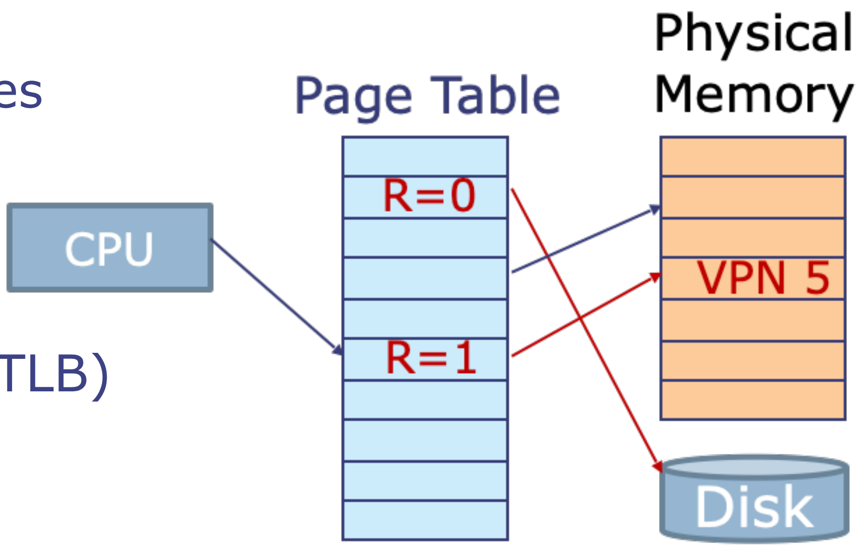
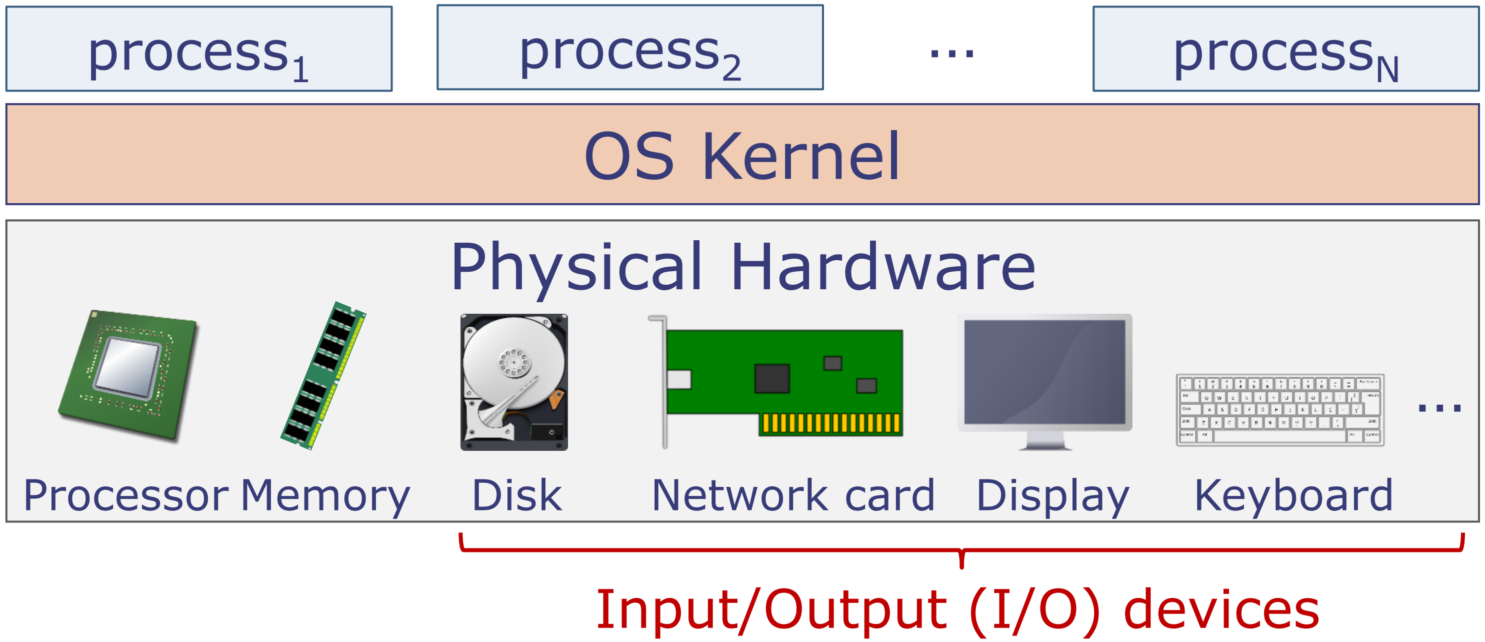
Lecture 19
I/O and Exceptions

Reminder: Virtual Memory

- Goal of virtual memory
  - Abstraction of the storage resources of the machine
  - Protection and privacy: Processes cannot access each other's data
- What we learned
  - Translation Lookaside Buffer (TLB) for address translation
  - Caches with virtual memory
  - Hierarchical page table
  - Page replacement algorithm
  - Page sharing and memory mapping
  - Copy-on-Write



