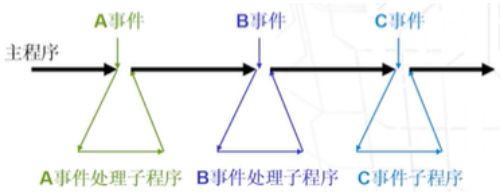
7 I/O Control & Interrupts

7.1 Interrupts & Exceptions

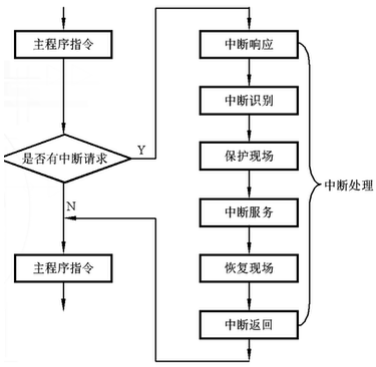
7.1.1 Basic Introductions

Interrupts: When the program is executing, some events may occur and need to be handled immediately. Then CPU will stop executing the current program and execute handler to handle these events. After the events are handled successfully, CPU will re-execute the program from the point it was interrupted.



- Categories: Internal interrupts & External interrupts.
  - **Internal interrupts** come from the system.
    - **Voluntary interrupts**（自愿中断）: the program request some services from OS, such as I/O operations. Use instruction `INT xx`. Very similar to function calls.
    - **Forced interrupts**（强迫中断）:
      - Software error (*exceptions*): such as divide zero, overflow, undefined instruction, segment fault.
      - Hardware error: such as voltage error, hardware breakdown.
  - **External interrupts** come from the peripheral devices, such as clock interrupt, keyboard signal, shutdown, end of transfer. External interrupts are all **forced interrupts**（强迫中断）.
- Interrupts give the computer the ability to solve the problems more dynamically and flexibly, which improve the handling ability of system.

Interrupt Handling



- Check if interrupts occur every time after one instruction is finished;
- Save the breakpoint and contexts;
- Handle interrupts (Use interrupt handle service);
- Retrieve contexts and recover from breakpoint;

Precise / Imprecise Interrupt

- **Imprecise interrupt**: Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.
    - For example, we only handle interrupt in every pipeline stage, therefore it's difficult to associate the interrupt with the exact instruction.
    - May cause some problems in pipelined execution.
- **Precise interrupt**: An interrupt or exception that is always associated with the correct instruction in pipelined computers.
    - Another way to define *precise interrupts* is: Precise interrupts if and only if:
        - All the instructions before have committed their states;
        - All the instructions after do not change the state of the computer.
    - Therefore, we can associate the interrupts with the exact instruction that was the cause of the interrupts.

7.1.2 Interrupt Handler

Detect the interrupt source

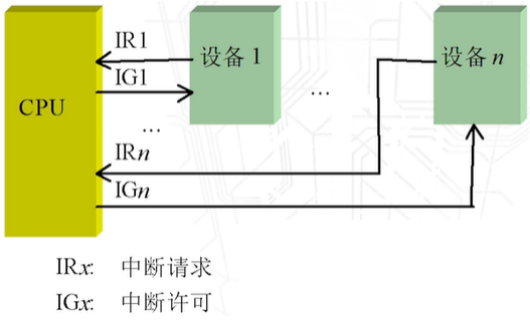
- **CPU round-robin detection** （CPU轮询）: CPU asks every device (read the status bit from every device) to find out the source of the interrupt; *Simpler hardware* but *slow*.
- **Cause register**（原因寄存器）: In MIPS, we use the cause register to store the reasons of exceptions, and CPU can find the source from the reasons.
- **Interrupt vector**（中断向量）: In x86, the interrupt device send a special code (the vector table index) to CPU, so CPU can find the source of an interrupt.

The entrance to the interrupt handle service

- **Non-vectored interrupts**（非向量中断）: Find the source of the interrupt, and according to offset value in the interrupt reasons, calculate the entrance of the interrupt handle service function.
- **Vectored interrupts**（向量中断）: (x86) Store the entrance to the interrupt handle service function in the **Interrupt Vector Table**（中断向量表）. Find the entrance with the special code (the vector table index).
    - The introductions of interrupts in 8086 can refer to the notes of EI209.

