Constructive Computer Architecture

Interrupts/Exceptions/Faults

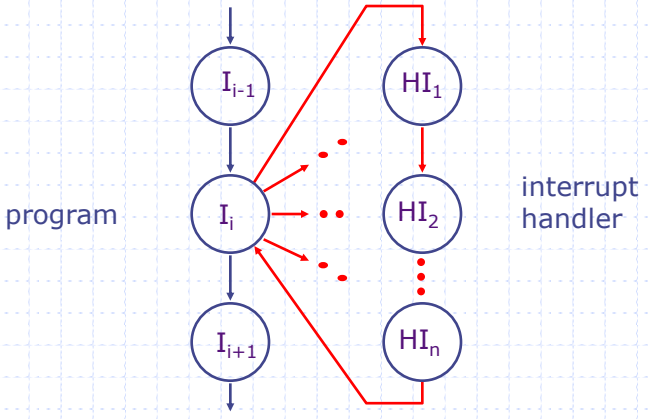
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

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 (a.k.a exceptions, faults and traps)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults, TLB misses, protection violations
  - *traps:* system calls, e.g., jumps 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 (EPC)
  - 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

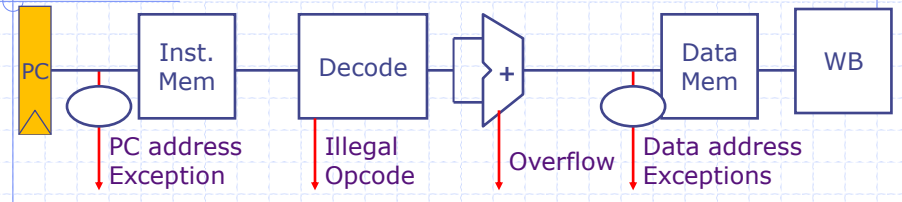
Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction ERET (*return-from-exception*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Interrupts

- A synchronous interrupt is caused by a *particular instruction* and causes a control hazard
  - requires undoing the effect of one or more partially executed instructions. Comes in two varieties:
- Exception: The instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - information about the exception has to be recorded and conveyed to the exception handler
- Faults (aka Trap): Like a system call and the instruction is considered to have been completed
  - requires a special jump instruction involving a change to privileged kernel mode

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

Additional Features for Exceptions/Faults

- register: `epc`
  - holds `pc+4` of instruction that causes exception/fault
- instruction: `eret`
  - returns from an exception/fault handler sub-routine using epc

As an example consider a complex instruction which may be implemented in SW

`mult ra, rb`

- causes a fault, sets epc to pc+4
- jumps to the exception handler for `mult`
- the `mult` instruction is considered to have been completed

Software Considerations

```
00001004 <handler>:
    1004:    00000000     li $t1, 1010
             // all exceptions jump to 1004
    1008:    00000000     addiu $t1, causeR
             // causeR contains 0 for mul, 4 for div, etc
    100c:    00000000     jr $t1
    1010:    08000408     j 1020 // <mult_excep>
    1014:    08000408     j 1060 // <div_excep>
    ...
00001020 <mult_excep>:
    1020:    24890000     addiu $t1,$a0,0
    1024:    24aa0000     addiu $t2,$a1,0
    ...
    104c:    42000018     eret
...
00001124 <main>:
    ...
    11a0:    00850018     mult $a0,$a1
```

We need to change the interfaces to handle exceptions wherever necessary

For example,

- a memory request will return a 2-tuple <mem-reponse, mException>;
- decoder will have to recognize new instructions `mult, eret, ...`

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) fcMULT  = 6'b011000;
Bit#(5) rsERET  = 5'b10000;
```

```
typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br,
Mult, 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);
  DecodedInst dInst = ?; ...
  opFUNC:                                  begin
    case (funct) ...
      fcMULT:
         dInst.iType   = Mult;
         dInst.brFunc  = AT;
         dInst.rDst    = Invalid;
         dInst.rSrc1   = validReg(rs);
         dInst.rSrc2   = validReg(rt); end
  opRS:                                    begin
    if(rs==rsERET)
      dInst.iType = ERet;
      dInst.brFunc  = AT;
      dInst.rDst    = Invalid;
      dInst.rSrc1   = Invalid;
      dInst.rSrc2   = Invalid; end
  return dInst;
endfunction
```


Branch Address Calculation

```
function Addr brAddrCalc(Addr pc, Data val,
                         IType iType, Data imm, Bool taken, Addr epc);
  Addr pcPlus4 = pc + 4;
  Addr targetAddr = case (iType)
    J  : {pcPlus4[31:28], imm[27:0]};
    Jr : val;
    Br : (taken? pcPlus4 + imm : pcPlus4);
    Mult: h'1004;  // Interrupt handler
    ERet: epc;
    Alu, Ld, St, Unsupported: pcPlus4;
  endcase;
  return targetAddr;
endfunction
```

Execute Function