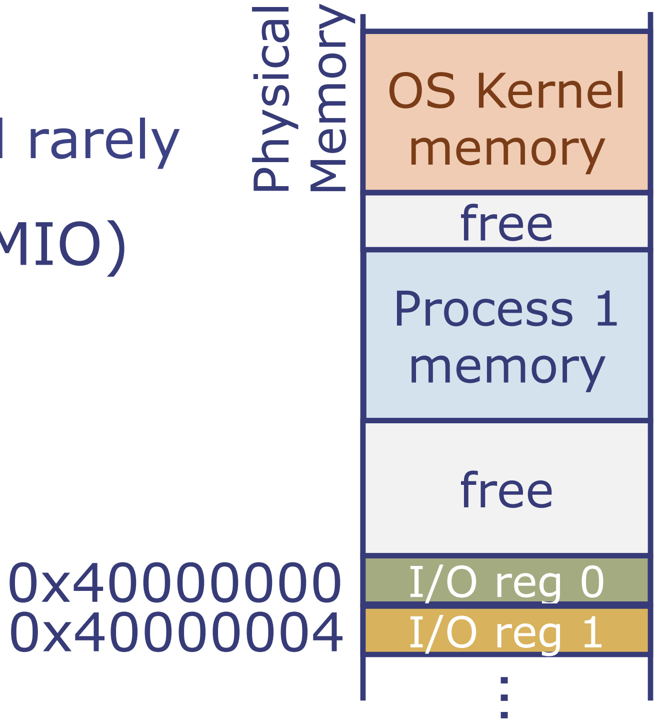
Communicating with I/O devices

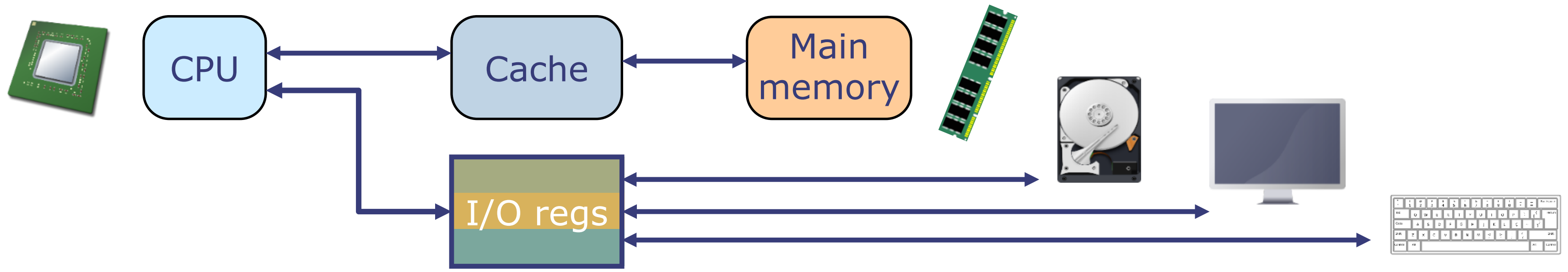


- The system has shared I/O registers that both the processor and the I/O devices can read and write
- Key questions:
  - How to access I/O registers?
  - How do processor and device coordinate on each transfer?

Accessing I/O registers

- Option 1: Use special instructions
  - e.g., `in` and `out` instructions in x86
  - Inflexible, adds instructions → used rarely
- Option 2: Memory-mapped I/O (MMIO)
  - I/O registers are mapped to physical memory locations
  - Processor accesses them with loads and stores
  - These loads and stores should not be cached!





Coordinating I/O Transfers

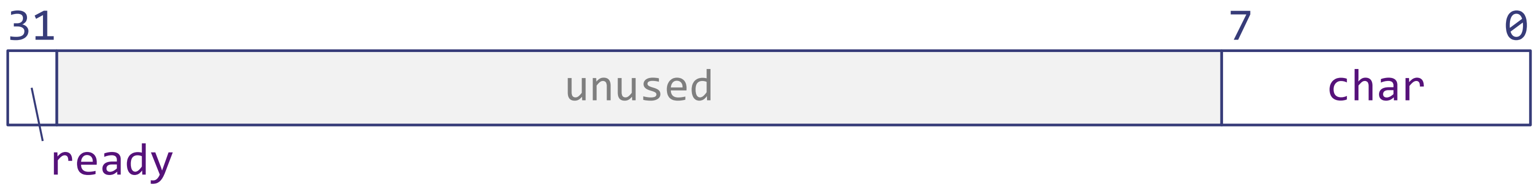
- Option 1: Polling (synchronous)
  - Processor periodically reads the register associated with a specific I/O device
- Option 2: Interrupts (asynchronous)
  - Processor initiates a request, then moves to other work
  - When the request is serviced, the I/O device interrupts the processor
- *Pros/Cons of each approach?*

Polling is simple but worse latency

Interrupts let the processor do useful computation while request is serviced + better latency but less resource efficient

Example 1: Polling-based I/O

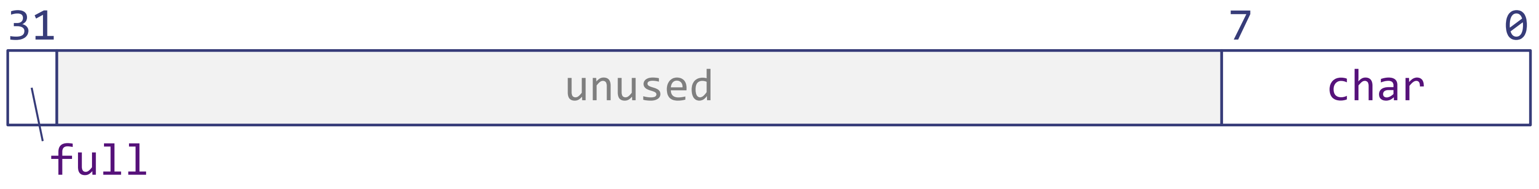
- Consider a simple character-based display
- Uses one I/O register with the following format:



  - The ready bit (bit 31) is set to 1 only when the display is ready to print a character
  - When the processor wants to display an 8-bit character, it writes it to char (bits 0-7) and sets the ready bit to 0
  - After the display has processed the character, it sets the ready bit to 1

Example 2: Interrupt-based I/O

- Consider a simple keyboard that uses a single I/O register with a similar format:



  - On a keystroke, the keyboard writes the typed character in char, sets full bit to 1, and raises a keyboard interrupt
  - Interrupt handler reads char and sets full bit to 0, so the keyboard can deliver future keystrokes

  - This works fine because keyboard is very slow compared to CPU. Faster devices use more sophisticated mechanisms (e.g., network cards write packets to main memory and use I/O registers only to indicate status of transfer)

Memory-mapped I/O (MMIO) Addresses

- Inputs
  - 0x 4000 4000 - performing a **lw** from this address will read one signed word from the keyboard.
  - Repeating a **lw** to this address will read the next input word and so on.

