中国科学院大学
University of Chinese Academy of Sciences

CS101

Systems Thinking

Seamless Transition-1:
The four principles of seamless transition

zxu@ict.ac.cn
zhangjialin@ict.ac.cn

Outline

- What is systems thinking?
- Three objectives of systems thinking
- Abstraction
- Modularization
- Seamless transition
  - The symphony of four principles
  - Yang's cycle principle
  - Postel's robustness principle
  - von Neumann's exhaustiveness principle
  - Amdahl's law
  - Landscape of computing systems

These slides acknowledge sources for additional data not cited in the textbook

5.1 The symphony of four principles

- 1-minute quiz
  - Q: Why can two students in San Jose and Shenzhen conduct a video talk online correctly? Please give a specific principle.
    - Why trillions of instructions can be automatically executed in a fraction of a second, across the globe, to produce correct computational results?

5.1 The symphony of four principles

- 1-minute quiz
  - Q: Why and how can two students in San Jose and Shenzhen conduct an online video talk correctly? Please give a specific principle.
    - Why trillions of instructions can be automatically executed in a fraction of a second, across the globe, to produce correct computational results?
  - A. The computers involved in the video talk execute their computational processes correctly and smoothly
  - More concretely, for each computational process involved, do a computational induction (similar to mathematic induction)
    - Ensure that the first step is correctly identified
    - Ensure that any identified step (i.e., any single step) is correctly executed
    - For each step just finishing execution, ensure that the correct next step is identified and the current step correctly transition to the next step
  - A step could be a program, a instruction, a gate, etc.

5.1 The symphony of four principles

- 1-minute quiz
  - Q: Why and how can two students in San Jose and Shenzhen conduct an online video talk correctly? Please give a specific principle.
    - Why trillions of instructions can be automatically executed in a fraction of a second, across the globe, to produce **correct** computational results?
  - A. The computers involved in the video talk execute their computational processes correctly and smoothly
  - More concretely, for each computational process involved, do a computational induction (similar to mathematic induction), to ensure **correctness**

    - Identify first step
    - Execute single step
    - Identify and transition to next step

- Yang's cycle principle
- Postel's robustness principle
- von Neumann's exhaustiveness principle

5.1 The symphony of four principles

- 1-minute quiz
  - Q: Why and how can two students in San Jose and Shenzhen conduct an online video talk correctly? Please give a specific principle.
    - Why trillions of instructions can be automatically executed **in a fraction of a second**, across the globe, to produce **correct** computational results?
  - A. The computers involved in the video talk execute their computational processes correctly and smoothly
  - More concretely, for each computational process involved, do a computational induction (similar to mathematic induction), to ensure **correctness**

    - Identify first step
    - Execute single step
    - Identify and transition to next step

    - Also need to consider **smoothly**

- Yang’s cycle principle
- Postel’s robustness principle
- von Neumann’s exhaustiveness principle
- Amdahl's law

5.2 Yang’s cycle principle

- In a multi-step computational process, how to ensure the seamless transition from one step to the next step?
- Yang’s cycle principle
  - A system executes a computational process in a sequence of cycles.
  - The system finishes one cycle and automatically returns to the beginning (of the next cycle),
  - So that different computational processes preserve their respective kinds.
    - Examples of different kinds, when step=instruction
      - MOV to register instruction, MOV to memory instruction
      - ADD instruction, INC instruction
      - CMP instruction, JL instruction

《太玄经·**周首**》𝌇:
阳气周神而反乎始，
物继其汇。

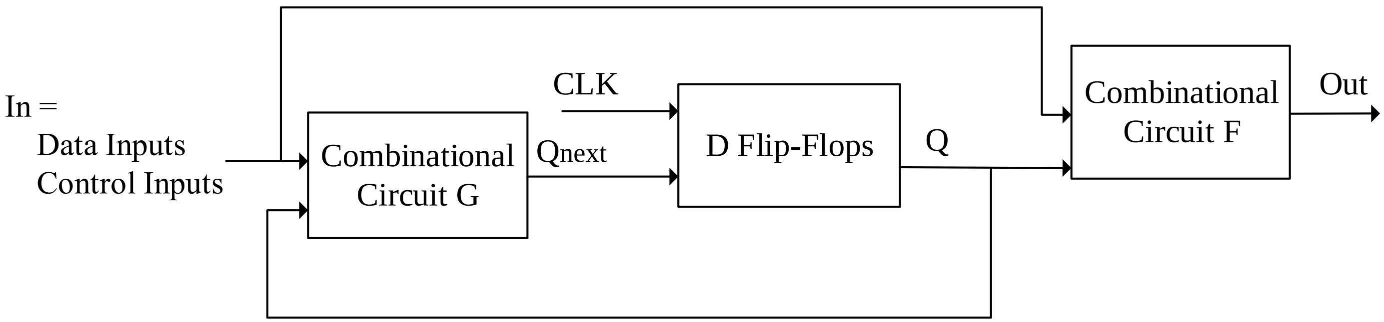
Head **Full Circle** 𝌇:
Yang qi comes full circle. Divinely, it returns to the beginning. Things go on to preserve their kinds.

扬雄，公元前2年
Yang Xiong, 2 BCE

English translation of 《太玄经·周首》adapted from
Nylan M. The Canon of Supreme Mystery by Yang Hsiung: A Translation with Commentary of the T'ai Hsuan Ching. SUNY Press, 1993.

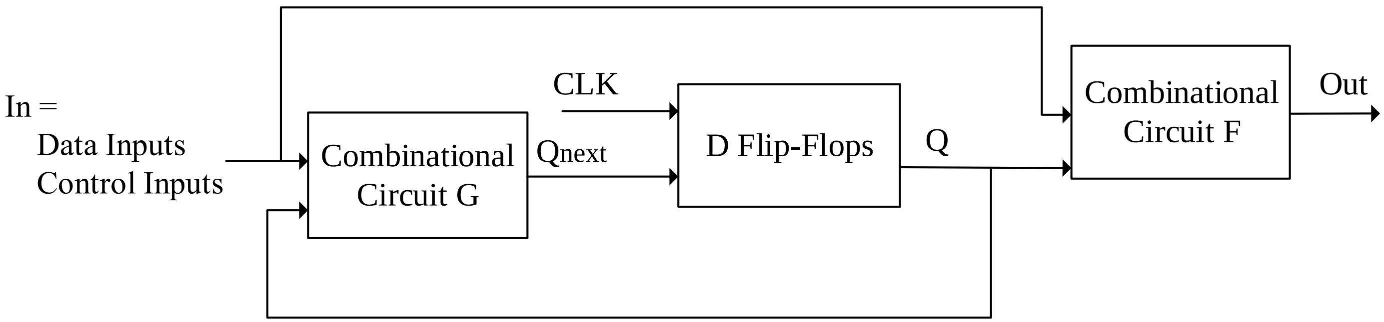
Crucial details

- Automatically return to the beginning of next cycle
  - Sequential circuit uses current state to generate next state
    - At step *k*, the system is in state Q, = output of the D flip-flops
    - Functionality of step *k*
      - Use Q and current input In to generate Q_{next} and current output Out
    - When step *k* finishes, i.e., when CLK switches to the next clock cycle
      - Q_{next} replaces Q to become the current state via the D flip-flops, and the system returns to the beginning of step *k* + 1



