ECE 2300
Digital Logic & Computer Organization

Spring 2025

Exceptions
Input/Output

Cornell University

Announcements

- HW 8 due tomorrow

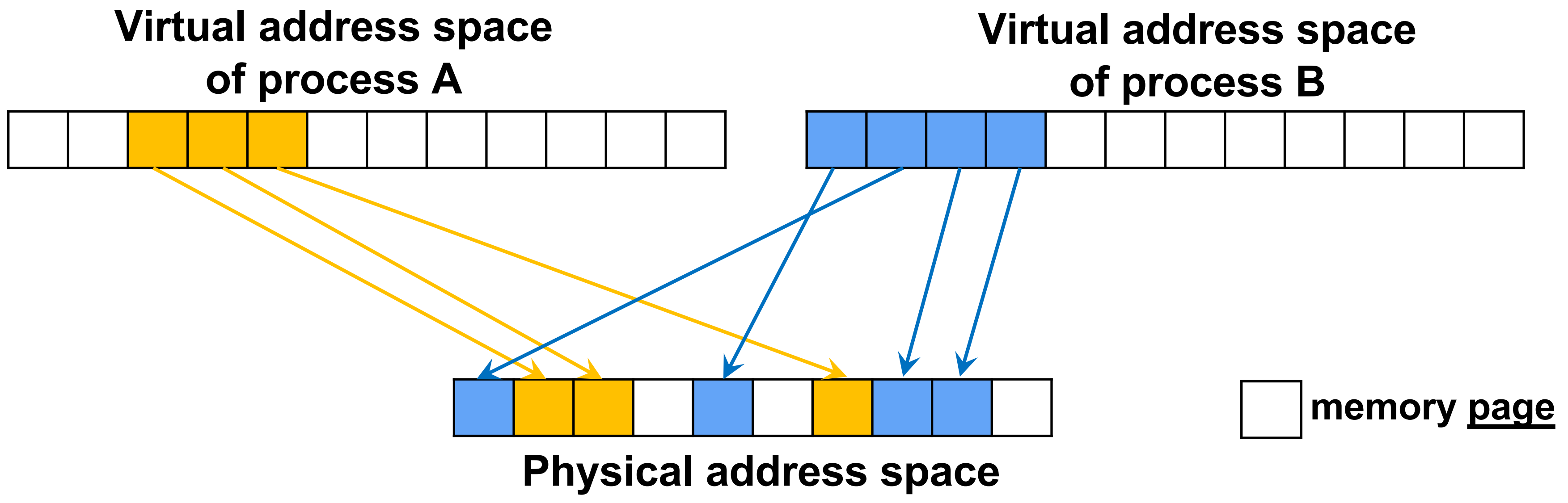
- Lab 5 due next Monday
  – Download latest zip file from CMS

- Final exam: Saturday May 10th, 9am @ PHL 101, 100 mins
  – Cumulative; More on coverage next lecture
  – Sample final is posted on CMS
  – TA-led review session on Thursday May 8, 7:30pm
  – OH schedule in final week will be announced soon

Virtual Memory (True or False)

- Program counter (PC) holds a virtual address
- Different processes (running programs) share the same page table
- The TLB is typically implemented using DRAM
- Page fault occurs when there is a TLB miss followed by a Page Table miss

Review: Virtual Memory Concepts

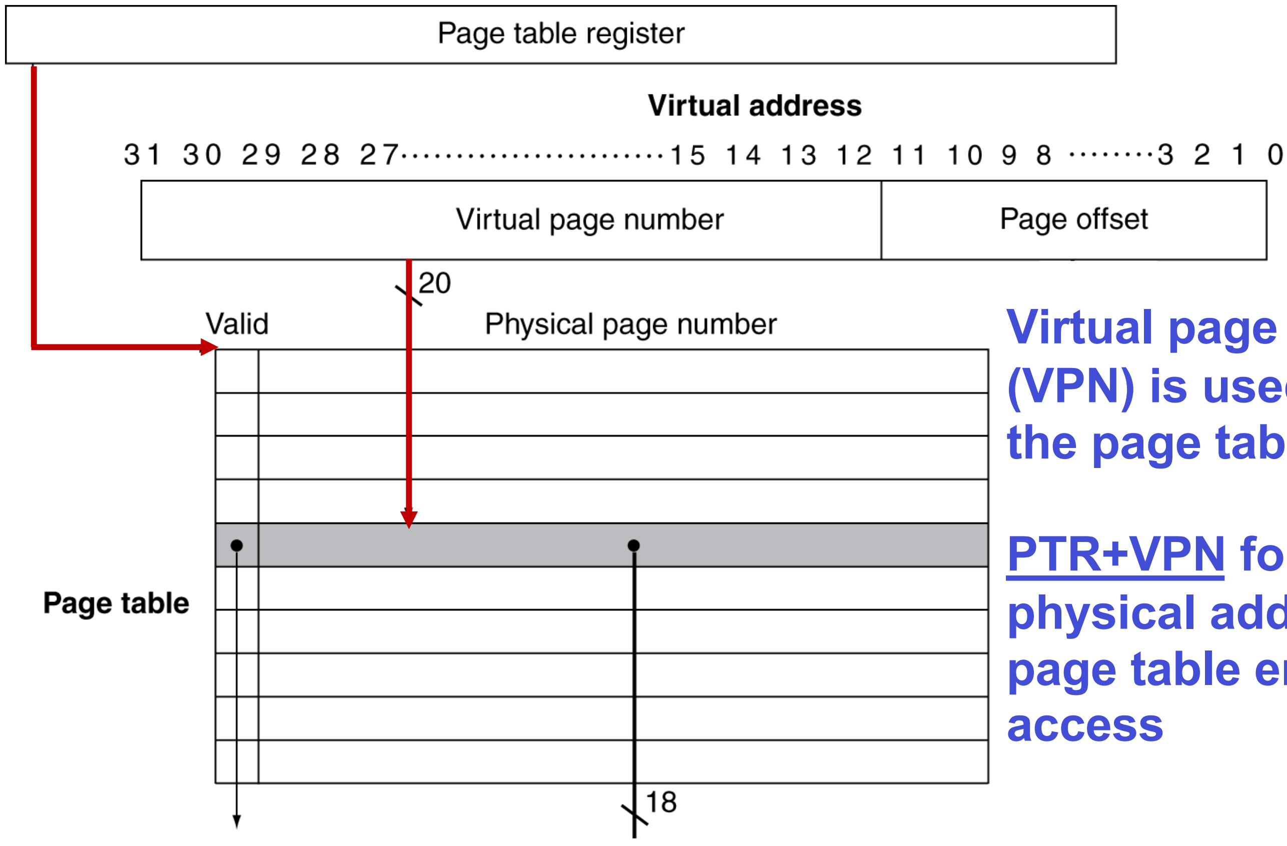

Virtual address space of process A
Virtual address space of process B
memory page
Physical address space

- **Each process (active instance of a program) has its own virtual address space**
  - **Allows developers to write software as if it owns all of the computer's memory**
  - **<u>Each process also has its own page table</u>**

- **The virtual page to physical page mapping is <u>dynamically</u> managed by the OS**

Review: Page Table Access

The Page Table Register (PTR) is a special CPU register for locating the page table in the physical MM

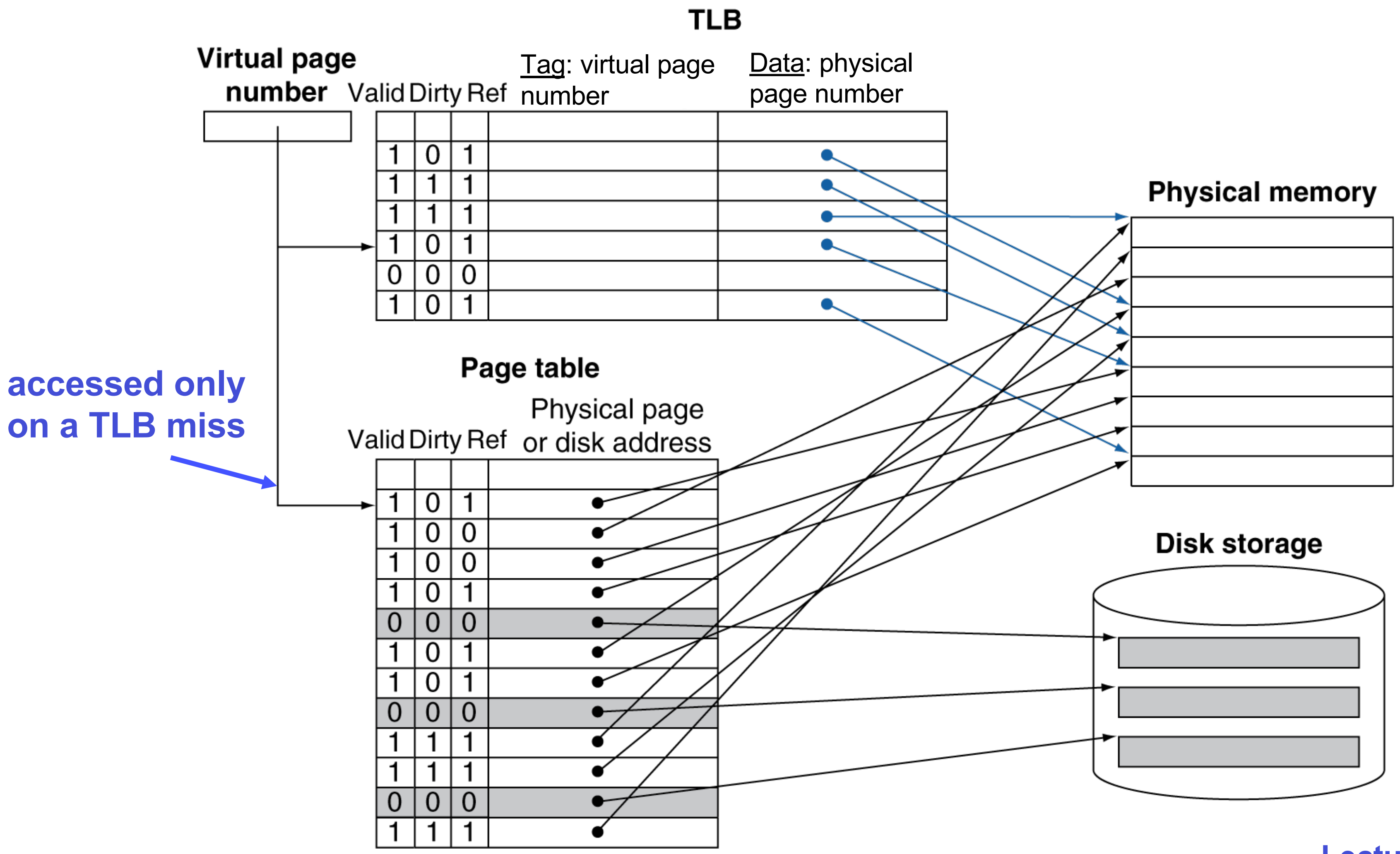


Virtual page number (VPN) is used to index the page table;

<u>PTR+VPN</u> form the physical address of the page table entry (PTE) to access

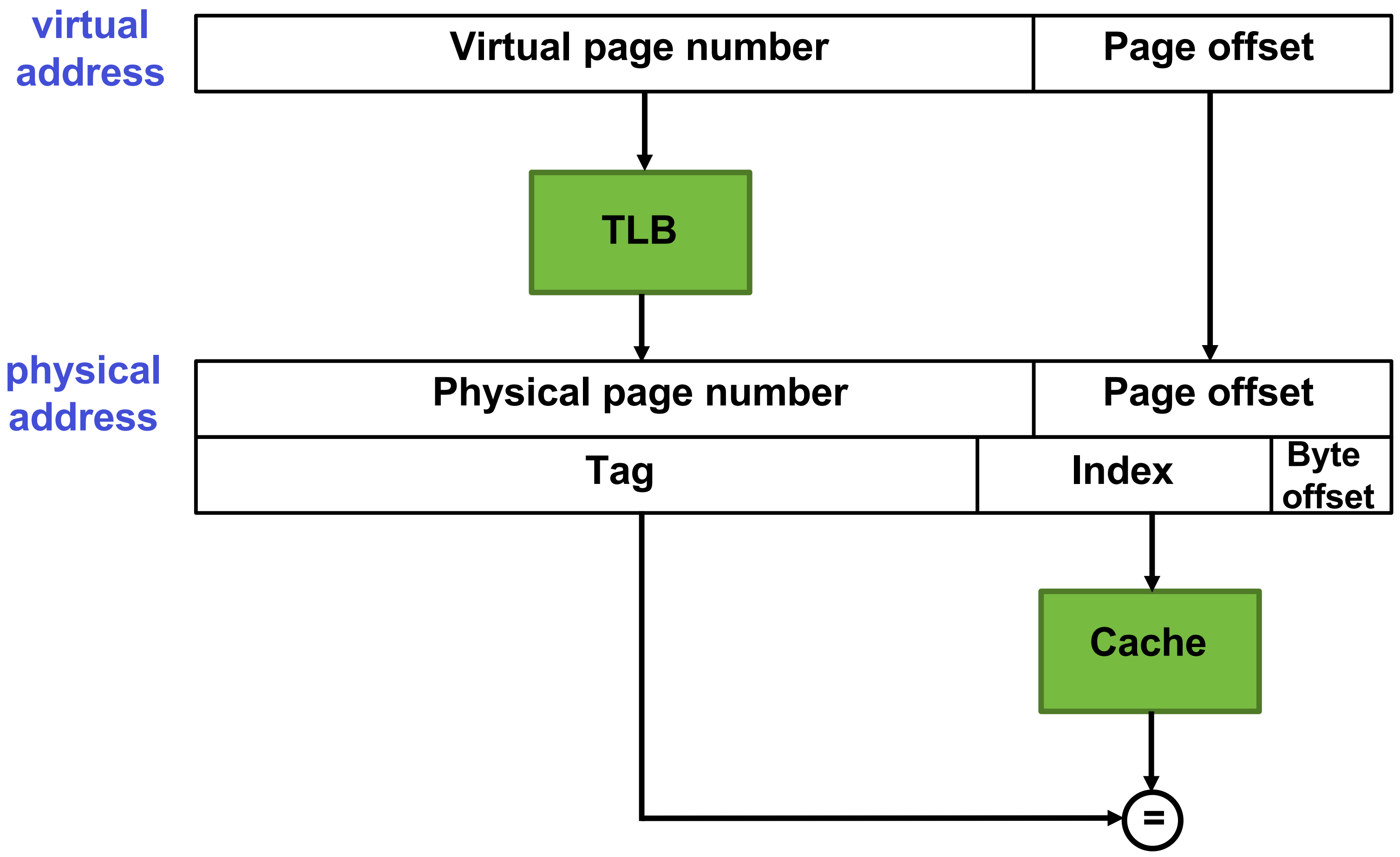
Review: Translation Lookaside Buffer (TLB)