- Outputs:
  - 0x 4000 0000 - performing a **sw** to this address prints an ASCII character to the console corresponding to the **ASCII** equivalent of the value stored at this address
  - 0x 4000 0004 - a **sw** to this address prints a **decimal** number
  - 0x 4000 0008 - a **sw** to this address prints a **hexadecimal** number

Pros and Cons of MMIO

- Advantages:
  - ✓ Single Address Space:
    - Unified memory and I/O addressing simplifies programming
  - ✓ Simplified Access:
    - Common memory read/write instructions for both memory and I/O operations, no new instructions needed
  - ✓ Cache Benefits:
    - Potential cache utilization for improved performance
- Disadvantages:
  - X Address Space Contention:
    - I/O devices share address space with memory, reducing available memory addresses
  - X Possible Latency:
    - Cache and memory access optimizations can cause additional latency for time-sensitive I/O operations

Memory Mapped IO Example

Program that reads two inputs from keyboard, adds them and displays result on monitor.

```
// load the read port into t0
li t0, 0x40004000

// read the first input
lw a0, 0(t0)
// read the second input
lw a1, 0(t0)

// add them together
add a0, a0, a1

// load the write port into t0
li t0, 0x40000004
// write the output in decimal
sw a0, 0(t0)
```

MMIO for Performance Measures

- Performance Measures
  - 0x 4000 5000 – **lw** to get **instruction count** from start of program execution
  - 0x 4000 6000 – **lw** get **performance counter** – number of instructions between turning the performance counter on and then off.
  - 0x 4000 6004
    - **sw 0** to turn **performance counting off**
    - **sw 1** to turn it **on**

Memory Mapped IO Example 2

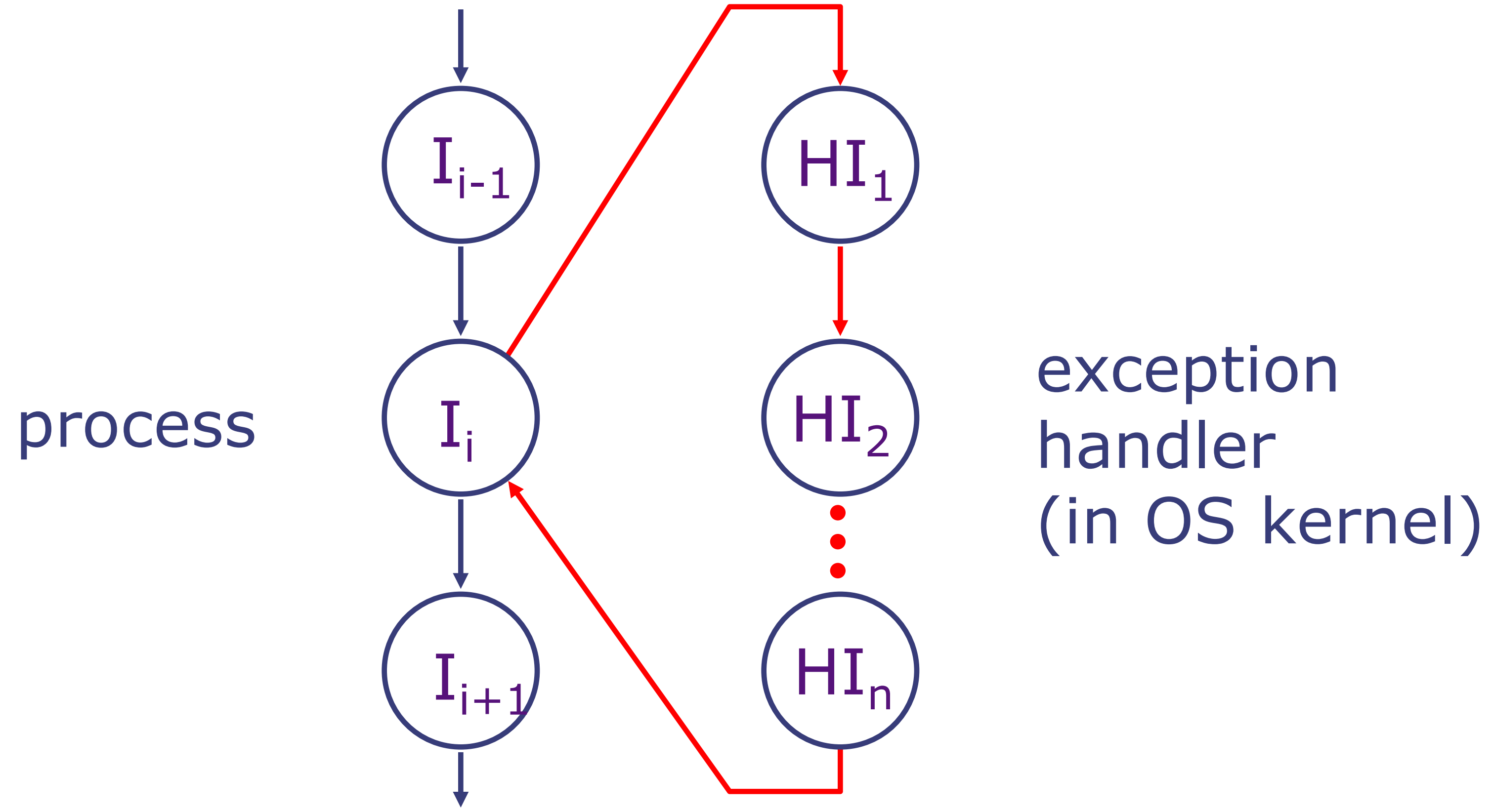
A program similar to Example 1, with an additional performance counter that measures the time taken for a simple addition operation.

```
// prepare to read input from console
li t0, 0x40004000
// get user input
lw a0, 0(t0)
lw a1, 0(t0)
// load the performance counter address into t1
li t1, 0x40006000
li t2, 1
// start the performance counter by storing 1 to the magic address
sw t2, 4(t1)
add a0, a0, a1
// stop the performance counter by storing 0 to the address
sw zero, 4(t1)
// prepare to print decimal to console
li t0, 0x40000004
// first print sum
sw a0, 0(t0)
// get the count from the performance counter
lw t2, 0(t1)
// print the count
sw t2, 0(t0)
```