Crucial details

- Automatically return to the beginning of next cycle
  - Sequential circuit uses current state to generate next state
    - At step *k*, the system is in state Q, = output of the D flip-flops
    - Functionality of step *k*
      - Use Q and current input In to generate Q_{next} and current output Out
    - When step *k* finishes, i.e., when CLK switches to the next clock cycle
      - Q_{next} replaces Q to become the current state via the D flip-flops, and the system returns to the beginning of step *k* + 1



- Use the same cycle mechanism to support diversity
  - By utilizing different control signals (control inputs)

Cycles with different granularities

- The task of sending a WeChat message involves the executions of several programs, and consists of a sequence of program cycles
- Execution of a program cycle consists of the executions of a sequence of instruction cycles
- Execution of a instruction cycle consists of the executions of a sequence of clock cycles
- A 1-GHz processor has a clock cycle of 1 ns

- At each clock cycle, the processor performs a state transition of one or more sequential circuits

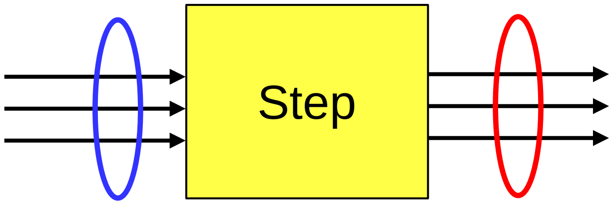
5.3 Postel's robustness principle

- Originally proposed by Jon Postel for the Internet
- Has become a systems principle
- When design, implement, and use a system, for every step,
  - Be tolerant of inputs and strict on outputs (宽进严出)
  - Be tolerant of inputs
    - System should still work when inputs deviate somewhat from "correct" values
  - Be strict on outputs
    - System should generate only "correct" outputs, not deviating from "correct" values
- Implication
  - Accumulation of errors, drifts, and distortions can often be avoided

TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

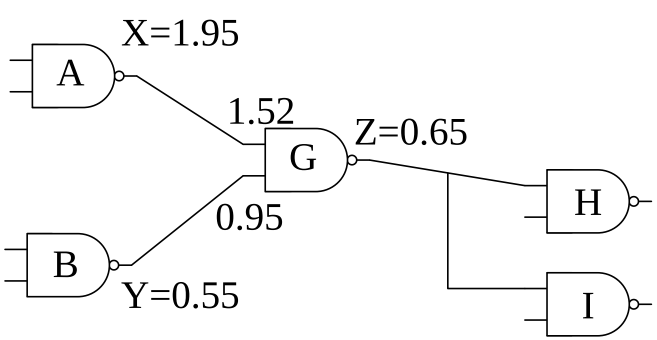
Jon Postel, 1980

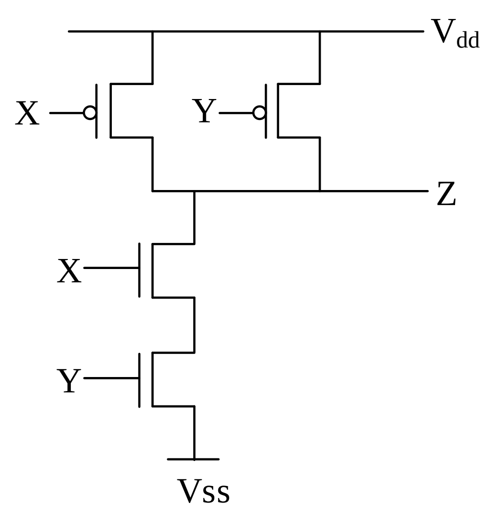
Be **strict in outputs**, and be **tolerant of inputs**



An example

- Consider gate G in the circuit of 5 NAND gates
  - It receives inputs from A, B, and outputs to H, I
    - All NAND gate have the same behavior and been implemented by a CMOS circuit
  - Naïve Design of the CMOS circuit without following Postel's robustness principle
    - There is not margin of gap near the threshold voltage Vth = 0.7 Volt
    - When A=HIGH=1.95 and B=LOW=0.55 Volt, Z should be HIGH>0.7 Volt
    - However, after B drifts +0.4 to reach 0.95 Volt, Z becomes LOW, an error



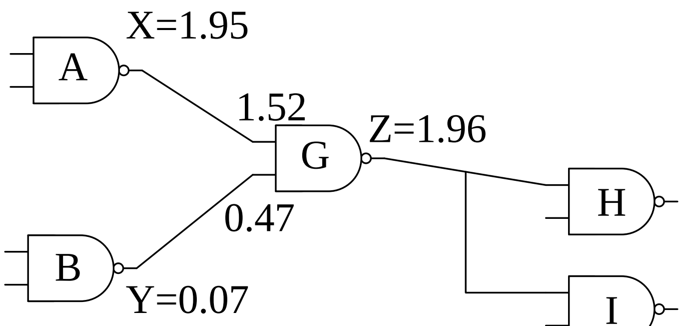


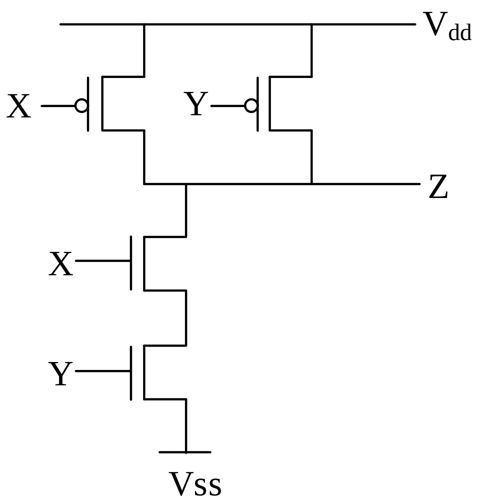
Vdd = 2 Volt
Vss = 0 Volt
Vth = 0.7 Volt

Naïve Design
Logic 1: > 0.7 Volt
Logic 0: < 0.7 Volt

An example

- Consider gate G in the circuit of 5 NAND gates
  - It receives inputs from A, B, and outputs to H, I
    - All NAND gate have the same behavior and been implemented by a CMOS circuit
- Better design of the CMOS circuit following Postel’s robustness principle
  - A minimal gap of 1 volt at input side and 1.8 volt at the output side
  - Note that the output of B cannot be 0.55 Volt. It has to be < 01 Volt
  - Let B=LOW=0.07<0.1 Volt. Even after a drifting value of +0.4 Volt, G still sees a LOW value, since B=0.47 Volt. Thus, Z is HIGH with Z > 1.9 Volt