Interrupt Award（中断裁决）: When multiple interrupts occur, the CPU must choose which one to handle first.

- **Independent interrupt request**（独立请求）: Every device has its own interrupt request line and interrupt grant line. Set the priority of every device, and the circuit in the CPU decides which one to respond first according to their priorities.



- **Chain interrupt request**（链式请求）: Several devices share the same interrupt pin in CPU, and CPU handle the interrupts one by one following the device chain. You can refer to the notes of Chap 9 in EI209 for detail information.
- **Group chain interrupt request**（分组链式请求）: Combine two methods above together.

Interrupt Mask（中断屏蔽）

- **Single interrupt**（单重中断）: Another interrupt can not occur when handling the current interrupt. Need to close the interrupt service before handling, and re-open it after handling.
  - The brief process: **close interrupt service**, save the context, identify the interrupt, find the entrance to interrupt service, specific interrupt handle service, retrieve the context, **re-open interrupt service**.
- **Multiple interrupts**（多重中断）: The interrupt handler can be interrupted by another program (usually the more urgent one), which forms nested interrupts（中断嵌套）.
  - The brief process: **close interrupt service**, save the context, identify the interrupt, find the entrance to interrupt service, open interrupt service, specific interrupt handle service, close interrupt service, retrieve the context, **open interrupt service**.
- In 8086, we can set the register `IF` (interrupt flag) to enable interrupt mask. (`IF = 1` means allowing CPU to mask interrupts). Use `STI` and `CLI` to set or clear the flag register. **IF** does not affect the non-maskable interrupts（对不可屏蔽中断不起作用）.

Return from Interrupt Handler

- In x86, we use `IRET` to return from interrupt handler. (`IRET` includes retrieving the contexts from stack).

7.2 Interrupts & Exceptions in MIPS

Registers About Exceptions & Interrupts in MIPS

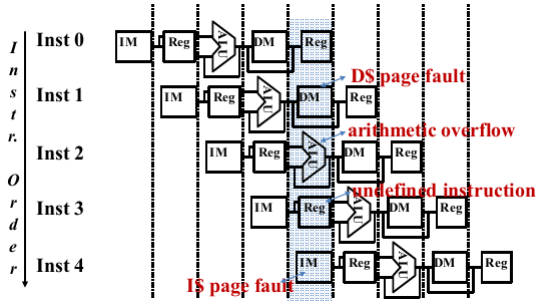
- **EPC**: register that stores the address of exceptions.
- **Cause**: register that stores the reasons of exceptions.
- **Status**: registers that control interruptions.
- No interrupt vector!

Handling Exception

- Set `EPC` to the address of exceptions;
- Set one control digit of `Status` to force CPU to go into kernel mode, "close interrupt" (forbid any interruptions).
- Set `Cause` register to the reason of exceptions.
- Go to the unique entrance of handling exception, and handle the exception with the help of software.
- *Handling Exceptions*...
- Execute `eret` to put the content of `EPC` into register `PC`, clear the control digit of `Status` to "re-open interrupt" (allow interruptions), and CPU go into user mode;
- Continue to execute the next instructions.

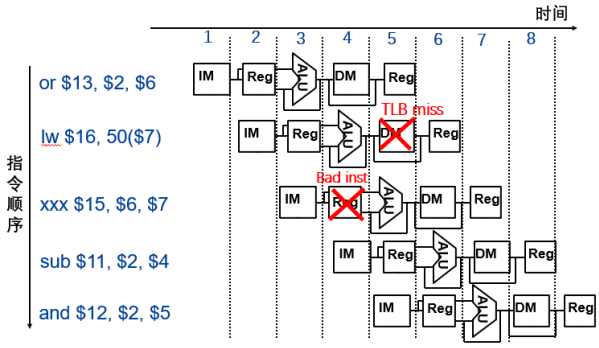
Imprecise Interrupts

- Handle the interrupts in the current stage of the instruction.
- Clear the status of next few instructions: using `flush` to clear the segment registers.
- If there are multiple interrupts at the same time: *handle the earliest interrupt first*.



