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

CS101

# 系统思维-3

## 无缝衔接

**zxu@ict.ac.cn**

# 提纲



- 系统思维概述
- 抽象化
- 模块化
- 无缝衔接
  - 扬雄周期原理
    - 指令周期与指令流水线
  - 波斯特尔鲁棒性原理（宽进严出原理）
  - 冯诺依曼穷举原理
  - 阿姆达尔定律

程序
进程
**指令**
**冯氏结构**
**指令流水线**
**时序电路**
**组合电路**

**以一耦万：**

**由模块化抽象表达的计算过程如何能够流畅运行**

课件中可能包含素材引用，特此致谢！

# 课堂小测验

- 姓名：　　　　　　　　　　　学号：
- 如果两个数字电路有相同真值表，称它们等价。等价的数字电路算一个。请指出下述哪一个断言是错误的（）

A. 一个输入一个输出的组合电路有4个

*B.* $N$个输入一个输出的组合电路有$2^{2^N}$个

C. 一个输入一个输出的时序电路有4个

D. 一个输入一个输出的时序电路有无穷多个

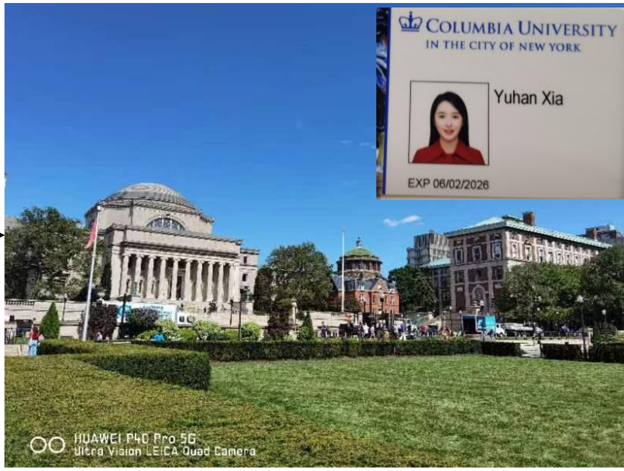
# 示例：无缝衔接的计算过程

- 教师与同学微信视频，讨论推荐信事宜
- 为什么能够完成协同计算任务（传递一个视频笑容）？
  - 相隔万里，万亿次运算，一秒之内完成
- 原理上是如何做到的？

← 15000公里 →

北京

纽约



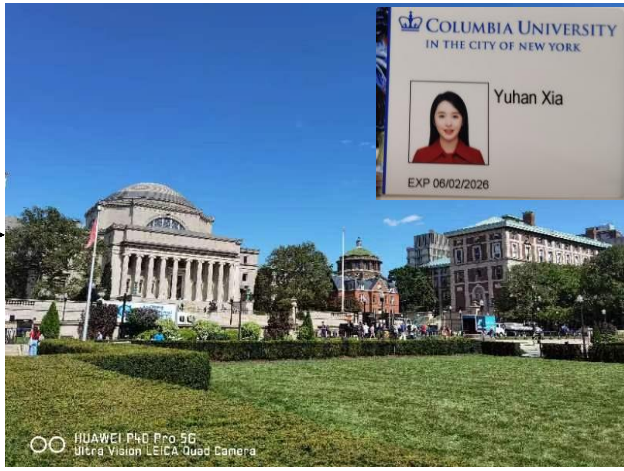
# 示例：无缝衔接的计算过程

- 教师与同学微信视频，讨论推荐信事宜
- 为什么能够完成协同计算任务（传递一个视频笑容）？
  - 相隔万里，万亿次运算，一秒之内完成
- 原理上是如何做到的？
  - 先不管速度，正确性是如何保证的？（功能实现）
  - 速度是如何保证的？（性能实现）
- 无缝衔接的计算过程

15000公里

北京

纽约



# 1. 无缝衔接

- 无缝衔接是自动执行的进阶
  - 计算过程在全系统中流畅地正确运行，避免缝隙和瓶颈
  - 相邻的步骤之间的形式与内容（格式与语义）要无缝衔接
    - 信息和计算无障碍地从一个模块、步骤过渡到下一个模块、步骤
  - 做不到完全流畅，至少也要控制瓶颈，使系统满足用户体验需求

# 无缝衔接

- 无缝衔接是高阶自动执行
  - 计算过程在全系统中流畅地正确运行，避免缝隙和瓶颈
  - 相邻的步骤之间的形式与内容（格式与语义）要无缝衔接
    - 信息和计算无障碍地从一个模块、步骤过渡到下一个模块、步骤
  - 做不到完全流畅，至少也要控制瓶颈，使系统满足用户体验需求
- 确保计算过程正确性的计算归纳法
    - 正确地确定**第一步**
    - 针对每个单一步骤，保证**当前步**骤正确执行
    - 每一步骤执行完毕，正确地确定并衔接到**下一步**骤

# 无缝衔接

- 无缝衔接是高阶自动执行
  - 计算过程在全系统中流畅地正确运行，避免缝隙和瓶颈
  - 相邻的步骤之间的形式与内容（格式与语义）要无缝衔接
    - 信息和计算无障碍地从一个模块、步骤过渡到下一个模块、步骤
  - 做不到完全流畅，至少也要控制瓶颈，使系统满足用户体验需求
- 确保正确性的计算过程归纳法
    - 正确地确定第一步
    - 针对每个单一步骤，保证该步骤正确执行
    - 每一步骤执行完毕，正确地确定并衔接到下一步骤
- 无缝衔接的四个原理
  - 扬雄周期原理 Yang’s cycle principle
  - 波斯特尔鲁棒原理 Postel’s robustness principle
  - 冯诺依曼穷举原理 von Neumann’s exhaustiveness principle
  - 阿姆达尔定律 Amdahl‘s law

# 无缝衔接

- 无缝衔接是高阶自动执行
  - 计算过程在全系统中流畅地正确运行，避免缝隙和瓶颈
  - 相邻的步骤之间的形式与内容（格式与语义）要无缝衔接
    - 信息和计算无障碍地从一个模块、步骤过渡到下一个模块、步骤
  - 做不到完全流畅，至少也要控制瓶颈，使系统满足用户体验需求

- 计算归纳法
  - 确定第一步
  - 保证当前步骤正确执行
  - 正确地衔接到下一步骤

**扬雄周期原理**

**波斯特尔鲁棒原理**

**冯诺依曼穷举原理**

# 无缝衔接

- 无缝衔接是高阶自动执行
  - 计算过程在全系统中流畅地正确运行，避免缝隙和瓶颈
  - 相邻的步骤之间的形式与内容（格式与语义）要无缝衔接
    - 信息和计算无障碍地从一个模块、步骤过渡到下一个模块、步骤
  - 做不到完全流畅，至少也要控制瓶颈，使系统满足用户体验需求

- 计算归纳法
  - 确定第一步
  - 保证当前步骤正确执行
  - 正确地衔接到下一步骤

- 性能的无缝衔接

**扬雄周期原理**

**波斯特尔鲁棒原理**

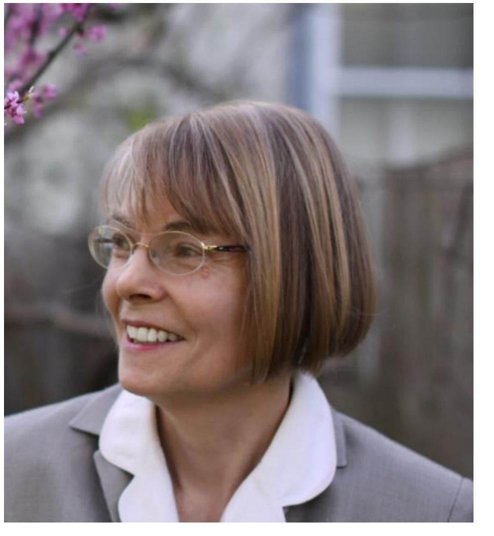
**冯诺依曼穷举原理**

**阿姆达尔定律**

# 2. 扬雄周期原理

## 优于尼采的永恒轮回：die ewige Wiederkunft

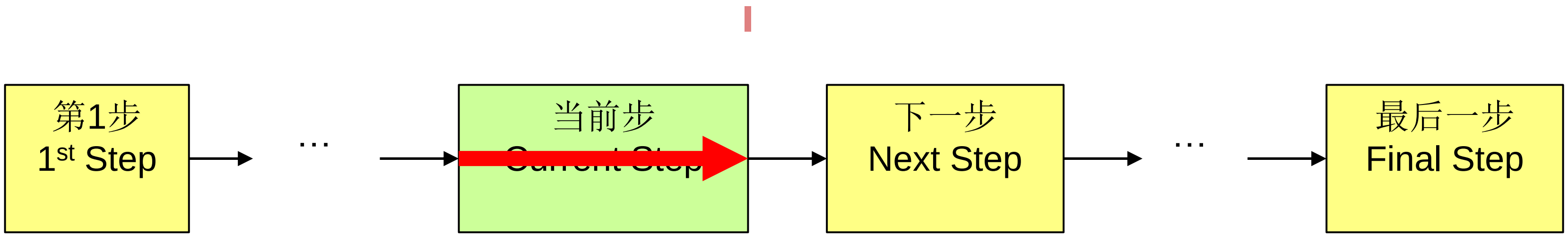
- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“𝌇 阳气周神而反乎始，物继其汇”
  - 宋代司马光诠释道：“岁功既毕，神化既周”



  - Head **Full Circle** 𝌇  
    Yang qi comes full circle.  
    Divinely, it returns to the beginning.  
    Things go on to preserve their kinds.
  - UC-Berkeley戴梅可教授的英文版译注  
    **The Canon of Supreme Mystery** by Yang Hsiung

# 扬雄周期原理
## 优于尼采的永恒轮回：die ewige Wiederkunft

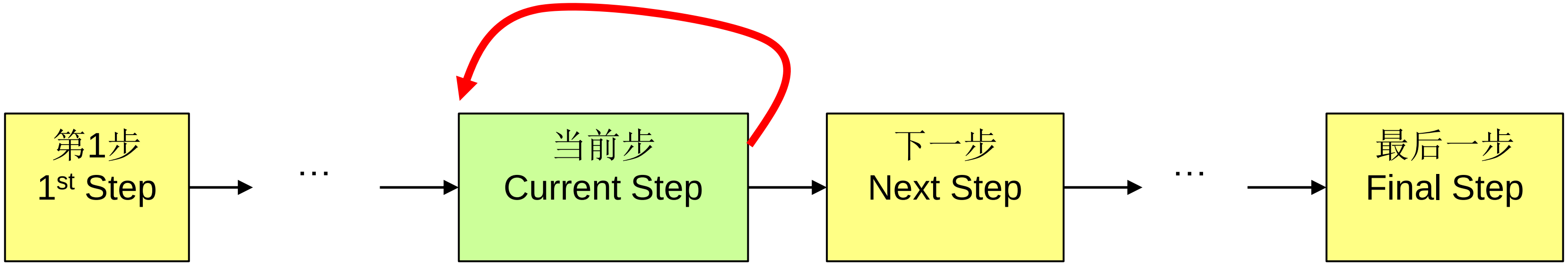
- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“阳气周神而反乎始，物继其汇”
    - 宋代司马光诠释道：“岁功既毕，神化既周”
  - 三个要点
    - 岁功既毕：当前步骤执行完毕
    - 周而复始：哪个步骤的开始？
    - 物继其类



# 扬雄周期原理

## 优于尼采的永恒轮回：die ewige Wiederkunft

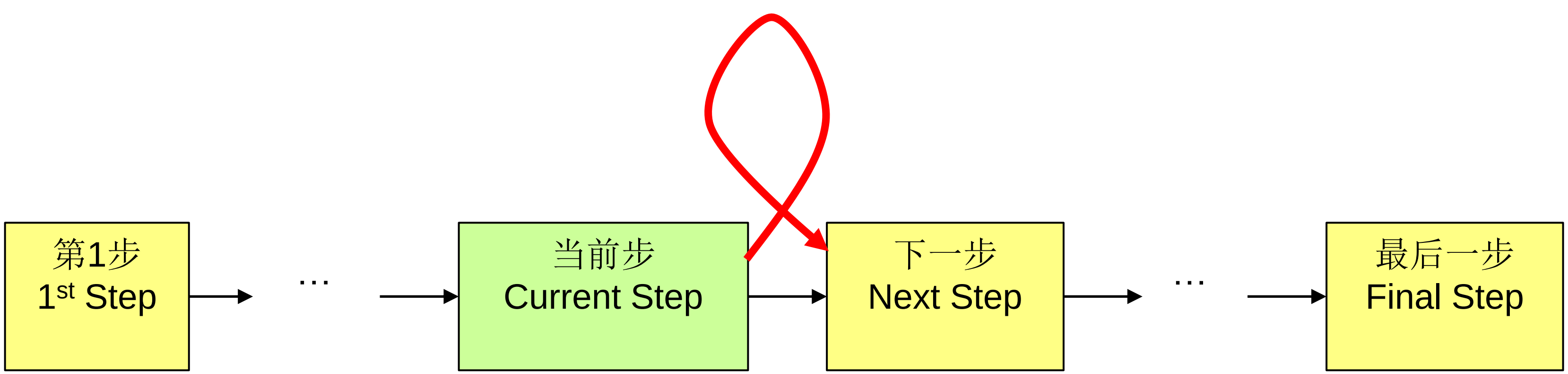
- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“阳气周神而反乎始，物继其汇”
    - 宋代司马光诠释道：“岁功既毕，神化既周”
  - 三个要点
    - 岁功既毕：当前步骤执行完毕
    - 周而复始：哪个步骤的开始？
    - 物继其类



# 扬雄周期原理

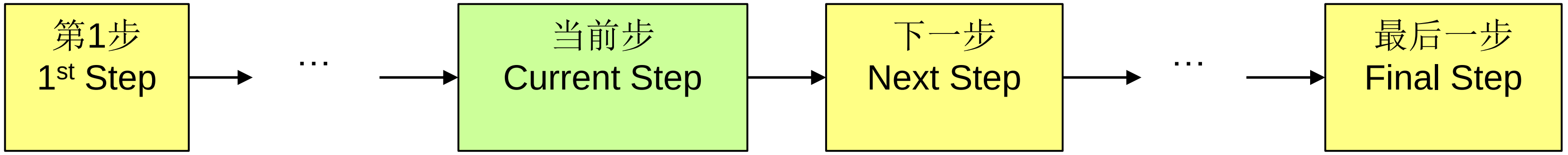
## 优于尼采的永恒轮回：die ewige Wiederkunft

- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“阳气周神而反乎始，物继其汇”
    - 宋代司马光诠释道：“岁功既毕，神化既周”
  - 三个要点
    - 岁功既毕：当前步骤执行完毕
    - 周而复始（哪个步骤的开始？下一步骤的开始）



# 扬雄周期原理
## 优于尼采的永恒轮回：die ewige Wiederkunft

- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“阳气周神而反乎始，物继其汇”
    - 宋代司马光诠释道：“岁功既毕，神化既周”
  - 三个要点
    - 岁功既毕：当前步骤执行完毕
    - 周而复始（哪个步骤的开始？下一步骤的开始）
    - 物继其类（每个步骤呈现其特色，周期原理支持多样性）
      - 当前指令的功能；下一条指令在哪里
      - 指令周期支持各种指令：加法、乘法、存数、跳转等等



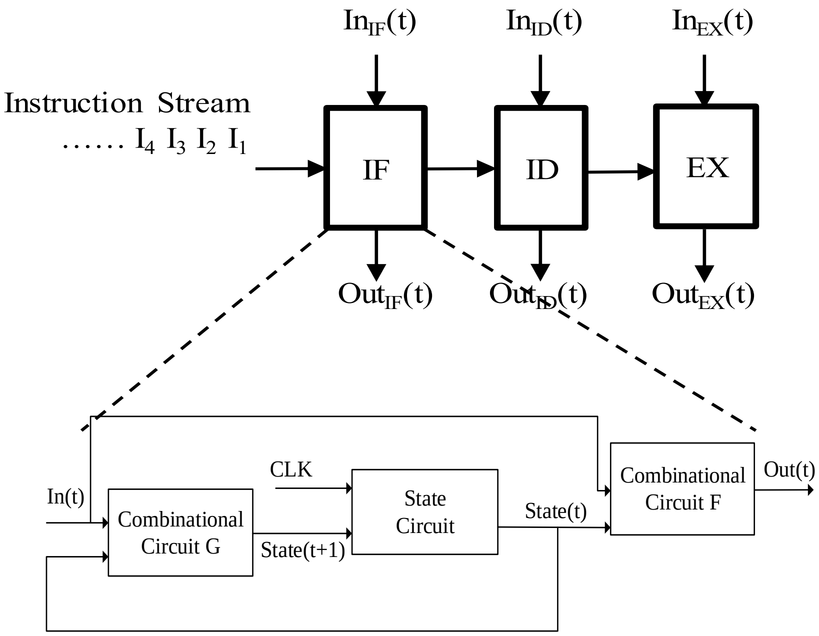
# 扬雄周期原理
## 优于尼采的永恒轮回：die ewige Wiederkunft

- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“阳气周神而反乎始，物继其汇”
    - 宋代司马光诠释道：“岁功既毕，神化既周”
- 执行完一个XX周期，周而复始执行下一个XX周期
    - 计算过程由**程序**周期组合而成
    - 程序周期由**指令**周期组合而成
    - 指令周期由**时钟**周期组合而成

# 扬雄周期原理
## 优于尼采的永恒轮回：die ewige Wiederkunft

- 汉代扬雄所著的《太玄经》
  - 《太玄经•周首》：“阳气周神而反乎始，物继其汇”
    - 宋代司马光诠释道：“岁功既毕，神化既周”
- 执行完一个XX周期，周而复始执行下一个XX周期
    - 计算过程由**程序**周期组合而成
    - 程序周期由**指令**周期组合而成
    - 指令周期由**时钟**周期组合而成
- 一个时钟周期的操作对应于一个自动机变换
  - 等同于时序电路的一个时钟周期的操作

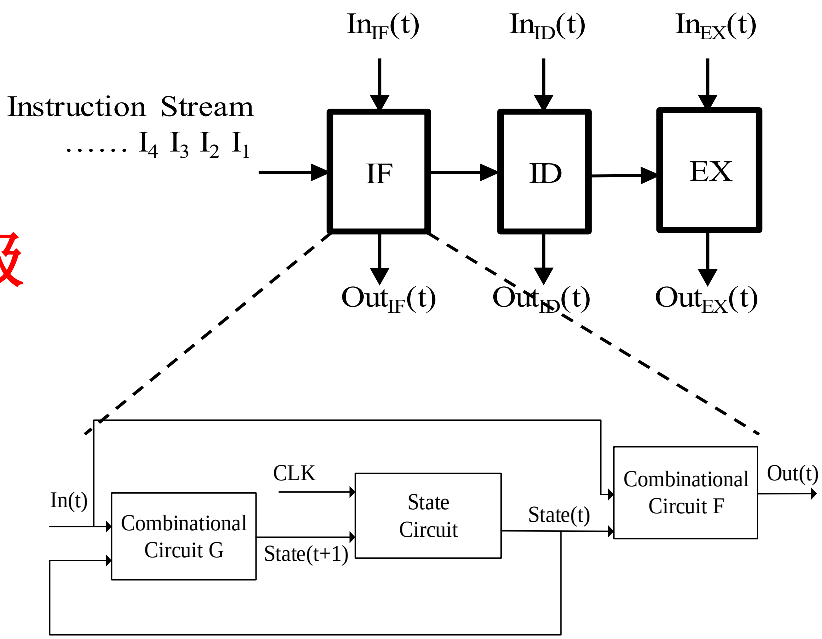
# 3. 指令周期与指令流水线




- 处理器的核心是指令流水线
  - 指令流水线的每一级
    是一个时序电路
- 例子：三级指令流水线
  - 一个指令周期包含三个步骤（三级）
    取指、译码、执行
- 三级指令流水线
  执行指令MOV 0, R1
  - Instruction Fetch (IF) stage：IR ← M[PC]　　取指（IF）
    - 将内存单元M[PC]的指令取到指令寄存器IR
  - Instruction Decode (ID): Control Signals = Decode(IR)　　译码（ID）
    - 从指令产生控制信号
  - Instruction Execute (EX): R1 ← 0; PC ← PC+1　　执行（EX）
    - 根据控制信号执行指令操作，并将程序计数器增量

# 指令周期与指令流水线

- 处理器的核心是指令流水线
  - 指令流水线的每一级
    是一个时序电路
- **真实处理器的指令流水线有5~31级**




- 例子：三级指令流水线
  执行指令MOV 0, R1
  - Instruction Fetch (IF) stage：IR ← M[PC]　　取指（IF）
    - 将内存单元M[PC]的指令取到指令寄存器IR
  - Instruction Decode (ID): Control Signals = Decode(IR)　　译码（ID）
    - 从指令产生控制信号
  - Instruction Execute (EX): R1 ← 0; PC ← PC+1　　执行（EX）
    - 根据控制信号执行指令操作，并将程序计数器增量

# 3.1 设计斐波那契计算机的指令集

- 继续英文教科书2.3节内容

```
fib[0] = 0                      MOV 0, R1
                                MOV R1, M[R0]              //R0=12 initially
fib[1] = 1                      MOV 1, R1
                                MOV R1, M[R0+8]
for i := 2; i < 51; i++ {       MOV 2, R2                  // i:=2
   fib[i] = fib[i-1] + fib[i-2] MOV 0, R1                  // label Loop
                                ADD M[R0+R2*8-16], R1
                                ADD M[R0+R2*8-8], R1
                                MOV R1, M[R0+R2*8-0]
                                INC R2                     // i++
                                CMP 51, R2                 // i < 51?
}                               JL Loop                    // if Yes, goto Loop
```

- Design an instruction set for FC
  - Any instruction consists of an opcode and one or more operands
    - E.g., **opcode** **operand** **operand**  每条指令包含一个操作码和若干操作数
  - In mnemonics, e.g., **MOV 0, R1**  汇编语言记号表达
  - In binary, e.g., **000 000000 01**  二进制表达

# 指令集设计过程

- **步骤0**：给定汇编语言代码
- **步骤1**：**确定（可见的）存储器与寄存器**
  - **FLAGS**: CPU status register **状态寄存器**
    - Holding status value of instruction execution, such as if the result is overflow, zero, less than, etc.
    - Only “less than” is used in this example
  - **PC**: program counter **程序计数器**
    - Holding the address of the next instruction to be executed
  - **R0, R1, and R2**: general purpose registers 三个**通用寄存器**
    - Holding operands of instructions

```
fib[0] = 0                        MOV 0, R1
                                  MOV R1, M[R0]
fib[1] = 1                        MOV 1, R1
                                  MOV R1, M[R0+8]
for i := 2; i < 51; i++ {         MOV 2, R2
  fib[i] = fib[i-1] + fib[i-2]    MOV 0, R1      // Loop
                                  ADD M[R0+R2*8-16], R1
                                  ADD M[R0+R2*8-8], R1
                                  MOV R1, M[R0+R2*8-0]
                                  INC R2
                                  CMP 51, R2
}                                 JL Loop
```

- **步骤2：合并同类指令，确定操作码**
  - 合并同类指令，例如，
  - MOV 0, R1
  - MOV 1, R1
  - MOV 2, R2

# 指令集设计过程：步骤2

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes (one for a type)
  - Merge similar instructions into a type
    合并同类指令，如 MOV 0, R1; MOV 1, R1; MOV 2, R2
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions, needing 3 bits to specify
  6种指令，需要3比特操作码

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2
  fib[i] = fib[i-1] + fib[i-2]      MOV 0, R1       // Loop
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

| Instruction Type | Opcode | Semantics 指令语义 |
|---|---|---|
| MOV to Register | 000 | Assign an immediate value to a register |
| MOV to Memory | 001 | Assign the content of a register to M[Address] |
| ADD | 010 | R1 + M[Address] → R1 |
| INC | 011 | R + 1 → R (R is a register) |
| CMP | 100 | Compare to a value, assign the result to FLAGS |
| JL | 101 | If FLAGS is '<' (less than), Loop → PC |

# 指令集设计过程：步骤2

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
  fib[i] = fib[i-1] + fib[i-2]      MOV 0, R1       // Loop
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

| Instruction Type | Opcode | Semantics |
|---|---|---|
| MOV to Register | 000 | Assign an immediate value to a register |
| **MOV to Memory** | **001** | **Assign the content of a register to M[Address]** |
| ADD | 010 | R1 + M[Address] → R1 |
| INC | 011 | R + 1 → R  (R is a register) |
| CMP | 100 | Compare to a value, assign the result to FLAGS |
| JL | 101 | If FLAGS is '<' (less than), Loop → PC |

# 指令集设计过程：步骤2

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

| fib code | Instructions |
| --- | --- |
| fib[0] = 0 | **MOV 0, R1** |
| | MOV R1, M[R0] √ |
| fib[1] = 1 | **MOV 1, R1** |
| | MOV R1, M[R0+8] √ |
| for i := 2; i < 51; i++ { | **MOV 2, R2** |
| fib[i] = fib[i-1] + fib[i-2] | **MOV 0, R1** // Loop |
| | ADD M[R0+R2*8-16], R1√ |
| | ADD M[R0+R2*8-8], R1√ |
| | MOV R1, M[R0+R2*8-0] √ |
| | INC R2 |
| | CMP 51, R2 |
| } | JL Loop |

| Instruction Type | Opcode | Semantics |
| --- | --- | --- |
| MOV to Register | 000 | Assign an immediate value to a register |
| MOV to Memory | 001 | Assign the content of a register to M[Address] |
| **ADD** | **010** | **R1 + M[Address] → R1** |
| INC | 011 | R + 1 → R (R is a register) |
| CMP | 100 | Compare to a value, assign the result to FLAGS |
| JL | 101 | If FLAGS is '<' (less than), Loop → PC |

# 指令集设计过程：步骤2

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                        MOV 0, R1
                                  MOV R1, M[R0]  √
fib[1] = 1                        MOV 1, R1
                                  MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {         MOV 2, R2
  fib[i] = fib[i-1] + fib[i-2]    MOV 0, R1      // Loop
                                  ADD M[R0+R2*8-16], R1√
                                  ADD M[R0+R2*8-8], R1√
                                  MOV R1, M[R0+R2*8-0] √
                                  INC R2 √
                                  CMP 51, R2
}                                 JL Loop
```

| Instruction Type | Opcode | Semantics |
|---|---|---|
| MOV to Register | 000 | Assign an immediate value to a register |
| MOV to Memory | 001 | Assign the content of a register to M[Address] |
| ADD | 010 | R1 + M[Address] → R1 |
| **INC** | **011** | **R + 1 → R  (R is a register)** |
| CMP | 100 | Compare to a value, assign the result to FLAGS |
| JL | 101 | If FLAGS is '<' (less than), Loop → PC |

# 指令集设计过程：步骤2

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1       // Loop
                                    ADD M[R0+R2*8-16], R1√
                                    ADD M[R0+R2*8-8], R1√
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2 √
                                    CMP 51, R2 √
}                                   JL Loop
```

| Instruction Type | Opcode | Semantics |
|---|---|---|
| MOV to Register | 000 | Assign an immediate value to a register |
| MOV to Memory | 001 | Assign the content of a register to M[Address] |
| ADD | 010 | R1 + M[Address] → R1 |
| INC | 011 | R + 1 → R (R is a register) |
| **CMP** | **100** | **Compare to a value, assign the result to FLAGS** |
| JL | 101 | If FLAGS is '<' (less than), Loop → PC |