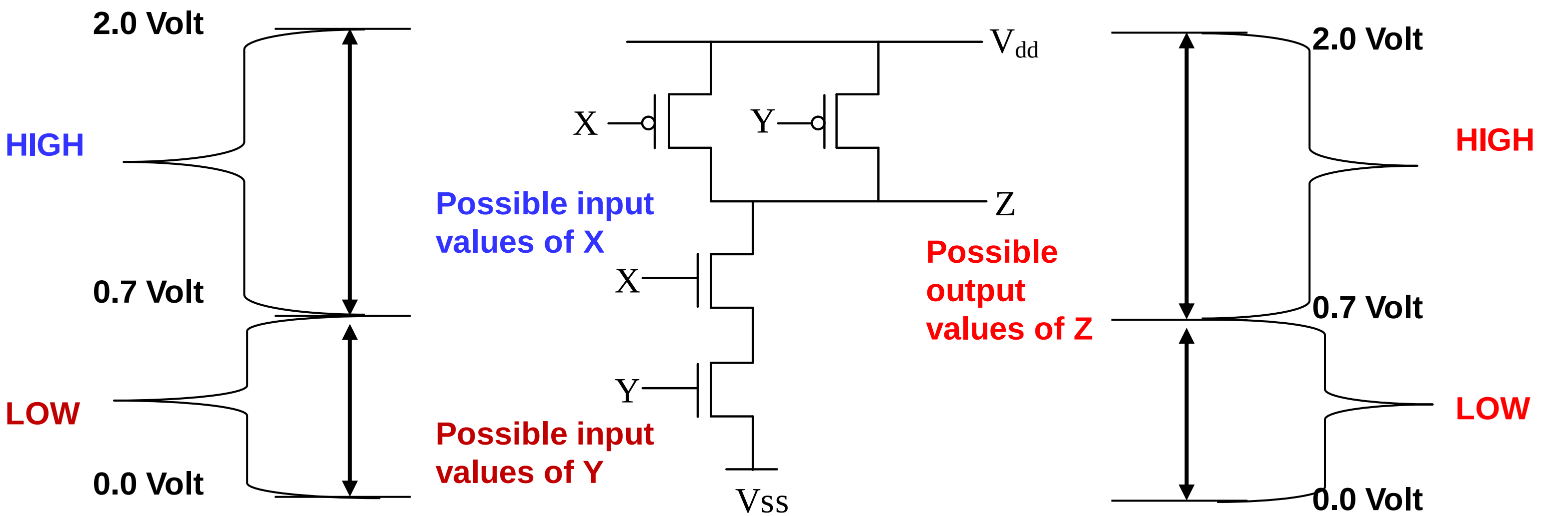
Better Design

For Input Voltages
Logic 1: > 1.5 Volt
Logic 0: < 0.5 Volt

For Output Voltages
Logic 1: > 1.9 Volt
Logic 0: < 0.1 Volt

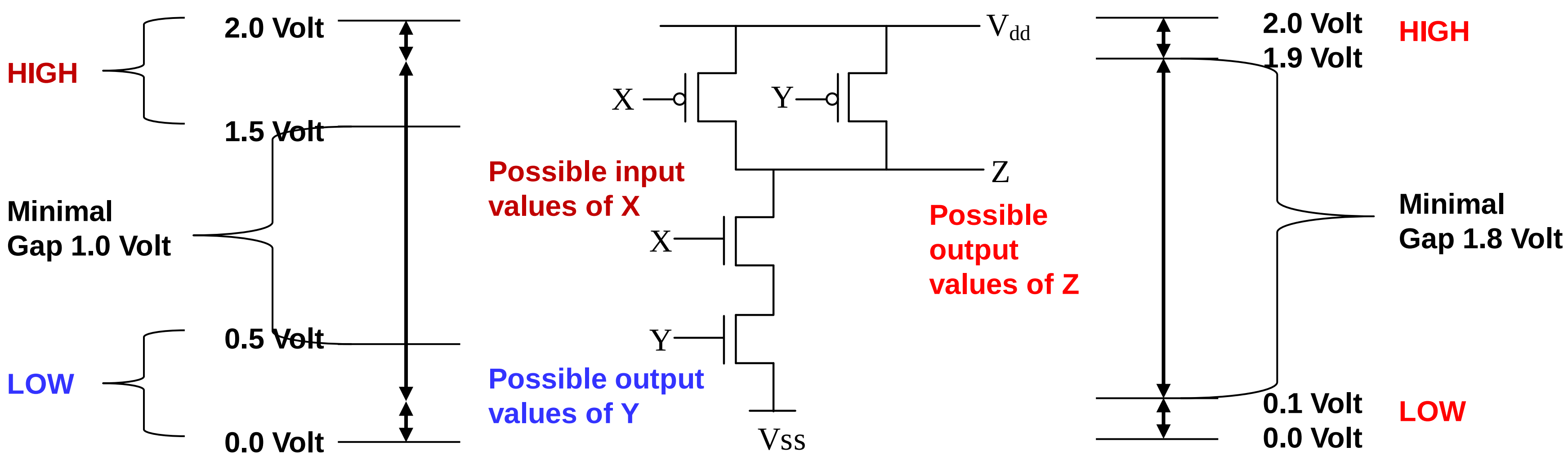
Summary of Naïve Design

- Assume X=HIGH and Y=LOW. Then Z should be HIGH
  - Could easily get the wrong result of Z = LOW
- Why?
- Treat inputs and outputs equally, and in a bad way
  - Both the input side and the output side
    - have **0 minimal gap** between HIGH and LOW
    - have **unsafe margins** of 0.7 Volt for LOW and 1.3 Volt for HIGH



Summary of Better Design

- Assume X=HIGH and Y=LOW. Then Z will be HIGH
- **Tolerance on inputs**
  - Input to a gate has a 0.5 Volt safe margin and a minimal gap of 1 Volt
    - Compared to Naïve Design: 0.7 and 1.3 unsafe margins and 0 minimal gap
- **Strictness on outputs**
  - Output from a gate has a 0.1 Volt safe margin and a minimal gap of 1.8 Volt
    - Compared to Naïve Design: 0.7 and 1.3 unsafe margins and 0 minimal gap



5.4 von Neumann's exhaustiveness principle

- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
- In the quote, two terms have specific meanings
  - *Operation* = Problem-solving Task
    - E.g., solving a non-linear partial differential equation
  - *Device* = Computer
    - An automatic computing system

> The instructions which govern this *operation* must be given to the *device* in absolutely exhaustive detail. …
>
> Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention.
>
> John von Neumann, 1945

5.4 von Neumann's exhaustiveness principle

- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
- 1-minute quiz
  - Q: How to cover "absolutely exhaustive detail"?

> The instructions which govern this operation must be given to the device in absolutely exhaustive detail. …
>
> Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention.
>
> John von Neumann, 1945

5.4 von Neumann's exhaustiveness principle

- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
- 1-minute quiz
  - Q: How to achieve "absolutely exhaustive detail"?
  - The challenge:
    - how to use finite instructions to achieve "absolutely exhaustive detail"?
  - List the types of instructions, and give an example for each type
    - Program code, e.g., hide.go

> The instructions which govern this operation must be given to the device in absolutely exhaustive detail. …
>
> Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention.
>
> John von Neumann, 1945

5.4 von Neumann's exhaustiveness principle

- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
- 1-minute quiz
  - Q: How to achieve "in absolutely exhaustive detail"? List the types of instructions, and give an example for each type
  - A: Instructions here mean not only a computer's instruction set, but include the following
    - Program code, e.g., hide.go
    - Input data, e.g., Autumn.bmp
    - Library of functions, e.g., fmt.go
    - Context information, e.g., hide.go is /cs101/Prj2 in my Linux laptop
  - A: Answers to more fundamental questions
    - Where and what is the first instruction, when the computer power is turned on?
    - How to determine the next instruction to execute?
    - What types of exceptions are there, to normal execution of programs?

