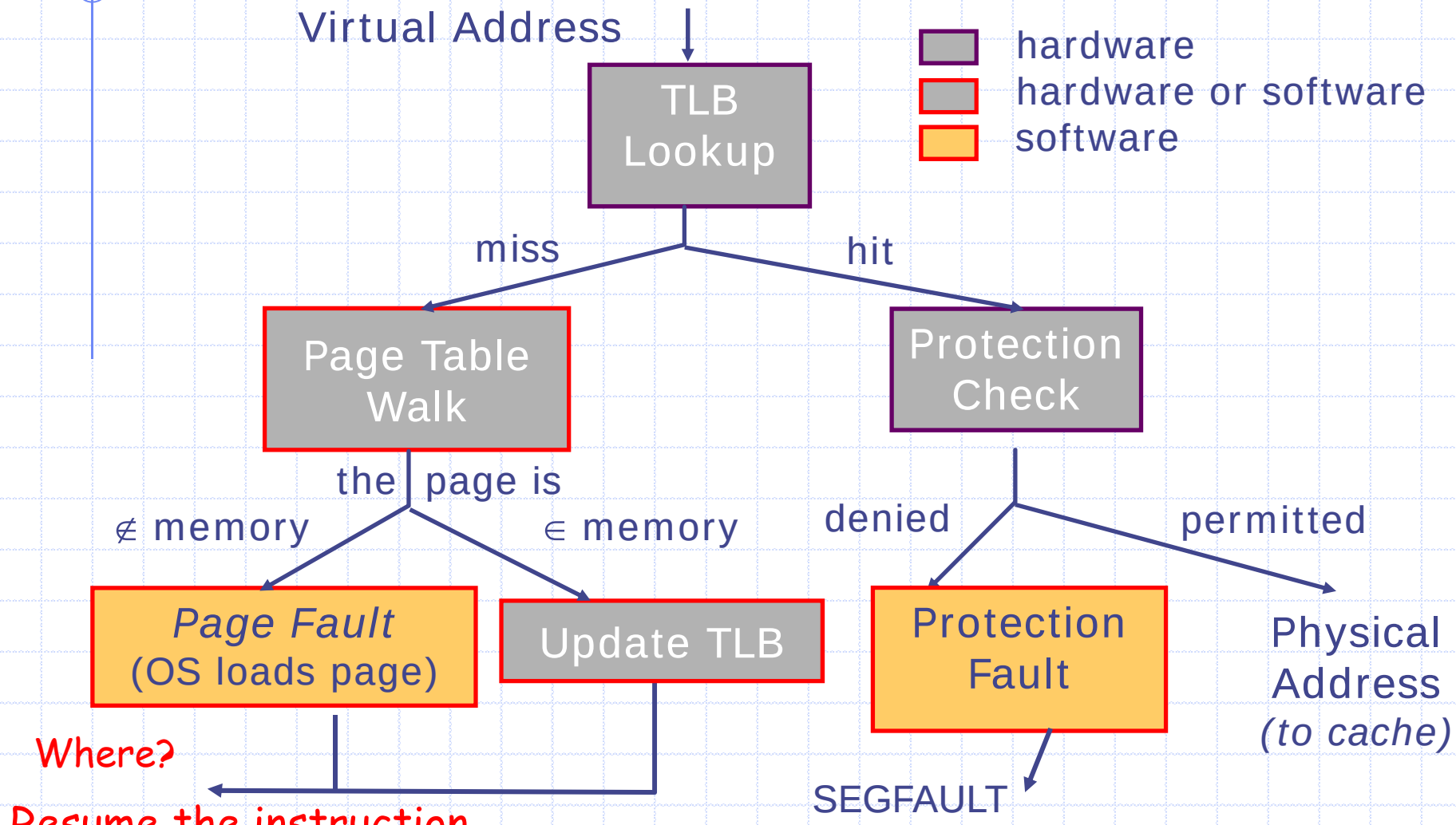
Constructive Computer Architecture

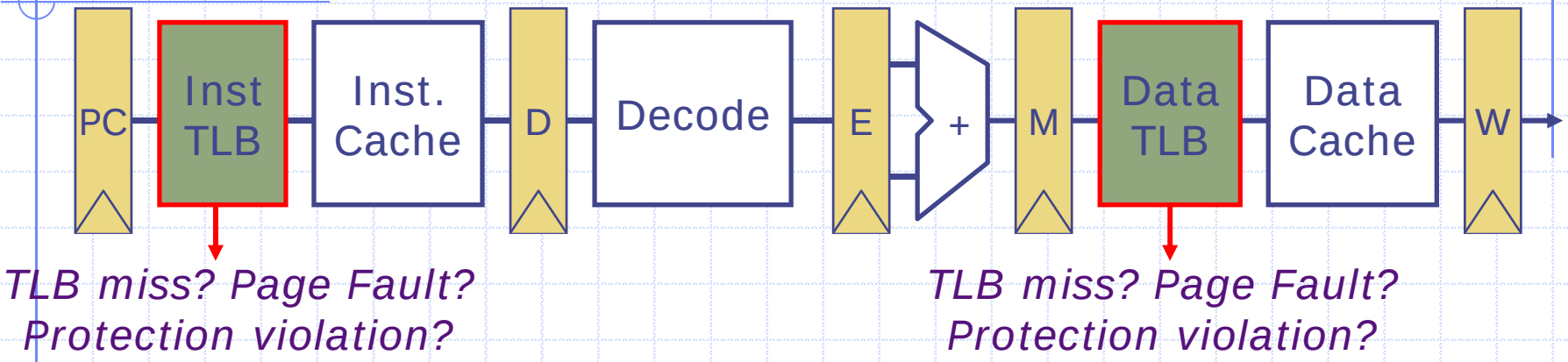
Virtual Memory and Interrupts

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

Address Translation:
putting it all together

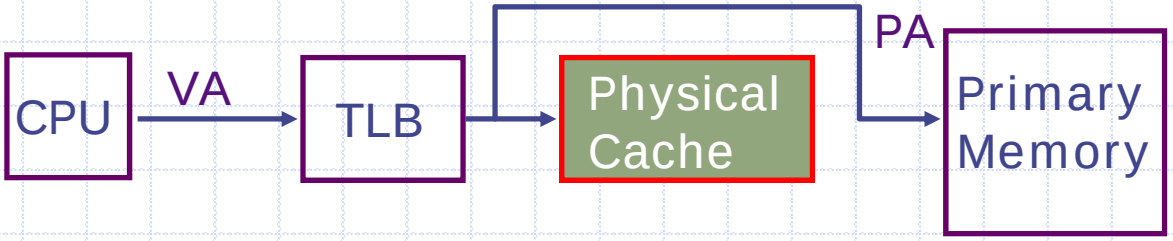


Address Translation in Pipeline Machines

