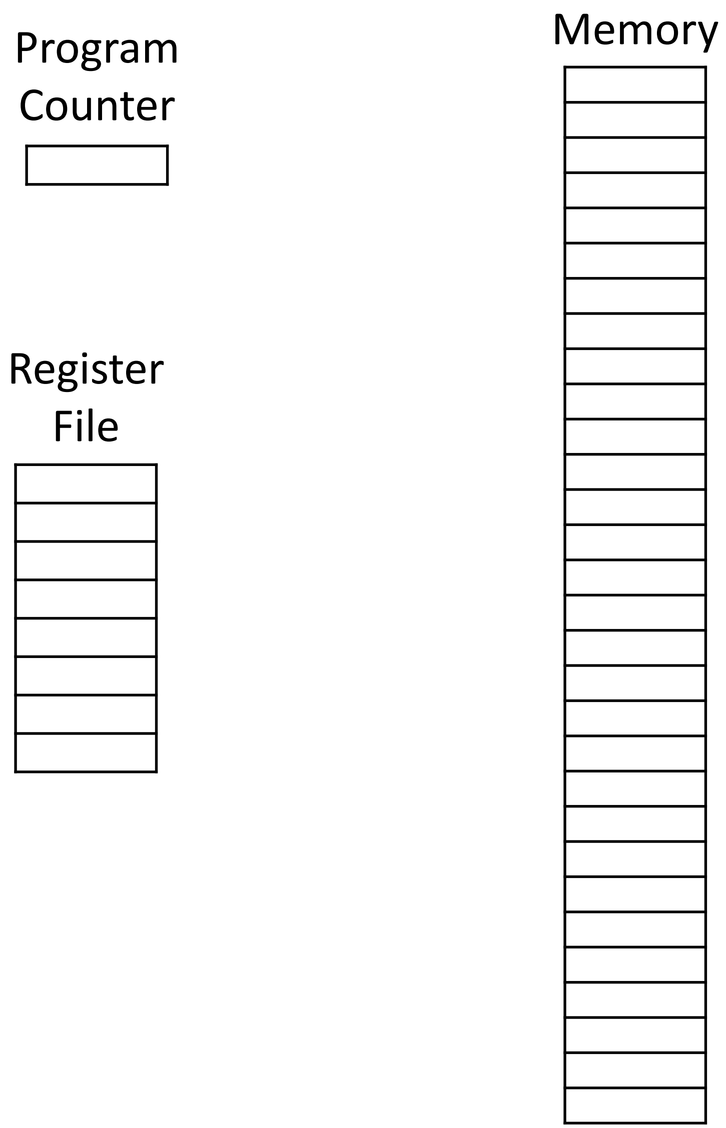
Interrupts and Exceptions

Today's lecture

- Use addressing to get data from the outside world
  - Data is moved from peripherals to memory
  - Addressing schemes
    - Memory-mapped vs. isolated I/O
  - Data movement schemes
    - Programmed I/O vs. Interrupt-driven I/O vs. Direct memory access

Most modern operating systems pre-emptively schedule programs

- If a computer is running two programs A and B, the O/S will periodically switch between them
  1. Stop A from running
  2. Copy A’s register values to memory
  3. Copy B’s register values from memory
  4. Start B running

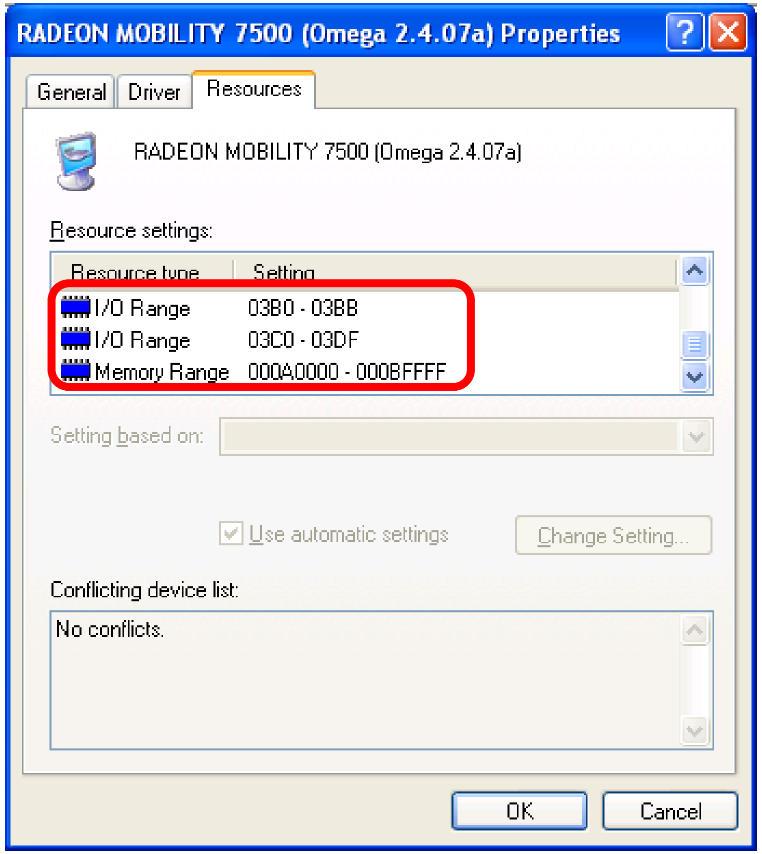


How does the O/S stop program A?

We can treat most devices “as if” they were memory with an “address” for reading/writing

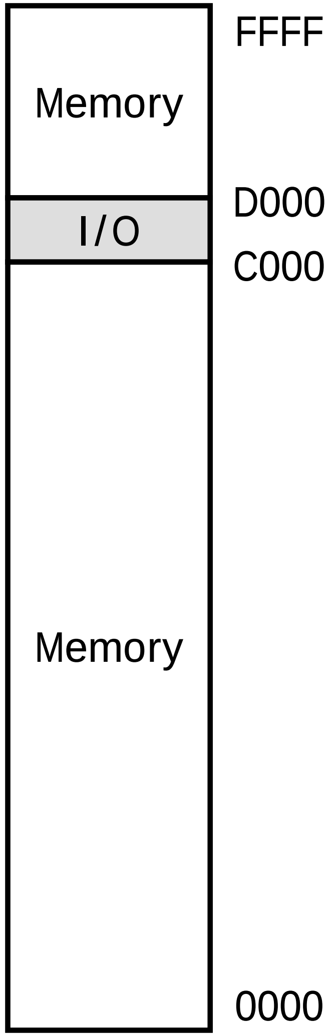
- Many ISAs often make this analogy explicit — to transfer data to/from a particular device, the CPU can access special addresses

- *Example*: Video card can be accessed via addresses 3B0-3BB, 3C0-3DF and A0000-BFFFF