Exceptions

Review: Exceptions

- Exception: Event that needs to be processed by the OS kernel. The event is usually unexpected or rare.



RISC-V Exception Handling

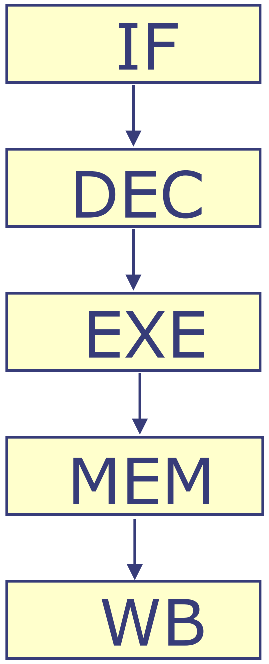
- RISC-V provides several privileged registers, called control and status registers (CSRs), e.g.,
  - mepc: exception PC
  - mcause: cause of the exception (interrupt, illegal instr, etc.)
  - mtvec: address of the exception handler
  - mstatus: status bits (privileged mode, interrupts enabled, etc.)

- RISC-V also provides privileged instructions, e.g.,
  - csrr and csrw to read/write CSRs
  - mret to return from the exception handler to the process
  - Trying to execute these instructions from user mode causes an exception → normal processes cannot take over the machine

Exceptions in the Pipeline

- On an exception, need to:
  - Save current PC in mepc
  - Save the cause of the exception in mcause
  - Set PC with the exception handler
- Exceptions cause control flow hazards!
  - They are implicit branches
- Want precise exceptions:
  - All preceding instructions must have completed
  - Instruction causing exception and future instructions must not have executed (no updates to register or memory)
  - Simple in single-cycle machines, more complex with pipelining

When Can Exceptions Happen?



Memory fault (e.g., illegal memory address)

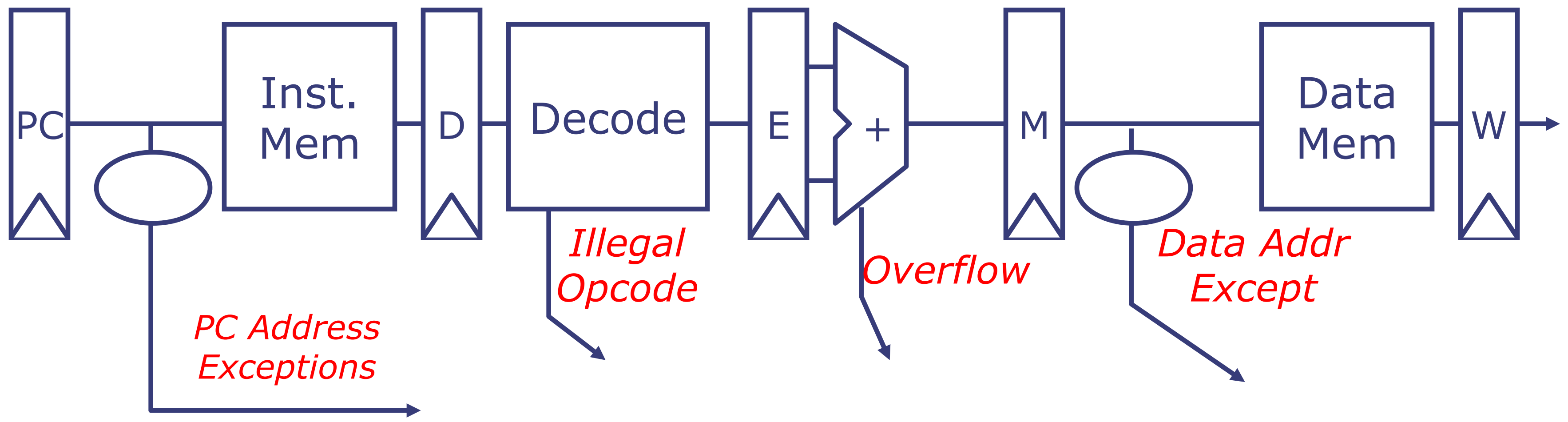
Illegal instruction (e.g., unknown instruction)

Arithmetic exception (e.g., divide by zero)

Memory fault (e.g., illegal memory address)