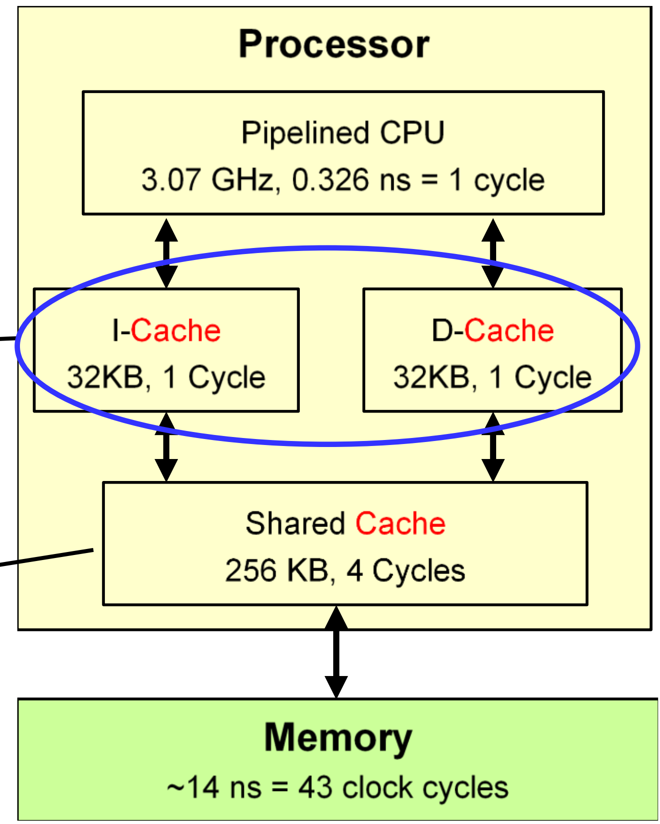
> The instructions which govern this operation must be given to the device in absolutely exhaustive detail. …
>
> Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention.
>
> John von Neumann, 1945

The first instruction to execute
when the computer power turns on

- Example with an x86 processor
  - Where: the first instruction to execute is at memory address 0xFFFFFFF0
  - What: a jump instruction, e.g., JUMP 000F0000
    - Address 000F0000 contains the entry instruction for the BIOS code
- Why?
  - Computer starts by executing the BIOS firmware code
    - To initialize the computer and to load the operating system
  - Using a jump instruction upfront increases flexibility
    - E.g., if we want the computer to start by executing another firmware code BIOS-2 at entry address 000FA000,then change
    - Address 0xFFFFFFF0 to hold JUMP 000FA000

Three ways to determine the next instruction to execute

- The earliest method is linear sequencing in Harvard Mark I computer, the *Automatic Sequence Controlled Calculator*
  - Instructions are linearly sequenced
    - There is no jump. Next instruction is located right after current instruction on the instruction tape
  - Storing data and code separately (This is called **Harvard architecture**)
    - Still widely used in the cache units of modern computers. A processor has separate instruction cache and data cache
  - In contrast, the **Princeton architecture** uses a single cache or memory to store both data and instructions
- Modern computers use both



Three ways to determine the next instruction to execute

- The earliest method is linear sequencing in Harvard Mark I computer
- The ENIAC method
  - Every instruction holds the address of the next instruction
    - Used by the revised version of the ENIAC computer

Opcode and operands of current instruction	Address of next instruction

- Modern computers mostly use the PC mechanism: the address of the next instruction to execute is stored in the program counter (PC)

Deal with exceptions to normal execution

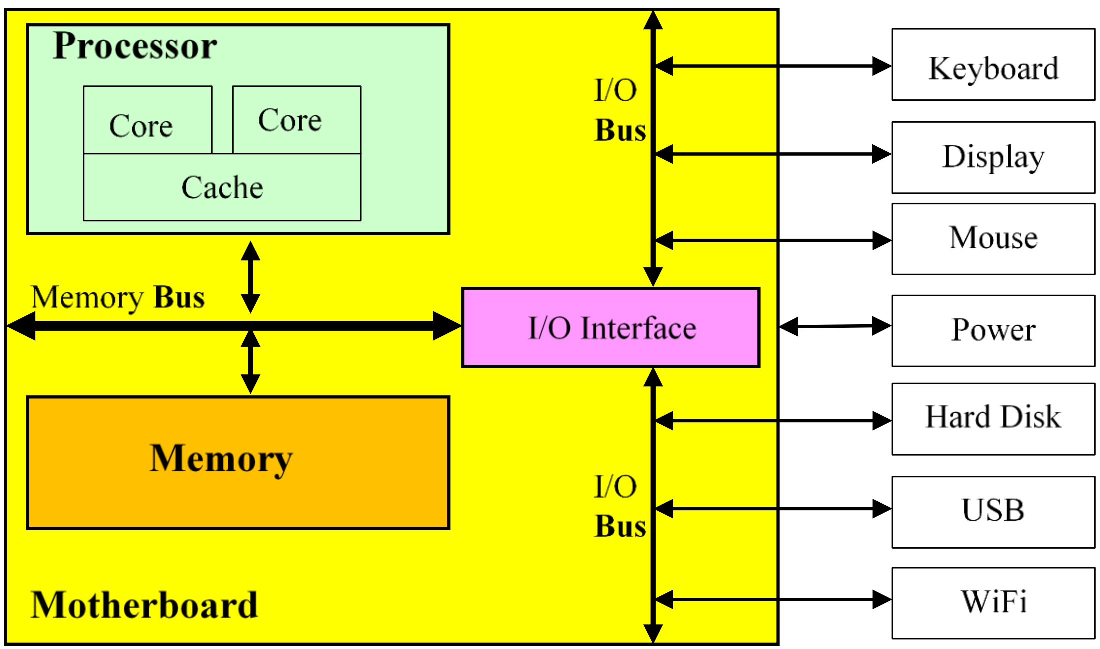
- We have seen exceptions in programming, e.g.,
  - in the Text Hider project, the statement

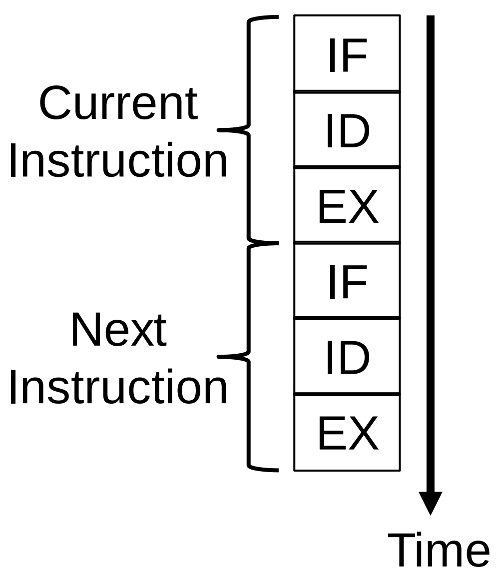
```
p, _ := ioutil.ReadFile("./Autumn.bmp") should really be

p, error := ioutil.ReadFile("./Autumn.bmp")
if error != nil {…// put exception-handling code here}
… // no error; continue normal execution
```