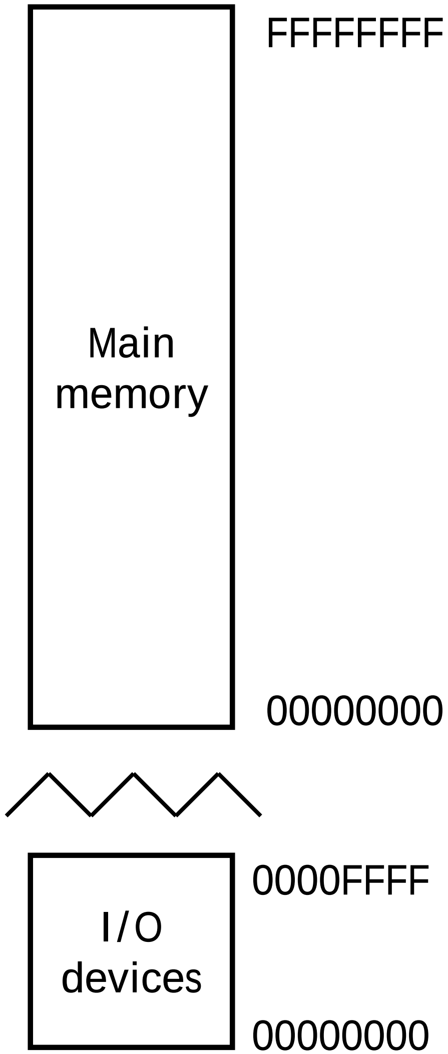


Most ISAs one of two protocols for addressing devices: memory-mapped I/O or isolated I/O

Memory-mapped I/O reserves a portion of main memory addresses for I/O

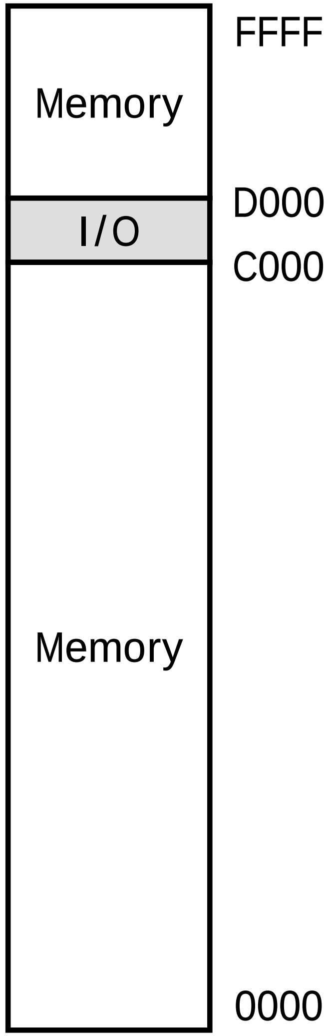


Isolated I/O creates a separate memory address space for devices

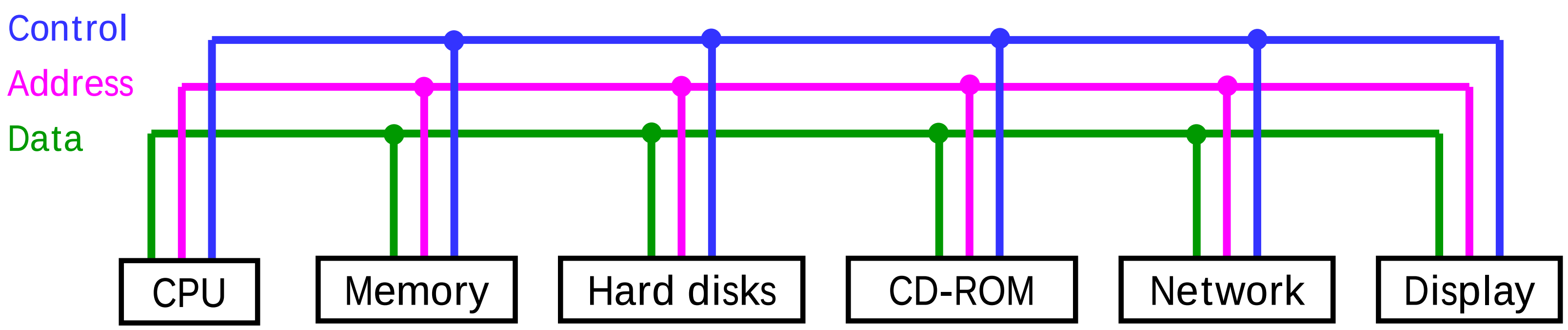


Memory-mapped I/O divides main memory addresses into actual memory and devices

- Apple IIe (right) had a 16-bit address bus
  - Addresses C000-CFFF accessed I/O devices.
  - No actual main memory at C000-CFFF
  - All other addresses reference main memory.
- I/O addresses are shared by many peripherals.
  - C010 → keyboard
  - C030 → speaker
- Some devices may need several I/O addresses.



We use control and addressing to determine when data goes to memory or devices

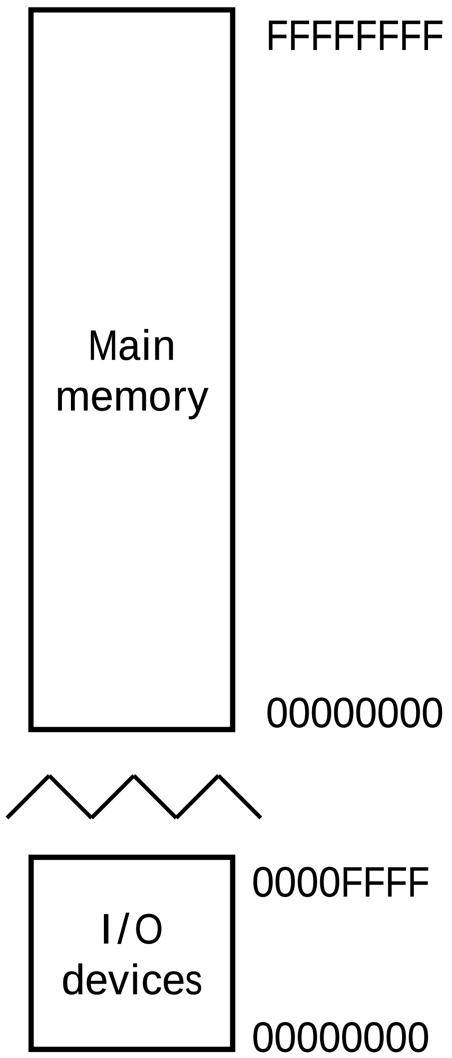


- Each device has to monitor the address bus to see if it is the target. (Apple IIe example)
  - Main memory ignores any transactions with addresses C000-CFFF.
  - The speaker only responds when C030 appears on the address bus.