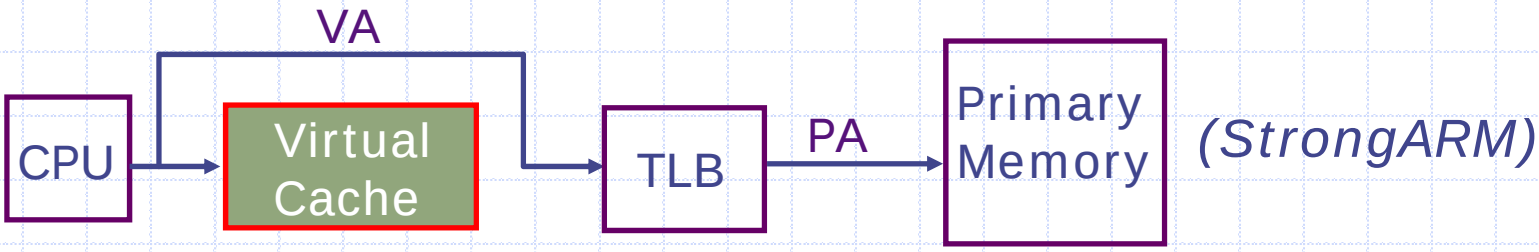


- Software handlers need a *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Methods to overcome the additional latency of a TLB:
  - slow down the clock
  - pipeline the TLB and cache access
  - √ virtual address caches
  - √ parallel TLB/cache access

Physical *vs* Virtual Address Caches?