- Small cache of recently accessed PTE (typically 16-512 entries, fully associative)



Accessing the TLB and the Cache

- Cache usually uses physical addresses since it holds a subset of what is in MM



Analogy for TLB

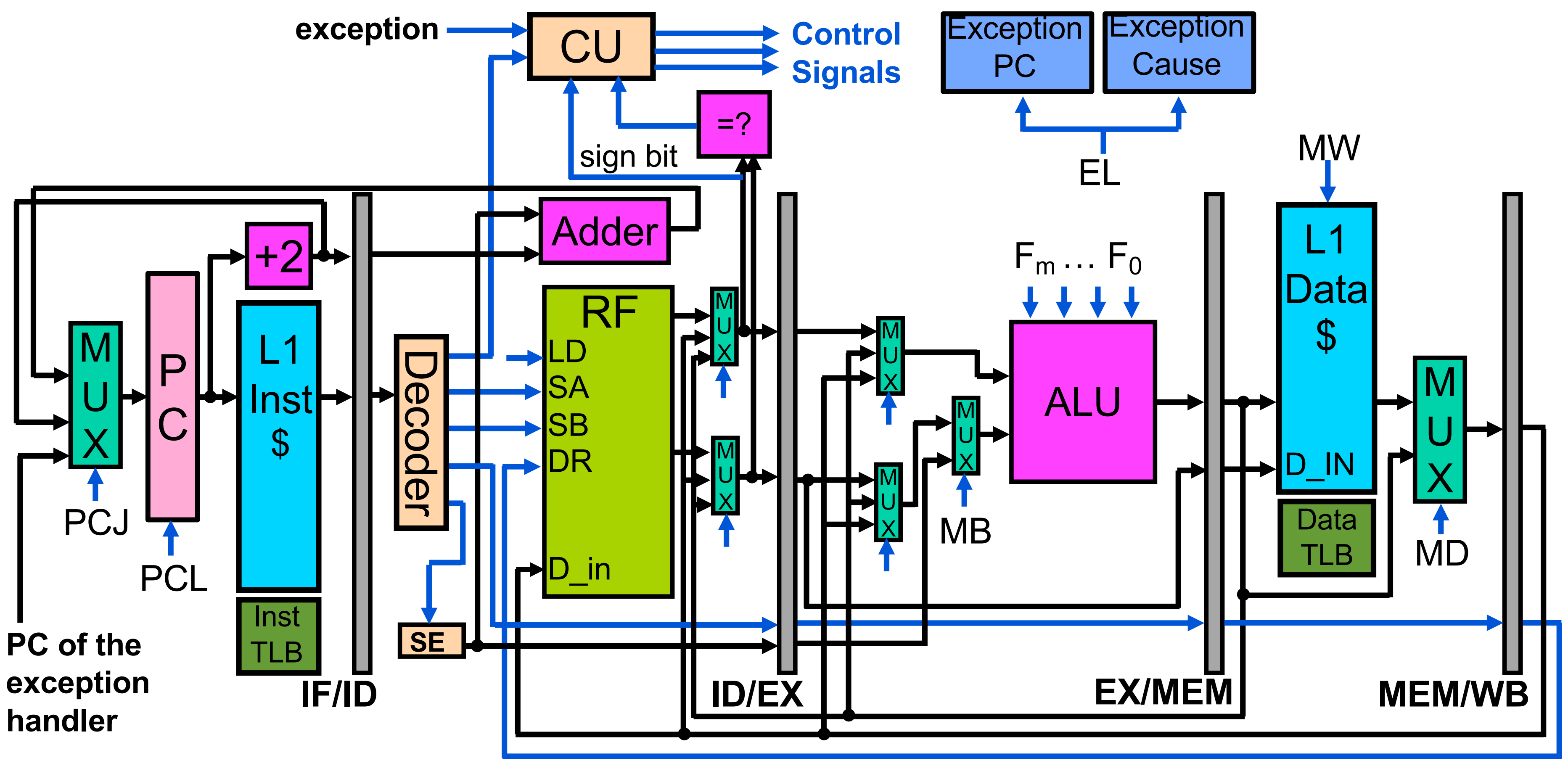
Exceptions and Interrupts

- Useful methods for signaling the CPU that some event has occurred that requires action
  - In response, the CPU may *suspend* the running program in order to handle the exception/interrupt
- Exceptions are used to handle conditions that arise when executing instructions on the processor
  - Detected by the processor itself
- Interrupts are used to handle (asynchronous) events external to the processor
  - I/O device request, external error or malfunction

Why are Exceptions Useful?

- Handle unexpected events
  - Overflow, divide-by-zero, invalid opcode, memory protection violation, etc.
- Handle page faults
- Allow user programs to get service from the OS
  - A system call creates an exception that kicks out the user program and transfers control to exception handler
- An exception handler is OS-managed code that responds to exceptions and dispatches the right OS service routine based on the exception type

Pipeline with Exception Handling

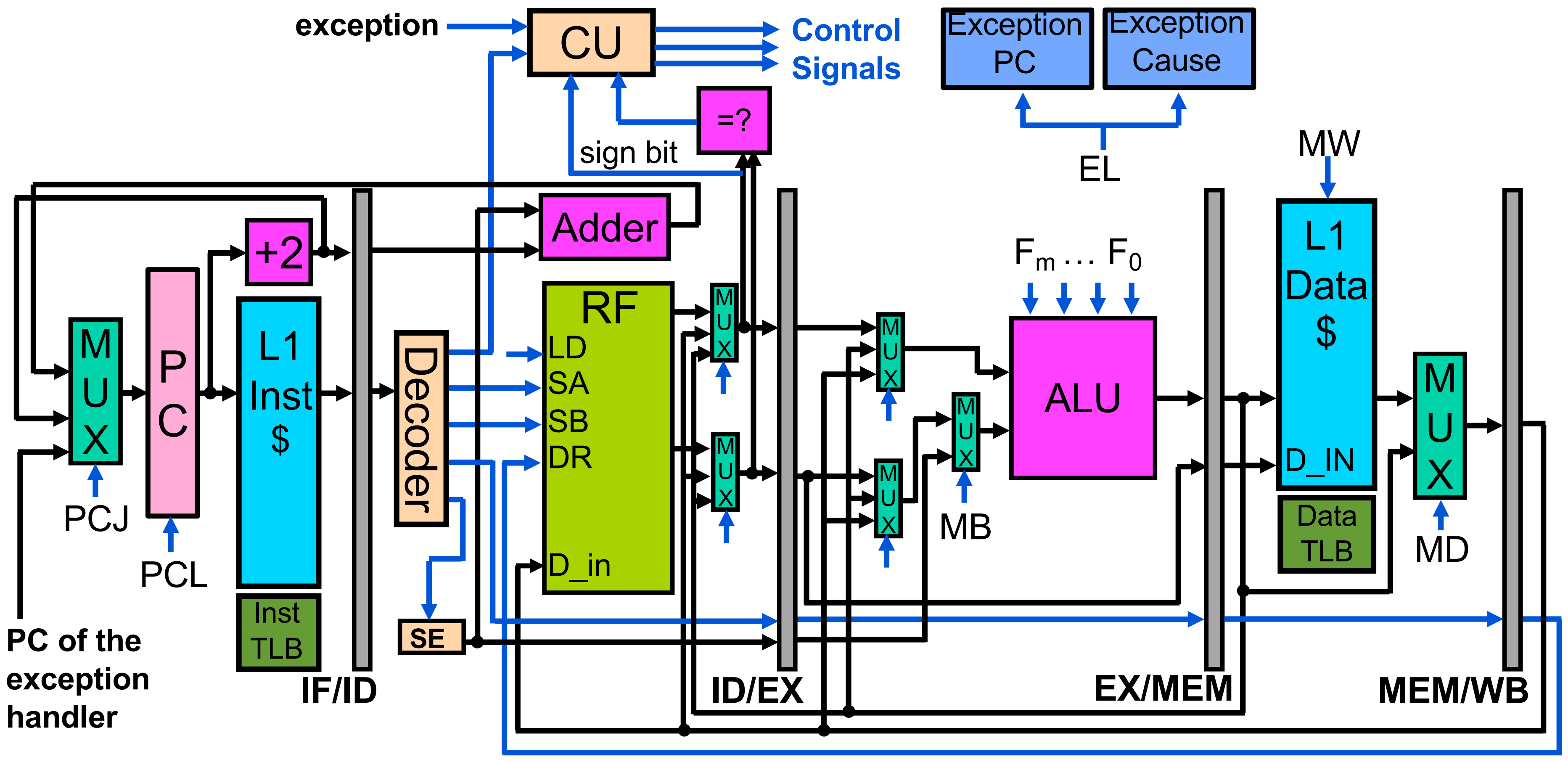


Pipeline with Exception Handling

When an Exception signal is raised

- The control unit (CU) sets the *Cause* of the exception and *Exception PC* (with the address of the faulting instruction)
- All instructions before the exception complete
- The *faulting instruction* (that causes the exception), and any behind it in the pipeline, are turned into NOPs
- The PC of the first instruction in the exception handler code is loaded into the PC register

Instruction Page Fault

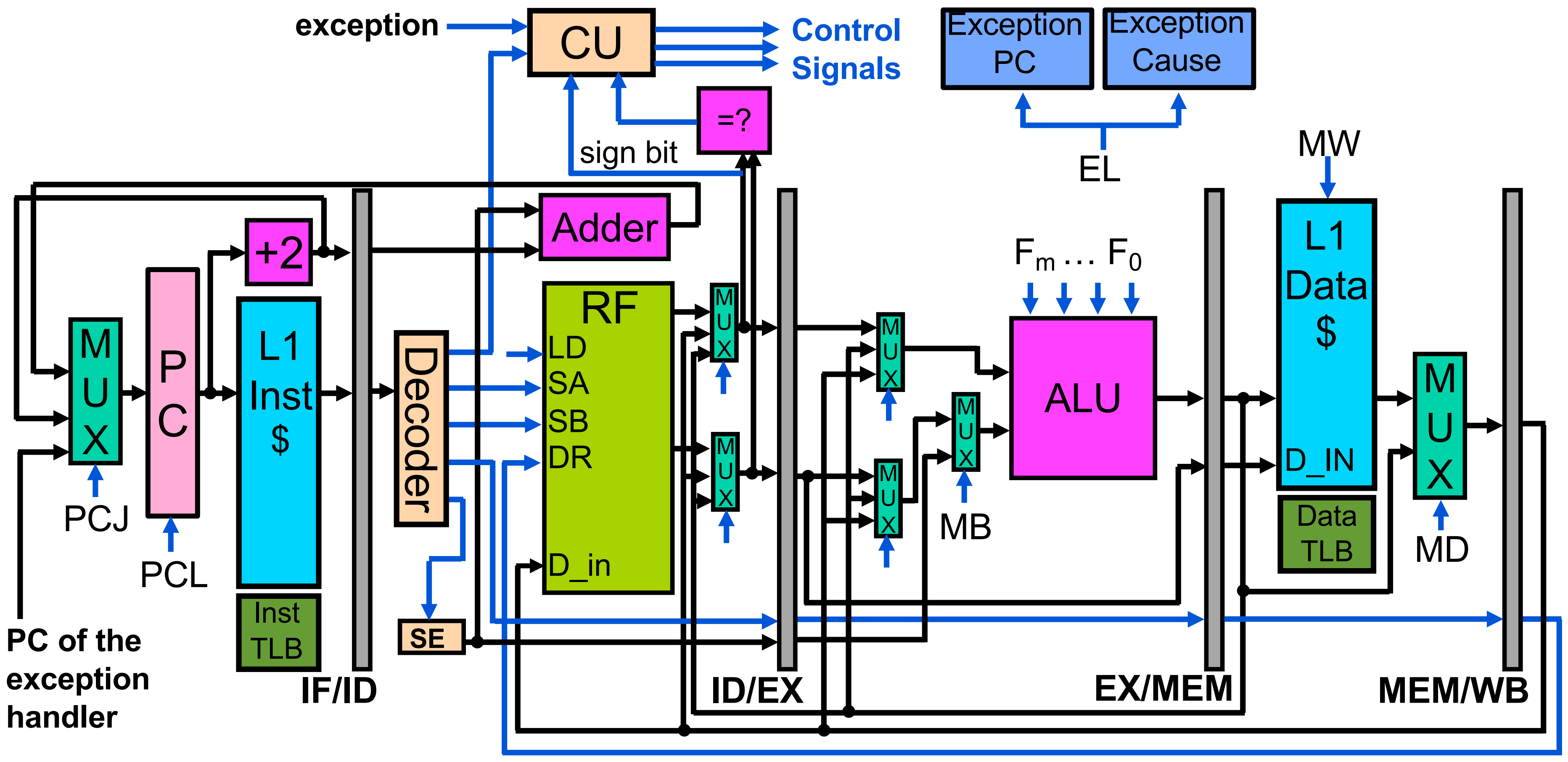


[SUB R5,R5,R7]
not in main memory

LW R4,0(R1)