Isolated I/O creates two separate address spaces and needs two sets of instructions

- *Example* (x86):
  - regular instructions like MOV reference RAM
  - special instructions IN and OUT access a separate I/O address space

- An address could refer to *either* main memory *or* an I/O device, depending on the instruction used



- Examples

```
lw    $reg, 0xffff0020($0)  # gets SPIMbot x-coord

sw    $reg, 0xffff0010($0)  # sets bot speed = $reg
```

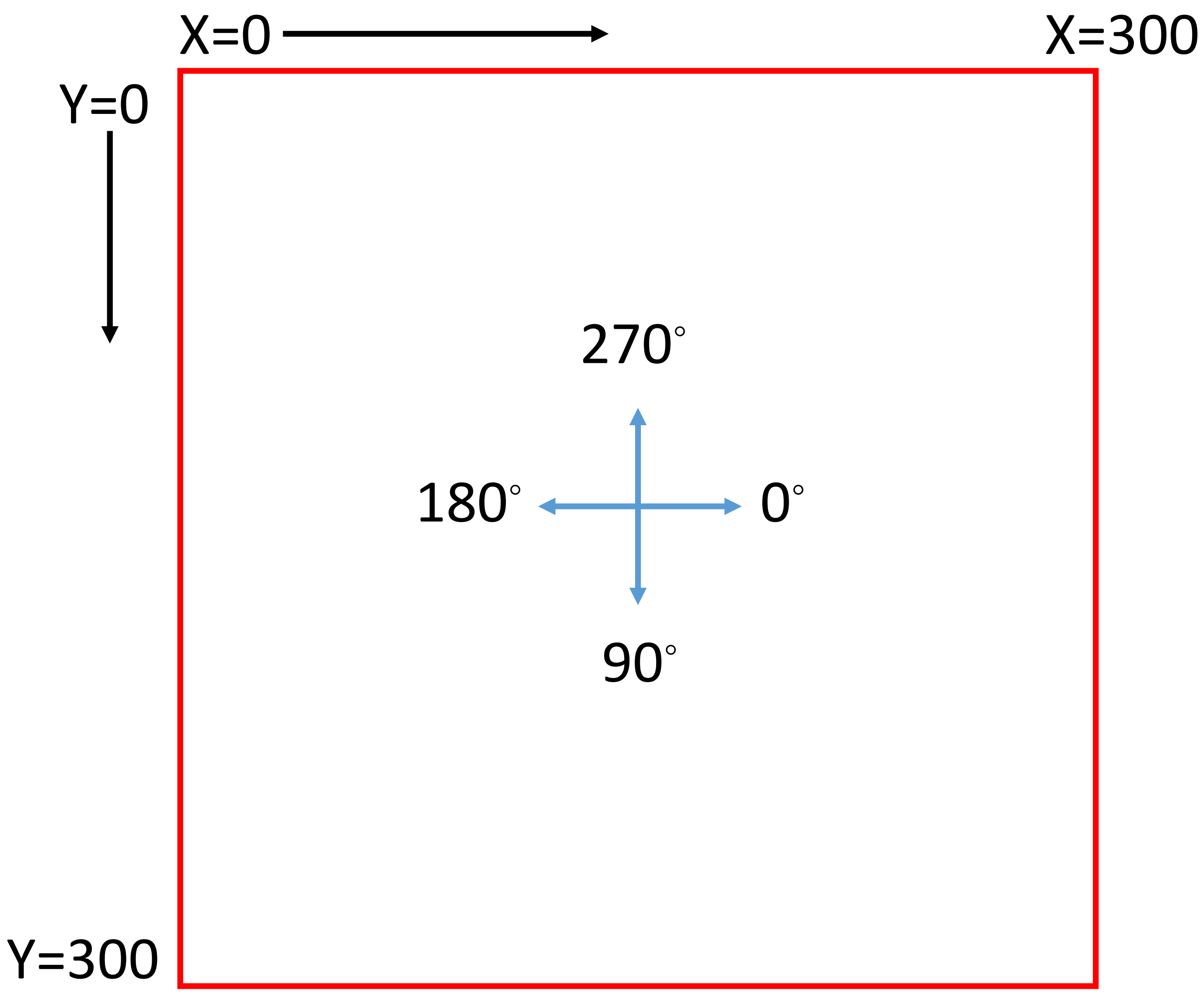
- Some control commands require a sequence of instructions

```
sw    $reg, 0xffff0014($0)
li     $t0, 1
sw     $t0, 0xffff0018        # sets bot angle = $reg
```

Example SPIMbot commands

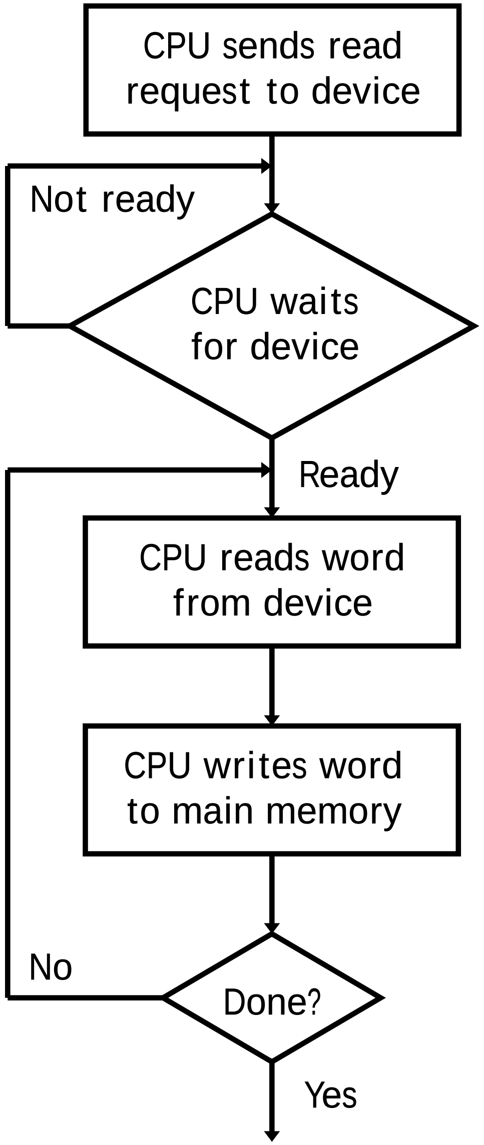
What	How
get SPIMbot's current x/y-coordinate	`lw` from `0xffff0020 (x)` `lw` from `0xffff0024 (y)`
set SPIMbot's angle (absolute)	`sw` the angle to `0xffff0014` `sw 1` to `0xffff0018`
set SPIMbot's angle (relative)	`sw` the angle to `0xffff0014` `sw 0` to `0xffff0018`
set SPIMbot's velocity	`sw` a number between `-10` and `10` to `0xffff0010`
read the current time	`lw` from `0xffff001c`
request a timer interrupt	`sw` the desired (future) time to `0xffff001c`
acknowledge a bonk interrupt	`sw` any value to `0xffff0060`
acknowledge a timer interrupt	`sw` any value to `0xffff006c`

SPIMbot coordinate system



In programmed I/O, the program or OS is responsible for transmitting data

- CPU makes a request and then waits (loops) until device is ready (loop 1)
- Buses are typically 32-64 bits wide, so loop 2 is repeated for large transfers
- Also called polling



Programmed I/O is generally bad

- A lot of CPU time is needed for this!
  - most devices *are* slow compared to CPUs
  - CPU also “wastes time” doing actual data transfer