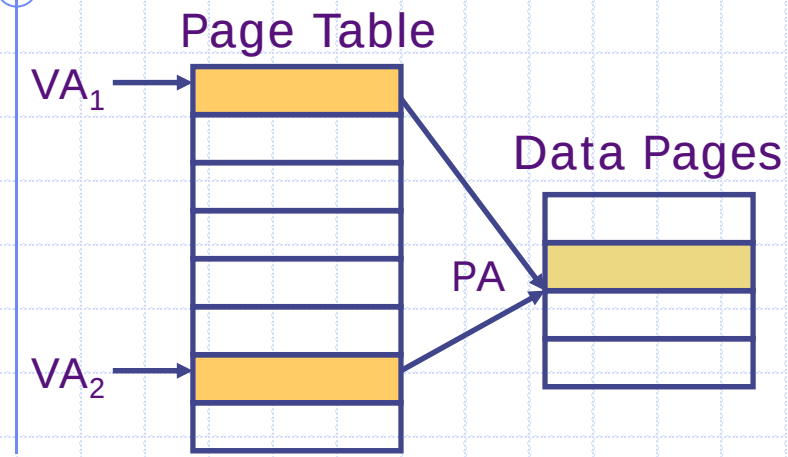
Alternative: place the cache before the TLB



(StrongARM)

- One cycle in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)

Aliasing in Virtual-Address Caches



Two virtual pages share one physical page

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

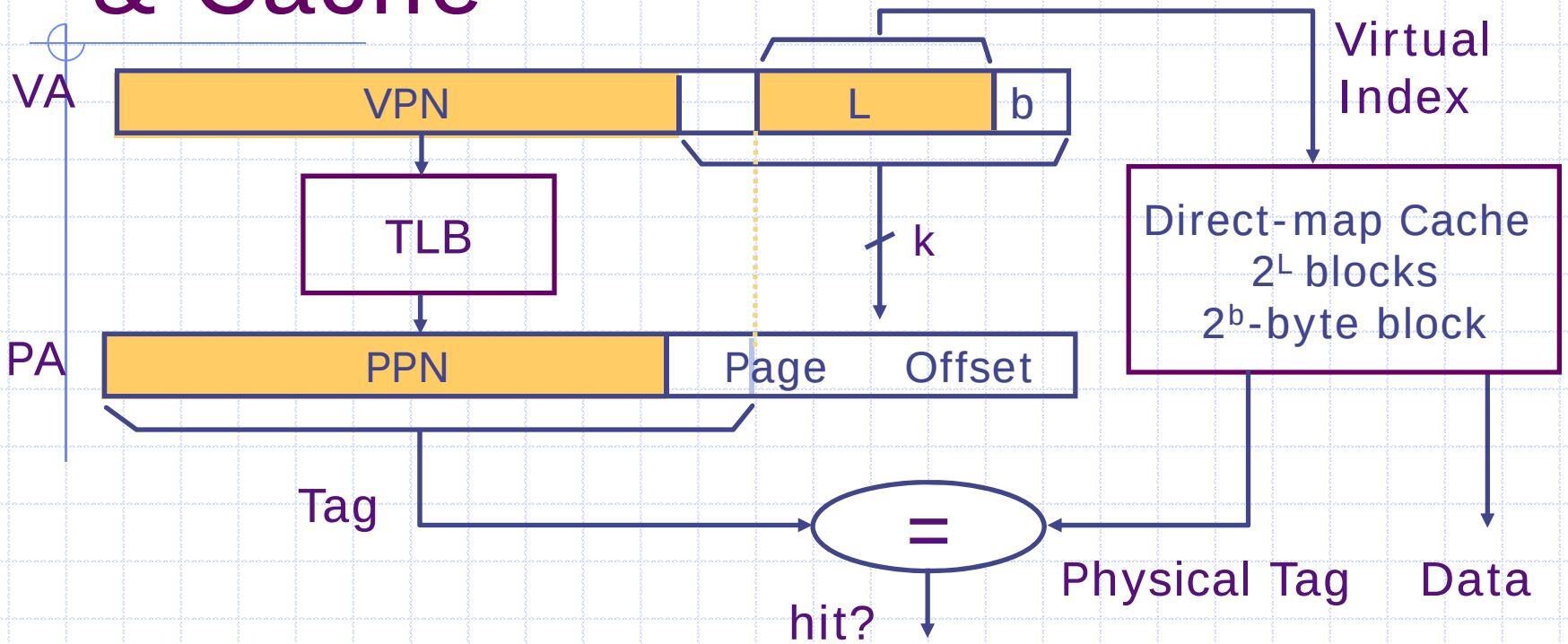
Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)