ADD R1,R2,R3

Instruction Page Fault

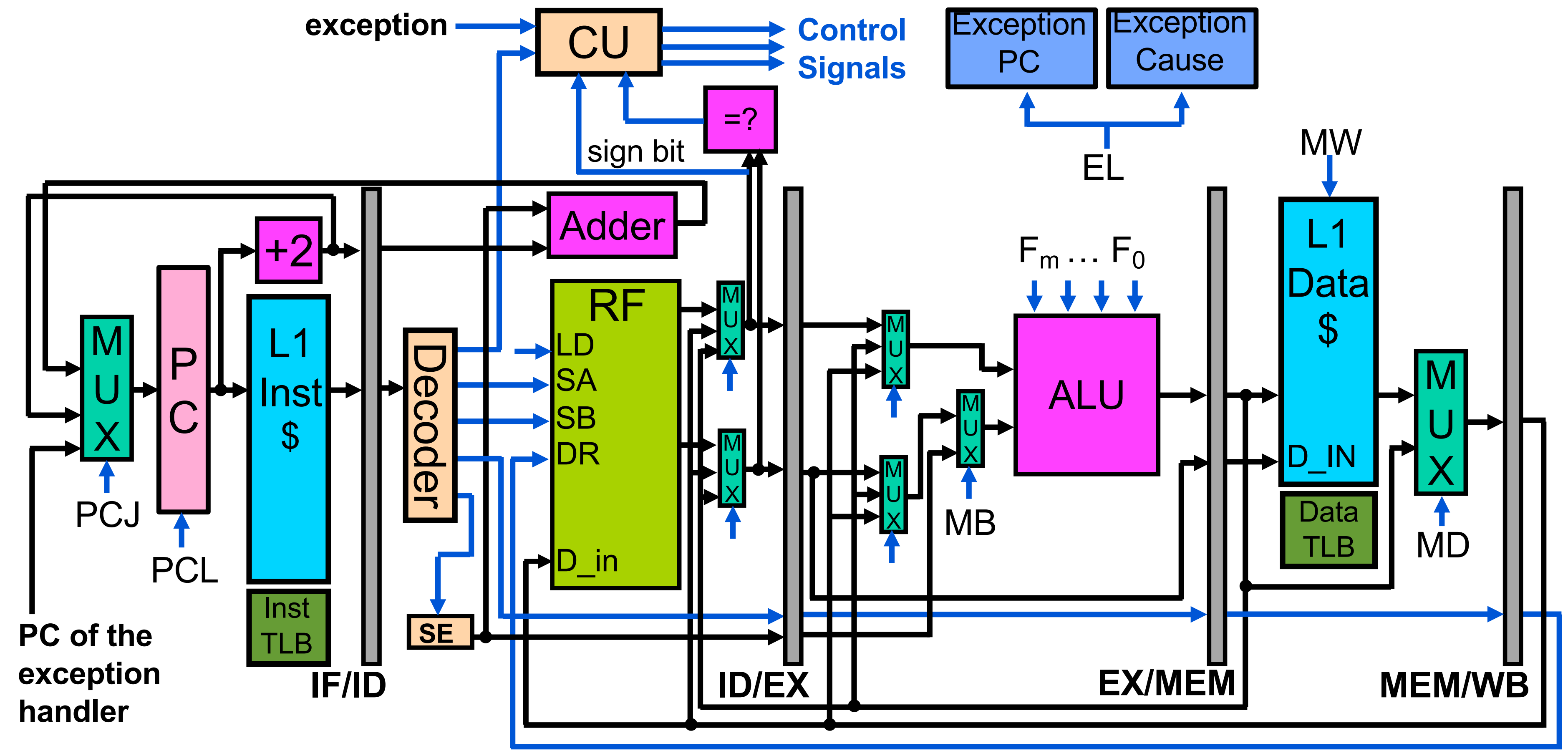


[SUB R5,R5,R7]
<TLB miss>

LW R4,0(R1)

ADD R1,R2,R3

Instruction Page Fault



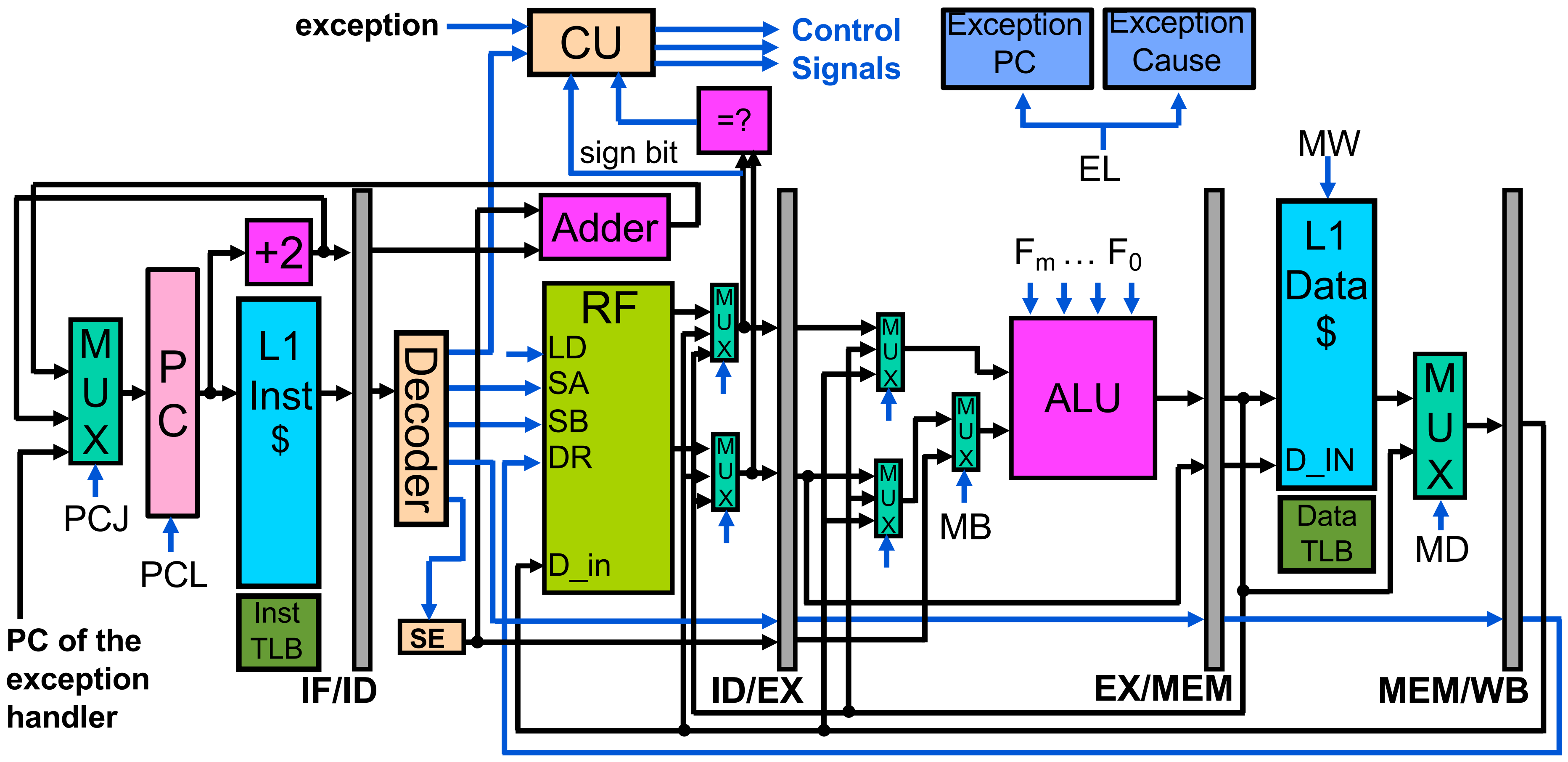
[SUB R5,R5,R7]

<stall>

<access page table>

<LW and ADD have completed>

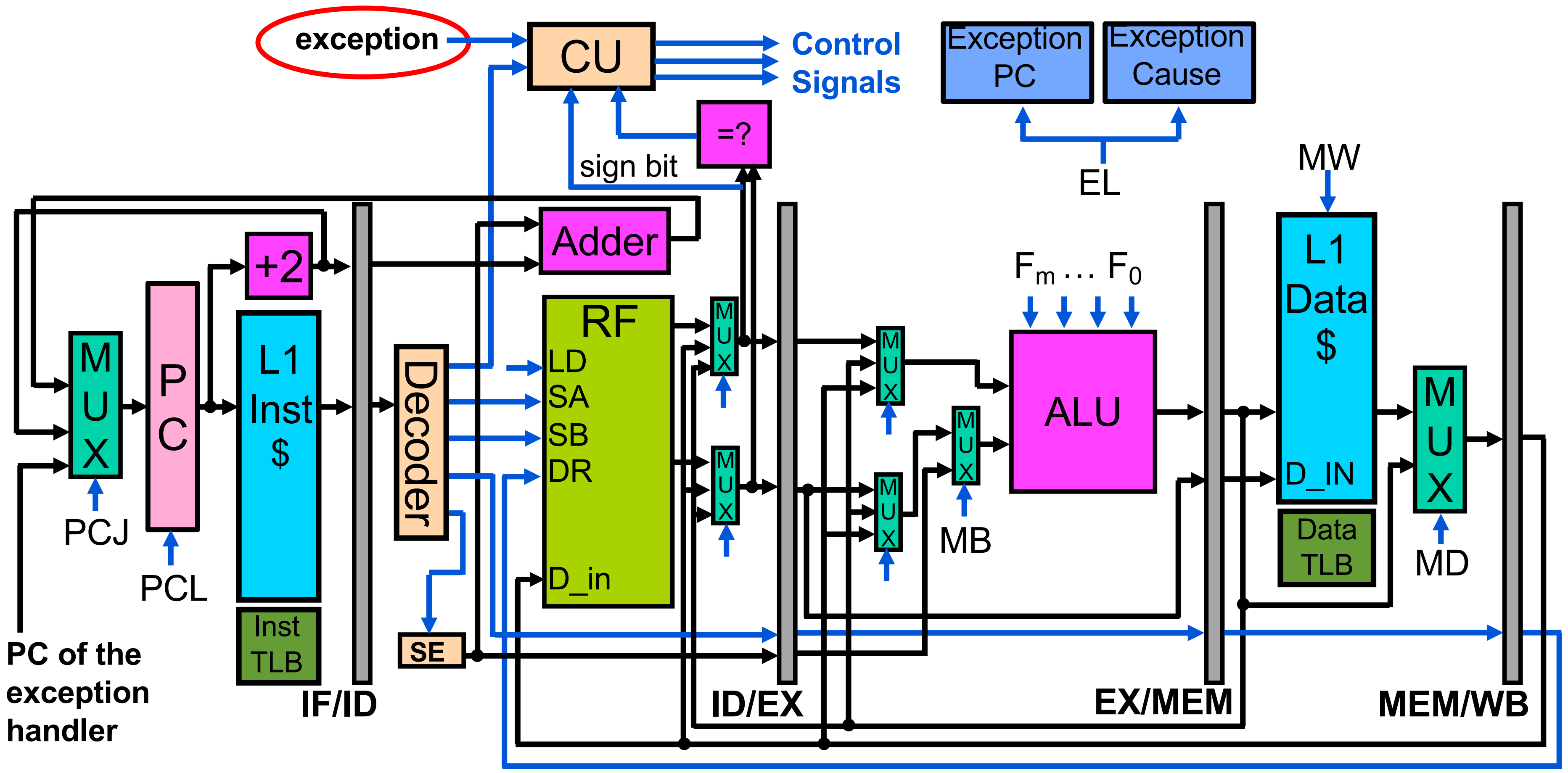
Instruction Page Fault



[SUB R5,R5,R7]