# 指令集设计过程：步骤2

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0] √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
  fib[i] = fib[i-1] + fib[i-2]      MOV 0, R1      // Loop
                                    ADD M[R0+R2*8-16], R1√
                                    ADD M[R0+R2*8-8], R1√
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2 √
                                    CMP 51, R2 √
}                                   JL Loop √
```

| Instruction Type | Opcode | Semantics |
|---|---|---|
| MOV to Register | 000 | Assign an immediate value to a register |
| MOV to Memory | 001 | Assign the content of a register to M[Address] |
| ADD | 010 | R1 + M[Address] → R1 |
| INC | 011 | R + 1 → R  (R is a register) |
| CMP | 100 | Compare to a value, assign the result to FLAGS |
| **JL** | **101** | **If FLAGS is '<' (less than), Loop → PC** |

# 步骤3：确定操作数

| | |
|---|---|
| fib[0] = 0 | MOV 0, R1 |
| | MOV R1, M[R0] |
| fib[1] = 1 | MOV 1, R1 |
| | MOV R1, M[R0+8] |
| for i := 2; i < 51; i++ { | MOV 2, R2 |
| fib[i] = fib[i-1] + fib[i-2] | MOV 0, R1 // Loop |
| | ADD M[R0+R2*8-16], R1 |
| | ADD M[R0+R2*8-8], R1 |
| | MOV R1, M[R0+R2*8-0] |
| | INC R2 |
| | CMP 51, R2 |
| } | JL Loop |

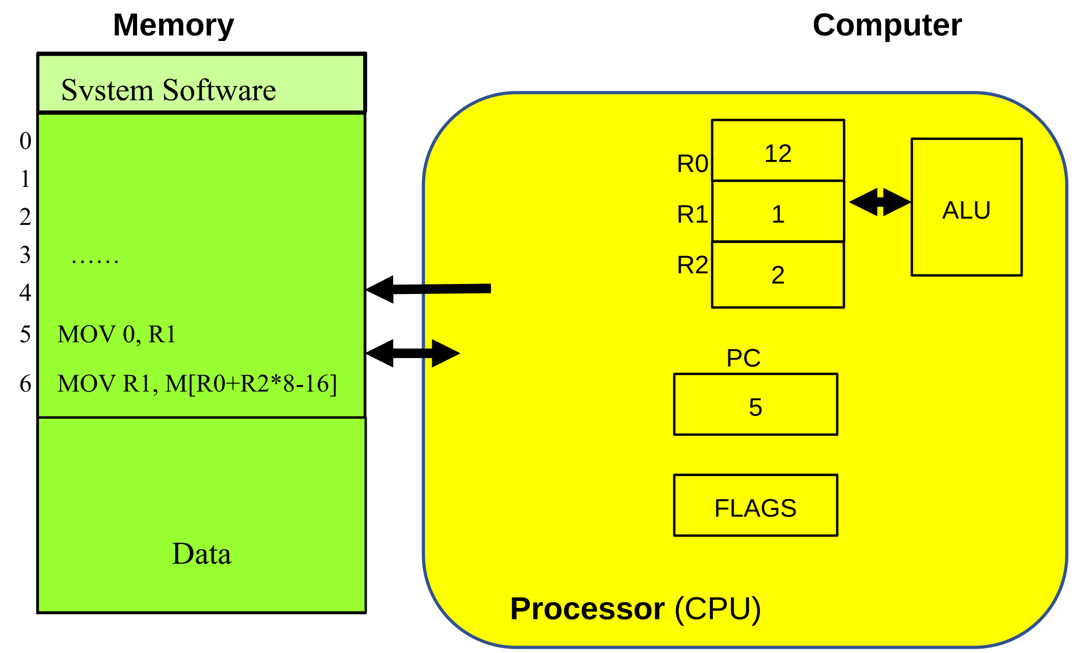
- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
- For each opcode, determine its operands 确定操作数编码
  - Assuming the instruction length = 11 bits In practice, assume 8, 16, 32 or 64 bits
  - There are 3 data registers, needing 2 bits
  - Leave 6 bits for immediate value
  - The base+index+offset mode
    - Address = R0 + R2*I + J, where
      R0, R2 are fixed
      I = 0, 1, 2, 4, 8
      J = 0, ± 4, ± 8, ± 16
    - 5×7 = 35 possible (I, J) pairs
    - $35<2^6$, 6 bits are enough
- Notes
  - For INC R2, operand 1 is not used
  - JL has only one operand

| Opcode 3-bit | Operand 1 Immediate Value, 6-bit | Operand 2 Register, 2-bit | Instruction |
|---|---|---|---|
| 000 | 000000 | 01 | MOV 0, R1 |
| 000 | 000001 | 01 | MOV 1, R1 |
| 000 | 000010 | 10 | MOV 2, R2 |
| 011 | | 10 | INC R2 |
| 100 | 110011 | 10 | CMP 51, R2 |
| 101 | 00000101 | | JL Loop |

| Opcode 3-bit | Operand 1 Memory Address, 6-bit | Operand 2 Register, 2-bit | Instruction |
|---|---|---|---|
| 001 | R0+R2*0+0 | 01 | MOV R1, M[R0] |
| 001 | R0+R2*0+8 | 01 | MOV R1, M[R0+8] |
| 001 | R0+R2*0-0 | 01 | MOV R1, M[R0+R2*8-0] |
| 010 | R0+R2*8-8 | 01 | ADD M[R0+R2*8-8], R1 |
| 010 | R0+R2*8-16 | 01 | ADD M[R0+R2*8-16], R1 |

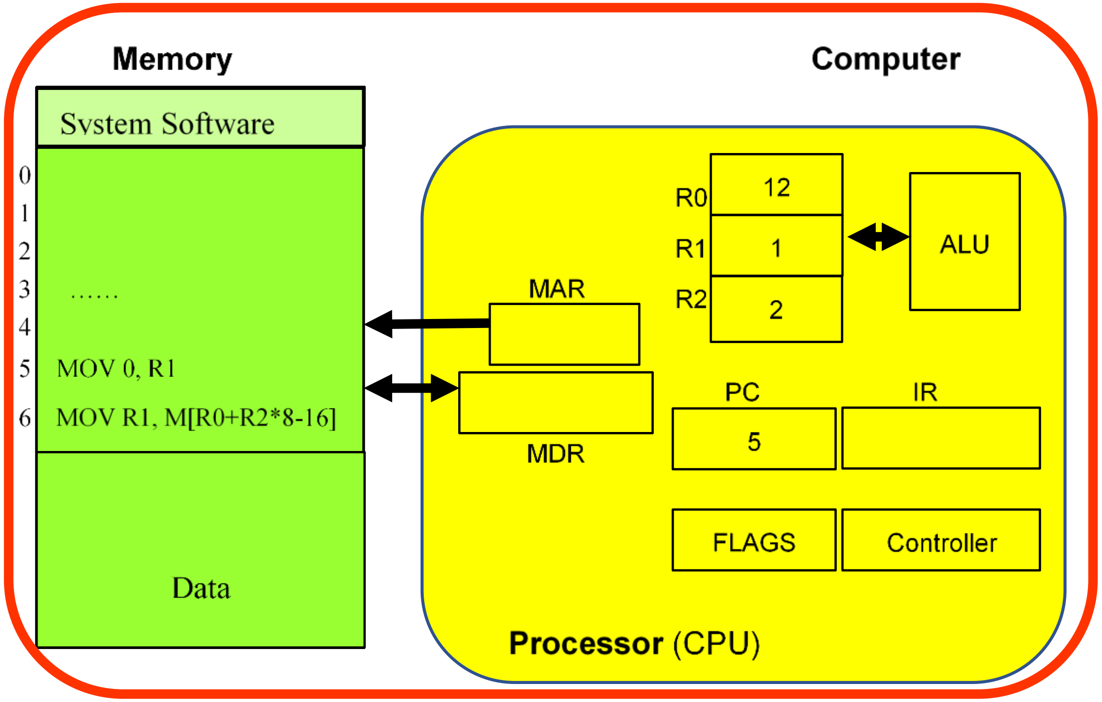
# 指令集只看见可见寄存器

- To see an example of executing instruction MOV 0, R1



# 3.2 一条指令在处理器内部是如何执行的？

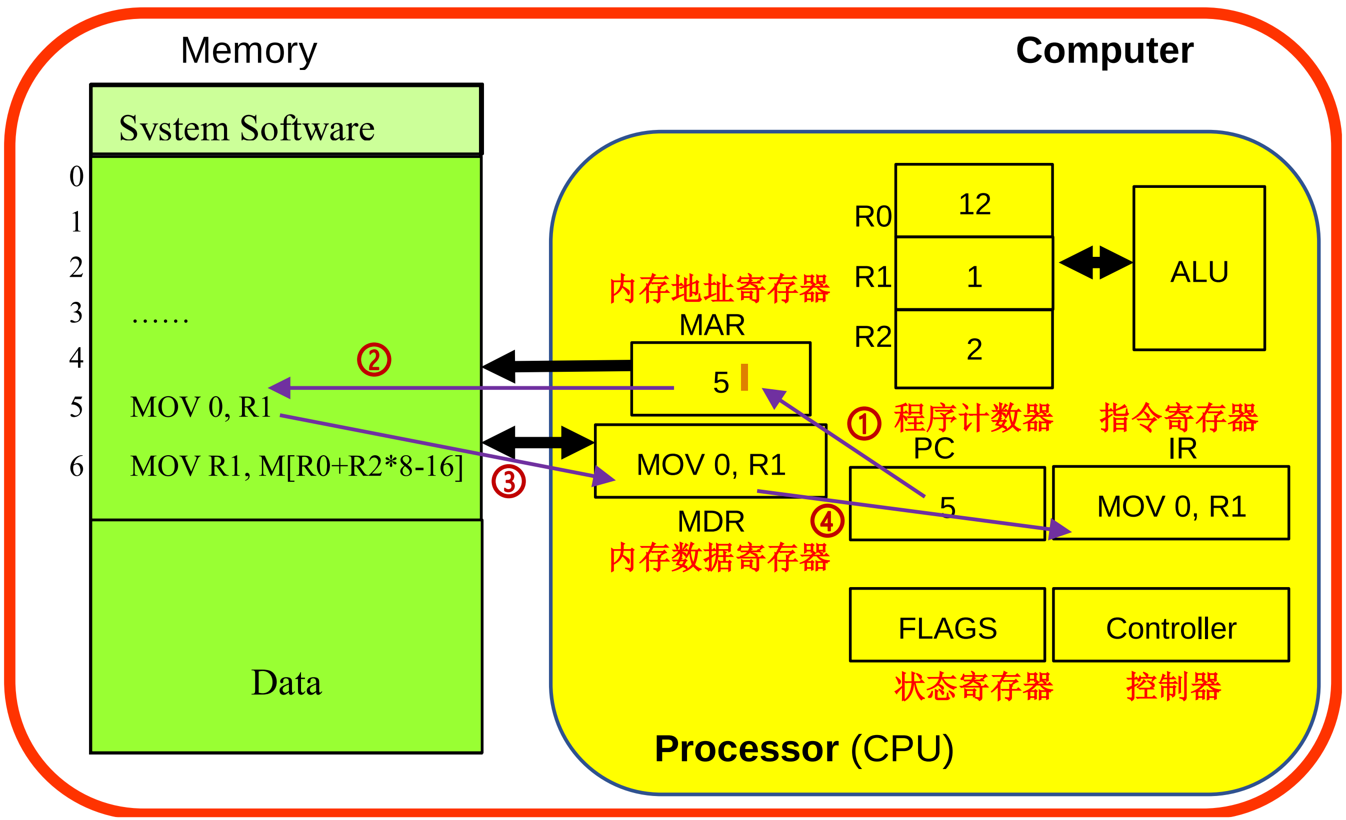
- To see an example of executing instruction MOV 0, R1
  - by a **3-stage instruction pipeline** 三级指令流水线
    - Instruction Fetch (IF): IR←M[PC] 取指
    - Instruction Decode (ID): Signals = Decode(IR) 译码
    - Instruction Execute (EX): R1 ← 0; PC ← PC+1 执行
- Internal components not visible to user
  需要关心内部部件（不可见）
  - IR: Instruction Register holding the instruction being executed
  - MAR: Memory Address Register, holding the memory address used
  - MDR: Memory Data Register, holding the data for a load or store
  - Controller: control circuit to generate control signals



# Execution details

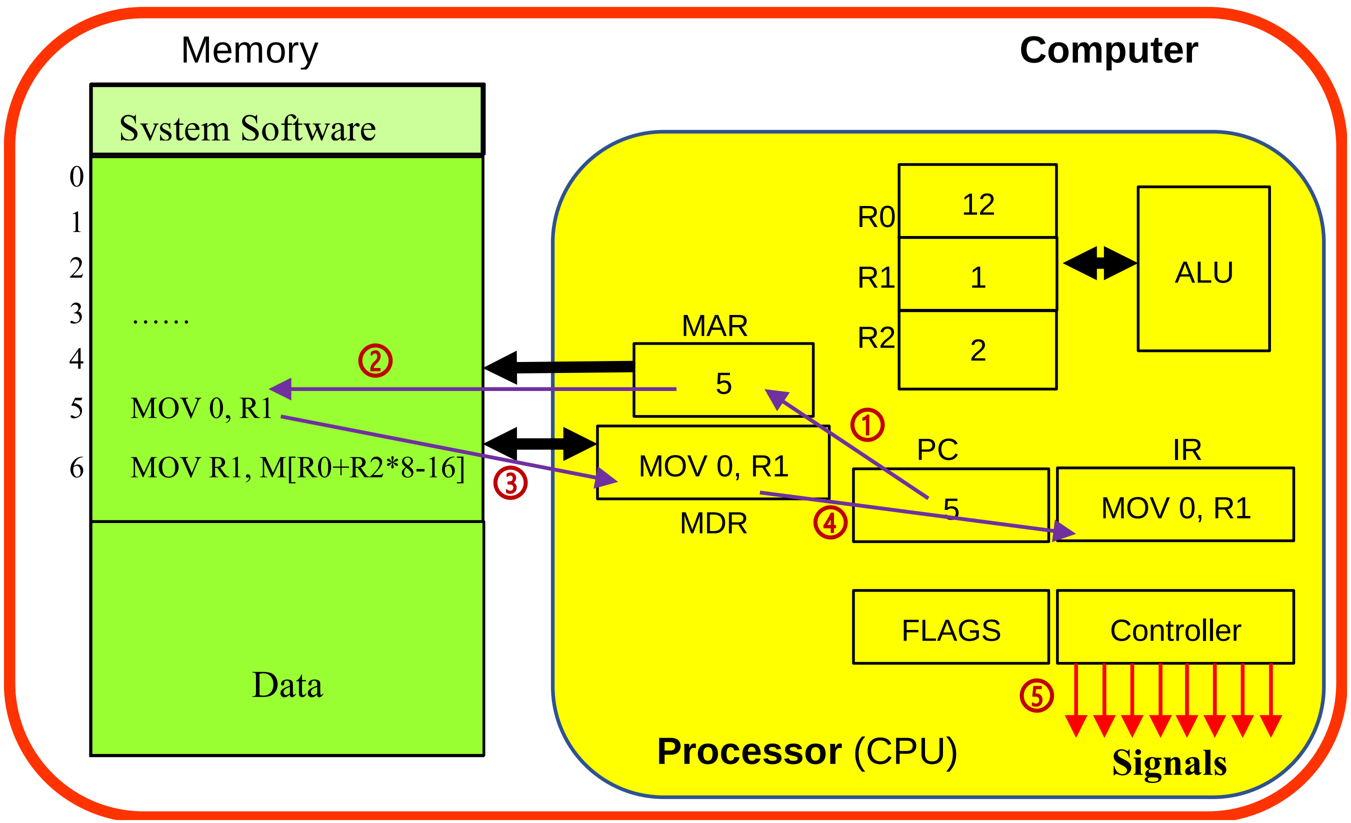
- Instruction Fetch (IF): IR←M[PC]
- Instruction Decode (ID): Signals = Decode(IR)
- Instruction Execute (EX): R1 ← 0; PC ← PC+1

- IF stage (micro operations ①②③④) 取指



- Instruction Fetch (IF): IR←M[PC]
- Instruction Decode (ID): Signals = Decode(IR)
- Instruction Execute (EX): R1 ← 0; PC ← PC+1

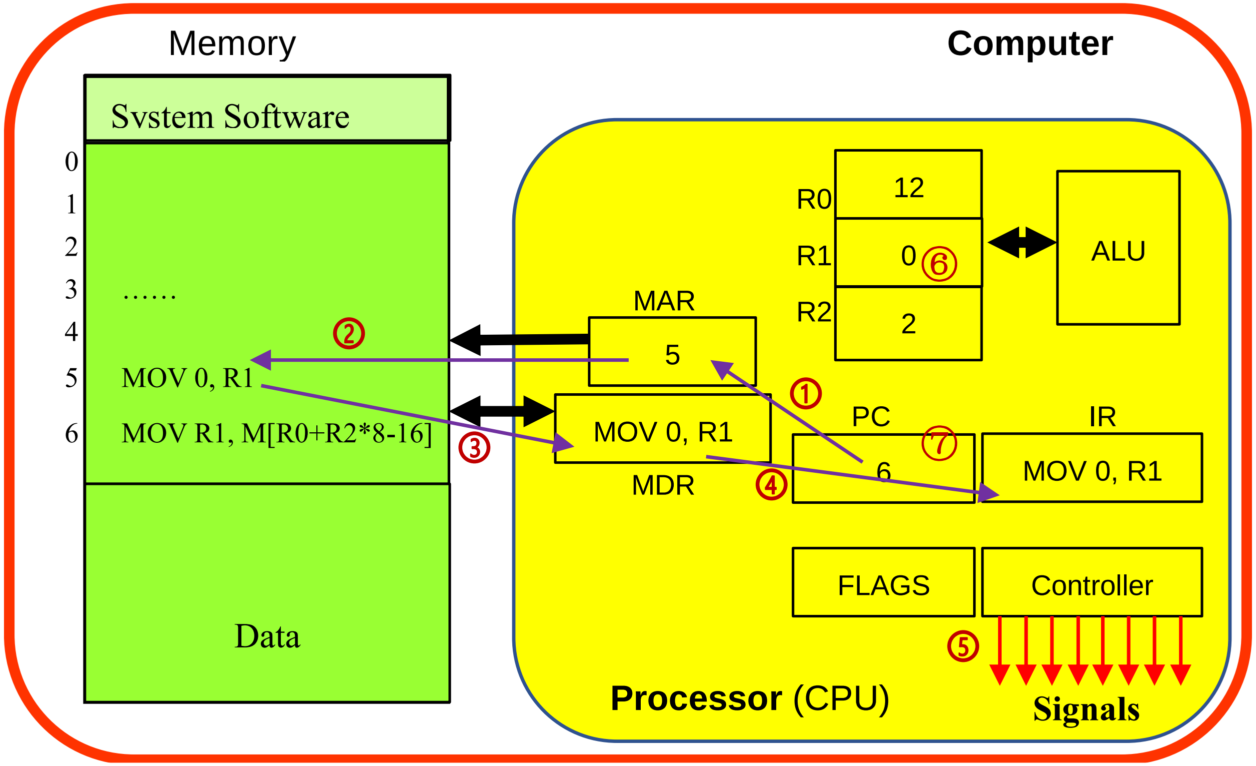
- IF stage (micro operations ①②③④) 取指
- ID stage (micro operation ⑤) 译码




**译码：**
控制器从IR的当前指令产生控制信号

- Instruction Fetch (IF): IR←M[PC]
- Instruction Decode (ID): Signals = Decode(IR)
- Instruction Execute (EX): R1 ← 0; PC ← PC+1

- IF stage (micro operations ①②③④) 取指
- ID stage (micro operation ⑤) 译码
- EX stage (micro operations ⑥⑦) 执行
  - 0→R1; PC+1→PC




执行：
控制信号驱动相关部件，例如使能相关**MUX**，执行指令要求的操作
0→R1
PC+1→PC

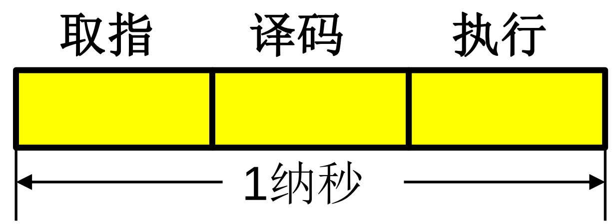