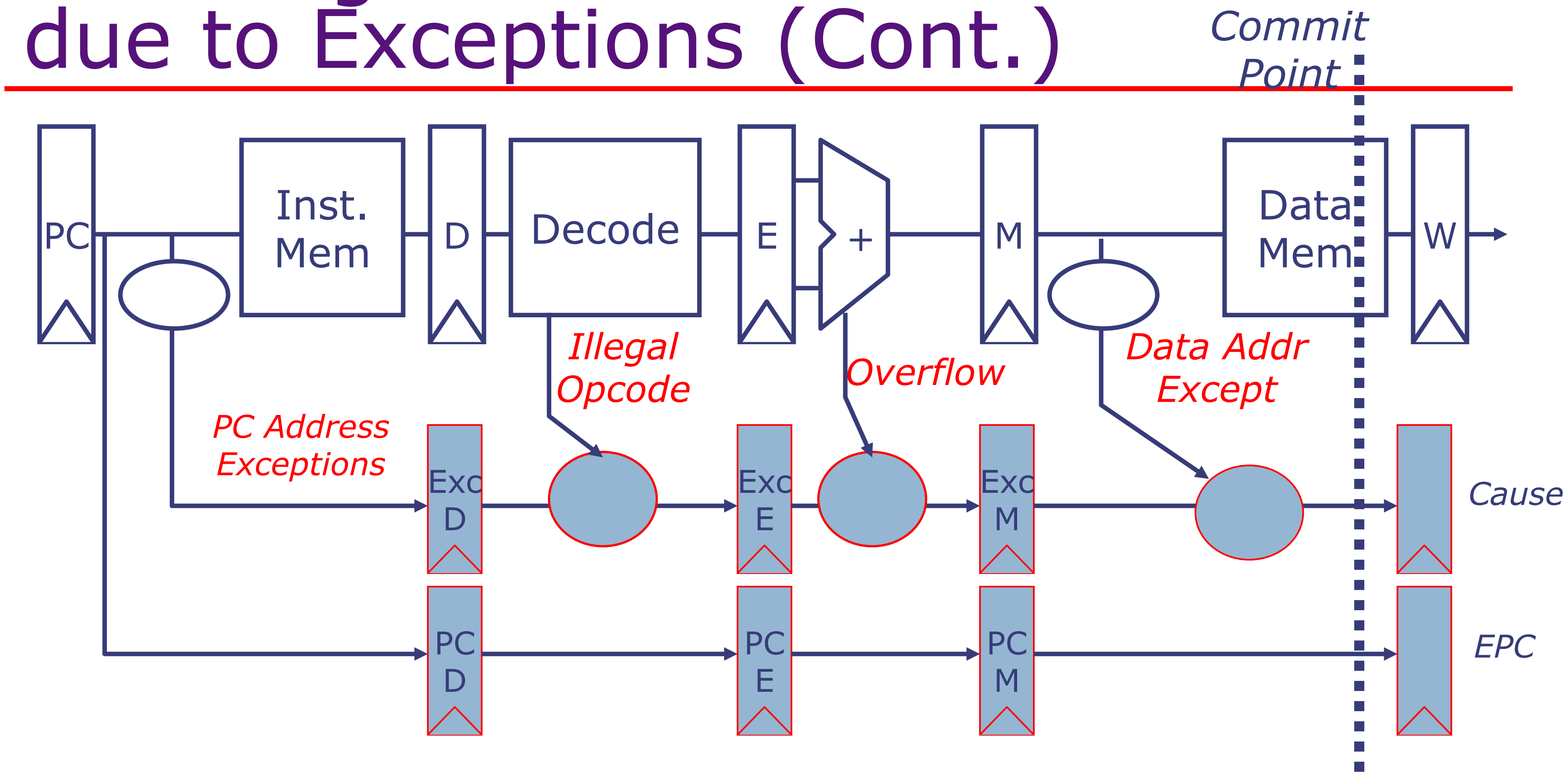
- Instructions following the one that causes the exception may already be in the pipeline…
- … but none has written registers or memory yet ☺