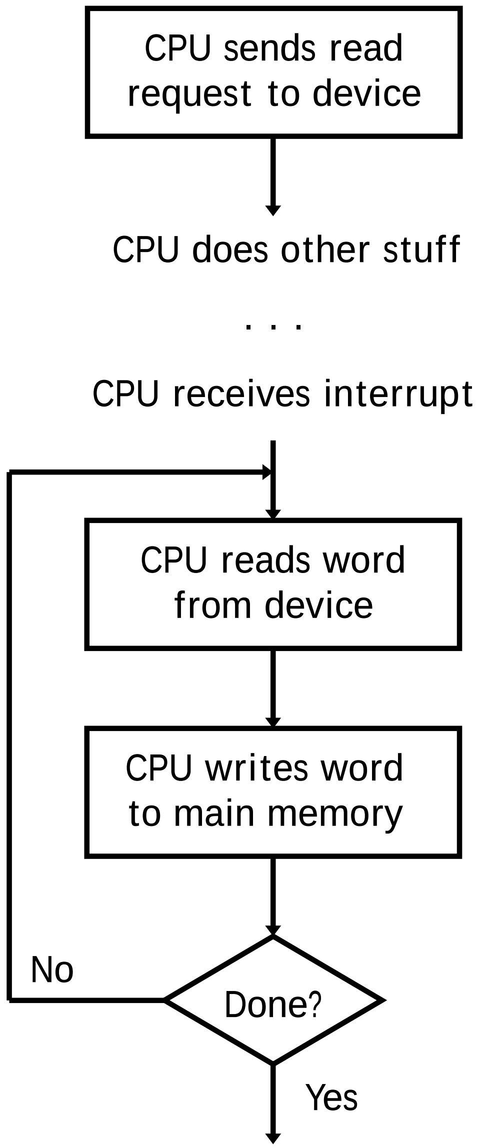
- CPU must ask repeatedly
- CPU must ask often enough to ensure that it doesn’t miss anything, which means it can’t do much else while waiting

Interrupt-driven I/O transfers data when devices interrupt the processor

Interrupt-driven I/O solves the inefficiencies of Programmed I/O

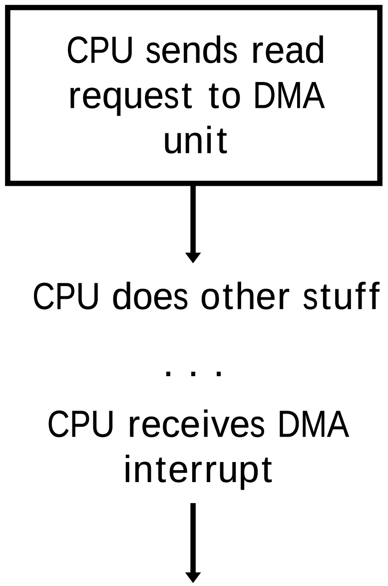
- Instead of waiting, the CPU continues with other calculations
- The device interrupts the processor when the data is ready

- CPU still does the data transfer

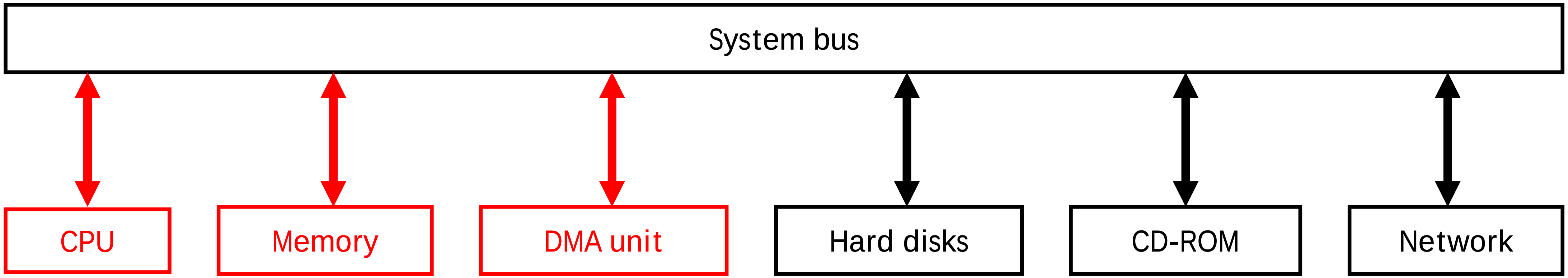


Direct memory access (DMA) parallelizes data transfer with a separate controller

- The DMA controller is a simple processor which manages I/O and memory data transfers
  - The CPU asks the DMA controller to transfer data between a device and main memory. After that, the CPU can continue with other tasks
  - The DMA controller issues requests to the right I/O device, waits, and manages the transfers between the device and main memory
  - Once finished, the DMA controller interrupts the CPU



Example of data transfer using DMA



Since both the processor and the DMA controller may need to access main memory, some form of arbitration is required

Interrupts vs. Exceptions

- External events that require the processor's attention
- Not an error, should be recoverable
- OS manages and resolves the interrupt

- Typically errors that are detected within the processor
- Always an error, may or may not be recoverable
- OS must resolve the exception or ask the program to resolve

More details on interrupts

- *Examples*: I/O device needs attention, timer interrupts to mark cycle
- All interrupts are recoverable: interrupted program should resume after the interrupt is handled
- OS responsible to do the right thing, such as:
  - Save the current state and shut down the hardware devices
  - Find and load the correct data from the hard disk
  - Transfer data to/from the I/O device, or install drivers

More details on exceptions

- *Examples*: illegal instruction opcode, arithmetic overflow, or attempts to divide by 0
- There are two possible ways of resolving these errors:
  - If the error is un-recoverable, the operating system kills the program
  - Less serious problems can often be fixed by OS or the program itself

Sometimes users want to handle their own exceptions:

- Example: numerical applications can scale values to avoid floating point overflow/underflow

- Many operating systems provide a mechanism for applications for handling their exceptions
  - Unix lets you register “signal handler” functions

- Modern languages like Java provide programmers with language features to “catch” exceptions (this is much cleaner)

ISA's are periodically extended creating backwards compatibility problems.

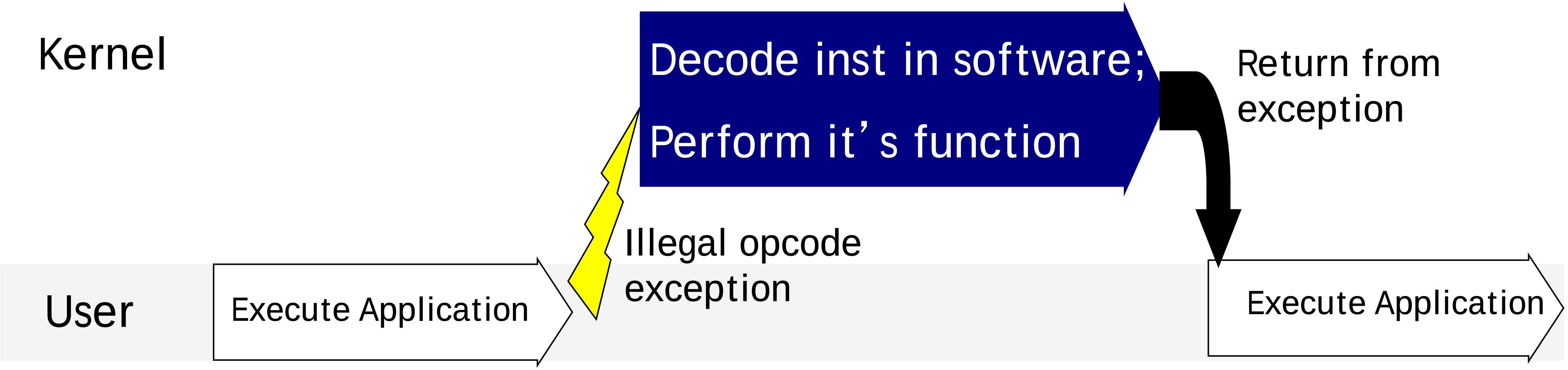
- Examples: MMX, SSE, SSE2, etc.