# 3.3 指令周期的重叠执行

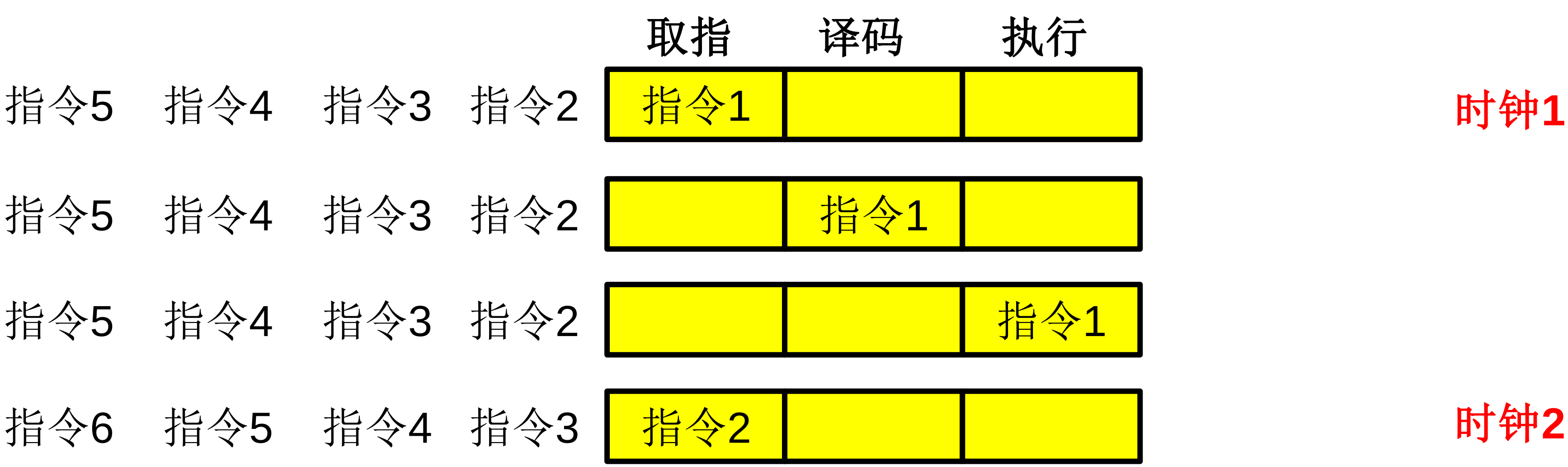
- 每个指令周期包含三个操作阶段（微操作）
  - 例如，执行指令MOV R1, M[R0] 的三步骤如下：
    - 取指操作：IR ← M(PC); PC←PC+1
    - 译码操作：Signals = Decode(IR)
    - 执行操作：M[R0] ← R1

| | | | | 取指 | 译码 | 执行 | |
|---|---|---|---|---|---|---|---|
| 指令5 | 指令4 | 指令3 | 指令2 | 指令1 | | | 时钟1 |
| 指令5 | 指令4 | 指令3 | 指令2 | | 指令1 | | 时钟2 |
| 指令5 | 指令4 | 指令3 | 指令2 | | | 指令1 | 时钟3 |
| 指令6 | 指令5 | 指令4 | 指令3 | 指令2 | | | 时钟4 |

# 指令流水线

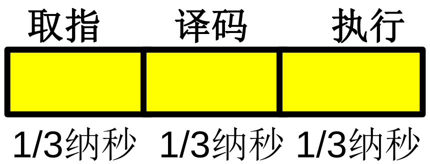
- 假设指令流水线总延时是1纳秒，流水线各阶段没有重叠（No overlap）
  - 执行完当前指令，再执行下一条指令
    - 处理器时钟周期 = 1纳秒
    - 处理器主频（frequency）= 1/(1纳秒) = 1 GHz
    - 峰值速度为1 GIPS
      (Giga Instructions Per Second)
      即每秒十亿条指令

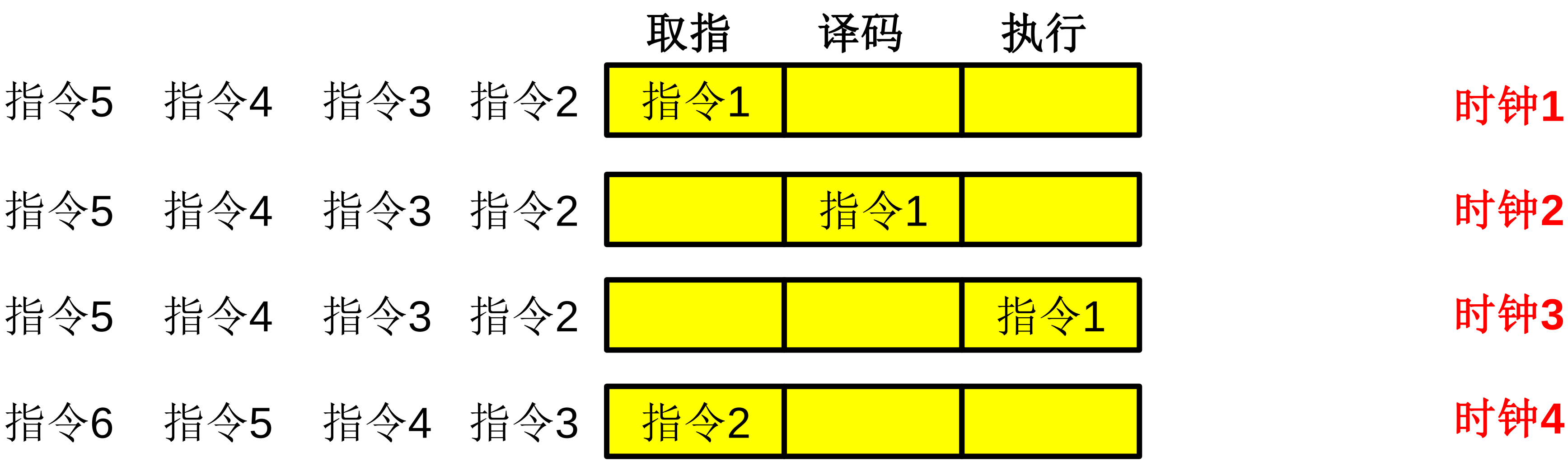


# 指令流水线

- 假设指令流水线总延时是1纳秒，流水线各阶段没有重叠（No overlap）
  - **执行完当前指令，再执行下一条指令**
  - **提升处理器主频到3 GHz**
    - 处理器时钟周期 = (1/3)纳秒
    - 处理器主频（frequency）= 1/(1/3) = 3 GHz
    - 执行一条指令需要三个时钟周期
    - 峰值速度为1 GIPS (Giga Instructions Per Second) 即每秒十亿条指令



# 指令流水线机制总是利用重叠（overlap）

**处理器主频= 3 GHz，指令流水线同时执行三条指令，平均每个时钟周期执行完毕一条指令**

**峰值速度：3 Giga Instructions Per Second (GIPS)，即每秒执行30亿条指令**

| 取指 | 译码 | 执行 |
|---|---|---|
| 1/3纳秒 | 1/3纳秒 | 1/3纳秒 |

- 假设*K*级指令流水线总延时是1纳秒，则有
  - 处理器时钟周期是每级延时 = 1/*K* 纳秒
  - 处理器主频 = *K* GHz
  - 指令流水线同时执行*K*条指令，平均每个时钟周期执行完毕一条指令
  - 处理器峰值速度：*K* GIPS

| | | | | | 取指 | 译码 | 执行 |
|---|---|---|---|---|---|---|---|
| **时钟0:** | 指令4 | 指令3 | 指令2 | 指令1 | | | |
| **时钟1:** | 指令5 | 指令4 | 指令3 | 指令2 | 指令1 | | |
| **时钟2:** | 指令6 | 指令5 | 指令4 | 指令3 | 指令2 | 指令1 | |
| **时钟3:** | 指令7 | 指令6 | 指令5 | 指令4 | 指令3 | 指令2 | 指令1 |
| **时钟4:** | 指令8 | 指令7 | 指令6 | 指令5 | 指令4 | 指令3 | 指令2 |
| **时钟5:** | 指令9 | 指令8 | 指令7 | 指令6 | 指令5 | 指令4 | 指令3 |

# 课堂小测验

- 姓名：　　　　　　　　　　学号：
- 请指出下述哪一个断言是错误的（）

A. 指令流水线采用重叠执行（overlapped execution）机制，可同时执行多条指令

B. 指令流水线的每一级都是一个时钟同步的时序电路，即自动机

C. 指令流水线的每一级都可能需要多个时钟周期才能完成

D. 指令流水线的每一级都是一个组合电路

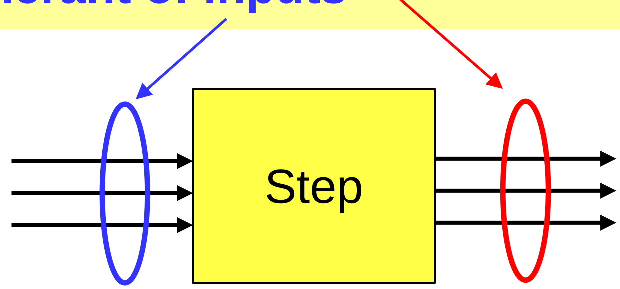
# 4. Postel's Robustness Principle 宽进严出原理

- 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 误差、漂移、失真不会积累

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**

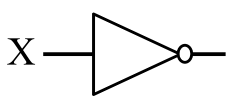


# 例子：非门电路设计

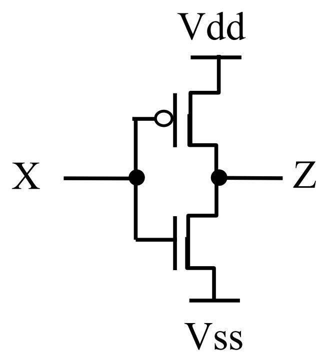
- 宽进严出，输出重设到正确值，即>Voh，或<Vol

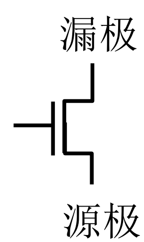
**真值表与符号**

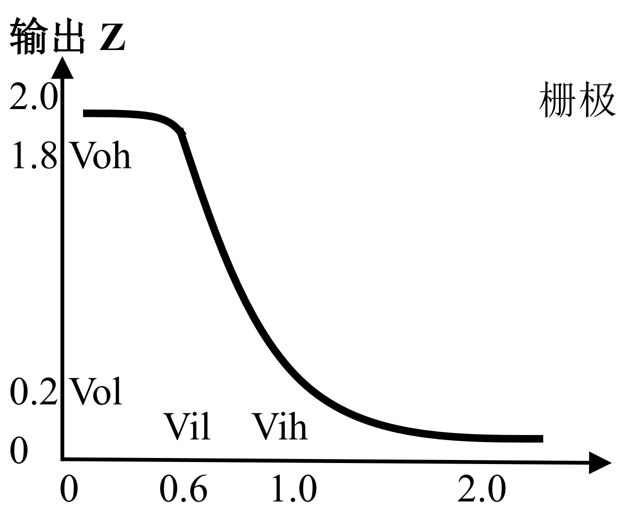
| X | Z |
| --- | --- |
| 0 | 1 |
| 1 | 0 |




**CMOS 电路**




**电路参数规定**

Vdd = 2 伏

Vss = 0 伏

Vth = 0.7 伏

**① 糟糕设计**

**输入与输出采用相同规定**

逻辑 1：0.7~2.0 伏

逻辑 0：0.0~0.7 伏

**② 宽进严出设计**

**输入电压**

逻辑 1：1.0~2.0 伏

逻辑 0：0.0~0.6 伏

**输出电压**

逻辑 1：1.8~2.0 伏

逻辑 0：0.0~0.2 伏

# 5. von Neumann’s Exhaustiveness Principle 冯诺依曼穷尽原理

- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
  当自动解决问题时，必须给计算机指令，穷尽所有细节
- 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

# 冯诺依曼穷尽原理

- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
- 1-minute quiz
  - Q: How to cover “absolutely exhaustive detail”? 怎么可能穷尽所有细节？
  - 可能有无穷多种细节，如何穷尽？
  - 使用前述**计算归纳法**
    - Identify first step
      确定第一步
    - Execute single step
      正确执行每一步（当前步骤）
      应对异常
    - Identify and transition to next step 确定并过渡到下一步

> 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

# 冯诺依曼穷尽原理

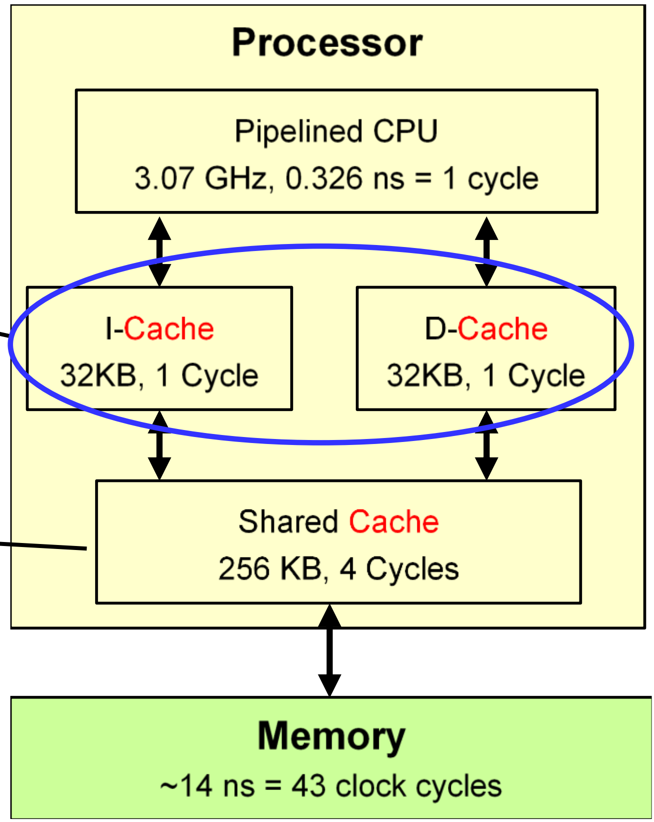
- Computer must be given instructions in absolutely exhaustive detail when automatically solving a problem
- 1-minute quiz
  - Q: How to achieve “in absolutely exhaustive detail”? List the types of instructions, and give an example for each type
  - A: Answers to more fundamental questions 必须考虑正常执行和异常执行
    - Where and what is the first instruction, when the computer power is turned on?
      开机之后执行的**第一条指令**在哪里？是什么？
    - How to determine the next instruction to execute?
      如何确定**下一条指令**？
    - What types of exceptions are there, to normal execution of programs?
      如何保证**当前指令**正确执行？
      没有正确执行，只有两种可能：（1）没有正确设计实现；（2）出现**异常**
      有哪些异常类别？如何应对？

# The first instruction to execute
## when the computer power turns on
## 开机后执行的第一条指令：在哪里？是什么？

- Example with an x86 processor **x86处理器例子**
  - Where: the first instruction to execute is at memory address 0xFFFFFFF0 在哪里？处理器规定的地址空间的某个高位地址
  - What: a jump instruction, e.g., JUMP 000F0000 是什么