Concurrent Access to TLB & Cache



Index L is available without consulting the TLB

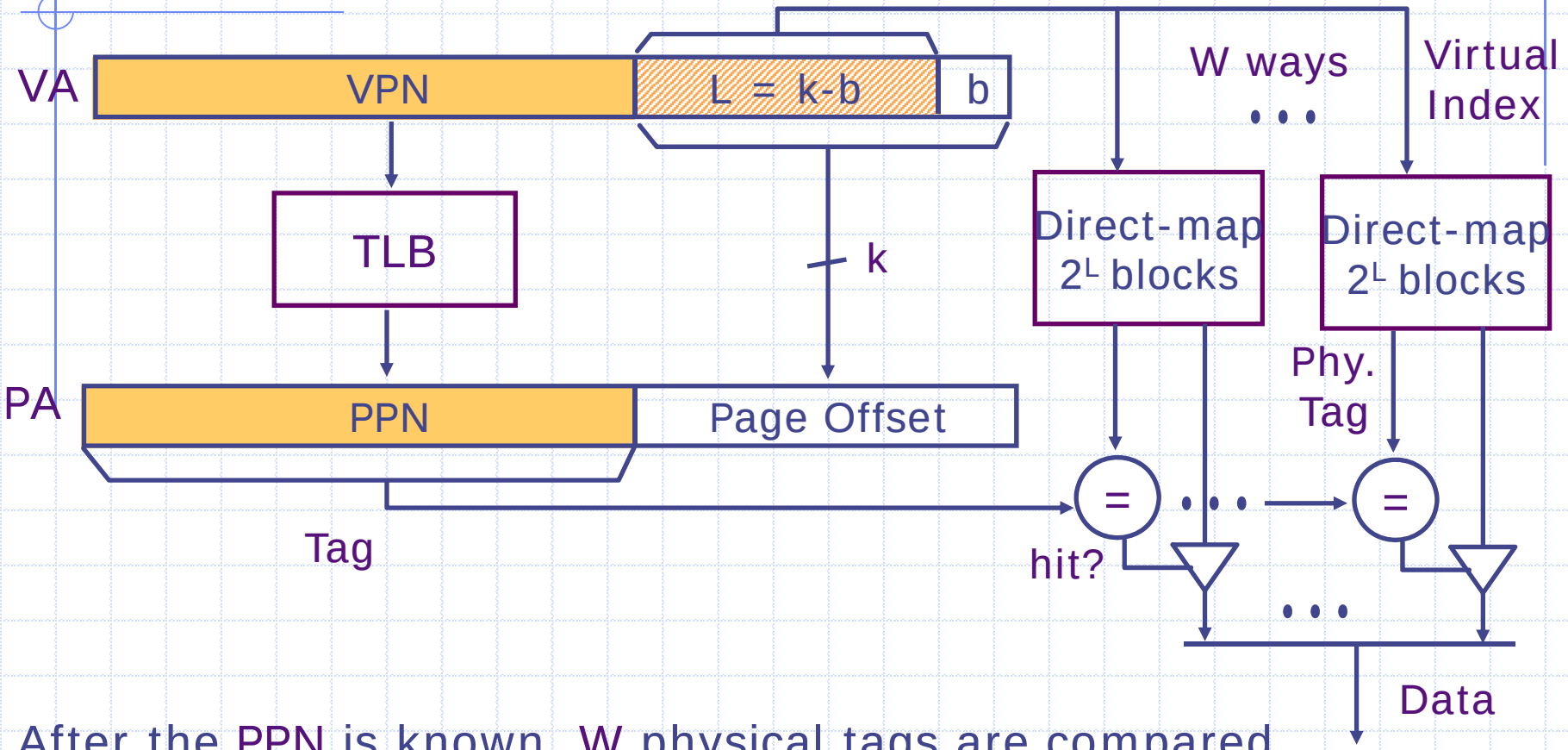
⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