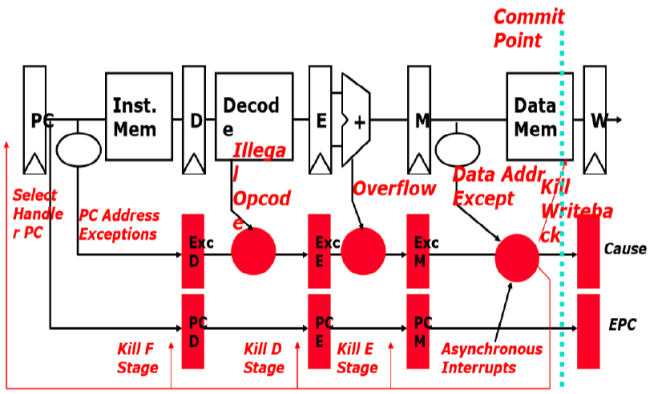
- Some problems may occur.
  - We detect exceptions at the 4th cycle, so we handle the interrupt. After the interrupt handling, the pipeline will restart from the 3rd instruction, which may cause the *result loss* of the 1st and 2nd instruction.

    An Possible Solution: do not restart the whole pipeline, restart the 3rd instruction and the instructions after it.



  - It's difficult for programmer to set the exact breakpoint when debugging, because when handling the interrupts, the previous instructions have not finished executing yet.

Precise Interrupts



- Hold exception flags in pipeline until commit point (M stage)
  - Exceptions in earlier pipe stages override later exceptions
  - Inject external interrupts at commit point (override others)
  - If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

- Do not handle the interrupt immediately;
- Set the exception flags in segment registers;
- Before writing back to memory or register, handle the interrupts.
- After handling the interrupts, restart the pipeline from the current instruction.
- All interrupts will be sorted in order, therefore the problem in the imprecise interrupts won't happen.

7.3 I/O Device Controller

Categories of peripheral

- According to speed:

    - **Low-speed peripheral**（低速设备）: Several bytes per second, such as keyboard and mouse.
    - **Middle-speed peripheral**（中速设备）: thousands of bytes per second, such as printer.
    - **High-speed peripheral**（高速设备）: millions of bytes per second, such as tape streamer（磁带机）.
- According to contents:
    - **Block peripheral**（块设备）: the information read/write unit is a block, such as disk.
    - **Character peripheral**（字符设备）: use for the input/output of characters, such as keyboard, monitor and printer.

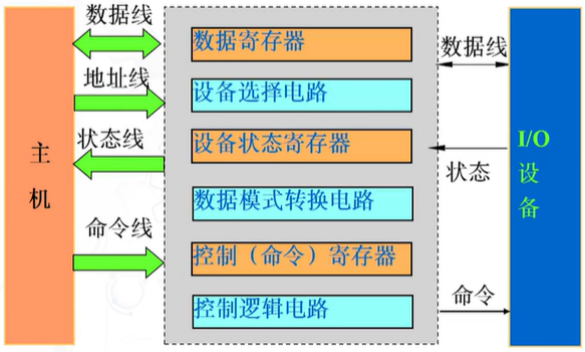
Features

- Much slower than CPU;
- Different format of information among different peripherals;
- Therefore, need I/O device controller（设备控制器）.

Different Type of I/O: refer to notes of EI209 for more details.

- Memory-mapped I/O: The port address of I/O devices shares the same address system with the memory address.
    - Pros:
        - Can directly perform write/read operation to the buffer and registers of I/O devices;
        - Can use the memory access instructions, do not need extra instructions;
        - Simple design and less pins.
    - Cons:
        - Less memory storage because of I/O ports;
        - Longer instruction than isolated I/O instruction, and slower speed.
- Isolated I/O: Special designed addresses and instructions for I/O devices.
    - Pros:
        - I/O does not use the storage of memory;
        - Shorter instruction, and higher speed;
        - Shorter I/O port address, easier to decode;
        - Programs are easy to understand.
    - Cons:
        - Less flexibility;
        - Need two control logic (memory & I/O ports), more complex.