    - Address 000F0000 contains the entry instruction for the BIOS code 跳转指令：跳转到BIOS（最底层系统软件）的入口地址
- Why? 什么道理？
  - Computer starts by executing the BIOS firmware code
    - To initialize the computer and to load the operating system 计算机开机后先执行固件，做初始化工作（如自检），然后载入系统软件
  - Using a jump instruction upfront increases flexibility 跳转指令增加灵活性
    - 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 确定下一条指令的方法1 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
    下一条指令紧跟当前指令
    - 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**)
    哈佛体系结构
    - Still widely used in the cache units of modern computers. A processor has separate instruction cache and data cache
  - In contrast, the **Princeton architecture**
    普林斯顿体系结构
    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 确定下一条指令的方法2、3

- The ENIAC method
  ENIAC方法
  - Every instruction holds the address of the next instruction
    当前指令指明下一条指令的地址
    - 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)
  当代计算机采用“程序计数器机制”
  **PC存放下一条指令的地址**

# Deal with exceptions to normal execution 应对异常

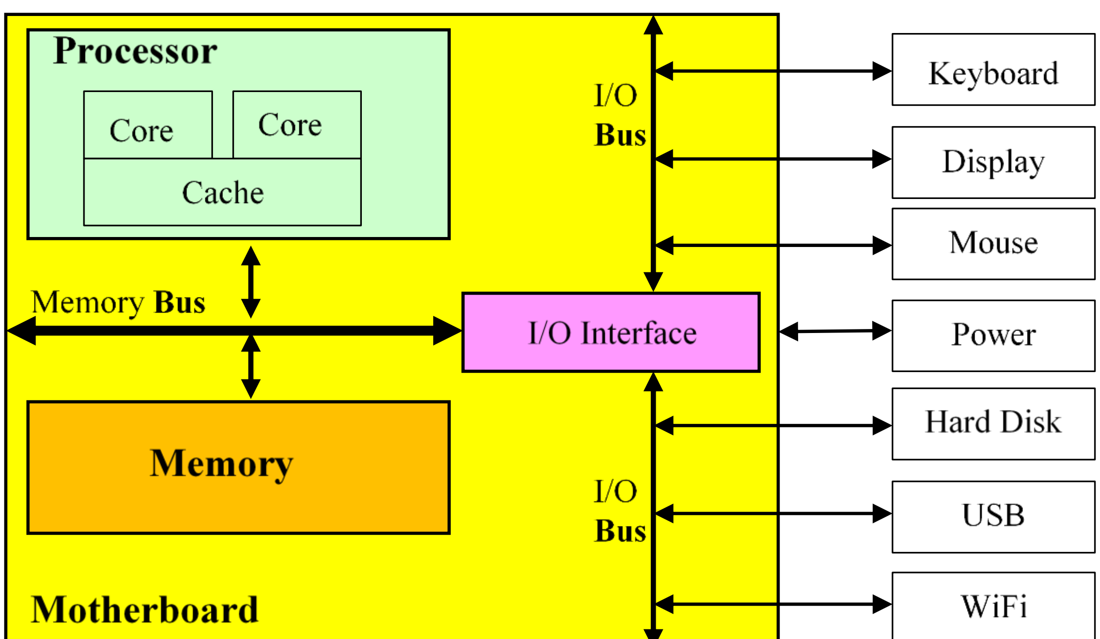
- 同学们已经体验过了Go语言编程中的异常
  - 读文件可能出错
- We have seen exceptions in programming, e.g.,

  in the Text Hider project, the statement
  - p, _ := ioutil.ReadFile(“./Autumn.bmp”) 例如，文件不存在

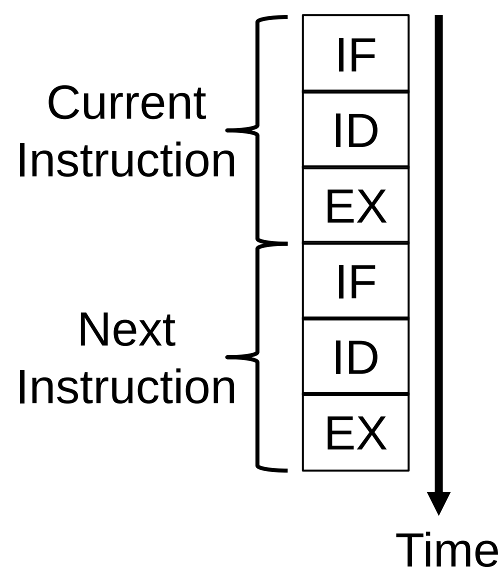
  should really be
  - p, error := ioutil.ReadFile(“./Autumn.bmp”)
    if error != nil {
    …// put exception-handling code here 出错了，执行异常处理代码
    }
    … // no error; continue normal execution 无错，继续正常执行

# Three types of exceptions are supported by computer hardware 三种异常

- In normal execution (without exception), the current instruction finishes and continue to execute the next instruction
  正常情况：执行完当前指令，继续执行下一条指令
- 异常情况：（1）不执行完当前指令；（2）不执行下一条指令；
  - Interrupt 中断
  - Hardware error 硬件出错
  - Machine check 保底异常

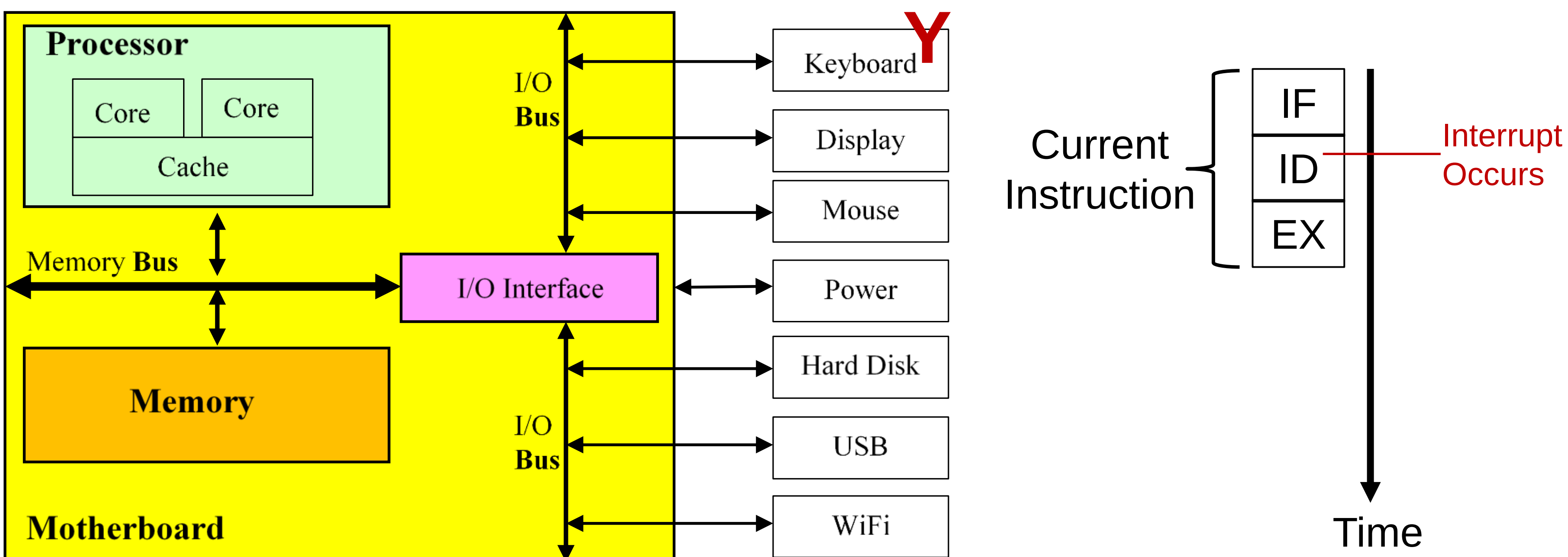



指令流水线
Stages of instruction pipeline



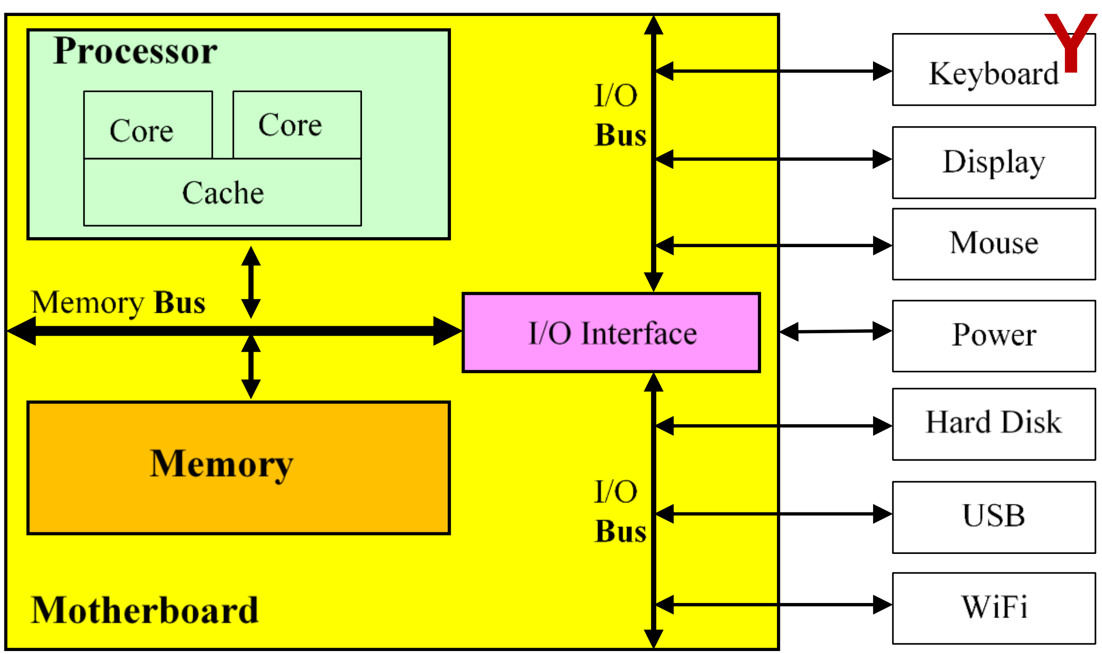
# Interrupt handling 中断处理

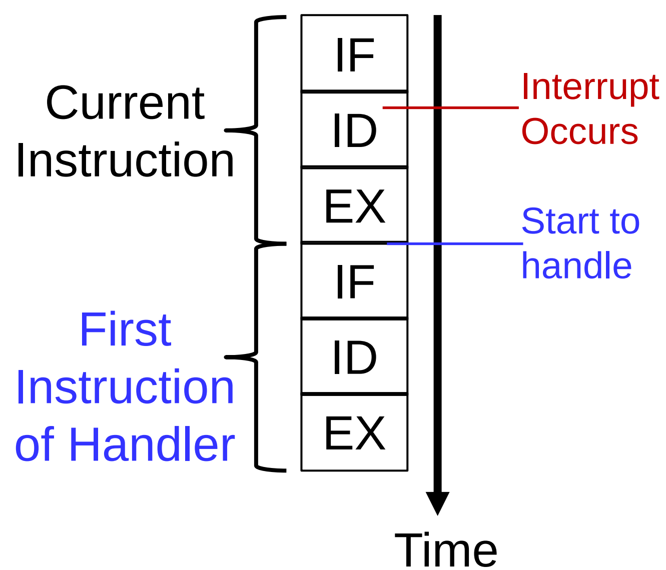
- When an interrupt occurs, e.g.,
  - When the user punches key ‘Y’ on the keyboard 用户敲了’Y’键 while the processor is executing the instruction decode stage
- What should the processor do?
  - Should it immediately take an exception-handling action?
  - Should it finish the current instruction first?



# Interrupt handling

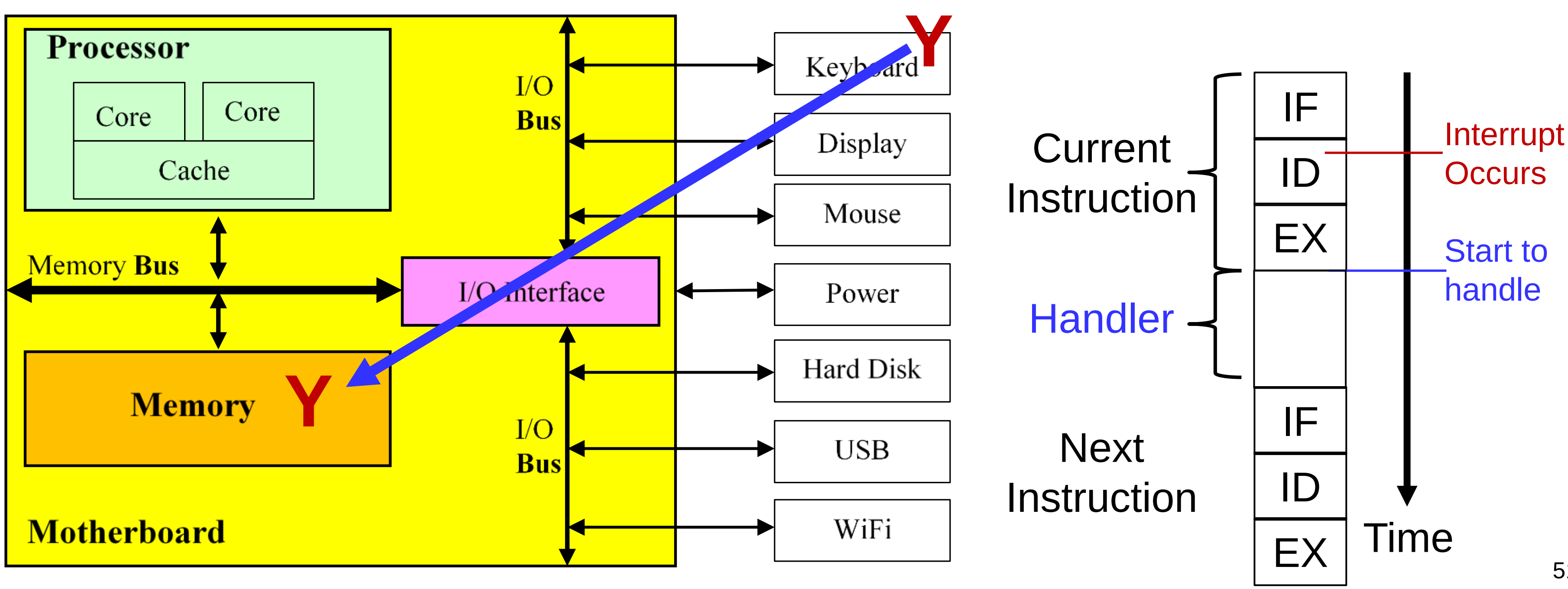
- 继续执行完毕当前指令，然后跳转到中断处理程序的入口
- The processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - 中断处理程序称为 interrupt handler



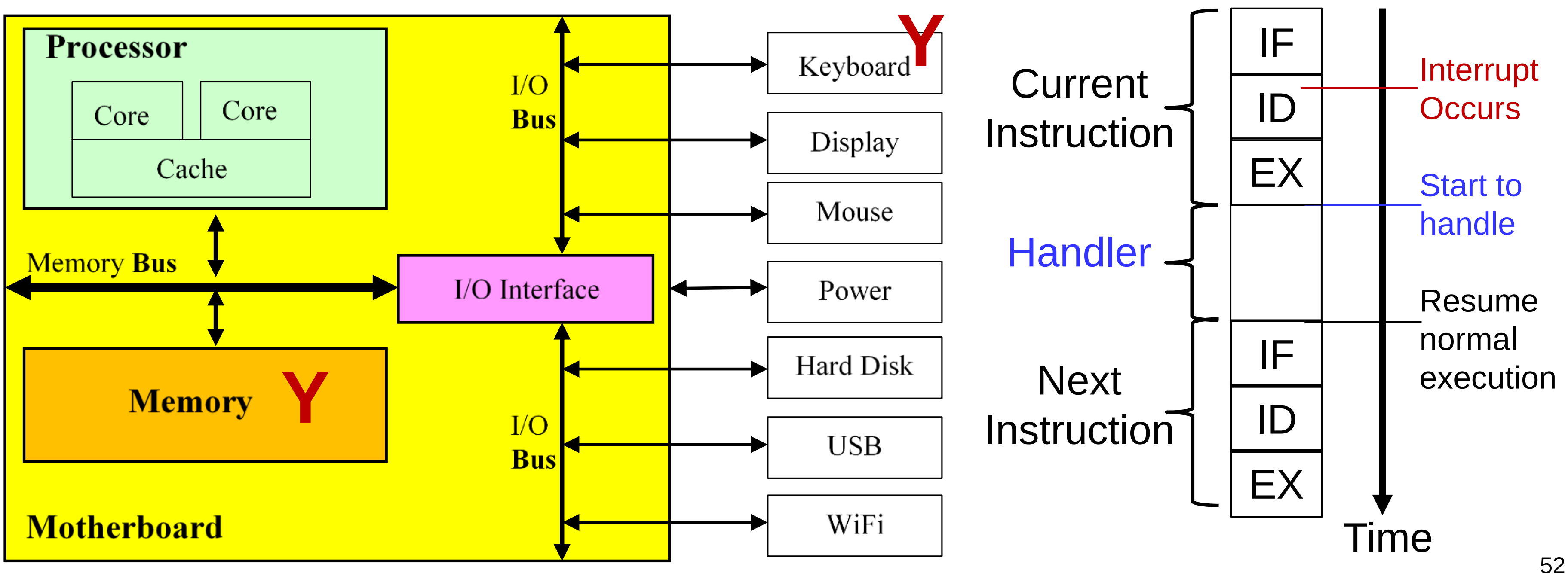
# Interrupt handling

- The processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - such as **copying the punched key value to memory**
    执行中断处理程序，包括将敲入的键值’Y’拷贝到内存



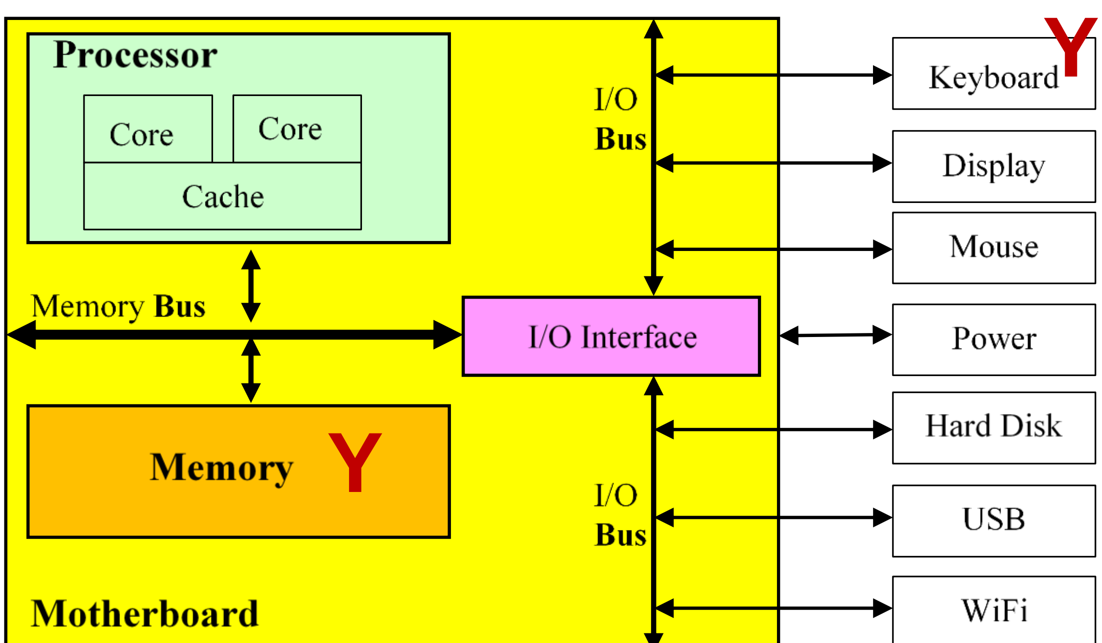
# Interrupt handling

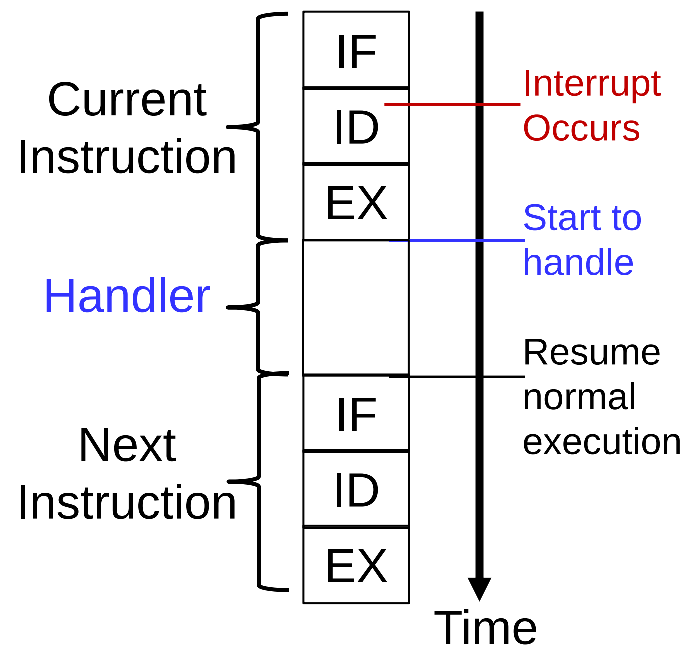
- When an interrupt occurs, the processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - such as **copying the punched key value to memory**
- Then, the processor resumes normal execution
  - by executing the original next instruction
    退出中断处理程序，恢复正常执行原来程序的下一条指令



# Interrupt handling

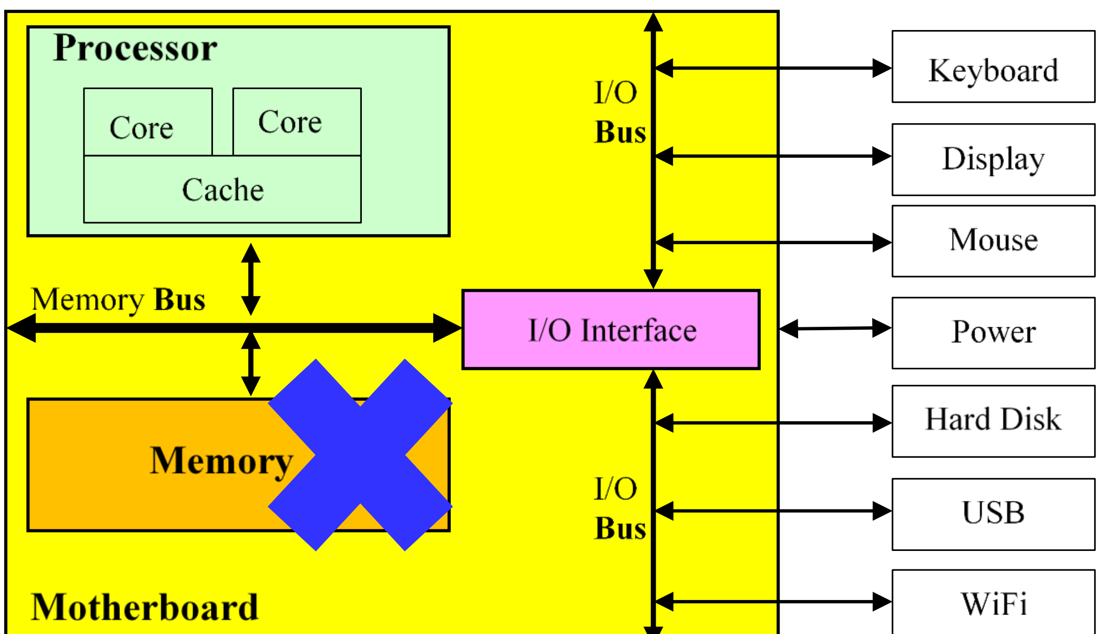
- When an interrupt occurs, the processor finishes the current instruction and jumps to an interrupt handling subprogram to handle the interrupt
  - such as **copying the punched key value to memory**
- Then, the processor resumes executing the next instruction
  - Q: how does the processor know the address of the next instruction?
  - A: need to **save context** before executing handler
    进入中断处理程序前，必须保存上下文，知道返回后该执行哪条指令

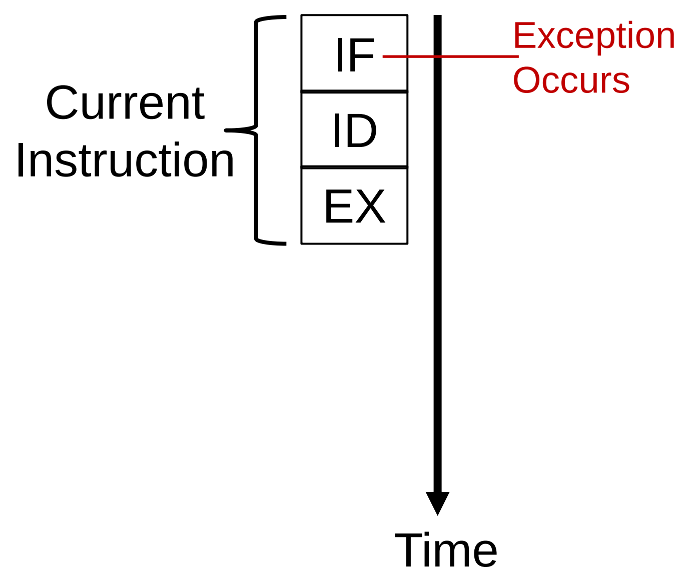


# Hardware error handling 硬件出错

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



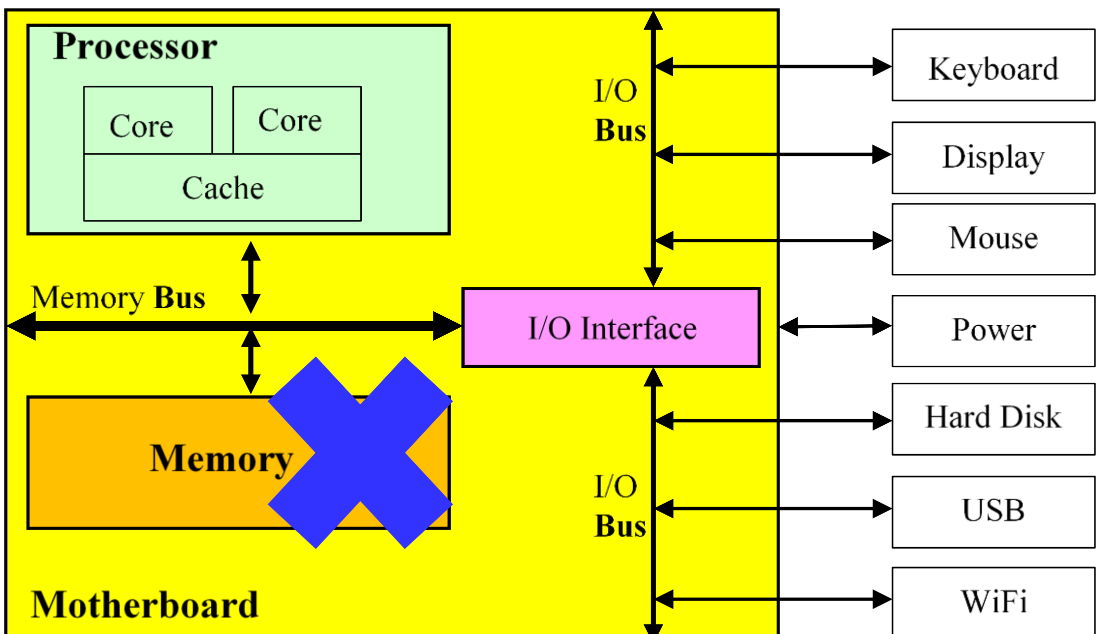



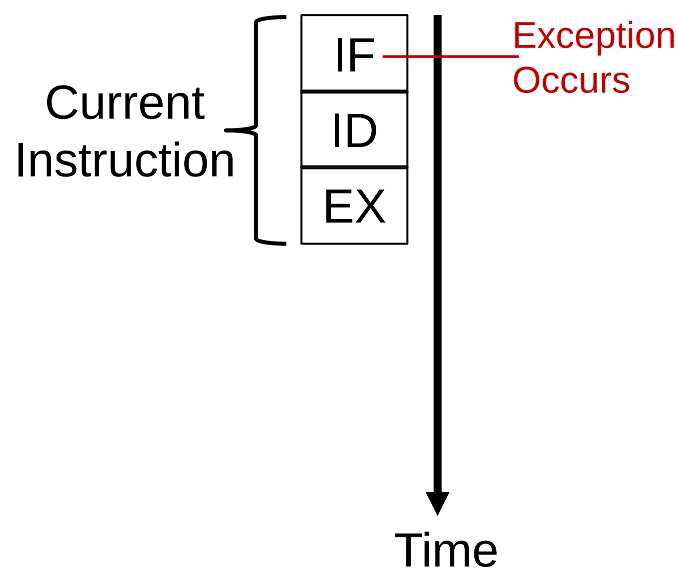
# Hardware error

- When the memory becomes faulty and generates a hardware error exception 内存条坏了
  - Should the processor finish the current instruction first?

还能够先完成当前指令再“中断”吗？

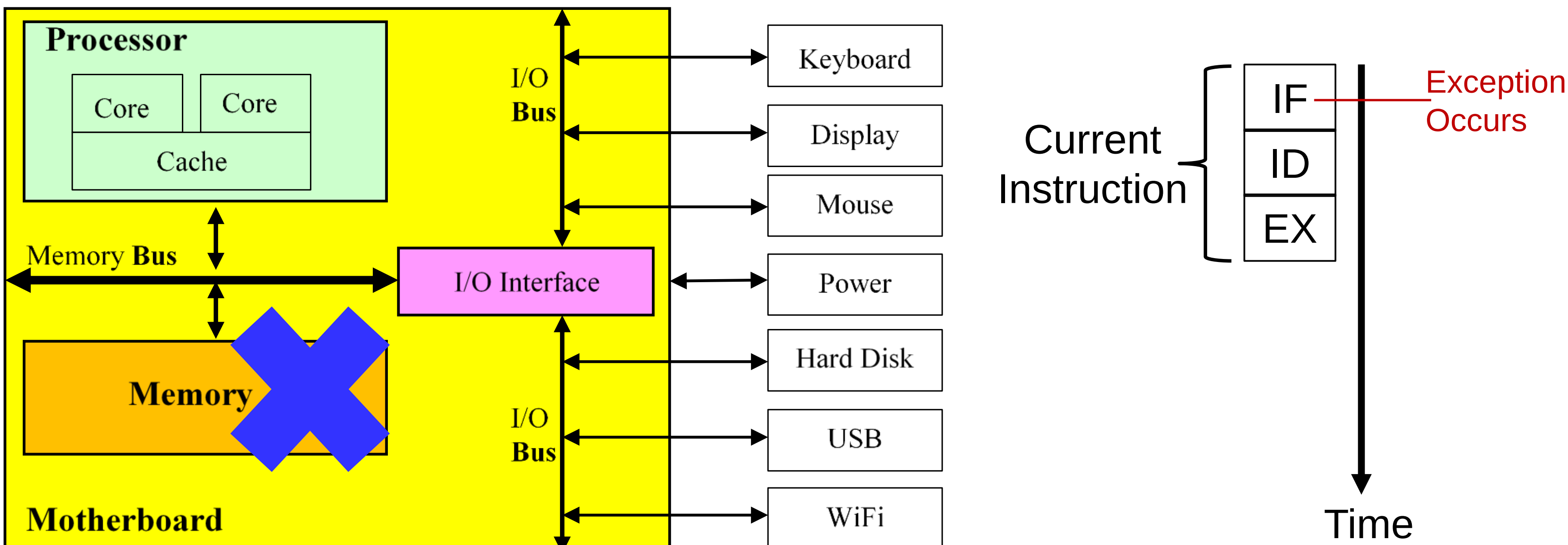