<page fault>

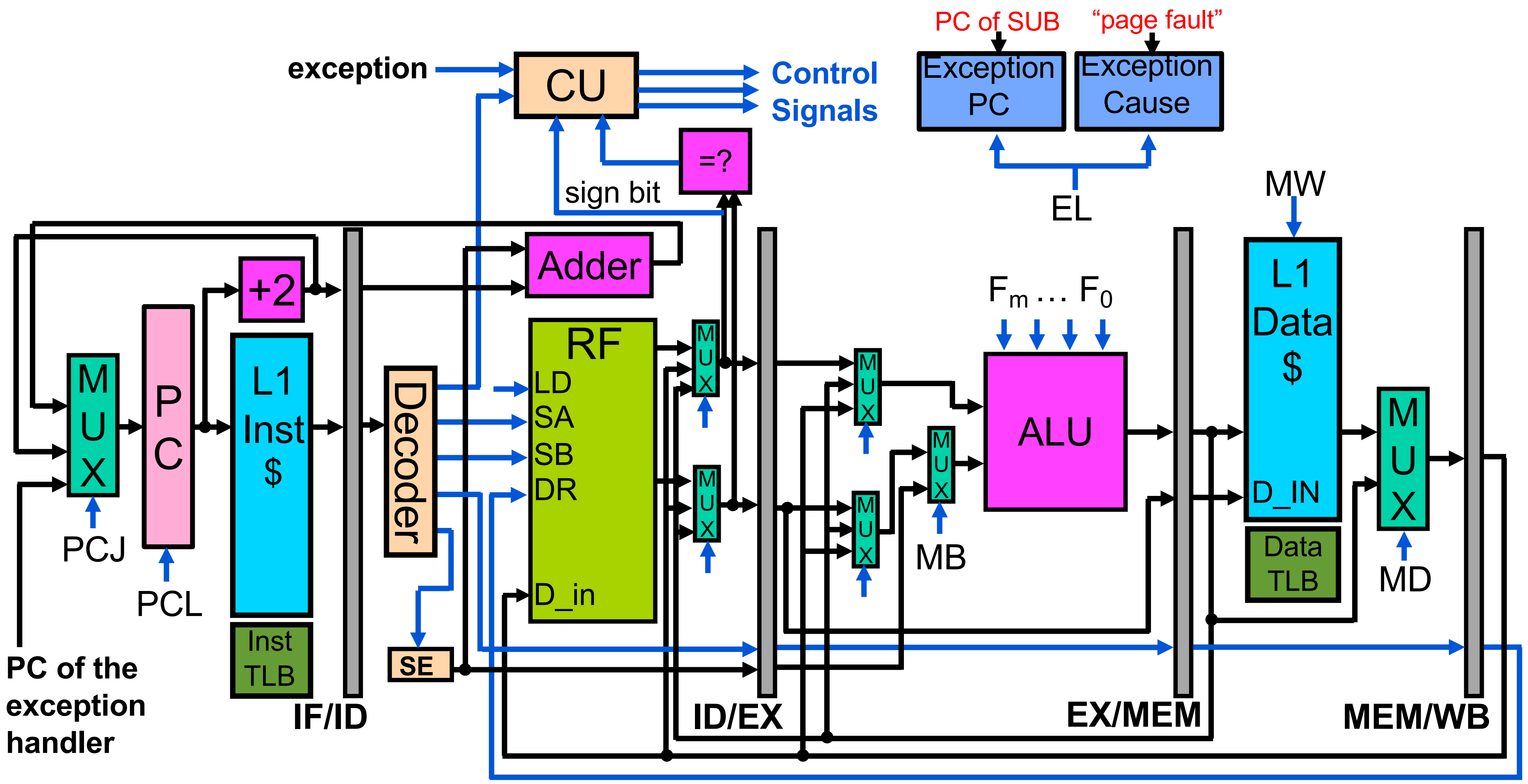
Instruction Page Fault



[SUB R5,R5,R7]

<page fault>

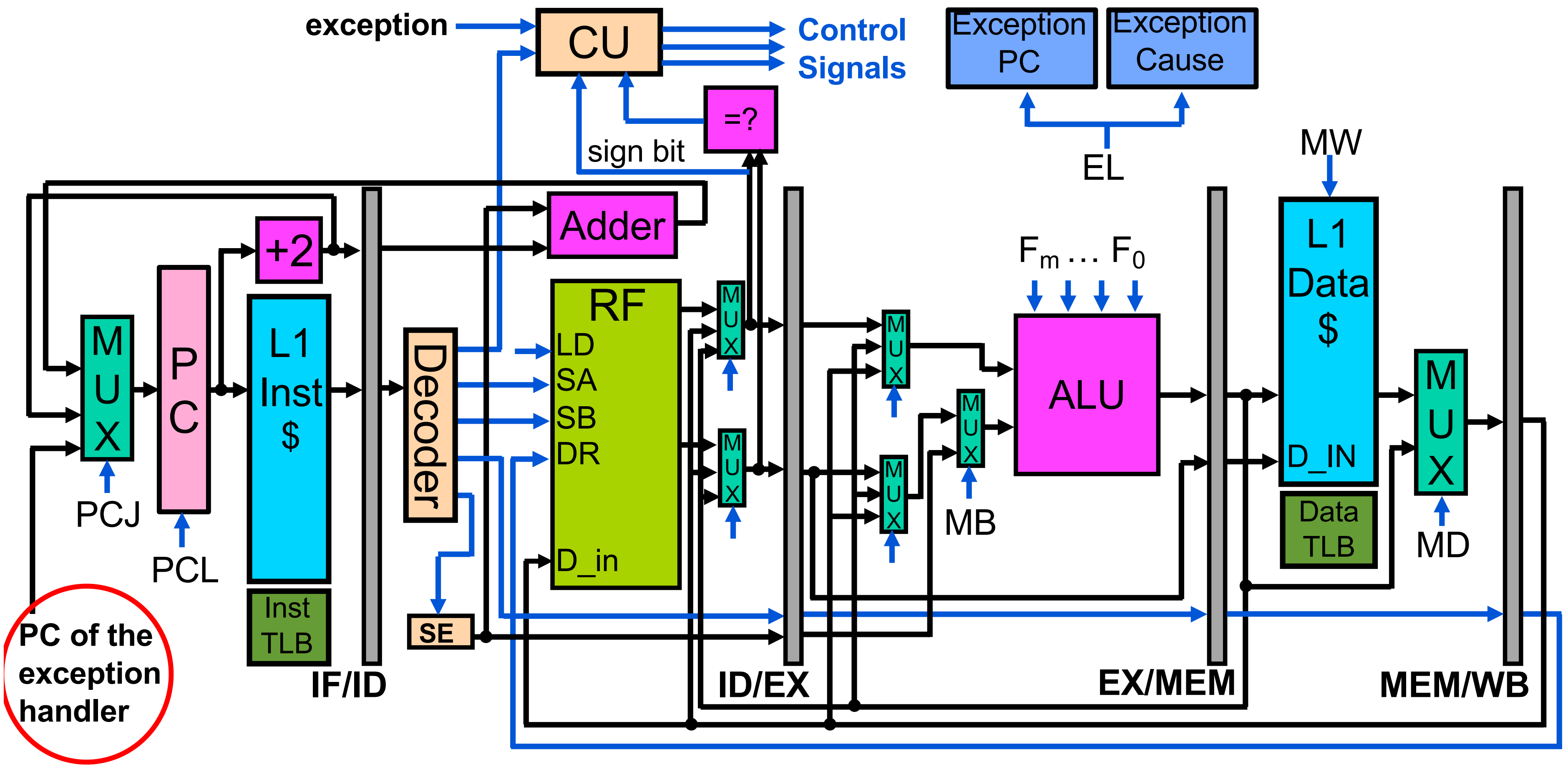
Instruction Page Fault



[SUB R5,R5,R7]

<page fault>

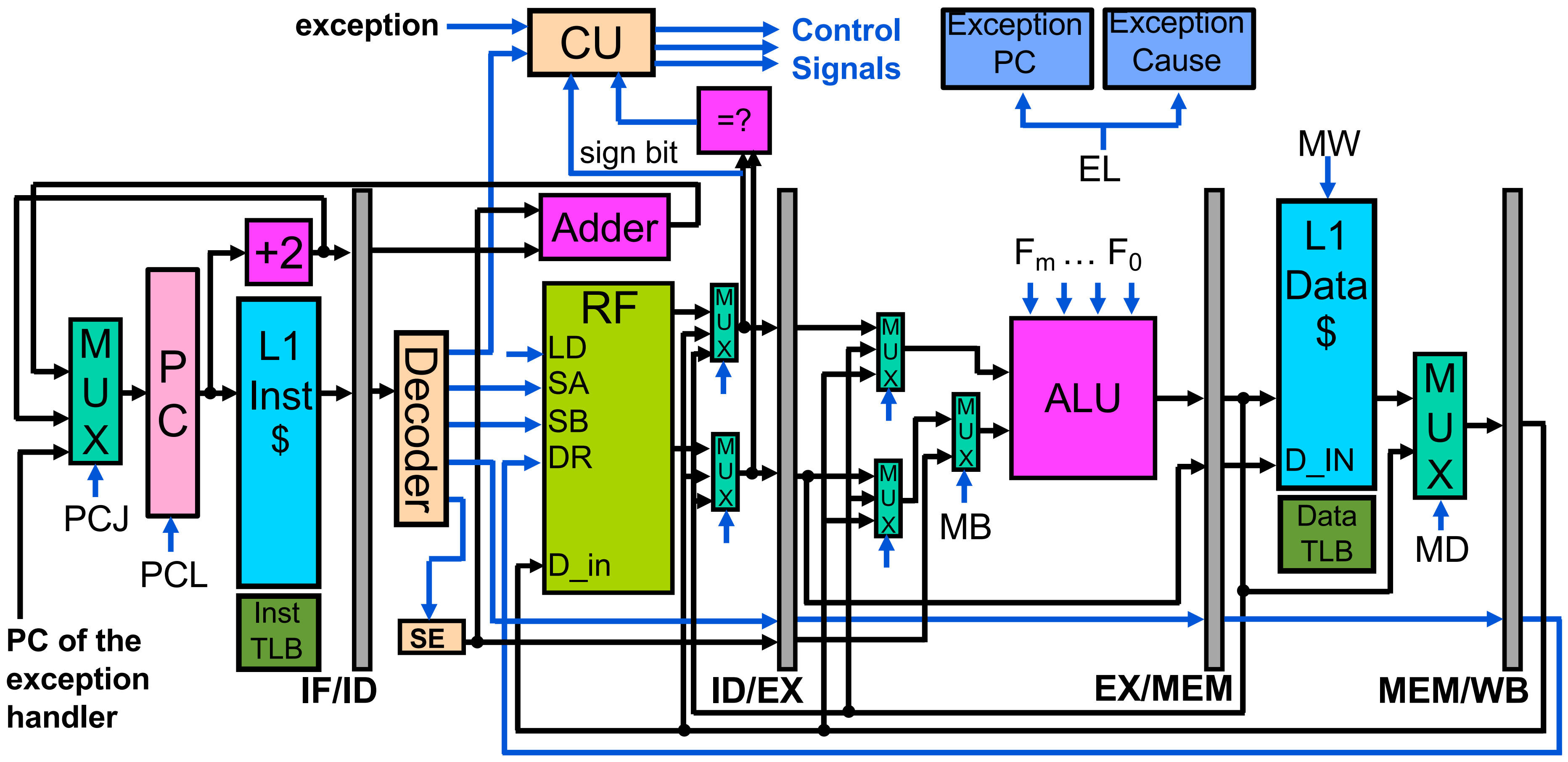
Instruction Page Fault



[SUB R5,R5,R7]