- Create illegal opcode exceptions

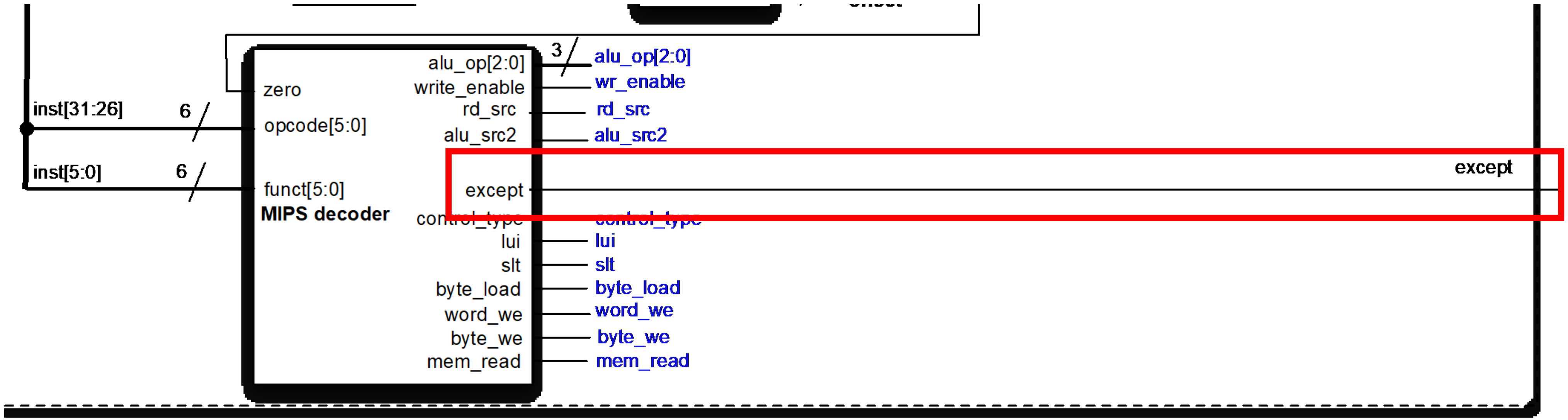
- Programs compiled with these new instructions will not run on older implementations (e.g., a 486)
  - "Forward compatibility" problem

Instruction Emulation makes these illegal opcode exceptions recoverable

Can’t change existing hardware, so we add software to “emulate” these instructions



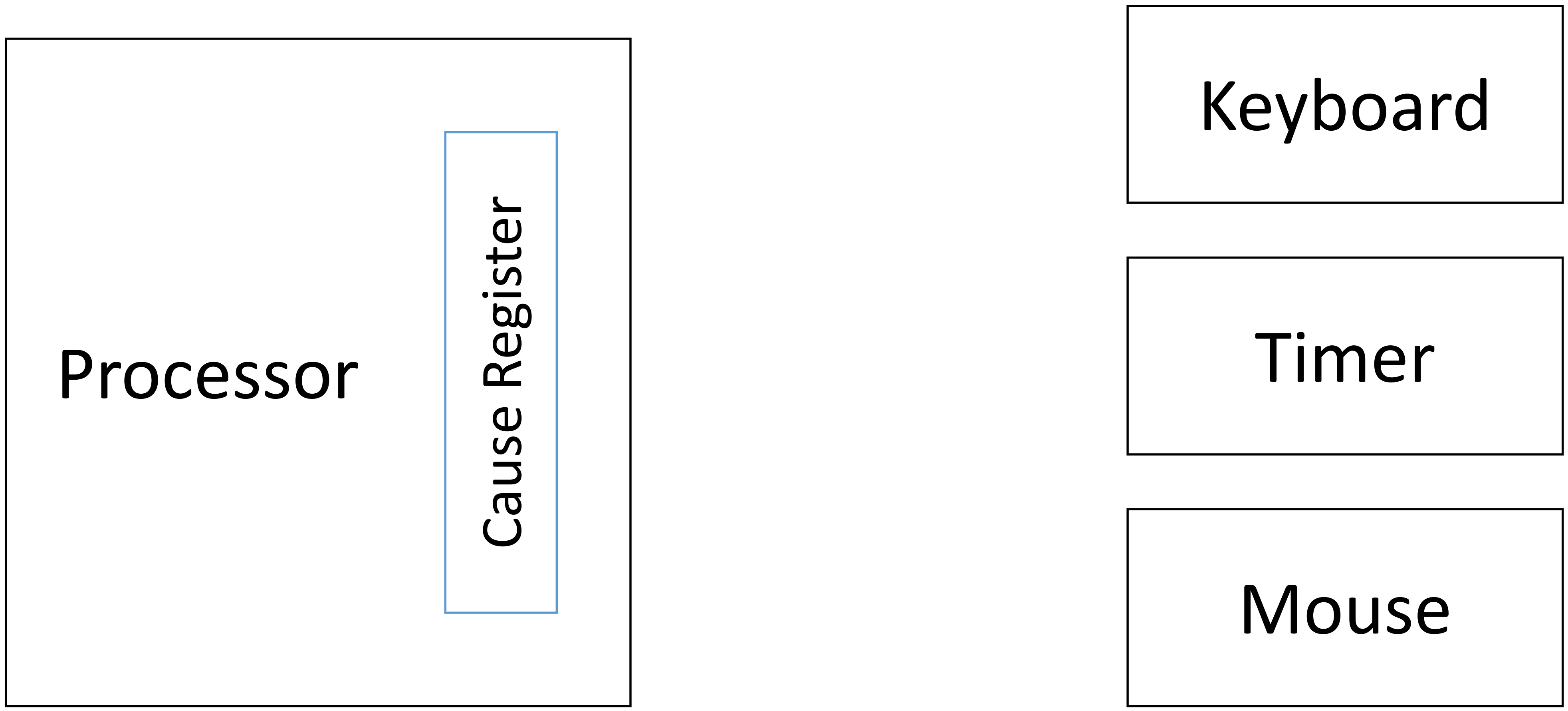
Hardware Decoder raises the Reserved Instruction Exception



Different types of exceptions in MIPS32 are encoded with different numbers

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

In hardware, devices send interrupts and the processor acknowledges when those interrupts have been processed or “handled”