Three types of exceptions are supported by computer hardware

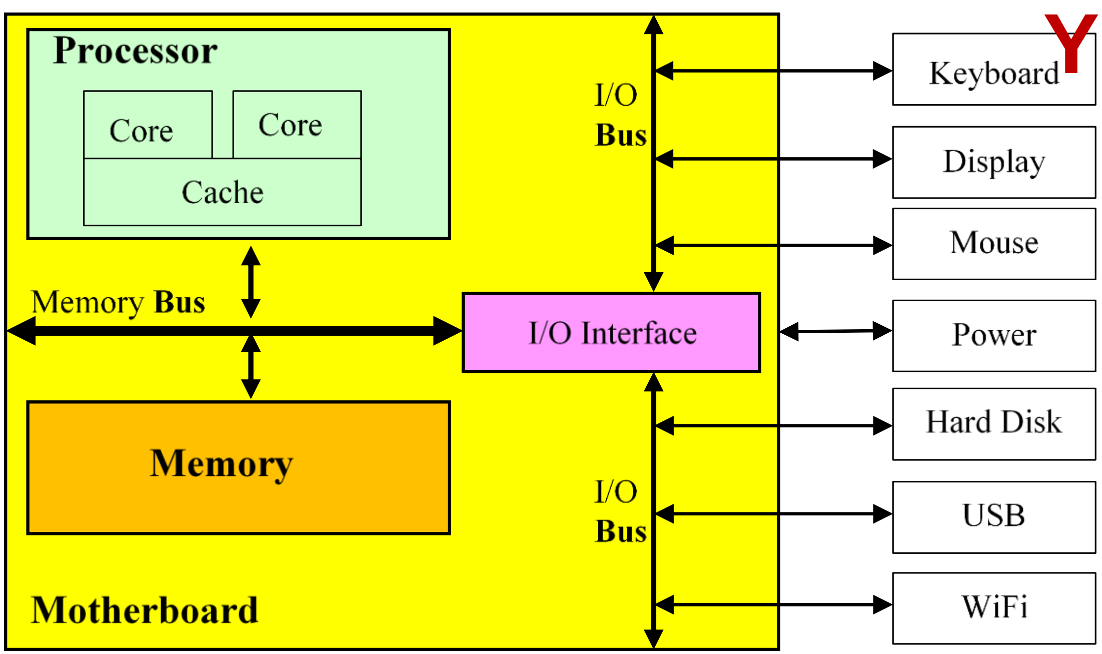
- In normal execution (without exception), the current instruction finishes and continue to execute the next instruction

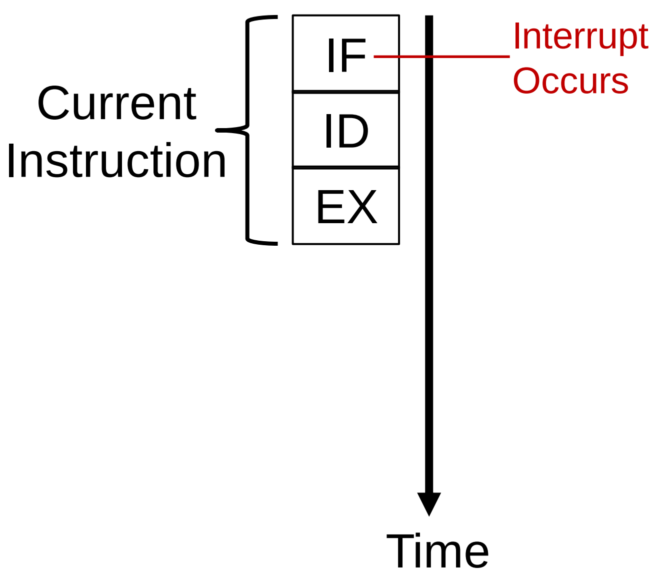




5.4.1 Interrupt handling

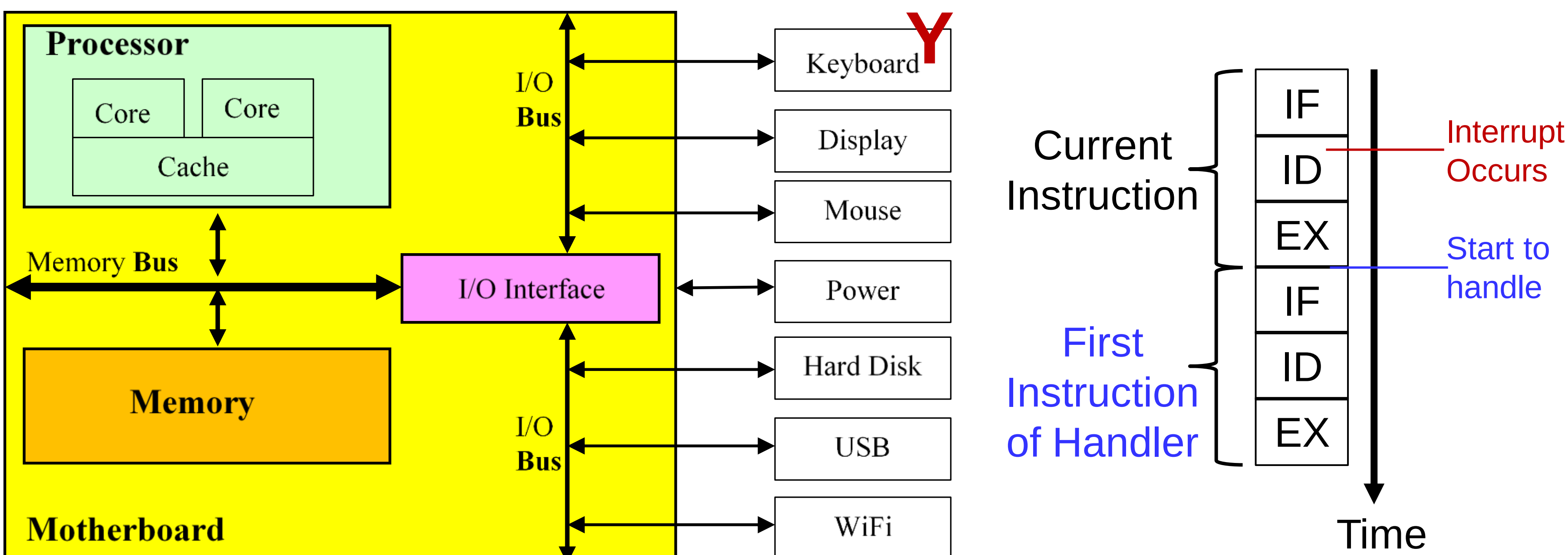
- When an interrupt occurs, e.g.,
  - When the user punches key ‘Y’ on the keyboard while the processor is executing the instruction fetch stage
- What should the processor do?
  - Should it immediately take an exception-handling action?
  - Should it finishes the current instruction first?