Handling Control Hazards due to Exceptions



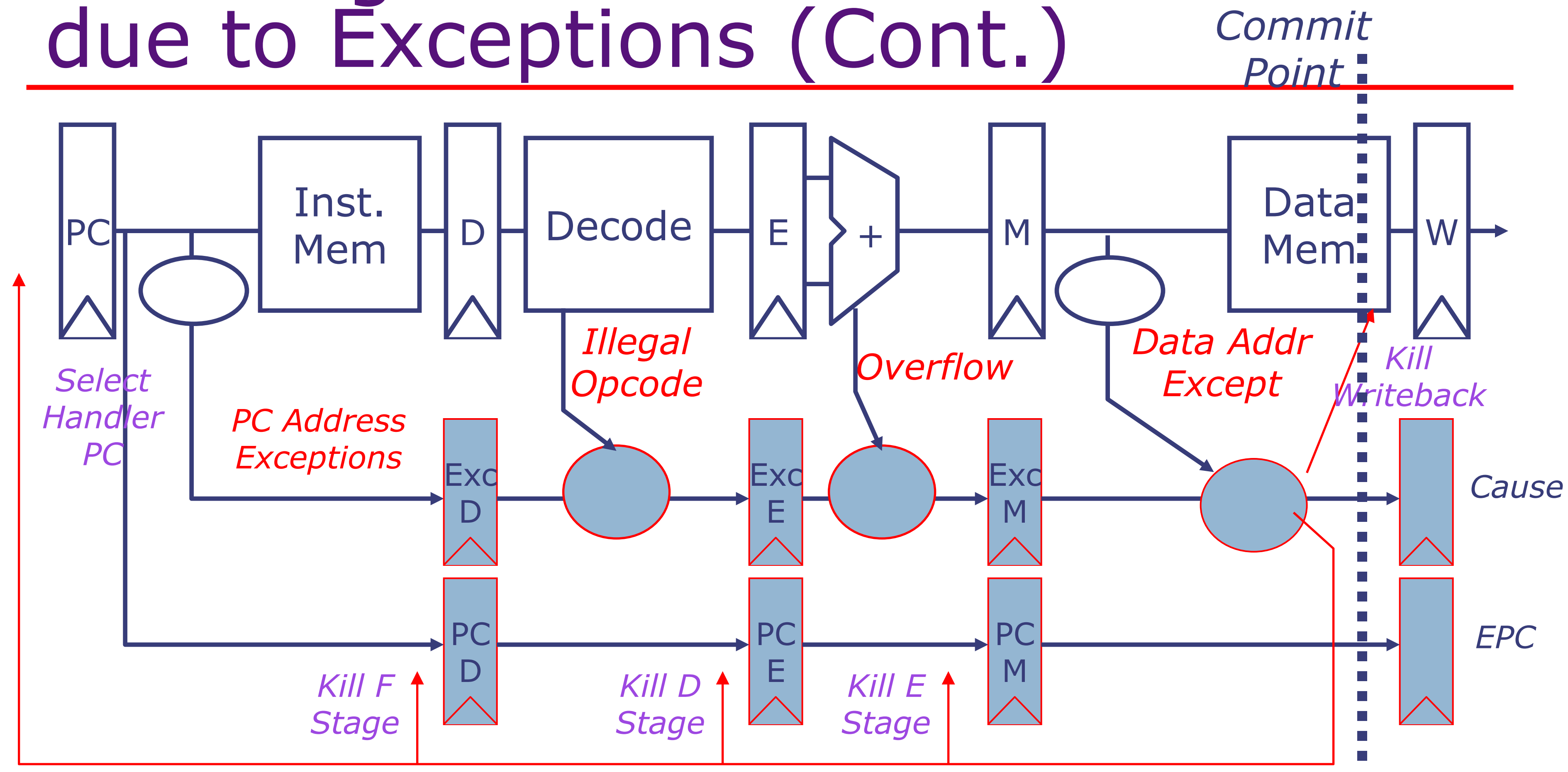
- Instructions may suffer exceptions in different pipeline stages
- Must prioritize exceptions from earlier instructions

Handling Control Hazards due to Exceptions (Cont.)



- Typical strategy: Record exceptions, process the first one to reach commit point (i.e., the point where architectural state is modified)

Handling Control Hazards due to Exceptions (Cont.)



- Typical strategy: Record exceptions, process the first one to reach commit point (i.e., the point where architectural state is modified)

Resolving Exceptions

- If an instruction has an exception at stage i
  - Kill instructions in stages i-1,…,1 (flush the pipeline)
  - Set PC to be the exception handler

```
sub x11, x2, x4
and x12, x2, x5
or  x13, x2, x6
??? x1, x2, x1  //unknown opcode
sub x15, x6, x7
lw  x16, 100(x7)
xor x1, x2, x3
```

```
//instructions to be invoked on
   an exception:
CH: sw x26, 1000(x10)
    sw x27, 1008(x10)
    …
```

unknown opcode

	1	2	3	4	5	6	7	8
IF	sub	and	or	???	sub	lw	xor	sw
DEC		sub	and	or	???	sub	lw	nop
EXE			sub	and	or	???	sub	nop
MEM				sub	and	or	???	nop
WB					sub	and	or	$\boxed{\text{nop}}$

commit point

Reference: Computer Organization and Design (RISC-V Edition)

Multiple Exceptions

Causes memory fault

Invalid opcode

```
lw x1, 4(x2)
???
mul x4, x5, x6
sub x7, x8, x9
```

```
//instructions to be invoked on an
   exception:
CH: sw x26, 1000(x10)
    sw x27, 1008(x10)
     …
```

	1	2	3	4	5
IF	lw	???	mul	sub	sw
DEC		lw	???	mul	nop
EXE			lw	???	nop
MEM				lw	nop
WB					$\boxed{\text{nop}}$

commit point

Invalid opcode detected

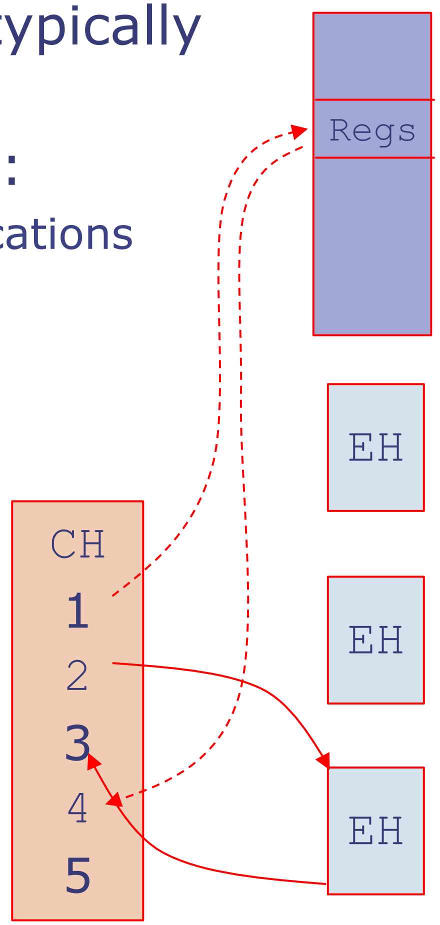
Memory fault detected

Works fine even if exception from latter instruction is detected first!