<page fault>

Instruction Page Fault



1st instruction in exception handler

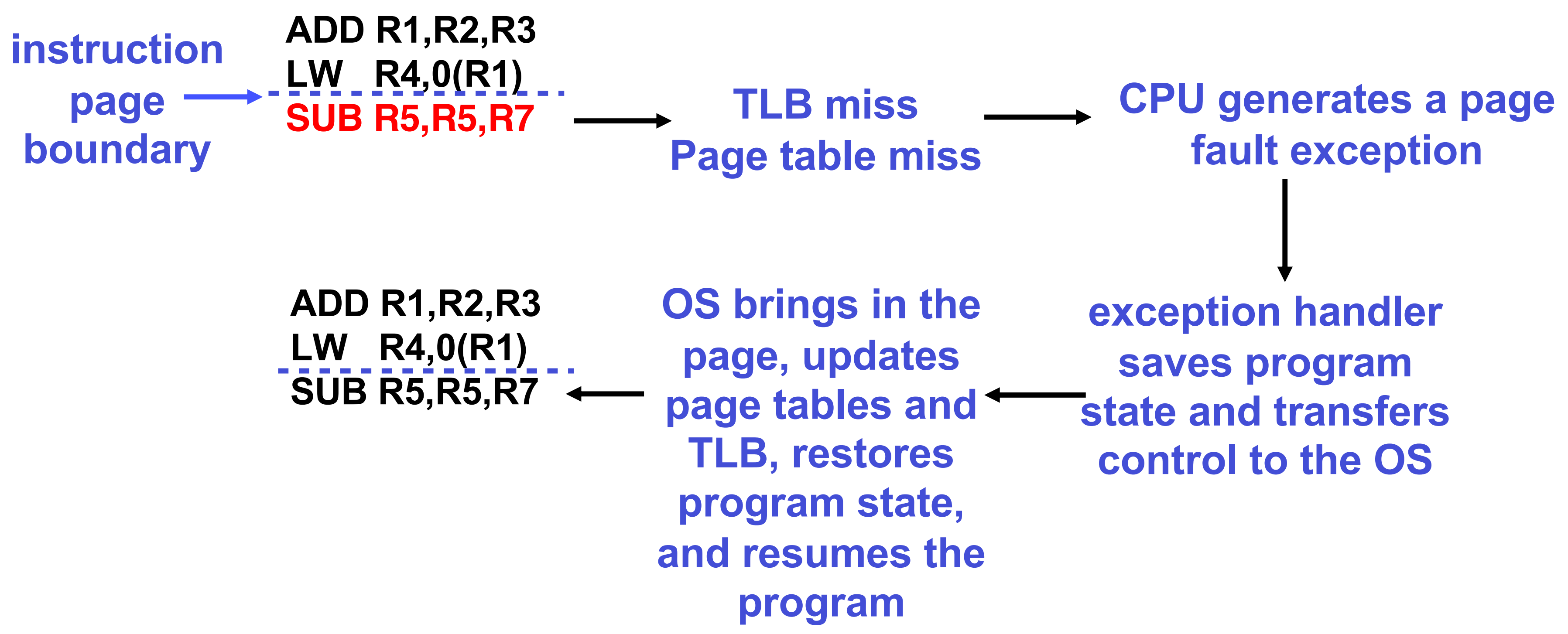
Enabling Program Restart and Calling OS

The exception handler then takes the following actions

- Saves the program state into memory so this program can later be restored when the exception has been handled

- Reads the *Cause register* and determines the appropriate service of the OS to invoke

A Recap: Handling a Page Fault



instruction page boundary

```
ADD R1,R2,R3
LW   R4,0(R1)
SUB R5,R5,R7
```

TLB miss
Page table miss

CPU generates a page fault exception

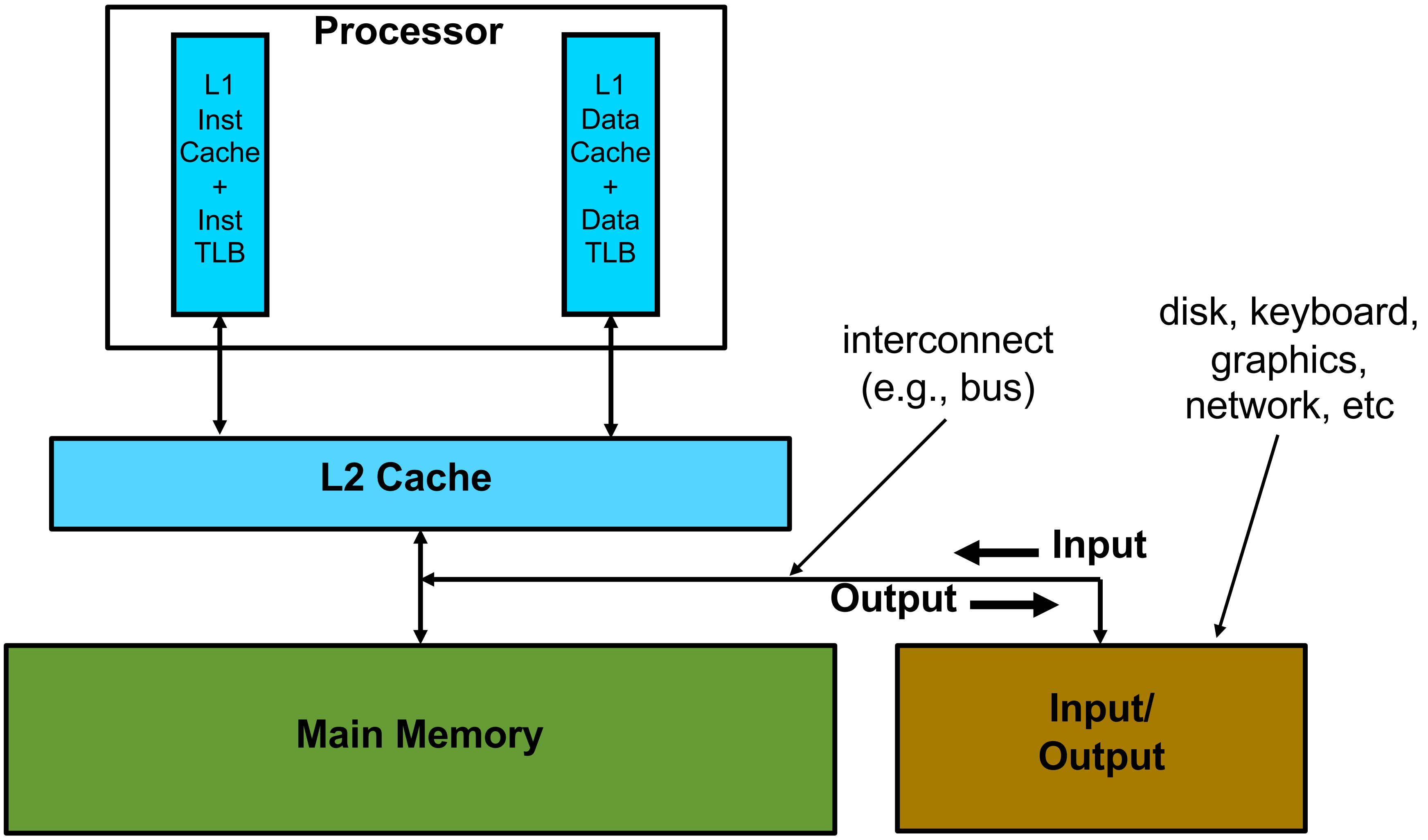
exception handler saves program state and transfers control to the OS

OS brings in the page, updates page tables and TLB, restores program state, and resumes the program

```
ADD R1,R2,R3
LW   R4,0(R1)
SUB R5,R5,R7
```

Analogy for Exception Handler

Computer with Input/Output



Input/Output Devices

- I/O devices are the media to allow computer systems to interact with the outside world

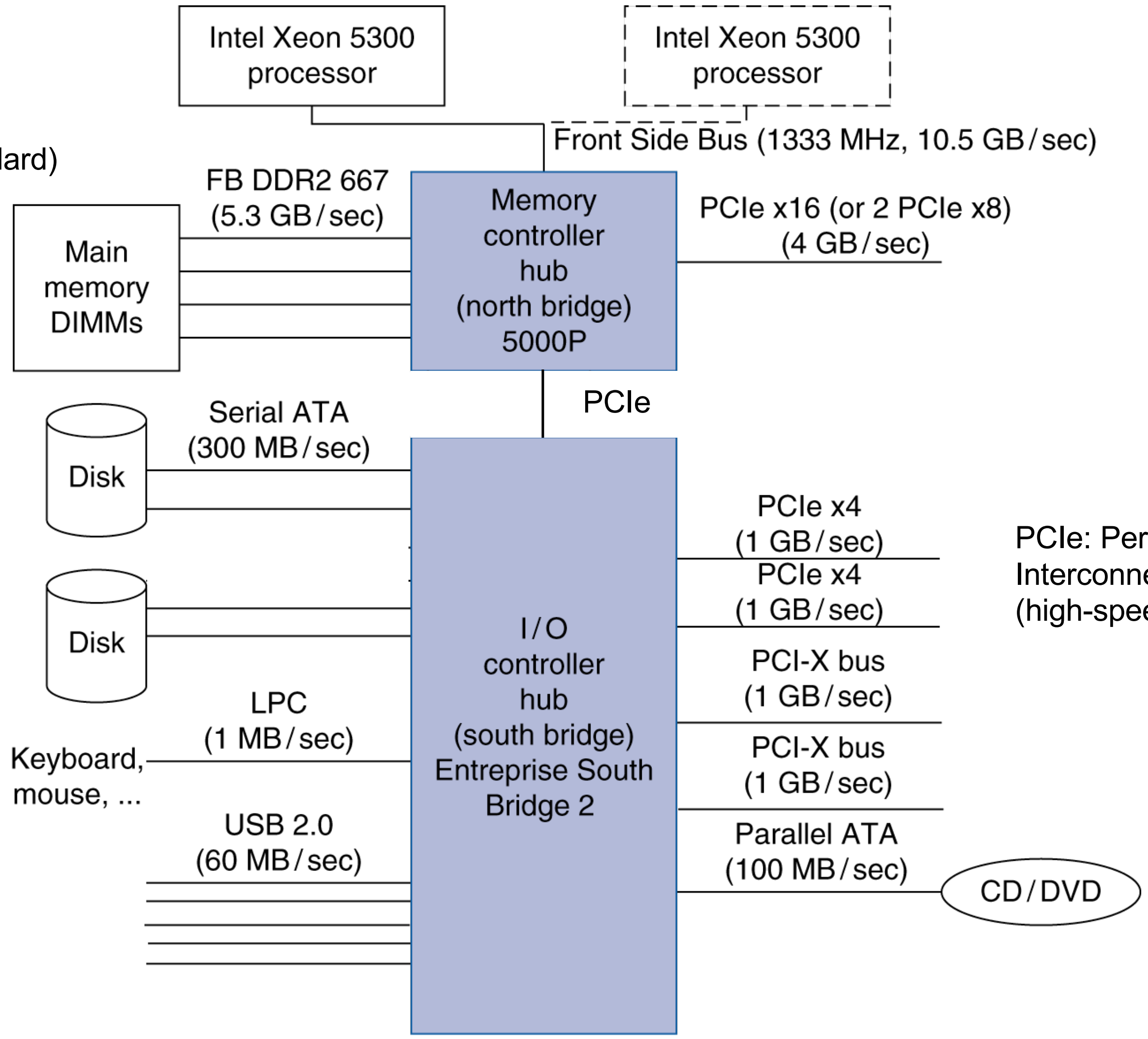
Example Server System with I/O

DDR: Double Data Rate (memory interface standard)

DIMM: Dual Inline Memory Module (circuit board with DRAMs on both sides)

Serial ATA = Serial Advanced Technology Attachment (interface standard for storage device)

PCIe: Peripheral Component Interconnect Express (high-speed bus standard)


Intel Xeon 5300 processor
Intel Xeon 5300 processor
Front Side Bus (1333 MHz, 10.5 GB/sec)
FB DDR2 667 (5.3 GB/sec)
Main memory DIMMs
Memory controller hub (north bridge) 5000P
PCIe x16 (or 2 PCIe x8) (4 GB/sec)
PCIe
Serial ATA (300 MB/sec)
Disk
Disk
PCIe x4 (1 GB/sec)
PCIe x4 (1 GB/sec)
PCI-X bus (1 GB/sec)
PCI-X bus (1 GB/sec)
I/O controller hub (south bridge) Entreprise South Bridge 2
LPC (1 MB/sec)
Keyboard, mouse, ...
USB 2.0 (60 MB/sec)
Parallel ATA (100 MB/sec)
CD/DVD

I/O Controller

- **An I/O controller manages one or more peripheral devices**
  - **Function: coordinates data transfers between the device(s) and the rest of computer system**
  - **Interface: contains a set of special registers for communication with the processor**
    - **Command registers**
      - Tells the device to do something
      - Written by CPU/OS
    - **Status registers (read by processor/OS)**
      - Indicates the status of the device (ready, busy, error)
      - Read by CPU/OS
    - **Data input/output registers**

Accessing I/O Devices

- How do we get a command/data to the right device?