  - No, because the IF stage cannot be finished 不行！指令都取不回来
    - The instruction cannot be fetched from memory 完不成**取指（IF）**操作



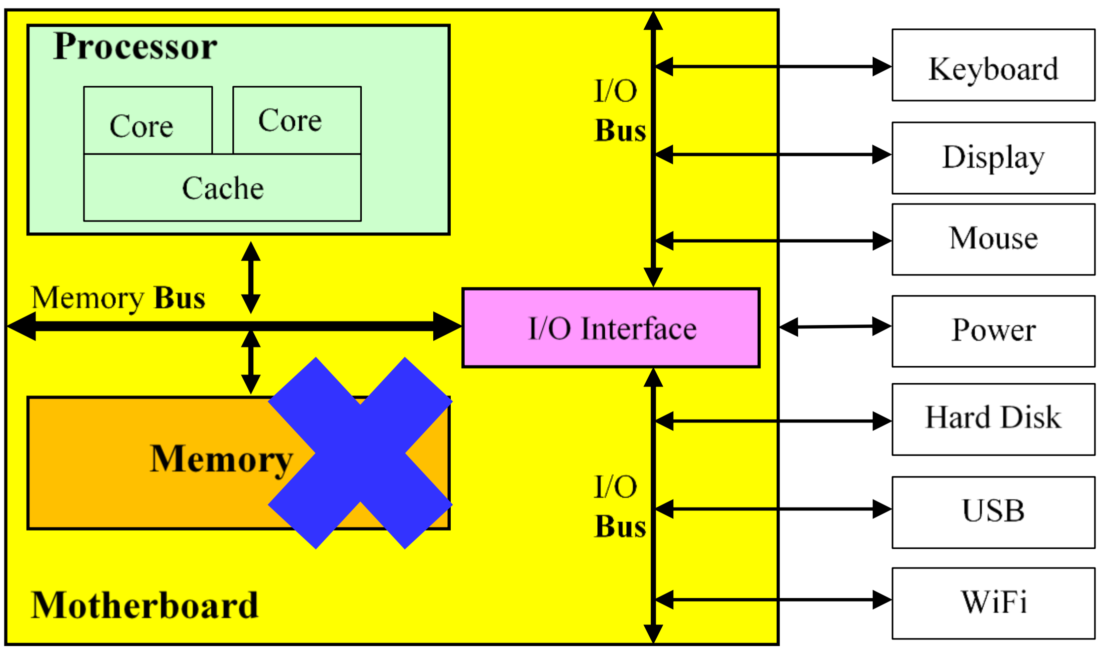
# Hardware error

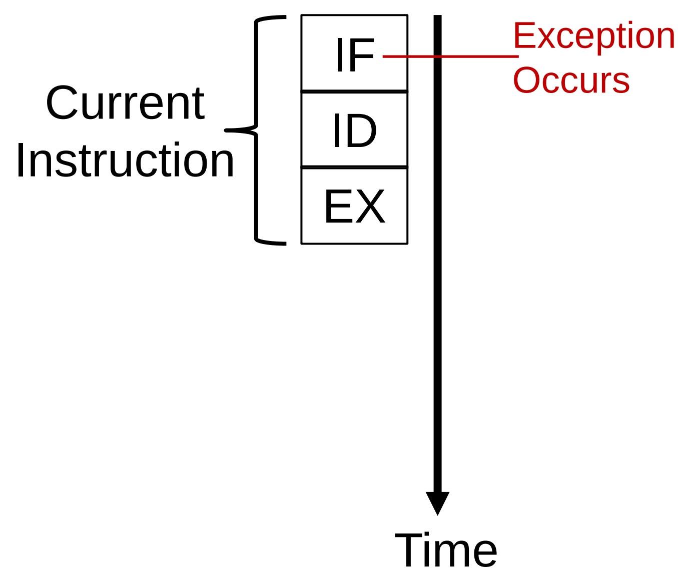
- When the memory becomes faulty and generates a hardware error exception
  - Should the processor immediately take an exception-handling action?
  - Yes, it executes an exception-handling sequence of steps without depending on the memory
    立即执行异常处理程序，无需访问内存



# Machine check 保底异常

- This is the "all other" exception, for exhaustiveness
  - Typically, an unrecoverable hardware error
- Example
  - While executing an exception-handling sequence of steps for the memory fault, the sequence experiences another error
  - The system generates minimal diagnostic information and crashes 输出最基本的诊断信息，宕机



# 6. Amdahl's law 阿姆达尔定律

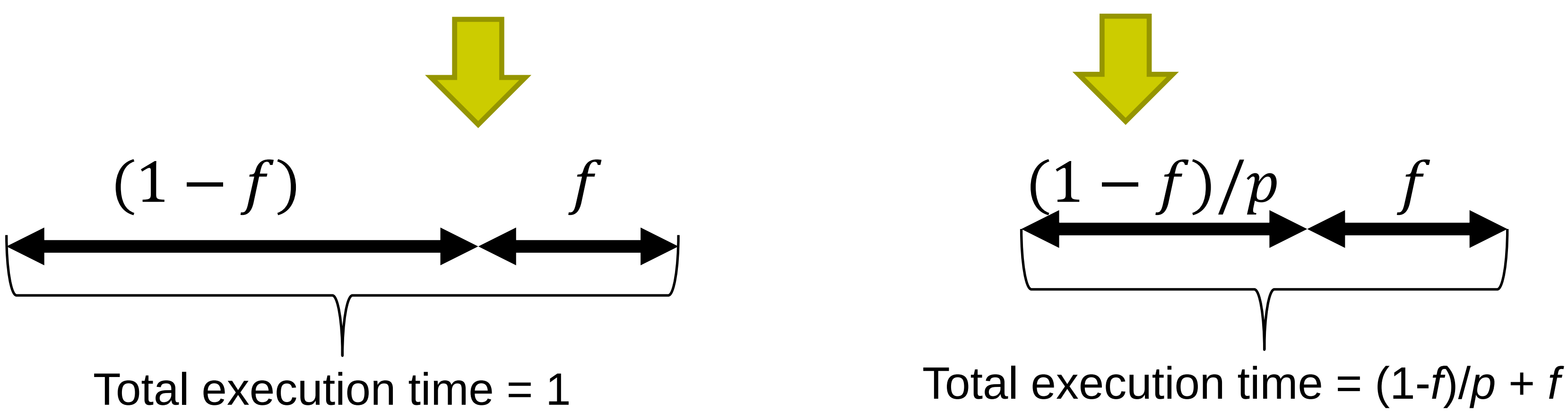
- Previous 3 principles regard correctness (seamless transition)
- Amdahl's law focuses on performance (smooth transition)

- 1-minute quiz
  - A student writes a program sort.go to read an 8-GB file of 64-bit integers, sort the integers, and write the sorted result into another file on his laptop computer. Suppose the total execution time is T, where 0.8*T is spent on I/O operations.
    - Q1: What is the problem size *N*, i.e., the number of integers?
    - A1: (a) 8 (b) 64 (c) 1024 (d) 64*1024*1024 (e) 1024*1024*1024

    - Q2: The student upgrades his laptop with a new wonder processor that is 100 times as fast as the old processor. How much time (approximately) is needed to execute sort.go on the same 8-GB file?
    - A1: (a) T/100 (b) T/5 (c) 0.1*T (d) 0.8*T

# 6.1 Amdahl's law　阿姆达尔定律

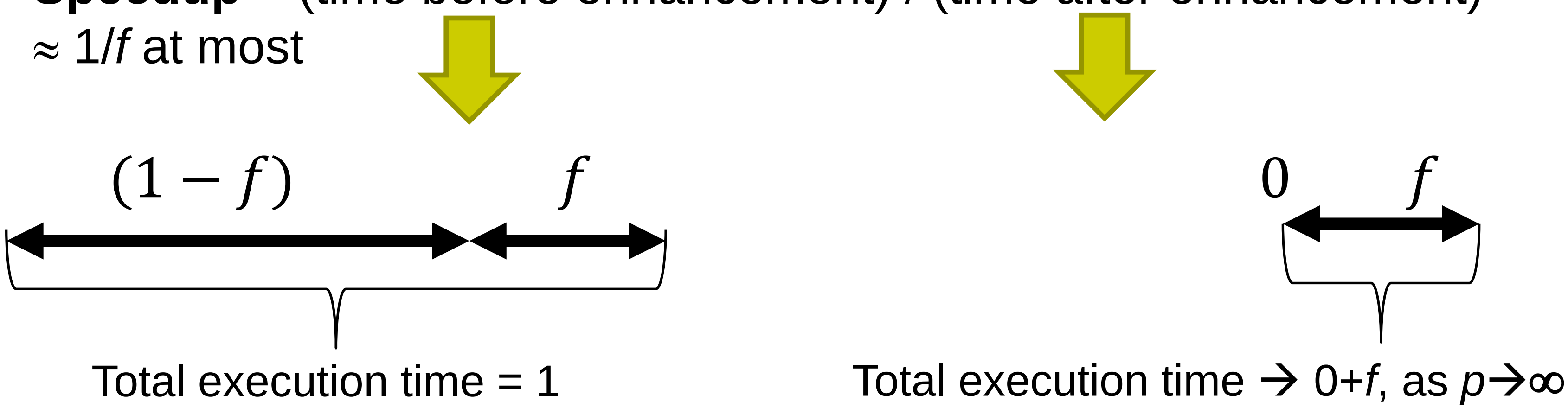
- After enhancing a portion of a system, the speedup obtained is upper bounded by the reciprocal of the other portion's time.
  改进系统的一部分后，系统加速比不会超过未改进部分执行时间的倒数
- Suppose a system's execution time is broken into two portions $(1-f)$ and $f$, such that $(1-f) > f$　系统执行时间1划分成两部分$(1-f)$, $f$
- Amdahl's law: Enhancement on the $(1-f)$ portion can lead to a speedup no more than 1/*f*
  - Speedup approaches but never exceeds 1/*f*
  - **Speedup** = (time before enhancement) / (time after enhancement)
    加速比 = 改进前的执行时间 / 改进后的执行时间



# Amdahl’s law

- After enhancing a portion of a system, the speedup obtained is upper bounded by the reciprocal of the other portion's time.

- Suppose a system's execution time is broken into two portions $(1-f)$ and $f$, such that $(1-f) > f$
- Amdahl’s law: Enhancement on the $(1-f)$ portion can lead to a speedup no more than 1/*f*
  - Speedup approaches but never exceeds 1/*f*
  - **Speedup** = (time before enhancement) / (time after enhancement) ≈ 1/*f* at most

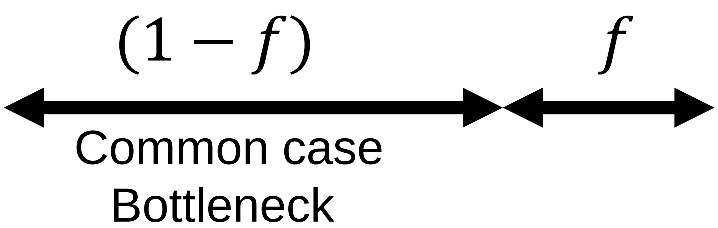


# Amdahl's law

- 1-minute quiz
  - A student writes a program sort.go to read an 8-GB file of 64-bit integers, sort the integers, and write the sorted result into another file on his laptop computer. Suppose the total execution time is T, where 0.8*T is spent on file I/O operations.
  - Q1: What is the problem size *N*, i.e., the number of integers?
  - A1: (a) 8 (b) 64 (c) 1024 (d) 64*1024*1024 (e) 1024*1024*1024
  - Q2: The student upgrades his laptop with a new wonder processor that is 100 times as fast as the old processor. How much time (approximately) is needed to execute sort.go on the same 8-GB file?
  - A1: (a) T/100 (b) T/5 (c) 0.1*T (d) 0.8*T
    - The processing time is about 0.2*T/100, but the I/O time stays the same. Thus, the total time is about 0.2*T/100 + 0.8*T = 0.802*T
    - The speedup is T / 0.802T ≈ 1.247, **only 24.7% faster**

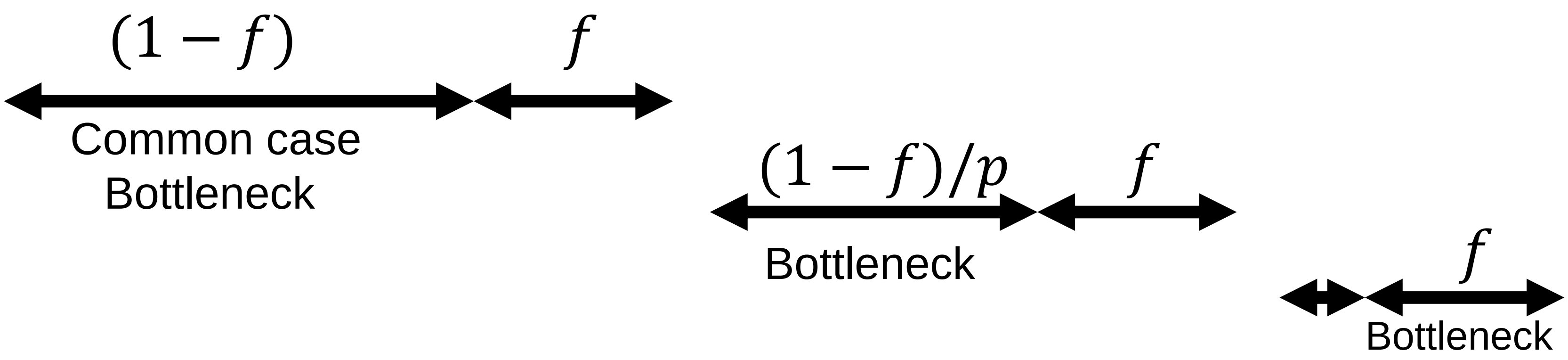
# Implications of Amdahl's law

- Amdahl's law offers two advices for system design
  - *Optimize the common case*. System enhancement, or system optimization, should focus on the common case, also known as the bottleneck, i.e., the most time consuming portion



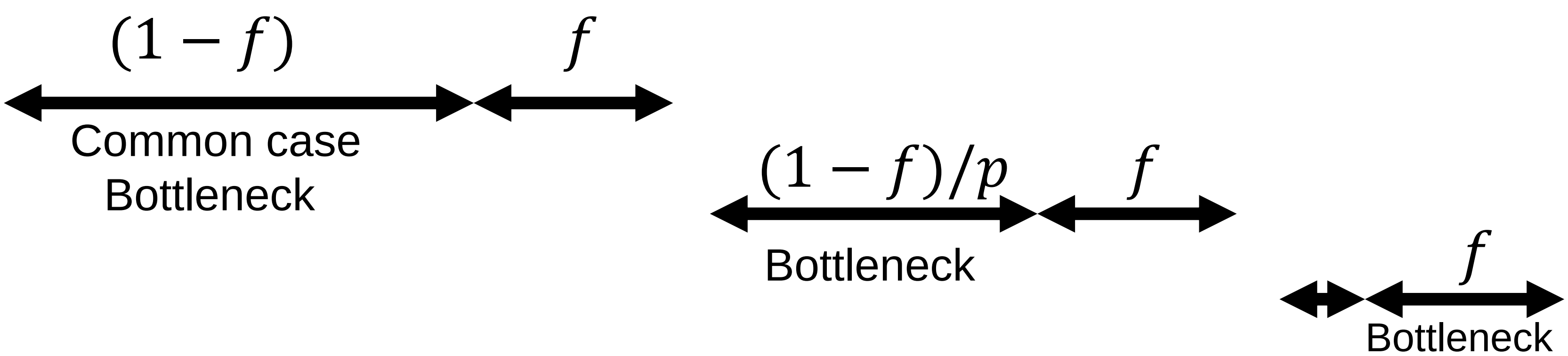
# Implications of Amdahl's law

- Amdahl's law offers two advices for system design