Cases: L + b = k L + b < k

L + b > k what happens here? Partially VA cache!

Virtual-Index Physical-Tag Caches: Associative Organization

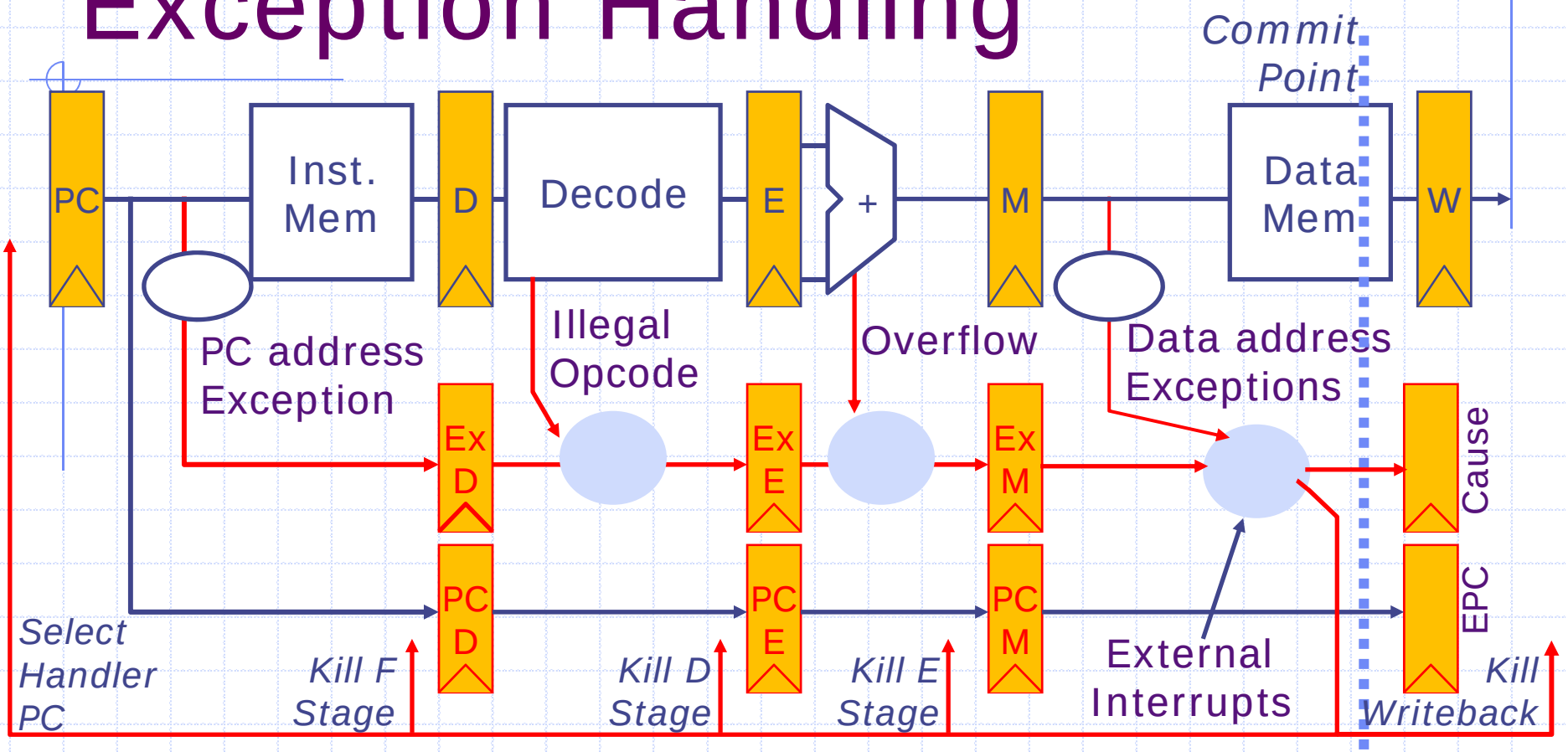


After the PPN is known, W physical tags are compared

Allows cache *size* to be greater than 2^{L+b} bytes