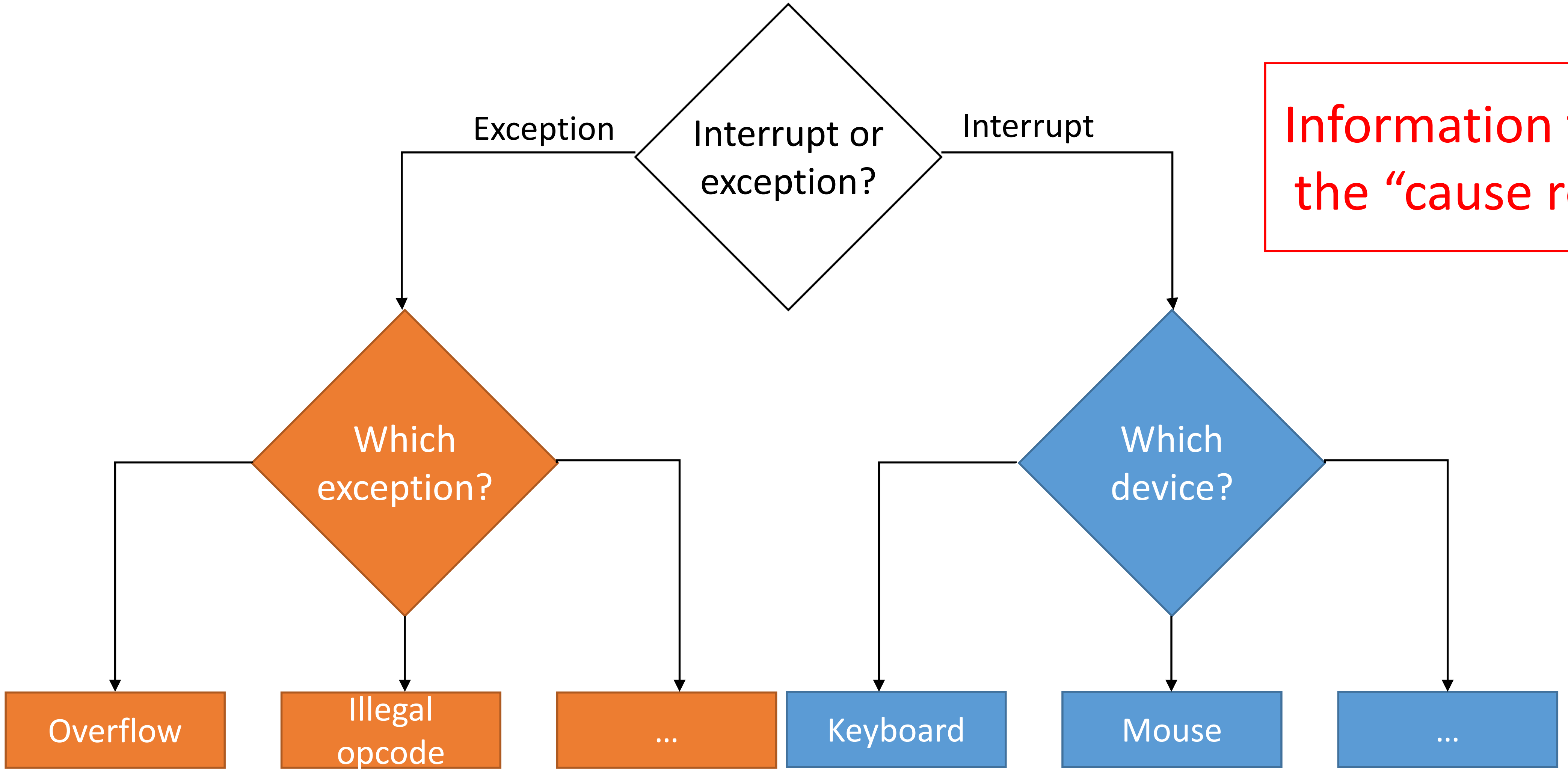
In software, exceptions and interrupts are processed by an “interrupt handler”

1. An exception/interrupt occurs
2. The program is stopped
3. Control of the processor is given to the operating system by changing PC to the address of the interrupt handler
   - In MIPS32, the interrupt handler starts at address 0x80000180
4. The interrupt handler code processes the exception/interrupt
   - If an interrupt, the handler will acknowledge the interrupt
5. If the exception/interrupt is recoverable, control of the processor is returned to the program

The interrupt handler must know which instruction was executing when the interrupt/exception occurred

- The program counter will be set to 0x80000180
- The program’s current PC must be saved
- Saving the PC also helps with error reporting

The interrupt handler must know the cause of the interrupt/exception

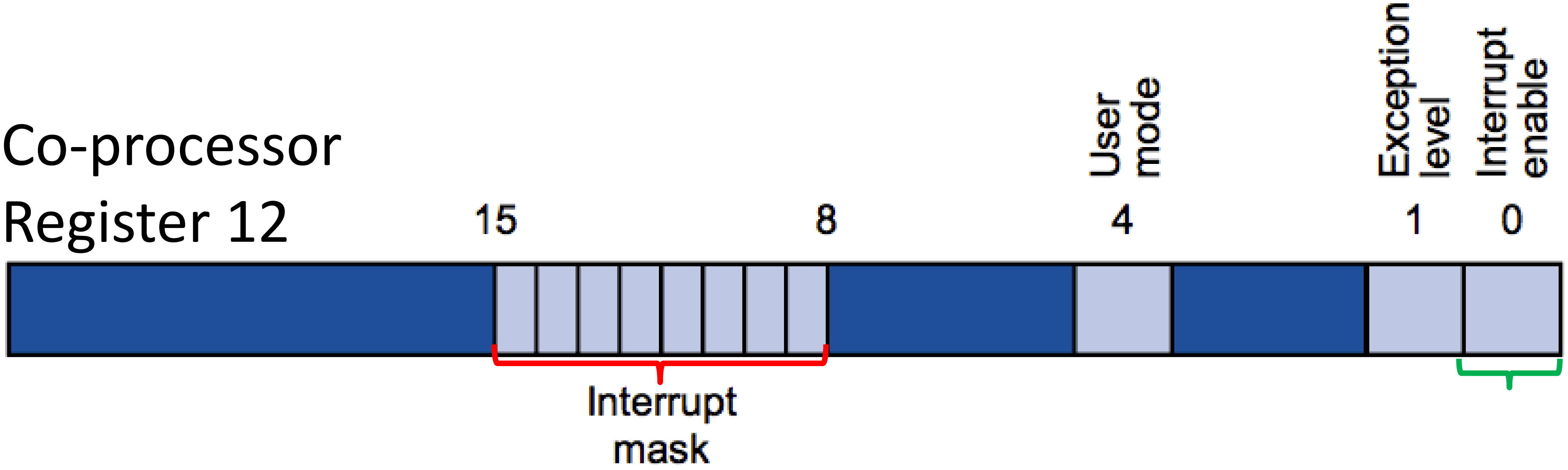


Information found in the "cause register"