  - *Optimize the common case*. System enhancement, or system optimization, should focus on the common case, also known as the bottleneck, i.e., the most time consuming portion
  - *Chase the bottleneck*. When the system bottleneck changes, so does our optimization target



# Implications of Amdahl's law

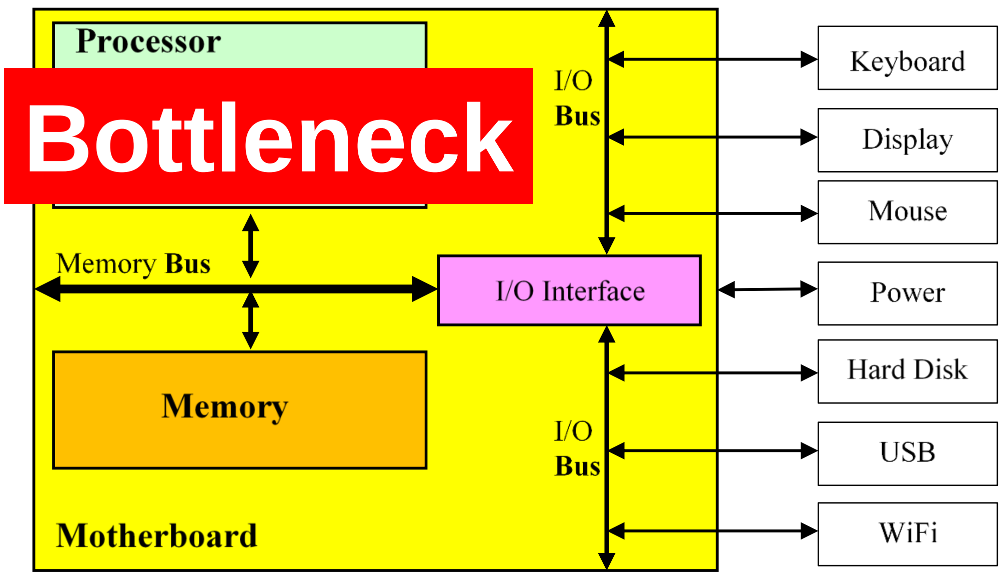
- Amdahl's law offers two advices for system design
  - *Optimize the common case*. System enhancement, or system optimization, should focus on the common case, also known as the bottleneck, i.e., the most time-consuming portion 改进瓶颈
  - *Chase the bottleneck*. When the system bottleneck changes, so does our optimization target 追逐瓶颈

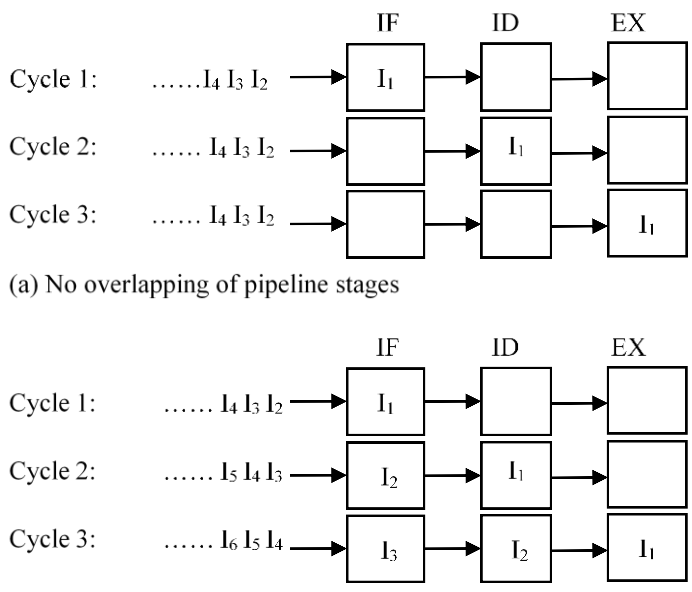



- Three techniques to utilize Amdahl's law
  - Pipelining, caching, parallel computing

# 6.2 Pipelining: multiple instructions overlap

- Optimize the bottleneck of the CPU's instruction pipeline
  - Key technique: overlapping pipeline stages with multiple instructions
    - Using about the same amount of resource, plus some overhead
- Assuming 1-GHz clock cycle, the average performances are
  - Without overlapping: executing an instruction needs 3 cycles (3 ns)
  - With overlapping: executing an instruction needs 1 cycle (1 ns)



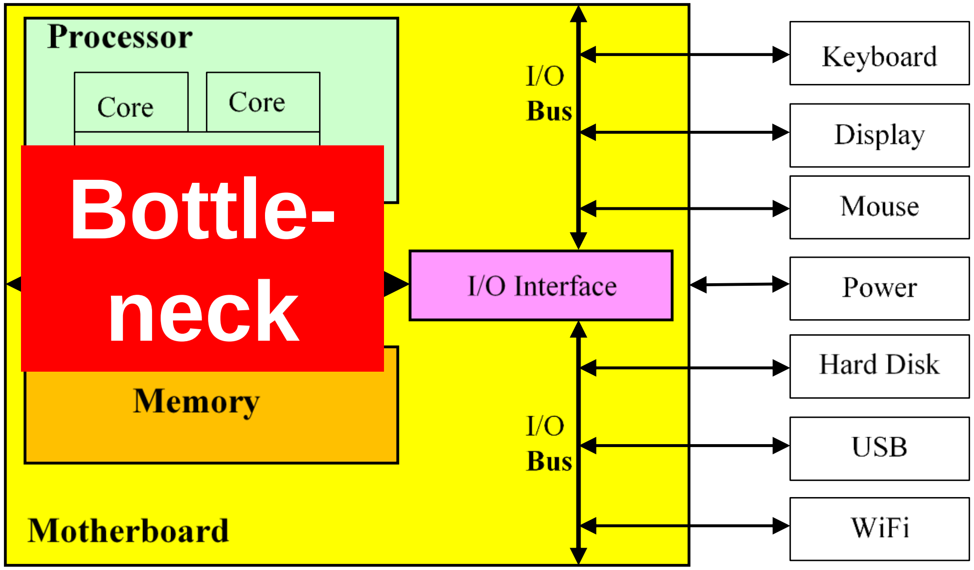




(a) No overlapping of pipeline stages

(b) Overlapping of pipeline stages

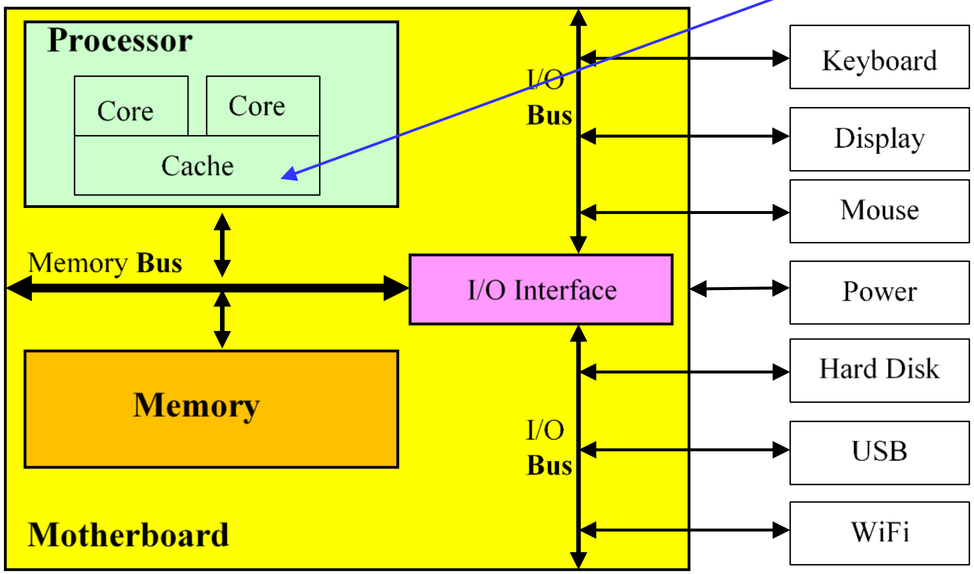
# 6.3 Caching: use a small but faster buffer

- Optimize the von Neumann bottleneck
  - Memory is too slow to feed data to CPU
  - On the other hand, programs have spatial and temporal localities
  - Key technique: caching, i.e., using a small but faster buffer



# 高速缓存 Caching: use a small but faster buffer

- Optimize the von Neumann bottleneck
  - Memory is too slow to feed data to CPU
  - On the other hand, programs have spatial and temporal localities
  - Key technique: caching, i.e., using a small but faster **buffer**
    - Without having to change the instruction set



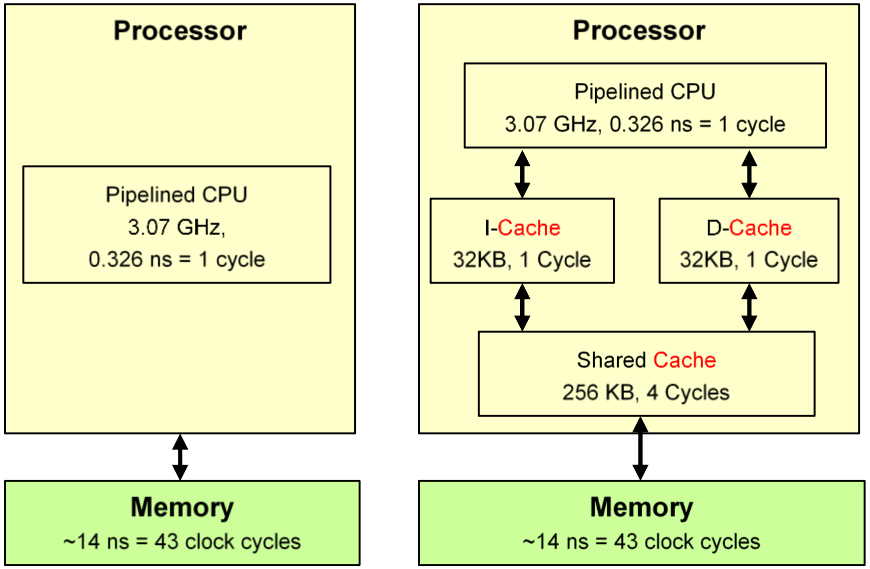
# ***Caching using the Fibonacci Computer

- Two-layer caching
  - Layer 1: separate instruction and data caches
  - Layer 2: shared cache for instruction and data
- Assume 92 iterations of the loop code in red are executed

Clock cycles of instructions without caching

| Instruction type | Times of memory accesses | Clock cycles needed |
|---|---|---|
| MOV 0, R1 | 1 | 1*43 **+1** → 43 |
| MOV R1, M[…] | 2 | 2*43 **+1** → 86 |
| ADD M[…], R1 | 2 | 2*43 **+1** → 86 |
| INC R2 | 1 | 1*43 **+1** → 43 |
| CMP 51, R2 | 1 | 1*43 **+1** → 43 |
| JL Loop | 1 | 1*43 **+1** → 43 |

Numbers in **blue** are canceled out due to pipeline overlapping. A simpler analysis does not consider such optimizations, and calculate the cycles for “MOV 0, R1” to be 1*43+1=44. This simplification leads to similar performance analysis results.

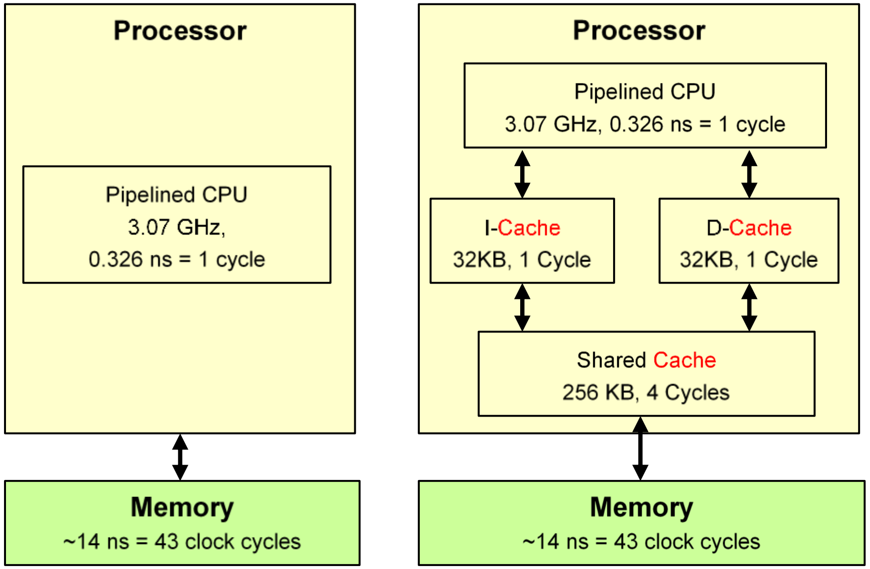



| | | No Caching | Caching |
|---|---|---|---|
| | MOV 0, R1 | 43 | |
| | MOV R1, M[R0] | 86 | |
| | MOV 1, R1 | 43 | |
| | MOV R1, M[R0+8] | 86 | |
| | MOV 2, R2 | 43 | |
| Loop: | MOV 0, R1 | 43 | 1 |
| | ADD M[R0+R2*8-16], R1 | 86 | 1 |
| | ADD M[R0+R2*8-8], R1 | 86 | 2 |
| | MOV R1, M[R0+R2*8-0] | 86 | 2 |
| | INC R2 | 43 | 1 |
| | CMP 51, R2 | 43 | 2 |
| | JL Loop | 43 | 3 |

Assume all internal operations together take 1 cycle.
Every instruction execution needs to access memory at least once to fetch instruction. Some instructions need two accesses to also fetch data.

# ***When code and data are stored in level-1 caches

- Instruction Fetch needs only 1 cycle
  - This is why 1st MOV needs 1 cycle
- Instruction Fetch and Operand Fetch can be done simultaneously, thanks to Harvard architecture
  - This is why 1st ADD needs 1 cycle
- Data and control dependencies may cause pipeline to stall (wait)
  - This is why 2nd ADD needs 2 cycles and JL needs 3 cycles




Clock cycles of instructions with caching

| Instruction type | Times of memory accesses | Clock cycles needed |
|---|---|---|
| MOV 0, R1 | 1 | 1*1 +1 → 1 |
| MOV R1, M[…] | 2 | 1*1 +1 → 1 |
| ADD M[…], R1 | 2 | 1*1 +1 → 1 |