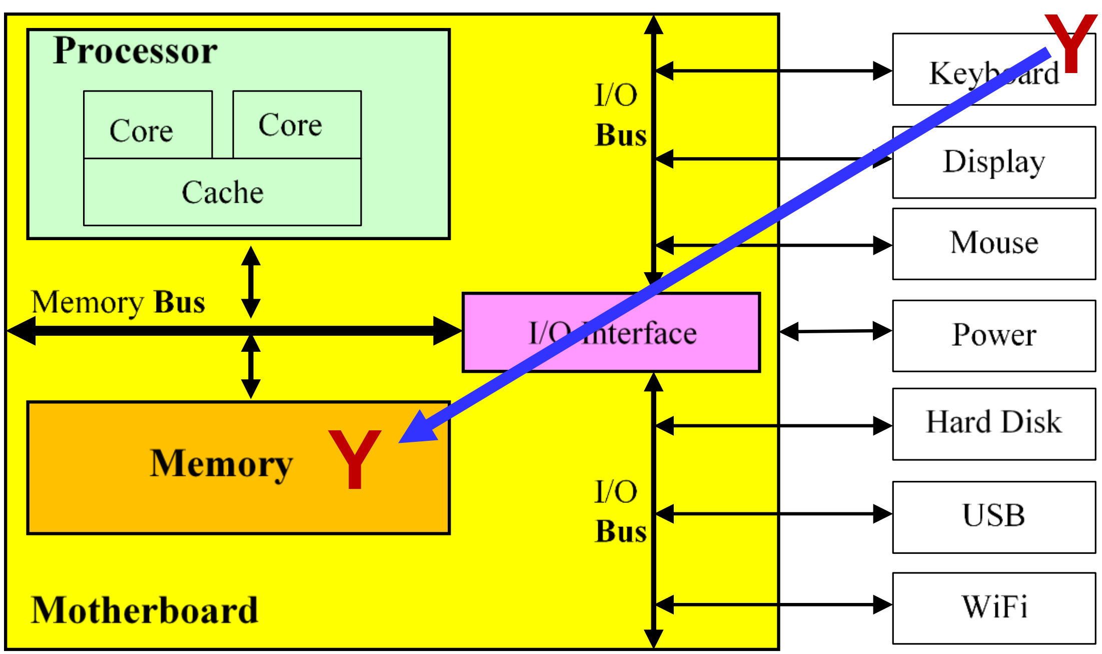
Interrupt handling

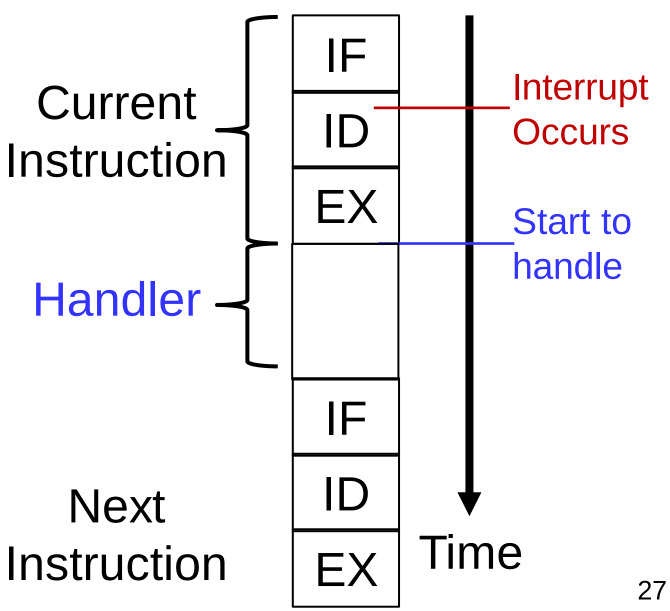
- When an interrupt occurs, e.g.,
  - When the user punches key ‘Y’ on the keyboard while the processor is executing the instruction decode stage
- The processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt



Interrupt handling

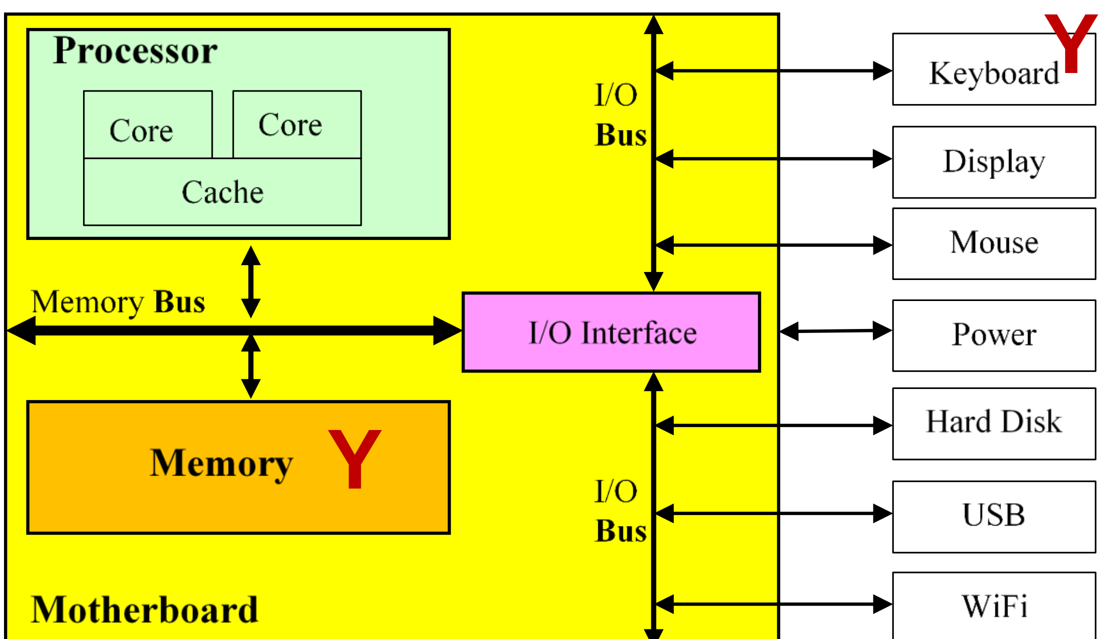
- When an interrupt occurs, e.g.,
  - When the user punches key 'Y' on the keyboard while the processor is executing the instruction decode stage
- The processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - such as **coping the punched key value to memory**