```
function ExecInst exec(DecodedInst dInst, Data rVal1,
                       Data rVal2, Addr pc, Addr epc);
  ...
  let brAddr = brAddrCalc(pc, rVal1, dInst.iType,
   validValue(dInst.imm), brTaken, epc);
  ...
  eInst.brAddr = ... brAddr ...;
  ...
  return eInst;
endfunction
```

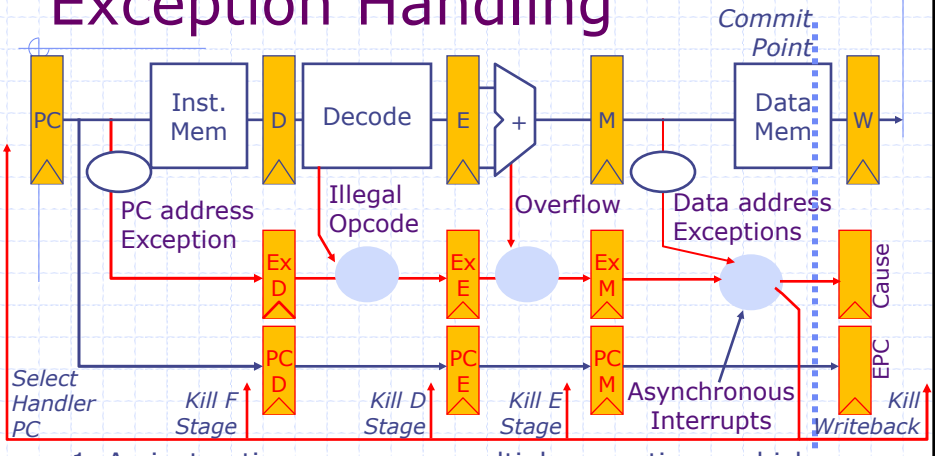
With these changes the single-cycle machine will handle exceptions

One-Cycle SMIPS

```
rule doExecute;
  let inst = iMem.req(pc); let dInst = decode(inst);
  let rVal1 = rf.rd1(validRegValue(dInst.src1));
  let rVal2 = rf.rd2(validRegValue(dInst.src2));
  let eInst = exec(dInst, rVal1, rVal2, pc, epc);
  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);
  pc <= eInst.brTaken ? eInst.addr : pc + 4;
  if(eInst.iType == Mult)              begin
    epc <= eInst.addr; causeR <= 0; end
endrule endmodule
```

Exception handling in pipeline machines

Exception Handling

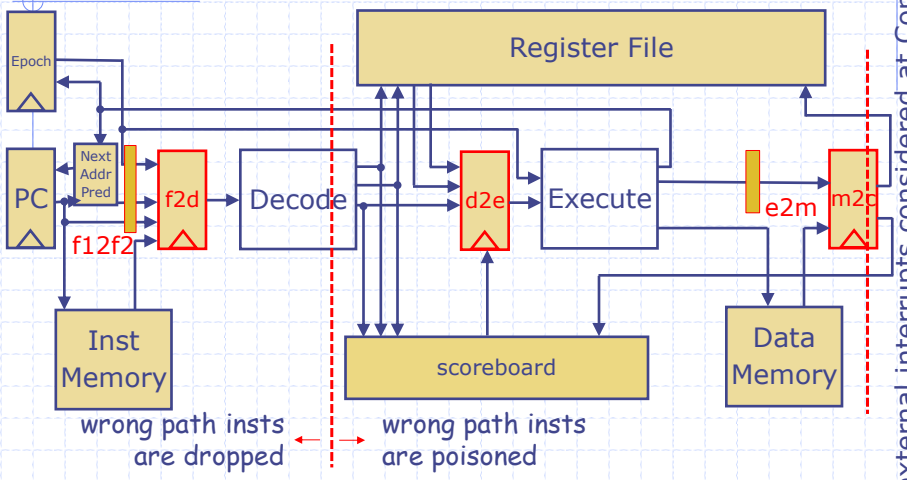


1. An instruction may cause multiple exceptions; which one should we process? from the earliest stage
2. When multiple instructions are causing exceptions; which one should we process first? from the oldest instruction

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 cause

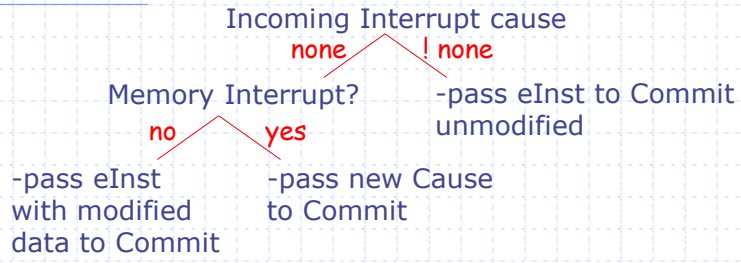
none ! none

-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