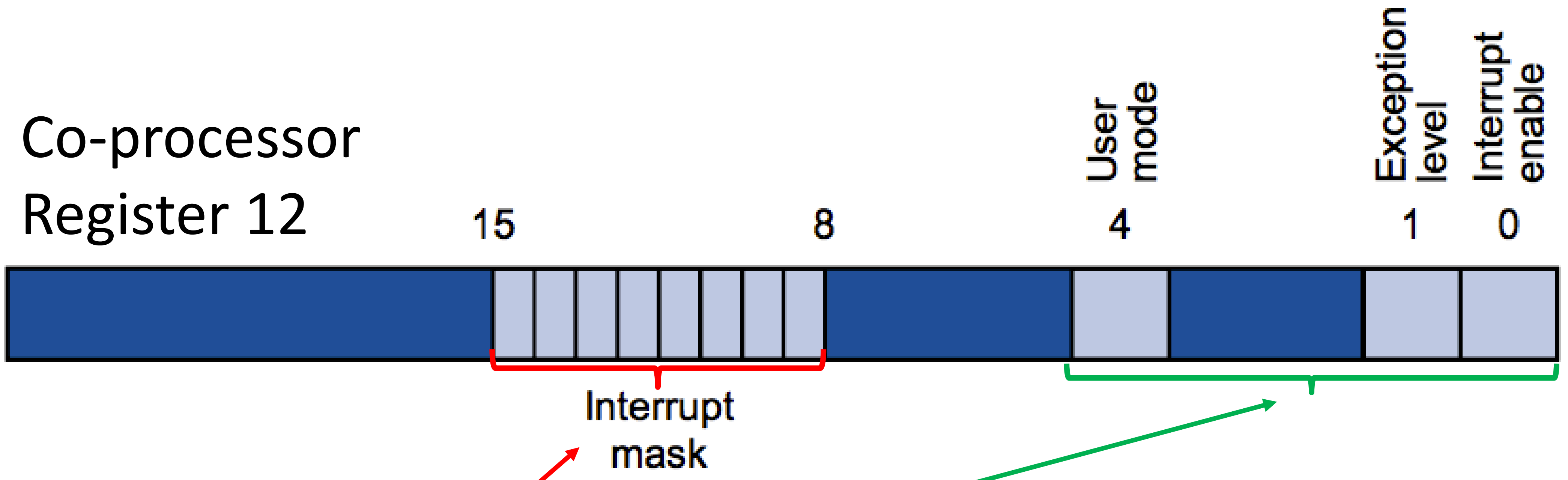
To receive interrupts, the software has to enable them

- MIPS: permissions set with the Status register (on the co-processor)
- Enable interrupts by setting bit zero
- Select which interrupts to receive by setting one or more of bits 8-15
- User mode is 0 for user, 1 for kernel

Co-processor
Register 12



Control the status register by moving data to the co-processor

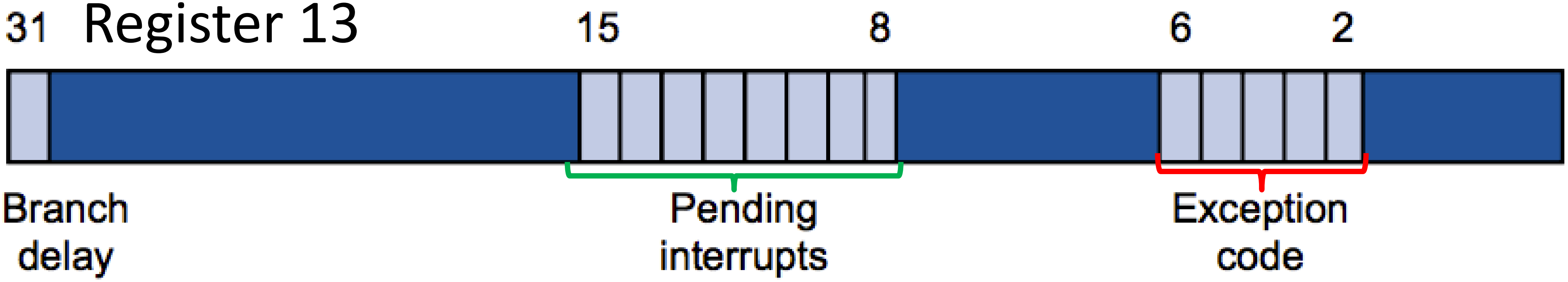



```
li     $t0, 0x1001  # enable interrupts and interrupt 12
mtc0   $t0, $12     # set Status register = $12
                    # move to coprocessor 0
```

When an interrupt/exception occurs, the Cause Register indicates which one

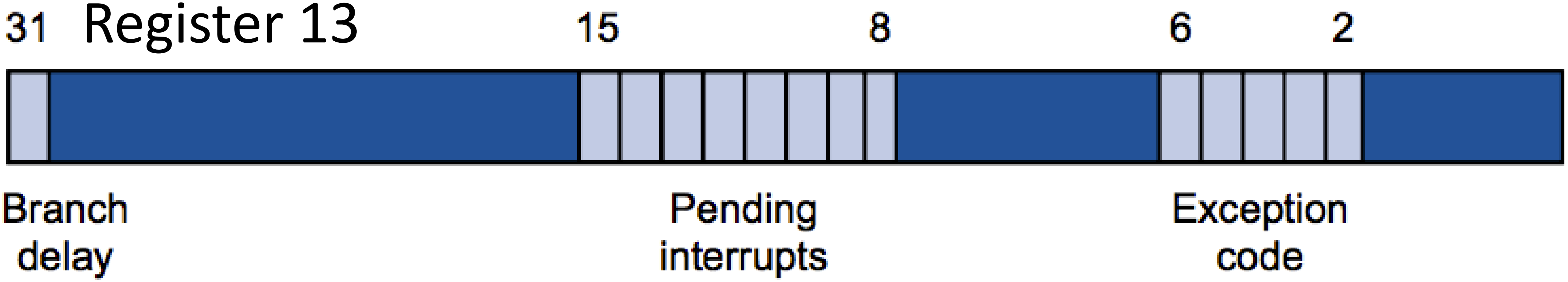
- For an exception, the exception code field holds the exception type
- For an interrupt, the exception code field is 00000
  - External devices set the bits for pending interrupts

Co-processor Register 13



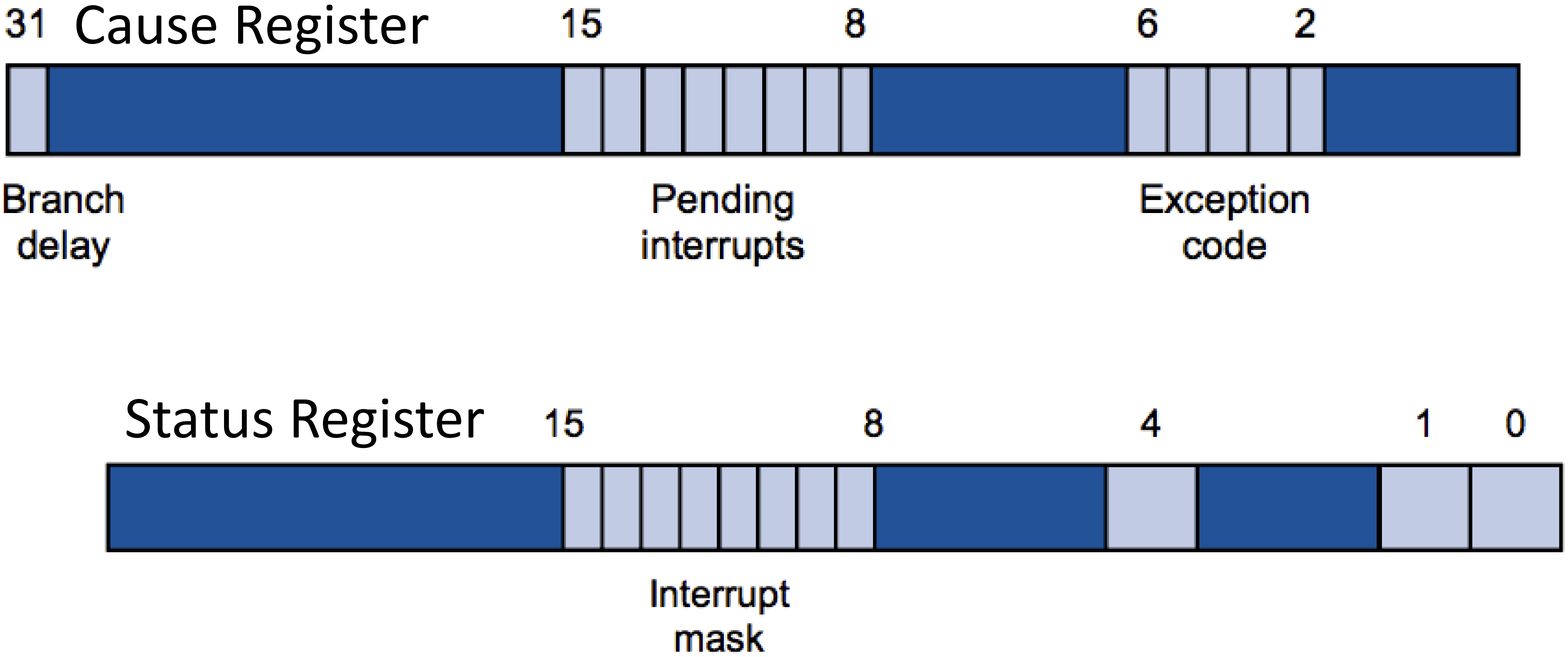
Handle interrupts/exceptions by moving data from the co-processor