- Dedicated I/O instructions
  - Separate Load/Store instructions to access I/O registers
  - Only the OS can use these instructions

- Memory-mapped I/O
  - Specific portions of the physical address space are assigned to I/O devices
  - Only the OS can access these addresses
  - Each I/O device register has a unique memory address

Data Transfer Between I/O and Memory

- Programmed I/O (PIO)
  - Processor completely arbitrates transfer of data from device to memory
    - Typically much less efficient than DMA

- Direct Memory Access (DMA)
  - I/O device transfers data directly to main memory
  - Processor/OS sets up the transfer through I/O commands
  - And then can do something else, like running another program

Informing the Processor

- A device needs to inform the processor when an I/O operation is completed

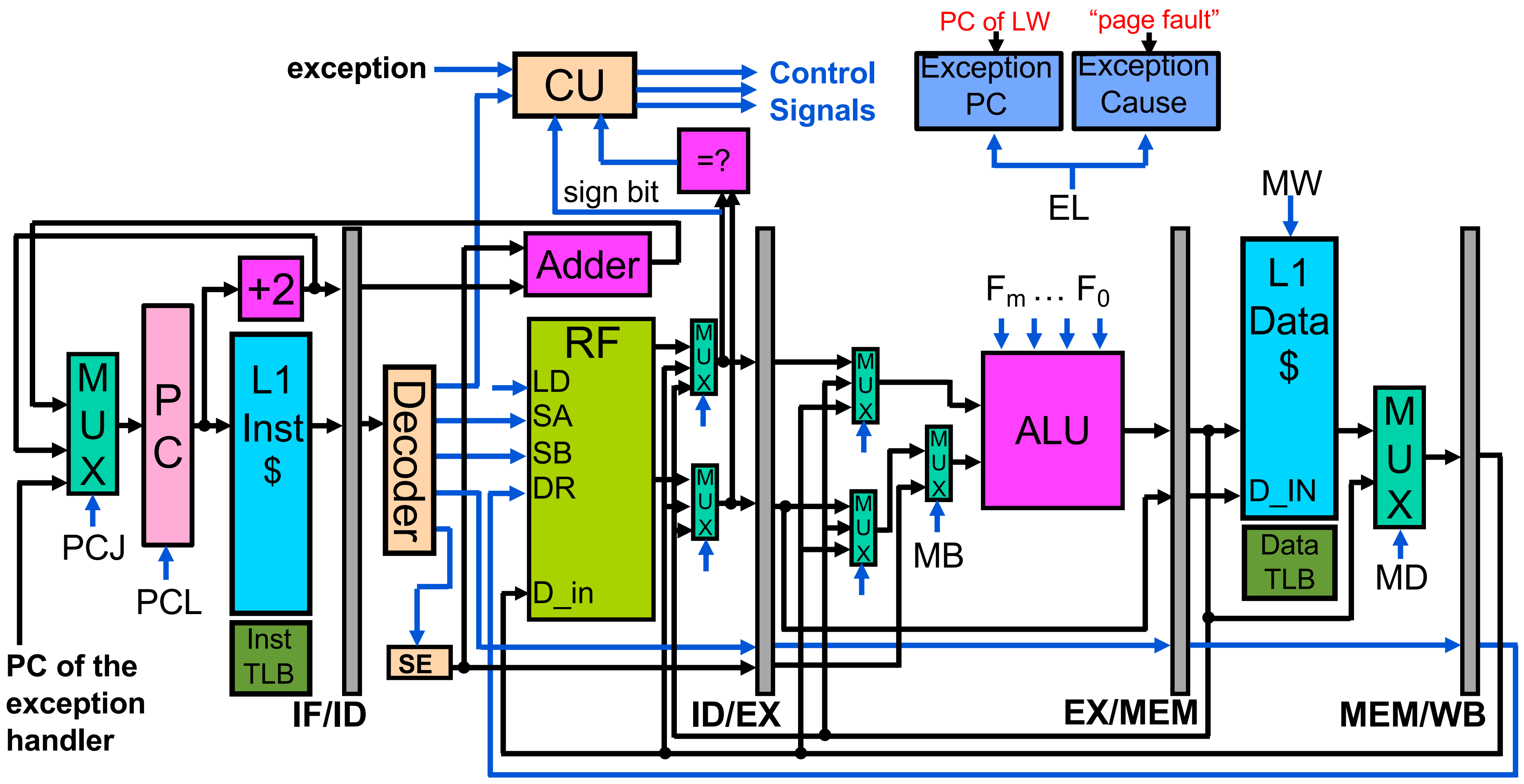
- Polling
  - Processor periodically reads the Status Register, which indicates when an operation is done

- Interrupt-driven I/O
  - I/O device signals the processor via an interrupt when the operation is done
  - Typically more efficient than polling

Let’s Pull Some Pieces Together

- Page fault occurs
- Exception handler gets loaded
- Exception handler takes action
- OS sets up disk transfer
- OS schedules another program
- Data is read from disk and transferred to main memory
- Disk controller interrupts processor
- Second program is interrupted
- First program can run again

Data Page Fault Occurs in Program A

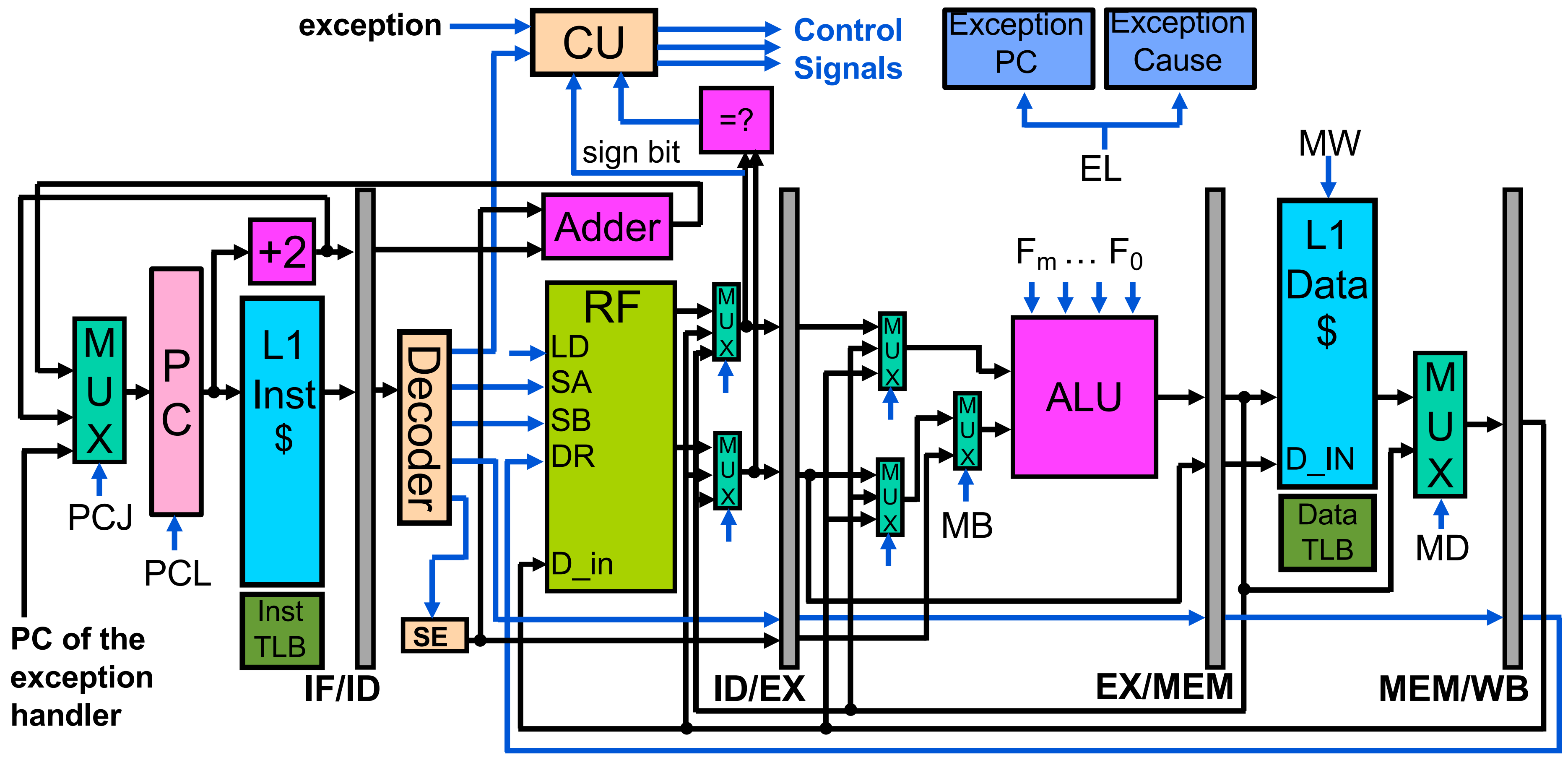


INSTR INSTR INSTR LW R1,0(R1)

<TLB miss>

<page fault>

Exception Handler Gets Loaded



1st instruction in exception handler

NOP

NOP

NOP

NOP

Exception Handler Takes Action

- Saves program A state
- Reads the Cause register and determines that a page fault occurred
- Calls the appropriate part of the OS

OS Switches from Program A to B

What states do we save into the main memory when program A is suspended?

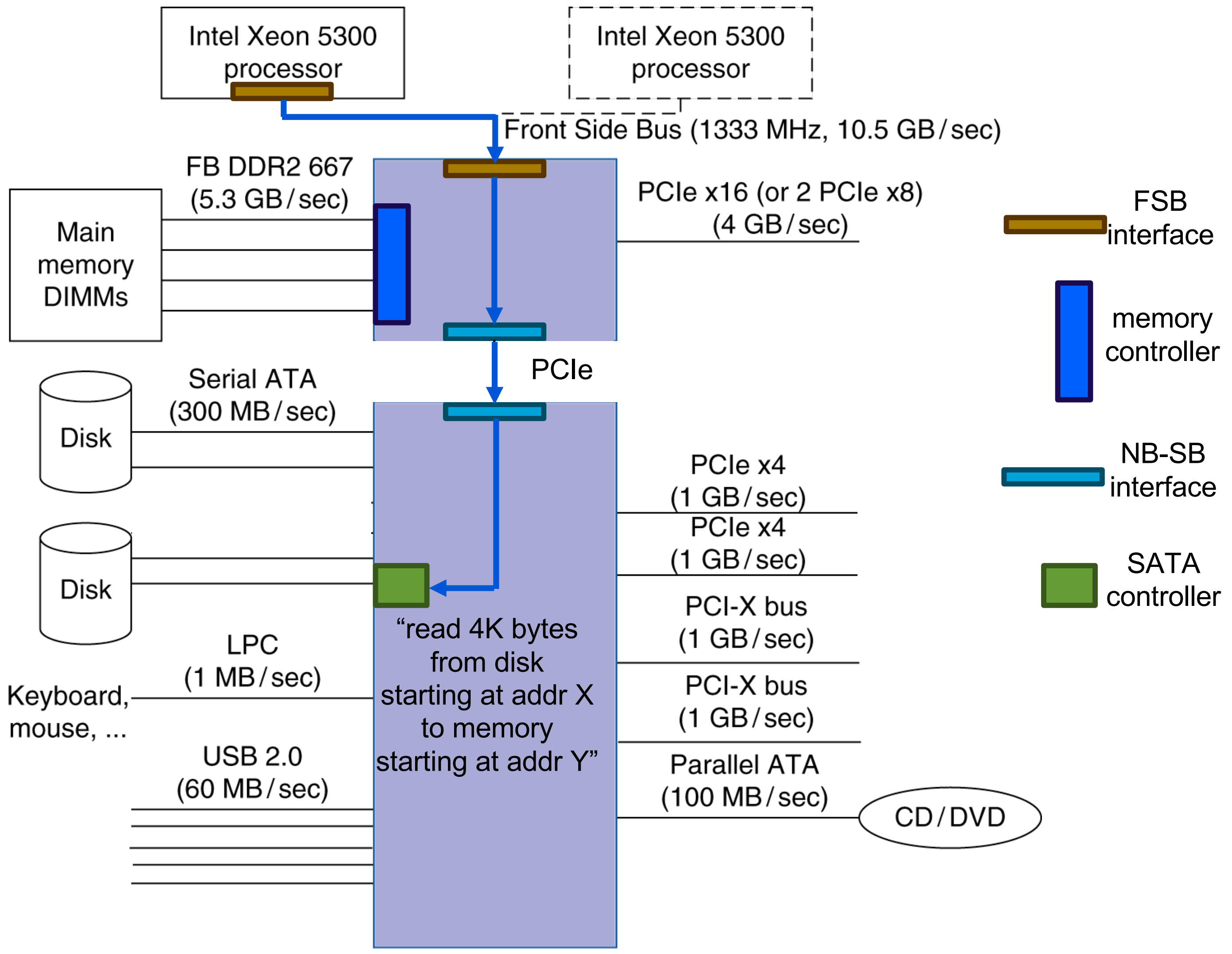
- PC ?
- Registers in RF ?
- Page table register (PTR) ?
- TLB ?

What about caches ?