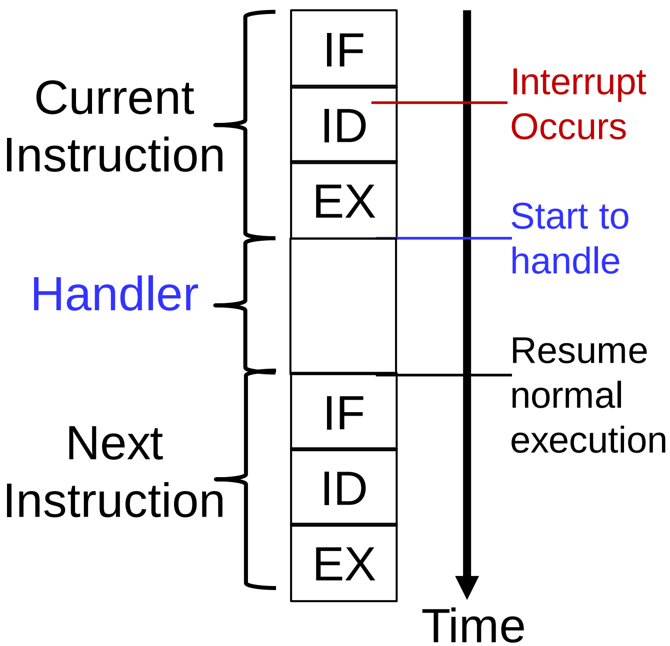




Interrupt handling

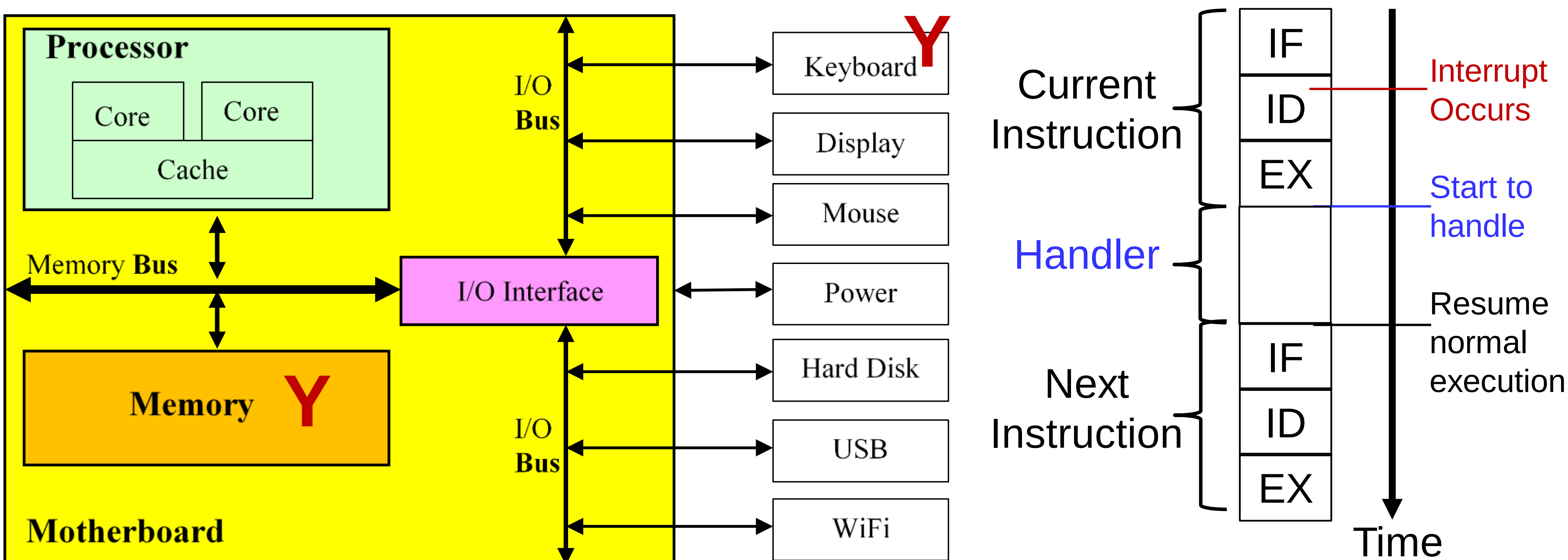
- When an interrupt occurs, the processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - such as **coping the punched key value to memory**
- Then, the processor resumes normal execution
  - by executing the original next instruction





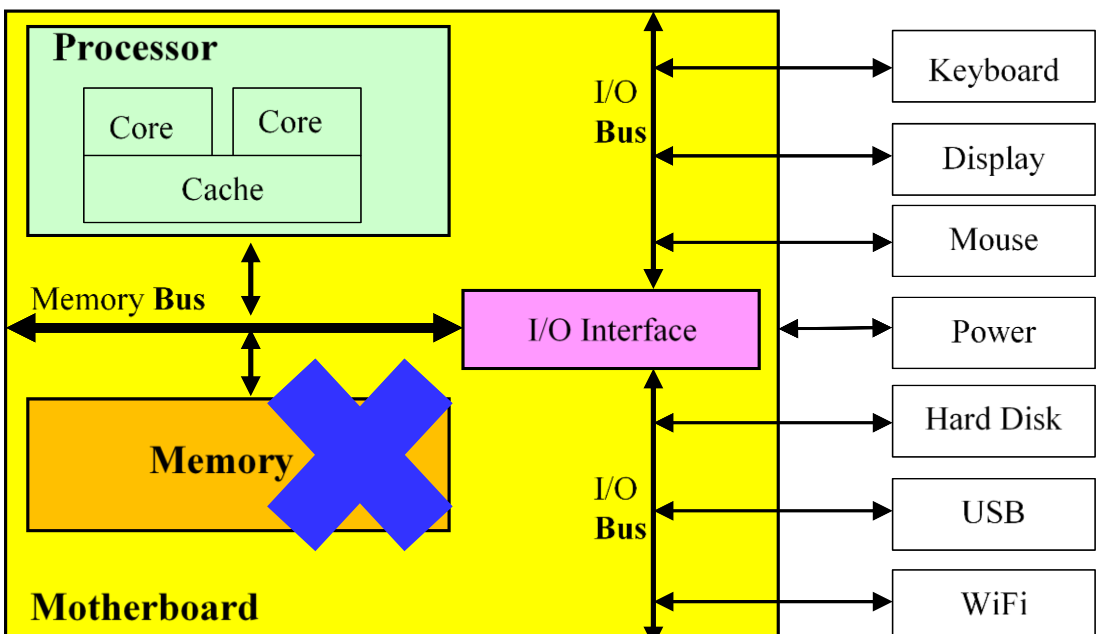
Interrupt handling

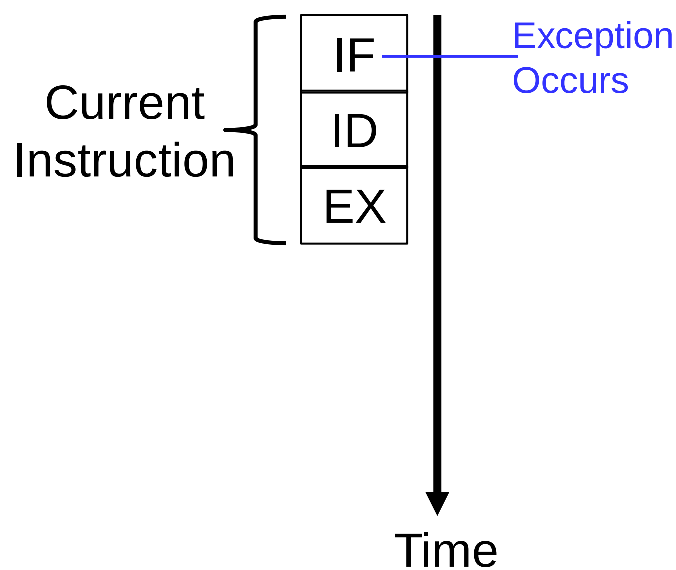
- When an interrupt occurs, the processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - such as **coping the punched key value to memory**
- Then, the processor resumes executing the next instruction
  - Q: how does the processor know the address of the next instruction?