Reference: Computer Organization and Design (RISC-V Edition)

Typical Exception Handler Structure

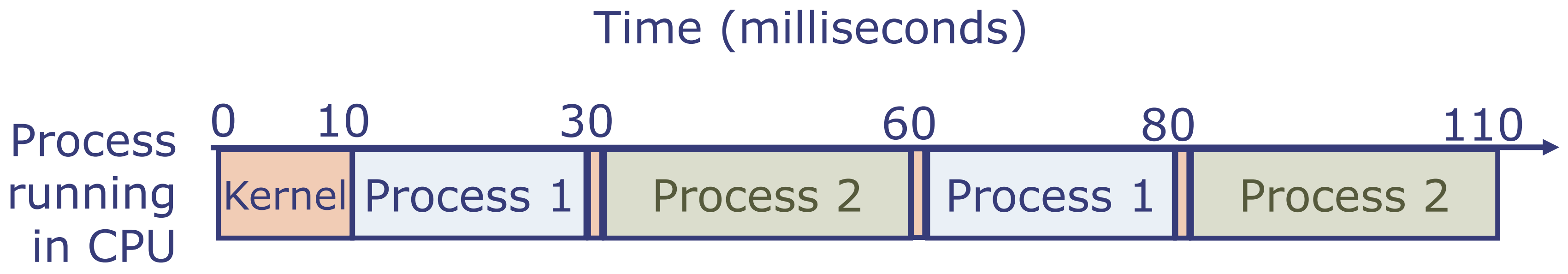
- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)
- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes `mcause`, process state to the right EH to handle the specific exception/interrupt
  3. EH returns which process should run next (could be the same or a different one)
  4. CH loads x1-x31, mepc from memory for the right process
  5. CH executes `mret`, which sets pc to mepc, disables supervisor mode, enables interrupts



Case Study 1 - CPU Scheduling

Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time



Case Study 1 - CPU Scheduling
Dispatches to a specific handler based on “cause”

- Schedule processes one after another, for a fixed amount of time.

```
typedef struct {
  int pc;
  int regs[31];
  …
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {
  if (cause == TIMER_INTERRUPT)
    return interrupt_timer(curProc); // process scheduling
  else if (cause == 0x08)
    // system call, e.g, OS service “write” to file
    return syscall_eh(curProc);
  else if (cause == 0x02) // illegal instruction
    return illegal_eh(curProc);
  else if (cause < 0) // external interrupt
    ...
}
```

Case Study 1 - CPU Scheduling
Implements a round robin scheduler

- Schedule processes one after another, for a fixed amount of time.

```
// an array to store state of all processes
ProcState procTbl[NUM_PROCS];

ProcState* interrupt_timer(ProcState* curProc) {
   int nextPid = curProc->pid + 1;
   if (nextPid >= NUM_PROCS)
      nextPid = 1; // Proc0 is kernel
   return &procTbl[nextPid];
}
```

Case Study 2 - Coordinating I/O Transfers

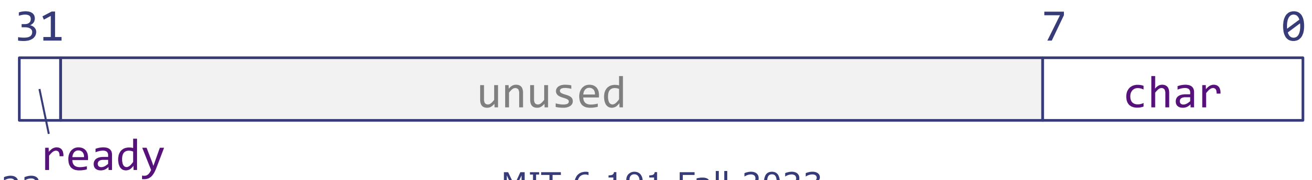
```
ProcState* eh_dispatcher(ProcState* curProc, int cause) {
  if (cause == TIMERINTERRUPT)
    return interrupt_timer(curProc); // process scheduling
  else if (cause == SYSCALL)
    // system call, e.g., OS service “write” to file
    return syscall_eh(curProc);
  else if (cause == 0x02) // illegal instruction
    return illegal_eh(curProc);
  else
    ...
}
ProcState* interrupt_timer(ProcState* curProc) {
  int nextPid = curProc->pid + 1;
  if (nextPid >= NUM_PROCS) nextPid = 1; // Proc0 is kernel
  print_string(“Switching processes on timer interrupt\n”);
  return &procTbl[nextPid];
}
```

Privileged mode print function

Case Study 2 - Coordinating I/O Transfers

```
void print_string(char* s) {
  // Privileged mode function for printing a string
  // iterate through a character array and call print_char
}

// Modification 1 (Wrong):
void print_char(int x) {
  // character display using mmio
  *display_mmio = x; // write to the mmio register
}
// Modification 2 (Correct):
void print_char(int x) {
  // polling-based character display using mmio
  while (*display_mmio < 0x80000000); // wait until ready
  *display_mmio = x; // write to the mmio register
}
```



Case Study 3 - System Calls

- The OS kernel lets processes invoke system services (e.g., access files) via system calls
- Processes invoke system calls by executing an instruction that causes an exception
  - Same mechanism as before!
- `ecall` instruction causes an exception, sets `mcause` CSR to a particular value
- Typically, similar conventions as a function call:
  - System call number in a7
  - Other arguments in a0-a6
  - Results in a0-a1 (or in memory)
  - All registers are preserved (treated as callee-saved)

Case Study 3 - System Calls

proc1.user.S

```
. = 0x0
start:
    // do this main loop forever
    mv t0, zero
    li t1, 0x1000
loop:
    addi t0, t0, 1
    blt t0, t1, next
    mv t0, zero
    la a0, hello_string
    li a7, 0x13
    ecall
next:
    j loop

hello_string:
    .ascii "Hello from process 1!\n\0"
```

proc2.user.S

```
. = 0x0
start:
    // do this main loop forever
    mv t0, zero
    li t1, 0x4000
loop:
    addi t0, t0, 1
    blt t0, t1, next
    mv t0, zero
    la a0, hello_string
    li a7, 0x13
    ecall
next:
    j loop

hello_string:
    .ascii "Hello from process 2!\n\0"
```