Exception handling in a pipeline machine

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

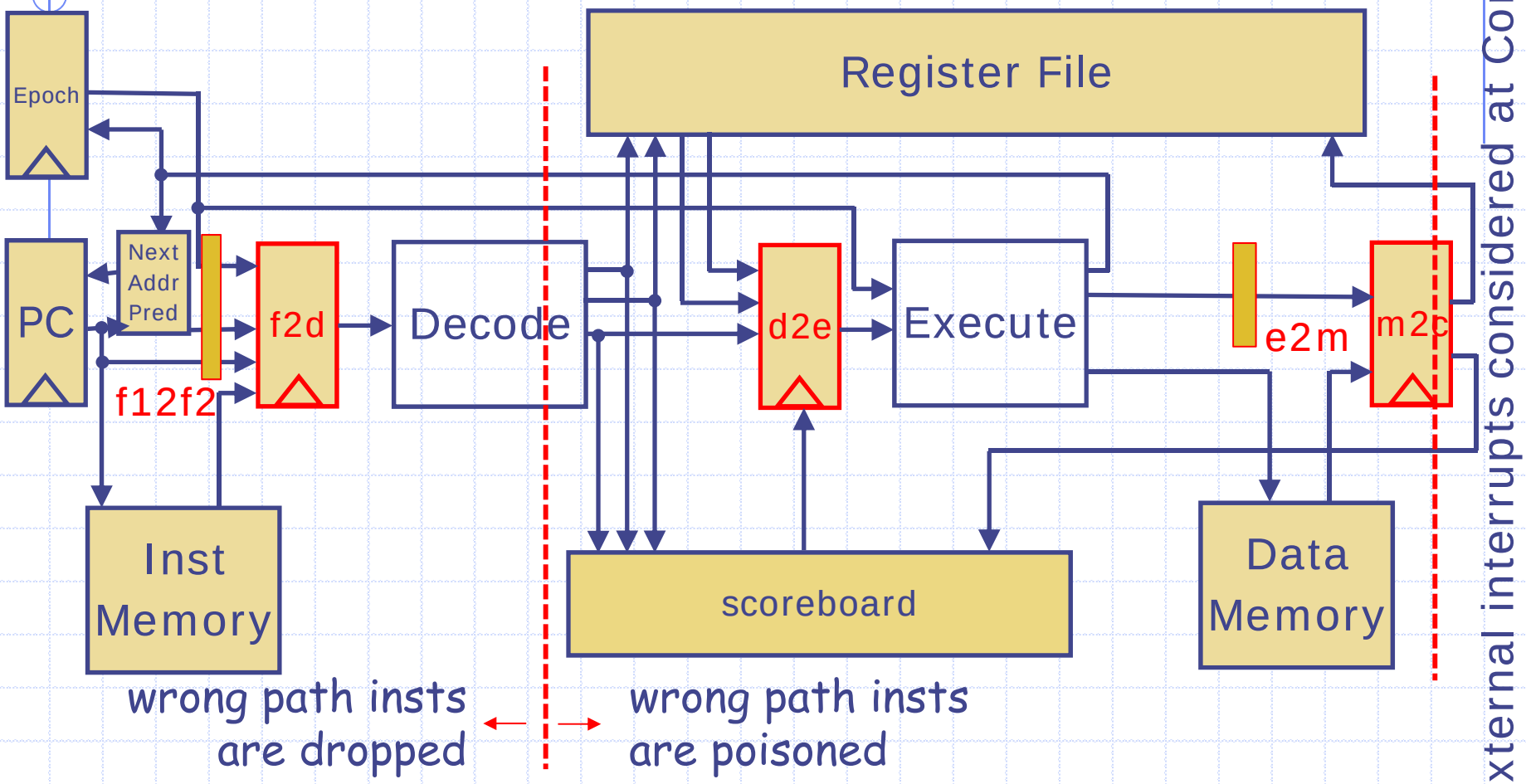
Interrupt processing

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

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
- 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

