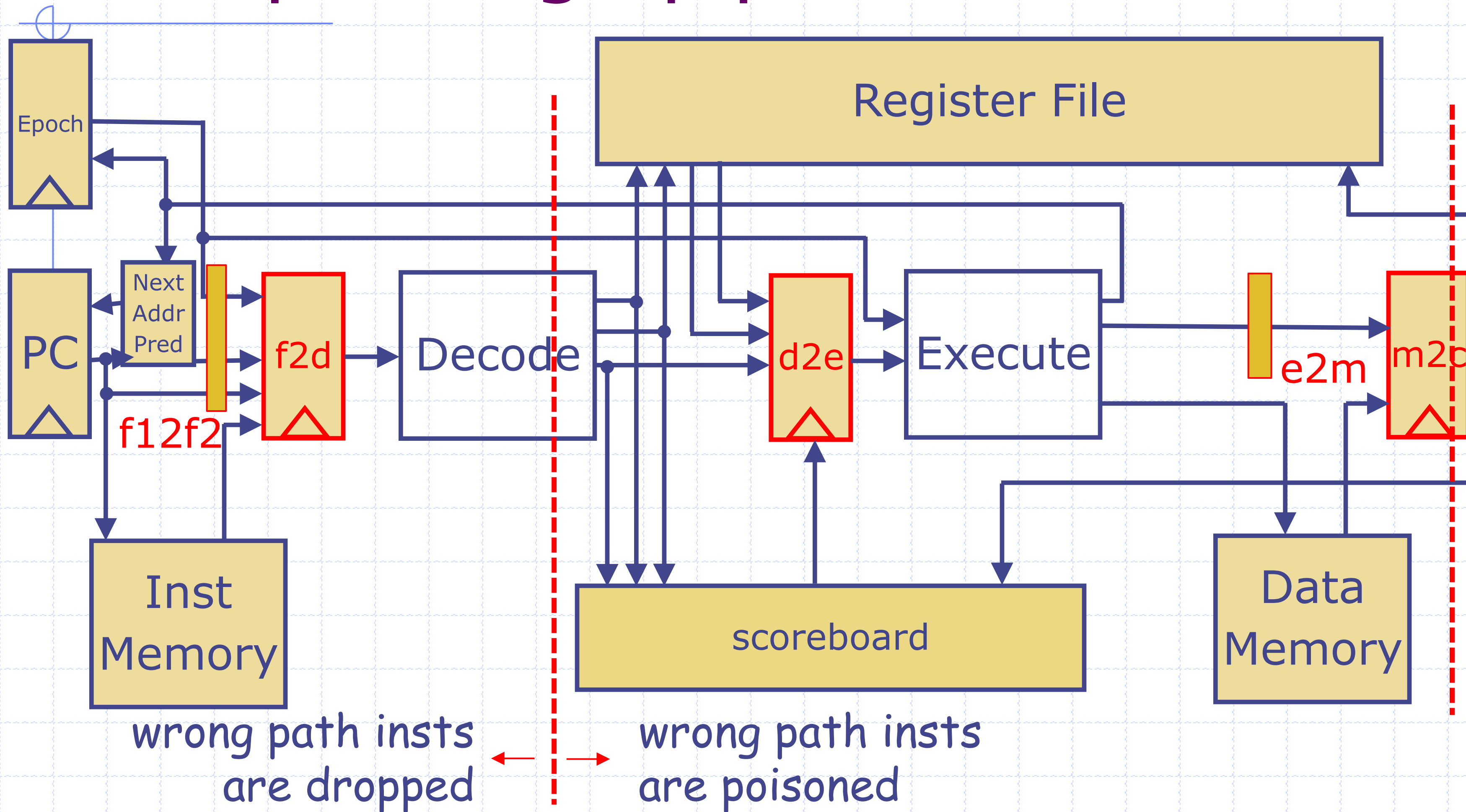
1. An instruction may cause multiple exceptions; which one should we process? from the earliest stage

2. When multiple instructions are causing exceptions; which one should we process first? from the oldest instruction

Exception Handling

- When instruction *x* in $stage_i$ raises an exception, its cause is recorded and passed down the pipeline
- For a given instruction, exceptions from the later stages of the pipeline do not override cause of exception from the earlier stages
- At commit point external interrupts, if present, override other internal interrupts
- If an exception is present at commit: Cause and EPC registers are set, and pc is redirected to the handler PC
  - Epoch mechanism takes care of redirecting the pc