”print” system call for user processes 1 and 2

`SYS_print = 0x13`

Case Study 3 - System Calls

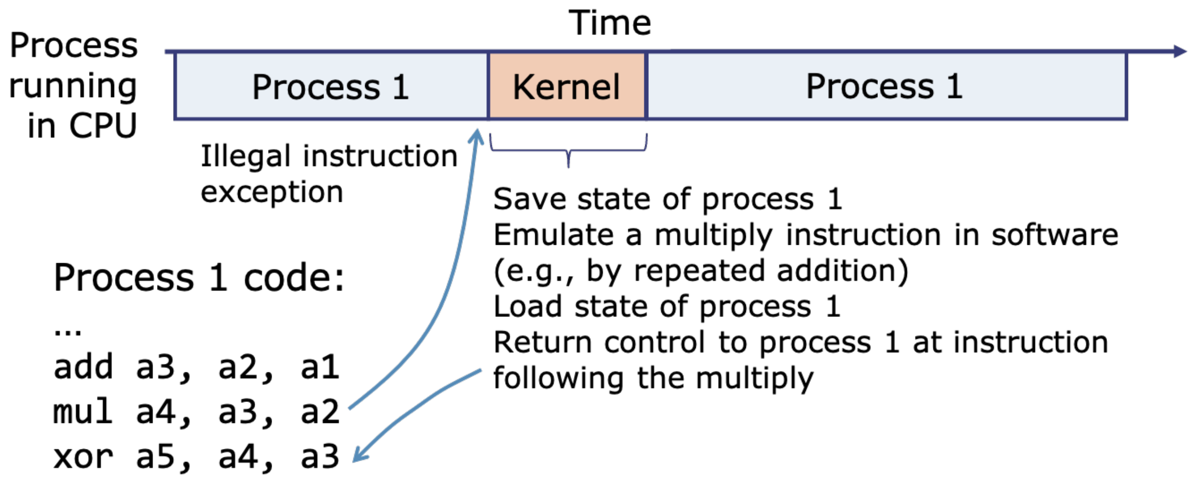
```
ProcState* syscall_eh(ProcState* curProc) {
  // Privileged mode function for printing a string
  // iterate through a character array and call print_char
  int syscall = curProc->regs[REG_a7];
  if (syscall == SYS_getpid) {
    curProc->regs[REG_a0] = curProc->pid;
  }
  else if (syscall == SYS_print) {
    print_string((char*)va_to_pa(curProc, curProc->regs[REG_a0]));
  }
  else {
    curProc->regs[REG_a0] = -128;
  }
}
```

Need to translate virtual (per-process) address to physical address

Case Study 4 - Emulating Instructions

Enabled by illegal instruction exceptions

- mul x1, x2, x3 is an instruction in the RISC-V 'M' extension (x1 := x2 * x3)
  - If 'M' is not implemented, this is an illegal instruction
- What happens if we run code from an RV32IM machine on an RV32I machine?
  - mul causes an illegal instruction exception (mcause = 2)
- The exception handler can emulate the instruction and return to the program at mepc+4
  - Requires incrementing mepc by 4 before mret
  - Result: Program believes it is executing in a RV32IM processor, when it's actually running in a RV32I


Time
Process running in CPU
Process 1
Kernel
Process 1
Illegal instruction exception
Save state of process 1
Emulate a multiply instruction in software (e.g., by repeated addition)
Load state of process 1
Return control to process 1 at instruction following the multiply
Process 1 code:

```
...
add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...
```

Case Study 4 - Emulating Instructions

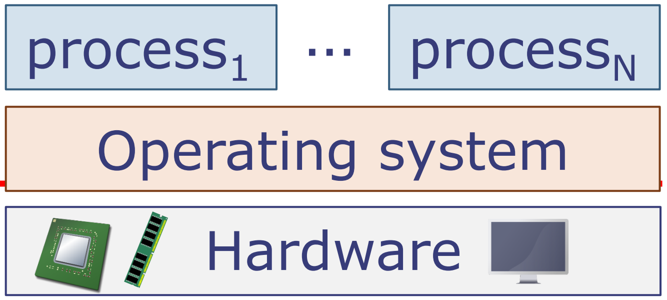
Enabled by illegal instruction exceptions

31 25	24 20	19 15	14 12	11 7	6 0
func7	rs2	rs1	func3	rd	opcode

R-type instruction

```
ProcState* illegal_eh(ProcState* curProc) {
    // load_mem fetches instruction
    int inst = load_mem(curProc->pc);
    // check opcode & function codes
    if ((inst & MASK_MUL) == MATCH_MUL) {
      // is MUL, extract rd, rs1, rs2 from inst
      int rd = (inst >> 7) & 0x01F;
      int rs1 = (inst >> 15) & 0x01F;
      int rs2 = (inst >> 20) & 0x01F;
      ...
      // emulate regs[rd] = regs[rs1] * regs[rs2]
      curProc->regs[rd] = multiply(curProc->regs[rs1],
                                   curProc->regs[rs2]);
      curProc->pc = curProc->pc + 4; // resume at pc+4
    }
    else abort();
    return curProc;
}
```

Summary



- I/O Interfaces and Methods:
  - Memory Mapped I/O (MMIO).
  - Polling
- Exceptions:
  - Unusual events or conditions that require immediate attention from the CPU.
  - Exceptions handling in the Pipeline
  - Exceptions Handlers (CH, EH)

Thank you!

Next Lecture: Parallel Processing