Killing vs Poisoning



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

Interrupt processing at Execute

Incoming Interrupt

no

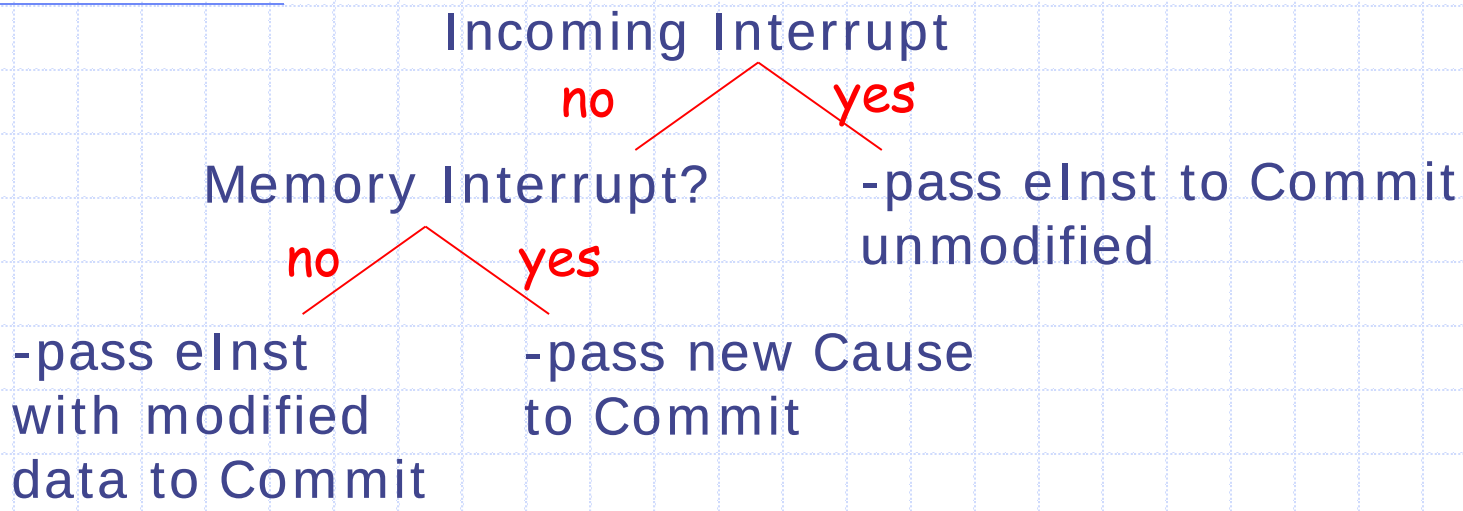
yes

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

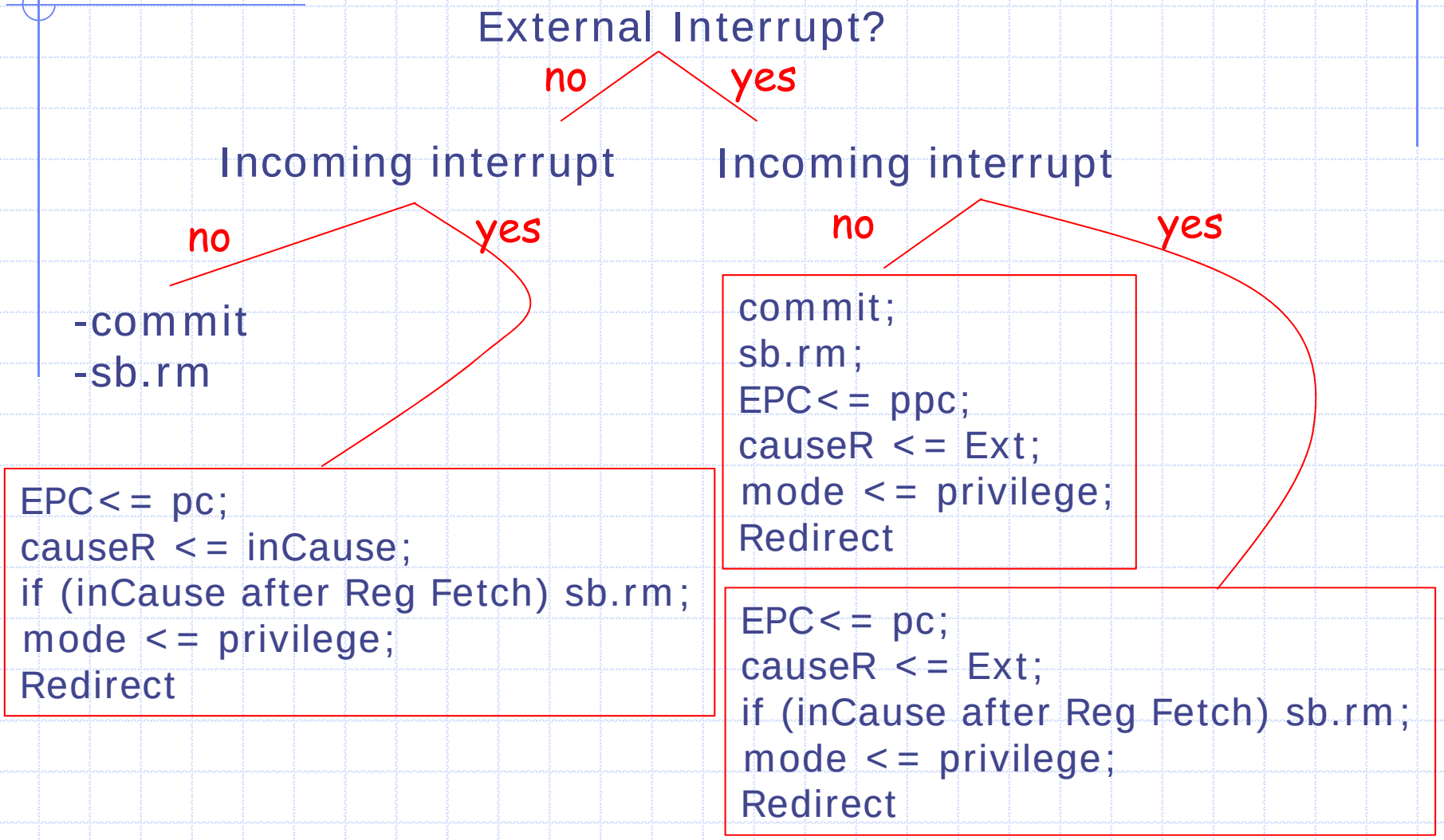
-pass eInst to M stage unmodified

eInst will contain information about any newly detected interrupts at Execute

Interrupt processing at Memory stage



Interrupt processing at Commit



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

The code slides follow

One-Cycle SMIPS

```
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;
```