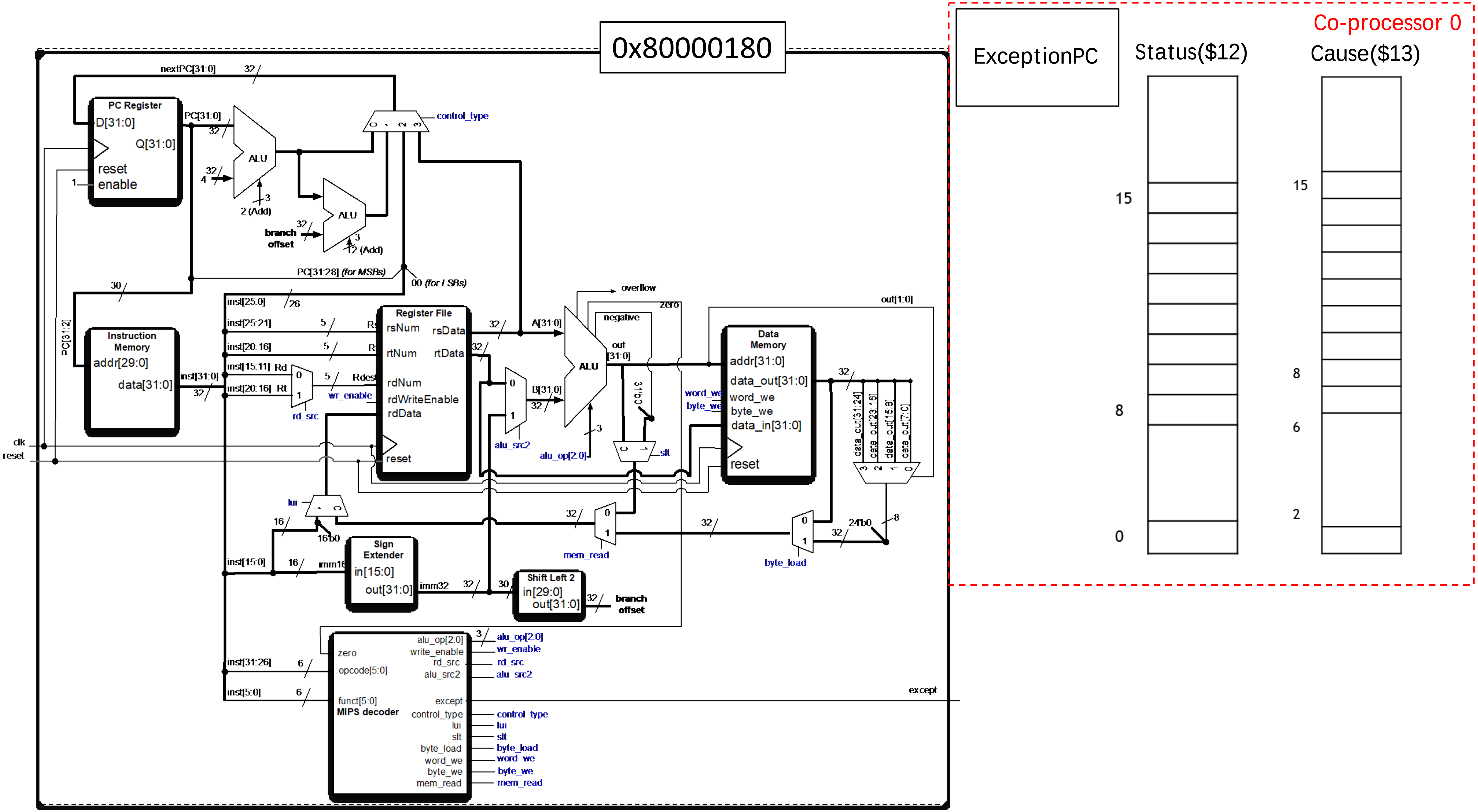
Co-processor Register 13




```
mfc0  $t0, $13 # Get Cause register
srl   $t1, $t0, 2
andi  $t1, $t1, 0x1F
bne   $t1, $0, non_interrupt
```

The status register and cause registers must both have a 1 in the same bit position to process an interrupt (interrupts need to be enabled)





0x80000180
Co-processor 0
ExceptionPC
Status($12)
Cause($13)
15
8
0
15
8
6
2
nextPC[31:0]
PC Register
D[31:0]
Q[31:0]
reset
enable
PC[31:0]
ALU
2 (Add)
branch offset
control_type
PC[31:28] (for MSBs)
00 (for LSBs)
Instruction Memory
addr[29:0]
data[31:0]
inst[31:0]
PC[31:2]
clk
reset
inst[25:0]
inst[25:21]
inst[20:16]
inst[15:11] Rd
inst[20:16] Rt
Rdest
rd_src
wr_enable
Register File
rsNum
rsData
rtNum
rtData
rdNum
rdWriteEnable
rdData
A[31:0]
B[31:0]
alu_src2
alu_op[2:0]
overflow
zero
negative
out[31:0]
31'b0
slt
word_we
byte_we
out[1:0]
Data Memory
addr[31:0]
data_out[31:0]
word_we
byte_we
data_in[31:0]
data_out[31:24]
data_out[23:16]
data_out[15:8]
data_out[7:0]
24'b0
byte_load
mem_read
lui
16'b0
inst[15:0]
imm16
Sign Extender
in[15:0]
out[31:0]
imm32
Shift Left 2
in[29:0]
out[31:0]
inst[31:26]
inst[5:0]
zero
opcode[5:0]
funct[5:0]
MIPS decoder
alu_op[2:0]
write_enable
rd_src
alu_src2
except
control_type
lui
slt
byte_load
word_we
byte_we
mem_read
except