| INC R2 | 1 | 1*1 +1 → 1 |
| CMP 51, R2 | 1 | 1*1 +1 +1 → 2 |
| JL Loop | 1 | 1*1 +1 +2 → 3 |

| | | No Caching | Caching |
|---|---|---|---|
| | MOV 0, R1 | 43 | |
| | MOV R1, M[R0] | 86 | |
| | MOV 1, R1 | 43 | |
| | MOV R1, M[R0+8] | 86 | |
| | MOV 2, R2 | 43 | |
| Loop: | MOV 0, R1 | 43 | 1 |
| | ADD M[R0+R2*8-16], R1 | 86 | 1 |
| | ADD M[R0+R2*8-8], R1 | 86 | 2 |
| | MOV R1, M[R0+R2*8-0] | 86 | 2 |
| | INC R2 | 43 | 1 |
| | CMP 51, R2 | 43 | 2 |
| | JL Loop | 43 | 3 |

Assume all internal operations together take 1 cycle.
Every instruction execution needs to access memory at least once to fetch instruction. Some instructions need two accesses to also fetch data.

# Simplified performance analysis

- Simplify by consider a single iteration of loop
- Without caching
  - Total time = 4*43 + 3*86
    = 430 clock cycles
    = 430 * 0.326 = 140 ns

| | | No Caching | Caching |
|---|---|---|---|
| Loop: | MOV 0, R1 | 43 | 1 |
| | ADD M[R0+R2*8-16], R1 | 86 | 1 |
| | ADD M[R0+R2*8-8], R1 | 86 | 2 |
| | MOV R1, M[R0+R2*8-0] | 86 | 2 |
| | INC R2 | 43 | 1 |
| | CMP 51, R2 | 43 | 2 |
| | JL Loop | 43 | 3 |

- With caching
  - Total time = 1*3 + 2*3 + 3*1
    = 12 clock cycles
    = 12 * 0.326 = 3.91 ns

- Caching brings about significant speedup
  - Speedup = 140 ns / 3.91 ns = 35.8, or 34.8 times faster

# Performance metrics of the 3.07-GHz FC

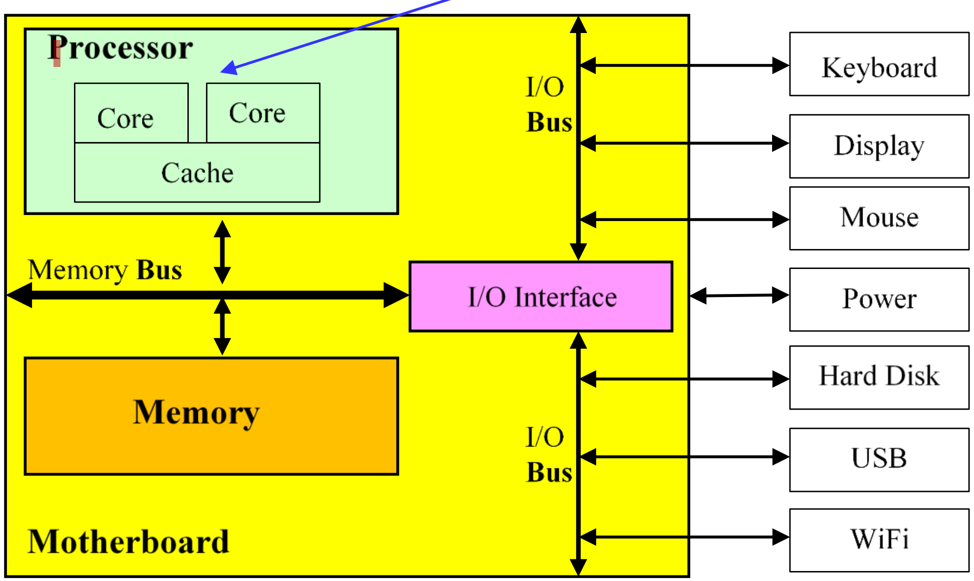
- The peak speed of the Fibonacci Computer is the maximal speed possible
  - Peak speed = 1 instruction per cycle = 3.07 GIPS
- The sustained speed of the Fibonacci Computer is the real speed achieved when executing the loop application code
  - Sustained speed = 7/140 = 0.05 GIPS
    - Each iteration takes 140 ns to execute 7 instructions
  - Efficiency = sustained speed / peak speed = 0.05 / 3.07 = 1.63%

# Performance metrics of the 3.07-GHz FC

- The peak speed of the Fibonacci Computer is the maximal speed possible
  - Peak speed = 1 instruction per cycle = 3.07 GIPS
- The sustained speed of the Fibonacci Computer is the achieved speed when executing the loop application code
  - Sustained speed = 7/140 = 0.05 GIPS
  - Efficiency = sustained speed / peak speed = 0.05 / 3.07 = 1.63%
- For the Fibonacci Computer with caching
  - Sustained speed = 7/12 = 1.79 GIPS
  - Speedup = 1.79 GIPS / 0.05 GIPS = 35.8
  - Efficiency = sustained speed / peak speed = 1.79 / 3.07 = 58.3%
- Both have the same peak speed, but caching significantly improves real speed (sustained speed)

# 6.4 Parallel computing

- Also known as parallel processing
- What if one CPU is not enough?
  - What if we need 10 GIPS, 1 million GIPS, 1 trillion GIPS?
- Use multiple CPUs/processors/computers in one system
  - A processor having multiple CPUs is called a **multicore** processor
    - Each CPU is called a **core**



# Supercomputer examples

- Top500.org is a list ranking the world's fastest supercomputers
  - Maintained since 1993 by scientists in Europe and USA
- Rank the 500 fastest supercomputers by their speeds in executing the Linpack benchmark
  - Speed = executed 64-bit floating-point operations per second (FLOPS)
- The main contributing factor of progress is parallel computing

**The Linpack benchmark** program for solving a system of linear equations using Gaussian elimination. It finds $\boldsymbol{x}$ in $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}$ is an $N \times N$ matrix, and $\boldsymbol{x}, \boldsymbol{b}$ are two $N$-dimensional vectors. For $N$=3, we have

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

| **Time of Test** | 1993 | 2020 | 1993-2020 Growth Factor |
|---|---|---|---|
| **Top-1 Name** | Thinking Machine CM-5 | Fujitsu Fugaku | N/A |
| **Problem Size** | N = 52,224 | N = 20,459,520 | 392 |
| **Speed** | 59.7 GFlop/s | 415,530 TFlop/s | 6,960,302 |
| **Clock Frequency** | 32 MHz | 2.2 GHz | 69 |
| **Parallelism** | **1,024 cores** | **7,299,072 cores** | **7,128** |
| **Memory** | 32 GB | 4,866,048 GB | 152,064 |
| **Power** | 96.5 KW | 28,334.5 KW | 294 |
| **Cost** | US $30 million | US $1 billion | 33 |