I/O Device Controller



- **Data register**: acts as data buffer between CPU and I/O devices;
- **State register**: store the state of each device;

- **Control register**: store the control word from CPU and send it to the device after instruction transformation.

7.4 The Control Method of I/O Devices

Program Control Method（程序控制）

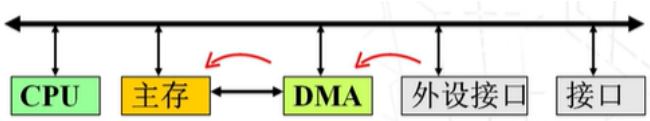
- **Program request**（程序查询） （Programmed I/O)
- **Interrupts**（中断）

Program request: CPU sends request to each I/O device to get the real-time status.

- When devices are not ready, the CPU must wait.
- Slow speed because of frequent requests.

Interrupts: when I/O devices are ready, they send interrupts to CPU. Refer to notes of Chap 2 in EI209 for details.

DMA: Direct Memory Access. An extra path to transfer data between I/O devices and memory, do not occupy the data bus. Refer to notes of Chap 2 in EI209 for details.

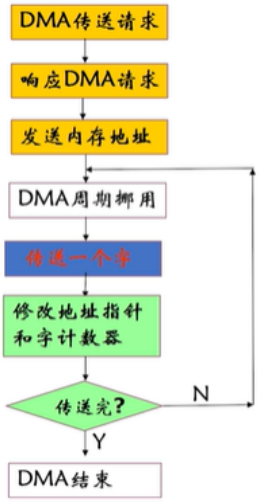


Memory Contention（内存争用）: CPU needs to access data in the memory while DMA needs to access data in memory, too.

- **Pause the CPU access process**（停止CPU使用主存）: CPU waits when DMA is transferring. Simple but low-efficient.
- **Cycle Stealing**（周期挪用/周期窃取）: DMA fully uses the cycles that CPU do not access memory to transfer data. If contentions still happen, then DMA accesses first.

DMA data transfer process

- CPU tell DMA controller (DMAC): read/write, device address, starting address of memory block for data, amount of data to be transferred, ...
- When the peripheral is ready, I/O ports send transfer request to DMAC;
- DMAC responds to the request;
- DMAC begins the transfer process between DMA and I/O devices;
- DMAC begins the transfer process between DMA and memory (using cycle stealing);



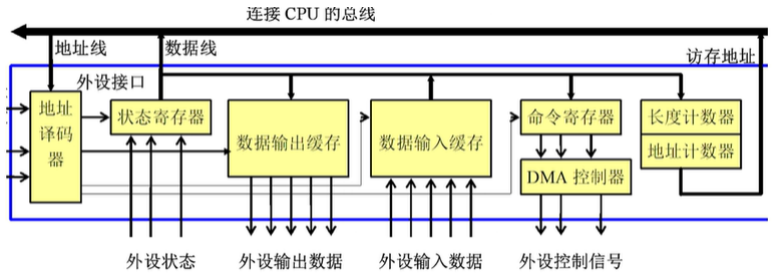
- Repeat the last two steps, until the data transfer is finished.

CPU does not involve in the whole transfer process!!!

DMA end stage（结束阶段）

- Two state:
  - Normally finish the process（正常结束）
  - Abnormally finish the process（异常结束）
- Send interrupt to CPU, CPU will handle the interrupts according to the state.

DMA controller



DMA v.s. Interrupts

- DMA: hardware; Interrupt: software;
- DMA response: between two memory cycle; Interrupt response: between two instruction cycles.
- DMA does not change the CPU context (with the cycle stealing method).
- DMA uses interrupt in the end.
- DMA has **a higher priority** than interrupts (if response is slow, then high-speed peripheral may lose its data).
- DMA can only perform data transfer, but interrupts can handle exceptions.
- Interrupts is suitable for irregular（不定时） small-data transfer; DMA is suitable for multiple-set data transfer（成组数据交换）

Summary of Different I/O Techniques

Interaction with I/O device	Programmed I/O	Interrupt-driven I/O	Direct Memory Access
Waiting for the deivce	Software (CPU)	Hardware	Hardware
Transfer the device data to memory	Software (CPU)	Software	Hardware