OS Switches from Program A to B

- Program counter (PC): Save
- Registers in RF: Save
- Page table register (PTR): Save
- TLB: Invalidate all entries
- Caches: Typically retained; not flushed during context switch as they hold physical addresses

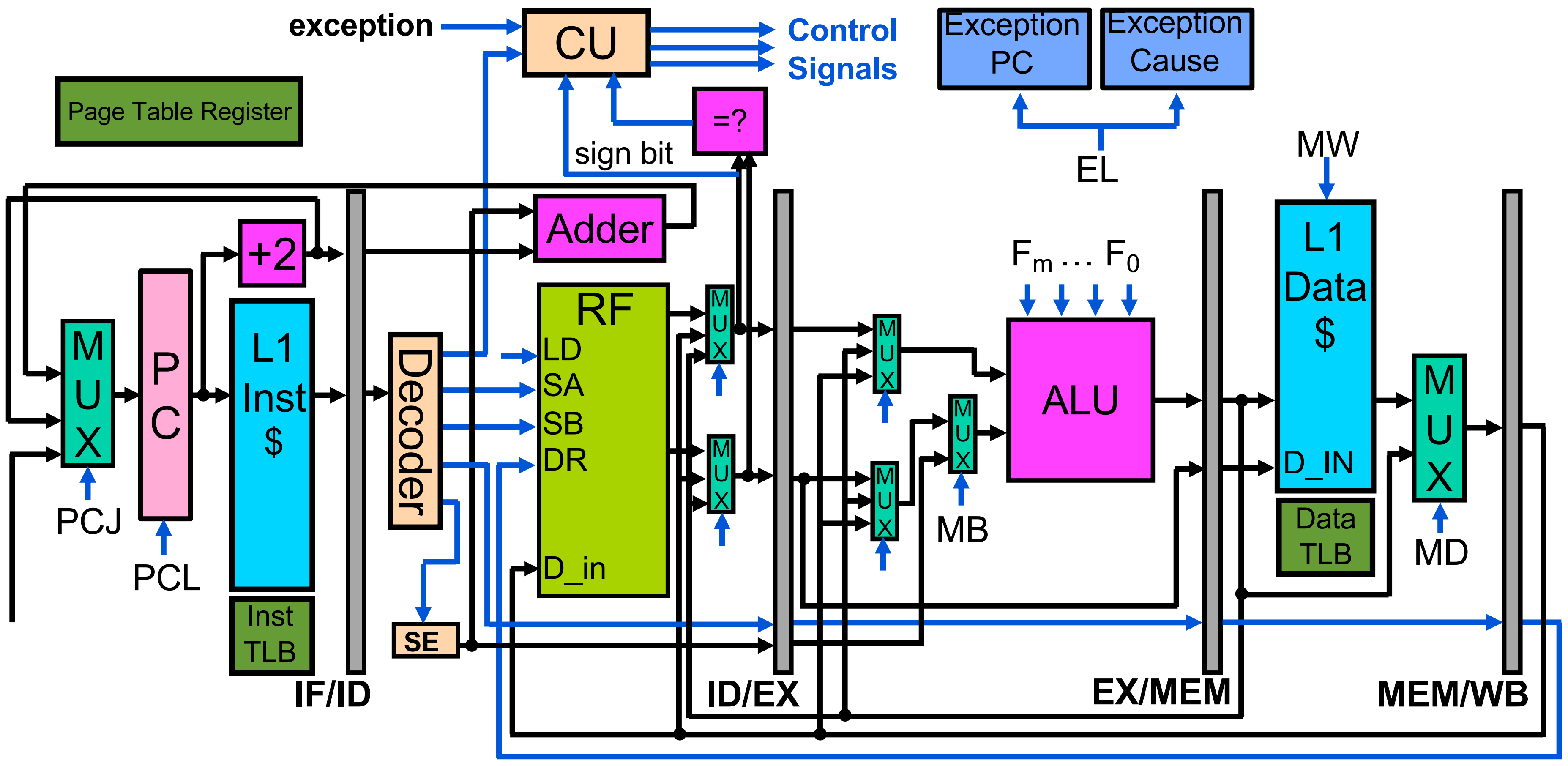
OS Sets Up Disk Transfer using DMA



OS Schedules Another Program

- While waiting for the disk read to complete for program A, the OS scheduler may run a different process (program B)
- It loads the processor with the state (PC, PTR, and RF) of program B

OS Switches from Program A to B

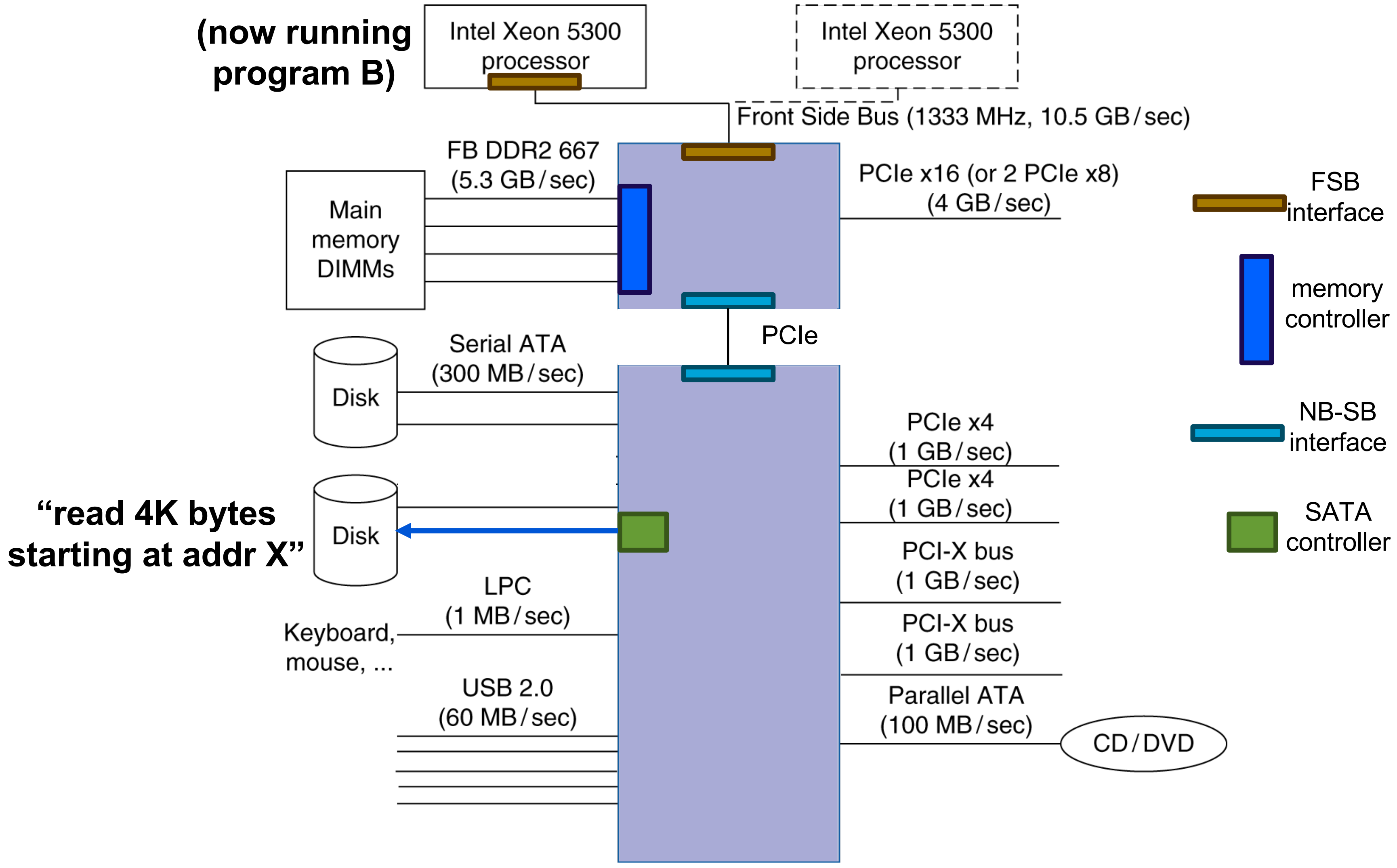


instruction of Program B

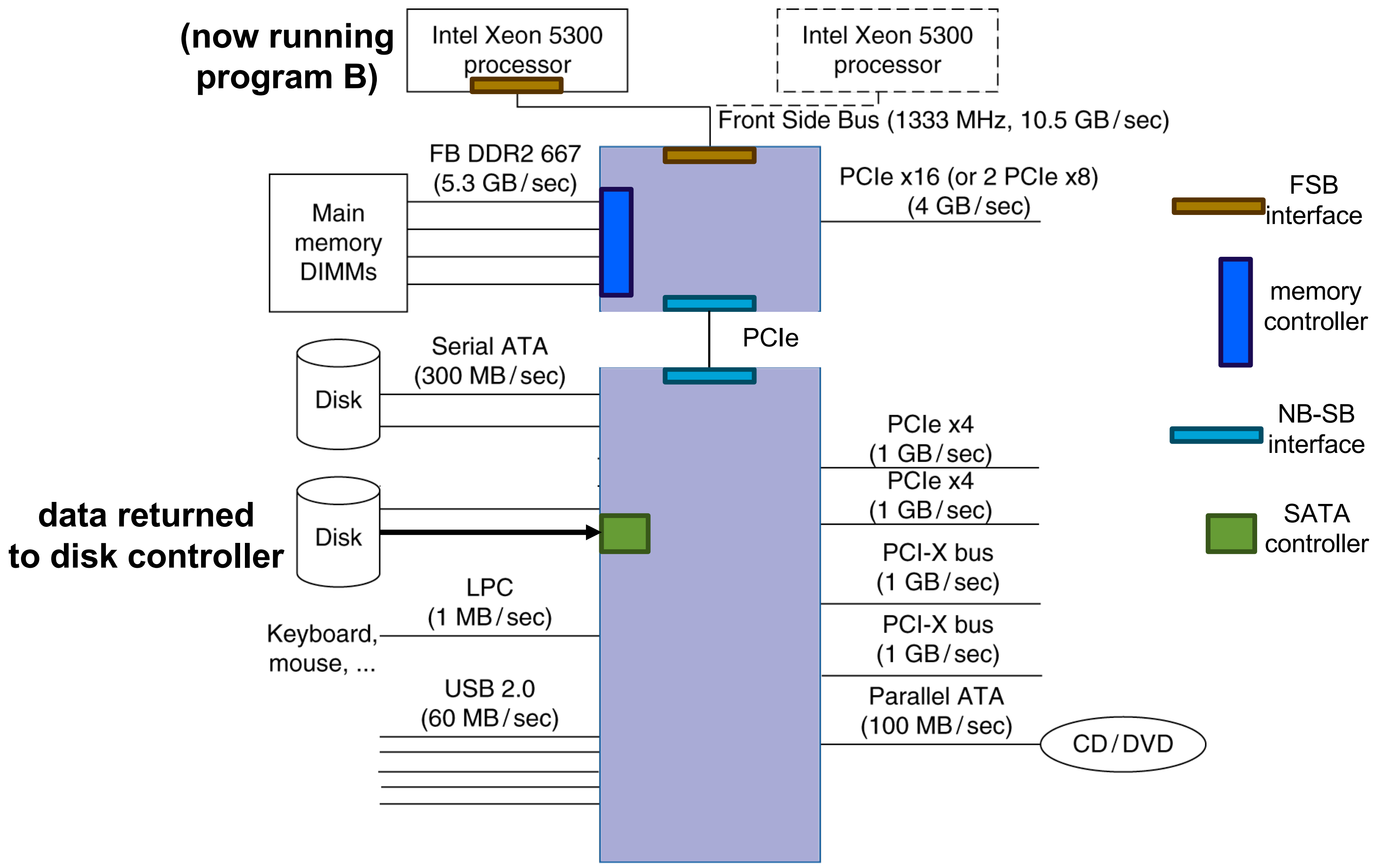
Page is Read from Disk

(now running program B)

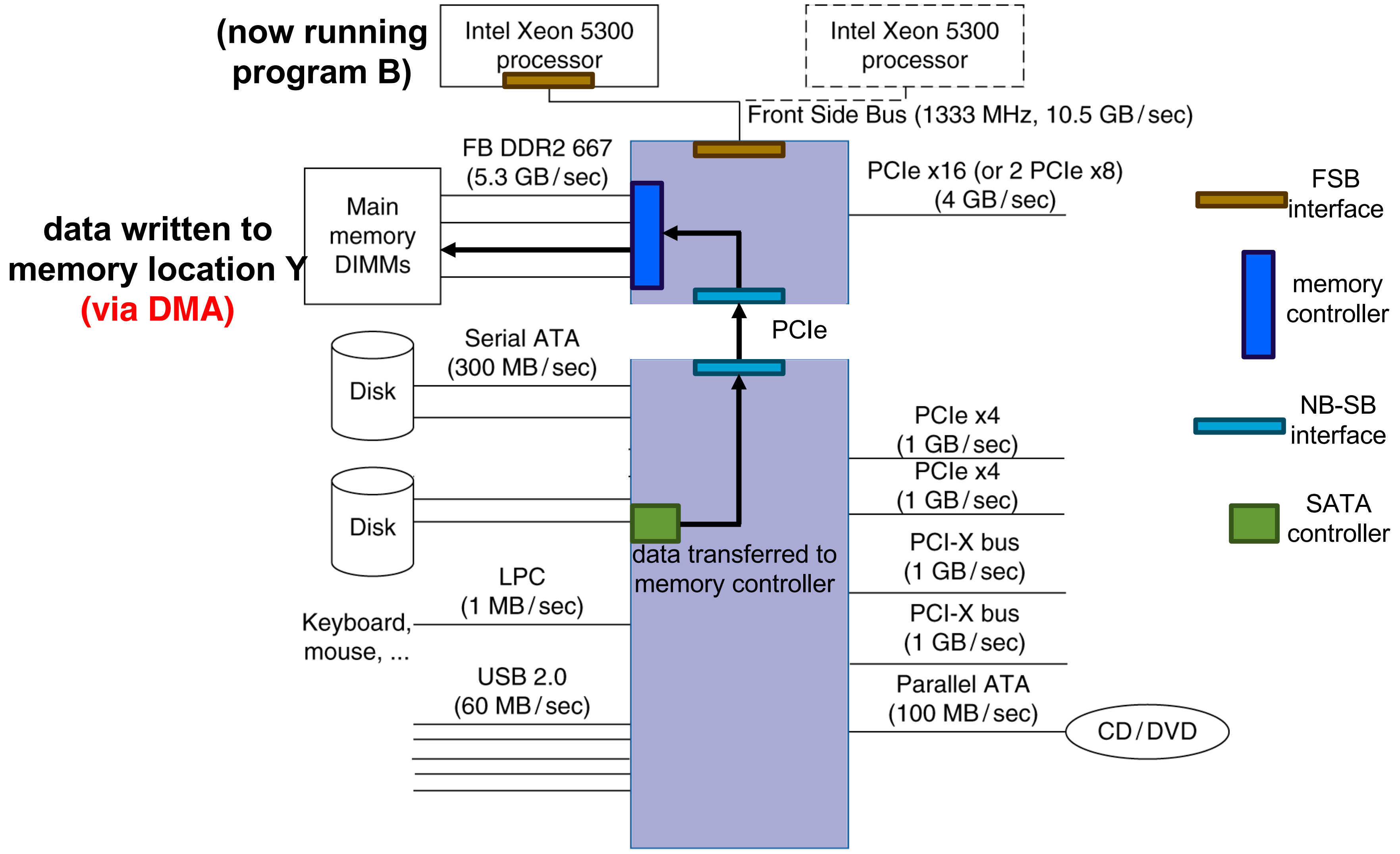
“read 4K bytes starting at addr X”