5.4.2 Hardware error handling

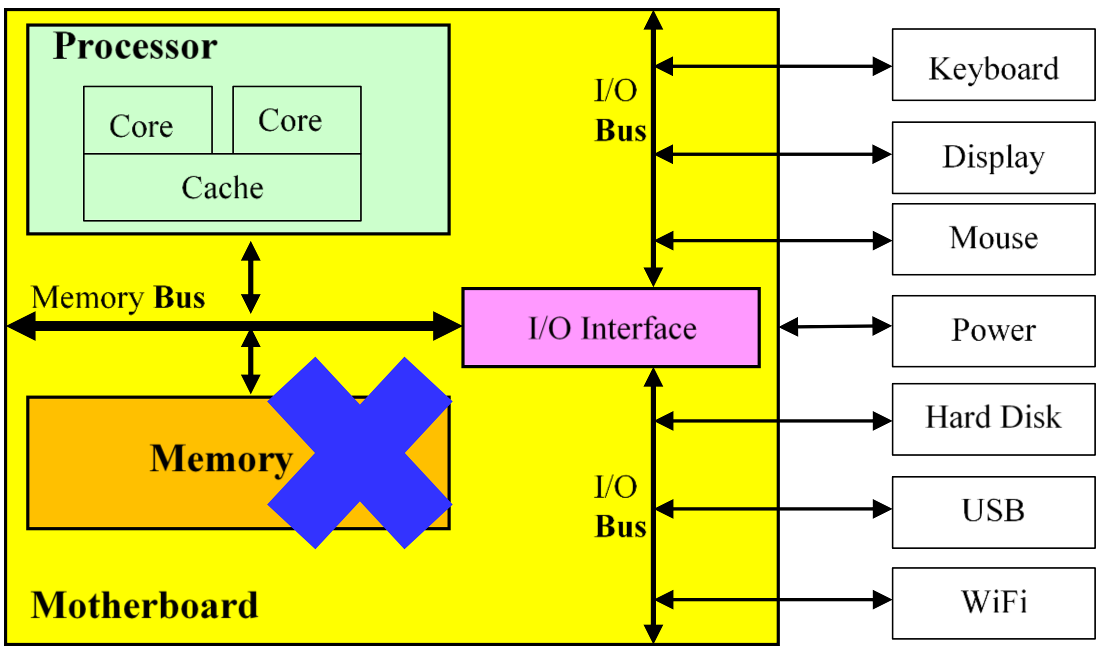
- When a hardware error occurs, e.g.,
  - When the memory becomes faulty and generates a hardware error exception
- What should the processor do?
  - Should it immediately take an exception-handling action?
  - Should it finishes the current instruction first?

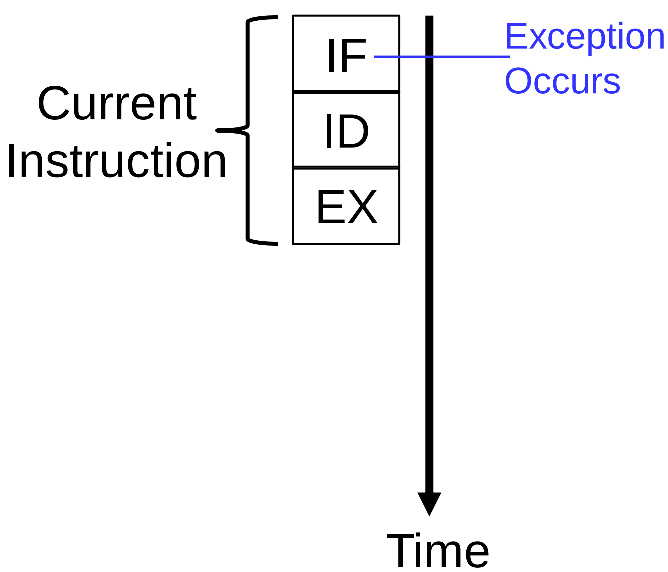




Hardware error

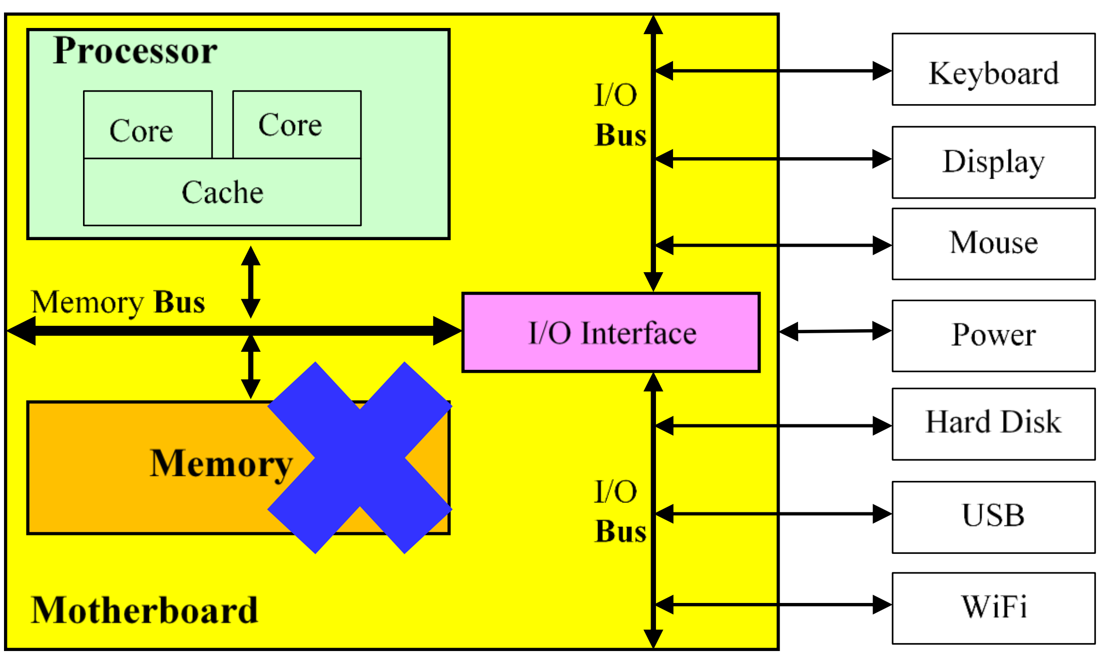
- When the memory becomes faulty and generates a hardware error exception
  - Should the processor finishes the current instruction first?
  - No, because the IF stage cannot be finished
    - The instruction cannot be fetched from memory

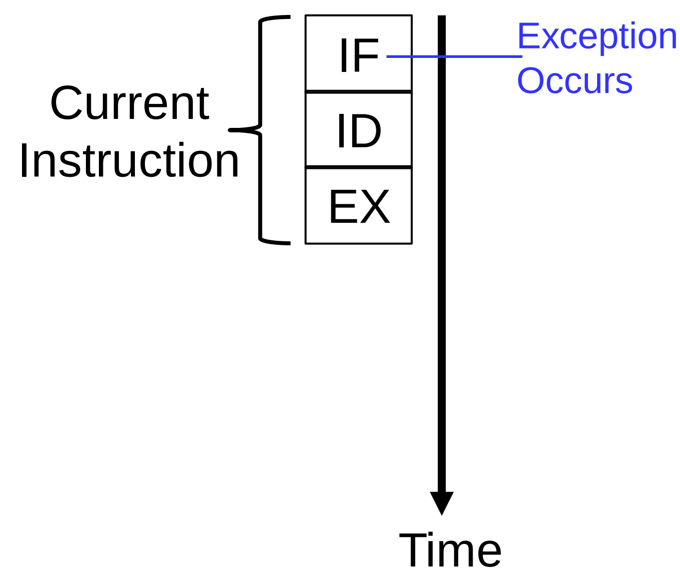




Hardware error

- When the memory becomes faulty and generates a hardware error exception
  - Should the processor immediately take an exception-handling action?
  - Yes, it executes an exception-handling sequence of steps without depending on the memory
  - The system returns to some well-defined crash state





5.4.3 Machine check

- This is the "all other" exception, for exhaustiveness
- Typically a unrecoverable hardware error
- Example
  - When executing an exception-handling sequence of steps for the memory fault, the sequence experiences another error
  - The system generates minimal diagnostic information and crashes