Multiple stage pipeline



This affects whether an instruction is removed from sb in case of an interrupt

Interrupt processing

- Internal interrupts can happen at any stage but cause a redirection only at Commit
- External interrupts are considered only at Commit
- Some instructions, like Store, cannot be undone once launched. So an instruction is considered to have completed before an external interrupt is taken
- If an instruction causes an interrupt then the external interrupt, if present, is given a priority and the instruction is executed again

Interrupt processing at Execute-1

Incoming Interrupt

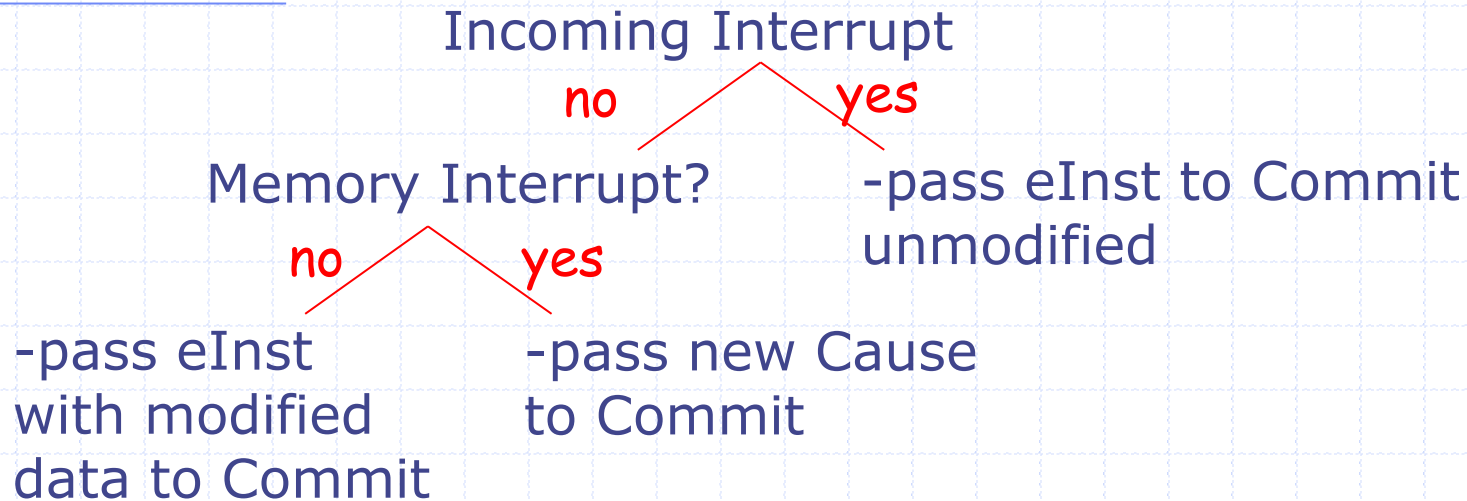
no / yes

-if (mem type) issue Ld/St
-if (mispred) redirect
-pass eInst to M stage

-pass eInst to M stage unmodified

eInst will contain information about any newly detected interrupts at Execute

Interrupt processing at Execute-2 or Mem stage



Interrupt processing at Commit

External Interrupt?

- **no** → Incoming interrupt
  - **no** →
    - -commit
    - -sb.rm
  - **yes** →
    EPC<= pc;
    causeR <= inCause;
    if (inCause after Reg Fetch) sb.rm;
    mode <= Kernel;
    Redirect
- **yes** → Incoming interrupt
  - **no** →
    commit;
    sb.rm;
    EPC<= ppc;
    causeR <= Ext;
    mode <= Kernel;
    Redirect
  - **yes** →
    EPC<= pc;
    causeR <= Ext;
    if (inCause after Reg Fetch) sb.rm;
    mode <= Kernel;
    Redirect

Final comment

- There is generally a lot of machinery associated with a plethora of exceptions in ISAs
- Precise exceptions are difficult to implement correctly in pipelined machines
- Performance is usually not the issue and therefore sometimes exceptions are implemented using microcode