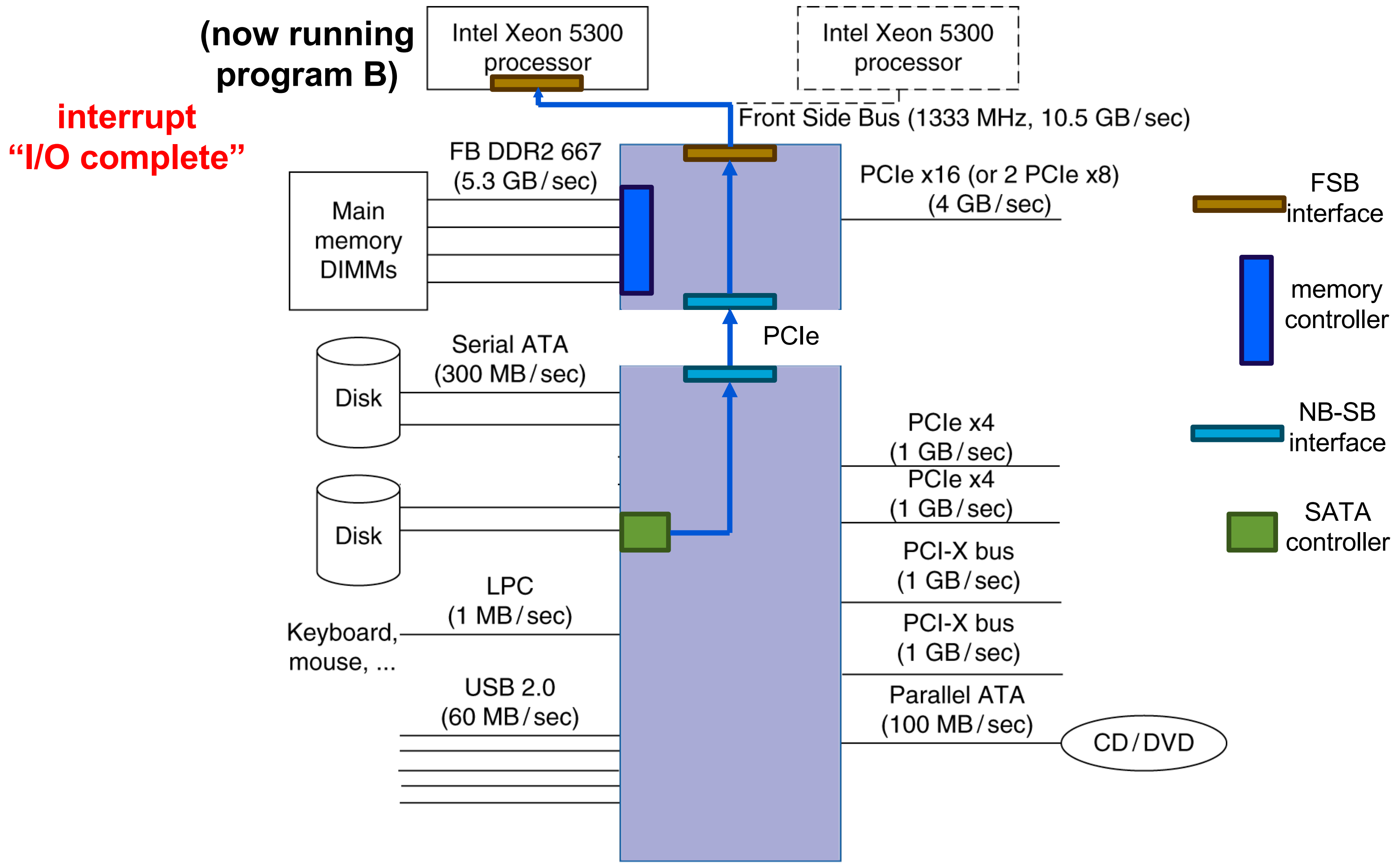
Page is Read from Disk



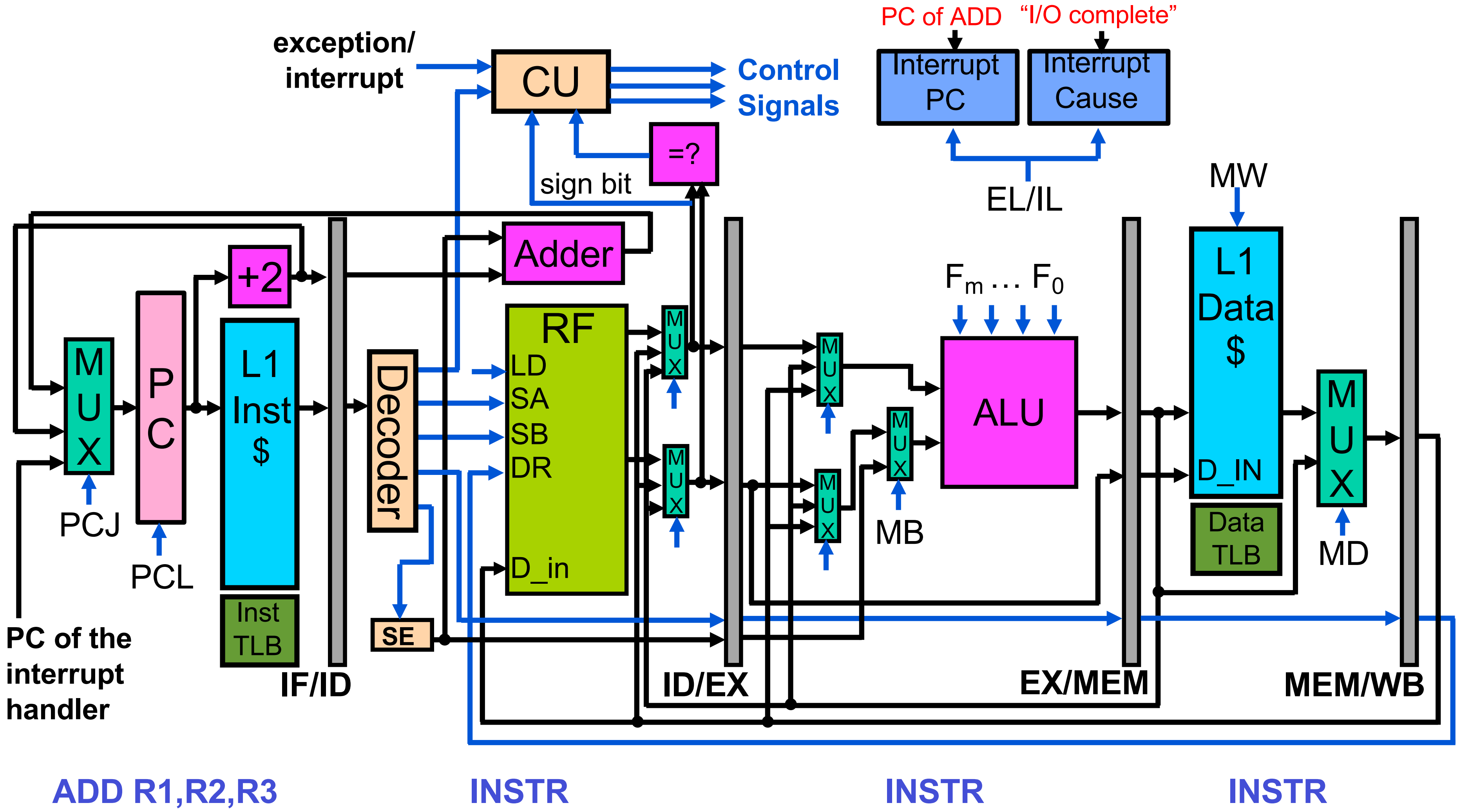
DMA Transfer of Page to Memory



I/O Controller Interrupts Processor



Program B is Interrupted

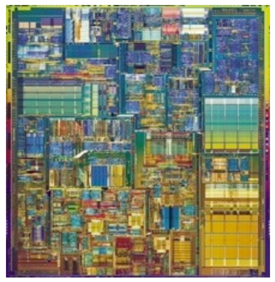


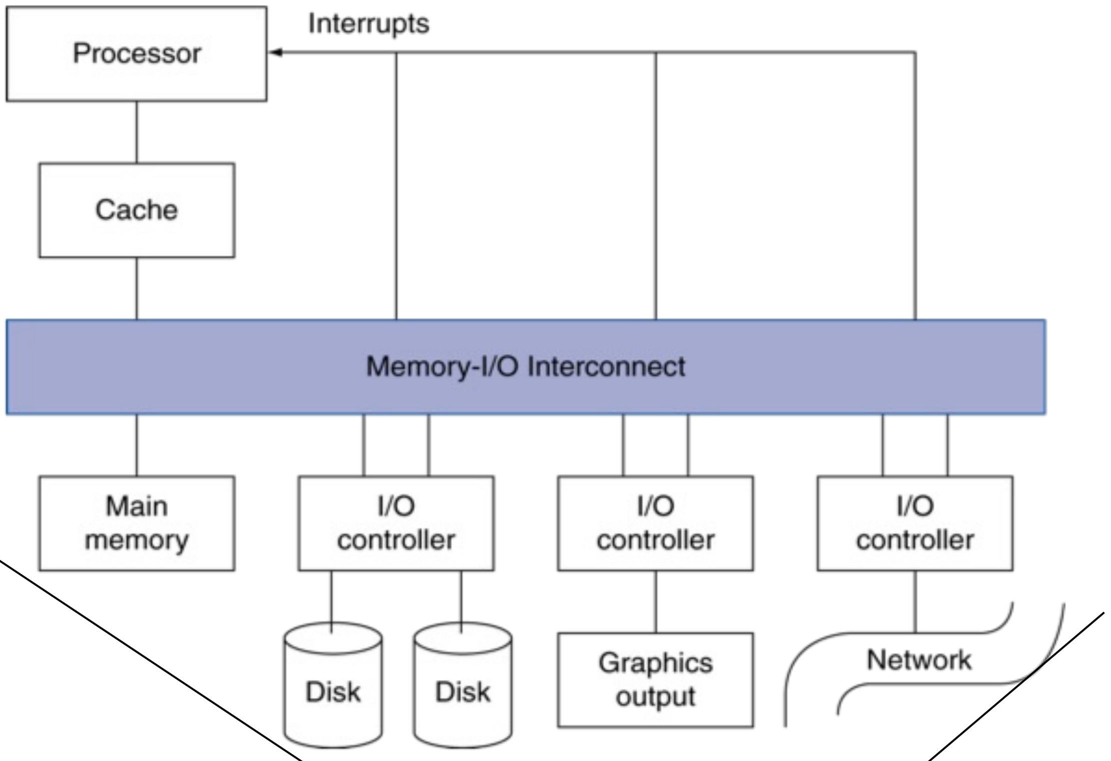
The state of program B is saved by the interrupt handler

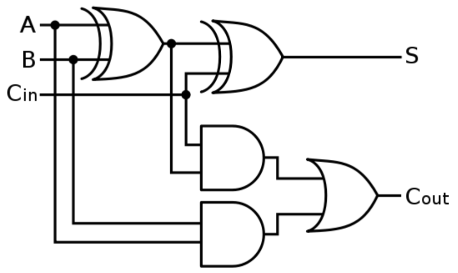
Program A Can Now Run Again

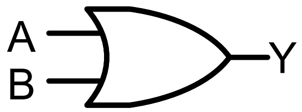
- Page fault was handled, so OS marks program A as runnable
- If OS scheduler chooses to run program A, it loads its state (PC of the LW, PTR, and registers)
- Key point: Processor was free to do other work during the long I/O transfer time
  - With DMA, processor did not have to directly handle data transfers from device to memory
  - With interrupt-driven I/O, processor did not have to poll the device to see when the I/O operation completed

Building a Complete Computer









Next Class

More on Final Exam

Advanced Topics